Drools Documentation

The JBoss Drools team [http://www.drools.org/community/team.html]

Drools Documentation

by

Version 6.3.1-SNAPSHOT

..... xiv
I. Welcome 1
1. Introduction 2
1.1. Introduction 2
1.2. Getting Involved 2
1.2.1. Sign up to jboss.org 3
1.2.2. Sign the Contributor Agreement 3
1.2.3. Submitting issues via JIRA 4
1.2.4. Fork GitHub 5
1.2.5. Writing Tests 5
1.2.6. Commit with Correct Conventions 7
1.2.7. Submit Pull Requests 8
1.3. Installation and Setup (Core and IDE) 10
1.3.1. Installing and using 10
1.3.2. Building from source 20
1.3.3. Eclipse 21
2. Release Notes 28
2.1. What is New and Noteworthy in Drools 6.3.0 28
2.1.1. Real Time Validation and Verification for the Decision Tables 28
2.1.2. Improved DRL Editor 28
2.1.3. Browsing graphs of objects with OOPath 29
2.1.4. Kie Navigator View for Eclipse 31
2.2. New and Noteworthy in KIE Workbench 6.3.0 31
2.2.1. Asset locking 31
2.2.2. Data Modeller Tool Windows 32
2.2.3. Generation of JPA enabled Data Models 33
2.2.4. Data Set Authoring 36
2.3. What is New and Noteworthy in Drools 6.2.0 36
2.3.1. Propagation modes 36
2.4. New and Noteworthy in KIE Workbench 6.2.0 38
2.4.1. Download Repository or Part of the Repository as a ZIP 38
2.4.2. Project Editor permissions 39
2.4.3. Unify validation style in Guided Decision Table Wizard. 40
2.4.4. Improved Wizards 40
2.4.5. Consistent behaviour of XLS, Guided Decision Tables and Guided Templates 41
2.4.6. Improved Metadata Tab 42
2.4.7. Improved Data Objects Editor 45
2.4.8. Execution Server Management UI 47
2.4.9. Social Activities 48
2.4.10. Contributors Dashboard 50
2.4.11. Package selector 51
2.4.12. Improved visual consistency 52
2.4.13. Guided Decision Tree Editor 54

2.4.14. Create Repository Wizard 55
2.4.15. Repository Structure Screen 56
2.5. New and Noteworthy in Integration 6.2.0 58
2.5.1. KIE Execution Server 58
2.6. What is New and Noteworthy in Drools 6.1.0 60
2.6.1. JMX support for KieScanner 60
2.7. New and Noteworthy in KIE Workbench 6.1.0 60
2.7.1. Data Modeler - round trip and source code preservation 60
2.7.2. Data Modeler - improved annotations 60
2.7.3. Standardization of the display of tabular data 60
2.7.4. Generation of `modify(x) {...}` blocks 61
2.8. New and Noteworthy in KIE API 6.0.0 62
2.8.1. New KIE name 62
2.8.2. Maven aligned projects and modules and Maven Deployment 62
2.8.3. Configuration and convention based projects 63
2.8.4. KieBase Inclusion 63
2.8.5. KieModules, KieContainer and KIE-CI 64
2.8.6. KieScanner 64
2.8.7. Hierarchical ClassLoader 65
2.8.8. Legacy API Adapter 65
2.8.9. KIE Documentation 65
2.9. What is New and Noteworthy in Drools 6.0.0 66
2.9.1. PHREAK - Lazy rule matching algorithm 66
2.9.2. Automatically firing timed rule in passive mode 66
2.9.3. Expression Timers 67
2.9.4. RuleFlowGroups and AgendaGroups are merged 68
2.10. New and Noteworthy in KIE Workbench 6.0.0 68
2.11. New and Noteworthy in Integration 6.0.0 71
2.11.1. CDI 71
2.11.2. Spring 72
2.11.3. Aries Blueprints 72
2.11.4. OSGi Ready 72
3. Compatibility matrix 73
II. KIE 74
4. KIE 75
4.1. Overview 75
4.1.1. Anatomy of Projects 75
4.1.2. Lifecycles 76
4.2. Build, Deploy, Utilize and Run 77
4.2.1. Introduction 77
4.2.2. Building 80
4.2.3. Deploying 97
4.2.4. Running 102
4.2.5. Installation and Deployment Cheat Sheets 117

4.2.6. Build, Deploy and Utilize Examples 118
4.3. Security 129
4.3.1. Security Manager 129
III. Drools Runtime and Language 132
5. Hybrid Reasoning 133
5.1. Artificial Intelligence 133
5.1.1. A Little History 133
5.1.2. Knowledge Representation and Reasoning 134
5.1.3. Rule Engines and Production Rule Systems (PRS) 135
5.1.4. Hybrid Reasoning Systems (HRS) 137
5.1.5. Expert Systems 140
5.1.6. Recommended Reading 141
5.2. Rete Algorithm 144
5.3. ReteOO Algorithm 151
5.4. PHREAK Algorithm 152
6. User Guide 161
6.1. The Basics 161
6.1.1. Stateless Knowledge Session 161
6.1.2. Stateful Knowledge Session 164
6.1.3. Methods versus Rules 169
6.1.4. Cross Products 169
6.2. Execution Control 171
6.2.1. Agenda 171
6.2.2. Rule Matches and Conflict Sets. 172
6.2.3. Declarative Agenda 178
6.3. Inference 181
6.3.1. Bus Pass Example 181
6.4. Truth Maintenance with Logical Objects 183
6.4.1. Overview 183
6.5. Decision Tables in Spreadsheets 188
6.5.1. When to Use Decision Tables 188
6.5.2. Overview 189
6.5.3. How Decision Tables Work 191
6.5.4. Spreadsheet Syntax 194
6.5.5. Creating and integrating Spreadsheet based Decision Tables 204
6.5.6. Managing Business Rules in Decision Tables 205
6.5.7. Rule Templates 206
6.6. Logging 209
7. Running 210
7.1. KieRuntime 210
7.1.1. EntryPoint 210
7.1.2. RuleRuntime 212
7.1.3. StatefulRuleSession 213
7.2. Agenda 214

7.2.1. Conflict Resolution 215
7.2.2. AgendaGroup 215
7.2.3. ActivationGroup 216
7.2.4. RuleFlowGroup 216
7.3. Event Model 217
7.4. StatelessKieSession 218
7.4.1. Sequential Mode 220
7.5. Propagation modes 221
7.6. Commands and the CommandExecutor 222
8. Rule Language Reference 228
8.1. Overview 228
8.1.1. A rule file 228
8.1.2. What makes a rule 229
8.2. Keywords 229
8.3. Comments 231
8.3.1. Single line comment 231
8.3.2. Multi-line comment 232
8.4. Error Messages 232
8.4.1. Message format 232
8.4.2. Error Messages Description 233
8.4.3. Other Messages 237
8.5. Package 237
8.5.1. import 238
8.5.2. global 238
8.6. Function 240
8.7. Type Declaration 241
8.7.1. Declaring New Types 242
8.7.2. Declaring Metadata 245
8.7.3. Declaring Metadata for Existing Types 251
8.7.4. Parametrized constructors for declared types 252
8.7.5. Non Typesafe Classes 252
8.7.6. Accessing Declared Types from the Application Code 253
8.7.7. Type Declaration 'extends' 254
8.7.8. Traits 255
8.8. Rule 261
8.8.1. Rule Attributes 262
8.8.2. Timers and Calendars 266
8.8.3. Left Hand Side (when) syntax 269
8.8.4. The Right Hand Side (then) 322
8.8.5. Conditional named consequences 324
8.8.6. A Note on Auto-boxing and Primitive Types 326
8.9. Query 327
8.10. Domain Specific Languages 330
8.10.1. When to Use a DSL 330

8.10.2. DSL Basics 330
8.10.3. Adding Constraints to Facts 333
8.10.4. Developing a DSL 334
8.10.5. DSL and DSLR Reference 335
9. Complex Event Processing 339
9.1. Complex Event Processing 339
9.2. Drools Fusion 340
9.3. Event Semantics 342
9.4. Event Processing Modes 343
9.4.1. Cloud Mode 344
9.4.2. Stream Mode 345
9.5. Session Clock 347
9.5.1. Available Clock Implementations 348
9.6. Sliding Windows 349
9.6.1. Sliding Time Windows 349
9.6.2. Sliding Length Windows 350
9.6.3. Window Declaration 352
9.7. Streams Support 352
9.7.1. Declaring and Using Entry Points 353
9.8. Memory Management for Events 355
9.8.1. Explicit expiration offset 355
9.8.2. Inferred expiration offset 356
9.9. Temporal Reasoning 356
9.9.1. Temporal Operators 357
10. Experimental Features 371
10.1. Browsing graphs of objects with OOPath 371
10.1.1. Reactive OOPath 372
IV. Drools Integration 373
11. Drools Commands 374
11.1. API 374
11.1.1. XStream 374
11.1.2. JSON 374
11.1.3. JAXB 374
11.2. Commands supported 375
11.2.1. BatchExecutionCommand 377
11.2.2. InsertObjectCommand 378
11.2.3. RetractCommand 380
11.2.4. ModifyCommand 381
11.2.5. GetObjectCommand 382
11.2.6. InsertElementsCommand 383
11.2.7. FireAllRulesCommand 385
11.2.8. StartProcessCommand 386
11.2.9. SignalEventCommand 387
11.2.10. CompleteWorkItemCommand 388

11.2.11. AbortWorkItemCommand 389
11.2.12. QueryCommand 390
11.2.13. SetGlobalCommand 391
11.2.14. GetGlobalCommand 393
11.2.15. GetObjectsCommand 394
12. CDI 396
12.1. Introduction 396
12.2. Annotations 396
12.2.1. @KReleaseId 396
12.2.2. @KContainer 396
12.2.3. @KBase 397
12.2.4. @KSession for KieSession 398
12.2.5. @KSession for StatelessKieSession 399
12.3. API Example Comparison 400
13. Integration with Spring 401
13.1. Important Changes for Drools 6.0 401
13.2. Integration with Drools Expert 401
13.2.1. KieModule 401
13.2.2. KieBase 402
13.2.3. IMPORTANT NOTE 403
13.2.4. KieSessions 404
13.2.5. Kie:ReleaseId 405
13.2.6. Kie:Import 405
13.2.7. Annotations 407
13.2.8. Event Listeners 411
13.2.9. Loggers 415
13.2.10. Defining Batch Commands 416
13.2.11. Persistence 417
13.2.12. Leveraging Other Spring Features 418
13.3. Integration with jBPM Human Task 420
13.3.1. How to configure Spring with jBPM Human task 420
14. Android Integration 424
14.1. Integration with Drools Expert 424
14.1.1. Pre-serialized Rules 424
14.1.2. KieContainer API with drools-compiler dependency 426
14.2. Integration with Roboguice 428
14.2.1. Pre-serialized Rules with Roboguice 428
14.2.2. KieContainer with drools-compiler dependency and Roboguice 429
15. Apache Camel Integration 432
15.1. Camel 432
16. Drools Camel Server 435
16.1. Introduction 435
16.2. Deployment 435
16.3. Configuration 435

16.3.1. REST/Camel Services configuration 435
17. JMX monitoring with RHQ/JON 441
17.1. Introduction 441
17.1.1. Enabling JMX monitoring in a Drools application 441
17.1.2. Installing and running the RHQ/JON plugin 441
V. Drools Workbench 443
18. Workbench 444
18.1. Installation 444
18.1.1. War installation 444
18.1.2. Workbench data 444
18.1.3. System properties 445
18.2. Quick Start 446
18.2.1. Add repository 446
18.2.2. Add project 449
18.2.3. Define Data Model 453
18.2.4. Define Rule 456
18.2.5. Build and Deploy 459
18.3. Administration 460
18.3.1. Administration overview 460
18.3.2. Organizational unit 460
18.3.3. Repositories 461
18.4. Configuration 463
18.4.1. User management 463
18.4.2. Roles 463
18.4.3. Restricting access to repositories 465
18.4.4. Command line config tool 465
18.5. Introduction 466
18.5.1. Log in and log out 466
18.5.2. Home screen 467
18.5.3. Workbench concepts 467
18.5.4. Initial layout 467
18.6. Changing the layout 468
18.6.1. Resizing 469
18.6.2. Repositioning 469
18.7. Authoring 471
18.7.1. Artifact Repository 471
18.7.2. Asset Editor 473
18.7.3. Tags Editor 477
18.7.4. Project Explorer 479
18.7.5. Project Editor 492
18.7.6. Validation 495
18.7.7. Data Modeller 497
18.7.8. Data Sets 537
18.8. Embedding Workbench In Your Application 551

18.9. Asset Management 552
18.9.1. Asset Management Overview 552
18.9.2. Managed vs Unmanaged Repositories 552
18.9.3. Asset Management Processes 552
18.9.4. Usage Flow 554
18.9.5. Repository Structure 556
18.9.6. Managed Repositories Operations 557
18.9.7. Remote APIs 563
19. Authoring Assets 564
19.1. Creating a package 564
19.1.1. Empty package 565
19.1.2. Copy, Rename and Delete Packages 566
19.2. Business rules with the guided editor 568
19.2.1. Parts of the Guided Rule Editor 568
19.2.2. The "WHEN" (left-hand side) of a Rule 569
19.2.3. The "THEN" (right-hand side) of a Rule 573
19.2.4. Optional attributes 576
19.2.5. Pattern/Action toolbar 576
19.2.6. User driven drop down lists 576
19.2.7. Augmenting with DSL sentences 577
19.2.8. A more complex example: 578
19.3. Templates of assets/rules 579
19.3.1. Creating a rule template 580
19.3.2. Define the template 580
19.3.3. Defining the template data 581
19.3.4. Generated DRL 585
19.4. Guided decision tables (web based) 587
19.4.1. Types of decision table 587
19.4.2. Main components\concepts 588
19.4.3. Defining a web based decision table 591
19.4.4. Rule definition 606
19.4.5. Audit Log 607
19.4.6. Real Time Validation and Verification 609
19.5. Guided Decision Trees 610
19.5.1. The initial editor layout 610
19.5.2. First steps 612
19.5.3. Editing Data Object nodes 613
19.5.4. Editing Field Constraint nodes 614
19.5.5. Editing Action nodes 615
19.5.6. Managing the tree 618
19.6. Spreadsheet decision tables 620
19.7. Scorecards 621
19.7.1. (a) Setup Parameters 622
19.7.2. (b) Characteristics 623

19.8. Test Scenario 625
19.8.1. Knowledge Session Selector 627
19.8.2. Given Section 628
19.8.3. Expect Section 628
19.8.4. Global Section 629
19.8.5. New Input Section 629
19.9. Functions 629
19.10. DSL editor 630
19.11. Data enumerations (drop down list configurations) 631
19.11.1. Advanced enumeration concepts 632
19.12. Technical rules (DRL) 633
20. Workbench Integration 635
20.1. REST 635
20.1.1. Job calls 635
20.1.2. Repository calls 636
20.1.3. Organizational unit calls 638
20.1.4. Maven calls 638
20.1.5. REST summary 639
21. Workbench High Availability 641
21.1. 641
21.1.1. VFS clustering 641
21.1.2. jBPM clustering 644
VI. KIE Server 645
22. KIE Execution Server 646
22.1. Overview 646
22.1.1. Glossary 646
22.2. Installing the KIE Server 647
22.2.1. Bootstrap switches 648
22.2.2. Installation details for different containers 650
22.3. Kie Server setup 652
22.3.1. Managed Kie Server 652
22.3.2. Unmanaged KIE Execution Server 654
22.4. Creating a Kie Container 654
22.5. Managing Containers 655
22.5.1. Starting a Container 655
22.5.2. Stopping and Deleting a Container 656
22.5.3. Updating a Container 656
22.6. REST API 656
22.6.1. [GET] / 657
22.6.2. [POST] / 657
22.6.3. [GET] /containers 657
22.6.4. [GET] /containers/{id} 658
22.6.5. [PUT] /containers/{id} 658
22.6.6. [DELETE] /containers/{id} 659

22.6.7. [POST] /containers/{id} 659
22.6.8. [GET] /containers/{id}/release-id 660
22.6.9. [POST] /containers/{id}/release-id 660
22.6.10. [GET] /containers/{id}/scanner 661
22.6.11. [POST] /containers/{id}/scanner 661
22.6.12. Native REST client for Execution Server 662
VII. Drools Examples 664
23. Examples 665
23.1. Getting the Examples 665
23.2. Hello World 665
23.3. State Example 671
23.3.1. Understanding the State Example 671
23.4. Fibonacci Example 678
23.5. Banking Tutorial 684
23.6. Pricing Rule Decision Table Example 697
23.6.1. Executing the example 697
23.6.2. The decision table 698
23.7. Pet Store Example 700
23.8. Honest Politician Example 711
23.9. Sudoku Example 715
23.9.1. Sudoku Overview 715
23.9.2. Running the Example 715
23.9.3. Java Source and Rules Overview 721
23.9.4. Sudoku Validator Rules (validate.drl) 721
23.9.5. Sudoku Solving Rules (sudoku.drl) 722
23.10. Number Guess 723
23.11. Conway's Game Of Life 730
23.12. Invaders 737
23.12.1. Invaders1Main 738
23.12.2. Invaders2Main 739
23.12.3. Invaders3Main 739
23.12.4. Invaders4Main 740
23.12.5. Invaders5Main 740
23.12.6. Invaders6Main 740
23.12.7. Invaders4Main 741
23.13. Adventures with Drools 741
23.13.1. Using the game. 742
23.13.2. The code 744
23.14. Pong 746
23.15. Wumpus World 747
23.16. Miss Manners and Benchmarking 750
23.16.1. Introduction 751
23.16.2. In depth Discussion 754
23.16.3. Output Summary 760

23.17. Backward-Chaining 763
23.17.1. Backward-Chaining Systems 764
23.17.2. Cloning Transitive Closures 765
23.17.3. Defining a Query 766
23.17.4. Transitive Closure Example 767
23.17.5. Reactive Transitive Queries 769
23.17.6. Queries with Unbound Arguments 770
23.17.7. Multiple Unbound Arguments 771

Drools

Part I. Welcome

Welcome and Release Notes

Chapter 1. Introduction

1.1. Introduction

It's been a busy year since the last 5.x series release and so much has change.

One of the biggest complaints during the 5.x series was the lack of defined methodology for deployment. The mechanism used by Drools and jBPM was very flexible, but it was too flexible. A big focus for 6.0 was streamlining the build, deploy and loading(utilization) aspects of the system. Building and deploying now align with Maven and the utilization is now convention and configuration oriented, instead of programmatic, with sane default to minimise the configuration.

The workbench has been rebuilt from the ground up, inspired by Eclipse, to provide a flexible and better integrated solution; with panels and perspectives via plugins. The base workbench has been spun off into a standalone project called UberFire, so that anyone now can build high quality web based workbenches. In the longer term it will facilitate user customised Drools and jBPM installations.

Git replaces JCR as the content repository, offering a fast and scalable back-end storage for content that has strong tooling support. There has been a refocus on simplicity away from databases with an aim of storing everything as text file, even meta data is just a file. The database is just there to provide fast indexing and search via Lucene. This will allow repositories now to be synced and published with established infrastructure, like GitHub.

jBPM has been dramatically beefed up, thanks to the Polymita acquisition, with human tasks, form builders, class modellers, execution servers and runtime management. All fully integrated into the new workbench.

OptaPlanner is now a top level project and getting full time attention.

A new umbrella name, KIE (Knowledge Is Everything), has been introduced to bring our related technologies together under one roof. It also acts as the core shared around for our projects. So expect to see it a lot.

1.2. Getting Involved

We are often asked "How do I get involved". Luckily the answer is simple, just write some code and submit it :) There are no hoops you have to jump through or secret handshakes. We have a very minimal "overhead" that we do request to allow for scalable project development. Below we provide a general overview of the tools and "workflow" we request, along with some general advice.

If you contribute some good work, don't forget to blog about it :)

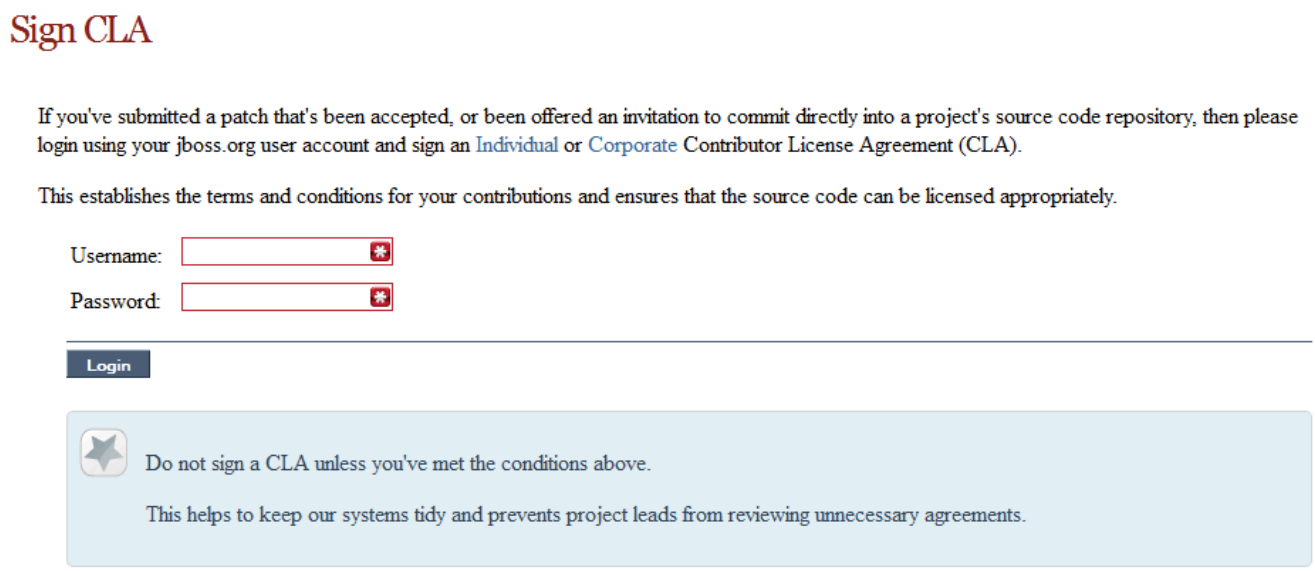
1.2.1. Sign up to jboss.org

Signing to jboss.org will give you access to the JBoss wiki, forums and JIRA. Go to http://www.jboss.org/ and click "Register".

1.2.2. Sign the Contributor Agreement

The only form you need to sign is the contributor agreement, which is fully automated via the web. As the image below says "This establishes the terms and conditions for your contributions and ensures that source code can be licensed appropriately"

https://cla.jboss.org/



1.2.3. Submitting issues via JIRA

To be able to interact with the core development team you will need to use JIRA, the issue tracker. This ensures that all requests are logged and allocated to a release schedule and all discussions captured in one place. Bug reports, bug fixes, feature requests and feature submissions should all go here. General questions should be undertaken at the mailing lists.

Minor code submissions, like format or documentation fixes do not need an associated JIRA issue created.

https://issues.jboss.org/browse/JBRULES [https://issues.jboss.org/browse/JBRULES](Drools)

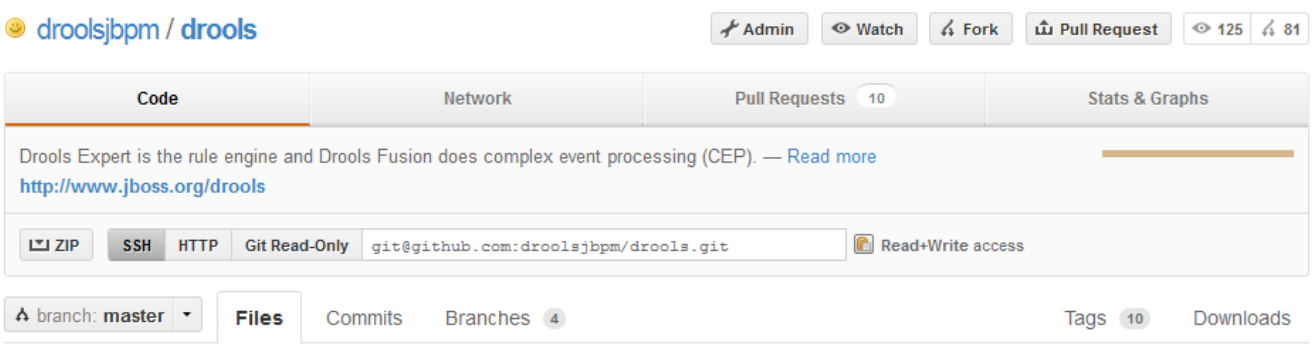
https://issues.jboss.org/browse/JBPM

https://issues.jboss.org/browse/GUVNOR

1.2.4. Fork GitHub

With the contributor agreement signed and your requests submitted to JIRA you should now be ready to code :) Create a GitHub account and fork any of the Drools, jBPM or Guvnor repositories. The fork will create a copy in your own GitHub space which you can work on at your own pace. If you make a mistake, don't worry blow it away and fork again. Note each GitHub repository provides you the clone (checkout) URL, GitHub will provide you URLs specific to your fork.

https://github.com/droolsjbpm



1.2.5. Writing Tests

When writing tests, try and keep them minimal and self contained. We prefer to keep the DRL fragments within the test, as it makes for quicker reviewing. If their are a large number of rules

then using a String is not practical so then by all means place them in separate DRL files instead to be loaded from the classpath. If your tests need to use a model, please try to use those that already exist for other unit tests; such as Person, Cheese or Order. If no classes exist that have the fields you need, try and update fields of existing classes before adding a new class.

There are a vast number of tests to look over to get an idea, MiscTest is a good place to start.

https://github.com/droolsjbpm/drools/blob/master/drools-compiler/src/test/java/org/drools/integrationtests/MiscTest.java [https://github.com/droolsjbpm]

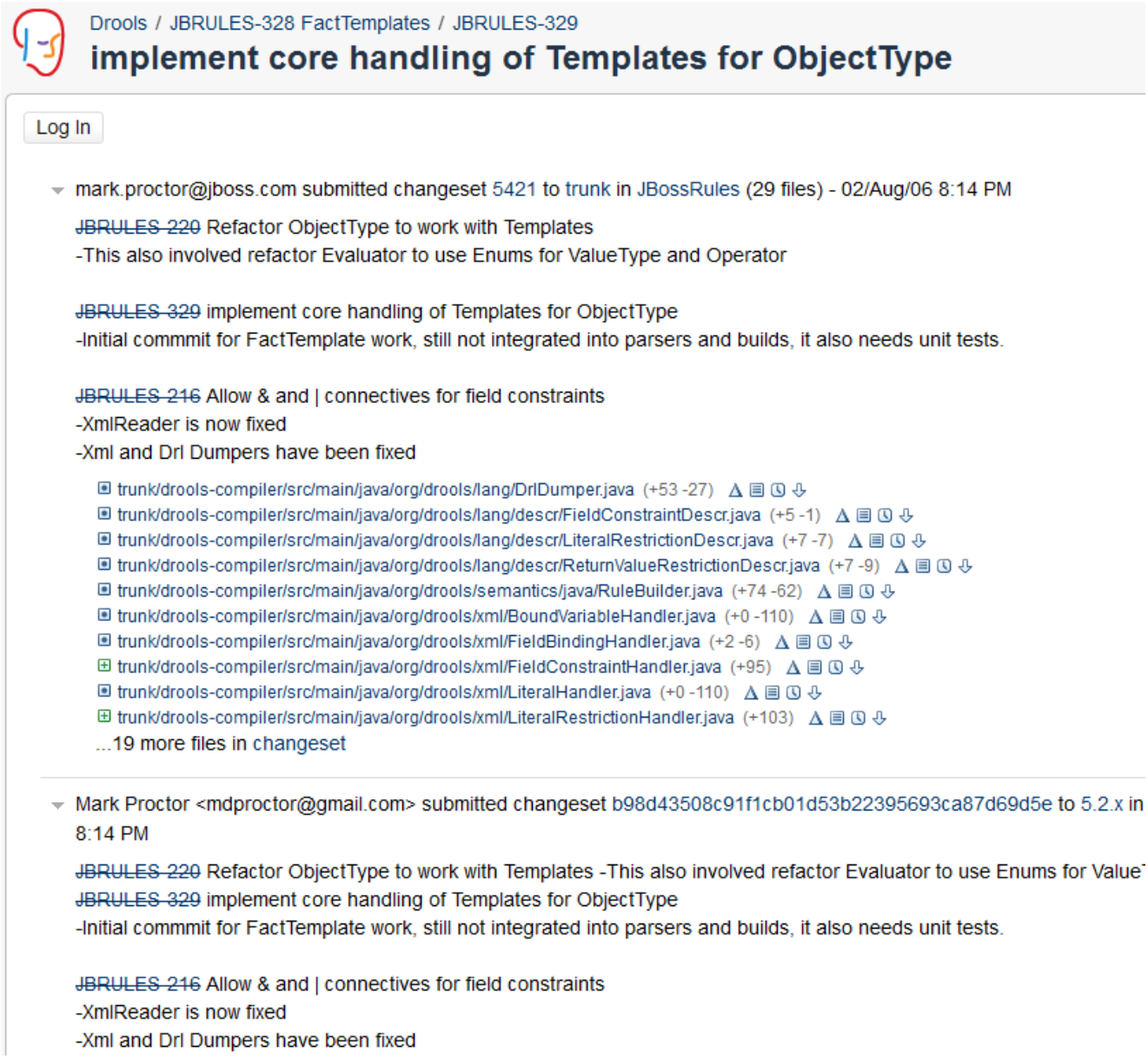
```
557     @Test
558     public void testEvalWithBigDecimal() throws Exception {
559         String str = "";
560         str += "package org.drools \n";
561         str += "import java.math.BigDecimal; \n";
562         str += "global java.util.List list \n";
563         str += "rule rule1 \n";
564         str += "    dialect \"java\" \n";
565         str += "when \n";
566         str += "    $bd : BigDecimal() \n";
567         str += "    eval( $bd.compareTo( BigDecimal.ZERO ) > 0 ) \n";
568         str += "then \n";
569         str += "    list.add( $bd ); \n";
570         str += "end \n";
571
572         KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
573
574         kbuilder.add( ResourceFactory.newByteArrayResource( str.getBytes() ),
575                       ResourceType.DRL );
576
577         if ( kbuilder.hasErrors() ) {
578             logger.warn( kbuilder.getErrors().toString() );
579         }
580         assertFalse( kbuilder.hasErrors() );
581
582         KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
583         kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
584
585         StatefulKnowledgeSession ksession = createKnowledgeSession(kbase);
586         List list = new ArrayList();
587         ksession.setGlobal( "list",
588                             list );
589         ksession.insert( new BigDecimal( 1.5 ) );
590
591         ksession.fireAllRules();
592
593         assertEquals( 1,
594                       list.size() );
595         assertEquals( new BigDecimal( 1.5 ),
596                       list.get( 0 ) );
597     }
598
```

1.2.6. Commit with Correct Conventions

When you commit, make sure you use the correct conventions. The commit must start with the JIRA issue id, such as JBRULES-220. This ensures the commits are cross referenced via JIRA, so we can see all commits for a given issue in the same place. After the id the title of the issue should come next. Then use a newline, indented with a dash, to provide additional information

related to this commit. Use an additional new line and dash for each separate point you wish to make. You may add additional JIRA cross references to the same commit, if it's appropriate. In general try to avoid combining unrelated issues in the same commit.

Don't forget to rebase your local fork from the original master and then push your commits back to your fork.



1.2.7. Submit Pull Requests

With your code rebased from original master and pushed to your personal GitHub area, you can now submit your work as a pull request. If you look at the top of the page in GitHub for your work area their will be a "Pull Request" button. Selecting this will then provide a gui to automate the submission of your pull request.

The pull request then goes into a queue for everyone to see and comment on. Below you can see a typical pull request. The pull requests allow for discussions and it shows all associated commits and the diffs for each commit. The discussions typically involve code reviews which provide helpful suggestions for improvements, and allows for us to leave inline comments on specific parts of the code. Don't be disheartened if we don't merge straight away, it can often take several revisions before we accept a pull request. Luckily GitHub makes it very trivial to go back to your code, do some more commits and then update your pull request to your latest and greatest.

It can take time for us to get round to responding to pull requests, so please be patient. Submitted tests that come with a fix will generally be applied quite quickly, where as just tests will often way until we get time to also submit that with a fix. Don't forget to rebase and resubmit your request from time to time, otherwise over time it will have merge conflicts and core developers will general ignore those.

1.3. Installation and Setup (Core and IDE)

1.3.1. Installing and using

Drools provides an Eclipse-based IDE (which is optional), but at its core only Java 1.5 (Java SE) is required.

A simple way to get started is to download and install the Eclipse plug-in - this will also require the Eclipse GEF framework to be installed (see below, if you don't have it installed already). This will provide you with all the dependencies you need to get going: you can simply create a new rule project and everything will be done for you. Refer to the chapter on the Rule Workbench and IDE for detailed instructions on this. Installing the Eclipse plug-in is generally as simple as unzipping a file into your Eclipse plug-in directory.

Use of the Eclipse plug-in is not required. Rule files are just textual input (or spreadsheets as the case may be) and the IDE (also known as the Rule Workbench) is just a convenience. People have integrated the rule engine in many ways, there is no "one size fits all".

Alternatively, you can download the binary distribution, and include the relevant JARs in your projects classpath.

1.3.1.1. Dependencies and JARs

Drools is broken down into a few modules, some are required during rule development/compiling, and some are required at runtime. In many cases, people will simply want to include all the dependencies at runtime, and this is fine. It allows you to have the most flexibility. However, some may prefer to have their "runtime" stripped down to the bare minimum, as they will be deploying rules in binary form - this is also possible. The core runtime engine can be quite compact, and only requires a few 100 kilobytes across 3 JAR files.

The following is a description of the important libraries that make up JBoss Drools

- knowledge-api.jar - this provides the interfaces and factories. It also helps clearly show what is intended as a user API and what is just an engine API.
- knowledge-internal-api.jar - this provides internal interfaces and factories.
- drools-core.jar - this is the core engine, runtime component. Contains both the RETE engine and the LEAPS engine. This is the only runtime dependency if you are pre-compiling rules (and deploying via Package or RuleBase objects).
- drools-compiler.jar - this contains the compiler/builder components to take rule source, and build executable rule bases. This is often a runtime dependency of your application, but it need not be if you are pre-compiling your rules. This depends on drools-core.
- drools-jsr94.jar - this is the JSR-94 compliant implementation, this is essentially a layer over the drools-compiler component. Note that due to the nature of the JSR-94 specification, not all

features are easily exposed via this interface. In some cases, it will be easier to go direct to the Drools API, but in some environments the JSR-94 is mandated.

- drools-decisiontables.jar - this is the decision tables 'compiler' component, which uses the drools-compiler component. This supports both excel and CSV input formats.

There are quite a few other dependencies which the above components require, most of which are for the drools-compiler, drools-jsr94 or drools-decisiontables module. Some key ones to note are "POI" which provides the spreadsheet parsing ability, and "antlr" which provides the parsing for the rule language itself.

NOTE: if you are using Drools in J2EE or servlet containers and you come across classpath issues with "JDT", then you can switch to the janino compiler. Set the system property "drools.compiler": For example: -Ddrools.compiler=JANINO.

For up to date info on dependencies in a release, consult the released POMs, which can be found on the Maven repository.

1.3.1.2. Use with Maven, Gradle, Ivy, Buildr or Ant

The JARs are also available in the central Maven repository [http://search.maven.org/#search|ga|1|org.drools] (and also in the JBoss Maven repository [https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.drools~~~~]).

If you use Maven, add KIE and Drools dependencies in your project's `pom.xml` like this:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-bom</artifactId>
      <type>pom</type>
      <version>...</version>
      <scope>import</scope>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <scope>runtime</scope>
  </dependency>
  ...
<dependencies>
```

This is similar for Gradle, Ivy and Buildr. To identify the latest version, check the Maven repository.

If you're still using Ant (without Ivy), copy all the JARs from the download zip's `binaries` directory and manually verify that your classpath doesn't contain duplicate JARs.

1.3.1.3. Runtime

The "runtime" requirements mentioned here are if you are deploying rules as their binary form (either as KnowledgePackage objects, or KnowledgeBase objects etc). This is an optional feature that allows you to keep your runtime very light. You may use drools-compiler to produce rule packages "out of process", and then deploy them to a runtime system. This runtime system only requires drools-core.jar and knowledge-api for execution. This is an optional deployment pattern, and many people do not need to "trim" their application this much, but it is an ideal option for certain environments.

1.3.1.4. Installing IDE (Rule Workbench)

The rule workbench (for Eclipse) requires that you have Eclipse 3.4 or greater, as well as Eclipse GEF 3.4 or greater. You can install it either by downloading the plug-in or using the update site.

Another option is to use the JBoss IDE, which comes with all the plug-in requirements pre packaged, as well as a choice of other tools separate to rules. You can choose just to install rules from the "bundle" that JBoss IDE ships with.

1.3.1.4.1. Installing GEF (a required dependency)

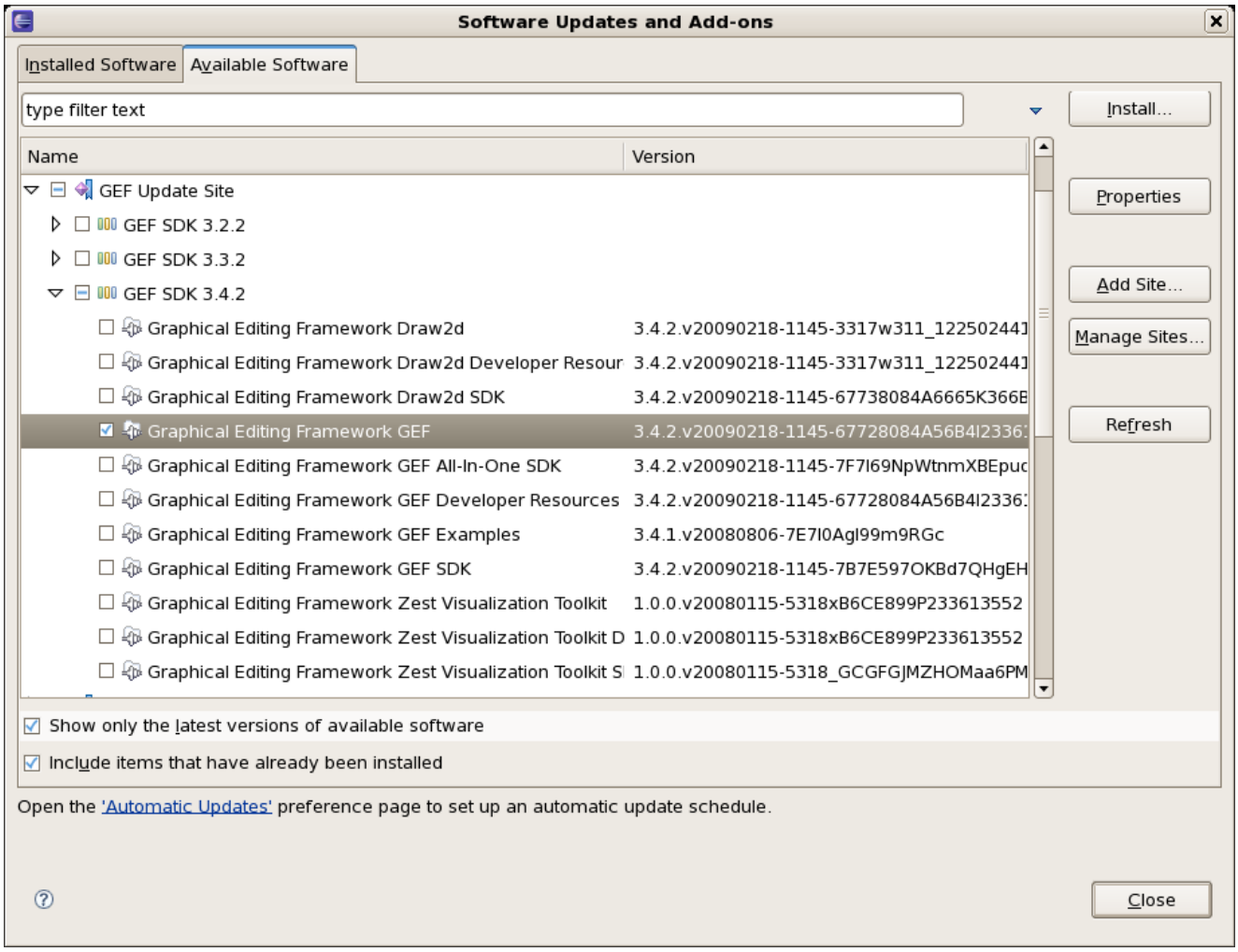
GEF is the Eclipse Graphical Editing Framework, which is used for graph viewing components in the plug-in.

If you don't have GEF installed, you can install it using the built in update mechanism (or downloading GEF from the Eclipse.org website not recommended). JBoss IDE has GEF already, as do many other "distributions" of Eclipse, so this step may be redundant for some people.

Open the Help->Software updates...->Available Software->Add Site... from the help menu. Location is:

```
http://download.eclipse.org/tools/gef/updates/releases/
```

Next you choose the GEF plug-in:



Press next, and agree to install the plug-in (an Eclipse restart may be required). Once this is completed, then you can continue on installing the rules plug-in.

1.3.1.4.2. Installing GEF from zip file

To install from the zip file, download and unzip the file. Inside the zip you will see a plug-in directory, and the plug-in JAR itself. You place the plug-in JAR into your Eclipse applications plug-in directory, and restart Eclipse.

1.3.1.4.3. Installing Drools plug-in from zip file

Download the Drools Eclipse IDE plugin from the link below. Unzip the downloaded file in your main eclipse folder (do not just copy the file there, extract it so that the feature and plugin JARs end up in the features and plugin directory of eclipse) and (re)start Eclipse.

http://www.drools.org/download/download.html

To check that the installation was successful, try opening the Drools perspective: Click the 'Open Perspective' button in the top right corner of your Eclipse window, select 'Other...' and pick the Drools perspective. If you cannot find the Drools perspective as one of the possible perspectives,

the installation probably was unsuccessful. Check whether you executed each of the required steps correctly: Do you have the right version of Eclipse (3.4.x)? Do you have Eclipse GEF installed (check whether the org.eclipse.gef_3.4.*.jar exists in the plugins directory in your eclipse root folder)? Did you extract the Drools Eclipse plugin correctly (check whether the org.drools.eclipse_*.jar exists in the plugins directory in your eclipse root folder)? If you cannot find the problem, try contacting us (e.g. on irc or on the user mailing list), more info can be found no our homepage here:

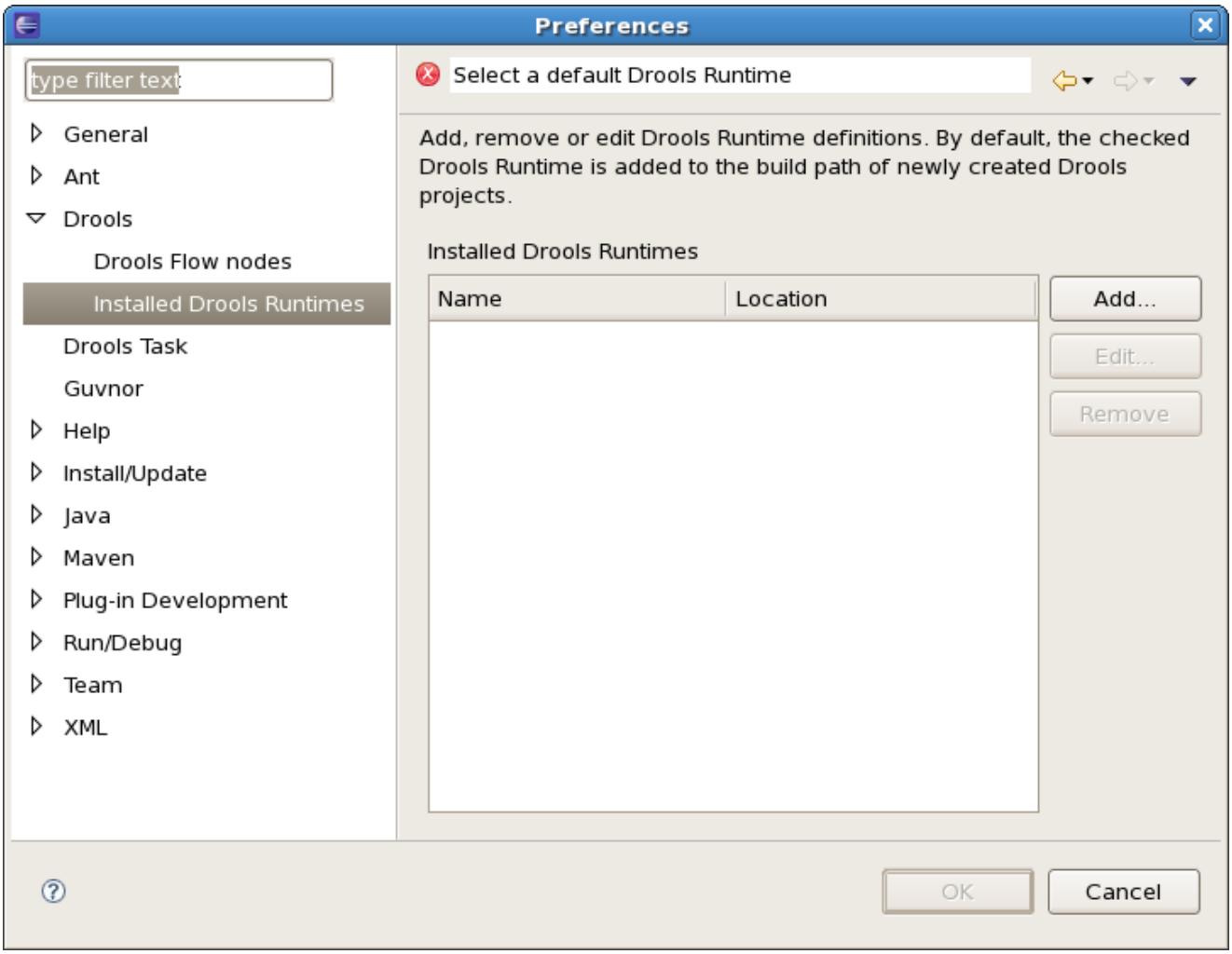
http://www.drools.org/

1.3.1.4.4. Drools Runtimes

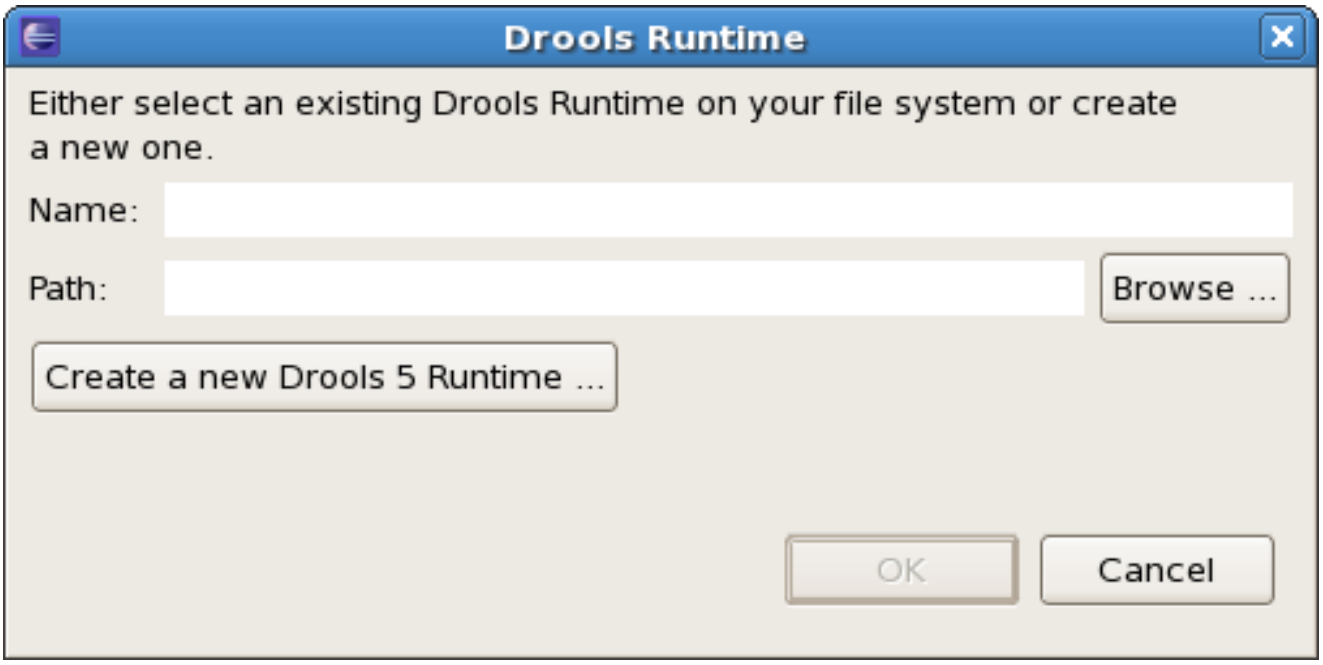
A Drools runtime is a collection of JARs on your file system that represent one specific release of the Drools project JARs. To create a runtime, you must point the IDE to the release of your choice. If you want to create a new runtime based on the latest Drools project JARs included in the plugin itself, you can also easily do that. You are required to specify a default Drools runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

1.3.1.4.4.1. Defining a Drools runtime

You are required to define one or more Drools runtimes using the Eclipse preferences view. To open up your preferences, in the menu Window select the Preferences menu item. A new preferences dialog should show all your preferences. On the left side of this dialog, under the Drools category, select "Installed Drools runtimes". The panel on the right should then show the currently defined Drools runtimes. If you have not yet defined any runtimes, it should like something like the figure below.



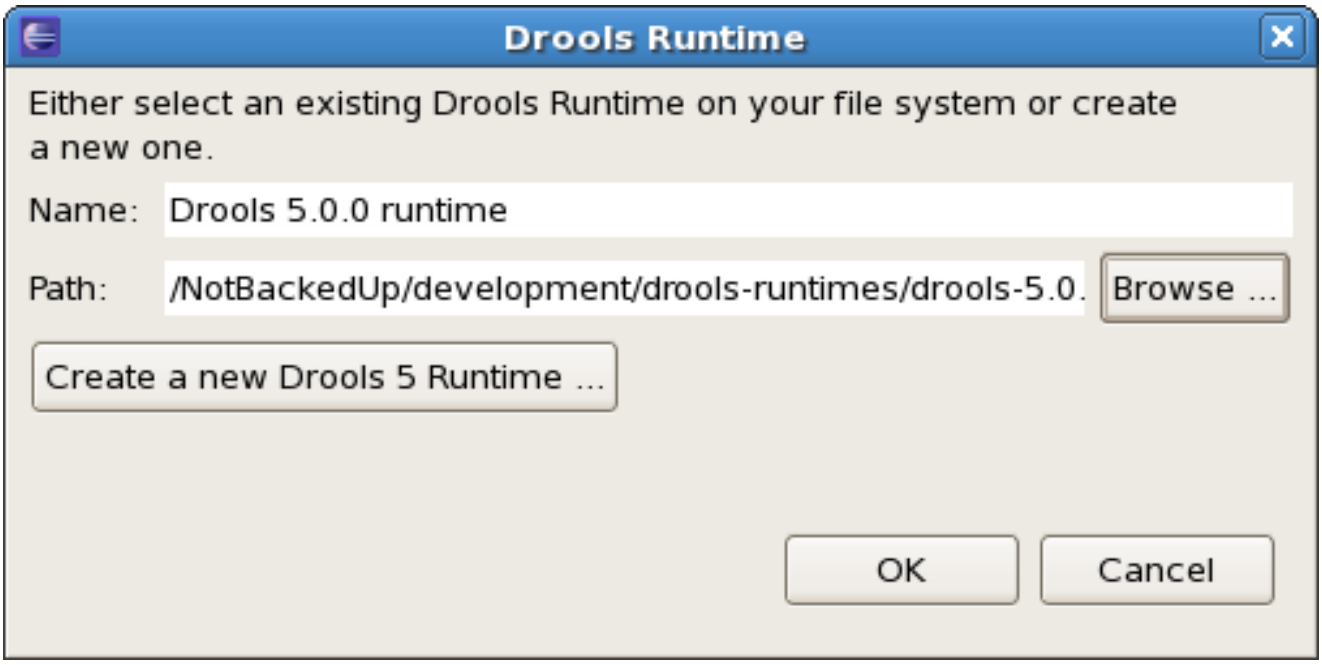
To define a new Drools runtime, click on the add button. A dialog as shown below should pop up, requiring the name for your runtime and the location on your file system where it can be found.



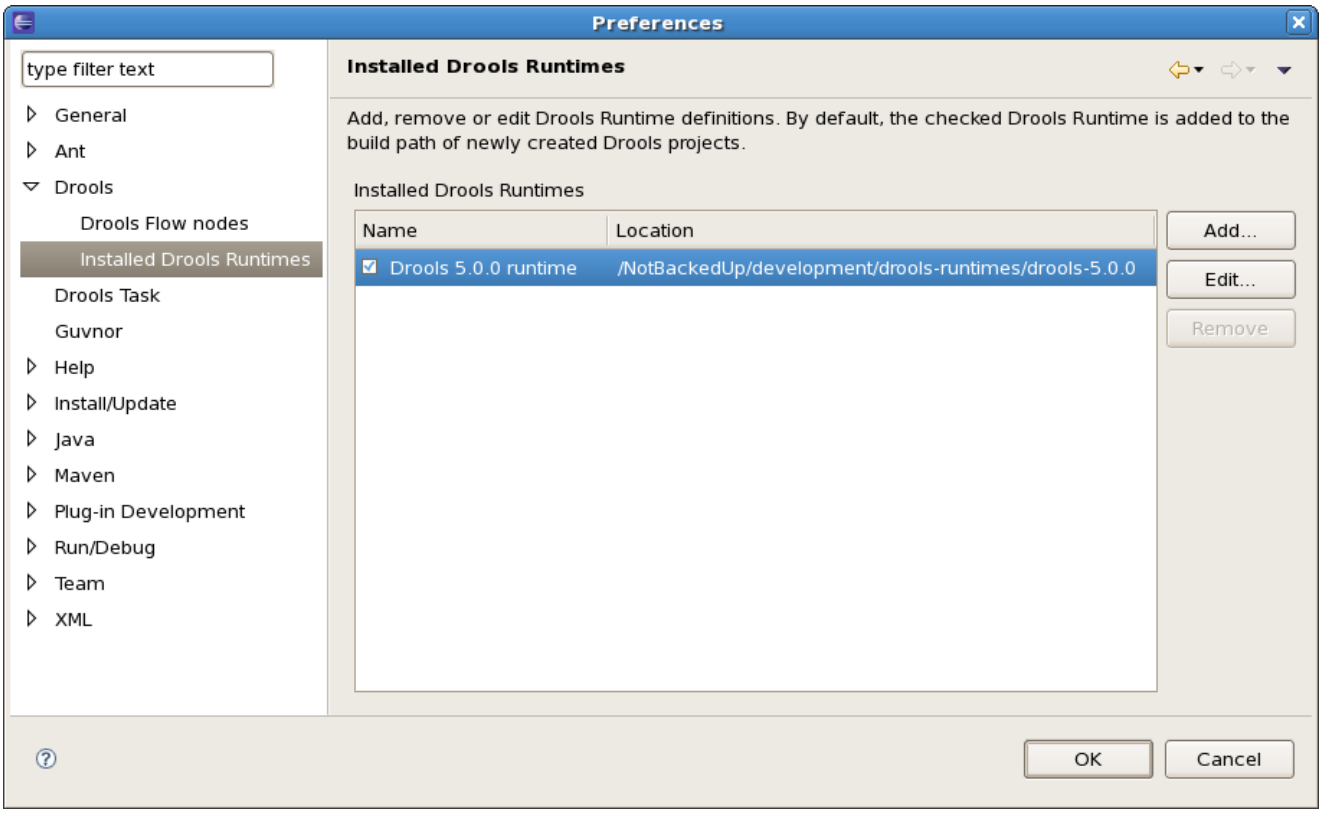
In general, you have two options:

1. If you simply want to use the default JARs as included in the Drools Eclipse plugin, you can create a new Drools runtime automatically by clicking the "Create a new Drools 5 runtime ..." button. A file browser will show up, asking you to select the folder on your file system where you want this runtime to be created. The plugin will then automatically copy all required dependencies to the specified folder. After selecting this folder, the dialog should look like the figure shown below.

2. If you want to use one specific release of the Drools project, you should create a folder on your file system that contains all the necessary Drools libraries and dependencies. Instead of creating a new Drools runtime as explained above, give your runtime a name and select the location of this folder containing all the required JARs.

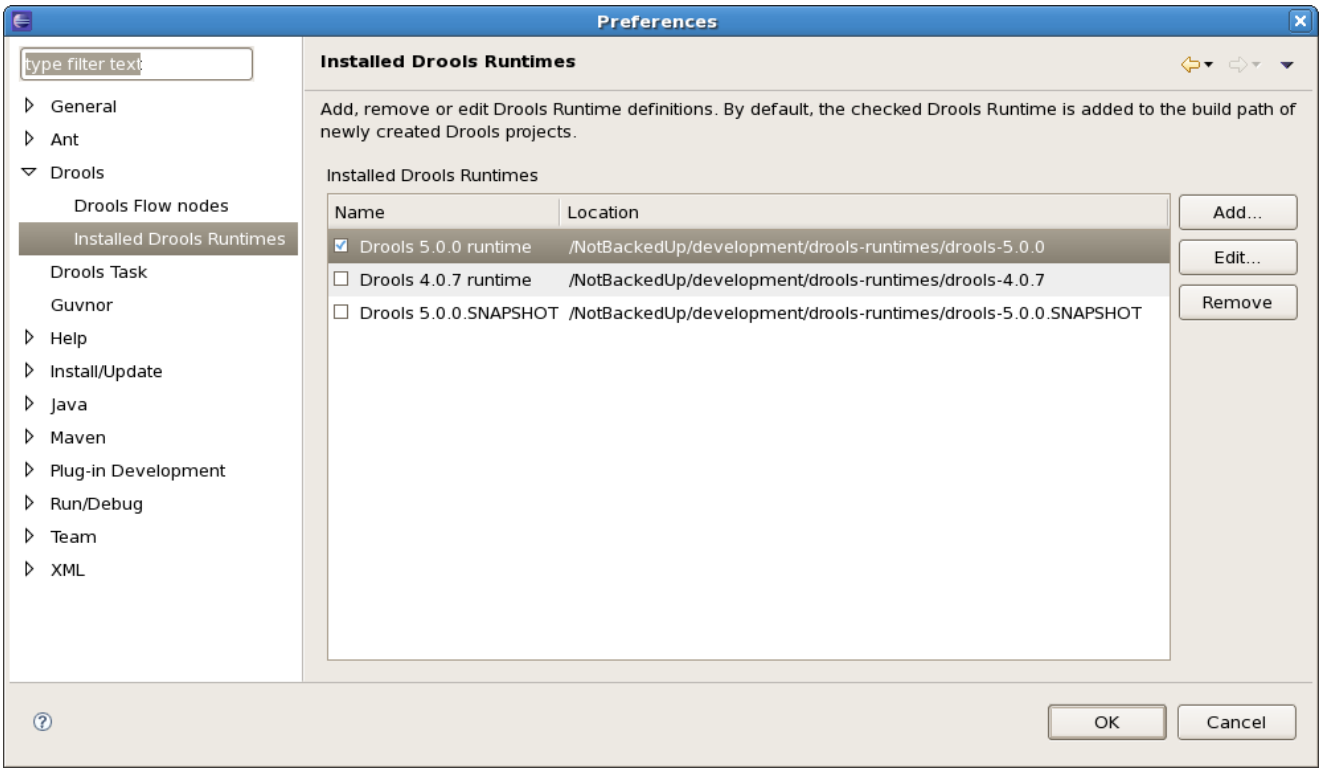


After clicking the OK button, the runtime should show up in your table of installed Drools runtimes, as shown below. Click on checkbox in front of the newly created runtime to make it the default Drools runtime. The default Drools runtime will be used as the runtime of all your Drools project that have not selected a project-specific runtime.



You can add as many Drools runtimes as you need. For example, the screenshot below shows a configuration where three runtimes have been defined: a Drools 4.0.7 runtime, a Drools 5.0.0

runtime and a Drools 5.0.0.SNAPSHOT runtime. The Drools 5.0.0 runtime is selected as the default one.

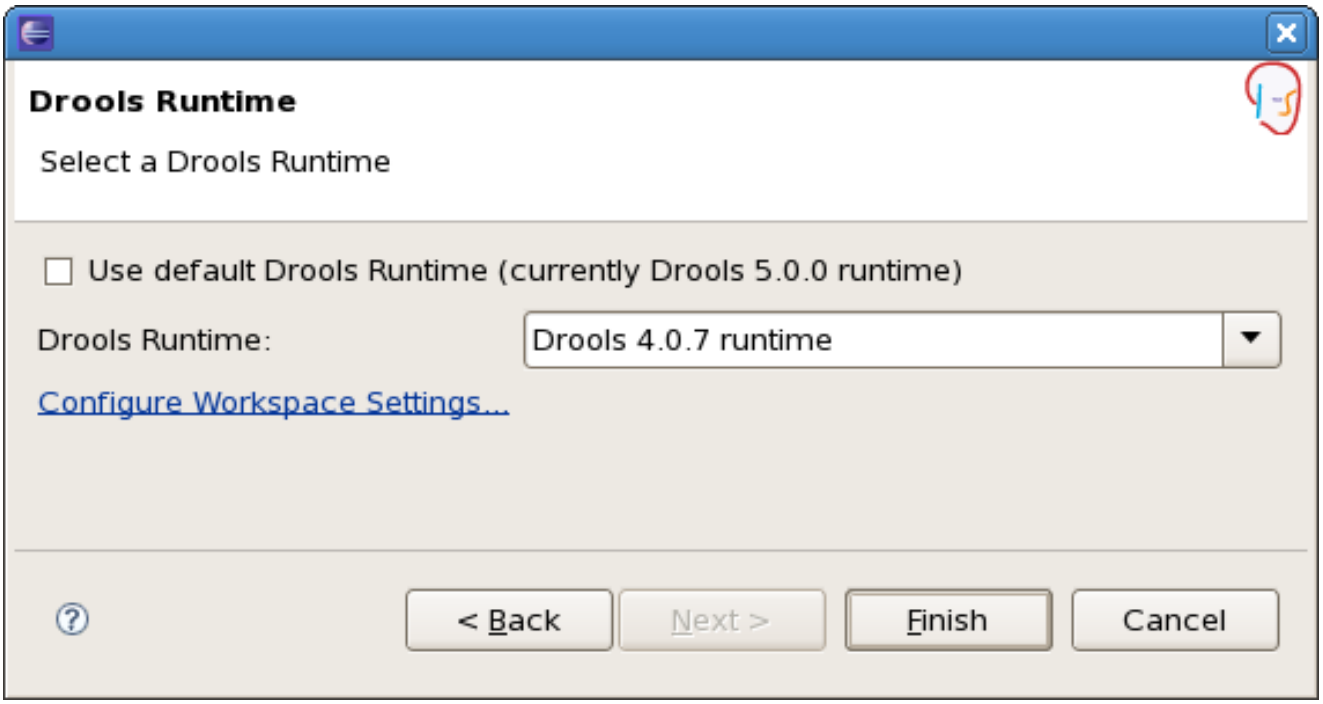


Note that you will need to restart Eclipse if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

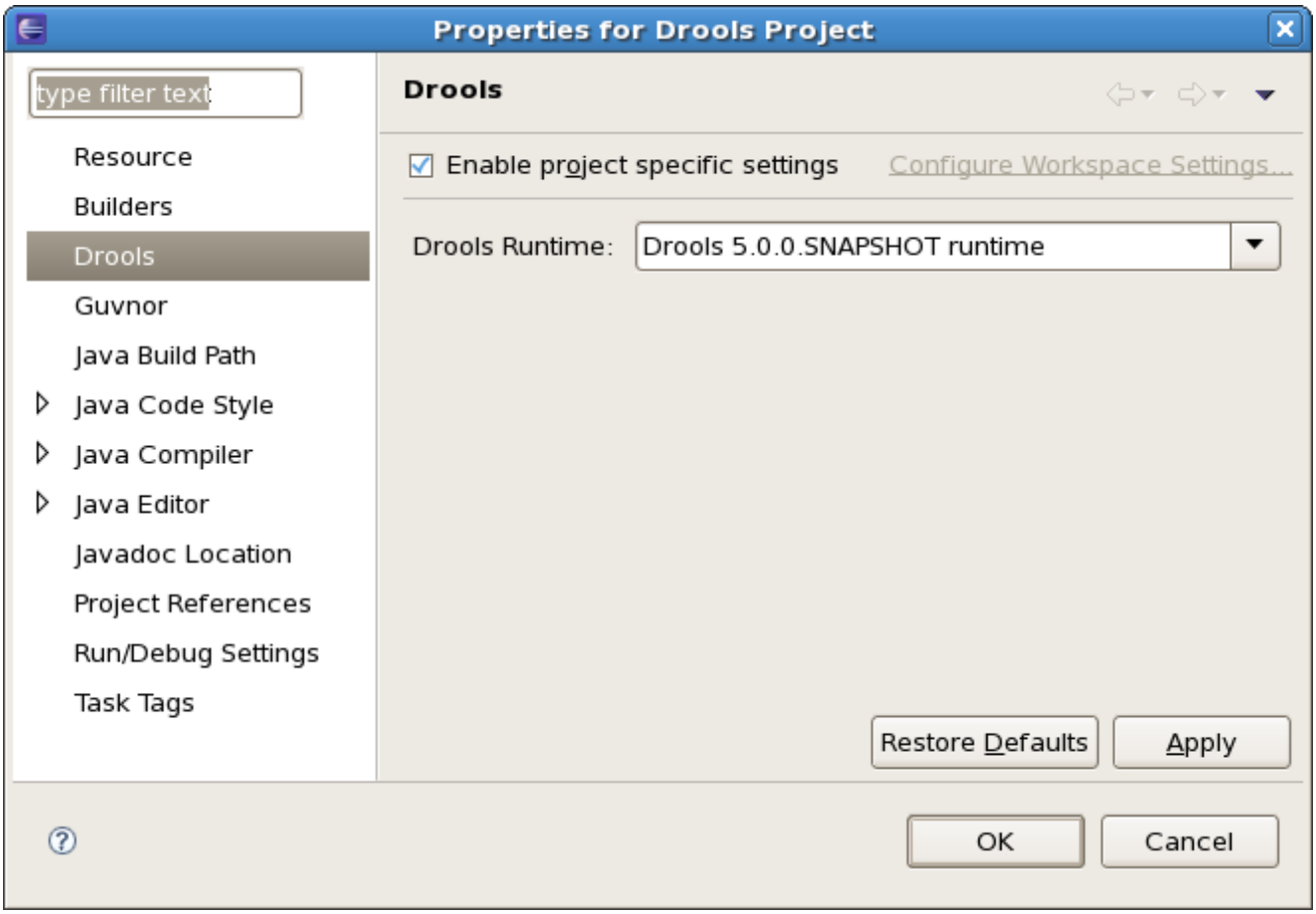
1.3.1.4.4.2. Selecting a runtime for your Drools project

Whenever you create a Drools project (using the New Drools Project wizard or by converting an existing Java project to a Drools project using the "Convert to Drools Project" action that is shown when you are in the Drools perspective and you right-click an existing Java project), the plugin will automatically add all the required JARs to the classpath of your project.

When creating a new Drools project, the plugin will automatically use the default Drools runtime for that project, unless you specify a project-specific one. You can do this in the final step of the New Drools Project wizard, as shown below, by deselecting the "Use default Drools runtime" checkbox and selecting the appropriate runtime in the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there.



You can change the runtime of a Drools project at any time by opening the project properties (right-click the project and select Properties) and selecting the Drools category, as shown below. Check the "Enable project specific settings" checkbox and select the appropriate runtime from the drop-down box. If you click the "Configure workspace settings ..." link, the workspace preferences showing the currently installed Drools runtimes will be opened, so you can add new runtimes there. If you deselect the "Enable project specific settings" checkbox, it will use the default runtime as defined in your global preferences.



1.3.2. Building from source

1.3.2.1. Getting the sources

The source code of each Maven artifact is available in the JBoss Maven repository as a source JAR. The same source JARs are also included in the download zips. However, if you want to build from source, it's highly recommended to get our sources from our source control.

Drools and jBPM use Git [http://git-scm.com/] for source control. The blessed git repositories are hosted on GitHub [https://github.com]:

- https://github.com/droolsjbpm

Git allows you to fork our code, independently make personal changes on it, yet still merge in our latest changes regularly and optionally share your changes with us. To learn more about git, read the free book Git Pro [http://progit.org/book/].

1.3.2.2. Building the sources

In essense, building from source is very easy, for example if you want to build the *guvnor* project:

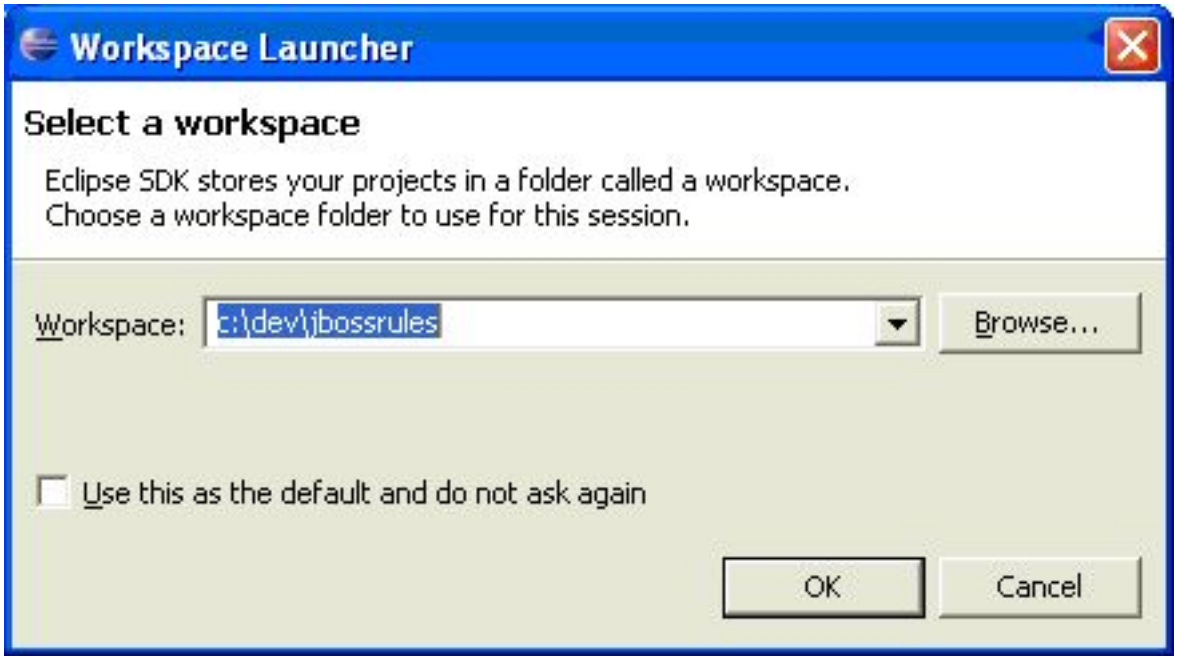
```
$ git clone git@github.com:droolsjbpm/guvnor.git
...
$ cd guvnor
$ mvn clean install -DskipTests -Dfull
...
```

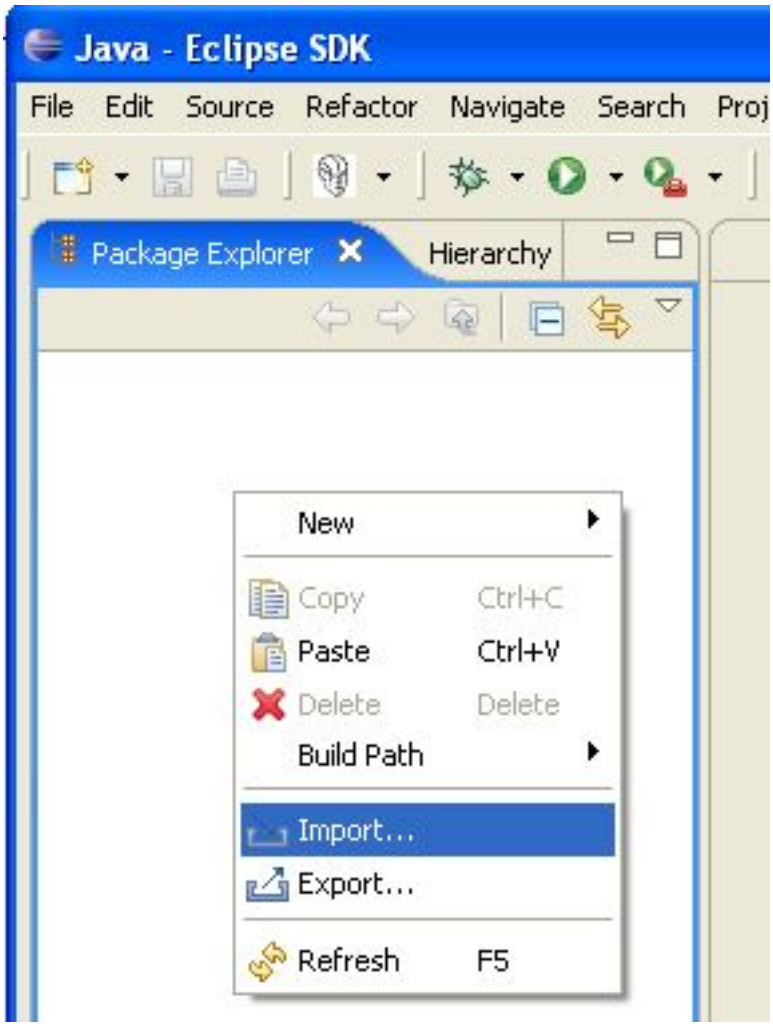
However, *there are a lot potential pitfalls*, so if you're serious about building from source and possibly contributing to the project, **follow the instructions in the README file in droolsjbpm-build-bootstrap [https://github.com/droolsjbpm/droolsjbpm-build-bootstrap/blob/master/README.md].**

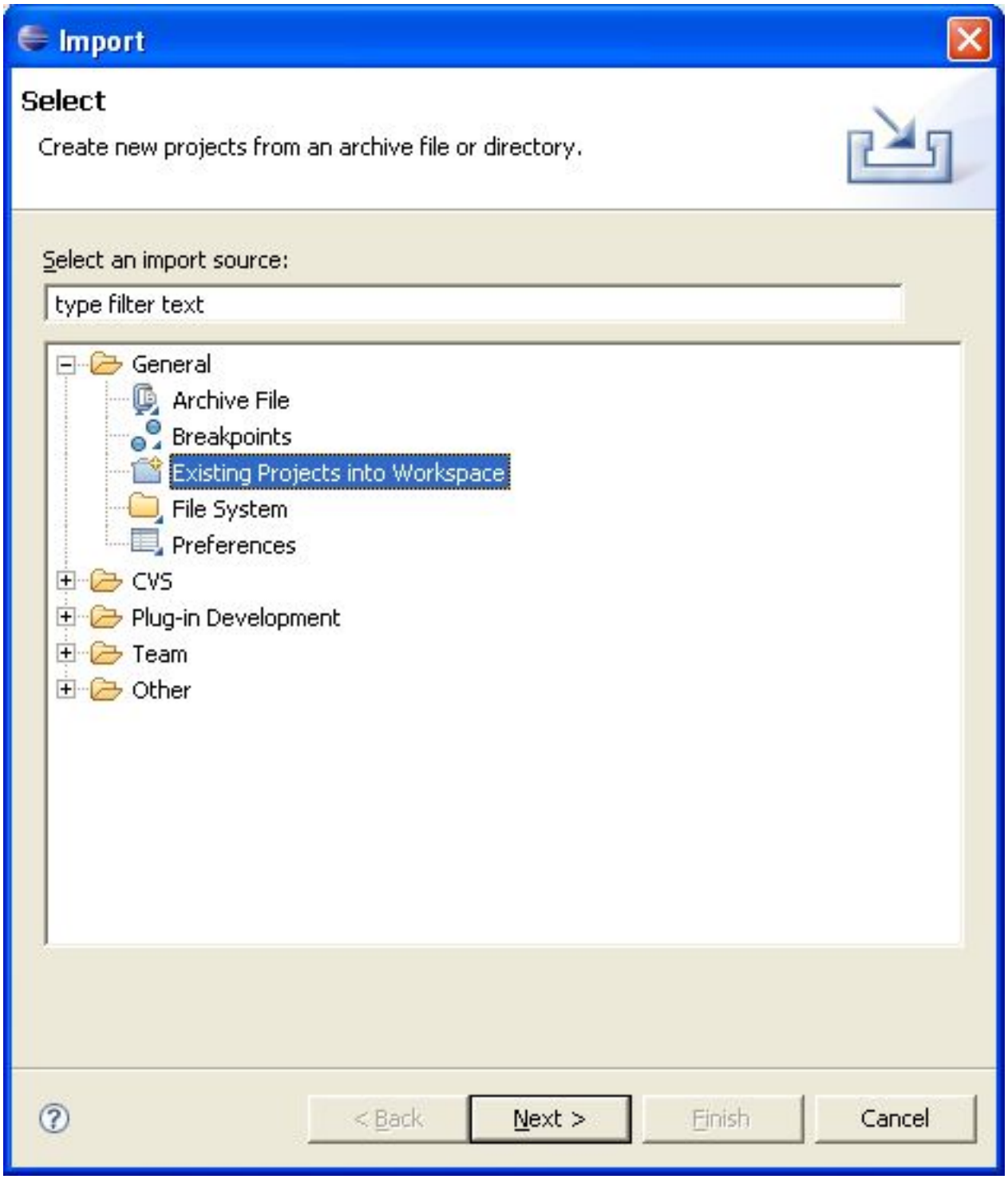
1.3.3. Eclipse

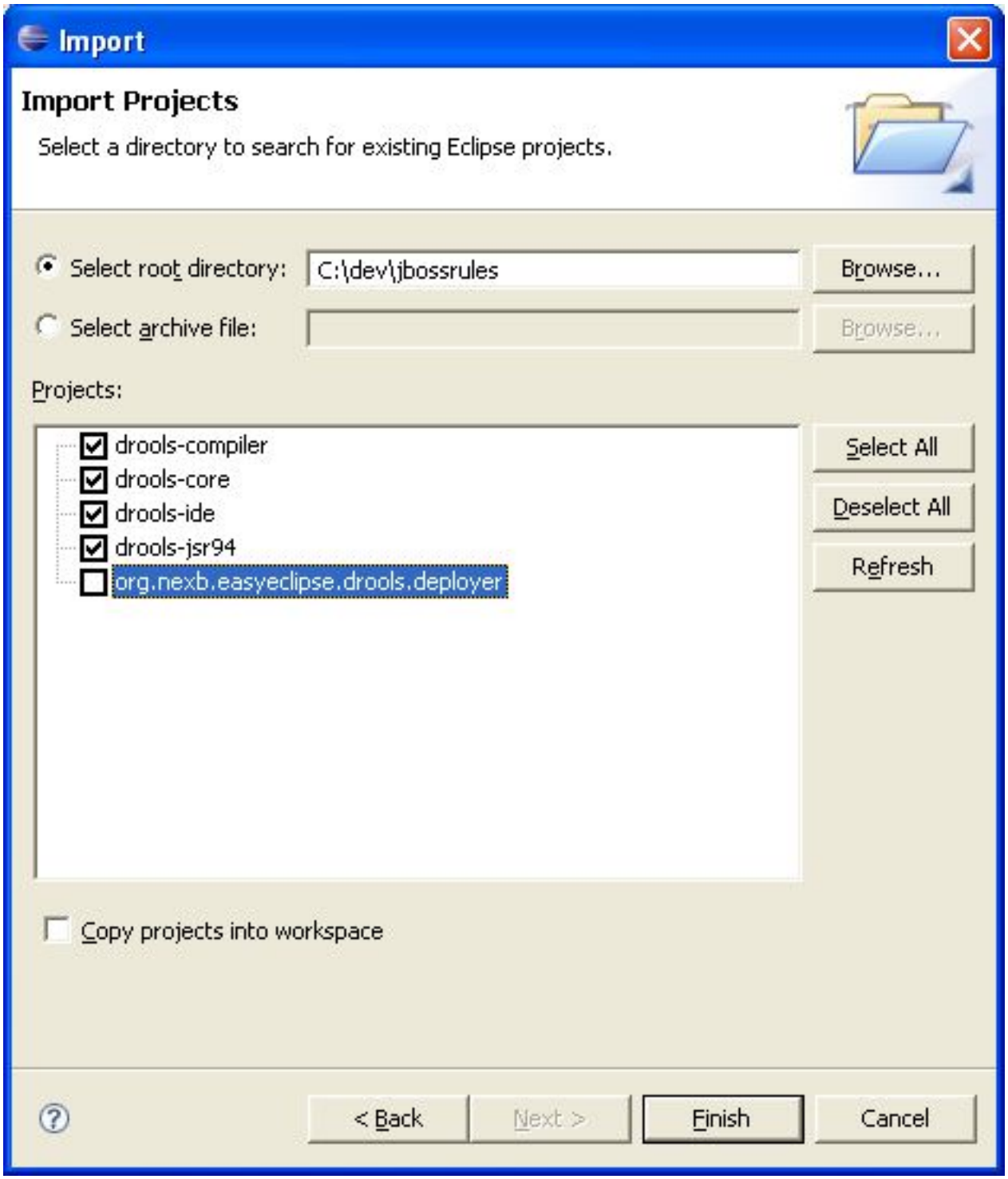
1.3.3.1. Importing Eclipse Projects

With the Eclipse project files generated they can now be imported into Eclipse. When starting Eclipse open the workspace in the root of your subversion checkout.

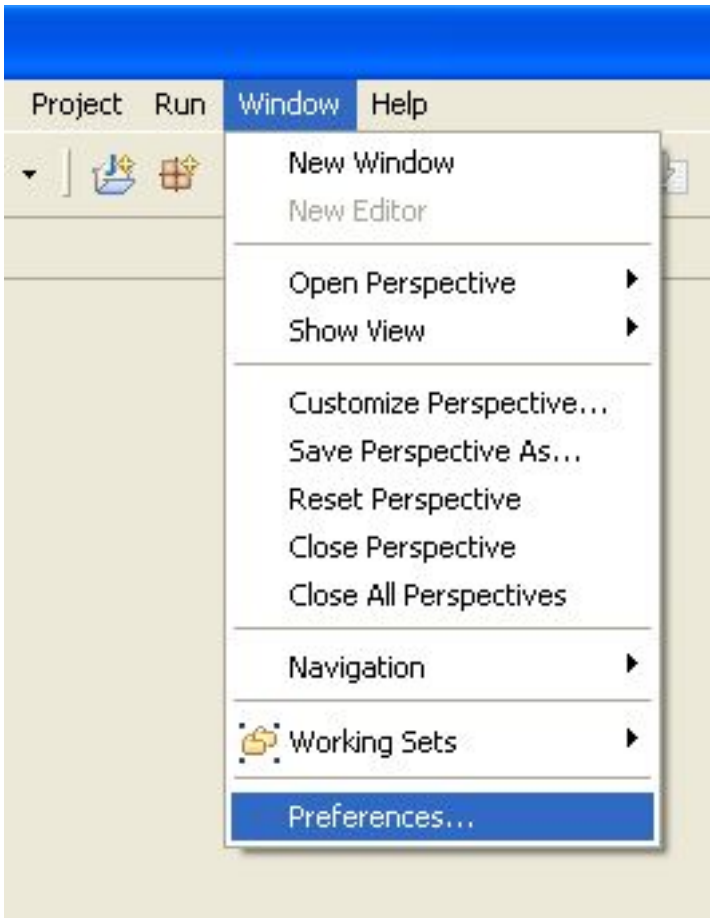


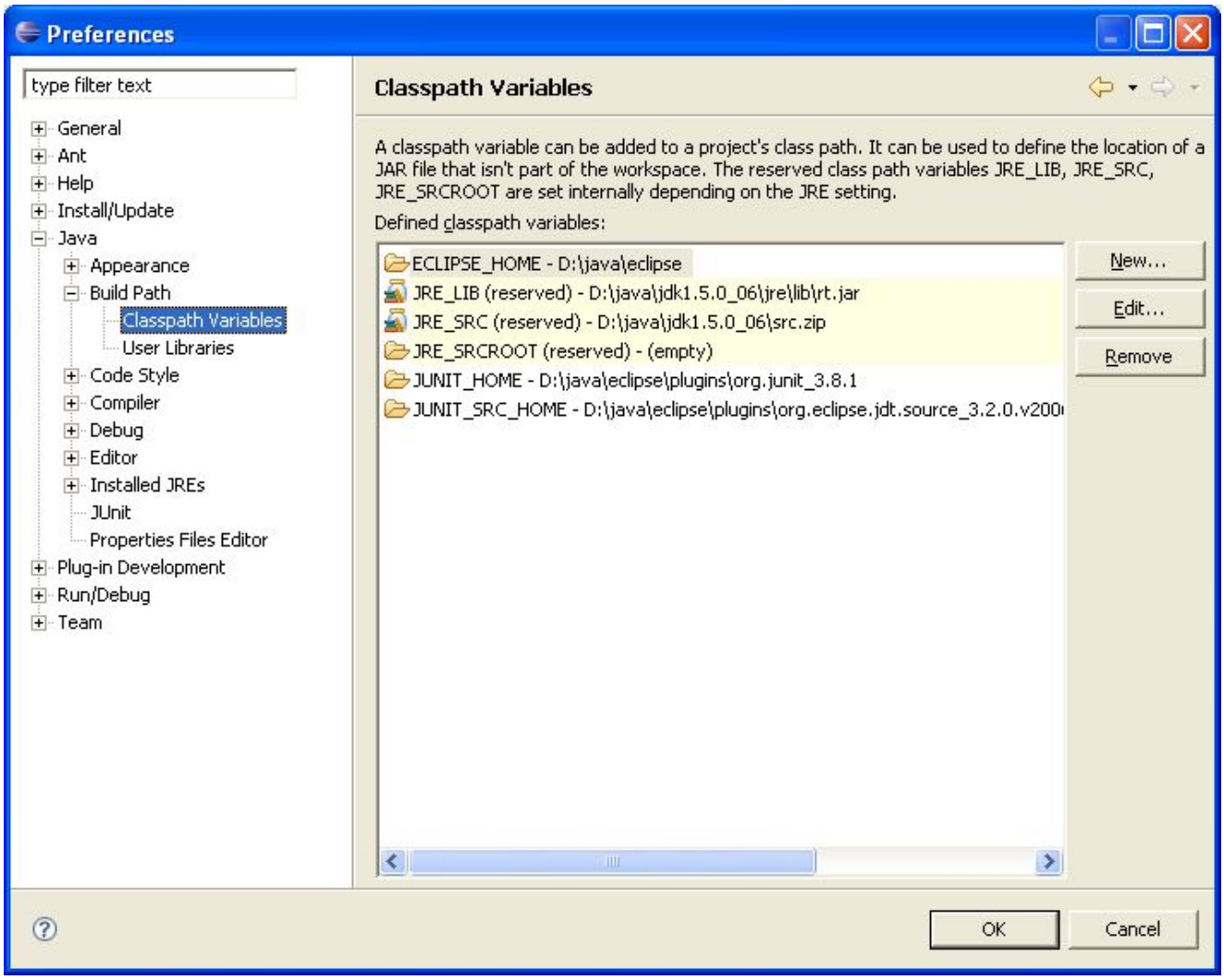

Java - Eclipse SDK
File
Edit
Source
Refactor
Navigate
Search
Proj
Package Explorer
Hierarchy
New
Copy
Ctrl+C
Paste
Ctrl+V
Delete
Delete
Build Path
Import...
Export...
Refresh
F5

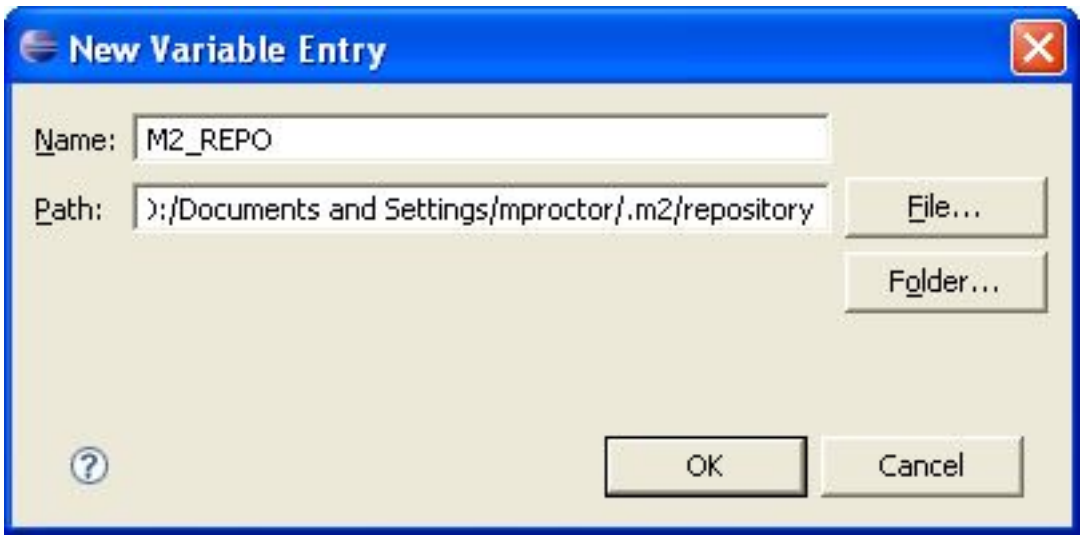

Import
Select
Create new projects from an archive file or directory.
Select an import source:
type filter text
General
Archive File
Breakpoints
Existing Projects into Workspace
File System
Preferences
CVS
Plug-in Development
Team
Other
< Back
Next >
Finish
Cancel

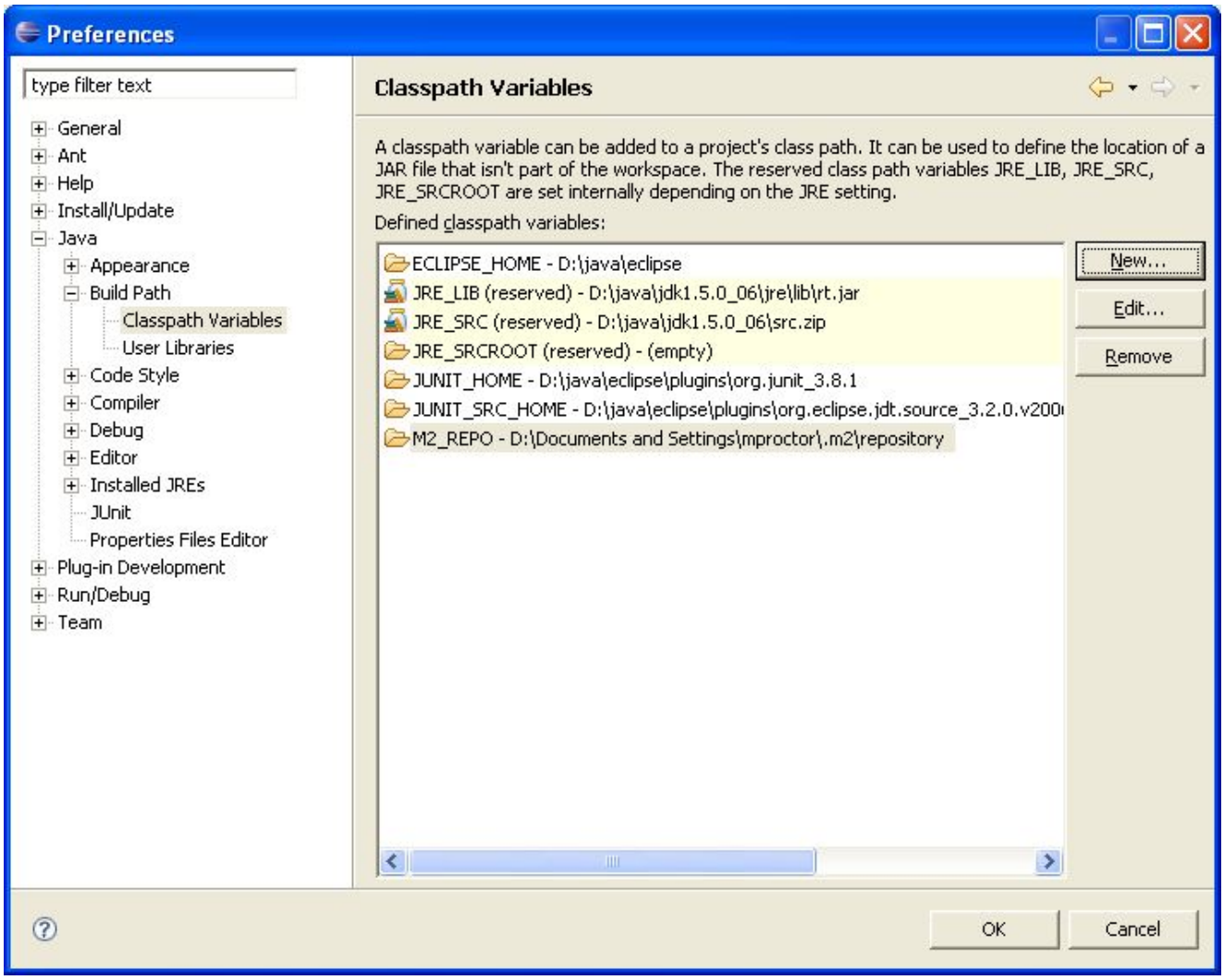


When calling **mvn install** all the project dependencies were downloaded and added to the local Maven repository. Eclipse cannot find those dependencies unless you tell it where that repository is. To do this setup an `M2_REPO` classpath variable.


Project
Run
Window
Help
New Window
New Editor
Open Perspective
Show View
Customize Perspective...
Save Perspective As...
Reset Perspective
Close Perspective
Close All Perspectives
Navigation
Working Sets
Preferences...


Preferences
type filter text
General
Ant
Help
Install/Update
Java
Appearance
Build Path
Classpath Variables
User Libraries
Code Style
Compiler
Debug
Editor
Installed JREs
JUnit
Properties Files Editor
Plug-in Development
Run/Debug
Team
Classpath Variables
A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. The reserved class path variables JRE_LIB, JRE_SRC, JRE_SRCROOT are set internally depending on the JRE setting.
Defined classpath variables:
ECLIPSE_HOME - D:\java\eclipse
JRE_LIB (reserved) - D:\java\jdk1.5.0_06\jre\lib\rt.jar
JRE_SRC (reserved) - D:\java\jdk1.5.0_06\src.zip
JRE_SRCROOT (reserved) - (empty)
JUNIT_HOME - D:\java\eclipse\plugins\org.junit_3.8.1
JUNIT_SRC_HOME - D:\java\eclipse\plugins\org.eclipse.jdt.source_3.2.0.v200
New...
Edit...
Remove
OK
Cancel


New Variable Entry
Name:
M2_REPO
Path:
):/Documents and Settings/mproctor/.m2/repository
File...
Folder...
OK
Cancel


Preferences
type filter text
General
Ant
Help
Install/Update
Java
Appearance
Build Path
Classpath Variables
User Libraries
Code Style
Compiler
Debug
Editor
Installed JREs
JUnit
Properties Files Editor
Plug-in Development
Run/Debug
Team
Classpath Variables
A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. The reserved class path variables JRE_LIB, JRE_SRC, JRE_SRCROOT are set internally depending on the JRE setting.
Defined classpath variables:
ECLIPSE_HOME - D:\java\eclipse
JRE_LIB (reserved) - D:\java\jdk1.5.0_06\jre\lib\rt.jar
JRE_SRC (reserved) - D:\java\jdk1.5.0_06\src.zip
JRE_SRCROOT (reserved) - (empty)
JUNIT_HOME - D:\java\eclipse\plugins\org.junit_3.8.1
JUNIT_SRC_HOME - D:\java\eclipse\plugins\org.eclipse.jdt.source_3.2.0.v200
M2_REPO - D:\Documents and Settings\mproctor\.m2\repository
New...
Edit...
Remove
OK
Cancel

Chapter 2. Release Notes

2.1. What is New and Noteworthy in Drools 6.3.0

2.1.1. Real Time Validation and Verification for the Decision Tables

Decision tables used to have a Validation-button for validating the table. This is now removed and the table is validated after each cell value change. The validation and verification checks include:

- Redundancy
- Subsumption
- Conflicts
- Missing Columns

These checks are explained in detail in the workbench documentation.

2.1.2. Improved DRL Editor

The DRL Editor has undergone a face lift; moving from a plain TextArea to using ACE Editor and a custom DRL syntax highlighter.

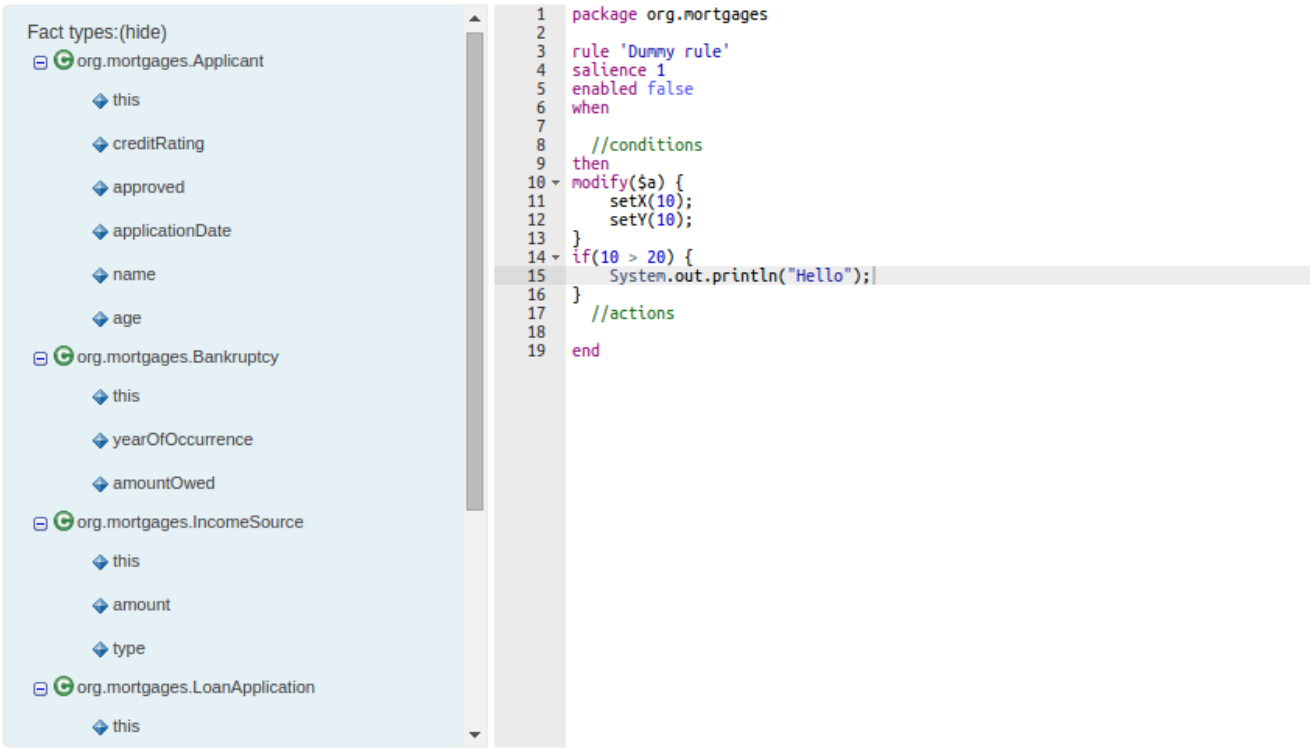


Figure 2.1. ACE Editor

2.1.3. Browsing graphs of objects with OOPath

Warning

This feature is experimental

When the field of a fact is a collection it is possible to bind and reason over all the items in that collection on by one using the `from` keyword. Nevertheless, when it is required to browse a graph of object the extensive use of the `from` conditional element may result in a verbose and cubersome syntax like in the following example:

Example 2.1. Browsing a graph of objects with from

```
rule "Find all grades for Big Data exam" when    $student: Student( $plan: plan )    $exam:
 Exam( course == "Big Data" ) from $plan.exams    $grade: Grade() from $exam.gradesthen /*
 RHS */ end
 when    $student: Student( $plan: plan
 )    $exam: Exam( course == "Big Data" ) from
 $plan.exams    $grade: Grade() from
 $exam.gradesthen /* RHS */
```

In this example it has been assumed to use a domain model consisting of a `Student` who has a `Plan` of study: a `Plan` can have zero or more `Exams` and an `Exam` zero or more `Grades`. Note that only the root object of the graph (the `Student` in this case) needs to be in the working memory in order to make this works.

By borrowing ideas from XPath, this syntax can be made more succinct, as XPath has a compact notation for navigating through related elements while handling collections and filtering constraints. This XPath-inspired notation has been called `OOPath` since it is explictly intended to browse graph of objects. Using this notation the former example can be rewritten as it follows:

Example 2.2. Browsing a graph of objects with OOPath

```
rule "Find all grades for Big Data exam" when    Student( $grade: /plan/exams{course == "Big
 Data"}/grades )then /* RHS */ end
 when    Student( $grade: /plan/exams{course == "Big Data"}/grades
 )then /* RHS */
```

Formally, the core grammar of an `OOPath` expression can be defined in EBNF notation in this way.

```
OOPExpr = "/" OOPSegment { ( "/" | "." ) OOPSegment } ;OOPSegment = [ID ( ":" | ":=" )] ID
 ["[" Number "]"] ["{" Constraints "}"];
 } ;OOPSegment = [ID ( ":" | ":=" )] ID ["[" Number "]"] ["{"
```

In practice an `OOPath` expression has the following features.

- It has to start with `/`.
- It can dereference a single property of an object with the `.` operator
- It can dereference a multiple property of an object using the `/` operator. If a collection is returned, it will iterate over the values in the collection
- While traversing referenced objects it can filter away those not satisfying one or more constraints, written as predicate expressions between curly brackets like in:

```
Student( $grade: /plan/exams{course == "Big Data"}/grades )
```

- Items can also be accessed by their index by putting it between square brackets like in:

```
Student( $grade: /plan/exams[0]/grades )
```

To adhere to Java convention OOPath indexes are 0-based, compared to XPath 1-based

2.1.3.1. Reactive OOPath

At the moment Drools is not able to react to updates involving a deeply nested traversed during the evaluation of an `OOPath` expression. To make these objects reactive to changes at the moment it is necessary to make them extend the class `org.drools.core.phreak.ReactiveObject`. It is planned to overcome this limitation by implementing a mechanism that automatically instruments the classes belonging to a specific domain model.

Having extendend that class, the domain objects can notify the engine when one of its field has been updated by invoking the inherited method `notifyModification` as in the following example:

Example 2.3. Notifying the engine that an exam has been moved to a different course

```
public void setCourse(String course) {    this.course = course;    notifyModification(this);}
{    this.course =
course;
notifyModification(this);
```

In this way if an exam is moved to a different course, the rule is re-triggered and the list of grades matching the rule recomputed.

2.1.4. Kie Navigator View for Eclipse

A new viewer has been added to the Eclipse Tooling. This Kie Navigator View is used to manage Kie Server installations and projects.

Please read the chapter **Kie Navigator View** for more information about this new feature

2.2. New and Noteworthy in KIE Workbench 6.3.0

2.2.1. Asset locking

To avoid conflicts when editing assets, a new locking mechanism has been introduced that makes sure that only one user at a time can edit an asset. When a user begins to edit an asset, a lock will automatically be acquired. This is indicated by a lock symbol appearing on the asset title bar as well as in the project explorer view. If a user starts editing an already locked asset a pop-up notification will appear to inform the user that the asset can't currently be edited, as it is being worked on by another user. As long as the editing user holds the lock, changes by other users will be prevented. Locks will automatically be released when the editing user saves or closes the asset, or logs out of the workbench. Every user further has the option to force a lock release in the metadata tab, if required.

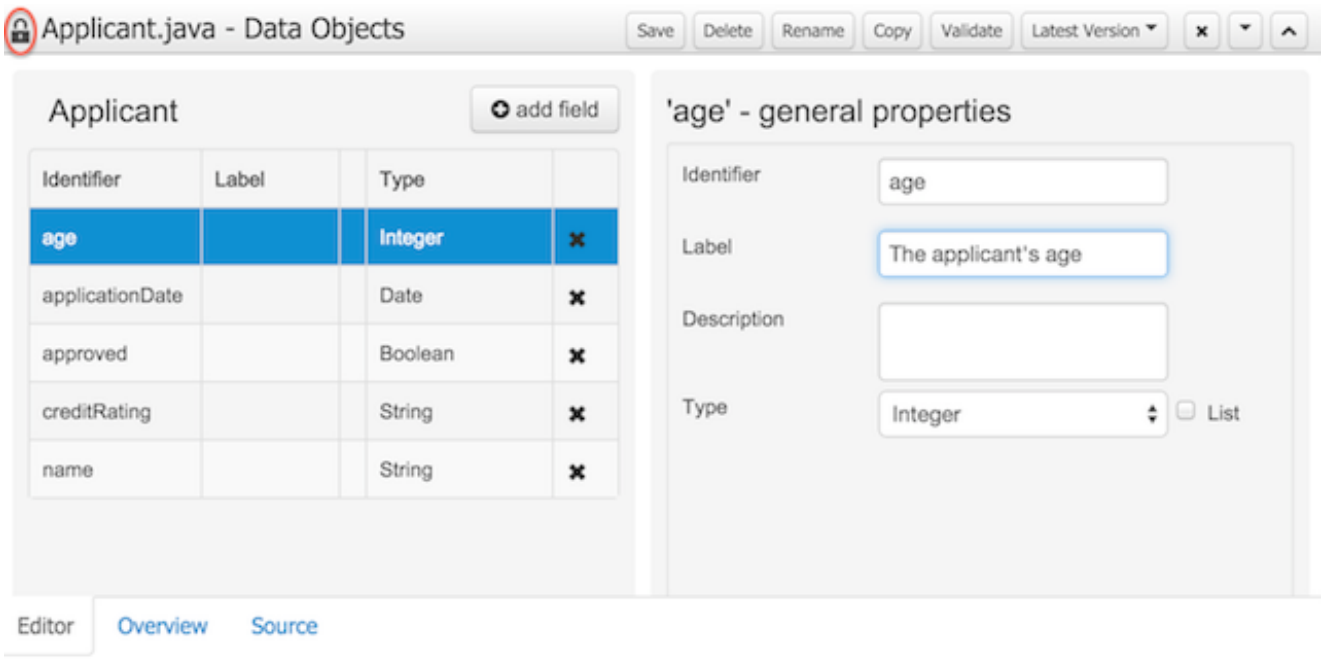


Figure 2.2. Editing an asset automatically acquires a lock

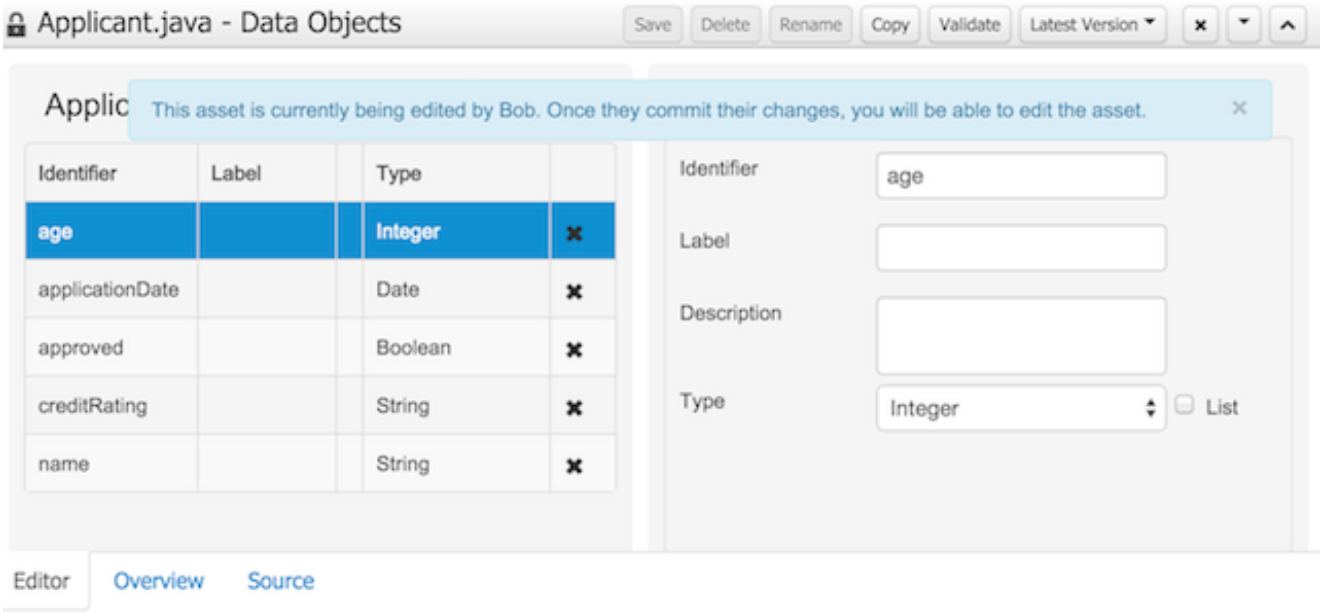


Figure 2.3. Locked assets cannot be edited by other users

2.2.2. Data Modeller Tool Windows

Drools and jBPM configurations, Persistence (see Generation of JPA enabled Data Models) and Advanced configurations were moved into "Tool Windows". "Tool Windows" are a new concept introduced in latest Uberfire version that enables the development of context aware screens. Each "Tool Window" will contain a domain editor that will manage a set of related Data Object parameters.

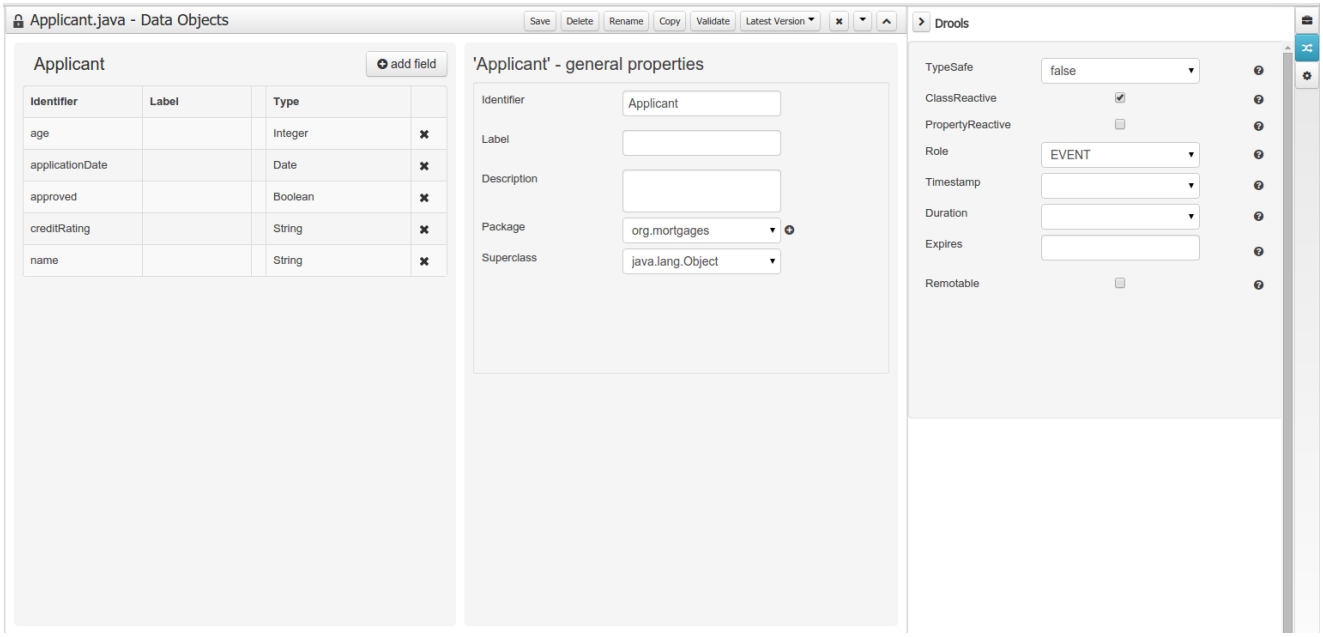


Figure 2.4. Drools and jBPM domain tool window

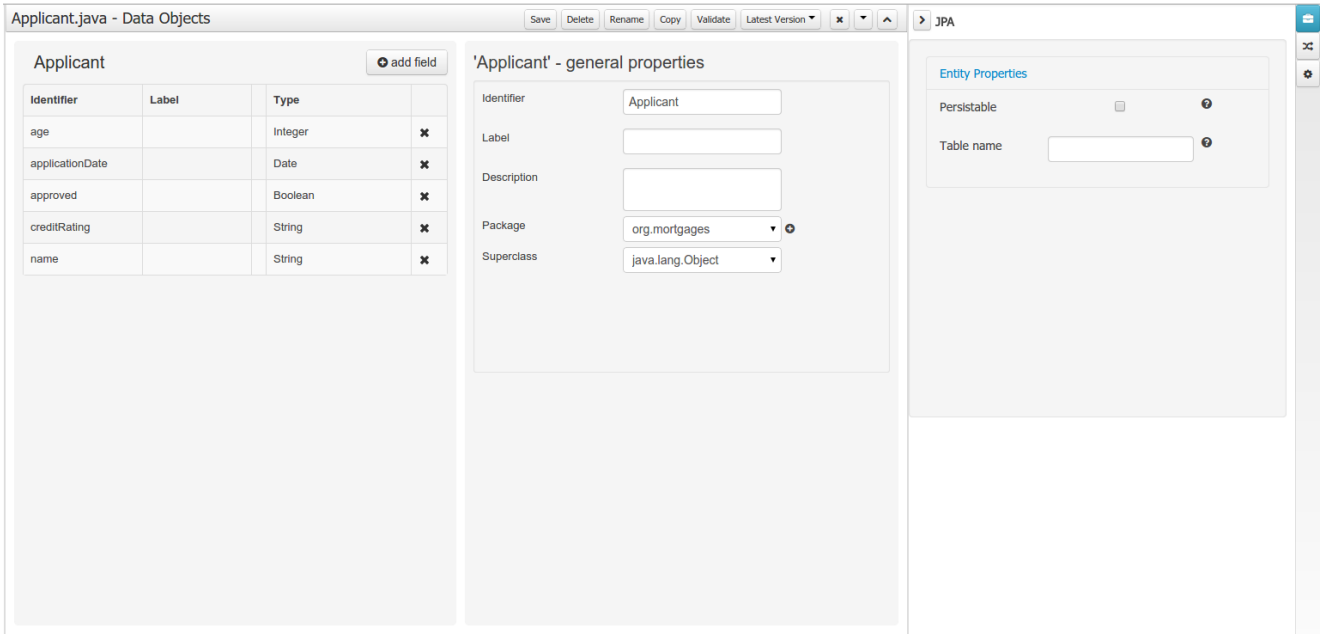


Figure 2.5. Persistence tool window

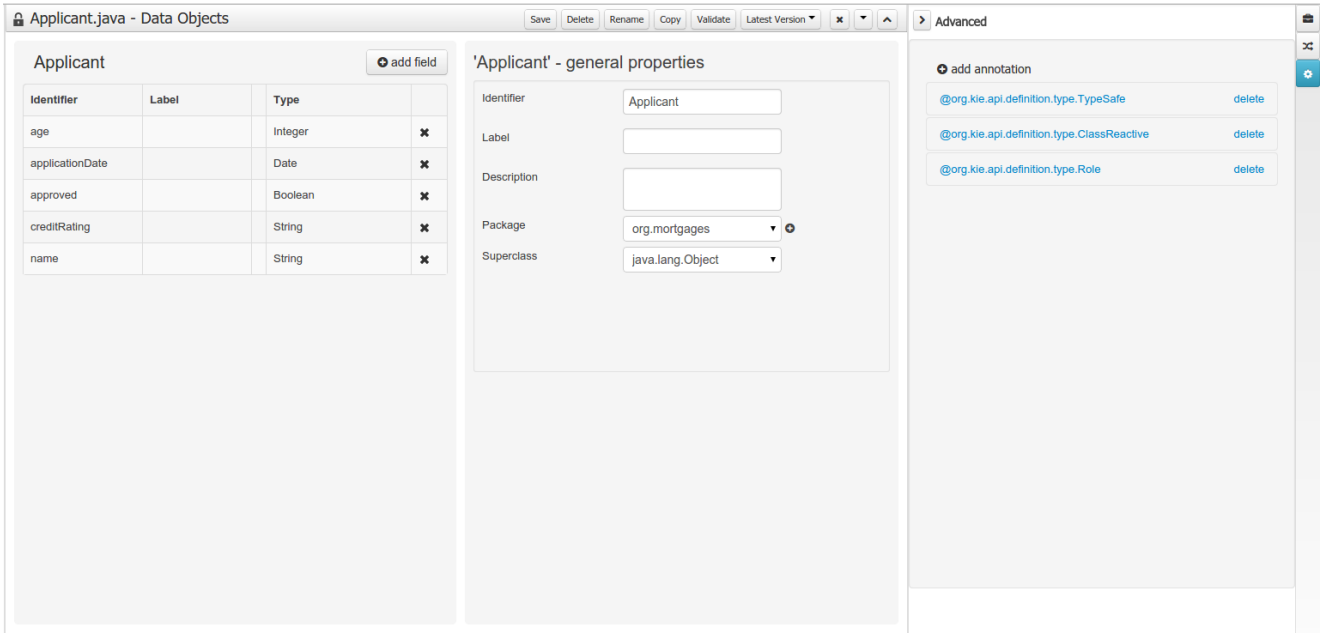


Figure 2.6. Advanced configurations tool window

2.2.3. Generation of JPA enabled Data Models

Data modeller was extended to support the generation of persistable Data Objects. The persistable Data Objects are based on the JPA specification and all the underlying metadata are automatically generated.

- "The New -> Data Object" Data Objects can be marked as persistable at creation time.

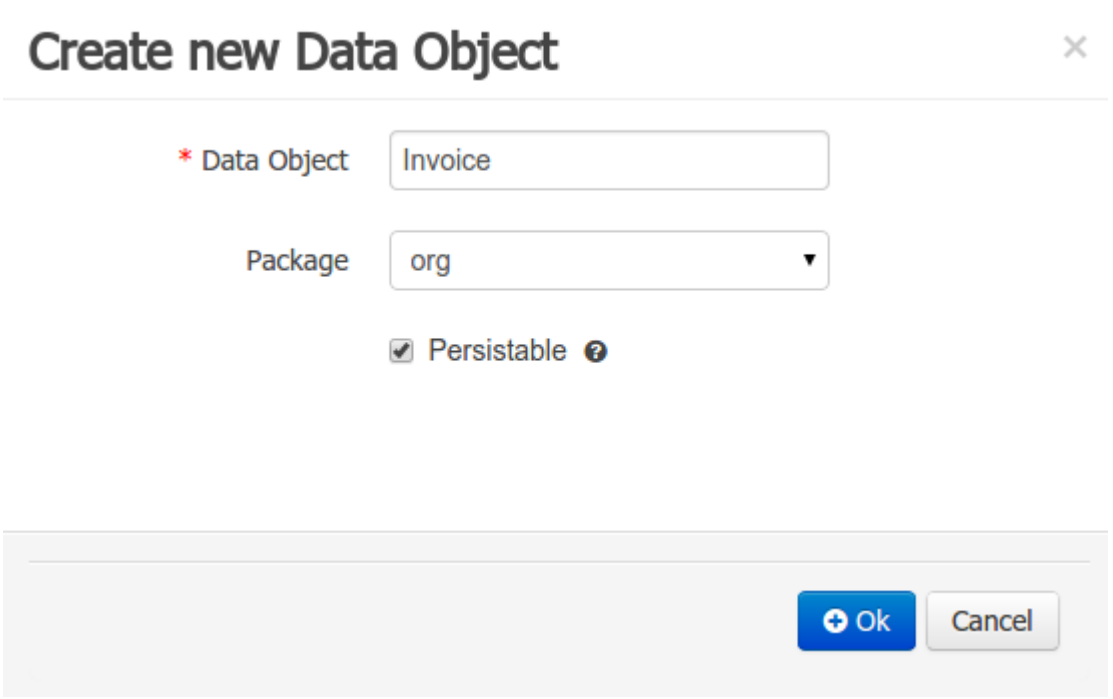


Figure 2.7. New Data Object

- The Persistence tool window contains the JPA Domain editors for both Data Object and Field. Each editor will manage the by default generated JPA metadata

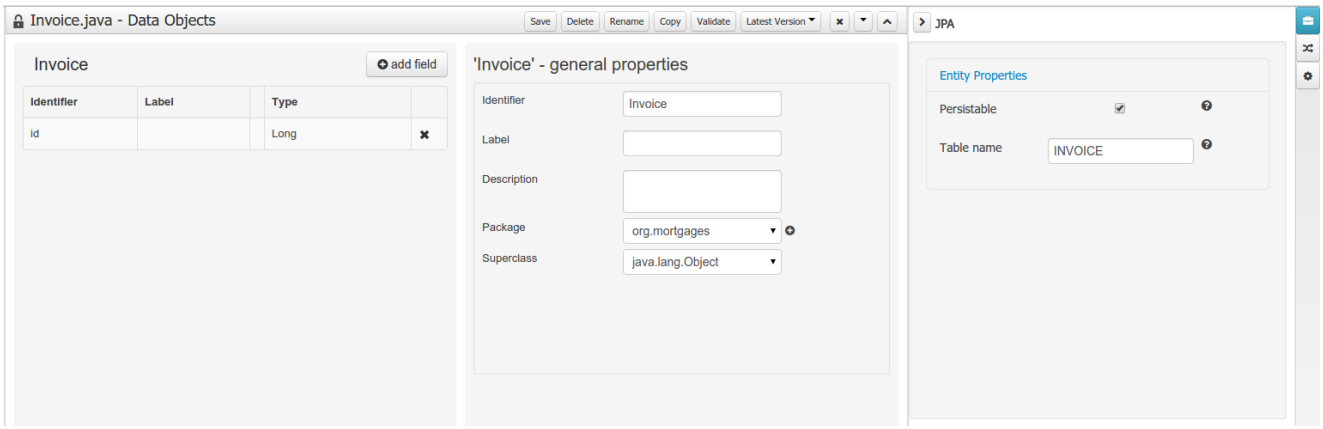


Figure 2.8. Data Object level JPA domain editor

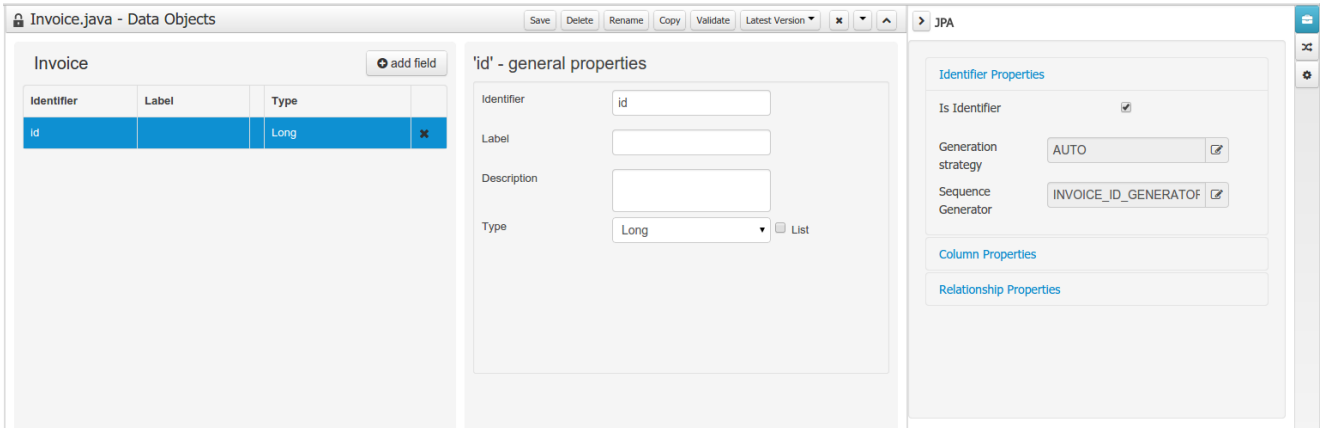


Figure 2.9. Field level JPA domain editor

- Persistence configuration screen was added to the project editor.

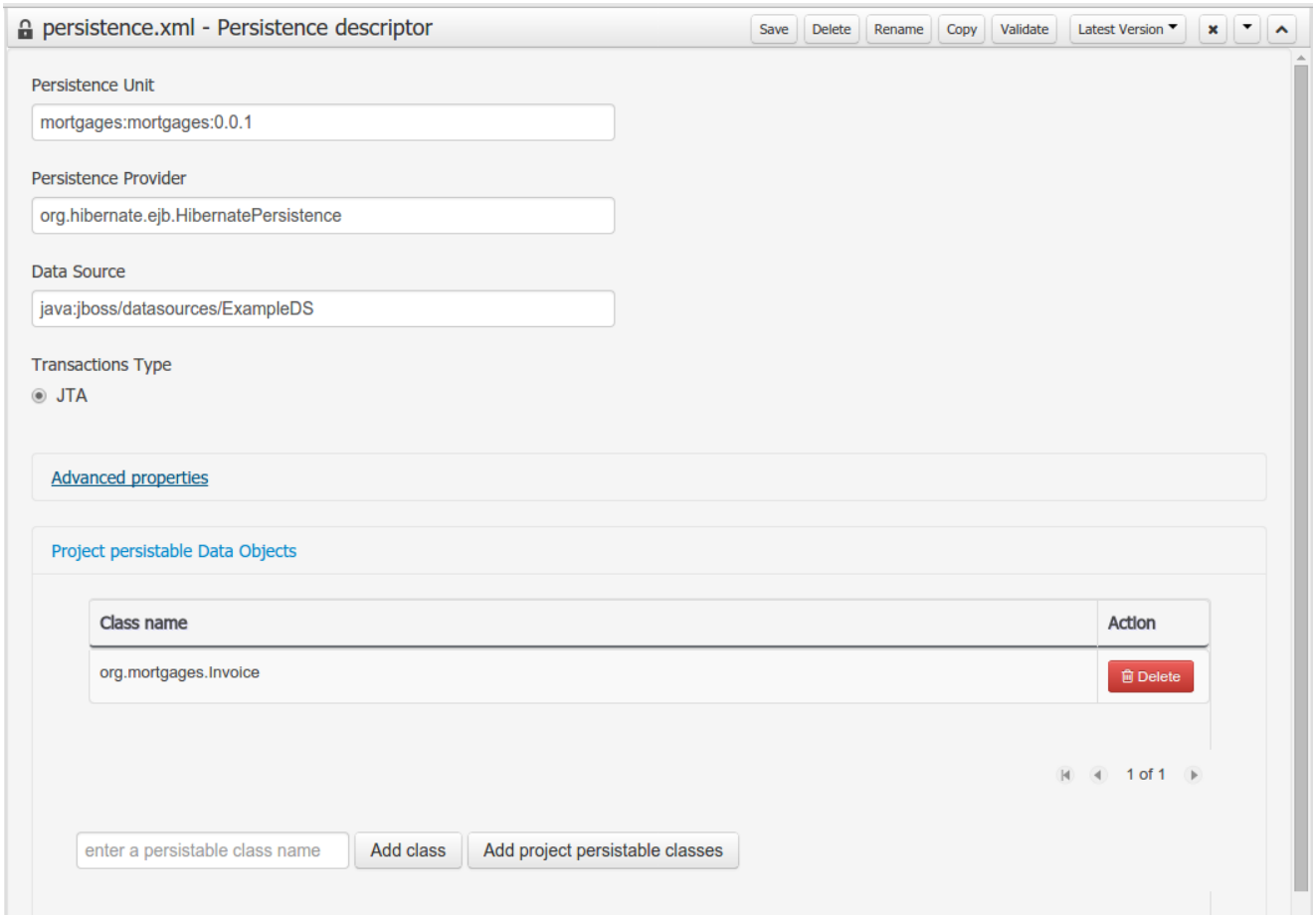


Figure 2.10. Persistence configuration

2.2.4. Data Set Authoring

A new perspective for authoring data set definitions has been added. Data set definitions make it possible to retrieve data from external systems like databases, CSV/Excel files or even use a Java class to generate the data. Once the data is available it can be used, for instance, to create charts and dashboards from the Perspective Editor just feeding the charts from any of the data sets available.

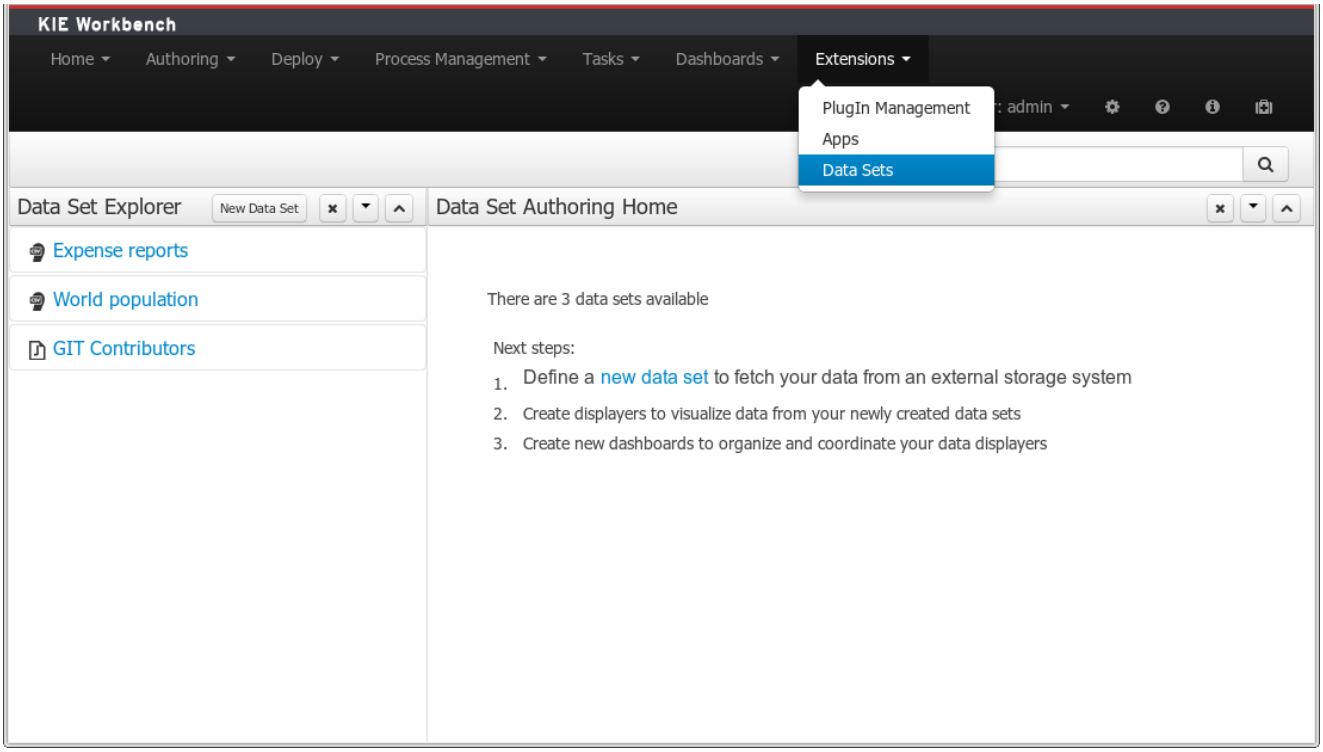


Figure 2.11. Data Sets Authoring Perspective

2.3. What is New and Noteworthy in Drools 6.2.0

2.3.1. Propagation modes

The introduction of PHREAK as default algorithm for the Drools engine made the rules' evaluation lazy. This new Drools lazy behavior allowed a relevant performance boost but, in some very specific cases, breaks the semantic of a few Drools features.

More precisely in some circumstances it is necessary to propagate the insertion of new fact into th session immediately. For instance Drools allows a query to be executed in pull only (or passive) mode by prepending a '?' symbol to its invocation as in the following example:

Example 2.4. A passive query

```
query Q (Integer i)
```

```
    String( this == i.toString() )
end
rule R when
    $i : Integer()
    ?Q( $i; )
then
    System.out.println( $i );
end
```

In this case, since the query is passive, it shouldn't react to the insertion of a String matching the join condition in the query itself. In other words this sequence of commands

```
KieSession ksession = ...
ksession.insert(1);
ksession.insert("1");
ksession.fireAllRules();
```

shouldn't cause the rule R to fire because the String satisfying the query condition has been inserted after the Integer and the passive query shouldn't react to this insertion. Conversely the rule should fire if the insertion sequence is inverted because the insertion of the Integer, when the passive query can be satisfied by the presence of an already existing String, will trigger it.

Unfortunately the lazy nature of PHREAK doesn't allow the engine to make any distinction regarding the insertion sequence of the two facts, so the rule will fire in both cases. In circumstances like this it is necessary to evaluate the rule eagerly as done by the old RETEOO-based engine.

In other cases it is required that the propagation is eager, meaning that it is not immedate, but anyway has to happen before the engine/agenda starts scheduled evaluations. For instance this is necessary when a rule has the no-loop or the lock-on-active attribute and in fact when this happens this propagation mode is automatically enforced by the engine.

To cover these use cases, and in all other situations where an immediate or eager rule evaluation is required, it is possible to declaratively specify so by annotating the rule itself with **@Propagation(Propagation.Type)**, where Propagation.Type is an enumeration with 3 possible values:

- **IMMEDIATE** means that the propagation is performed immediately.
- **EAGER** means that the propagation is performed lazily but eagerly evaluated before scheduled evaluations.
- **LAZY** means that the propagation is totally lazy and this is default PHREAK behaviour

This means that the following drl:

Example 2.5. A data-driven rule using a passive query

```
query Q (Integer i)
```

```
    String( this == i.toString() )
end
rule R @Propagation(IMMEDIATE) when
    $i : Integer()
    ?Q( $i; )
then
    System.out.println( $i );
end
```

will make the rule R to fire if and only if the Integer is inserted after the String, thus behaving in accordance with the semantic of the passive query.

2.4. New and Noteworthy in KIE Workbench 6.2.0

2.4.1. Download Repository or Part of the Repository as a ZIP

This feature makes it possible to download a repository or a folder from the repository as a ZIP file.

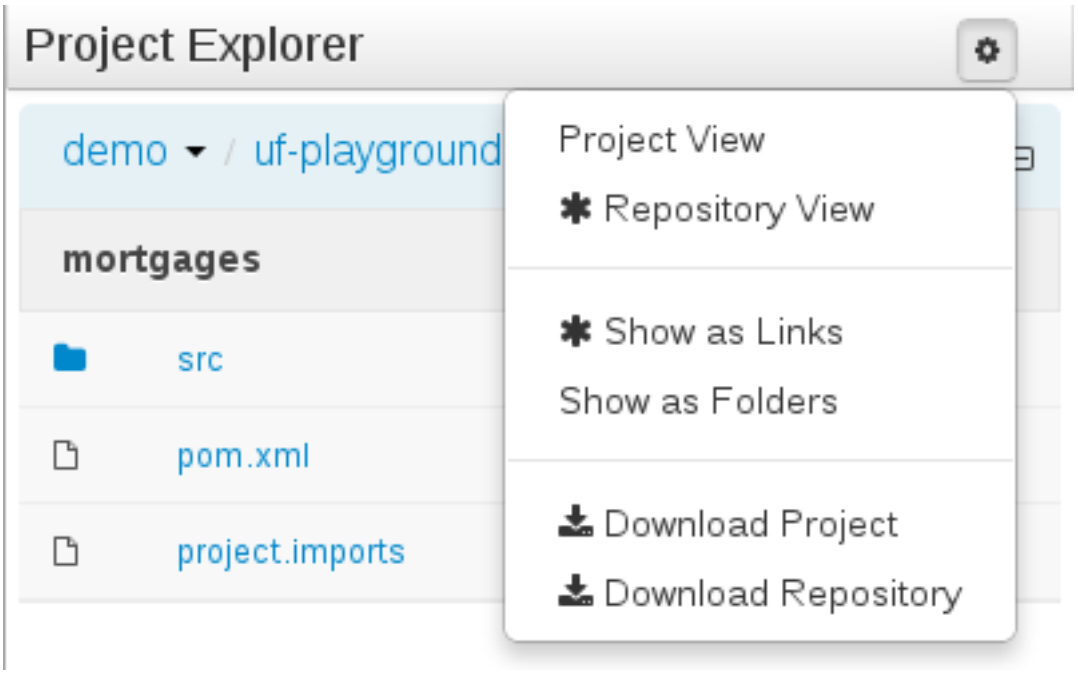


Figure 2.12. Download current repository or project

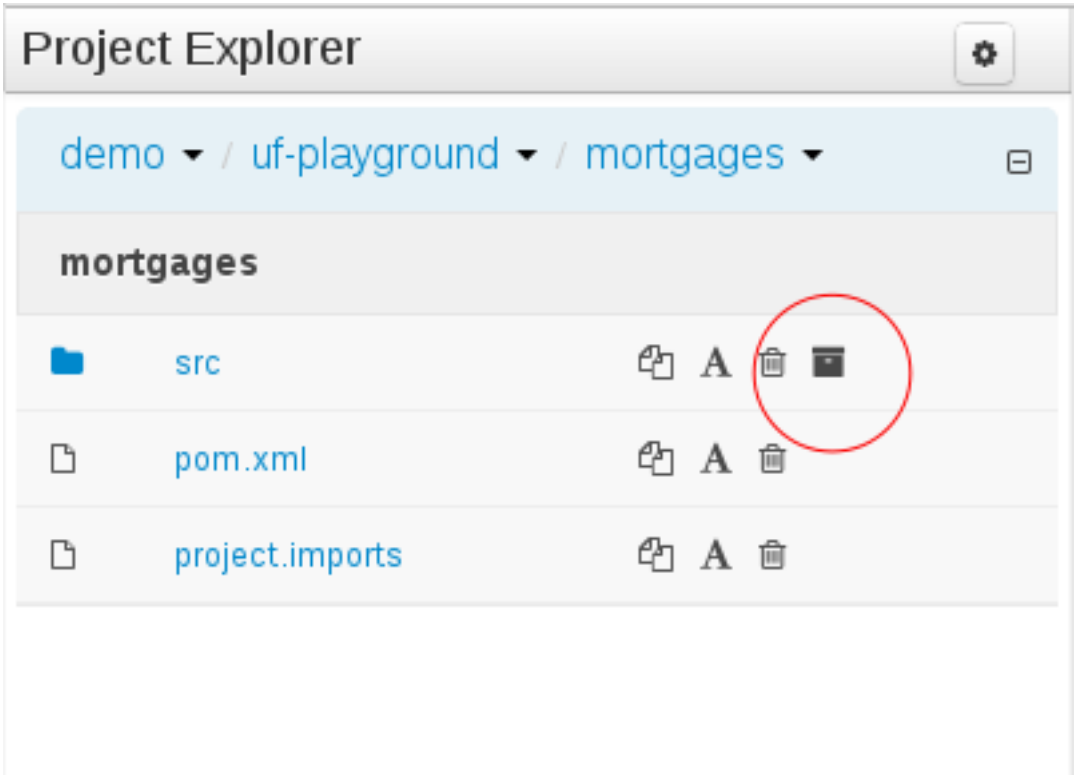


Figure 2.13. Download a folder

2.4.2. Project Editor permissions

The ability to configure role-based permissions for the Project Editor have been added.

Permissions can be configured using the `WEB-INF/classes/workbench-policy.properties` file.

The following permissions are supported:

- Save button

  `feature.wb_project_authoring_save`

- Delete button

  `feature.wb_project_authoring_delete`

- Copy button

  `feature.wb_project_authoring_copy`

- Rename button

  `feature.wb_project_authoring_rename`

- Build & Deploy button

```
feature.wb_project_authoring_buildAndDeploy
```

2.4.3. Unify validation style in Guided Decision Table Wizard.

All of our new screens use GWT-Bootstrap widgets and alert users to input errors in a consistent way.

One of the most noticable differences was the Guided Decision Table Wizard that alerted errors in a way inconsistent with our use of GWT-Bootstrap.

This Wizard has been updated to use the new look and feel.

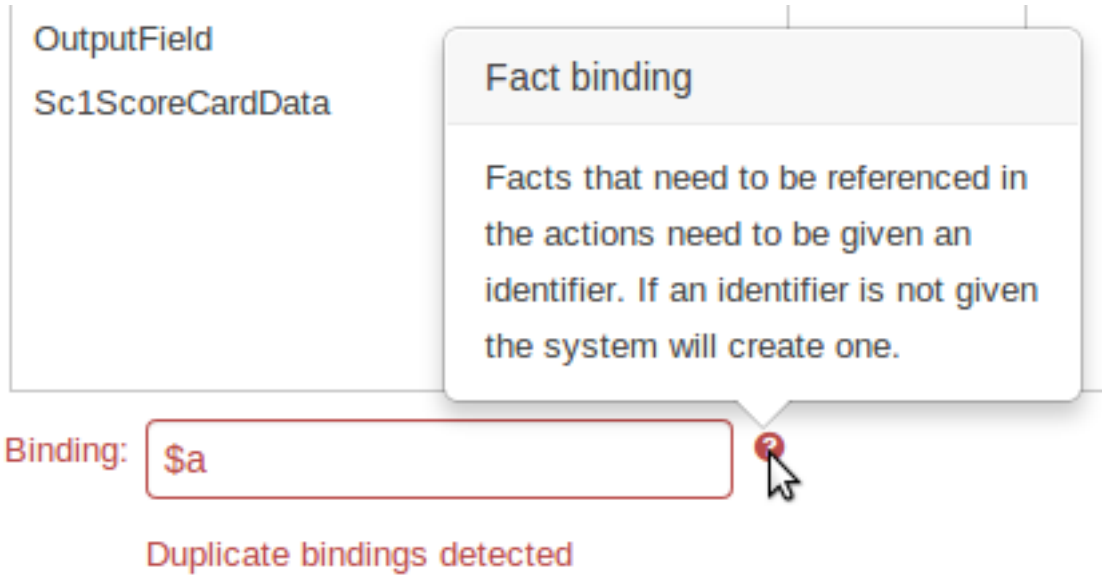


Figure 2.14. New Guided Decision Table Wizard validation

2.4.4. Improved Wizards

During the re-work of the Guided Decision Table's Wizard to make it's validation consistent with other areas of the application we took the opportunity to move the Wizard Framework to GWT-Bootstrap too.

The resulting appearance is much more pleasing. We hope to migrate more legacy editors to GWT-Bootstrap as time and priorities permit.

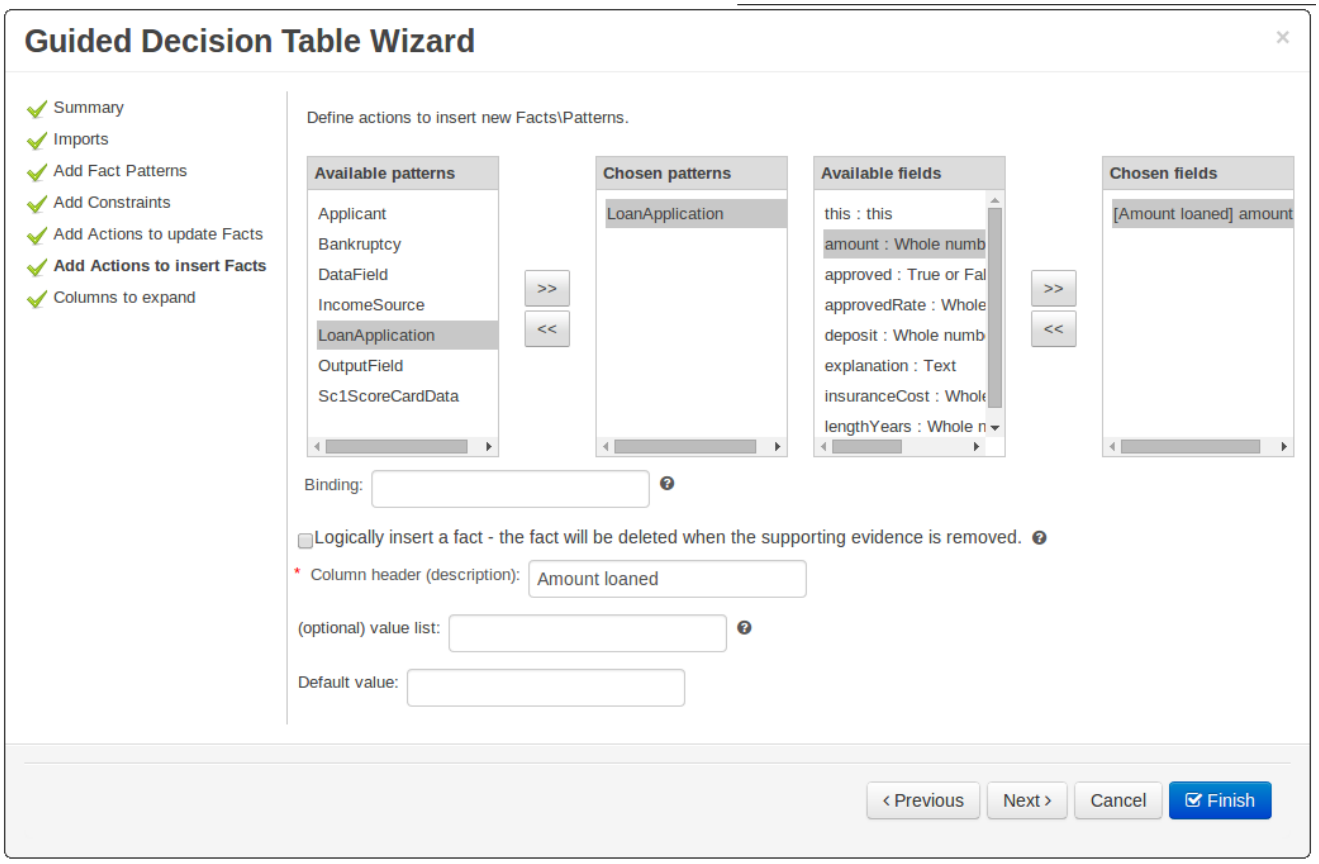


Figure 2.15. New Wizard Framework

2.4.5. Consistent behaviour of XLS, Guided Decision Tables and Guided Templates

Consistency is a good thing for everybody. Users can expect different authoring metaphores to produce the same rule behaviour (and developers know when something is a bug!).

There were a few inconsistencies in the way XLS Decision Tables, Guidied Decision Tables and Guided Rule Templates generated the underlying rules for empty cells. These have been eliminated making their operation consistent.

- If all constraints have null values (empty cells) the Pattern is not created.

  Should you need the Pattern but no constraints; you will need to include the constraint `this != null`.

  This operation is consistent with how XLS and Guided Decision Tables have always worked.

- You can define a constraint on a String field for an empty String or white-space by delimiting it with double-quotation marks. The enclosing quotation-marks are removed from the value when generating the rules.

The use of quotation marks for other String values is not required and they can be omitted. Their use is however essential to differentiate a constraint for an empty String from an empty cell - in which case the constraint is omitted.

2.4.6. Improved Metadata Tab

The Metadata tab provided in previous versions was redesigned to provide a better asset versioning information browsing and recovery. Now every workbench editor will provide an "Overview tab" that will enable the user to manage the following information.

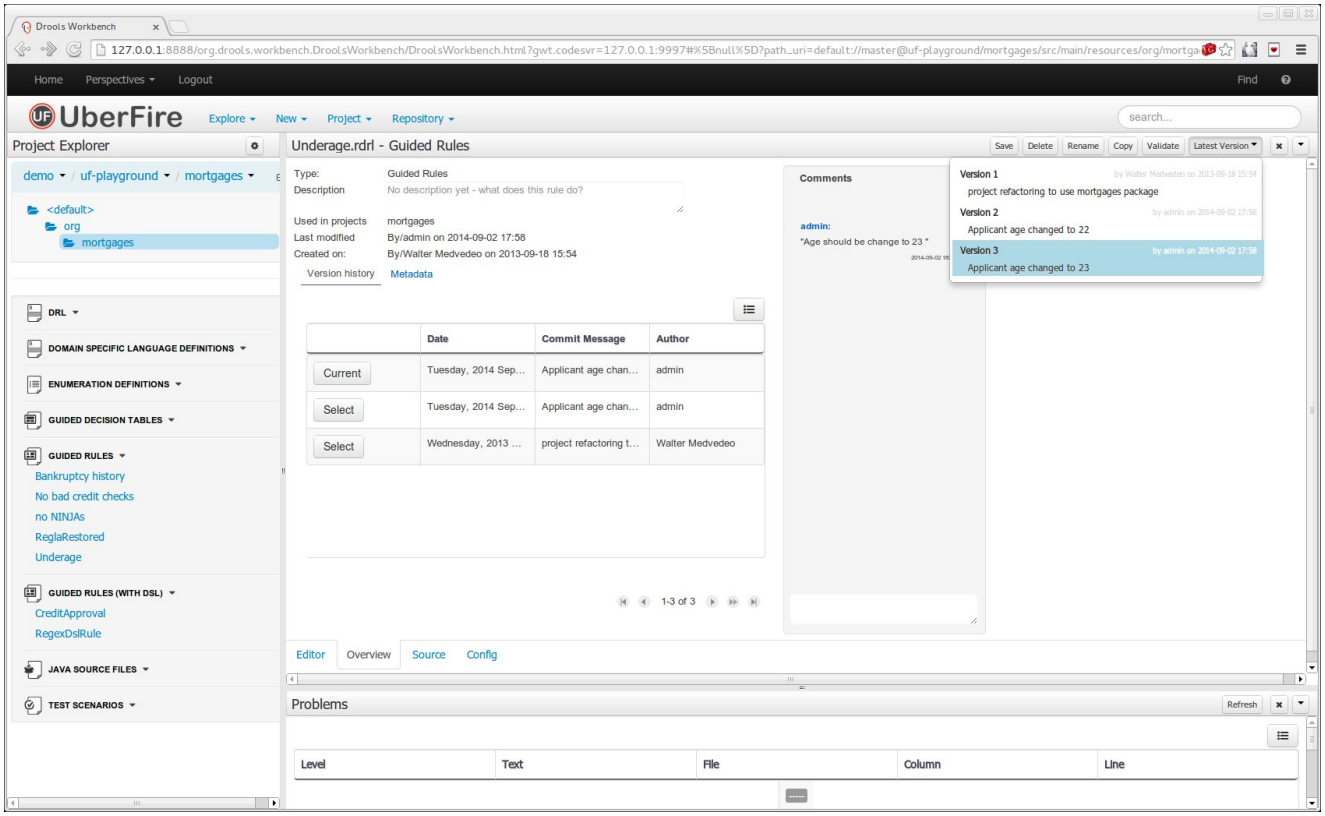


Figure 2.16. Improved Metadata Tab

- Versions history

  The versions history shows a tabular view of the asset versions and provides a "Select" button that will enable the user to load a previously created version.

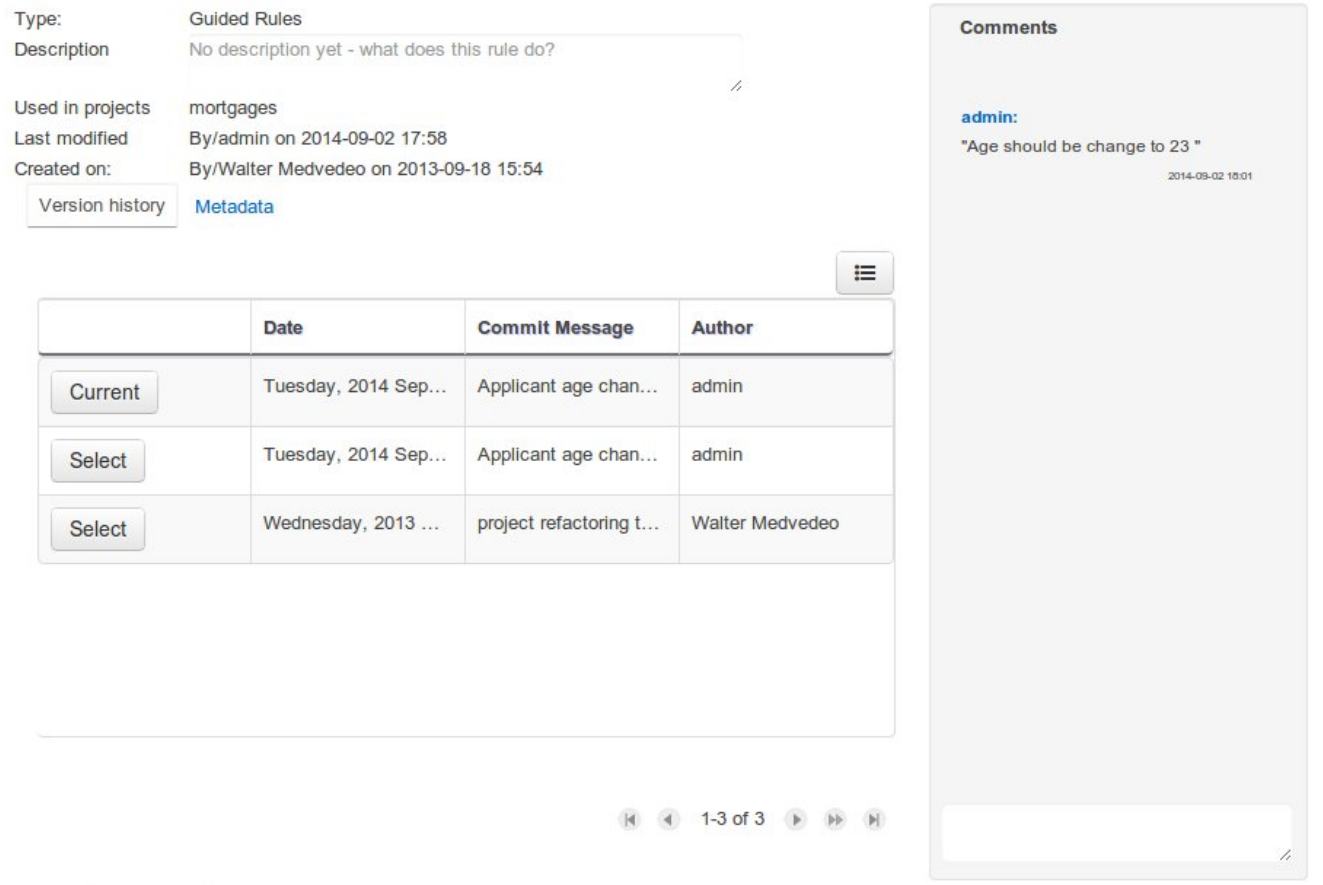


Figure 2.17. Versions history

- Metadata

  The metadata section gets access to additional file attributes.

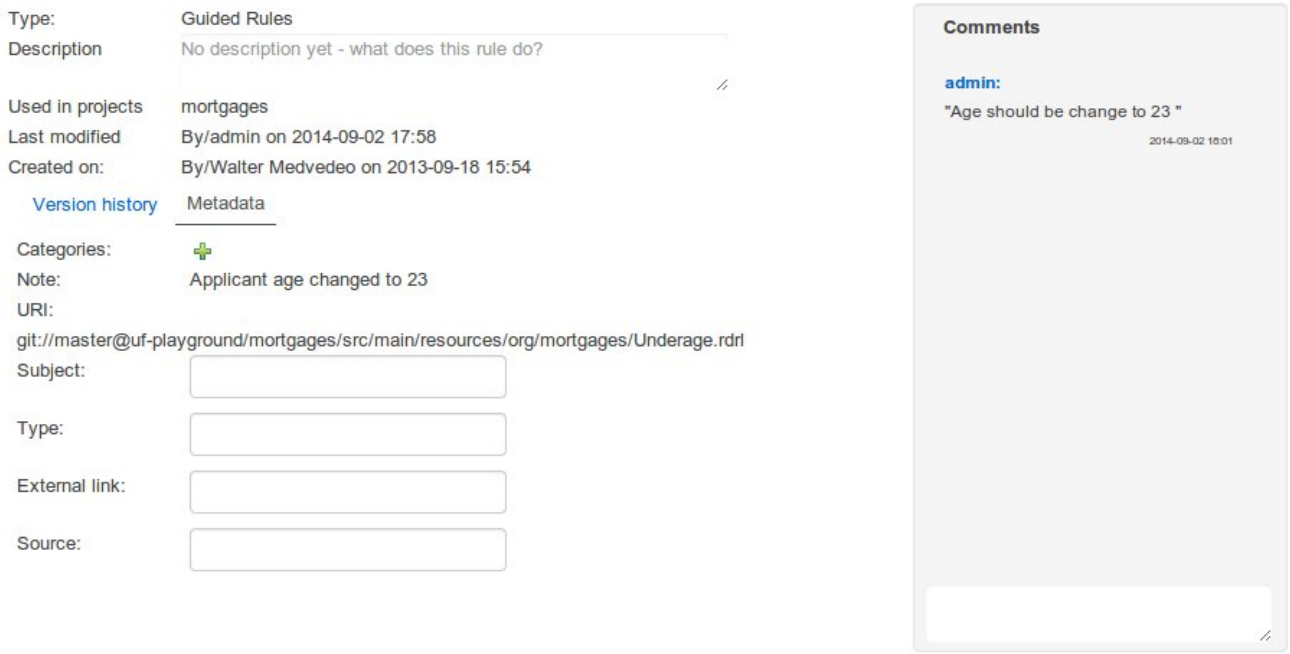


Figure 2.18. Metadata section

- Comments area

  The redesigned comments area enables much clearer discussions on a file.

- Version selection dropdown

  The "Version selector dropdown" located at the menu bar provides the ability to load and restore previous versions from the "Editor tab", without having to open the "Overview tab" to load the "Version history".

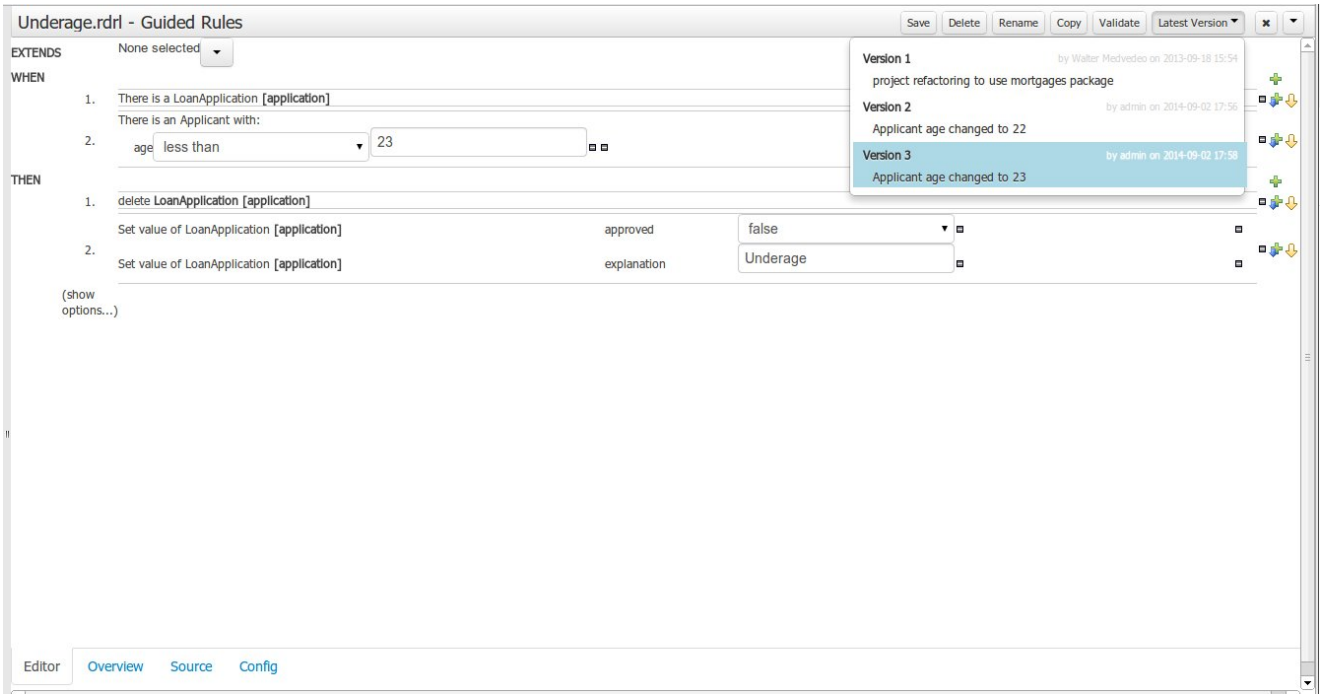


Figure 2.19. Version selection dropdown

2.4.7. Improved Data Objects Editor

The Java editor was unified to the standard workbench editors functioning. It means that and now every data object is edited on his own editor window.

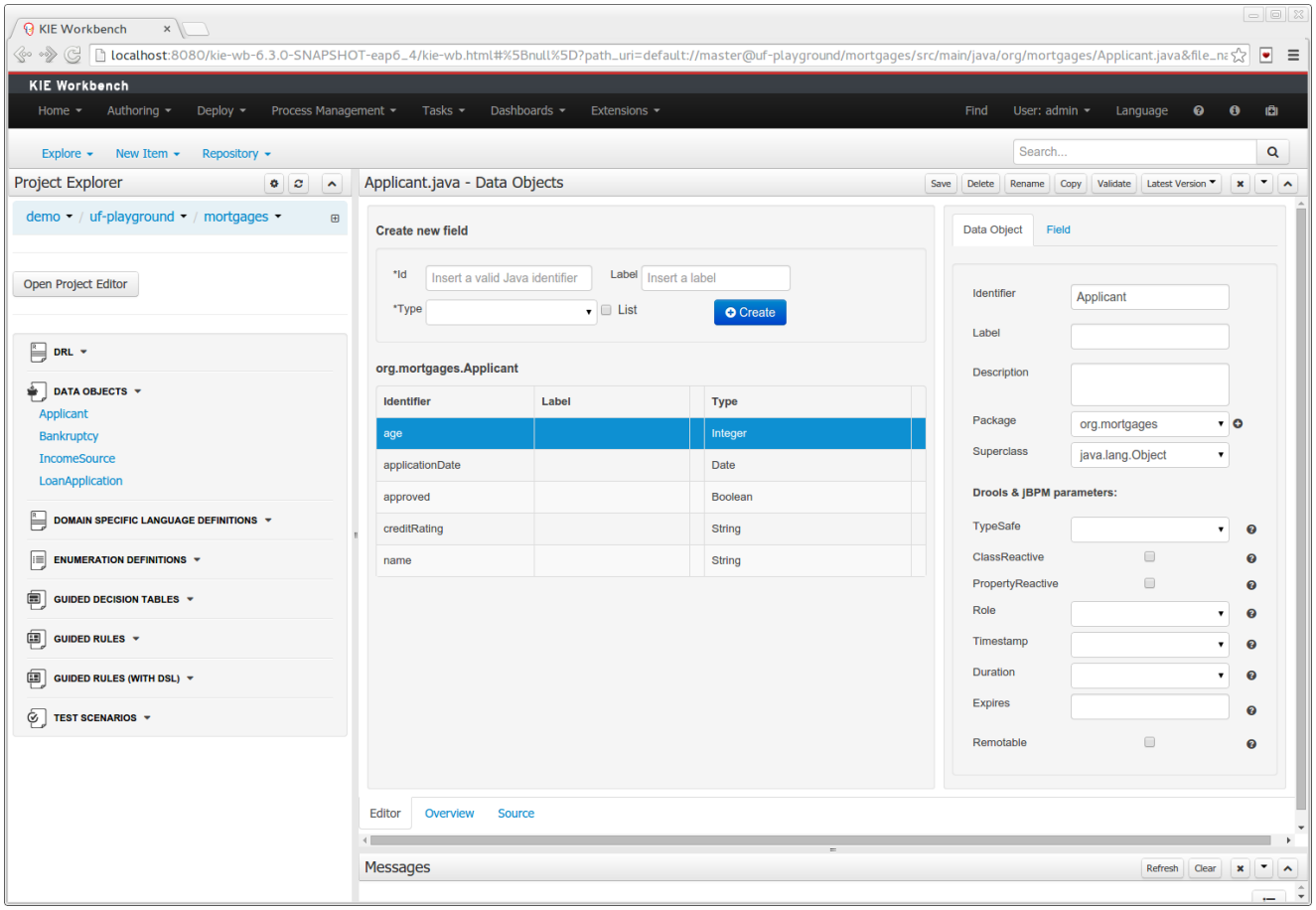


Figure 2.20. Improved Data Object Editor

- "New -> Data Object" option was added to create the data objects.
- Overview tab was added for every file to manage the file metadata and have access to the file versions history.
- Editable "Source Tab" tab was added. Now the Java code can be modified by administrators using the workbench.
- "Editor" - "Source Tab" round trip is provided. This will let administrators to do manual changes on the generated Java code and go back to the editor tab to continue working.
- Class usages detection. Whenever a Data Object is about to be deleted or renamed, the project will be scanned for the class usages. If usages are found (e.g. in drl files, decision tables, etc.) the user will receive an alert. This will prevent the user from breaking the project build.

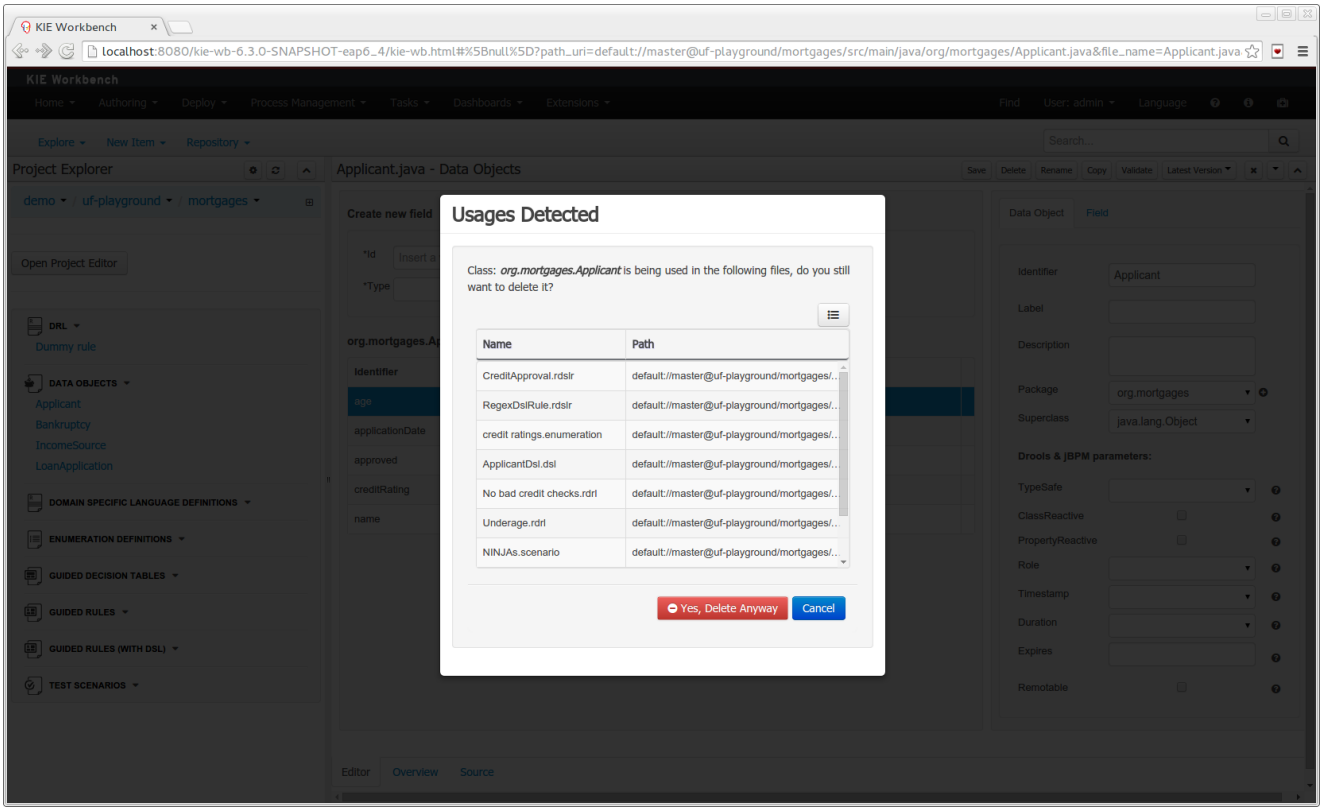


Figure 2.21. Usages detection

2.4.8. Execution Server Management UI

A new perspective called **Management** has been added under **Servers** top level menu. This perspective provides users the ability to manage multiple execution servers with multiple containers. Available features includes connect to already deployed execution servers; create new, start, stop, delete or upgrade containers.

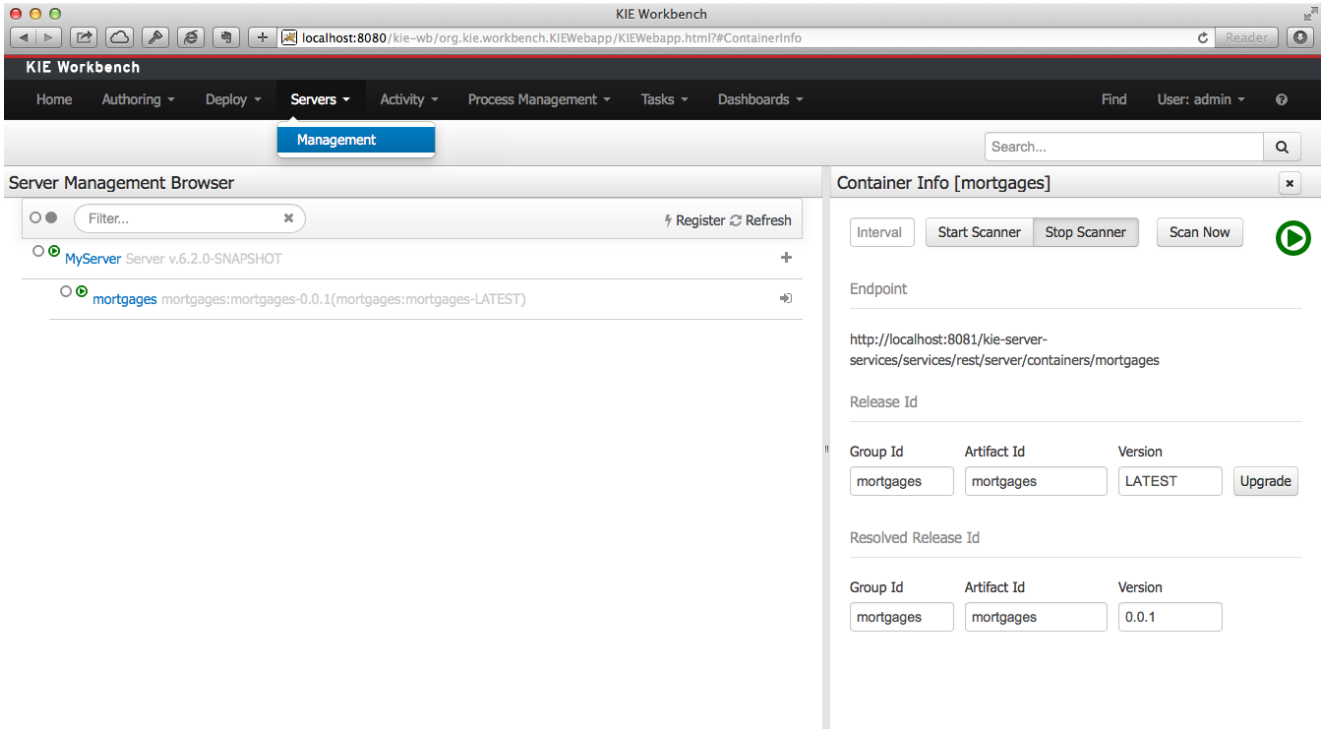


Figure 2.22. Management perspective

Note

Current version of Execution Server just supports rule based execution.

2.4.9. Social Activities

A brand new feature called **Social Activities** has been added under a new top level menu item group called **Activity**.

This new feature is divided in two different perspectives: **Timeline Perspective** and **People Perspective**.

The **Timeline Perspective** shows on left side the recent assets created or edited by the logged user. In the main window there is the "Latest Changes" screen, showing all the recent updated assets and an option to filter the recent updates by repository.

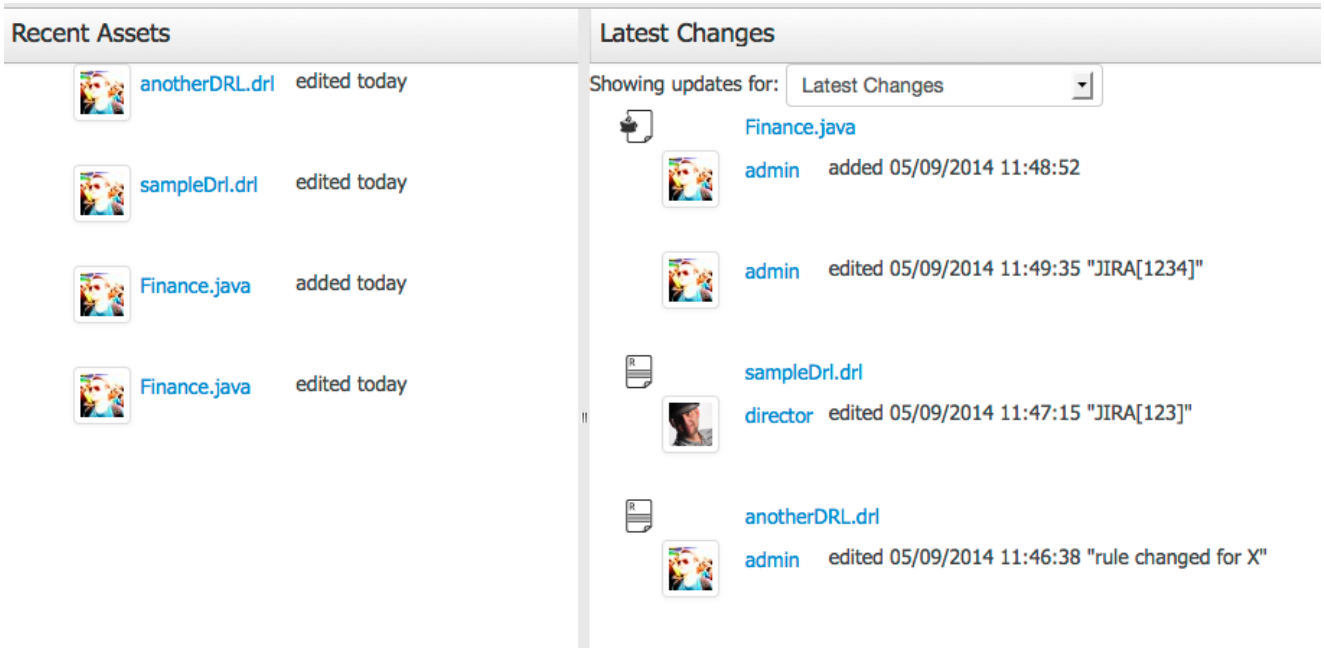


Figure 2.23. Timeline Perspective

The **People Perspective** is the home page of an user. Showing his infos (including a gravatar picture from user e-mail), user connections (people that user follow) and user recent activities. There is also a way to edit an user info. The search suggestion can be used to navigate to a user profile, follow him and see his updates on your timeline.

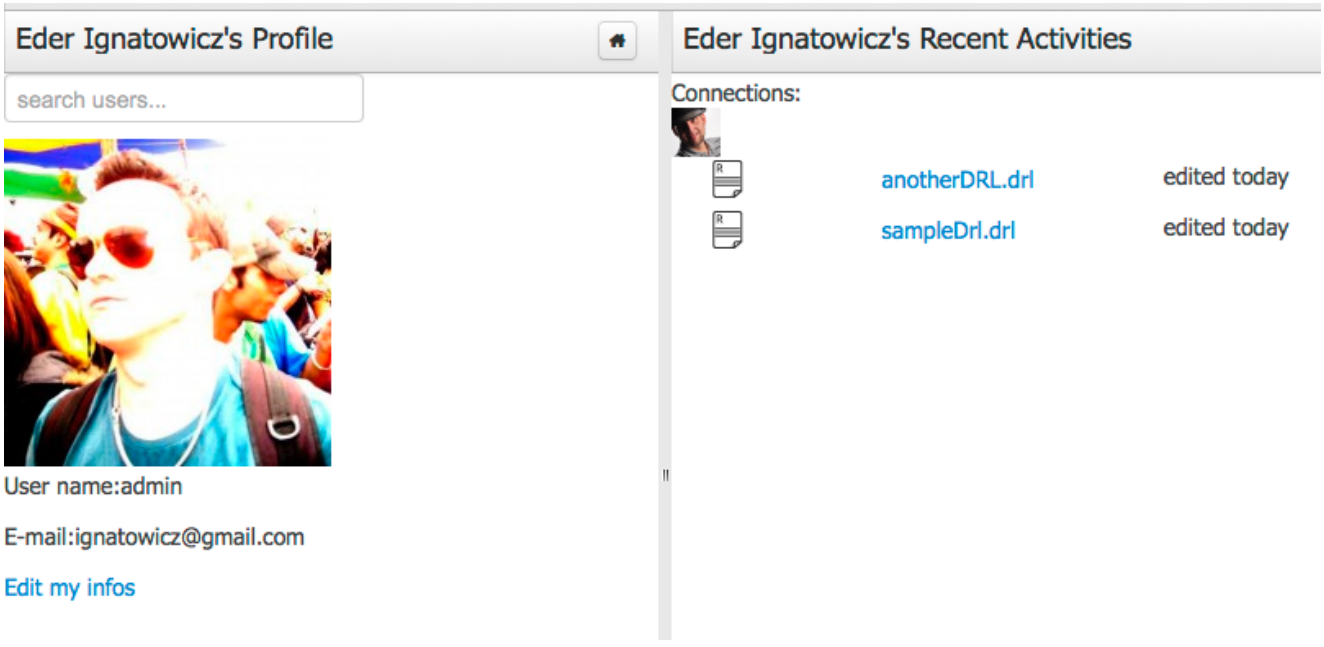


Figure 2.24. People Perspective

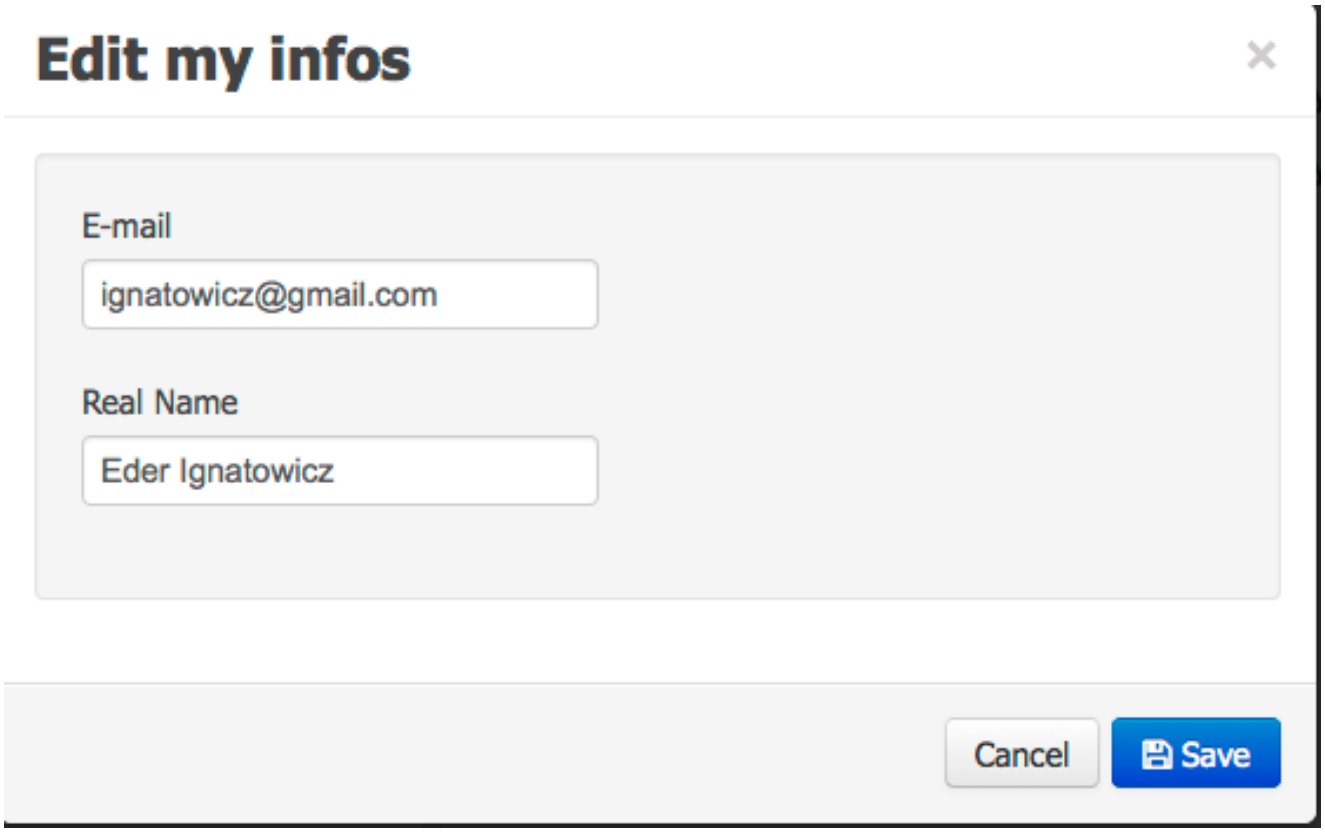


Figure 2.25. Edit User Info

2.4.10. Contributors Dashboard

A brand new perspective called **Contributors** has been added under a new top level menu item group called **Activity**. The perspective itself is a dashboard which shows several indicators about the contributions made to the managed organizations / repositories within the workbench. Every time a organization/repository is added/removed from the workbench the dashboard itself is updated accordingly.

This new perspective allows for the monitoring of the underlying activity on the managed repositories.

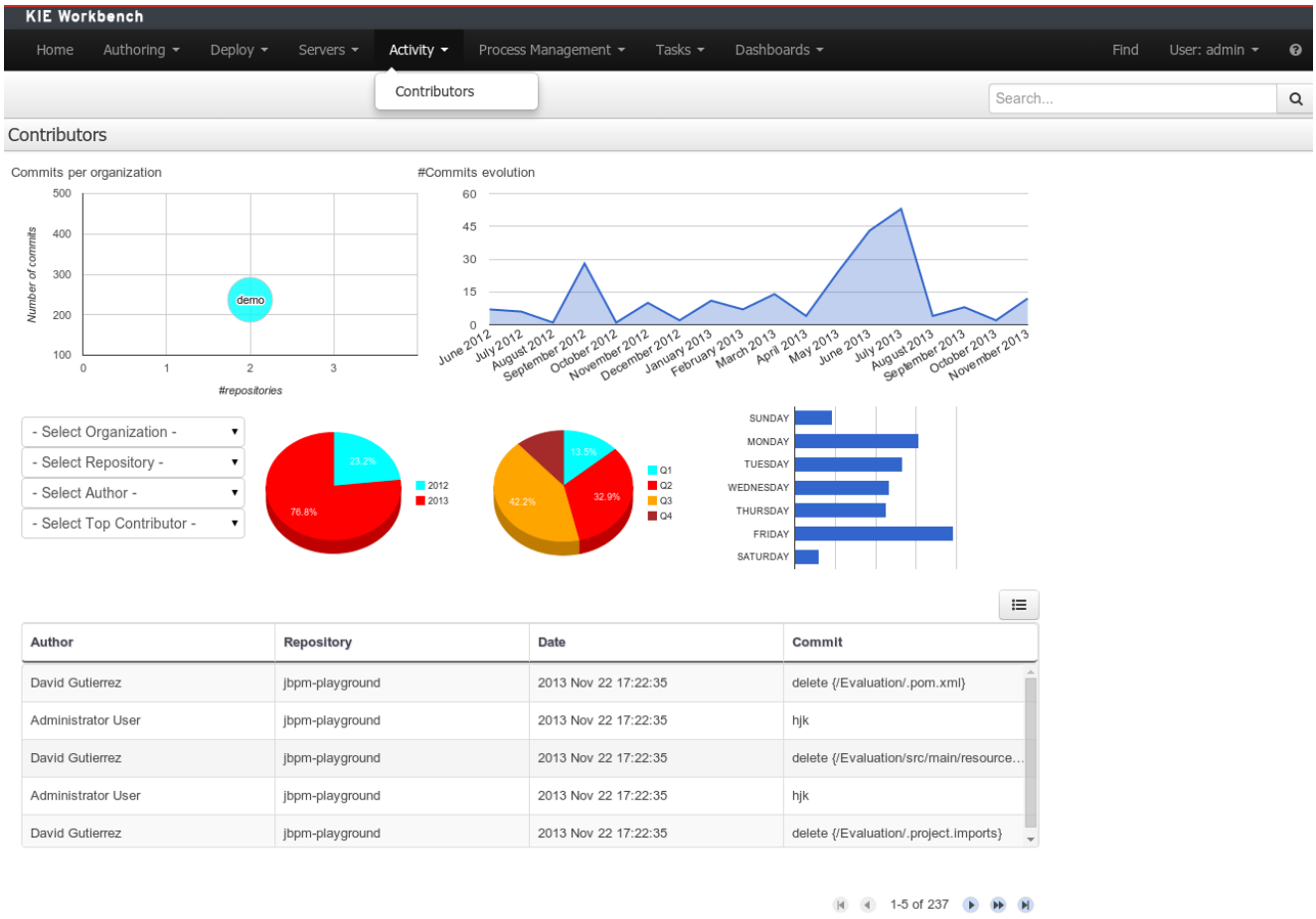


Figure 2.26. Contributors perspective

2.4.11. Package selector

The location of new assets whilst authoring was driven by the context of the Project Explorer.

This has been replaced with a Package Selector in the New Resource Popup.

The location defaults to the Project Explorer context but different packages can now be more easily chosen.

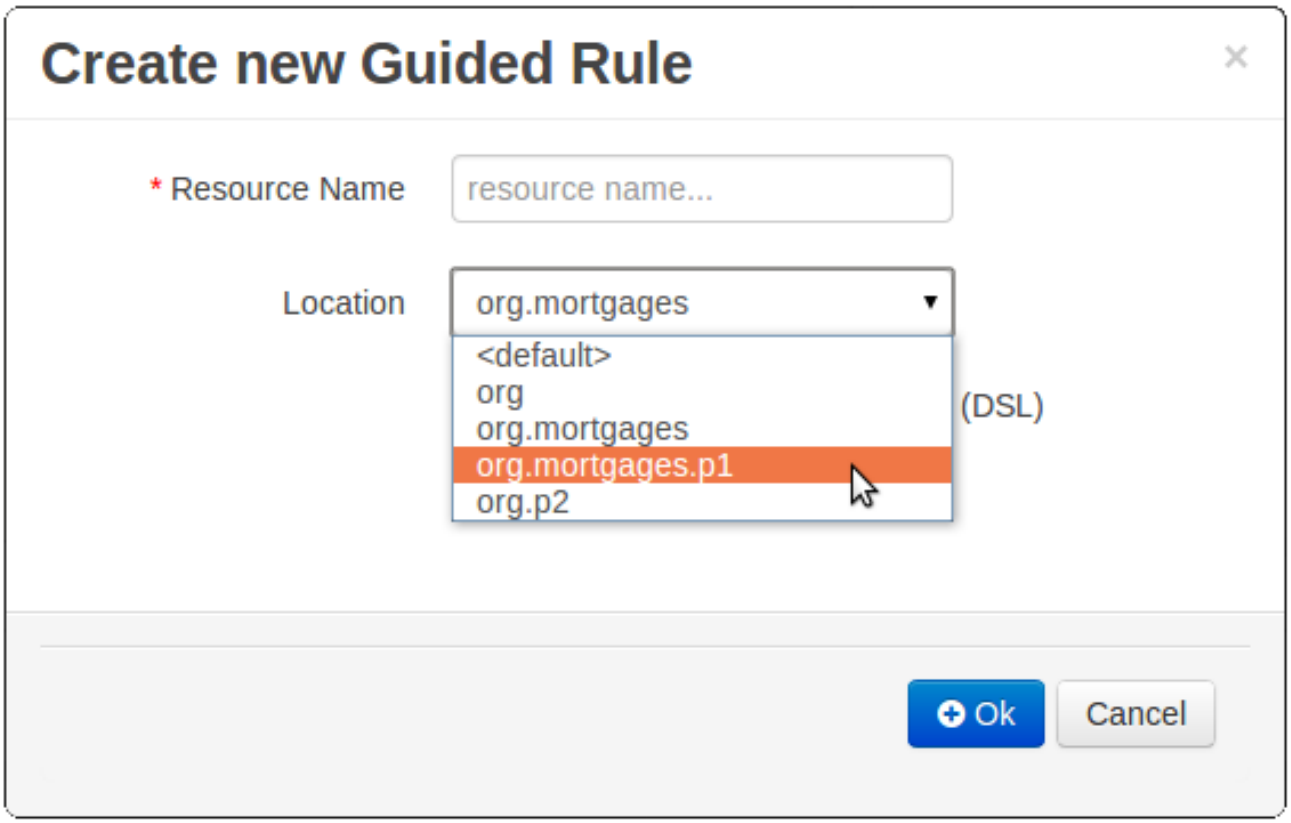


Figure 2.27. Package selector

2.4.12. Improved visual consistency

All Popups have been refactored to use GWT-Bootstrap widgets.

Whilst a simple change it brings greater visual consistency to the application as a whole.

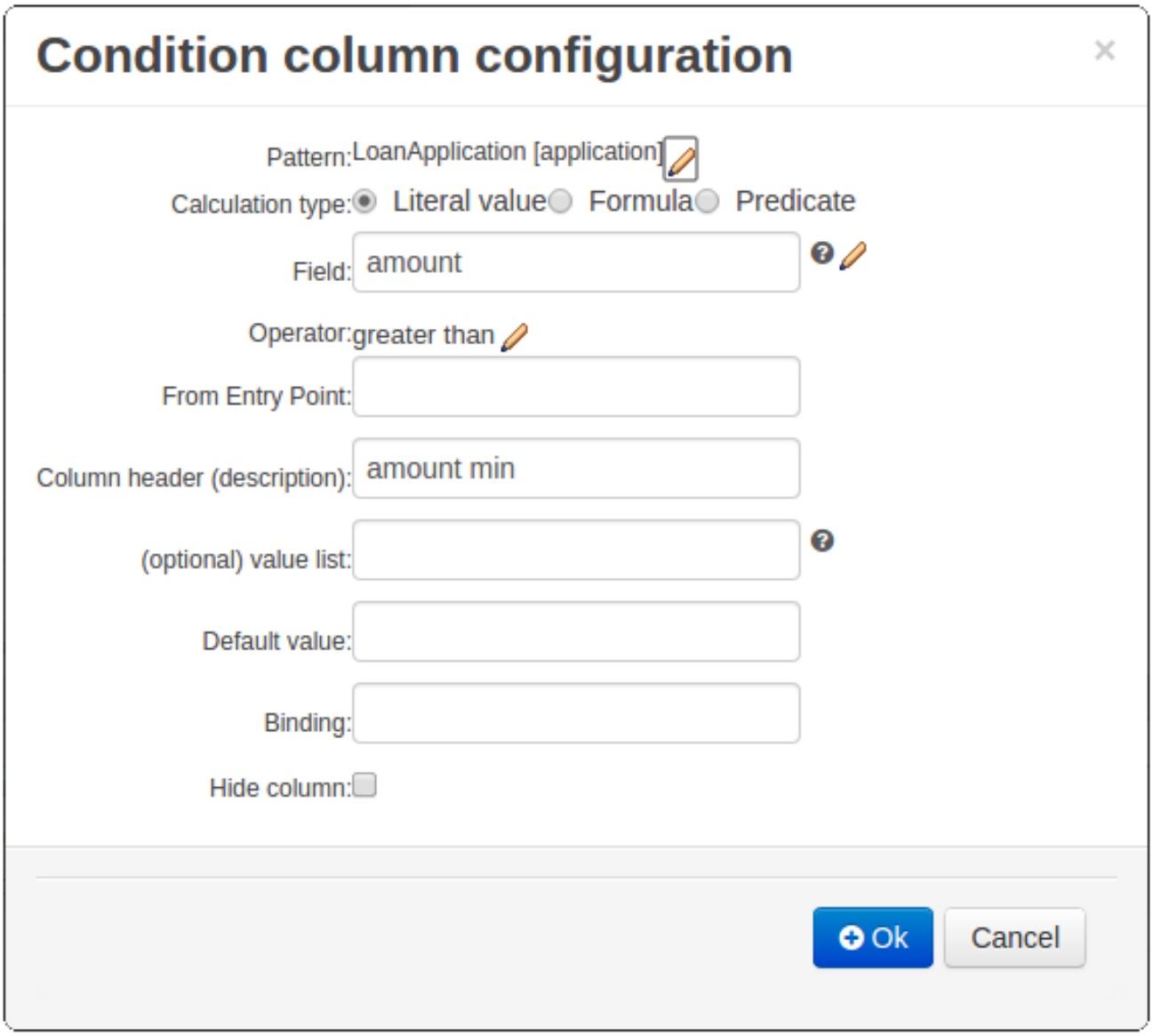


Figure 2.28. Example Guided Decision Table Editor popup

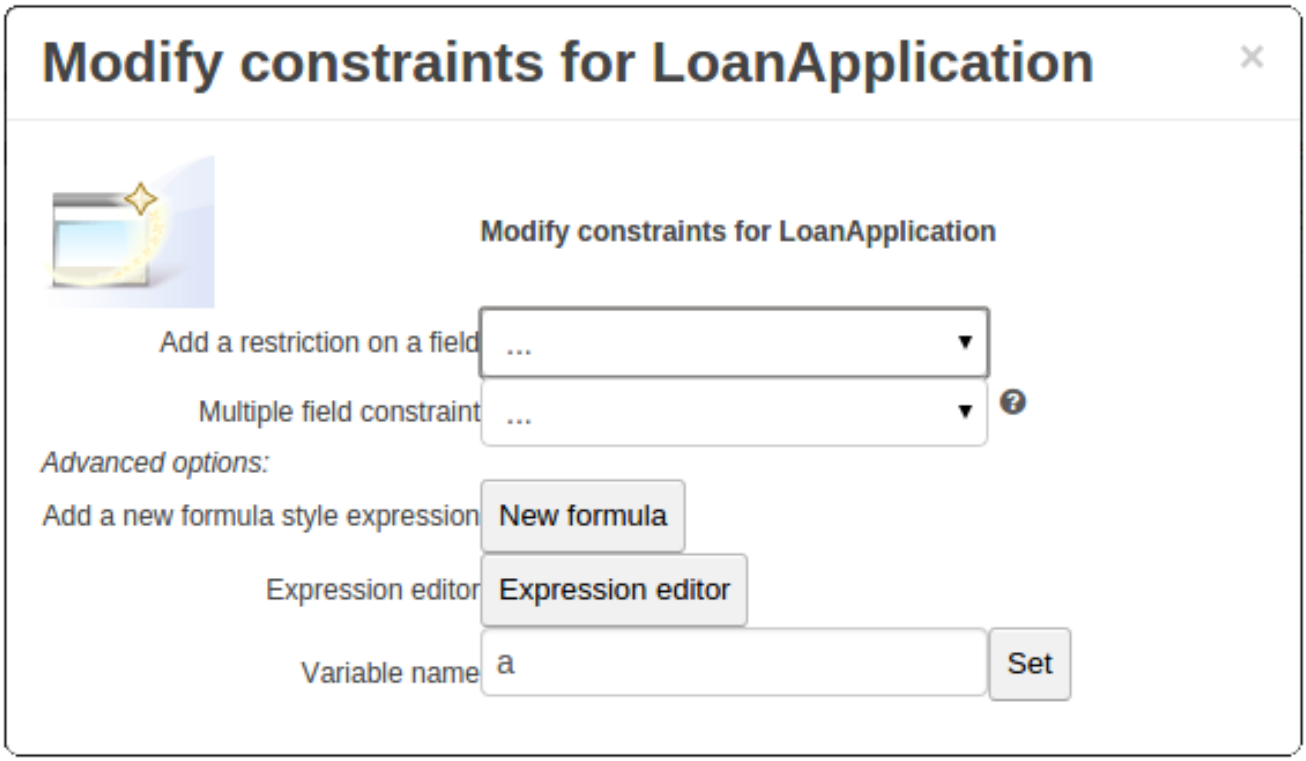


Figure 2.29. Example Guided Rule Editor popup

2.4.13. Guided Decision Tree Editor

A new editor has been added to support modelling of simple decision trees.

See the applicable section within the User Guide for more information about usage.

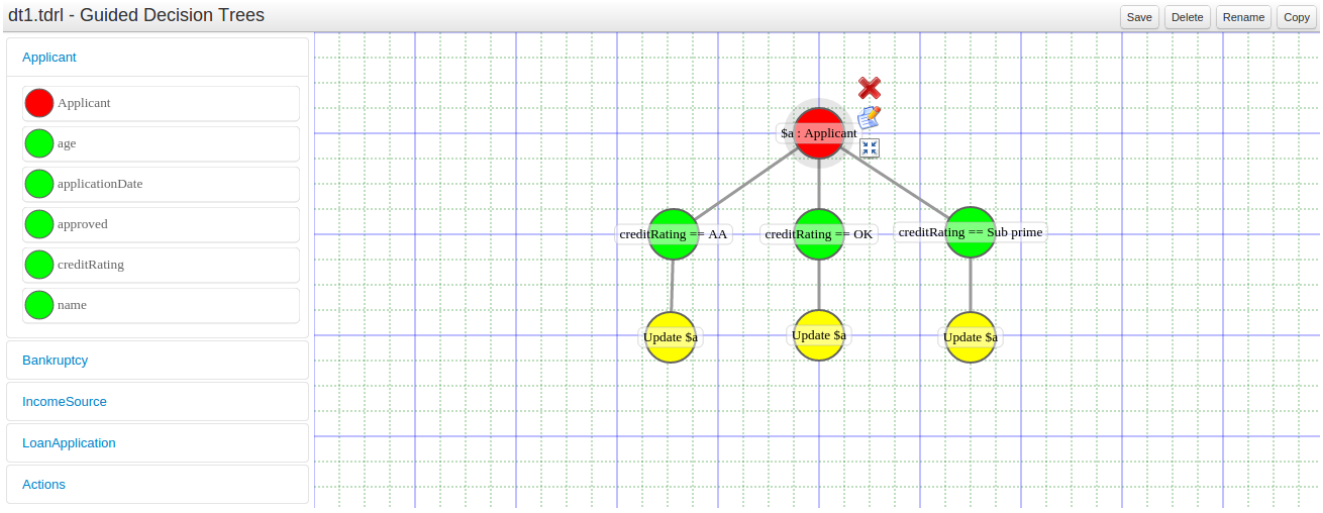


Figure 2.30. Example Guided Decision Tree

2.4.14. Create Repository Wizard

A wizard has been created to guide the repository creation process. Now the user can decide at repository creation time if it should be a managed or unmanaged repository and configure all related parameters.

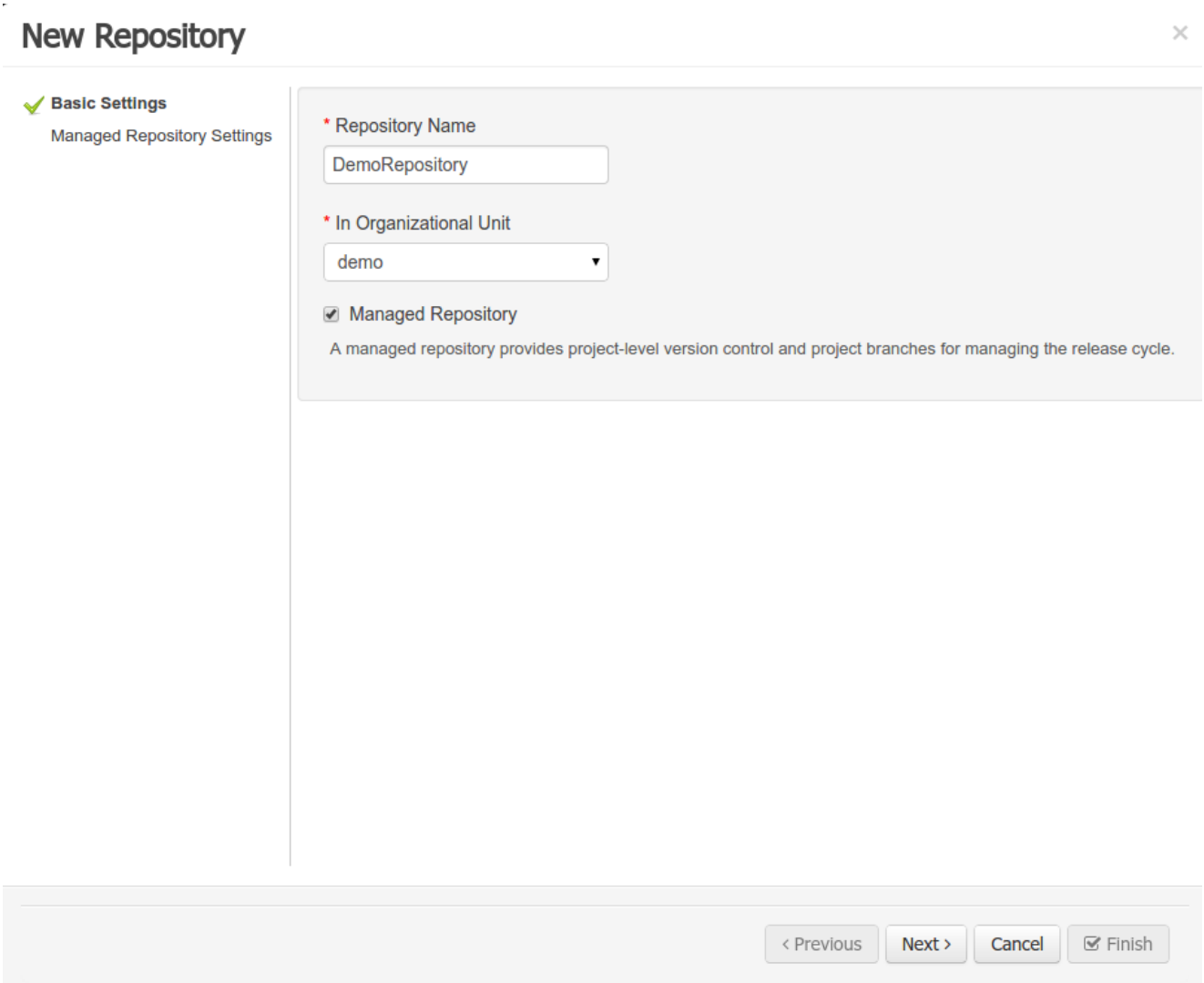


Figure 2.31. Create Repository Wizard 1/2

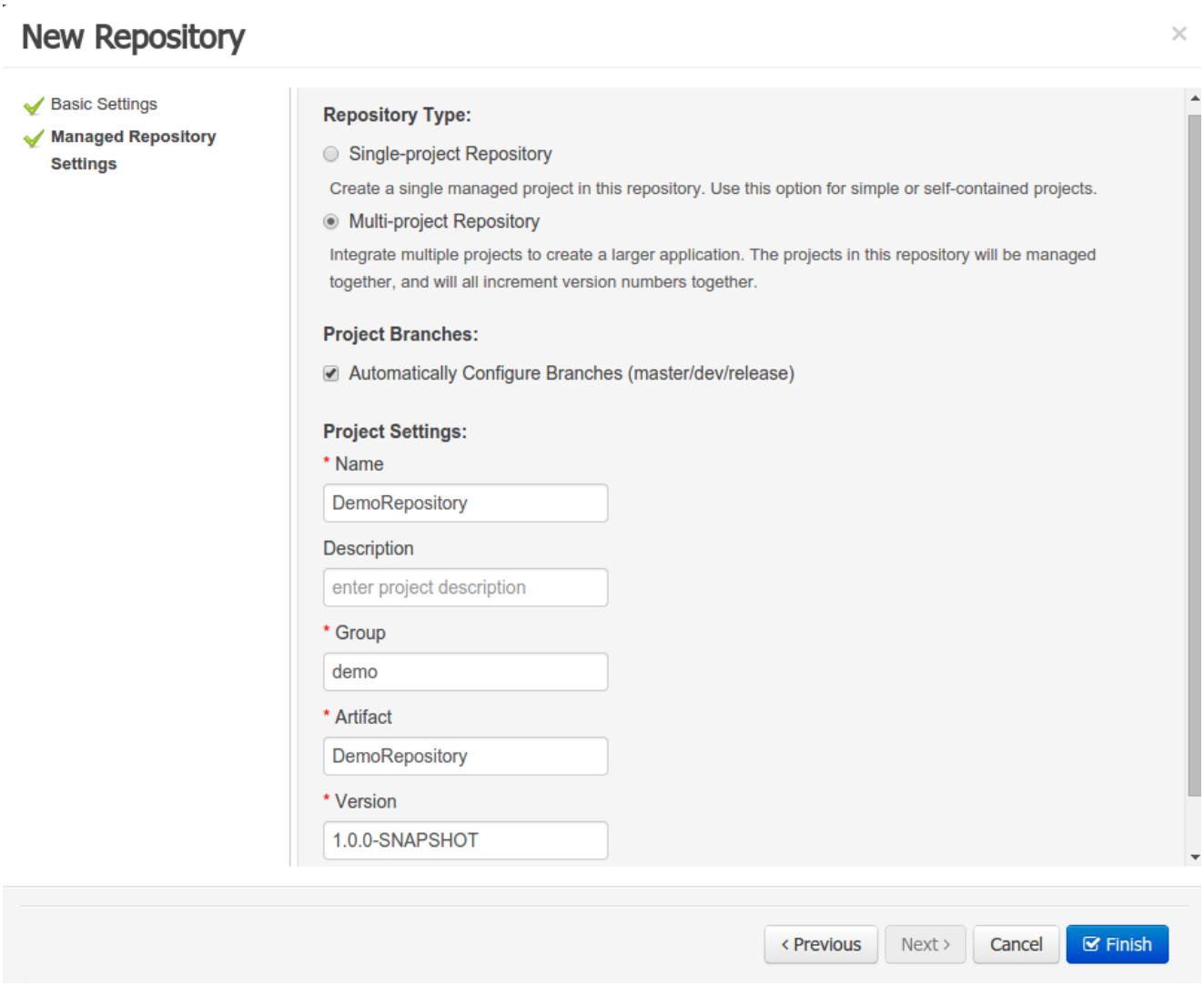


Figure 2.32. Create Repository Wizard 2/2

2.4.15. Repository Structure Screen

The new Repository Structure Screen will let users to manage the projects for a given repository, as well as other operations related to managed repositories like: branch creation, assets promotion and project release.

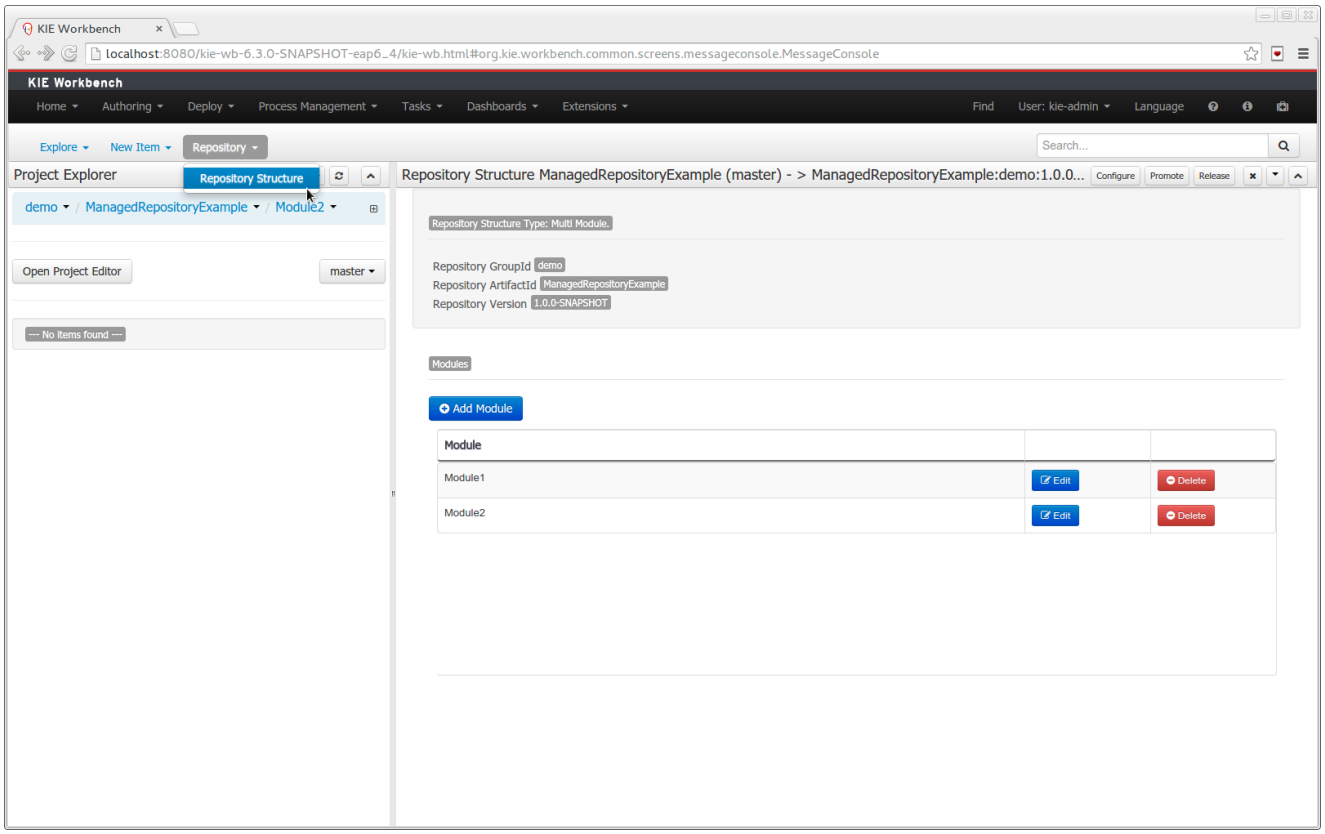


Figure 2.33. Repository Structure Screen for a Managed Repository

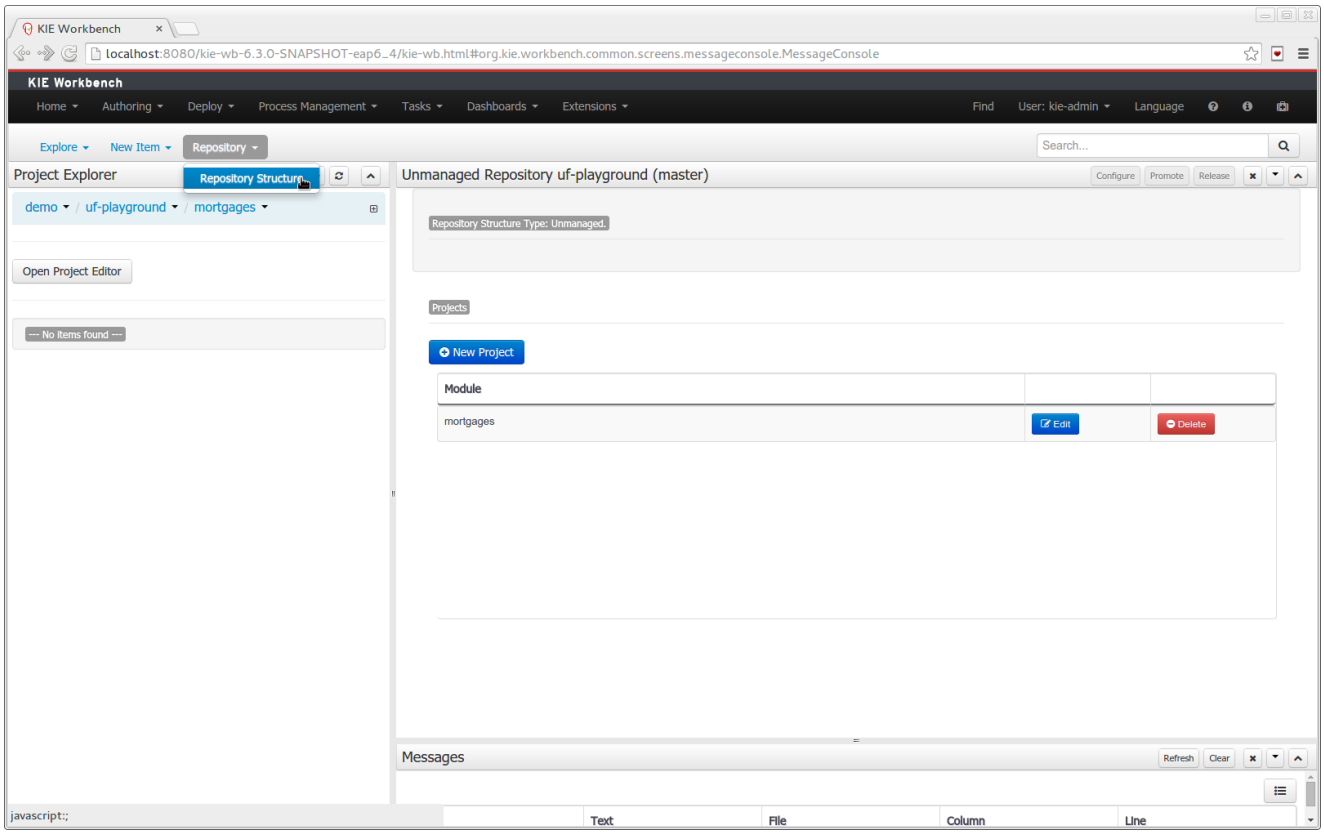


Figure 2.34. Repository Structure Screen for an Unmanaged Repository

2.5. New and Noteworthy in Integration 6.2.0

2.5.1. KIE Execution Server

A new KIE Execution Server was created with the goal of supporting the deployment of kjars and the automatic creation of REST endpoints for remote rules execution. This initial implementation supports provisioning and execution of kjars via REST without any glue code.

A user interface was also integrated into the workbench for remote provisioning. See the workbench's New&Noteworthy for details.

```
@Path("/server")
public interface KieServer {

    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response getInfo();

    @POST
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response execute( CommandScript command );

    @GET
```

```
    @Path("containers")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response listContainers();

    @GET
    @Path("containers/{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response getContainerInfo( @PathParam("id") String id );

    @PUT
    @Path("containers/{id}")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
   public Response createContainer( @PathParam("id") String id, KieContainerResource container );

    @DELETE
    @Path("containers/{id}")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response disposeContainer( @PathParam("id") String id );

    @POST
    @Path("containers/{id}")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response execute( @PathParam("id") String id, String cmdPayload );

    @GET
    @Path("containers/{id}/release-id")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response getReleaseId( @PathParam("id") String id);

    @POST
    @Path("containers/{id}/release-id")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response updateReleaseId( @PathParam("id") String id, ReleaseId releaseId );

    @GET
    @Path("containers/{id}/scanner")
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response getScannerInfo( @PathParam("id") String id );

    @POST
    @Path("containers/{id}/scanner")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response updateScanner( @PathParam("id") String id, KieScannerResource resource );

}
```

Figure 2.35. Kie Server interface

2.6. What is New and Noteworthy in Drools 6.1.0

2.6.1. JMX support for KieScanner

Added support for JMX monitoring and management on KieScanner and KieContainer. To enable, set the property `kie.scanner.mbeans` to `enabled`, for example via Java command line: `-Dkie.scanner.mbeans=enabled .`

KieScannerMBean will register under the name:

It exposes the following properties:

- **Scanner Release Id:** the release ID the scanner was configured with. May include maven range versions and special keywords like LATEST, SNAPSHOT, etc.
- **Current Release Id:** the actual release ID the artifact resolved to.
- **Status:** STARTING, SCANNING, UPDATING, RUNNING, STOPPED, SHUTDOWN

It also exposes the following operations:

- **scanNow():** forces an immediate scan of the maven repository looking for artifact updates
- **start():** starts polling the maven repository for artifact updates based on the polling interval parameter
- **stop():** stops automatically polling the maven repository

2.7. New and Noteworthy in KIE Workbench 6.1.0

2.7.1. Data Modeler - round trip and source code preservation

Full round trip between Data modeler and Java source code is now supported. No matter where the Java code was generated (e.g. Eclipse, Data modeller), data modeler will only update the necessary code blocks to maintain the model updated.

2.7.2. Data Modeler - improved annotations

New annotations @TypeSafe, @ClassReactive, @PropertyReactive, @Timestamp, @Duration and @Expires were added in order enrich current Drools annotations manged by the data modeler.

2.7.3. Standardization of the display of tabular data

We have standardized the display of tabular data with a new table widget.

The new table supports the following features:

- Selection of visible columns
- Resizable columns

- Moveable columns

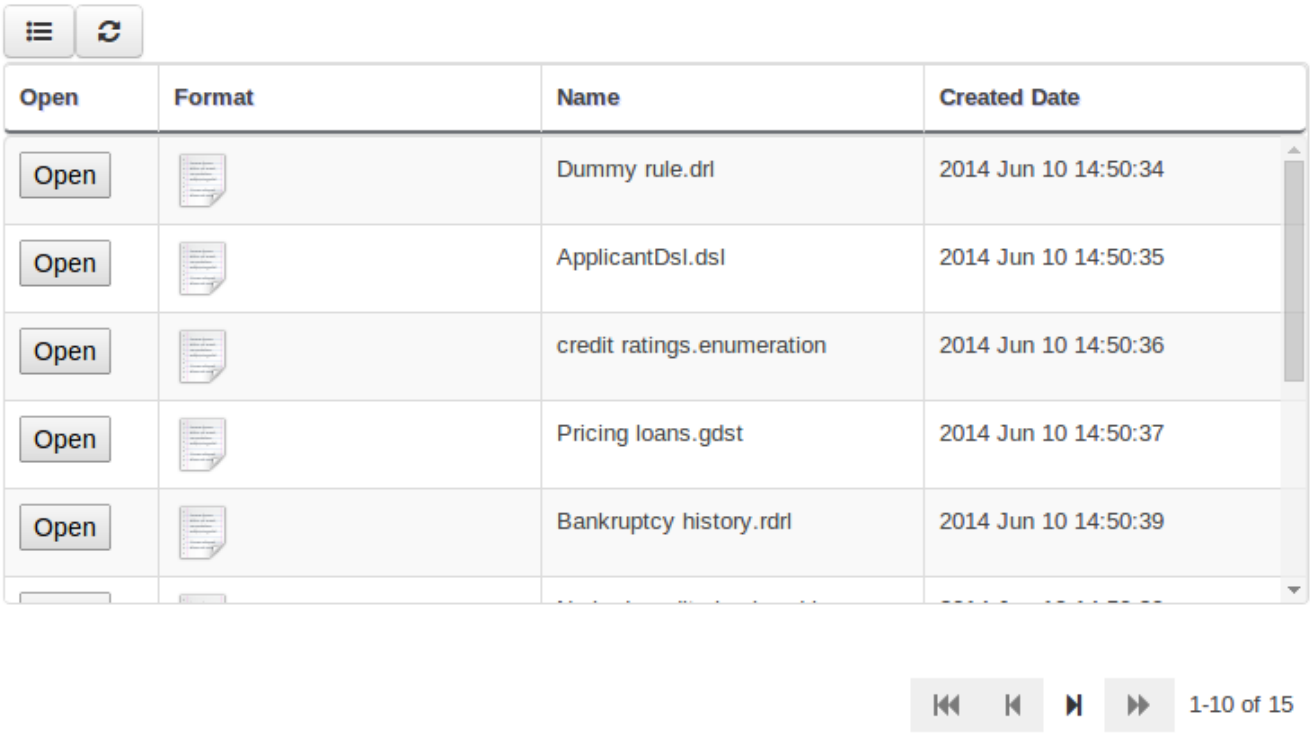


Figure 2.36. New table

The table is used in the following scenarios:

- Inbox (Incoming changes)
- Inbox (Recently edited)
- Inbox (Recently opened)
- Project Problems summary
- Artifact Repository browser
- Project Editor Dependency grid
- Project Editor KSession grid
- Project Editor Work Item Handlers Configuration grid
- Project Editor Listeners Configuration grid
- Search Results grid

2.7.4. Generation of `modify(x) {...}` blocks

The Guided Rule Editor, Guided Template Editor and Guided Decision Table Editor have been changed to generate `modify(x){...}`

Historically these editors supported the older `update(x)` syntax and hence rules created within the Workbench would not respond correctly to `@PropertyReactive` and associated annotations within a model. This has now been rectified with the use of `modify(x){...}` blocks.

2.8. New and Noteworthy in KIE API 6.0.0

2.8.1. New KIE name

KIE is the new umbrella name used to group together our related projects; as the family continues to grow. KIE is also used for the generic parts of unified API; such as building, deploying and loading. This replaces the droolsjbpm and knowledge keywords that would have been used before.

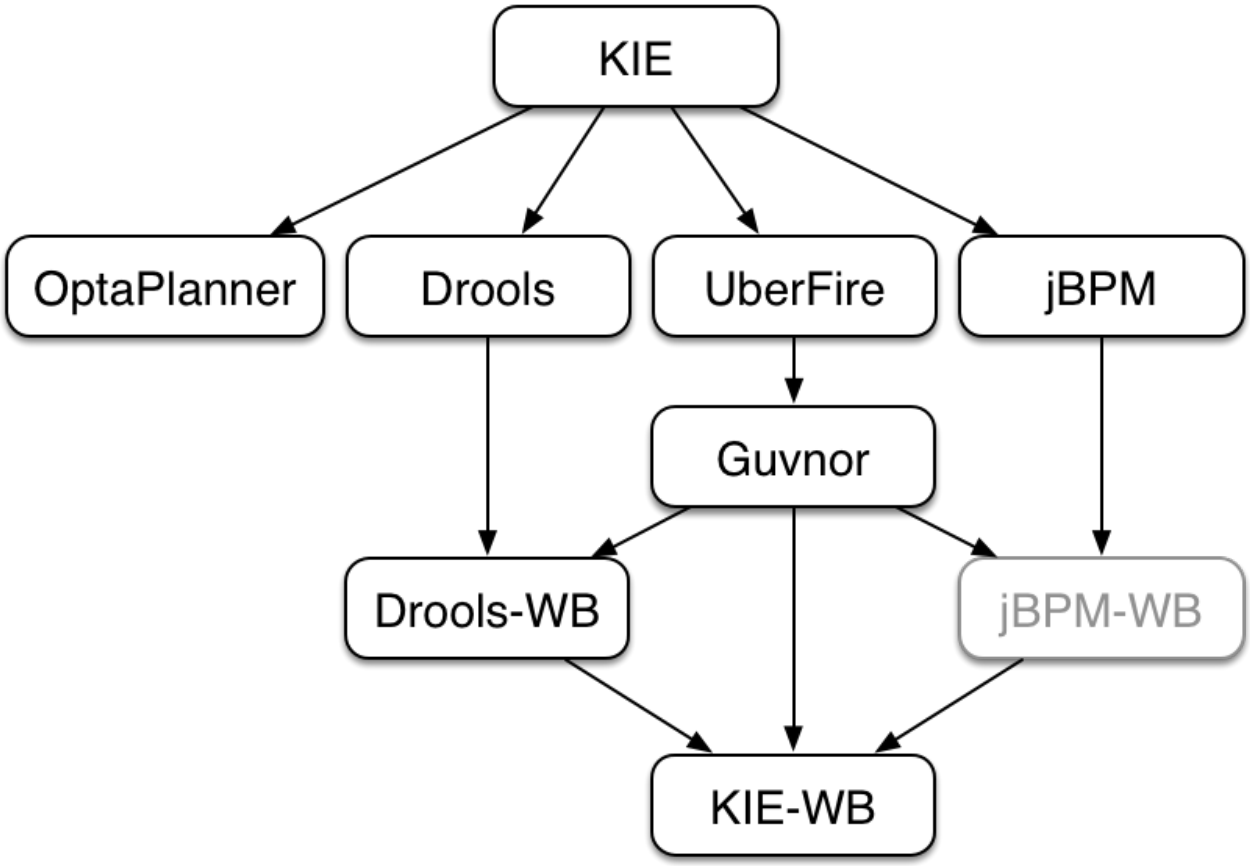


Figure 2.37. KIE Anatomy

2.8.2. Maven aligned projects and modules and Maven Deployment

One of the biggest complaints during the 5.x series was the lack of defined methodology for deployment. The mechanism used by Drools and jBPM was very flexible, but it was too flexible. A big focus for 6.0 was streamlining the build, deploy and loading (utilization) aspects of the sys-

tem. Building and deploying activities are now aligned with Maven and Maven repositories. The utilization for loading rules and processess is now convention and configuration oriented, instead of programmatic, with sane defaults to minimise the configuration.

Projects can be built with Maven and installed to the local M2_REPO or remote Maven repositories. Maven is then used to declare and build the classpath of dependencies, for KIE to access.

2.8.3. Configuration and convention based projects

The 'kmodule.xml' provides declarative configuration for KIE projects. Conventions and defaults are used to reduce the amount of configuration needed.

Example 2.6. Declare KieBases and KieSessions

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase1" packages="org.mypackages>
    <ksession name="ksession1"/>
  </kbase>
</kmodule>
```

Example 2.7. Utilize the KieSession

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

2.8.4. KieBase Inclusion

It is possible to include all the KIE artifacts belonging to a KieBase into a second KieBase. This means that the second KieBase, in addition to all the rules, function and processes directly defined into it, will also contain the ones created in the included KieBase. This inclusion can be done declaratively in the kmodule.xml file

Example 2.8. Including a KieBase into another declaratively

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="kbase2" includes="kbase1">
    <ksession name="ksession2"/>
  </kbase>
</kmodule>
```

or programmatically using the `KieModuleModel`.

Example 2.9. Including a KieBase into another programmatically

```
KieModuleModel kmodule = KieServices.Factory.get().newKieModuleModel();
KieBaseModel kieBaseModel1 = kmodule.newKieBaseModel("KBase2").addInclude("KBase1");
```

2.8.5. KieModules, KieContainer and KIE-CI

Any Maven produced JAR with a 'kmodule.xml' in it is considered a KieModule. This can be loaded from the classpath or dynamically at runtime from a Resource location. If the kie-ci dependency is on the classpath it embeds Maven and all resolving is done automatically using Maven and can access local or remote repositories. Settings.xml is obeyed for Maven configuration.

The KieContainer provides a runtime to utilize the KieModule, versioning is built in throughout, via Maven. Kie-ci will create a classpath dynamically from all the Maven declared dependencies for the artifact being loaded. Maven LATEST, SNAPSHOT, RELEASE and version ranges are supported.

Example 2.10. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.newKieContainer(
        ks.newReleaseId("org.mygroup", "myartefact", "1.0") );

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

KieContainers can be dynamically updated to a specific version, and resolved through Maven if KIE-CI is on the classpath. For stateful KieSessions the existing sessions are incrementally updated.

Example 2.11. Dynamically Update - Java

```
KieContainer kContainer.updateToVersion(
                ks.newReleaseId("org.mygroup", "myartefact", "1.1") );
```

2.8.6. KieScanner

The `KieScanner` is a Maven-oriented replacement of the KnowledgeAgent present in Drools 5. It continuously monitors your Maven repository to check if a new release of a Kie project has been installed and if so, deploys it in the `KieContainer` wrapping that project. The use of the `KieScanner` requires kie-ci.jar to be on the classpath.

A `KieScanner` can be registered on a `KieContainer` as in the following example.

Example 2.12. Registering and starting a KieScanner on a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "myartifact", "1.0-SNAPSHOT" );
KieContainer kContainer = kieServices.newKieContainer( releaseId );
KieScanner kScanner = kieServices.newKieScanner( kContainer );

// Start the KieScanner polling the Maven repository every 10 seconds
kScanner.start( 10000L );
```

In this example the `KieScanner` is configured to run with a fixed time interval, but it is also possible to run it on demand by invoking the `scanNow()` method on it. If the `KieScanner` finds, in the Maven repository, an updated version of the Kie project used by that `KieContainer` it automatically downloads the new version and triggers an incremental build of the new project. From this moment all the new `KieBase`s and `KieSession`s created from that `KieContainer` will use the new project version.

2.8.7. Hierarchical ClassLoader

The CompositeClassLoader is no longer used; as it was a constant source of performance problems and bugs. Traditional hierarchical classloaders are now used. The root classloader is at the KieContext level, with one child ClassLoader per namespace. This makes it cleaner to add and remove rules, but there can now be no referencing between namespaces in DRL files; i.e. functions can only be used by the namespaces that declared them. The recommendation is to use static Java methods in your project, which is visible to all namespaces; but those cannot (like other classes on the root KieContainer ClassLoader) be dynamically updated.

2.8.8. Legacy API Adapter

The 5.x API for building and running with Drools and jBPM is still available through Maven dependency "knowledge-api-legacy5-adapter". Because the nature of deployment has significantly changed in 6.0, it was not possible to provide an adapter bridge for the KnowledgeAgent. If any other methods are missing or problematic, please open a JIRA, and we'll fix for 6.1

2.8.9. KIE Documentation

While a lot of new documentation has been added for working with the new KIE API, the entire documentation has not yet been brought up to date. For this reason there will be continued references to old terminologies. Apologies in advance, and thank you for your patience. We hope those in the community will work with us to get the documentation updated throughout, for 6.1

2.9. What is New and Noteworthy in Drools 6.0.0

2.9.1. PHREAK - Lazy rule matching algorithm

The main work done for Drools in 6.0 involves the new PREAK algorithm. This is a lazy algorithm that should enable Drools to handle a larger number of rules and facts. AngendaGroups can now help improvement performance, as rules are not evaluated until it attempts to fire them.

Sequential mode continues to be supported for PHREAK but now 'modify' is allowed. While there is no 'inference' with sequential configuration, as rules are lazily evaluated, any rule not yet evaluated will see the more recent data as a result of 'modify'. This is more inline with how people intuitively think sequential works.

The conflict resolution order has been tweaked for PHREAK, and now is ordered by salience and then rule order; based on the rule position in the file.. Prior to Drools 6.0.0, after salience, it was considered arbitrary. When KieModules and updateToVersion are used for dynamic deployment, the rule order in the file is preserved via the diff processing.

2.9.2. Automatically firing timed rule in passive mode

When the rule engine runs in passive mode (i.e.: using fireAllRules) by default it doesn't fire consequences of timed rules unless fireAllRules isn't invoked again. Now it is possible to change this default behavior by configuring the KieSession with a `TimedRuleExectionOption` as shown in the following example.

Example 2.13. Configuring a KieSession to automatically execute timed rules

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption( TimedRuleExectionOption.YES );
KSession ksession = kbase.newKieSession(ksconf, null);
```

It is also possible to have a finer grained control on the timed rules that have to be automatically executed. To do this it is necessary to set a `FILTERED TimedRuleExectionOption` that allows to define a callback to filter those rules, as done in the next example.

Example 2.14. Configuring a filter to choose which timed rules should be automatically executed

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
conf.setOption( new TimedRuleExectionOption.FILTERED(new TimedRuleExecutionFilter() {
    public boolean accept(Rule[] rules) {
        return rules[0].getName().equals("MyRule");
    }
```

```
}) );
```

2.9.3. Expression Timers

It is now possible to define both the delay and interval of an interval timer as an expression instead of a fixed value. To do that it is necessary to declare the timer as an expression one (indicated by "expr:") as in the following example:

Example 2.15. An Expression Timer Example

```
declare Bean
    delay   : String = "30s"
    period  : long = 60000
end

rule "Expression timer"
    timer( expr: $d, $p )
when
    Bean( $d : delay, $p : period )
then
end
```

The expressions, $d and $p in this case, can use any variable defined in the pattern matching part of the rule and can be any String that can be parsed in a time duration or any numeric value that will be internally converted in a long representing a duration expressed in milliseconds.

Both interval and expression timers can have 3 optional parameters named "start", "end" and "repeat-limit". When one or more of these parameters are used the first part of the timer definition must be followed by a semicolon ';' and the parameters have to be separated by a comma ',' as in the following example:

Example 2.16. An Interval Timer with a start and an end

```
timer (int: 30s 10s; start=3-JAN-2010, end=5-JAN-2010)
```

The value for start and end parameters can be a Date, a String representing a Date or a long, or more in general any Number, that will be transformed in a Java Date applying the following conversion:

```
new Date( ((Number) n).longValue() )
```

Conversely the repeat-limit can be only an integer and it defines the maximum number of repetitions allowed by the timer. If both the end and the repeat-limit parameters are set the timer will stop when the first of the two will be matched.

The using of the start parameter implies the definition of a phase for the timer, where the beginning of the phase is given by the start itself plus the eventual delay. In other words in this case the timed rule will then be scheduled at times:

```
start + delay + n*period
```

for up to repeat-limit times and no later than the end timestamp (whichever first). For instance the rule having the following interval timer

```
timer ( int: 30s 1m; start="3-JAN-2010" )
```

will be scheduled at the 30th second of every minute after the midnight of the 3-JAN-2010. This also means that if for example you turn the system on at midnight of the 3-FEB-2010 it won't be scheduled immediately but will preserve the phase defined by the timer and so it will be scheduled for the first time 30 seconds after the midnight. If for some reason the system is paused (e.g. the session is serialized and then deserialized after a while) the rule will be scheduled only once to recover from missing activations (regardless of how many activations we missed) and subsequently it will be scheduled again in phase with the timer.

2.9.4. RuleFlowGroups and AgendaGroups are merged

These two groups have been merged and now RuleFlowGroups behave the same as AgendaGroups. The get methods have been left, for deprecation reasons, but both return the same underlying data. When jBPM activates a group it now just calls setFocus. RuleFlowGroups and AgendaGroups when used together was a continued source of errors. It also aligns the codebase, towards PHREAK and the multi-core explotation that is planned in the future.

2.10. New and Noteworthy in KIE Workbench 6.0.0

The workbench has had a big overhaul using a new base project called UberFire. UberFire is inspired by Eclipse and provides a clean, extensible and flexible framework for the workbench. The end result is not only a richer experience for our end users, but we can now develop more rapidly with a clean component based architecture. If you like he Workbench experience you can use UberFire today to build your own web based dashboard and console efforts.

As well as the move to a UberFire the other biggest change is the move from JCR to Git; there is an utility project to help with migration. Git is the most scalable and powerful source repository bar none. JGit provides a solid OSS implementation for Git. This addresses the continued performance problems with the various JCR implementations, which would slow down once the number of files and number of versions become too high. There has been a big "low tech" drive, to remove complexity. Everything is now stored as a file, including meta data. The database is only there to provide fast indexing and search. So importing and exporting is all standard Git and external sites, like GitHub, can be used to exchange repositories.

In 5.x developers would work with their own source repository and then push JCR, via the team provider. This team provider was not full featured and not available outside Eclipse. Git enables our repository to work any existing Git tool or team provider. While not yet supported in the UI, this will be added over time, it is possible to connect to the repo and tag and branch and restore things.

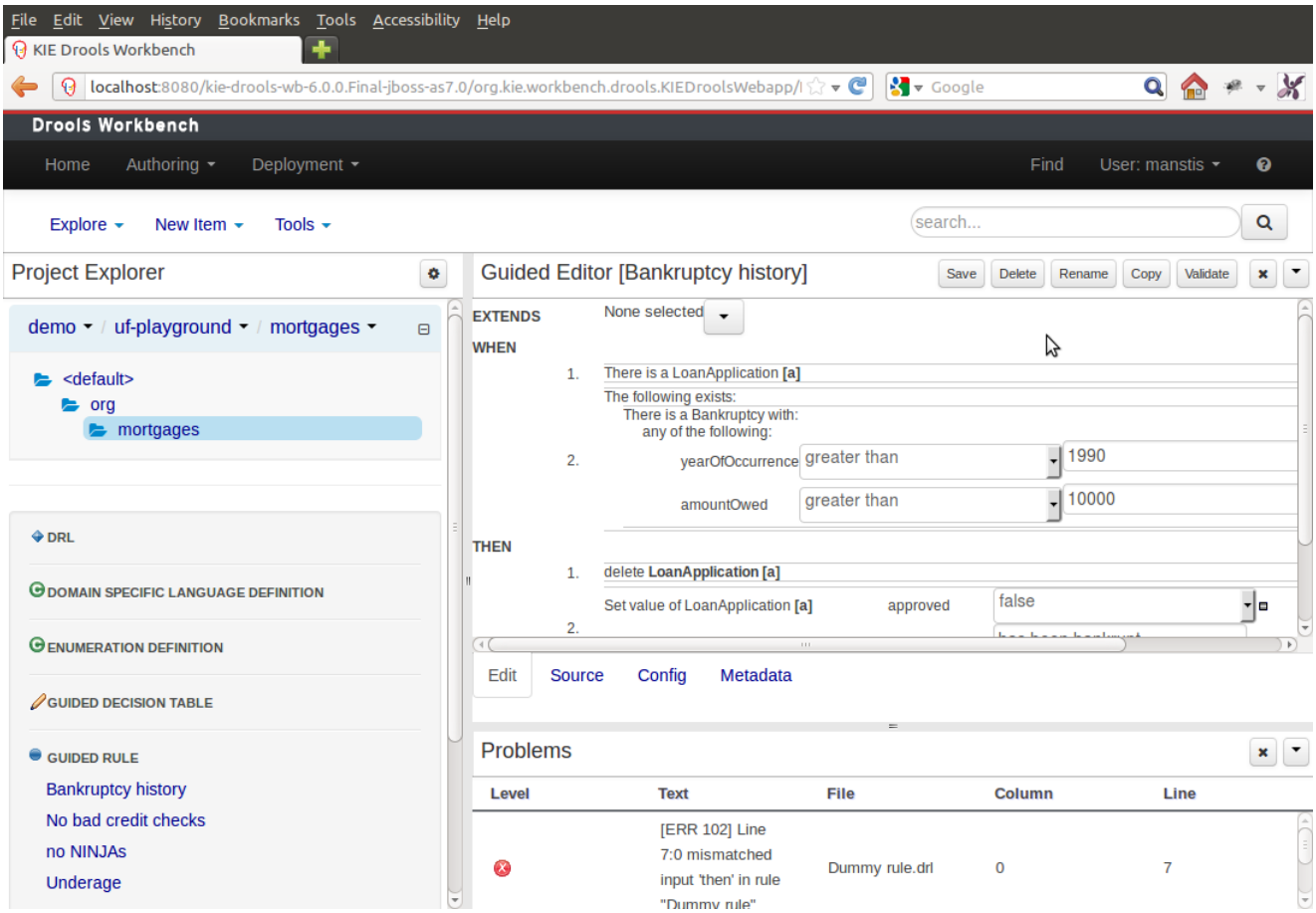


Figure 2.38. Workbench

The Guvnor brand leaked too much from its intended role; such as the authoring metaphors, like Decision Tables, being considered Guvnor components instead of Drools components. This wasn't helped by the monolithic projects structure used in 5.x for Guvnor. In 6.0 Guvnor 's focus has been narrowed to encapsulates the set of UberFire plugins that provide the basis for building a web based IDE. Such as Maven integration for building and deploying, management of Maven repositories and activity notifications via inboxes. Drools and jBPM build workbench distributions using Uberfire as the base and including a set of plugins, such as Guvnor, along with their own plugins for things like decision tables, guided editors, BPMN2 designer, human tasks.

The "Model Structure" diagram outlines the new project anatomy. The Drools workbench is called KIE-Drools-WB. KIE-WB is the uber workbench that combines all the Guvnor, Drools and jBPM plugins. The jBPM-WB is ghosted out, as it doesn't actually exist, being made redundant by KIE-WB.

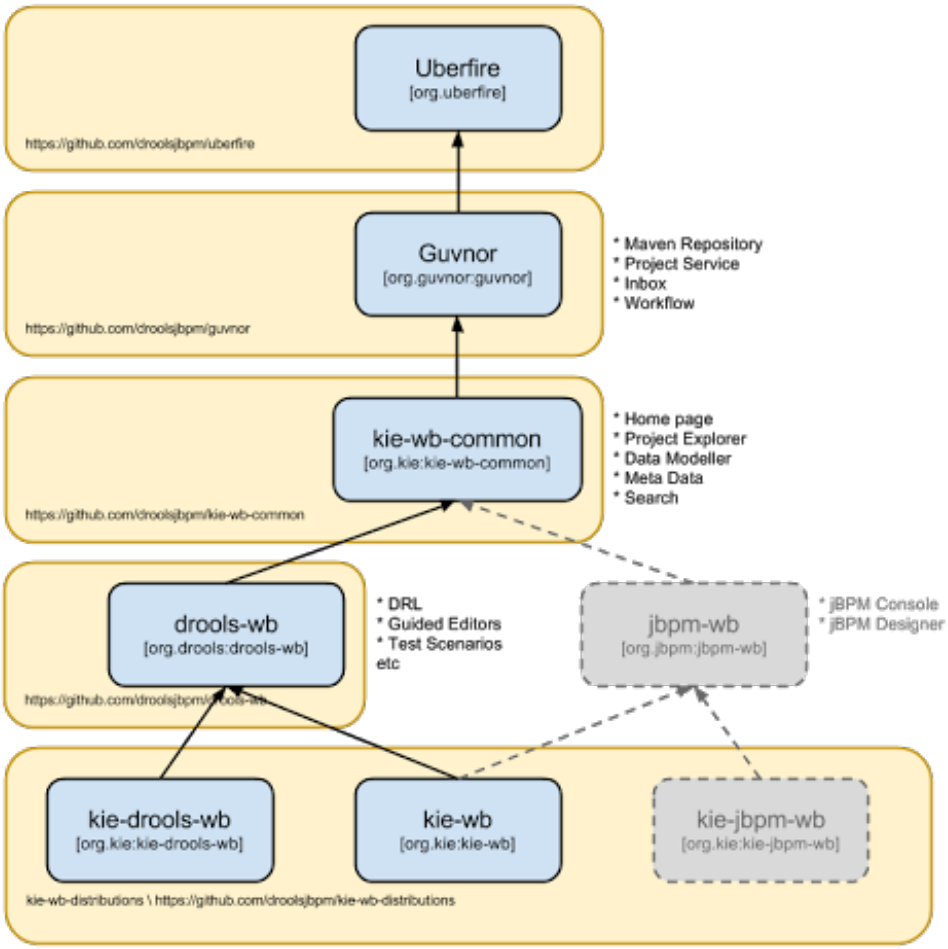


Figure 2.39. Module Structure

Important

KIE Drools Workbench and KIE Workbench share a common set of components for generic workbench functionality such as Project navigation, Project definitions, Maven based Projects, Maven Artifact Repository. These common features are described in more detail throughout this documentation.

The two primary distributions consist of:

- KIE Drools Workbench
  - Drools Editors, for rules and supporting assets.
  - jBPM Designer, for Rule Flow and supporting assets.
- KIE Workbench

- Drools Editors, for rules and supporting assets.
- jBPM Designer, for BPMN2 and supporting assets.
- jBPM Console, runtime and Human Task support.
- jBPM Form Builder.
- BAM.

Workbench highlights:

- New flexible Workbench environment, with perspectives and panels.
- New packaging and build system following KIE API.
  - Maven based projects.
  - Maven Artifact Repository replaces Global Area, with full dependency support.
- New Data Modeller replaces the declarative Fact Model Editor; bringing authoring of Java classes to the authoring environment. Java classes are packaged into the project and can be used within rules, processes etc and externally in your own applications.
- Virtual File System replaces JCR with a default Git based implementation.
  - Default Git based implementation supports remote operations.
  - External modifications appear within the Workbench.
- Incremental Build system showing, near real-time validation results of your project and assets.

The editors themselves are largely unchanged; however of note imports have moved from the package definition to individual editors so you need only import types used for an asset and not the package as a whole.

2.11. New and Noteworthy in Integration 6.0.0

2.11.1. CDI

CDI is now tightly integrated into the KIE API. It can be used to inject versioned KieSession and KieBases.

```
@Inject
@KSession("kbase1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private KieBase kbase1v10;

@Inject
@KBase("kbase1")
```

```
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private KieBase kbase1v10;
```

Figure 2.40. Side by side version loading for 'jar1.KBase1' KieBase

```
@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private KieSession ksessionv10;

@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private KieSession ksessionv11;
```

Figure 2.41. Side by side version loading for 'jar1.KBase1' KieBase

2.11.2. Spring

Spring has been revamped and now integrated with KIE. Spring can replace the 'kmodule.xml' with a more powerful spring version. The aim is for consistency with kmodule.xml

2.11.3. Aries Blueprints

Aries blueprints is now also supported, and follows the work done for spring. The aim is for consistency with spring and kmodule.xml

2.11.4. OSGi Ready

All modules have been refactored to avoid package splitting, which was a problem in 5.x. Testing has been moved to PAX.

Chapter 3. Compatibility matrix

Starting from KIE 6.0, Drools (including workbench), jBPM (including designer and console) and OptaPlanner follow the same version numbering.

Part II. KIE

KIE is the shared core for Drools and jBPM. It provides a unified methodology and programming model for building, deploying and utilizing resources.

Chapter 4. KIE

4.1. Overview

4.1.1. Anatomy of Projects

The process of researching an integration knowledge solution for Drools and jBPM has simply used the "droolsjbpm" group name. This name permeates GitHub accounts and Maven POMs. As scopes broadened and new projects were spun KIE, an acronym for Knowledge Is Everything, was chosen as the new group name. The KIE name is also used for the shared aspects of the system; such as the unified build, deploy and utilization.

KIE currently consists of the following subprojects:

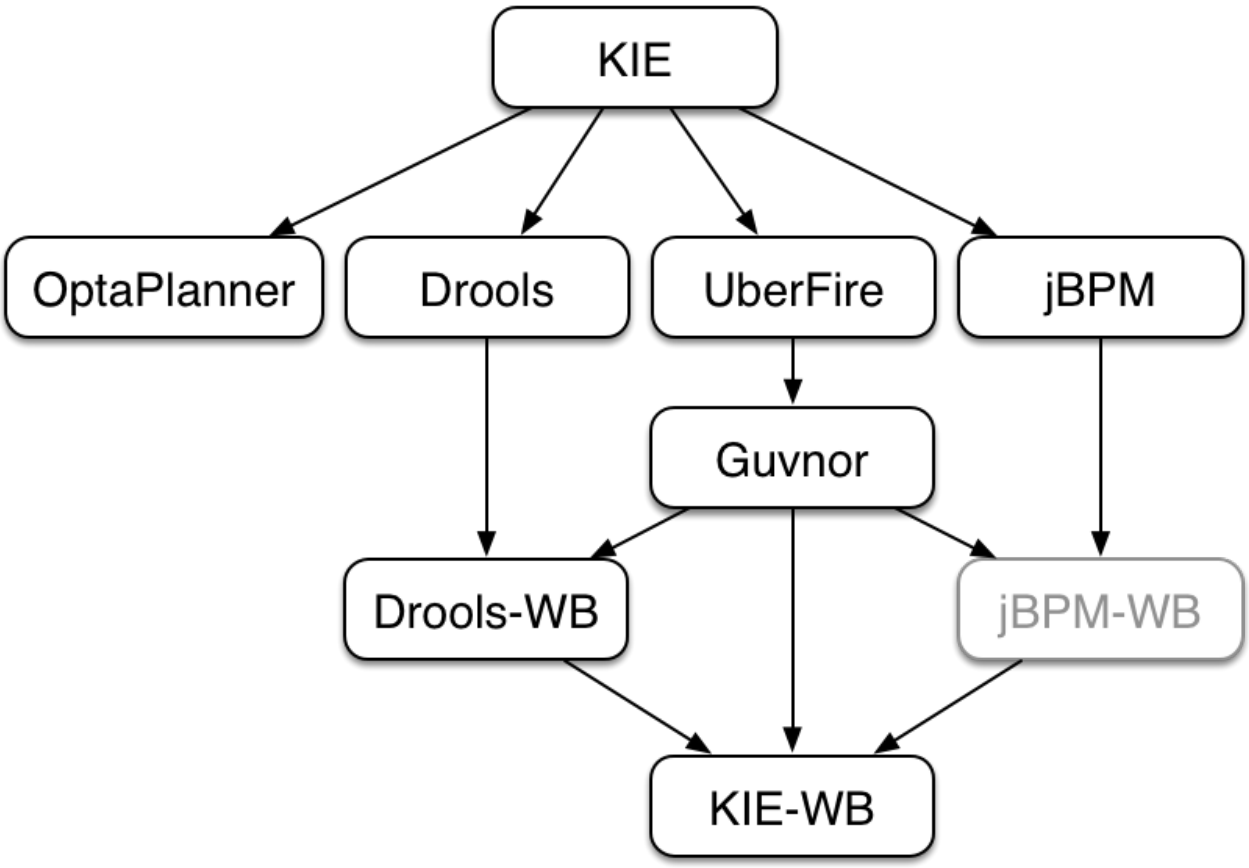


Figure 4.1. KIE Anatomy

OptaPlanner, a local search and optimization tool, has been spun off from Drools Planner and is now a top level project with Drools and jBPM. This was a natural evolution as Optaplanner, while having strong Drools integration, has long been independant of Drools.

From the Polymita acquisition, along with other things, comes the powerful Dashboard Builder which provides powerful reporting capabilities. Dashboard Builder is currently a temporary name and after the 6.0 release a new name will be chosen. Dashboard Builder is completely independant of Drools and jBPM and will be used by many projects at JBoss, and hopefully outside of JBoss :)

UberFire is the new base workbench project, spun off from the ground up rewrite. UberFire provides Eclipse-like workbench capabilities, with panels and perspectives from plugins. The project is independant of Drools and jBPM and anyone can use it as a basis of building flexible and powerful workbenches. UberFire will be used for console and workbench development throughout JBoss.

It was determined that the Guvnor brand leaked too much from its intended role; such as the authoring metaphors, like Decision Tables, being considered Guvnor components instead of Drools components. This wasn't helped by the monolithic projects structure used in 5.x for Guvnor. In 6.0 Guvnor's focus has been narrowed to encapsulate the set of UberFire plugins that provide the basis for building a web based IDE. Such as Maven integration for building and deploying, management of Maven repositories and activity notifications via inboxes. Drools and jBPM build workbench distributions using Uberfire as the base and including a set of plugins, such as Guvnor, along with their own plugins for things like decision tables, guided editors, BPMN2 designer, human tasks. The Drools workbench is called Drools-WB. KIE-WB is the uber workbench that combined all the Guvnor, Drools and jBPM plugins. The jBPM-WB is ghosted out, as it doesn't actually exist, being made redundant by KIE-WB.

4.1.2. Lifecycles

The different aspects, or life cycles, of working with KIE system, whether it's Drools or jBPM, can typically be broken down into the following:

- **Author**
  - Authoring of knowledge using a UI metaphor, such as: DRL, BPMN2, decision table, class models.
- **Build**
  - Builds the authored knowledge into deployable units.
  - For KIE this unit is a JAR.
- **Test**
  - Test KIE knowedge before it's deployed to the application.
- **Deploy**
  - Deploys the unit to a location where applications may utilize (consume) them.

- KIE uses Maven style repository.

- **Utilize**
  - The loading of a JAR to provide a KIE session (KieSession), for which the application can interact with.
  - KIE exposes the JAR at runtime via a KIE container (KieContainer).
  - KieSessions, for the runtime's to interact with, are created from the KieContainer.
- **Run**
  - System interaction with the KieSession, via API.
- **Work**
  - User interaction with the KieSession, via command line or UI.
- **Manage**
  - Manage any KieSession or KieContainer.

4.2. Build, Deploy, Utilize and Run

4.2.1. Introduction

6.0 introduces a new configuration and convention approach to building knowledge bases, instead of using the programmatic builder approach in 5.x. The builder is still available to fall back on, as it's used for the tooling integration.

Building now uses Maven, and aligns with Maven practices. A KIE project or module is simply a Maven Java project or module; with an additional metadata file META-INF/kmodule.xml. The kmodule.xml file is the descriptor that selects resources to knowledge bases and configures those knowledge bases and sessions. There is also alternative XML support via Spring and OSGi Blue-Prints.

While standard Maven can build and package KIE resources, it will not provide validation at build time. There is a Maven plugin which is recommended to use to get build time validation. The plugin also generates many classes, making the runtime loading faster too.

The example project layout and Maven POM descriptor is illustrated in the screenshot

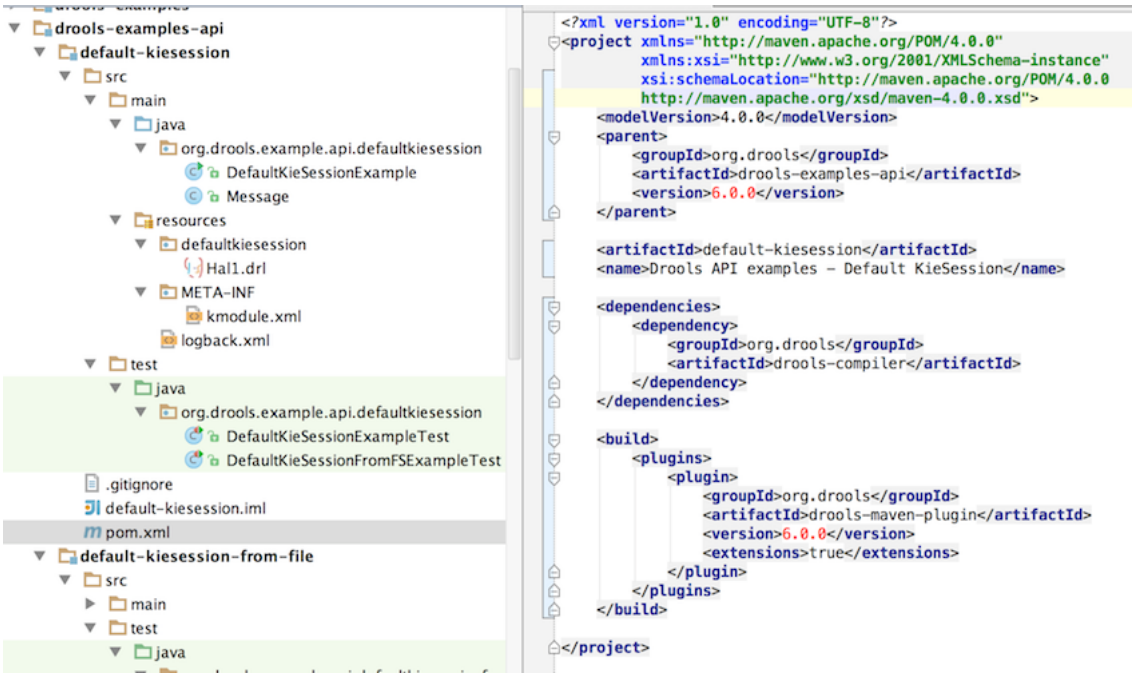


Figure 4.2. Example project layout and Maven POM

KIE uses defaults to minimise the amount of configuration. With an empty kmodule.xml being the simplest configuration. There must always be a kmodule.xml file, even if empty, as it's used for discovery of the JAR and its contents.

Maven can either 'mvn install' to deploy a KieModule to the local machine, where all other applications on the local machine use it. Or it can 'mvn deploy' to push the KieModule to a remote Maven repository. Building the Application will pull in the KieModule and populate the local Maven repository in the process.

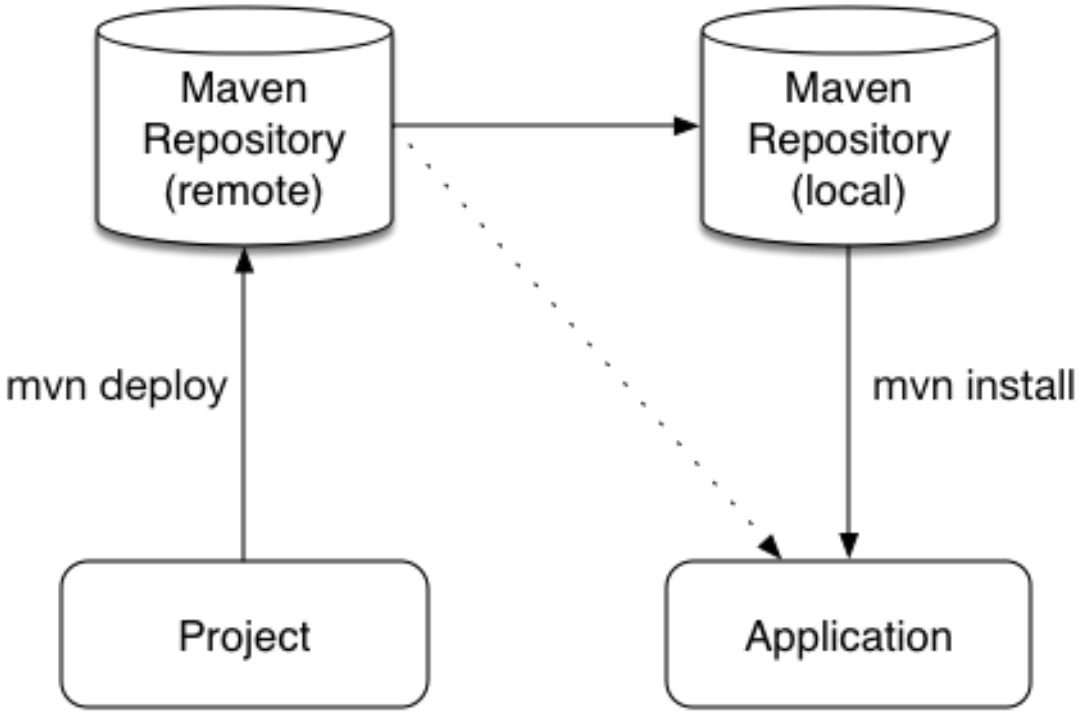


Figure 4.3. Example project layout and Maven POM

JARs can be deployed in one of two ways. Either added to the classpath, like any other JAR in a Maven dependency listing, or they can be dynamically loaded at runtime. KIE will scan the classpath to find all the JARs with a kmodule.xml in it. Each found JAR is represented by the KieModule interface. The terms classpath KieModule and dynamic KieModule are used to refer to the two loading approaches. While dynamic modules supports side by side versioning, classpath modules do not. Further once a module is on the classpath, no other version may be loaded dynamically.

Detailed references for the API are included in the next sections, the impatient can jump straight to the examples section, which is fairly self-explanatory on the different use cases.

4.2.2. Building

Figure 4.4. org.kie.api.core.builder

4.2.2.1. Creating and building a Kie Project

A Kie Project has the structure of a normal Maven project with the only peculiarity of including a kmodule.xml file defining in a declaratively way the `KieBase`s and `KieSession`s that can be created from it. This file has to be placed in the resources/META-INF folder of the Maven project while all the other Kie artifacts, such as DRL or a Excel files, must be stored in the resources folder or in any other subfolder under it.

Since meaningful defaults have been provided for all configuration aspects, the simplest kmodule.xml file can contain just an empty kmodule tag like the following:

Example 4.1. An empty kmodule.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"/>
```

In this way the kmodule will contain one single default `KieBase`. All Kie assets stored under the resources folder, or any of its subfolders, will be compiled and added to it. To trigger the building of these artifacts it is enough to create a `KieContainer` for them.

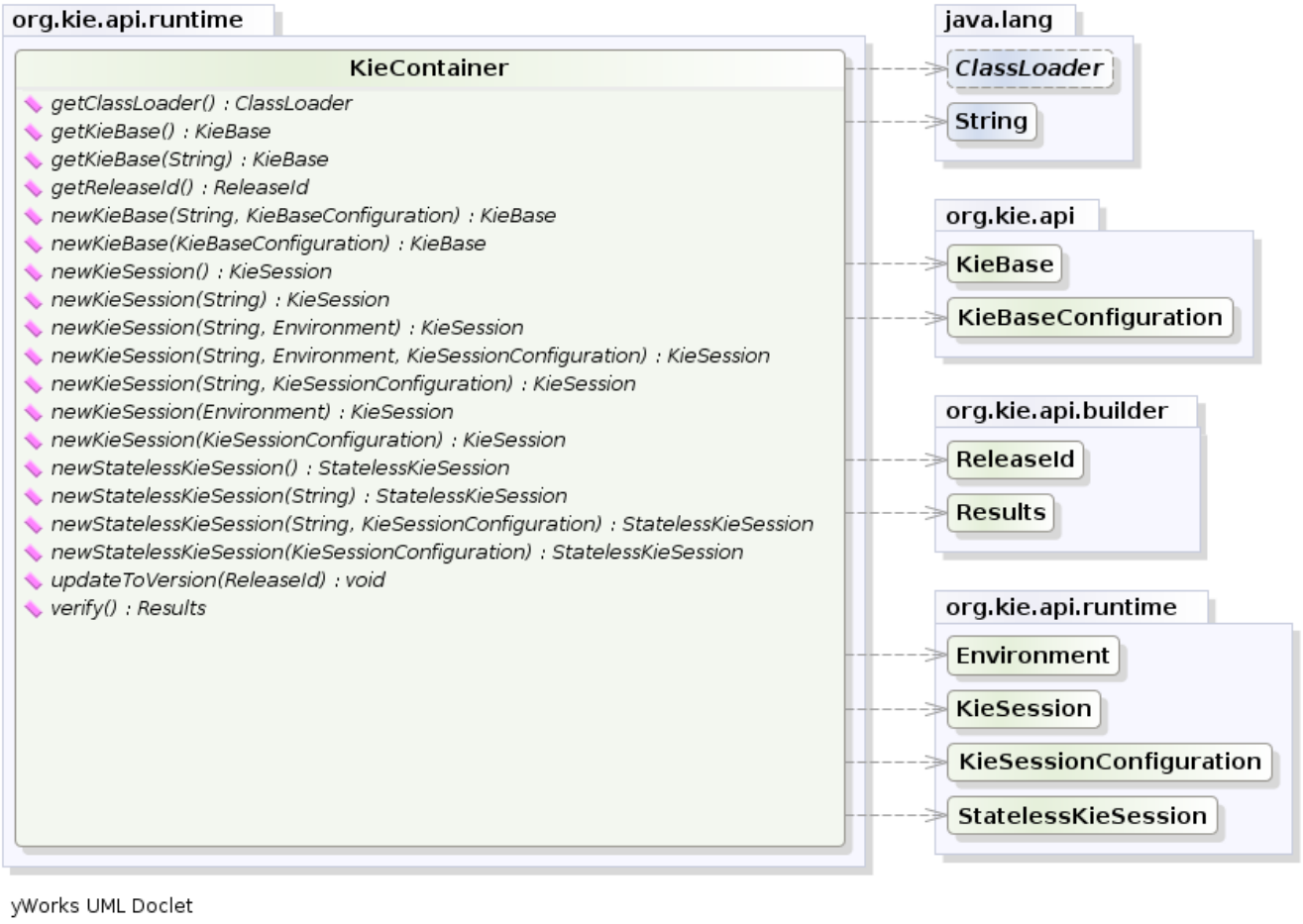


Figure 4.5. KieContainer

For this simple case it is enough to create a `KieContainer` that reads the files to be built from the classpath:

Example 4.2. Creating a KieContainer from the classpath

```
KieServices kieServices = KieServices.Factory.get();
```

```
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

`KieServices` is the interface from where it possible to access all the Kie building and runtime facilities:

Figure 4.6. KieServices

In this way all the Java sources and the Kie resources are compiled and deployed into the KieContainer which makes its contents available for use at runtime.

4.2.2.2. The kmodule.xml file

As explained in the former section, the kmodule.xml file is the place where it is possible to declaratively configure the `KieBase`(s) and `KieSession`(s) that can be created from a KIE project.

In particular a `KieBase` is a repository of all the application's knowledge definitions. It will contain rules, processes, functions, and type models. The `KieBase` itself does not contain data; instead, sessions are created from the `KieBase` into which data can be inserted and from which process instances may be started. Creating the `KieBase` can be heavy, whereas session creation is very light, so it is recommended that `KieBase` be cached where possible to allow for repeated session creation. However end-users usually shouldn't worry about it, because this caching mechanism is already automatically provided by the `KieContainer`.

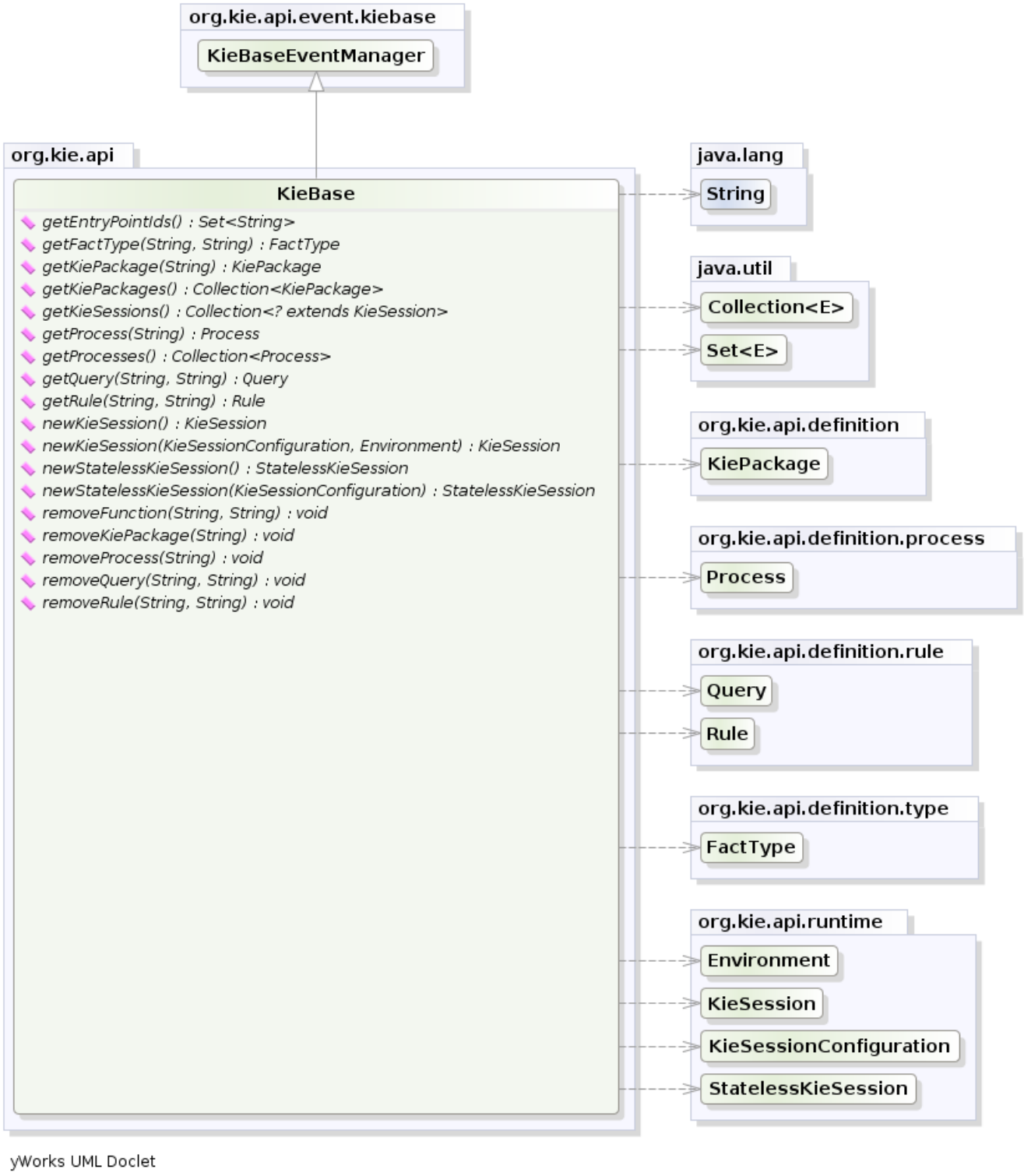


Figure 4.7. KieBase

Conversely the `KieSession` stores and executes on the runtime data. It is created from the `KieBase` or more easily can be created directly from the `KieContainer` if it has been defined in the kmodule.xml file

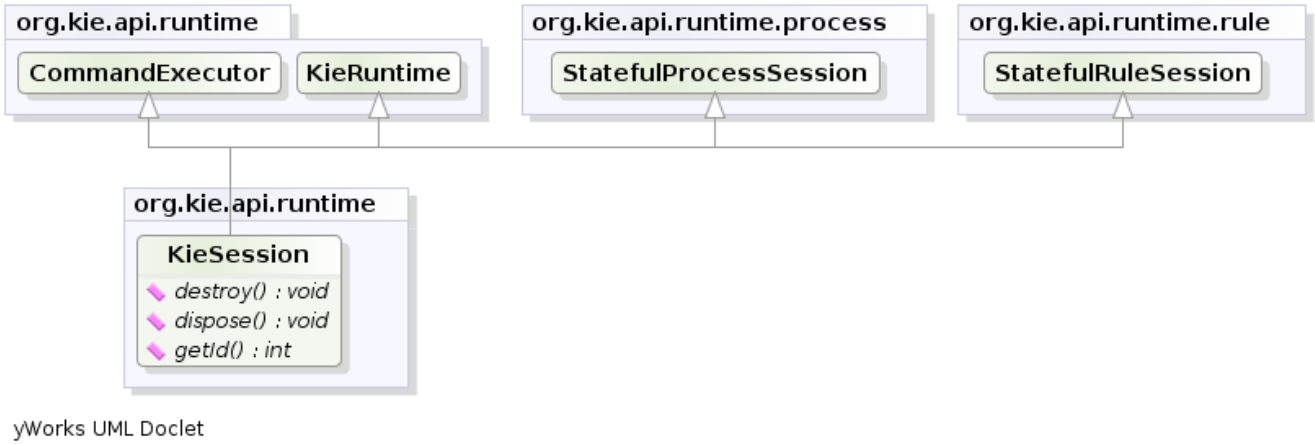


Figure 4.8. KieSession

The kmodule.xml allows to define and configure one or more `KieBase`s and for each `KieBase` all the different `KieSession`s that can be created from it, as showed by the follwing example:

Example 4.3. A sample kmodule.xml file

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <configuration>

<propertykey="drools.evaluator.supersetOf"value="org.mycompany.SupersetOfEvaluatorDefinition"/
>
  </configuration>
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality" declarativeAgenda="ena
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_2" type="stateless" default="false" beliefSystem="jtms"/>
  </kbase>
name="KBase2" default="false" eventProcessingMode="equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,
 org.domain.pkg3" includes="KBase1">
    <ksession name="KSession3_1" type="stateful" default="false" clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <workItemHandlers>
        <workItemHandler name="name" type="org.domain.WorkItemHandler"/>
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener"/>
        <agendaEventListener type="org.domain.FirstAgendaListener"/>
        <agendaEventListener type="org.domain.SecondAgendaListener"/>
        <processEventListener type="org.domain.ProcessListener"/>
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

Here the <configuration> tag contains a list of key-value pairs that are the optional properties used to configure the `KieBase`s building process. For instance this sample kmodule.xml

file defines an additional custom operator named `supersetOf` and implemented by the `org.mycompany.SupersetOfEvaluatorDefinition` class.

After this 2 `KieBase`s have been defined and it is possible to instance 2 different types of `KieSessions` from the first one, while only one from the second. A list of the attributes that can be defined on the kbase tag, together with their meaning and default values follows:

Table 4.1. kbase Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	The name with which retrieve this KieBase from the KieContainer. This is the only mandatory attribute.
includes	none	any comma separated list	A comma separated list of other KieBases contained in this kmodule. The artifacts of all these KieBases will be also included in this one.
packages	all	any comma separated list	By default all the Drools artifacts under the resources folder, at any level, are included into the KieBase. This attribute allows to limit the artifacts that will be compiled in this KieBase to only the ones belonging to the list of packages.
default	false	true, false	Defines if this KieBase is the default one for this module, so it can be created from the KieContainer without passing any name to it. There can be at most one default KieBase in each module.

Attribute name	Default value	Admitted values	Meaning
equalsBehavior	identity	identity, equality	Defines the behavior of Drools when a new fact is inserted into the Working Memory. With identity it always create a new FactHandle unless the same object isn't already present in the Working Memory, while with equality only if the newly inserted object is not equal (according to its equal method) to an already existing fact.
eventProcessing-Mode	cloud	cloud, stream	When compiled in cloud mode the KieBase treats events as normal facts, while in stream mode allow temporal reasoning on them.
declarativeAgenda	disabled	disabled, enabled	Defines if the Declarative Agenda is enabled or not.

Similarly all attributes of the ksession tag (except of course the name) have meaningful default. They are listed and described in the following table:

Table 4.2. ksession Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	Unique name of this KieSession. Used to fetch the KieSession from the KieContainer. This is the only mandatory attribute.
type	stateful	stateful, stateless	A stateful session allows to iteratively work with the Working Memory, while a state-

Attribute name	Default value	Admitted values	Meaning
			less one is a one-off execution of a Working Memory with a provided data set.
default	false	true, false	Defines if this KieSession is the default one for this module, so it can be created from the KieContainer without passing any name to it. In each module there can be at most one default KieSession for each type.
clockType	realtime	realtime, pseudo	Defines if events timestamps are determined by the system clock or by a psuedo clock controlled by the application. This clock is specially useful for unit testing temporal rules.
beliefSystem	simple	simple, jtms, defeasible	Defines the type of belief system used by the KieSession.

As outlined in the former kmodule.xml sample, it is also possible to declaratively create on each `KieSession` a file (or a console) logger, one or more `WorkItemHandlers` and some listeners that can be of 3 different types: ruleRuntimeEventListener, agendaEventListener and processEventListener

Having defined a kmodule.xml like the one in the former sample, it is now possible to simply retrieve the KieBases and KieSessions from the KieContainer using their names.

Example 4.4. Retriving KieBases and KieSessions from the KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();

KieBase kBase1 = kContainer.getKieBase("KBase1");
KieSession kieSession1 = kContainer.newKieSession("KSession2_1");
StatelessKieSession kieSession2 = kContainer.newStatelessKieSession("KSession2_2");
```

It has to be noted that since KSession2_1 and KSession2_2 are of 2 different types (the first is stateful, while the second is stateless) it is necessary to invoke 2 different methods on the `KieContainer` according to their declared type. If the type of the `KieSession` requested to the `KieContainer` doesn't correspond with the one declared in the kmodule.xml file the `KieContainer` will throw a `RuntimeException`. Also since a `KieBase` and a `KieSession` have been flagged as default is it possible to get them from the `KieContainer` without passing any name.

Example 4.5. Retriving default KieBases and KieSessions from the KieContainer

```
KieContainer kContainer = ...

KieBase kBase1 = kContainer.getKieBase(); // returns KBase1
KieSession kieSession1 = kContainer.newKieSession(); // returns KSession2_1
```

Since a Kie project is also a Maven project the groupId, artifactId and version declared in the pom.xml file are used to generate a `ReleaseId` that uniquely identifies this project inside your application. This allows creation of a new KieContainer from the project by simply passing its `ReleaseId` to the `KieServices`.

Example 4.6. Creating a KieContainer of an existing project by ReleaseId

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "myartifact", "1.0" );
KieContainer kieContainer = kieServices.newKieContainer( releaseId );
```

4.2.2.3. Building with Maven

The KIE plugin for Maven ensures that artifact resources are validated and pre-compiled, it is recommended that this is used at all times. To use the plugin simply add it to the build section of the Maven pom.xml

Example 4.7. Adding the KIE plugin to a Maven pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>${project.version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Building a KIE module without the Maven plugin will copy all the resources, as is, into the resulting JAR. When that JAR is loaded by the runtime, it will attempt to build all the resources then. If there are compilation issues it will return a null KieContainer. It also pushes the compilation overhead to the runtime. In general this is not recommended, and the Maven plugin should always be used.

4.2.2.4. Defining a KieModule programmatically

It is also possible to define the `KieBase`s and `KieSession`s belonging to a KieModule programmatically instead of the declarative definition in the kmodule.xml file. The same programmatic API also allows in explicitly adding the file containing the Kie artifacts instead of automatically read them from the resources folder of your project. To do that it is necessary to create a `KieFileSystem`, a sort of virtual file system, and add all the resources contained in your project to it.

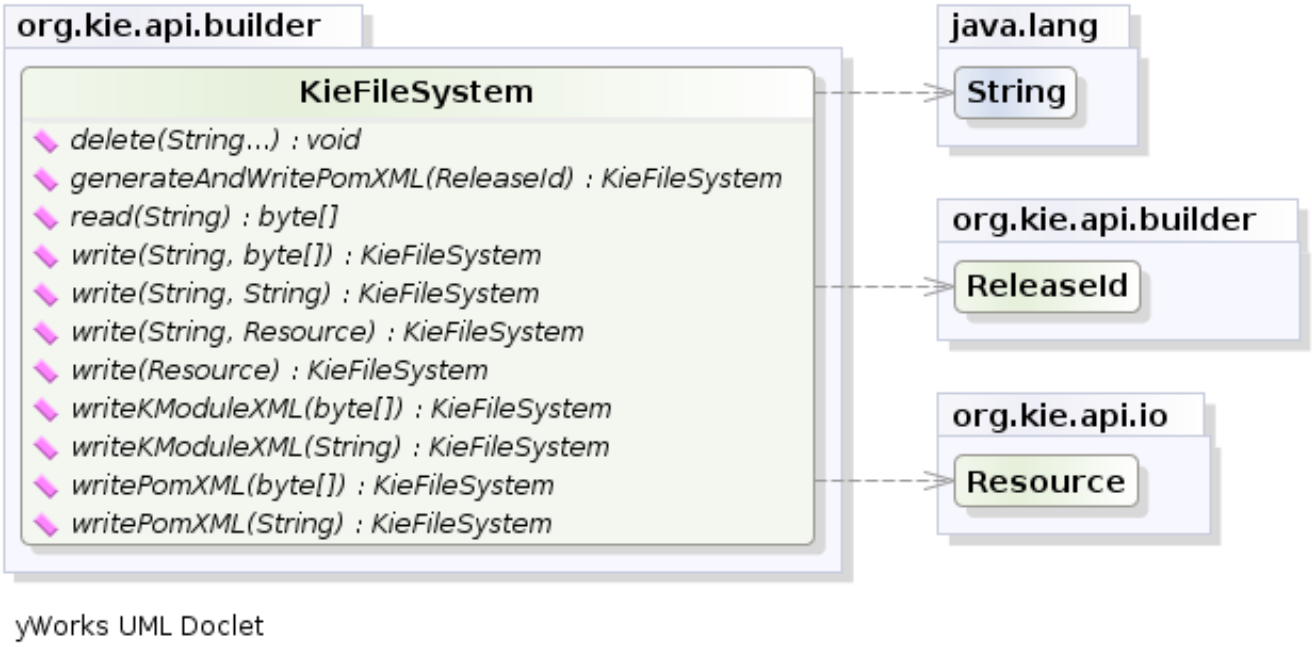


Figure 4.9. KieFileSystem

Like all other Kie core components you can obtain an instance of the `KieFileSystem` from the `KieServices`. The kmodule.xml configuration file must be added to the filesystem. This is a mandatory step. Kie also provides a convenient fluent API, implemented by the `KieModuleModel`, to programmatically create this file.

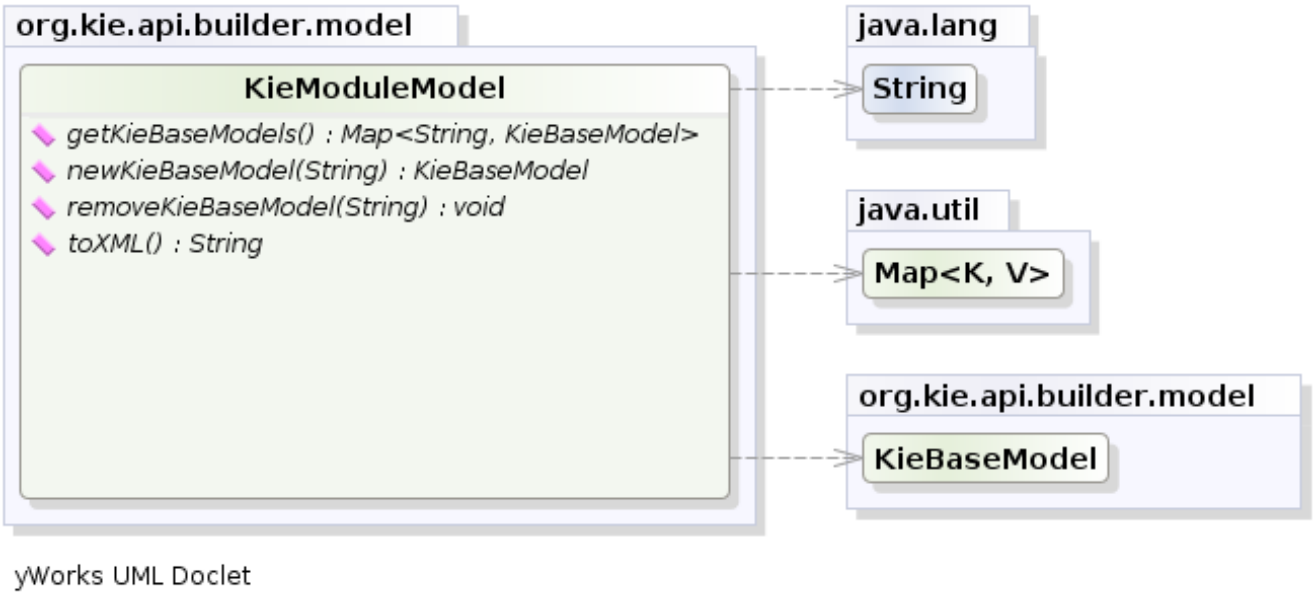


Figure 4.10. KieModuleModel

To do this in practice it is necessary to create a `KieModuleModel` from the `KieServices`, configure it with the desired `KieBases` and `KieSessions`, convert it in XML and add the XML to the `KieFileSystem`. This process is shown by the following example:

Example 4.8. Creating a kmodule.xml programmatically and adding it to a KieFileSystem

```
KieServices kieServices = KieServices.Factory.get();
KieModuleModel kieModuleModel = kieServices.newKieModuleModel();

KieBaseModel kieBaseModel1 = kieModuleModel.newKieBaseModel( "KBase1 ")
        .setDefault( true )
        .setEqualsBehavior( EqualityBehaviorOption.EQUALITY )
        .setEventProcessingMode( EventProcessingOption.STREAM );

KieSessionModel ksessionModel1 = kieBaseModel1.newKieSessionModel( "KSession1" )
        .setDefault( true )
        .setType( KieSessionModel.KieSessionType.STATEFUL )
        .setClockType( ClockTypeOption.get("realtime") );

KieFileSystem kfs = kieServices.newKieFileSystem();
kfs.writeKModuleXML(kieModuleModel.toXML());
```

At this point it is also necessary to add to the `KieFileSystem`, through its fluent API, all others Kie artifacts composing your project. These artifacts have to be added in the same position of a corresponding usual Maven project.

Example 4.9. Adding Kie artifacts to a KieFileSystem

```
KieFileSystem kfs = ...
kfs.write( "src/main/resources/KBase1/ruleSet1.drl", stringContainingAValidDRL )
        .write( "src/main/resources/dtable.xls",
                kieServices.getResources().newInputStreamResource( dtableFileStream ) );
```

This example shows that it is possible to add the Kie artifacts both as plain Strings and as `Resources`. In the latter case the `Resources` can be created by the `KieResources` factory, also provided by the `KieServices`. The `KieResources` provides many convenient factory methods to convert an `InputStream`, a `URL`, a `File`, or a `String` representing a path of your file system to a `Resource` that can be managed by the `KieFileSystem`.

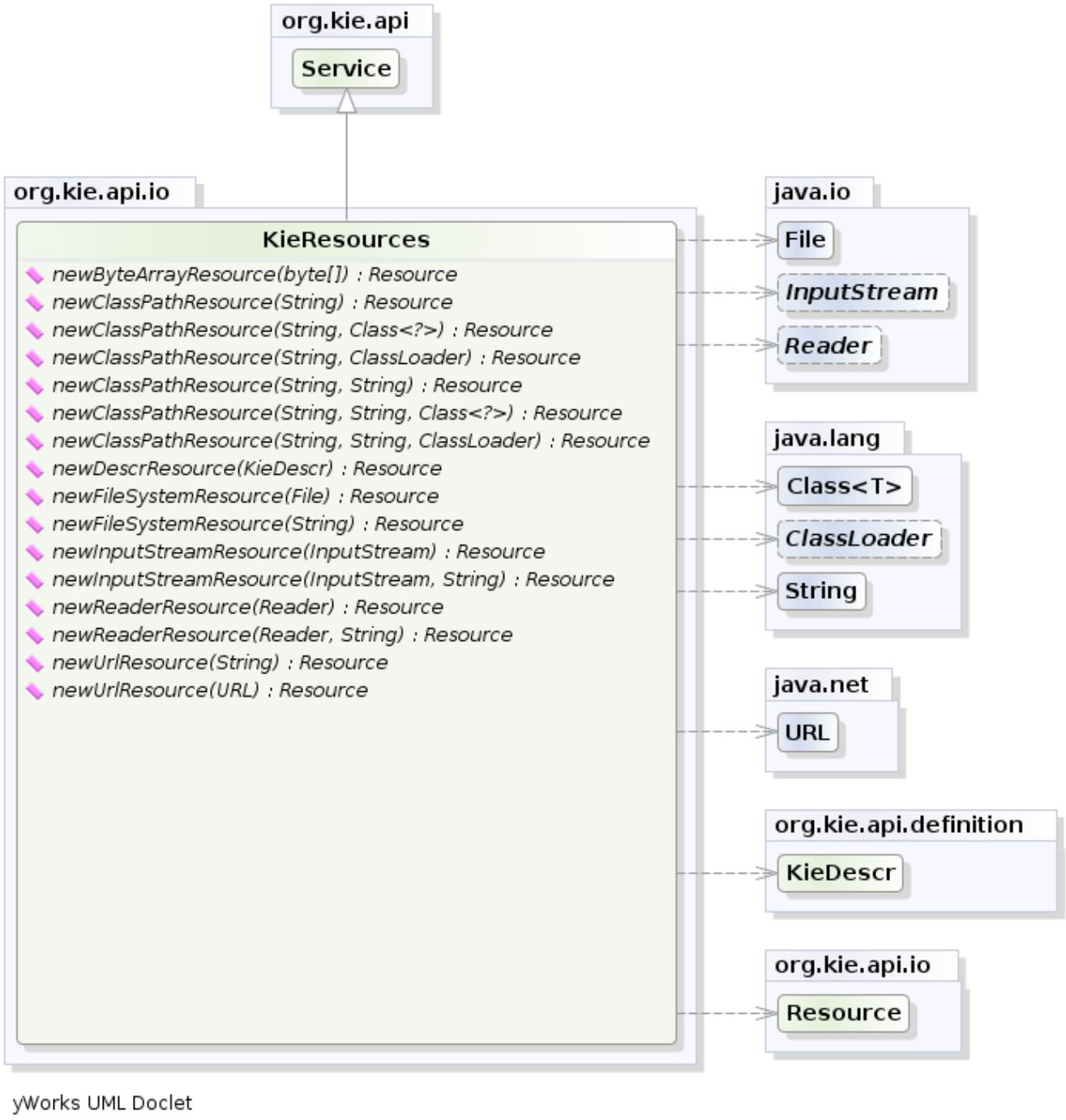


Figure 4.11. KieResources

Normally the type of a `Resource` can be inferred from the extension of the name used to add it to the `KieFileSystem`. However it also possible to not follow the Kie conventions about file extensions and explicitly assign a specific `ResourceType` to a `Resource` as shown below:

Example 4.10. Creating and adding a Resource with an explicit type

```
KieFileSystem kfs = ...
kfs.write( "src/main/resources/myDrl.txt",
           kieServices.getResources().newInputStreamResource( drlStream )
                      .setResourceType(ResourceType.DRL) );
```

Add all the resources to the `KieFileSystem` and build it by passing the `KieFileSystem` to a `KieBuilder`

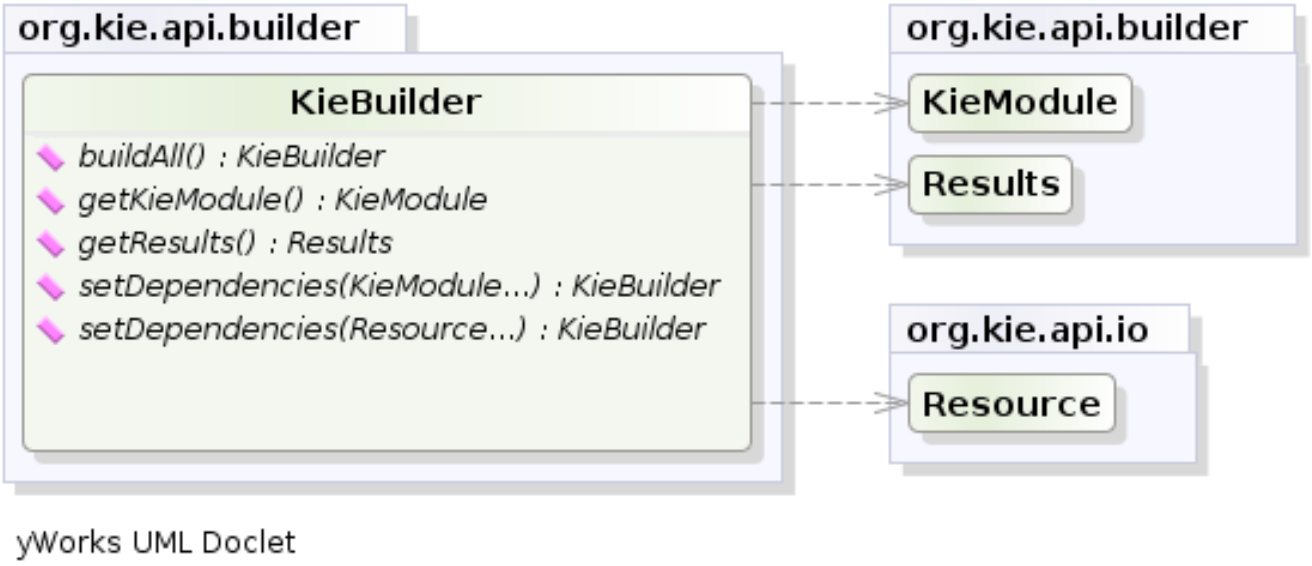


Figure 4.12. KieBuilder

When the contents of a `KieFileSystem` are successfully built, the resulting `KieModule` is automatically added to the `KieRepository`. The `KieRepository` is a singleton acting as a repository for all the available `KieModule`s.

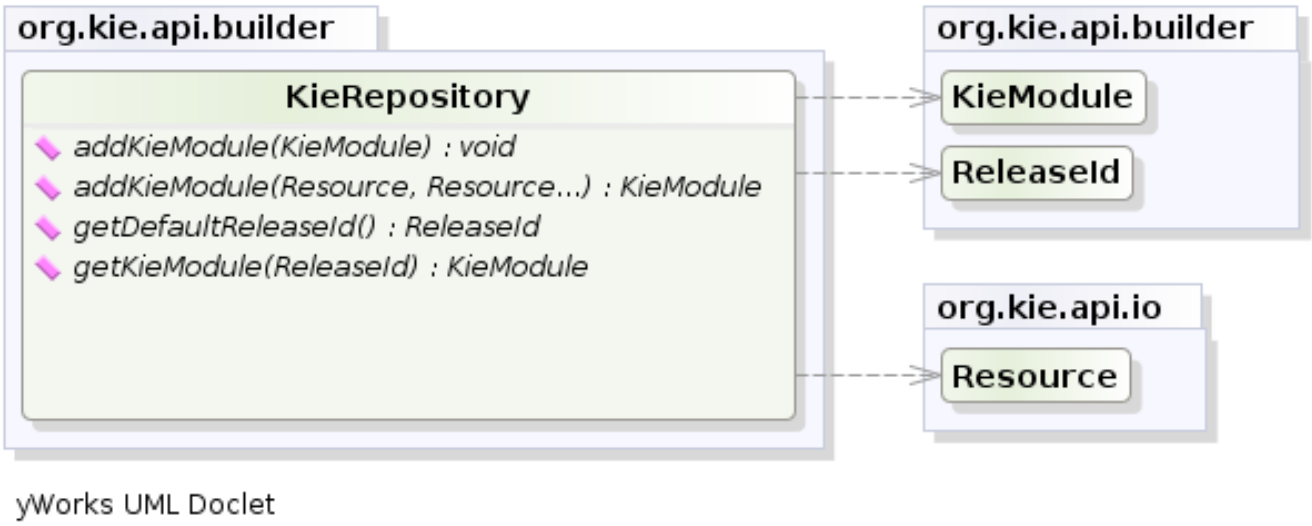


Figure 4.13. KieRepository

After this it is possible to create through the `KieServices` a new `KieContainer` for that `KieModule` using its `ReleaseId`. However, since in this case the `KieFileSystem` doesn't contain any pom.xml file (it is possible to add one using the `KieFileSystem.writePomXML` method), Kie cannot determine the `ReleaseId` of the `KieModule` and assign to it a default one. This default `ReleaseId` can be obtained from the `KieRepository` and used to identify the `KieModule` inside the `KieRepository` itself. The following example shows this whole process.

Example 4.11. Building the contents of a KieFileSystem and creating a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieFileSystem kfs = ...
kieServices.newKieBuilder( kfs ).buildAll();
KieContainer kieContainer = kieServices.newKieContainer(kieServices.getRepository().getDefaultReleaseId());
```

At this point it is possible to get `KieBase`s and create new `KieSession`s from this `KieContainer` exactly in the same way as in the case of a `KieContainer` created directly from the classpath.

It is a best practice to check the compilation results. The `KieBuilder` reports compilation results of 3 different severities: ERROR, WARNING and INFO. An ERROR indicates that the compilation of the project failed and in the case no `KieModule` is produced and nothing is added to the `KieRepository`. WARNING and INFO results can be ignored, but are available for inspection.

Example 4.12. Checking that a compilation didn't produce any error

```
KieBuilder kieBuilder = kieServices.newKieBuilder( kfs ).buildAll();
assertEquals( 0, kieBuilder.getResults().getMessages( Message.Level.ERROR ).size() );
```

4.2.2.5. Changing the Default Build Result Severity

In some cases, it is possible to change the default severity of a type of build result. For instance, when a new rule with the same name of an existing rule is added to a package, the default behavior is to replace the old rule by the new rule and report it as an INFO. This is probably ideal for most use cases, but in some deployments the user might want to prevent the rule update and report it as an error.

Changing the default severity for a result type, configured like any other option in Drools, can be done by API calls, system properties or configuration files. As of this version, Drools supports configurable result severity for rule updates and function updates. To configure it using system properties or configuration files, the user has to use the following properties:

Example 4.13. Setting the severity using properties

```
// sets the severity of rule updatesdrools.kbuilder.severity.duplicateRule = <INFO|WARNING|
ERROR>// sets the severity of function updatesdrools.kbuilder.severity.duplicateFunction = <INFO|
WARNING|ERROR>
 updatesdrools.kbuilder.severity.duplicateRule =
<INFO|WARNING|ERROR>// sets the severity of
function updatesdrools.kbuilder.severity.duplicateFunction =
```

4.2.3. Deploying

4.2.3.1. KieBase

The `KieBase` is a repository of all the application's knowledge definitions. It will contain rules, processes, functions, and type models. The `KieBase` itself does not contain data; instead, sessions are created from the `KieBase` into which data can be inserted and from which process instances may be started. The `KieBase` can be obtained from the `KieContainer` containing the `KieModule` where the `KieBase` has been defined.

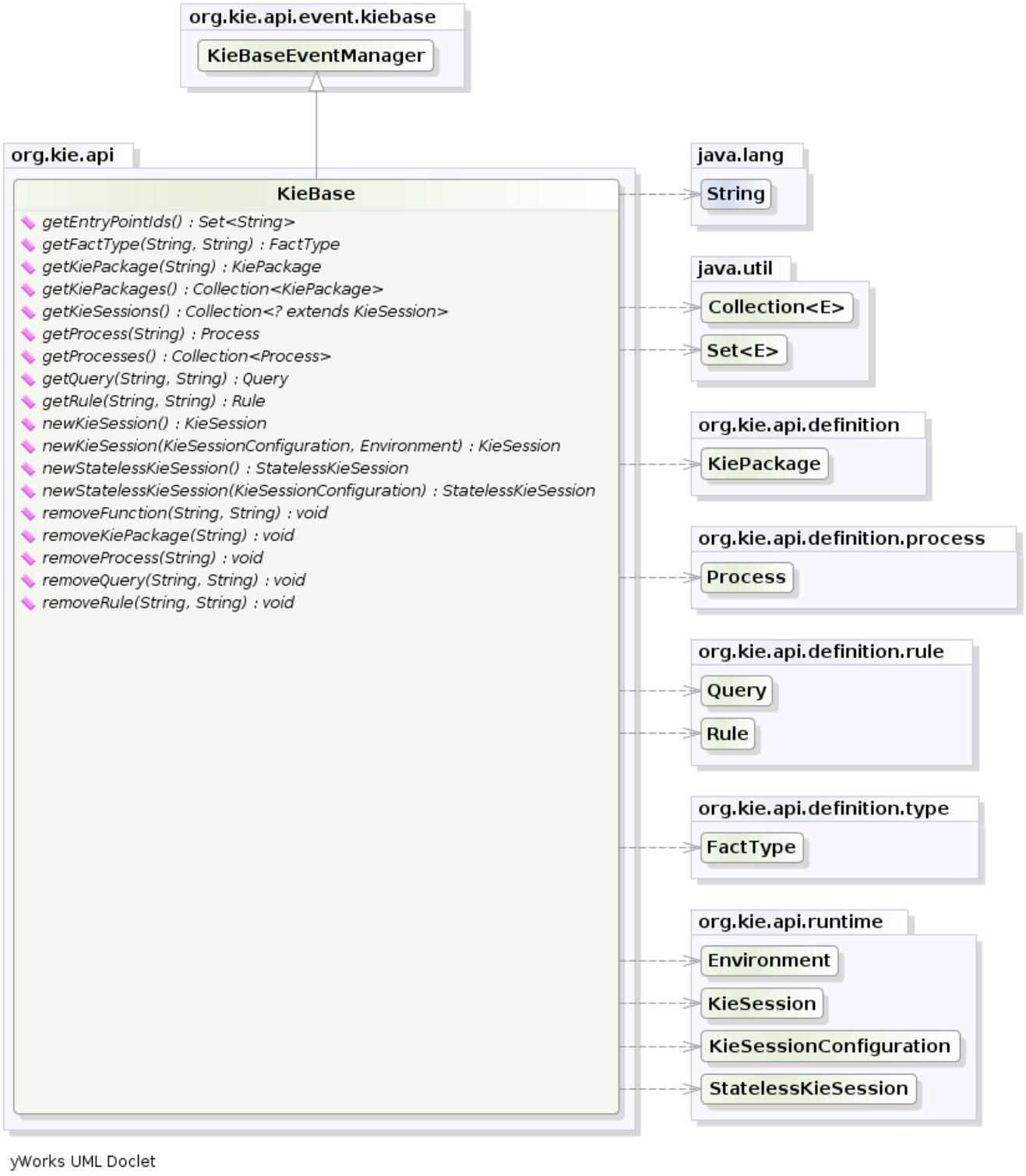


Figure 4.14. KieBase

Sometimes, for instance in a OSGi environment, the `KieBase` needs to resolve types that are not in the default class loader. In this case it will be necessary to create a `KieBaseConfiguration` with an additional class loader and pass it to `KieContainer` when creating a new `KieBase` from it.

Example 4.14. Creating a new KieBase with a custom ClassLoader

```
KieServices kieServices = KieServices.Factory.get();
KieBaseConfiguration kbaseConf = kieServices.newKieBaseConfiguration( null, MyType.class.getClassLoader() );
KieBase kbase = kieContainer.newKieBase( kbaseConf );
```

4.2.3.2. KieSessions and KieBase Modifications

KieSessions will be discussed in more detail in section "Running". The `KieBase` creates and returns `KieSession` objects, and it may optionally keep references to those. When `KieBase` modifications occur those modifications are applied against the data in the sessions. This reference is a weak reference and it is also optional, which is controlled by a boolean flag.

4.2.3.3. KieScanner

The `KieScanner` allows continuous monitoring of your Maven repository to check whether a new release of a Kie project has been installed. A new release is deployed in the `KieContainer` wrapping that project. The use of the `KieScanner` requires kie-ci.jar to be on the classpath.

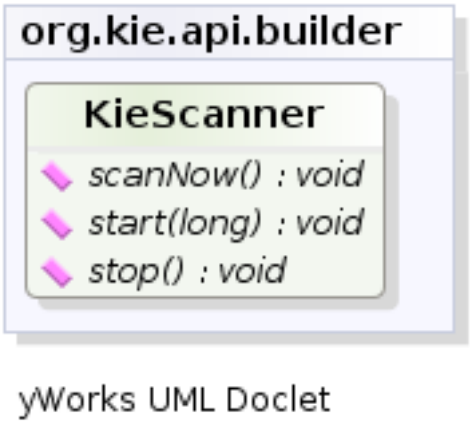


Figure 4.15. KieScanner

A `KieScanner` can be registered on a `KieContainer` as in the following example.

Example 4.15. Registering and starting a KieScanner on a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "myartifact", "1.0-SNAPSHOT" );
KieContainer kContainer = kieServices.newKieContainer( releaseId );
KieScanner kScanner = kieServices.newKieScanner( kContainer );

// Start the KieScanner polling the Maven repository every 10 seconds
kScanner.start( 10000L );
```

In this example the `KieScanner` is configured to run with a fixed time interval, but it is also possible to run it on demand by invoking the `scanNow()` method on it. If the `KieScanner` finds in the Maven repository an updated version of the Kie project used by that `KieContainer` it automatically downloads the new version and triggers an incremental build of the new project. From this moment all the new `KieBase`s and `KieSession`s created from that `KieContainer` will use the new project version.

The `KieScanner` will only pickup changes to deployed jars if it is using a SNAPSHOT, version range, the LATEST, or the RELEASE setting. Fixed versions will not automatically update at runtime.

4.2.3.4. Maven Versions and Dependencies

Maven supports a number of mechanisms to manage versioning and dependencies within applications. Modules can be published with specific version numbers, or they can use the SNAPSHOT suffix. Dependencies can specify version ranges to consume, or take avantage of SNAPSHOT mechanism.

StackOverflow provides a very good description for this, which is reproduced below.

http://stackoverflow.com/questions/30571/how-do-i-tell-maven-to-use-the-latest-version-of-a-dependency [http://stackoverflow.com/questions/30571/how-do-i-tell-maven-to-use-the-latest-version-of-a-dependency]

If you always want to use the newest version, Maven has two keywords you can use as an alternative to version ranges. You should use these options with care as you are no longer in control of the plugins/dependencies you are using.

When you depend on a plugin or a dependency, you can use the a version value of LATEST or RELEASE. LATEST refers to the latest released or snapshot version of a particular artifact, the most recently deployed artifact in a particular repository. RELEASE refers to the last non-snapshot release in the repository. In general, it is not a best practice to design software which depends on a non-specific version of an artifact. If you are developing software, you might want to use RELEASE or LATEST as a convenience so that you don't have to update version numbers when a new release of a third-party library is released. When you release software, you should always make sure that your project depends on specific versions to reduce the chances of your build or your project being affected by a software release not under your control. Use LATEST and RELEASE with caution, if at all.

See the POM Syntax section of the Maven book for more details.

http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom-syntax.html [http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-pom-syntax.html]

http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-project-dependencies.html

Here's an example illustrating the various options. In the Maven repository, com.foo:my-foo has the following metadata:

```
<metadata>
  <groupId>com.foo</groupId>
  <artifactId>my-foo</artifactId>
  <version>2.0.0</version>
  <versioning>
    <release>1.1.1</release>
    <versions>
      <version>1.0</version>
      <version>1.0.1</version>
      <version>1.1</version>
      <version>1.1.1</version>
      <version>2.0.0</version>
    </versions>
    <lastUpdated>20090722140000</lastUpdated>
  </versioning>
</metadata>
```

If a dependency on that artifact is required, you have the following options (other version ranges can be specified of course, just showing the relevant ones here): Declare an exact version (will always resolve to 1.0.1):

```
<version>[1.0.1]</version>
```

Declare an explicit version (will always resolve to 1.0.1 unless a collision occurs, when Maven will select a matching version):

```
<version>1.0.1</version>
```

Declare a version range for all 1.x (will currently resolve to 1.1.1):

```
<version>[1.0.0,2.0.0)</version>
```

Declare an open-ended version range (will resolve to 2.0.0):

```
<version>[1.0.0,)</version>
```

Declare the version as LATEST (will resolve to 2.0.0):

```
<version>LATEST</version>
```

Declare the version as RELEASE (will resolve to 1.1.1):

```
<version>RELEASE</version>
```

Note that by default your own deployments will update the "latest" entry in the Maven metadata, but to update the "release" entry, you need to activate the "release-profile" from the Maven super POM. You can do this with either "-Prelease-profile" or "-DperformRelease=true"

4.2.3.5. Settings.xml and Remote Repository Setup

The maven settings.xml is used to configure Maven execution. Detailed instructions can be found at the Maven website:

http://maven.apache.org/settings.html

The settings.xml file can be located in 3 locations, the actual settings used is a merge of those 3 locations.

- The Maven install: `$M2_HOME/conf/settings.xml`
- A user's install: `${user.home}/.m2/settings.xml`
- Folder location specified by the system property `kie.maven.settings.custom`

The settings.xml is used to specify the location of remote repositories. It is important that you activate the profile that specifies the remote repository, typically this can be done using "activeByDefault":

```
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</profiles>
```

Maven provides detailed documentation on using multiple remote repositories:

http://maven.apache.org/guides/mini/guide-multiple-repositories.html

4.2.4. Running

4.2.4.1. KieBase

The `KieBase` is a repository of all the application's knowledge definitions. It will contain rules, processes, functions, and type models. The `KieBase` itself does not contain data; instead, ses-

sions are created from the `KieBase` into which data can be inserted and from which process instances may be started. The `KieBase` can be obtained from the `KieContainer` containing the `KieModule` where the `KieBase` has been defined.

Example 4.16. Getting a KieBase from a KieContainer

```
KieBase kBase = kContainer.getKieBase();
```

4.2.4.2. KieSession

The `KieSession` stores and executes on the runtime data. It is created from the `KieBase`.

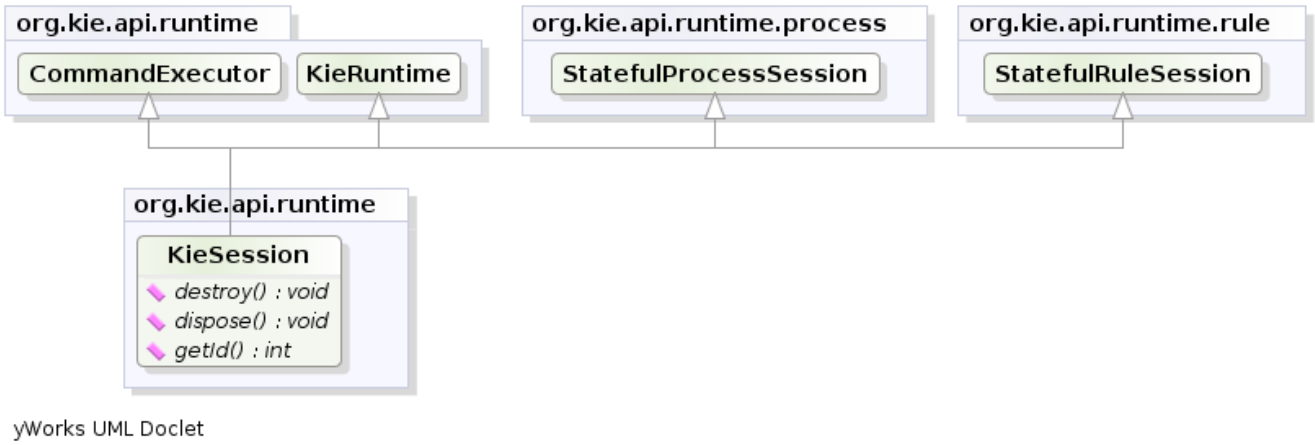


Figure 4.16. KieSession

Example 4.17. Create a KieSession from a KieBase

```
KieSession ksession = kbase.newKieSession();
```

4.2.4.3. KieRuntime

4.2.4.3.1. KieRuntime

The `KieRuntime` provides methods that are applicable to both rules and processes, such as setting globals and registering channels. ("Exit point" is an obsolete synonym for "channel".)

Figure 4.17. KieRuntime

4.2.4.3.1.1. Globals

Globals are named objects that are made visible to the rule engine, but in a way that is fundamentally different from the one for facts: changes in the object backing a global do not trigger reevaluation of rules. Still, globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or, at least, don't expect changes to have any effect on the behavior of your rules.

A global must be declared in a rules file, and then it needs to be backed up with a Java object.

```
global java.util.List list
```

With the Knowledge Base now aware of the global identifier and its type, it is now possible to call `ksession.setGlobal()` with the global's name and an object, for any session, to associate the object with the global. Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

```
List list = new ArrayList();
ksession.setGlobal("list", list);
```

Make sure to set any global before it is used in the evaluation of a rule. Failure to do so results in a `NullPointerException`.

4.2.4.4. Event Model

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows separation of logging and auditing activities from the main part of your application (and the rules).

The `KieRuntimeEventManager` interface is implemented by the `KieRuntime` which provides two interfaces, `RuleRuntimeEventManager` and `ProcessEventManager`. We will only cover the `RuleRuntimeEventManager` here.

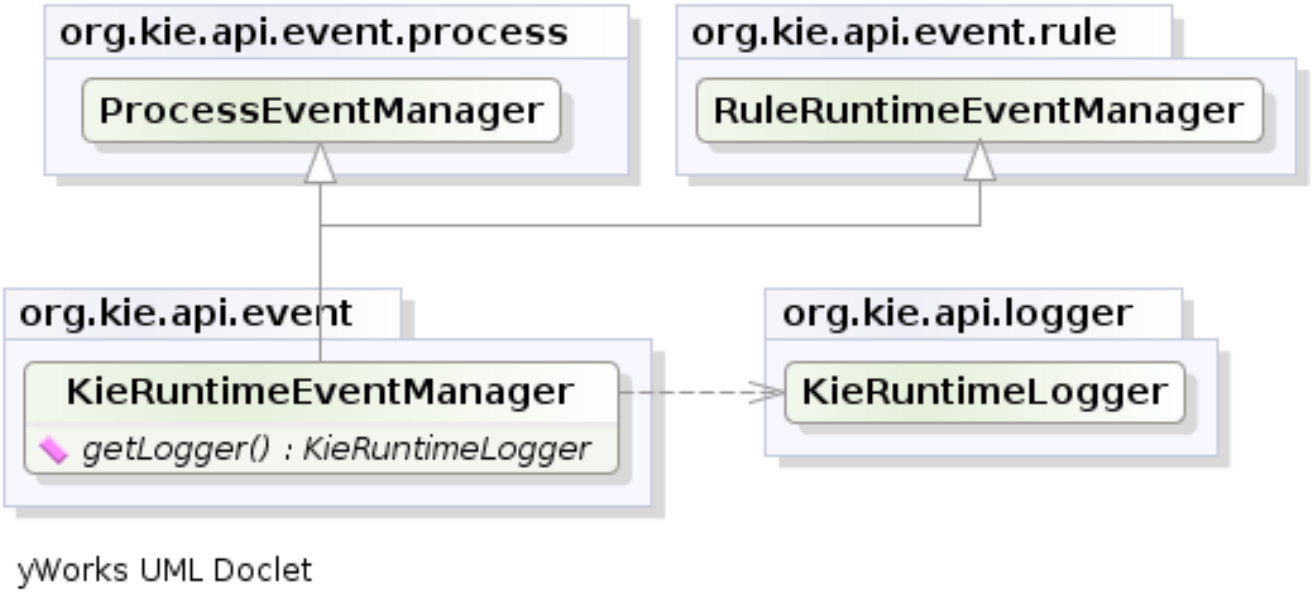


Figure 4.18. KieRuntimeEventManager

The `RuleRuntimeEventManager` allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

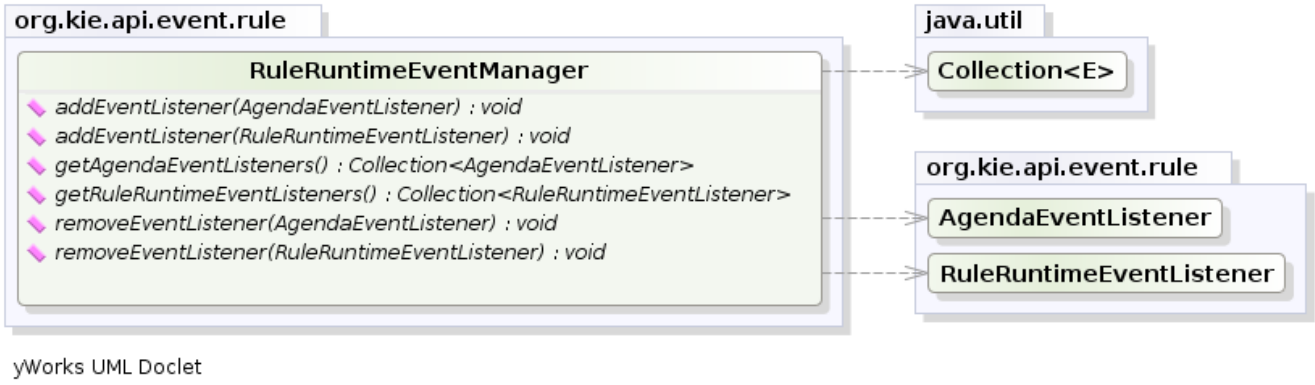


Figure 4.19. RuleRuntimeEventManager

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print matches after they have fired.

Example 4.18. Adding an AgendaEventListener

```
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterMatchFired(AfterMatchFiredEvent event) {
        super.afterMatchFired( event );
        System.out.println( event );
    }
});
```

Drools also provides `DebugRuleRuntimeEventListener` and `DebugAgendaEventListener` which implement each method with a debug print statement. To print all Working Memory events, you add a listener like this:

Example 4.19. Adding a DebugRuleRuntimeEventListener

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

All emitted events implement the `KieRuntimeEvent` interface which can be used to retrieve the actual `KnowlegeRuntime` the event originated from.

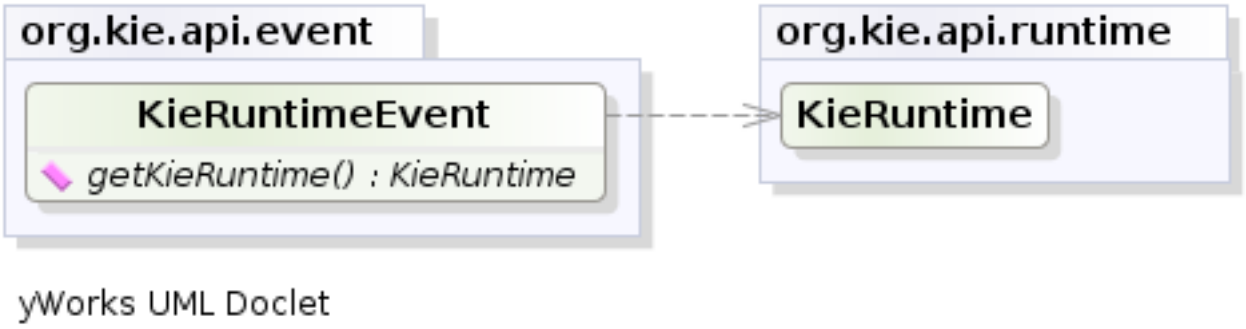


Figure 4.20. KieRuntimeEvent

The events currently supported are:

- MatchCreatedEvent
- MatchCancelledEvent
- BeforeMatchFiredEvent
- AfterMatchFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectDeletedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

4.2.4.5. KieRuntimeLogger

The KieRuntimeLogger uses the comprehensive event system in Drools to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

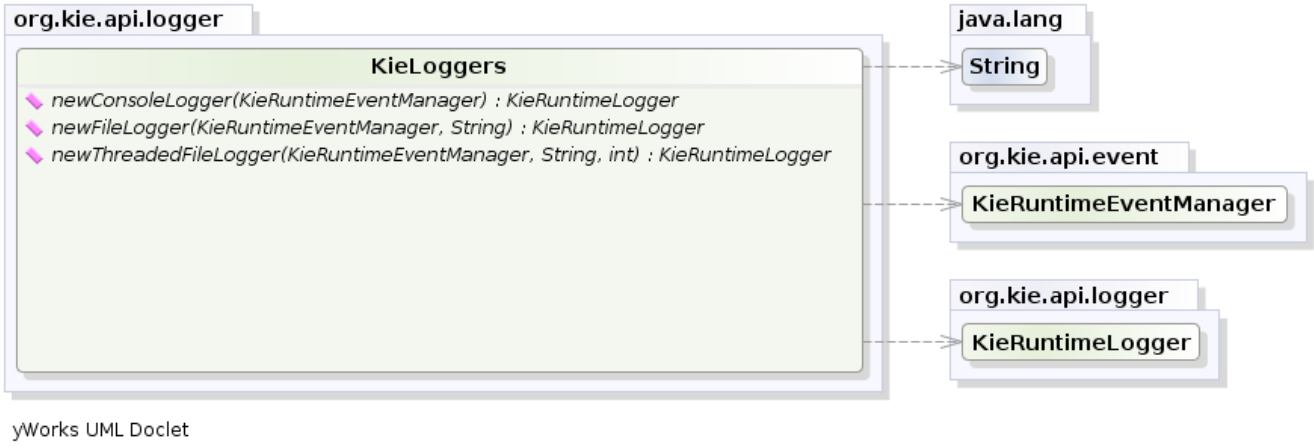


Figure 4.21. KieLoggers

Example 4.20. FileLogger

```
KieRuntimeLogger logger =
  KieServices.Factory.get().getLoggers().newFileLogger(ksession, "logdir/mylogfile");
...
logger.close();
```

4.2.4.6. Commands and the CommandExecutor

KIE has the concept of stateful or stateless sessions. Stateful sessions have already been covered, which use the standard KieRuntime, and can be worked with iteratively over time. Stateless is a one-off execution of a KieRuntime with a provided data set. It may return some results, with the session being disposed at the end, prohibiting further iterative interactions. You can think of stateless as treating an engine like a function call with optional return results.

The foundation for this is the `CommandExecutor` interface, which both the stateful and stateless interfaces extend. This returns an `ExecutionResults`:

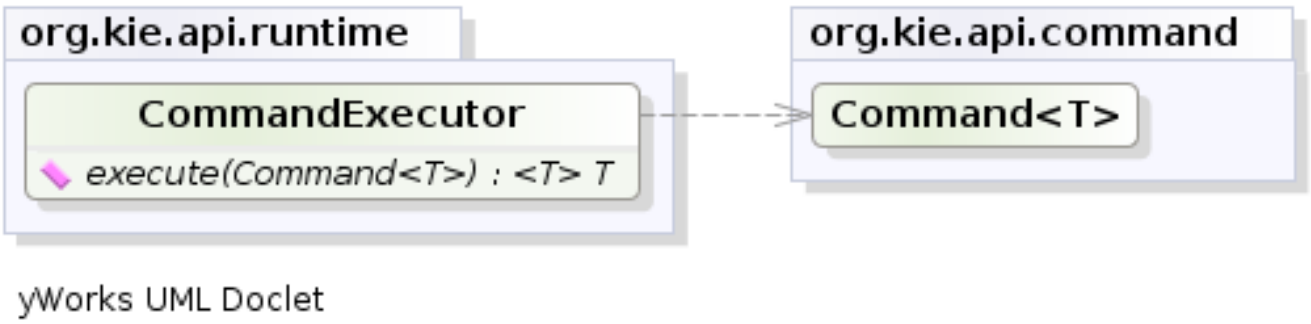


Figure 4.22. CommandExecutor

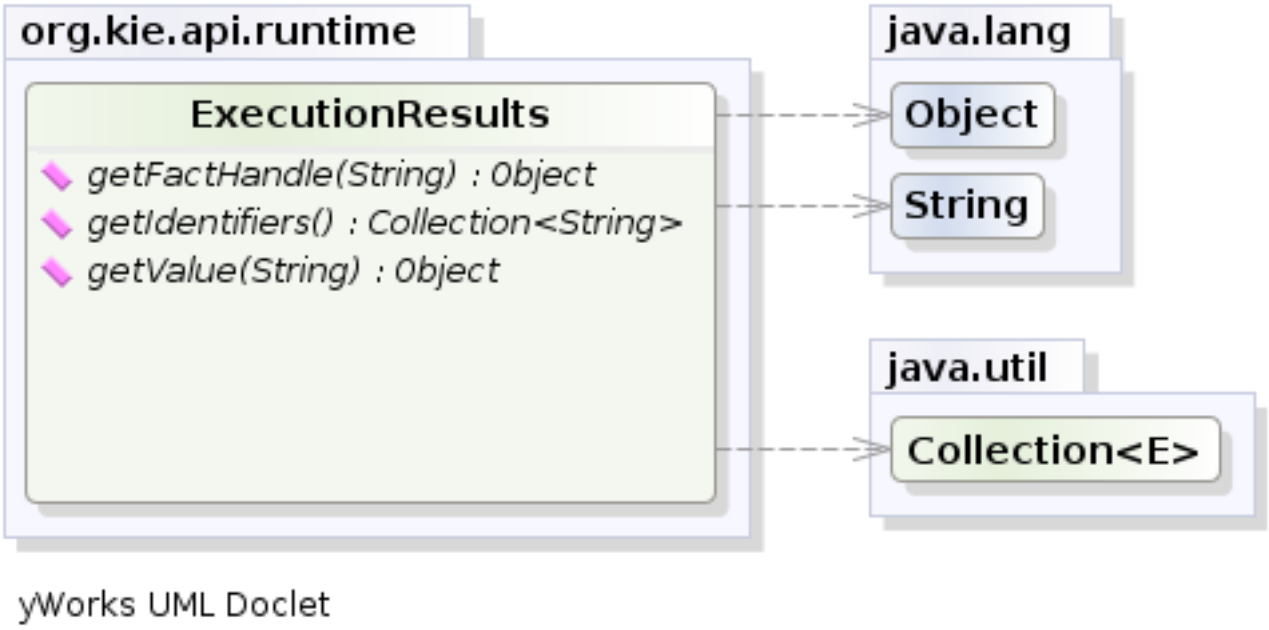


Figure 4.23. ExecutionResults

The `CommandExecutor` allows for commands to be executed on those sessions, the only difference being that the StatelessKieSession executes `fireAllRules()` at the end before disposing the session. The commands can be created using the `CommandExecutor` .The Javadocs provide the full list of the allowed comands using the `CommandExecutor`.

setGlobal and getGlobal are two commands relevant to both Drools and jBPM.

Set Global calls setGlobal underneath. The optional boolean indicates whether the command should return the global's value as part of the `ExecutionResults`. If true it uses the same name as the global name. A String can be used instead of the boolean, if an alternative name is desired.

Example 4.21. Set Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.newSetGlobal( "stilton", new Cheese( "stilton" ), true);
Cheese stilton = bresults.getValue( "stilton" );
```

Allows an existing global to be returned. The second optional String argument allows for an alternative return name.

Example 4.22. Get Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.getGlobal( "stilton" );
```

```
Cheese stilton = bresults.getValue( "stilton" );
```

All the above examples execute single commands. The `BatchExecution` represents a composite command, created from a list of commands. It will iterate over the list and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query, all in a single `execute(...)` call, which is quite powerful.

The StatelessKieSession will execute `fireAllRules()` automatically at the end. However the keen-eyed reader probably has already noticed the `FireAllRules` command and wondered how that works with a StatelessKieSession. The `FireAllRules` command is allowed, and using it will disable the automatic execution at the end; think of using it as a sort of manual override function.

Any command, in the batch, that has an out identifier set will add its results to the returned `ExecutionResults` instance. Let's look at a simple example to see how this works. The example presented includes command from the Drools and jBPM, for the sake of illustration. They are covered in more detail in the Drool and jBPM specific sections.

Example 4.23. BatchExecution Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();

List cmds = new ArrayList();
cmds.add( CommandFactory.newInsertObject( new Cheese( "stilton", 1), "stilton") );
cmds.add( CommandFactory.newStartProcess( "process cheeses" ) );
cmds.add( CommandFactory.newQuery( "cheeses" ) );
ExecutionResults bresults = ksession.execute( CommandFactory.newBatchExecution( cmds ) );
Cheese stilton = ( Cheese ) bresults.getValue( "stilton" );
QueryResults qresults = ( QueryResults ) bresults.getValue( "cheeses" );
```

In the above example multiple commands are executed, two of which populate the `ExecutionResults`. The query command defaults to use the same identifier as the query name, but it can also be mapped to a different identifier.

All commands support XML and jSON marshalling using XStream, as well as JAXB marshalling. This is covered in section Commands API.

4.2.4.7. StatelessKieSession

The `StatelessKieSession` wraps the `KieSession`, instead of extending it. Its main focus is on the decision service type scenarios. It avoids the need to call `dispose()`. Stateless sessions do not support iterative insertions and the method call `fireAllRules()` from Java code; the act of calling `execute()` is a single-shot method that will internally instantiate a `KieSession`, add all the user data and execute user commands, call `fireAllRules()`, and then call `dispose()`. While the main way to work with this class is via the `BatchExecution` (a subinterface of `Command`) as supported by the `CommandExecutor` interface, two convenience methods are provided for when simple object insertion is all that's required. The `CommandExecutor` and `BatchExecution` are talked about in detail in their own section.

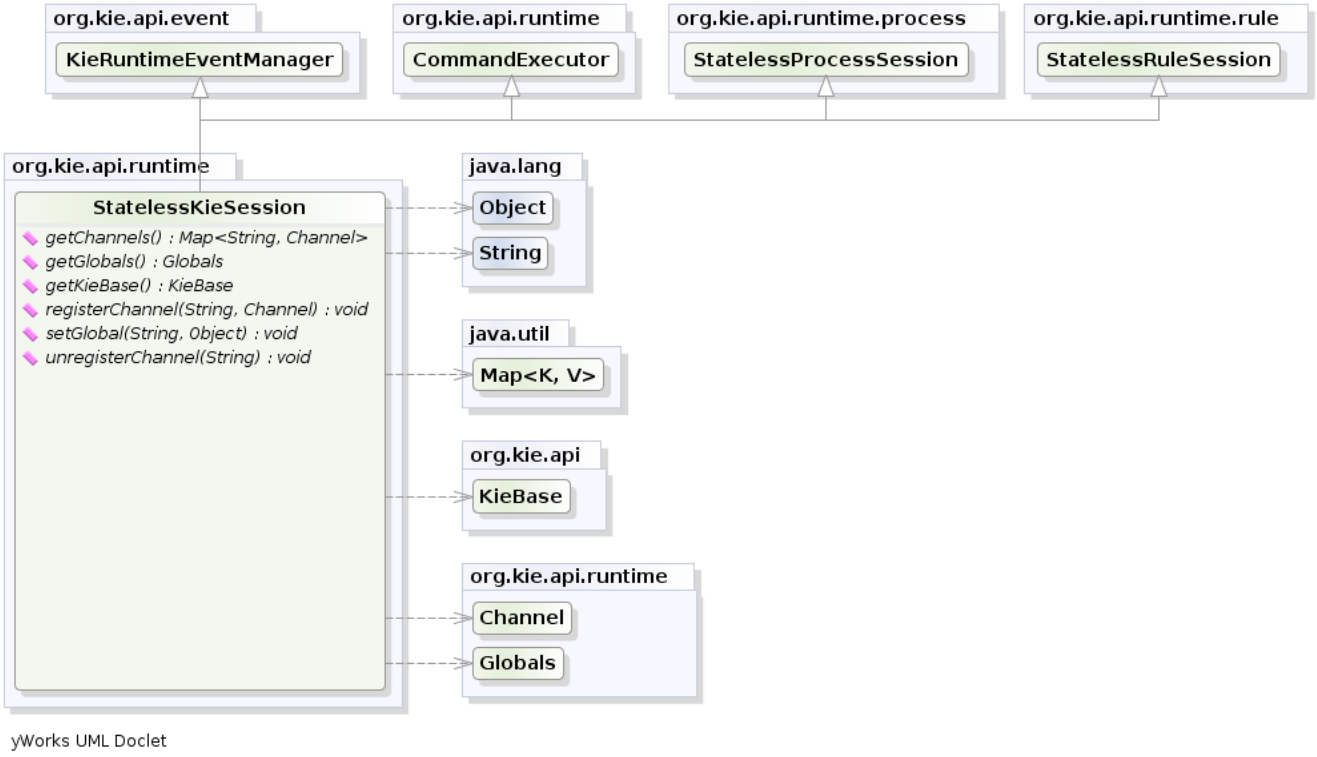


Figure 4.24. StatelessKieSession

Our simple example shows a stateless session executing a given collection of Java objects using the convenience API. It will iterate the collection, inserting each element in turn.

Example 4.24. Simple StatelessKieSession execution with a Collection

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ksession.execute( collection );
```

If this was done as a single Command it would be as follows:

Example 4.25. Simple StatelessKieSession execution with InsertElements Command

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

If you wanted to insert the collection itself, and the collection's individual elements, then `CommandFactory.newInsert(collection)` would do the job.

Methods of the `CommandFactory` create the supported commands, all of which can be marshalled using XStream and the `BatchExecutionHelper`. `BatchExecutionHelper` provides details on the

XML format as well as how to use Drools Pipeline to automate the marshalling of `BatchExecution` and `ExecutionResults`.

`StatelessKieSession` supports globals, scoped in a number of ways. We cover the non-command way first, as commands are scoped to a specific execution call. Globals can be resolved in three ways.

- The StatelessKieSession method `getGlobals()` returns a Globals instance which provides access to the session's globals. These are shared for *all* execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

Example 4.26. Session scoped global

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
// Set a global hbnSession, that can be used for DB interactions in the rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// Execute while being able to resolve the "hbnSession" identifier.
ksession.execute( collection );
```

- Using a delegate is another way of global resolution. Assigning a value to a global (with `setGlobal(String, Object)`) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. Only if an identifier cannot be found in this internal collection, the delegate global (if any) will be used.
- The third way of resolving globals is to have execution scoped globals. Here, a `Command` to set a global is passed to the `CommandExecutor`.

The `CommandExecutor` interface also offers the ability to export data via "out" parameters. Inserted facts, globals and query results can all be returned.

Example 4.27. Out identifiers

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true ) );
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" ) );
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
```

```
results.getValue( "Get People" );
```

4.2.4.8. Marshalling

The `KieMarshallers` are used to marshal and unmarshal KieSessions.

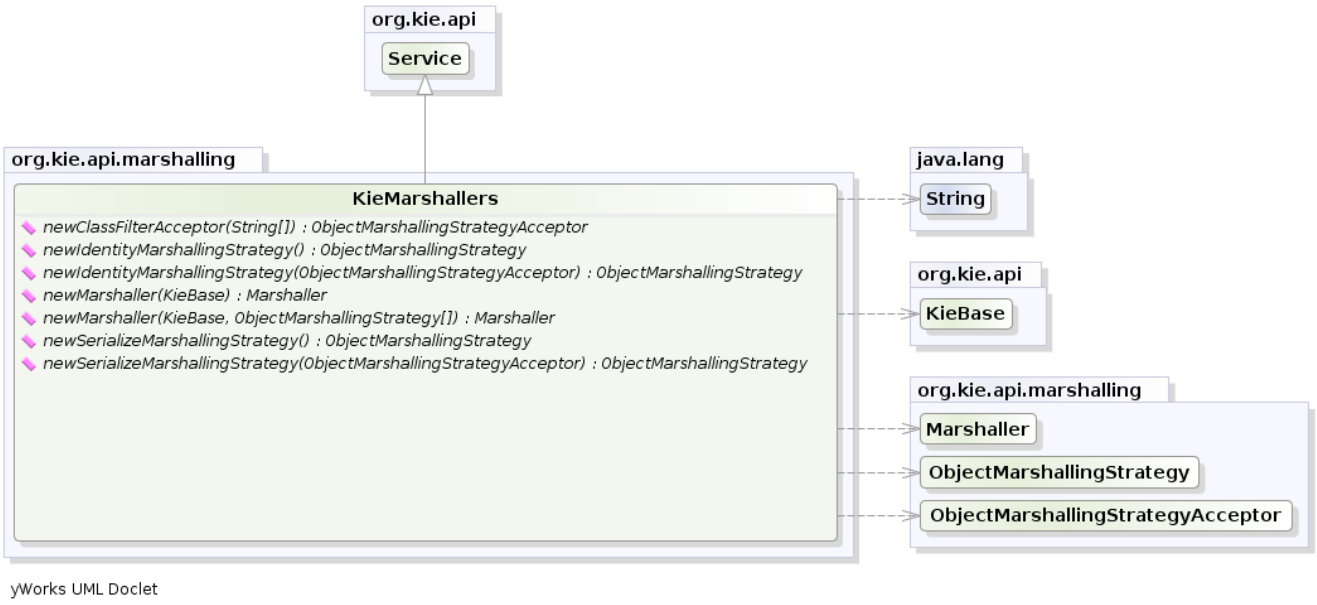


Figure 4.25. KieMarshallers

An instance of the `KieMarshallers` can be retrieved from the `KieServices`. A simple example is shown below:

Example 4.28. Simple Marshaller Example

```
// ksession is the KieSession
// kbase is the KieBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = KieServices.Factory.get().getMarshallers().newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

However, with marshalling, you will need more flexibility when dealing with referenced user data. To achieve this use the `ObjectMarshallingStrategy` interface. Two implementations are provided, but users can implement their own. The two supplied strategies are `IdentityMarshallingStrategy` and `SerializeMarshallingStrategy`. `SerializeMarshallingStrategy` is the default, as shown in the example above, and it just calls the `Serializable` or `Externalizable` methods on a user instance. `IdentityMarshallingStrategy` creates an integer id for each user object and stores them in a Map, while the id is written to the stream. When unmarshalling it accesses the `IdentityMarshallingStrategy` map to retrieve the instance. This means that if you use the `IdentityMarshallingStrategy`, it is stateful for the life of the Marshaller instance and will create

ids and keep references to all objects that it attempts to marshal. Below is the code to use an Identity Marshalling Strategy.

Example 4.29. IdentityMarshallingStrategy

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategy oms = kMarshallers.newIdentityMarshallingStrategy()
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase, new ObjectMarshallingStrategy[]{ oms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Im most cases, a single strategy is insufficient. For added flexibility, the `ObjectMarshallingStrategyAcceptor` interface can be used. This Marshaller has a chain of strategies, and while reading or writing a user object it iterates the strategies asking if they accept responsibility for marshalling the user object. One of the provided implementations is `ClassFilterAcceptor`. This allows strings and wild cards to be used to match class names. The default is "*.*", so in the above example the Identity Marshalling Strategy is used which has a default "*.*" acceptor.

Assuming that we want to serialize all classes except for one given package, where we will use identity lookup, we could do the following:

Example 4.30. IdentityMarshallingStrategy with Acceptor

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategyAcceptor identityAcceptor =
        kMarshallers.newClassFilterAcceptor( new String[] { "org.domain.pkg1.*" } );
ObjectMarshallingStrategy identityStrategy =
        kMarshallers.newIdentityMarshallingStrategy( identityAcceptor );
ObjectMarshallingStrategy sms = kMarshallers.newSerializeMarshallingStrategy();
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase,
                                    new ObjectMarshallingStrategy[]{ identityStrategy, sms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Note that the acceptance checking order is in the natural order of the supplied elements.

Also note that if you are using scheduled matches (i.e. some of your rules use timers or calendars) they are marshallable only if, before you use it, you configure your KieSession to use a trackable timer job factory manager as follows:

Example 4.31. Configuring a trackable timer job factory manager

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption(TimerJobFactoryOption.get("trackable"));
```

```
KSession ksession = kbase.newKieSession(ksconf, null);
```

4.2.4.9. Persistence and Transactions

Longterm out of the box persistence with Java Persistence API (JPA) is possible with Drools. It is necessary to have some implementation of the Java Transaction API (JTA) installed. For development purposes the Bitronix Transaction Manager is suggested, as it's simple to set up and works embedded, but for production use JBoss Transactions is recommended.

Example 4.32. Simple example using transactions

```
KieServices kieServices = KieServices.Factory.get();
Environment env = kieServices.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
         Persistence.createEntityManagerFactory( "emf-name" ) );
env.set( EnvironmentName.TRANSACTION_MANAGER,
         TransactionManagerServices.getTransactionManager() );

// KieSessionConfiguration may be null, and a default will be used
KieSession ksession =
        kieServices.getStoreServices().newKieSession( kbase, null, env );
int sessionId = ksession.getId();

UserTransaction ut =
  (UserTransaction) new InitialContext().lookup( "java:comp/UserTransaction" );
ut.begin();
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```

To use a JPA, the Environment must be set with both the `EntityManagerFactory` and the `TransactionManager`. If rollback occurs the ksession state is also rolled back, hence it is possible to continue to use it after a rollback. To load a previously persisted KieSession you'll need the id, as shown below:

Example 4.33. Loading a KieSession

```
KieSession ksession =
        kieServices.getStoreServices().loadKieSession( sessionId, kbase, null, env );
```

To enable persistence several classes must be added to your persistence.xml, as in the example below:

Example 4.34. Configuring JPA

```
<persistence-unit name="org.drools.persistence.jpa" transaction-type="JTA">
```

```
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/BitronixJTADataSource</jta-data-source>
    <class>org.drools.persistence.info.SessionInfo</class>
    <class>org.drools.persistence.info.WorkItemInfo</class>
    <properties>
          <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
          <property name="hibernate.max_fetch_depth" value="3"/>
          <property name="hibernate.hbm2ddl.auto" value="update" />
          <property name="hibernate.show_sql" value="true" />
          <property name="hibernate.transaction.manager_lookup_class"
                     value="org.hibernate.transaction.BTMTransactionManagerLookup" />
    </properties>
</persistence-unit>
```

The jdbc JTA data source would have to be configured first. Bitronix provides a number of ways of doing this, and its documentation should be consulted for details. For a quick start, here is the programmatic approach:

Example 4.35. Configuring JTA DataSource

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADataSource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions( true );
ds.getDriverProperties().put( "user", "sa" );
ds.getDriverProperties().put( "password", "sasa" );
ds.getDriverProperties().put( "URL", "jdbc:h2:mem:mydb" );
ds.init();
```

Bitronix also provides a simple embedded JNDI service, ideal for testing. To use it, add a jndi.properties file to your META-INF folder and add the following line to it:

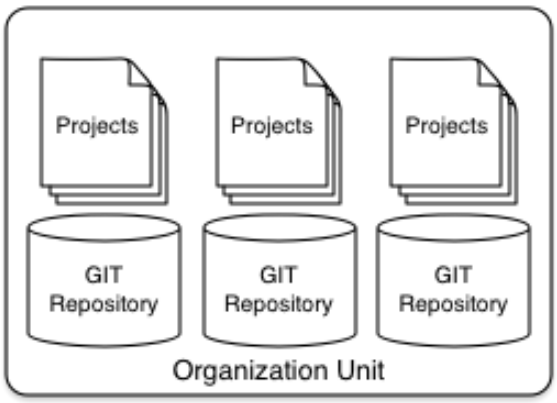
Example 4.36. JNDI properties

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

4.2.5. Installation and Deployment Cheat Sheets

Content Structuring

There are 3 layers to structure your content.



Organizational Unit

This is the top level. An installation may have one or more organisational units.

Repository

Each Unit can have one or more repositories.
A repository is a physical git repository, stored on disk.

Project

Each project can have one or more projects.
A project forms the deployable unit and compiles down to a jar. A project can depend on one or more other project.

Useful GIT System Properties

org.uberfire.nio.git.dir: Location of the directory .niogit. Default: working directory
org.uberfire.nio.git.daemon.enabled: Enables/disables git daemon. Default: true
org.uberfire.nio.git.daemon.host: If daemon enabled, uses this property as local host identifier. Default: localhost
org.uberfire.nio.git.daemon.port: If daemon enabled, uses this property as port number. Default 9418
org.uberfire.nio.git.daemon.upload: If daemon enabled, uses this information to define if it's possible to push (upload) data to git. Default: true
org.uberfire.metadata.index.dir: Place where lucene .index folder will be stored. Default: working directory

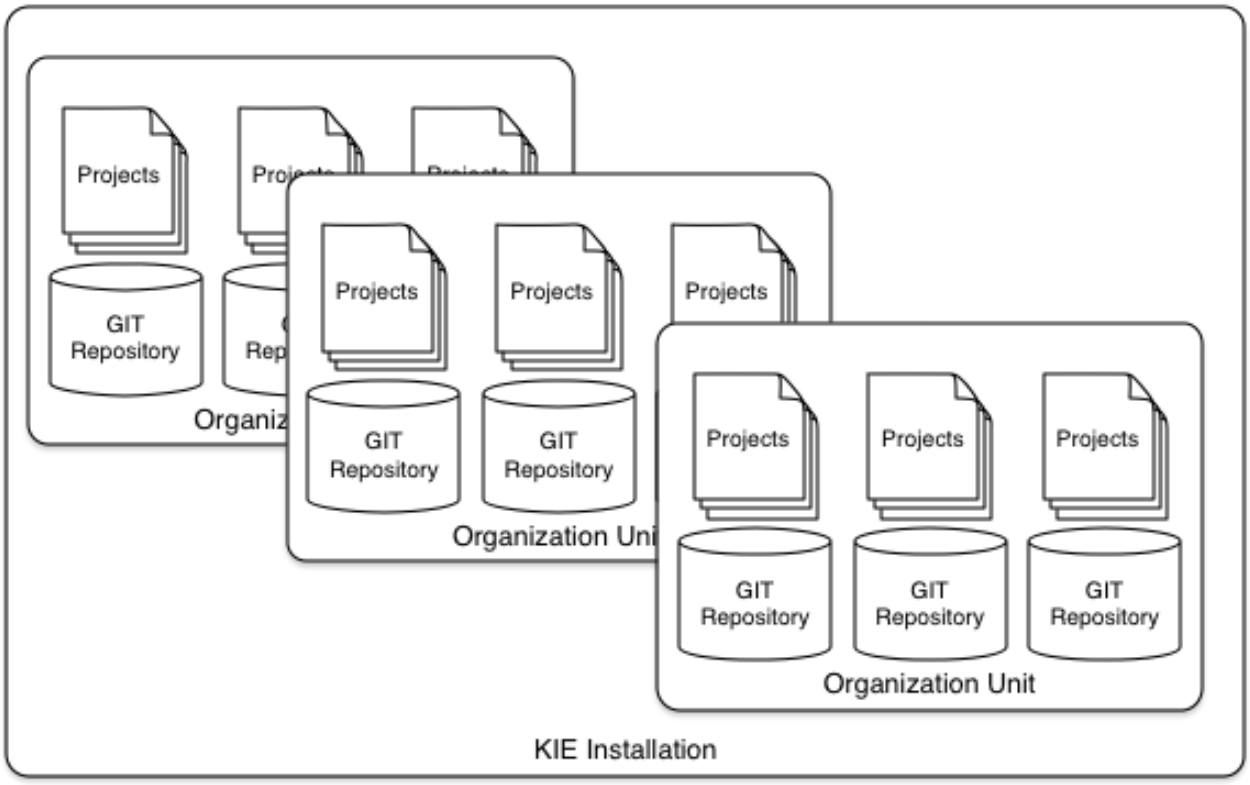
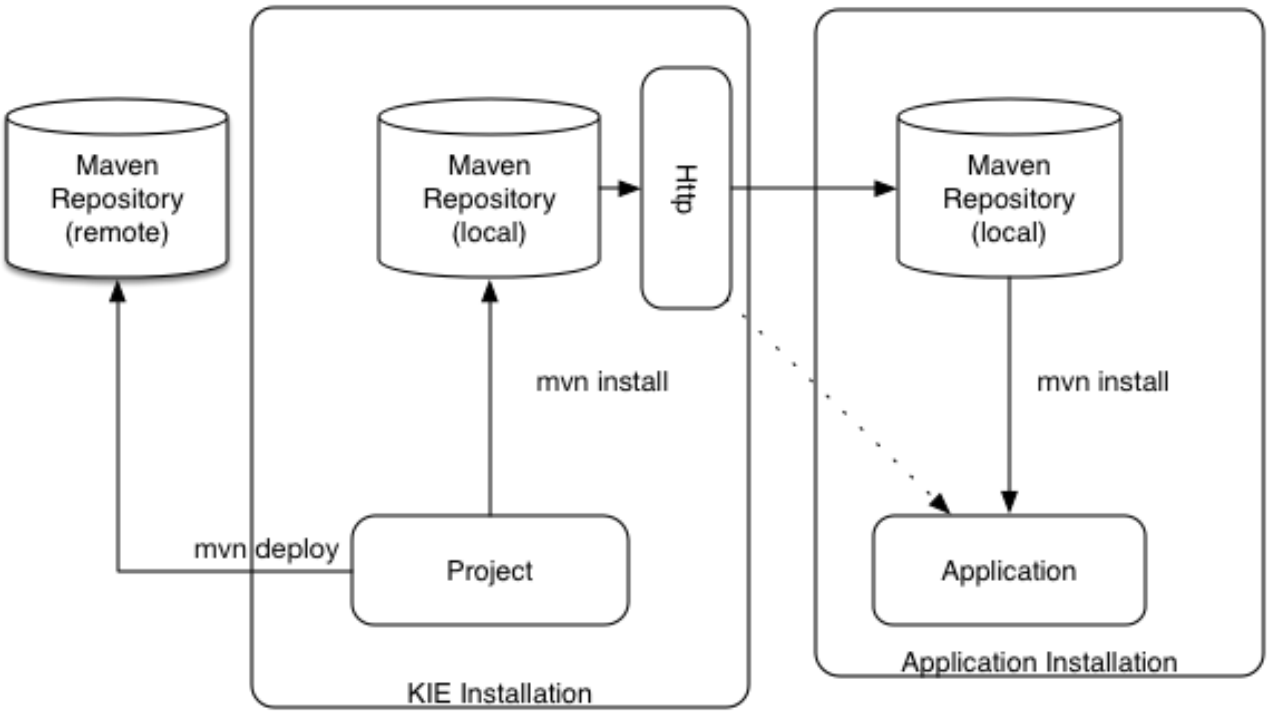


Figure 4.26. Installation Overview



Maven Repository - Server Side
Built projects are installed into the local maven repository.
Default location: <working-directory>/repositories/kie
System property: org.guvnor.m2repo.dir

The repository is exposed via httpd for applications to access.
URL: http://localhost:8080/<app context>/maven2/
Example: http://localhost:8080/kie-drools-wb-6.0.0-SNAPSHOT-jboss-as7.0/
maven2/org/mydomain/proj1/1.0.0/proj1-1.0.0.jar

Maven Repository Location Configuration - Application Side
Applications may specify the remote repositories either in the applications pom.xml or via Maven settings.xml.

There are three locations where a settings.xml file may live:
The Maven install: $M2_HOME/conf/settings.xml
A user's install: ${user.home}/.m2/settings.xml
System Property for file location: kie.maven.settings.custom

Figure 4.27. Deployment Overview

4.2.6. Build, Deploy and Utilize Examples

The best way to learn the new build system is by example. The source project "drools-examples-api" contains a number of examples, and can be found at GitHub:

https://github.com/droolsjbpm/drools/tree/6.0.x/drools-examples-api

Each example is described below, the order starts with the simplest (most of the options are defaulted) and working its way up to more complex use cases.

The Deploy use cases shown below all involve `mvn install`. Remote deployment of JARs in Maven is well covered in Maven literature. Utilize refers to the initial act of loading the resources and providing access to the KIE runtimes. Where as Run refers to the act of interacting with those runtimes.

4.2.6.1. Default KieSession

- Project: default-kesession.
- Summary: Empty kmodule.xml KieModule on the classpath that includes all resources in a single default KieBase. The example shows the retrieval of the default KieSession from the classpath.

An empty kmodule.xml will produce a single KieBase that includes all files found under resources path, be it DRL, BPMN2, XLS etc. That single KieBase is the default and also includes a single default KieSession. Default means they can be created without knowing their names.

Example 4.37. Author - kmodule.xml

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"> </kmodule>
```

Example 4.38. Build and Install - Maven

```
mvn install
```

ks.getKieClasspathContainer() returns the KieContainer that contains the KieBases deployed onto the environment classpath. kContainer.newKieSession() creates the default KieSession. Notice that you no longer need to look up the KieBase, in order to create the KieSession. The KieSession knows which KieBase it's associated with, and use that, which in this case is the default KieBase.

Example 4.39. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();

KieSession kSession = kContainer.newKieSession();
```

```
kSession.setGlobal("out", out);
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

4.2.6.2. Named KieSession

- Project: named-kiesession.
- Summary: kmodule.xml that has one named KieBase and one named KieSession. The examples shows the retrieval of the named KieSession from the classpath.

kmodule.xml will produce a single named KieBase, 'kbase1' that includes all files found under resources path, be it DRL, BPMN2, XLS etc. KieSession 'ksession1' is associated with that KieBase and can be created by name.

Example 4.40. Author - kmodule.xml

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
    <kbase name="kbase1">
        <ksession name="ksession1"/>
    </kbase>
</kmodule>
```

Example 4.41. Build and Install - Maven

```
mvn install
```

ks.getKieClasspathContainer() returns the KieContainer that contains the KieBases deployed onto the environment classpath. This time the KieSession uses the name 'ksession1'. You do not need to lookup the KieBase first, as it knows which KieBase 'ksession1' is assocaited with.

Example 4.42. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.setGlobal("out", out);
kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

4.2.6.3. KieBase Inheritence

- Project: kiebase-inclusion.
- Summary: 'kmodule.xml' demonstrates that one KieBase can include the resources from another KieBase, from another KieModule. In this case it inherits the named KieBase from the 'name-kiesession' example. The included KieBase can be from the current KieModule or any other KieModule that is in the pom.xml dependency list.

kmodule.xml will produce a single named KieBase, 'kbase2' that includes all files found under resources path, be it DRL, BPMN2, XLS etc. Further it will include all the resources found from the KieBase 'kbase1', due to the use of the 'includes' attribute. KieSession 'ksession2' is associated with that KieBase and can be created by name.

Example 4.43. Author - kmodule.xml

```
<kbase name="kbase2" includes="kbase1">
    <ksession name="ksession2"/>
</kbase>
```

This example requires that the previous example, 'named-kiesession', is built and installed to the local Maven repository first. Once installed it can be included as a dependency, using the standard Maven <dependencies> element.

Example 4.44. Author - pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.drools</groupId>
    <artifactId>drools-examples-api</artifactId>
    <version>6.0.0/version>
  </parent>

  <artifactId>kiebase-inclusion</artifactId>
  <name>Drools API examples - KieBase Inclusion</name>

  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-compiler</artifactId>
    </dependency>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>named-kiesession</artifactId>
      <version>6.0.0</version>
    </dependency>
```

```
  </dependencies>

</project>
```

Once 'named-kiesession' is built and installed this example can be built and installed as normal. Again the act of installing, will force the unit tests to run, demonstrating the use case.

Example 4.45. Build and Install - Maven

```
mvn install
```

ks.getKieClasspathContainer() returns the KieContainer that contains the KieBases deployed onto the environment classpath. This time the KieSession uses the name 'ksession2'. You do not need to lookup the KieBase first, as it knows which KieBase 'ksession1' is assocaited with. Notice two rules fire this time, showing that KieBase 'kbase2' has included the resources from the dependency KieBase 'kbase1'.

Example 4.46. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieSession kSession = kContainer.newKieSession("ksession2");
kSession.setGlobal("out", out);

kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();

kSession.insert(new Message("Dave", "Open the pod bay doors, HAL."));
kSession.fireAllRules();
```

4.2.6.4. Multiple KieBases

- Project: 'multiple-kbases.
- Summary: Demonstrates that the 'kmodule.xml' can contain any number of KieBase or KieSession declarations. Introduces the 'packages' attribute to select the folders for the resources to be included in the KieBase.

kmodule.xml produces 6 different named KieBases. 'kbase1' includes all resources from the KieModule. The other KieBases include resources from other selected folders, via the 'packages' attribute. Note the use of wildcard '*', to select this package and all packages below it.

Example 4.47. Author - kmodule.xml

```
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
```

```
  <kbase name="kbase1">
    <ksession name="ksession1"/>
  </kbase>

  <kbase name="kbase2" packages="org.some.pkg">
    <ksession name="ksession2"/>
  </kbase>

  <kbase name="kbase3" includes="kbase2" packages="org.some.pkg2">
    <ksession name="ksession3"/>
  </kbase>

  <kbase name="kbase4" packages="org.some.pkg, org.other.pkg">
    <ksession name="ksession4"/>
  </kbase>

  <kbase name="kbase5" packages="org.*">
    <ksession name="ksession5"/>
  </kbase>

  <kbase name="kbase6" packages="org.some.*">
    <ksession name="ksession6"/>
  </kbase>
</kmodule>
```

Example 4.48. Build and Install - Maven

```
mvn install
```

Only part of the example is included below, as there is a test method per KieSession, but each one is a repetition of the other, with different list expectations.

Example 4.49. Utilize and Run - Java

```
@Test
public void testSimpleKieBase() {
    List<Integer> list = useKieSession("ksession1");
    // no packages imported means import everything
    assertEquals(4, list.size());
    assertTrue( list.containsAll( asList(0, 1, 2, 3) ) );
}

//.. other tests for ksession2 to ksession6 here

private List<Integer> useKieSession(String name) {
    KieServices ks = KieServices.Factory.get();
    KieContainer kContainer = ks.getKieClasspathContainer();
    KieSession kSession = kContainer.newKieSession(name);

    List<Integer> list = new ArrayList<Integer>();
    kSession.setGlobal("list", list);
    kSession.insert(1);
```

```
    kSession.fireAllRules();

    return list;
}
```

4.2.6.5. KieContainer from KieRepository

- Project: kcontainer-from-repository
- Summary: The project does not contain a kmodule.xml, nor does the pom.xml have any dependencies for other KieModules. Instead the Java code demonstrates the loading of a dynamic KieModule from a Maven repository.

The pom.xml must include kie-ci as a depdency, to ensure Maven is available at runtime. As this uses Maven under the hood you can also use the standard Maven settings.xml file.

Example 4.50. Author - pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.drools</groupId>
    <artifactId>drools-examples-api</artifactId>
    <version>6.0.0</version>
  </parent>

  <artifactId>kiecontainer-from-kierepo</artifactId>
  <name>Drools API examples - KieContainer from KieRepo</name>

  <dependencies>
    <dependency>
      <groupId>org.kie</groupId>
      <artifactId>kie-ci</artifactId>
    </dependency>
  </dependencies>

</project>
```

Example 4.51. Build and Install - Maven

```
mvn install
```

In the previous examples the classpath KieContainer used. This example creates a dynamic KieContainer as specified by the ReleaseId. The ReleaseId uses Maven conventions for group id, artifact id and version. It also obeys LATEST and SNAPSHOT for versions.

Example 4.52. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();

// Install example1 in the local Maven repo before to do this
KieContainer    kContainer    =    ks.newKieContainer(ks.newReleaseId("org.drools",    "named-
kiesession", "6.0.0-SNAPSHOT"));

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.setGlobal("out", out);

Object msg1 = createMessage(kContainer, "Dave", "Hello, HAL. Do you read me, HAL?");
kSession.insert(msg1);
kSession.fireAllRules();
```

4.2.6.6. Default KieSession from File

- Project: default-kiesession-from-file
- Summary: Dynamic KieModules can also be loaded from any Resource location. The loaded KieModule provides default KieBase and KieSession definitions.

No kmodule.xml file exists. The project 'default-kiesession' must be built first, so that the resulting JAR, in the target folder, can be referenced as a File.

Example 4.53. Build and Install - Maven

```
mvn install
```

Any KieModule can be loaded from a Resource location and added to the KieRepository. Once deployed in the KieRepository it can be resolved via its ReleaseId. Note neither Maven or kie-ci are needed here. It will not set up a transitive dependency parent classloader.

Example 4.54. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieRepository kr = ks.getRepository();

KieModule  kModule  =  kr.addKieModule(ks.getResources().newFileSystemResource(getFile("default-
kiesession")));

KieContainer kContainer = ks.newKieContainer(kModule.getReleaseId());

KieSession kSession = kContainer.newKieSession();
kSession.setGlobal("out", out);

Object msg1 = createMessage(kContainer, "Dave", "Hello, HAL. Do you read me, HAL?");
```

```
kSession.insert(msg1);
kSession.fireAllRules();
```

4.2.6.7. Named KieSession from File

- Project: named-kiesession-from-file
- Summary: Dynamic KieModules can also be loaded from any Resource location. The loaded KieModule provides named KieBase and KieSession definitions.

No kmodule.xml file exists. The project 'named-kiesession' must be built first, so that the resulting JAR, in the target folder, can be referenced as a File.

Example 4.55. Build and Install - Maven

```
mvn install
```

Any KieModule can be loaded from a Resource location and added to the KieRepository. Once in the KieRepository it can be resolved via its ReleaseId. Note neither Maven or kie-ci are needed here. It will not setup a transitive dependency parent classloader.

Example 4.56. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieRepository kr = ks.getRepository();

KieModule  kModule  =  kr.addKieModule(ks.getResources().newFileSystemResource(getFile("named-
kiesession")));

KieContainer kContainer = ks.newKieContainer(kModule.getReleaseId());

KieSession kSession = kContainer.newKieSession("ksession1");
kSession.setGlobal("out", out);

Object msg1 = createMessage(kContainer, "Dave", "Hello, HAL. Do you read me, HAL?");
kSession.insert(msg1);
kSession.fireAllRules();
```

4.2.6.8. KieModule with Dependent KieModule

- Project: kie-module-form-multiple-files
- Summary: Programmatically provide the list of dependant KieModules, without using Maven to resolve anything.

No kmodule.xml file exists. The projects 'named-kiesession' and 'kiebase-include' must be built first, so that the resulting JARs, in the target folders, can be referenced as Files.

Example 4.57. Build and Install - Maven

```
mvn install
```

Creates two resources. One is for the main KieModule 'exRes1' the other is for the dependency 'exRes2'. Even though kie-ci is not present and thus Maven is not available to resolve the dependencies, this shows how you can manually specify the dependent KieModules, for the vararg.

Example 4.58. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieRepository kr = ks.getRepository();

Resource ex1Res = ks.getResources().newFileSystemResource(getFile("kiebase-inclusion"));
Resource ex2Res = ks.getResources().newFileSystemResource(getFile("named-kiesession"));

KieModule kModule = kr.addKieModule(ex1Res, ex2Res);
KieContainer kContainer = ks.newKieContainer(kModule.getReleaseId());

KieSession kSession = kContainer.newKieSession("ksession2");
kSession.setGlobal("out", out);

Object msg1 = createMessage(kContainer, "Dave", "Hello, HAL. Do you read me, HAL?");
kSession.insert(msg1);
kSession.fireAllRules();

Object msg2 = createMessage(kContainer, "Dave", "Open the pod bay doors, HAL.");
kSession.insert(msg2);
kSession.fireAllRules();
```

4.2.6.9. Programmaticaly build a Simple KieModule with Defaults

- Project: kiemoduelmodel-example
- Summary: Programmaticaly buid a KieModule from just a single file. The POM and models are all defaulted. This is the quickest out of the box approach, but should not be added to a Maven repository.

Example 4.59. Build and Install - Maven

```
mvn install
```

This programmatically builds a KieModule. It populates the model that represents the ReleaseId and kmodule.xml, and it adds the relevant resources. A pom.xml is generated from the ReleaseId.

Example 4.60. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieRepository kr = ks.getRepository();
KieFileSystem kfs = ks.newKieFileSystem();

kfs.write("src/main/resources/org/kie/example5/HAL5.drl", getRule());

KieBuilder kb = ks.newKieBuilder(kfs);

kb.buildAll(); // kieModule is automatically deployed to KieRepository if successfully built.
if (kb.getResults().hasMessages(Level.ERROR)) {
    throw new RuntimeException("Build Errors:\n" + kb.getResults().toString());
}

KieContainer kContainer = ks.newKieContainer(kr.getDefaultReleaseId());

KieSession kSession = kContainer.newKieSession();
kSession.setGlobal("out", out);

kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
kSession.fireAllRules();
```

4.2.6.10. Programmaticaly build a KieModule using Meta Models

- Project: kiemoduelmodel-example
- Summary: Programmaticaly build a KieModule, by creating its kmodule.xml meta model resources.

Example 4.61. Build and Install - Maven

```
mvn install
```

This programmatically builds a KieModule. It populates the model that represents the ReleaseId and kmodule.xml, as well as add the relevant resources. A pom.xml is generated from the ReleaseId.

Example 4.62. Utilize and Run - Java

```
KieServices ks = KieServices.Factory.get();
KieFileSystem kfs = ks.newKieFileSystem();

Resource ex1Res = ks.getResources().newFileSystemResource(getFile("named-kiesession"));
Resource ex2Res = ks.getResources().newFileSystemResource(getFile("kiebase-inclusion"));

ReleaseId rid = ks.newReleaseId("org.drools", "kiemodulemodel-example", "6.0.0-SNAPSHOT");
kfs.generateAndWritePomXML(rid);
```

```
KieModuleModel kModuleModel = ks.newKieModuleModel();
kModuleModel.newKieBaseModel("kiemodulemodel")
            .addInclude("kiebase1")
            .addInclude("kiebase2")
            .newKieSessionModel("ksession6");

kfs.writeKModuleXML(kModuleModel.toXML());
kfs.write("src/main/resources/kiemodulemodel/HAL6.drl", getRule());

KieBuilder kb = ks.newKieBuilder(kfs);
kb.setDependencies(ex1Res, ex2Res);
kb.buildAll(); // kieModule is automatically deployed to KieRepository if successfully built.
if (kb.getResults().hasMessages(Level.ERROR)) {
    throw new RuntimeException("Build Errors:\n" + kb.getResults().toString());
}

KieContainer kContainer = ks.newKieContainer(rid);

KieSession kSession = kContainer.newKieSession("ksession6");
kSession.setGlobal("out", out);

Object msg1 = createMessage(kContainer, "Dave", "Hello, HAL. Do you read me, HAL?");
kSession.insert(msg1);
kSession.fireAllRules();

Object msg2 = createMessage(kContainer, "Dave", "Open the pod bay doors, HAL.");
kSession.insert(msg2);
kSession.fireAllRules();

Object msg3 = createMessage(kContainer, "Dave", "What's the problem?");
kSession.insert(msg3);
kSession.fireAllRules();
```

4.3. Security

4.3.1. Security Manager

The KIE engine is a platform for the modelling and execution of business behavior, using a multitude of declarative abstractions and metaphores, like rules, processes, decision tables and etc.

Many times, the authoring of these metaphores is done by third party groups, be it a different group inside the same company, a group from a partner company, or even anonymous third parties on the internet.

Rules and Processes are designed to execute arbitrary code in order to do their job, but in such cases it might be necessary to constrain what they can do. For instance, it is unlikely a rule should be allowed to create a classloader (what could open the system to an attack) and certainly it should not be allowed to make a call to `System.exit()`.

The Java Platform provides a very comprehensive and well defined security framework that allows users to define policies for what a system can do. The KIE platform leverages that framework and allow application developers to define a specific policy to be applied to any execution of user provided code, be it in rules, processes, work item handlers and etc.

4.3.1.1. How to define a KIE Policy

Rules and processes can run with very restrict permissions, but the engine itself needs to perform many complex operations in order to work. Examples are: it needs to create classloaders, read system properties, access the file system, etc.

Once a security manager is installed, though, it will apply restrictions to all the code executing in the JVM according to the defined policy. For that reason, KIE allows the user to define two different policy files: one for the engine itself and one for the assets deployed into and executed by the engine.

One easy way to setup the enviroment is to give the engine itself a very permissive policy, while providing a constrained policy for rules and processes.

Policy files follow the standard policy file syntax as described in the Java documentation. For more details, see:

http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax

A permissive policy file for the engine can look like the following:

Example 4.63. A sample engine.policy file

```
grant {
    permission java.security.AllPermission;
}
```

An example security policy for rules could be:

Example 4.64. A sample rules.policy file

```
grant {
    permission java.util.PropertyPermission "*", "read";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
}
```

Please note that depending on what the rules and processes are supposed to do, many more permissions might need to be granted, like accessing files in the filesystem, databases, etc.

In order to use these policy files, all that is necessary is to execute the application with these files as parameters to the JVM. Three parameters are required:

Table 4.3. Parameters

Parameter	Meaning
-Djava.security.manager	Enables the security manager

Parameter	Meaning
-Djava.security.policy=<jvm_policy_file>	Defines the global policy file to be applied to the whole application, including the engine
-Dkie.security.policy=<kie_policy_file>	Defines the policy file to be applied to rules and processes

For instance:

```
java -Djava.security.manager -Djava.security.policy=global.policy -Dkie.security.policy=rules.policy foo.bar.MyApp
```

Note

When executing the engine inside a container, use your container's documentation to find out how to configure the Security Manager and how to define the global security policy. Define the kie security policy as described above and set the `kie.security.policy` system property in order to configure the engine to use it.

Note

Please note that unless a Security Manager is configured, the `kie.security.policy` will be ignored.

Note

A Security Manager has a high performance impact in the JVM. Applications with strict performance requirements are strongly discouraged of using a Security Manager. An alternative is the use of other security procedures like the auditing of rules/processes before testing and deployment to prevent malicious code from being deployed to the environment.

Part III. Drools
Runtime and Language

Drools is a powerful Hybrid Reasoning System.

Chapter 5. Hybrid Reasoning

5.1. Artificial Intelligence

5.1.1. A Little History

Over the last few decades *artificial intelligence* (AI) became an unpopular term, with the well-known "AI Winter" [http://en.wikipedia.org/wiki/AI_winter]. There were large boasts from scientists and engineers looking for funding, which never lived up to expectations, resulting in many failed projects. Thinking Machines Corporation [http://en.wikipedia.org/wiki/Thinking_Machines_Corporation] and the 5th Generation Computer [http://en.wikipedia.org/wiki/Fifth-generation_computer] (5GP) project probably exemplify best the problems at the time.

Thinking Machines Corporation was one of the leading AI firms in 1990, it had sales of nearly $65 million. Here is a quote from its brochure:

“Some day we will build a thinking machine. It will be a truly intelligent machine. One that can see and hear and speak. A machine that will be proud of us.”

Yet 5 years later it filed for bankruptcy protection under Chapter 11. The site inc.com has a fascinating article titled "The Rise and Fall of Thinking Machines" [http://www.inc.com/magazine/19950915/2622.html]. The article covers the growth of the industry and how a cosy relationship with Thinking Machines and DARPA [http://en.wikipedia.org/wiki/DARPA] over-heated the market, to the point of collapse. It explains how and why commerce moved away from AI and towards more practical number-crunching super computers.

The 5th Generation Computer project was a USD 400 million project in Japan to build a next generation computer. Valves (or Tubes) was the first generation, transistors the second, integrated circuits the third and finally microprocessors was the fourth. The fifth was intended to be a machine capable of effective Artificial Intelligence. This project spurred an "arms" race with the UK and USA, that caused much of the AI bubble. The 5GP would provide massive multi-cpu parallel processing hardware along with powerful knowledge representation and reasoning software via *Prolog*; a type of *expert system*. By 1992 the project was considered a failure and cancelled. It was the largest and most visible commercial venture for Prolog, and many of the failures are pinned on the problems of trying to run a logic based programming language concurrently on multi CPU hardware with effective results. Some believe that the failure of the 5GP project tainted Prolog and relegated it to academia, see "Whatever Happened to Prolog" [http://www.dvorak.org/blog/whatever-happened-to-prolog/] by John C. Dvorak.

However while research funding dried up and the term AI became less used, many green shoots where planted and continued more quietly under discipline specific names: *cognitive systems*, *machine learning*, *intelligent systems*, *knowledge representation and reasoning*. Offshoots of these then made their way into commercial systems, such as expert systems in the *Business Rules Management System* (BRMS) market.

Imperative, system based languages, languages such as C, C++, Java and C#/.Net have dominated the last 20 years, enabled by the practicality of the languages and ability to run with good performance on commodity hardware. However many believe there is a renaissance underway in the field of AI, spurred by advances in hardware capabilities and AI research. In 2005 Heather Havenstein authored "Spring comes to AI winter" [http://www.computerworld.com/s/article/99691/Spring_comes_to_AI_winter] which outlines a case for this resurgence. Norvig and Russel dedicate several pages to what factors allowed the industry to overcome it's problems and the research that came about as a result:

> Recent years have seen a revolution in both the content and the methodology of work in artificial intelligence. It is now more common to build on existing theories than to propose brand-new ones, to base claims on rigorous theorems or hard experimental evidence rather than on intuition, and to show relevance to real-world applications rather than toy examples.
>
> —Artificial Intelligence: A Modern Approach

Computer vision, *neural networks*, *machine learning* and *knowledge representation and reasoning* (KRR) have made great strides towards becoming practical in commercial environments. For example, vision-based systems can now fully map out and navigate their environments with strong recognition skills. As a result we now have self-driving cars about to enter the commercial market. *Ontological* research, based around description logic, has provided very rich semantics to represent our world. Algorithms such as the tableaux algorithm have made it possible to use those rich semantics effectively in large complex ontologies. Early KRR systems, like Prolog in 5GP, were dogged by the limited semantic capabilities and memory restrictions on the size of those ontologies.

5.1.2. Knowledge Representation and Reasoning

In A Little History talks about AI as a broader subject and touches on Knowledge Representation and Reasoning (KRR) and also Expert Systems, I'll come back to Expert Systems later.

KRR is about how we represent our knowledge in symbolic form, i.e. how we describe something. Reasoning is about how we go about the act of thinking using this knowledge. System based object-oriented languages, like C++, Java and C#, have data definitions called classes for describing the composition and behaviour of modeled entities. In Java we call exemplars of these described things beans or instances. However those classification systems are limited to ensure computational efficiency. Over the years researchers have developed increasingly sophisticated ways to represent our world. Many of you may already have heard of OWL (Web Ontology Language). There is always a gap between what can be theoretically represented and what can be used computationally in practically timely manner, which is why OWL has different sub-languages from Lite to Full. It is not believed that any reasoning system can support OWL Full. However, algorithmic advances continue to narrow that gap and improve the expressiveness available to reasoning engines.

There are also many approaches to how these systems go about thinking. You may have heard discussions comparing the merits of forward chaining, which is reactive and data driven, with

backward chaining, which is passive and query driven. Many other types of reasoning techniques exist, each of which enlarges the scope of the problems we can tackle declaratively. To list just a few: imperfect reasoning (fuzzy logic, certainty factors), defeasible logic, belief systems, temporal reasoning and correlation. You don't need to understand all these terms to understand and use Drools. They are just there to give an idea of the range of scope of research topics, which is actually far more extensive, and continues to grow as researchers push new boundaries.

KRR is often referred to as the core of Artificial Intelligence. Even when using biological approaches like neural networks, which model the brain and are more about pattern recognition than thinking, they still build on KRR theory. My first endeavours with Drools were engineering oriented, as I had no formal training or understanding of KRR. Learning KRR has allowed me to get a much wider theoretical background. Allowing me to better understand both what I've done and where I'm going, as it underpins nearly all of the theoretical side to our Drools R&D. It really is a vast and fascinating subject that will pay dividends for those who take the time to learn. I know it did and still does for me. Bracham and Levesque have written a seminal piece of work, called "Knowledge Representation and Reasoning" that is a must read for anyone wanting to build strong foundations. I would also recommend the Russel and Norvig book "Artificial Intelligence, a modern approach" which also covers KRR.

5.1.3. Rule Engines and Production Rule Systems (PRS)

We've now covered a brief history of AI and learnt that the core of AI is formed around KRR. We've shown than KRR is a vast and fascinating subject which forms the bulk of the theory driving Drools R&D.

The rule engine is the computer program that delivers KRR functionality to the developer. At a high level it has three components:

- Ontology
- Rules
- Data

As previously mentioned the ontology is the representation model we use for our "things". It could use records or Java classes or full-blown OWL based ontologies. The rules perform the reasoning, i.e., they facilitate "thinking". The distinction between rules and ontologies blurs a little with OWL based ontologies, whose richness is rule based.

The term "rules engine" is quite ambiguous in that it can be any system that uses rules, in any form, that can be applied to data to produce outcomes. This includes simple systems like form validation and dynamic expression engines. The book "How to Build a Business Rules Engine" (2004) by Malcolm Chisholm exemplifies this ambiguity. The book is actually about how to build and alter a database schema to hold validation rules. The book then shows how to generate Visual Basic code from those validation rules to validate data entry. While perfectly valid, this is very different to what we are talking about.

Drools started life as a specific type of rule engine called a Production Rule System (PRS) and was based around the Rete algorithm (usually pronounced as two syllables, e.g., REH-te or RAY-tay). The Rete algorithm, developed by Charles Forgy in 1974, forms the brain of a Production Rule System and is able to scale to a large number of rules and facts. A Production Rule is a two-part structure: the engine matches facts and data against Production Rules - also called Productions or just Rules - to infer conclusions which result in actions.

```
when
    <conditions>
then
    <actions>;
```

The process of matching the new or existing facts against Production Rules is called *pattern matching*, which is performed by the *inference engine*. Actions execute in response to changes in data, like a database trigger; we say this is a *data driven* approach to reasoning. The actions themselves can change data, which in turn could match against other rules causing them to fire; this is referred to as forward chaining

Drools 5.x implements and extends the *Rete* algorithm. This extended Rete algorithm is named *ReteOO*, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for object oriented systems. Other Rete based engines also have marketing terms for their proprietary enhancements to Rete, like RetePlus and Rete III. The most common enhancements are covered in "Production Matching for Large Learning Systems" (1995) by Robert B. Doorenbos' thesis, which presents Rete/UL. Drools 6.x introduces a new lazy algorithm named *PHREAK*; which is covered in more detail in the PHEAK algorithm section.

The Rules are stored in the Production Memory and the facts that the Inference Engine matches against are kept in the Working Memory. Facts are asserted into the Working Memory where they may then be modified or retracted. A system with a large number of rules and facts may result in many rules being true for the same fact assertion; these rules are said to be in conflict. The Agenda manages the execution order of these conflicting rules using a Conflict Resolution strategy.

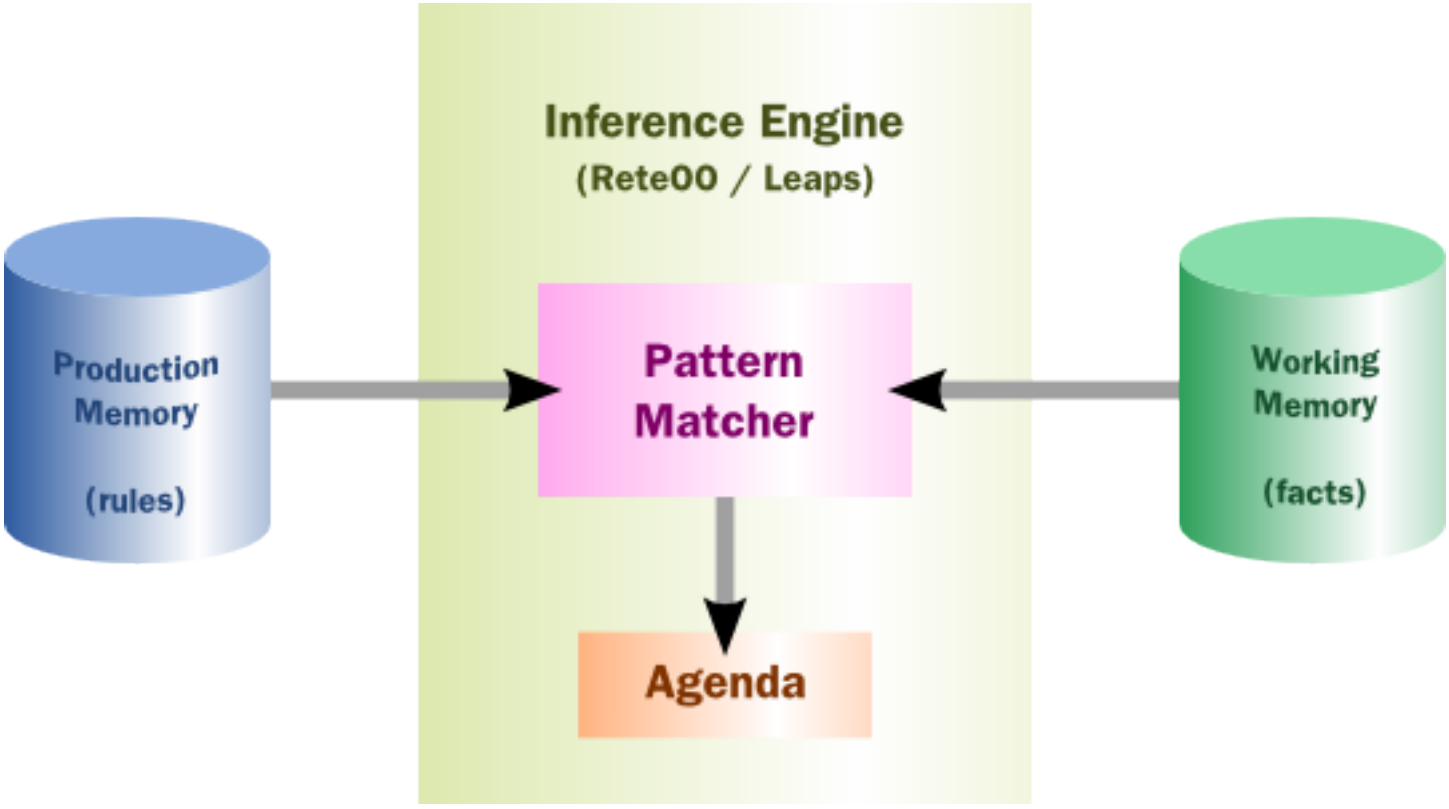


Figure 5.1. High-level View of a Production Rule System

5.1.4. Hybrid Reasoning Systems (HRS)

You may have read discussions comparing the merits of forward chaining (reactive and data driven) or backward chaining (passive query). Here is a quick explanation of these two main types of reasoning.

Forward chaining is "data-driven" and thus reactionary, with facts being asserted into working memory, which results in one or more rules being concurrently true and scheduled for execution by the Agenda. In short, we start with a fact, it propagates through the rules, and we end in a conclusion.

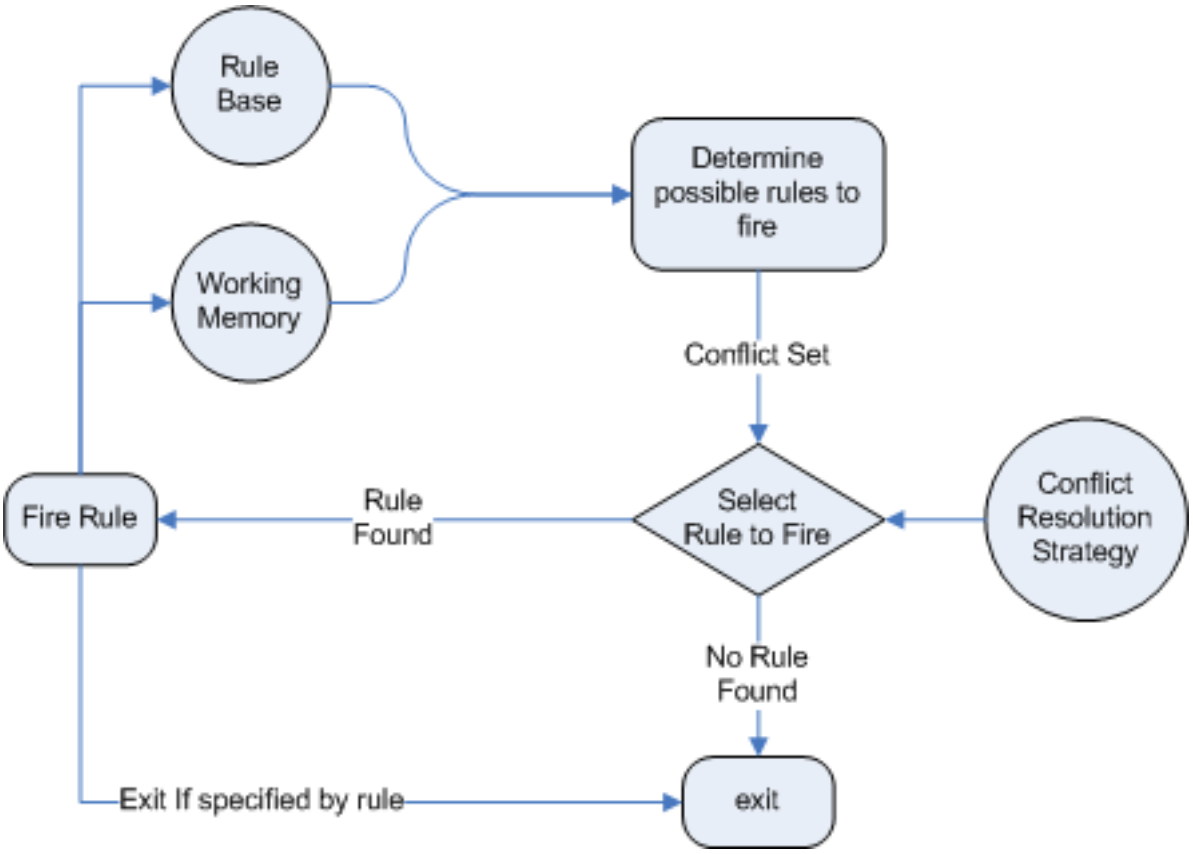


Figure 5.2. Forward Chaining

Backward chaining is "goal-driven", meaning that we start with a conclusion which the engine tries to satisfy. If it can't, then it searches for conclusions that it can satisfy. These are known as subgoals, that will help satisfy some unknown part of the current goal. It continues this process until either the initial conclusion is proven or there are no more subgoals. Prolog is an example of a Backward Chaining engine. Drools can also do backward chaining, which we refer to as derivation queries.

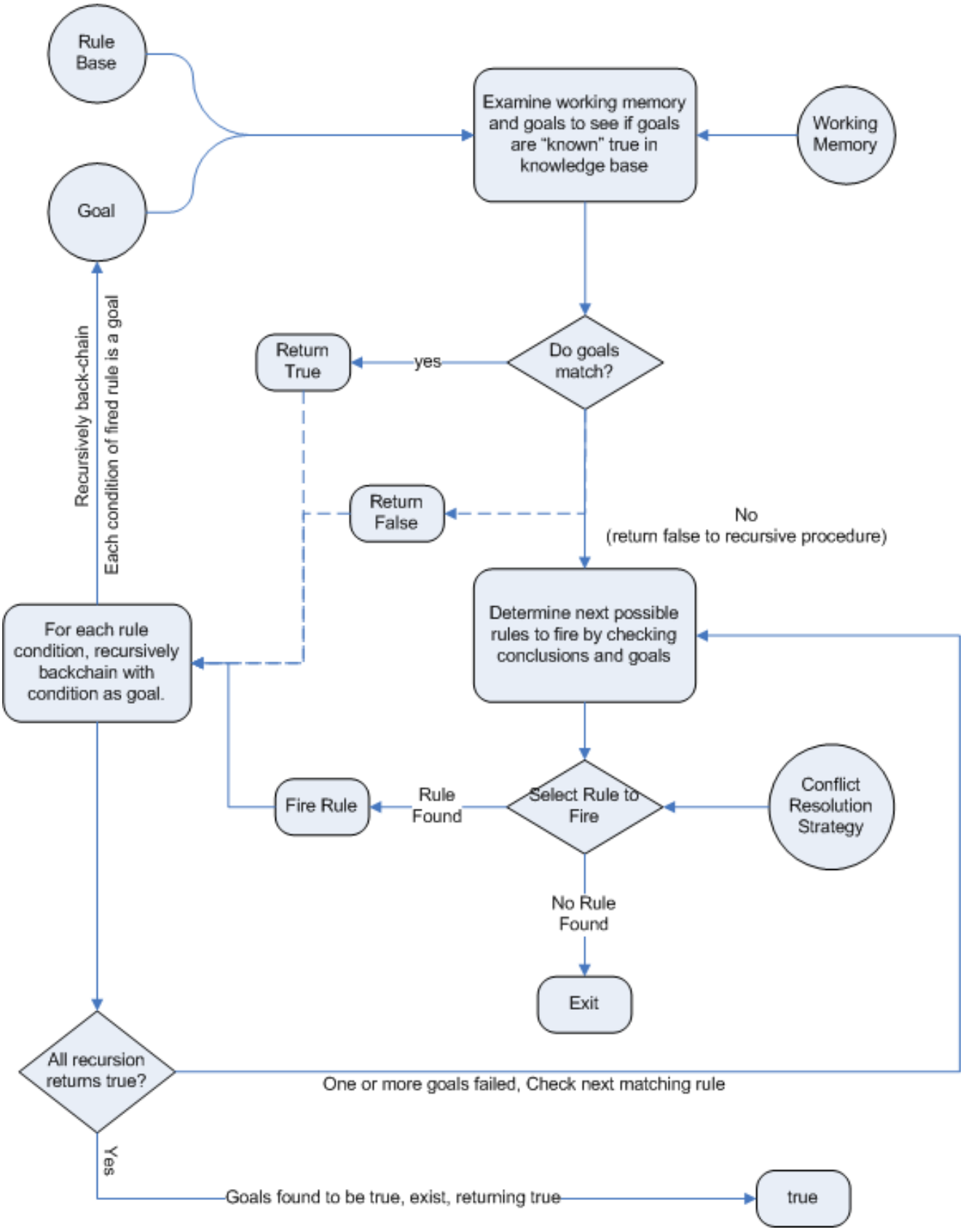


Figure 5.3. Backward Chaining

Historically you would have to make a choice between systems like OPS5 (forward) or Prolog (backward). Nowadays many modern systems provide both types of reasoning capabilities. There are also many other types of reasoning techniques, each of which enlarges the scope of the problems we can tackle declaratively. To list just a few: imperfect reasoning (fuzzy logic, certainty factors), defeasible logic, belief systems, temporal reasoning and correlation. Modern systems are merging these capabilities, and others not listed, to create *hybrid reasoning systems* (HRS).

While Drools started out as a PRS, 5.x introduced Prolog style backward chaining reasoning as well as some functional programming styles. For this reason we now prefer the term Hybrid Reasoning System when describing Drools.

Drools currently provides crisp reasoning, but imperfect reasoning is almost ready. Initially this will be imperfect reasoning with fuzzy logic; later we'll add support for other types of uncertainty. Work is also under way to bring OWL based ontological reasoning, which will integrate with our *traits* system. We also continue to improve our functional programming capabilities.

5.1.5. Expert Systems

You will often hear the terms *expert systems* used to refer to *production rule systems* or *Prolog*-like systems. While this is normally acceptable, it's technically incorrect as these are frameworks to build expert systems with, rather than expert systems themselves. It becomes an expert system once there is an ontological model to represent the domain and there are facilities for knowledge acquisition and explanation.

Mycin is the most famous expert system, built during the 70s. It is still heavily covered in academic literature, such as the recommended book "Expert Systems" by Peter Jackson.

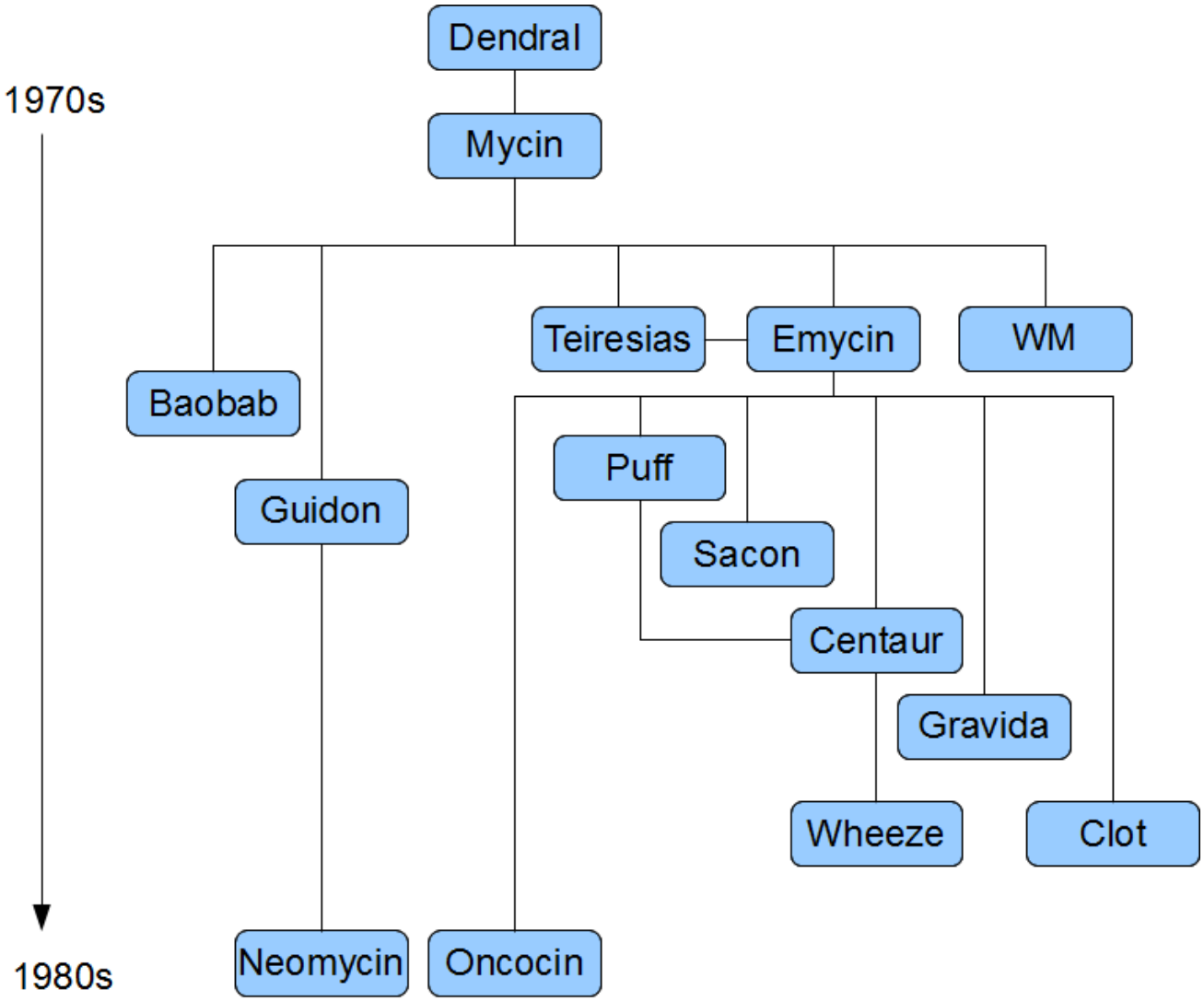


Figure 5.4. Early History of Expert Systems

5.1.6. Recommended Reading

General AI, KRR and Expert System Books

For those wanting to get a strong theoretical background in KRR and expert systems, I'd strongly recommend the following books. "Artificial Intelligence: A Modern Approach" is a must have, for anyone's bookshelf.

- Introduction to Expert Systems
  - Peter Jackson
- Expert Systems: Principles and Programming

  - Joseph C. Giarratano, Gary D. Riley

- Knowledge Representation and Reasoning
  - Ronald J. Brachman, Hector J. Levesque
- Artificial Intelligence : A Modern Approach.
  - Stuart Russell and Peter Norvig

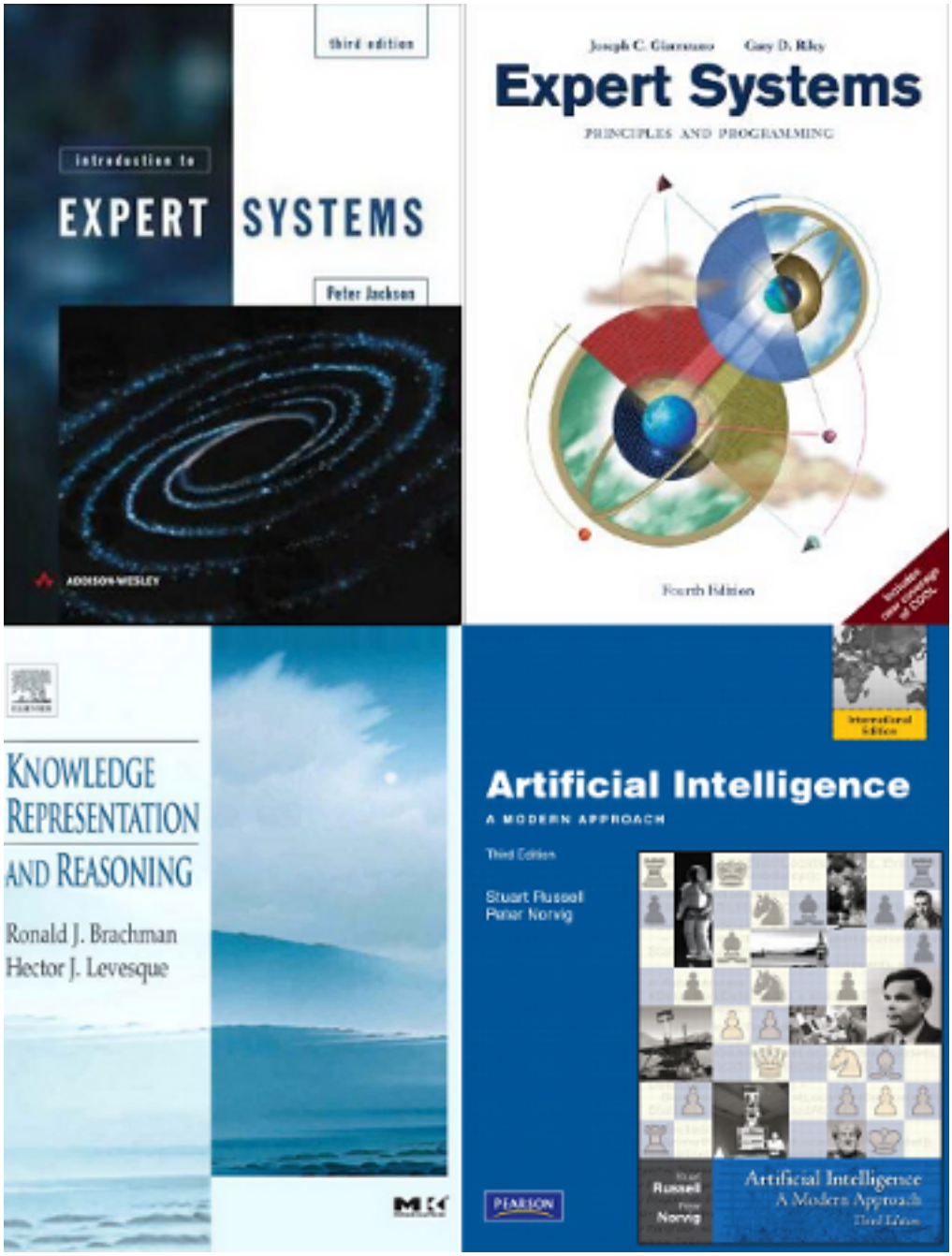


Figure 5.5. Recommended Reading

Papers

Here are some recommended papers that cover interesting areas in rule engine research:

- Production Matching for Large Learning Systems: Rete/UL (1993)
  - Robert B. Doorenbos
- Advances In Rete Pattern Matching
  - Marshall Schor, Timothy P. Daly, Ho Soo Lee, Beth R. Tibbitts (AAAI 1986)
- Collection-Oriented Match
  - Anurag Acharya and Milind Tambe (1993)
- The Leaps Algorithm
  - Don Batery (1990)
- Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing
  - Eric Hanson , Mohammed S. Hasan (1993)

Drools Books

There are currently three Drools books, all from Packt Publishing.

- JBoss Drools Business Rules
  - Paul Browne
- Drools JBoss Rules 5.0 Developers Guide
  - Michal Bali
- Drools Developer's Cookbook
  - Lucas Amador

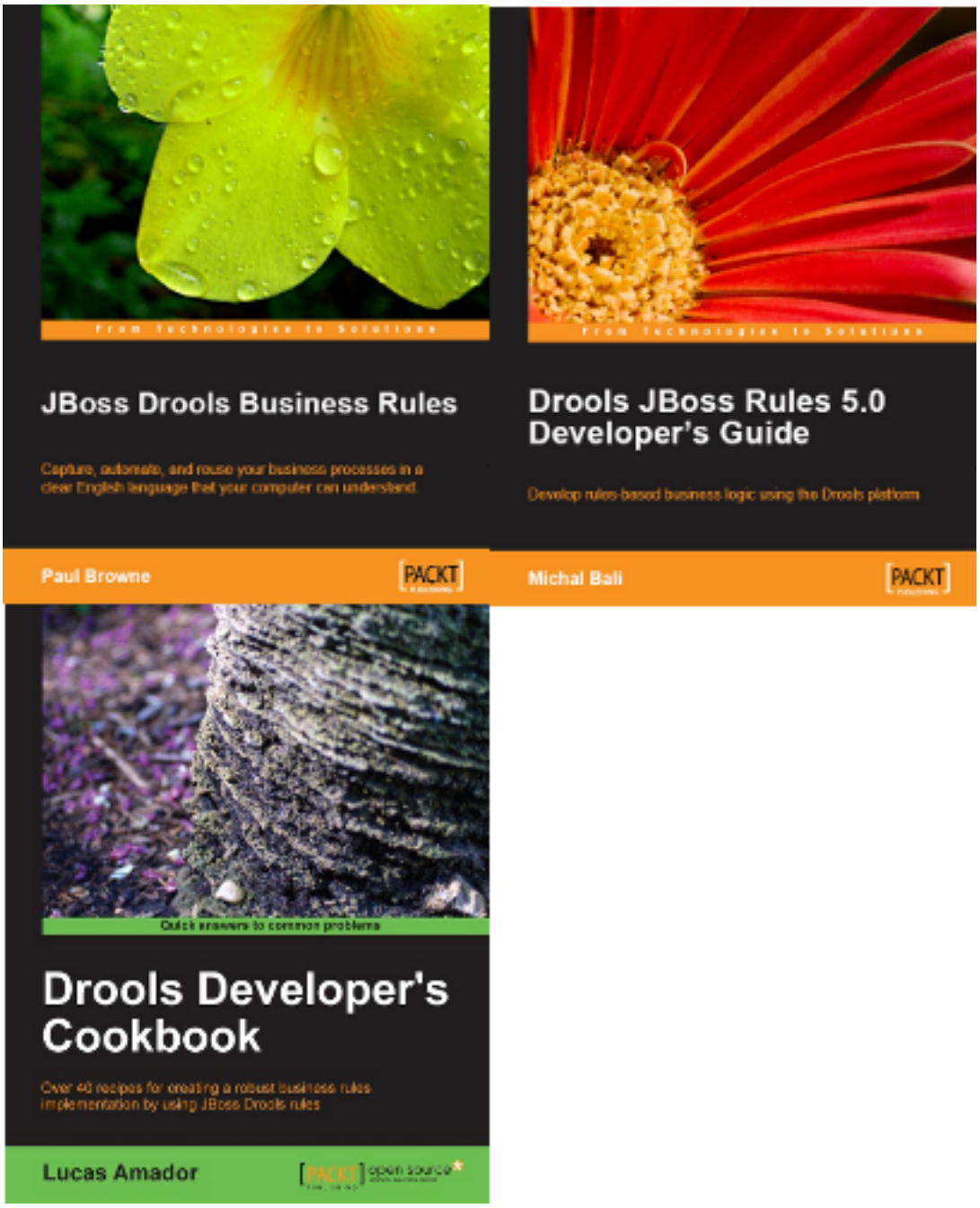


Figure 5.6. Recommended Reading

5.2. Rete Algorithm

The *Rete* algorithm was invented by Dr. Charles Forgy and documented in his PhD thesis in 1978-79. A simplified version of the paper was published in 1982 (http://citeseer.ist.psu.edu/context/505087/0). The latin word "rete" means "net" or "network". The Rete algorithm can be broken into 2 parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory are processed to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data as it propagates through the network. The nodes at the top of the network would have many matches, and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. In Dr. Forgy's 1982 paper, he described 4 basic nodes: root, 1-input, 2-input and terminal.

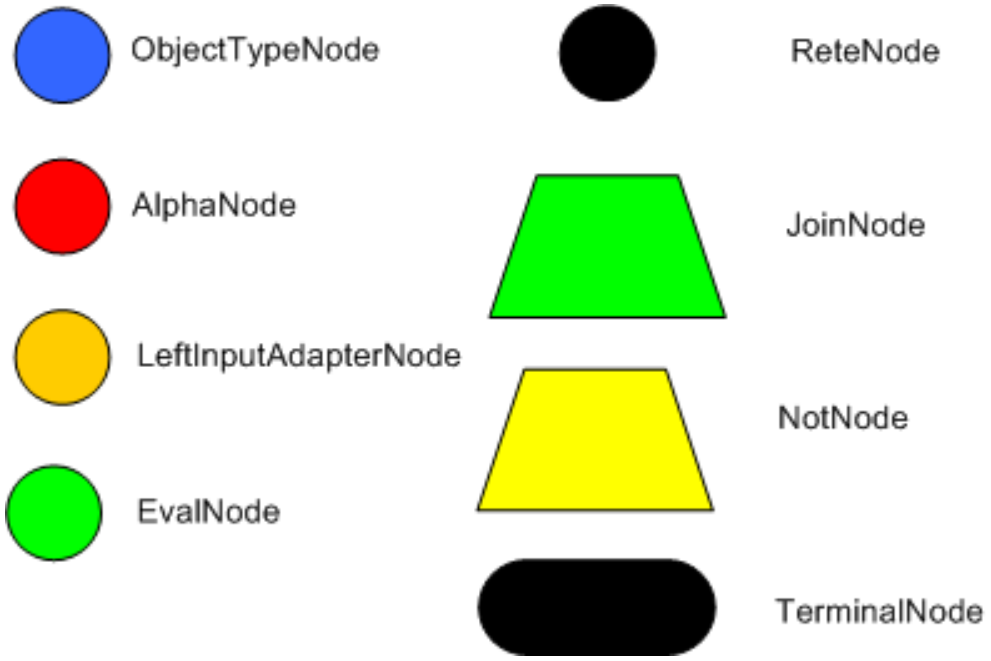


Figure 5.7. Rete Nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode. The purpose of the ObjectTypeNode is to make sure the engine doesn't do more work than it needs to. For example, say we have 2 objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeNode and have all 1-input and 2-input nodes descend from it. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an `instanceof` check.

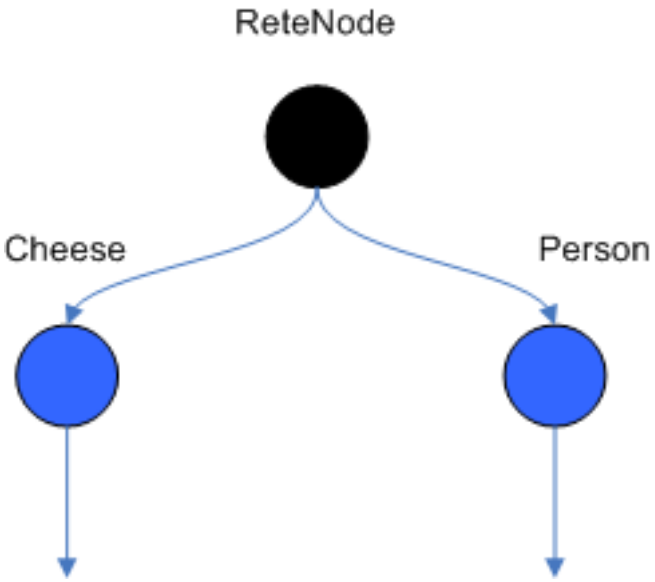


Figure 5.8. ObjectTypeNodes

ObjectTypeNodes can propagate to AlphaNodes, LeftInputAdapterNodes and BetaNodes. AlphaNodes are used to evaluate literal conditions. Although the 1982 paper only covers equality conditions, many RETE implementations support other operations. For example, `Account.name == "Mr Trout"` is a literal condition. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode. In Dr. Forgy's paper, he refers to these as IntraElement conditions. The following diagram shows the AlphaNode combinations for Cheese(name == "cheddar", strength == "strong"):

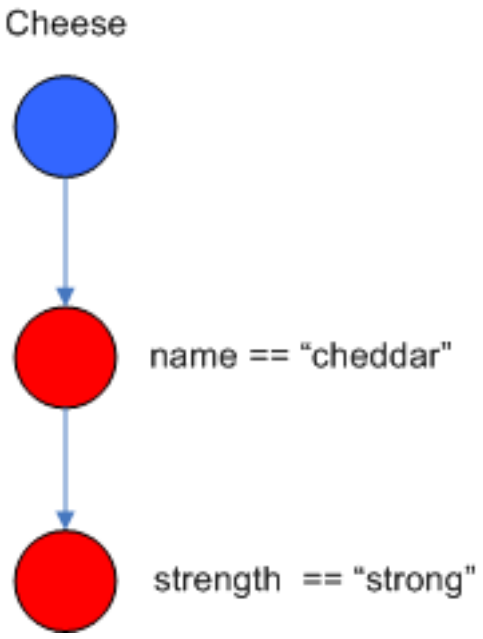


Figure 5.9. AlphaNodes

Drools extends Rete by optimizing the propagation from ObjectTypeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap,thereby avoiding unnecessary literal checks.

There are two two-input nodes, JoinNode and NotNode, and both are types of BetaNodes. BetaNodes are used to compare 2 objects, and their fields, to each other. The objects may be the same or different types. By convention we refer to the two inputs as left and right. The left input for a BetaNode is generally a list of objects; in Drools this is a Tuple. The right input is a single object. Two Nodes can be used to implement 'exists' checks. BetaNodes also have memory. The left input is called the Beta Memory and remembers all incoming tuples. The right input is called the Alpha Memory and remembers all incoming objects. Drools extends Rete by performing indexing on the BetaNodes. For instance, if we know that a BetaNode is performing a check on a String field, as each object enters we can do a hash lookup on that String value. This means when facts enter from the opposite side, instead of iterating over all the facts to find valid joins, we do a lookup returning potentially valid candidates. At any point a valid join is found the Tuple is joined with the Object; which is referred to as a partial match; and then propagated to the next node.

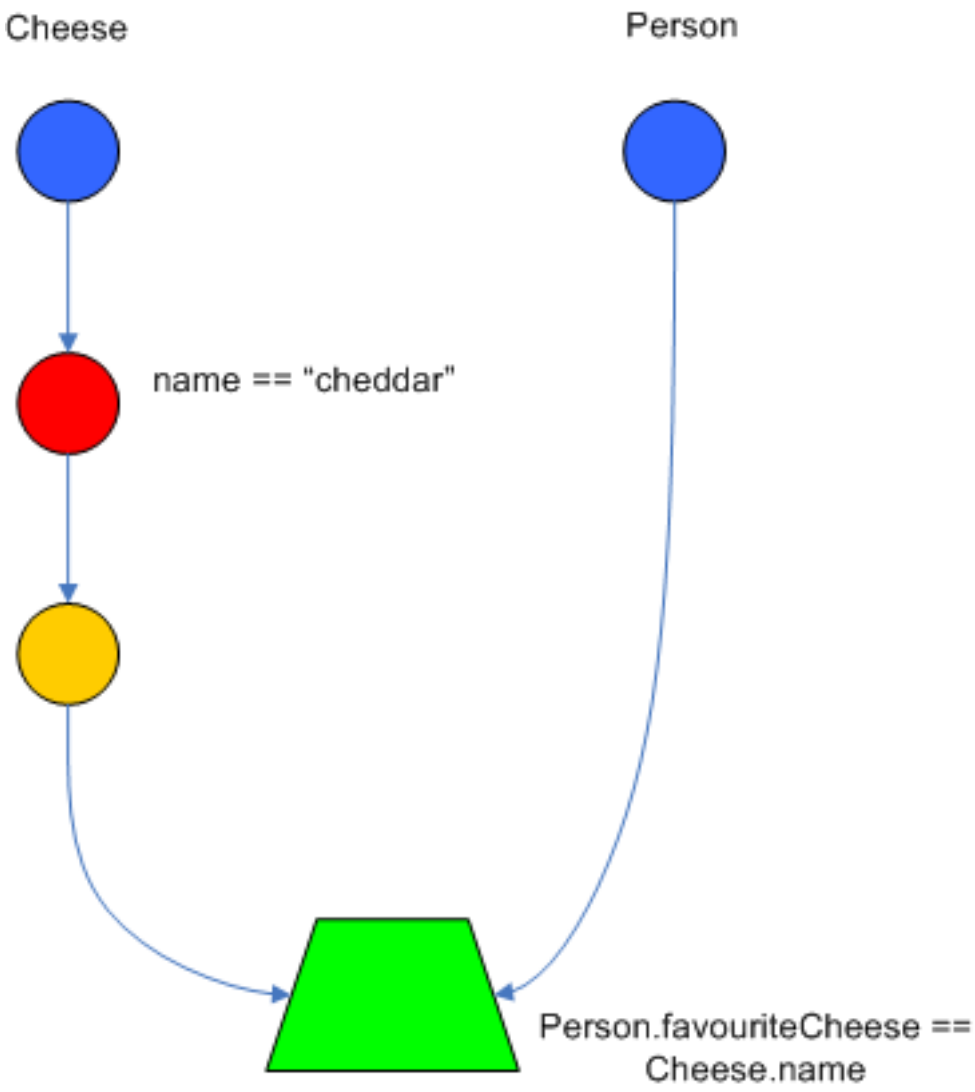


Figure 5.10. JoinNode

To enable the first Object, in the above case Cheese, to enter the network we use a LeftInputNodeAdapter - this takes an Object as an input and propagates a single Object Tuple.

Terminal nodes are used to indicate a single rule having matched all its conditions; at this point we say the rule has a full match. A rule with an 'or' conditional disjunctive connective results in subrule generation for each possible logically branch; thus one rule can have multiple terminal nodes.

Drools also performs node sharing. Many rules repeat the same patterns, and node sharing allows us to collapse those patterns so that they don't have to be re-evaluated for every single instance. The following two rules share the first pattern, but not the last:

```
rule
when
    Cheese( $cheddar : name == "cheddar" )
    $person : Person( favouriteCheese == $cheddar )
then
```

```
    System.out.println( $person.getName() + " likes cheddar" );
end
```

```
rule
when
    Cheese( $cheddar : name == "cheddar" )
    $person : Person( favouriteCheese != $cheddar )
then
    System.out.println( $person.getName() + " does not like cheddar" );
end
```

As you can see below, the compiled Rete network shows that the alpha node is shared, but the beta nodes are not. Each beta node has its own TerminalNode. Had the second pattern been the same it would have also been shared.

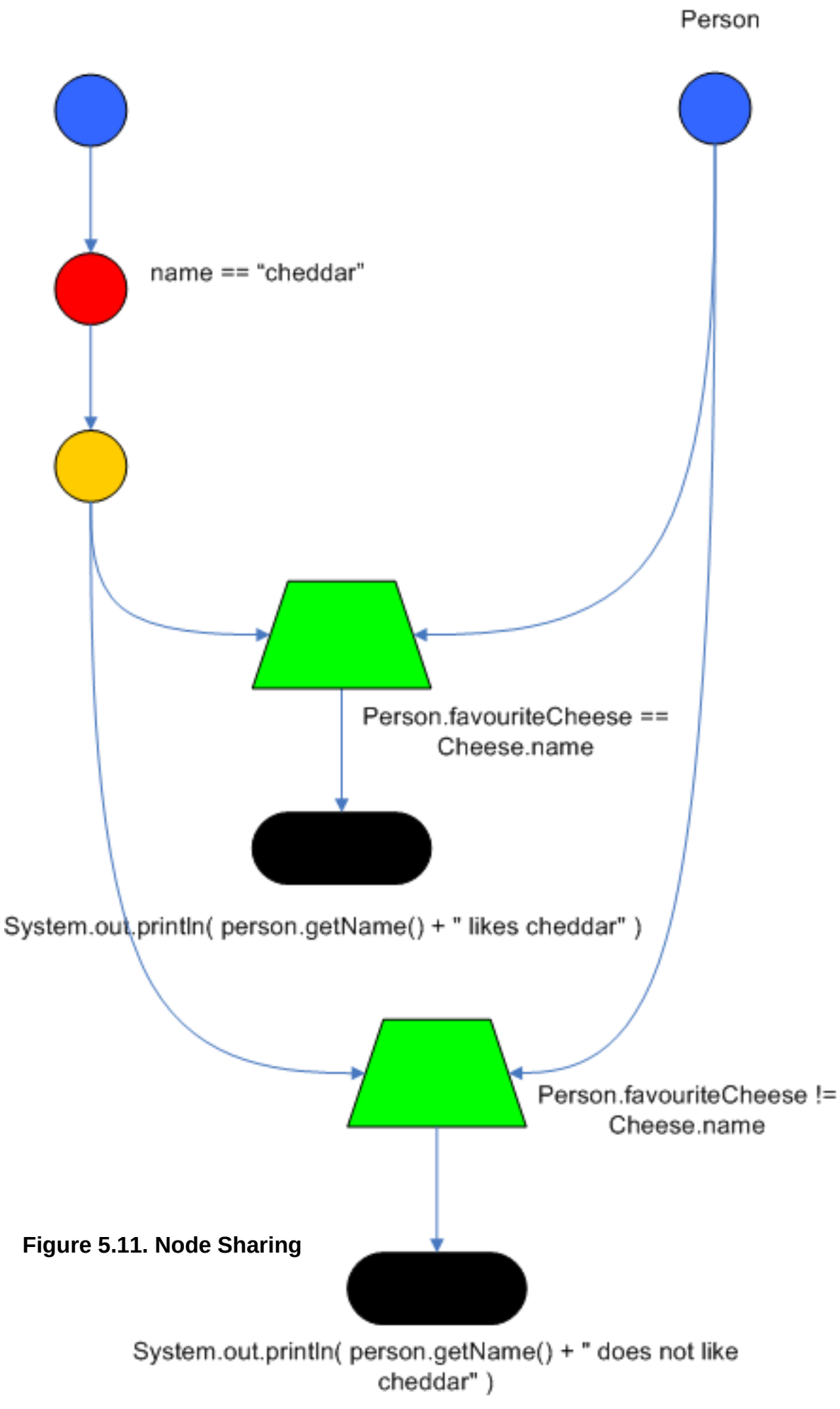


Figure 5.11. Node Sharing

5.3. ReteOO Algorithm

The *ReteOO* was developed throughout the 3, 4 and 5 series releases. It takes the RETE algorithm and applies well known enhancements, all of which are covered by existing academic literature:

Node sharing

- Sharing is applied to both the alpha and beta network. The beta network sharing is always from the root pattern.

Alpha indexing

- Alpha Nodes with many children use a hash lookup mechanism, to avoid testing each result.

Beta indexing

- Join, Not and Exist nodes indexing their memories using a hash. This reduces the join attempts for equal checks. Recently range indexing was added to Not and Exists.

Tree based graphs

- Join matches did not contain any references to their parent or children matches. Deletions would have to recalculate all join matches again, which involves recreating all those join match objects, to be able to find the parts of the network where the tuples should be deleted. This is called symmetrical propagation. A tree graph provides parent and children references, so a deletion is just a matter of following those references. This is asymmetrical propagation. The result is faster and less impact on the GC, and more robust because changes in values will not cause memory leaks if they happen without the engine being notified.

Modify-in-place

- Traditional RETE implements a modify as a delete + insert. This causes all join tuples to be GC'd, many of which are recreated again as part of the insert. Modify-in-place instead propagates as a single pass, every node is inspected

Property reactive

- Also called "new trigger condition". Allows more fine grained reactivity to updates. A Pattern can react to changes to specific properties and ignore others. This alleviates problems of recursion and also helps with performance.

Sub-networks

- Not, Exists and Accumulate can each have nested conditional elements, which forms sub networks.

Backward Chaining

- Prolog style derivation trees for backward chaining are supported. The implementation is stack based, so does not have method recursion issues for large graphs.

Lazy Truth Maintenance

- Truth maintenance has a runtime cost, which is incurred whether TMS is used or not. Lazy TMS only turns it on, on first use. Further it's only turned on for that object type, so other object types do not incur the runtime cost.

Heap based agenda

- The agenda uses a binary heap queue to sort rule matches by salience, rather than any linear search or maintenance approach.

Dynamic Rules

- Rules can be added and removed at runtime, while the engine is still populated with data.

5.4. PHREAK Algorithm

Drools 6 introduces a new algorithm, that attempts to address some of the core issues of RETE. The algorithm is not a rewrite form scratch and incorporates all of the existing code from ReteOO, and all its enhancements. While PHREAK is an evolution of the RETE algorithm, it is no longer classified as a RETE implementation. In the same way that once an animal evolves beyond a certain point and key characteristics are changed, the animal becomes classified as new species. There are two key RETE characteristics that strongly identify any derivative strains, regardless of optimizations. That it is an eager, data oriented algorithm. Where all work is doing done the insert, update or delete actions; eagerly producing all partial matches for all rules. PHREAK in contrast is characterised as a lazy, goal oriented algorithm; where partial matching is aggressively delayed.

This eagerness of RETE can lead to a lot of churn in large systems, and much wasted work. Where wasted work is classified as matching efforts that do not result in a rule firing.

PHREAK was heavily inspired by a number of algorithms; including (but not limited to) LEAPS, RETE/UL and Collection-Oriented Match. PHREAK has all enhancements listed in the ReteOO section. In addition it adds the following set of enhancements, which are explained in more detail in the following paragraphs.

- Three layers of contextual memory; Node, Segment and Rule memories.
- Rule, segment and node based linking.
- Lazy (delayed) rule evaluation.

- Isolated rule evaluation.
- Set oriented propagations.
- Stack based evaluations, with pause and resume.

When the PHREAK engine is started all rules are said to be unlinked, no rule evaluation can happen while rules are unlinked. The insert, update and deletes actions are queued before entering the beta network. A simple heuristic, based on the rule most likely to result in firings, is used to select the next rule for evaluation; this delays the evaluation and firing of the other rules. Only once a rule has all right inputs populated will the rule be considered linked in, although no work is yet done. Instead a goal is created, that represents the rule, and placed into a priority queue; which is ordered by salience. Each queue itself is associated with an AgendaGroup. Only the active AgendaGroup will inspect its queue, popping the goal for the rule with the highest salience and submitting it for evaluation. So the work done shifts from the insert, update, delete phase to the fireAllRules phase. Only the rule for which the goal was created is evaluated, other potential rule evaluations from those facts are delayed. While individual rules are evaluated, node sharing is still achieved through the process of segmentation, which is explained later.

Each successful join attempt in RETE produces a tuple (or token, or partial match) that will be propagated to the child nodes. For this reason it is characterised as a tuple oriented algorithm. For each child node that it reaches it will attempt to join with the other side of the node, again each successful join attempt will be propagated straight away. This creates a descent recursion effect. Thrashing the network of nodes as it ripples up and down, left and right from the point of entry into the beta network to all the reachable leaf nodes.

PHREAK propagation is set oriented (or collection-oriented), instead of tuple oriented. For the rule being evaluated it will visit the first node and process all queued insert, update and deletes. The results are added to a set and the set is propagated to the child node. In the child node all queued inset, update and deletes are processed, adding the results to the same set. Once finished that set is propagated to the next child node, and so on until the terminal node is reached. This creates a single pass, pipeline type effect, that is isolated to the current rule being evaluated. This creates a batch process effect which can provide performance advantages for certain rule constructs; such as sub-networks with accumulates. In the future it will leans itself to being able to exploit multi-core machines in a number of ways.

The Linking and Unlinking uses a layered bit mask system, based on a network segmentation. When the rule network is built segments are created for nodes that are shared by the same set of rules. A rule itself is made up from a path of segments, although if there is no sharing that will be a single segment. A bit-mask offset is assigned to each node in the segment. Also another bit mask (the layering) is assigned to each segment in the rule's path. When there is at least one input (data propagation) the node's bit is set to on. When each node has its bit set to on the segment's bit is also set to on. Conversely if any node's bit is set to off, the segment is then also set to off. If each segment in the rule's path is set to on, the rule is said to be linked in and a goal is created to schedule the rule for evaluation. The same bit-mask technique is used to also track dirty node, segments and rules; this allows for a rule already link in to be scheduled for evaluation if it's considered dirty since it was last evaluated.

This ensures that no rule will ever evaluate partial matches, if it's impossible for it to result in rule instances because one of the joins has no data. This is possible in RETE and it will merrily churn away producing martial match attempts for all nodes, even if the last join is empty.

While the incremental rule evaluation always starts from the root node, the dirty bit masks are used to allow nodes and segments that are not dirty to be skipped.

Using the existence of at at least one items of data per node, is a fairly basic heuristic. Future work would attempt to delay the linking even further; using techniques such as arc consistency to determine whether or not matching will result in rule instance firings.

Where as RETE has just a singe unit of memory, the node memory, PHREAK has 3 levels of memory. This allows for much more contextual understanding during evaluation of a Rule.

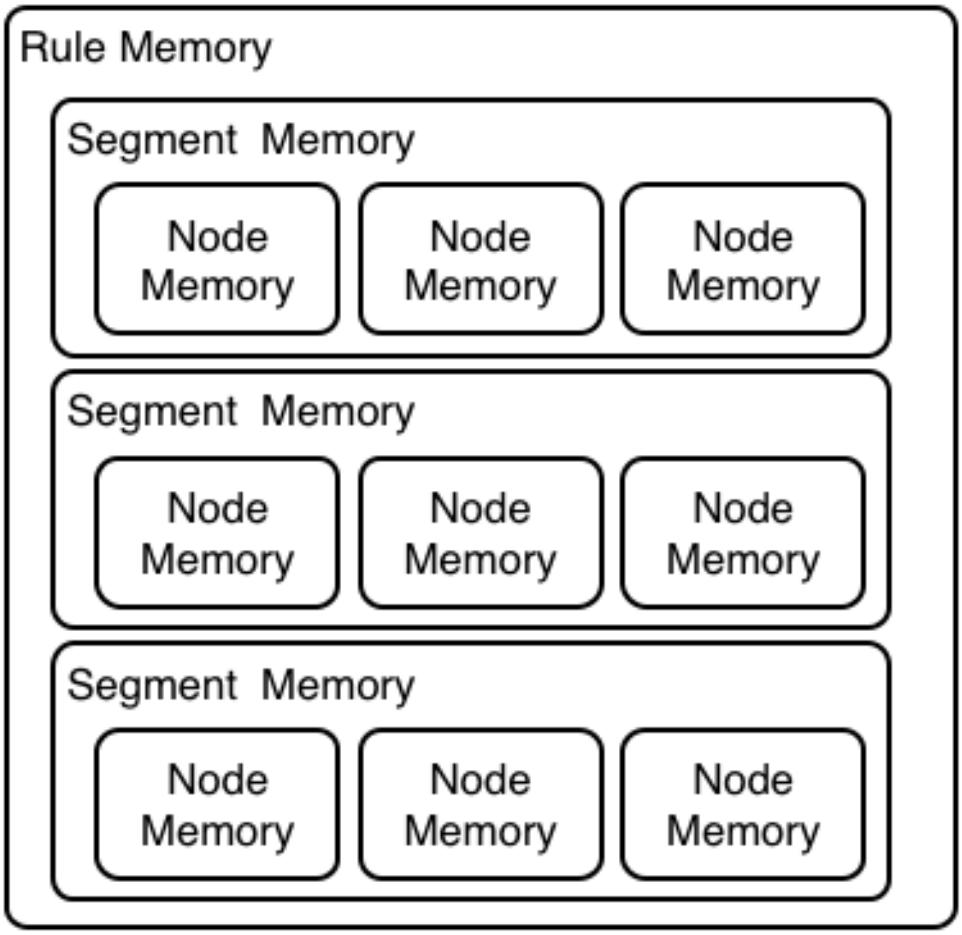


Figure 5.12. PHREAK 3 Layered memory system

Example 1 shows a single rule, with three patterns; A, B and C. It forms a single segment, with bits 1, 2 and 4 for the nodes. The single segment has a bit offset of 1.

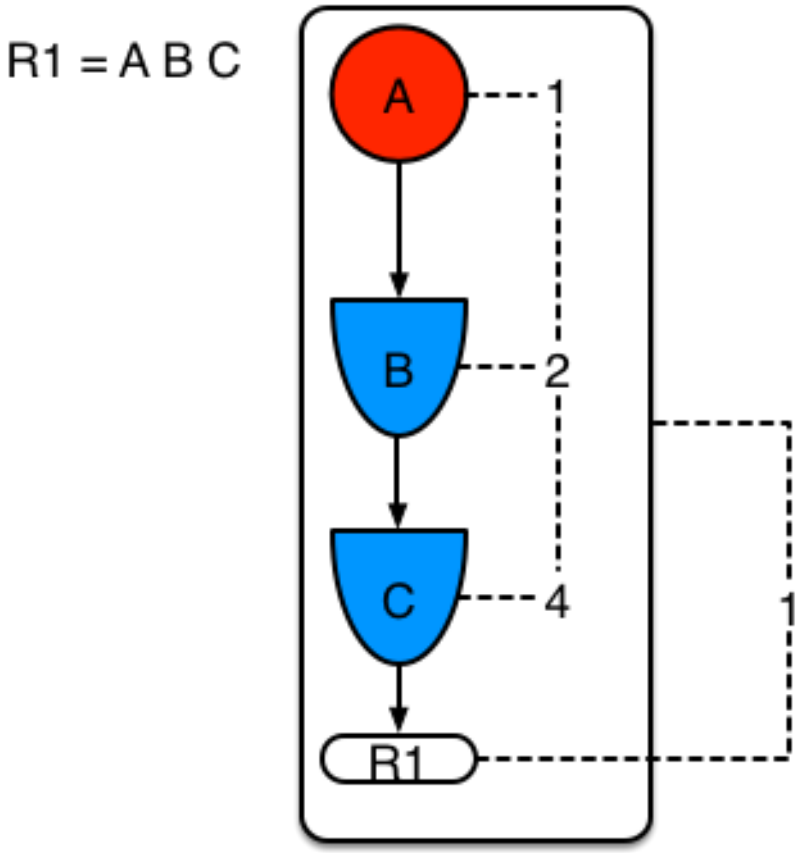


Figure 5.13. Example1: Single rule, no sharing

Example 2 demonstrates what happens when another rule is added that shares the pattern A. A is placed in its own segment, resulting in two segments per rule. Those two segments form a path, for their respective rules. The first segment is shared by both paths. When A is linked the segment becomes linked, it then iterates each path the segment is shared by, setting the bit 1 to on. If B and C are later turned on, the second segment for path R1 is linked in; this causes bit 2 to be turned on for R1. With bit 1 and bit 2 set to on for R1, the rule is now linked and a goal created to schedule the rule for later evaluation and firing.

When a rule is evaluated it is the segments that allow the results of matching to be shared. Each segment has a staging memory to queue all insert, update and deletes for that segment. If R1 was to evaluated it would process A and result in a set of tuples. The algorithm detects that there is a segmentation split and will create peered tuples for each insert, update and delete in the set and add them to R2's staging memory. Those tuples will be merged with any existing staged tuples and wait for R2 to eventually be evaluated.

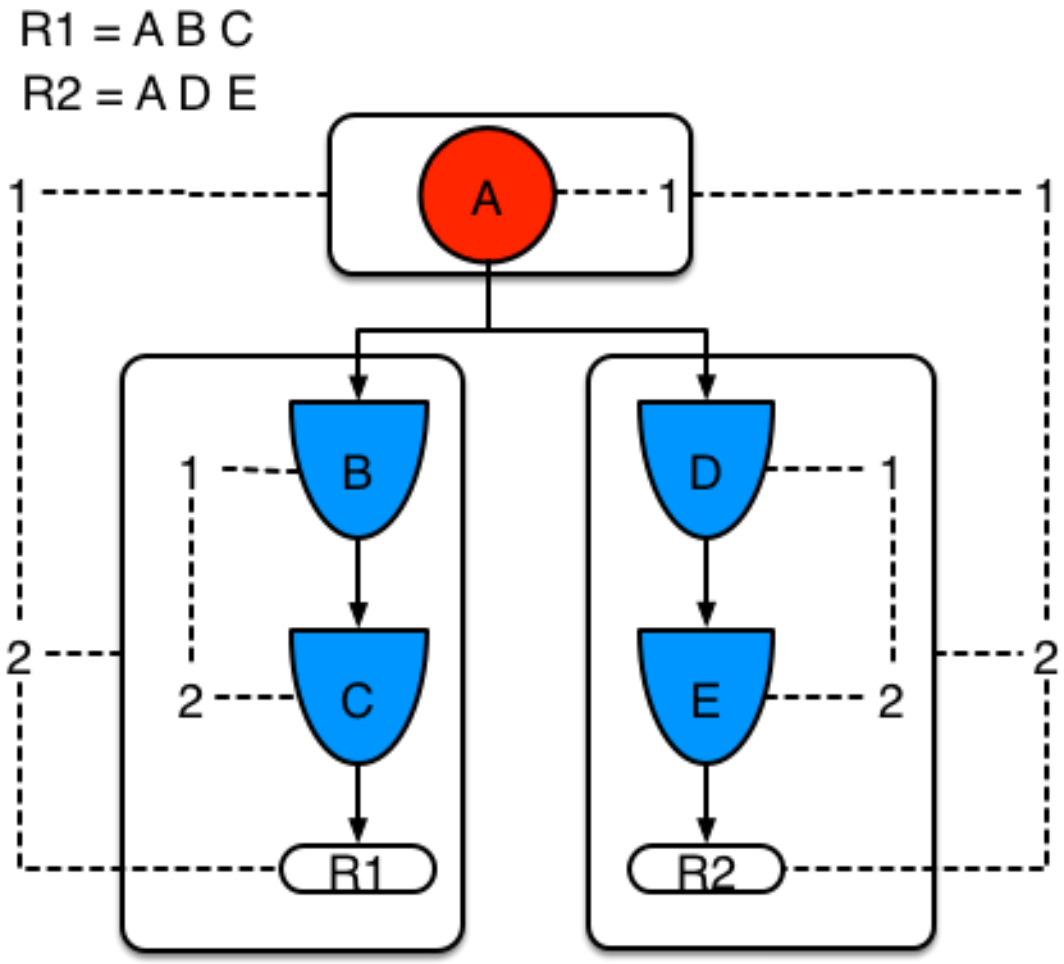


Figure 5.14. Example 2: Two rules, with sharing

Example 3 adds a third rule and demonstrates what happens when A and B are shared. Only the bits for the segments are shown this time. Demonstrating that R4 has 3 segments, R3 has 3 segments and R1 has 2 segments. A and B are shared by R1, R3 and R4. While D is shared by R3 and R4.

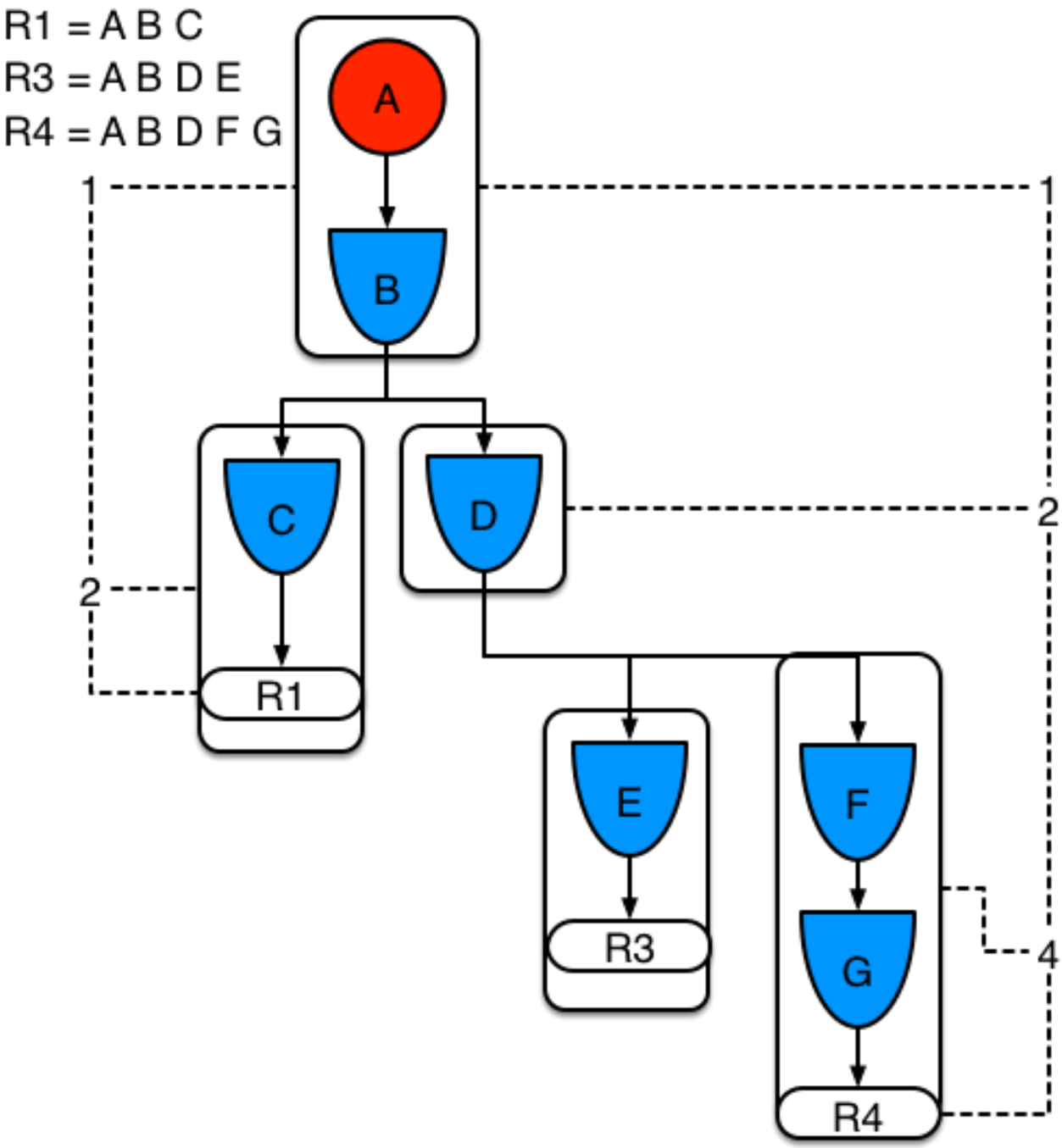


Figure 5.15. Example 3: Three rules, with sharing

Sub-networks are formed when a Not, Exists or Accumulate node contain more than one element. In Example 4 "B not(C)" forms the sub network, note that "not(C)" is a single element and does not require a sub network and is merged inside of the Not node.

The sub network gets its own segment. R1 still has a path of two segments. The sub network forms another "inner" path. When the sub network is linked in, it will link in the outer segment.

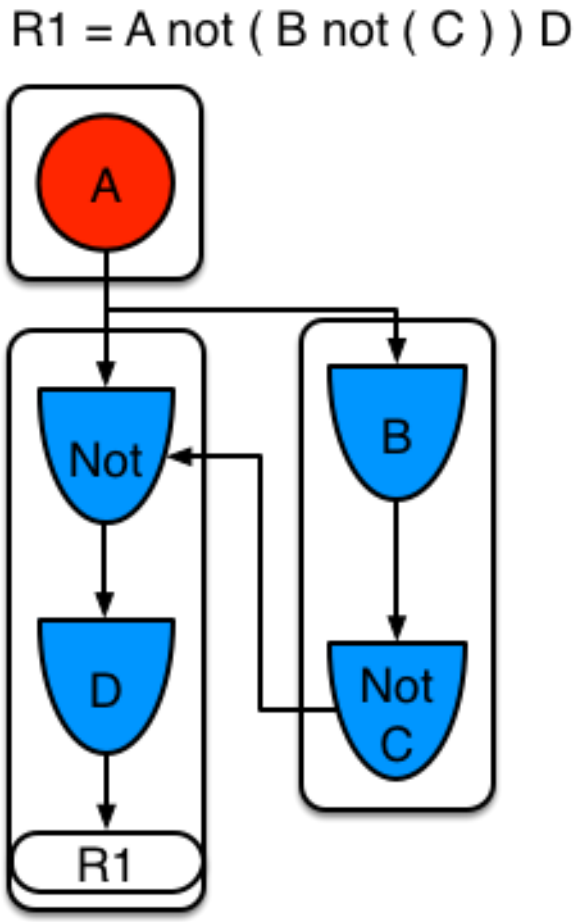


Figure 5.16. Example 4 : Single rule, with sub-network and no sharing

Example 5 shows that the sub-network nodes can be shard by a rule that does not have a sub-network. This results in the sub-network segment being split into two.

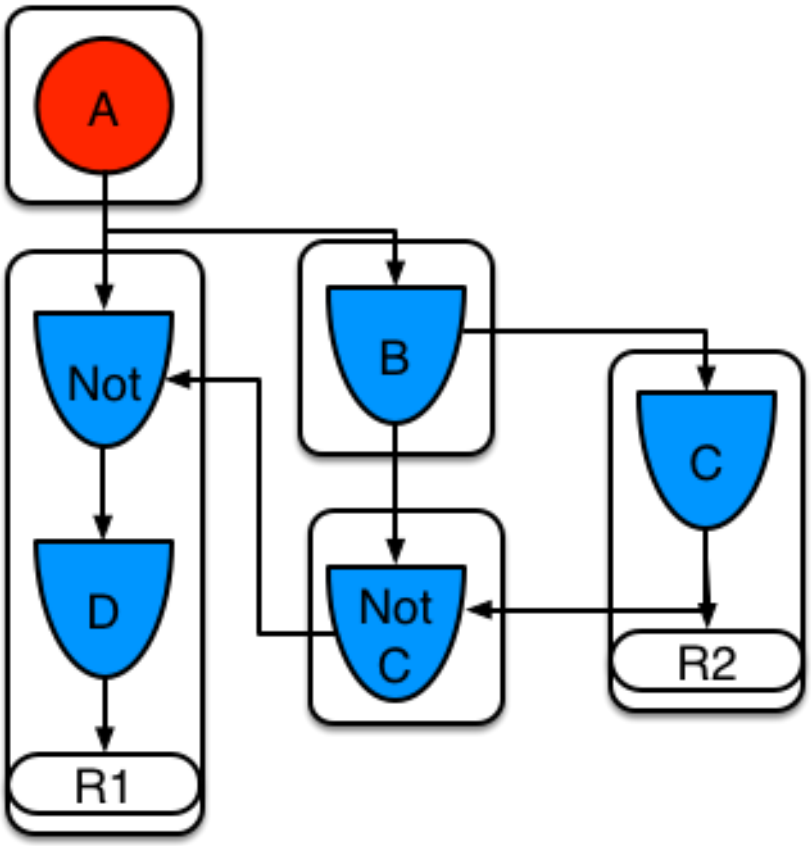


Figure 5.17. Example 5: Two rules, one with a sub-network and sharing

Not nodes with constraints and accumulate nodes have special behaviour and can never unlink a segment, and are always considered to have their bits on.

All rule evaluations are incremental, and will not waste work recomputing matches that it has already produced.

The evaluation algorithm is stack based, instead of method recursion. Evaluation can be paused and resumed at any time, via the use of a StackEntry to represent current node being evaluated.

When a rule evaluation reaches a sub-network a StackEntry is created for the outer path segment and the sub-network segment. The sub-network segment is evaluated first, when the set reaches the end of the sub-network path it is merged into a staging list for the outer node it feeds into. The previous StackEntry is then resumed where it can process the results of the sub network. This has the added benefit that all work is processed in a batch, before propagating to the child node; which is much more efficient for accumulate nodes.

The same stack system can be used for efficient backward chaining. When a rule evaluation reaches a query node it again pauses the current evaluation, by placing it on the stack. The query is then evaluated which produces a result set, which is saved in a memory location for the resumed StackEntry to pick up and propagate to the child node. If the query itself called other queries the

process would repeat, with the current query being paused and a new evaluation setup for the current query node.

One final point on performance. One single rule in general will not evaluate any faster with PHREAK than it does with RETE. For a given rule and same data set, which using a root context object to enable and disable matching, both attempt the same amount of matches and produce the same number of rule instances, and take roughly the same time. Except for the use case with subnetworks and accumulates.

PHREAK can however be considered more forgiving that RETE for poorly written rule bases and with a more graceful degradation of performance as the number of rules and complexity increases.

RETE will also churn away producing partial machines for rules that do not have data in all the joins; where as PHREAK will avoid this.

So it's not that PHREAK is faster than RETE, it just won't slow down as much as your system grows :)

AgendaGroups did not help in RETE performance, as all rules where evaluated at all times, regardless of the group. The same is true for salience. Which is why root context objects are often used, to limit matching attempts. PHREAK only evaluates rules for the active AgendaGroup, and within that group will attempt to avoid evaluation of rules (via salience) that do not result in rule instance firings.

With PHREAK AgendaGroups and salience now become useful performance tools. The root context objects are no longer needed and potentially counter productive to performance, as they force the flushing and recreation of matches for rules.

Chapter 6. User Guide

6.1. The Basics

6.1.1. Stateless Knowledge Session

So where do we get started? There are so many use cases and so much functionality in a rule engine such as Drools that it becomes beguiling. Have no fear my intrepid adventurer, the complexity is layered and you can ease yourself in with simple use cases.

Stateless session, not utilising inference, forms the simplest use case. A stateless session can be called like a function passing it some data and then receiving some results back. Some common use cases for stateless sessions are, but not limited to:

- Validation
  - Is this person eligible for a mortgage?
- Calculation
  - Compute a mortgage premium.
- Routing and Filtering
  - Filter incoming messages, such as emails, into folders.
  - Send incoming messages to a destination.

So let's start with a very simple example using a driving license application.

```
public class Applicant {
    private String name;
    private int age;
    private boolean valid;
    // getter and setter methods here
}
```

Now that we have our data model we can write our first rule. We assume that the application uses rules to reject invalid applications. As this is a simple validation use case we will add a single rule to disqualify any applicant younger than 18.

```
package com.company.licenserule "Is of valid age"when     $a : Applicant( age < 18 )then
 $a.setValid( false );end

com.company.licenserule "Is of valid
```

```
age"
when    $a : Applicant( age < 18
)
then    $a.setValid( false
);
```

To make the engine aware of data, so it can be processed against the rules, we have to *insert* the data, much like with a database. When the Applicant instance is inserted into the engine it is evaluated against the constraints of the rules, in this case just two constraints for one rule. We say *two* because the type Applicant is the first object type constraint, and `age < 18` is the second field constraint. An object type constraint plus its zero or more field constraints is referred to as a pattern. When an inserted instance satisfies both the object type constraint and all the field constraints, it is said to be matched. The `$a` is a binding variable which permits us to reference the matched object in the consequence. There its properties can be updated. The dollar character ('$') is optional, but it helps to differentiate variable names from field names. The process of matching patterns against the inserted data is, not surprisingly, often referred to as *pattern matching*.

To use this rule it is necessary to put it a Drools file, just a plain text file with .drl extension , short for "Drools Rule Language". Let's call this file licenseApplication.drl, and store it in a Kie Project. A Kie Project has the structure of a normal Maven project with an additional file (kmodule.xml) defining the `KieBase`s and `KieSession`s that can be created. This file has to be placed in the resources/META-INF folder of the Maven project while all the other Drools artifacts, such as the licenseApplication.drl containing the former rule, must be stored in the resources folder or in any other subfolder under it.

Since meaningful defaults have been provided for all configuration aspects, the simplest kmodule.xml file can contain just an empty kmodule tag like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"/>
```

At this point it is possible to create a `KieContainer` that reads the files to be built, from the class-path.

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

The above code snippet compiles all the DRL files found on the classpath and put the result of this compilation, a `KieModule`, in the `KieContainer`. If there are no errors, we are now ready to create our session from the `KieContainer` and execute against some data:

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();
Applicant applicant = new Applicant( "Mr John Smith", 16 );
assertTrue( applicant.isValid() );
ksession.execute( applicant );
```

```
assertFalse( applicant.isValid() );
```

The preceding code executes the data against the rules. Since the applicant is under the age of 18, the application is marked as invalid.

So far we've only used a single instance, but what if we want to use more than one? We can execute against any object implementing Iterable, such as a collection. Let's add another class called `Application`, which has the date of the application, and we'll also move the boolean valid field to the `Application` class.

```
public class Applicant {
    private String name;
    private int age;
    // getter and setter methods here
}

public class Application {
    private Date dateApplied;
    private boolean valid;
    // getter and setter methods here
}
```

We will also add another rule to validate that the application was made within a period of time.

```
package com.company.licenserule "Is of valid age"when     Applicant( age < 18 )     $a :
 Application()     then    $a.setValid( false );endrule "Application was made this year"when
  $a : Application( dateApplied > "01-jan-2009" )     then    $a.setValid( false );end

com.company.licenserule "Is of valid
age"
when    Applicant( age < 18
)    $a : Application()

then    $a.setValid( false
);

endrule "Application was made this
year"
when    $a : Application( dateApplied > "01-jan-2009" )

then    $a.setValid( false
);
```

Unfortunately a Java array does not implement the `Iterable` interface, so we have to use the JDK converter method `Arrays.asList(...)`. The code shown below executes against an iterable list, where all collection elements are inserted before any matched rules are fired.

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();
Applicant applicant = new Applicant( "Mr John Smith", 16 );
```

```
Application application = new Application();
assertTrue( application.isValid() );
ksession.execute( Arrays.asList( new Object[] { application, applicant } ) );
assertFalse( application.isValid() );
```

The two execute methods `execute(Object object)` and `execute(Iterable objects)` are actually convenience methods for the interface `BatchExecutor`'s method `execute(Command command)`.

The `KieCommands` commands factory, obtainable from the `KieServices` like all other factories of the KIE API, is used to create commands, so that the following is equivalent to `execute(Iterable it)`:

```
ksession.execute( kieServices.getCommands().newInsertElements( Arrays.asList( new Object[] { ap
plication, applicant } ) );
```

Batch Executor and Command Factory are particularly useful when working with multiple Commands and with output identifiers for obtaining results.

```
KieCommands kieCommands = kieServices.getCommands();
List<Command> cmds = new ArrayList<Command>();
cmds.add( kieCommands.newInsert( new Person( "Mr John Smith" ), "mrSmith", true, null ) );
cmds.add( kieCommands.newInsert( new Person( "Mr John Doe" ), "mrDoe", true, null ) );
BatchExecutionResults results = ksession.execute( kieCommands.newBatchExecution( cmds ) );
assertEquals( new Person( "Mr John Smith" ), results.getValue( "mrSmith" ) );
```

`CommandFactory` supports many other Commands that can be used in the `BatchExecutor` like `StartProcess`, `Query`, and `SetGlobal`.

6.1.2. Stateful Knowledge Session

Stateful Sessions are long lived and allow iterative changes over time. Some common use cases for Stateful Sessions are, but not limited to:

- Monitoring
  - Stock market monitoring and analysis for semi-automatic buying.
- Diagnostics
  - Fault finding, medical diagnostics
- Logistics
  - Parcel tracking and delivery provisioning
- Compliance

- Validation of legality for market trades.

In contrast to a Stateless Session, the `dispose()` method must be called afterwards to ensure there are no memory leaks, as the KieBase contains references to Stateful Knowledge Sessions when they are created. Since Stateful Knowledge Session is the most commonly used session type it is just named `KieSession` in the KIE API. `KieSession` also supports the `BatchExecutor` interface, like `StatelessKieSession`, the only difference being that the `FireAllRules` command is not automatically called at the end for a Stateful Session.

We illustrate the monitoring use case with an example for raising a fire alarm. Using just four classes, we represent rooms in a house, each of which has one sprinkler. If a fire starts in a room, we represent that with a single `Fire` instance.

```
public class Room {
    private String name
    // getter and setter methods here
}
public class Sprinkler {
    private Room room;
    private boolean on;
    // getter and setter methods here
}
public class Fire {
    private Room room;
    // getter and setter methods here
}
public class Alarm {
}
```

In the previous section on Stateless Sessions the concepts of inserting and matching against data were introduced. That example assumed that only a single instance of each object type was ever inserted and thus only used literal constraints. However, a house has many rooms, so rules must express relationships between objects, such as a sprinkler being in a certain room. This is best done by using a binding variable as a constraint in a pattern. This "join" process results in what is called cross products, which are covered in the next section.

When a fire occurs an instance of the `Fire` class is created, for that room, and inserted into the session. The rule uses a binding on the `room` field of the `Fire` object to constrain matching to the sprinkler for that room, which is currently off. When this rule fires and the consequence is executed the sprinkler is turned on.

```
rule "When there is a fire turn on the sprinkler"when    Fire($room : room)    $sprinkler :
 Sprinkler( room == $room, on == false )then    modify( $sprinkler ) { setOn( true ) };
 System.out.println( "Turn on the sprinkler for room " + $room.getName() );end
kler"
when    Fire($room :
room)    $sprinkler : Sprinkler( room == $room, on == false
)
then    modify( $sprinkler ) { setOn( true )
```

```
};    System.out.println( "Turn on the sprinkler for room " + $room.getName()
);
```

Whereas the Stateless Session uses standard Java syntax to modify a field, in the above rule we use the `modify` statement, which acts as a sort of "with" statement. It may contain a series of comma separated Java expressions, i.e., calls to setters of the object selected by the `modify` statement's control expression. This modifies the data, and makes the engine aware of those changes so it can reason over them once more. This process is called inference, and it's essential for the working of a Stateful Session. Stateless Sessions typically do not use inference, so the engine does not need to be aware of changes to data. Inference can also be turned off explicitly by using the *sequential mode*.

So far we have rules that tell us when matching data exists, but what about when it does *not* exist? How do we determine that a fire has been extinguished, i.e., that there isn't a `Fire` object any more? Previously the constraints have been sentences according to Propositional Logic, where the engine is constraining against individual instances. Drools also has support for First Order Logic that allows you to look at sets of data. A pattern under the keyword `not` matches when something does not exist. The rule given below turns the sprinkler off as soon as the fire in that room has disappeared.

```
rule "When the fire is gone turn off the sprinkler"when     $room : Room( )     $sprinkler :
 Sprinkler( room == $room, on == true )    not Fire( room == $room )then    modify( $sprinkler )
  {  setOn(  false  )  };       System.out.println(  "Turn  off  the  sprinkler  for  room  "  +
 $room.getName() );end
kler"
when    $room : Room(
)    $sprinkler : Sprinkler( room == $room, on == true
)    not Fire( room == $room
)
then    modify( $sprinkler ) { setOn( false )
};    System.out.println( "Turn off the sprinkler for room " + $room.getName()
);
```

While there is one sprinkler per room, there is just a single alarm for the building. An `Alarm` object is created when a fire occurs, but only one `Alarm` is needed for the entire building, no matter how many fires occur. Previously `not` was introduced to match the absence of a fact; now we use its complement `exists` which matches for one or more instances of some category.

```
rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end
```

Likewise, when there are no fires we want to remove the alarm, so the `not` keyword can be used again.

```
rule "Cancel the alarm when all the fires have gone"when    not Fire()    $alarm : Alarm()then    delete( $alarm );    System.out.println( "Cancel the alarm" );end
gone"
when    not
Fire()    $alarm :
Alarm()
then    delete( $alarm
);    System.out.println( "Cancel the alarm"
);
```

Finally there is a general health status message that is printed when the application first starts and after the alarm is removed and all sprinklers have been turned off.

```
rule "Status output when things are ok"when    not Alarm()    not Sprinkler( on == true ) then    System.out.println( "Everything is ok" );end

ok"when    not
 Alarm()    not Sprinkler( on == true
)
 then    System.out.println( "Everything is ok"
```

As we did in the Stateless Session example, the above rules should be placed in a single DRL file and saved into the resouces folder of your Maven project or any of its subfolder. As before, we can then obtain a `KieSession` from the `KieContainer`. The only difference is that this time we create a Stateful Session, whereas before we created a Stateless Session.

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
KieSession ksession = kContainer.newKieSession();
```

With the session created it is now possible to iteratively work with it over time. Four `Room` objects are created and inserted, as well as one `Sprinkler` object for each room. At this point the engine has done all of its matching, but no rules have fired yet. Calling `ksession.fireAllRules()` allows the matched rules to fire, but without a fire that will just produce the health message.

```
String[] names = new String[]{"kitchen", "bedroom", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();
for( String name: names ){
    Room room = new Room( name );
    name2room.put( name, room );
    ksession.insert( room );
    Sprinkler sprinkler = new Sprinkler( room );
```

```
    ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

```
> Everything is ok
```

We now create two fires and insert them; this time a reference is kept for the returned `FactHandle`. A Fact Handle is an internal engine reference to the inserted instance and allows instances to be retracted or modified at a later point in time. With the fires now in the engine, once `fireAllRules()` is called, the alarm is raised and the respective sprinklers are turned on.

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

```
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

After a while the fires will be put out and the `Fire` instances are retracted. This results in the sprinklers being turned off, the alarm being cancelled, and eventually the health message is printed again.

```
ksession.delete( kitchenFireHandle );
ksession.delete( officeFireHandle );

ksession.fireAllRules();
```

```
> Cancel the alarm> Turn off the sprinkler for room office> Turn off the sprinkler for room
 kitchen> Everything is ok
alarm> Turn off the sprinkler for room
office> Turn off the sprinkler for room
kitchen> Everything is
```

Everyone still with me? That wasn't so hard and already I'm hoping you can start to see the value and power of a declarative rule system.

6.1.3. Methods versus Rules

People often confuse methods and rules, and new rule users often ask, "How do I call a rule?" After the last section, you are now feeling like a rule expert and the answer to that is obvious, but let's summarize the differences nonetheless.

```
public void helloWorld(Person person) {
    if ( person.getName().equals( "Chuck" ) ) {
        System.out.println( "Hello Chuck" );
    }
}
```

- Methods are called directly.
- Specific instances are passed.
- One call results in a single execution.

```
rule "Hello World" when   Person( name == "Chuck" )then   System.out.println( "Hello Chuck" );end
when    Person( name == "Chuck"
)
then    System.out.println( "Hello Chuck"
);
```

- Rules execute by matching against any data as long it is inserted into the engine.
- Rules can never be called directly.
- Specific instances cannot be passed to a rule.
- Depending on the matches, a rule may fire once or several times, or not at all.

6.1.4. Cross Products

Earlier the term "cross product" was mentioned, which is the result of a join. Imagine for a moment that the data from the fire alarm example were used in combination with the following rule where there are no field constraints:

```
rule  "Show  Sprinklers"  when       $room  :  Room()       $sprinkler  :  Sprinkler()then
 System.out.println( "room:" + $room.getName() +                              " sprinkler:" +
 $sprinkler.getRoom().getName() );end
when    $room :
Room()    $sprinkler :
Sprinkler()
then    System.out.println( "room:" + $room.getName()
+                       " sprinkler:" + $sprinkler.getRoom().getName()
);
```

In SQL terms this would be like doing `select * from Room, Sprinkler` and every row in the Room table would be joined with every row in the Sprinkler table resulting in the following output:

```
room:office sprinkler:office
room:office sprinkler:kitchen
room:office sprinkler:livingroom
room:office sprinkler:bedroom
room:kitchen sprinkler:office
room:kitchen sprinkler:kitchen
room:kitchen sprinkler:livingroom
room:kitchen sprinkler:bedroom
room:livingroom sprinkler:office
room:livingroom sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:livingroom sprinkler:bedroom
room:bedroom sprinkler:office
room:bedroom sprinkler:kitchen
room:bedroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

These cross products can obviously become huge, and they may very well contain spurious data. The size of cross products is often the source of performance problems for new rule authors. From this it can be seen that it's always desirable to constrain the cross products, which is done with the variable constraint.

```
rule
when
    $room : Room()
    $sprinkler : Sprinkler( room == $room )
then
    System.out.println( "room:" + $room.getName() +
                        " sprinkler:" + $sprinkler.getRoom().getName() );
end
```

This results in just four rows of data, with the correct Sprinkler for each Room. In SQL (actually HQL) the corresponding query would be `select * from Room, Sprinkler where Room == Sprinkler.room`.

```
room:office sprinkler:office
room:kitchen sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

6.2. Execution Control

6.2.1. Agenda

The Agenda is a *Rete* feature. It maintains set of rules that are able to execute, its job is to schedule that execution in a deterministic order.

During actions on the `RuleRuntime`, rules may become fully matched and eligible for execution; a single Rule Runtime Action can result in multiple eligible rules. When a rule is fully matched a Rule Match is created, referencing the rule and the matched facts, and placed onto the Agenda. The Agenda controls the execution order of these Matches using a Conflict Resolution strategy.

The engine cycles repeatedly through two phases:

1. Rule Runtime Actions. This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls `fireAllRules()` the engine switches to the Agenda Evaluation phase.
2. Agenda Evaluation. This attempts to select a rule to fire. If no rule is found it exits, otherwise it fires the found rule, switching the phase back to Rule Runtime Actions.

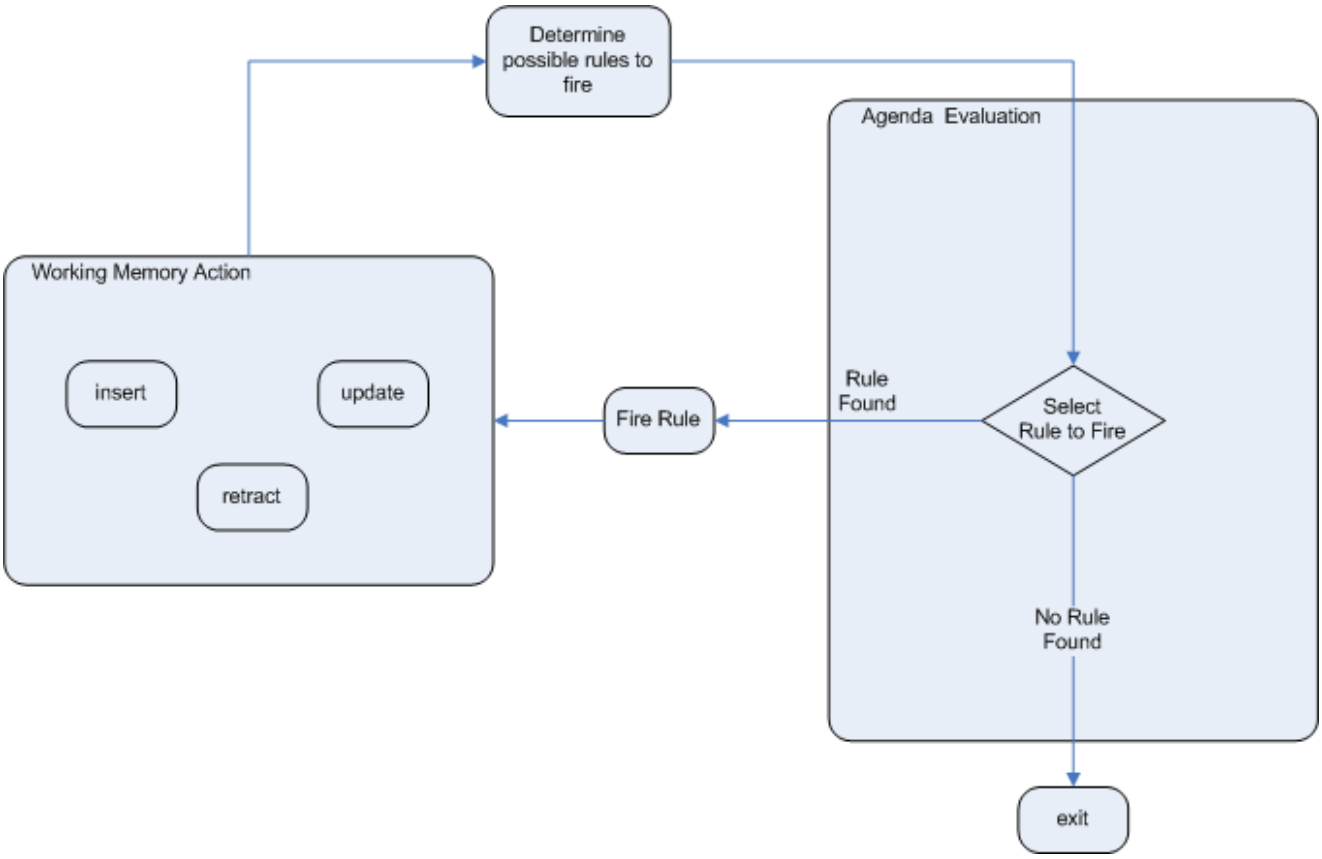


Figure 6.1. Two Phase Execution

The process repeats until the agenda is clear, in which case control returns to the calling application. When Rule Runtime Actions are taking place, no rules are being fired.

6.2.2. Rule Matches and Conflict Sets.

6.2.2.1. Cashflow Example

So far the data and the matching process has been simple and small. To mix things up a bit a new example will be explored that handles cashflow calculations over date periods. The state of the engine will be illustratively shown at key stages to help get a better understanding of what is actually going on under the hood. Three classes will be used, as shown below. This will help us grow our understanding of pattern matching and joins further. We will then use this to illustate different techniques for execution control.

```
public class CashFlow {
    private Date   date;
    private double amount;
    private int    type;
    long           accountNo;
    // getter and setter methods here
}

public class Account {
    private long   accountNo;
    private double balance;
    // getter and setter methods here
}

public AccountPeriod {
    private Date start;
    private Date end;
    // getter and setter methods here
}
```

By now you already know how to create KieBases and how to instantiate facts to populate the `KieSession`, so tables will be used to show the state of the inserted data, as it makes things clearer for illustration purposes. The tables below show that a single fact was inserted for the `Account`. Also inserted are a series of debits and credits as `CashFlow` objects for that account, extending over two quarters.

CashFlow			
date	amount	type	accountNo
12-Jan-07	100	CREDIT	1
2-Feb-07	200	DEBIT	1
18-May-07	50	CREDIT	1
9-Mar-07	75	CREDIT	1

Account	
accountNo	balance
1	0

Figure 6.2. CashFlows and Account

Two rules can be used to determine the debit and credit for that quarter and update the Account balance. The two rules below constrain the cashflows for an account for a given time period. Notice the "&&" which use short cut syntax to avoid repeating the field name twice.

```
rule "increase balance for credits"when  ap :
 AccountPeriod()  acc : Account( $accountNo :
 accountNo )  CashFlow( type == CREDIT,
         accountNo == $accountNo,
   date >= ap.start && <= ap.end,
   $amount : amount )then  acc.balance  +=
 $amount;end

credits"when  ap
: AccountPeriod()  acc : Account( $accountNo
accountNo )  CashFlow( type
== CREDIT,            accountNo
== $accountNo,            date >= ap.start &&
<= ap.end,            $amount :
amount
 )then  acc.balance
+=
```

```
rule "decrease balance for deb
its" when   ap : AccountPeriod()   acc :
 Account( $accountNo : accountNo )
 CashFlow( type == DEBIT,
 accountNo == $accountNo,            date >=
 ap.start && <= ap.end,             $amount :
 amount ) then   acc.balance -= $amount; end
debits"
 when   ap :
 AccountPeriod()   acc : Account( $accountNo : accountNo
 )   CashFlow( type ==
 DEBIT,             accountNo
== $accountNo,            date >= ap.start && <=
 ap.end,             $amount : amount
 )
 then   acc.balance -=
```

Earlier we showed how rules would equate to SQL, which can often help people with an SQL background to understand rules. The two rules above can be represented with two views and a trigger for each view, as below:

Table 6.1.

```
select * from Account acc,
 Cashflow cf,              AccountPeriod ap
     where acc.accountNo == cf.accountNo and
      cf.type == CREDIT and      cf.date >=
 ap.start and       cf.date <= ap.end
 acc,              Cashflow
 cf,              AccountPeriod ap
  where acc.accountNo == cf.accountNo
and       cf.type == CREDIT
 and      cf.date >= ap.start
and       cf.date <=
```

```
select * from Account acc,
 Cashflow cf,              AccountPeriod
 ap where acc.accountNo == cf.accountNo and
      cf.type == DEBIT and      cf.date >=
 ap.start and       cf.date <= ap.end
acc,               Cashflow
 cf,              AccountPeriod
ap where acc.accountNo == cf.accountNo
and       cf.type == DEBIT
 and      cf.date >= ap.start
and       cf.date <=
```

```
trigger : acc.balance += cf.amount
```

```
trigger : acc.balance -= cf.amount
```

If the `AccountPeriod` is set to the first quarter we constrain the rule "increase balance for credits" to fire on two rows of data and "decrease balance for debits" to act on one row of data.

AccountingPeriod	
start	end
01-Jan-07	31-Mar-07

CashFlow		
date	amount	type
12-Jan-07	100	CREDIT
9-Mar-07	75	CREDIT

CashFlow		
date	amount	type
2-Feb-07	200	DEBIT

Figure 6.3. AccountingPeriod, CashFlows and Account

The two cashflow tables above represent the matched data for the two rules. The data is matched during the insertion stage and, as you discovered in the previous chapter, does not fire straight away, but only after `fireAllRules()` is called. Meanwhile, the rule plus its matched data is placed on the Agenda and referred to as an Rule Match or Rule Instance. The Agenda is a table of Rule Matches that are able to fire and have their consequences executed, as soon as fireAllRules() is called. Rule Matches on the Agenda are referred to as a *conflict set* and their execution is determine by a conflict resolution strategy. Notice that the order of execution so far is considered arbitrary.

Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	

Figure 6.4. CashFlows and Account

After all of the above activations are fired, the account has a balance of -25.

Account	
accountNo	balance
1	-25

Figure 6.5. CashFlows and Account

If the `AccountPeriod` is updated to the second quarter, we have just a single matched row of data, and thus just a single Rule Match on the Agenda.

The firing of that Activation results in a balance of 25.

AccountingPeriod	
start	end
01-Apr-07	30-Jun-07

CashFlow		
date	amount	type
18-May-07	50	CREDIT

Figure 6.6. CashFlows and Account

accountNo	balance
1	25

Figure 6.7. CashFlows and Account

6.2.2.2. Conflict Resolution

What if you don't want the order of rule execution to be arbitrary? When there is one or more Rule Match on the Agenda they are said to be in conflict, and a conflict resolution strategy is used to determine the order of execution. The Drools strategy is very simple and based around a salience value, which assigns a priority to a rule. Each rule has a default value of 0, the higher the value the higher the priority.

As a general rule, it is a good idea not to count on rules firing in any particular order, and to author the rules without worrying about a "flow". However when a flow is needed a number of possibilities exist beyond salience: agenda groups, rule flow groups, activation groups and control/semaphore facts.

As of Drools 6.0 rule definition order in the source file is used to set priority after salience.

6.2.2.3. Salience

To illustrate Salience we add a rule to print the account balance, where we want this rule to be executed after all the debits and credits have been applied for all accounts. We achieve this by assigning a negative salience to this rule so that it fires after all rules with the default salience 0.

Table 6.2.

```
rule "Print balance for AccountPeriod"       salience -50    when       ap : AccountPeriod()
         acc : Account()              then          System.out.println( acc.accountNo + " :
  " + acc.balance );    end
```

```
AccountPeriod"
  salience -50
    when
ap : AccountPeriod()        acc : Account()

    then        System.out.println( acc.accountNo + " : " + acc.balance
);
```

The table below depicts the resulting Agenda. The three debit and credit rules are shown to be in arbitrary order, while the print rule is ranked last, to execute afterwards.

Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	
4	print balance	

Figure 6.8. CashFlows and Account

6.2.2.4. Agenda Groups

Agenda groups allow you to place rules into groups, and to place those groups onto a stack. The stack has push/pop bevaviour. Calling "setFocus" places the group onto the stack:

```
ksession.getAgenda().getAgendaGroup( "Group A" ).setFocus();
```

The agenda always evaluates the top of the stack. When all the rules have fired for a group, it is poped from the stack and the next group is evaluated.

Table 6.3.

```
rule "increase balance for credits"  agenda-
group "calculation"when  ap : AccountPeriod()
  acc : Account( $accountNo : accountNo )
  CashFlow( type == CREDIT,
 accountNo == $accountNo,            date >=
 ap.start && <= ap.end,            $amount :
 amount )then  acc.balance  += $amount;end
 credits"  agenda-
group
 "calculation"when  ap
: AccountPeriod()  acc : Account( $accountNo
accountNo )  CashFlow( type
== CREDIT,            accountNo
== $accountNo,            date >= ap.start &&
<= ap.end,            $amount :
amount
 )then  acc.balance
```

```
rule "Print balance for AccountPeri
od"  agenda-group "report"when  ap :
 AccountPeriod()  acc : Account()then
 System.out.println( acc.accountNo +
             " : " + acc.balance );    end
AccountPeriod"

agenda-group "report"when
ap : AccountPeriod()
acc
 : Account()then
System.out.println( acc.accountNo +                    " : "
+ acc.balance
);
```

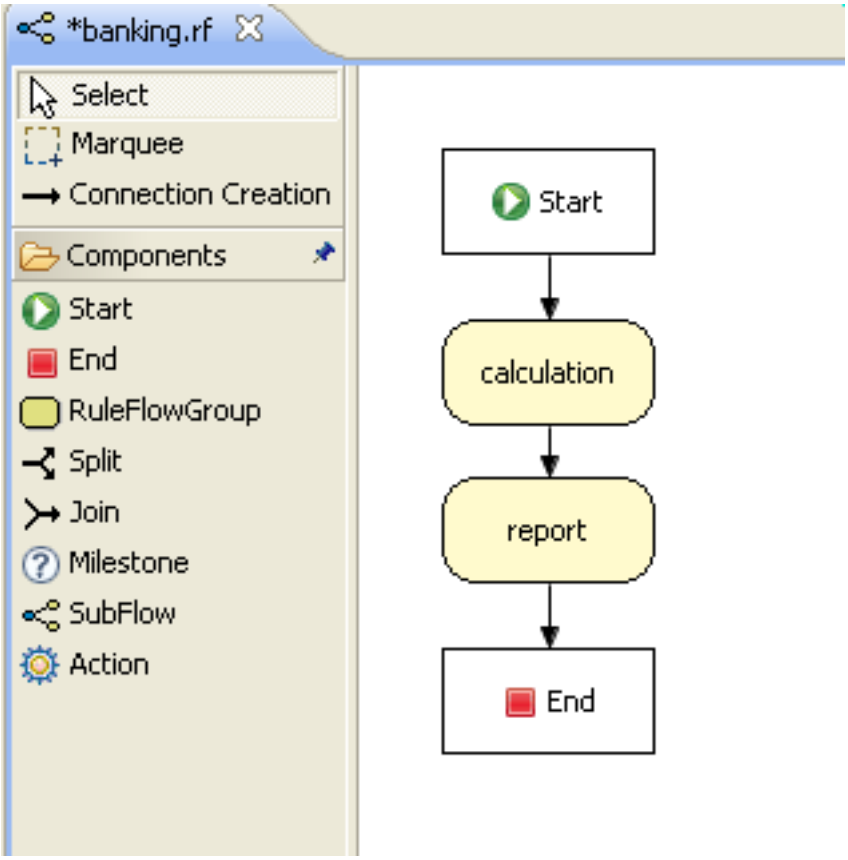
```
+=
```

First set the focus to the "report" group and then by placing the focus on "calculation" we ensure that group is evaluated first.

```
Agenda agenda = ksession.getAgenda();
agenda.getAgendaGroup( "report" ).setFocus();
agenda.getAgendaGroup( "calculation" ).setFocus();
ksession.fireAllRules();
```

6.2.2.5. Rule Flow

Drools also features ruleflow-group attributes which allows workflow diagrams to declaratively specify when rules are allowed to fire. The screenshot below is taken from Eclipse using the Drools plugin. It has two ruleflow-group nodes which ensures that the calculation rules are executed before the reporting rules.



The use of the ruleflow-group attribute in a rule is shown below.

Table 6.4.

`rule "increase balance for credits" ruleflow-group "calculation"when ap :`	`rule "Print balance for AccountPeri od" ruleflow-group "report"when ap :`

```
 AccountPeriod()  acc : Account( $accountNo :
 accountNo )  CashFlow( type == CREDIT,
         accountNo == $accountNo,
   date >= ap.start && <= ap.end,
   $amount : amount )then  acc.balance  +=
 $amount;end
 credits"  ruleflow-
group
 "calculation"when  ap
: AccountPeriod()  acc : Account( $accountNo
accountNo )  CashFlow( type
== CREDIT,            accountNo
== $accountNo,            date >= ap.start &&
<= ap.end,            $amount :
amount
 )then  acc.balance
+=
```

```
 AccountPeriod()  acc : Account()then
 System.out.println( acc.accountNo +
             " : " + acc.balance );    end
AccountPeriod"

ruleflow-group "report"when
ap : AccountPeriod()
acc
 : Account()then
System.out.println( acc.accountNo +                    " : "
+ acc.balance
);
```

6.2.3. Declarative Agenda

Warning

Declarative Agenda is experimental, and all aspects are highly likely to change in the future. @Eager and @Direct are temporary annotations to control the behaviour of rules, which will also change as Declarative Agenda evolves. Annotations instead of attributes where chosen, to reflect their experimental nature.

The declarative agenda allows to use rules to control which other rules can fire and when. While this will add a lot more overhead than the simple use of salience, the advantage is it is declarative and thus more readable and maintainable and should allow more use cases to be achieved in a simpler fashion.

This feature is off by default and must be explicitly enabled, that is because it is considered highly experimental for the moment and will be subject to change, but can be activated on a given KieBase by adding the declarativeAgenda='enabled' attribute in the corresponding kbase tag of the kmodule.xml file as in the following example.

Example 6.1. Enabling the Declarative Agenda

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="DeclarativeKBase" declarativeAgenda="enabled">
    <ksession name="KSession">
  </kbase>
</kmodule>
```

The basic idea is:

- All rule's Matches are inserted into WorkingMemory as facts. So you can now do pattern matching against a Match. The rule's metadata and declarations are available as fields on the Match object.
- You can use the kcontext.blockMatch(Match match) for the current rule to block the selected match. Only when that rule becomes false will the match be eligible for firing. If it is already eligible for firing and is later blocked, it will be removed from the agenda until it is unblocked.
- A match may have multiple blockers and a count is kept. All blockers must became false for the counter to reach zero to enable the Match to be eligible for firing.
- kcontext.unblockAllMatches(Match match) is an over-ride rule that will remove all blockers regardless
- An activation may also be cancelled, so it never fires with cancelMatch
- An unblocked Match is added to the Agenda and obeys normal salience, agenda groups, ruleflow groups etc.
- The @Direct annotations allows a rule to fire as soon as it's matched, this is to be used for rules that block/unblock matches, it is not desirable for these rules to have side effects that impact else where.

Example 6.2. New RuleContext methods

```
void blockMatch(Match match);
void unblockAllMatches(Match match);
void cancelMatch(Match match);
```

Here is a basic example that will block all matches from rules that have metadata @department('sales'). They will stay blocked until the blockerAllSalesRules rule becomes false, i.e. "go2" is retracted.

Example 6.3. Block rules based on rule metadata

```
rule rule1 @Eager @department('sales') when $s : String( this == 'go1' )then
 list.add( kcontext.rule.name + ':' + $s );endrule rule2 @Eager @department('sales') when
   $s : String( this == 'go1' )then list.add( kcontext.rule.name + ':' + $s );endrule
blockerAllSalesRules @Direct @Eager when $s : String( this == 'go2' ) $i : Match( department
 == 'sales' )then list.add( $i.rule.name + ':' + $s ); kcontext.blockMatch( $i );end
when    $s : String( this == 'go1'
)
then    list.add( kcontext.rule.name + ':' + $s
);
endrule rule2 @Eager @department('sales')
when    $s : String( this == 'go1'
)
then    list.add( kcontext.rule.name + ':' + $s
);
```

```
endrule blockerAllSalesRules @Direct @Eager
when    $s : String( this == 'go2'
)    $i : Match( department == 'sales'
)
then    list.add( $i.rule.name + ':' + $s
);    kcontext.blockMatch( $i
);
```

Warning

Further than annotate the blocking rule with @Direct, it is also necessary to annotate all the rules that could be potentially blocked by it with @Eager. This is because, since the Match has to be evaluated by the pattern matching of the blocking rule, the potentially blocked ones cannot be evaluated lazily, otherwise won't be any Match to be evaluated.

This example shows how you can use active property to count the number of active or inactive (already fired) matches.

Example 6.4. Count the number of active/inactive Matches

```
rule  rule1  @Eager  @department('sales')  when       $s  :  String(  this  ==  'go1'  )then
 list.add( kcontext.rule.name + ':' + $s );endrule rule2 @Eager @department('sales') when    $s :
 String( this == 'go1' )then    list.add( kcontext.rule.name + ':' + $s );endrule rule3 @Eager
 @department('sales') when    $s : String( this == 'go1' )then    list.add( kcontext.rule.name
 + ':' + $s );endrule countActivateInActive @Direct @Eager when    $s : String( this == 'go2' )
    $active : Number( this == 1 ) from accumulate( $a : Match( department == 'sales', active ==
 true ), count( $a ) )    $inActive : Number( this == 2 ) from  accumulate( $a : Match( department
 == 'sales', active == false ), count( $a ) )then    kcontext.halt( );end
when    $s : String( this == 'go1'
)
then    list.add( kcontext.rule.name + ':' + $s
);
endrule rule2 @Eager @department('sales')
when    $s : String( this == 'go1'
)
then    list.add( kcontext.rule.name + ':' + $s
);
endrule rule3 @Eager @department('sales')
when    $s : String( this == 'go1'
)
then    list.add( kcontext.rule.name + ':' + $s
);
endrule countActivateInActive @Direct @Eager
when    $s : String( this == 'go2'
)    $active : Number( this == 1 ) from accumulate( $a : Match( department == 'sales',
active == true ), count( $a )
)    $inActive : Number( this == 2 ) from  accumulate( $a : Match( department == 'sales',
active == false ), count( $a )
)
then    kcontext.halt(
```

```
);end
```

6.3. Inference

6.3.1. Bus Pass Example

Inference has a bad name these days, as something not relevant to business use cases and just too complicated to be useful. It is true that contrived and complicated examples occur with inference, but that should not detract from the fact that simple and useful ones exist too. But more than this, correct use of inference can crate more agile and less error prone business rules, which are easier to maintain.

So what is inference? Something is inferred when we gain knowledge of something from using previous knowledge. For example, given a Person fact with an age field and a rule that provides age policy control, we can infer whether a Person is an adult or a child and act on this.

```
rule "Infer Adult"when  $p : Person( age >= 18 )then  insert( new IsAdult( $p ) )end
Adult"
when  $p : Person( age >= 18
)
then  insert( new IsAdult( $p )
)
```

Due to the preceding rule, every Person who is 18 or over will have an instance of IsAdult inserted for them. This fact is special in that it is known as a relation. We can use this inferred relation in any rule:

```
$p : Person()IsAdult( person == $p )
son()IsAdult( person == $p
```

So now we know what inference is, and have a basic example, how does this facilitate good rule design and maintenance?

Let's take a government department that are responsible for issuing ID cards when children become adults, henceforth referred to as ID department. They might have a decision table that includes logic like this, which says when an adult living in London is 18 or over, issue the card:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person		
	location	age >= $1	issueIdCard($1)
	Select Person	Select Adults	Issue ID Card
Issue ID Card to Adults	London	18	p

However the ID department does not set the policy on who an adult is. That's done at a central government level. If the central government were to change that age to 21, this would initiate a change management process. Someone would have to liaise with the ID department and make sure their systems are updated, in time for the law going live.

This change management process and communication between departments is not ideal for an agile environment, and change becomes costly and error prone. Also the card department is managing more information than it needs to be aware of with its "monolithic" approach to rules management which is "leaking" information better placed elsewhere. By this I mean that it doesn't care what explicit "age >= 18" information determines whether someone is an adult, only that they are an adult.

In contrast to this, let's pursue an approach where we split (de-couple) the authoring responsibilities, so that both the central government and the ID department maintain their own rules.

It's the central government's job to determine who is an adult. If they change the law they just update their central repository with the new rules, which others use:

	RuleTable Age Policy	
	CONDITION	ACTION
	p : Person	
	age >= $1	insert($1)
	Adult Age Policy	Add Adult Relation
Infer Adult	18	new IsAdult(p)

The IsAdult fact, as discussed previously, is inferred from the policy rules. It encapsulates the seemingly arbitrary piece of logic "age >= 18" and provides semantic abstractions for its meaning. Now if anyone uses the above rules, they no longer need to be aware of explicit information that determines whether someone is an adult or not. They can just use the inferred fact:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person	IsAdult	
	location	person == $1	issueIdCard($1)
	Select Person	**Select Adults**	**Issue ID Card**
Issue ID Card to Adults	**London**	**p**	**p**

While the example is very minimal and trivial it illustrates some important points. We started with a monolithic and leaky approach to our knowledge engineering. We created a single decision table that had all possible information in it and that leaks information from central government that the ID department did not care about and did not want to manage.

We first de-coupled the knowledge process so each department was responsible for only what it needed to know. We then encapsulated this leaky knowledge using an inferred fact IsAdult. The use of the term IsAdult also gave a semantic abstraction to the previously arbitrary logic "age >= 18".

So a general rule of thumb when doing your knowledge engineering is:

- **Bad**
  - Monolithic
  - Leaky
- **Good**
  - De-couple knowledge responsibilities
  - Encapsulate knowledge
  - Provide semantic abstractions for those encapsulations

6.4. Truth Maintenance with Logical Objects

6.4.1. Overview

After regular inserts you have to retract facts explicitly. With *logical* assertions, the fact that was asserted will be automatically retracted when the conditions that asserted it in the first place are no longer true. Actually, it's even cleverer then that, because it will be retracted only if there isn't any single condition that supports the logical assertion.

Normal insertions are said to be *stated*, i.e., just like the intuitive meaning of "stating a fact" implies. Using a `HashMap` and a counter, we track how many times a particular equality is *stated*; this means we count how many different instances are equal.

When we *logically* insert an object during a RHS execution we are said to *justify* it, and it is considered to be justified by the firing rule. For each logical insertion there can only be one equal object, and each subsequent equal logical insertion increases the justification counter for this logical assertion. A justification is removed by the LHS of the creating rule becoming untrue, and the counter is decreased accordingly. As soon as we have no more justifications the logical object is automatically retracted.

If we try to *logically* insert an object when there is an equal *stated* object, this will fail and return null. If we *state* an object that has an existing equal object that is *justified* we override the Fact; how this override works depends on the configuration setting `WM_BEHAVIOR_PRESERVE`. When the property is set to discard we use the existing handle and replace the existing instance with the new Object, which is the default behavior; otherwise we override it to *stated* but we create an new `FactHandle`.

This can be confusing on a first read, so hopefully the flow charts below help. When it says that it returns a new `FactHandle`, this also indicates the `Object` was propagated through the network.

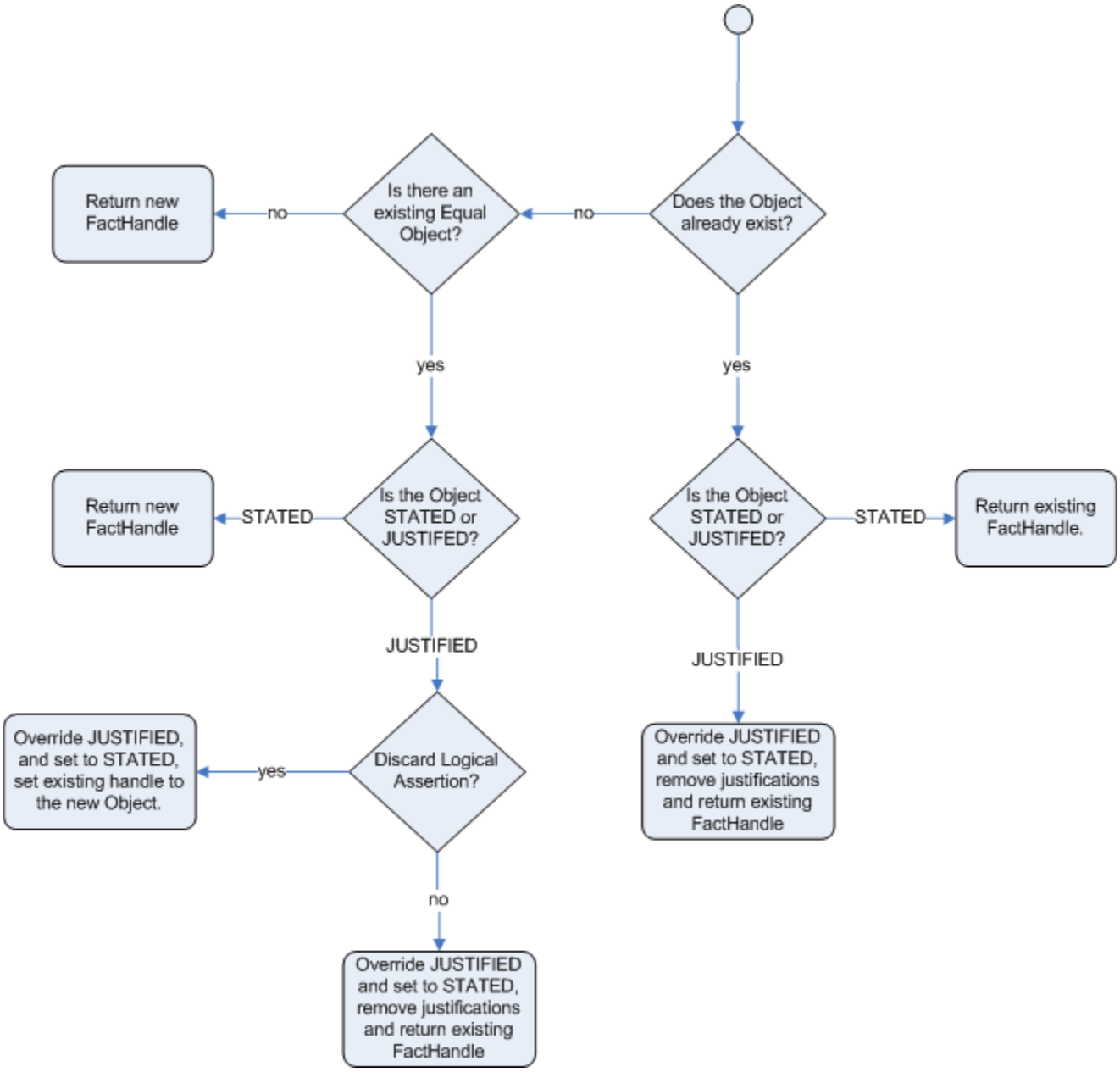


Figure 6.9. Stated Insertion

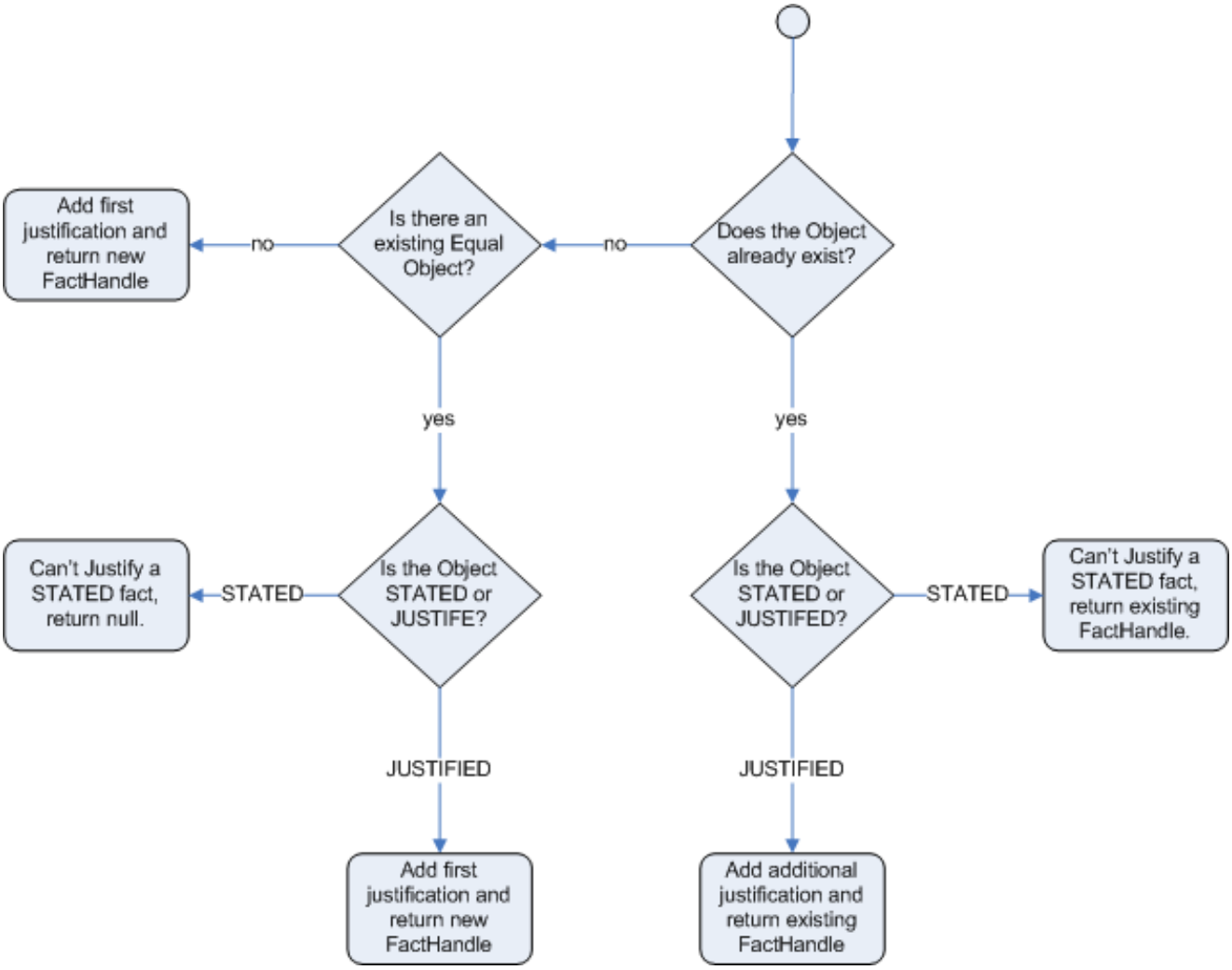


Figure 6.10. Logical Insertion

6.4.1.1. Bus Pass Example With Inference and TMS

The previous example was issuing ID cards to over 18s, in this example we now issue bus passes, either a child or adult pass.

```
rule "Issue Child Bus Pass" when $p : Person( age < 16 )then insert(new
 ChildBusPass( $p ) );end rule "Issue Adult Bus Pass" when $p : Person( age >= 16 )then
   insert(new AdultBusPass( $p ) );end
when    $p : Person( age < 16
)
then    insert(new ChildBusPass( $p )
);
end
 rule "Issue Adult Bus Pass"
when    $p : Person( age >= 16
)
then    insert(new AdultBusPass( $p )
);
```

As before the above example is considered monolithic, leaky and providing poor separation of concerns.

As before we can provide a more robust application with a separation of concerns using inference. Notice this time we don't just insert the inferred object, we use "insertLogical":

```
rule "Infer Child" when   $p : Person( age < 16 )then   insertLogical( new IsChild( $p ) )endrule
 "Infer Adult" when    $p : Person( age >= 16 )then    insertLogical( new IsAdult( $p ) )end
when    $p : Person( age < 16
)
then    insertLogical( new IsChild( $p )
)
endrule "Infer Adult"
when    $p : Person( age >= 16
)
then    insertLogical( new IsAdult( $p )
)
```

A "insertLogical" is part of the Drools Truth Maintenance System (TMS). When a fact is logically inserted, this fact is dependant on the truth of the "when" clause. It means that when the rule becomes false the fact is automatically retracted. This works particularly well as the two rules are mutually exclusive. So in the above rules if the person is under 16 it inserts an IsChild fact, once the person is 16 or over the IsChild fact is automatically retracted and the IsAdult fact inserted.

Returning to the code to issue bus passes, these two rules can + logically insert the ChildBusPass and AdultBusPass facts, as the TMS + supports chaining of logical insertions for a cascading set of retracts.

```
rule "Issue Child Bus Pass" when     $p : Person( )           IsChild( person == $p )then
 insertLogical(new ChildBusPass( $p ) );end rule "Issue Adult Bus Pass" when    $p : Person( age
 >= 16 )         IsAdult( person =$p )then    insertLogical(new AdultBusPass( $p ) );end
when    $p : Person(
)         IsChild( person == $p
)
then    insertLogical(new ChildBusPass( $p )
);
end
 rule "Issue Adult Bus Pass"
when    $p : Person( age >= 16
)         IsAdult( person =$p
)
then    insertLogical(new AdultBusPass( $p )
);
```

Now when a person changes from being 15 to 16, not only is the IsChild fact automatically retracted, so is the person's ChildBusPass fact. For bonus points we can combine this with the 'not' conditional element to handle notifications, in this situation, a request for the returning of the pass. So when the TMS automatically retracts the ChildBusPass object, this rule triggers and sends a request to the person:

```
rule "Return ChildBusPass Request "when    $p : Person( )          not( ChildBusPass( person
 == $p ) )then    requestChildBusPass( $p );end
Request "when    $p :
 Person( )         not( ChildBusPass( person == $p

) )then    requestChildBusPass(
```

6.4.1.2. Important note: Equality for Java objects

It is important to note that for Truth Maintenance (and logical assertions) to work at all, your Fact objects (which may be JavaBeans) must override equals and hashCode methods (from java.lang.Object) correctly. As the truth maintenance system needs to know when two different physical objects are equal in value, *both* equals and hashCode must be overridden correctly, as per the Java standard.

Two objects are equal if and only if their equals methods return true for each other and if their hashCode methods return the same values. See the Java API for more details (but do keep in mind you *MUST* override both equals and hashCode).

TMS behaviour is not affected by theruntime configuration of Identity vs Equality, TMS is always equality.

6.5. Decision Tables in Spreadsheets

Decision tables are a "precise yet compact" (ref. Wikipedia) way of representing conditional logic, and are well suited to *business* level rules.

Drools supports managing rules in a spreadsheet format. Supported formats are Excel (XLS), and CSV, which means that a variety of spreadsheet programs (such as Microsoft Excel, OpenOffice.org Calc amongst others) can be utilized. It is expected that web based decision table editors will be included in a near future release.

Decision tables are an old concept (in software terms) but have proven useful over the years. Very briefly speaking, in Drools decision tables are a way to generate rules driven from the data entered into a spreadsheet. All the usual features of a spreadsheet for data capture and manipulation can be taken advantage of.

6.5.1. When to Use Decision Tables

Consider decision tables as a course of action if rules exist that can be expressed as rule templates and data: each row of a decision table provides data that is combined with a template to generate a rule.

Many businesses already use spreadsheets for managing data, calculations, etc. If you are happy to continue this way, you can also manage your business rules this way. This also assumes you are happy to manage packages of rules in `.xls` or `.csv` files. Decision tables are not recommended

for rules that do not follow a set of templates, or where there are a small number of rules (or if there is a dislike towards software like Excel or OpenOffice.org). They are ideal in the sense that there can be control over what *parameters* of rules can be edited, without exposing the rules directly.

Decision tables also provide a degree of insulation from the underlying object model.

6.5.2. Overview

Here are some examples of real world decision tables (slightly edited to protect the innocent).

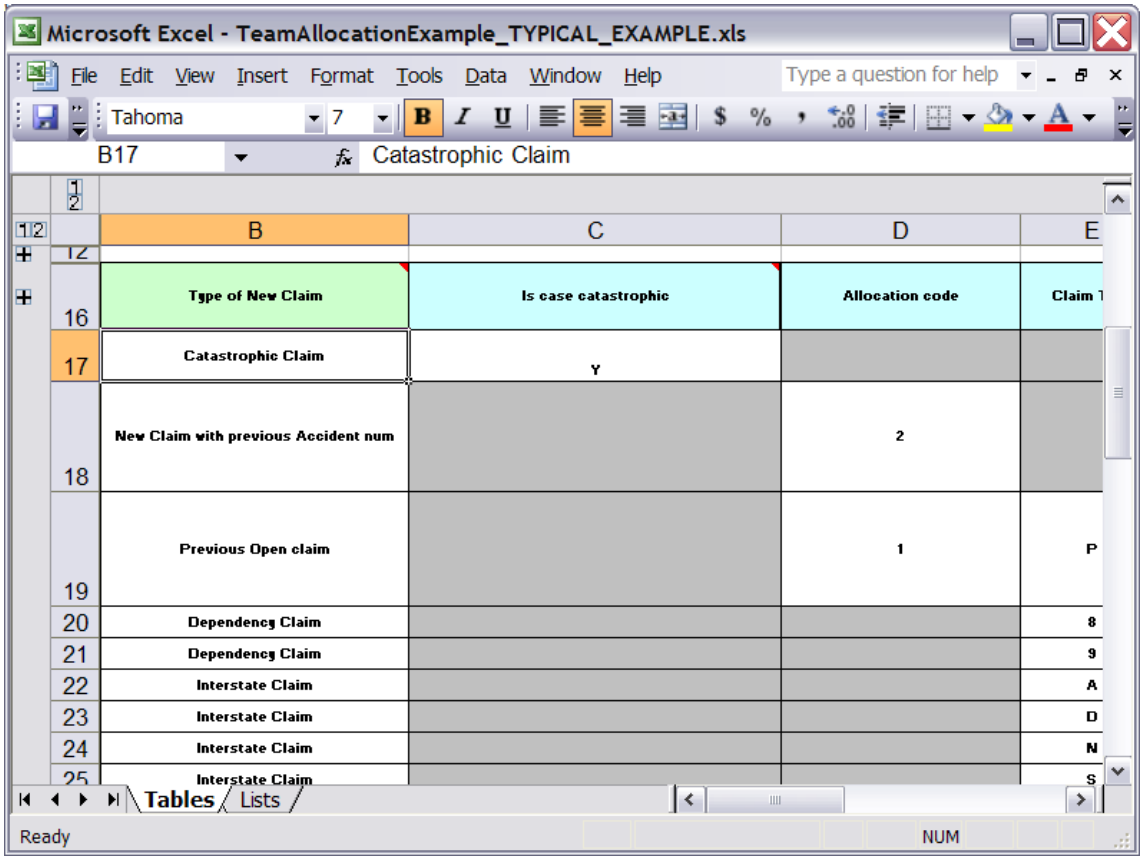


Figure 6.11. Using Excel to edit a decision table

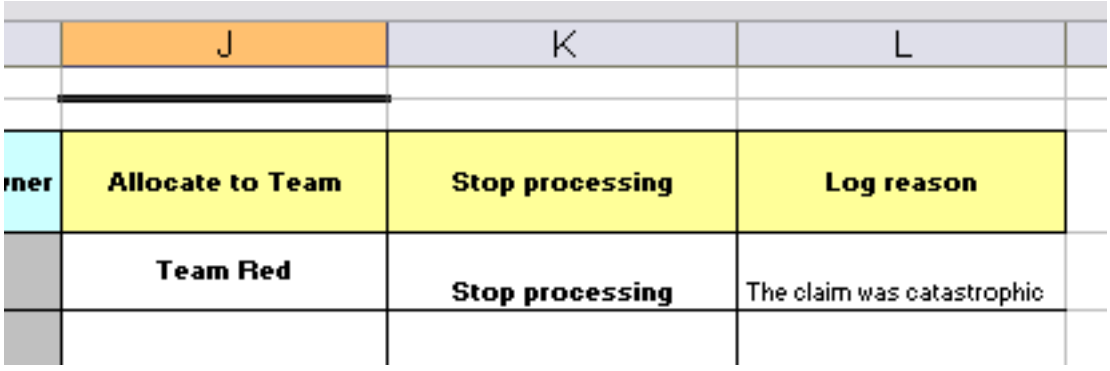


Figure 6.12. Multiple actions for a rule row

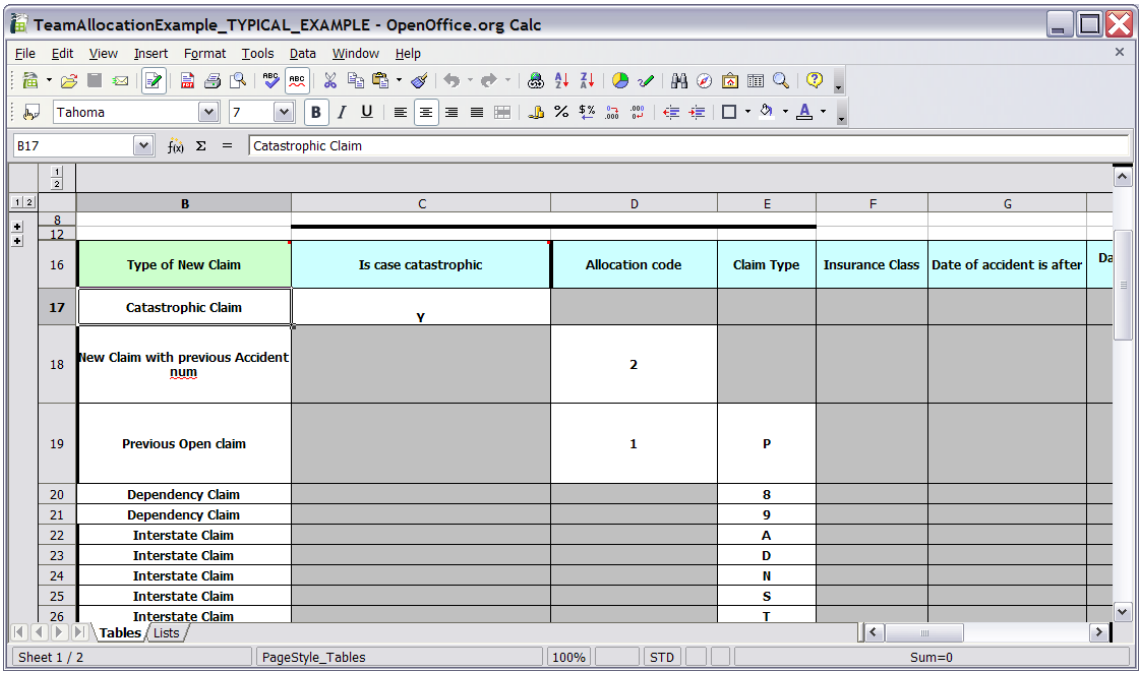


Figure 6.13. Using OpenOffice.org

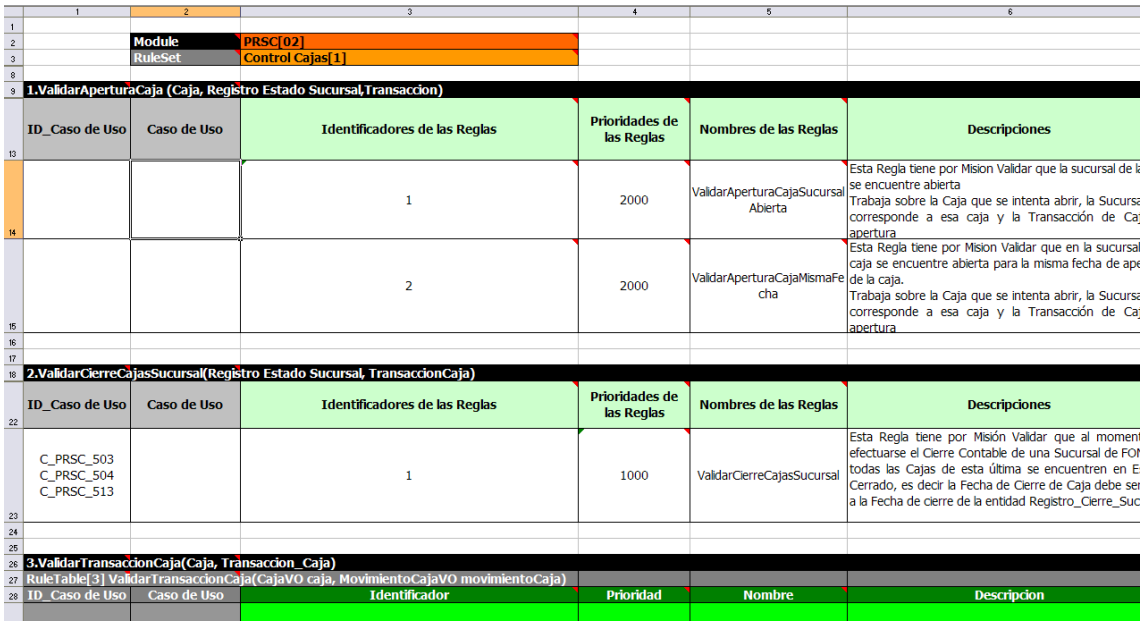
In the above examples, the technical aspects of the decision table have been collapsed away (using a standard spreadsheet feature).

The rules start from row 17, with each row resulting in a rule. The conditions are in columns C, D, E, etc., the actions being off-screen. The values in the cells are quite simple, and their meaning is indicated by the headers in Row 16. Column B is just a description. It is customary to use color to make it obvious what the different areas of the table mean.

Note

Note that although the decision tables look like they process top down, this is not necessarily the case. Ideally, rules are authored without regard for the order of rows, simply because this makes maintenance easier, as rows will not need to be shifted around all the time.

As each row is a rule, the same principles apply. As the rule engine processes the facts, any rules that match may fire. (Some people are confused by this. It is possible to clear the agenda when a rule fires and simulate a very simple decision table where only the first match effects an action.) Also note that you can have multiple tables on one spreadsheet. This way, rules can be grouped where they share common templates, yet at the end of the day they are all combined into one rule package. Decision tables are essentially a tool to generate DRL rules automatically.



	Module	PRSC[02]			
	RuleSet	Control Cajas[1]			
1.ValidarAperturaCaja (Caja, Registro Estado Sucursal,Transaccion)					
ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas	Descripciones
		1	2000	ValidarAperturaCajaSucursal Abierta	Esta Regla tiene por Mision Validar que la sucursal de l se encuentre abierta Trabaja sobre la Caja que se intenta abrir, la Sucursa corresponde a esa caja y la Transacción de Ca apertura
		2	2000	ValidarAperturaCajaMismaFe cha	Esta Regla tiene por Mision Validar que en la sucursal caja se encuentre abierta para la misma fecha de ape de la caja. Trabaja sobre la Caja que se intenta abrir, la Sucursa corresponde a esa caja y la Transacción de Ca apertura
2.ValidarCierreCajasSucursal(Registro Estado Sucursal, TransaccionCaja)					
ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas	Descripciones
C_PRSC_503 C_PRSC_504 C_PRSC_513		1	1000	ValidarCierreCajasSucursal	Esta Regla tiene por Misión Validar que al moment efectuarse el Cierre Contable de una Sucursal de FON todas las Cajas de esta última se encuentren en E Cerrado, es decir la Fecha de Cierre de Caja debe ser a la Fecha de cierre de la entidad Registro_Cierre_Suc
3.ValidarTransaccionCaja(Caja, Transaccion_Caja)					
RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja)					
ID_Caso de Uso	Caso de Uso	Identificador	Prioridad	Nombre	Descripcion

Figure 6.14. A real world example using multiple tables for grouping like rules

6.5.3. How Decision Tables Work

The key point to keep in mind is that in a decision table each row is a rule, and each column in that row is either a condition or action for that rule.

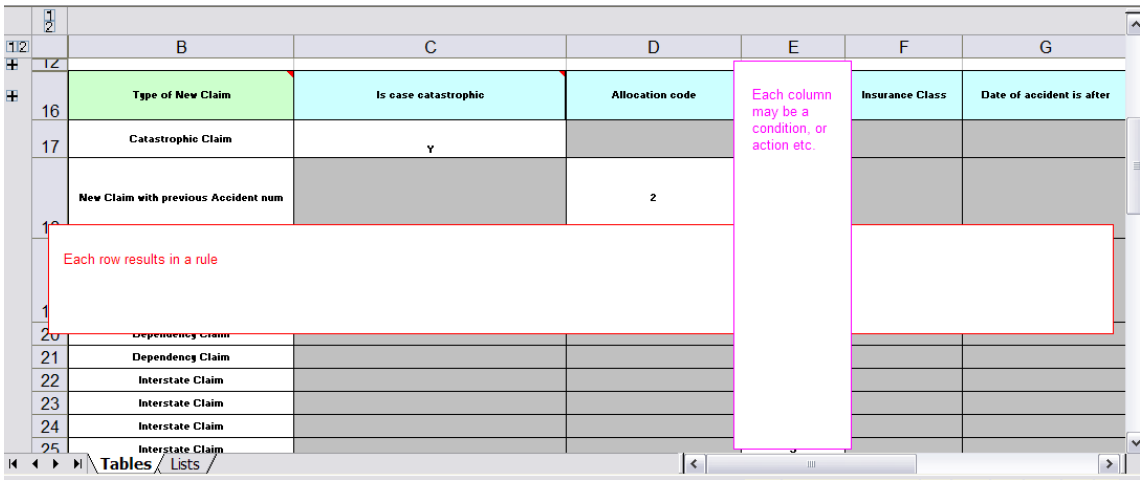


Figure 6.15. Rows and columns

The spreadsheet looks for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column). Other keywords are also used to define other package level attributes (covered later). It is important to keep the keywords in one column. By convention the second column ("B") is used for this, but it can be any column (convention is to leave a margin on the left for notes). In the following diagram, C is actually the column where it starts. Everything to the left of this is ignored.

If we expand the hidden sections, it starts to make more sense how it works; note the keywords in column C.

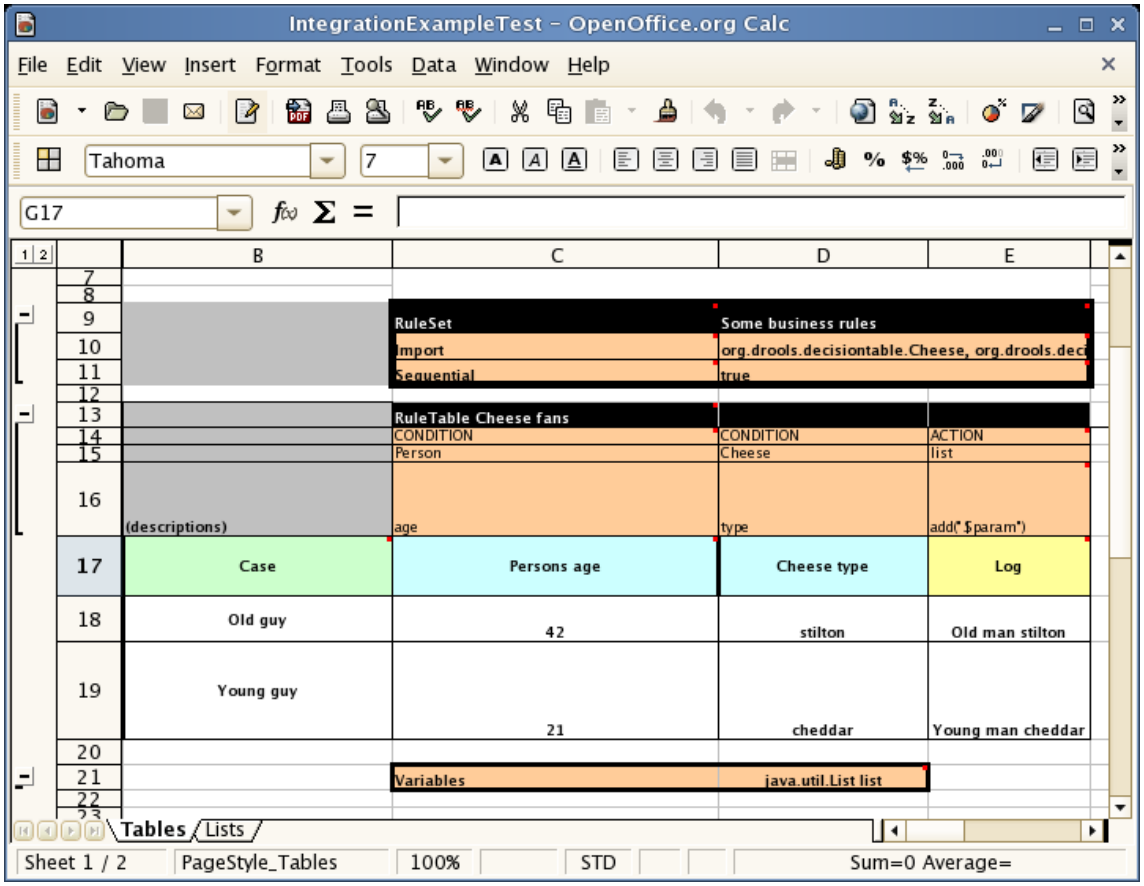


Figure 6.16. Expanded for rule templates

Now the hidden magic which makes it work can be seen. The RuleSet keyword indicates the name to be used in the *rule package* that will encompass all the rules. This name is optional, using a default, but it *must* have the *RuleSet* keyword in the cell immediately to the right.

The other keywords visible in Column C are Import and Sequential which will be covered later. The RuleTable keyword is important as it indicates that a chunk of rules will follow, based on some rule templates. After the RuleTable keyword there is a name, used to prefix the names of the generated rules. The sheet name and row numbers are appended to guarantee unique rule names.

Warning

The RuleTable name combined with the sheet name must be unique across all spreadsheet files in the same KieBase. If that's not the case, some rules might have the same name and only 1 of them will be applied. To show such ignored rules, raise the severity of such rule name conflicts.

The column of RuleTable indicates the column in which the rules start; columns to the left are ignored.

Note

In general the keywords make up name-value pairs.

Referring to row 14 (the row immediately after RuleTable), the keywords CONDITION and ACTION indicate that the data in the columns below are for either the LHS or the RHS parts of a rule. There are other attributes on the rule which can also be optionally set this way.

Row 15 contains declarations of *ObjectTypes*. The content in this row is optional, but if this option is not in use, the row must be left blank; however this option is usually found to be quite useful. When using this row, the values in the cells below (row 16) become constraints on that object type. In the above case, it generates `Person(age=="42")` and `Cheese(type=="stilton")`, where 42 and "stilton" come from row 18. In the above example, the "==" is implicit; if just a field name is given the translator assumes that it is to generate an exact match.

Note

An ObjectType declaration can span columns (via merged cells), meaning that all columns below the merged range are to be combined into one set of constraints within a single pattern matching a single fact at a time, as opposed to non-merged cells containing the same ObjectType, but resulting in different patterns, potentially matching different or identical facts.

Row 16 contains the rule templates themselves. They can use the "$param" placeholder to indicate where data from the cells below should be interpolated. (For multiple insertions, use "$1", "$2", etc., indicating parameters from a comma-separated list in a cell below.) Row 17 is ignored; it may contain textual descriptions of the column's purpose.

Rows 18 and 19 show data, which will be combined (interpolated) with the templates in row 15, to generate rules. If a cell contains no data, then its template is ignored. (This would mean that some condition or action does not apply for that rule row.) Rule rows are read until there is a blank row. Multiple RuleTables can exist in a sheet. Row 20 contains another keyword, and a value. The row positions of keywords like this do not matter (most people put them at the top) but their column should be the same one where the RuleTable or RuleSet keywords should appear. In our case column C has been chosen to be significant, but any other column could be used instead.

In the above example, rules would be rendered like the following (as it uses the "ObjectType" row):

```
//row 18
rule "Cheese_fans_18"
when
```

```
    Person(age=="42")
    Cheese(type=="stilton")
then
    list.add("Old man stilton");
end
```

Note

The constraints `age=="42"` and `type=="stilton"` are interpreted as single constraints, to be added to the respective ObjectType in the cell above. If the cells above were spanned, then there could be multiple constraints on one "column".

Warning

Very large decision tables may have very large memory requirements.

6.5.4. Spreadsheet Syntax

6.5.4.1. Spreadsheet Structure

There are two types of rectangular areas defining data that is used for generating a DRL file. One, marked by a cell labelled `RuleSet`, defines all DRL items except rules. The other one may occur repeatedly and is to the right and below a cell whose contents begin with `RuleTable`. These areas represent the actual decision tables, each area resulting in a set of rules of similar structure.

A Rule Set area may contain cell pairs, one below the `RuleSet` cell and containing a keyword designating the kind of value contained in the other one that follows in the same row.

The columns of a Rule Table area define patterns and constraints for the left hand sides of the rules derived from it, actions for the consequences of the rules, and the values of individual rule attributes. Thus, a Rule Table area should contain one or more columns, both for conditions and actions, and an arbitrary selection of columns for rule attributes, at most one column for each of these. The first four rows following the row with the cell marked with `RuleTable` are earmarked as header area, mostly used for the definition of code to construct the rules. It is any additional row below these four header rows that spawns another rule, with its data providing for variations in the code defined in the Rule Table header.

All keywords are case insensitive.

Only the first worksheet is examined for decision tables.

6.5.4.2. Rule Set Entries

Entries in a Rule Set area may define DRL constructs (except rules), and specify rule attributes. While entries for constructs may be used repeatedly, each rule attribute may be given at most

once, and it applies to all rules unless it is overruled by the same attribute being defined within the Rule Table area.

Entries must be given in a vertically stacked sequence of cell pairs. The first one contains a keyword and the one to its right the value, as shown in the table below. This sequence of cell pairs may be interrupted by blank rows or even a Rule Table, as long as the column marked by `RuleSet` is upheld as the one containing the keyword.

Table 6.5. Entries in the Rule Set area

Keyword	Value	Usage
RuleSet	The package name for the generated DRL file. Optional, the default is `rule_table`.	Must be First entry.
Sequential	"true" or "false". If "true", then salience is used to ensure that rules fire from the top down.	Optional, at most once. If omitted, no firing order is imposed.
EscapeQuotes	"true" or "false". If "true", then quotation marks are escaped so that they appear literally in the DRL.	Optional, at most once. If omitted, quotation marks are escaped.
Import	A comma-separated list of Java classes to import.	Optional, may be used repeatedly.
Variables	Declarations of DRL globals, i.e., a type followed by a variable name. Multiple global definitions must be separated with a comma.	Optional, may be used repeatedly.
Functions	One or more function definitions, according to DRL syntax.	Optional, may be used repeatedly.
Queries	One or more query definitions, according to DRL syntax.	Optional, may be used repeatedly.
Declare	One or more declarative types, according to DRL syntax.	Optional, may be used repeatedly.

Warning

In some locales, MS Office, LibreOffice and OpenOffice will encode a double quoth " differently, which will cause a compilation error. The difference is often hard to see. For example: `“A”` will fail, but `"A"` will work.

For defining rule attributes that apply to all rules in the generated DRL file you can use any of the entries in the following table. Notice, however, that the proper keyword must be used. Also, each of these attributes may be used only once.

Important

Rule attributes specified in a Rule Set area will affect all rule assets in the same package (not only in the spreadsheet). Unless you are sure that the spreadsheet is the only one rule asset in the package, the recommendation is to specify rule attributes not in a Rule Set area but in a Rule Table columns for each rule instead.

Table 6.6. Rule attribute entries in the Rule Set area

Keyword	Initial	Value
PRIORITY	P	An integer defining the "salience" value for the rule. Overridden by the "Sequential" flag.
DURATION	D	A long integer value defining the "duration" value for the rule.
TIMER	T	A timer definition. See "Timers and Calendars".
ENABLED	B	A Boolean value. "true" enables the rule; "false" disables the rule.
CALENDARS	E	A calendars definition. See "Timers and Calendars".
NO-LOOP	U	A Boolean value. "true" inhibits looping of rules due to changes made by its consequence.
LOCK-ON-ACTIVE	L	A Boolean value. "true" inhibits additional activations of all rules with this flag set within the same ruleflow or agenda group.
AUTO-FOCUS	F	A Boolean value. "true" for a rule within an agenda group causes activations of the rule to automatically give the focus to the group.

Keyword	Initial	Value
ACTIVATION-GROUP	X	A string identifying an activation (or XOR) group. Only one rule within an activation group will fire, i.e., the first one to fire cancels any existing activations of other rules within the same group.
AGENDA-GROUP	G	A string identifying an agenda group, which has to be activated by giving it the "focus", which is one way of controlling the flow between groups of rules.
RULEFLOW-GROUP	R	A string identifying a rule-flow group.

6.5.4.3. Rule Tables

All Rule Tables begin with a cell containing "RuleTable", optionally followed by a string within the same cell. The string is used as the initial part of the name for all rules derived from this Rule Table, with the row number appended for distinction. (This automatic naming can be overridden by using a NAME column.) All other cells defining rules of this Rule Table are below and to the right of this cell.

The next row defines the column type, with each column resulting in a part of the condition or the consequence, or providing some rule attribute, the rule name or a comment. The table below shows which column headers are available; additional columns may be used according to the table showing rule attribute entries given in the preceding section. Note that each attribute column may be used at most once. For a column header, either use the keyword or any other word beginning with the letter given in the "Initial" column of these tables.

Table 6.7. Column Headers in the Rule Table

Keyword	Initial	Value	Usage
NAME	N	Provides the name for the rule generated from that row. The default is constructed from the text following the RuleTable tag and the row number.	At most one column

Keyword	Initial	Value	Usage
DESCRIPTION	I	A text, resulting in a comment within the generated rule.	At most one column
CONDITION	C	Code snippet and interpolated values for constructing a constraint within a pattern in a condition.	At least one per rule table
ACTION	A	Code snippet and interpolated values for constructing an action for the consequence of the rule.	At least one per rule table
METADATA	@	Code snippet and interpolated values for constructing a metadata entry for the rule.	Optional, any number of columns

Given a column headed CONDITION, the cells in successive lines result in a conditional element.

- Text in the first cell below CONDITION develops into a pattern for the rule condition, with the snippet in the next line becoming a constraint. If the cell is merged with one or more neighbours, a single pattern with multiple constraints is formed: all constraints are combined into a parenthesized list and appended to the text in this cell. The cell may be left blank, which means that the code snippet in the next row must result in a valid conditional element on its own.

  To include a pattern without constraints, you can write the pattern in front of the text for another pattern.

  The pattern may be written with or without an empty pair of parentheses. A "from" clause may be appended to the pattern.

  If the pattern ends with "eval", code snippets are supposed to produce boolean expressions for inclusion into a pair of parentheses after "eval".

- Text in the second cell below CONDITION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. If you want to create a constraint consisting of a comparison using "==" with the value from the cells below, the field selector alone is sufficient. Any other comparison operator must be specified as the last item within the snippet, and the value from the cells below is appended. For all other constraint forms, you must mark the position for including the contents of a cell with the symbol `$param`. Multiple insertions are possible by using the symbols `$1`, `$2`, etc., and a comma-separated list of values in the cells below.

A text according to the pattern `forall(`*delimiter*`){`*snippet*`}` is expanded by repeating the *snippet* once for each of the values of the comma-separated list of values in each of the cells below, inserting the value in place of the symbol `$` and by joining these expansions by the given *delimiter*. Note that the forall construct may be surrounded by other text.

2. If the cell in the preceding row is not empty, the completed code snippet is added to the conditional element from that cell. A pair of parentheses is provided automatically, as well as a separating comma if multiple constraints are added to a pattern in a merged cell.

   If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below CONDITION is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the conditional element or constraint for this rule.

Given a column headed ACTION, the cells in successive lines result in an action statement.

- Text in the first cell below ACTION is optional. If present, it is interpreted as an object reference.
- Text in the second cell below ACTION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. For a singular insertion, mark the position for including the contents of a cell with the symbol `$param`. Multiple insertions are possible by using the symbols `$1`, `$2`, etc., and a comma-separated list of values in the cells below.

     A method call without interpolation can be achieved by a text without any marker symbols. In this case, use any non-blank entry in a row below to include the statement.

     The forall construct is available here, too.

  2. If the first cell is not empty, its text, followed by a period, the text in the second cell and a terminating semicolon are stringed together, resulting in a method call which is added as an action statement for the consequence.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below ACTION is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the action statement for this rule.

Note

Using `$1` instead of `$param` works in most cases, but it will fail if the replacement text contains a comma: then, only the part preceding the first comma is inserted. Use this "abbreviation" judiciously.

Given a column headed METADATA, the cells in successive lines result in a metadata annotation for the generated rules.

- Text in the first cell below METADATA is ignored.
- Text in the second cell below METADATA is subject to interpolation, as described above, using values from the cells in the rule rows. The metadata marker character `@` is prefixed automatically, and thus it should not be included in the text for this cell.
- Text in the third cell below METADATA is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the metadata annotation for this rule.

6.5.4.4. Examples

The various interpolations are illustrated in the following example.

Example 6.5. Interpolating cell data

If the template is `Foo(bar == $param)` and the cell is `42`, then the result is `Foo(bar == 42)`.

If the template is `Foo(bar < $1, baz == $2)` and the cell contains `42,43`, the result will be `Foo(bar < 42, baz ==43)`.

The template `forall(&&){bar != $}` with a cell containing `42,43` results in `bar != 42 && bar != 43`.

The next example demonstrates the joint effect of a cell defining the pattern type and the code snippet below it.

13	RuleTable Cheese fans	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton
19	21	cheddar

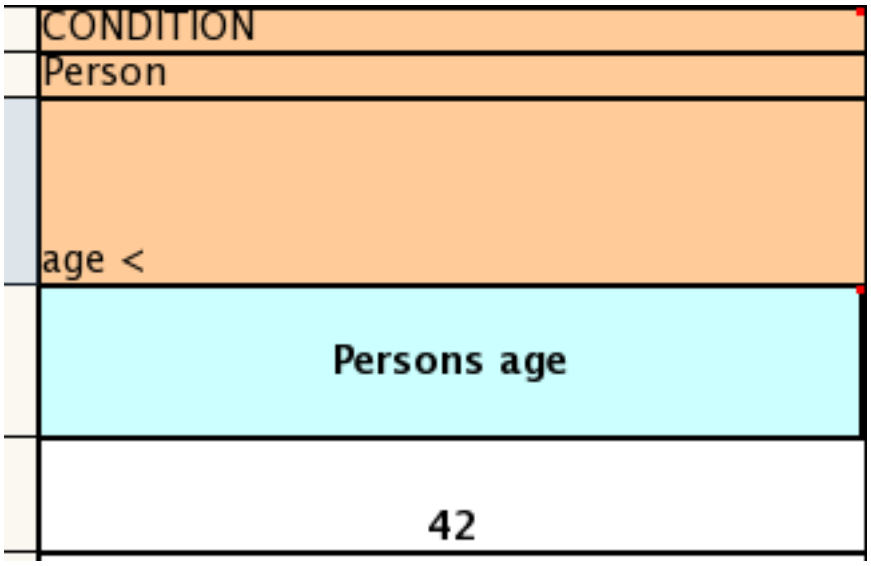
This spreadsheet section shows how the `Person` type declaration spans 2 columns, and thus both constraints will appear as `Person(age == ..., type == ...)`. Since only the field names are present in the snippet, they imply an equality test.

In the following example the marker symbol `$param` is used.

CONDITION
Person
age==“$param”
Persons age
42

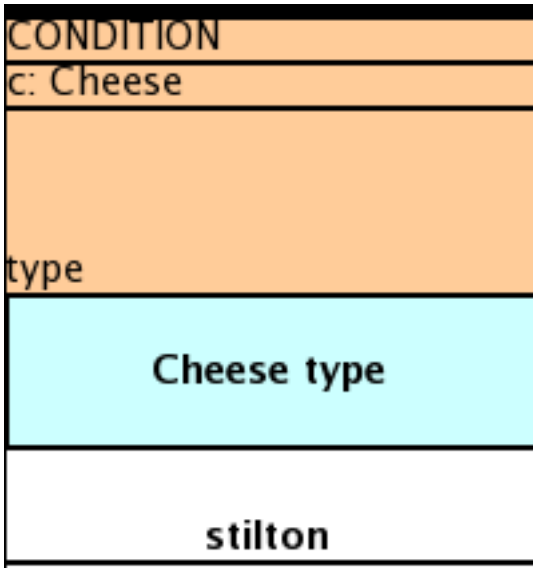
The result of this column is the pattern `Person(age == "42"))`. You may have noticed that the marker and the operator "==" are redundant.

The next example illustrates that a trailing insertion marker can be omitted.



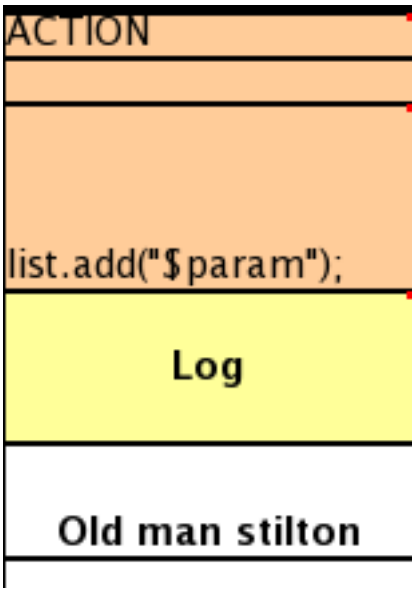
Here, appending the value from the cell is implied, resulting in `Person(age < "42")).`

You can provide the definition of a binding variable, as in the example below. .



Here, the result is `c: Cheese(type == "stilton").` Note that the quotes are provided automatically. Actually, anything can be placed in the object type row. Apart from the definition of a binding variable, it could also be an additional pattern that is to be inserted literally.

A simple construction of an action statement with the insertion of a single value is shown below.



The cell below the ACTION header is left blank. Using this style, anything can be placed in the consequence, not just a single method call. (The same technique is applicable within a CONDITION column as well.)

Below is a comprehensive example, showing the use of various column headers. It is not an error to have no value below a column header (as in the NO-LOOP column): here, the attribute will not be applied in any of the rules.

	B	C	D	E	F	G	H
1							
2		RuleSet	org.acme.insurance.base				
3		import	import org.acme.insurance.base.Approve, import org.acme.insurance.base.Driver				
4		Package	org.acme.insurance.base				
5							
6		RuleTable Old Driver					
7		CONDITION	CONDITION	RULEFLOW-GROUP	NO-LOOP	ACTION	ACTION
8		$driver: Driver					
9	iptions)	licenceYears	priorClaims			insert(new Approve("$param"));	System.out.println("$pa
10	ase	Persons age	Prior Claims			Inserting approvment	Log
11	d guy	30	1	risk assessment		Safe and mature	Old driver Approved
12							
13							
14							
15							
16							

Figure 6.17. Example usage of keywords for imports, headers, etc.

And, finally, here is an example of Import, Variables and Functions.

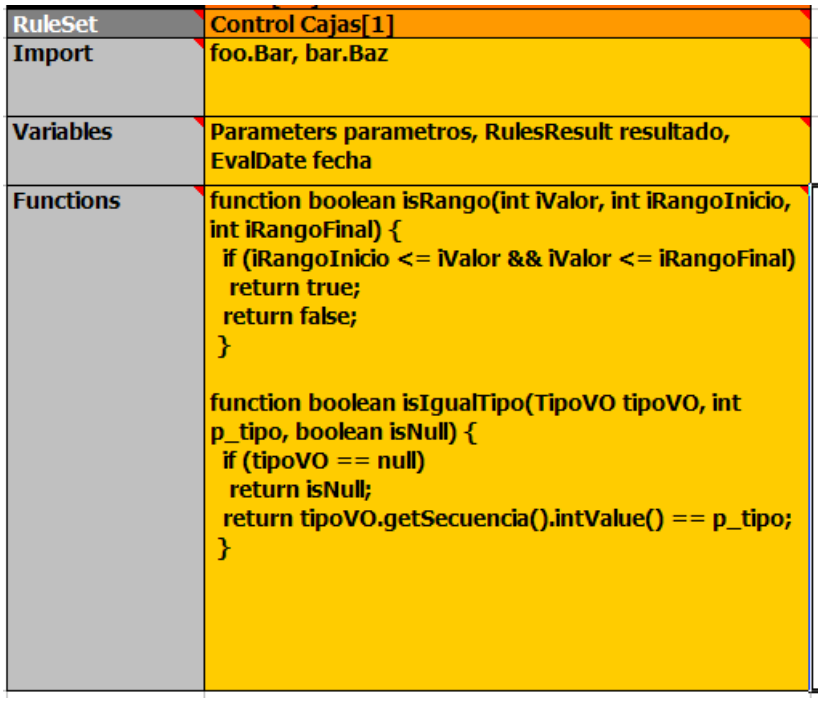


Figure 6.18. Example usage of keywords for functions, etc.

Multiple package names within the same cell must be separated by a comma. Also, the pairs of type and variable names must be comma-separated. Functions, however, must be written as they appear in a DRL file. This should appear in the same column as the "RuleSet" keyword; it could be above, between or below all the rule rows.

Note

It may be more convenient to use Import, Variables, Functions and Queries repeatedly rather than packing several definitions into a single cell.

6.5.5. Creating and integrating Spreadsheet based Decision Tables

The API to use spreadsheet based decision tables is in the drools-decisiontables module. There is really only one class to look at: `SpreadsheetCompiler`. This class will take spreadsheets in various formats, and generate rules in DRL (which you can then use in the normal way). The `SpreadsheetCompiler` can just be used to generate partial rule files if it is wished, and assemble it into a complete rule package after the fact (this allows the separation of technical and non-technical aspects of the rules if needed).

To get started, a sample spreadsheet can be used as a base. Alternatively, if the plug-in is being used (Rule Workbench IDE), the wizard can generate a spreadsheet from a template (to edit it an xls compatible spreadsheet editor will need to be used).

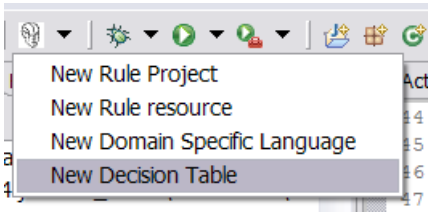


Figure 6.19. Wizard in the IDE

6.5.6. Managing Business Rules in Decision Tables

6.5.6.1. Workflow and Collaboration

Spreadsheets are well established business tools (in use for over 25 years). Decision tables lend themselves to close collaboration between IT and domain experts, while making the business rules clear to business analysts, it is an ideal separation of concerns.

Typically, the whole process of authoring rules (coming up with a new decision table) would be something like:

1. Business analyst takes a template decision table (from a repository, or from IT)
2. Decision table business language descriptions are entered in the table(s)
3. Decision table rules (rows) are entered (roughly)
4. Decision table is handed to a technical resource, who maps the business language (descriptions) to scripts (this may involve software development of course, if it is a new application or data model)
5. Technical person hands back and reviews the modifications with the business analyst.
6. The business analyst can continue editing the rule rows as needed (moving columns around is also fine etc).
7. In parallel, the technical person can develop test cases for the rules (liaising with business analysts) as these test cases can be used to verify rules and rule changes once the system is running.

6.5.6.2. Using spreadsheet features

Features of applications like Excel can be used to provide assistance in entering data into spreadsheets, such as validating fields. Lists that are stored in other worksheets can be used to provide valid lists of values for cells, like in the following diagram.

<title> Wizard in the IDE </title>

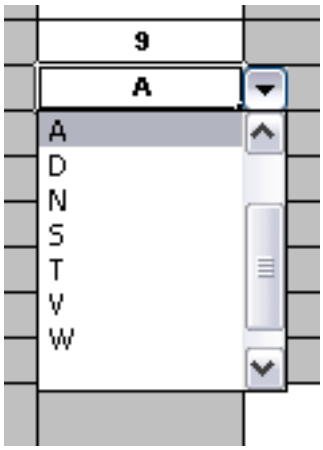


Figure 6.20.

Some applications provide a limited ability to keep a history of changes, but it is recommended to use an alternative means of revision control. When changes are being made to rules over time, older versions are archived (many open source solutions exist for this, such as Subversion or Git).

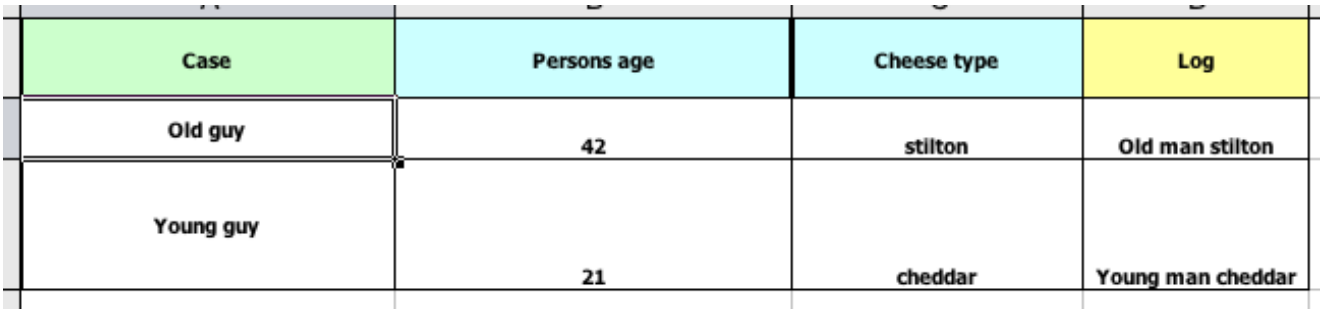
6.5.7. Rule Templates

Related to decision tables (but not necessarily requiring a spreadsheet) are "Rule Templates" (in the drools-templates module). These use any tabular data source as a source of rule data - populating a template to generate many rules. This can allow both for more flexible spreadsheets, but also rules in existing databases for instance (at the cost of developing the template up front to generate the rules).

With Rule Templates the data is separated from the rule and there are no restrictions on which part of the rule is data-driven. So whilst you can do everything you could do in decision tables you can also do the following:

- store your data in a database (or any other format)
- conditionally generate rules based on the values in the data
- use data for any part of your rules (e.g. condition operator, class name, property name)
- run different templates over the same data

As an example, a more classic decision table is shown, but without any hidden rows for the rule meta data (so the spreadsheet only contains the raw data to generate the rules).



Case	Persons age	Cheese type	Log
Old guy	42	stilton	Old man stilton
Young guy	21	cheddar	Young man cheddar

Figure 6.21. Template data

See the `ExampleCheese.xls` in the examples download for the above spreadsheet.

If this was a regular decision table there would be hidden rows before row 1 and between rows 1 and 2 containing rule metadata. With rule templates the data is completely separate from the rules. This has two handy consequences - you can apply multiple rule templates to the same data and your data is not tied to your rules at all. So what does the template look like?

```
1  template header
2  age
3  type
4  log
5
6  package org.drools.examples.templates;
7
8  global java.util.List list;
9
10 template "cheesefans"
11
12 rule "Cheese fans_@{row.rowNumber}"
13 when
14    Person(age == @{age})
15    Cheese(type == "@{type}")
16 then
17    list.add("@{log}");
18 end
19
20 end template
```

Annotations to the preceding program listing:

- Line 1: All rule templates start with `template header`.
- Lines 2-4: Following the header is the list of columns in the order they appear in the data. In this case we are calling the first column `age`, the second `type` and the third `log`.
- Line 5: An empty line signifies the end of the column definitions.

- Lines 6-9: Standard rule header text. This is standard rule DRL and will appear at the top of the generated DRL. Put the package statement and any imports and global and function definitions into this section.
- Line 10: The keyword `template` signals the start of a rule template. There can be more than one template in a template file, but each template should have a unique name.
- Lines 11-18: The rule template - see below for details.
- Line 20: The keywords `end template` signify the end of the template.

The rule templates rely on MVEL to do substitution using the syntax @{token_name}. There is currently one built-in expression, @{row.rowNumber} which gives a unique number for each row of data and enables you to generate unique rule names. For each row of data a rule will be generated with the values in the data substituted for the tokens in the template. With the example data above the following rule file would be generated:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Cheese fans_1"
when
  Person(age == 42)
  Cheese(type == "stilton")
then
  list.add("Old man stilton");
end

rule "Cheese fans_2"
when
  Person(age == 21)
  Cheese(type == "cheddar")
then
  list.add("Young man cheddar");
end
```

The code to run this is simple:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
dtableconfiguration.setInputType( DecisionTableInputType.XLS );

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource( getSpreadsheetName(),
                                                    getClass() ),
              ResourceType.DTABLE,
              dtableconfiguration );
```

6.6. Logging

One way to illuminate the black box that is a rule engine, is to play with the logging level.

Everything is logged to SLF4J [http://www.slf4j.org/], which is a simple logging facade that can delegate any log to Logback, Apache Commons Logging, Log4j or java.util.logging. Add a dependency to the logging adaptor for your logging framework of choice. If you're not using any logging framework yet, you can use Logback by adding this Maven dependency:

```
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>1.x</version>
    </dependency>
```

Note

If you're developing for an ultra light environment, use `slf4j-nop` or `slf4j-simple` instead.

Configure the logging level on the package `org.drools`. For example:

In Logback, configure it in your `logback.xml` file:

```
<configuration>

    <logger name="org.drools" level="debug"/>

    ...

<configuration>
```

In Log4J, configure it in your `log4j.xml` file:

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <category name="org.drools">
      <priority value="debug" />
    </category>

    ...

</log4j:configuration>
```

Chapter 7. Running

Ths sections extends the KIE Running section, which should be read first, with specifics for the Drools runtime.

7.1. KieRuntime

7.1.1. EntryPoint

The `EntryPoint` provides the methods around inserting, updating and deleting facts. The term "entry point" is related to the fact that we have multiple partitions in a Working Memory and you can choose which one you are inserting into. The use of multiple entry points is more common in event processing use cases, but they can be used by pure rule applications as well.

The `KieRuntime` interface provides the main interaction with the engine. It is available in rule consequences and process actions. In this manual the focus is on the methods and interfaces related to rules, and the methods pertaining to processes will be ignored for now. But you'll notice that the `KieRuntime` inherits methods from both the `WorkingMemory` and the `ProcessRuntime`, thereby providing a unified API to work with processes and rules. When working with rules, three interfaces form the `KieRuntime`: `EntryPoint`, `WorkingMemory` and the `KieRuntime` itself.

Figure 7.1. EntryPoint

7.1.1.1. Insert

In order for a fact to be evaluated against the rules in a `KieBase`, it has to be inserted into the session. This is done by calling the method `insert(yourObject)`. When a fact is inserted into the session, some of its properties might be immediately evaluated (eager evaluation) and some might be deferred for later evaluation (lazy evaluation). The exact behaviour depends on the rules engine algorithm being used.

Note

Expert systems typically use the term *assert* or *assertion* to refer to facts made available to the system. However, due to "assert" being a keyword in most languages, we have decided to use the `insert` keyword; In this manual, the two terms are used interchangeably.

When an Object is inserted it returns a `FactHandle`. This `FactHandle` is the token used to represent your inserted object within the `WorkingMemory`. It is also used for interactions with the `WorkingMemory` when you wish to delete or modify an object.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );
```

As mentioned in the KieBase section, a Working Memory may operate in two assertion modes: either equality or identity. Identity is the default.

Identity means that the Working Memory uses an `IdentityHashMap` to store all asserted objects. New instance assertions always result in the return of new `FactHandle`, but if an instance is asserted again then it returns the original fact handle, i.e., it ignores repeated insertions for the same object.

Equality means that the Working Memory uses a `HashMap` to store all asserted objects. An object instance assertion will only return a new `FactHandle` if the inserted object is not equal (according to its `equal()/hashcode()` methods) to an already existing fact.

7.1.1.2. Delete

In order to remove a fact from the session, the method `delete()` is used. When a fact is deleted, any matches that are active and depend on that fact will be cancelled. Note that it is possible to have rules that depend on the nonexistence of a fact, in which case deleting a fact may cause a rule to activate. (See the `not` and `exists` keywords).

Note

Expert systems typically use the term *retract* or *retraction* to refer to the operation of removing facts from the Working Memory. Drools prefers the keyword `delete` for symmetry with the keyword `insert`; Drools also supports the keyword `retract`, but it was deprecated in favor of `delete`. In this manual, the two terms are used interchangeably.

Retraction may be done using the `FactHandle` that was returned by the insert call. On the right hand side of a rule the `delete` statement is used, which works with a simple object reference.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );
....
ksession.delete( stiltonHandle );
```

7.1.1.3. Update

The Rule Engine must be notified of modified facts, so that they can be reprocessed. You must use the `update()` method to notify the `WorkingMemory` of changed objects for those objects that are not able to notify the `WorkingMemory` themselves. Notice that `update()` always takes the modified object as a second parameter, which allows you to specify new instances for immutable

objects. On the right hand side of a rule the `modify` statement is recommended, as it makes the changes and notifies the engine in a single statement. Alternatively, after changing a fact object's field values through calls of setter methods you must invoke `update` immediately, event before changing another fact, or you will cause problems with the indexing within the rule engine. The modify statement avoids this problem.

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = workingMemory.insert( stilton );
...
stilton.setPrice( 100 );
workingMemory.update( stiltonHandle, stilton );
```

7.1.2. RuleRuntime

The RuleRuntime provides access to the Agenda, permits query executions, and lets you access named Entry Points.

Figure 7.2. RuleRuntime

7.1.2.1. Query

Queries are used to retrieve fact sets based on patterns, as they are used in rules. Patterns may make use of optional parameters. Queries can be defined in the Knowledge Base, from where they are called up to return the matching results. While iterating over the result collection, any identifier bound in the query can be used to access the corresponding fact or fact field by calling the `get` method with the binding variable's name as its argument. If the binding refers to a fact object, its FactHandle can be retrieved by calling `getFactHandle`, again with the variable's name as the parameter.

Figure 7.3. QueryResults

Figure 7.4. QueryResultsRow

Example 7.1. Simple Query Example

```
QueryResults results =
    ksession.getQueryResults( "my query", new Object[] { "string" } );
for ( QueryResultsRow row : results ) {
```

```
    System.out.println( row.get( "varName" ) );
}
```

7.1.2.2. Live Queries

Invoking queries and processing the results by iterating over the returned set is not a good way to monitor changes over time.

To alleviate this, Drools provides Live Queries, which have a listener attached instead of returning an iterable result set. These live queries stay open by creating a view and publishing change events for the contents of this view. To activate, you start your query with parameters and listen to changes in the resulting view. The `dispose` method terminates the query and discontinues this reactive scenario.

Example 7.2. Implementing ViewChangedEventListener

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
 public void rowUpdated(Row row) {
  updated.add( row.get( "$price" ) );
 }

 public void rowRemoved(Row row) {
  removed.add( row.get( "$price" ) );
 }

 public void rowAdded(Row row) {
  added.add( row.get( "$price" ) );
 }
};

// Open the LiveQuery
LiveQuery query = ksession.openLiveQuery( "cheeses",
                                          new Object[] { "cheddar", "stilton" },
                                          listener );
...
...
query.dispose() // calling dispose to terminate the live query
```

A Drools blog article contains an example of Glazed Lists integration for live queries:

http://blog.athico.com/2010/07/glazed-lists-examples-for-drools-live.html

7.1.3. StatefulRuleSession

The `StatefulRuleSession` is inherited by the `KieSession` and provides the rule related methods that are relevant from outside of the engine.

Figure 7.5. StatefulRuleSession

7.1.3.1. Agenda Filters

Figure 7.6. AgendaFilters

`AgendaFilter` objects are optional implementations of the filter interface which are used to allow or deny the firing of a match. What you filter on is entirely up to the implementation. Drools 4.0 used to supply some out of the box filters, which have not be exposed in drools 5.0 knowledge-api, but they are simple to implement and the Drools 4.0 code base can be referred to.

To use a filter specify it while calling `fireAllRules()`. The following example permits only rules ending in the string `"Test"`. All others will be filtered out.

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

7.2. Agenda

The Agenda is a *Rete* feature. During actions on the `WorkingMemory`, rules may become fully matched and eligible for execution; a single Working Memory Action can result in multiple eligible rules. When a rule is fully matched a Match is created, referencing the rule and the matched facts, and placed onto the Agenda. The Agenda controls the execution order of these Matches using a Conflict Resolution strategy.

The engine cycles repeatedly through two phases:

1. Working Memory Actions. This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls `fireAllRules()` the engine switches to the Agenda Evaluation phase.

2. Agenda Evaluation. This attempts to select a rule to fire. If no rule is found it exits, otherwise it fires the found rule, switching the phase back to Working Memory Actions.

Figure 7.7. Two Phase Execution

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

Figure 7.8. Agenda

7.2.1. Conflict Resolution

Conflict resolution is required when there are multiple rules on the agenda. (The basics to this are covered in chapter "Quick Start".) As firing a rule may have side effects on the working memory, the rule engine needs to know in what order the rules should fire (for instance, firing ruleA may cause ruleB to be removed from the agenda).

The default conflict resolution strategies employed by Drools are: Salience and LIFO (last in, first out).

The most visible one is *salience* (or priority), in which case a user can specify that a certain rule has a higher priority (by giving it a higher number) than other rules. In that case, the rule with higher salience will be preferred. LIFO priorities are based on the assigned Working Memory Action counter value, with all rules created during the same action receiving the same value. The execution order of a set of firings with the same priority value is arbitrary.

As a general rule, it is a good idea not to count on rules firing in any particular order, and to author the rules without worrying about a "flow". However when a flow is needed a number of possibilities exist, including but not limited to: agenda groups, rule flow groups, activation groups, control/semaphore facts. These are discussed in later sections.

Drools 4.0 supported custom conflict resolution strategies; while this capability still exists in Drools it has not yet been exposed to the end user via knowledge-api in Drools 5.0.

7.2.2. AgendaGroup

Figure 7.9. AgendaGroup

Agenda groups are a way to partition rules (matches, actually) on the agenda. At any one time, only one group has "focus" which means that matches for rules in that group only will take effect. You can also have rules with "auto focus" which means that the focus is taken for its agenda group when that rule's conditions are true.

Agenda groups are known as "modules" in CLIPS terminology. While it best to design rules that do not need control flow, this is not always possible. Agenda groups provide a handy way to create a "flow" between grouped rules. You can switch the group which has focus either from within the rule engine, or via the API. If your rules have a clear need for multiple "phases" or "sequences" of processing, consider using agenda-groups for this purpose.

Each time `setFocus()` is called it pushes that Agenda Group onto a stack. When the focus group is empty it is popped from the stack and the focus group that is now on top evaluates. An Agenda

Group can appear in multiple locations on the stack. The default Agenda Group is "MAIN", with all rules which do not specify an Agenda Group being in this group. It is also always the first group on the stack, given focus initially, by default.

```
ksession.getAgenda().getAgendaGroup( "Group A" ).setFocus();
```

The `clear()` method can be used to cancel all the activations generated by the rules belonging to a given Agenda Group before one has had a chance to fire.

```
ksession.getAgenda().getAgendaGroup( "Group A" ).clear();
```

Note that, due to the lazy nature of the phreak algorithm used by Drools, the activations are by default materialized only at firing time, but it is possible to anticipate the evaluation and then the activation of a given rule at the moment when a fact is inserted into the session by annotating it with `@Propagation(IMMEDIATE)` as explained in the Propagation modes section.

7.2.3. ActivationGroup

Figure 7.10. ActivationGroup

An activation group is a set of rules bound together by the same "activation-group" rule attribute. In this group only one rule can fire, and after that rule has fired all the other rules are cancelled from the agenda. The `clear()` method can be called at any time, which cancels all of the activations before one has had a chance to fire.

```
ksession.getAgenda().getActivationGroup( "Group B" ).clear();
```

7.2.4. RuleFlowGroup

Figure 7.11. RuleFlowGroup

A rule flow group is a group of rules associated by the "ruleflow-group" rule attribute. These rules can only fire when the group is activate. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group. Here too, the `clear()` method can be called at any time to cancels all matches still remaining on the Agenda.

```
ksession.getAgenda().getRuleFlowGroup( "Group C" ).clear();
```

7.3. Event Model

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you, for instance, to separate logging and auditing activities from the main part of your application (and the rules).

The `WorkingMemoryEventManager` allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

Figure 7.12. WorkingMemoryEventManager

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print matches after they have fired.

Example 7.3. Adding an AgendaEventListener

```
ksession.addEventListener( new DefaultAgendaEventListener() {
   public void afterMatchFired(AfterMatchFiredEvent event) {
       super.afterMatchFired( event );
       System.out.println( event );
   }
});
```

Drools also provides `DebugRuleRuntimeEventListener` and `DebugAgendaEventListener` which implement each method with a debug print statement. To print all Working Memory events, you add a listener like this:

Example 7.4. Adding a DebugRuleRuntimeEventListener

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

The events currently supported are:

- MatchCreatedEvent
- MatchCancelledEvent
- BeforeMatchFiredEvent
- AfterMatchFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent

- ObjectDeletedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

7.4. StatelessKieSession

The `StatelessKieSession` wraps the `KieSession`, instead of extending it. Its main focus is on decision service type scenarios. It avoids the need to call `dispose()`. Stateless sessions do not support iterative insertions and the method call `fireAllRules()` from Java code; the act of calling `execute()` is a single-shot method that will internally instantiate a `KieSession`, add all the user data and execute user commands, call `fireAllRules()`, and then call `dispose()`. While the main way to work with this class is via the `BatchExecution` (a subinterface of `Command`) as supported by the `CommandExecutor` interface, two convenience methods are provided for when simple object insertion is all that's required. The `CommandExecutor` and `BatchExecution` are talked about in detail in their own section.

Figure 7.13. StatelessKieSession

Our simple example shows a stateless session executing a given collection of Java objects using the convenience API. It will iterate the collection, inserting each element in turn.

Example 7.5. Simple StatelessKieSession execution with a Collection

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ksession.execute( collection );
```

If this was done as a single Command it would be as follows:

Example 7.6. Simple StatelessKieSession execution with InsertElements Command

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

If you wanted to insert the collection itself, and the collection's individual elements, then `CommandFactory.newInsert(collection)` would do the job.

Methods of the `CommandFactory` create the supported commands, all of which can be marshalled using XStream and the `BatchExecutionHelper`. `BatchExecutionHelper` provides details on the XML format as well as how to use Drools Pipeline to automate the marshalling of `BatchExecution` and `ExecutionResults`.

`StatelessKieSession` supports globals, scoped in a number of ways. I'll cover the non-command way first, as commands are scoped to a specific execution call. Globals can be resolved in three ways.

- The StatelessKieSession method `getGlobals()` returns a Globals instance which provides access to the session's globals. These are shared for *all* execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

Example 7.7. Session scoped global

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
// Set a global hbnSession, that can be used for DB interactions in the rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// Execute while being able to resolve the "hbnSession" identifier.
ksession.execute( collection );
```

- Using a delegate is another way of global resolution. Assigning a value to a global (with `setGlobal(String, Object)`) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. Only if an identifier cannot be found in this internal collection, the delegate global (if any) will be used.
- The third way of resolving globals is to have execution scoped globals. Here, a `Command` to set a global is passed to the `CommandExecutor`.

The `CommandExecutor` interface also offers the ability to export data via "out" parameters. Inserted facts, globals and query results can all be returned.

Example 7.8. Out identifiers

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true ) );
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" ) );
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
```

```
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

7.4.1. Sequential Mode

With Rete you have a stateful session where objects can be asserted and modified over time, and where rules can also be added and removed. Now what happens if we assume a stateless session, where after the initial data set no more data can be asserted or modified and rules cannot be added or removed? Certainly it won't be necessary to re-evaluate rules, and the engine will be able to operate in a simplified way.

1. Order the Rules by salience and position in the ruleset (by setting a sequence attribute on the rule terminal node).
2. Create an elements, one element for each possible rule match; element position indicates firing order.
3. Turn off all node memories, except the right-input Object memory.
4. Disconnect the Left Input Adapter Node propagation, and let the Object plus the Node be referenced in a Command object, which is added to a list on the Working Memory for later execution.
5. Assert all objects, and, when all assertions are finished and thus right-input node memories are populated, check the Command list and execute each in turn.
6. All resulting Matches should be placed in the elements, based upon the determined sequence number of the Rule. Record the first and last populated elements, to reduce the iteration range.
7. Iterate the elements of Matches, executing populated element in turn.
8. If we have a maximum number of allowed rule executions, we can exit our network evaluations early to fire all the rules in the elements.

The `LeftInputAdapterNode` no longer creates a Tuple, adding the Object, and then propagate the Tuple – instead a Command object is created and added to a list in the Working Memory. This Command object holds a reference to the `LeftInputAdapterNode` and the propagated object. This stops any left-input propagations at insertion time, so that we know that a right-input propagation will never need to attempt a join with the left-inputs (removing the need for left-input memory). All nodes have their memory turned off, including the left-input Tuple memory but excluding the right-input object memory, which means that the only node remembering an insertion propagation is the right-input object memory. Once all the assertions are finished and all right-input memories populated, we can then iterate the list of `LeftInputAdatperNode` Command objects calling each in turn. They will propagate down the network attempting to join with the right-input objects, but they won't be remembered in the left input as we know there will be no further object assertions and thus propagations into the right-input memory.

There is no longer an Agenda, with a priority queue to schedule the Tuples; instead, there is simply an elements for the number of rules. The sequence number of the `RuleTerminalNode`

indicates the element within the elements where to place the Match. Once all Command objects have finished we can iterate our elements, checking each element in turn, and firing the Matches if they exist. To improve performance, we remember the first and the last populated cell in the elements. The network is constructed, with each `RuleTerminalNode` being given a sequence number based on a salience number and its order of being added to the network.

Typically the right-input node memories are Hash Maps, for fast object deletion; here, as we know there will be no object deletions, we can use a list when the values of the object are not indexed. For larger numbers of objects indexed Hash Maps provide a performance increase; if we know an object type has only a few instances, indexing is probably not advantageous, and a list can be used.

Sequential mode can only be used with a Stateless Session and is off by default. To turn it on, either call `RuleBaseConfiguration.setSequential(true)`, or set the rulebase configuration property `drools.sequential` to true. Sequential mode can fall back to a dynamic agenda by calling `setSequentialAgenda` with `SequentialAgenda.DYNAMIC`. You may also set the "drools.sequential.agenda" property to "sequential" or "dynamic".

7.5. Propagation modes

The introduction of PHREAK as default algorithm for the Drools engine made the rules' evaluation lazy. This new Drools lazy behavior allowed a relevant performance boost but, in some very specific cases, breaks the semantic of a few Drools features.

More precisely in some circumstances it is necessary to propagate the insertion of new fact into th session immediately. For instance Drools allows a query to be executed in pull only (or passive) mode by prepending a '?' symbol to its invocation as in the following example:

Example 7.9. A passive query

```
query Q (Integer i)
    String( this == i.toString() )
end
rule R when
    $i : Integer()
    ?Q( $i; )
then
    System.out.println( $i );
end
```

In this case, since the query is passive, it shouldn't react to the insertion of a String matching the join condition in the query itself. In other words this sequence of commands

```
KieSession ksession = ...
ksession.insert(1);
ksession.insert("1");
ksession.fireAllRules();
```

shouldn't cause the rule R to fire because the String satisfying the query condition has been inserted after the Integer and the passive query shouldn't react to this insertion. Conversely the rule should fire if the insertion sequence is inverted because the insertion of the Integer, when the passive query can be satisfied by the presence of an already existing String, will trigger it.

Unfortunately the lazy nature of PHREAK doesn't allow the engine to make any distinction regarding the insertion sequence of the two facts, so the rule will fire in both cases. In circumstances like this it is necessary to evaluate the rule eagerly as done by the old RETEOO-based engine.

In other cases it is required that the propagation is eager, meaning that it is not immedate, but anyway has to happen before the engine/agenda starts scheduled evaluations. For instance this is necessary when a rule has the no-loop or the lock-on-active attribute and in fact when this happens this propagation mode is automatically enforced by the engine.

To cover these use cases, and in all other situations where an immediate or eager rule evaluation is required, it is possible to declaratively specify so by annotating the rule itself with **@Propagation(Propagation.Type)**, where Propagation.Type is an enumeration with 3 possible values:

- **IMMEDIATE** means that the propagation is performed immediately.
- **EAGER** means that the propagation is performed lazily but eagerly evaluated before scheduled evaluations.
- **LAZY** means that the propagation is totally lazy and this is default PHREAK behaviour

This means that the following drl:

Example 7.10. A data-driven rule using a passive query

```
query Q (Integer i)
    String( this == i.toString() )
end
rule R @Propagation(IMMEDIATE) when
    $i : Integer()
    ?Q( $i; )
then
    System.out.println( $i );
end
```

will make the rule R to fire if and only if the Integer is inserted after the String, thus behaving in accordance with the semantic of the passive query.

7.6. Commands and the CommandExecutor

The `CommandFactory` allows for commands to be executed on those sessions, the only difference being that the Stateless Knowledge Session executes `fireAllRules()` at the end before disposing the session. The currently supported commands are:

- FireAllRules
- GetGlobal
- SetGlobal
- InsertObject
- InsertElements
- Query
- StartProcess
- BatchExecution

`InsertObject` will insert a single object, with an optional "out" identifier. `InsertElements` will iterate an Iterable, inserting each of the elements. What this means is that a Stateless Knowledge Session is no longer limited to just inserting objects, it can now start processes or execute queries, and do this in any order.

Example 7.11. Insert Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
  ksession.execute( CommandFactory.newInsert( new Cheese( "stilton" ), "stilton_id" ) );
Stilton stilton = bresults.getValue( "stilton_id" );
```

The execute method always returns an `ExecutionResults` instance, which allows access to any command results if they specify an out identifier such as the "stilton_id" above.

Example 7.12. InsertElements Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
Command cmd = CommandFactory.newInsertElements( Arrays.asList( Object[] {
                  new Cheese( "stilton" ),
                  new Cheese( "brie" ),
                  new Cheese( "cheddar" ),
              });
ExecutionResults bresults = ksession.execute( cmd );
```

The execute method only allows for a single command. That's where `BatchExecution` comes in, which represents a composite command, created from a list of commands. Now, execute will iterate over the list and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query, all in a single `execute(...)` call, which is quite powerful.

As mentioned previosly, the StatelessKieSession will execute `fireAllRules()` automatically at the end. However the keen-eyed reader probably has already noticed the `FireAllRules` com-

mand and wondered how that works with a StatelessKieSession. The `FireAllRules` command is allowed, and using it will disable the automatic execution at the end; think of using it as a sort of manual override function.

A custom XStream marshaller can be used with the Drools Pipeline to achieve XML scripting, which is perfect for services. Here are two simple XML samples, one for the BatchExecution and one for the `ExecutionResults`.

Example 7.13. Simple BatchExecution XML

```
<batch-execution>
   <insert out-identifier='outStilton'>
      <org.drools.compiler.Cheese>
         <type>stilton</type>
         <price>25</price>
         <oldPrice>0</oldPrice>
      </org.drools.compiler.Cheese>
   </insert>
</batch-execution>
```

Example 7.14. Simple ExecutionResults XML

```
<execution-results>
   <result identifier='outStilton'>
      <org.drools.compiler.Cheese>
         <type>stilton</type>
         <oldPrice>25</oldPrice>
         <price>30</price>
      </org.drools.compiler.Cheese>
   </result>
</execution-results>
```

Spring and Camel, covered in the integrations book, facilitate declarative services.

Example 7.15. BatchExecution Marshalled to XML

```
<batch-execution>
  <insert out-identifier="stilton">
    <org.drools.compiler.Cheese>
      <type>stilton</type>
      <price>1</price>
      <oldPrice>0</oldPrice>
    </org.drools.compiler.Cheese>
  </insert>
  <query out-identifier='cheeses2' name='cheesesWithParams'>
    <string>stilton</string>
    <string>cheddar</string>
  </query>
```

```
</batch-execution>
```

The `CommandExecutor` returns an `ExecutionResults`, and this is handled by the pipeline code snippet as well. A similar output for the <batch-execution> XML sample above would be:

Example 7.16. ExecutionResults Marshalled to XML

```
<execution-results>
  <result identifier="stilton">
    <org.drools.compiler.Cheese>
      <type>stilton</type>
      <price>2</price>
    </org.drools.compiler.Cheese>
  </result>
  <result identifier='cheeses2'>
    <query-results>
      <identifiers>
        <identifier>cheese</identifier>
      </identifiers>
      <row>
        <org.drools.compiler.Cheese>
          <type>cheddar</type>
          <price>2</price>
          <oldPrice>0</oldPrice>
        </org.drools.compiler.Cheese>
      </row>
      <row>
        <org.drools.compiler.Cheese>
          <type>cheddar</type>
          <price>1</price>
          <oldPrice>0</oldPrice>
        </org.drools.compiler.Cheese>
      </row>
    </query-results>
  </result>
</execution-results>
```

The `BatchExecutionHelper` provides a configured XStream instance to support the marshalling of Batch Executions, where the resulting XML can be used as a message format, as shown above. Configured converters only exist for the commands supported via the Command Factory. The user may add other converters for their user objects. This is very useful for scripting stateless or stateful knowledge sessions, especially when services are involved.

There is currently no XML schema to support schema validation. The basic format is outlined here, and the drools-pipeline module has an illustrative unit test in the `XStreamBatchExecution-Test` unit test. The root element is <batch-execution> and it can contain zero or more commands elements.

Example 7.17. Root XML element

```
<batch-execution>
...
</batch-execution>
```

This contains a list of elements that represent commands, the supported commands is limited to those Commands provided by the Command Factory. The most basic of these is the <insert> element, which inserts objects. The contents of the insert element is the user object, as dictated by XStream.

Example 7.18. Insert

```
<batch-execution>
   <insert>
      ...<!-- any user object -->
   </insert>
</batch-execution>
```

The insert element features an "out-identifier" attribute, demanding that the inserted object will also be returned as part of the result payload.

Example 7.19. Insert with Out Identifier Command

```
<batch-execution>
   <insert out-identifier='userVar'>
      ...
   </insert>
</batch-execution>
```

It's also possible to insert a collection of objects using the <insert-elements> element. This command does not support an out-identifier. The `org.domain.UserClass` is just an illustrative user object that XStream would serialize.

Example 7.20. Insert Elements command

```
<batch-execution>
   <insert-elements>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
      <org.domain.UserClass>
         ...
```

```
      </org.domain.UserClass>
   </insert-elements>
</batch-execution>
```

While the `out` attribute is useful in returning specific instances as a result payload, we often wish to run actual queries. Both parameter and parameterless queries are supported. The `name` attribute is the name of the query to be called, and the `out-identifier` is the identifier to be used for the query results in the `<execution-results>` payload.

Example 7.21. Query Command

```
<batch-execution>
   <query out-identifier='cheeses' name='cheeses'/>
   <query out-identifier='cheeses2' name='cheesesWithParams'>
      <string>stilton</string>
      <string>cheddar</string>
   </query>
</batch-execution>
```

Chapter 8. Rule Language Reference

8.1. Overview

Drools has a "native" rule language. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain. This chapter is mostly concerted with this native rule format. The diagrams used to present the syntax are known as "railroad" diagrams, and they are basically flow charts for the language terms. The technically very keen may also refer to `DRL.g` which is the Antlr3 grammar for the rule language. If you use the Rule Workbench, a lot of the rule structure is done for you with content assistance, for example, type "ru" and press ctrl+space, and it will build the rule structure for you.

8.1.1. A rule file

A rule file is typically a file with a .drl extension. In a DRL file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required) - spreading rules across files can help with managing large numbers of rules. A DRL file is simply a text file.

The overall structure of a rule file is:

Example 8.1. Rules file

```
package package-name

imports

globals

functions

queries

rules
```

The order in which the elements are declared is not important, except for the package name that, if declared, must be the first element in the rules file. All elements are optional, so you will use only those you need. We will discuss each of them in the following sections.

8.1.2. What makes a rule

For the impatient, just as an early view, a rule has the following rough structure:

```
rule "name"
    attributes
    when
        LHS
    then
        RHS
end
```

It's really that simple. Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave. LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

It is important to note that white space is not important, *except* in the case of domain specific languages, where lines are processed one by one and spaces may be significant to the domain language.

8.2. Keywords

Drools 5 introduces the concept of *hard* and *soft* keywords.

Hard keywords are reserved, you cannot use any hard keyword when naming your domain objects, properties, methods, functions and other elements that are used in the rule text.

Here is the list of hard keywords that must be avoided as identifiers when writing rules:

- `true`
- `false`
- `null`

Soft keywords are just recognized in their context, enabling you to use these words in any other place if you wish, although, it is still recommended to avoid them, to avoid confusions, if possible. Here is a list of the soft keywords:

- `lock-on-active`
- `date-effective`
- `date-expires`
- `no-loop`

- `auto-focus`
- `activation-group`
- `agenda-group`
- `ruleflow-group`
- `entry-point`
- `duration`
- `package`
- `import`
- `dialect`
- `salience`
- `enabled`
- `attributes`
- `rule`
- `extend`
- when
- then
- `template`
- `query`
- `declare`
- `function`
- `global`
- `eval`
- `not`
- `in`
- `or`
- `and`
- `exists`

- `forall`
- accumulate
- collect
- from
- `action`
- `reverse`
- `result`
- `end`
- over
- `init`

Of course, you can have these (hard and soft) words as part of a method name in camel case, like notSomething() or accumulateSomething() - there are no issues with that scenario.

Although the 3 hard keywords above are unlikely to be used in your existing domain models, if you absolutely need to use them as identifiers instead of keywords, the DRL language provides the ability to escape hard keywords on rule text. To escape a word, simply enclose it in grave accents, like this:

```
Holiday( `true` == "yes" ) // please note that Drools will resolve that reference to the method
 Holiday.isTrue()
```

8.3. Comments

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks, like the RHS of a rule.

8.3.1. Single line comment

To create single line comments, you can use '//'. The parser will ignore anything in the line after the comment symbol. Example:

```
rule "Testing Comments"when    // this is a single line comment    eval( true ) // this is a
 comment in the same line of a patternthen   // this is a comment inside a semantic code blockend
Com
ments"when    // this is a single line
 comment    eval( true ) // this is a comment in the same line of a

patternthen    // this is a comment inside a semantic code
```

Warning

'#' for comments has been removed.

8.3.2. Multi-line comment

Figure 8.1. Multi-line comment

Multi-line comments are used to comment blocks of text, both in and outside semantic code blocks. Example:

```
rule "Test Multi-line Comments"when    /* this is a multi-line comment       in the left hand
 side of a rule */     eval( true )then     /* and this is a multi-line comment        in the
 right hand side of a rule */end
Com
ments"when    /* this is a multi-line
 comment       in the left hand side of a rule
 */    eval( true

)then    /* and this is a multi-line
 comment       in the right hand side of a rule
 */
```

8.4. Error Messages

Drools 5 introduces standardized error messages. This standardization aims to help users to find and resolve problems in a easier and faster way. In this section you will learn how to identify and interpret those error messages, and you will also receive some tips on how to solve the problems associated with them.

8.4.1. Message format

The standardization includes the error message format and to better explain this format, let's use the following example:

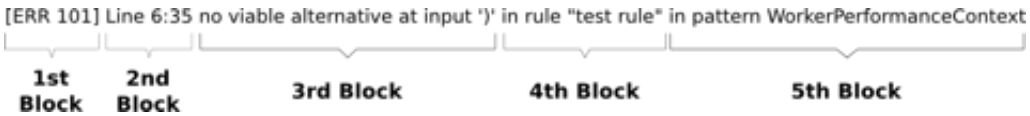


Figure 8.2. Error Message Format

1st Block: This area identifies the error code.

2nd Block: Line and column information.

3rd Block: Some text describing the problem.

4th Block: This is the first context. Usually indicates the rule, function, template or query where the error occurred. This block is not mandatory.

5th Block: Identifies the pattern where the error occurred. This block is not mandatory.

8.4.2. Error Messages Description

8.4.2.1. 101: No viable alternative

Indicates the most common errors, where the parser came to a decision point but couldn't identify an alternative. Here are some examples:

Example 8.2.

```
1: rule one
2:   when
3:     exists Foo()
4:     exits Bar()
5:   then
6: end
```

The above example generates this message:

- [ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one

At first glance this seems to be valid syntax, but it is not (exits != exists). Let's take a look at next example:

Example 8.3.

```
1: package org.drools.examples;2: rule3:   when4:    Object()5:   then6:    System.out.println("A
 RHS");7: end
org.drools.examples;2:
rule3:
when4:
Object()5:
then6:     System.out.println("A
RHS");7:
```

Now the above code generates this message:

- [ERR 101] Line 3:2 no viable alternative at input 'WHEN'

This message means that the parser encountered the token **WHEN**, actually a hard keyword, but it's in the wrong place since the the rule name is missing.

The error "no viable alternative" also occurs when you make a simple lexical mistake. Here is a sample of a lexical problem:

Example 8.4.

```
1: rule simple_rule
2:   when
3:     Student( name == "Andy )
4:   then
5: end
```

The above code misses to close the quotes and because of this the parser generates this error message:

- [ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern Student

Note

Usually the Line and Column information are accurate, but in some cases (like unclosed quotes), the parser generates a 0:-1 position. In this case you should check whether you didn't forget to close quotes, apostrophes or parentheses.

8.4.2.2. 102: Mismatched input

This error indicates that the parser was looking for a particular symbol that it didn't #nd at the current input position. Here are some samples:

Example 8.5.

```
1: rule simple_rule
2:   when
3:     foo3 : Bar(
```

The above example generates this message:

- [ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern Bar

To fix this problem, it is necessary to complete the rule statement.

Note

Usually when you get a 0:-1 position, it means that parser reached the end of source.

The following code generates more than one error message:

Example 8.6.

```
1:  package  org.drools.examples;2:3:  rule  "Avoid  NPE  on  wrong  syntax"4:     when5:
 not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie", price == 15 ) ) from
 $cheeseList )6:   then7:     System.out.println("OK");8: end
org.drools.examples;
2:3: rule "Avoid NPE on wrong
syntax"4:
when5:      not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie", price == 15 )
) from $cheeseList
)6:
then7:
System.out.println("OK");8:
```

These are the errors associated with this source:

- [ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Avoid NPE on wrong syntax" in pattern Cheese
- [ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Avoid NPE on wrong syntax"
- [ERR 102] Line 5:106 mismatched input ')' expecting 'then' in rule "Avoid NPE on wrong syntax"

Note that the second problem is related to the first. To fix it, just replace the commas (',') by AND operator ('&&').

Note

In some situations you can get more than one error message. Try to fix one by one, starting at the first one. Some error messages are generated merely as consequences of other errors.

8.4.2.3. 103: Failed predicate

A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords. This sample shows exactly this situation:

Example 8.7.

```
 1:  package  nesting;  2:  dialect  "mvel"  3:  4:  import  org.drools.compiler.Person  5:  import
 org.drools.compiler.Address 6:  7:
nesting; 2: dialect
 "mvel"
 3: 4: import
 org.drools.compiler.Person 5: import
```

```
 org.drools.compiler.Address
6:  fdsfdsfds
 8:
 9: rule "test something"
10:   when
11:     p: Person( name=="Michael" )
12:   then
13:     p.name = "other";
14:     System.out.println(p.name);
15: end
```

With this sample, we get this error message:

- [ERR 103] Line 7:0 rule 'rule_key' failed predicate: {(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule

The **fdsfdsfds** text is invalid and the parser couldn't identify it as the soft keyword `rule`.

Note

This error is very similar to 102: Mismatched input, but usually involves soft keywords.

8.4.2.4. 104: Trailing semi-colon not allowed

This error is associated with the `eval` clause, where its expression may not be terminated with a semicolon. Check this example:

Example 8.8.

```
1: rule simple_rule
2:   when
3:     eval(abc();)
4:   then
5: end
```

Due to the trailing semicolon within eval, we get this error message:

- [ERR 104] Line 3:4 trailing semi-colon not allowed in rule simple_rule

This problem is simple to fix: just remove the semi-colon.

8.4.2.5. 105: Early Exit

The recognizer came to a subrule in the grammar that must match an alternative at least once, but the subrule did not match anything. Simply put: the parser has entered a branch from where there is no way out. This example illustrates it:

Example 8.9.

```
1: template test_error2:   aa s  11;3: end
test_error2:   aa s
11;3:
```

This is the message associated to the above sample:

- [ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'aa' in template test_error

To fix this problem it is necessary to remove the numeric value as it is neither a valid data type which might begin a new template slot nor a possible start for any other rule file construct.

8.4.3. Other Messages

Any other message means that something bad has happened, so please contact the development team.

8.5. Package

A package is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other - perhaps HR rules, for instance. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). Although, it is not possible to merge into the same package resources declared under different names. A single Rulebase may, however, contain multiple packages built on it. A common structure is to have all the rules for a package in the same file as the package declaration (so that is it entirely self-contained).

The following railroad diagram shows all the components that may make up a package. Note that a package *must* have a namespace and be declared using standard Java conventions for package names; i.e., no spaces, unlike rule names which allow spaces. In terms of the order of elements, they can appear in any order in the rule file, with the exception of the `package` statement, which must be at the top of the file. In all cases, the semicolons are optional.

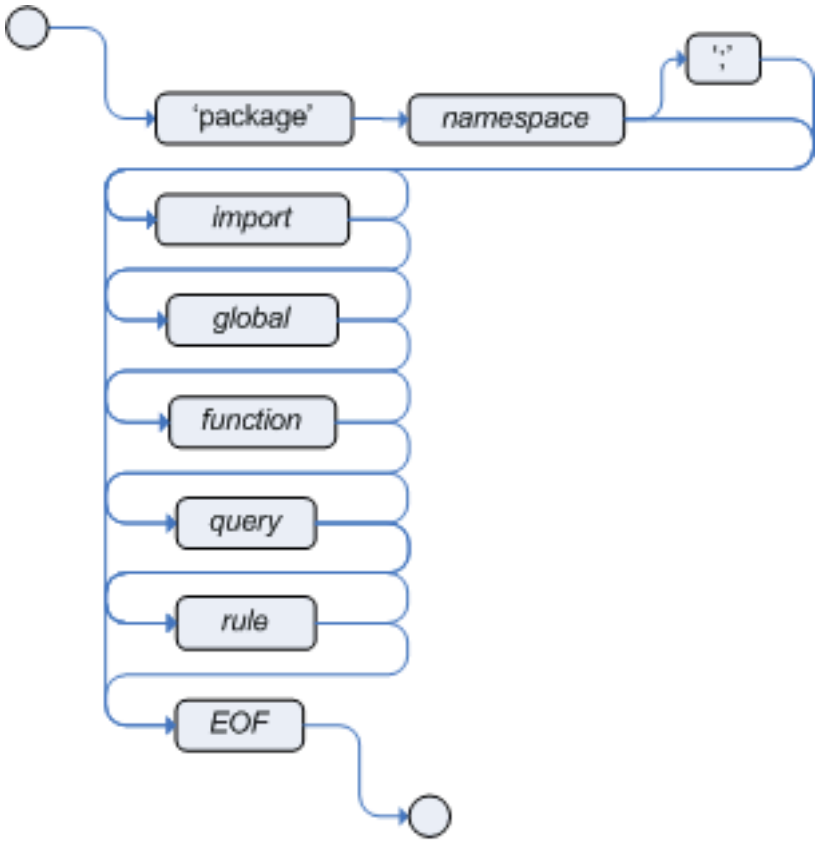


Figure 8.3. package

Notice that any rule attribute (as described the section Rule Attributes) may also be written at package level, superseding the attribute's default value. The modified default may still be replaced by an attribute setting within a rule.

8.5.1. import

Figure 8.4. import

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. Drools automatically imports classes from the Java package of the same name, and also from the package `java.lang`.

8.5.2. global

Figure 8.5. global

With `global` you define global variables. They are used to make application objects available to the rules. Typically, they are used to provide data or services that the rules use, especially application services used in rule consequences, and to return data from the rules, like logs or values added in rule consequences, or for the rules to interact with the application, doing callbacks. Globals are not inserted into the Working Memory, and therefore a global should never be used to establish conditions in rules except when it has a constant immutable value. The engine cannot be notified about value changes of globals and does not track their changes. Incorrect use of globals in constraints may yield surprising results - surprising in a bad way.

If multiple packages declare globals with the same identifier they must be of the same type and all of them will reference the same global value.

In order to use globals you must:

1. Declare your global variable in your rules file and use it in rules. Example:

```
global java.util.List myGlobalList;rule "Using a global"when      eval( true )then
 myGlobalList.add( "Hello World" );end

myGlobalList;rule "Using

a global"when    eval(

true )then    myGlobalList.add( "Hello
```

2. Set the global value on your working memory. It is a best practice to set all global values before asserting any fact to the working memory. Example:

```
List list = new ArrayList();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

Note that these are just named instances of objects that you pass in from your application to the working memory. This means you can pass in any object you want: you could pass in a service locator, or perhaps a service itself. With the new `from` element it is now common to pass a Hibernate session as a global, to allow `from` to pull data from a named Hibernate query.

One example may be an instance of a Email service. In your integration code that is calling the rule engine, you obtain your emailService object, and then set it in the working memory. In the DRL, you declare that you have a global of type EmailService, and give it the name "email". Then in your rule consequences, you can use things like email.sendSMS(number, message).

Globals are not designed to share data between rules and they should never be used for that purpose. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory.

Care must be taken when changing data held by globals because the rule engine is not aware of those changes, hence cannot react to them.

8.6. Function

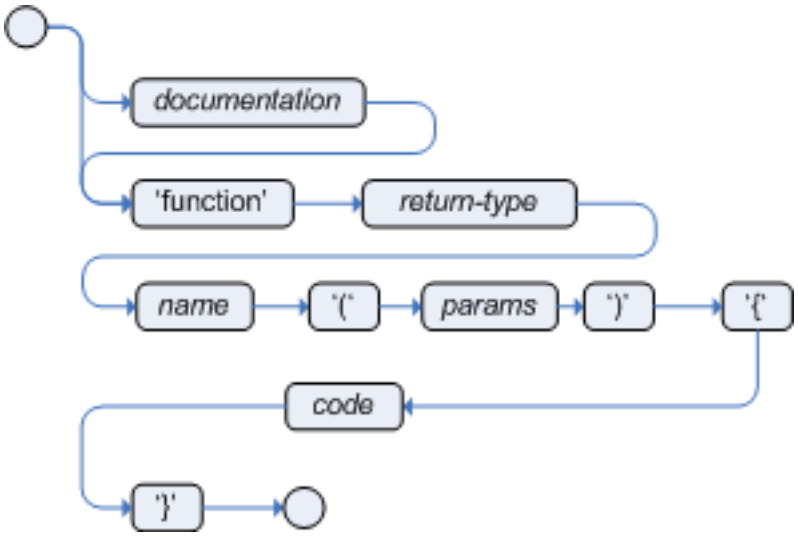


Figure 8.6. function

Functions are a way to put semantic code in your rule source file, as opposed to in normal Java classes. They can't do anything more than what you can do with helper classes. (In fact, the compiler generates the helper class for you behind the scenes.) The main advantage of using functions in a rule is that you can keep the logic all in one place, and you can change the functions as needed (which can be a good or a bad thing). Functions are most useful for invoking actions on the consequence (`then`) part of a rule, especially if that particular action is used over and over again, perhaps with only differing parameters for each rule.

A typical function declaration looks like:

```
function String hello(String name) {    return "Hello "+name+"!";}
{    return "Hello "+name
+"!";
```

Note that the `function` keyword is used, even though its not really part of Java. Parameters to the function are defined as for a method, and you don't have to have parameters if they are not needed. The return type is defined just like in a regular method.

Alternatively, you could use a static method in a helper class, e.g., `Foo.hello()`. Drools supports the use of function imports, so all you would need to do is:

```
import function my.package.Foo.hello
```

Irrespective of the way the function is defined or imported, you use a function by calling it by its name, in the consequence or inside a semantic code block. Example:

```
rule "using a static function"when    eval( true )then   System.out.println( hello( "Bob" ) );end
ic
function"when     eval(

true )then    System.out.println( hello( "Bob"
```

8.7. Type Declaration

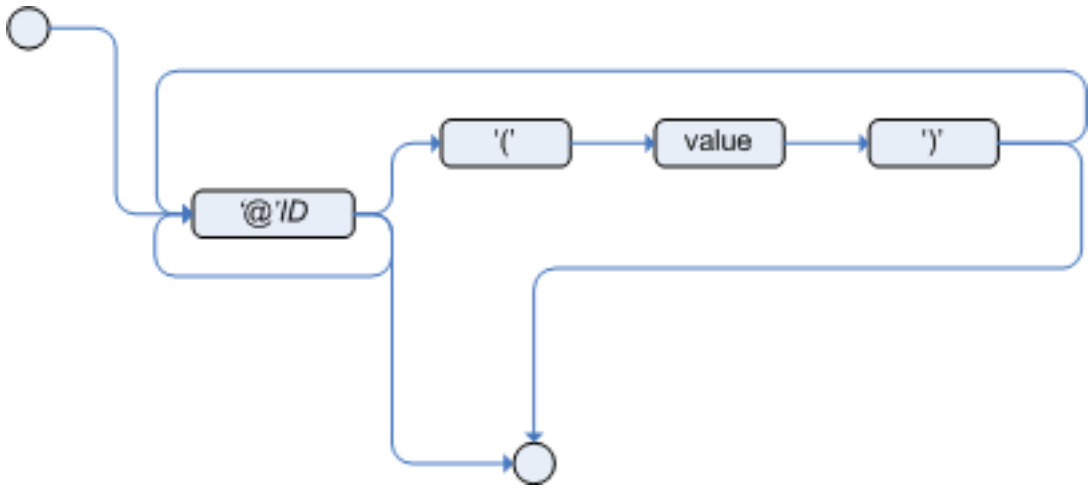


Figure 8.7. meta_data

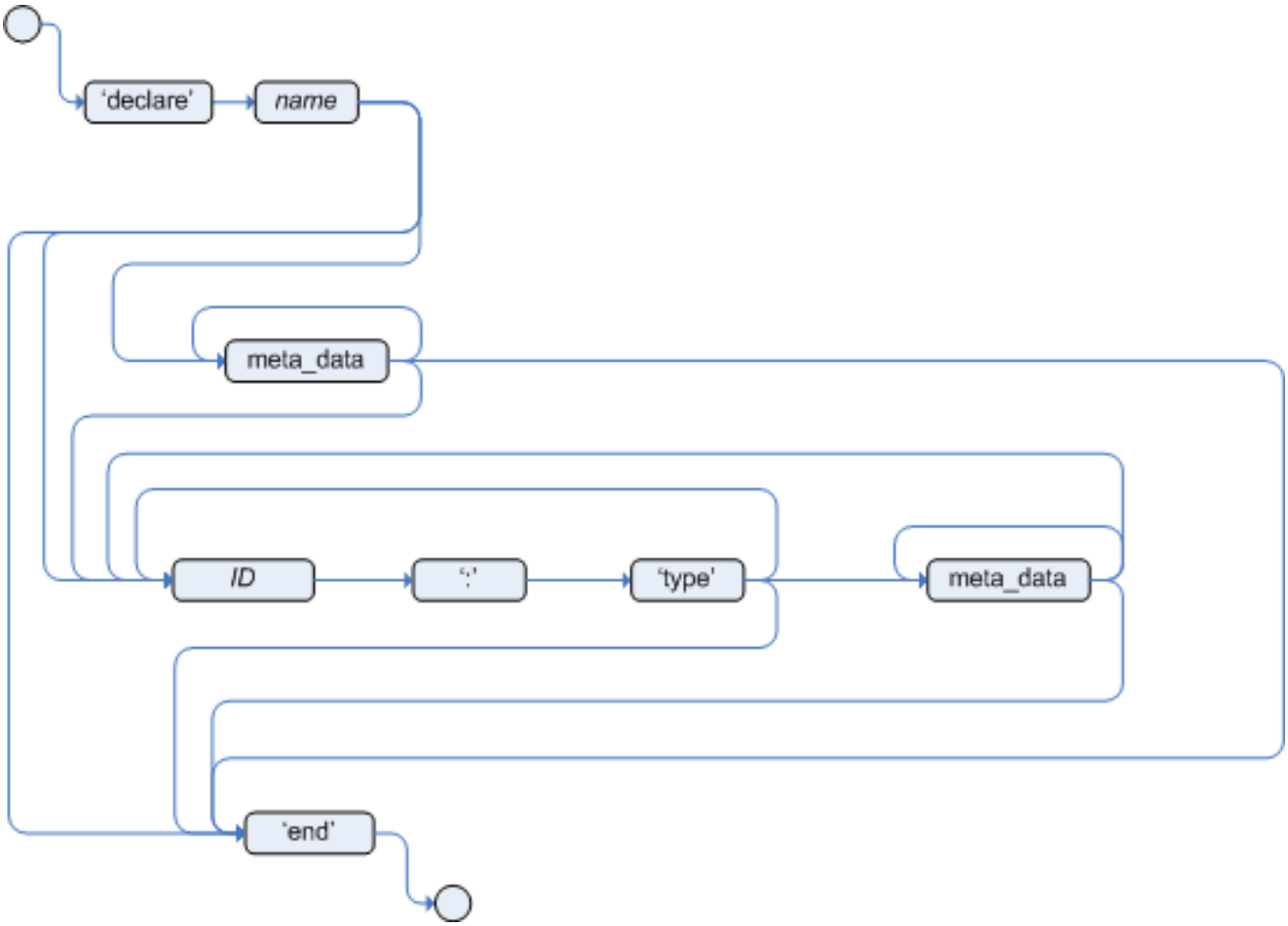


Figure 8.8. type_declaration

Type declarations have two main goals in the rules engine: to allow the declaration of new types, and to allow the declaration of metadata for types.

- **Declaring new types:** Drools works out of the box with plain Java objects as facts. Sometimes, however, users may want to define the model directly to the rules engine, without worrying about creating models in a lower level language like Java. At other times, there is a domain model already built, but eventually the user wants or needs to complement this model with additional entities that are used mainly during the reasoning process.
- **Declaring metadata:** facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

8.7.1. Declaring New Types

To declare a new type, all you need to do is use the keyword `declare`, followed by the list of fields, and the keyword `end`. A new fact must have a list of fields, otherwise the engine will look for an existing fact class in the classpath and raise an error if not found.

Example 8.10. Declaring a new fact type: Address

```
declare Address   number : int   streetName : String   city : String
dress   number :
int   streetName :
String   city :
end
```

The previous example declares a new fact type called `Address`. This fact type will have three attributes: `number`, `streetName` and `city`. Each attribute has a type that can be any valid Java type, including any other class created by the user or even other fact types previously declared.

For instance, we may want to declare another fact type `Person`:

Example 8.11. declaring a new fact type: Person

```
declare Person    name : String    dateOfBirth : java.util.Date    address : Address
son    name :
String    dateOfBirth :
java.util.Date    address :
end
```

As we can see on the previous example, `dateOfBirth` is of type `java.util.Date`, from the Java API, while `address` is of the previously defined fact type Address.

You may avoid having to write the fully qualified name of a class every time you write it by using the `import` clause, as previously discussed.

Example 8.12. Avoiding the need to use fully qualified class names by using import

```
import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

When you declare a new fact type, Drools will, at compile time, generate bytecode that implements a Java class representing the fact type. The generated Java class will be a one-to-one Java Bean mapping of the type definition. So, for the previous example, the generated Java class would be:

Example 8.13. generated Java class for the previous Person fact type declaration

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // empty constructor
    public Person() {...}

    // constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // if keys are defined, constructor with keys
    public Person( ...keys... ) {...}

    // getters and setters
    // equals/hashCode
    // toString
}
```

Since the generated class is a simple Java class, it can be used transparently in the rules, like any other fact.

Example 8.14. Using the declared types in rules

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    // Insert Mark, who is Bob's mate.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

8.7.1.1. Declaring enumerative types

DRL also supports the declaration of enumerative types. Such type declarations require the additional keyword *enum*, followed by a comma separated list of admissible values terminated by a semicolon.

Example 8.15.

```
declare enum DaysOfWeek
   SUN,MON,TUE,WED,THU,FRI,SAT;
```

```
end
```

The compiler will generate a valid Java enum, with static methods valueOf() and values(), as well as instance methods ordinal(), compareTo() and name().

Complex enums are also partially supported, declaring the internal fields similarly to a regular type declaration. Notice that as of version 6.x, enum fields do *NOT* support other declared types or enums

Example 8.16.

```
declare enum DaysOfWeek

 SUN("Sunday"),MON("Monday"),TUE("Tuesday"),WED("Wednesday"),THU("Thursday"),FRI("Friday"),SAT("Saturday");

   fullName : String
end
```

Enumeratives can then be used in rules

Example 8.17. Using declarative enumerations in rules

```
rule "Using a declared Enum"
when
   $p : Employee( dayOff == DaysOfWeek.MONDAY )
then
   ...
end
```

8.7.2. Declaring Metadata

Metadata may be assigned to several different constructions in Drools: fact types, fact attributes and rules. Drools uses the at sign ('@') to introduce metadata, and it always uses the form:

```
@metadata_key( metadata_value )
```

The parenthesized *metadata_value* is optional.

For instance, if you want to declare a metadata attribute like `author`, whose value is *Bob*, you could simply write:

Example 8.18. Declaring a metadata attribute

```
@author( Bob )
```

Drools allows the declaration of any arbitrary metadata attribute, but some will have special meaning to the engine, while others are simply available for querying at runtime. Drools allows the declaration of metadata both for fact types and for fact attributes. Any metadata that is declared before the attributes of a fact type are assigned to the fact type, while metadata declared after an attribute are assigned to that particular attribute.

Example 8.19. Declaring metadata attributes for fact types and attributes

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

In the previous example, there are two metadata items declared for the fact type (`@author` and `@dateOfCreation`) and two more defined for the name attribute (`@key` and `@maxLength`). Please note that the `@key` metadata has no required value, and so the parentheses and the value were omitted.:

8.7.2.1. Predefined class level annotations

Some annotations have predefined semantics that are interpreted by the engine. The following is a list of some of these predefined annotations and their meaning.

8.7.2.1.1. @role(<fact | event>)

The @role annotation defines how the engine should handle instances of that type: either as regular facts or as events. It accepts two possible values:

- fact : this is the default, declares that the type is to be handled as a regular fact.
- event : declares that the type is to be handled as an event.

The following example declares that the fact type StockTick in a stock broker application is to be handled as an event.

Example 8.20. declaring a fact type as an event

```
import some.package.StockTick

declare StockTick
    @role( event )
end
```

The same applies to facts declared inline. If StockTick was a fact type declared in the DRL itself, instead of a previously existing class, the code would be:

Example 8.21. declaring a fact type and assigning it the event role

```
declare StockTick
    @role( event )

    datetime : java.util.Date
    symbol : String
    price : double
end
```

8.7.2.1.2. @typesafe(<boolean>)

By default all type declarations are compiled with type safety enabled; @typesafe(false) provides a means to override this behaviour by permitting a fall-back, to type unsafe evaluation where all constraints are generated as MVEL constraints and executed dynamically. This can be important when dealing with collections that do not have any generics or mixed type collections.

8.7.2.1.3. @timestamp(<attribute name>)

Every event has an associated timestamp assigned to it. By default, the timestamp for a given event is read from the Session Clock and assigned to the event at the time the event is inserted into the working memory. Although, sometimes, the event has the timestamp as one of its own attributes. In this case, the user may tell the engine to use the timestamp from the event's attribute instead of reading it from the Session Clock.

```
@timestamp( <attributeName> )
```

To tell the engine what attribute to use as the source of the event's timestamp, just list the attribute name as a parameter to the @timestamp tag.

Example 8.22. declaring the VoiceCall timestamp attribute

```
declare VoiceCall
```

```
Call   @role( event )
    @timestamp( callDateTime )
end
```

8.7.2.1.4. @duration(<attribute name>)

Drools supports both event semantics: point-in-time events and interval-based events. A point-in-time event is represented as an interval-based event whose duration is zero. By default, all events have duration zero. The user may attribute a different duration for an event by declaring which attribute in the event type contains the duration of the event.

```
@duration( <attributeName> )
```

So, for our VoiceCall fact type, the declaration would be:

Example 8.23. declaring the VoiceCall duration attribute

```
declare VoiceCall
Call   @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
end
```

8.7.2.1.5. @expires(<time interval>)

Important

This tag is only considered when running the engine in STREAM mode. Also, additional discussion on the effects of using this tag is made on the Memory Management section. It is included here for completeness.

Events may be automatically expired after some time in the working memory. Typically this happens when, based on the existing rules in the knowledge base, the event can no longer match and activate any rules. Although, it is possible to explicitly define when an event should expire.

```
@expires( <timeOffset> )
```

The value of *timeOffset* is a temporal interval in the form:

```
[#d][#h][#m][#s][#[ms]]
```

Where *[ ]* means an optional parameter and # means a numeric value.

So, to declare that the VoiceCall facts should be expired after 1 hour and 35 minutes after they are inserted into the working memory, the user would write:

Example 8.24. declaring the expiration offset for the VoiceCall events

```
declare VoiceCall
Call   @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
    @expires( 1h35m )
end
```

The @expires policy will take precedence and override the implicit expiration offset calculated from temporal constraints and sliding windows in the knowledge base.

8.7.2.1.6. @propertyChangeSupport

Facts that implement support for property changes as defined in the Javabean(tm) spec, now can be annotated so that the engine register itself to listen for changes on fact properties. The boolean parameter that was used in the insert() method in the Drools 4 API is deprecated and does not exist in the drools-api module.

Example 8.25. @propertyChangeSupport

```
declare Person
son @propertyChangeSupport
end
```

8.7.2.1.7. @propertyReactive

Make this type property reactive. See Fine grained property change listeners section for details.

8.7.2.2. Predefined attribute level annotations

As noted before, Drools also supports annotations in type attributes. Here is a list of predefined attribute annotations.

8.7.2.2.1. @key

Declaring an attribute as a key attribute has 2 major effects on generated types:

1. The attribute will be used as a key identifier for the type, and as so, the generated class will implement the equals() and hashCode() methods taking the attribute into account when comparing instances of this type.

2. Drools will generate a constructor using all the key attributes as parameters.
For instance:

Example 8.26. example of @key declarations for a type

```
declare Person    firstName : String @key    lastName : String @key    age : int
son    firstName : String
@key    lastName : String
@key    age :
end
```

For the previous example, Drools will generate equals() and hashCode() methods that will check the firstName and lastName attributes to determine if two instances of Person are equal to each other, but will not check the age attribute. It will also generate a constructor taking firstName and lastName as parameters, allowing one to create instances with a code like this:

Example 8.27. creating an instance using the key constructor

```
Person person = new Person( "John", "Doe" );
```

8.7.2.2.2. @position

Patterns support positional arguments on type declarations.

Positional arguments are ones where you don't need to specify the field name, as the position maps to a known named field. i.e. Person(name == "mark") can be rewritten as Person("mark";). The semicolon ';' is important so that the engine knows that everything before it is a positional argument. Otherwise we might assume it was a boolean expression, which is how it could be interpreted after the semicolon. You can mix positional and named arguments on a pattern by using the semicolon ';' to separate them. Any variables used in a positional that have not yet been bound will be bound to the field that maps to that position.

```
declare Cheese
    name : String
    shop : String
    price : int
end
```

The default order is the declared order, but this can be overridden using @position

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
```

```
end
```

The @Position annotation, in the org.drools.definition.type package, can be used to annotate original pojos on the classpath. Currently only fields on classes can be annotated. Inheritance of classes is supported, but not interfaces of methods yet.

Example patterns, with two constraints and a binding. Remember semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in positional arguments, but not variables.

```
Cheese( "stilton", "Cheese Shop", p; )
Cheese( "stilton", "Cheese Shop"; p : price )
Cheese( "stilton"; shop == "Cheese Shop", p : price )
Cheese( name == "stilton"; shop == "Cheese Shop", p : price )
```

@Position is inherited when beans extend each other; while not recommended, two fields may have the same @position value, and not all consecutive values need be declared. If a @position is repeated, the conflict is solved using inheritance (fields in the superclass have the precedence) and the declaration order. If a @position value is missing, the first field without an explicit @position (if any) is selected to fill the gap. As always, conflicts are resolved by inheritance and declaration order.

```
declare Cheese
    name : String
    shop : String @position(2)
    price : int @position(0)
end

declare SeasonedCheese extends Cheese
    year : Date @position(0)
    origin : String @position(6)
    country : String
end
```

In the example, the field order would be : price (@position 0 in the superclass), year (@position 0 in the subclass), name (first field with no @position), shop (@position 2), country (second field without @position), origin.

8.7.3. Declaring Metadata for Existing Types

Drools allows the declaration of metadata attributes for existing types in the same way as when declaring metadata attributes for new fact types. The only difference is that there are no fields in that declaration.

For instance, if there is a class org.drools.examples.Person, and one wants to declare metadata for it, it's possible to write the following code:

Example 8.28. Declaring metadata for an existing type

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

Instead of using the import, it is also possible to reference the class by its fully qualified name, but since the class will also be referenced in the rules, it is usually shorter to add the import and use the short class name everywhere.

Example 8.29. Declaring metadata using the fully qualified class name

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

8.7.4. Parametrized constructors for declared types

Generate constructors with parameters for declared types.

Example: for a declared type like the following:

```
declare Person    firstName : String
son    firstName : String@key
    lastName : String @key
    age : int
end
```

The compiler will implicitly generate 3 constructors: one without parameters, one with the @key fields, and one with all fields.

```
Person() // parameterless constructorPerson( String firstName, String lastName )Person( String
 firstName, String lastName, int age )
meterless constructorPerson( String
firstName, String lastName )Person( String firstName, String
```

8.7.5. Non Typesafe Classes

@typesafe(<boolean>) has been added to type declarations. By default all type declarations are compiled with type safety enabled; @typesafe(false) provides a means to override this behaviour

by permitting a fall-back, to type unsafe evaluation where all constraints are generated as MVEL constraints and executed dynamically. This can be important when dealing with collections that do not have any generics or mixed type collections.

8.7.6. Accessing Declared Types from the Application Code

Declared types are usually used inside rules files, while Java models are used when sharing the model between rules and applications. Although, sometimes, the application may need to access and handle facts from the declared types, especially when the application is wrapping the rules engine and providing higher level, domain specific user interfaces for rules management.

In such cases, the generated classes can be handled as usual with the Java Reflection API, but, as we know, that usually requires a lot of work for small results. Therefore, Drools provides a simplified API for the most common fact handling the application may want to do.

The first important thing to realize is that a declared fact will belong to the package where it was declared. So, for instance, in the example below, `Person` will belong to the `org.drools.examples` package, and so the fully qualified name of the generated class will be `org.drools.examples.Person`.

Example 8.30. Declaring a type in the org.drools.examples package

```
package org.drools.examples

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

Declared types, as discussed previously, are generated at knowledge base compilation time, i.e., the application will only have access to them at application run time. Therefore, these classes are not available for direct reference from the application.

Drools then provides an interface through which users can handle declared types from the application code: `org.drools.definition.type.FactType`. Through this interface, the user can instantiate, read and write fields in the declared fact types.

Example 8.31. Handling declared fact types through the API

```
// get a reference to a knowledge base with a declared type:KieBase kbase = ...// get the declared
 FactTypeFactType personType = kbase.getFactType( "org.drools.examples",
                   "Person" );// handle the type as necessary:// create instances:Object bob =
 personType.newInstance();// set attributes valuespersonType.set( bob,               "name",
           "Bob" );personType.set( bob,              "age",              42 );// insert fact
 into a sessionKieSession ksession = ...ksession.insert( bob );ksession.fireAllRules();// read
 attributesString name = personType.get( bob, "name" );int age = personType.get( bob, "age" );
```

```
declared type:KieBase kbase

 = ...// get the
 declared FactTypeFactType personType =
 kbase.getFactType( "org.drools.examples",

 "Person" );// handle the type
 as necessary://
 create instances:Object bob

 = personType.newInstance();// set
 attributes
valuespersonType.set( bob,
   "name",
 "Bob"
);personType.set( bob,
   "age",

 42 );// insert fact into
 a sessionKieSession ksession
 = ...ksession.insert(


bob );ksession.fireAllRules();//
 read attributesString name = personType.get( bob,
 "name" );int age = personType.get( bob,
```

The API also includes other helpful methods, like setting all the attributes at once, reading values from a Map, or reading all attributes at once, into a Map.

Although the API is similar to Java reflection (yet much simpler to use), it does not use reflection underneath, relying on much more performant accessors implemented with generated bytecode.

8.7.7. Type Declaration 'extends'

Type declarations now support 'extends' keyword for inheritance

In order to extend a type declared in Java by a DRL declared subtype, repeat the supertype in a declare statement without any fields.

```
b org.people.Person

declare Person end

declare Student extends Person
    school : String
end

declare LongTermStudent extends Student
    years : int
    course : String
end
```

8.7.8. Traits

WARNING : this feature is still experimental and subject to changes

The same fact may have multiple dynamic types which do not fit naturally in a class hierarchy. Traits allow to model this very common scenario. A trait is an interface that can be applied (and eventually removed) to an individual object at runtime. To create a trait rather than a traditional bean, one has to declare them explicitly as in the following example:

Example 8.32.

```
declare trait GoldenCustomer    // fields will map to getters/setters    code     : String
  balance  : long    discount : int    maxExpense : long
enCustomer    // fields will map to
 getters/setters    code
: String    balance
: long    discount
: int    maxExpense
end
```

At runtime, this declaration results in an interface, which can be used to write patterns, but can not be instantiated directly. In order to apply a trait to an object, we provide the new don keyword, which can be used as simply as this:

Example 8.33.

```
when
    $c : Customer()
then
    GoldenCustomer gc = don( $c, GoldenCustomer.class );
end
```

when a core object dons a trait, a proxy class is created on the fly (one such class will be generated lazily for each core/trait class combination). The proxy instance, which wraps the core object and implements the trait interface, is inserted automatically and will possibly activate other rules. An immediate advantage of declaring and using interfaces, getting the implementation proxy for free from the engine, is that multiple inheritance hierarchies can be exploited when writing rules. The core classes, however, need not implement any of those interfaces statically, also facilitating the use of legacy classes as cores. In fact, any object can don a trait, provided that they are declared as @Traitable. Notice that this annotation used to be optional, but now is mandatory.

Example 8.34.

```
import org.drools.core.factmodel.traits.Traitable;
declare Customer
```

```
    @Traitable
    code    : String
    balance : long
end
```

The only connection between core classes and trait interfaces is at the proxy level: a trait is not specifically tied to a core class. This means that the same trait can be applied to totally different objects. For this reason, the trait does not transparently expose the fields of its core object. So, when writing a rule using a trait interface, only the fields of the interface will be available, as usual. However, any field in the interface that corresponds to a core object field, will be mapped by the proxy class:

Example 8.35.

```
when
    $o: OrderItem( $p : price, $code : custCode )
    $c: GoldenCustomer( code == $code, $a : balance, $d: discount )
then
    $c.setBalance( $a - $p*$d );
end
```

In this case, the code and balance would be read from the underlying Customer object. Likewise, the setAccount will modify the underlying object, preserving a strongly typed access to the data structures. A hard field must have the same name and type both in the core class and all donned interfaces. The name is used to establish the mapping: if two fields have the same name, then they must also have the same declared type. The annotation @org.drools.core.factmodel.traits.Alias allows to relax this restriction. If an @Alias is provided, its value string will be used to resolve mappings instead of the original field name. @Alias can be applied both to traits and core beans.

Example 8.36.

```
import org.drools.core.factmodel.traits.*;
declare trait GoldenCustomer
    balance : long @Alias( "org.acme.foo.accountBalance" )
end

declare Person
    @Traitable
    name : String
    savings : long @Alias( "org.acme.foo.accountBalance" )
end

when
    GoldenCustomer( balance > 1000 ) // will react to new Person( 2000 )
then
end
```

More work is being done on reaxing this constraint (see the experimental section on "logical" traits later). Now, one might wonder what happens when a core class does NOT provide the implementation for a field defined in an interface. We call hard fields those trait fields which are also core fields and thus readily available, while we define soft those fields which are NOT provided by the core class. Hidden fields, instead, are fields in the core class not exposed by the interface.

So, while hard field management is intuitive, there remains the problem of soft and hidden fields. Hidden fields are normally only accessible using the core class directly. However, the "fields" Map can be used on a trait interface to access a hidden field. If the field can't be resolved, null will be returned. Notice that this feature is likely to change in the future.

Example 8.37.

```
when
     $sc : GoldenCustomer( fields[ "age" ] > 18 )  // age is declared by the underlying core
 class, but not by GoldenCustomer
then
```

Soft fields, instead, are stored in a Map-like data structure that is specific to each core object and referenced by the proxy(es), so that they are effectively shared even when an object dons multiple traits.

Example 8.38.

```
when
    $sc : GoldenCustomer( $c : code, // hard getter
                          $maxExpense : maxExpense > 1000 // soft getter
    )
then
    $sc.setDiscount( ... ); // soft setter
end
```

A core object also holds a reference to all its proxies, so that it is possible to track which type(s) have been added to an object, using a sort of dynamic "instanceof" operator, which we called isA. The operator can accept a String, a class literal or a list of class literals. In the latter case, the constraint is satisfied only if all the traits have been donned.

Example 8.39.

```
$sc : GoldenCustomer( $maxExpense : maxExpense > 1000,                          this isA
 "SeniorCustomer", this isA [ NationalCustomer.class, OnlineCustomer.class ])
maxExpense > 1000,                              this isA "SeniorCustomer",
this isA
[
```

Eventually, the business logic may require that a trait is removed from a wrapped object. To this end, we provide two options. The first is a "logical don", which will result in a logical insertion of the proxy resulting from the traiting operation. The TMS will ensure that the trait is removed when its logical support is removed in the first place.

Example 8.40.

```
then
    don( $x, // core object
         Customer.class, // trait class
         true // optional flag for logical insertion
    )
```

The second is the use of the "shed" keyword, which causes the removal of any type that is a subtype (or equivalent) of the one passed as an argument. Notice that, as of version 5.5, shed would only allow to remove a single specific trait.

Example 8.41.

```
then
    Thing t = shed( $x, GoldenCustomer.class ) // also removes any trait that
```

This operation returns another proxy implementing the org.drools.core.factmodel.traits.Thing interface, where the getFields() and getCore() methods are defined. Internally, in fact, all declared traits are generated to extend this interface (in addition to any others specified). This allows to preserve the wrapper with the soft fields which would otherwise be lost.

A trait and its proxies are also correlated in another way. Starting from version 5.6, whenever a core object is "modified", its proxies are "modified" automatically as well, to allow trait-based patterns to react to potential changes in hard fields. Likewise, whenever a trait proxy (mached by a trait pattern) is modified, the modification is propagated to the core class and the other traits. Morover, whenever a don operation is performed, the core object is also modified automatically, to reevaluate any "isA" operation which may be triggered.

Potentially, this may result in a high number of modifications, impacting performance (and correctness) heavily. So two solutions are currently implemented. First, whenever a core object is modified, only the most specific traits (in the sense of inheritance between trait interfaces) are updated and an internal blocking mechanism is in place to ensure that each potentially matching pattern is evaluated once and only once. So, in the following situation:

```
declare trait GoldenCustomer end
declare trait NationalGoldenustomer extends GoldenCustomer end
declare trait SeniorGoldenCustomer extends GoldenCustomer end
```

a modification of an object that is both a GoldenCustomer, a NationalGoldenCustomer and a SeniorGoldenCustomer wold cause only the latter two proxies to be actually modified. The first would match any pattern for GoldenCustomer and NationalGoldenCustomer; the latter would instead be prevented from rematching GoldenCustomer, but would be allowed to match SeniorGoldenCustomer patterns. It is not necessary, instead, to modify the GoldenCustomer proxy since it is already covered by at least one other more specific trait.

The second method, up to the usr, is to mark traits as @PropertyReactive. Property reactivity is trait-enabled and takes into account the trait field mappings, so to block unnecessary propagations.

8.7.8.1. Cascading traits

WARNING : This feature is extremely experimental and subject to changes

Normally, a hard field must be exposed with its original type by all traits donned by an object, to prevent situations such as

Example 8.42.

```
declare Person  @Traitable  name : String  id : String
son
@Traitable  name :
String  id :
end

declare trait Customer
  id : String
end

declare trait Patient
  id : long  // Person can't don Patient, or an exception will be thrown
end
```

Should a Person don both Customer and Patient, the type of the hard field id would be ambiguous. However, consider the following example, where GoldenCustomers refer their best friends so that they become Customers as well:

Example 8.43.

```
declare Person  @Traitable( logical=true )  bestFriend : Person
son  @Traitable( logical=true
)  bestFriend :
end

declare trait Customer end

declare trait GoldenCustomer extends Customer
  refers : Customer @Alias( "bestFriend" )
```

```
end
```

Aside from the @Alias, a Person-as-GoldenCustomer's best friend might be compatible with the requirements of the trait GoldenCustomer, provided that they are some kind of Customer themselves. Marking a Person as "logically traitable" - i.e. adding the annotation @Traitable(logical = true) - will instruct the engine to try and preserve the logical consistency rather than throwing an exception due to a hard field with different type declarations (Person vs Customer). The following operations would then work:

Example 8.44.

```
Person p1 = new Person();Person p2 = new Person();p1.setBestFriend( p2 );...Customer
    c2 = don( p2, Customer.class );...GoldenCustomer gc1 = don( p1,
 GoldenCustomer.class );...p1.getBestFriend(); // returns p2gc1.getRefers(); // returns c2, a
 Customer proxy wrapping p2
Person();Person p2 = new
 Person();p1.setBestFriend( p2

);...Customer c2 = don( p2, Customer.class

);...GoldenCustomer gc1 = don( p1, GoldenCustomer.class

);...p1.getBestFriend(); // returns
 p2gc1.getRefers(); // returns c2, a Customer proxy wrapping
```

Notice that, by the time p1 becomes GoldenCustomer, p2 must have already become a Customer themselves, otherwise a runtime exception will be thrown since the very definition of GoldenCustomer would have been violated.

In some cases, however, one might want to infer, rather than verify, that p2 is a Customer by virtue that p1 is a GoldenCustomer. This modality can be enabled by marking Customer as "logical", using the annotation @org.drools.core.factmodel.traits.Trait(logical = true). In this case, should p2 not be a Customer by the time that p1 becomes a GoldenCustomer, it will be automatically don the trait Customer to preserve the logical integrity of the system.

Notice that the annotation on the core class enables the dynamic type management for its fields, whereas the annotation on the traits determines whether they will be enforced as integrity constraints or cascaded dynamically.

Example 8.45.

```
import org.drools.factmodel.traits.*;

declare trait Customer
    @Trait( logical = true )
end
```

8.8. Rule

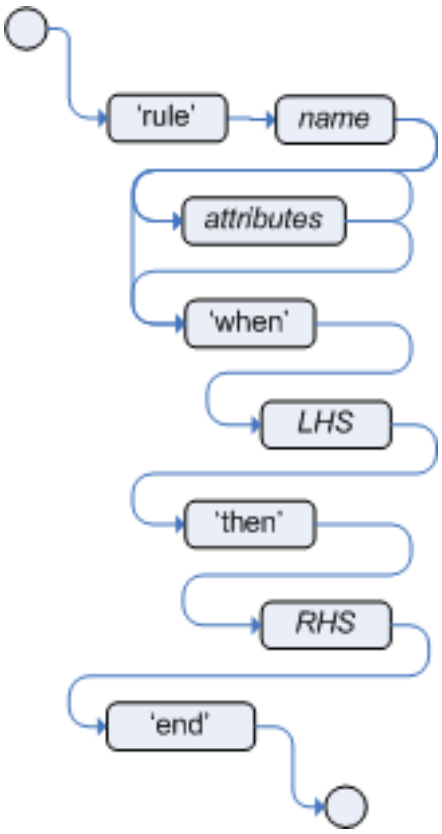


Figure 8.9. rule

A rule specifies that *when* a particular set of conditions occur, specified in the Left Hand Side (LHS), *then* do what queryis specified as a list of actions in the Right Hand Side (RHS). A common question from users is "Why use when instead of if?" "When" was chosen over "if" because "if" is normally part of a procedural execution flow, where, at a specific point in time, a condition is to be checked. In contrast, "when" indicates that the condition evaluation is not tied to a specific evaluation sequence or point in time, but that it happens continually, at any time during the life time of the engine; whenever the condition is met, the actions are executed.

A rule must have a name, unique within its rule package. If you define a rule twice in the same DRL it produces an error while loading. If you add a DRL that includes a rule name already in the package, it replaces the previous rule. If a rule name is to have spaces, then it will need to be enclosed in double quotes (it is best to always use double quotes).

Attributes - described below - are optional. They are best written one per line.

The LHS of the rule follows the `when` keyword (ideally on a new line), similarly the RHS follows the `then` keyword (again, ideally on a newline). The rule is terminated by the keyword `end`. Rules cannot be nested.

Example 8.46. Rule Syntax Overview

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

Example 8.47. A simple rule

```
rule "Approve if not rejected"  salience -100   agenda-group "approval"     when         not
 Rejection()        p : Policy(approved == false, policyState:status)       exists Driver(age >
 25)       Process(status == policyState)    then       log("APPROVED: due to no objections.");
        p.setApproved(true);end
rejected"  salience
 -100   agenda-
group "approval"
  when        not
 Rejection()         p : Policy(approved ==
false, policyState:status)        exists Driver(age
> 25)        Process(status
== policyState)
  then        log("APPROVED: due to no
 objections.");
```

8.8.1. Rule Attributes

Rule attributes provide a declarative way to influence the behavior of the rule. Some are quite simple, while others are part of complex subsystems such as ruleflow. To get the most from Drools you should make sure you have a proper understanding of each attribute.

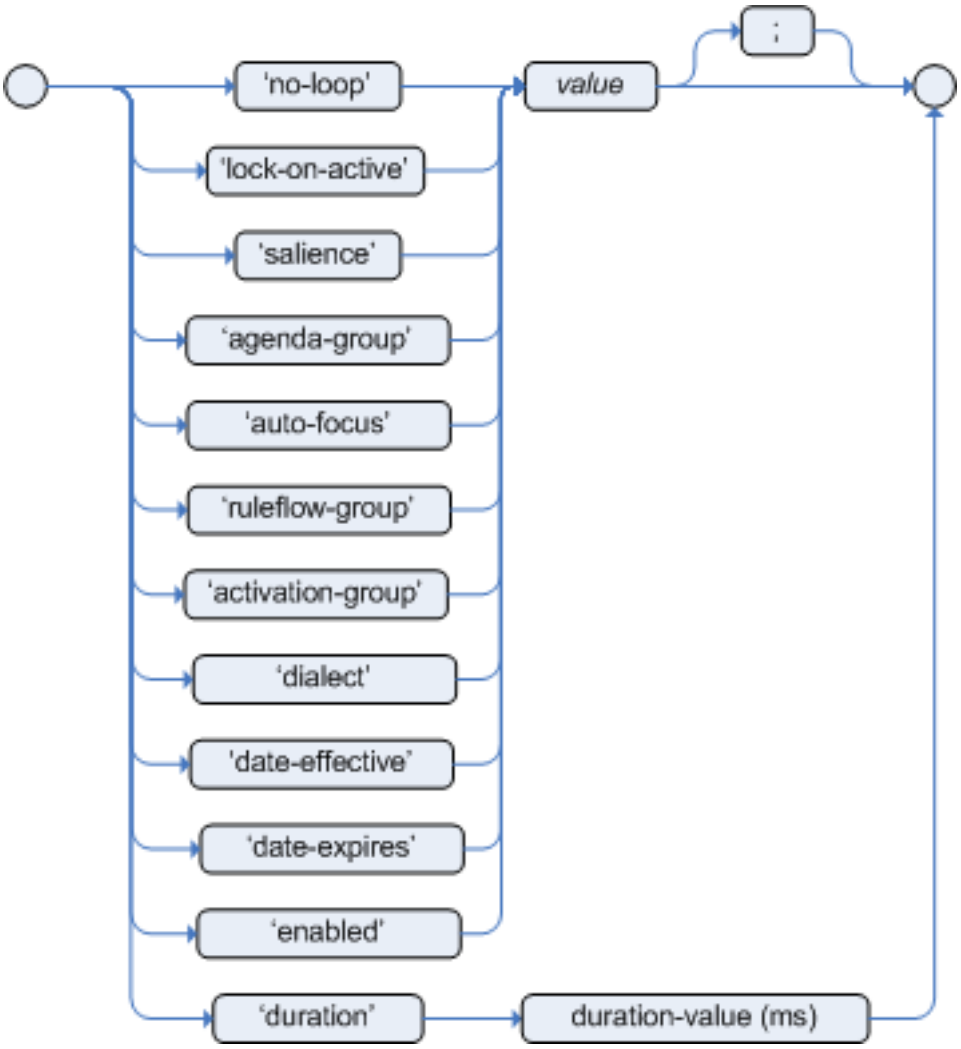


Figure 8.10. rule attributes

`no-loop`
default value: `false`

type: Boolean

When a rule's consequence modifies a fact it may cause the rule to activate again, causing an infinite loop. Setting no-loop to true will skip the creation of another Activation for the rule with the current set of facts.

`ruleflow-group`
default value: N/A

type: String

Ruleflow is a Drools feature that lets you exercise control over the firing of rules. Rules that are assembled by the same ruleflow-group identifier fire only when their group is active.

`lock-on-active`

default value: `false`

type: Boolean

Whenever a ruleflow-group becomes active or an agenda-group receives the focus, any rule within that group that has lock-on-active set to true will not be activated any more; irrespective of the origin of the update, the activation of a matching rule is discarded. This is a stronger version of no-loop, because the change could now be caused not only by the rule itself. It's ideal for calculation rules where you have a number of rules that modify a fact and you don't want any rule re-matching and firing again. Only when the ruleflow-group is no longer active or the agenda-group loses the focus those rules with lock-on-active set to true become eligible again for their activations to be placed onto the agenda.

`salience`

default value: `0`

type: integer

Each rule has an integer salience attribute which defaults to zero and can be negative or positive. Salience is a form of priority where rules with higher salience values are given higher priority when ordered in the Activation queue.

Drools also supports dynamic salience where you can use an expression involving bound variables.

Example 8.48. Dynamic Salience

```
rule "Fire in rank order 1,2,.."        salience( -$rank )    when        Element( $rank :
 rank,... )    then        ...end
1,2,.."        salience( -$rank
)
when        Element( $rank : rank,...
)
then
...
```

`agenda-group`

default value: MAIN

type: String

Agenda groups allow the user to partition the Agenda providing more execution control. Only rules in the agenda group that has acquired the focus are allowed to fire.

`auto-focus`

default value: `false`

type: Boolean

When a rule is activated where the `auto-focus` value is true and the rule's agenda group does not have focus yet, then it is given focus, allowing the rule to potentially fire.

`activation-group`
default value: N/A

type: String

Rules that belong to the same activation-group, identified by this attribute's string value, will only fire exclusively. More precisely, the first rule in an activation-group to fire will cancel all pending activations of all rules in the group, i.e., stop them from firing.

Note: This used to be called Xor group, but technically it's not quite an Xor. You may still hear people mention Xor group; just swap that term in your mind with activation-group.

`dialect`
default value: as specified by the package

type: String

possible values: "java" or "mvel"

The dialect species the language to be used for any code expressions in the LHS or the RHS code block. Currently two dialects are available, Java and MVEL. While the dialect can be specified at the package level, this attribute allows the package definition to be overridden for a rule.

`date-effective`
default value: N/A

type: String, containing a date and time definition

A rule can only activate if the current date and time is after date-effective attribute.

`date-expires`
default value: N/A

type: String, containing a date and time definition

A rule cannot activate if the current date and time is after the date-expires attribute.

`duration`
default value: no default value

type: long

The duration dictates that the rule will fire after a specified duration, if it is still true.

Example 8.49. Some attribute examples

```
rule "my rule"
```

```
  salience 42
  agenda-group "number 1"
    when ...
```

8.8.2. Timers and Calendars

Rules now support both interval and cron based timers, which replace the now deprecated duration attribute.

Example 8.50. Sample timer attribute uses

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron:* 0/15 * * * ? )
```

Interval (indicated by "int:") timers follow the semantics of java.util.Timer objects, with an initial delay and an optional repeat interval. Cron (indicated by "cron:") timers follow standard Unix cron expressions:

Example 8.51. A Cron Example

```
rule "Send SMS every 15 minutes"
    timer (cron:* 0/15 * * * ?)
when
    $a : Alarm( on == true )
then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on" );
end
```

A rule controlled by a timer becomes active when it matches, and once for each individual match. Its consequence is executed repeatedly, according to the timer's settings. This stops as soon as the condition doesn't match any more.

Consequences are executed even after control returns from a call to fireUntilHalt. Moreover, the Engine remains reactive to any changes made to the Working Memory. For instance, removing a fact that was involved in triggering the timer rule's execution causes the repeated execution to terminate, or inserting a fact so that some rule matches will cause that rule to fire. But the Engine is not continually active, only after a rule fires, for whatever reason. Thus, reactions to an insertion done asynchronously will not happen until the next execution of a timer-controlled rule. Disposing a session puts an end to all timer activity.

Conversely when the rule engine runs in passive mode (i.e.: using fireAllRules instead of fireUntil-Halt) by default it doesn't fire consequences of timed rules unless fireAllRules isn't invoked again.

However it is possible to change this default behavior by configuring the KieSession with a `TimedRuleExectionOption` as shown in the following example.

Example 8.52. Configuring a KieSession to automatically execute timed rules

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption( TimedRuleExectionOption.YES );
KSession ksession = kbase.newKieSession(ksconf, null);
```

It is also possible to have a finer grained control on the timed rules that have to be automatically executed. To do this it is necessary to set a `FILTERED TimedRuleExectionOption` that allows to define a callback to filter those rules, as done in the next example.

Example 8.53. Configuring a filter to choose which timed rules should be automatically executed

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
conf.setOption( new TimedRuleExectionOption.FILTERED(new TimedRuleExecutionFilter() {
    public boolean accept(Rule[] rules) {
        return rules[0].getName().equals("MyRule");
    }
}) );
```

For what regards interval timers it is also possible to define both the delay and interval as an expression instead of a fixed value. To do that it is necessary to use an expression timer (indicated by "expr:") as in the following example:

Example 8.54. An Expression Timer Example

```
declare Bean
    delay   : String = "30s"
    period  : long = 60000
end

rule "Expression timer"
    timer( expr: $d, $p )
when
    Bean( $d : delay, $p : period )
then
end
```

The expressions, $d and $p in this case, can use any variable defined in the pattern matching part of the rule and can be any String that can be parsed in a time duration or any numeric value that will be internally converted in a long representing a duration expressed in milliseconds.

Both interval and expression timers can have 3 optional parameters named "start", "end" and "repeat-limit". When one or more of these parameters are used the first part of the timer definition must be followed by a semicolon ';' and the parameters have to be separated by a comma ',' as in the following example:

Example 8.55. An Interval Timer with a start and an end

```
timer (int: 30s 10s; start=3-JAN-2010, end=5-JAN-2010)
```

The value for start and end parameters can be a Date, a String representing a Date or a long, or more in general any Number, that will be transformed in a Java Date applying the following conversion:

```
new Date( ((Number) n).longValue() )
```

Conversely the repeat-limit can be only an integer and it defines the maximum number of repetitions allowed by the timer. If both the end and the repeat-limit parameters are set the timer will stop when the first of the two will be matched.

The using of the start parameter implies the definition of a phase for the timer, where the beginning of the phase is given by the start itself plus the eventual delay. In other words in this case the timed rule will then be scheduled at times:

```
start + delay + n*period
```

for up to repeat-limit times and no later than the end timestamp (whichever first). For instance the rule having the following interval timer

```
timer ( int: 30s 1m; start="3-JAN-2010" )
```

will be scheduled at the 30th second of every minute after the midnight of the 3-JAN-2010. This also means that if for example you turn the system on at midnight of the 3-FEB-2010 it won't be scheduled immediately but will preserve the phase defined by the timer and so it will be scheduled for the first time 30 seconds after the midnight. If for some reason the system is paused (e.g. the session is serialized and then deserialized after a while) the rule will be scheduled only once to recover from missing activations (regardless of how many activations we missed) and subsequently it will be scheduled again in phase with the timer.

Calendars are used to control when rules can fire. The Calendar API is modelled on Quartz [http://www.quartz-scheduler.org/]:

Example 8.56. Adapting a Quartz Calendar

```
Calendar weekDayCal = QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)
```

Calendars are registered with the KieSession:

Example 8.57. Registering a Calendar

```
ksession.getCalendars().set( "weekday", weekDayCal );
```

They can be used in conjunction with normal rules and rules including timers. The rule attribute "calendars" may contain one or more comma-separated calendar names written as string literals.

Example 8.58. Using Calendars and Timers together

```
rule "weekdays are high priority"
   calendars "weekday"
   timer (int:0 1h)
when
    Alarm()
then
    send( "priority high - we have an alarm" );
end

rule "weekend are low priority"
   calendars "weekend"
   timer (int:0 4h)
when
    Alarm()
then
    send( "priority low - we have an alarm" );
end
```

8.8.3. Left Hand Side (when) syntax

8.8.3.1. What is the Left Hand Side?

The Left Hand Side (LHS) is a common name for the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is empty, it will be considered as a condition element that is always true and it will be activated once, when a new WorkingMemory session is created.

Figure 8.11. Left Hand Side

Example 8.59. Rule without a Conditional Element

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end

// The above rule is internally rewritten as:

rule "eval(true)"
when
    eval( true )
then
    ... // actions (executed once)
end
```

Conditional elements work on one or more *patterns* (which are described below). The most common conditional element is "`and`". Therefore it is implicit when you have multiple patterns in the LHS of a rule that are not connected in any way:

Example 8.60. Implicit and

```
rule "2 unconnected patterns"when    Pattern1()    Pattern2()then    ... // actionsend// The
 above rule is internally rewritten as:rule "2 and connected patterns"when    Pattern1()
 and Pattern2()then    ... // actionsend
ed
 patterns"when
   Pattern1()

  Pattern2()then    ...


// actionsend// The above rule is internally

 rewritten as:rule "2 and

connected patterns"when
   Pattern1()

and Pattern2()then    ...
```

Note

An "`and`" cannot have a leading declaration binding (unlike for example `or`). This is obvious, since a declaration can only reference a single fact at a time, and when

the "and" is satisfied it matches both facts - so which fact would the declaration bind to?

```
// Compile error$person : (Person( name == "Romeo" ) and Person( name == "Juliet"))
error$person : (Person( name == "Romeo" ) and Person( name ==
```

8.8.3.2. Pattern (conditional element)

8.8.3.2.1. What is a pattern?

A pattern element is the most important Conditional Element. It can potentially match on each fact that is inserted in the working memory.

A pattern contains of zero or more constraints and has an optional pattern binding. The railroad diagram below shows the syntax for this.

Figure 8.12. Pattern

In its simplest form, with no constraints, a pattern matches against a fact of the given type. In the following case the type is `Cheese`, which means that the pattern will match against all `Person` objects in the Working Memory:

```
Person()
```

The type need not be the actual class of some fact object. Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes.

```
Object() // matches all objects in the working memory
```

Inside of the pattern parenthesis is where all the action happens: it defines the constraints for that pattern. For example, with a age related constraint:

```
Person( age == 100 )
```

Note

For backwards compatibility reasons it's allowed to suffix patterns with the ; character. But it is not recommended to do that.

8.8.3.2.2. Pattern binding

For referring to the matched object, use a pattern binding variable such as `$p`.

Example 8.61. Pattern with a binding variable

```
rule ...
when
    $p : Person()
then
    System.out.println( "Person " + $p );
end
```

The prefixed dollar symbol (`$`) is just a convention; it can be useful in complex rules where it helps to easily differentiate between variables and fields, but it is not mandatory.

8.8.3.3. Constraint (part of a pattern)

8.8.3.3.1. What is a constraint?

A constraint is an expression that returns `true` or `false`. This example has a constraint that states *5 is smaller than 6*:

```
Person( 5 < 6 )  // just an example, as constraints like this would be useless in a real pattern
```

In essence, it's a Java expression with some enhancements (such as property access) and a few differences (such as `equals()` semantics for ==). Let's take a deeper look.

8.8.3.3.2. Property access on Java Beans (POJO's)

Any bean property can be used directly. A bean property is exposed using a standard Java bean getter: a method `getMyProperty()` (or `isMyProperty()` for a primitive boolean) which takes no arguments and return something. For example: the age property is written as `age` in DRL instead of the getter `getAge()`:

```
Person( age == 50 )// this is the same as:Person( getAge() == 50 )

)// this is the same
as:Person( getAge() == 50
```

Drools uses the standard JDK `Introspector` class to do this mapping, so it follows the standard Java bean specification.

Note

We recommend using property access (`age`) over using getters explicitly (`getAge()`) because of performance enhancements through field indexing.

Warning

Property accessors must not change the state of the object in a way that may effect the rules. Remember that the rule engine effectively caches the results of its matching in between invocations to make it faster.

```
public int getAge() {    age++; // Do NOT do this    return age;}
{    age++; // Do NOT do
this    return
age;
```

```
public int getAge() {    Date now = DateUtil.now(); // Do NOT do this    return
 DateUtil.differenceInYears(now, birthday);}
{    Date now = DateUtil.now(); // Do NOT do
this    return DateUtil.differenceInYears(now,
birthday);
```

To solve this latter case, insert a fact that wraps the current date into working memory and update that fact between `fireAllRules` as needed.

Note

The following fallback applies: if the getter of a property cannot be found, the compiler will resort to using the property name as a method name and without arguments:

```
Person( age == 50 )// If Person.getAge() does not exists, this falls back
 to:Person( age() == 50 )

)// If Person.getAge() does not exists, this falls back
to:Person( age() == 50
```

Nested property access is also supported:

```
Person( address.houseNumber == 50 )// this is the same as:Person( getAddress().getHouseNumber()
 == 50 )

)// this is the same
as:Person( getAddress().getHouseNumber() == 50
```

Nested properties are also indexed.

Warning

In a stateful session, care should be taken when using nested accessors as the Working Memory is not aware of any of the nested values, and does not know when they change. Either consider them immutable while any of their parent references are inserted into the Working Memory. Or, instead, if you wish to modify a nested value you should mark all of the outer facts as updated. In the above example, when the `houseNumber` changes, any `Person` with that `Address` must be marked as updated.

8.8.3.3.3. Java expression

You can use any Java expression that returns a `boolean` as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access:

```
Person( age == 50 )
```

It is possible to change the evaluation priority by using parentheses, as in any logic or mathematical expression:

```
Person( age > 100 && ( age % 10 == 0 ) )
```

It is possible to reuse Java methods:

```
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```

Warning

As for property accessors, methods must not change the state of the object in a way that may affect the rules. Any method executed on a fact in the LHS should be a *read only* method.

```
Person( incrementAndGetAge() == 10 ) // Do NOT do this
```

Warning

The state of a fact should not change between rule invocations (unless those facts are marked as updated to the working memory on every change):

```
Person( System.currentTimeMillis() % 1000 == 0 ) // Do NOT do this
```

Normal Java operator precedence applies, see the operator precedence list below.

Important

All operators have normal Java semantics except for == and `!=`.

The == operator has null-safe `equals()` semantics:

```
// Similar to: java.util.Objects.equals(person.getFirstName(), "John")//
   so (because "John" is not null) similar to:
"John".equals(person.getFirstName())Person( firstName == "John" )
"John")// so (because "John" is not null) similar
to://
"John".equals(person.getFirstName())Person( firstName == "John"
```

The `!=` operator has null-safe `!equals()` semantics:

```
//      Similar      to:      !java.util.Objects.equals(person.getFirstName(),
"John")Person( firstName != "John" )
"John")Person( firstName != "John"
```

Type coercion is always attempted if the field and the value are of different types; exceptions will be thrown if a bad coercion is attempted. For instance, if "ten" is provided as a string in a numeric evaluator, an exception is thrown, whereas "10" would coerce to a numeric 10. Coercion is always in favor of the field type and not the value type:

```
Person( age == "10" ) // "10" is coerced to 10
```

8.8.3.3.4. Comma separated AND

The comma character (',') is used to separate constraint groups. It has implicit *AND* connective semantics.

```
// Person is at least 50 and weighs at least 80 kgPerson( age > 50, weight > 80 )
kgPerson( age > 50, weight > 80
```

```
// Person is at least 50, weighs at least 80 kg and is taller than 2 meter.Person( age > 50,
 weight > 80, height > 2 )
meter.Person( age > 50, weight > 80, height > 2
```

Note

Although the `&&` and `,` operators have the same semantics, they are resolved with different priorities: The `&&` operator precedes the `||` operator. Both the `&&` and `||` operator precede the `,` operator. See the operator precedence list below.

The comma operator should be preferred at the top level constraint, as it makes constraints easier to read and the engine will often be able to optimize them better.

The comma (`,`) operator cannot be embedded in a composite constraint expression, such as parentheses:

```
Person( ( age > 50, weight > 80 ) || height > 2 ) // Do NOT do this: compile error// Use this
 insteadPerson( ( age > 50 && weight > 80 ) || height > 2 )

 error// Use
this insteadPerson( ( age > 50 && weight > 80 ) || height >
```

8.8.3.3.5. Binding variables

A property can be bound to a variable:

```
// 2 persons of the same agePerson( $firstAge : age ) // bindingPerson( age == $firstAge ) //
 constraint expression
agePerson( $firstAge : age ) //
bindingPerson( age == $firstAge ) // constraint
```

The prefixed dollar symbol (`$`) is just a convention; it can be useful in complex rules where it helps to easily differentiate between variables and fields.

Note

For backwards compatibility reasons, It's allowed (but not recommended) to mix a constraint binding and constraint expressions as such:

```
// Not recommendedPerson( $age : age * 2 < 100 )
ommendedPerson( $age : age * 2 <
```

```
// Recommended (separates bindings and constraint expressions)Person( age * 2 <
 100, $age : age )
 and constraint expressions)Person( age * 2 <
```

Bound variable restrictions using the operator == provide for very fast execution as it use hash indexing to improve performance.

8.8.3.3.6. Unification

Drools does not allow bindings to the same declaration. However this is an important aspect to derivation query unification. While positional arguments are always processed with unification a special unification symbol, ':=', was introduced for named arguments named arguments. The following "unifies" the age argument across two people.

```
Person( $age := age ) Person( $age := age)
 Person( $age := age)
```

In essence unification will declare a binding for the first occurrence and constrain to the same value of the bound field for sequence occurrences.

8.8.3.3.7. Grouped accessors for nested objects

Often it happens that it is necessary to access multiple properties of a nested object as in the following example

```
Person( name == "mark", address.city == "london", address.country == "uk" )
```

These accessors to nested objects can be grouped with a '.(...)' syntax providing more readable rules as in

```
Person( name == "mark", address.( city == "london", country == "uk") )
```

Note the '.' prefix, this is necessary to differentiate the nested object constraints from a method call.

8.8.3.3.8. Inline casts and coercion

When dealing with nested objects, it also quite common the need to cast to a subtype. It is possible to do that via the # symbol as in:

```
Person( name == "mark", address#LongAddress.country == "uk" )
```

This example casts Address to LongAddress, making its getters available. If the cast is not possible (instanceof returns false), the evaluation will be considered false. Also fully qualified names are supported:

```
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )
```

It is possible to use multiple inline casts in the same expression:

```
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000 )
```

moreover, since we also support the instanceof operator, if that is used we will infer its results for further uses of that field, within that pattern:

```
Person( name == "mark", address instanceof LongAddress, address.country == "uk" )
```

8.8.3.3.9. Special literal support

Besides normal Java literals (including Java 5 enums), this literal is also supported:

8.8.3.3.9.1. Date literal

The date format `dd-mmm-yyyy` is supported by default. You can customize this by providing an alternative date format mask as the System property named `drools.dateformat`. If more control is required, use a restriction.

Example 8.62. Date Literal Restriction

```
Cheese( bestBefore < "27-Oct-2009" )
```

8.8.3.3.10. List and Map access

It's possible to directly access a `List` value by index:

```
// Same as childList(0).getAge() == 18
Person( childList[0].age == 18 )
```

It's also possible to directly access a `Map` value by key:

```
// Same as credentialMap.get("jsmith").isValid()
Person( credentialMap["jsmith"].valid )
```

8.8.3.3.11. Abbreviated combined relation condition

This allows you to place more than one restriction on a field using the restriction connectives `&&` or `||`. Grouping via parentheses is permitted, resulting in a recursive syntax pattern.

Figure 8.13. Abbreviated combined relation condition

Figure 8.14. Abbreviated combined relation condition with parentheses

```
// Simple abbreviated combined relation condition using a single &&Person( age > 30 && < 40 )
dition using a single &&Person( age
```

```
// Complex abbreviated combined relation using groupingsPerson( age ( (> 30 && < 40) ||
          (> 20 && < 25) ) )
 relation using groupingsPerson( age ( (>
 30 && < 40) ||              (> 20
```

```
// Mixing abbreviated combined relation with constraint connectivesPerson( age > 30 && < 40
 || location == "london" )
lation with constraint connectivesPerson( age > 30 && <
```

8.8.3.3.12. Special DRL operators

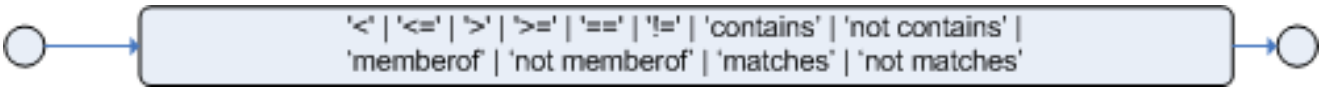


Figure 8.15. Operators

Coercion to the correct value for the evaluator and the field will be attempted.

8.8.3.3.12.1. The operators `<` `<=` `>` `>=`

These operators can be used on properties with natural ordering. For example, for Date fields, < means *before*, for `String` fields, it means alphabetically lower.

```
Person( firstName < $otherFirstName )
```

```
Person( birthDate < $otherBirthDate )
```

Only applies on `Comparable` properties.

8.8.3.3.12.2. Null-safe dereferencing operator

The !. operator allows to derefencing in a null-safe way. More in details the matching algorithm requires the value to the left of the !. operator to be not null in order to give a positive result for pattern matching itself. In other words the pattern:

```
Person( $streetName : address!.street )
```

will be internally translated in:

```
Person( address != null, $streetName : address.street )
```

8.8.3.3.12.3. The operator `matches`

Matches a field against any valid Java Regular Expression. Typically that regexp is a string literal, but variables that resolve to a valid regexp are also allowed.

Example 8.63. Regular Expression Constraint

```
Cheese( type matches "(Buffalo)?\\S*Mozarella" )
```

Note

Like in Java, regular expressions written as string literals *need to escape* '\'.

Only applies on `String` properties. Using `matches` against a `null` value always evaluates to false.

8.8.3.3.12.4. The operator `not matches`

The operator returns true if the String does not match the regular expression. The same rules apply as for the `matches` operator. Example:

Example 8.64. Regular Expression Constraint

```
Cheese( type not matches "(Buffulo)?\\S*Mozarella" )
```

Only applies on `String` properties. Using `not matches` against a `null` value always evaluates to true.

8.8.3.3.12.5. The operator `contains`

The operator `contains` is used to check whether a field that is a Collection or elements contains the specified value.

Example 8.65. Contains with Collections

```
CheeseCounter( cheeses contains "stilton" ) // contains with a String liter
alCheeseCounter( cheeses contains $var ) // contains with a variable
 String literalCheeseCounter( cheeses contains $var ) // contains
```

Only applies on `Collection` properties.

8.8.3.3.12.6. The operator `not contains`

The operator `not contains` is used to check whether a field that is a Collection or elements does *not* contain the specified value.

Example 8.66. Literal Constraint with Collections

```
CheeseCounter( cheeses not contains "cheddar" ) // not contains with a String liter
alCheeseCounter( cheeses not contains $var ) // not contains with a variable
 String literalCheeseCounter( cheeses not contains $var ) // not contains
```

Only applies on `Collection` properties.

Note

For backward compatibility, the `excludes` operator is supported as a synonym for `not contains`.

8.8.3.3.12.7. The operator `memberOf`

The operator `memberOf` is used to check whether a field is a member of a collection or elements; that collection must be a variable.

Example 8.67. Literal Constraint with Collections

```
CheeseCounter( cheese memberOf $matureCheeses )
```

8.8.3.3.12.8. The operator `not memberOf`

The operator `not memberOf` is used to check whether a field is not a member of a collection or elements; that collection must be a variable.

Example 8.68. Literal Constraint with Collections

```
CheeseCounter( cheese not memberOf $matureCheeses )
```

8.8.3.3.12.9. The operator `soundslike`

This operator is similar to `matches`, but it checks whether a word has almost the same sound (using English pronunciation) as the given value. This is based on the Soundex algorithm (see `http://en.wikipedia.org/wiki/Soundex`).

Example 8.69. Test with soundslike

```
// match cheese "fubar" or "foobar"Cheese( name soundslike 'foobar' )
bar"Cheese( name soundslike 'foobar'
```

8.8.3.3.12.10. The operator `str`

This operator `str` is used to check whether a field that is a `String` starts with or ends with a certain value. It can also be used to check the length of the String.

```
Message( routingValue str[startsWith] "R1" )
```

```
Message( routingValue str[endsWith] "R2" )
```

```
Message( routingValue str[length] 17 )
```

8.8.3.3.12.11. The operators `in` and `not in` (compound value restriction)

The compound value restriction is used where there is more than one possible value to match. Currently only the `in` and `not in` evaluators support this. The second operand of this operator must be a comma-separated list of values, enclosed in parentheses. Values may be given as variables, literals, return values or qualified identifiers. Both evaluators are actually *syntactic sugar*, internally rewritten as a list of multiple restrictions using the operators `!=` and ==.

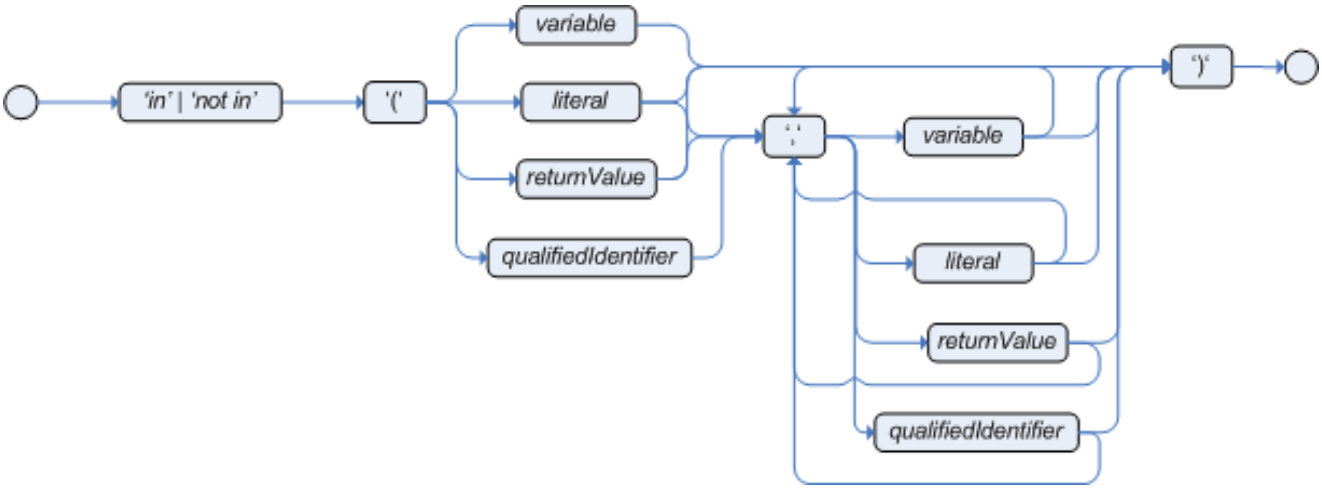


Figure 8.16. compoundValueRestriction

Example 8.70. Compound Restriction using "in"

```
Person( $cheese : favouriteCheese )Cheese( type in ( "stilton", "cheddar", $cheese ) )
)Cheese( type in ( "stilton", "cheddar", $cheese )
```

8.8.3.3.13. Inline eval operator (deprecated)

Figure 8.17. Inline Eval Expression

An inline eval constraint can use any valid dialect expression as long as it results to a primitive boolean. The expression must be constant over time. Any previously bound variable, from the current or previous pattern, can be used; autovivification is also used to auto-create field binding variables. When an identifier is found that is not a current variable, the builder looks to see if the

identifier is a field on the current object type, if it is, the field binding is auto-created as a variable of the same name. This is called autovivification of field variables inside of inline eval's.

This example will find all male-female pairs where the male is 2 years older than the female; the variable `age` is auto-created in the second pattern by the autovivification process.

Example 8.71. Return Value operator

```
Person( girlAge : age, sex = "F" )Person( eval( age == girlAge + 2 ), sex = 'M' ) // eval()
 is actually obsolete in this example
)Person( eval( age == girlAge + 2 ), sex = 'M' ) // eval() is actually obsolete in this
```

Note

Inline eval's are effectively obsolete as their inner syntax is now directly supported. It's recommended not to use them. Simply write the expression without wrapping eval() around it.

8.8.3.3.14. Operator precedence

The operators are evaluated in this precedence:

Table 8.1. Operator precedence

Operator type	Operators	Notes
(nested / null safe) property access	`. !.`	Not normal Java semantics
List/Map access	`[ ]`	Not normal Java semantics
constraint binding	`:`	Not normal Java semantics
multiplicative	`* / %`	
additive	`+ -`	
shift	`<< >> >>>`	
relational	`< > <= >= instanceof`	
equality	`== !=`	Does not use normal Java (*not*) *same* semantics: uses (*not*) *equals* semantics instead.
non-short circuiting AND	`&`	
non-short circuiting exclusive OR	`^`	
non-short circuiting inclusive OR	`|`	

Operator type	Operators	Notes
logical AND	`&&`	
logical OR	`||`	
ternary	`? :`	
Comma separated AND	`,`	Not normal Java semantics

8.8.3.4. Positional Arguments

Patterns now support positional arguments on type declarations.

Positional arguments are ones where you don't need to specify the field name, as the position maps to a known named field. i.e. Person(name == "mark") can be rewritten as Person("mark";). The semicolon ';' is important so that the engine knows that everything before it is a positional argument. Otherwise we might assume it was a boolean expression, which is how it could be interpreted after the semicolon. You can mix positional and named arguments on a pattern by using the semicolon ';' to separate them. Any variables used in a positional that have not yet been bound will be bound to the field that maps to that position.

```
declare Cheese    name : String    shop : String    price : intend
Cheese    name :
String    shop :
String    price :
int
```

Example patterns, with two constraints and a binding. Remember semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in positional arguments, but not variables. Positional arguments are always resolved using unification.

```
Cheese(   "stilton",   "Cheese   Shop",   p;   )Cheese(   "stilton",   "Cheese   Shop";   p   :
 price )Cheese( "stilton"; shop == "Cheese Shop", p : price )Cheese( name == "stilton"; shop
 == "Cheese Shop", p : price )
)Cheese( "stilton", "Cheese Shop"; p : price
)Cheese( "stilton"; shop == "Cheese Shop", p : price
)Cheese( name == "stilton"; shop == "Cheese Shop", p : price
```

Positional arguments that are given a previously declared binding will constrain against that using unification; these are referred to as input arguments. If the binding does not yet exist, it will create the declaration binding it to the field represented by the position argument; these are referred to as output arguments.

8.8.3.5. Fine grained property change listeners

When you call modify() (see the modify statement section) on a given object it will trigger a revaluation of all patterns of the matching object type in the knowledge base. This can can lead to un-

wanted and useless evaluations and in the worst cases to infinite recursions. The only workaround to avoid it was to split up your objects into smaller ones having a 1 to 1 relationship with the original object.

This feature allows the pattern matching to only react to modification of properties actually constrained or bound inside of a given pattern. That will help with performance and recursion and avoid artificial object splitting.

By default this feature is off in order to make the behavior of the rule engine backward compatible with the former releases. When you want to activate it on a specific bean you have to annotate it with @propertyReactive. This annotation works both on DRL type declarations:

```
declare Person@propertyReactive    firstName : String    lastName : Stringend
Per
son@propertyReactive    firstName :
 String    lastName :
```

and on Java classes:

```
@PropertyReactive     public static class Person {     private String firstName;     private
 String lastName;}
ertyReactive    public static class
 Person {    private
 String firstName;    private
```

In this way, for instance, if you have a rule like the following:

```
rule "Every person named Mario is a male" when    $person : Person( firstName == "Mario" )then
    modify ( $person )  { setMale( true ) }end
when    $person : Person( firstName == "Mario"
)
then    modify ( $person )  { setMale( true )
}
```

you won't have to add the no-loop attribute to it in order to avoid an infinite recursion because the engine recognizes that the pattern matching is done on the 'firstName' property while the RHS of the rule modifies the 'male' one. Note that this feature does not work for update(), and this is one of the reasons why we promote modify() since it encapsulates the field changes within the statement. Moreover, on Java classes, you can also annotate any method to say that its invocation actually modifies other properties. For instance in the former Person class you could have a method like:

```
@Modifies( { "firstName", "lastName" } )public void setName(String name) {    String[] names =
 name.split("\\s");    this.firstName = names[0];    this.lastName = names[1];}
 } )public void setName(String
```

```
 name) {    String[] names
= name.split("\\s");    this.firstName
 = names[0];    this.lastName
```

That means that if a rule has a RHS like the following:

```
modify($person) { setName("Mario Fusco") }
```

it will correctly recognize that the values of both properties 'firstName' and 'lastName' could have potentially been modified and act accordingly, not missing of reevaluating the patterns constrained on them. At the moment the usage of @Modifies is not allowed on fields but only on methods. This is coherent with the most common scenario where the @Modifies will be used for methods that are not related with a class field as in the Person.setName() in the former example. Also note that @Modifies is not transitive, meaning that if another method internally invokes the Person.setName() one it won't be enough to annotate it with @Modifies({ "name" }), but it is necessary to use @Modifies({ "firstName", "lastName" }) even on it. Very likely @Modifies transitivity will be implemented in the next release.

For what regards nested accessors, the engine will be notified only for top level fields. In other words a pattern matching like:

```
Person ( address.city.name == "London )
```

will be revaluated only for modification of the 'address' property of a Person object. In the same way the constraints analysis is currently strictly limited to what there is inside a pattern. Another example could help to clarify this. An LHS like the following:

```
$p : Person( )Car( owner = $p.name )
)Car( owner = $p.name
```

will not listen on modifications of the person's name, while this one will do:

```
Person( $name : name )Car( owner = $name )
)Car( owner = $name
```

To overcome this problem it is possible to annotate a pattern with @watch as it follows:

```
$p : Person( ) @watch ( name )Car( owner = $p.name )
)Car( owner = $p.name
```

Indeed, annotating a pattern with @watch allows you to modify the inferred set of properties for which that pattern will react. Note that the properties named in the @watch annotation are actually added to the ones automatically inferred, but it is also possible to explicitly exclude one or more of them prepending their name with a ! and to make the pattern to listen for all or none of the properties of the type used in the pattern respectively with the wildcrds * and !*. So, for example, you can annotate a pattern in the LHS of a rule like:

```
// listens for changes on both firstName (inferred) and lastNamePerson( firstName ==
  $expectedFirstName ) @watch( lastName )// listens for all the properties of the Person
 beanPerson( firstName == $expectedFirstName ) @watch( * )// listens for changes on lastName
 and explicitly exclude firstNamePerson( firstName == $expectedFirstName ) @watch( lastName, !
firstName )// listens for changes on all the properties except the age onePerson( firstName ==
 $expectedFirstName ) @watch( *, !age )
and lastNamePerson( firstName == $expectedFirstName ) @watch(

 lastName )// listens for all the properties of the
 Person beanPerson( firstName == $expectedFirstName ) @watch(

 * )// listens for changes on lastName and explicitly
 exclude firstNamePerson( firstName == $expectedFirstName ) @watch( lastName,

!firstName )// listens for changes on all the properties except the
 age onePerson( firstName == $expectedFirstName ) @watch( *,
```

Since doesn't make sense to use this annotation on a pattern using a type not annotated with @PropertyReactive the rule compiler will raise a compilation error if you try to do so. Also the duplicated usage of the same property in @watch (for example like in: @watch(firstName, ! first-Name)) will end up in a compilation error. In a next release we will make the automatic detection of the properties to be listened smarter by doing analysis even outside of the pattern.

It also possible to enable this feature by default on all the types of your model or to completely disallow it by using on option of the KnowledgeBuilderConfiguration. In particular this new PropertySpecificOption can have one of the following 3 values:

```
- DISABLED => the feature is turned off and all the other related annotations are just ignored-
 ALLOWED => this is the default behavior: types are not property reactive unless they are not
 annotated with @PropertySpecific- ALWAYS => all types are property reactive by default
tions are just ignored- ALLOWED => this is the default behavior: types are not property reactive
 unless they are
 not annotated with @PropertySpecific- ALWAYS => all types are
```

So, for example, to have a KnowledgeBuilder generating property reactive types by default you could do:

```
KnowledgeBuilderConfiguration                              config                              =

 kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
tion
```

```
config = KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();
```

In this last case it will be possible to disable the property reactivity feature on a specific type by annotating it with @ClassReactive.

8.8.3.6. Basic conditional elements

8.8.3.6.1. Conditional Element `and`

The Conditional Element `"and"` is used to group other Conditional Elements into a logical conjunction. Drools supports both prefix `and` and infix `and`.

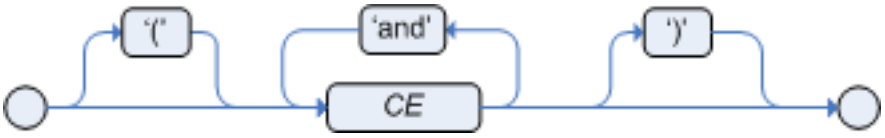


Figure 8.18. infixAnd

Traditional infix `and` is supported:

```
//infixAndCheese( cheeseType : type ) and Person( favouriteCheese == cheeseType )
fixAndCheese( cheeseType : type ) and Person( favouriteCheese == cheeseType
```

Explicit grouping with parentheses is also supported:

```
//infixAnd with grouping( Cheese( cheeseType : type ) and   ( Person( favouriteCheese ==
 cheeseType ) or     Person( favouriteCheese == cheeseType ) )
grouping( Cheese( cheeseType : type )
 and  ( Person( favouriteCheese == cheeseType )
or     Person( favouriteCheese == cheeseType )
```

Note

The symbol `&&` (as an alternative to `and`) is deprecated. But it is still supported in the syntax for backwards compatibility.

Figure 8.19. prefixAnd

Prefix `and` is also supported:

```
(and Cheese( cheeseType : type )     Person( favouriteCheese == cheeseType ) )
)     Person( favouriteCheese == cheeseType )
```

The root element of the LHS is an implicit prefix `and` and doesn't need to be specified:

Example 8.72. implicit root prefixAnd

```
when
    Cheese( cheeseType : type )
    Person( favouriteCheese == cheeseType )
then
    ...
```

8.8.3.6.2. Conditional Element `or`

The Conditional Element `or` is used to group other Conditional Elements into a logical disjunction. Drools supports both prefix `or` and infix `or`.

Figure 8.20. infixOr

Traditional infix `or` is supported:

```
//infixOrCheese( cheeseType : type ) or Person( favouriteCheese == cheeseType )
fixOrCheese( cheeseType : type ) or Person( favouriteCheese == cheeseType
```

Explicit grouping with parentheses is also supported:

```
//infixOr with grouping( Cheese( cheeseType : type ) or  ( Person( favouriteCheese == cheeseType )
 and    Person( favouriteCheese == cheeseType ) )
grouping( Cheese( cheeseType : type )
 or  ( Person( favouriteCheese == cheeseType )
 and    Person( favouriteCheese == cheeseType )
```

Note

The symbol `||` (as an alternative to `or`) is deprecated. But it is still supported in the syntax for backwards compatibility.

Figure 8.21. prefixOr

Prefix `or` is also supported:

```
(or Person( sex == "f", age > 60 )    Person( sex == "m", age > 65 )
)    Person( sex == "m", age > 65
```

Note

The behavior of the Conditional Element `or` is different from the connective `||` for constraints and restrictions in field constraints. The engine actually has no understanding of the Conditional Element `or`. Instead, via a number of different logic transformations, a rule with `or` is rewritten as a number of subrules. This process ultimately results in a rule that has a single `or` as the root node and one subrule for each of its CEs. Each subrule can activate and fire like any normal rule; there is no special behavior or interaction between these subrules. - This can be most confusing to new rule authors.

The Conditional Element `or` also allows for optional pattern binding. This means that each resulting subrule will bind its pattern to the pattern binding. Each pattern must be bound separately, using eponymous variables:

```
pensioner : ( Person( sex == "f", age > 60 ) or Person( sex == "m", age > 65 ) )
```

```
(or pensioner : Person( sex == "f", age > 60 )     pensioner : Person( sex == "m", age > 65 ) )
 )     pensioner : Person( sex == "m", age > 65
```

Since the conditional element `or` results in multiple subrule generation, one for each possible logically outcome, the example above would result in the internal generation of two rules. These two rules work independently within the Working Memory, which means both can match, activate and fire - there is no shortcutting.

The best way to think of the conditional element `or` is as a shortcut for generating two or more similar rules. When you think of it that way, it's clear that for a single rule there could be multiple activations if two or more terms of the disjunction are true.

8.8.3.6.3. Conditional Element `not`

Figure 8.22. not

The CE `not` is first order logic's non-existential quantifier and checks for the non-existence of something in the Working Memory. Think of "not" as meaning "there must be none of...".

The keyword `not` may be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may optionally omit the parentheses.

Example 8.73. No Busses

```
not Bus()
```

Example 8.74. No red Busses

```
// Brackets are optional:not Bus(color == "red")// Brackets are optional:not ( Bus(color ==
 "red", number == 42) )// "not" with nested infix
optional:not Bus(color
== "red")// Brackets
are optional:not ( Bus(color == "red", number ==
42) )// "not" with nestedand - two patterns,
// brackets are requires:
not ( Bus(color == "red") and
      Bus(color == "blue") )
```

8.8.3.6.4. Conditional Element `exists`

Figure 8.23. exists

The CE `exists` is first order logic's existential quantifier and checks for the existence of something in the Working Memory. Think of "exists" as meaning "there is at least one..". It is different from just having the pattern on its own, which is more like saying "for each one of...". If you use `exists` with a pattern, the rule will only activate at most once, regardless of how much data there is in working memory that matches the condition inside of the `exists` pattern. Since only the existence matters, no bindings will be established.

The keyword `exists` must be followed by parentheses around the CEs that it applies to. In the simplest case of a single pattern (like below) you may omit the parentheses.

Example 8.75. At least one Bus

```
exists Bus()
```

Example 8.76. At least one red Bus

```
exists Bus(color == "red")// brackets are optional:exists ( Bus(color == "red", number == 42) )//
 "exists" with nested infix
"red")// brackets are
optional:exists ( Bus(color == "red", number == 42)
)// "exists" with nested infixand,
// brackets are required:
exists ( Bus(color == "red") and
         Bus(color == "blue") )
```

8.8.3.7. Advanced conditional elements

8.8.3.7.1. Conditional Element `forall`

Figure 8.24. forall

The Conditional Element `forall` completes the First Order Logic support in Drools. The Conditional Element `forall` evaluates to true when all facts that match the first pattern match all the remaining patterns. Example:

```
rule "All English buses are red"when    forall( $bus : Bus( type == 'english')
    Bus( this == $bus, color = 'red' ) )then    // all English buses are redend

red"when    forall( $bus : Bus( type ==
'english')                    Bus( this == $bus, color = 'red' )

)then    // all English buses are
```

In the above rule, we "select" all Bus objects whose type is "english". Then, for each fact that matches this pattern we evaluate the following patterns and if they match, the forall CE will evaluate to true.

To state that all facts of a given type in the working memory must match a set of constraints, `forall` can be written with a single pattern for simplicity. Example:

Example 8.77. Single Pattern Forall

```
rule "All Buses are Red"when    forall( Bus( color == 'red' ) )then    // all Bus facts are redend
Red"
when    forall( Bus( color == 'red' )
)
then    // all Bus facts are
red
```

Another example shows multiple patterns inside the `forall`:

Example 8.78. Multi-Pattern Forall

```
rule "all employees have health and dental care programs"when    forall( $emp : Employee()
          HealthCare( employee == $emp )              DentalCare( employee == $emp )
 )then    // all employees have health and dental careend
care
 programs"when    forall( $emp
 : Employee()             HealthCare( employee ==
 $emp )             DentalCare( employee ==
 $emp )

  )then    // all employees have health and
```

Forall can be nested inside other CEs. For instance, `forall` can be used inside a `not` CE. Note that only single patterns have optional parentheses, so that with a nested `forall` parentheses must be used:

Example 8.79. Combining Forall with Not CE

```
rule "not all employees have health and dental care"when     not ( forall( $emp : Employee()
               HealthCare( employee == $emp )                  DentalCare( employee == $emp ) )
        )then    // not all employees have health and dental careend
tal
care"when     not ( forall( $emp
: Employee()                   HealthCare( employee ==
$emp )                  DentalCare( employee == $emp )
 )

 )then    // not all employees have health and
dental
```

As a side note, `forall( p1 p2 p3...)` is equivalent to writing:

```
not(p1 and not(and p2 p3...))
```

Also, it is important to note that `forall` is a *scope delimiter*. Therefore, it can use any previously bound variable, but no variable bound inside it will be available for use outside of it.

8.8.3.7.2. Conditional Element `from`

Figure 8.25. from

The Conditional Element `from` enables users to specify an arbitrary source for data to be matched by LHS patterns. This allows the engine to reason over data not in the Working Memory. The data source could be a sub-field on a bound variable or the results of a method call. It is a powerful construction that allows out of the box integration with other application components and frameworks. One common example is the integration with data retrieved on-demand from databases using hibernate named queries.

The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, it allows you to easily use object property navigation, execute method calls and access maps and collections elements.

Here is a simple example of reasoning and binding on another pattern sub-field:

```
rule "validate zipcode"when     Person( $personAddress : address )      Address( zipcode ==
 "23920W") from $personAddress then    // zip code is okend

zipcode"when    Person( $personAddress : address
 )     Address( zipcode == "23920W") from

$personAddress then    // zip code
is
```

With all the flexibility from the new expressiveness in the Drools engine you can slice and dice this problem many ways. This is the same but shows how you can use a graph notation with the 'from':

```
rule "validate zipcode"when     $p : Person( )      $a : Address( zipcode == "23920W") from
 $p.address then    // zip code is okend

zipcode"when    $p : Person(
 )     $a : Address( zipcode == "23920W") from

$p.address then    // zip code
is
```

Previous examples were evaluations using a single pattern. The CE `from` also support object sources that return a collection of objects. In that case, `from` will iterate over all objects in the collection and try to match each of them individually. For instance, if we want a rule that applies 10% discount to each item in an order, we could do:

```
rule "apply 10% discount to all items over US$ 100,00 in an order"when    $order : Order()
 $item  : OrderItem( value > 100 ) from $order.itemsthen    // apply discount to $itemend

order"when    $order
: Order()    $item  : OrderItem( value > 100 )
from
 $order.itemsthen    // apply discount
to
```

The above example will cause the rule to fire once for each item whose value is greater than 100 for each given order.

You must take caution, however, when using `from`, especially in conjunction with the `lock-on-active` rule attribute as it may produce unexpected results. Consider the example provided earlier, but now slightly modified as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"ruleflow-group "test"lock-on-active
 truewhen     $p : Person( )      $a : Address( state == "NC") from $p.address then     modify
 ($p) {} // Assign person to sales region 1 in a modify blockendrule "Apply a discount to people
 in the city of Raleigh"ruleflow-group "test"lock-on-active truewhen     $p : Person( )
 $a : Address( city == "Raleigh") from $p.address then     modify ($p) {} // Apply discount
 to person in a modify blockend
 region
1"ruleflow-group "test"
lock-
on-active truewhen    $p :
 Person( )     $a : Address( state == "NC")

from $p.address then    modify ($p) {} // Assign person to sales region 1 in
a

 modify blockendrule "Apply a discount to people in the
city of
Raleigh"ruleflow-group "test"
lock-
on-active truewhen    $p :
 Person( )     $a : Address( city == "Raleigh")

from $p.address then    modify ($p) {} // Apply discount to person in
a
```

In the above example, persons in Raleigh, NC should be assigned to sales region 1 and receive a discount; i.e., you would expect both rules to activate and fire. Instead you will find that only the second rule fires.

If you were to turn on the audit log, you would also see that when the second rule fires, it deactivates the first rule. Since the rule attribute `lock-on-active` prevents a rule from creating new activations when a set of facts change, the first rule fails to reactivate. Though the set of facts have not changed, the use of `from` returns a new fact for all intents and purposes each time it is evaluated.

First, it's important to review why you would use the above pattern. You may have many rules across different rule-flow groups. When rules modify working memory and other rules downstream of your RuleFlow (in different rule-flow groups) need to be reevaluated, the use of `modify` is critical. You don't, however, want other rules in the same rule-flow group to place activations on one another recursively. In this case, the `no-loop` attribute is ineffective, as it would only prevent a rule from activating itself recursively. Hence, you resort to `lock-on-active`.

There are several ways to address this issue:

- Avoid the use of `from` when you can assert all facts into working memory or use nested object references in your constraint expressions (shown below).
- Place the variable assigned used in the modify block as the last sentence in your condition (LHS).
- Avoid the use of `lock-on-active` when you can explicitly manage how rules within the same rule-flow group place activations on one another (explained below).

The preferred solution is to minimize use of `from` when you can assert all your facts into working memory directly. In the example above, both the Person and Address instance can be asserted into working memory. In this case, because the graph is fairly simple, an even easier solution is to modify your rules as follows:

```
rule "Assign people in North Carolina (NC) to sales region 1"ruleflow-group "test"lock-on-active
 truewhen    $p : Person(address.state == "NC" )  then    modify ($p) {} // Assign person to sales
 region 1 in a modify blockendrule "Apply a discount to people in the city of Raleigh"ruleflow-
group "test"lock-on-active truewhen    $p : Person(address.city == "Raleigh" )  then    modify
 ($p) {} //Apply discount to person in a modify blockend
 region
1"ruleflow-group "test"
lock-
on-active truewhen    $p : Person(address.state ==
"NC"
 )  then    modify ($p) {} // Assign person to sales region 1 in
a

 modify blockendrule "Apply a discount to people in the
city of
Raleigh"ruleflow-group "test"
lock-
on-active truewhen    $p : Person(address.city ==
"Raleigh"
 )  then    modify ($p) {} //Apply discount to person in
a
```

Now, you will find that both rules fire as expected. However, it is not always possible to access nested facts as above. Consider an example where a Person holds one or more Addresses and you wish to use an existential quantifier to match people with at least one address that meets certain conditions. In this case, you would have to resort to the use of `from` to reason over the collection.

There are several ways to use `from` to achieve this and not all of them exhibit an issue with the use of `lock-on-active`. For example, the following use of `from` causes both rules to fire as expected:

```
rule "Assign people in North Carolina (NC) to sales region 1"ruleflow-group "test"lock-on-active
 truewhen   $p : Person($addresses : addresses)   exists (Address(state == "NC") from $addresses)
  then    modify ($p) {} // Assign person to sales region 1 in a modify blockendrule "Apply a
 discount to people in the city of Raleigh"ruleflow-group "test"lock-on-active truewhen    $p :
 Person($addresses : addresses)    exists (Address(city == "Raleigh") from $addresses)  then
  modify ($p) {} // Apply discount to person in a modify blockend
 region
1"ruleflow-group "test"
lock-
on-active truewhen    $p :
Person($addresses : addresses)    exists (Address(state == "NC")
from
 $addresses)  then    modify ($p) {} // Assign person to sales region 1 in
a

 modify blockendrule "Apply a discount to people in the
city of
Raleigh"ruleflow-group "test"
lock-
on-active truewhen    $p :
Person($addresses : addresses)    exists (Address(city == "Raleigh")
from
 $addresses)  then    modify ($p) {} // Apply discount to person in
a
```

However, the following slightly different approach does exhibit the problem:

```
rule "Assign people in North Carolina (NC) to sales region 1"ruleflow-group "test"lock-on-active
 truewhen    $assessment : Assessment()    $p : Person()    $addresses : List() from $p.addresses
    exists (Address( state == "NC") from $addresses) then    modify ($assessment) {} // Modify
 assessment in a modify blockendrule "Apply a discount to people in the city of Raleigh"ruleflow-
group "test"lock-on-active truewhen    $assessment : Assessment()    $p : Person()    $addresses :
 List() from $p.addresses     exists (Address( city == "Raleigh") from $addresses)then    modify
 ($assessment) {} // Modify assessment in a modify blockend
 region
1"ruleflow-group "test"
lock-
on-active truewhen
$assessment : Assessment()
$p : Person()    $addresses :
List() from $p.addresses    exists (Address( state == "NC")

from $addresses) then    modify ($assessment) {} // Modify assessment in
a
```

```
 modify blockendrule "Apply a discount to people in the
city of
Raleigh"ruleflow-group "test"
lock-
on-active truewhen
$assessment : Assessment()
$p : Person()    $addresses : List()
 from $p.addresses     exists (Address( city ==
"Raleigh")
 from $addresses)then    modify ($assessment) {} // Modify assessment in
a
```

In the above example, the $addresses variable is returned from the use of `from`. The example also introduces a new object, assessment, to highlight one possible solution in this case. If the $assessment variable assigned in the condition (LHS) is moved to the last condition in each rule, both rules fire as expected.

Though the above examples demonstrate how to combine the use of `from` with `lock-on-active` where no loss of rule activations occurs, they carry the drawback of placing a dependency on the order of conditions on the LHS. In addition, the solutions present greater complexity for the rule author in terms of keeping track of which conditions may create issues.

A better alternative is to assert more facts into working memory. In this case, a person's addresses may be asserted into working memory and the use of `from` would not be necessary.

There are cases, however, where asserting all data into working memory is not practical and we need to find other solutions. Another option is to reevaluate the need for `lock-on-active`. An alternative to `lock-on-active` is to directly manage how rules within the same rule-flow group activate one another by including conditions in each rule that prevent rules from activating each other recursively when working memory is modified. For example, in the case above where a discount is applied to citizens of Raleigh, a condition may be added to the rule that checks whether the discount has already been applied. If so, the rule does not activate.

8.8.3.7.3. Conditional Element `collect`

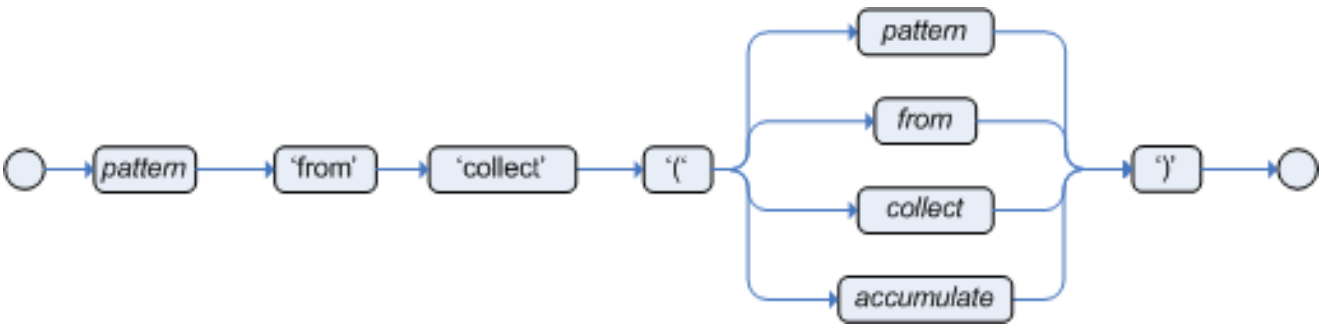


Figure 8.26. collect

The Conditional Element `collect` allows rules to reason over a collection of objects obtained from the given source or from the working memory. In First Oder Logic terms this is the cardinality quantifier. A simple example:

```
import java.util.ArrayListrule "Raise priority if system has more than 3 pending alarms"when
 $system : System()    $alarms : ArrayList( size >= 3 )             from collect( Alarm( system
 == $system, status == 'pending' ) )then    // Raise priority, because system $system has    //
 3 or more alarms pending. The pending alarms    // are $alarms.end

java.util.ArrayListrule "Raise priority if system has more than 3 pending
alarms"
when    $system :
System()    $alarms : ArrayList( size >= 3
)              from collect( Alarm( system == $system, status == 'pending' )
)
then    // Raise priority, because system $system
has    // 3 or more alarms pending. The pending
alarms    // are
$alarms.
```

In the above example, the rule will look for all pending alarms in the working memory for each given system and group them in ArrayLists. If 3 or more alarms are found for a given system, the rule will fire.

The result pattern of `collect` can be any concrete class that implements the `java.util.Collection` interface and provides a default no-arg public constructor. This means that you can use Java collections like ArrayList, LinkedList, HashSet, etc., or your own class, as long as it implements the `java.util.Collection` interface and provide a default no-arg public constructor.

Both source and result patterns can be constrained as any other pattern.

Variables bound before the `collect` CE are in the scope of both source and result patterns and therefore you can use them to constrain both your source and result patterns. But note that `collect` is a scope delimiter for bindings, so that any binding made inside of it is not available for use outside of it.

Collect accepts nested `from` CEs. The following example is a valid use of "collect":

```
import java.util.LinkedList;rule "Send a message to all mothers"when    $town : Town( name ==
 'Paris' )     $mothers : LinkedList()                  from collect( Person( gender == 'F',
 children > 0 )                                  from $town.getPeople()
         )then    // send a message to all mothersend

java.util.LinkedList;rule "Send a message to all
mothers"
when    $town : Town( name == 'Paris'
)    $mothers : LinkedList()
                from collect( Person( gender == 'F', children > 0 )
                              from $town.getPeople()
```

```
)
then    // send a message to all
mothers
```

8.8.3.7.4. Conditional Element `accumulate`

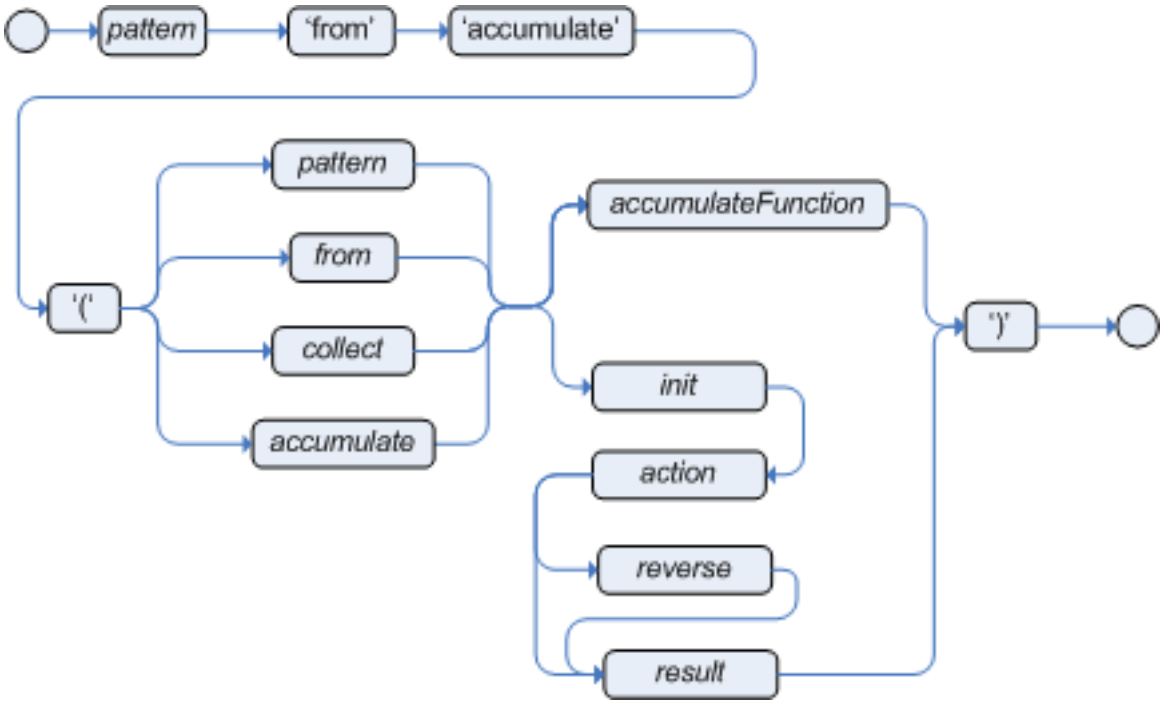


Figure 8.27. accumulate

The Conditional Element `accumulate` is a more flexible and powerful form of `collect`, in the sense that it can be used to do what `collect` does and also achieve results that the CE `collect` is not capable of achieving. Accumulate allows a rule to iterate over a collection of objects, executing custom actions for each of the elements, and at the end, it returns a result object.

Accumulate supports both the use of pre-defined accumulate functions, or the use of inline custom code. Inline custom code should be avoided though, as it is harder for rule authors to maintain, and frequently leads to code duplication. Accumulate functions are easier to test and reuse.

The Accumulate CE also supports multiple different syntaxes. The preferred syntax is the top level accumulate, as noted bellow, but all other syntaxes are supported for backward compatibility.

8.8.3.7.4.1. Accumulate CE (preferred syntax)

The top level accumulate syntax is the most compact and flexible syntax. The simplified syntax is as follows:

```
accumulate( <source pattern>; <functions> [;<constraints>] )
```

For instance, a rule to calculate the minimum, maximum and average temperature reading for a given sensor and that raises an alarm if the minimum temperature is under 20C degrees and the average is over 70C degrees could be written in the following way, using Accumulate:

Note

The DRL language defines "`acc`" as a synonym of "`accumulate`". The author might prefer to use "`acc`" as a less verbose keyword or the full keyword "`accumulate`" for legibility.

```
rule "Raise alarm"
when
    $s : Sensor()
    accumulate( Reading( sensor == $s, $temp : temperature );
                $min : min( $temp ),
                $max : max( $temp ),
                $avg : average( $temp );
                $min < 20, $avg > 70 )
then
    // raise the alarm
end
```

In the above example, min, max and average are Accumulate Functions and will calculate the minimum, maximum and average temperature values over all the readings for each sensor.

Drools ships with several built-in accumulate functions, including:

- average
- min
- max
- count
- sum
- collectList
- collectSet

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
rule "Average profit"when    $order : Order()    accumulate( OrderItem( order == $order, $cost :
 cost, $price : price );                  $avgProfit : average( 1 - $cost / $price ) )then
  // average profit for $order is $avgProfitend

profit"when    $order
: Order()    accumulate( OrderItem( order == $order, $cost : cost, $price :
```

```
price );                $avgProfit : average( 1 - $cost / $price
)
 )then    // average profit for $order
is
```

Accumulate Functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Function all one needs to do is to create a Java class that implements the `org.drools.core.runtime.rule.TypedAccumulateFunction` interface. As an example of an Accumulate Function implementation, the following is the implementation of the `average` function:

```
/**
 * An implementation of an accumulator capable of calculating average values
 */
public class AverageAccumulateFunction implements org.drools.core.runtime.rule.TypedAccumulateFunction {

    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

    public static class AverageData implements Externalizable {
        public int    count = 0;
        public double total = 0;

        public AverageData() {}

        public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
            count   = in.readInt();
            total   = in.readDouble();
        }

        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeInt(count);
            out.writeDouble(total);
        }

    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Serializable createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.drools.core.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
```

```
        data.count = 0;
        data.total = 0;
    }

    /* (non-Javadoc)
      * @see org.drools.core.base.accumulators.AccumulateFunction#accumulate(java.lang.Object,
 java.lang.Object)
     */
    public void accumulate(Serializable context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
         * @see org.drools.core.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
 java.lang.Object)
     */
    public void reverse(Serializable context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.drools.core.base.accumulators.AccumulateFunction#getResult(java.lang.Object)
     */
    public Object getResult(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count );
    }

    /* (non-Javadoc)
     * @see org.drools.core.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

    /**
     * {@inheritDoc}
     */
    public Class< ? > getResultType() {
        return Number.class;
    }

}
```

The code for the function is very simple, as we could expect, as all the "dirty" integration work is done by the engine. Finally, to use the function in the rules, the author can import it using the "import accumulate" statement:

```
import accumulate <class_name> <function_name>
```

For instance, if one implements the class `some.package.VarianceFunction` function that implements the `variance` function and wants to use it in the rules, he would do the following:

Example 8.80. Example of importing and using the custom "`variance`" accumulate function

```
import accumulate some.package.VarianceFunction variance

rule "Calculate Variance"
when
    accumulate( Test( $s : score ), $v : variance( $s ) )
then
    // the variance of the test scores is $v
end
```

Note

The built in functions (sum, average, etc) are imported automatically by the engine. Only user-defined custom accumulate functions need to be explicitly imported.

Note

For backward compatibility, Drools still supports the configuration of accumulate functions through configuration files and system properties, but this is a deprecated method. In order to configure the variance function from the previous example using the configuration file or system property, the user would set a property like this:

```
drools.accumulate.function.variance = some.package.VarianceFunction
```

Please note that "`drools.accumulate.function.`" is a prefix that must always be used, "`variance`" is how the function will be used in the drl files, and "`some.package.VarianceFunction`" is the fully qualified name of the class that implements the function behavior.

8.8.3.7.4.2. Alternate Syntax: single function with return type

The accumulate syntax evolved over time with the goal of becoming more compact and expressive. Nevertheless, Drools still supports previous syntaxes for backward compatibility purposes.

In case the rule is using a single accumulate function on a given accumulate, the author may add a pattern for the result object and use the "from" keyword to link it to the accumulate result. Example: a rule to apply a 10% discount on orders over $100 could be written in the following way:

```
rule "Apply 10% discount to orders over US$ 100,00"when     $order : Order()     $total :
 Number( doubleValue > 100 )             from accumulate( OrderItem( order == $order, $value :
 value ),                               sum( $value ) )then    # apply discount to $orderend
$
 100,00"when    $order
: Order()    $total : Number( doubleValue > 100
 )              from accumulate( OrderItem( order == $order, $value :
value ),                              sum( $value
)
 )then    # apply discount
to
```

In the above example, the accumulate element is using only one function (sum), and so, the rules author opted to explicitly write a pattern for the result type of the accumulate function (Number) and write the constraints inside it. There are no problems in using this syntax over the compact syntax presented before, except that is is a bit more verbose. Also note that it is not allowed to use both the return type and the functions binding in the same accumulate statement.

8.8.3.7.4.3. Accumulate with inline custom code

Warning

The use of accumulate with inline custom code is not a good practice for several reasons, including difficulties on maintaining and testing rules that use them, as well as the inability of reusing that code. Implementing your own accumulate functions is very simple and straightforward, they are easy to unit test and to use. This form of accumulate is supported for backward compatibility only.

Another possible syntax for the accumulate is to define inline custom code, instead of using accumulate functions. As noted on the previous warned, this is discouraged though for the stated reasons.

The general syntax of the `accumulate` CE with inline custom code is:

```
<result pattern> from accumulate( <source pattern>,
                                  init( <init code> ),
                                  action( <action code> ),
                                  reverse( <reverse code> ),
                                  result( <result expression> ) )
```

The meaning of each of the elements is the following:

- *<source pattern>*: the source pattern is a regular pattern that the engine will try to match against each of the source objects.

- <*init code*>: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- <*action code*>: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- <*reverse code*>: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to undo any calculation done in the <*action code*> block, so that the engine can do decremental calculation when a source object is modified or deleted, hugely improving performance of these operations.
- <*result expression*>: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.
- <*result pattern*>: this is a regular pattern that the engine tries to match against the object returned from the <*result expression*>. If it matches, the `accumulate` conditional element evaluates to *true* and the engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the `accumulate` CE evaluates to *false* and the engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

```
rule "Apply 10% discount to orders over US$ 100,00"when     $order : Order()     $total :
 Number( doubleValue > 100 )             from accumulate( OrderItem( order == $order, $value :
 value ),                                init( double total = 0; ),
   action( total += $value; ),                                reverse( total -= $value; ),
                         result( total ) )then    # apply discount to $orderend
$
 100,00"when    $order
: Order()    $total : Number( doubleValue > 100
 )              from accumulate( OrderItem( order == $order, $value :
value ),                              init( double total =
0; ),                                action( total +=
$value; ),                                reverse( total -=
$value; ),                                result( total
)
 )then    # apply discount
to
```

In the above example, for each `Order` in the Working Memory, the engine will execute the *init code* initializing the total variable to zero. Then it will iterate over all `OrderItem` objects for that order, executing the *action* for each one (in the example, it will sum the value of all items into the total variable). After iterating over all `OrderItem` objects, it will return the value corresponding to the *result expression* (in the above example, the value of variable `total`). Finally, the engine will try to match the result with the `Number` pattern, and if the double value is greater than 100, the rule will fire.

The example used Java as the semantic dialect, and as such, note that the usage of the semicolon as statement delimiter is mandatory in the init, action and reverse code blocks. The result is an

expression and, as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect's specific syntax.

As mentioned before, the *reverse code* is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and delete*.

The `accumulate` CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

```
rule "Accumulate using custom objects"when    $person   : Person( $likes : likes )    $cheesery :
 Cheesery( totalAmount > 100 )              from accumulate( $cheese : Cheese( type == $likes ),
                                       init( Cheesery cheesery = new Cheesery(); ),
                             action( cheesery.addCheese( $cheese ); ),
             reverse( cheesery.removeCheese( $cheese ); ),
  result( cheesery ) );then    // do somethingend

custom objects"when    $person   : Person( $likes
 : likes )    $cheesery : Cheesery( totalAmount
 > 100 )                from accumulate( $cheese : Cheese( type
 == $likes ),                                  init( Cheesery cheesery =
 new Cheesery(); ),                                  action( cheesery.addCheese(
 $cheese ); ),                                  reverse( cheesery.removeCheese(
 $cheese ); ),                                  result(

cheesery ) );then
```

8.8.3.8. Conditional Element `eval`

Figure 8.28. eval

The conditional element `eval` is essentially a catch-all which allows any semantic code (that returns a primitive boolean) to be executed. This code can refer to variables that were bound in the LHS of the rule, and functions in the rule package. Overuse of eval reduces the declarativeness of your rules and can result in a poorly performing engine. While `eval` can be used anywhere in the patterns, the best practice is to add it as the last conditional element in the LHS of a rule.

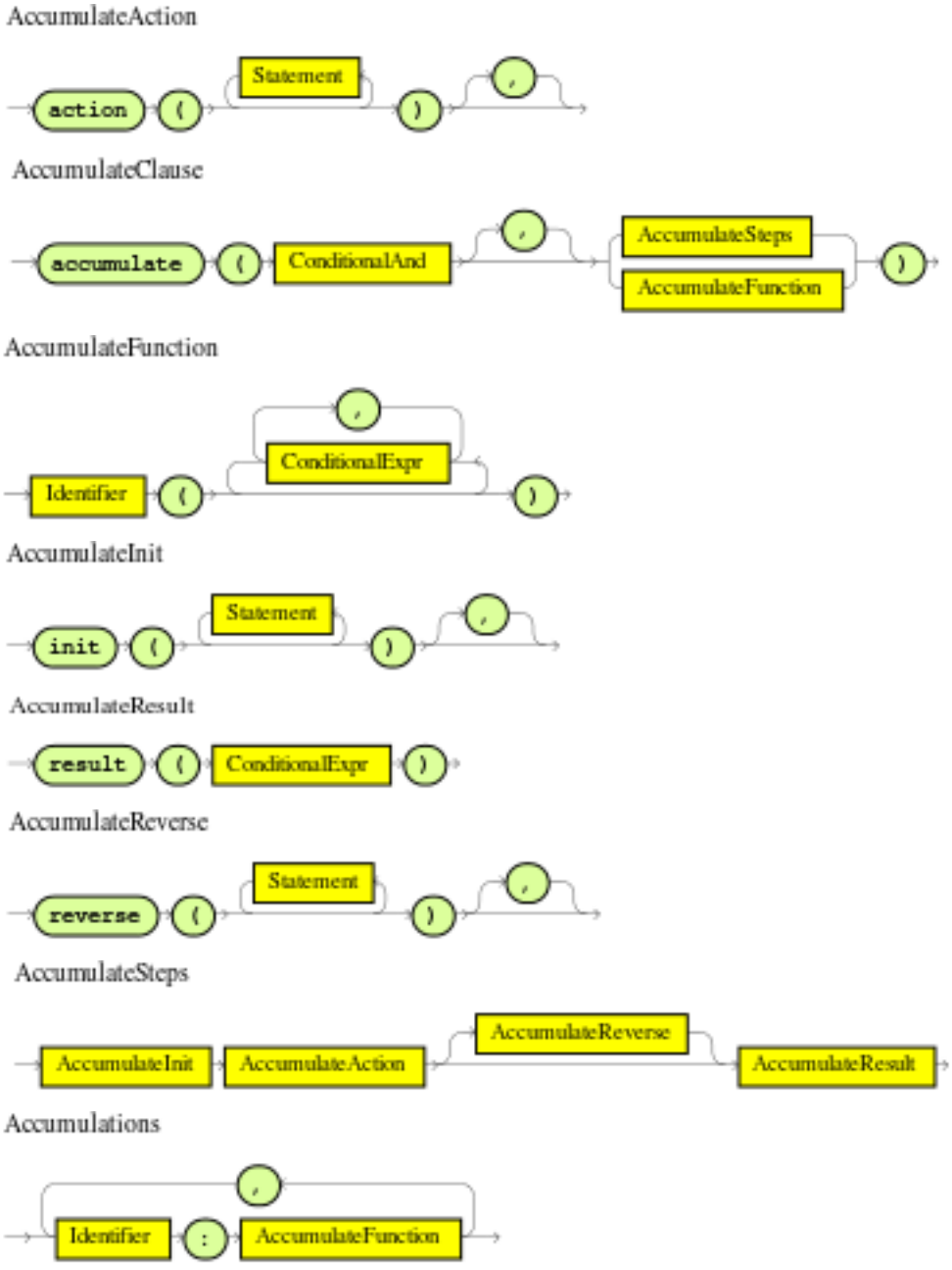
Evals cannot be indexed and thus are not as efficient as Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

For folks who are familiar with Drools 2.x lineage, the old Drools parameter and condition tags are equivalent to binding a variable to an appropriate type, and then using it in an eval node.

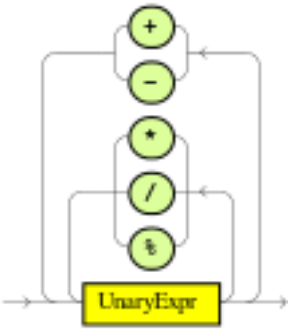
```
p1 : Parameter()p2 : Parameter()eval( p1.getList().containsKey( p2.getItem() ) )
rameter()p2
: Parameter()eval( p1.getList().containsKey( p2.getItem()
```

```
p1 : Parameter()p2 : Parameter()// call function isValid in the LHSeval( isValid( p1, p2 ) )
rameter()p2
: Parameter()// call function isValid in
the LHSeval( isValid( p1, p2
```

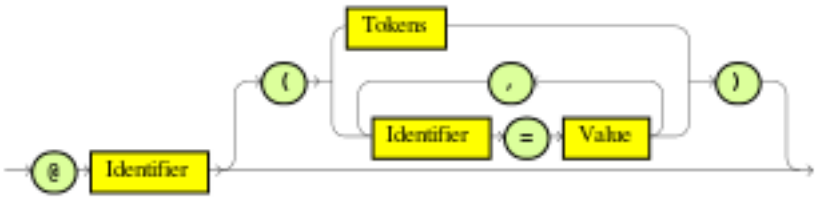
8.8.3.9. Railroad diagrams



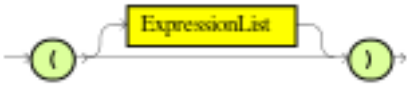
AdditiveExpr



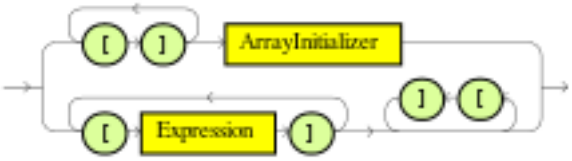
Annotation



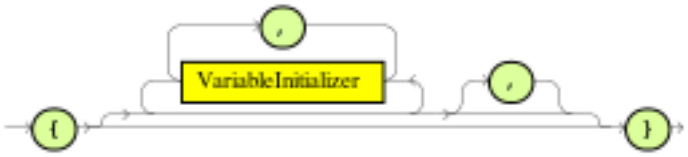
Arguments



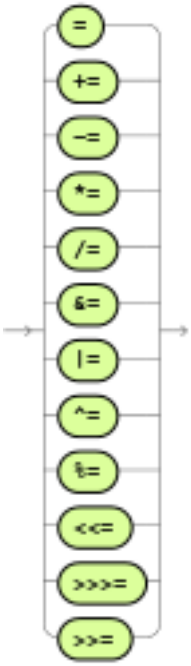
ArrayCreatorRest



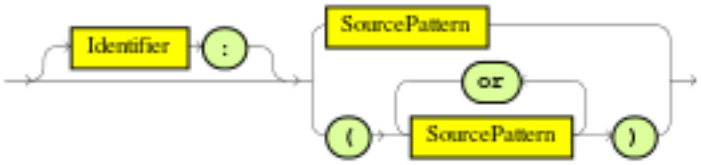
ArrayInitializer



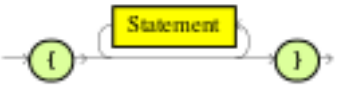
AssignmentOperator



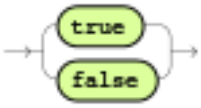
BindingPattern



Block

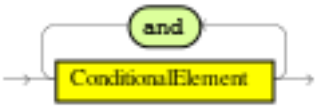


BooleanLiteral



CompilationUnit

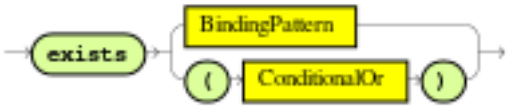
ConditionalAnd



ConditionalElementAccumulate

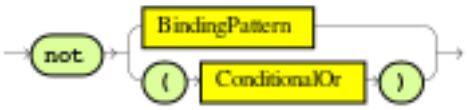
ConditionalElementEval

ConditionalElementExists

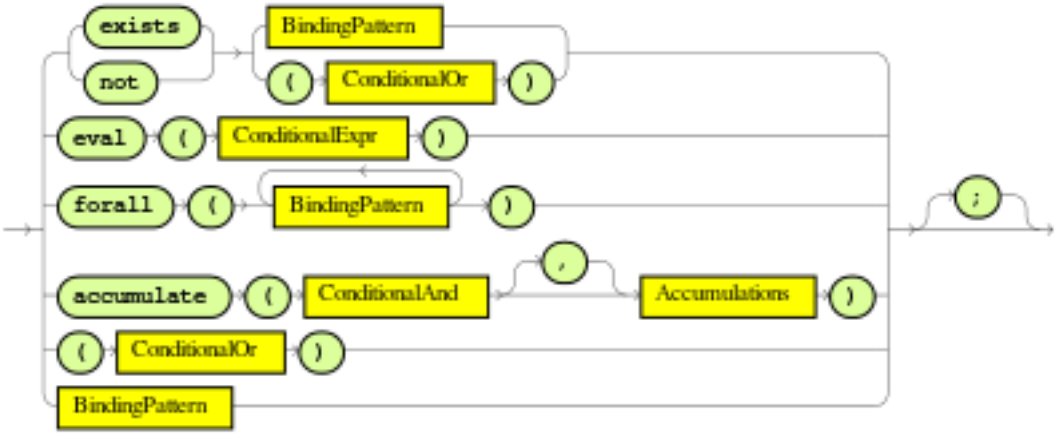


ConditionalElementForall

ConditionalElementNot

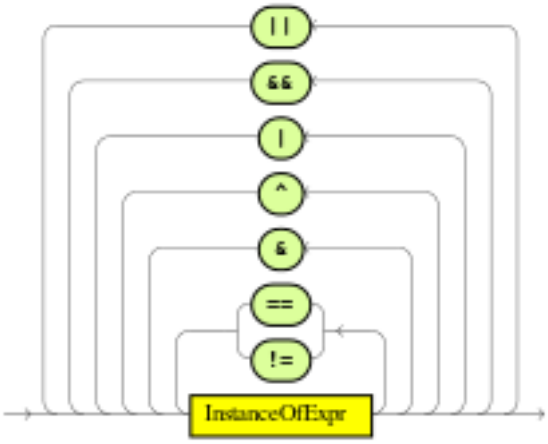


ConditionalElement

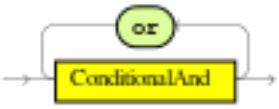


ConditionalExpr

ConditionalOrExpr

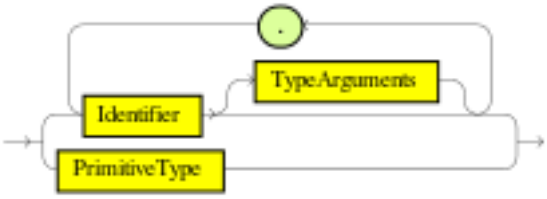


ConditionalOr

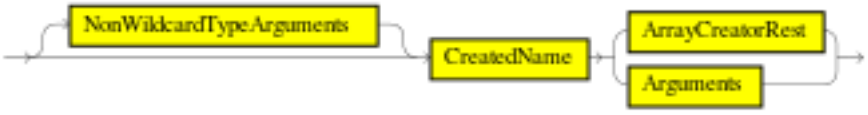


Constraints

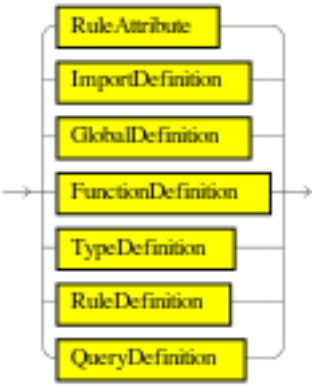
CreatedName



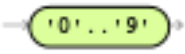
Creator



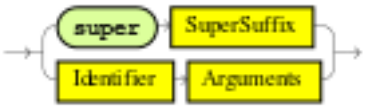
Definition



Digit

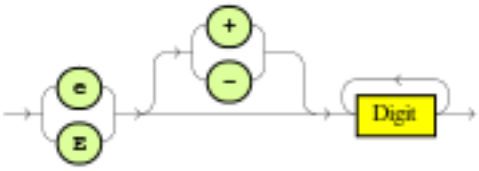


ExplicitGenericInvocationSuffix



ExplicitGenericInvocation

Exponent

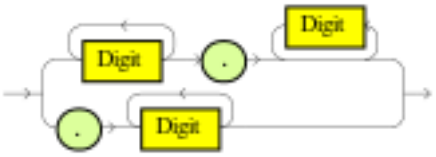


ExpressionList

Expression

Field

Fraction



FromAccumulateClause

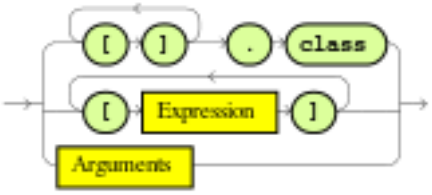
FromClause

FromCollectClause

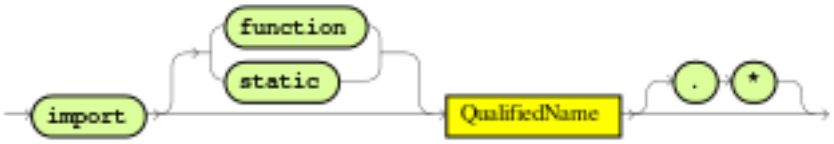
FunctionDefinition

GlobalDefinition

IdentifierSuffix

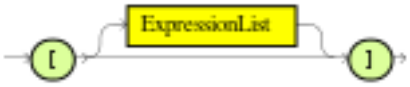


ImportDefinition

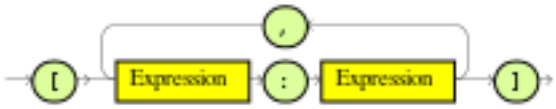


InExpr

InlineListExpr

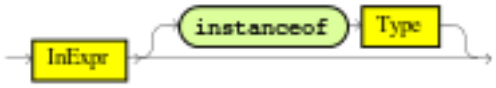


InlineMapExpr

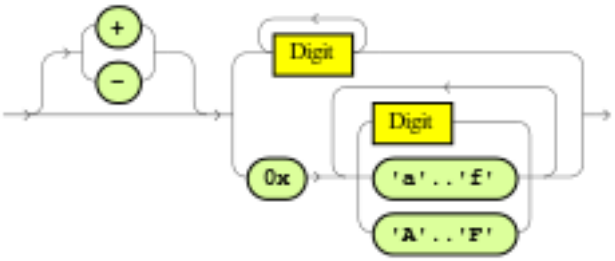


InnerCreator

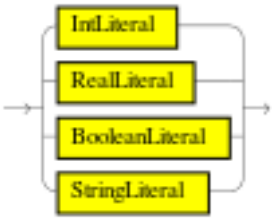
InstanceOfExpr



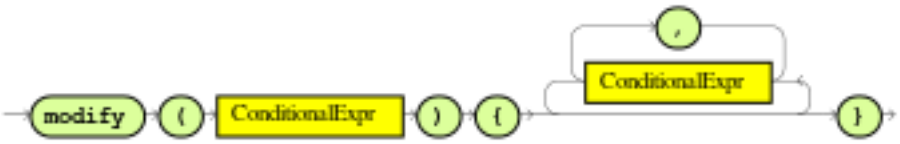
IntLiteral



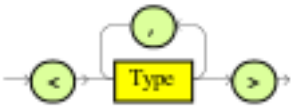
Literal



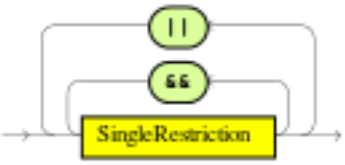
ModifyStatement



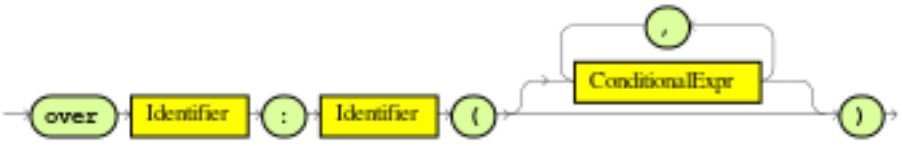
NonWildcardTypeArguments



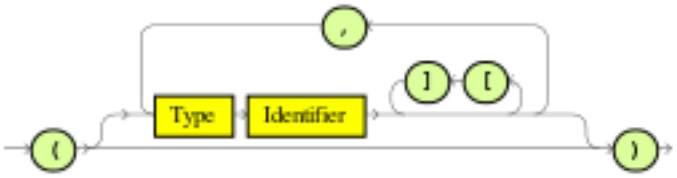
OrRestriction



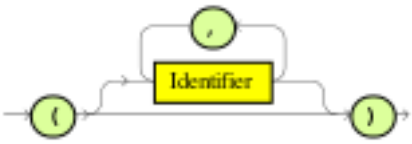
OverClause



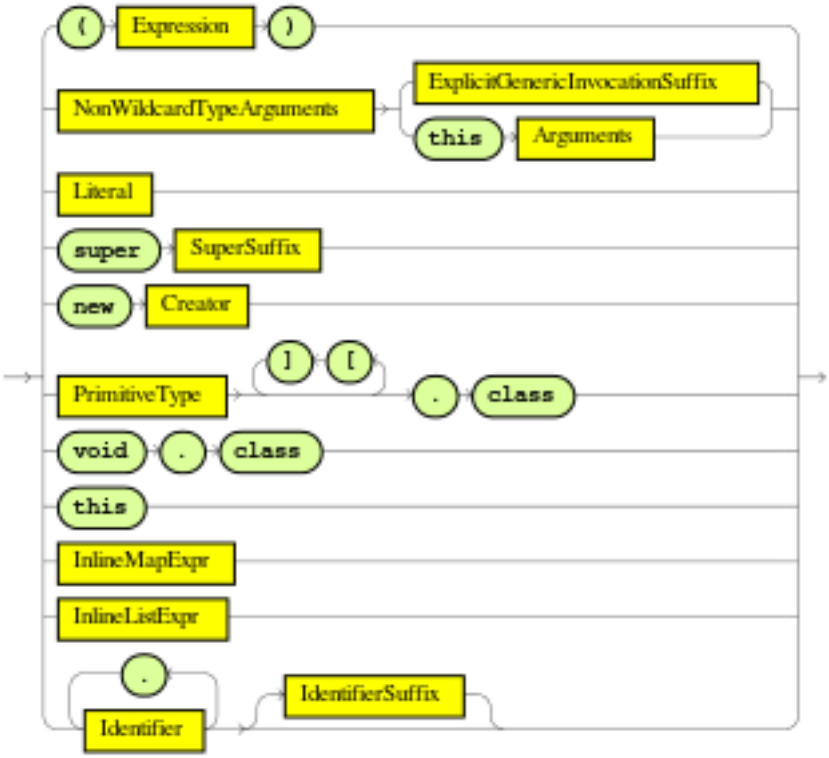
Parameters



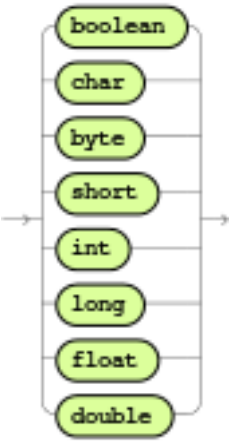
Placeholders



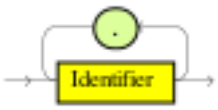
Primary



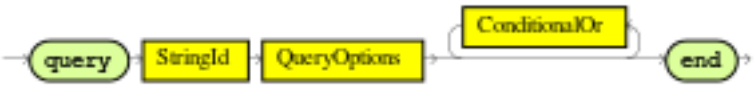
PrimitiveType



QualifiedName

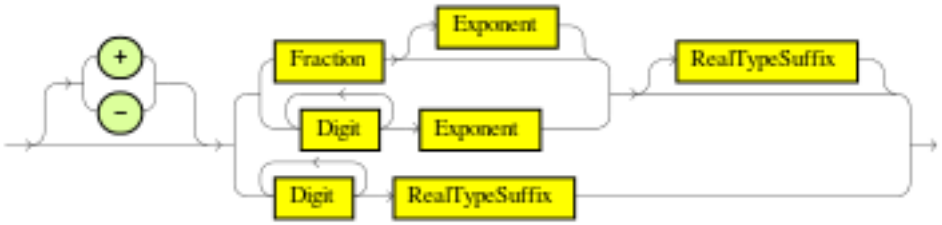


QueryDefinition

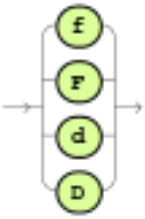


QueryOptions

RealLiteral

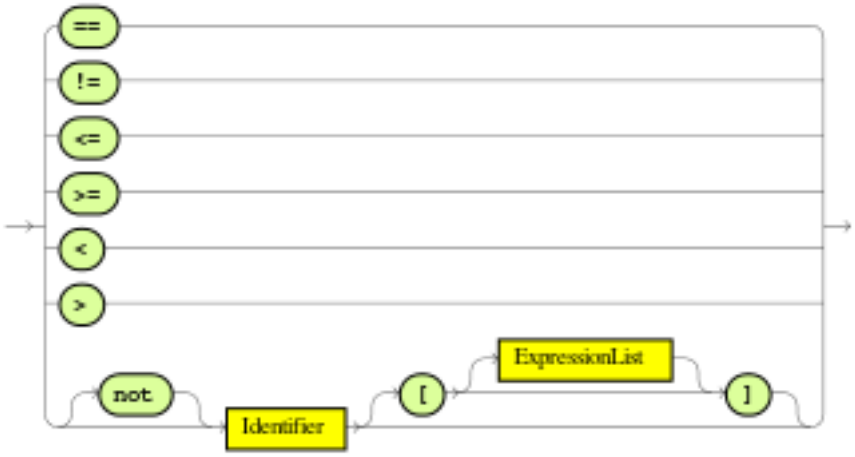


RealTypeSuffix



RelationalExpr

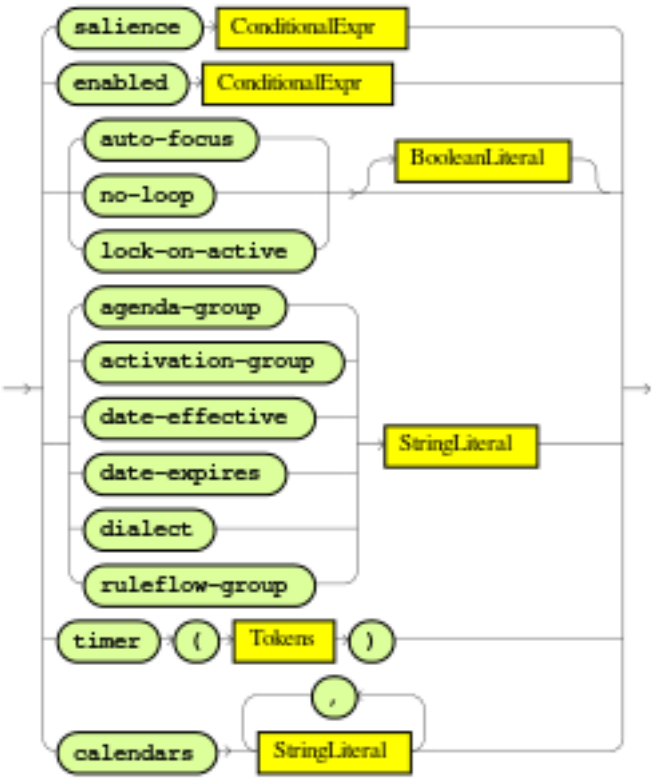
RelationalOperator



RhsStatement

RuleAttributes

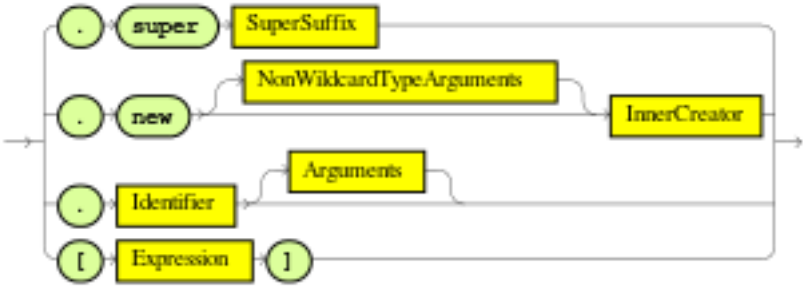
RuleAttribute



RuleDefinition

RuleOptions

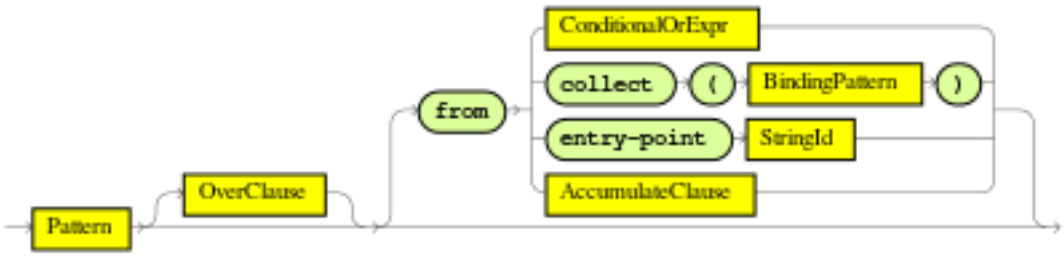
Selector



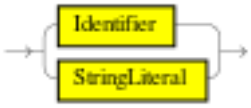
ShiftExpr

SingleRestriction

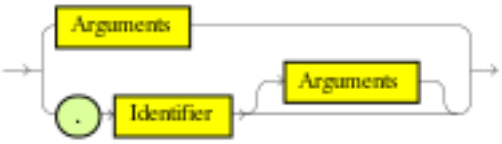
SourcePattern



StringId

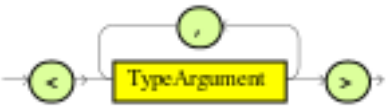


SuperSuffix

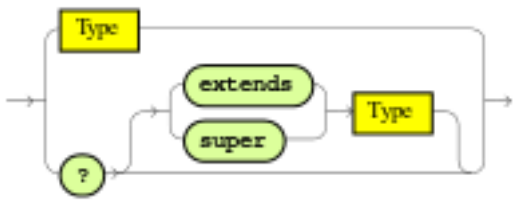


ThenPart

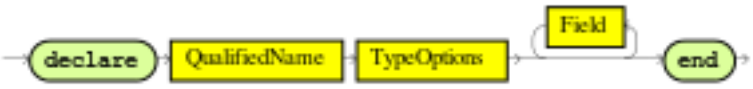
TypeArguments



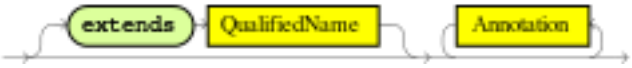
TypeArgument



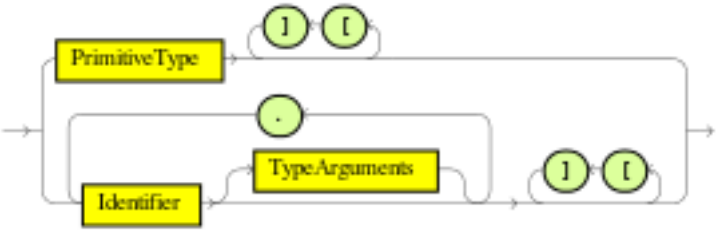
TypeDefinition



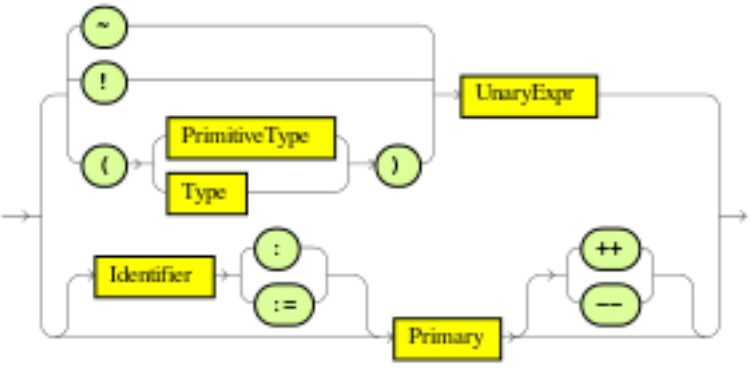
TypeOptions



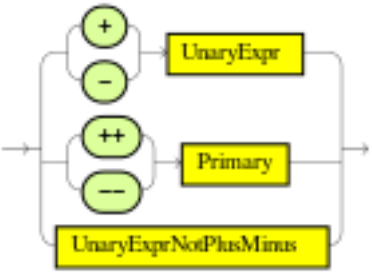
Type



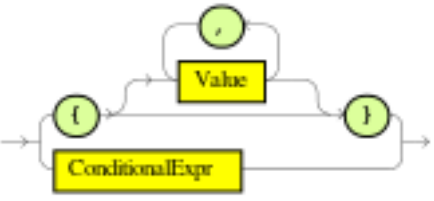
UnaryExprNotPlusMinus

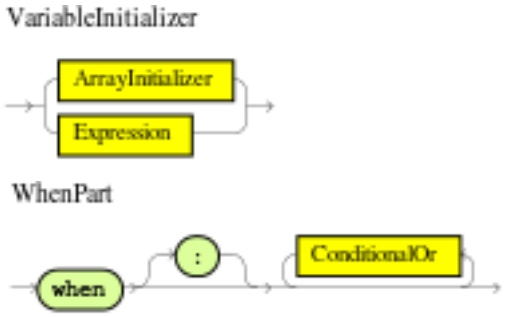


UnaryExpr



Value





8.8.4. The Right Hand Side (then)

8.8.4.1. Usage

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule; this part should contain a list of actions to be executed. It is bad practice to use imperative or conditional code in the RHS of a rule; as a rule should be atomic in nature - "when this, then do this", not "when this, maybe do this". The RHS part of a rule should also be kept small, thus keeping it declarative and readable. If you find you need imperative and/or conditional code in the RHS, then maybe you should be breaking that rule down into multiple rules. The main purpose of the RHS is to insert, delete or modify working memory data. To assist with that there are a few convenience methods you can use to modify working memory; without having to first reference a working memory instance.

`update(`*object, handle*`);` will tell the engine that an object has changed (one that has been bound to something on the LHS) and rules may need to be reconsidered.

`update(`*object*`);` can also be used; here the Knowledge Helper will look up the facthandle for you, via an identity check, for the passed object. (Note that if you provide Property Change Listeners to your Java beans that you are inserting into the engine, you can avoid the need to call `update()` when the object changes.). After a fact's field values have changed you must call update before changing another fact, or you will cause problems with the indexing within the rule engine. The modify keyword avoids this problem.

`insert(new` *Something*`());` will place a new object of your creation into the Working Memory.

`insertLogical(new` *Something*`());` is similar to insert, but the object will be automatically deleted when there are no more facts to support the truth of the currently firing rule.

`delete(`*handle*`);` removes an object from Working Memory.

These convenience methods are basically macros that provide short cuts to the `KnowledgeHelper` instance that lets you access your Working Memory from rules files. The predefined variable `drools` of type `KnowledgeHelper` lets you call several other useful methods. (Refer to the `KnowledgeHelper` interface documentation for more advanced operations).

- The call `drools.halt()` terminates rule execution immediately. This is required for returning control to the point whence the current session was put to work with `fireUntilHalt()`.

- Methods `insert(Object o)`, `update(Object o)` and `delete(Object o)` can be called on `drools` as well, but due to their frequent use they can be called without the object reference.
- `drools.getWorkingMemory()` returns the `WorkingMemory` object.
- `drools.setFocus( String s)` sets the focus to the specified agenda group.
- `drools.getRule().getName()`, called from a rule's RHS, returns the name of the rule.
- `drools.getTuple()` returns the Tuple that matches the currently executing rule, and `drools.getActivation()` delivers the corresponding Activation. (These calls are useful for logging and debugging purposes.)

The full Knowledge Runtime API is exposed through another predefined variable, `kcontext`, of type `KieContext`. Its method `getKieRuntime()` delivers an object of type `KieRuntime`, which, in turn, provides access to a wealth of methods, many of which are quite useful for coding RHS logic.

- The call `kcontext.getKieRuntime().halt()` terminates rule execution immediately.
- The accessor `getAgenda()` returns a reference to this session's `Agenda`, which in turn provides access to the various rule groups: activation groups, agenda groups, and rule flow groups. A fairly common paradigm is the activation of some agenda group, which could be done with the lengthy call:

```
// give focus to the agenda group CleanUp
kcontext.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp" ).setFocus();
```

  (You can achieve the same using `drools.setFocus( "CleanUp" )`.)
- To run a query, you call `getQueryResults(String query)`, whereupon you may process the results, as explained in section Query.
- A set of methods dealing with event management lets you, among other things, add and remove event listeners for the Working Memory and the Agenda.
- Method `getKieBase()` returns the `KieBase` object, the backbone of all the Knowledge in your system, and the originator of the current session.
- You can manage globals with `setGlobal(...)`, `getGlobal(...)` and `getGlobals()`.
- Method `getEnvironment()` returns the runtime's `Environment` which works much like what you know as your operating system's environment.

8.8.4.2. The `modify` Statement

This language extension provides a structured approach to fact updates. It combines the update operation with a number of setter calls to change the object's fields. This is the syntax schema for the `modify` statement:

```
modify ( <fact-expression> ) {
    <expression> [ , <expression> ]*
}
```

The parenthesized *<fact-expression>* must yield a fact object reference. The expression list in the block should consist of setter calls for the given object, to be written without the usual object reference, which is automatically prepended by the compiler.

The example illustrates a simple fact modification.

Example 8.81. A modify statement

```
rule "modify stilton"when    $stilton : Cheese(type == "stilton")then    modify( $stilton ){
       setPrice( 20 ),        setAge( "overripe" )    }end

stilton"when    $stilton : Cheese(type
==
 "stilton")then    modify( $stilton
 ){        setPrice(
20 ),        setAge(
"overripe" )
```

The advantages in using the modify statment are particularly clear when used in conjuction with fine grained property change listeners. See the corresponding section for more details.

8.8.5. Conditional named consequences

Sometimes the constraint of having one single consequence for each rule can be somewhat limiting and leads to verbose and difficult to be maintained repetitions like in the following example:

```
rule "Give 10% discount to customers older than 60"when    $customer : Customer( age > 60 )then
    modify($customer) { setDiscount( 0.1 ) };endrule "Give free parking to customers older than
 60"when     $customer : Customer( age > 60 )     $car : Car ( owner == $customer )then
 modify($car) { setFreeParking( true ) };end
than
 60"when    $customer : Customer( age >
60
 )then    modify($customer) { setDiscount( 0.1
)

 };endrule "Give free parking to customers older
than
 60"when    $customer : Customer( age >
60 )    $car : Car ( owner ==
$customer
 )then    modify($car) { setFreeParking( true
)
```

It is already possible to partially overcome this problem by making the second rule extending the first one like in:

```
rule "Give 10% discount to customers older than 60"when    $customer : Customer( age > 60 )then
    modify($customer) { setDiscount( 0.1 ) };endrule "Give free parking to customers older than
 60"     extends "Give 10% discount to customers older than 60"when     $car : Car ( owner ==
 $customer )then    modify($car) { setFreeParking( true ) };end
than
 60"when    $customer : Customer( age >
60
 )then    modify($customer) { setDiscount( 0.1
)

 };endrule "Give free parking to customers older
than 60"    extends "Give 10% discount to customers older
than
 60"when    $car : Car ( owner ==
$customer
 )then    modify($car) { setFreeParking( true
)
```

Anyway this feature makes it possible to define more labelled consequences other than the default one in a single rule, so, for example, the 2 former rules can be compacted in only one like it follows:

```
rule "Give 10% discount and free parking to customers older than 60"when      $customer :
 Customer( age > 60 )    do[giveDiscount]    $car : Car ( owner == $customer )then    modify($car)
 { setFreeParking( true ) };then[giveDiscount]    modify($customer) { setDiscount( 0.1 ) };end

than 60"when    $customer : Customer( age >
 60 )
   do[giveDiscount]    $car : Car ( owner ==

$customer )then    modify($car) { setFreeParking( true

) };then[giveDiscount]    modify($customer) { setDiscount( 0.1
```

This last rule has 2 consequences, the usual default one, plus another one named "giveDiscount" that is activated, using the keyword do, as soon as a customer older than 60 is found in the knowledge base, regardless of the fact that he owns a car or not. The activation of a named consequence can be also guarded by an additional condition like in this further example:

```
rule "Give free parking to customers older than 60 and 10% discount to golden ones among them"when
     $customer : Customer( age > 60 )     if ( type == "Golden" ) do[giveDiscount]     $car :
 Car ( owner == $customer )then    modify($car) { setFreeParking( true ) };then[giveDiscount]
    modify($customer) { setDiscount( 0.1 ) };end
ones
 among them"when    $customer : Customer( age
> 60 )    if ( type ==
"Golden" ) do[giveDiscount]    $car : Car ( owner
==
```

```
 $customer )then    modify($car) { setFreeParking(
true
 ) };then[giveDiscount]    modify($customer) { setDiscount(
0.1
```

The condition in the if statement is always evaluated on the pattern immediately preceding it. In the end this last, a bit more complicated, example shows how it is possible to switch over different conditions using a nested if/else statement:

```
rule "Give free parking and 10% discount to over 60 Golden customer and 5% to Silver ones"when
      $customer : Customer( age > 60 )      if ( type == "Golden" ) do[giveDiscount10]
 else if ( type == "Silver" ) break[giveDiscount5]     $car : Car ( owner == $customer )then
       modify($car) { setFreeParking( true ) };then[giveDiscount10]       modify($customer)
 { setDiscount( 0.1 ) };then[giveDiscount5]    modify($customer) { setDiscount( 0.05 ) };end

Silver ones"when    $customer : Customer( age
> 60 )    if ( type ==
"Golden" ) do[giveDiscount10]    else if ( type ==
"Silver" ) break[giveDiscount5]    $car : Car ( owner
==
 $customer )then    modify($car) { setFreeParking(
true
 ) };then[giveDiscount10]    modify($customer) { setDiscount(
0.1
 ) };then[giveDiscount5]    modify($customer) { setDiscount(
0.05
```

Here the purpose is to give a 10% discount AND a free parking to Golden customers over 60, but only a 5% discount (without free parking) to the Silver ones. This result is achieved by activating the consequence named "giveDiscount5" using the keyword break instead of do. In fact do just schedules a consequence in the agenda, allowing the remaining part of the LHS to continue of being evaluated as per normal, while break also blocks any further pattern matching evaluation. Note, of course, that the activation of a named consequence not guarded by any condition with break doesn't make sense (and generates a compile time error) since otherwise the LHS part following it would be never reachable.

8.8.6. A Note on Auto-boxing and Primitive Types

Drools attempts to preserve numbers in their primitive or object wrapper form, so a variable bound to an int primitive when used in a code block or expression will no longer need manual unboxing; unlike Drools 3.0 where all primitives were autoboxed, requiring manual unboxing. A variable bound to an object wrapper will remain as an object; the existing JDK 1.5 and JDK 5 rules to handle auto-boxing and unboxing apply in this case. When evaluating field constraints, the system attempts to coerce one of the values into a comparable format; so a primitive is comparable to an object wrapper.

8.9. Query

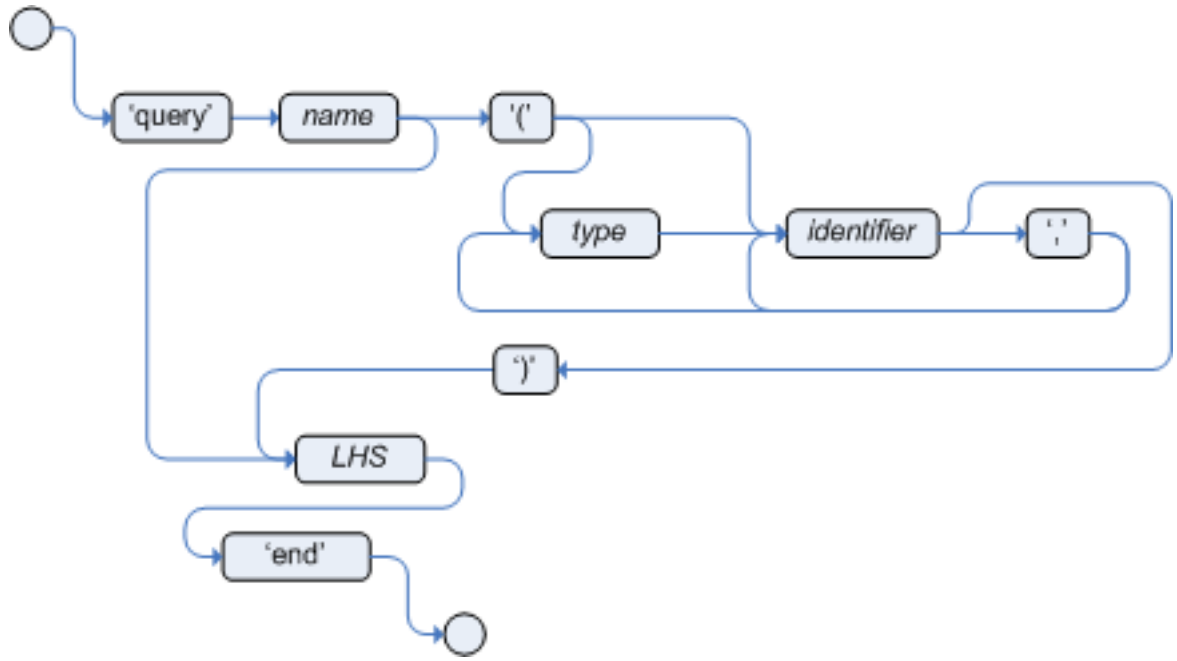


Figure 8.29. query

A query is a simple way to search the working memory for facts that match the stated conditions. Therefore, it contains only the structure of the LHS of a rule, so that you specify neither "when" nor "then". A query has an optional set of parameters, each of which can be optionally typed. If the type is not given, the type Object is assumed. The engine will attempt to coerce the values as needed. Query names are global to the KieBase; so do not add queries of the same name to different packages for the same RuleBase.

To return the results use `ksession.getQueryResults("name")`, where "name" is the query's name. This returns a list of query results, which allow you to retrieve the objects that matched the query.

The first example presents a simple query for all the people over the age of 30. The second one, using parameters, combines the age limit with a location.

Example 8.82. Query People over the age of 30

```
query "people over the age of 30"     person : Person( age > 30 )end
     person : Person( age > 30
)
```

Example 8.83. Query People over the age of x, and who live in y

```
query "people over the age of x"  (int x, String y)    person : Person( age > x, location == y )end
```

```
y)    person : Person( age > x, location == y
)
```

We iterate over the returned QueryResults using a standard "for" loop. Each element is a QueryResultsRow which we can use to access each of the columns in the tuple. These columns can be accessed by bound declaration name or index position.

Example 8.84. Query People over the age of 30

```
QueryResults    results    =    ksession.getQueryResults(    "people    over    the    age    of
  30"  );System.out.println(  "we  have  "  +  results.size()  +  "  people  over  the  age    of
 30" );System.out.println( "These people are are over 30:" );for ( QueryResultsRow row : results )
 {    Person person = ( Person ) row.get( "person" );    System.out.println( person.getName()
 + "\n" );}
30" );System.out.println( "we have " + results.size() + " people over the age  of

30" );System.out.println( "These people are are over

30:" );for ( QueryResultsRow row : results
) {    Person person = ( Person ) row.get(
"person" );    System.out.println( person.getName() +
"\n"
```

Support for positional syntax has been added for more compact code. By default the declared type order in the type declaration matches the argument position. But it possible to override these using the @position annotation. This allows patterns to be used with positional arguments, instead of the more verbose named arguments.

```
declare Cheese     name : String @position(1)     shop : String @position(2)     price : int
 @position(0)end
Cheese    name : String
@position(1)    shop : String
@position(2)    price : int
@position(0)
```

The @Position annotation, in the org.drools.definition.type package, can be used to annotate original pojos on the classpath. Currently only fields on classes can be annotated. Inheritance of classes is supported, but not interfaces or methods. The isContainedIn query below demonstrates the use of positional arguments in a pattern; `Location(x, y;)` instead of `Location( thing == x, location == y)`.

Queries can now call other queries, this combined with optional query arguments provides derivation query style backward chaining. Positional and named syntax is supported for arguments. It is also possible to mix both positional and named, but positional must come first, separated by a semi colon. Literal expressions can be passed as query arguments, but at this stage you cannot mix expressions with variables. Here is an example of a query that calls another query. Note that

'z' here will always be an 'out' variable. The '?' symbol means the query is pull only, once the results are returned you will not receive further results as the underlying data changes.

```
declare Location    thing : String    location : String endquery isContainedIn( String x, String
 y )     Location(x, y;)    or     ( Location(z, y;) and ?isContainedIn(x, z;) )end
Location    thing :
 String     location :


String endquery isContainedIn( String x, String y
 )
Location(x, y;)
 or     ( Location(z, y;) and ?isContainedIn(x,
z;)
```

As previously mentioned you can use live "open" queries to reactively receive changes over time from the query results, as the underlying data it queries against changes. Notice the "look" rule calls the query without using '?'.

```
query isContainedIn( String x, String y )      Location(x, y;)     or      ( Location(z, y;)
 and isContainedIn(x, z;) )endrule look when      Person( $l : likes )      isContainedIn( $l,
 'office'; )then   insertLogical( $l 'is in the office' );end
)     Location(x,
 y;)
or     ( Location(z, y;) and isContainedIn(x, z;)


)endrule look
when     Person( $l : likes
)     isContainedIn( $l, 'office';

)then   insertLogical( $l 'is in the office'
 );
```

Drools supports unification for derivation queries, in short this means that arguments are optional. It is possible to call queries from Java leaving arguments unspecified using the static field org.drools.core.runtime.rule.Variable.v - note you must use 'v' and not an alternative instance of Variable. These are referred to as 'out' arguments. Note that the query itself does not declare at compile time whether an argument is in or an out, this can be defined purely at runtime on each use. The following example will return all objects contained in the office.

```
results = ksession.getQueryResults( "isContainedIn", new Object[] {  Variable.v, "office" } );l =
 new ArrayList<List<String>>();for ( QueryResultsRow r : results ) {    l.add( Arrays.asList( new
 String[] { (String) r.get( "x" ), (String) r.get( "y" ) } ) );}
fice" } );l
= new ArrayList<List<String>>();for ( QueryResultsRow r :
results ) {    l.add( Arrays.asList( new String[] { (String) r.get( "x" ), (String) r.get( "y" )
} )
```

The algorithm uses stacks to handle recursion, so the method stack will not blow up.

The following is not yet supported:

- List and Map unification
- Variables for the fields of facts
- Expression unification - pred(X, X + 1, X * Y / 7)

8.10. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. Given a DSL, you write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files, and you can use any text editor to create and modify them. But there are also DSL and DSLR editors, both in the IDE as well as in the web based BRMS, and you can use those as well, although they may not provide you with the full DSL functionality.

8.10.1. When to Use a DSL

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modelling of domain object and the rule engine's native language and methods. If your rules need to be read and validated by domain experts (such as business analysts, for instance) who are not programmers, you should consider using a DSL; it hides implementation details and focuses on the rule logic proper. DSL sentences can also act as "templates" for conditional elements and consequence actions that are used repeatedly in your rules, possibly with minor variations. You may define DSL sentences as being mapped to these repeated phrases, with parameters providing a means for accommodating those variations.

DSLs have no impact on the rule engine at runtime, they are just a compile time feature, requiring a special parser and transformer.

8.10.2. DSL Basics

The Drools DSL mechanism allows you to customise conditional expressions and consequence actions. A global substitution mechanism ("keyword") is also available.

Example 8.85. Example DSL mapping

```
[when]Something is {colour}=Something(colour=="{colour}")
```

In the preceding example, `[when]` indicates the scope of the expression, i.e., whether it is valid for the LHS or the RHS of a rule. The part after the bracketed keyword is the expression that you

use in the rule; typically a natural language expression, but it doesn't have to be. The part to the right of the equal sign ("=") is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it ought to be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation. First, it extracts the string values appearing where the expression contains variable names in braces (here: `{colour}`). Then, the values obtained from these captures are then interpolated wherever that name, again enclosed in braces, occurs on the right hand side of the mapping. Finally, the interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

Note that the expressions (i.e., the strings on the left hand side of the equal sign) are used as regular expressions in a pattern matching operation against a line of the DSL rule file, matching all or part of a line. This means you can use (for instance) a '?' to indicate that the preceding character is optional. One good reason to use this is to overcome variations in natural language phrases of your DSL. But, given that these expressions are regular expression patterns, this also means that all "magic" characters of Java's pattern syntax have to be escaped with a preceding backslash ('\').

It is important to note that the compiler transforms DSL rule files line by line. In the above example, all the text after "Something is " to the end of the line is captured as the replacement value for "{colour}", and this is used for interpolating the target string. This may not be exactly what you want. For instance, when you intend to merge different DSL expressions to generate a composite DRL pattern, you need to transform a DSLR line in several independent operations. The best way to achieve this is to ensure that the captures are surrounded by characteristic text - words or even single characters. As a result, the matching operation done by the parser plucks out a substring from somewhere within the line. In the example below, quotes are used as distinctive characters. Note that the characters that surround the capture are not included during interpolation, just the contents between them.

As a rule of thumb, use quotes for textual data that a rule editor may want to enter. You can also enclose the capture with words to ensure that the text is correctly matched. Both is illustrated by the following example. Note that a single line such as `Something is "green" and another solid thing` is now correctly expanded.

Example 8.86. Example with quotes

```
[when]something is "{colour}"=Something(colour=="{colour}")
[when]another {state} thing=OtherThing(state=="{state})"
```

It is a good idea to avoid punctuation (other than quotes or apostrophes) in your DSL expressions as much as possible. The main reason is that punctuation is easy to forget for rule authors using your DSL. Another reason is that parentheses, the period and the question mark are magic characters, requiring escaping in the DSL definition.

In a DSL mapping, the braces "{" and "}" should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\"):

```
[then]do something= if (foo) \{ doSomething(); \}
 \}
```

Note

If braces "{" and "}" should appear in the replacement string of a DSL definition, escape them with a backslash ('\').

Example 8.87. Examples of DSL mapping entries

```
# This is a comment to be ignored.[when]There is a person with name of
   "{name}"=Person(name=="{name}")[when]Person is at least {age} years old and lives
   in "{location}"= Person(age >= {age}, location=="{location}")[then]Log
 "{message}"=System.out.println("{message}");[when]And = and
ignored.[when]There is a person with name of
 "{name}"=Person(name=="{name}")[when]Person is at least {age} years old and lives in
 "{location}"= Person(age >= {age},
 location=="{location}")[then]Log
 "{message}"=System.out.println("{message}");[when]And =
```

Given the above DSL examples, the following examples show the expansion of various DSLR snippets:

Example 8.88. Examples of DSL expansions

```
There is a person with name of "Kitty" ==> Person(name="Kitty")Person is at least 42
 years old and lives in "Atlanta" ==> Person(age >= 42, location="Atlanta")Log "boo" ==>
 System.out.println("boo");There is a person with name of "Bob" and Person is at least 30 years
 old and lives in "Utah" ==> Person(name="Bob") and Person(age >= 30, location="Utah")
"Kitty" ==>
 Person(name="Kitty")Person is at least 42 years old and lives in
 "Atlanta" ==> Person(age >= 42,
 location="Atlanta")Log
 "boo" ==>
 System.out.println("boo");There is a person with name of "Bob" and Person is at least 30 years old and lives in
 "Utah" ==> Person(name="Bob") and Person(age >= 30,
```

Note

Don't forget that if you are capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

You can chain DSL expressions together on one line, as long as it is clear to the parser where one ends and the next one begins and where the text representing a parameter ends. (Otherwise you risk getting all the text until the end of the line as a parameter value.) The DSL expressions are tried, one after the other, according to their order in the DSL definition file. After any match, all remaining DSL expressions are investigated, too.

The resulting DRL text may consist of more than one line. Line ends are in the replacement text are written as `\n`.

8.10.3. Adding Constraints to Facts

A common requirement when writing rule conditions is to be able to add an arbitrary combination of constraints to a pattern. Given that a fact type may have many fields, having to provide an individual DSL statement for each combination would be plain folly.

The DSL facility allows you to add constraints to a pattern by a simple convention: if your DSL expression starts with a hyphen (minus character, "-") it is assumed to be a field constraint and, consequently, is is added to the last pattern line preceding it.

For an example, lets take look at class `Cheese`, with the following fields: type, price, age and country. We can express some LHS condition in normal DRL like the following

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

The DSL definitions given below result in three DSL phrases which may be used to create any combination of constraint involving these fields.

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

You can then write rules with conditions like the following:

```
There is a Cheese with
        - age is less than 42
        - type is 'stilton'
```

The parser will pick up a line beginning with "-" and add it as a constraint to the preceding pattern, inserting a comma when it is required. For the preceding example, the resulting DRL is:

```
Cheese(age<42, type=='stilton')
```

Combining all all numeric fields with all relational operators (according to the DSL expression "age is less than..." in the preceding example) produces an unwieldy amount of DSL entries. But you can define DSL phrases for the various operators and even a generic expression that handles any field constraint, as shown below. (Notice that the expression definition contains a regular expression in addition to the variable name.)

```
[when][]is less than or equal to=<=
[when][]is less than=<
[when][]is greater than or equal to=>=
[when][]is greater than=>
[when][]is equal to===
[when][]equals===
[when][]There is a Cheese with=Cheese()
[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}
```

Given these DSL definitions, you can write rules with conditions such as:

```
There is a Cheese with
   - age is less than 42
   - rating is greater than 50
   - type equals 'stilton'
```

In this specific case, a phrase such as "is less than" is replaced by <, and then the line matches the last DSL entry. This removes the hyphen, but the final result is still added as a constraint to the preceding pattern. After processing all of the lines, the resulting DRL text is:

```
Cheese(age<42, rating > 50, type=='stilton')
```

Note

The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.

8.10.4. Developing a DSL

A good way to get started is to write representative samples of the rules your application requires, and to test them as you develop. This will provide you with a stable framework of conditional

elements and their constraints. Rules, both in DRL and in DSLR, refer to entities according to the data model representing the application data that should be subject to the reasoning process defined in rules. Notice that writing rules is generally easier if most of the data model's types are facts.

Given an initial set of rules, it should be possible to identify recurring or similar code snippets and to mark variable parts as parameters. This provides reliable leads as to what might be a handy DSL entry. Also, make sure you have a full grasp of the jargon the domain experts are using, and base your DSL phrases on this vocabulary.

You may postpone implementation decisions concerning conditions and actions during this first design phase by leaving certain conditional elements and actions in their DRL form by prefixing a line with a greater sign (">"). (This is also handy for inserting debugging statements.)

During the next development phase, you should find that the DSL configuration stabilizes pretty quickly. New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.

Try to keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints. But do not exaggerate: authors using the DSL should still be able to identify DSL phrases by some fixed text.

8.10.5. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file according to DRL syntax.

- A line starting with "#" or "//" (with or without preceding white space) is treated as a comment. A comment line starting with "#/" is scanned for words requesting a debug option, see below.
- Any line starting with an opening bracket ("[") is assumed to be the first line of a DSL entry definition.
- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

A DSL entry consists of the following four parts:

- A scope definition, written as one of the keywords "when" or "condition", "then" or "consequence", "*" and "keyword", enclosed in brackets ("[" and "]"). This indicates whether the DSL entry is valid for the condition or the consequence of a rule, or both. A scope indication of "keyword" means that the entry has global significance, i.e., it is recognized anywhere in a DSLR file.
- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the the next part begins with an opening bracket. An empty pair of brackets is valid, too.
- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions,* terminated by an equal sign ("="). A variable definition is enclosed in braces

("{" and "}"). It consists of a variable name and two optional attachments, separated by colons (":"). If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable; if there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash ("\") if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in braces. Optionally, the variable name may be followed by an exclamation mark ("!") and a transformation function, see below.

  Note that braces ("{" and "}") must be escaped with a preceding backslash ("\") if they should occur literally within the replacement string.

Debugging of DSL expansion can be turned on, selectively, by using a comment line starting with "#/" which may contain one or more words from the table presented below. The resulting output is written to standard output.

Table 8.2. Debug options for DSL expansion

Word	Description
result	Prints the resulting DRL text, with line numbers.
steps	Prints each expansion step of condition and consequence lines.
keyword	Dumps the internal representation of all DSL entries with scope "keyword".
when	Dumps the internal representation of all DSL entries with scope "when" or "*".
then	Dumps the internal representation of all DSL entries with scope "then" or "*".
usage	Displays a usage statistic of all DSL entries.

Below are some sample DSL definitions, with comments describing the language features they illustrate.

```
# Comment: DSL examples#/ debug: display result and usage# keyword definition: replaces "regula"
 by "rule"[keyword][]regula=rule# conditional element: "T" or "t", "a" or "an", convert matched
 word[when][][Tt]here is an? {entity:\w+}=       ${entity!lc}: {entity!ucfirst} ()# consequence
 statement: convert matched word, literal braces[then][]update {entity:\w+}=modify( ${entity!
lc} )\{ \}

examples#/ debug: display result

and usage# keyword definition: replaces "regula"
by
```

```
"rule"[keyword][]regula=rule# conditional element: "T" or "t", "a" or "an", convert
matched word[when][][Tt]here is an?
{entity:\w+}=        ${entity!lc}: {entity!

ucfirst} ()# consequence statement: convert matched word,
literal braces[then][]update {entity:\w+}=modify( ${entity!lc}
```

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.

2. Each of the "keyword" entries is applied to the entire text. First, the regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default (".*?"). Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.

3. Sections of the DSLR text between "when" and "then", and "then" and "end", respectively, are located and processed in a uniform manner, line by line, as described below.

   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ".*?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

   Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

   If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, i.e., a type name followed by a pair of parentheses. if this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma (",") is inserted beforehand.

   If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a "modify" statement, ending in a pair of braces ("{" and "}"). If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma (",") is inserted beforehand.

Note

It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., "accumulate") or it may only work for the first insertion (e.g., "eval").

All string transformation functions are described in the following table.

Table 8.3. String transformation functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.
ucfirst	Converts the first letter to upper case, and all other letters to lower case.
num	Extracts all digits and "-" from the string. If the last two digits in the original string are preceded by "." or ",", a decimal period is inserted in the corresponding position.
a?*b*/*c*	Compares the string with string *a*, and if they are equal, replaces it with *b*, otherwise with *c*. But *c* can be another triplet *a*, *b*, *c*, so that the entire structure is, in fact, a translation table.

The following DSL examples show how to use string transformation functions.

```
# definitions for conditions[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()[when]
[]- with an? {attr} greater than {amount}={attr} <= {amount!num}[when][]- with a {what}
 {attr}={attr} {what!positive?>0/negative?%lt;0/zero?==0/ERROR}
tions for conditions[when][]There is an?
{entity}=${entity!lc}: {entity!ucfirst}()[when][]- with an? {attr} greater than
 {amount}={attr} <= {amount!num}[when][]- with a {what} {attr}={attr} {what!
```

A file containing a DSL definition has to be put under the resources folder or any of its subfolders like any other drools artifact. It must have the extension `.dsl`, or alternatively be marked with type `ResourceType.DSL`. when programmatically added to a `KieFileSystem`. For a file using DSL definition, the extension `.dslr` should be used, while it can be added to a `KieFileSystem` with type `ResourceType.DSLR`.

For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. Thus, the parser can "recognize" the DSL expressions and transform them into native rule language expressions.

Chapter 9. Complex Event Processing

9.1. Complex Event Processing

There is no broadly accepted definition on the term Complex Event Processing. The term Event by itself is frequently overloaded and used to refer to several different things, depending on the context it is used. Defining terms is not the goal of this guide and as so, lets adopt a loose definition that, although not formal, will allow us to proceed with a common understanding.

So, in the scope of this guide:

Important

Event, is a record of a significant change of state in the application domain at a given point in time.

For instance, on a Stock Broker application, when a sale operation is executed, it causes a change of state in the domain. This change of state can be observed on several entities in the domain, like the price of the securities that changed to match the value of the operation, the ownership of the traded assets that changed from the seller to the buyer, the balance of the accounts from both seller and buyer that are credited and debited, etc. Depending on how the domain is modelled, this change of state may be represented by a single event, multiple atomic events or even hierarchies of correlated events. In any case, in the context of this guide, Event is the record of the change of a particular piece of data in the domain.

Events are processed by computer systems since they were invented, and throughout the history, systems responsible for that were given different names and different methodologies were employed. It wasn't until the 90's though, that a more focused work started on EDA (Event Driven Architecture) with a more formal definition on the requirements and goals for event processing. Old messaging systems started to change to address such requirements and new systems started to be developed with the single purpose of event processing. Two trends were born under the names of Event Stream Processing and Complex Event Processing.

In the very beginnings, Event Stream Processing was focused on the capabilities of processing streams of events in (near) real time, while the main focus of Complex Event Processing was on the correlation and composition of atomic events into complex (compound) events. An important (maybe the most important) milestone was the publishing of Dr. David Luckham's book "The Power of Events" in 2002. In the book, Dr Luckham introduces the concept of Complex Event Processing and how it can be used to enhance systems that deal with events. Over the years, both trends converged to a common understanding and today these systems are all referred to as CEP systems.

This is a very simplistic explanation to a really complex and fertile field of research, but sets a high level and common understanding of the concepts that this guide will introduce.

The current understanding of what Complex Event Processing is may be briefly described as the following quote from Wikipedia:

Important

"**Complex Event Processing**, or CEP, is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud. CEP employs techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events such as causality, membership, and timing, and event-driven processes."

—Wikipedia [http://en.wikipedia.org/wiki/Complex_event_processing]

In other words, CEP is about detecting and selecting the interesting events (and only them) from an event cloud, finding their relationships and inferring new data from them and their relationships.

Note

For the remaining of this guide, we will use the terms **Complex Event Processing** and **CEP** as a broad reference for any of the related technologies and techniques, including but not limited to, CEP, Complex Event Processing, ESP, Event Stream Processing and Event Processing in general.

9.2. Drools Fusion

Event Processing use cases, in general, share several requirements and goals with Business Rules use cases. These overlaps happen both on the business side and on the technical side.

On the Business side:

- Business rules are frequently defined based on the occurrence of scenarios triggered by events. Examples could be:
  - On an algorithmic trading application: take an action if the security price increases X% compared to the day opening price, where the price increases are usually denoted by events on a Stock Trade application.
  - On a monitoring application: take an action if the temperature on the server room increases X degrees in Y minutes, where sensor readings are usually denoted by events.

- Both business rules and event processing queries change frequently and require immediate response for the business to adapt itself to new market conditions, new regulations and new enterprise policies.

From a technical perspective:

- Both require seamless integration with the enterprise infrastructure and applications, specially on autonomous governance, including, but not limited to, lifecycle management, auditing, security, etc.
- Both have functional requirements like pattern matching and non-functional requirements like response time and query/rule explanation.

Even sharing requirements and goals, historically, both fields were born appart and although the industry evolved and one can find good products on the market, they either focus on event processing or on business rules management. That is due not only because of historical reasons but also because, even overlapping in part, use cases do have some different requirements.

Important

Drools was also born as a rules engine several years ago, but following the vision of becoming a single platform for behavioral modelling, it soon realized that it could only achieve this goal by crediting the same importance to the three complementary business modelling techniques:

- Business Rules Management
- Business Processes Management
- Complex Event Processing

In this context, Drools Fusion is the module responsible for adding event processing capabilities into the platform.

Supporting Complex Event Processing, though, is much more than simply understanding what an event is. CEP scenarios share several common and distinguishing characteristics:

- Usually required to process huge volumes of events, but only a small percentage of the events are of real interest.
- Events are usually immutable, since they are a record of state change.
- Usually the rules and queries on events must run in reactive modes, i.e., react to the detection of event patterns.

- Usually there are strong temporal relationships between related events.
- Individual events are usually not important. The system is concerned about patterns of related events and their relationships.
- Usually, the system is required to perform composition and aggregation of events.

Based on this general common characteristics, Drools Fusion defined a set of goals to be achieved in order to support Complex Event Processing appropriately:

- Support Events, with their proper semantics, as first class citizens.
- Allow detection, correlation, aggregation and composition of events.
- Support processing of Streams of events.
- Support temporal constraints in order to model the temporal relationships between events.
- Support sliding windows of interesting events.
- Support a session scoped unified clock.
- Support the required volumes of events for CEP use cases.
- Support to (re)active rules.
- Support adapters for event input into the engine (pipeline).

The above list of goals are based on the requirements not covered by Drools Expert itself, since in a unified platform, all features of one module are leveraged by the other modules. This way, Drools Fusion is born with enterprise grade features like Pattern Matching, that is paramount to a CEP product, but that is already provided by Drools Expert. In the same way, all features provided by Drools Fusion are leveraged by Drools Flow (and vice-versa) making process management aware of event processing and vice-versa.

For the remaining of this guide, we will go through each of the features Drools Fusion adds to the platform. All these features are available to support different use cases in the CEP world, and the user is free to select and use the ones that will help him model his business use case.

9.3. Event Semantics

An *event* is a fact that present a few distinguishing characteristics:

- **Usually immutables:** since, by the previously discussed definition, events are a record of a state change in the application domain, i.e., a record of something that already happened, and the past can not be "changed", events are immutables. This constraint is an important require-

ment for the development of several optimizations and for the specification of the event lifecycle. This does not mean that the Java object representing the object must be immutable. Quite the contrary, the engine does not enforce immutability of the object model, because one of the most common use cases for rules is event data enrichment.

Note

As a best practice, the application is allowed to populate un-populated event attributes (to enrich the event with inferred data), but already populated attributes should never be changed.

- **Strong temporal constraints:** rules involving events usually require the correlation of multiple events, specially temporal correlations where events are said to happen at some point in time relative to other events.
- **Managed lifecycle:** due to their immutable nature and the temporal constraints, events usually will only match other events and facts during a limited window of time, making it possible for the engine to manage the lifecycle of the events automatically. In other words, one an event is inserted into the working memory, it is possible for the engine to find out when an event can no longer match other facts and automatically delete it, releasing its associated resources.
- **Use of sliding windows:** since all events have timestamps associated to them, it is possible to define and use sliding windows over them, allowing the creation of rules on aggregations of values over a period of time. Example: average of an event value over 60 minutes.

Drools supports the declaration and usage of events with both semantics: **point-in-time** events and **interval-based** events.

Note

A simplistic way to understand the unitification of the semantics is to consider a *point-in-time* event as an *interval-based* event whose *duration is zero*.

9.4. Event Processing Modes

Rules engines in general have a well known way of processing data and rules and provide the application with the results. Also, there is not many requirements on how facts should be presented to the rules engine, specially because in general, the processing itself is time independent. That is a good assumption for most scenarios, but not for all of them. When the requirements include the processing of real time or near real time events, time becomes and important variable of the reasoning process.

The following sections will explain the impact of time on rules reasoning and the two modes provided by Drools for the reasoning process.

9.4.1. Cloud Mode

The CLOUD processing mode is the default processing mode. Users of rules engine are familiar with this mode because it behaves in exactly the same way as any pure forward chaining rules engine, including previous versions of Drools.

When running in CLOUD mode, the engine sees all facts in the working memory, does not matter if they are regular facts or events, as a whole. There is no notion of flow of time, although events have a timestamp as usual. In other words, although the engine knows that a given event was created, for instance, on January 1st 2009, at 09:35:40.767, it is not possible for the engine to determine how "old" the event is, because there is no concept of "now".

In this mode, the engine will apply its usual many-to-many pattern matching algorithm, using the rules constraints to find the matching tuples, activate and fire rules as usual.

This mode does not impose any kind of additional requirements on facts. So for instance:

- There is no notion of time. No requirements clock synchronization.
- There is no requirement on event ordering. The engine looks at the events as an unordered cloud against which the engine tries to match rules.

On the other hand, since there is no requirements, some benefits are not available either. For instance, in CLOUD mode, it is not possible to use sliding windows, because sliding windows are based on the concept of "now" and there is no concept of "now" in CLOUD mode.

Since there is no ordering requirement on events, it is not possible for the engine to determine when events can no longer match and as so, there is no automatic life-cycle management for events. I.e., the application must explicitly delete events when they are no longer necessary, in the same way the application does with regular facts.

Cloud mode is the default execution mode for Drools, but in any case, as any other configuration in Drools, it is possible to change this behavior either by setting a system property, using configuration property files or using the API. The corresponding property is:

```
KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();
config.setOption( EventProcessingOption.CLOUD );
```

The equivalent property is:

```
drools.eventProcessingMode = cloud
```

9.4.2. Stream Mode

The STREAM processing mode is the mode of choice when the application needs to process streams of events. It adds a few common requirements to the regular processing, but enables a whole lot of features that make stream event processing a lot simpler.

The main requirements to use STREAM mode are:

- Events in each stream must be time-ordered. I.e., inside a given stream, events that happened first must be inserted first into the engine.
- The engine will force synchronization between streams through the use of the session clock, so, although the application does not need to enforce time ordering between streams, the use of non-time-synchronized streams may result in some unexpected results.

Given that the above requirements are met, the application may enable the STREAM mode using the following API:

```
KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();
config.setOption( EventProcessingOption.STREAM );
```

Or, the equivalent property:

```
drools.eventProcessingMode = stream
```

When using the STREAM, the engine knows the concept of flow of time and the concept of "now", i.e., the engine understands how old events are based on the current timestamp read from the Session Clock. This characteristic allows the engine to provide the following additional features to the application:

- Sliding Window support
- Automatic Event Lifecycle Management
- Automatic Rule Delaying when using Negative Patterns

All these features are explained in the following sections.

9.4.2.1. Role of Session Clock in Stream mode

When running the engine in CLOUD mode, the session clock is used only to time stamp the arriving events that don't have a previously defined timestamp attribute. Although, in STREAM mode, the Session Clock assumes an even more important role.

In STREAM mode, the session clock is responsible for keeping the current timestamp, and based on it, the engine does all the temporal calculations on event's aging, synchronizes streams from multiple sources, schedules future tasks and so on.

Check the documentation on the Session Clock section to know how to configure and use different session clock implementations.

9.4.2.2. Negative Patterns in Stream Mode

Negative patterns behave different in STREAM mode when compared to CLOUD mode. In CLOUD mode, the engine assumes that all facts and events are known in advance (there is no concept of flow of time) and so, negative patterns are evaluated immediately.

When running in STREAM mode, negative patterns with temporal constraints may require the engine to wait for a time period before activating a rule. The time period is automatically calculated by the engine in a way that the user does not need to use any tricks to achieve the desired result.

For instance:

Example 9.1. a rule that activates immediately upon matching

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

The above rule has no temporal constraints that would require delaying the rule, and so, the rule activates immediately. The following rule on the other hand, must wait for 10 seconds before activating, since it may take up to 10 seconds for the sprinklers to activate:

Example 9.2. a rule that automatically delays activation due to temporal constraints

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

This behaviour allows the engine to keep consistency when dealing with negative patterns and temporal constraints at the same time. The above would be the same as writing the rule as below, but does not burden the user to calculate and explicitly write the appropriate duration parameter:

Example 9.3. same rule with explicit duration parameter

```
rule "Sound the alarm"
```

```
    duration( 10s )
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

The following rule expects every 10 seconds at least one “Heartbeat” event, if not the rule fires. The special case in this rule is that we use the same type of the object in the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait between 0 to 10 seconds before firing and it excludes the Heartbeat bound to $h. Excluding the bound Heartbeat is important since the temporal constraint [0s, ...] does not exclude by itself the bound event $h from being matched again, thus preventing the rule to fire.

Example 9.4. excluding bound events in negative patterns

```
rule "Sound the alarm"
when
    $h: Heartbeat( ) from entry-point "MonitoringStream"
    not( Heartbeat( this != $h, this after[0s,10s] $h ) from entry-point "MonitoringStream" )
then
    // Sound the alarm
end
```

9.5. Session Clock

Reasoning over time requires a reference clock. Just to mention one example, if a rule reasons over the average price of a given stock over the last 60 minutes, how the engine knows what stock price changes happened over the last 60 minutes in order to calculate the average? The obvious response is: by comparing the timestamp of the events with the "current time". How the engine knows what **time is now**? Again, obviously, by querying the Session Clock.

The session clock implements a strategy pattern, allowing different types of clocks to be plugged and used by the engine. This is very important because the engine may be running in an elements of different scenarios that may require different clock implementations. Just to mention a few:

- **Rules testing:** testing always requires a controlled environment, and when the tests include rules with temporal constraints, it is necessary to not only control the input rules and facts, but also the flow of time.
- **Regular execution:** usually, when running rules in production, the application will require a real time clock that allows the rules engine to react immediately to the time progression.
- **Special environments:** specific environments may have specific requirements on time control. Cluster environments may require clock synchronization through heart beats, or JEE environments may require the use of an AppServer provided clock, etc.

- **Rules replay or simulation:** to replay scenarios or simulate scenarios it is necessary that the application also controls the flow of time.

9.5.1. Available Clock Implementations

Drools 5 provides 2 clock implementations out of the box. The default real time clock, based on the system clock, and an optional pseudo clock, controlled by the application.

9.5.1.1. Real Time Clock

By default, Drools uses a real time clock implementation that internally uses the system clock to determine the current timestamp.

To explicitly configure the engine to use the real time clock, just set the session configuration parameter to real time:

```
KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("realtime") );
```

9.5.1.2. Pseudo Clock

Drools also offers out of the box an implementation of a clock that is controlled by the application that is called Pseudo Clock. This clock is specially useful for unit testing temporal rules since it can be controlled by the application and so the results become deterministic.

To configure the pseudo session clock, do:

```
KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("pseudo") );
```

As an example of how to control the pseudo session clock:

```
KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();
conf.setOption( ClockTypeOption.get( "pseudo" ) );
KieSession session = kbase.newKieSession( conf, null );

SessionPseudoClock clock = session.getSessionClock();

// then, while inserting facts, advance the clock as necessary:
FactHandle handle1 = session.insert( tick1 );
clock.advanceTime( 10, TimeUnit.SECONDS );
FactHandle handle2 = session.insert( tick2 );
clock.advanceTime( 30, TimeUnit.SECONDS );
FactHandle handle3 = session.insert( tick3 );
```

9.6. Sliding Windows

Sliding Windows are a way to scope the events of interest by defining a window that is constantly moving. The two most common types of sliding window implementations are time based windows and length based windows.

The next sections will detail each of them.

Important

Sliding Windows are only available when running the engine in STREAM mode. Check the Event Processing Mode section for details on how the STREAM mode works.

Important

Sliding windows start to match immediately and defining a sliding window does not imply that the rule has to wait for the sliding window to be "full" in order to match. For instance, a rule that calculates the average of an event property on a window:length(10) will start calculating the average immediately, and it will start at 0 (zero) for no-events, and will update the average as events arrive one by one.

9.6.1. Sliding Time Windows

Sliding Time Windows allow the user to write rules that will only match events occurring in the last X time units.

For instance, if the user wants to consider only the Stock Ticks that happened in the last 2 minutes, the pattern would look like this:

```
StockTick() over window:time( 2m )
```

Drools uses the "over" keyword to associate windows to patterns.

On a more elaborate example, if the user wants to sound an alarm in case the average temperature over the last 10 minutes read from a sensor is above the threshold value, the rule would look like:

Example 9.5. aggregating values over time windows

```
rule "Sound the alarm in case temperature rises above threshold"when
 TemperatureThreshold( $max : max )    Number( doubleValue > $max ) from accumulate(
 SensorReading( $temp : temperature ) over window:time( 10m ),         average( $temp ) )then
    // sound the alarmend
```

```
above threshold"when    TemperatureThreshold( $max
: max )    Number( doubleValue > $max
) from accumulate(        SensorReading( $temp : temperature ) over
window:time( 10m ),        average(
$temp
 ) )then    //
sound
```

The engine will automatically disregard any SensorReading older than 10 minutes and keep the calculated average consistent.

Important

Please note that time based windows are considered when calculating the interval an event remains in the working memory before being expired, but an event falling off a sliding window does not mean by itself that the event will be discarded from the working memory, as there might be other rules that depend on that event. The engine will discard events only when no other rules depend on that event and the expiration policy for that event type is fulfilled.

9.6.2. Sliding Length Windows

Sliding Length Windows work the same way as Time Windows, but consider events based on order of their insertion into the session instead of flow of time.

For instance, if the user wants to consider only the last 10 RHT Stock Ticks, independent of how old they are, the pattern would look like this:

```
StockTick( company == "RHT" ) over window:length( 10 )
```

As you can see, the pattern is similar to the one presented in the previous section, but instead of using window:time to define the sliding window, it uses window:length.

Using a similar example to the one in the previous section, if the user wants to sound an alarm in case the average temperature over the last 100 readings from a sensor is above the threshold value, the rule would look like:

Example 9.6. aggregating values over length windows

```
rule   "Sound   the   alarm   in   case   temperature   rises   above   threshold"when
 TemperatureThreshold( $max : max )     Number( doubleValue > $max ) from accumulate(
 SensorReading( $temp : temperature ) over window:length( 100 ),        average( $temp ) )then
    // sound the alarmend
```

```
above threshold"when    TemperatureThreshold( $max
: max )    Number( doubleValue > $max
) from accumulate(        SensorReading( $temp : temperature ) over
window:length( 100 ),        average(
$temp
 ) )then    //
sound
```

The engine will keep only consider the last 100 readings to calculate the average temperature.

Important

Please note that falling off a length based window is not criteria for event expiration in the session. The engine disregards events that fall off a window when calculating that window, but does not remove the event from the session based on that condition alone as there might be other rules that depend on that event.

Important

Please note that length based windows do not define temporal constraints for event expiration from the session, and the engine will not consider them. If events have no other rules defining temporal constraints and no explicit expiration policy, the engine will keep them in the session indefinitely.

When using a sliding window, alpha constraints are evaluated before the window is considered, but beta (join) constraints are evaluated afterwards. This usually doesn't make a difference when time windows are concerned, but it's important when using a length window. For example this pattern:

```
StockTick( company == "RHT" ) over window:length( 10 )
```

defines a window of (at most) 10 StockTicks all having company equal to "RHT", while the following one:

```
$s : String()
StockTick( company == $s ) over window:length( 10 )
```

first creates a window of (at most) 10 StockTicks regardless of the value of their company attribute and then filters among them only the ones having the company equal to the String selected from the working memory.

9.6.3. Window Declaration

The engine also supports the declaration of Windows. This promotes a clear separation between what are the filters applied to the window and what are the constraints applied to the result of window. It also allows easy reuse of windows among multiple rules.

Another benefit is a new implementation of the basic window support in the engine, increasing the overall performance of the rules that use sliding windows.

The simplified EBNF to declare a window is:

```
windowDeclaration := DECLARE WINDOW ID annotation* lhsPatternBind END
```

For example a window containing only the last 10 stock ticks from a given source can be defined like:

```
declare window Ticks     StockTick( source == "NYSE" )          over window:length( 10 )
    from entry-point STStreamend
 Ticks    StockTick( source == "NYSE"
 )        over window:length( 10
 )        from entry-point
```

Rules can then use this declared window by using it as a source for a FROM as in:

```
rule "RHT ticks in the window"     when         accumulate( StockTick( company == "RHT" ) from
 window Ticks,                        $cnt : count(1) )     then         // there has been $cnt
 RHT ticks over the last 10 ticksend
dow"
when        accumulate( StockTick( company == "RHT" ) from window
Ticks,                    $cnt : count(1)
)
then        // there has been $cnt RHT ticks over the last 10
ticks
```

Note that this example also demonstrates how the window declaration allows to separate the constraints applied to the window (only the StockTicks having "NYSE" as source are among the 10 events included into window) and the constraints applied to the window result (among the last 10 events having "NYSE" as source only the ones with company equal to "RHT" are selected).

9.7. Streams Support

Most CEP use cases have to deal with streams of events. The streams can be provided to the application in various forms, from JMS queues to flat text files, from database tables to raw sockets or even through web service calls. In any case, the streams share a common set of characteristics:

- events in the stream are ordered by a timestamp. The timestamp may have different semantics for different streams but they are always ordered internally.
- volumes of events are usually high.
- atomic events are rarely useful by themselves. Usually meaning is extracted from the correlation between multiple events from the stream and also from other sources.
- streams may be homogeneous, i.e. contain a single type of events, or heterogeneous, i.e. contain multiple types of events.

Drools generalized the concept of a stream as an "entry point" into the engine. An entry point is for drools a gate from which facts come. The facts may be regular facts or special facts like events.

In Drools, facts from one entry point (stream) may join with facts from any other entry point or event with facts from the working memory. Although, they never mix, i.e., they never lose the reference to the entry point through which they entered the engine. This is important because one may have the same type of facts coming into the engine through several entry points, but one fact that is inserted into the engine through entry point A will never match a pattern from a entry point B, for example.

9.7.1. Declaring and Using Entry Points

Entry points are declared implicitly in Drools by directly making use of them in rules. I.e. referencing an entry point in a rule will make the engine, at compile time, to identify and create the proper internal structures to support that entry point.

So, for instance, lets imagine a banking application, where transactions are fed into the system coming from streams. One of the streams contains all the transactions executed in ATM machines. So, if one of the rules says: a withdraw is authorized if and only if the account balance is over the requested withdraw amount, the rule would look like:

Example 9.7. Example of Stream Usage

```
rule "authorize withdraw"when   WithdrawRequest( $ai : accountId, $am : amount ) from entry-point
 "ATM Stream"   CheckingAccount( accountId == $ai, balance > $am )then   // authorize withdrawend

withdraw"when    WithdrawRequest( $ai : accountId, $am : amount ) from entry-point
"ATM Stream"    CheckingAccount( accountId == $ai, balance >
$am
 )then    //
authorize
```

In the previous example, the engine compiler will identify that the pattern is tied to the entry point "ATM Stream" and will both create all the necessary structures for the rulebase to support the "ATM Stream" and will only match WithdrawRequests coming from the "ATM Stream". In the previous example, the rule is also joining the event from the stream with a fact from the main working memory (CheckingAccount).

Now, lets imagine a second rule that states that a fee of $2 must be applied to any account for which a withdraw request is placed at a bank branch:

Example 9.8. Using a different Stream

```
rule "apply fee on withdraws on branches"when     WithdrawRequest( $ai : accountId, processed
 == true ) from entry-point "Branch Stream"     CheckingAccount( accountId == $ai )then     //
 apply a $2 fee on the accountend

branches"when    WithdrawRequest( $ai : accountId, processed == true ) from entry-point
"Branch Stream"    CheckingAccount( accountId ==
$ai
 )then    // apply a $2 fee on
the
```

The previous rule will match events of the exact same type as the first rule (WithdrawRequest), but from two different streams, so an event inserted into "ATM Stream" will never be evaluated against the pattern on the second rule, because the rule states that it is only interested in patterns coming from the "Branch Stream".

So, entry points, besides being a proper abstraction for streams, are also a way to scope facts in the working memory, and a valuable tool for reducing cross products explosions. But that is a subject for another time.

Inserting events into an entry point is equally simple. Instead of inserting events directly into the working memory, insert them into the entry point as shown in the example below:

Example 9.9. Inserting facts into an entry point

```
// create your rulebase and your session as usual
KieSession session = ...

// get a reference to the entry point
EntryPoint atmStream = session.getEntryPoint( "ATM Stream" );

// and start inserting your facts into the entry point
atmStream.insert( aWithdrawRequest );
```

The previous example shows how to manually insert facts into a given entry point. Although, usually, the application will use one of the many adapters to plug a stream end point, like a JMS queue, directly into the engine entry point, without coding the inserts manually. The Drools pipeline API has several adapters and helpers to do that as well as examples on how to do it.

9.8. Memory Management for Events

Important

The automatic memory management for events is only performed when running the engine in STREAM mode. Check the Event Processing Mode section for details on how the STREAM mode works.

One of the benefits of running the engine in STREAM mode is that the engine can detect when an event can no longer match any rule due to its temporal constraints. When that happens, the engine can safely delete the event from the session without side effects and release any resources used by that event.

There are basically 2 ways for the engine to calculate the matching window for a given event:

- explicitly, using the expiration policy
- implicitly, analyzing the temporal constraints on events

9.8.1. Explicit expiration offset

The first way of allowing the engine to calculate the window of interest for a given event type is by explicitly setting it. To do that, just use the declare statement and define an expiration for the fact type:

Example 9.10. explicitly defining an expiration offset of 30 minutes for StockTick events

```
declare StockTick    @expires( 30m )end
StockTick    @expires( 30m
```

The above example declares an expiration offset of 30 minutes for StockTick events. After that time, assuming no rule still needs the event, the engine will expire and remove the event from the session automatically.

Important

An explicit expiration policy for a given event type overrides any inferred expiration offset for that same type.

9.8.2. Inferred expiration offset

Another way for the engine to calculate the expiration offset for a given event is implicitly, by analyzing the temporal constraints in the rules. For instance, given the following rule:

Example 9.11. example rule with temporal constraints

```
rule "correlate orders"when    $bo : BuyOrderEvent( $id : id )     $ae : AckEvent( id == $id,
 this after[0,10s] $bo )then    // do somethingend

orders"when    $bo : BuyOrderEvent( $id : id
 )     $ae : AckEvent( id == $id, this after[0,10s]
$bo
 )then    //
do
```

Analyzing the above rule, the engine automatically calculates that whenever a BuyOrderEvent matches, it needs to store it for up to 10 seconds to wait for matching AckEvent's. So, the implicit expiration offset for BuyOrderEvent will be 10 seconds. AckEvent, on the other hand, can only match existing BuyOrderEvent's, and so its expiration offset will be zero seconds.

The engine will make this analysis for the whole rulebase and find the offset for every event type.

Important

An explicit expiration policy for a given event type overrides any inferred expiration offset for that same type.

9.9. Temporal Reasoning

Temporal reasoning is another requirement of any CEP system. As discussed previously, one of the distinguishing characteristics of events is their strong temporal relationships.

Temporal reasoning is an extensive field of research, from its roots on Temporal Modal Logic to its more practical applications in business systems. There are hundreds of papers and thesis written and approaches are described for several applications. Drools once more takes a pragmatic and simple approach based on several sources, but specially worth noting the following papers:

[ALLEN81] Allen, J.F.. *An Interval-based Representation of Temporal Knowledge*. 1981.

[ALLEN83] Allen, J.F.. *Maintaining knowledge about temporal intervals*. 1983.

[BENNE00] Bennet, Brandon and Galton, Antony P.. *A Unifying Semantics for Time and Events*. 2005.

[YONEK05] Yoneki, Eiko and Bacon, Jean. *Unified Semantics for Event Correlation Over Time and Space in Hybrid Network Environments*. 2005.

Drools implements the Interval-based Time Event Semantics described by Allen, and represents Point-in-Time Events as Interval-based evens with duration 0 (zero).

Note

For all temporal operator intervals, the "*" (star) symbol is used to indicate *positive infinity* and the "-*" (minus star) is used to indicate *negative infinity*.

9.9.1. Temporal Operators

Drools implements all 13 operators defined by Allen and also their logical complement (negation). This section details each of the operators and their parameters.

9.9.1.1. After

The after evaluator correlates two events and matches when the temporal distance from the current event to the event being correlated belongs to the distance range declared for the operator.

Lets look at an example:

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

The previous pattern will match if and only if the temporal distance between the time when $eventB finished and the time when $eventA started is between (3 minutes and 30 seconds) and (4 minutes). In other words:

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The temporal distance interval for the after operator is optional:

- If two values are defined (like in the example below), the interval starts on the first value and finishes on the second.
- If only one value is defined, the interval starts on the value and finishes on the positive infinity.
- If no value is defined, it is assumed that the initial value is 1ms and the final value is the positive infinity.

Note

It is possible to define negative distances for this operator. Example:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
```

Note

If the first value is greater than the second value, the engine automatically reverses them, as there is no reason to have the first value greater than the second value. Example: the following two patterns are considered to have the same semantics:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB ) $eventA : EventA( this
 after[ -2m, -3m30s ] $eventB )
 ) $eventA : EventA( this after[ -2m, -3m30s ] $eventB
```

Note

The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:

```
EventA( this after $someDate )
```

9.9.1.2. Before

The before evaluator correlates two events and matches when the temporal distance from the event being correlated to the current correlated belongs to the distance range declared for the operator.

Lets look at an example:

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

The previous pattern will match if and only if the temporal distance between the time when $eventA finished and the time when $eventB started is between (3 minutes and 30 seconds) and (4 minutes). In other words:

```
 3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The temporal distance interval for the before operator is optional:

- If two values are defined (like in the example below), the interval starts on the first value and finishes on the second.
- If only one value is defined, then the interval starts on the value and finishes on the positive infinity.
- If no value is defined, it is assumed that the initial value is 1ms and the final value is the positive infinity.

Note

It is possible to define negative distances for this operator. Example:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

Note

If the first value is greater than the second value, the engine automatically reverses them, as there is no reason to have the first value greater than the second value. Example: the following two patterns are considered to have the same semantics:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB ) $eventA : EventA( this
 before[ -2m, -3m30s ] $eventB )
 ) $eventA : EventA( this before[ -2m, -3m30s ] $eventB
```

Note

The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:

```
EventA( this after $someDate )
```

9.9.1.3. Coincides

The coincides evaluator correlates two events and matches when both happen at the same time. Optionally, the evaluator accept thresholds for the distance between events' start and finish timestamps.

Lets look at an example:

```
$eventA : EventA( this coincides $eventB )
```

The previous pattern will match if and only if the start timestamps of both $eventA and $eventB are the same AND the end timestamp of both $eventA and $eventB also are the same.

Optionally, this operator accepts one or two parameters. These parameters are the thresholds for the distance between matching timestamps.

- If only one parameter is given, it is used for both start and end timestamps.
- If two parameters are given, then the first is used as a threshold for the start timestamp and the second one is used as a threshold for the end timestamp.

In other words:

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

Above pattern will match if and only if:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 15s &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 10s
```

Warning

It makes no sense to use negative interval values for the parameters and the engine will raise an error if that happens.

Note

The *after*, *before* and *coincides* operators can be used to define constraints between events, java.util.Date attributes, and long attributes (interpreted as timestamps since epoch) in any combination. Example:

```
EventA( this after $someDate )
```

9.9.1.4. During

The during evaluator correlates two events and matches when the current event happens during the occurrence of the event being correlated.

Lets look at an example:

```
$eventA : EventA( this during $eventB )
```

The previous pattern will match if and only if the $eventA starts after $eventB starts and finishes before $eventB finishes.

In other words:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp < $eventB.endTimestamp
```

The during operator accepts 1, 2 or 4 optional parameters as follow:

- If one value is defined, this will be the maximum distance between the start timestamp of both event and the maximum distance between the end timestamp of both events in order to operator match. Example:

```
$eventA : EventA( this during[ 5s ] $eventB )
```

  Will match if and only if:

```
0 < $eventA.startTimestamp - $eventB.startTimestamp <= 5s &&
0 < $eventB.endTimestamp - $eventA.endTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance between the timestamps of both events, while the second value will be the maximum distance between the timestamps of both events. Example:

```
$eventA : EventA( this during[ 5s, 10s ] $eventB )
```

  Will match if and only if:

```
5s <= $eventA.startTimestamp - $eventB.startTimestamp <= 10s &&
```

```
5s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

- If four values are defined, the first two values will be the minimum and maximum distances between the start timestamp of both events, while the last two values will be the minimum and maximum distances between the end timestamp of both events. Example:

```
$eventA : EventA( this during[ 2s, 6s, 4s, 10s ] $eventB )
```

Will match if and only if:

```
2s <= $eventA.startTimestamp - $eventB.startTimestamp <= 6s &&
4s <= $eventB.endTimestamp - $eventA.endTimestamp <= 10s
```

9.9.1.5. Finishes

The finishes evaluator correlates two events and matches when the current event's start timestamp happens after the correlated event's start timestamp, but both end timestamps occur at the same time.

Lets look at an example:

```
$eventA : EventA( this finishes $eventB )
```

The previous pattern will match if and only if the $eventA starts after $eventB starts and finishes at the same time $eventB finishes.

In other words:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
$eventA.endTimestamp == $eventB.endTimestamp
```

The finishes evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of both events in order for the operator to match. Example:

```
$eventA : EventA( this finishes[ 5s ] $eventB )
```

Will match if and only if:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

9.9.1.6. Finished By

The finishedby evaluator correlates two events and matches when the current event start timestamp happens before the correlated event start timestamp, but both end timestamps occur at the same time. This is the symmetrical opposite of finishes evaluator.

Lets look at an example:

```
$eventA : EventA( this finishedby $eventB )
```

The previous pattern will match if and only if the $eventA starts before $eventB starts and finishes at the same time $eventB finishes.

In other words:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
$eventA.endTimestamp == $eventB.endTimestamp
```

The finishedby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of both events in order for the operator to match. Example:

```
$eventA : EventA( this finishedby[ 5s ] $eventB )
```

Will match if and only if:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

9.9.1.7. Includes

The includes evaluator correlates two events and matches when the event being correlated happens during the current event. It is the symmetrical opposite of during evaluator.

Lets look at an example:

```
$eventA : EventA( this includes $eventB )
```

The previous pattern will match if and only if the $eventB starts after $eventA starts and finishes before $eventA finishes.

In other words:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp < $eventA.endTimestamp
```

The includes operator accepts 1, 2 or 4 optional parameters as follow:

- If one value is defined, this will be the maximum distance between the start timestamp of both event and the maximum distance between the end timestamp of both events in order to operator match. Example:

```
$eventA : EventA( this includes[ 5s ] $eventB )
```

  Will match if and only if:

```
0 < $eventB.startTimestamp - $eventA.startTimestamp <= 5s &&
0 < $eventA.endTimestamp - $eventB.endTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance between the timestamps of both events, while the second value will be the maximum distance between the timestamps of both events. Example:

```
$eventA : EventA( this includes[ 5s, 10s ] $eventB )
```

  Will match if and only if:

```
5s <= $eventB.startTimestamp - $eventA.startTimestamp <= 10s &&
```

```
5s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

- If four values are defined, the first two values will be the minimum and maximum distances between the start timestamp of both events, while the last two values will be the minimum and maximum distances between the end timestamp of both events. Example:

```
$eventA : EventA( this includes[ 2s, 6s, 4s, 10s ] $eventB )
```

Will match if and only if:

```
2s <= $eventB.startTimestamp - $eventA.startTimestamp <= 6s &&
4s <= $eventA.endTimestamp - $eventB.endTimestamp <= 10s
```

9.9.1.8. Meets

The meets evaluator correlates two events and matches when the current event's end timestamp happens at the same time as the correlated event's start timestamp.

Lets look at an example:

```
$eventA : EventA( this meets $eventB )
```

The previous pattern will match if and only if the $eventA finishes at the same time $eventB starts.

In other words:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp ) == 0
```

The meets evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of current event and the start timestamp of the correlated event in order for the operator to match. Example:

```
$eventA : EventA( this meets[ 5s ] $eventB )
```

Will match if and only if:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

9.9.1.9. Met By

The metby evaluator correlates two events and matches when the current event's start timestamp happens at the same time as the correlated event's end timestamp.

Lets look at an example:

```
$eventA : EventA( this metby $eventB )
```

The previous pattern will match if and only if the $eventA starts at the same time $eventB finishes.

In other words:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp ) == 0
```

The metby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the end timestamp of the correlated event and the start timestamp of the current event in order for the operator to match. Example:

```
$eventA : EventA( this metby[ 5s ] $eventB )
```

Will match if and only if:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

9.9.1.10. Overlaps

The overlaps evaluator correlates two events and matches when the current event starts before the correlated event starts and finishes after the correlated event starts, but before the correlated event finishes. In other words, both events have an overlapping period.

Lets look at an example:

```
$eventA : EventA( this overlaps $eventB )
```

The previous pattern will match if and only if:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp < $eventB.endTimestamp
```

The overlaps operator accepts 1 or 2 optional parameters as follow:

- If one parameter is defined, this will be the maximum distance between the start timestamp of the correlated event and the end timestamp of the current event. Example:

```
$eventA : EventA( this overlaps[ 5s ] $eventB )
```

  Will match if and only if:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp < $eventB.endTimestamp
 &&
0 <= $eventA.endTimestamp - $eventB.startTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance and the second value will be the maximum distance between the start timestamp of the correlated event and the end timestamp of the current event. Example:

```
$eventA : EventA( this overlaps[ 5s, 10s ] $eventB )
```

  Will match if and only if:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp < $eventB.endTimestamp
 &&
5s <= $eventA.endTimestamp - $eventB.startTimestamp <= 10s
```

9.9.1.11. Overlapped By

The overlappedby evaluator correlates two events and matches when the correlated event starts before the current event starts and finishes after the current event starts, but before the current event finishes. In other words, both events have an overlapping period.

Lets look at an example:

```
$eventA : EventA( this overlappedby $eventB )
```

The previous pattern will match if and only if:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp < $eventA.endTimestamp
```

The overlappedby operator accepts 1 or 2 optional parameters as follow:

- If one parameter is defined, this will be the maximum distance between the start timestamp of the current event and the end timestamp of the correlated event. Example:

```
$eventA : EventA( this overlappedby[ 5s ] $eventB )
```

  Will match if and only if:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp < $eventA.endTimestamp
 &&
0 <= $eventB.endTimestamp - $eventA.startTimestamp <= 5s
```

- If two values are defined, the first value will be the minimum distance and the second value will be the maximum distance between the start timestamp of the current event and the end timestamp of the correlated event. Example:

```
$eventA : EventA( this overlappedby[ 5s, 10s ] $eventB )
```

  Will match if and only if:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp < $eventA.endTimestamp
 &&
5s <= $eventB.endTimestamp - $eventA.startTimestamp <= 10s
```

9.9.1.12. Starts

The starts evaluator correlates two events and matches when the current event's end timestamp happens before the correlated event's end timestamp, but both start timestamps occur at the same time.

Lets look at an example:

```
$eventA : EventA( this starts $eventB )
```

The previous pattern will match if and only if the $eventA finishes before $eventB finishes and starts at the same time $eventB starts.

In other words:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
$eventA.endTimestamp < $eventB.endTimestamp
```

The starts evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the start timestamp of both events in order for the operator to match. Example:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

Will match if and only if:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
$eventA.endTimestamp < $eventB.endTimestamp
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

9.9.1.13. Started By

The startedby evaluator correlates two events and matches when the correlating event's end timestamp happens before the current event's end timestamp, but both start timestamps occur at the same time. Lets look at an example:

```
$eventA : EventA( this startedby $eventB )
```

The previous pattern will match if and only if the $eventB finishes before $eventA finishes and starts at the same time $eventB starts.

In other words:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
$eventA.endTimestamp > $eventB.endTimestamp
```

The startedby evaluator accepts one optional parameter. If it is defined, it determines the maximum distance between the start timestamp of both events in order for the operator to match. Example:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

Will match if and only if:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
$eventA.endTimestamp > $eventB.endTimestamp
```

Warning

It makes no sense to use a negative interval value for the parameter and the engine will raise an exception if that happens.

Chapter 10. Experimental Features

10.1. Browsing graphs of objects with OOPath

When the field of a fact is a collection it is possible to bind and reason over all the items in that collection on by one using the `from` keyword. Nevertheless, when it is required to browse a graph of object the extensive use of the `from` conditional element may result in a verbose and cubersome syntax like in the following example:

Example 10.1. Browsing a graph of objects with from

```
rule "Find all grades for Big Data exam" when       $student: Student( $plan: plan )
 $exam: Exam( course == "Big Data" ) from $plan.exams       $grade: Grade() from $exam.grades
      then /* RHS */ end
 when      $student: Student( $plan: plan
 )      $exam: Exam( course == "Big Data" ) from
 $plan.exams      $grade: Grade() from
 $exam.grades      then /* RHS */
```

In this example it has been assumed to use a domain model consisting of a `Student` who has a `Plan` of study: a `Plan` can have zero or more `Exam`s and an `Exam` zero or more `Grade`s. Note that only the root object of the graph (the `Student` in this case) needs to be in the working memory in order to make this works.

By borrowing ideas from XPath, this syntax can be made more succinct, as XPath has a compact notation for navigating through related elements while handling collections and filtering constraints. This XPath-inspired notation has been called `OOPath` since it is explictly intended to browse graph of objects. Using this notation the former example can be rewritten as it follows:

Example 10.2. Browsing a graph of objects with OOPath

```
rule "Find all grades for Big Data exam" when      Student( $grade: /plan/exams{course == "Big
 Data"}/grades )      then /* RHS */ end
 when      Student( $grade: /plan/exams{course == "Big Data"}/grades
 )      then /* RHS */
```

Formally, the core grammar of an `OOPath` expression can be defined in EBNF notation in this way.

```
OOPExpr = "/" OOPSegment { ( "/" | "." ) OOPSegment } ;     OOPSegment = [ID ( ":" | ":=" )]
 ID ["[" Number "]"] ["{" Constraints "}"];
 } ;    OOPSegment = [ID ( ":" | ":=" )] ID ["[" Number "]"] ["{"
```

In practice an `OOPath` expression has the following features.

- It has to start with `/`.
- It can dereference a single property of an object with the `.` operator
- It can dereference a multiple property of an object using the `/` operator. If a collection is returned, it will iterate over the values in the collection
- While traversing referenced objects it can filter away those not satisfying one or more constraints, written as predicate expressions between curly brackets like in:

```
Student( $grade: /plan/exams{course == "Big Data"}/grades )
```

- Items can also be accessed by their index by putting it between square brackets like in:

```
Student( $grade: /plan/exams[0]/grades )
```

To adhere to Java convention OOPath indexes are 0-based, compared to XPath 1-based

10.1.1. Reactive OOPath

At the moment Drools is not able to react to updates involving a deeply nested traversed during the evaluation of an `OOPath` expression. To make these objects reactive to changes at the moment it is necessary to make them extend the class `org.drools.core.phreak.ReactiveObject`. It is planned to overcome this limitation by implementing a mechanism that automatically instruments the classes belonging to a specific domain model.

Having extendend that class, the domain objects can notify the engine when one of its field has been updated by invoking the inherited method `notifyModification` as in the following example:

Example 10.3. Notifying the engine that an exam has been moved to a different course

```
public void setCourse(String course) {      this.course = course;      notifyModification(this);
        }
{        this.course =
course;
notifyModification(this);
```

In this way if an exam is moved to a different course, the rule is re-triggered and the list of grades matching the rule recomputed.

Part IV. Drools Integration

Integration Documentation

Chapter 11. Drools Commands

11.1. API

XML marshalling/unmarshalling of the Drools Commands requires the use of special classes, which are going to be described in the following sections.

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD

11.1.1. XStream

To use the XStream commands marshaller you need to use the DroolsHelperProvider to obtain an XStream instance. We need to use this because it has the commands converters registered.

- Marshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().fromXML(xml)

11.1.2. JSON

JSON API to marshalling/unmarshalling is similar to XStream API:

- Marshalling

  BatchExecutionHelper.newJSonMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelper.newJSonMarshaller().fromXML(xml)

11.1.3. JAXB

There are two options for using JAXB, you can define your model in an XSD file or you can have a POJO model. In both cases you have to declare your model inside JAXBContext, and in order

to do that you need to use Drools Helper classes. Once you have the JAXBContext you need to create the Unmarshaller/Marshaller as needed.

11.1.3.1. Using an XSD file to define the model

With your model defined in a XSD file you need to have a KnowledgeBase that has your XSD model added as a resource.

To do this, the XSD file must be added as a XSD ResourceType into the KnowledgeBuilder. Finally you can create the JAXBContext using the KnowledgeBase created with the KnowledgeBuilder

```
Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage(Language.XMLSCHEMA);
JaxbConfiguration jaxbConfiguration = KnowledgeBuilderFactory.newJaxbConfiguration( xjcOpts, "xsd" );
kbuilder.add(ResourceFactory.newClassPathResource("person.xsd", getClass()), ResourceType.XSD, jaxbConfiguration);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();

List<String> classesName = new ArrayList<String>();
classesName.add("org.drools.compiler.test.Person");

JAXBContext jaxbContext = KnowledgeBuilderHelper.newJAXBContext(classesName.toArray(new String[classesName.size()]
```

11.1.3.2. Using a POJO model

In this case you need to use DroolsJaxbHelperProviderImpl to create the JAXBContext. This class has two parameters:

1. classNames: A List with the canonical name of the classes that you want to use in the marshalling/unmarshalling process.
2. properties: JAXB custom properties

```
List<String> classNames = new ArrayList<String>();
classNames.add("org.drools.compiler.test.Person");
JAXBContext jaxbContext = DroolsJaxbHelperProviderImpl.createDroolsJaxbContext(classNames, null);
Marshaller marshaller = jaxbContext.createMarshaller();
```

11.2. Commands supported

Currently, the following commands are supported:

- BatchExecutionCommand
- InsertObjectCommand

- RetractCommand
- ModifyCommand
- GetObjectCommand
- InsertElementsCommand
- FireAllRulesCommand
- StartProcessCommand
- SignalEventCommand
- CompleteWorkItemCommand
- AbortWorkItemCommand
- QueryCommand
- SetGlobalCommand
- GetGlobalCommand
- GetObjectsCommand

Note

In the next snippets code we are going to use a POJO org.drools.compiler.test.Person that has two fields

- name: String
- age: Integer

Note

In the next examples, to marshall the commands we have used the next snippet codes:

- XStream

```
String xml = BatchExecutionHelper.newXStreamMarshaller().toXML(command);
```

- JSON

```
String xml = BatchExecutionHelper.newJSonMarshaller().toXML(command);
```

- JAXB

```
Marshaller marshaller = jaxbContext.createMarshaller();
StringWriter xml = new StringWriter();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(command, xml);
```

11.2.1. BatchExecutionCommand

- Description: The command that contains a list of commands, which will be sent and executed.
- Attributes

Table 11.1. BatchExecutionCommand attributes

Name	Description	required
lookup	Sets the knowledge session id on which the commands are going to be executed	true
commands	List of commands to be executed	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
InsertObjectCommand insertObjectCommand = new InsertObjectCommand(new Person("john", 25));
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
command.getCommands().add(insertObjectCommand);
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert>
    <org.drools.compiler.test.Person>
```

```
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
  <fire-all-rules/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":
[{"insert":{"object":{"org.drools.compiler.test.Person":{"name":"john","age":25}}}},
{"fire-all-rules":""}]}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert>
        <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
    <fire-all-rules max="-1"/>
</batch-execution>
```

11.2.2. InsertObjectCommand

- Description: Insert an object in the knowledge session.
- Attributes

Table 11.2. InsertObjectCommand attributes

Name	Description	required
object	The object to be inserted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the	false

Name	Description	required
	execution results. Default value: true	
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();

Command insertObjectCommand = CommandFactory.newInsert(new Person("john", 25), "john", false, null);
cmds.add( insertObjectCommand );

BatchExecutionCommand command = CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert out-identifier="john" entry-point="my stream" return-object="false">
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert":{"entry-point":"my  stream",

{"org.drools.compiler.test.Person":{"name":"john","age":25}}}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert out-identifier="john" entry-point="my stream" >
        <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
</batch-execution>
```

11.2.3. RetractCommand

- Description: Retract an object from the knowledge session.
- Attributes

Table 11.3. RetractCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true

- Command creation: we have two options, with the same output result:

  1. Create the Fact Handle from a string

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand();
retractCommand.setFactHandleFromString("123:234:345:456:567");
command.getCommands().add(retractCommand);
```

  2. Set the Fact Handle that you received when the object was inserted

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand(factHandle);
command.getCommands().add(retractCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"retract":{"fact-
handle":"0:234:345:456:567"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

11.2.4. ModifyCommand

- Description: Allows you to modify a previously inserted object in the knowledge session.
- Attributes

Table 11.4. ModifyCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true
setters	List of setters object's modifications	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
ModifyCommand modifyCommand = new ModifyCommand();
modifyCommand.setFactHandleFromString("123:234:345:456:567");
List<Setter> setters = new ArrayList<Setter>();
setters.add(new SetterImpl("age", "30"));
modifyCommand.setSetters(setters);
command.getCommands().add(modifyCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <modify fact-handle="0:234:345:456:567">
    <set accessor="age" value="30"/>
  </modify>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"modify":{"fact-
handle":"0:234:345:456:567","setters":{"accessor":"age","value":30}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <modify fact-handle="0:234:345:456:567">
        <set value="30" accessor="age"/>
    </modify>
</batch-execution>
```

11.2.5. GetObjectCommand

- Description: Used to get an object from a knowledge session
- Attributes

Table 11.5. GetObjectCommand attributes

Name	Description	required
factHandle	The FactHandle associated to the object to be retracted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectCommand getObjectCommand = new GetObjectCommand();
getObjectCommand.setFactHandleFromString("123:234:345:456:567");
getObjectCommand.setOutIdentifier("john");
command.getCommands().add(getObjectCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <get-object fact-handle="0:234:345:456:567" out-identifier="john"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-object":{"fact-
handle":"0:234:345:456:567","out-identifier":"john"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-object out-identifier="john" fact-handle="0:234:345:456:567"/>
</batch-execution>
```

11.2.6. InsertElementsCommand

- Description: Used to insert a list of objects.
- Attributes

Table 11.6. InsertElementsCommand attributes

Name	Description	required
objects	The list of objects to be inserted on the knowledge session	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the execution results. Default value: true	false
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();

List<Object> objects = new ArrayList<Object>();
objects.add(new Person("john", 25));
objects.add(new Person("sarah", 35));

Command insertElementsCommand = CommandFactory.newInsertElements( objects );
cmds.add( insertElementsCommand );

BatchExecutionCommand command = CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert-elements>
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
    <org.drools.compiler.test.Person>
      <name>sarah</name>
      <age>35</age>
    </org.drools.compiler.test.Person>
  </insert-elements>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":
{"insert-elements":{"objects":[{"containedOb
ject":{"@class":"org.drools.compiler.test.Person","name":"john","age":25}},
{"containedObject":{"@class":"Person","name":"sarah","age":35}}]}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert-elements return-objects="true">
        <list>
          <element xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>25</age>
                <name>john</name>
            </element>
          <element xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>35</age>
```

```
                <name>sarah</name>
            </element>
        <list>
    </insert-elements>
</batch-execution>
```

11.2.7. FireAllRulesCommand

- Description: Allow execution of the rules activations created.
- Attributes

Table 11.7. FireAllRulesCommand attributes

Name	Description	required
max	The max number of rules activations to be executed. default is -1 and will not put any restriction on execution	false
outIdentifier	Add the number of rules activations fired on the execution results	false
agendaFilter	Allow the rules execution using an Agenda Filter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
fireAllRulesCommand.setMax(10);
fireAllRulesCommand.setOutIdentifier("firedActivations");
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <fire-all-rules max="10" out-identifier="firedActivations"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"fire-all-rules":{"max":10,"out-
identifier":"firedActivations"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
   <fire-all-rules out-identifier="firedActivations" max="10"/>
</batch-execution>
```

11.2.8. StartProcessCommand

- Description: Allows you to start a process using the ID. Also you can pass parameters and initial data to be inserted.
- Attributes

Table 11.8. StartProcessCommand attributes

Name	Description	required
processId	The ID of the process to be started	true
parameters	A Map<String, Object> to pass parameters in the process startup	false
data	A list of objects to be inserted in the knowledge session before the process startup	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
StartProcessCommand startProcessCommand = new StartProcessCommand();
startProcessCommand.setProcessId("org.drools.task.processOne");
command.getCommands().add(startProcessCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <start-process processId="org.drools.task.processOne"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"start-process":{"process-
id":"org.drools.task.processOne"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <start-process processId="org.drools.task.processOne">
        <parameter/>
    </start-process>
</batch-execution>
```

11.2.9. SignalEventCommand

- Description: Send a signal event.
- Attributes

Table 11.9. SignalEventCommand attributes

Name	Description	required
event-type		true
processInstanceId		false
event		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SignalEventCommand signalEventCommand = new SignalEventCommand();
signalEventCommand.setProcessInstanceId(1001);
signalEventCommand.setEventType("start");
signalEventCommand.setEvent(new Person("john", 25));
command.getCommands().add(signalEventCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <signal-event process-instance-id="1001" event-type="start">
    <org.drools.pipeline.camel.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.pipeline.camel.Person>
  </signal-event>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"signal-event":{"process-
instance-id":1001,"@event-type":"start","event-type":"start","object":
{"org.drools.pipeline.camel.Person":{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <signal-event event-type="start" process-instance-id="1001">
        <event xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </event>
    </signal-event>
</batch-execution>
```

11.2.10. CompleteWorkItemCommand

- Description: Allows you to complete a WorkItem.
- Attributes

Table 11.10. CompleteWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true
results		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
CompleteWorkItemCommand completeWorkItemCommand = new CompleteWorkItemCommand();
completeWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(completeWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <complete-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"complete-work-item":{"id":1001}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <complete-work-item id="1001"/>
</batch-execution>
```

11.2.11. AbortWorkItemCommand

- Description: Allows you abort an WorkItem. The same as session.getWorkItemManager().abortWorkItem(workItemId)
- Attributes

Table 11.11. AbortWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
AbortWorkItemCommand abortWorkItemCommand = new AbortWorkItemCommand();
abortWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(abortWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <abort-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"abort-work-item":{"id":1001}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <abort-work-item id="1001"/>
</batch-execution>
```

11.2.12. QueryCommand

- Description: Executes a query defined in knowledge base.
- Attributes

Table 11.12. QueryCommand attributes

Name	Description	required
name	The query name	true
outIdentifier	The identifier of the query results. The query results are	false

Name	Description	required
	going to be added in the execution results with this identifier	
arguments	A list of objects to be passed as a query parameter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
QueryCommand queryCommand = new QueryCommand();
queryCommand.setName("persons");
queryCommand.setOutIdentifier("persons");
command.getCommands().add(queryCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <query out-identifier="persons" name="persons"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"query":{"out-
identifier":"persons","name":"persons"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <query name="persons" out-identifier="persons"/>
</batch-execution>
```

11.2.13. SetGlobalCommand

- Description: Allows you to set a global.
- Attributes

Table 11.13. SetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
object	The object to be set into the global	false
out	A boolean to add, or not, the set global result into the execution results	false
outIdentifier	The identifier of the global execution result	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SetGlobalCommand setGlobalCommand = new SetGlobalCommand();
setGlobalCommand.setIdentifier("helper");
setGlobalCommand.setObject(new Person("kyle", 30));
setGlobalCommand.setOut(true);
setGlobalCommand.setOutIdentifier("output");
command.getCommands().add(setGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <set-global identifier="helper" out-identifier="output">
    <org.drools.compiler.test.Person>
      <name>kyle</name>
      <age>30</age>
    </org.drools.compiler.test.Person>
  </set-global>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"set-
global":{"identifier":"helper","out-identifier":"output","object":
{"org.drools.compiler.test.Person":{"name":"kyle","age":30}}}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <set-global out="true" out-identifier="output" identifier="helper">
        <object xsi:type="person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>30</age>
            <name>kyle</name>
        </object>
    </set-global>
</batch-execution>
```

11.2.14. GetGlobalCommand

- Description: Allows you to get a global previously defined.
- Attributes

Table 11.14. GetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetGlobalCommand getGlobalCommand = new GetGlobalCommand();
getGlobalCommand.setIdentifier("helper");
getGlobalCommand.setOutIdentifier("helperOutput");
command.getCommands().add(getGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-global identifier="helper" out-identifier="helperOutput"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
global":{"identifier":"helper","out-identifier":"helperOutput"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-global out-identifier="helperOutput" identifier="helper"/>
</batch-execution>
```

11.2.15. GetObjectsCommand

- Description: Returns all the objects from the current session as a Collection.
- Attributes

Table 11.15. GetObjectsCommand attributes

Name	Description	required
objectFilter	An ObjectFilter to filter the objects returned from the current session	false
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectsCommand getObjectsCommand = new GetObjectsCommand();
getObjectsCommand.setOutIdentifier("objects");
command.getCommands().add(getObjectsCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-objects out-identifier="objects"/>
```

```
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-objects":{"out-
identifier":"objects"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-objects out-identifier="objects"/>
</batch-execution>
```

Chapter 12. CDI

12.1. Introduction

CDI [http://www.cdi-spec.org], Contexts and Dependency Injection, is Java specification that provides declarative controls and strucutres to an application. KIE can use it to automatically instantiate and bind things, without the need to use the programmatic API.

12.2. Annotations

@KContainer, @KBase and @KSession all support an optional 'name' attribute. CDI typically does "getOrCreate" when it injects, all injections receive the same instance for the same set of annotations. the 'name' annotation forces a unique instance for each name, although all instance for that name will be identity equals.

12.2.1. @KReleaseId

Used to bind an instance to a specific version of a KieModule. If kie-ci is on the classpath this will resolve dependencies automatically, downloading from remote repositories.

12.2.2. @KContainer

@KContainer is optional as it can be detected and added by the use of @Inject and variable type inferrence.

```
@Inject
private KieContainer kContainer;
```

Figure 12.1. Injects Classpath KieContainer

```
@Inject
@KReleaseId(groupId = "jar1", artifactId = "art1", version = "1.1")
private KieContainer kContainer;
```

Figure 12.2. Injects KieContainer for Dynamic KieModule

```
@Inject
@KContainer(name = "kc1")
@KReleaseId(groupId = "jar1", artifactId = "art1", version = "1.1")
private KieContainer kContainer;
```

Figure 12.3. Injects named KieContainer for Dynamic KieModule

12.2.3. @KBase

@KBase is optional as it can be detected and added by the use of @Inject and variable type inference.

The default argument, if given, maps to the value attribute and specifies the name of the KieBase from the kmodule.xml file.

```
@Inject
private KieBase kbase;
```

Figure 12.4. Injects the Default KieBase from the Classpath KieContainer

```
@Inject
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase;
```

Figure 12.5. Injects the Default KieBase from a Dynamic KieModule

```
@Inject
@KBase("kbase1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase1v10;

@Inject
@KBase("kbase1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.1")
private KieBase kbase1v10;
```

Figure 12.6. Side by side version loading for 'jar1.KBase1' KieBase

```
@Inject
@KSession(value="kbase1", name="kb1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase1kb1;

@Inject
@KSession(value="kbase1", name="kb2")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase1kb2;
```

Figure 12.7. Use 'name' attribute to force new Instance for 'jar1.KBase1' KieBase

12.2.4. @KSession for KieSession

@KSession is optional as it can be detected and added by the use of @Inject and variable type inference.

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml file

```
@Inject
private KieSession ksession;
```

Figure 12.8. Injects the Default KieSession from the Classpath KieContainer

```
@Inject
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession;
```

Figure 12.9. Injects the Default KieSession from a Dynamic KieModule

```
@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksessionv10;

@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.1")
private KieSession ksessionv11;
```

Figure 12.10. Side by side version loading for 'jar1.KBase1' KieBase

```
@Inject
@KSession(value="ksession1", name="ks1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession1ks1

@Inject
@KSession(value="ksession1", name="ks2")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession1ks2
```

Figure 12.11. Use 'name' attribute to force new Instance for 'jar1.KBase1' KieSession

12.2.5. @KSession for StatelessKieSession

@KSession is optional as it can be detected and added by the use of @Inject and variable type inference.

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml file.

```
@Inject
private StatelessKieSession ksession;
```

Figure 12.12. Injects the Default StatelessKieSession from the Classpath KieContainer

```
@Inject
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private StatelessKieSession ksession;
```

Figure 12.13. Injects the Default StatelessKieSession from a Dynamic KieModule

```
@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private StatelessKieSession ksessionv10;

@Inject
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private StatelessKieSession ksessionv11;
```

Figure 12.14. Side by side version loading for 'jar1.KBase1' KieBase

```
@Inject
@KSession(value="ksession1", name="ks1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private StatelessKieSession ksession1ks1

@Inject
@KSession(value="ksession1", name="ks2")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
```

```
private StatelessKieSession ksession1ks2
```

Figure 12.15. Use 'name' attribute to force new Instance for 'jar1.KBase1' StatelessKieSession

12.3. API Example Comparison

CDI can inject instances into fields, or even pass them as arguments. In this example field injection is used.

```
@Inject
@KSession("ksession1")
KieSession kSession;

public void go(PrintStream out) {
    kSession.setGlobal("out", out);
    kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
    kSession.fireAllRules();
}
```

Figure 12.16. CDI example for a named KieSession

This is less code and more declarative than the API approach.

```
public void go(PrintStream out) {
    KieServices ks = KieServices.Factory.get();
    KieContainer kContainer = ks.getKieClasspathContainer();

    KieSession kSession = kContainer.newKieSession("ksession1");
    kSession.setGlobal("out", out);
    kSession.insert(new Message("Dave", "Hello, HAL. Do you read me, HAL?"));
    kSession.fireAllRules();
}
```

Figure 12.17. API equivalent example for a named KieSession

Chapter 13. Integration with Spring

13.1. Important Changes for Drools 6.0

Drools Spring integration has undergone a complete makeover inline with the changes for Drools 6.0. The following are some of the major changes

- The recommended prefix for the Drools Spring has changed from 'drools:' to 'kie:'
- New Top Level Tags in 6.0
  - kie:kmodule
  - kie:import (from version 6.2)
  - kie:releaseId (from version 6.2)
- The following tags are no longer valid as top level tags.
  - kie:kbase - A child of the *kie:kmodule* tag.
  - kie:ksession - A child of the *kie:kbase* tag.
- Removed Tags from previous versions Drools 5.x
  - drools:resources
  - drools:resource
  - drools:grid
  - drools:grid-node

13.2. Integration with Drools Expert

In this section we will explain the *kie* namespace.

13.2.1. KieModule

The <*kie:kmodule*> defines a collection of KieBase and associated KieSession's. The *kmodule* tag has one MANDATORY parameter 'id'.

Table 13.1. Sample

Attribute	Description	Required
id	Bean's *id* is the name to be referenced from other beans.	Yes

Attribute	Description	Required
	Standard Spring ID semantics apply.	

A *kmodule* tag can contain only the following tags as children.

- kie:kbase

Refer to the documentation of kmodule.xml in the Drools Expert documentation for detailed explanation of the need for kmodule.

13.2.2. KieBase

13.2.2.1. <kie:kbase>'s parameters as attributes:

Table 13.2. Sample

Attribute	Description	Required
name	Name of the KieBase	Yes
packages	Comma separated list of resource packages to be included in this kbase	No
includes	kbase names to be included. All resources from the corresponding kbase are included in this kbase.	No
default	Boolean (TRUE/FALSE). Default kbase, if not provided, it is assumed to be FALSE	No
scope	prototype | singleton. If not provided assumed to be singleton (default)	No
eventProcessingMode	Event Processing Mode. Valid options are STREAM, CLOUD	No
equalsBehavior	Valid options are IDENTITY, EQUALITY	No
declarativeAgenda	Valid options are enabled, disabled, true, false	No

13.2.2.2. A *kbase* tag can contain only the following tags as children.

- kie:ksession

13.2.2.3. <kie:kbase>'s definition example

A kmodule can contain multiple (1..n) kbase elements.

Example 13.1. kbase definition example

```
<kie:kmodule id="sample_module">
   <kie:kbase name="kbase1" packages="org.drools.spring.sample">
     ...
   </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.2.4. Spring Bean Scope (for KieBase and KieSession)

When defining a KieBase or a KieSession, you have the option of declaring a scope for that bean. For example, To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be 'prototype'. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be 'singleton'.

13.2.3. IMPORTANT NOTE

For proper initialization of the kmodule objects (kbase/ksession), it is mandatory for a bean of type `org.kie.spring.KModuleBeanFactoryPostProcessor` or `org.kie.spring.annotations.KModuleAnnotationPostProcessor` be defined.

Example 13.2. Regular kie-spring post processorbean definition

```
<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

Example 13.3. kie-spring post processorbean definition when annotations are used

```
<bean id="kiePostProcessor"
          class="org.kie.spring.annotations.KModuleAnnotationPostProcessor"/>
```

Note

Without the org.kie.spring.KModuleBeanFactoryPostProcessor or org.kie.spring.annotations.KModuleAnnotationPostProcessor bean definition, the kie-spring integration will fail to work.

13.2.4. KieSessions

<kie:ksession> element defines KieSessions. The same tag is used to define both Stateful (*org.kie.api.runtime.KieSession*) and Stateless (*org.kie.api.runtime.StatelessKieSession*) sessions.

13.2.4.1. <kie:ksession>'s parameters as attributes:

Table 13.3. Sample

Attribute	Description	Required
name	ksession's name.	Yes
type	is the session *stateful* or *stateless?*. If this attribute is empty or missing, the session is assumed to be of type Stateful.	No
default	Is this the default session?	no
scope	prototype | singleton. If not provided assumed to be singleton (default)	no
clockType	REALTIME or PSEUDO	no
listeners-ref	Specifies the reference to the event listeners group (see 'Defining a Group of listeners' section below).	no

Example 13.4. ksession definition example

```
<kie:kmodule id="sample-kmodule">
  <kie:kbase name="drl_kiesample3" packages="drl_kiesample3">
    <kie:ksession name="ksession1" type="stateless"/>
    <kie:ksession name="ksession2"/>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
```

```
            class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.4.2. Spring Bean Scope (for KieBase and KieSession)

When defining a KieBase or a KieSession, you have the option of declaring a scope for that bean. For example, To force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be 'prototype'. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be 'singleton'.

13.2.5. Kie:ReleaseId

13.2.5.1. <kie:releaseId>'s parameters as attributes:

Table 13.4. Sample

Attribute	Description	Required
id	Bean's *id* is the name to be referenced from other beans. Standard Spring ID semantics apply.	Yes
groupId	groupId from Maven GAV	Yes
artifactId	artifactId from Maven GAV	Yes
version	version from Maven GAV	Yes

Example 13.5. releaseId definition example

```
<kie:releaseId id="beanId" groupId="org.kie.spring"
            artifactId="named-artifactId" version="1.0.0-SNAPSHOT"/>
```

13.2.6. Kie:Import

Starting with version 6.2, kie-spring allows for importing of kie objects from kjars found on the classpath. Two modes of importing the kie objects are currently supported.

Table 13.5.

Attribute	Description	Required
releaseId	Reference to a Bean ID. Standard Spring ID semantics apply.	No

Attribute	Description	Required
enableScanner	Enable Scanner. This attribute is used only if 'releaseId' is specified.	No
scannerInterval	Scanning Interval in milli seconds. This attribute is used only if 'releaseId' is specified.	No

13.2.6.1. Global Import

The *import* tag will force the automatic scan of all the jars on the classpath, initialize the Kie Objects (Kbase/KSessions) and import these objects into the spring context.

```
<kie:import />
```

Figure 13.1. Global Import

13.2.6.2. Specific Import - ReleaseId

Using the *releaseId-ref* attribute on the import tag will initialize the specific Kie Objects (Kbase/KSessions) and import these objects into the spring context.

```
<kie:import releaseId-ref="namedKieSession"/>
<kie:releaseId id="namedKieSession" groupId="org.drools"
            artifactId="named-kiesession" version="6.3.1-SNAPSHOT"/>
```

Figure 13.2. Import Kie Objects using a releaseId

Kie Scanning feature can be enabled for KieBase's imported with a specific releaseId. This feature is currently not available for global imports.

```
<kie:import releaseId-ref="namedKieSession"
            enableScanner="true" scannerInterval="1000"/>

<kie:releaseId id="namedKieSession" groupId="org.drools"
            artifactId="named-kiesession" version="6.3.1-SNAPSHOT"/>
```

Figure 13.3. Import Kie Objects using a releaseId - Enable Scanner

If the scanner is defined and enabled, an implicit *KieScanner* object is created and inserted into the spring context. It can be retrived from the spring context.

```
// the implicit name would be releaseId#scanner
KieScanner releaseIdScanner = context.getBean("namedKieSession#scanner", KieScanner.class);
releaseIdScanner.scanNow();
```

Figure 13.4. Retriving the KieScanner from Spring Context

Note

kie-ci must be available on the classpath for the releaseId importing feature to work.

13.2.7. Annotations

@KContainer, @KBase and @KSession all support an optional 'name' attribute. Spring typically does "get" when it injects, all injections receive the same instance for the same set of annotations. the 'name' annotation forces a unique instance for each name, although all instance for that name will be identity equals.

13.2.7.1. @KReleaseId

Used to bind an instance to a specific version of a KieModule. If kie-ci is on the classpath this will resolve dependencies automatically, downloading from remote repositories.

13.2.7.2. @KContainer

```
@KContainer
private KieContainer kContainer;
```

Figure 13.5. Injects Classpath KieContainer

```
@KContainer
@KReleaseId(groupId = "jar1", artifactId = "art1", version = "1.1")
private KieContainer kContainer;
```

Figure 13.6. Injects KieContainer for Dynamic KieModule

```
@KContainer(name = "kc1")
@KReleaseId(groupId = "jar1", artifactId = "art1", version = "1.1")
private KieContainer kContainer;
```

Figure 13.7. Injects named KieContainer for Dynamic KieModule

13.2.7.3. @KBase

The default argument, if given, maps to the value attribute and specifies the name of the KieBase from the spring xml file.

```
@KBase
private KieBase kbase;
```

Figure 13.8. Injects the Default KieBase from the Classpath KieContainer

```
@KBase
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase;
```

Figure 13.9. Injects the Default KieBase from a Dynamic KieModule

```
@KBase("kbase1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieBase kbase1v10;

@KBase("kbase1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.1")
private KieBase kbase1v11;
```

Figure 13.10. Side by side version loading for 'jar1.KBase1' KieBase

```
@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession11kb2;

@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.1")
private KieSession ksession11kb2;
```

Figure 13.11. Side by side version loading for 'jar1.ksession1' KieSession

13.2.7.4. @KSession for KieSession

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml or spring xml file

```
@KSession
```

```
private KieSession ksession;
```

Figure 13.12. Injects the Default KieSession from the Classpath KieContainer

```
@KSession
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession;
```

Figure 13.13. Injects the Default KieSession from a Dynamic KieModule

```
@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksessionv10;

@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.1")
private KieSession ksessionv11;
```

Figure 13.14. Side by side version loading for 'jar1.KBase1' KieBase

```
@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession1ks1

@KSession("ksession1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private KieSession ksession1ks2
```

Figure 13.15. Use 'name' attribute to force new Instance for 'jar1.KBase1' KieSession

13.2.7.5. @KSession for StatelessKieSession

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml or spring xml file.

```
@KSession
private StatelessKieSession ksession;
```

Figure 13.16. Injects the Default StatelessKieSession from the Classpath KieContainer

```
@KSession
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private StatelessKieSession ksession;
```

Figure 13.17. Injects the Default StatelessKieSession from a Dynamic KieModule

```
@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.0")
private StatelessKieSession ksessionv10;

@KSession("ksession1")
@KReleaseId( groupId = "jar1", rtifactId = "art1", version = "1.1")
private StatelessKieSession ksessionv11;
```

Figure 13.18. Side by side version loading for 'jar1.KBase1' KieBase

```
@KSession(value="ksession1", name="ks1")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private StatelessKieSession ksession1ks1

@KSession(value="ksession1", name="ks2")
@KReleaseId( groupId = "jar1", artifactId = "art1", version = "1.0")
private StatelessKieSession ksession1ks2
```

Figure 13.19.

13.2.7.6. IMPORTANT NOTE

When annotations are used, For proper initialization of the kmodule objects (kbase/ksession), it is mandatory for either a bean of type `org.kie.spring.annotations.KModuleAnnotationPostProcessor` be defined or spring component-scan be enabled. One of the code snippets as shown below is required.

Example 13.6. kie-spring annotations post processor bean definition

```
<bean id="kiePostProcessor"
           class="org.kie.spring.annotations.KModuleAnnotationPostProcessor"/>
```

Example 13.7. kie-spring annotations - Component Scanning

```
<context:component-scan base-package="org.kie.spring.annotations"/>
```

Note

The post processor is different when annotations are used.

13.2.8. Event Listeners

Drools supports adding 3 types of listeners to KieSessions - *AgendaListener*, *WorkingMemoryListener*, *ProcessEventListener*

The kie-spring module allows you to configure these listeners to KieSessions using XML tags. These tags have identical names as the actual listener interfaces i.e., <kie:agendaEventListener....>, <kie:ruleRuntimeEventListener....> and <kie:processEventListener....>.

kie-spring provides features to define the listeners as standalone (individual) listeners and also to define them as a group.

13.2.8.1. Defining Stand alone Listeners:

13.2.8.1.1. Attributes:

Table 13.6. Sample

Attribute	Required	Description
ref	No	A reference to another declared bean.

Example 13.8. Listener configuration example - using a bean:ref.

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>

<kie:kmodule id="listeners_kmodule">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksession2">
      <kie:agendaEventListener ref="mock-agenda-listener"/>
      <kie:processEventListener ref="mock-process-listener"/>
      <kie:ruleRuntimeEventListener ref="mock-rr-listener"/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.8.1.2. Nested Elements:

- bean
  - class = String
  - name = String (optional)

Example 13.9. Listener configuration example - using nested bean.

```
<kie:kmodule id="listeners_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
   <kie:ksession name="ksession1">
      <kie:agendaEventListener>
      <bean class="mocks.MockAgendaEventListener"/>
      </kie:agendaEventListener>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.8.1.3. Empty Tag : Declaration with no 'ref' and without a nested bean

When a listener is defined without a reference to a implementing bean and does not contain a nested bean, *<drools:ruleRuntimeEventListener/>* the underlying implementation adds the Debug version of the listener defined in the API.

The debug listeners print the corresponding Event toString message to *System.err.*

Example 13.10. Listener configuration example - defaulting to the debug versions provided by the Knowledge-API .

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>

<kie:kmodule id="listeners_module">
 <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksession2">
      <kie:agendaEventListener />
      <kie:processEventListener />
      <kie:ruleRuntimeEventListener />
    </kie:ksession>
 </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
```

```
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.8.1.4. Mix and Match of different declaration styles

The drools-spring module allows you to mix and match the different declarative styles within the same KieSession. The below sample provides more clarity.

Example 13.11. Listener configuration example - mix and match of 'ref'/ nested-bean/empty styles.

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>

<kie:kmodule id="listeners_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksession1">
      <kie:agendaEventListener>
          <bean class="org.kie.spring.mocks.MockAgendaEventListener"/>
      </kie:agendaEventListener>
    </kie:ksession>
    <kie:ksession name="ksession2">
      <kie:agendaEventListener ref="mock-agenda-listener"/>
      <kie:processEventListener ref="mock-process-listener"/>
      <kie:ruleRuntimeEventListener ref="mock-rr-listener"/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.8.1.5. Defining multiple listeners of the same type

It is also valid to define multiple beans of the same event listener types for a KieSession.

Example 13.12. Listener configuration example - multiple listeners of the same type.

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>

<kie:kmodule id="listeners_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksession1">
      <kie:agendaEventListener ref="mock-agenda-listener"/>
      <kie:agendaEventListener>
          <bean class="org.kie.spring.mocks.MockAgendaEventListener"/>
      </kie:agendaEventListener>
    </kie:ksession>
  </kie:kbase>
```

```
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.8.2. Defining a Group of listeners:

drools-spring allows for grouping of listeners. This is particularly useful when you define a set of listeners and want to attach them to multiple sessions. The grouping feature is also very useful, when we define a set of listeners for 'testing' and then want to switch them for 'production' use.

13.2.8.2.1. Attributes:

Table 13.7. Sample

Attribute	Required	Description
ID	yes	Unique identifier

13.2.8.2.2. Nested Elements:

- kie:agendaEventListener...
- kie:ruleRuntimeEventListener...
- kie:processEventListener...

Note

The above mentioned child elements can be declared in any order. Only one declaration of each type is allowed in a group.

13.2.8.2.3. Example:

Example 13.13. Group of listeners - example

```
<bean id="mock-agenda-listener" class="mocks.MockAgendaEventListener"/>
<bean id="mock-rr-listener" class="mocks.MockRuleRuntimeEventListener"/>
<bean id="mock-process-listener" class="mocks.MockProcessEventListener"/>

<kie:kmodule id="listeners_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="statelessWithGroupedListeners" type="stateless"
             listeners-ref="debugListeners"/>
  </kie:kbase>
</kie:kmodule>
```

```
  <kie:eventListeners id="debugListeners">
  <kie:agendaEventListener ref="mock-agenda-listener"/>
  <kie:processEventListener ref="mock-process-listener"/>
  <kie:ruleRuntimeEventListener ref="mock-rr-listener"/>
</kie:eventListeners>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.9. Loggers

Drools supports adding 2 types of loggers to KieSessions - *ConsoleLogger*, *FileLogger.*

The kie-spring module allows you to configure these loggers to KieSessions using XML tags. These tags have identical names as the actual logger interfaces i.e., <kie:consoleLogger....> and <kie:fileLogger....>.

13.2.9.1. Defining a console logger:

A console logger can be attached to a KieSession by using the *<kie:consoleLogger/>* tag. This tag has no attributes and must be present directly under a <kie:ksession....> element.

Example 13.14. Defining a console logger - example

```
<kie:kmodule id="loggers_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ConsoleLogger-statefulSession" type="stateful">
      <kie:consoleLogger/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.9.2. Defining a file logger:

A file logger can be attached to a KieSession by using the *<kie:fileLogger/>* tag. This tag has the following attributes and must be present directly under a <kie:ksession....> element.

Table 13.8. Sample

Attribute	Required	Description
ID	yes	Unique identifier
file	yes	Path to the actual file on the disk

Attribute	Required	Description
threaded	no	Defaults to false. Valid values are 'true'or 'false'
interval	no	Integer. Specifies the interval for flushing the contents from memory to the disk.

Example 13.15. Defining a file logger - example

```
<kie:kmodule id="loggers_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ConsoleLogger-statefulSession" type="stateful">
      <kie:fileLogger id="fl_logger" file="#{ systemProperties['java.io.tmpdir'] }/log1"/>
      <kie:fileLogger id="tfl_logger" file="#{ systemProperties['java.io.tmpdir'] }/log2"
                      threaded="true" interval="5"/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
         class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.9.2.1. Closing a FileLogger

To prevent leaks, it is advised to close the *<kie:fileLogger ...>* programmatically.

```
LoggerAdaptor adaptor = (LoggerAdaptor) context.getBean("fl_logger");
adaptor.close();
```

13.2.10. Defining Batch Commands

A *<kie:batch>* element can be used to define a set of batch commands for a given ksession.This tag has no attributes and must be present directly under a <kie:ksession....> element. The commands supported are

- insert-object
  - ref = String (optional)
  - Anonymous bean
- set-global
  - identifier = String (required)
  - reg = String (optional)

    - Anonymous bean
- fire-all-rules
    - max : n
- fire-until-halt
- start-process
    - parameter
        - identifier = String (required)
        - ref = String (optional)
        - Anonymous bean
- signal-event
    - ref = String (optional)
    - event-type = String (required)
    - process-instance-id =n (optional)

Figure 13.20. Initialization Batch Commands

Example 13.16. Batch commands - example

```
<kie:kmodule id="batch_commands_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ksessionForCommands" type="stateful">
      <kie:batch>
        <kie:insert-object ref="person2"/>
        <kie:set-global identifier="persons" ref="personsList"/>
        <kie:fire-all-rules max="10"/>
      </kie:batch>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
         class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.11. Persistence

- jpa-persistence
    - transaction-manager

- ref = String

- entity-manager-factory
  - ref = String

Figure 13.21. Persistence Configuration Options

Example 13.17. ksession JPA configuration example

```
<kie:kstore id="kstore" /> <!-- provides KnowledgeStoreService implementation -->

<bean id="myEmf"
       class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
   <property name="dataSource" ref="ds" />
   <property name="persistenceUnitName"
       value="org.drools.persistence.jpa.local" />
</bean>

<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory" ref="myEmf" />
</bean>

<kie:kmodule id="persistence_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="jpaSingleSessionCommandService">
      <kie:configuration>
         <kie:jpa-persistence>
           <kie:transaction-manager ref="txManager"/>
           <kie:entity-manager-factory ref="myEmf"/>
         </kie:jpa-persistence>
      </kie:configuration>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
         class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.12. Leveraging Other Spring Features

This section provides details on leveraging other standard spring features when integrating with Drools Expert.

13.2.12.1. Using Spring Expressions (Spel)

```
<kie:kmodule id="batch_commands_module">
  <kie:kbase name="drl_kiesample" packages="#{packageRepository.packages}">
    <kie:ksession name="ksessionForCommands" type="stateful"/>
  </kie:kbase>
```

```
</kie:kmodule>

<bean id="kiePostProcessor"
      class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>

<bean id="packageRepository" class="sample.package.class.PackageRepo">
  <property name="packages" value="drl_kiesample3">
</bean>
```

```
<kie:kmodule id="loggers_module">
  <kie:kbase name="drl_kiesample" packages="drl_kiesample">
    <kie:ksession name="ConsoleLogger-statefulSession" type="stateful">
      <kie:fileLogger id="fl" file="#{ systemProperties['java.io.tmpdir'] }/log1"/>
      <kie:fileLogger id="tfl" file="#{ systemProperties['java.io.tmpdir'] }/log2"
            threaded="true" interval="5"/>
    </kie:ksession>
  </kie:kbase>
</kie:kmodule>

<bean id="kiePostProcessor"
            class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>
```

13.2.12.2. Using Spring Profiles

Spring 3.1 introduces a new profile attribute to the beans element of the spring-beans schema. This attribute acts as a switch when enabling and disabling profiles in different environments. One potential use of this attribute can be to have the same kbase defined with debug loggers in 'dev' environment and without loggers in 'prod' environment.

The below code snippet illustrates the concept of 'profiles'.

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:kie="http://drools.org/schema/kie-spring"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://drools.org/schema/kie-spring http://drools.org/schema/kie-spring.xsd">
  <beans profile="development">
    <kie:kmodule id="test-kmodule">
      <kie:kbase name="drl_kiesample" packages="drl_kiesample">
        <kie:ksession name="ksession1" type="stateless">
            <kie:consoleLogger />
        </kie:ksession>
      </kie:kbase>
    </kie:kmodule>
    ...
  </beans>

  <beans profile="production">
    <kie:kmodule id="test-kmodule">
      <kie:kbase name="drl_kiesample" packages="drl_kiesample">
```

```
        <kie:ksession name="ksession1" type="stateless"/>
      </kie:kbase>
    </kie:kmodule>
    ...
  </beans>
</beans>
```

As shown above, the Spring XML contains the definition of the profiles. While loading the *ApplicationContext* you have to tell Spring which profile you're loading.

There are several ways of selecting your profile and the most useful is by using the "spring.profiles.active" system property.

```
System.setProperty("spring.profiles.active",   "development");ApplicationContext   ctx   =   new
 ClassPathXmlApplicationContext("beans.xml");
velopment");ApplicationContext ctx =
```

Obviously, it is not a good practice to hard code things as shown above and the recommended practice is to keep the system properties definitions independent of the application.

```
-Dspring.profiles.active="development"
```

The profiles can also be loaded and enabled programmtically

```
...
GenericXmlApplicationContext ctx = new GenericXmlApplicationContext("beans.xml");
ConfigurableEnvironment env = ctx.getEnvironment();
env.setActiveProfiles("development");
ctx.refresh();
...
```

13.3. Integration with jBPM Human Task

This chapter describes the infrastructure used when configuring a human task server with Spring as well as a little bit about the infrastructure used when doing this.

13.3.1. How to configure Spring with jBPM Human task

The jBPM human task server can be configured to use Spring persistence. Example 13.18, "Configuring Human Task with Spring" is an example of this which uses local transactions and Spring's thread-safe EntityManager proxy.

The following diagram shows the dependency graph used in Example 13.18, "Configuring Human Task with Spring".

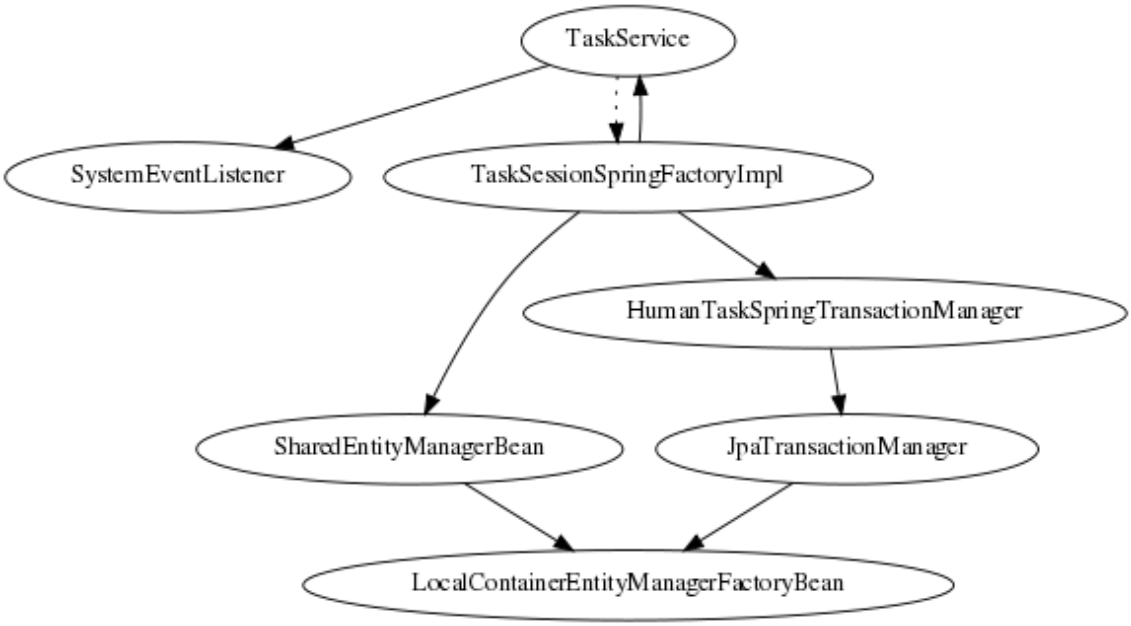


Figure 13.22. Spring Human Task integration injection dependencies

A `TaskService` instance is dependent on two other bean types: a drools `SystemEventListener` bean as well as a `TaskSessionSpringFactoryImpl` bean. The `TaskSessionSpringFactoryImpl` bean is howerver *not* injected into the `TaskService` bean because this would cause a circular dependency. To solve this problem, when the `TaskService` bean is injected into the `TaskSessionSpringFactoryImpl` bean, the setter method used secretly injects the `TaskSessionSpringFactoryImpl` instance back into the `TaskService` bean and initializes the `TaskService` bean as well.

The `TaskSessionSpringFactoryImpl` bean is responsible for creating all the internal instances in human task that deal with transactions and persistence context management. Besides a `TaskService` instance, this bean also requires a transaction manager and a persistence context to be injected. Specifically, it requires an instance of a `HumanTaskSpringTransactionManager` bean (as a transaction manager) and an instance of a `SharedEntityManagerBean` bean (as a persistence context instance).

We also use some of the standard Spring beans in order to configure persistence: there's a bean to hold the `EntityManagerFactory` instance as well as the `SharedEntityManagerBean` instance. The `SharedEntityManagerBean` provides a shared, thread-safe proxy for the actual `EntityManager`.

The `HumanTaskSpringTransactionManager` bean serves as a wrapper around the Spring transaction manager, in this case the `JpaTransactionManager`. An instance of a `JpaTransactionManager` bean is also instantiated because of this.

Example 13.18. Configuring Human Task with Spring

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jbpm="http://drools.org/schema/drools-spring"
                      xsi:schemaLocation="http://www.springframework.org/schema/beans   http://
www.springframework.org/schema/beans/spring-beans-3.0.xsd
                    http://drools.org/schema/drools-spring   org/drools/container/spring/drools-
spring-1.2.0.xsd">

  <!-- persistence & transactions-->
  <bean id="htEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="org.jbpm.task" />
  </bean>

  <bean id="htEm" class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
    <property name="entityManagerFactory" ref="htEmf"/>
  </bean>

  <bean id="jpaTxMgr" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="htEmf" />
    <!-- this must be true if using the SharedEntityManagerBean, and false otherwise -->
    <property name="nestedTransactionAllowed" value="true"/>
  </bean>

  <bean id="htTxMgr" class="org.drools.container.spring.beans.persistence.HumanTaskSpringTransactionManager">
    <constructor-arg ref="jpaTxMgr" />
  </bean>

  <!-- human-task beans -->

     <bean   id="systemEventListener"   class="org.drools.SystemEventListenerFactory"   factory-
method="getSystemEventListener" />

  <bean id="taskService" class="org.jbpm.task.service.TaskService" >
    <property name="systemEventListener" ref="systemEventListener" />
  </bean>

  <bean id="springTaskSessionFactory" class="org.jbpm.task.service.persistence.TaskSessionSpringFactoryImpl"
        init-method="initialize" depends-on="taskService" >
    <!-- if using the SharedEntityManagerBean, make sure to enable nested transactions -->
    <property name="entityManager" ref="htEm" />
    <property name="transactionManager" ref="htTxMgr" />
    <property name="useJTA" value="false" />
    <property name="taskService" ref="taskService" />
  </bean>

</beans>
```

When using the `SharedEntityManagerBean` instance, it's important to configure the Spring transaction manager to use nested transactions. This is because the `SharedEntityManagerBean` is a *transactional* persistence context and will close the persistence context after every operation. However, the human task server needs to be able to access (persisted) entities after operations. Nested transactions allow us to still have access to entities that otherwise would have been detached and are no longer accessible, especially when using an ORM framework that uses lazy-initialization of entities.

Also, while the `TaskSessionSpringFactoryImpl` bean takes an “useJTA” parameter, at the moment, JTA transactions with Spring have not yet been fully tested.

Chapter 14. Android Integration

14.1. Integration with Drools Expert

Drools Android integration comes in two flavors, with or without the *drools-compiler* dependency. Without the drools-compiler dependency the knowledge bases are pre-serialized at buildtime using the kie-maven-plugin. They can be then deserialized using the API, or directly injected using Roboguice. When using the drools-compiler dependency there are two options: (1) standard KieContainer API or (2) CDI-like injection with Roboguice.

14.1.1. Pre-serialized Rules

It is possible to use Drools without the drools-compiler dependency, which results in a smaller apk, by pre-serializing the compiled knowledge bases during build-time and then de-serializing them at runtime.

14.1.1.1. Maven Configuration

The KieBase must be serialized during build time using *kie-maven-plugin*.

Example 14.1. Pre-serialized KieBase Maven pom.xml

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-bom</artifactId>
      <version>6.3.1-SNAPSHOT</version>
      <type>pom</type>
      <scope>import</scope>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-android</artifactId>
    <exclusions>
     <exclusion>
       <groupId>org.slf4j</groupId>
       <artifactId>slf4j-api</artifactId>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-core</artifactId>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
```

```
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>6.3.1-SNAPSHOT</version>
      <configuration>
        <kiebases>
        <kiebase>HelloKB</kiebase>
        </kiebases>
        <resDirectory>${basedir}/src/main/res/raw</resDirectory>
      </configuration>
        <executions>
          <execution>
            <id>touch</id>
            <goals>
              <goal>touch</goal>
            </goals>
            <phase>initialize</phase>
          </execution>
          <execution>
            <id>compile-kbase</id>
              <goals>
                <goal>build</goal>
              </goals>
              <phase>compile</phase>
          </execution>
          <execution>
            <id>serialize</id>
            <goals>
              <goal>serialize</goal>
            </goals>
            <phase>compile</phase>
          </execution>
        </executions>
     </plugin>
  </plugins>
</build>
```

14.1.1.2. Loading the KieBase

The KieBase must be de-serialized before creating the sessions.

```
private class LoadKieBaseTask extends AsyncTask<InputStream, Void, KieBase> {
   @Override
   protected KieBase doInBackground(InputStream... params) {
      try {
         logger.debug("Loading knowledge base");
         final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
         kbase.addKnowledgePackages((List<KnowledgePackage>) DroolsStreamUtils.streamIn(params[0]));
         return kbase;
      }catch(Exception e) {
         logger.error("Drools exception", e);
         return null;
      }
   }
}
```

Figure 14.1. Loading serialized KieBase

14.1.2. KieContainer API with drools-compiler dependency

With the drools-compiler dependency standard KieContainer API can be used. This comes at a cost of a larger apk. To avoid the 65K limit, multidex (or proguard) can be used.

14.1.2.1. Maven Configuration with drools-compiler and multidex

The kie-maven-plugin must be configured to build the kiebase. Multidex must be used to allow for the increased dependencies. There are also some settings for merging various Drools XML files within the apk.

Example 14.2. pom.xml with drools-compiler and multidex

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.drools</groupId>
      <artifactId>drools-bom</artifactId>
      <version>6.3.1-SNAPSHOT</version>
      <type>pom</type>
      <scope>import</scope>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-android</artifactId>
    <exclusions>
      <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <exclusions>
      <exclusion>
         <groupId>org.slf4j</groupId>
         <artifactId>slf4j-api</artifactId>
      </exclusion>
      <exclusion>
         <groupId>xmlpull</groupId>
         <artifactId>xmlpull</artifactId>
      </exclusion>
      <exclusion>
         <groupId>xpp3</groupId>
         <artifactId>xpp3_min</artifactId>
      </exclusion>
```

```
      <exclusion>
         <groupId>org.slf4j</groupId>
         <artifactId>slf4j-api</artifactId>
      </exclusion>
      <exclusion>
         <groupId>org.eclipse.jdt.core.compiler</groupId>
         <artifactId>ecj</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>6.3.1-SNAPSHOT</version>
      <executions>
        <execution>
          <id>compile-kbase</id>
          <goals>
            <goal>build</goal>
          </goals>
          <phase>compile</phase>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>com.simpligility.maven.plugins</groupId>
      <artifactId>android-maven-plugin</artifactId>
      <version>4.2.1</version>
      <extensions>true</extensions>
      <configuration>
        <sdk>
          <platform>21</platform>
        </sdk>
        <dex>
          <coreLibrary>true</coreLibrary>
          <jvmArguments><jvmArgument>-Xmx2048m</jvmArgument></jvmArguments>
          <multiDex>true</multiDex>
          <mainDexList>maindex.txt</mainDexList>
        </dex>
        <extractDuplicates>true</extractDuplicates>
        <apk>
          <metaInf>
            <includes>
              <include>services/**</include>
              <include>kmodule.*</include>
              <include>HelloKB/**</include>
              <include>drools**</include>
              <include>maven/${project.groupId}/${project.artifactId}/**</include>
            </includes>
          </metaInf>
        </apk>
      </configuration>
    </plugin>
  </plugins>
</build>
```

14.2. Integration with Roboguice

14.2.1. Pre-serialized Rules with Roboguice

With Roboguice pre-serialized knowledge bases can be injected using the @KBase annotation.

14.2.1.1. Annotations

@KBase supports an optional 'name' attribute. CDI typically does "getOrCreate" when it injects, all injections receive the same instance for the same set of annotations. the 'name' annotation forces a unique instance for each name, although all instances for that name will be identity equals.

14.2.1.1.1. @KBase

The default argument maps to the value attribute and specifies the name of the KieBase from the kmodule.xml file.

```
@KBase("kbase1")
private KieBase kbase;
```

Figure 14.2. Injects KieBase by name from pre-serialized resource

14.2.1.2. AndroidManifest.xml configuration

The Roboguice module needs to be specified in the manifest.

Example 14.3. Roboguice manifest with pre-serialized knowledge base

```
<application
   android:largeHeap="true"
   android:allowBackup="true"
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme">
      <meta-data
         android:name="roboguice.modules"
         android:value="org.drools.android.roboguice.DroolsModule"/>
      <activity
         android:label="@string/app_name"
         android:name="org.drools.examples.android.SplashActivity">
      <intent-filter>
         <action android:name="android.intent.action.MAIN"/>
         <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
   </activity>
</application>
```

14.2.2. KieContainer with drools-compiler dependency and Roboguice

With Roboguice and drools-compiler almost the full CDI syntax can be used to inject KieContainers, KieBases, and KieSessions.

14.2.2.1. Annotations

@KContainer, @KBase and @KSession all support an optional 'name' attribute. CDI typically does "getOrCreate" when it injects, all injections receive the same instance for the same set of annotations. the 'name' annotation forces a unique instance for each name, although all instance for that name will be identity equals.

14.2.2.1.1. @KContainer

```
@Inject
private KieContainer kContainer;
```

Figure 14.3. Injects Classpath KieContainer

14.2.2.1.2. @KBase

The default argument, if given, maps to the value attribute and specifies the name of the KieBase from the kmodule.xml file.

```
@Inject
private KieBase kbase;
```

Figure 14.4. Injects the Default KieBase from the Classpath KieContainer

```
@Inject
@KBase("kbase1")
private KieBase kbase;
```

Figure 14.5. Injects KieBase by name from the Classpath KieContainer

14.2.2.1.3. @KSession for KieSession

@KSession is optional as it can be detected and added by the use of @Inject and variable type inference.

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml file

```
@Inject
private KieSession ksession;
```

Figure 14.6. Injects the Default KieSession from the Classpath KieContainer

```
@Inject
@KSession("ksession1")
private KieSession ksession;
```

Figure 14.7. Injects StatelessKieSession by name from the Classpath KieContainer

14.2.2.1.4. @KSession for StatelessKieSession

@KSession is optional as it can be detected and added by the use of @Inject and variable type inference.

The default argument, if given, maps to the value attribute and specifies the name of the KieSession from the kmodule.xml file.

```
@Inject
private StatelessKieSession ksession;
```

Figure 14.8. Injects the Default StatelessKieSession from the Classpath KieContainer

```
@Inject
@KSession("ksession1")
private StatelessKieSession ksession;
```

Figure 14.9. Injects StatelessKieSession by name from the Classpath KieContainer

14.2.2.2. AndroidManifest.xml configuration

The Roboguice module needs to be specified in the manifest.

Example 14.4. Roboguice manifest configuration

```
<application
   android:largeHeap="true"
   android:allowBackup="true"
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name"
   android:theme="@style/AppTheme">
   <meta-data
      android:name="roboguice.modules"
      android:value="org.drools.android.roboguice.DroolsContainerModule"/>
   <activity
      android:label="@string/app_name"
      android:name="org.drools.examples.android.SplashActivity">
      <intent-filter>
         <action android:name="android.intent.action.MAIN"/>
         <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
   </activity>
</application>
```

Chapter 15. Apache Camel Integration

15.1. Camel

Camel provides a light weight bus framework for getting information into and out of Drools.

Drools introduces two elements to make easy integration.

- Drools Policy

  Augments any JAXB or XStream data loaders. For JAXB it adds drools related paths ot the contextpath, for XStream it adds custom converters and aliases for Drools classes. It also handles setting the ClassLoader to the targeted ksession.

- Drools Endpoint

  Executes the payload against the specified drools session

Drools can be configured like any normal camel component, but notice the policy that wraps the drools related segments. This will route all payloads to ksession1

Example 15.1. Drools EndPoint configured with the CXFRS producer

```
<bean id="kiePolicy" class="org.kie.camel.component.KiePolicy" />

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="cxfrs://bean://rsServer"/>
         <policy ref="kiePolicy">
            <unmarshal ref="xstream" />
            <to uri="kie:ksession1" />
            <marshal ref="xstream" />
       </policy>
    </route>
</camelContext>
```

It is possible to not specify the session in the drools endpoint uri, and instead "multiplex" based on an attribute or header. In this example the policy will check either the header field "DroolsLookup" for the named session to execute and if that isn't specified it'll check the "lookup" attribute on the incoming payload.

Example 15.2. Drools EndPoint configured with the CXFRS producer

```
<bean id="kiePolicy" class="org.kie.camel.component.KiePolicy" />
```

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="cxfrs://bean://rsServer"/>
         <policy ref="kiePolicy">
            <unmarshal ref="xstream" />
            <to uri="kie:dynamic" />
            <marshal ref="xstream" />
       </policy>
    </route>
</camelContext>
```

Example 15.3. Java Code to execute against Route from a Spring and Camel Context

```
public class MyTest extends CamelSpringTestSupport {             @Override       protected
 AbstractXmlApplicationContext createApplicationContext() {                        return new
ClassPathXmlApplicationContext("org/drools/camel/component/CxfRsSpring.xml");    }      public
void test1() throws Exception {        String cmd = "";        cmd += "<batch-execution lookup=
\"ksession1\">\n";         cmd += "  <insert out-identifier=\"salaboy\">\n";         cmd += "
    <org.drools.pipeline.camel.Person>\n";         cmd += "          <name>salaboy</name>\n";
      cmd += "       </org.drools.pipeline.camel.Person>\n";         cmd += "    </insert>\n";
       cmd += "    <fire-all-rules/>\n";         cmd += "</batch-execution>\n";
          Object object = this.context.createProducerTemplate().requestBody("direct://client",
cmd);        System.out.println( object );     }}
port {

  @Override    protected AbstractXmlApplicationContext createApplicationContext() {
        return new ClassPathXmlApplicationContext("org/drools/
camel/component/CxfRsSpring.xml");

}      public void test1() throws
Exception {        String cmd
= "";        cmd += "<batch-execution
lookup=\"ksession1\">\n";        cmd += "  <insert out-identifier=
\"salaboy\">\n";        cmd += "
 <org.drools.pipeline.camel.Person>\n";        cmd += "
<name>salaboy</name>\n";        cmd += "
</org.drools.pipeline.camel.Person>\n";        cmd += "
</insert>\n";        cmd += "   <fire-all-
rules/>\n";        cmd += "</
batch-execution>\n";
       Object object = this.context.createProducerTemplate().requestBody("direct://
client", cmd);        System.out.println( object
 );
  }}
```

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD

- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD

Chapter 16. Drools Camel Server

16.1. Introduction

The drools camel server (drools-camel-server) module is a war which you can deploy to execute KnowledgeBases remotely for any sort of client application. This is not limited to JVM application clients, but any technology that can use HTTP, through a REST interface. This version of the execution server supports stateless and stateful sessions in a native way.

16.2. Deployment

Drools Camel Server is a war file, which can be deployed in a application server (such as JBoss AS). As the service is stateless, it is possible to have have as many of these services deployed as you need to serve the client load. Deploy on JBoss AS 4.x / Tomcat 6.x works out-of-the-box, instead some external dependencies must be added and the configuration must be changed to be deployed in JBoss AS 5

16.3. Configuration

Inside the war file you will find a few XML configuration files.

- beans.xml
  - Skeleton XML that imports knowledge-services.xml and camel-server.xml
- camel-server.xml
  - Configures CXF endpoints with Camel Routes
  - Came Routes pipeline messages to various configured knowledge services
- knowledge-services.xml
  - Various Knowledge Bases and Sessions
- camel-client.xml
  - Sample camel client showing how to send and receive a message
  - Used by "out of the box" test.jsp

16.3.1. REST/Camel Services configuration

The next step is configure the services that are going to be exposed through drools-server. You can modify this configuration in camel-server.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cxf="http://camel.apache.org/schema/cxf"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xsi:schemaLocation="
     http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/
spring-beans-2.5.xsd
  http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/camel-cxf.xsd
  http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
        http://camel.apache.org/schema/spring     http://camel.apache.org/schema/spring/camel-
spring.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

  <!--
    !    If you are running on JBoss you will need to copy a camel-jboss.jar into the lib and
 set this ClassLoader configuration
    !  http://camel.apache.org/camel-jboss.html
    !   <bean id="jbossResolver" class="org.apache.camel.jboss.JBossPackageScanClassResolver"/>
    -->

  <!--
    !   Define the server end point.
    !   Copy and paste this element, changing id and the address, to expose services on different
 urls.
    !   Different Camel routes can handle different end point paths.
    -->
  <cxf:rsServer id="rsServer"
                address="/rest"
                serviceClass="org.kie.jax.rs.CommandExecutorImpl">
       <cxf:providers>
           <bean class="org.kie.jax.rs.CommandMessageBodyReader"/>
       </cxf:providers>
  </cxf:rsServer>

  <cxf:cxfEndpoint id="soapServer"
            address="/soap"
             serviceName="ns:CommandExecutor"
             endpointName="ns:CommandExecutorPort"
          wsdlURL="soap.wsdl"
          xmlns:ns="http://soap.jax.drools.org/" >
    <cxf:properties>
      <entry key="dataFormat" value="MESSAGE"/>
      <entry key="defaultOperationName" value="execute"/>
    </cxf:properties>
  </cxf:cxfEndpoint>

  <!-- Leave this, as it's needed to make Camel "drools" aware -->
  <bean id="kiePolicy" class="org.kie.camel.component.KiePolicy" />

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <!--
     ! Routes incoming messages from end point id="rsServer".
     ! Example route unmarshals the messages with xstream and executes against ksession1.
     ! Copy and paste this element, changing marshallers and the 'to' uri, to target different
 sessions, as needed.
     !-->
```

```
    <route>
       <from uri="cxfrs://bean://rsServer"/>
       <policy ref="kiePolicy">
         <unmarshal ref="xstream" />
         <to uri="kie:ksession1" />
         <marshal ref="xstream" />
       </policy>
    </route>

    <route>
      <from uri="cxf://bean://soapServer"/>
      <policy ref="kiePolicy">
        <unmarshal ref="xstream" />
        <to uri="kie:ksession1" />
        <marshal ref="xstream" />
      </policy>
    </route>

  </camelContext>

</beans>
```

16.3.1.1. RESTful service endpoint creation

In the next XML snippet code we are creating a RESTful (JAX-RS) endpoint bound to /kservice/rest address and using org.drools.jax.rs.CommandExecutorImpl as the service implementer. This class is only used to instantiate the service endpoint because all the internal implementation is managed by Camel, and you can see in the source file that the exposed execute service must be never called.

Also a JAX-RS Provider is provided to determine if the message transported can be processed in this service endpoint.

```
<cxf:rsServer id="rsServer"
              address="/rest"
              serviceClass="org.kie.jax.rs.CommandExecutorImpl">
     <cxf:providers>
         <bean class="org.kie.jax.rs.CommandMessageBodyReader"/>
     </cxf:providers>
</cxf:rsServer>
```

Ideally this configuration doesn't need to be modified, at least the Service Class and the JAX-RS Provider, but you can add more endpoints associated to different addresses to use them in other Camel Routes.

After all this initial configuration, you can start config your own Knowledge Services.

16.3.1.2. Camel Kie Policy & Context creation

KiePolicy is used to add Drools support in Camel, basically what it does is to add interceptors into the camel route to create Camel Processors on the fly and modify the internal navigation route.

If you want to have SOAP support you need to create your custom Drools Policy, but it's going to be added in the next release.

But you don't need to know more internal details, only instantiate this bean:

```
<bean id="kiePolicy" class="org.kie.camel.component.KiePolicy" />
```

The next is create the camel route that will have the responsibility to execute the commands sent through JAX-RS. Basically we create a route definition associated with the JAX-RS definition as the data input, the camel policy to be used and inside the "execution route" or ProcessorDefinitions. As you can see, we set XStream as the marshaller/unmarshaller and the drools execution route definition

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="cxfrs://bean://rsServer"/>
      <policy ref="kiePolicy">
        <unmarshal ref="xstream" />
        <to uri="kie:ksession1" />
        <marshal ref="xstream" />
      </policy>
   </route>
   <route>
     <from uri="cxf://bean://soapServer"/>
     <policy ref="kiePolicy">
       <unmarshal ref="xstream" />
       <to uri="kie:ksession1" />
       <marshal ref="xstream" />
     </policy>
   </route>
</camelContext>
```

The drools endpoint creation has the next arguments

```
<to uri="kie:{1}/{2}" />
```

1. Execution Node identifier that is registered in the CamelContext
2. Knowledge Session identifier that was registered in the Execution Node with identifier {1}

Both parameters are configured in knowledge-services.xml file.

16.3.1.3. Knowledge Services configuration

The next step is create the Knowledge Sessions that you are going to use.

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:kie="http://drools.org/schema/kie-spring"
                    xsi:schemaLocation="http://www.springframework.org/schema/beans   http://
www.springframework.org/schema/beans/spring-beans-3.0.xsd
                                http://drools.org/schema/kie-spring http://drools.org/schema/
kie-spring.xsd">

  <kie:kmodule id="drools-camel-server">
    <kie:kbase name="kbase1" packages="org.drools.server">
      <kie:ksession name="ksession1" type="stateless"/>
    </kie:kbase>
  </kie:kmodule>

  <bean id="kiePostProcessor"
          class="org.kie.spring.KModuleBeanFactoryPostProcessor"/>

</beans>
```

The execution-node is a context or registered kbases and ksessions, here kbase1 and ksession1 are planed in the node1 context. The kbase itself consists of two knowledge definitions, a DRL and an XSD. The Spring documentation contains a lot more information on configuring these knowledge services.

16.3.1.4. Test

With drools-server war unzipped you should be able to see a test.jsp and run it. This example just executes a simple "echo" type application. It sends a message to the rule server that pre-appends the word "echo" to the front and sends it back. By default the message is "Hello World", different messages can be passed using the url parameter msg - test.jsp?msg="My Custom Message".

Under the hood the jsp invokes the Test.java class, this then calls out to Camel which is where the meet happens. The camel-client.xml defines the client with just a few lines of XML:

```
<!-- Leave this, as it's needed to make Camel "drools" aware -->
<bean id="kiePolicy" class="org.kie.camel.component.KiePolicy" />

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
     <from uri="direct://kservice/rest"/>
     <policy ref="kiePolicy">
       <to uri="cxfrs://http://localhost:8080/drools-server/kservice/rest"/>
     </policy>
  </route>
  <route>
    <from uri="direct://kservice/soap"/>
    <policy ref="kiePolicy">
      <to uri="cxfrs://http://localhost:8080/drools-server/kservice/soap"/>
    </policy>
  </route>
</camelContext>
```

"direct://kservice" is just a named hook, allowing Java to grab a reference and push data into it. In this example the data is already in XML, so we don't need to add any `DataFormats` to do the marshalling. The KiePolicy adds some smarts to the route and you'll see it used on the server side too. If JAXB or XStream were used, it would inject custom paths and converters, it can also set the ClassLoader too on the server side, on the client side it automatically unwraps the Response object.

The rule itself can be found here: test.drl. Notice the type Message is declared part of the DRL and is thus not present on the Classpath.

```
declare Message   text : Stringend      rule "echo" dialect "mvel"when   $m : Message();then
   $m.text = "echo:" + $m.text;end
Message   text :

Stringend

  rule "echo" dialect

"mvel"when   $m :

Message();then   $m.text = "echo:" +
```

Chapter 17. JMX monitoring with RHQ/JON

17.1. Introduction

The Drools engine supports runtime monitoring through JMX standard MBeans. These MBeans expose configuration and metrics data, from live knowledge bases and sessions, to internal details like rule execution times. Any JMX compatible console can be used to access that data. This chapter details how to use RHQ/JON to do it, but similar steps can be used for any other console.

17.1.1. Enabling JMX monitoring in a Drools application

To enable JMX monitoring in a Drools application, it is necessary to enable remote monitoring in the JVM. There are several tutorials on how to do that in the internet, but we recommend that you check the documentation of your specific JVM. Using the Oracle/Sun JVM, it can be as simple as running the engine with a few command line system properties.

For instance, to enable remote monitoring on port 19988 with disabled authentication (should be only used for tests/demos, as in production authentication should be enabled), just run the application with the following command line parameters:

```
    -Dcom.sun.management.jmxremote.port=19988    -Dcom.sun.management.jmxremote.ssl=false    -
Dcom.sun.management.jmxremote.authenticate=false
```

The second step is to enable the Drools MBeans. As any Drools configuration, that can be done by setting a system property, or adding the property to a configuration file, or using the API.

To enable it in the command line, use:

```
-Dkie.mbeans=enabled
```

To enable id using the API, use:

```
KieBaseConfiguration conf = ...
conf.setOption( MBeansOption.ENABLED );
```

17.1.2. Installing and running the RHQ/JON plugin

The following sequence of steps can be used to configure JON to monitor a Drools application:

1. Download the JON server and agent.

2. Download Drools plugin included in the "Drools and jBPM tools" bundle (http://www.jboss.org/drools/downloads.html).
3. Install server, agent, and the plugin.
4. Check that the server is running, agent is running and plugin is installed.
5. Execute the drools application [see details in the previous section].
6. On the agent console, type "discovery" command for the agent to find the drools application, which it will find on port 19988.
7. On JON console, click on auto-discovery queue.
8. Select the JMX Server process that is showing there, running on port 19988.
9. Click import.
10. Click on Resources->servers.
11. Click on the JMX Server.
12. Under JMXServer on the left hand side, you have Drools Service.

Part V. Drools Workbench

The Drools workbench is built with the UberFire framework and uses the Guvnor plugin. Drools provides an additional rich set of plugins for rule authoring metaphors.

Chapter 18. Workbench

18.1. Installation

18.1.1. War installation

Use the `war` from the workbench distribution zip that corrsponds to your application server. The differences between these `war` files are mainly superficial. For example, some JARs might be excluded if the application server already supplies them.

- `eap6_4`: tailored for Red Hat JBoss Enterprise Application Platform 6.4
- `tomcat7`: tailored for Apache Tomcat 7

Note

Apache Tomcat requires additional configuration to correctly install the Workbench. Please consult the `README.md` in the `war` for the most up to date procedure.

- `was8`: tailored for IBM WebSphere Application Server 8
- `weblogic12`: tailored for Oracle WebLogic Server 12c

Note

Oracle WebLogic requires additional configuration to correctly install the Workbench. Please consult the `README.md` in the `war` for the most up to date procedure.

- `wildfly8`: tailored for Red Hat JBoss Wildfly 8

18.1.2. Workbench data

The workbench stores its data, by default in the directory `$WORKING_DIRECTORY/.niogit`, for example `wildfly-8.0.0.Final/bin/.gitnio`, but it can be overridden with the system property `-Dorg.uberfire.nio.git.dir`.

Note

In production, make sure to back up the workbench data directory.

18.1.3. System properties

Here's a list of all system properties:

- **`org.uberfire.nio.git.dir`**: Location of the directory `.niogit`. Default: working directory
- **`org.uberfire.nio.git.daemon.enabled`**: Enables/disables git daemon. Default: `true`
- **`org.uberfire.nio.git.daemon.host`**: If git daemon enabled, uses this property as local host identifier. Default: `localhost`
- **`org.uberfire.nio.git.daemon.port`**: If git daemon enabled, uses this property as port number. Default: `9418`
- **`org.uberfire.nio.git.ssh.enabled`**: Enables/disables ssh daemon. Default: `true`
- **`org.uberfire.nio.git.ssh.host`**: If ssh daemon enabled, uses this property as local host identifier. Default: `localhost`
- **`org.uberfire.nio.git.ssh.port`**: If ssh daemon enabled, uses this property as port number. Default: `8001`
- **`org.uberfire.nio.git.ssh.passphrase`**: Passphrase to access your Operating Systems public keystore when cloning `git` repositories with `scp` style URLs; e.g. `git@github.com:user/repository.git`.
- **`org.uberfire.nio.git.ssh.cert.dir`**: Location of the directory `.security` where local certtificates will be stored. Default: working directory
- **`org.uberfire.metadata.index.dir`**: Place where Lucene `.index` folder will be stored. Default: working directory
- **`org.uberfire.cluster.id`**: Name of the helix cluster, for example: `kie-cluster`
- **`org.uberfire.cluster.zk`**: Connection string to zookeeper. This is of the form `host1:port1,host2:port2,host3:port3`, for example: `localhost:2188`
- **`org.uberfire.cluster.local.id`**: Unique id of the helix cluster node, note that '`:`' is replaced with '_', for example: `node1_12345`
- **`org.uberfire.cluster.vfs.lock`**: Name of the resource defined on helix cluster, for example: `kie-vfs`
- **`org.uberfire.cluster.autostart`**: Delays VFS clustering until the application is fully initialized to avoid conflicts when all cluster members create local clones. Default: `false`
- **`org.uberfire.sys.repo.monitor.disabled`**: Disable configuration monitor (do not disable unless you know what you're doing). Default: `false`

- **`org.uberfire.secure.key`**: Secret password used by password encryption. Default: `org.uberfire.admin`
- **`org.uberfire.secure.alg`**: Crypto algorithm used by password encryption. Default: `PBEWithMD5AndDES`
- **`org.uberfire.domain`**: security-domain name used by uberfire. Default: `ApplicationRealm`
- **`org.guvnor.m2repo.dir`**: Place where Maven repository folder will be stored. Default: working-directory/repositories/kie
- **`org.kie.example.repositories`**: Folder from where demo repositories will be cloned. The demo repositories need to have been obtained and placed in this folder. Demo repositories can be obtained from the kie-wb-6.2.0-SNAPSHOT-example-repositories.zip artifact. This System Property takes precedence over org.kie.demo and org.kie.example. Default: Not used.
- **`org.kie.demo`**: Enables external clone of a demo application from GitHub. This System Property takes precedence over org.kie.example. Default: `true`
- **`org.kie.example`**: Enables example structure composed by Repository, Organization Unit and Project. Default: `false`

To change one of these system properties in a WildFly or JBoss EAP cluster:

1. Edit the file `$JBOSS_HOME/domain/configuration/host.xml`.
2. Locate the XML elements `server` that belong to the `main-server-group` and add a system property, for example:

```
<system-properties>
  <property name="org.uberfire.nio.git.dir" value="..." boot-time="false"/>
  ...
</system-properties>
```

18.2. Quick Start

These steps help you get started with minimum of effort.

They should not be a substitute for reading the documentation in full.

18.2.1. Add repository

Create a new repository to hold your project by selecting the Administration Perspective.

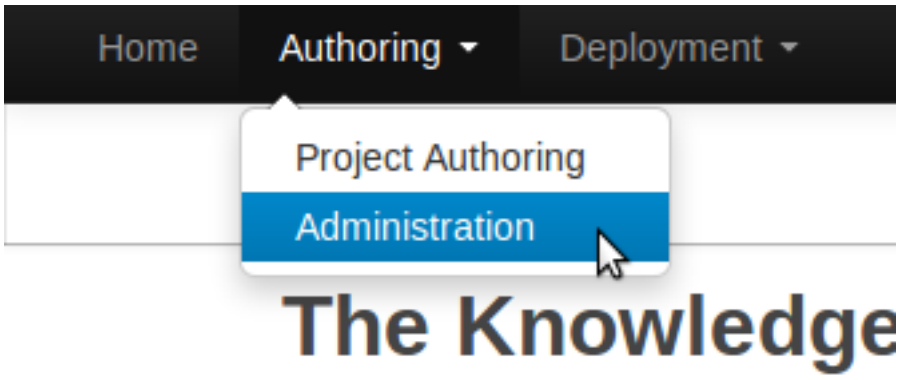


Figure 18.1. Selecting Administration perspective

Select the "New repository" option from the menu.

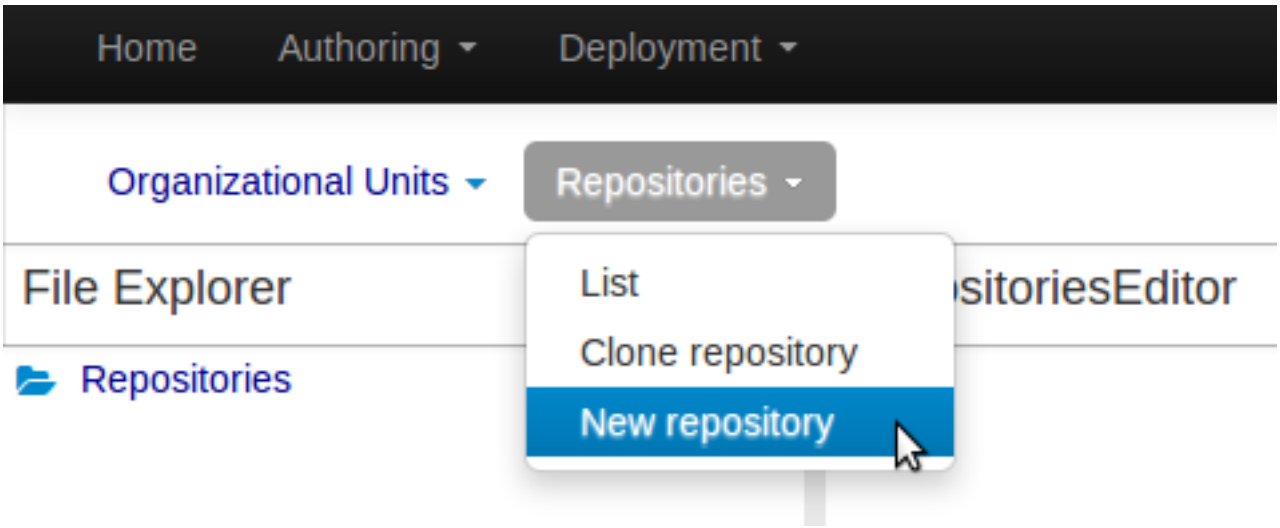


Figure 18.2. Creating new repository

Enter the required information.

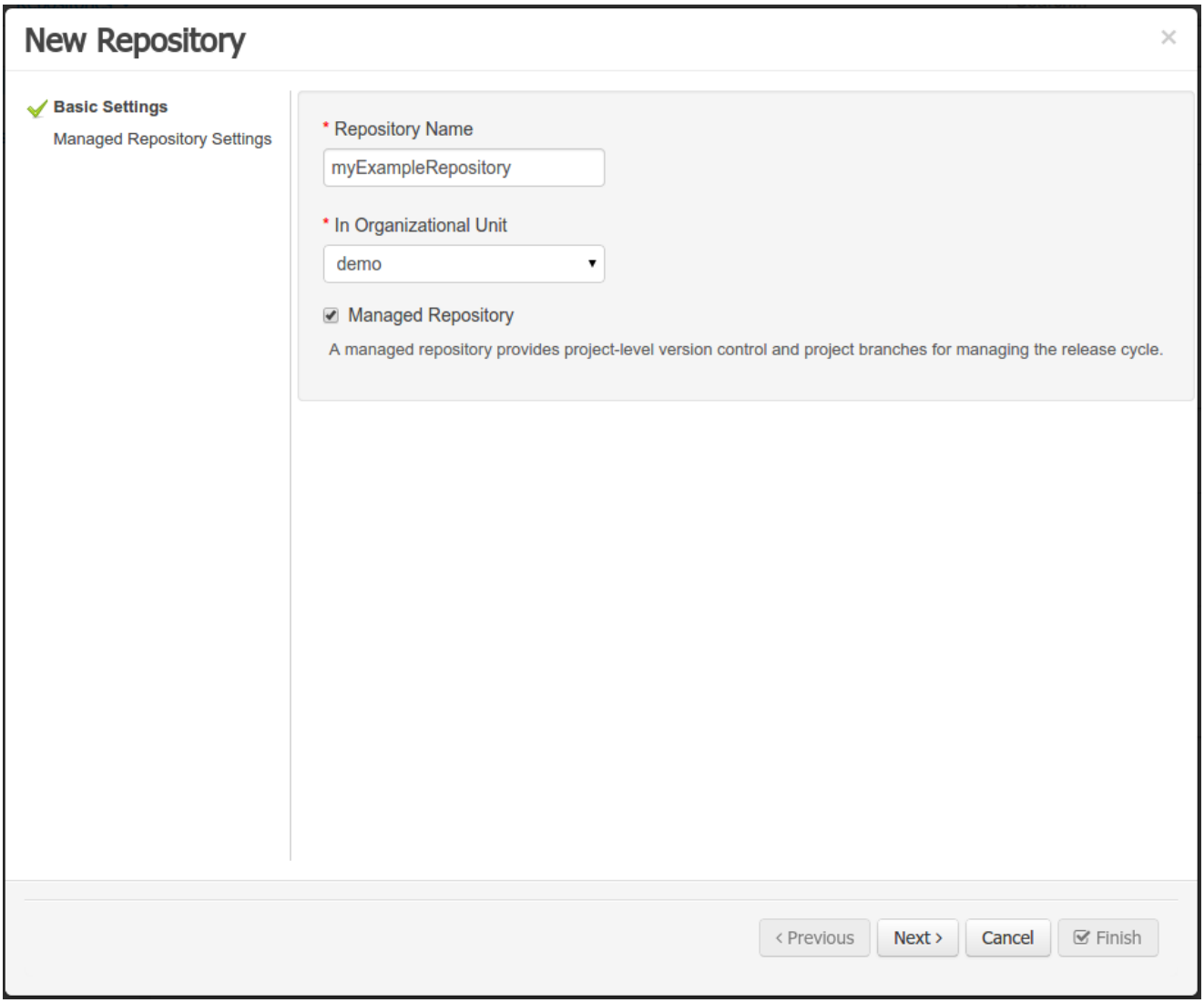


Figure 18.3. Entering repository information step 1/2

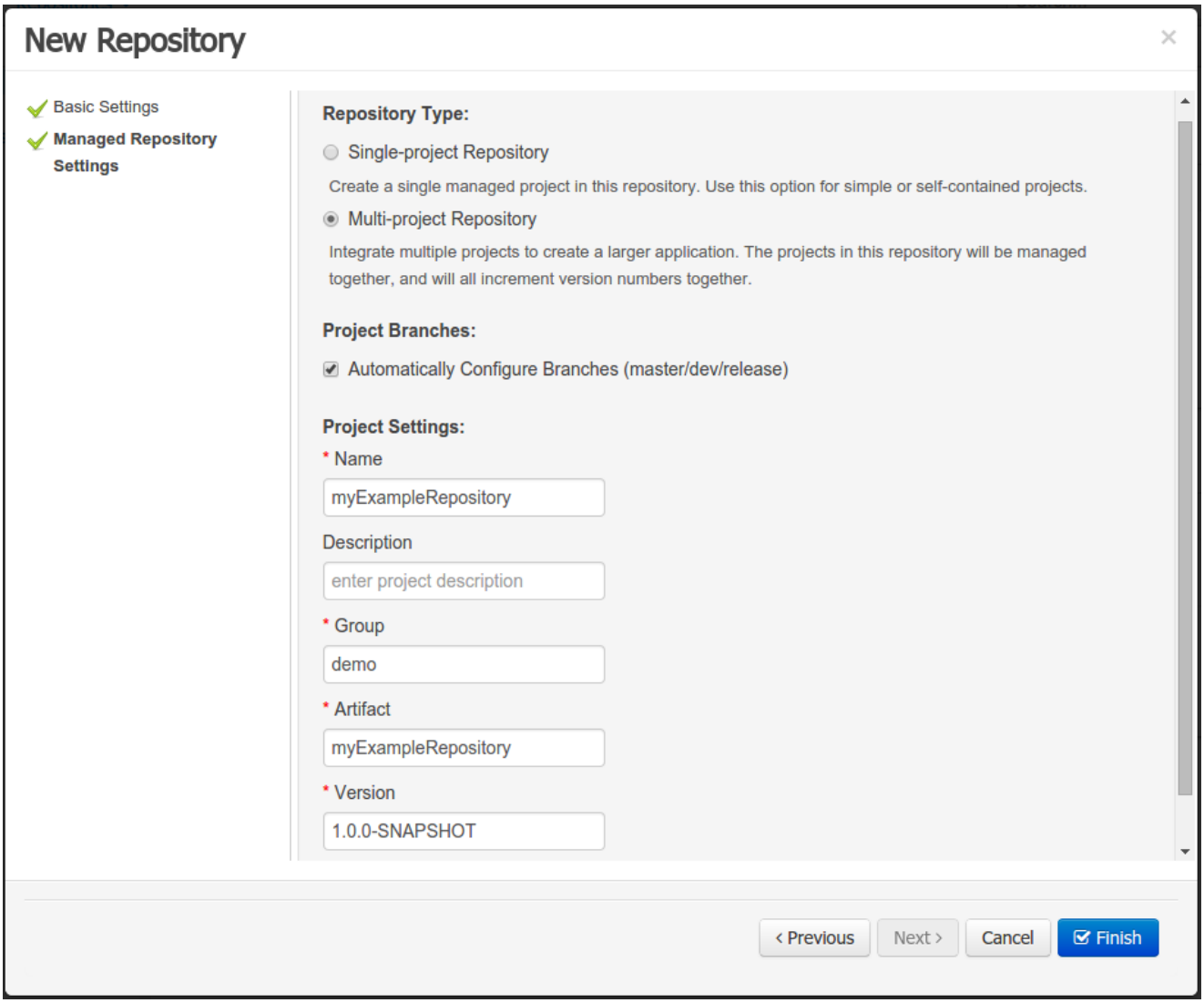


Figure 18.4. Entering repository information step 2/2 (only for managed repositories)

18.2.2. Add project

Select the Authoring Perspective to create a new project.

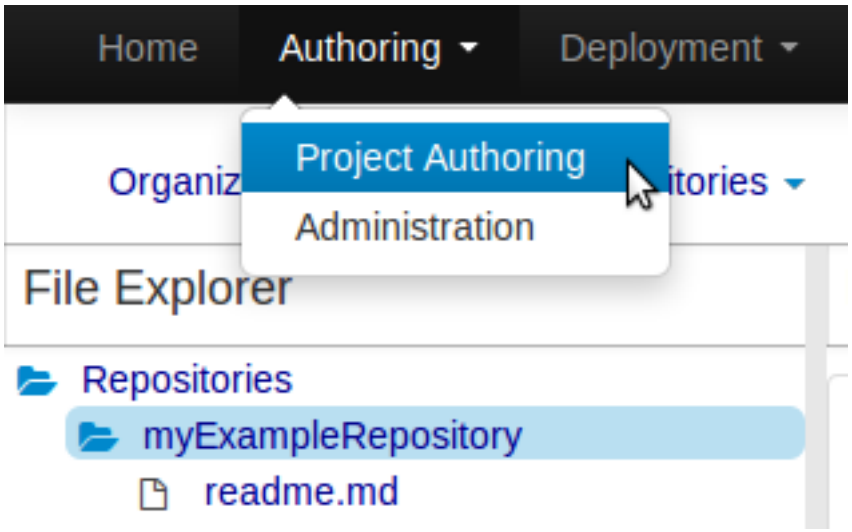


Figure 18.5. Selecting Authoring perspective

Select "Project" from the "New Item" menu.

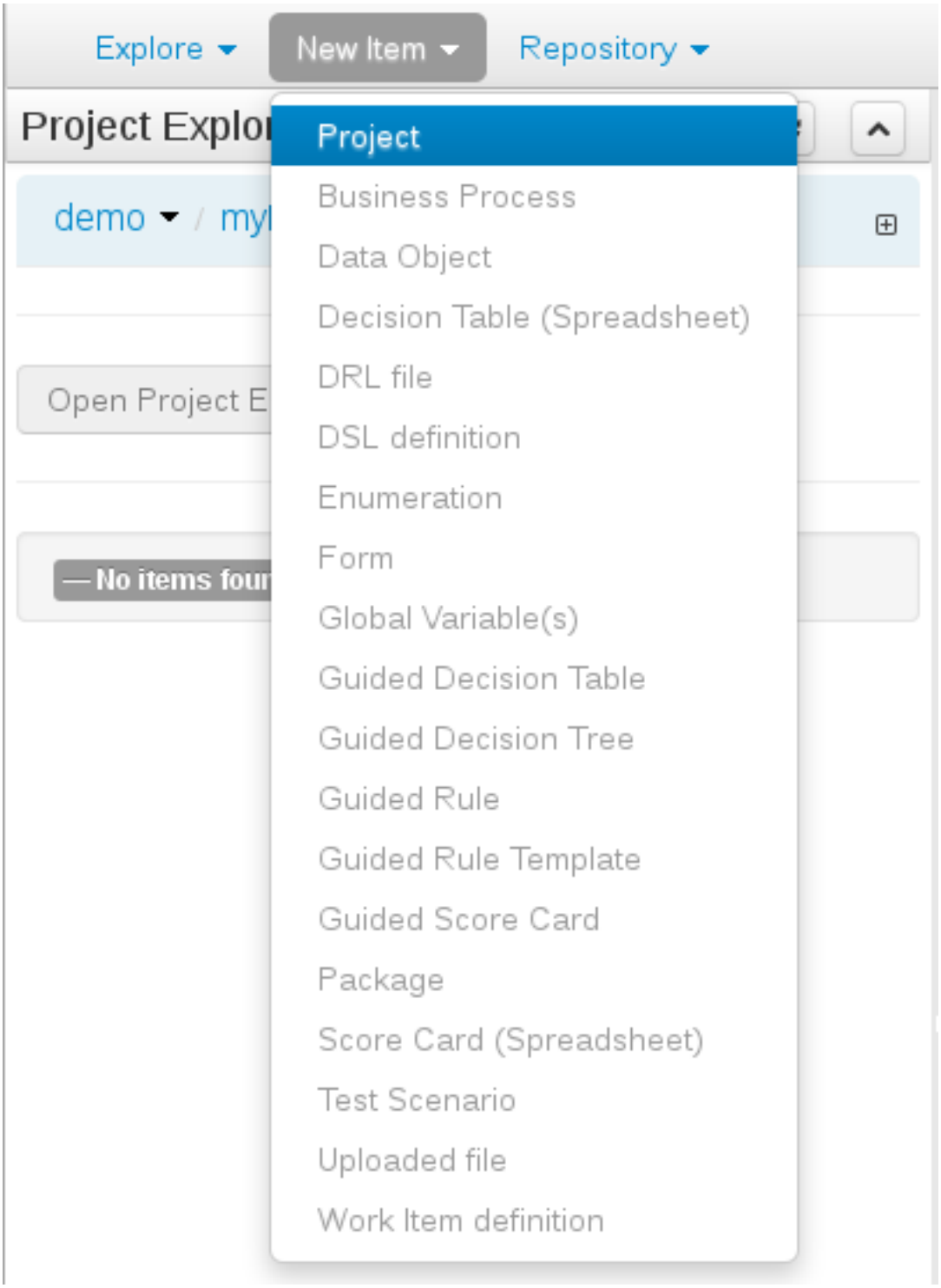


Figure 18.6. Creating new project

Enter a project name first.

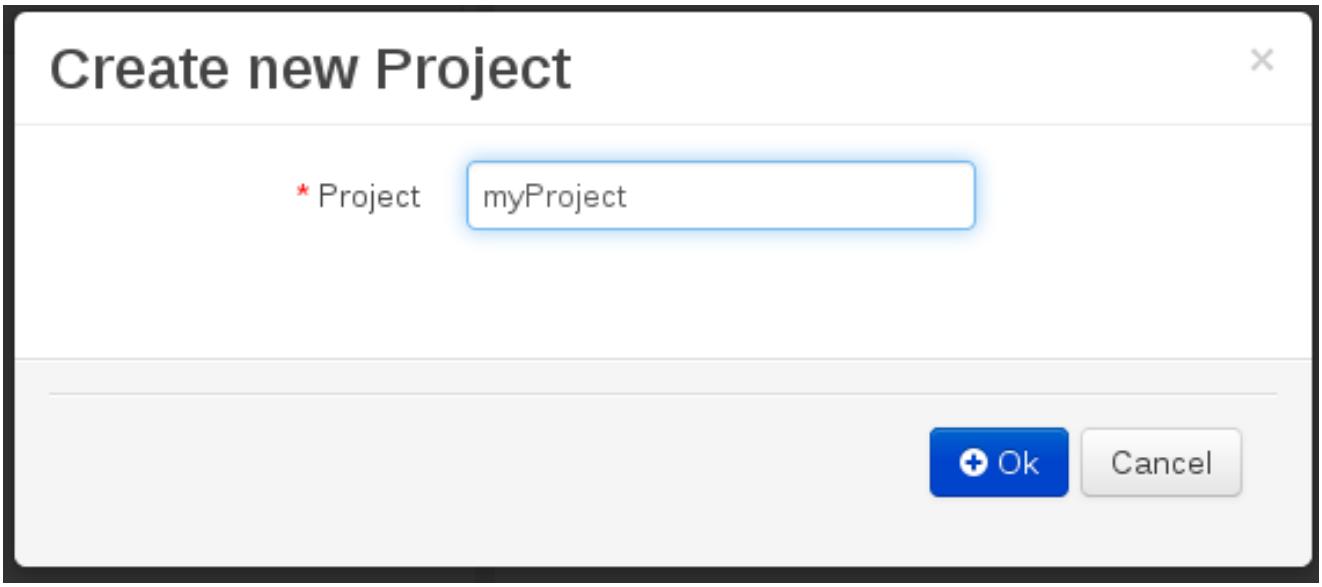


Figure 18.7. Entering project name

Enter the project details next.

- Group ID follows Maven conventions.
- Artifact ID is pre-populated from the project name.
- Version is set as 1.0 by default.

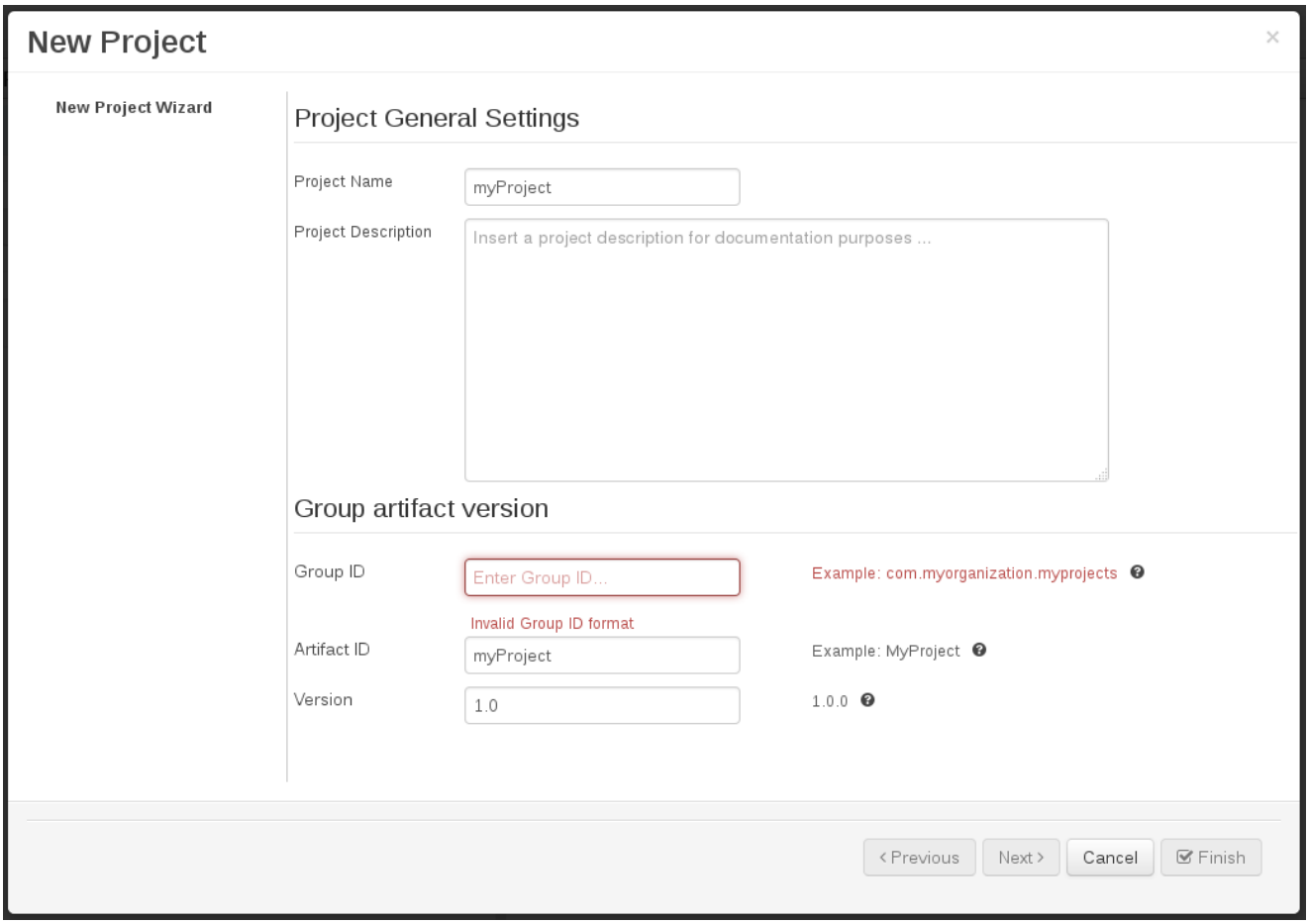


Figure 18.8. Entering project details

18.2.3. Define Data Model

After a project has been created you need to define Types to be used by your rules.

Select "Data Object" from the "New Item" menu.

Note

You can also use types contained in existing JARs.

Please consult the full documentation for details.

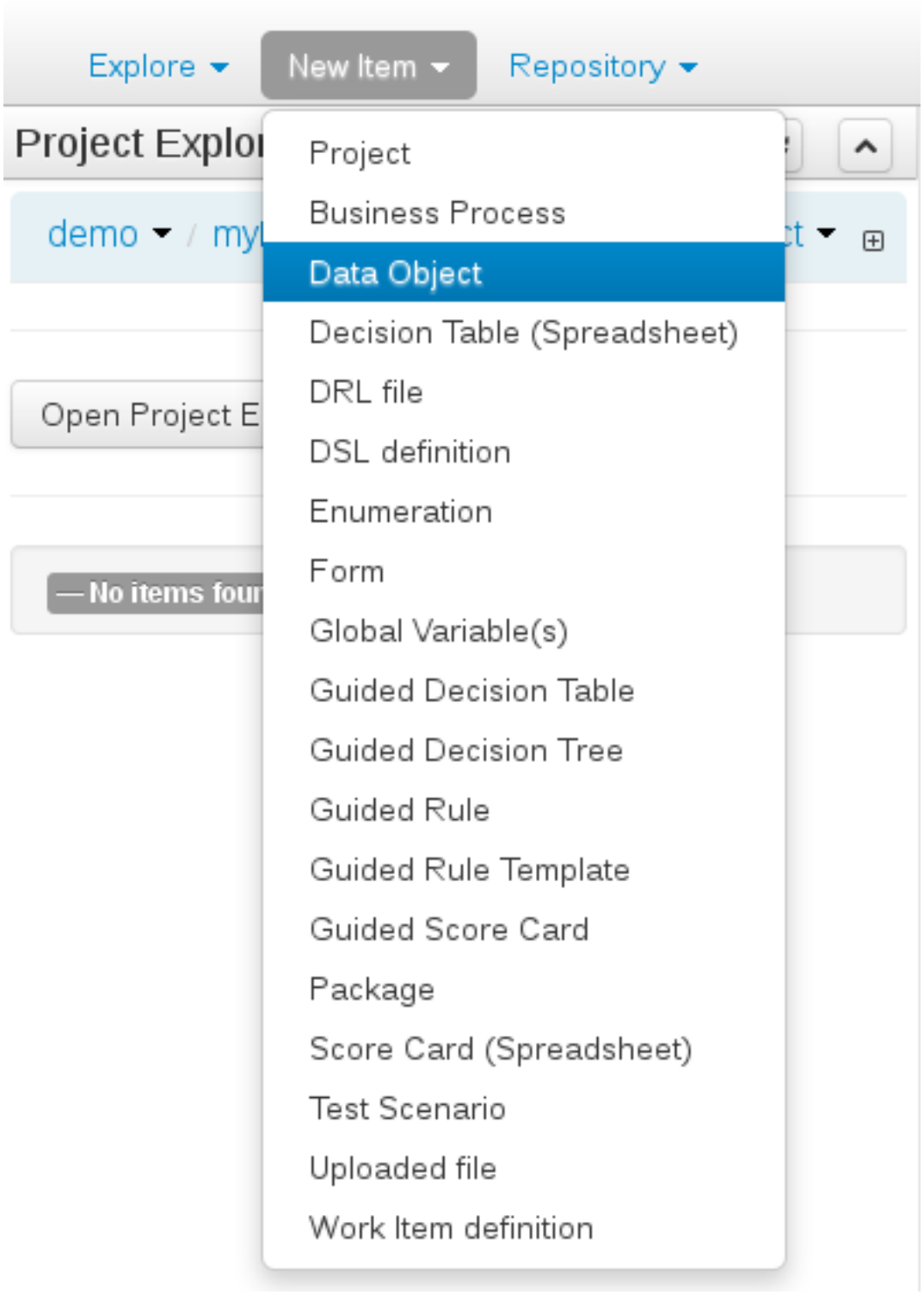


Figure 18.9. Creating "Data Object"

Set the name and select a package for the new type.

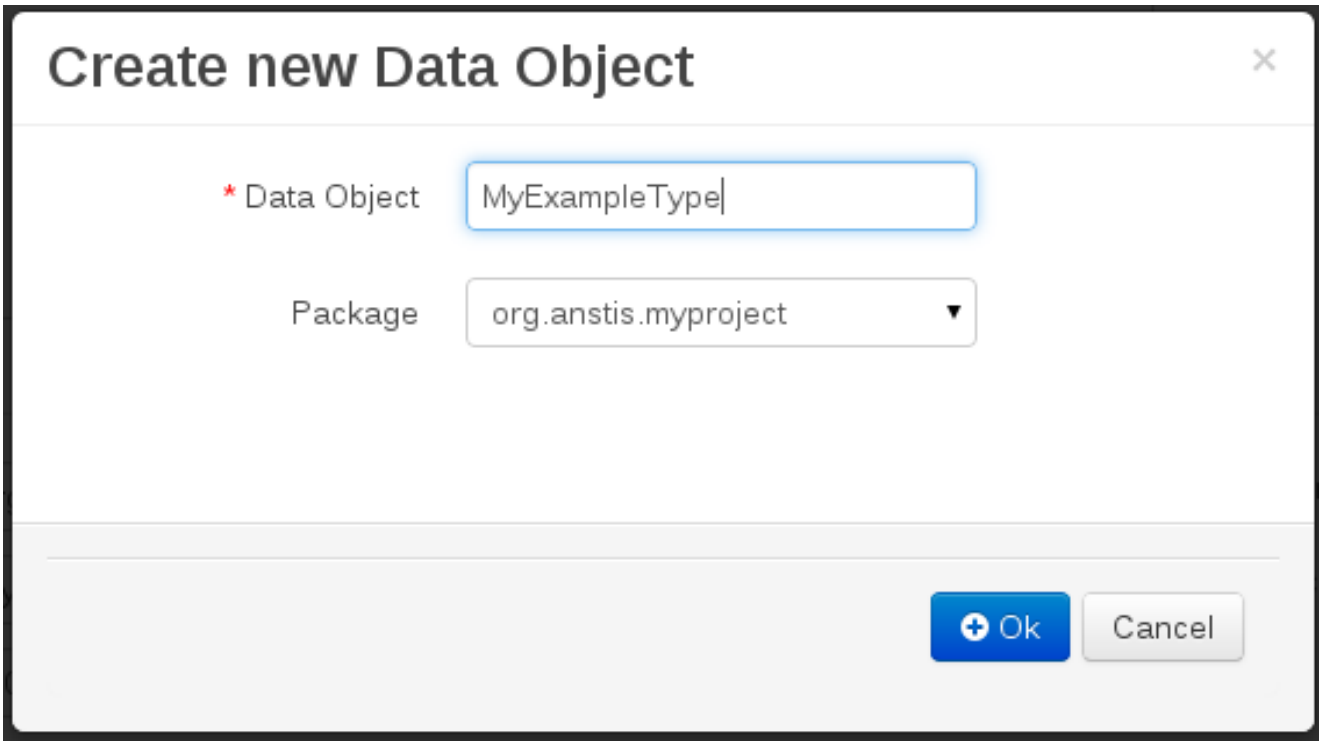


Figure 18.10. Creating a new type

Set field name and type and click on "Create" to create a field for the type.

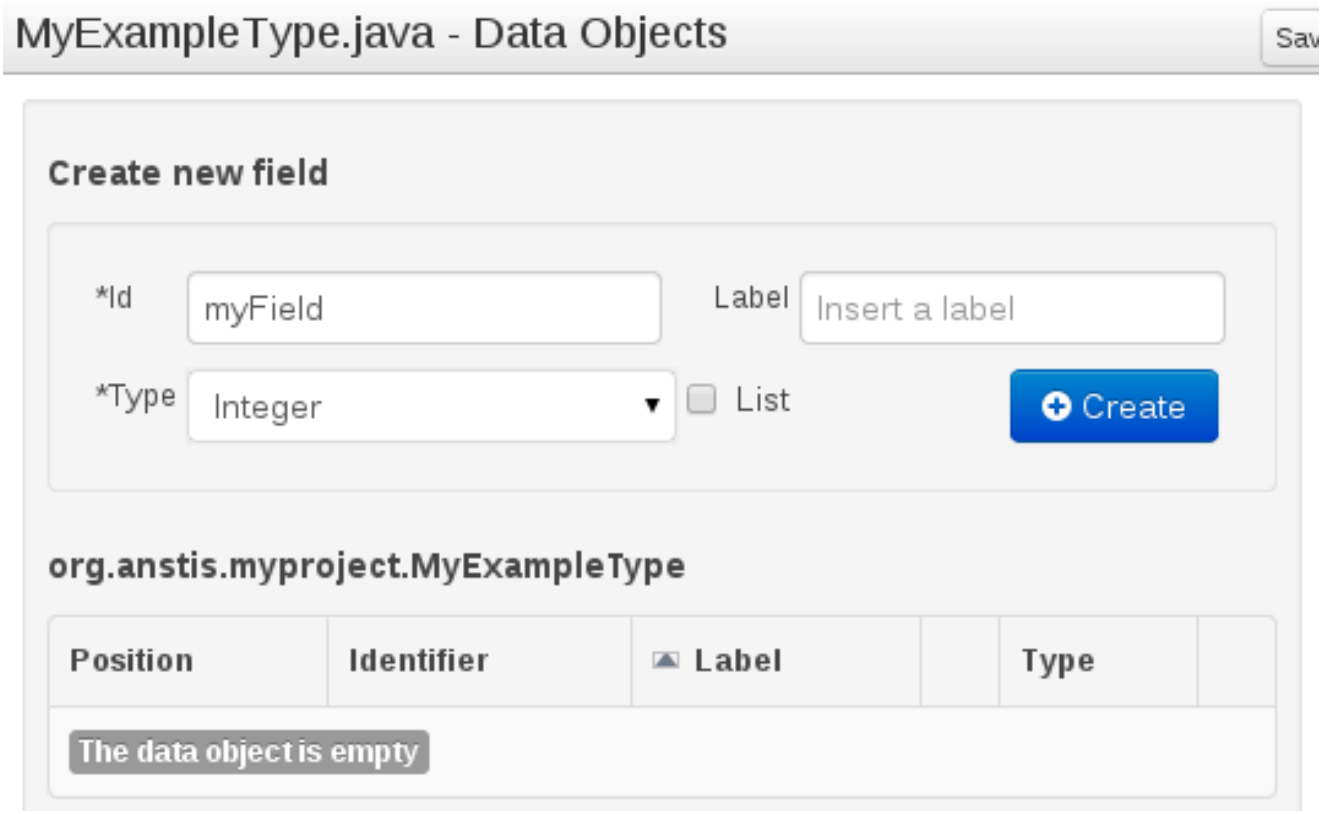


Figure 18.11. Click "Create" and add the field

Click "Save" to update the model.

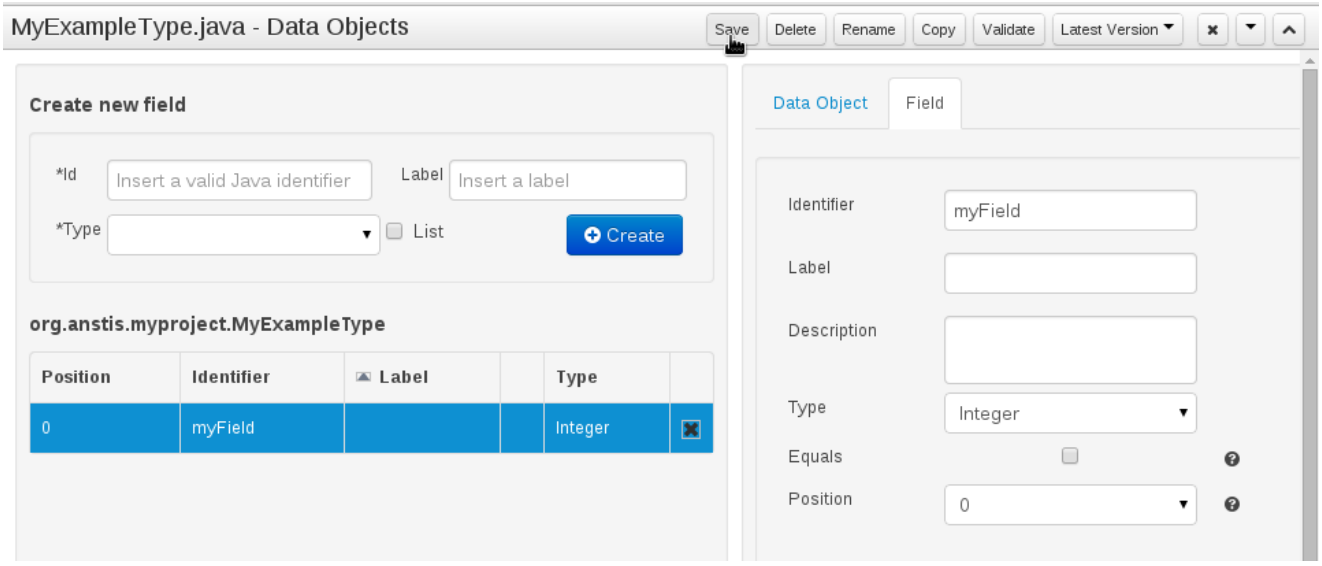


Figure 18.12. Clicking "Save"

18.2.4. Define Rule

Select "DRL file" (for example) from the "New Item" menu.

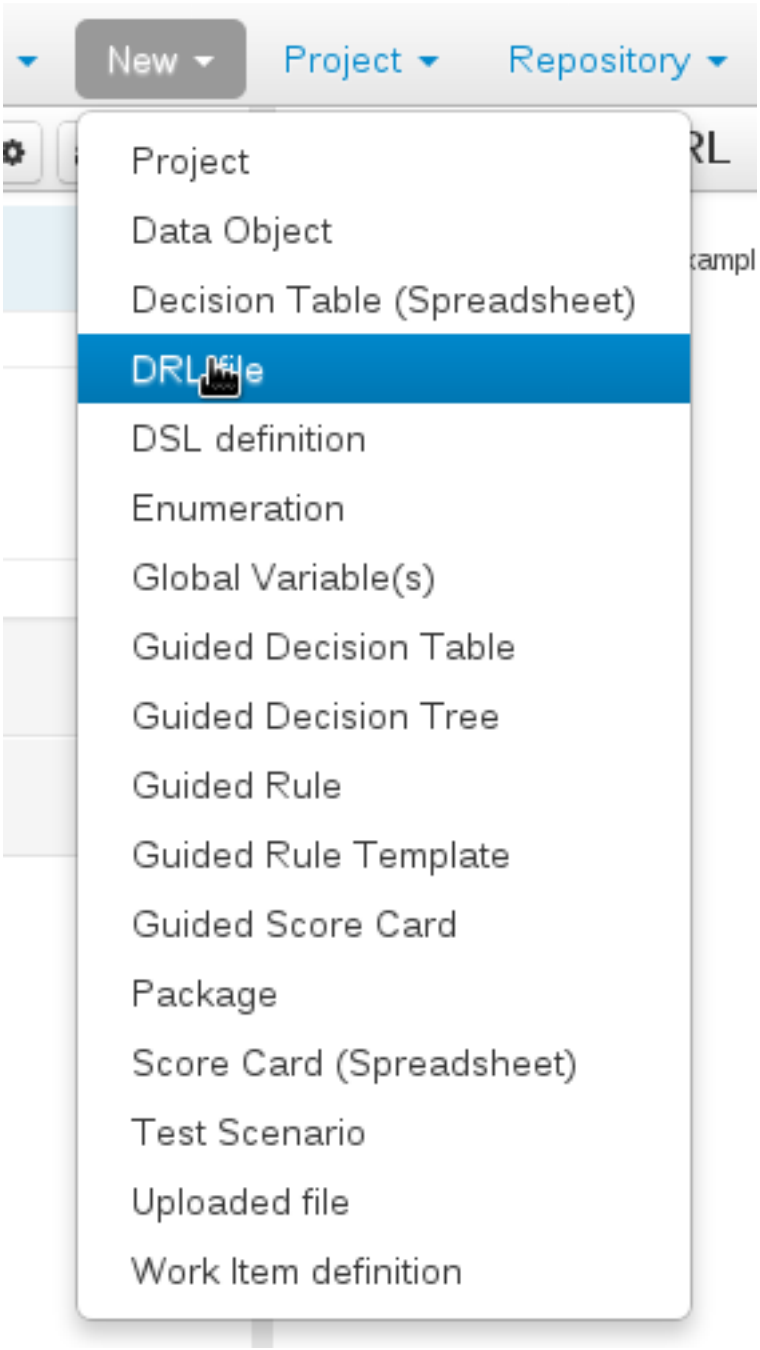


Figure 18.13. Selecting "DRL file" from the "New Item" menu

Enter a file name for the new rule.

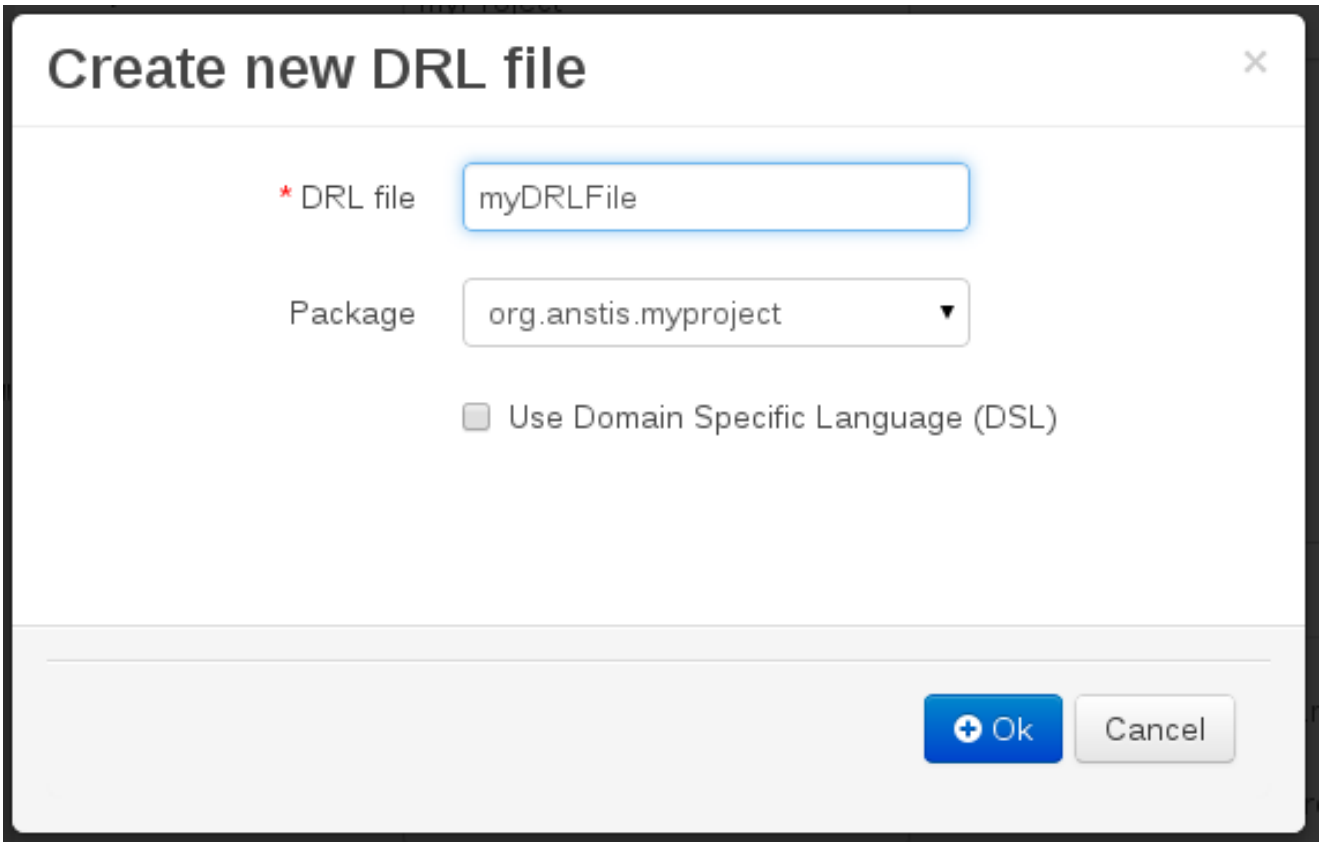


Figure 18.14. Entering file name for rule

Enter a definition for the rule.

The definition process differs from asset type to asset type.

The full documentation has details about the different editors.

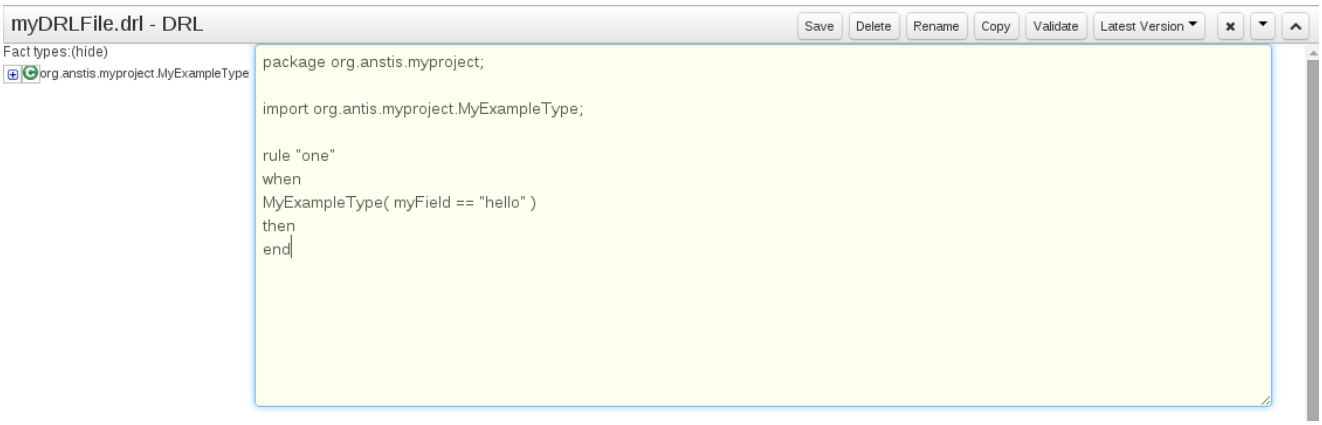


Figure 18.15. Defining a rule

Once the rule has been defined it will need to be saved.

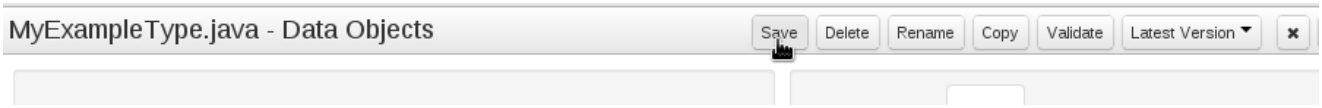


Figure 18.16. Saving the rule

18.2.5. Build and Deploy

Once rules have been defined within a project; the project can be built and deployed to the Workbench's Maven Artifact Repository.

To build a project select the "Project Editor" from the "Project" menu.

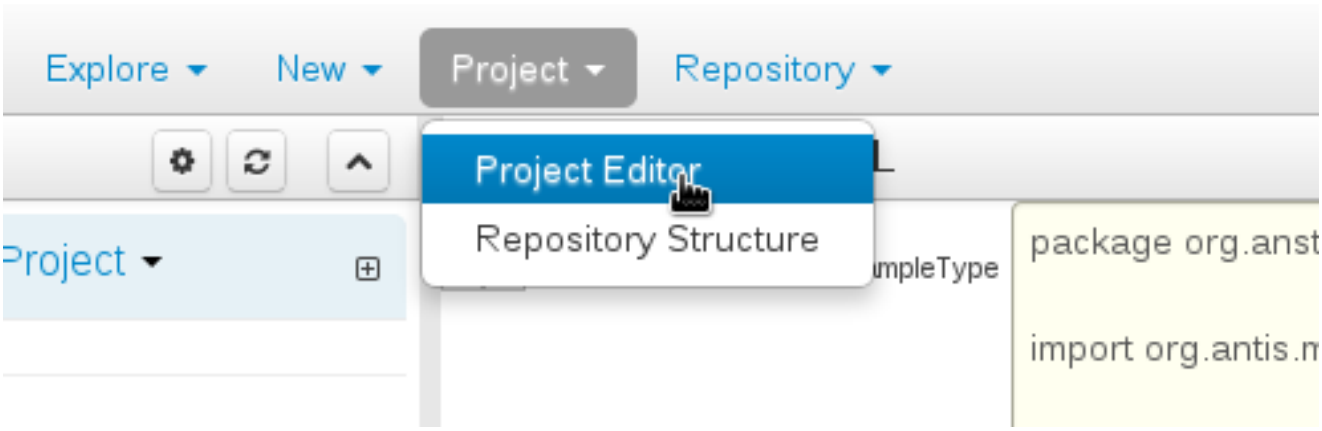


Figure 18.17. Selecting "Project Editor"

Click "Build and Deploy" to build the project and deploy it to the Workbench's Maven Artifact Repository.

When you select Build & Deploy the workbench will deploy to any repositories defined in the Dependency Management section of the pom in your workbench project. You can edit the pom.xml file associated with your workbench project under the Repository View of the project explorer. Details on dependency management in maven can be found here : http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

If there are errors during the build process they will be reported in the "Problems Panel".

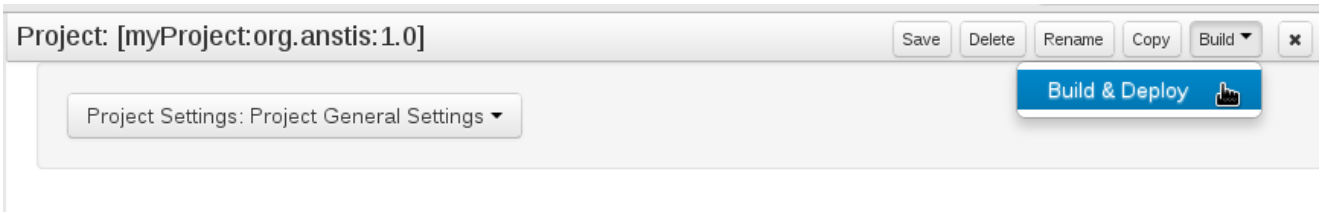


Figure 18.18. Building and deploying a project

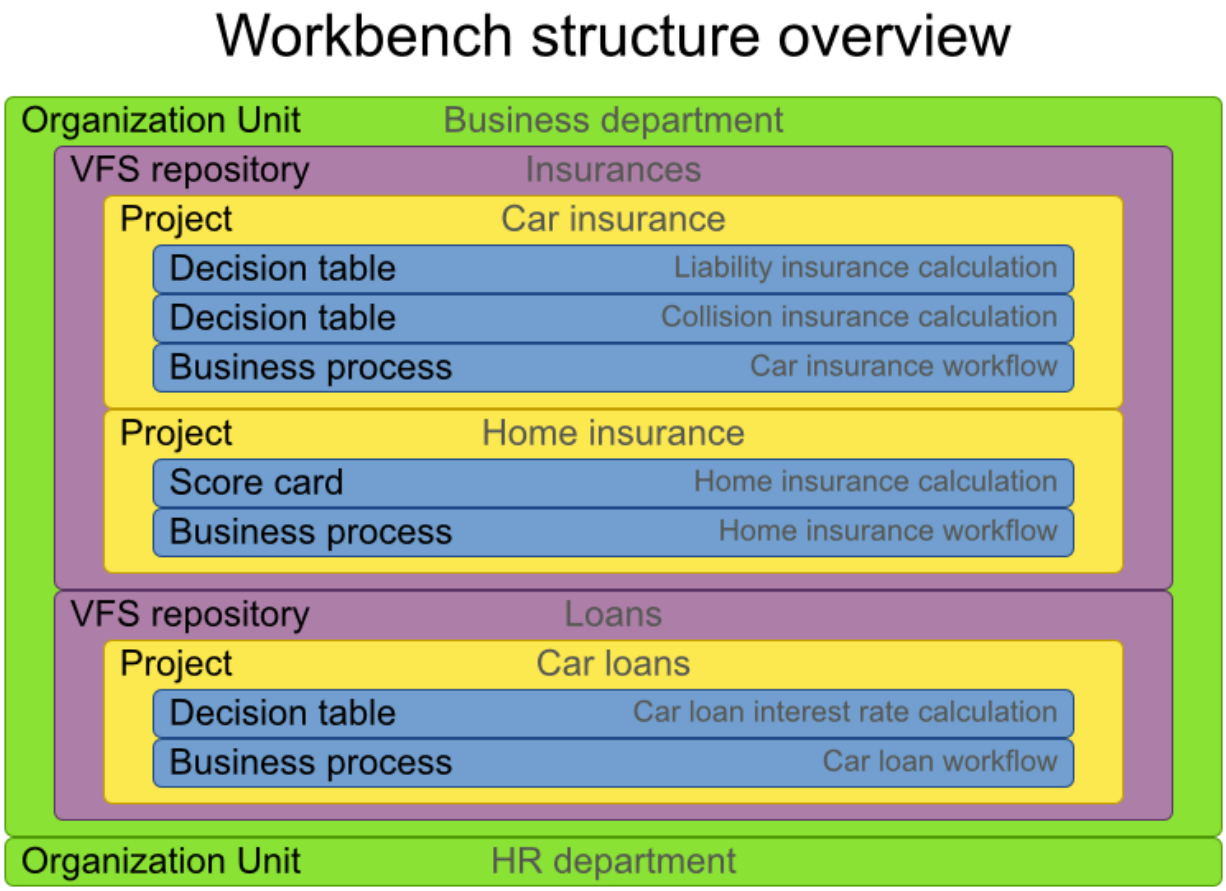
Now the project has been built and deployed; it can be referenced from your own projects as any other Maven Artifact.

The full documentation contains details about integrating projects with your own applications.

18.3. Administration

18.3.1. Administration overview

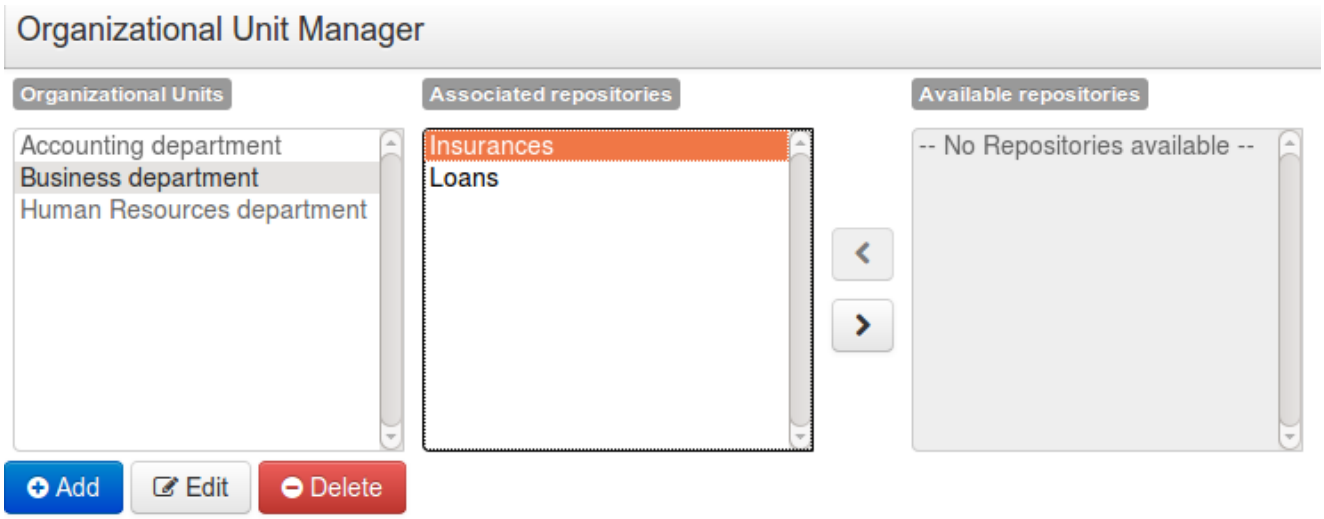
A workbench is structured with Organization Units, VFS repositories and projects:



18.3.2. Organizational unit

Organization units are useful to model departments and divisions.

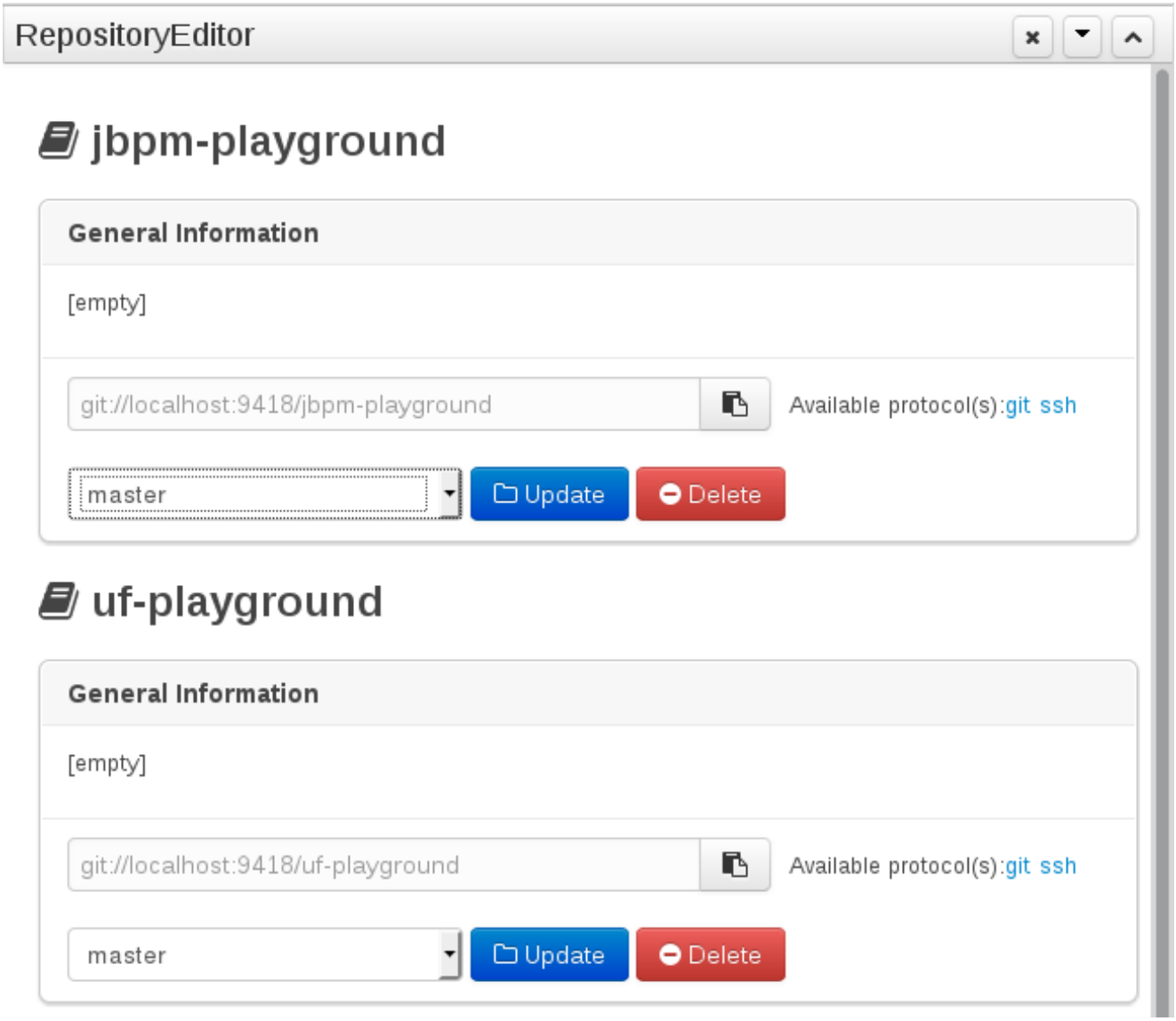
An organization unit can hold multiple repositories.



18.3.3. Repositories

Repositories are the place where assets are stored and each repository is organized by projects and belongs to a single organization unit.

Repositories are in fact a Virtual File System based storage, that by default uses GIT as backend. Such setup allows workbench to work with multiple backends and, in the same time, take full advantage of backend specifics features like in GIT case versioning, branching and even external access.



A new repository can be created from scratch or cloned from an existing repository.

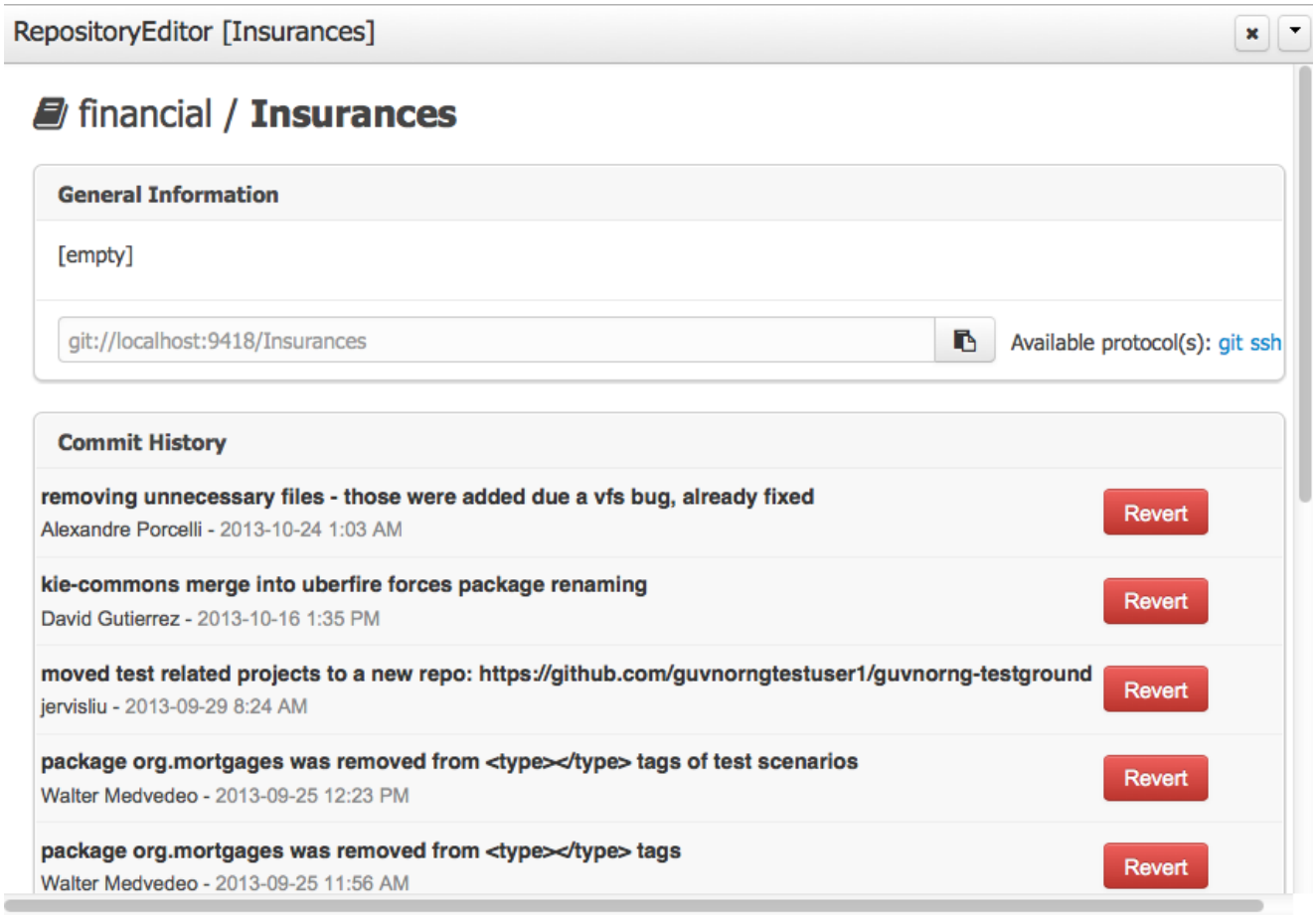
One of the biggest advantage of using GIT as backend is the ability to clone a repository from external and use your preferred tools to edit and build your assets.

Warning

Never clone your repositories directly from .niogit directory. Use always the available protocol(s) displayed in repositories editor.

18.3.3.1. Repository Editor

One additional advantage to use GIT as backend is the possibility to revert your repository to a previous state. You can do it directly from the repository editor by browsing its commit history and clicking the **Revert** button.



18.4. Configuration

18.4.1. User management

The workbench authenticates its users against the application server's authentication and authorization (JAAS).

On JBoss EAP and WildFly, add a user with the script `$JBOSS_HOME/bin/add-user.sh` (or `.bat`):

```
$ ./add-user.sh
// Type: Application User
// Realm: empty (defaults to ApplicationRealm)
// Role: admin
```

There is no need to restart the application server.

18.4.2. Roles

The Workbench uses the following roles:

- admin

- analyst
- developer
- manager
- user

18.4.2.1. Admin

Administrates the BPMS system.

- Manages users
- Manages VFS Repositories
- Has full access to make any changes necessary

18.4.2.2. Developer

Developer can do almost everything admin can do, except clone repositories.

- Manages rules, models, process flows, forms and dashboards
- Manages the asset repository
- Can create, build and deploy projects
- Can use the JBDS connection to view processes

18.4.2.3. Analyst

Analyst is a weaker version of developer and does not have access to the asset repository or the ability to deploy projects.

18.4.2.4. Business user

Daily user of the system to take actions on business tasks that are required for the processes to continue forward. Works primarily with the task lists.

- Does process management
- Handles tasks and dashboards

18.4.2.5. Manager/Viewer-only User

Viewer of the system that is interested in statistics around the business processes and their performance, business indicators, and other reporting of the system and people who interact with the system.

- Only has access to dashboards

18.4.3. Restricting access to repositories

It is possible to restrict access to repositories using roles and organizational groups. To let an user access a repository.

The user either has to belong into a role that has access to the repository or to a role that belongs into an orgazinational group that has access to the repository. These restrictions can be managed with the command line config tool.

18.4.4. Command line config tool

Provides capabilities to manage the system repository from command line. System repository contains the data about general workbench settings: how editors behave, organizational groups, security and other settings that are not editable by the user. System repository exists in the .niogit folder, next to all the repositories that have been created or cloned into the workbench.

18.4.4.1. Config Tool Modes

- Online (default and recommended) - Connects to the Git repository on startup, using Git server provided by the KIE Workbench. All changes are made locally and published to upstream when:
  - "push-changes" command is explicitly executed
  - "exit" is used to close the tool
- Offline - Creates and manipulates system repository directly on the server (no discard option)

18.4.4.2. Available Commands

Table 18.1. Available Commands

exit	Publishes local changes, cleans up temporary directories and quits the command line tool
discard	Discards local changes without publishing them, cleans up temporary directories and quits the command line tool
help	Prints a list of available commands
list-repo	List available repositories
list-org-units	List available organizational units
list-deployment	List available deployments
create-org-unit	Creates new organizational unit
remove-org-unit	Removes existing organizational unit

add-deployment	Adds new deployment unit
remove-deployment	Removes existing deployment
create-repo	Creates new git repository
remove-repo	Removes existing repository (only from config)
add-repo-org-unit	Adds repository to the organizational unit
remove-repo-org-unit	Removes repository from the organizational unit
add-role-repo	Adds role(s) to repository
remove-role-repo	Removes role(s) from repository
add-role-org-unit	Adds role(s) to organizational unit
remove-role-org-unit	Removes role(s) from organizational unit
add-role-project	Adds role(s) to project
remove-role-project	Removes role(s) from project
push-changes	Pushes changes to upstream repository (only in online mode)

18.4.4.3. How to use

The tool can be found from kie-config-cli-${version}-dist.zip. Execute the kie-config-cli.sh script and by default it will start in online mode asking for a Git url to connect to (the default value is ssh://localhost/system). To connect to a remote server, replace the host and port with appropriate values, e.g. ssh://kie-wb-host/system.

```
./kie-config-cli.sh
```

To operate in offline mode, append the offline parameter to the kie-config-cli.sh command. This will change the behaviour and ask for a folder where the .niogit (system repository) is. If .niogit does not yet exist, the folder value can be left empty and a brand new setup is created.

```
./kie-config-cli.sh offline
```

18.5. Introduction

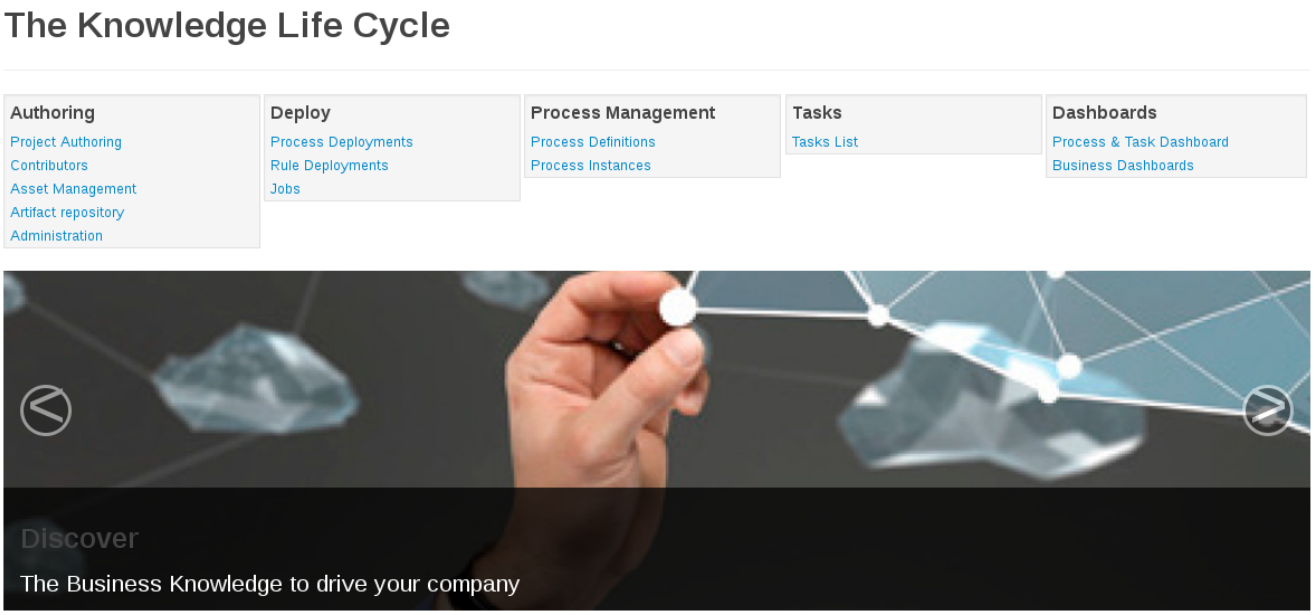
18.5.1. Log in and log out

Create a user with the role `admin` and log in with those credentials.

After successfully logging in, the account username is displayed at the top right. Click on it to review the roles of the current account.

18.5.2. Home screen

After logging in, the home screen shows. The actual content of the home screen depends on the workbench variant (Drools, jBPM, ...).



18.5.3. Workbench concepts

The Workbench is comprised of different logical entities:

- Part

  A Part is a screen or editor with which the user can interact to perform operations.

  Example Parts are "Project Explorer", "Project Editor", "Guided Rule Editor" etc. Parts can be repositioned.

- Panel

  A Panel is a container for one or more Parts.

  Panels can be resized.

- Perspective

  A perspective is a logical grouping of related Panels and Parts.

  The user can switch between perspectives by clicking on one of the top-level menu items; such as "Home", "Authoring", "Deploy" etc.

18.5.4. Initial layout

The Workbench consists of three main sections to begin; however its layout and content can be changed.

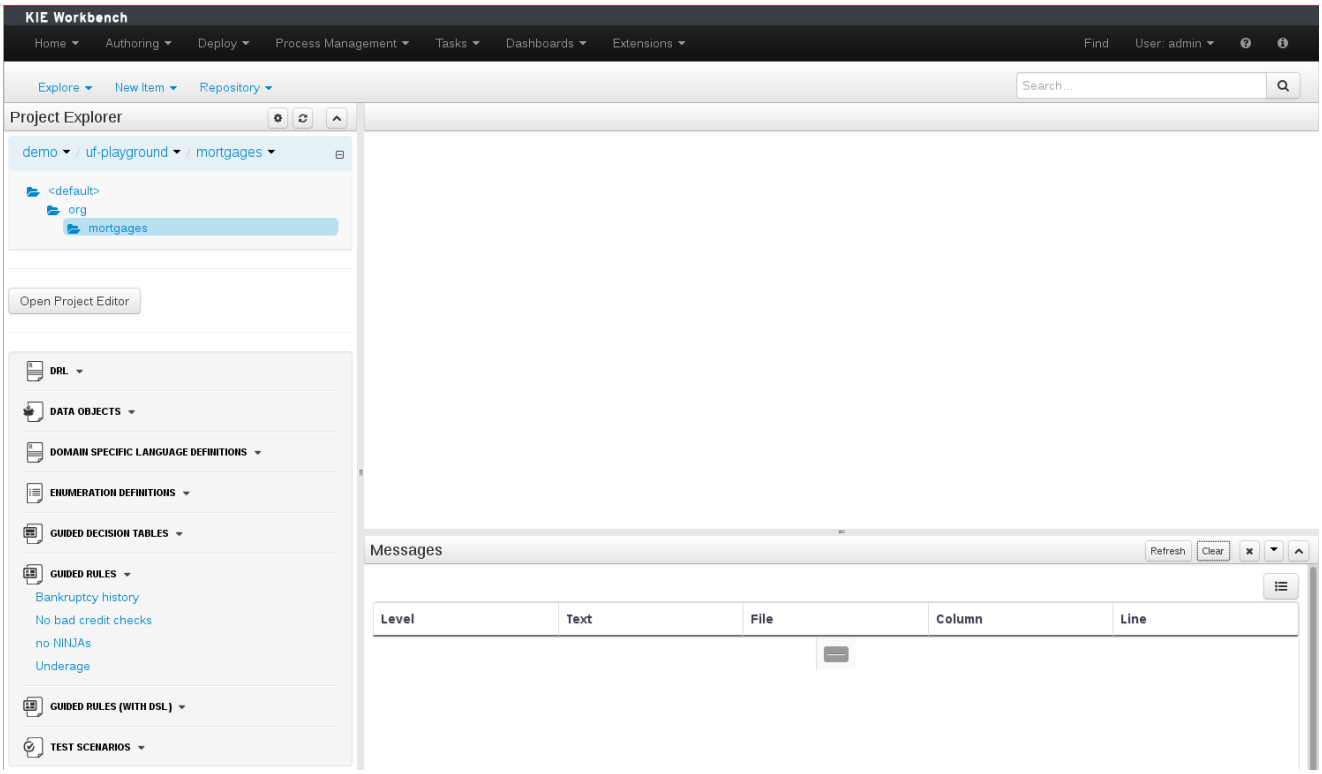


Figure 18.19. The Workbench

The initial Workbench shows the following components:-

- Project Explorer

  This provides the ability for the user to browse their configuration; of Organizational Units (in the above "example" is the Organizational Unit), Repositories (in the above "uf-playground" is the Repository) and Project (in the above "mortgages" is the Project).

- Problems

  This provides the user will real-time feedback about errors in the active Project.

- Empty space

  This empty space will contain an editor for assets selected from the Project Explorer.

  Other screens will also occupy this space by default; such as the Project Editor.

18.6. Changing the layout

The default layout may not be suitable for a user. Panels can therefore be either resized or repositioned.

This, for example, could be useful when running tests; as the test defintion and rule can be repositioned side-by-side.

18.6.1. Resizing

The following screenshot shows a Panel being resized.

Move the mouse pointer over the panel splitter (a grey horizontal or vertical line in between panels).

The cursor will changing indicating it is positioned correctly over the splitter. Press and hold the left mouse button and drag the splitter to the required position; then release the left mouse button.

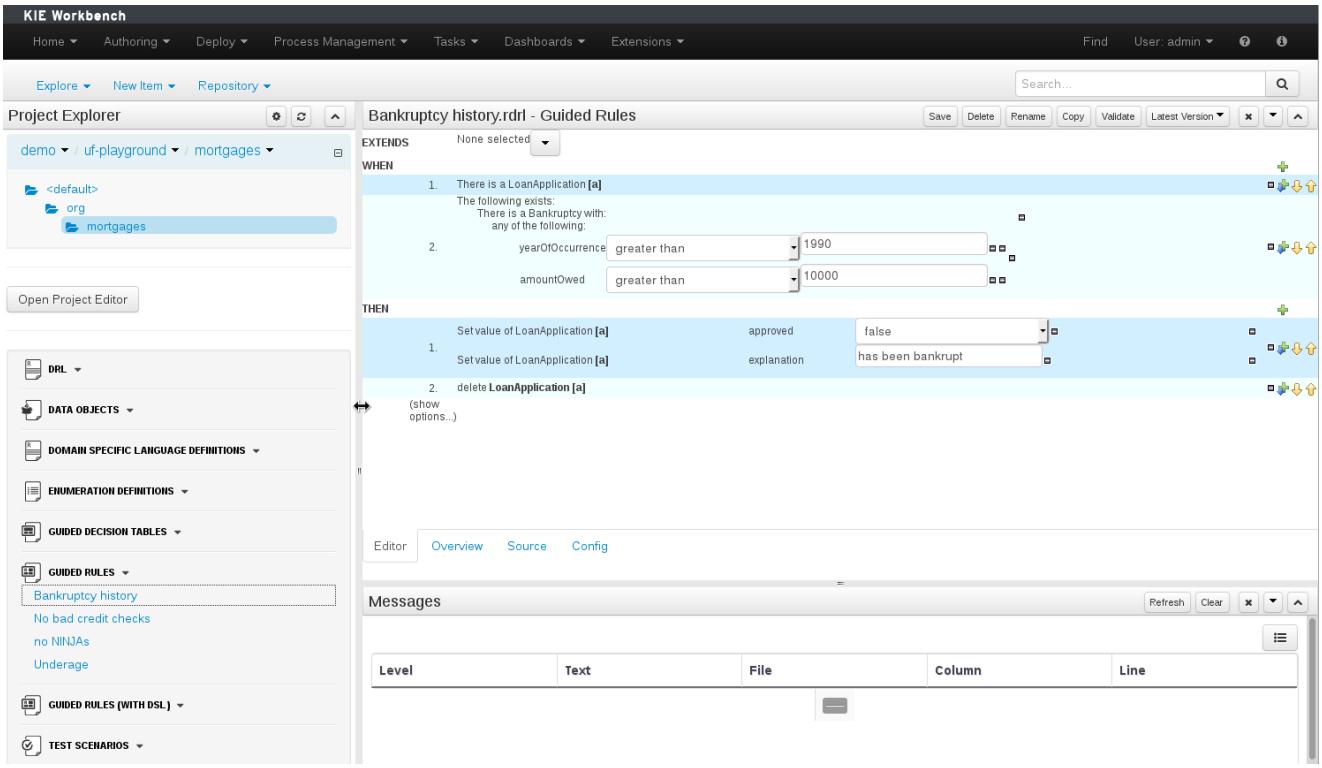


Figure 18.20. Resizing

18.6.2. Repositioning

The following screenshot shows a Panel being repositioned.

Move the mouse pointer over the Panel title ("Guided Editor [No bad credit checks]" in this example).

The cursor will change indicating it is positioned correctly over the Panel title. Press and hold the left mouse button. Drag the mouse to the required location. The target position is indicated with a pale blue rectangle. Different positions can be chosen by hovering the mouse pointer over the different blue arrows.

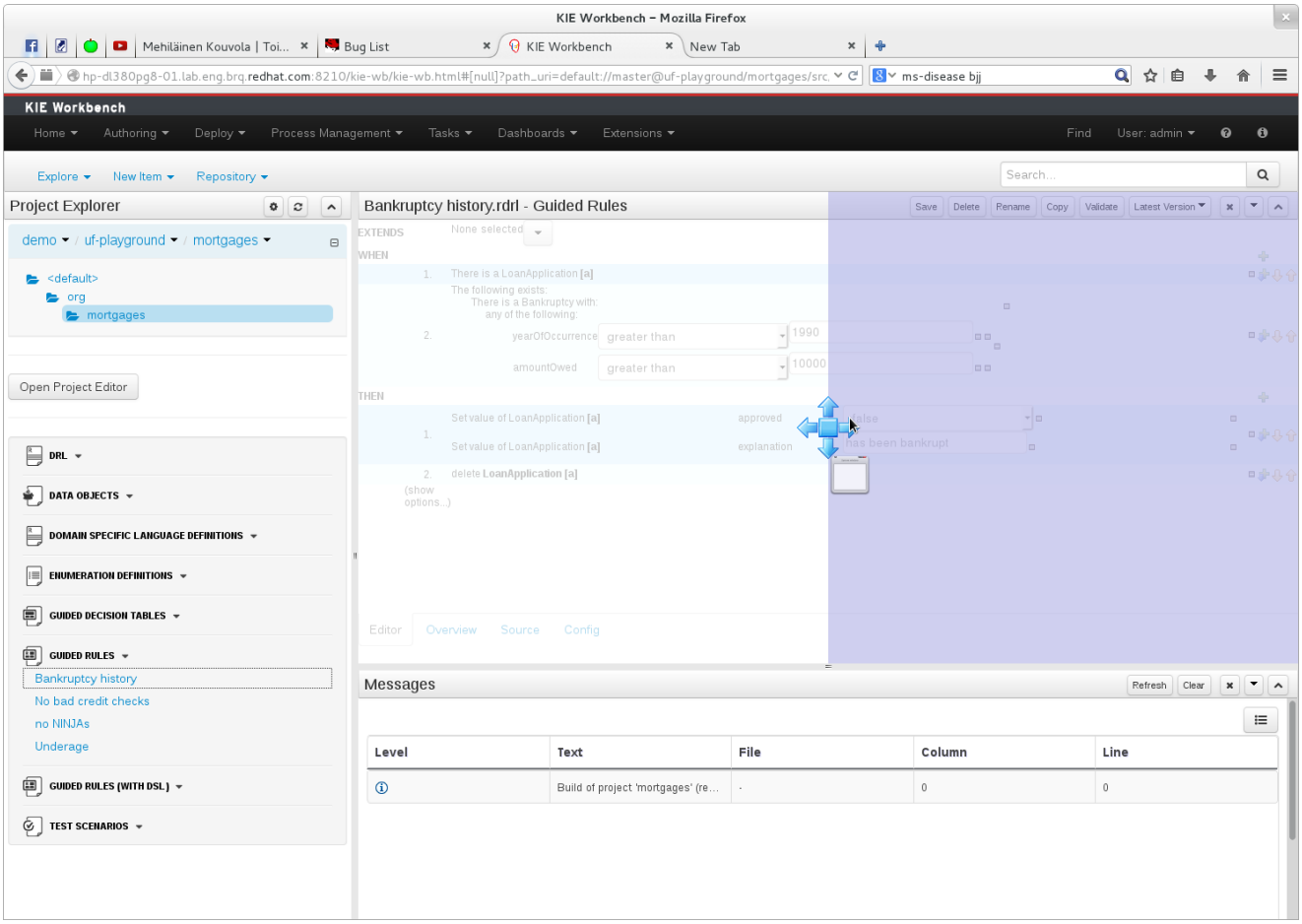


Figure 18.21. Repositioning - dragging

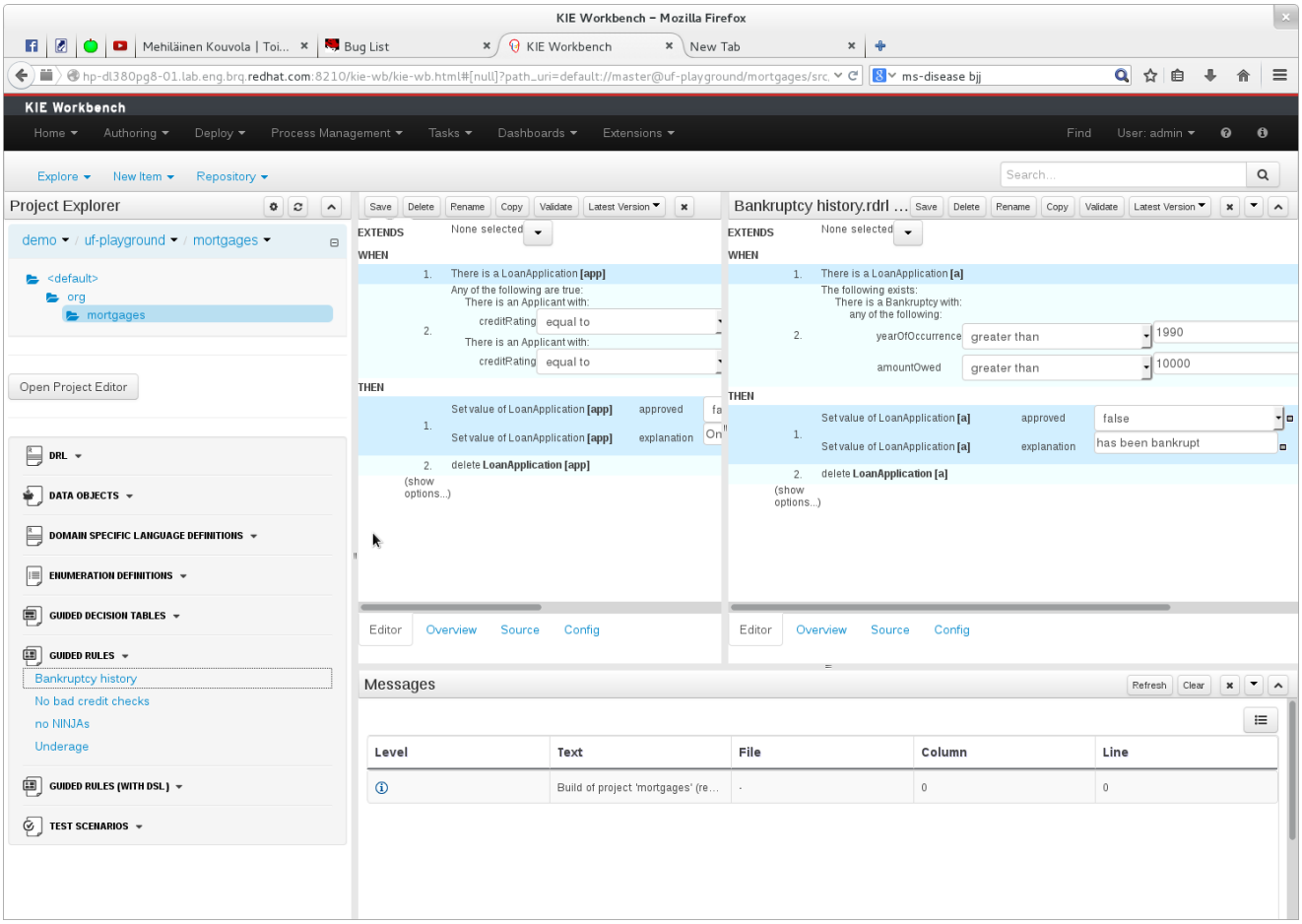


Figure 18.22. Repositioning - complete

18.7. Authoring

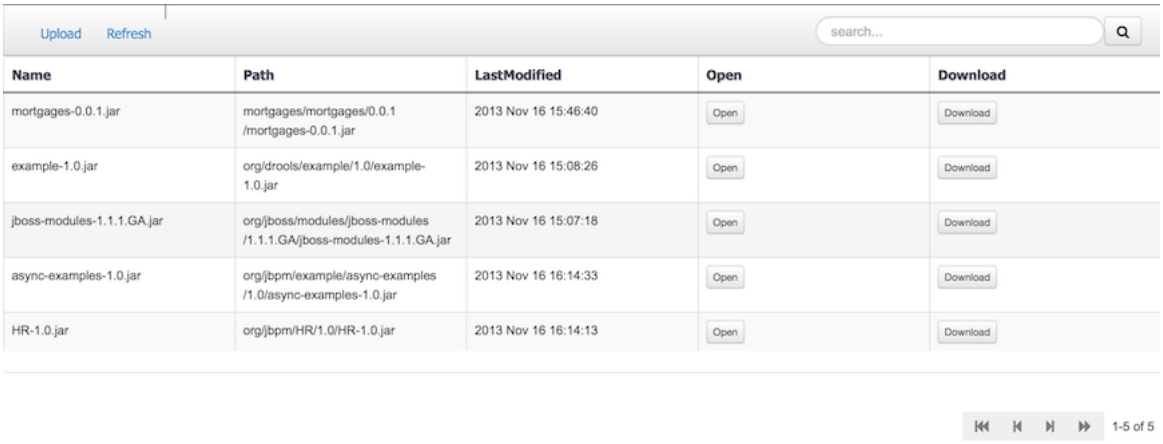
18.7.1. Artifact Repository

Projects often need external artifacts in their classpath in order to build, for example a domain model JARs. The artifact repository holds those artifacts.

The Artifact Repository is a full blown Maven repository. It follows the semantics of a Maven remote repository: all snapshots are timestamped. But it is often stored on the local hard drive.

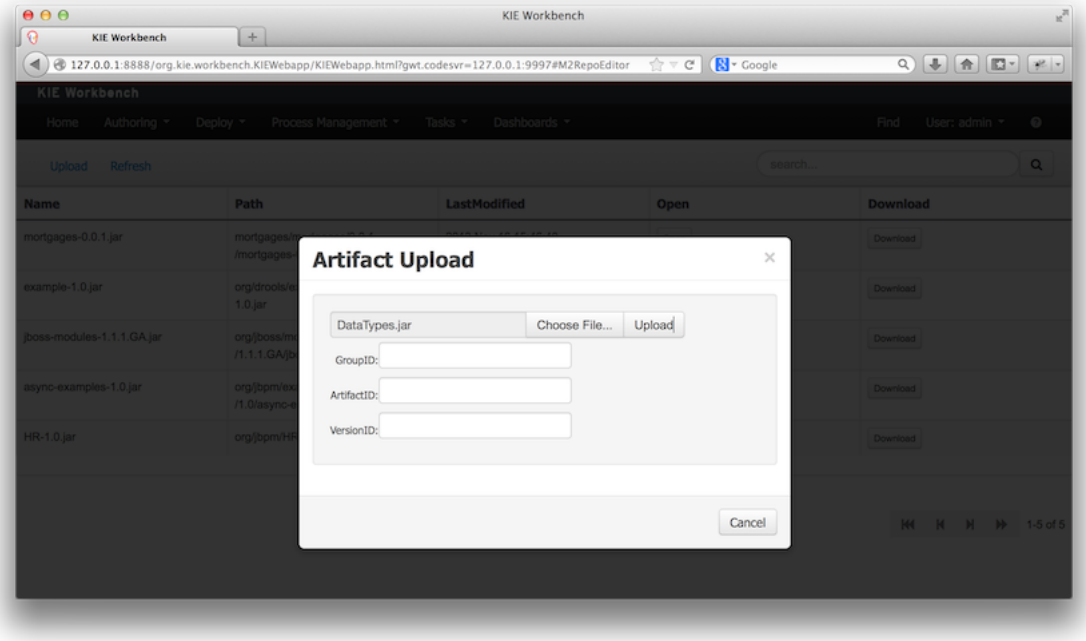
By default the artifact repository is stored under `$WORKING_DIRECTORY/repositories/kie`, but it can be overridden with the system property `-Dorg.guvnor.m2repo.dir`. There is only 1 Maven repository per installation.

The Artifact Repository screen shows a list of the artifacts in the Maven repository:



To add a new artifact to that Maven repository, either:

- Use the upload button and select a JAR. If the JAR contains a POM file under `META-INF/maven` (which every JAR build by Maven has), no further information is needed. Otherwise, a groupId, artifactId and version need be given too.



- Using Maven, `mvn deploy` to that Maven repository. Refresh the list to make it show up.

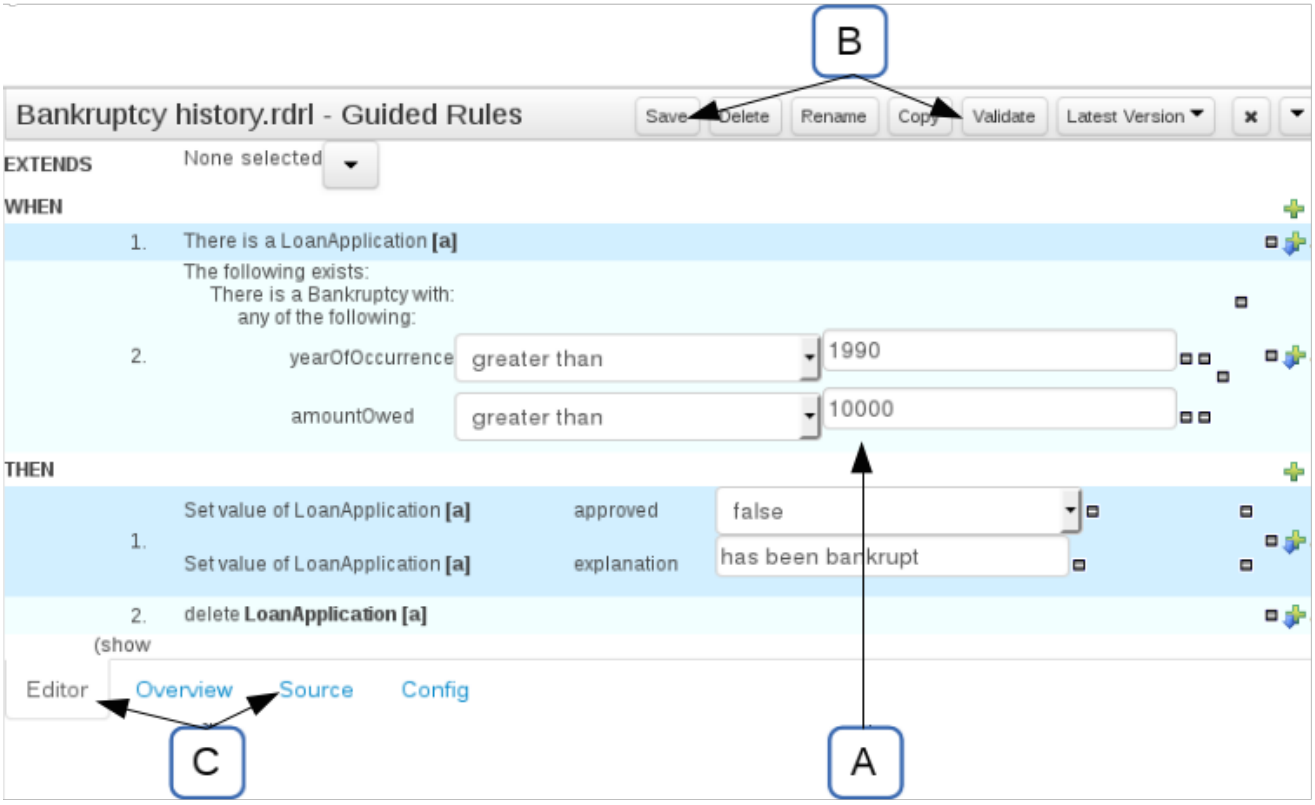
Note

This remote Maven repository is relatively simple. It does not support proxying, mirroring, ... like Nexus or Archiva.

18.7.2. Asset Editor

The Asset Editor is the principle component of the workbench User-Interface. It consists of two main views Editor and Overview.

- The views
  - A : The editing area - exactly what form the editor takes depends on the Asset type. An asset can only be edited by one user at a time to avoid conflicts. When a user begins to edit an asset, a lock will automatically be acquired. This is indicated by a lock symbol appearing on the asset title bar as well as in the project explorer view (see Section 18.7.4, “Project Explorer” for details). If a user starts editing an already locked asset a pop-up notification will appear to inform the user that the asset can't currently be edited, as it is being worked on by another user. Changes will be prevented until the editing user saves or closes the asset, or logs out of the workbench. Session timeouts will also cause locks to be released. Every user further has the option to force a lock release, if required (see the Metadata section below).
  - B : This menu bar contains various actions for the Asset; such as Save, Rename, Copy etc. Note that saving, renaming and deleting are deactivated if the asset is locked by a different user.
  - C : Different views for asset content or asset information.
    - Editor shows the main editor for the asset
    - Overview contains the metadata and conversation views for this editor. Explained in more detail below.
    - Source shows the asset in plain DRL. Note: This tab is only visible if the asset content can be generated into DRL.
    - Config contains the model imports used by the asset.



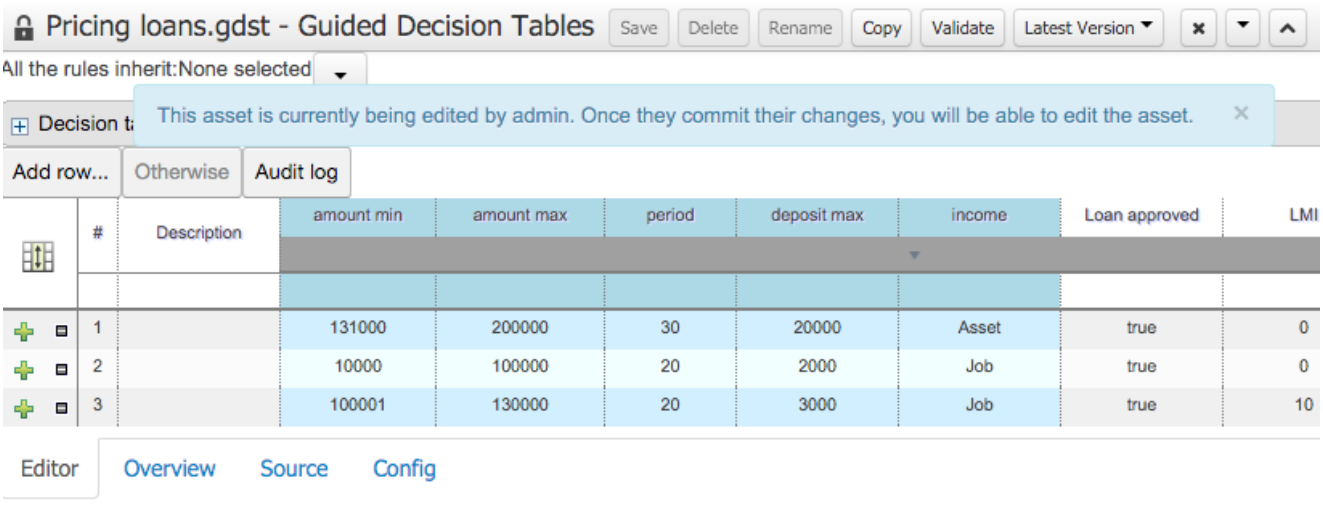


Figure 18.24. The Asset Editor - Locked assets cannot be edited by other users

Figure 18.23. The Asset Editor - Editor tab

- Overview
  - A : General information about the asset and the asset's description.

"Type:" The format name of the type of Asset.

"Description:" Description for the asset.

"Used in projects:" Names the projects where this rule is used.

"Last Modified:" Who made the last change and when.

"Created on:" Who created the asset and when.

- B : Version history for the asset. Selecting a version loads the selected version into this editor.
- C : Meta data (from the "Dublin Core" standard)
- D : Comments regarding the development of the Asset can be recorded here.

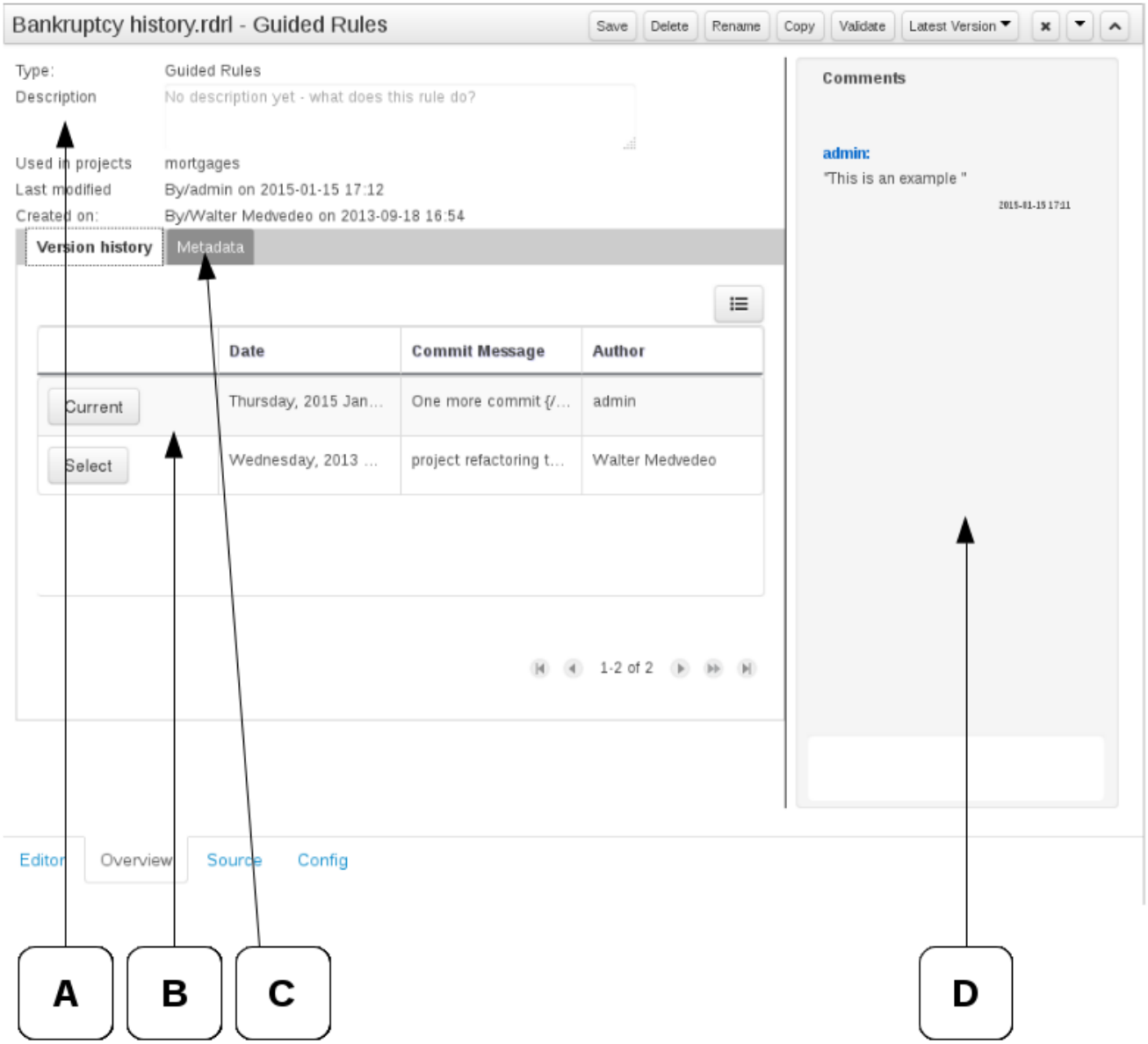


Figure 18.25. The Asset Editor - Overview tab

- Metadata
  - A : Meta data:-

    "Tags:" A tagging system for grouping the assets.

    "Note:" A comment made when the Asset was last updated (i.e. why a change was made)

    "URI:" URI to the asset inside the Git repository.

    "Subject/Type/External link/Source" : Other miscellaneous meta data for the Asset.

"Lock status" : Shows the lock status of the asset and, if locked, allows to force unlocking the asset.

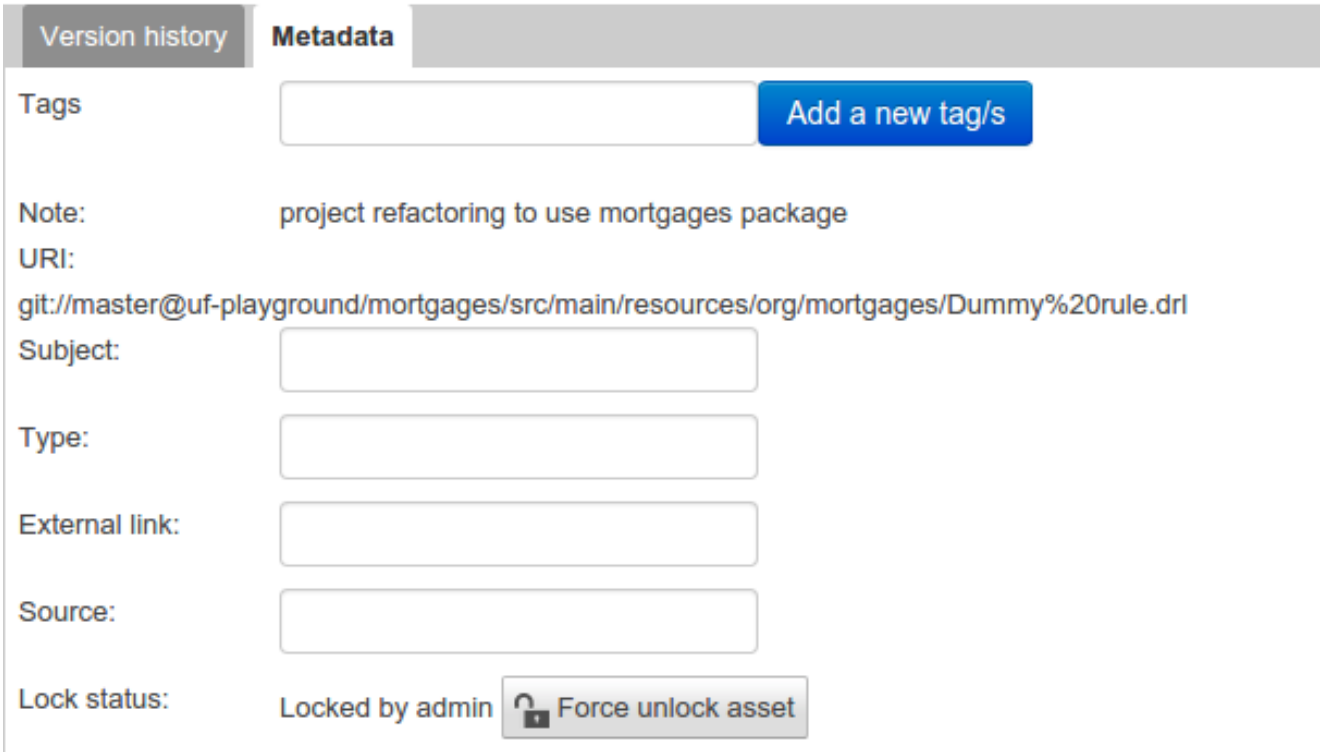


Figure 18.26. The Metadata tab

18.7.3. Tags Editor

Tags allow assets to be labelled with any number of tags that you define. These tags can be used to filter assets on the Project Explorer enabling "Tag filtering".

18.7.3.1. Creating Tags

To create tags you simply have to write them on the Tags input and press the "Add new Tag/s" button. The Tag Editor allows creating tags one by one or writing more than one separated with a white space.

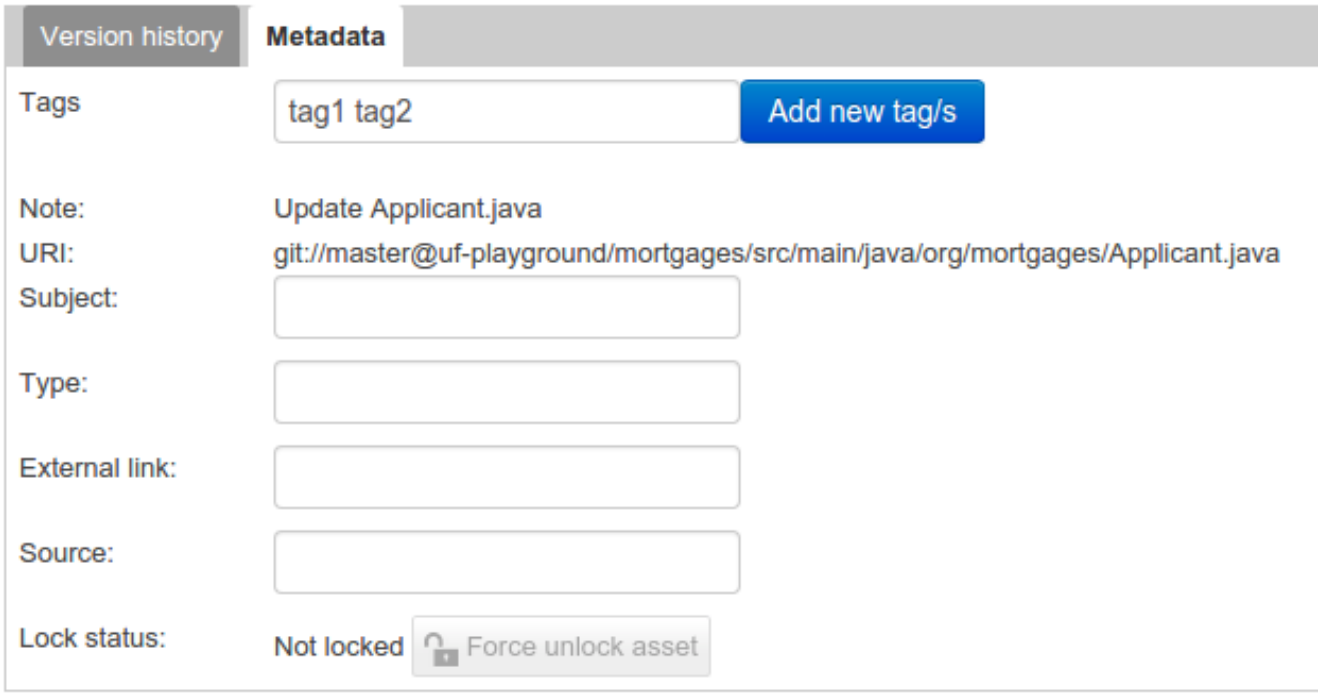


Figure 18.27. Creating Tags

Once you created new Tags they will appear over the Editor allowing you to remove them by pressing on them if you want.

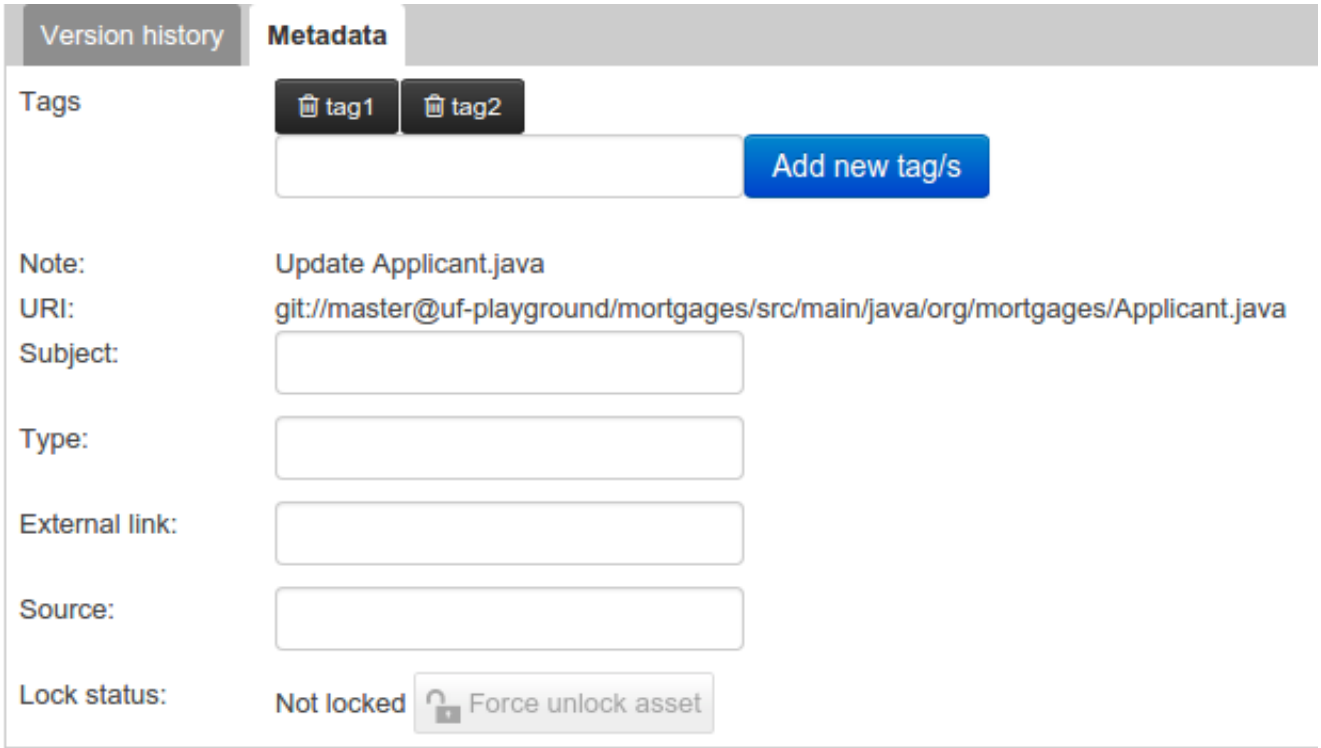


Figure 18.28. Existing Tags

18.7.4. Project Explorer

The Project Explorer provides the ability to browse different Organizational Units, Repositories, Projects and their files.

18.7.4.1. Initial view

The initial view could be empty when first opened.

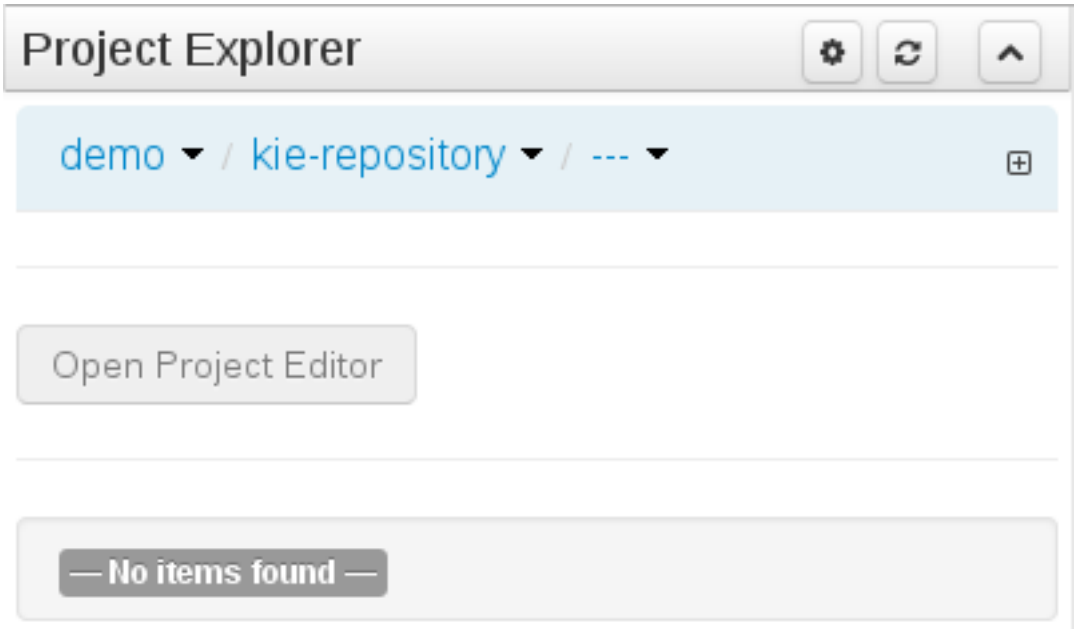


Figure 18.29. An empty initial view

The user may have to select an Organizational Unit, Repository and Project from the drop-down boxes.

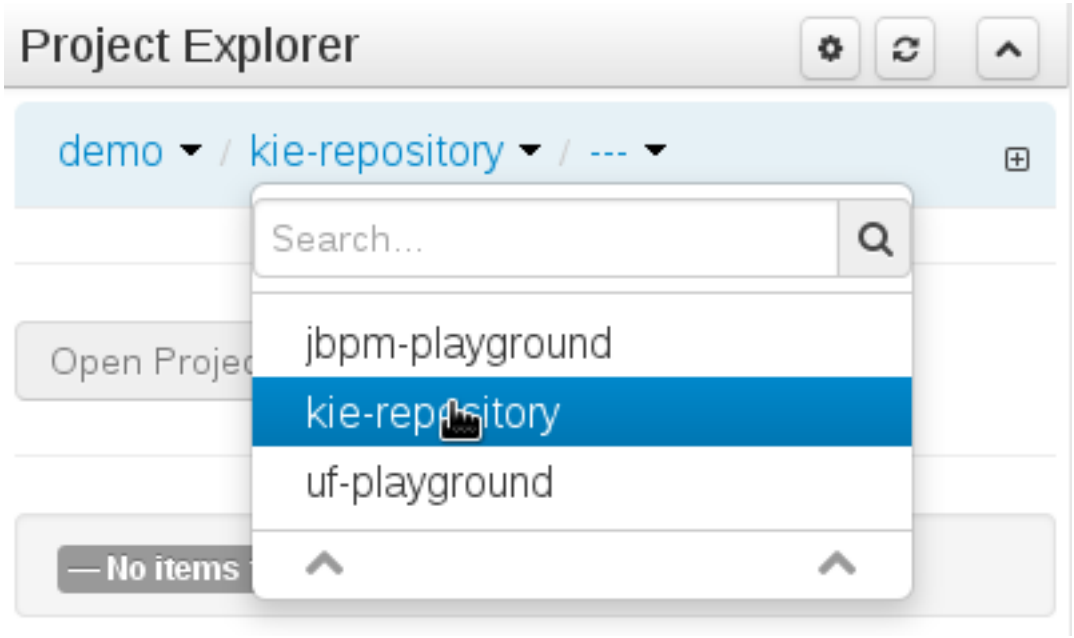


Figure 18.30. Selecting a repository

The default configuration hides Package details from view.

In order to reveal packages click on the icon as indicated in the following screen-shot.

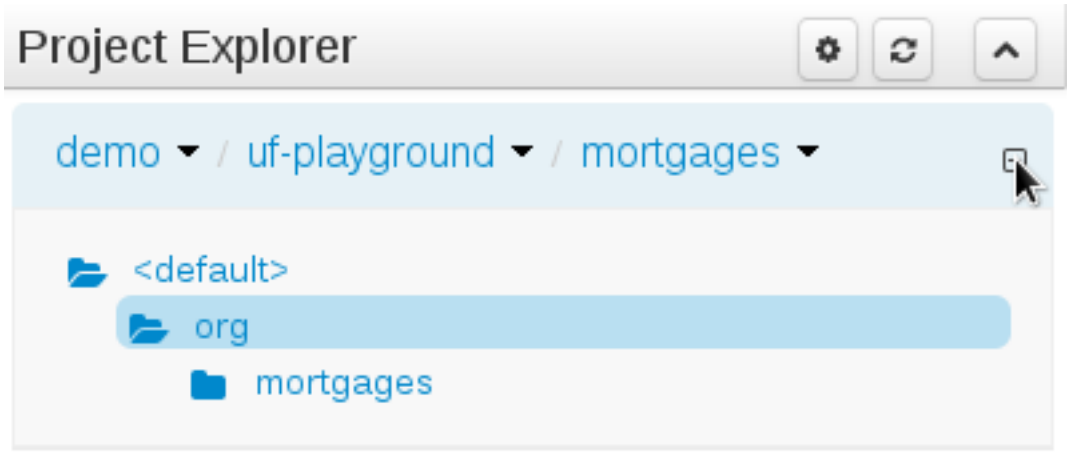


Figure 18.31. Showing packages

After a suitable combination of Organizational Unit, Repository, Project and Package have been selected the Project Explorer will show the contents. The exact combination of selections depends wholly on the structures defined within the Workbench installation and projects. Each section contains groups of related files. If a file is currently being edited by another user, a lock symbol will be displayed in front of the file name. The symbol is blue in case the lock is owned by the currently authenticated user, otherwise black. Moving the mouse pointer over the lock symbol will display a tooltip providing the name of the user who is currently editing the file (and therefore owning the lock). To learn more about locking see Section 18.7.2, “Asset Editor” for details.

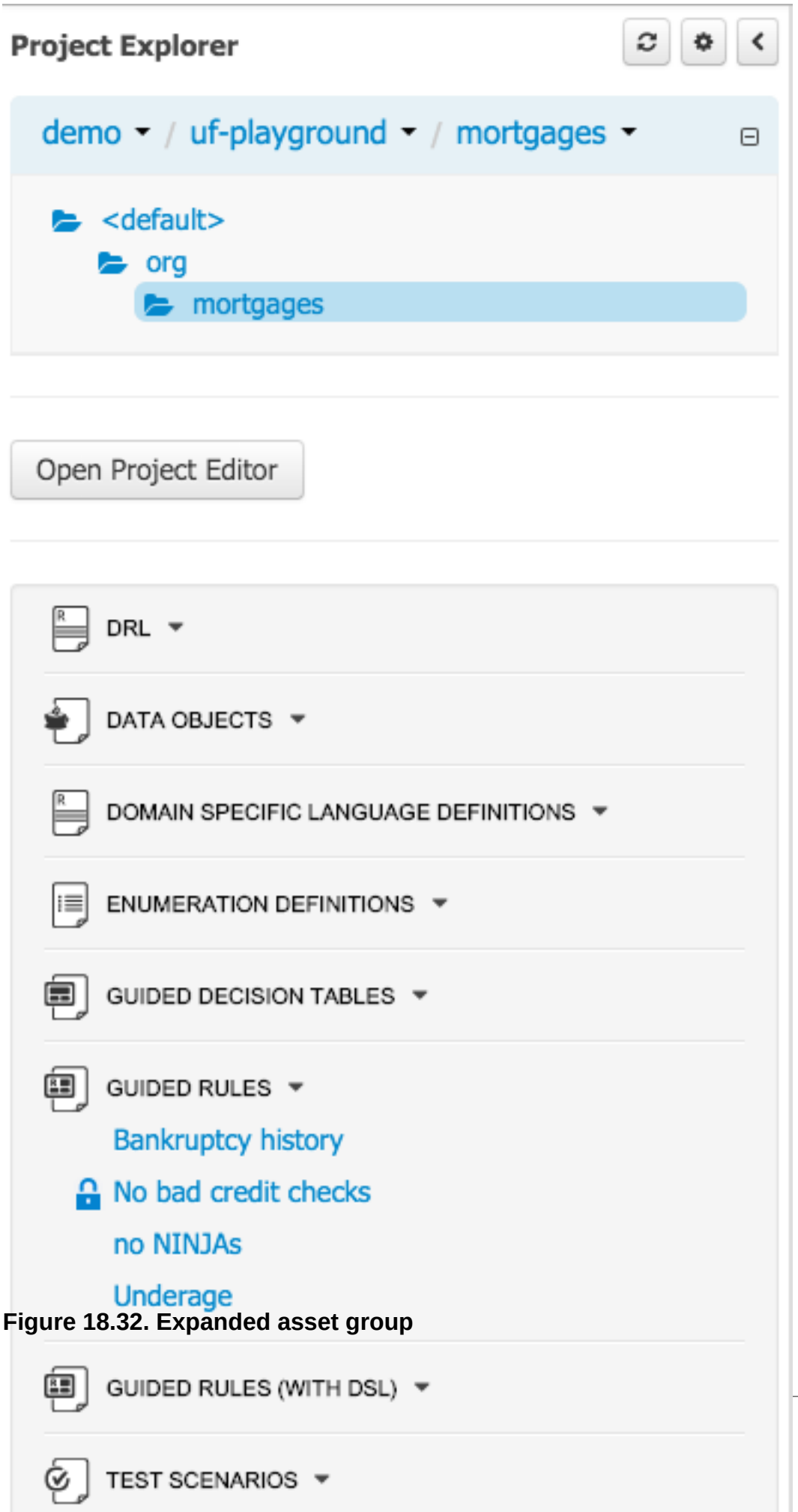


Figure 18.32. Expanded asset group

18.7.4.2. Different views

Project Explorer supports multiple views.

- Project View

  A simplified view of the underlying project structure. Certain system files are hidden from view.

- Repository View

  A complete view of the underlying project structure including all files; either user-defined or system generated.

Views can be selected by clicking on the icon within the Project Explorer, as shown below.

Both Project and Repository Views can be further refined by selecting either "Show as Folders" or "Show as Links".

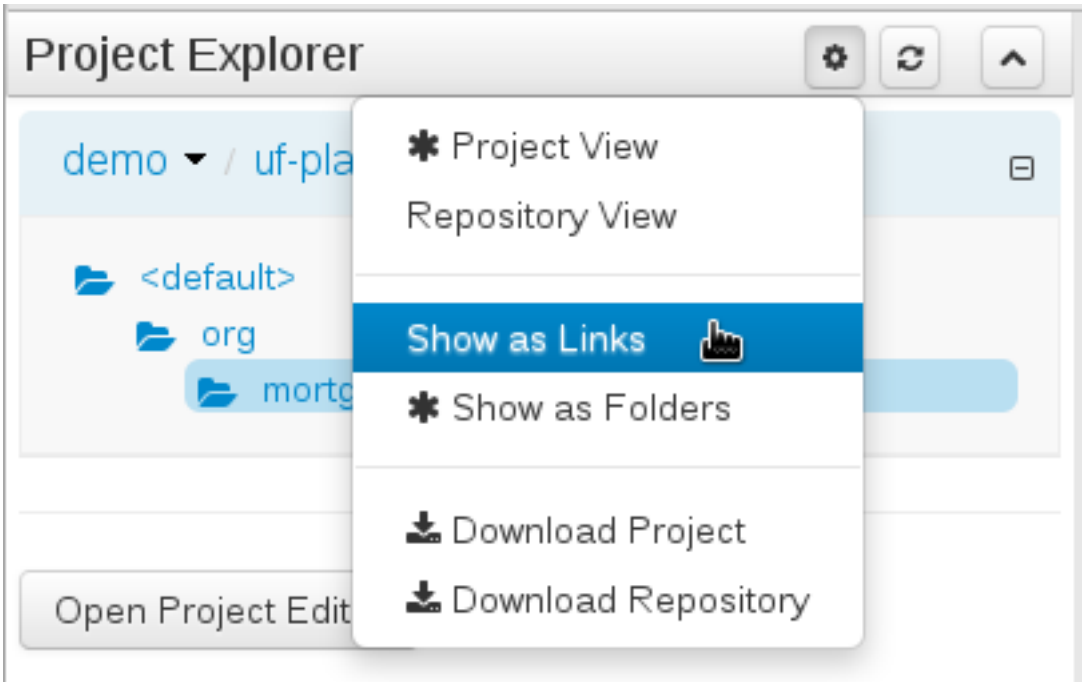


Figure 18.33. Switching view

18.7.4.2.1. Project View examples

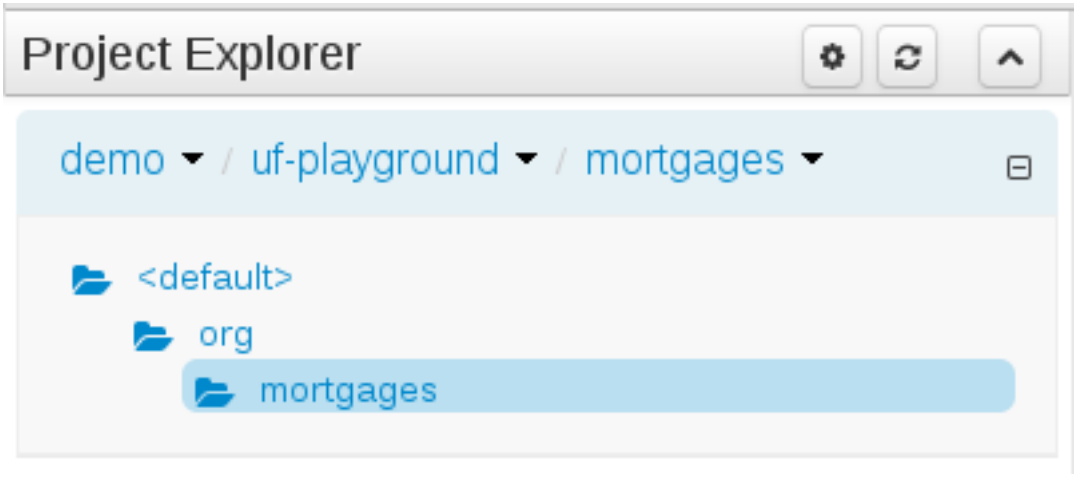


Figure 18.34. Project View - Folders

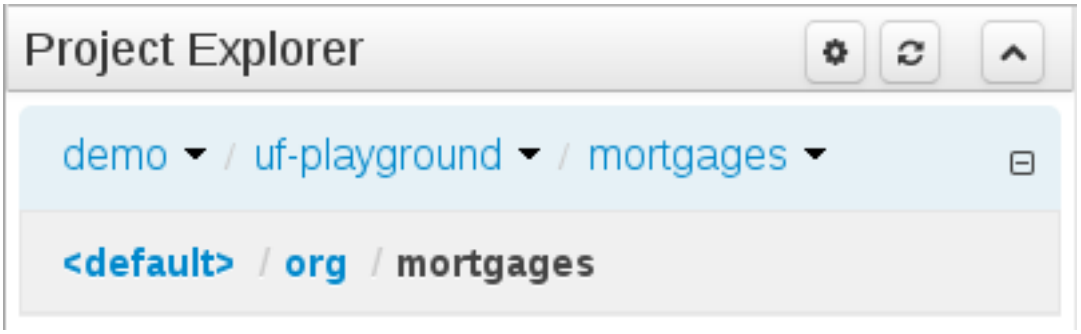


Figure 18.35. Project View - Links

18.7.4.2.2. Repository View examples

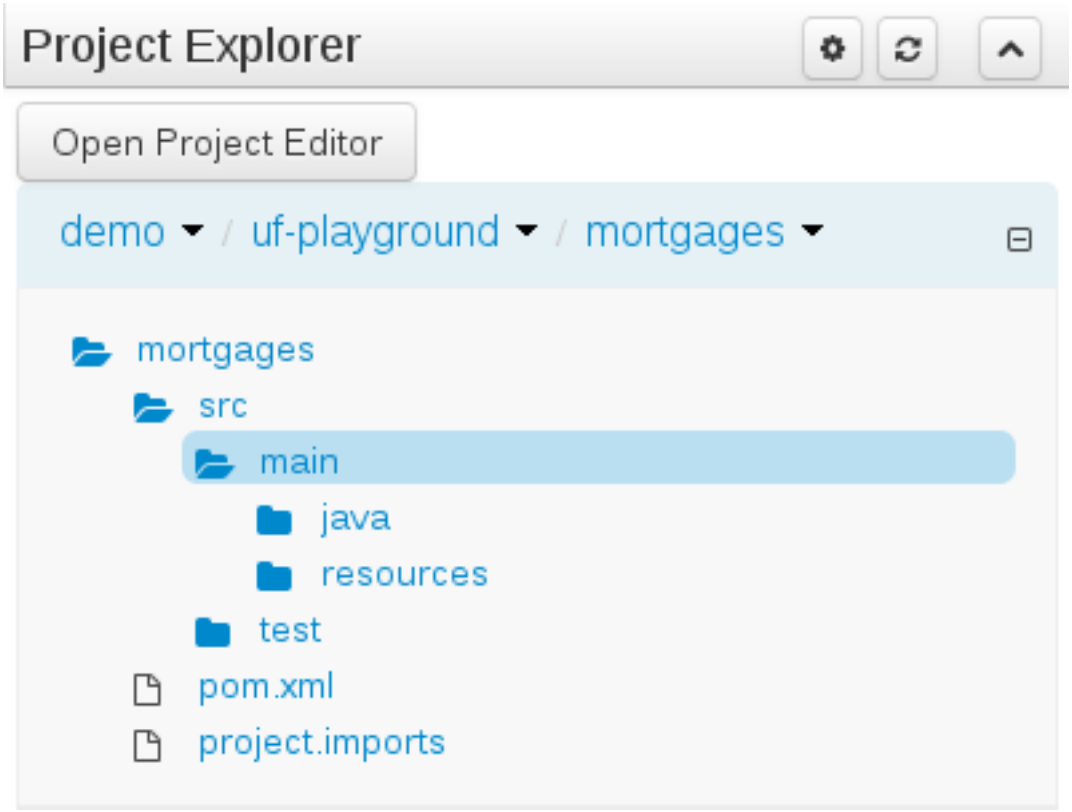


Figure 18.36. Repository View - Folders

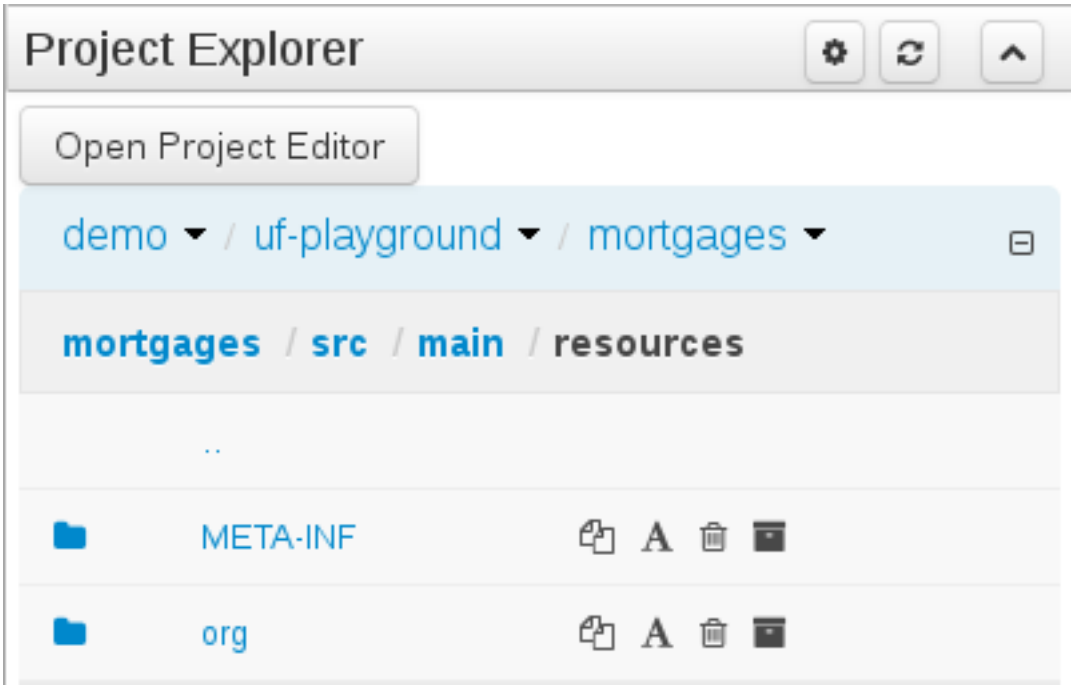


Figure 18.37. Repository View - Links

18.7.4.3. Download Project or Repository

Download Download and Download Repository make it possible to download the project or repository as a zip file.

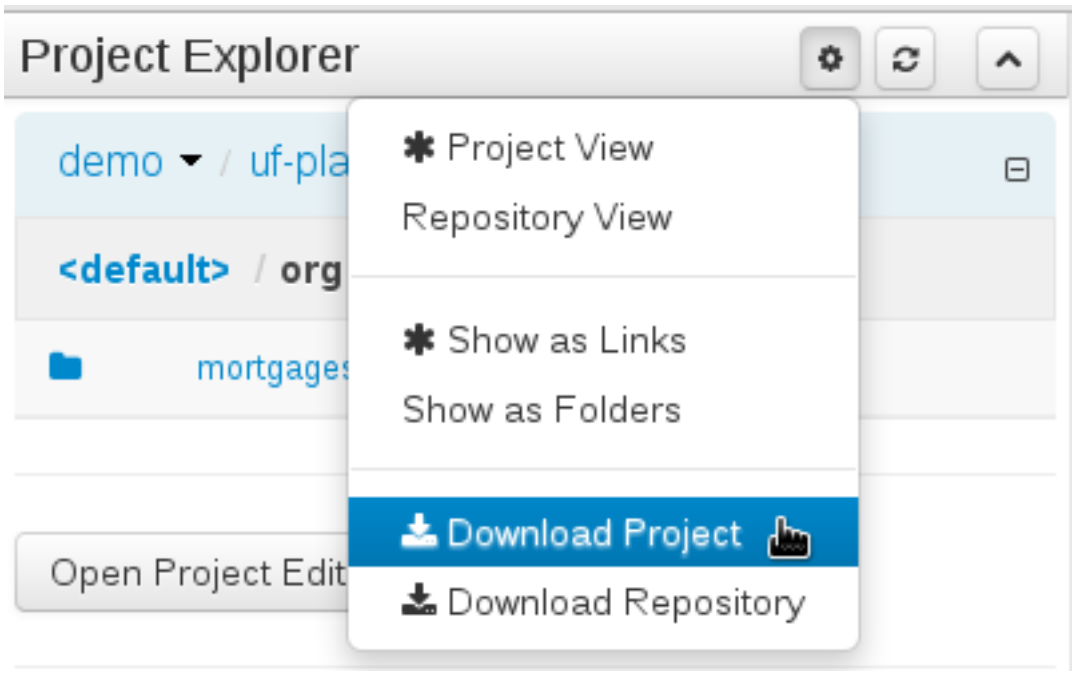


Figure 18.38. Repository and Project Downloads

18.7.4.4. Branch selector

A branch selector will be visible if the repository has more than a single branch.

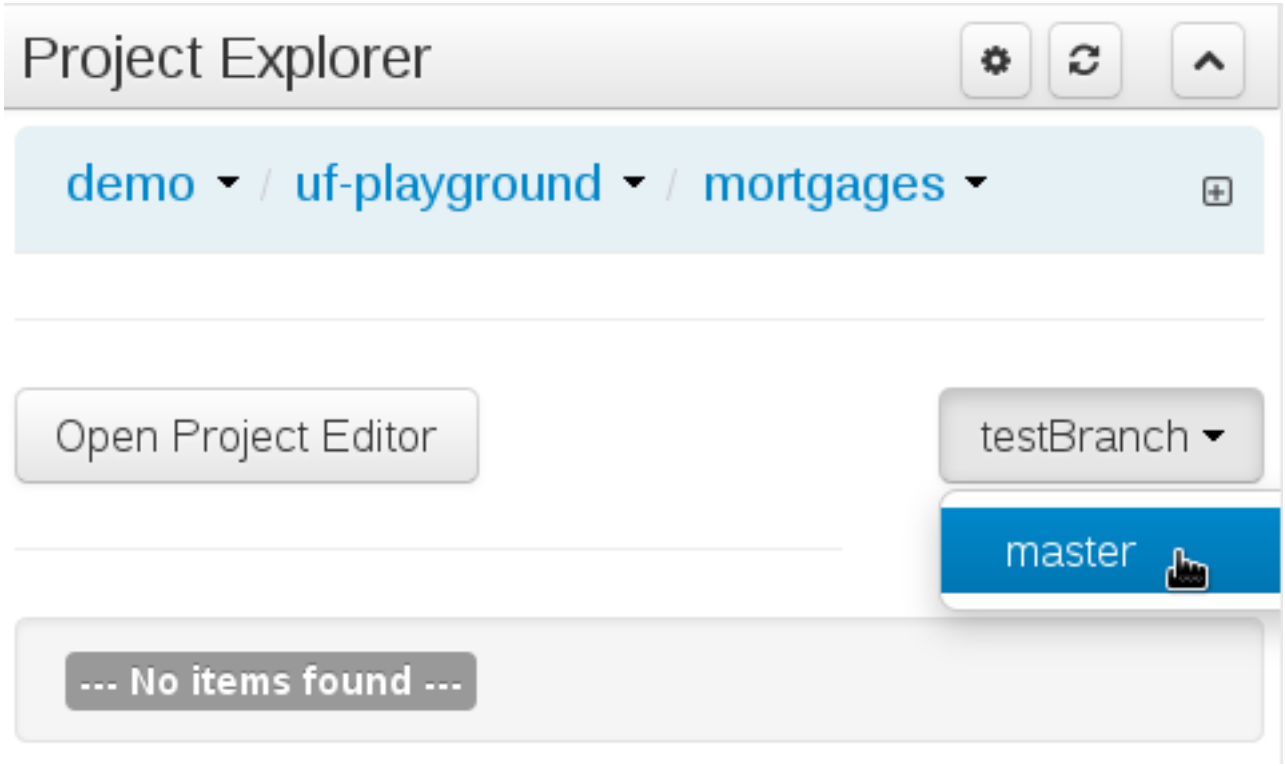


Figure 18.39. Branch selector

18.7.4.5. Filtering by Tag

To make easy view the elements on packages that contain a lot of assets, is possible to enabling the Tag filter, whichs allows you to filter the assets by their tags.

To see how to add tags to an asset look at: Section 18.7.3, “Tags Editor”

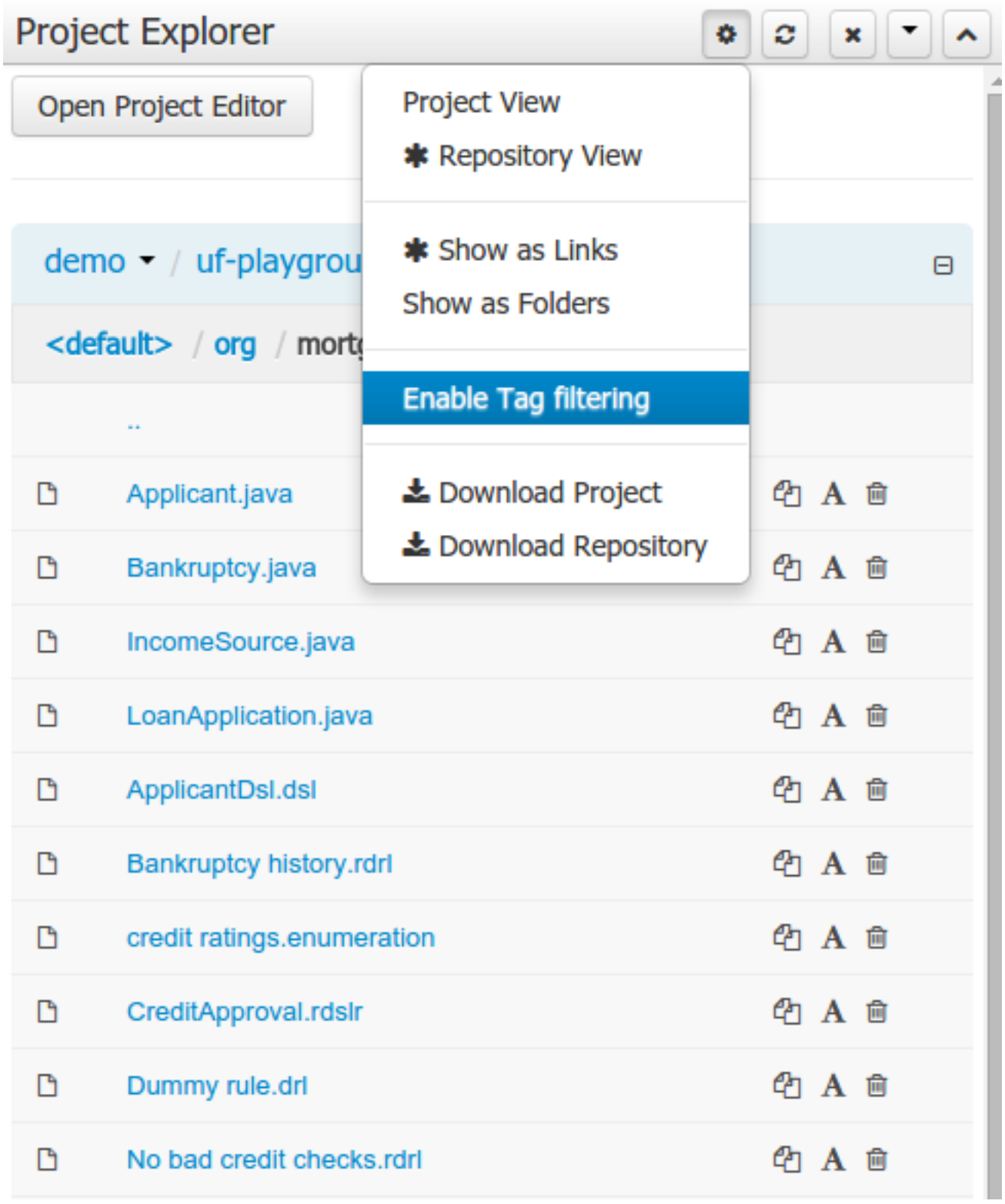


Figure 18.40. Enabling Filter by Tag

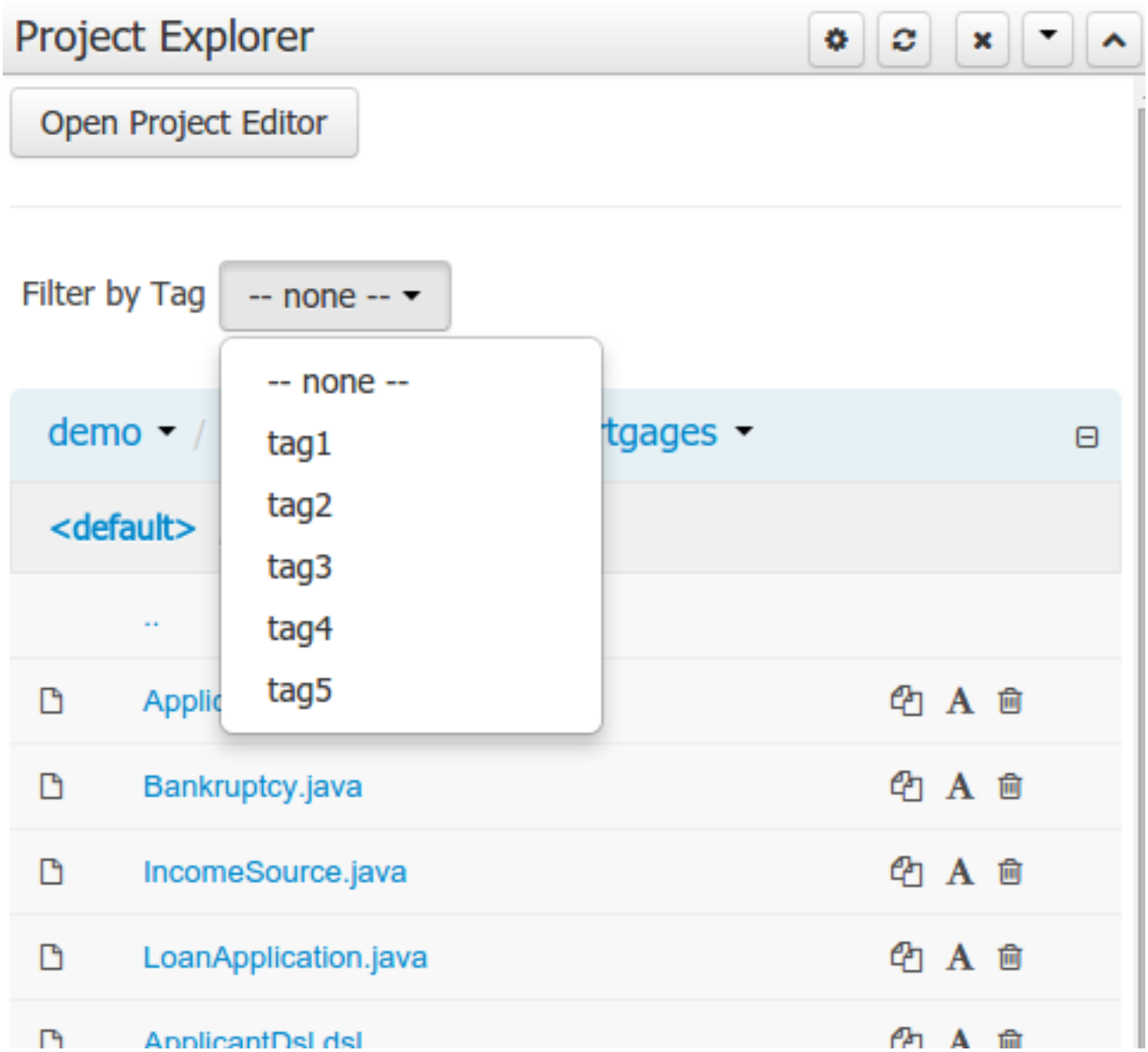


Figure 18.41. Filter by Tag

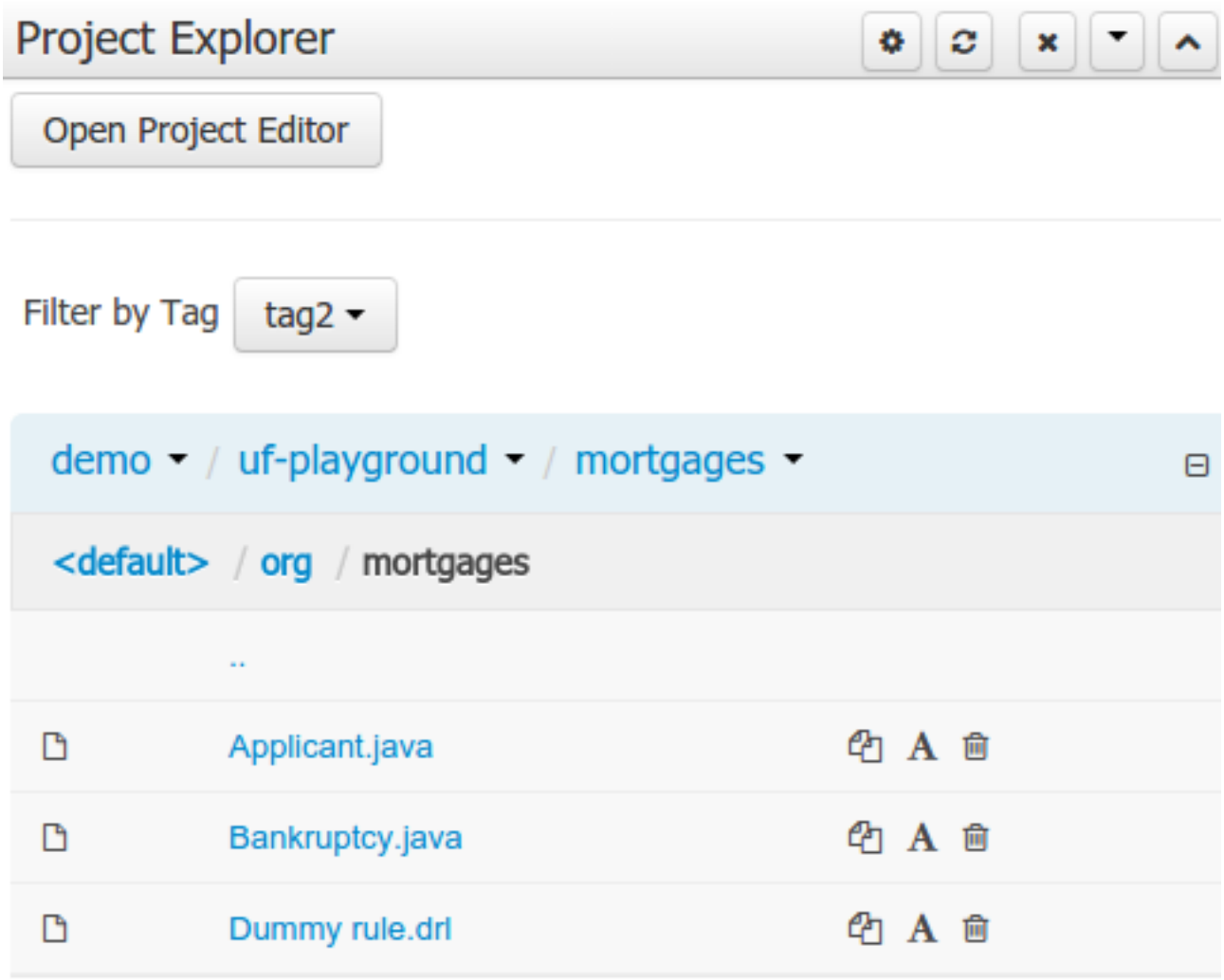


Figure 18.42. Filtering by Tag

18.7.4.6. Copy, Rename, Delete and Download Actions

Copy, rename and delete actions are available on *Links* mode, for packages (in of Project View) and for files and directories as well (in Repository View). Download action is available for directories. Download downloads the selected the selected directory as a zip file.

- A : Copy
- B : Rename
- C : Delete
- D : Download

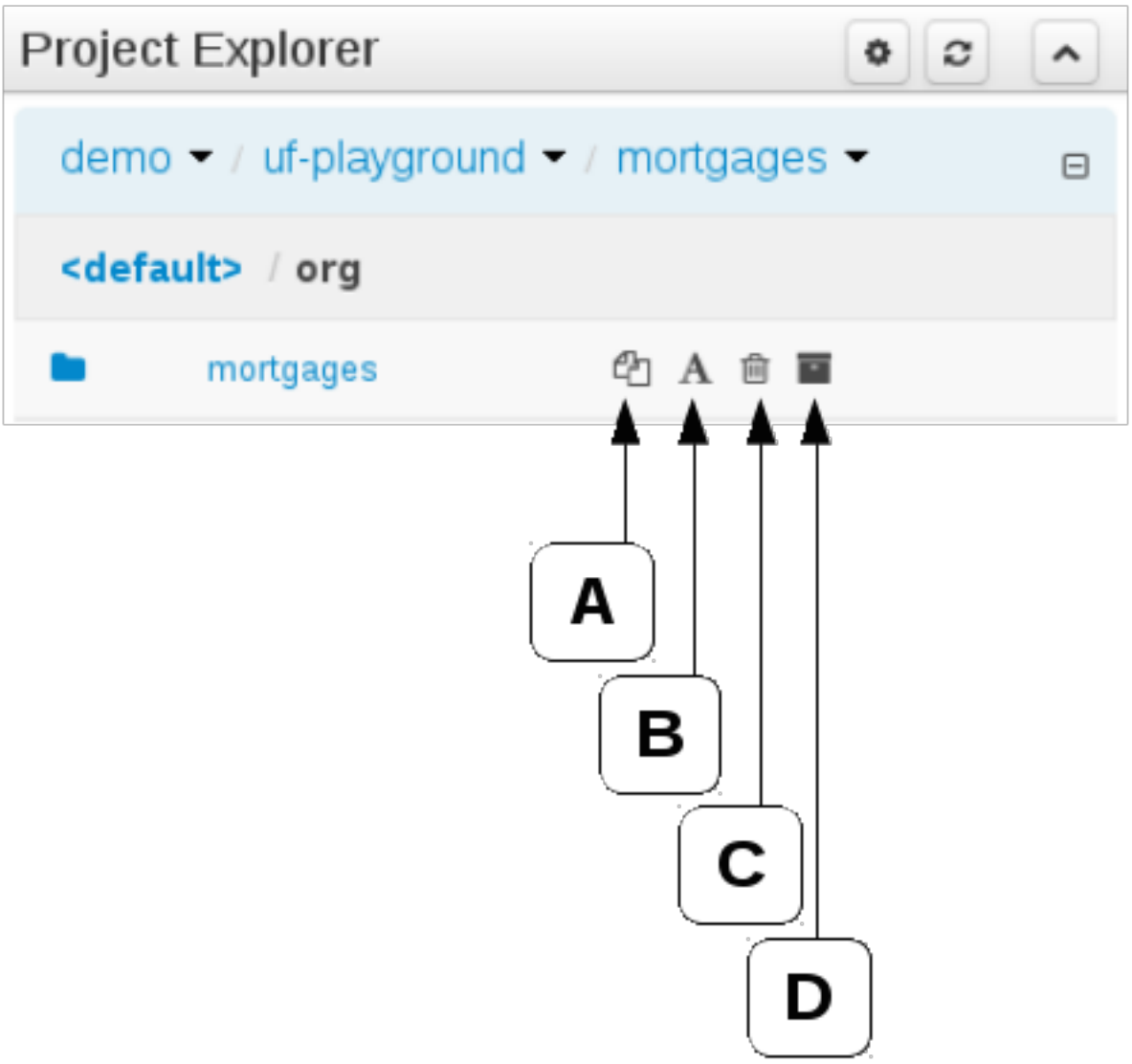


Figure 18.43. Project View - Package actions

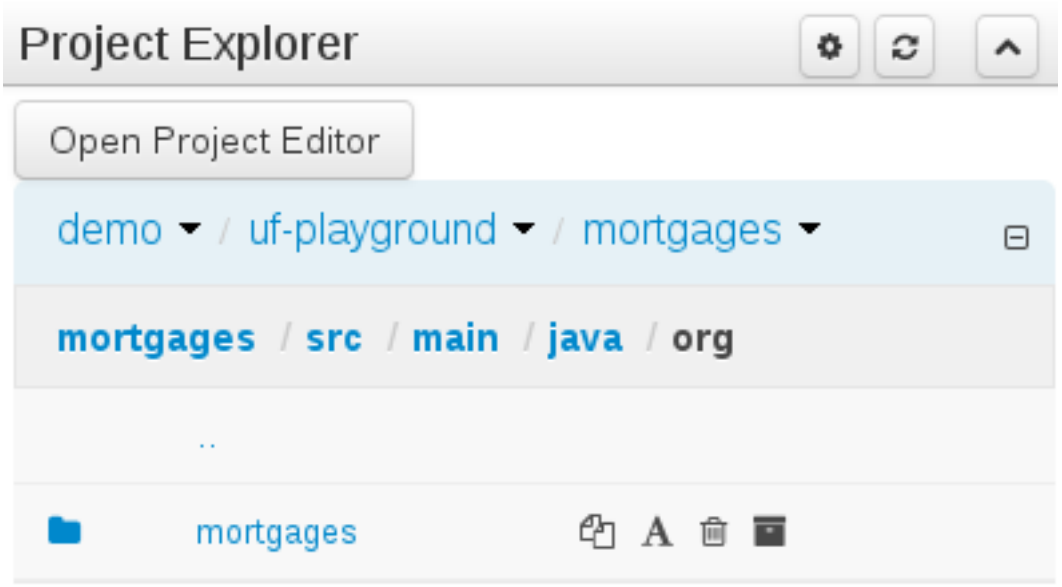


Figure 18.44. Repository View - Files and directories actions

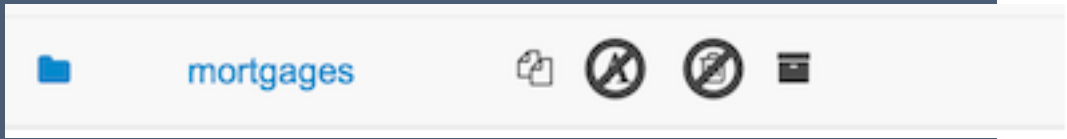
Warning

Workbench roadmap includes a refactoring and an impact analyses tools, but currenctly doesn't have it. Until both tools are provided make sure that your changes (copy/rename/delete) on packages, files or directories doesn't have a major impact on your project.

In cases that your change had an unexcepcted impact, Workbench allows you to restore your repository using the Repository editor.

Important

Files locked by other users as well as directories that contain such files cannot be renamed or deleted until the corresponding locks are released. If that is the case the rename and delete symbols will be deactivated. To learn more about locking see Section 18.7.2, “Asset Editor” for details.



18.7.5. Project Editor

The Project Editor screen can be accessed from Project Explorer. Project Editor shows the settings for the currently active project.

Unlike most of the workbench editors, project editor edits more than one file. Showing everything that is needed for configuring the KIE project in one place.

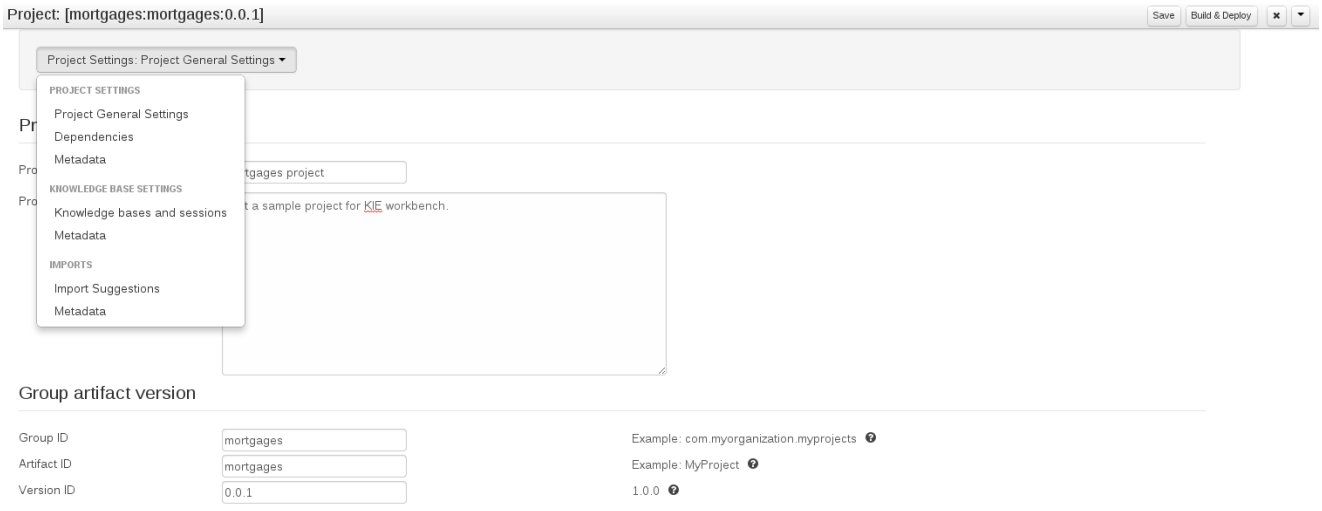


Figure 18.45. Project Screen and the different views

18.7.5.1. Build & Deploy

Build & Depoy builds the current project and deploys the KJAR into the workbench internal Maven repository.

18.7.5.2. Project Settings

Project Settings edits the pom.xml file used by Maven.

18.7.5.2.1. Project General Settings

General settings provide tools for project name and GAV-data (Group, Artifact, Version). GAV values are used as identifiers to differentiate projects and versions of the same project.

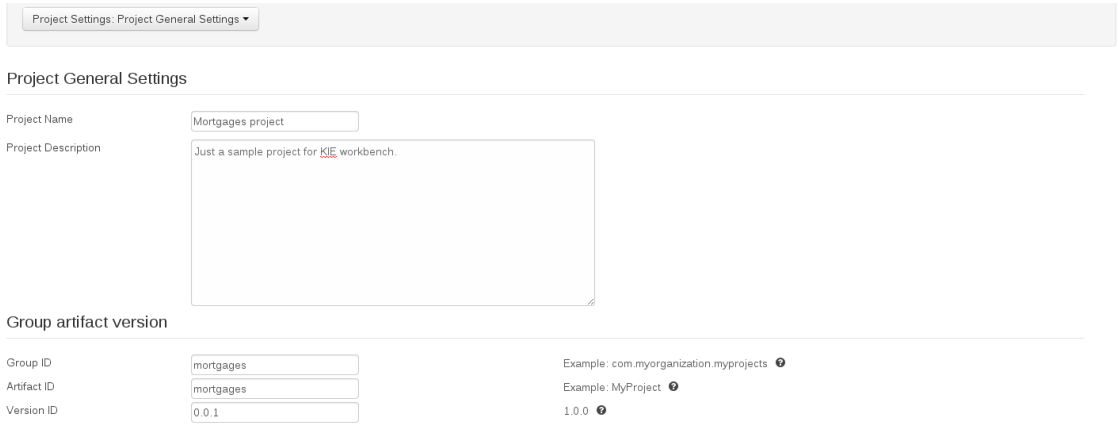


Figure 18.46. Project Settings

18.7.5.2.2. Dependencies

The project may have any number of either internal or external dependencies. Dependency is a project that has been built and deployed to a Maven repository. Internal dependencies are projects build and deployed in the same workbench as the project. External dependencies are retrieved from repositories outside of the current workbench. Each dependency uses the GAV-values to specify the project name and version that is used by the project.

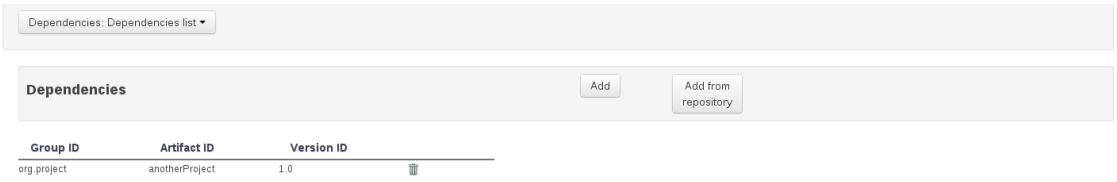


Figure 18.47. Dependencies

18.7.5.2.3. Metadata

Metadata for the pom.xml file.

18.7.5.3. Knowledge Base Settings

Knowledge Base Settings edits the kmodule.xml file used by Drools.

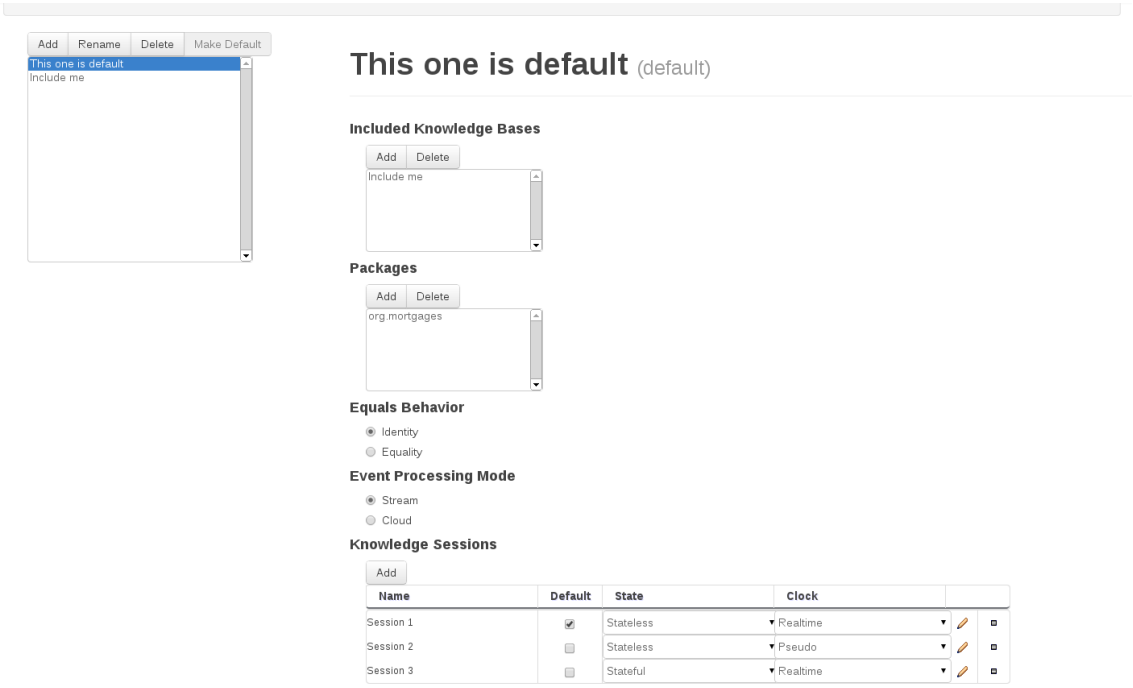


Figure 18.48. Knowledge Base Settings

Note

For more information about the Knowledge Base properties, check the Drools Expert documentation for kmodule.xml.

18.7.5.3.1. Knowledge bases and sessions

Knowledge bases and sessions lists the knowledge bases and the knowledge sessions specified for the project.

18.7.5.3.1.1. Knowledge base list

Lists all the knowledge bases by name. Only one knowledge base can be set as default.

18.7.5.3.1.2. Knowledge base properties

Knowledge base can include other knowledge bases. The models, rules and any other content in the included knowledge base will be visible and usable by the currently selected knowledge base.

Rules and models are stored in packages. The packages property specifies what packages are included into this knowledge base.

Equals behavior is explained in the Drools Expert part of the documentation.

Event processing mode is explained in the Drools Fusion part of the documentation.

18.7.5.3.1.3. Knowledge sessions

The table lists all the knowledge sessions in the selected knowledge base. There can be only one default of each type. The types are stateless and stateful. Clicking the pen-icon opens a popup that shows more properties for the knowledge session.

18.7.5.3.2. Metadata

Metadata for the kmodule.xml

18.7.5.4. Imports

Settings edits the project.imports file used by the workbench editors.

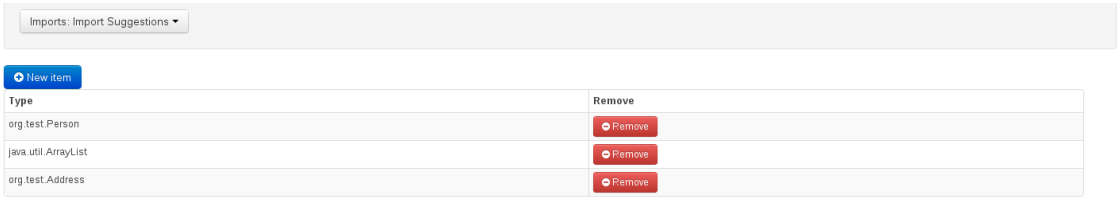


Figure 18.49. Imports

18.7.5.4.1. Import Suggestions

Import Suggestions lists imports that are used as suggestions when using the guided editors the workbench has. Making it easier to work with the workbench, as there is no need to type each import in each file that uses the import.

Note

Unlike in the previous version of Guvnor. The imports listed in the import suggestions are not automatically added into the knowledge base or into the packages of the workbench. Each import needs to be explicitly added into each file.

18.7.5.4.2. Metadata

Metadata for the project.imports file.

18.7.6. Validation

The Workbench provides a common and consistent service for users to understand whether files authored within the environment are valid.

18.7.6.1. Problem Panel

The Problems Panel shows real-time validation results of assets within a Project.

When a Project is selected from the Project Explorer the Problems Panel will refresh with validation results of the chosen Project.

When files are created, saved or deleted the Problems Panel content will update to show either new validation errors, or remove existing if a file was deleted.

Here an invalid DRL file has been created and saved.

The Problems Panel shows the validation errors.

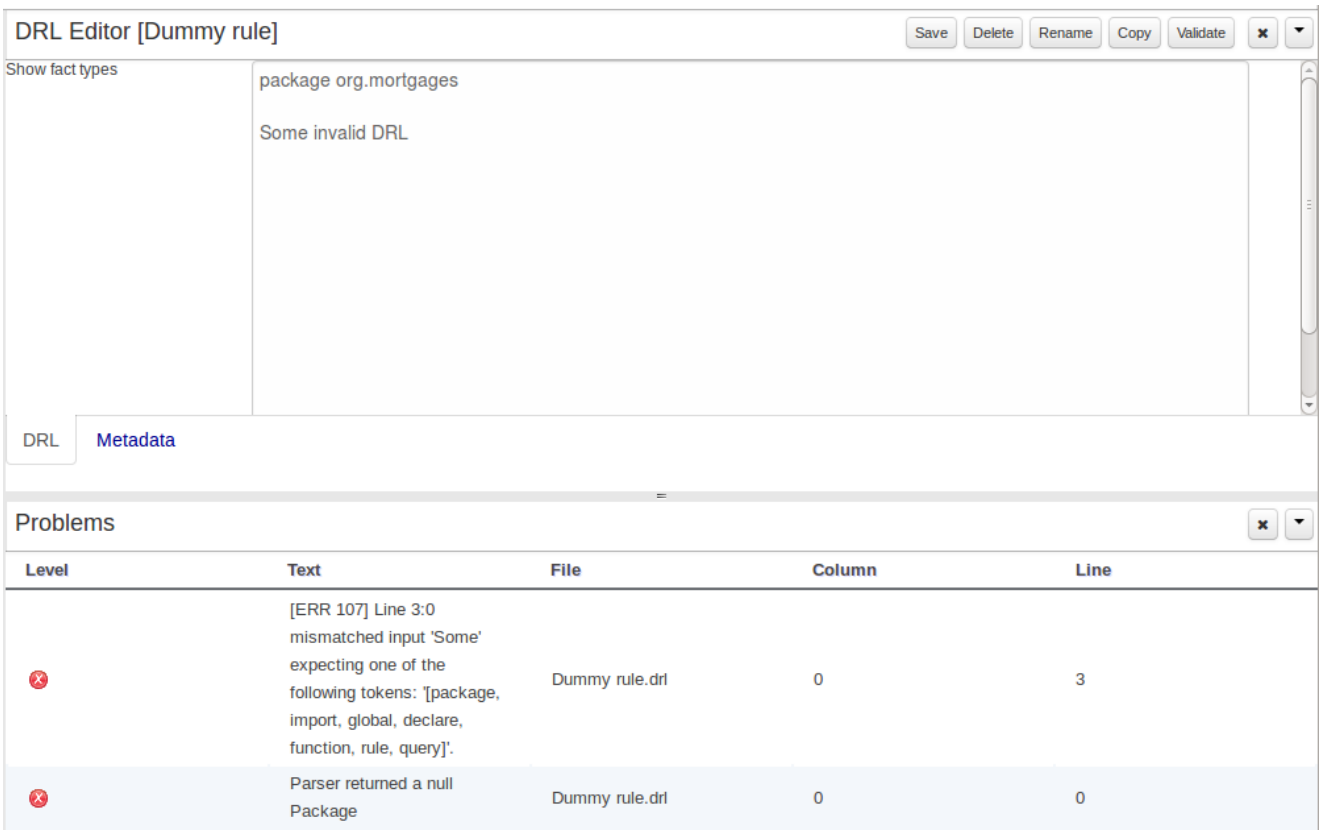


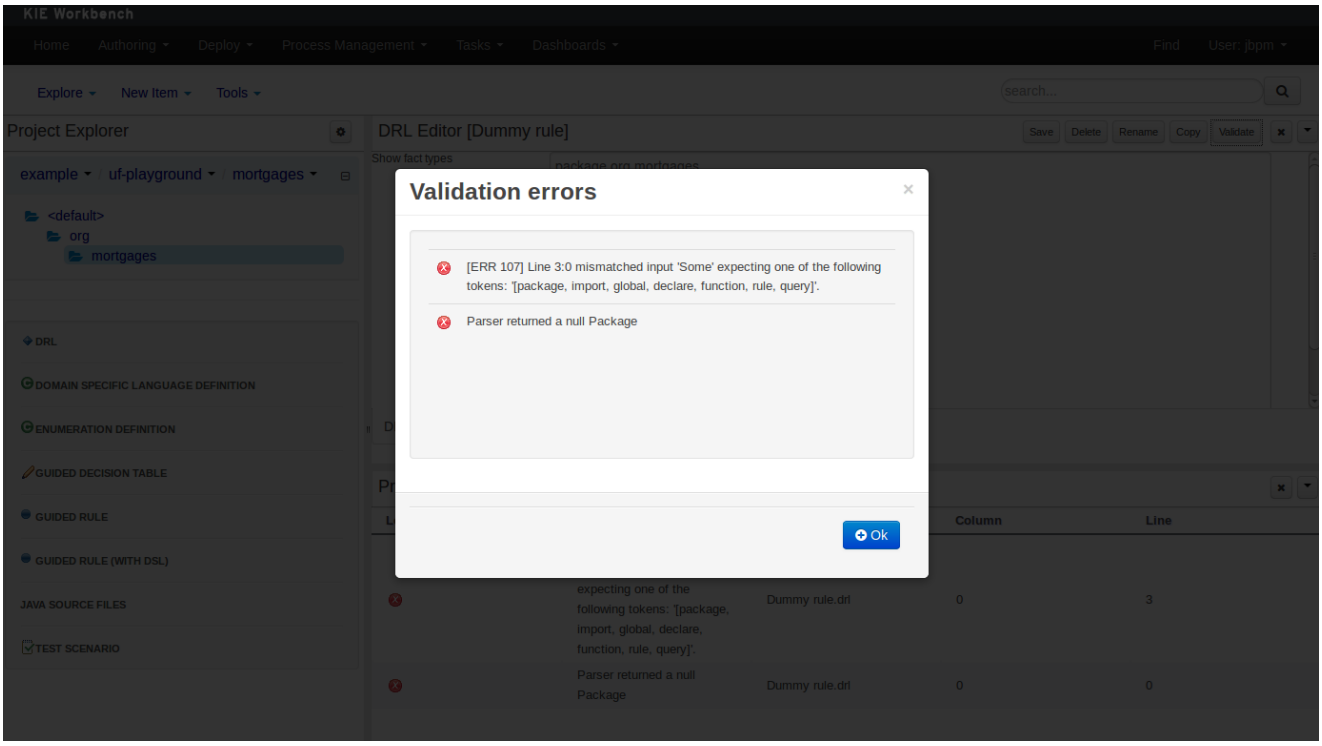
Figure 18.50. The Problems Panel

18.7.6.2. On demand validation

It is not always desirable to save a file in order to determine whether it is in a valid state.

All of the file editors provide the ability to validate the content before it is saved.

Clicking on the 'Validate' button shows validation errors, if any.



18.7.7. Data Modeller

18.7.7.1. First steps to create a data model

By default, a data model is always constrained to the context of a project. For the purpose of this tutorial, we will assume that a correctly configured project already exists and the authoring perspective is open.

To start the creation of a data model inside a project, take the following steps:

1. From the home panel, select the authoring perspective and use the project explorer to browse to the given project.

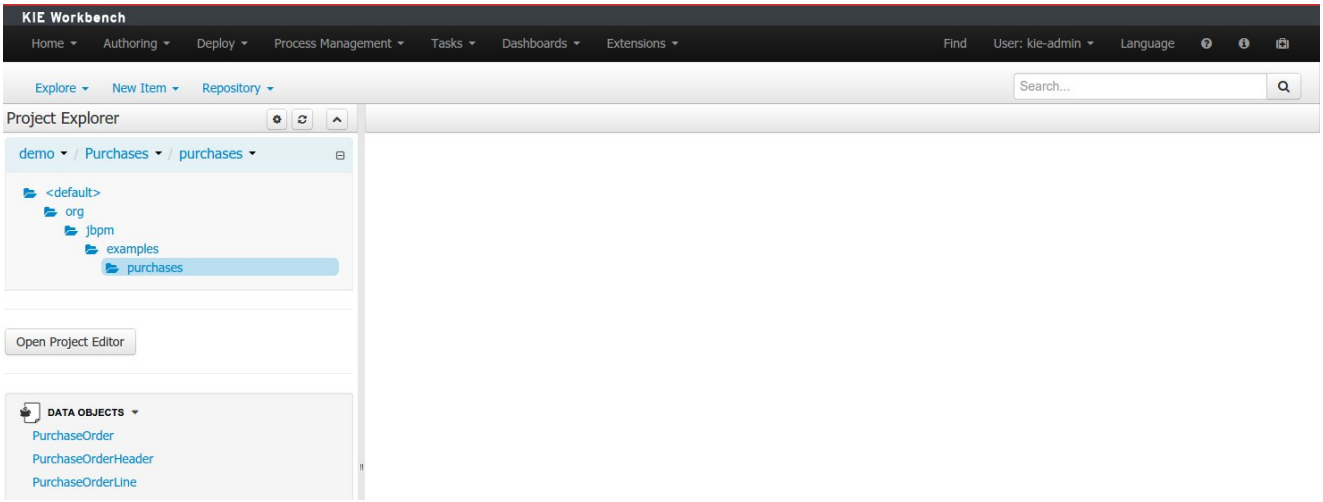


Figure 18.51. Go to authoring perspective and select a project

2. Open the Data Modeller tool by clicking on a Data Object file, or using the "New Item -> Data Object" menu option.

Figure 18.52. Click on a Data Object

This will start up the Data Modeller tool, which has the following general aspect:

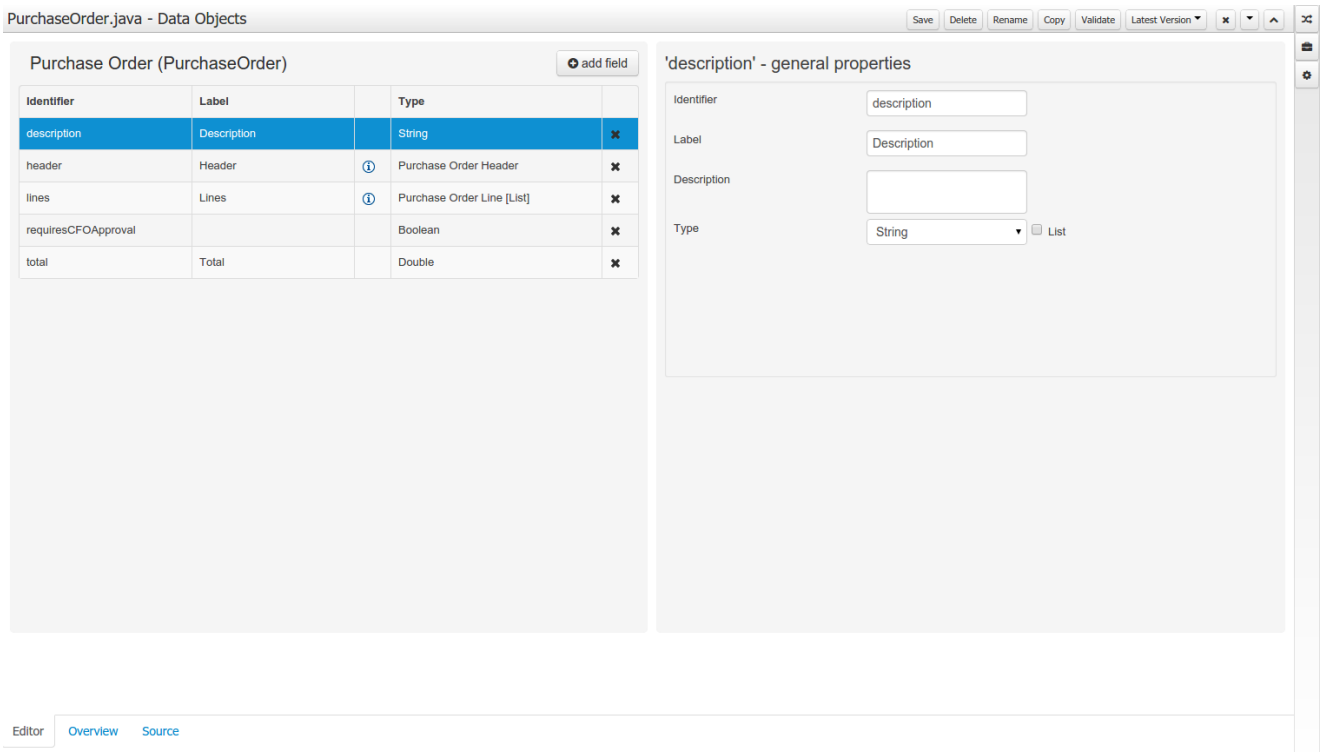


Figure 18.53. Data modeller overview

The "Editor" tab is divided into the following sections:

- The new field section is dedicated to the creation of new fields, and is opened when the "add field" button is pressed.

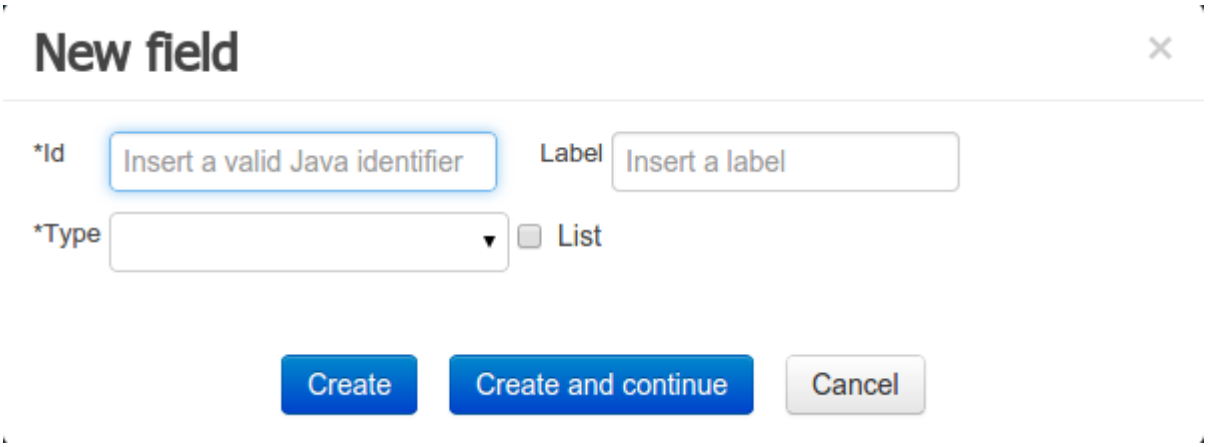


Figure 18.54. New field creation

- The Data Object's "field browser" section displays a list with the data object fields.

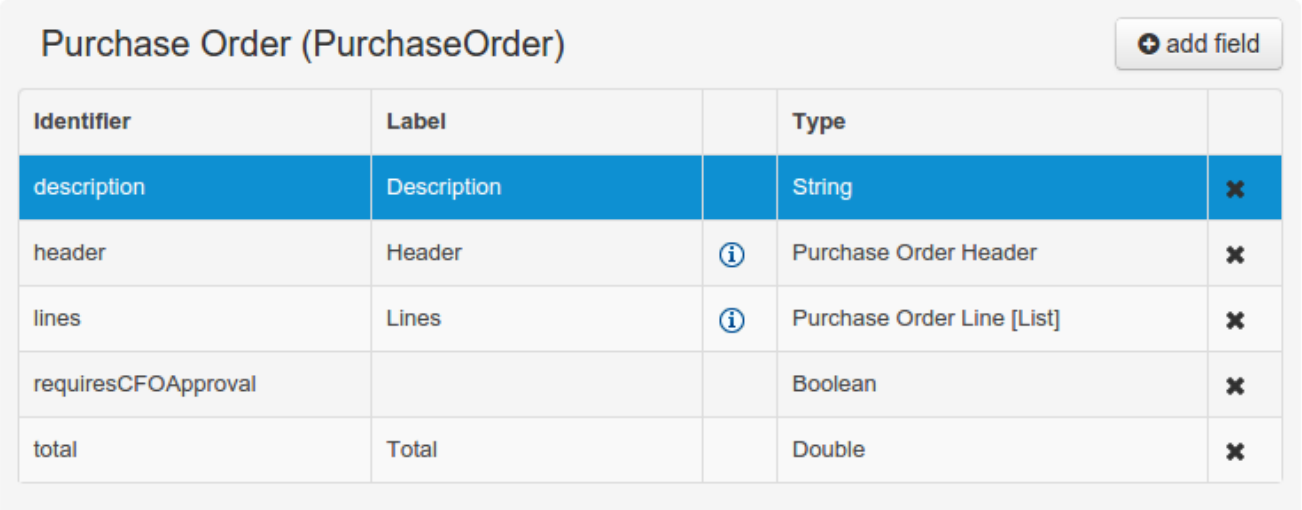


Figure 18.55. The Data Object's field browser

- The "Data Object / Field general properties" section. This is the rightmost section of the Data Modeller editor and visualizes the "Data Object" or "Field" general properties, depending on user selection.

  Data Object general properties can be selected by clicking on the Data Object Selector.

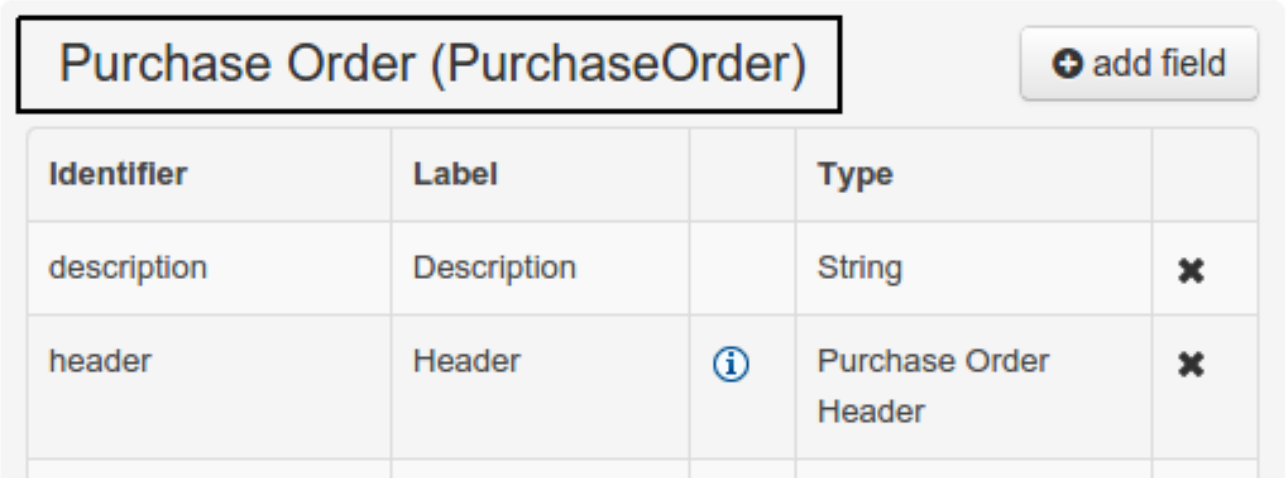


Figure 18.56. Data Object selector

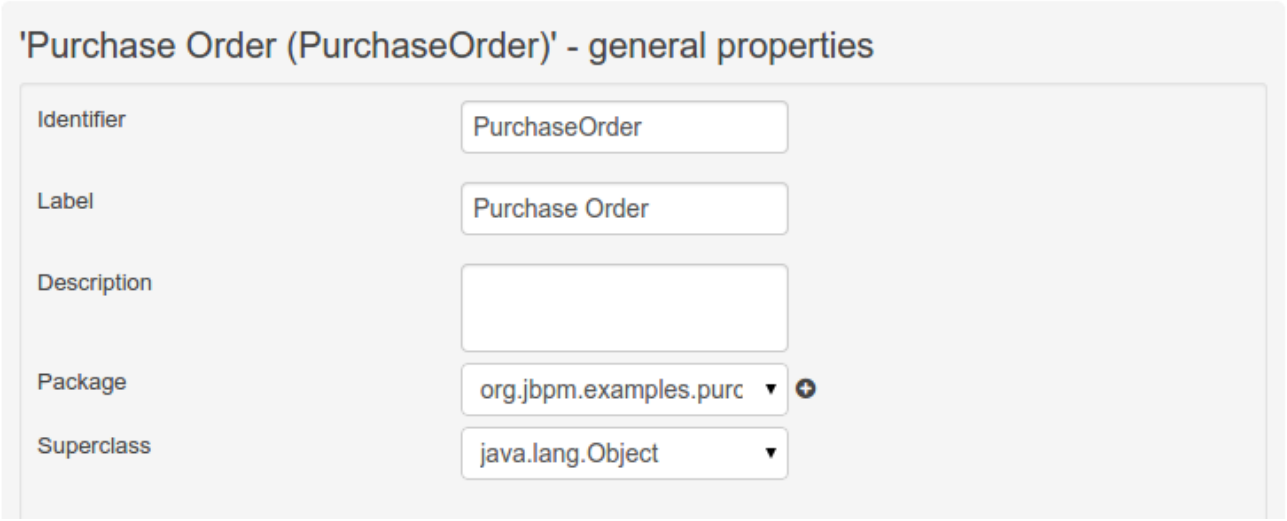


Figure 18.57. Data Object general properties

Field general properties can be selected by clicking on a field.

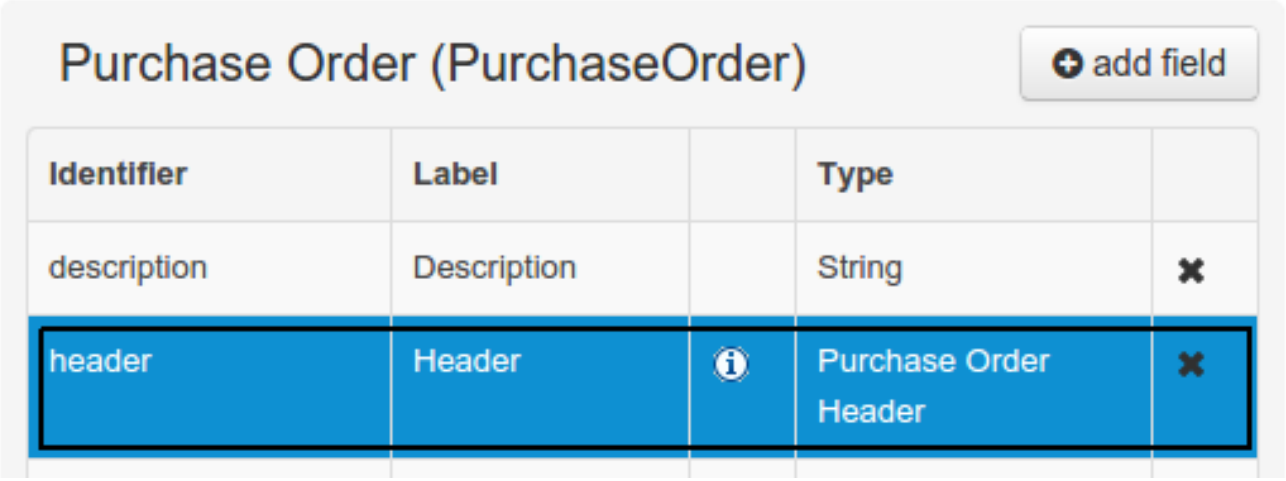


Figure 18.58. Field selector

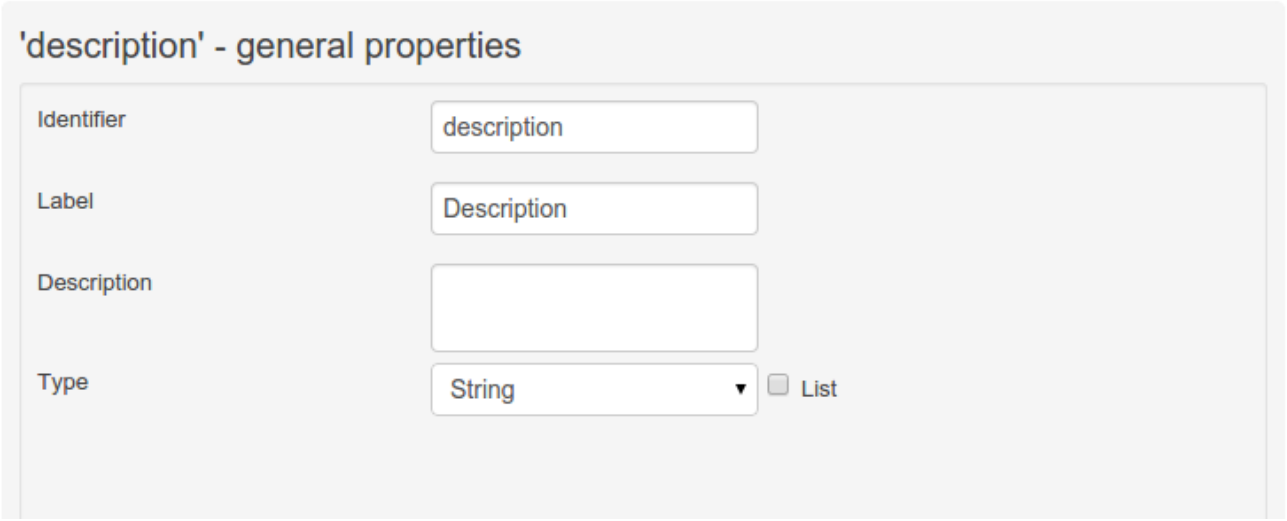


Figure 18.59. Field general properties

- On workbench's right side a new "Tool Bar" is provided that enables the selection of different context sensitive tool windows that will let the user do domain specific configurations. Currently three tool windows are provided for the following domains "Drools & jBPM", "Persistence" and "Advanced" configurations.

Figure 18.60. Data modeller Tool Bar

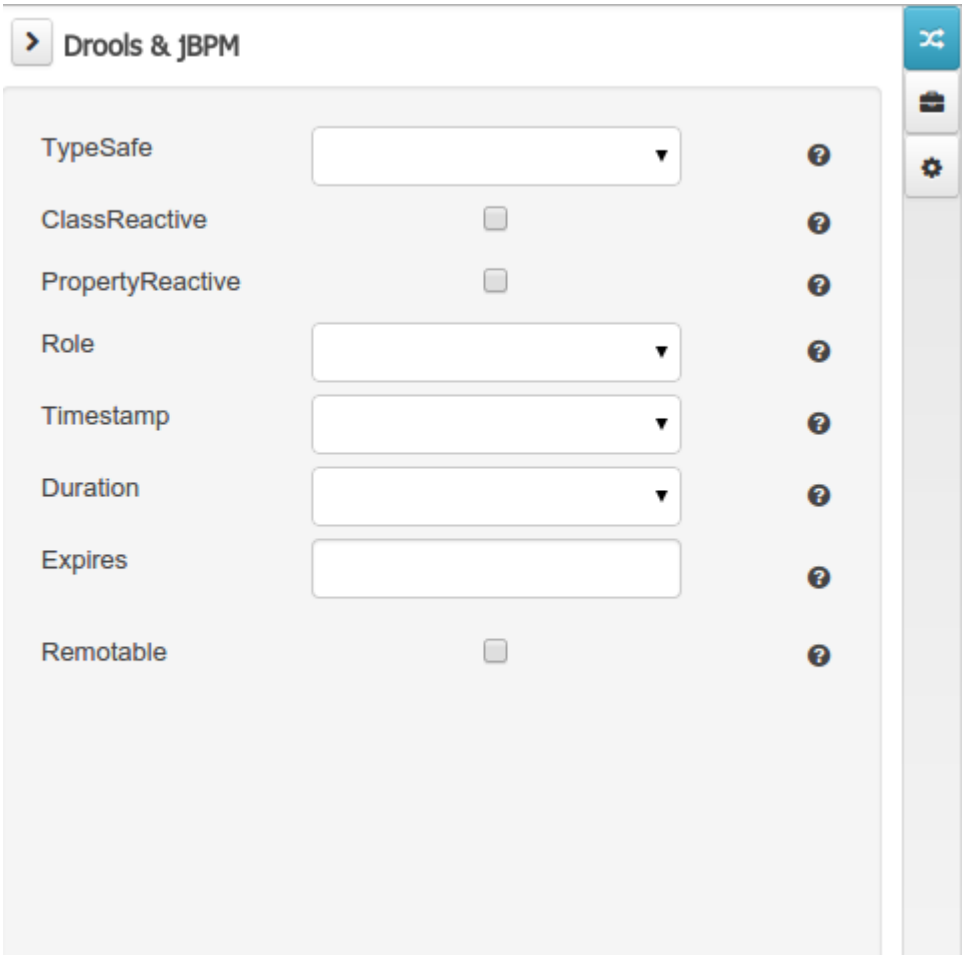


Figure 18.61. Drools & jBPM tool window

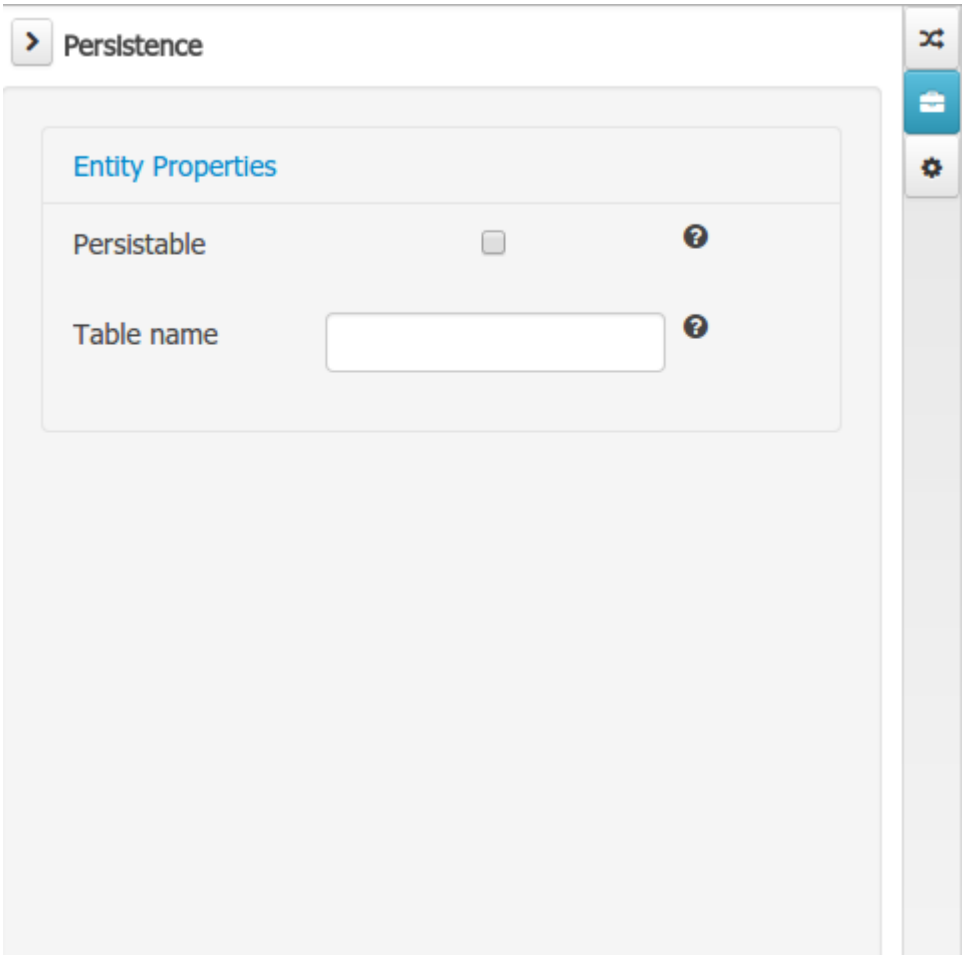


Figure 18.62. Persistence tool window

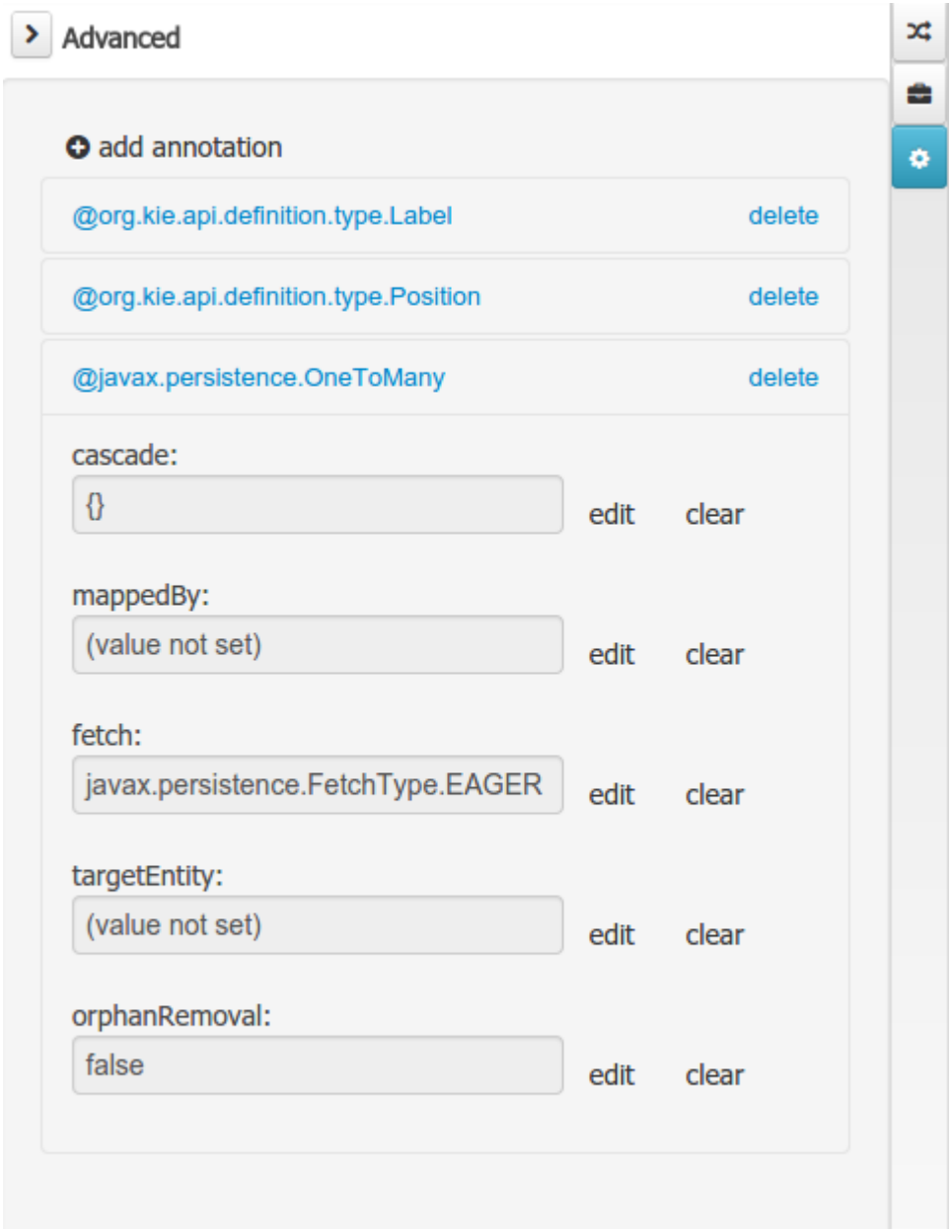


Figure 18.63. Advanced tool window

The "Source" tab shows an editor that allows the visualization and modification of the generated java code.

- Round trip between the "Editor" and "Source" tabs is possible, and also source code preservation is provided. It means that not matter where the Java code was generated (e.g. Eclipse, Data modeller), the data modeller will only update the necessary code blocks to maintain the model updated.

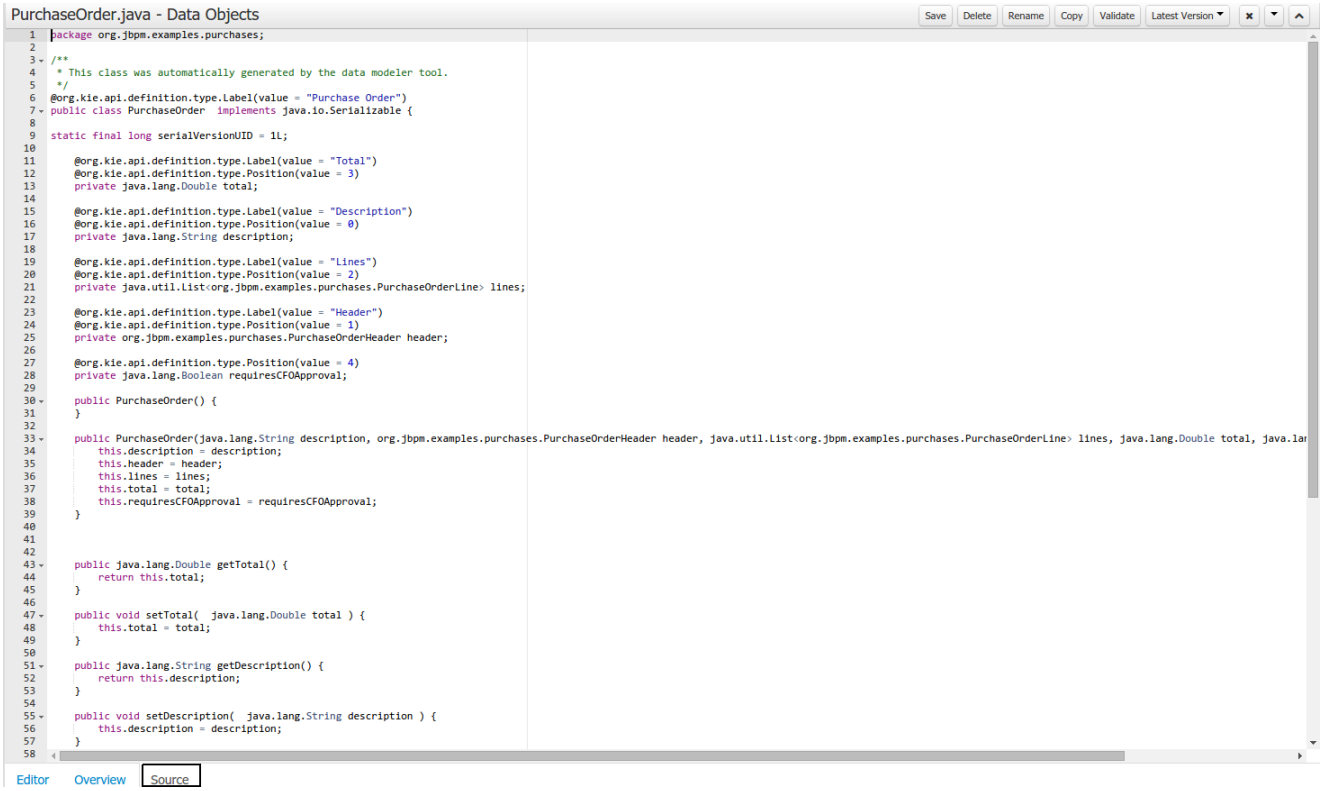


Figure 18.64. Source editor

The "Overview" tab shows the standard metadata and version information as the other workbench editors.

18.7.7.2. Data Objects

A data model consists of data objects which are a logical representation of some real-world data. Such data objects have a fixed set of modeller (or application-owned) properties, such as its internal identifier, a label, description, package etc. Besides those, a data object also has a variable set of user-defined fields, which are an abstraction of a real-world property of the type of data that this logical data object represents.

Creating a data object can be achieved using the workbench "New Item - Data Object" menu option.

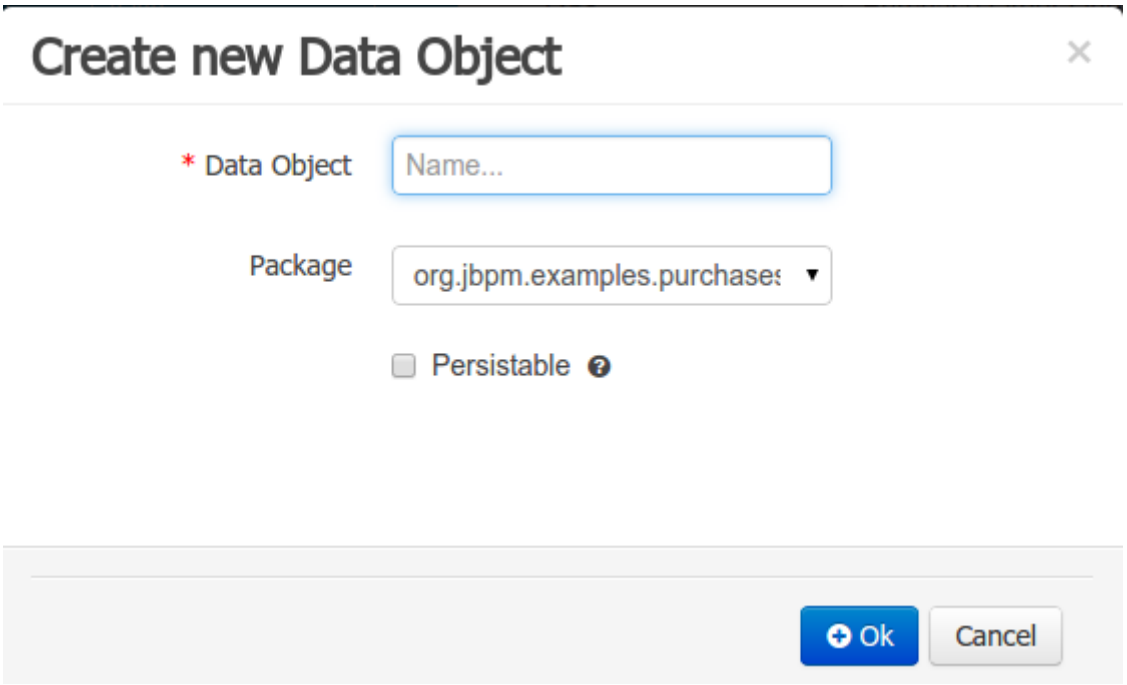


Figure 18.65. New Data Object menu option

Both resource name and location are mandatory parameters. When the "Ok" button is pressed a new Java file will be created and a new editor instance will be opened for the file edition. The optional "Persistable" attribute will add by default configurations on the data object in order to make it a JPA entity. Use this option if your jBPM project needs to store data object's information in a data base.

18.7.7.3. Properties & relationships

Once the data object has been created, it now has to be completed by adding user-defined properties to its definition. This can be achieved by pressing the "add field" button. The "New Field" dialog will be opened and the new field can be created by pressing the "Create" button. The "Create and continue" button will also add the new field to the Data Object, but won't close the dialog. In this way multiple fields can be created avoiding the popup opening multiple times. The following fields can (or must) be filled out:

- The field's internal identifier (mandatory). The value of this field must be unique per data object, i.e. if the proposed identifier already exists within current data object, an error message will be displayed.
- A label (optional): as with the data object definition, the user can define a user-friendly label for the data object field which is about to be created. This has no further implications on how fields from objects of this data object will be treated. If a label is defined, then this is how the field will be displayed throughout the data modeller tool.
- A field type (mandatory): each data object field needs to be assigned with a type.

This type can be either of the following:

1. A 'primitive java object' type: these include most of the object equivalents of the standard Java primitive types, such as Boolean, Short, Float, etc, as well as String, Date, BigDecimal and BigInteger.

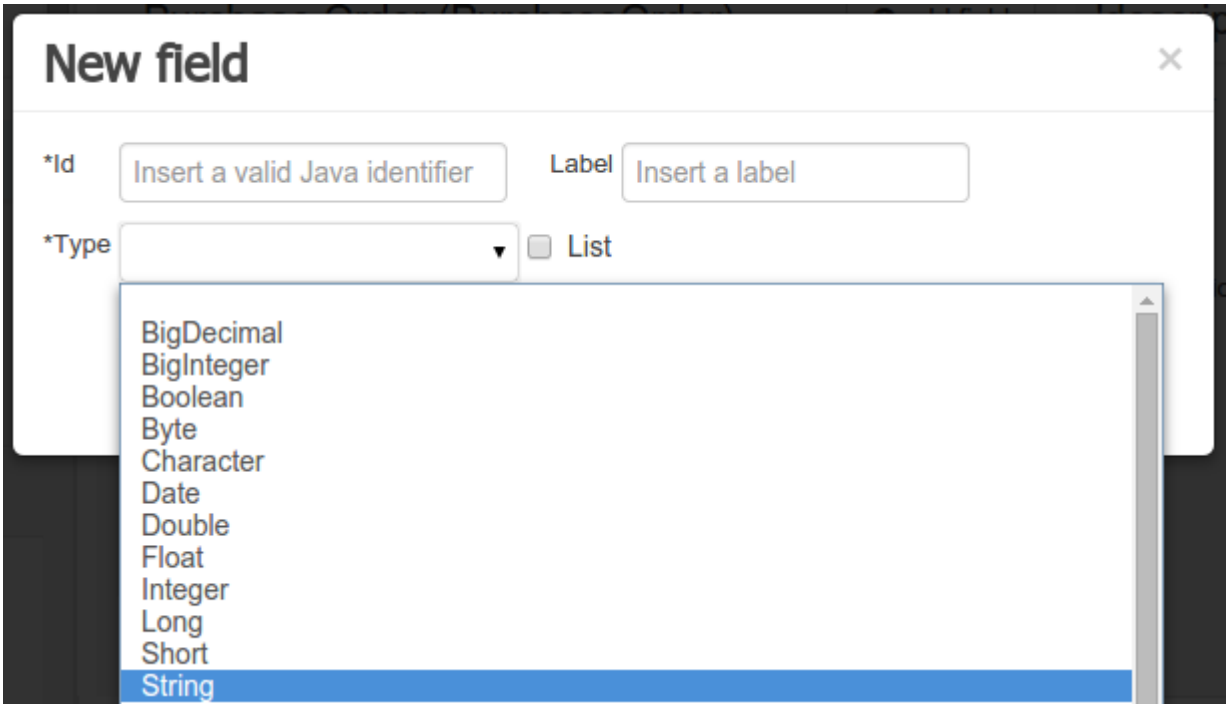


Figure 18.66. Primitive object field types

2. A 'data object' type: any user defined data object automatically becomes a candidate to be defined as a field type of another data object, thus enabling the creation of relationships between them. A data object field can be created either in 'single' or in 'multiple' form, the latter implying that the field will be defined as a collection of this type, which will be indicated by selecting "List" checkbox.

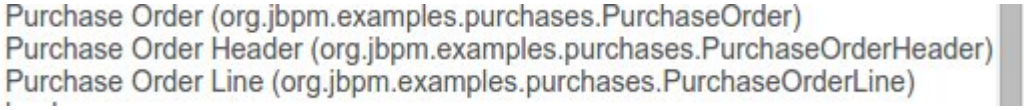


Figure 18.67. Data object field types

3. A 'primitive java' type: these include java primitive types byte, short, int, long, float, double, char and boolean.

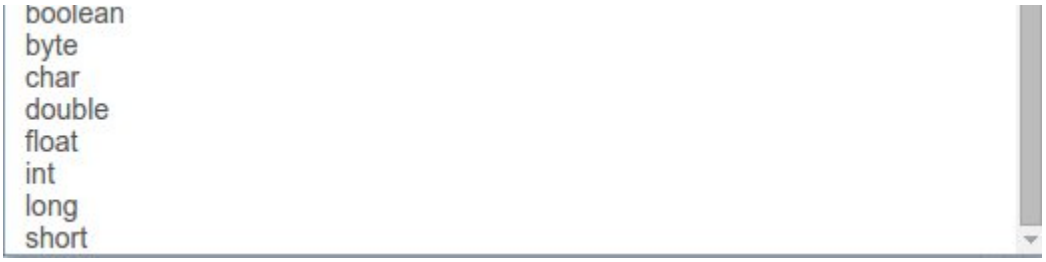


Figure 18.68. Primitive field types

When finished introducing the initial information for a new field, clicking the 'Create' button will add the newly created field to the end of the data object's fields table below:

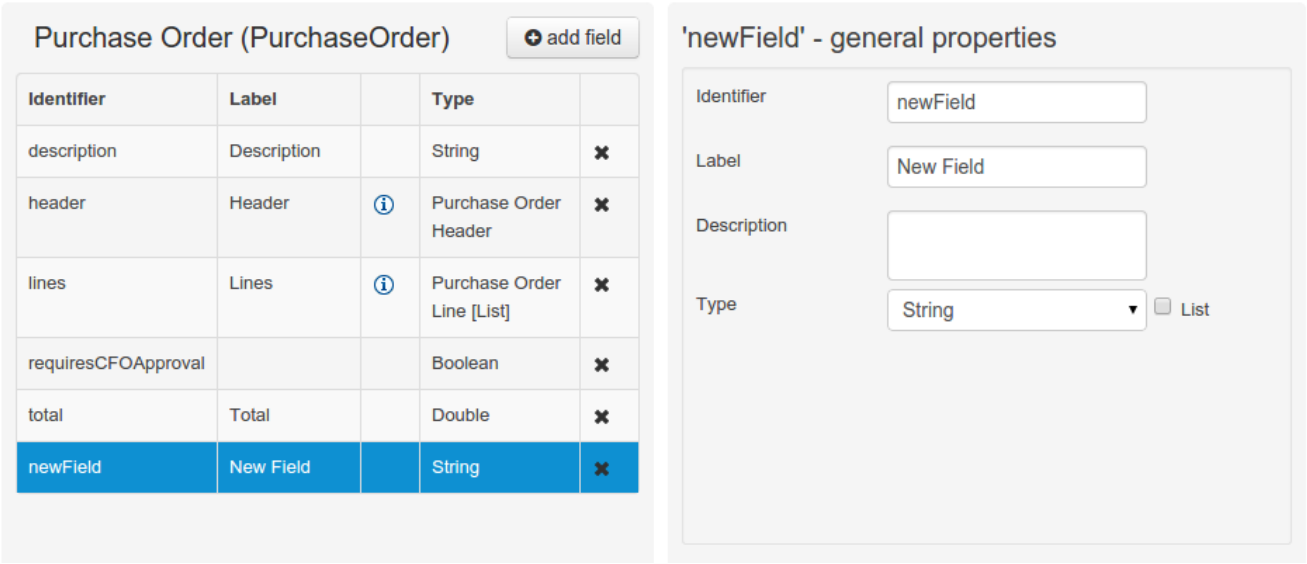


Figure 18.69. New field has been created

The new field will also automatically be selected in the data object's field list, and its properties will be shown in the Field general properties editor. Additionally the field properties will be loaded in the different tool windows, in this way the field will be ready for edition in whatever selected tool window.

At any time, any field (without restrictions) can be deleted from a data object definition by clicking on the corresponding 'x' icon in the data object's fields table.

18.7.7.4. Additional options

As stated before, both Data Objects as well as Fields require some of their initial properties to be set upon creation. Additionally there are three domains of properties that can be configured for a given Data Object. A domain is basically a set of properties related to a given business area.

Current available domains are, "Drools & jJBPM", "Persistence" and the "Advanced" domain. To work on a given domain the user should select the corresponding "Tool window" (see below) on the right side toolbar. Every tool window usually provides two editors, the "Data Object" level editor and the "Field" level editor, that will be shown depending on the last selected item, the Data Object or the Field.

18.7.7.4.1. Drools & jBPM domain

The Drools & jBPM domain editors manages the set of Data Object or Field properties related to drools applications.

18.7.7.4.1.1. Drools & jBPM object editor

The Drools & jBPM object editor manages the object level drools properties

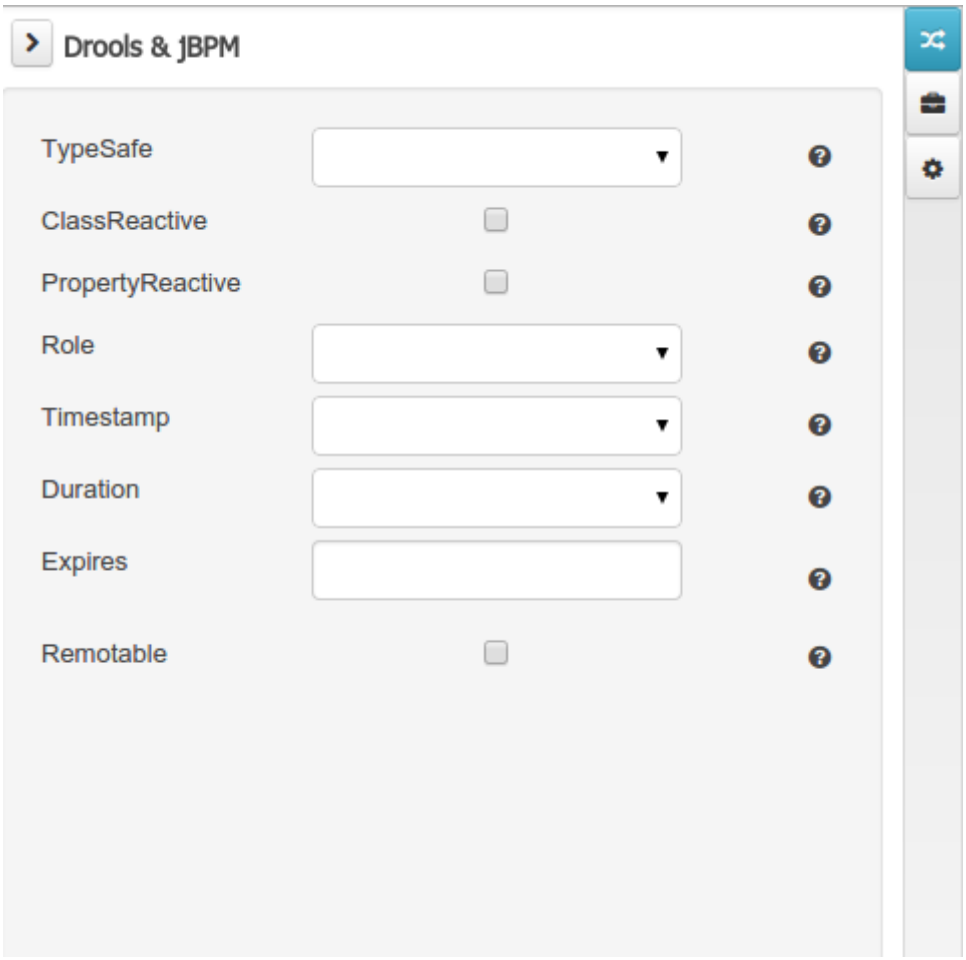


Figure 18.70. The data object's properties

- TypeSafe: this property allows to enable/disable the type safe behaviour for current type. By default all type declarations are compiled with type safety enabled. (See Drools for more information on this matter).

- ClassReactive: this property allows to mark this type to be treated as "Class Reactive" by the Drools engine. (See Drools for more information on this matter).
- PropertyReactive: this property allows to mark this type to be treated as "Property Reactive" by the Drools engine. (See Drools for more information on this matter).
- Role: this property allows to configure how the Drools engine should handle instances of this type: either as regular facts or as events. By default all types are handled as a regular fact, so for the time being the only value that can be set is "Event" to declare that this type should be handled as an event. (See Drools Fusion for more information on this matter).
- Timestamp: this property allows to configure the "timestamp" for an event, by selecting one of his attributes. If set the engine will use the timestamp from the given attribute instead of reading it from the Session Clock. If not, the engine will automatically assign a timestamp to the event. (See Drools Fusion for more information on this matter).
- Duration: this property allows to configure the "duration" for an event, by selecting one of his attributes. If set the engine will use the duration from the given attribute instead of using the default event duration = 0. (See Drools Fusion for more information on this matter).
- Expires: this property allows to configure the "time offset" for an event expiration. If set, this value must be a temporal interval in the form: [#d][#h][#m][#s][#[ms]] Where [ ] means an optional parameter and # means a numeric value. e.g.: 1d2h, means one day and two hours. (See Drools Fusion for more information on this matter).
- Remotable: If checked this property makes the Data Object available to be used with jBPM remote services as REST, JMS and WS. (See jBPM for more information on this matter).

18.7.7.4.1.2. Drools & jBPM field editor

The Drools & jBPM object editor manages the field level drools properties

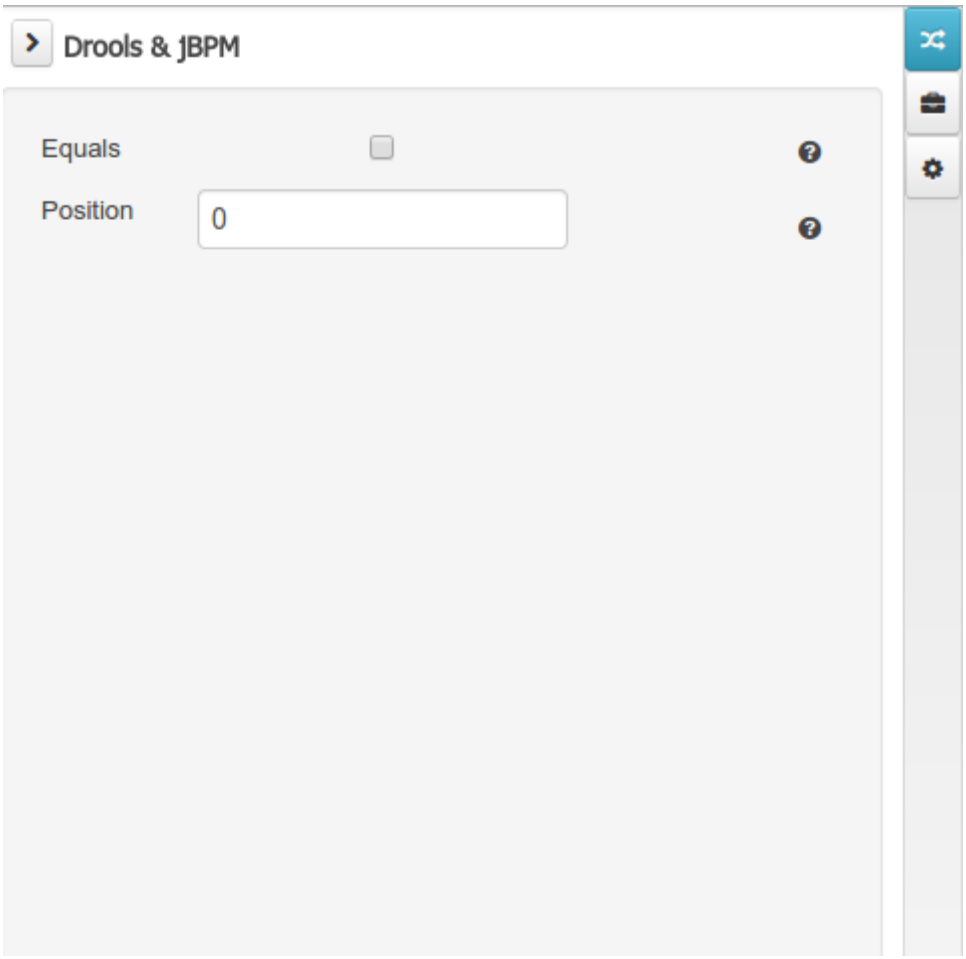


Figure 18.71. The data object's field properties

- Equals: checking this property for a Data Object field implies that it will be taken into account, at the code generation level, for the creation of both the equals() and hashCode() methods in the generated Java class. We will explain this in more detail in the following section.
- Position: this field requires a zero or positive integer. When set, this field will be interpreted by the Drools engine as a positional argument (see the section below and also the Drools documentation for more information on this subject).

18.7.7.4.2. Persistence domain

The Persistence domain editors manages the set of Data Object or Field properties related to persistence.

18.7.7.4.2.1. Persistence domain object editor

Persistence domain object editor manages the object level persistence properties

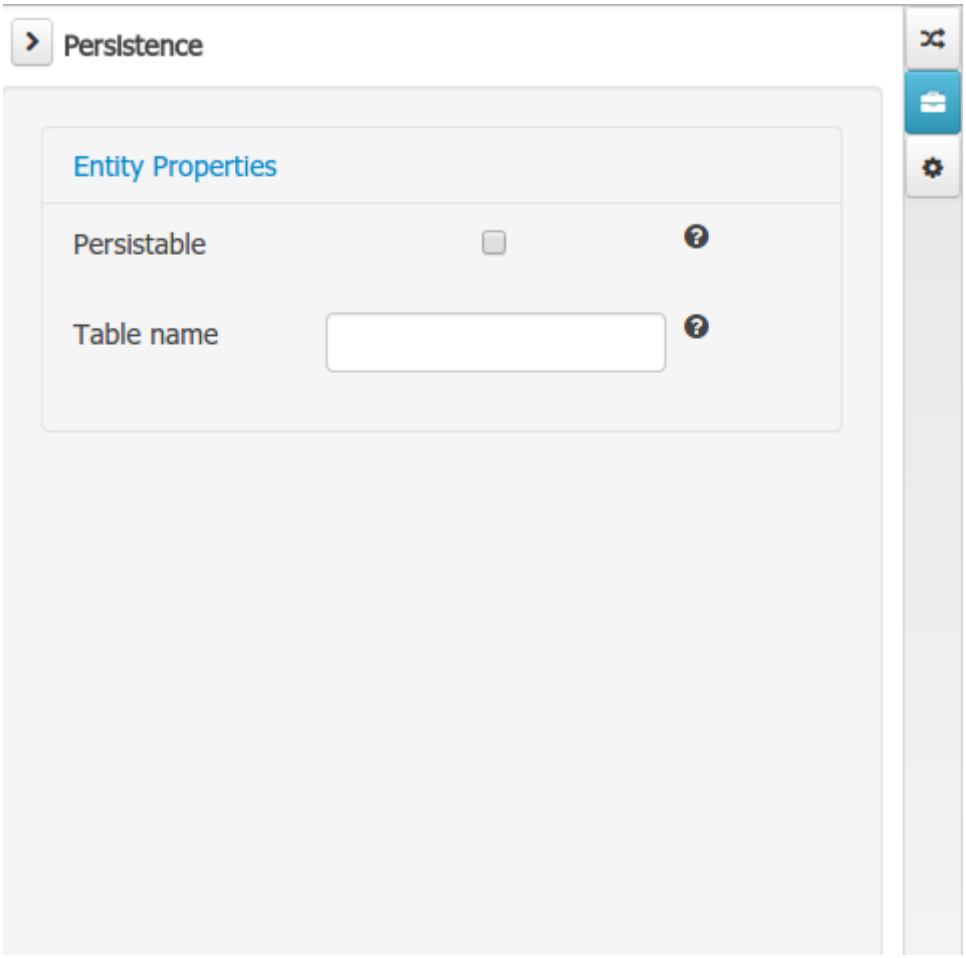


Figure 18.72. The data object's properties

- Persistable: this property allows to configure current Data Object as persistable.
- Table name: this property allows to set a user defined database table name for current Data Object.

18.7.7.4.2.2. Persistence domain field editor

The persistence domain field editor manages the field level persistence properties and is divided in three sections.

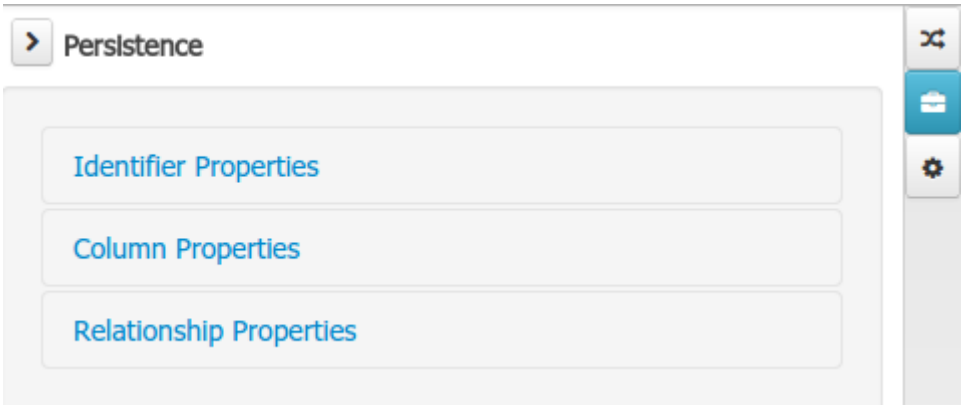
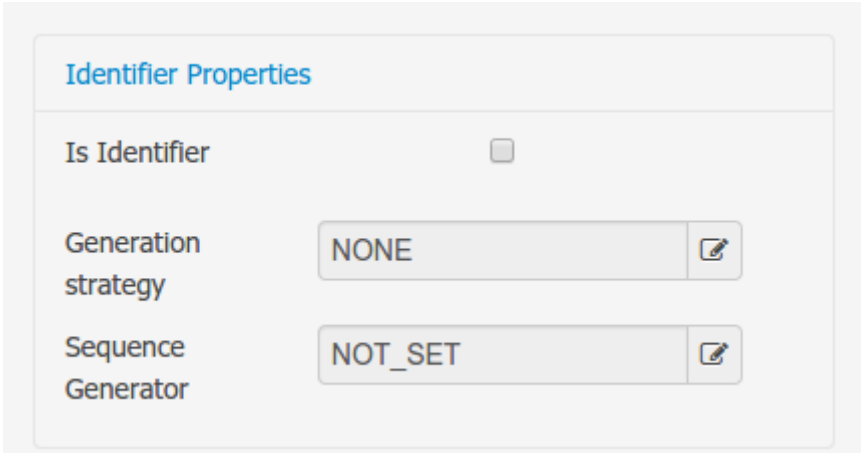


Figure 18.73. Persistence domain field editor sections

18.7.7.4.2.2.1. Identifier:



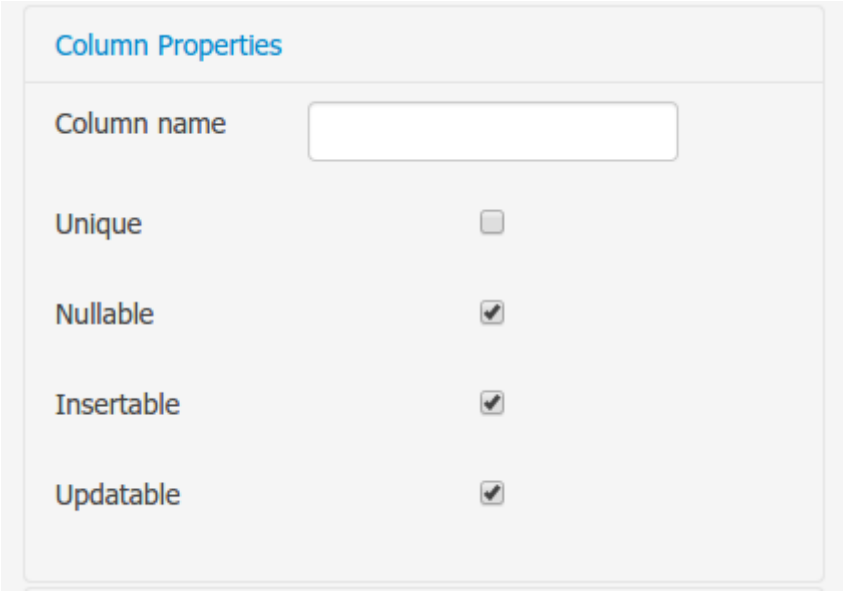
A persistable Data Object should have one and only one field defined as the Data Object identifier. The identifier is typically a unique number that distinguishes a given Data Object instance from all other instances of the same class.

- Is Identifier: marks current field as the Data Object identifier. A persistable Data Object should have one and only one field marked as identifier, and it should be a base java type, like String, Integer, Long, etc. A field that references a Data Object, or is a multiple field can not be marked as identifier. And also composite identifiers are not supported in this version. When a persistable Data Object is created an identifier field is created by default with the properly initializations, it's strongly recommended to use this identifier.
- Generation Strategy: the generation strategy establishes how the identifier values will be automatically generated when the Data Object instances are created and stored in a database. (e.g. by the forms associated to jBPM processes human tasks.) When the by default Identifier field is created, the generation strategy will be also automatically set and it's strongly recommended to use this configuration.

- Sequence Generator: the generator represents the seed for the values that will be used by the Generation Strategy. When the by default Identifier field is created the Sequence Generator will be also automatically generated and properly configured to be used by the Generation Strategy.

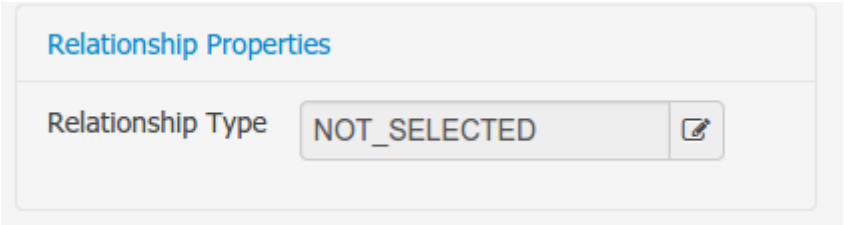
18.7.7.4.2.2.2. Column Properties:

The column properties section enables the customization of some properties of the database column that will store the field value.



- Column name: optional value that sets the database column name for the given field.
- Unique: When checked the unique property establishes that current field value should be a unique key when stored in the database. (if not set the default value is false)
- Nullable: When checked establishes that current field value can be null when stored in a database. (if not set the default value is true)
- Insertable: When checked establishes that column will be included in SQL INSERT statements generated by the persistence provider. (if not set the default value is true)
- Updatable: When checked establishes that the column will be included SQL UPDATE statements generated by the persistence provider. (if not set the default value is true)

18.7.7.4.2.2.3. Relationship Properties:



When the field's type is a Data Object type, or a list of a Data Object type a relationship type should be set in order to let the persistence provider to manage the relation. Fortunately this relation type is automatically set when such kind of fields are added to an already marked as persistable Data Object. The relationship type is set by the following popup.

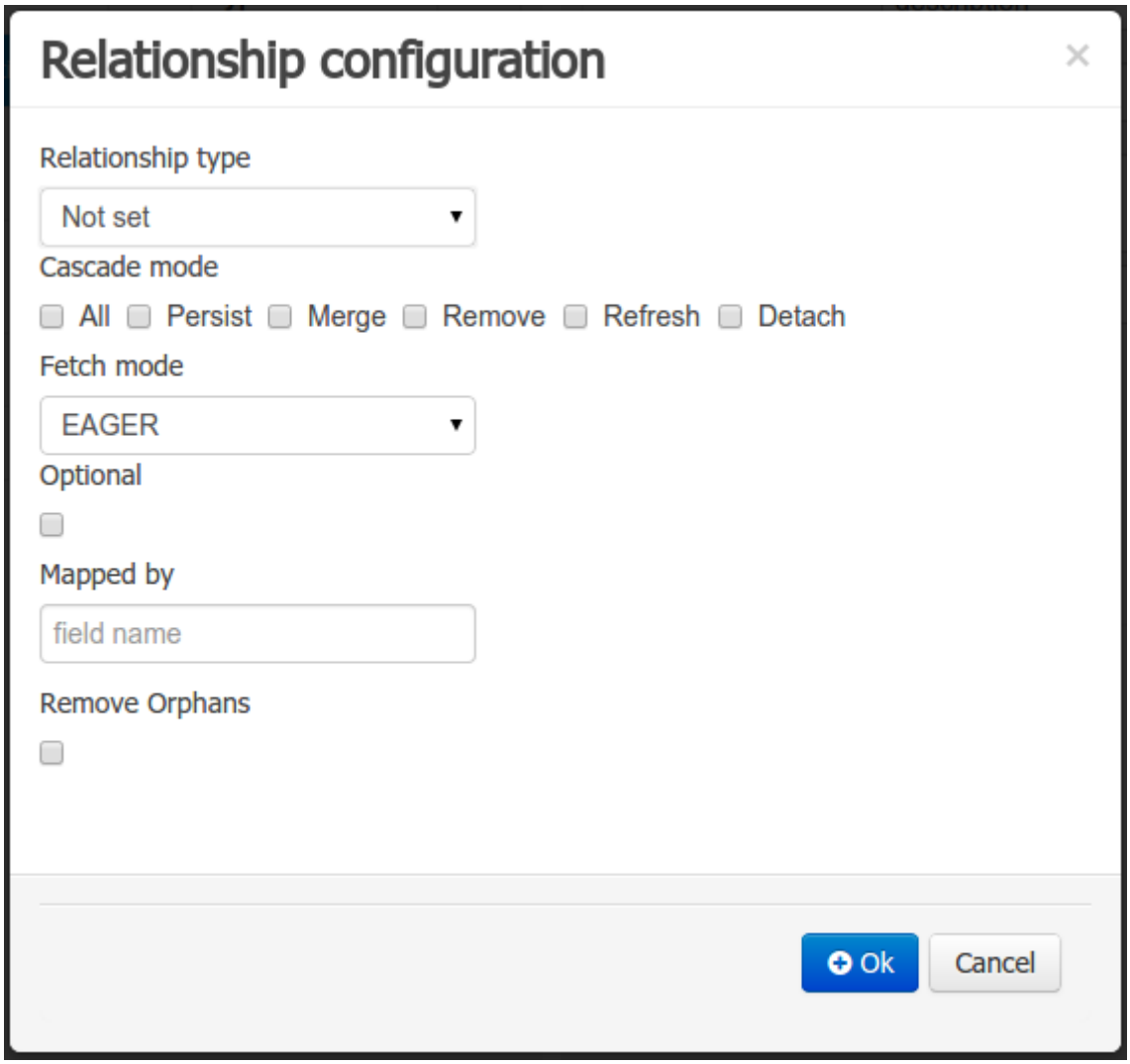


Figure 18.74. Relationship configuration popup

- Relationship type: sets the type of relation from one of the following options:

  One to one: typically used for 1:1 relations where "A is related to one instance of B", and B exists only when A exists. e.g. PurchaseOrder -> PurchaseOrderHeader (a PurchaseOrderHeader exists only if the PurchaseOrder exists)

  One to many: typically used for 1:N relations where "A is related to N instances of B", and the related instances of B exists only when A exists. e.g. PurchaseOrder -> PurchaseOrderLine (a PurchaseOrderLine exists only if the PurchaseOrder exists)

Many to one: typically used for 1:1 relations where "A is related to one instance of B", and B can exist even without A. e.g. PurchaseOrder -> Client (a Client can exist in the database even without an associated PurchaseOrder)

Many to many: typically used for N:N relations where "A can be related to N instances of B, and B can be related to M instances of A at the same time", and both B an A instances can exits in the database independently of the related instances. e.g. Course -> Student. (Course can be related to N Students, and a given Student can attend to M courses)

When a field of type "Data Object" is added to a given persistable Data Object, the "Many to One" relationship type is generated by default.

And when a field of type "list of Data Object" is added to a given persistable Data Object , the "One to Many" relationship is generated by default.

- Cascade mode: Defines the set of cascadable operations that are propagated to the associated entity. The value cascade=ALL is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH}. e.g. when A -> B, and cascade "PERSIST or ALL" is set, if A is saved, then B will be also saved.

  The by default cascade mode created by the data modeller is "ALL" and it's strongly recommended to use this mode when Data Objects are being used by jBPM processes and forms.

- Fetch mode: Defines how related data will be fetched from database at reading time.

  EAGER: related data will be read at the same time. e.g. If A -> B, when A is read from database B will be read at the same time.

  LAZY: reading of related data will be delayed usually to the moment they are required. e.g. If PurchaseOrder -> PurchaseOrderLine the lines reading will be postponed until a method "getLines()" is invoked on a PurchaseOrder instance.

  The default fetch mode created by the data modeller is "EAGER" and it's strongly recommended to use this mode when Data Objects are being used by jBPM processes and forms.

- Optional: establishes if the right side member of a relationship can be null.

- Mapped by: used for reverse relations.

18.7.7.4.3. Advanced domain

The advanced domain enables the configuration of whatever parameter set by the other domains as well as the adding of arbitrary parameters. As it will be shown in the code generation section every "Data Object / Field" parameter is represented by a java annotation. The advanced mode enables the configuration of this annotations.

18.7.7.4.3.1. Advanced domain Data Object / Field editor.

The advanced domain editor has the same shape for both Data Object and Field.

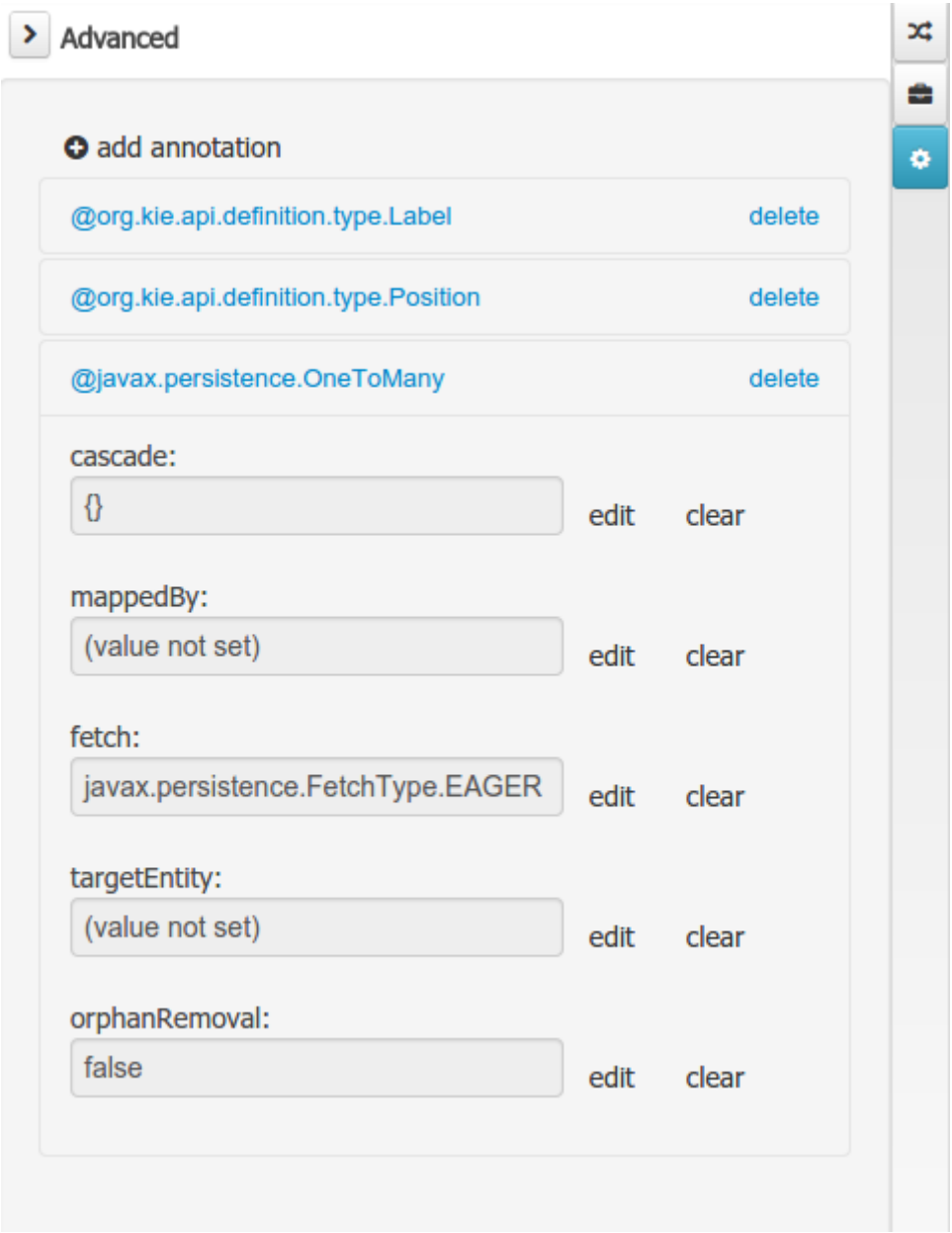


Figure 18.75. Advanced domain editor.

The following operations are available

- delete: enables the deletion of a given Data Object or Field annotation.
- clear: clears a given annotation parameter value.
- edit: enables the edition of a given annotation parameter value.
- add annotation: The add annotation button will start a wizard that will let the addition of whatever java annotation available in the project dependencies.

Add annotation wizard step #1: the first step of the wizard requires the entering of a fully qualified class name of an annotation, and by pressing the "search" button the annotation definition will be loaded into the wizard. Additionally when the annotation definition is loaded, different wizard steps will be created in order to enable the completion of the different annotation parameters. Required parameters will be marked with "*".

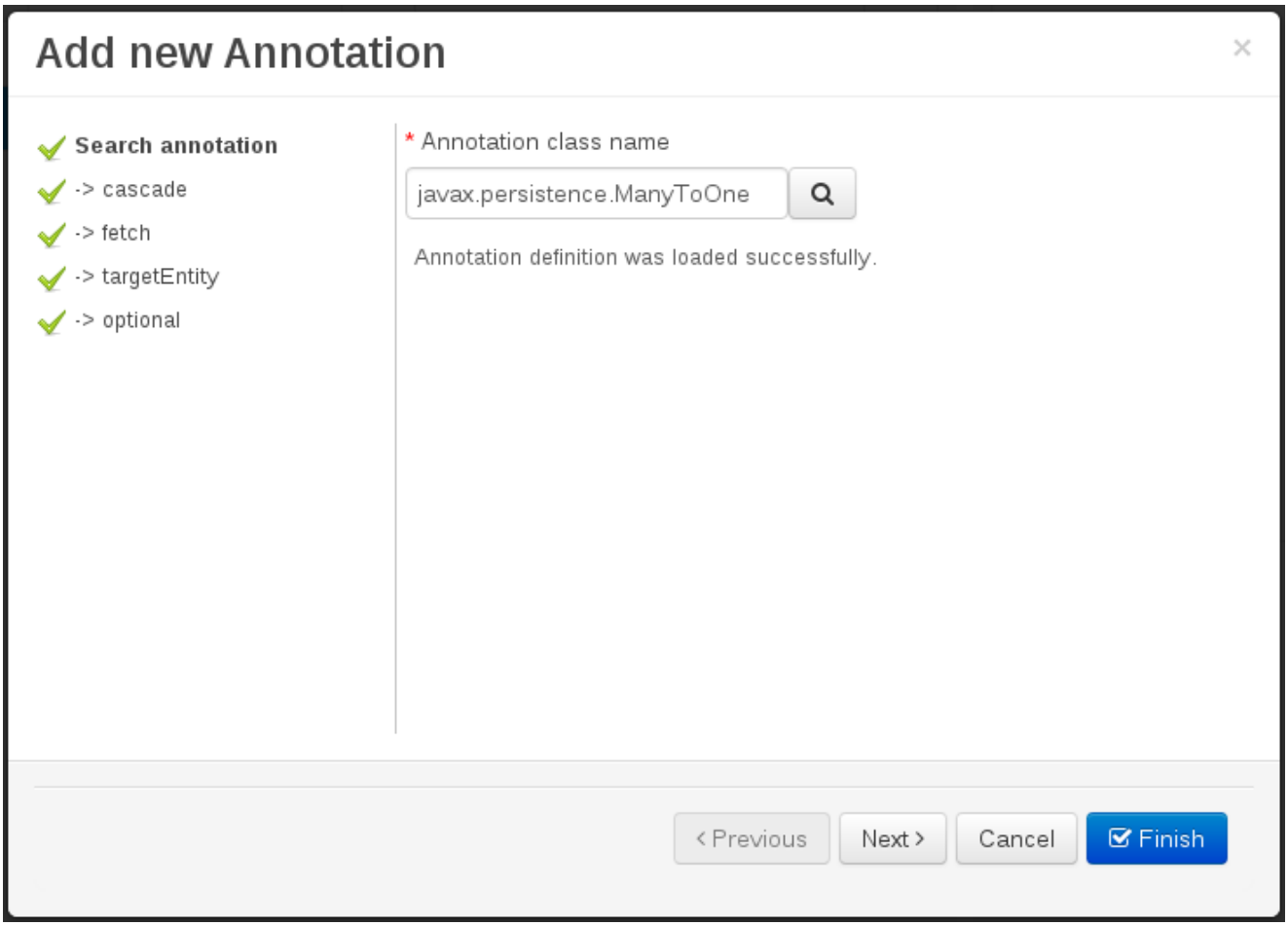


Figure 18.76. Annotation definition loaded into the wizard.

Whenever it's possible the wizard will provide a suitable editor for the given parameters.

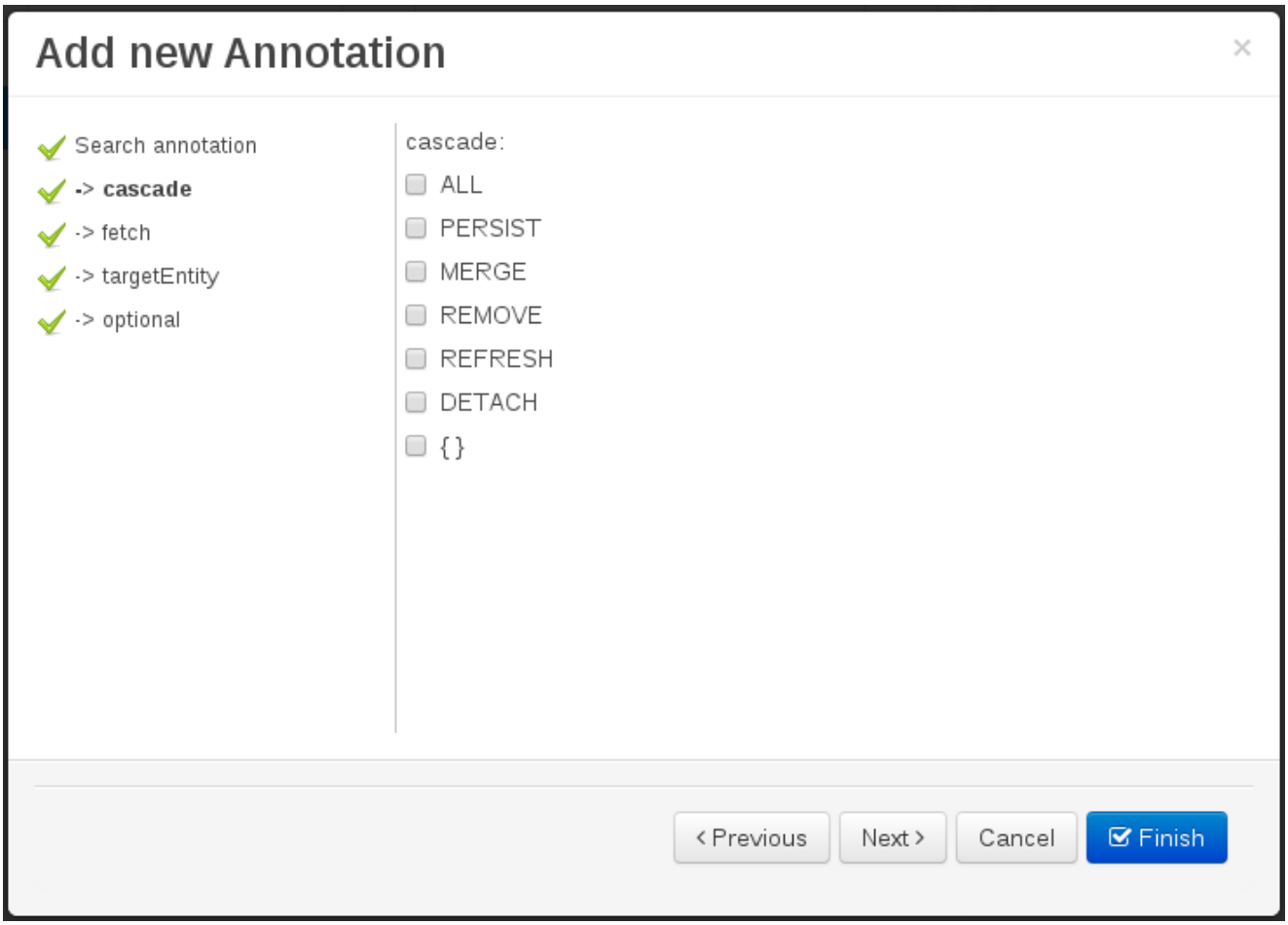


Figure 18.77. Automatically generated enum values editor for an Enumeration annotation parameter.

A generic parameter editor will be provided when it's not possible to calculate a customized editor

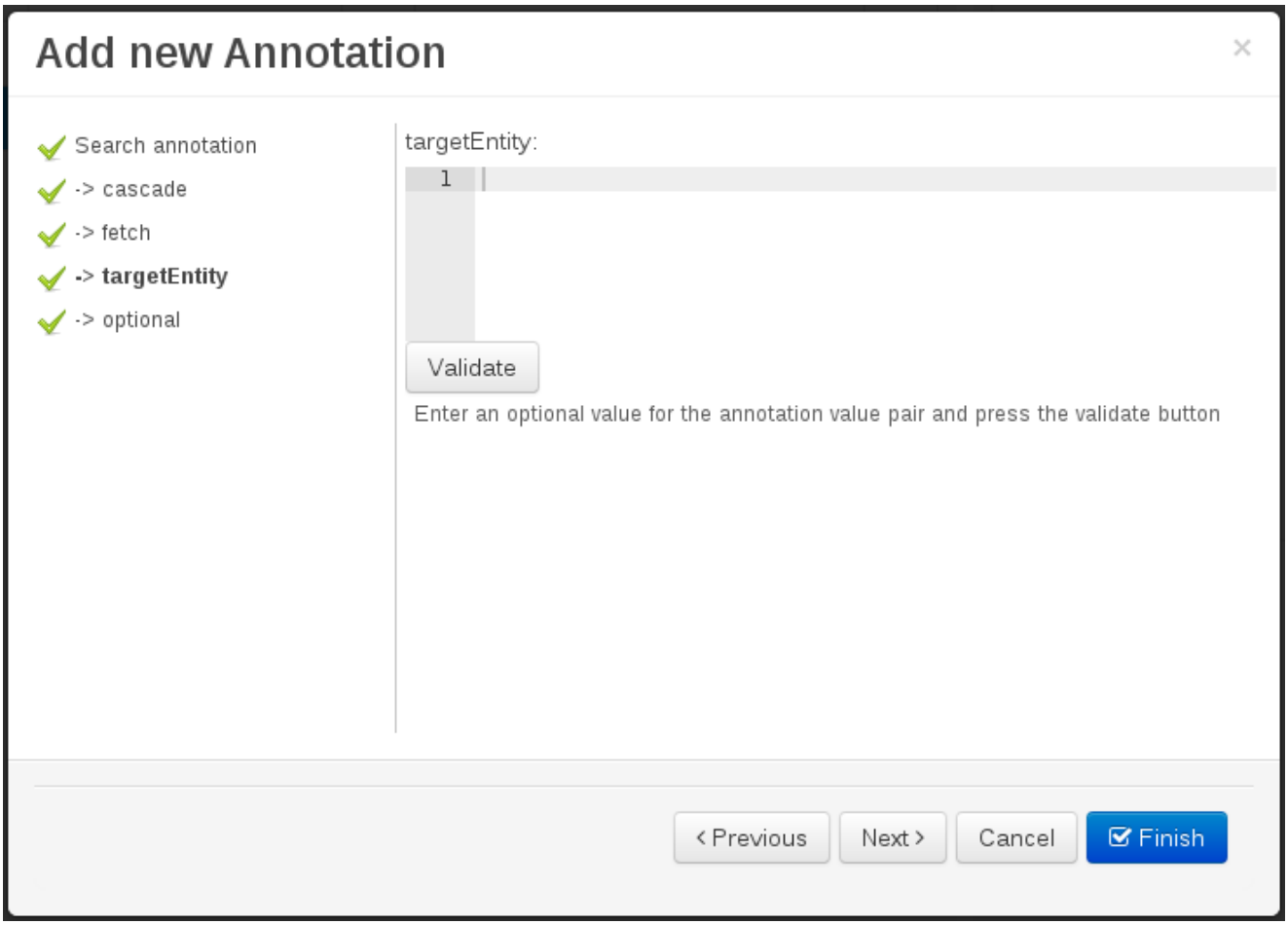


Figure 18.78. Generic annotation parameter editor

When all required parameters has been entered and validated, the finish button will be enabled and the wizard can be completed by adding the annotation to the given Data Object or Field.

18.7.7.5. Generate data model code.

The data model in itself is merely a visual tool that allows the user to define high-level data structures, for them to interact with the Drools Engine on the one hand, and the jBPM platform on the other. In order for this to become possible, these high-level visual structures have to be transformed into low-level artifacts that can effectively be consumed by these platforms. These artifacts are Java POJOs (Plain Old Java Objects), and they are generated every time the data model is saved, by pressing the "Save" button in the top Data Modeller Menu. Additionally when the user round trip between the "Editor" and "Source" tab, the code is auto generated to maintain the consistency with the Editor view and vice versa.

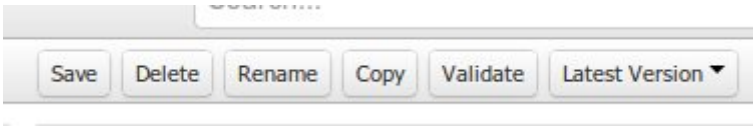


Figure 18.79. Save the data model from the top menu

The resulting code is generated according to the following transformation rules:

- The data object's identifier property will become the Java class's name. It therefore needs to be a valid Java identifier.
- The data object's package property becomes the Java class's package declaration.
- The data object's superclass property (if present) becomes the Java class's extension declaration.
- The data object's label and description properties will translate into the Java annotations "@org.kie.api.definition.type.Label" and "@org.kie.api.definition.type.Description", respectively. These annotations are merely a way of preserving the associated information, and as yet are not processed any further.
- The data object's role property (if present) will be translated into the "@org.kie.api.definition.type.Role" Java annotation, that *IS* interpreted by the application platform, in the sense that it marks this Java class as a Drools Event Fact-Type.
- The data object's type safe property (if present) will be translated into the "@org.kie.api.definition.type.TypeSafe Java annotation. (see Drools)
- The data object's class reactive property (if present) will be translated into the "@org.kie.api.definition.type.ClassReactive Java annotation. (see Drools)
- The data object's property reactive property (if present) will be translated into the "@org.kie.api.definition.type.PropertyReactive Java annotation. (see Drools)
- The data object's timestamp property (if present) will be translated into the "@org.kie.api.definition.type.Timestamp Java annotation. (see Drools)
- The data object's duration property (if present) will be translated into the "@org.kie.api.definition.type.Duration Java annotation. (see Drools)
- The data object's expires property (if present) will be translated into the "@org.kie.api.definition.type.Expires Java annotation. (see Drools)
- The data object's remotable property (if present) will be translated into the "@org.kie.api.remote.Remotable Java annotation. (see jBPM)

A standard Java default (or no parameter) constructor is generated, as well as a full parameter constructor, i.e. a constructor that accepts as parameters a value for each of the data object's user-defined fields.

The data object's user-defined fields are translated into Java class fields, each one of them with its own getter and setter method, according to the following transformation rules:

- The data object field's identifier will become the Java field identifier. It therefore needs to be a valid Java identifier.
- The data object field's type is directly translated into the Java class's field type. In case the field was declared to be multiple (i.e. 'List'), then the generated field is of the "java.util.List" type.
- The equals property: when it is set for a specific field, then this class property will be annotated with the "@org.kie.api.definition.type.Key" annotation, which is interpreted by the Drools Engine, and it will 'participate' in the generated equals() method, which overwrites the equals() method of the Object class. The latter implies that if the field is a 'primitive' type, the equals method will simply compares its value with the value of the corresponding field in another instance of the class. If the field is a sub-entity or a collection type, then the equals method will make a method-call to the equals method of the corresponding data object's Java class, or of the java.util.List standard Java class, respectively.

  If the equals property is checked for *ANY* of the data object's user defined fields, then this also implies that in addition to the default generated constructors another constructor is generated, accepting as parameters all of the fields that were marked with Equals. Furthermore, generation of the equals() method also implies that also the Object class's hashCode() method is overwritten, in such a manner that it will call the hashCode() methods of the corresponding Java class types (be it 'primitive' or user-defined types) for all the fields that were marked with Equals in the Data Model.
- The position property: this field property is automatically set for all user-defined fields, starting from 0, and incrementing by 1 for each subsequent new field. However the user can freely changes the position among the fields. At code generation time this property is translated into the "@org.kie.api.definition.type.Position" annotation, which can be interpreted by the Drools Engine. Also, the established property order determines the order of the constructor parameters in the generated Java class.

As an example, the generated Java class code for the Purchase Order data object, corresponding to its definition as shown in the following figure purchase_example.jpg is visualized in the figure at the bottom of this chapter. Note that the two of the data object's fields, namely 'header' and 'lines' were marked with Equals, and have been assigned with the positions 2 and 1, respectively).

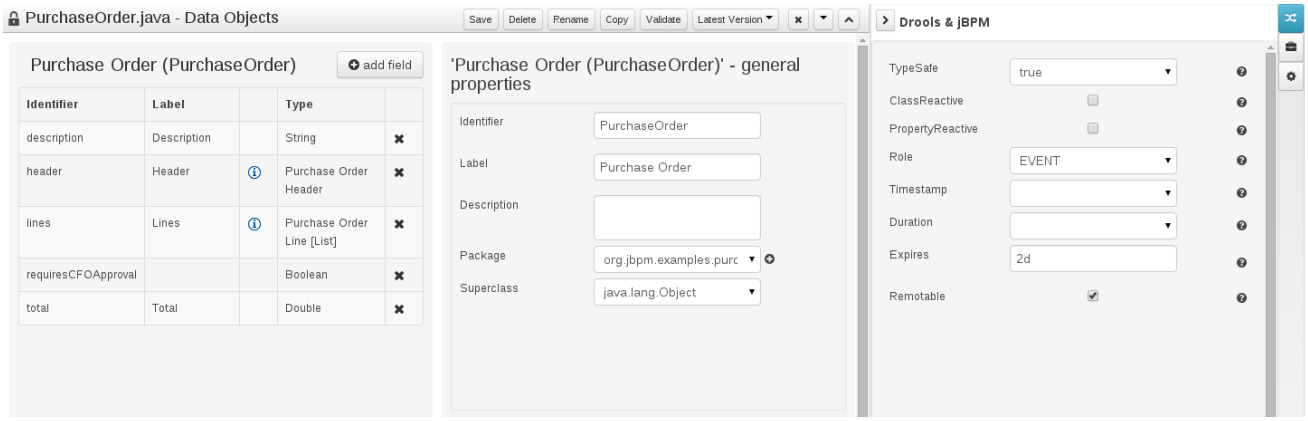


Figure 18.80. Purchase Order configuration

```
package org.jbpm.examples.purchases;

/**
* This class was automatically generated by the data modeler tool.
*/
@org.kie.api.definition.type.Label("Purchase Order")
@org.kie.api.definition.type.TypeSafe(true)
@org.kie.api.definition.type.Role(org.kie.api.definition.type.Role.Type.EVENT)
@org.kie.api.definition.type.Expires("2d")
@org.kie.api.remote.Remotable
public class PurchaseOrder implements java.io.Serializable
{

static final long serialVersionUID = 1L;

@org.kie.api.definition.type.Label("Total")
@org.kie.api.definition.type.Position(3)
private java.lang.Double total;

@org.kie.api.definition.type.Label("Description")
@org.kie.api.definition.type.Position(0)
private java.lang.String description;

@org.kie.api.definition.type.Label("Lines")
@org.kie.api.definition.type.Position(2)
@org.kie.api.definition.type.Key
private java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines;

@org.kie.api.definition.type.Label("Header")
@org.kie.api.definition.type.Position(1)
@org.kie.api.definition.type.Key
private org.jbpm.examples.purchases.PurchaseOrderHeader header;

@org.kie.api.definition.type.Position(4)
private java.lang.Boolean requiresCFOApproval;

public PurchaseOrder()
```

```
    {
    }

    public java.lang.Double getTotal()
    {
    return this.total;
    }

    public void setTotal(java.lang.Double total)
    {
    this.total = total;
    }

    public java.lang.String getDescription()
    {
    return this.description;
    }

    public void setDescription(java.lang.String description)
    {
    this.description = description;
    }

    public java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> getLines()
    {
    return this.lines;
    }

    public void setLines(java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines)
    {
    this.lines = lines;
    }

    public org.jbpm.examples.purchases.PurchaseOrderHeader getHeader()
    {
    return this.header;
    }

    public void setHeader(org.jbpm.examples.purchases.PurchaseOrderHeader header)
    {
    this.header = header;
    }

    public java.lang.Boolean getRequiresCFOApproval()
    {
    return this.requiresCFOApproval;
    }

    public void setRequiresCFOApproval(java.lang.Boolean requiresCFOApproval)
    {
    this.requiresCFOApproval = requiresCFOApproval;
    }

    public PurchaseOrder(java.lang.Double total, java.lang.String description,
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
    org.jbpm.examples.purchases.PurchaseOrderHeader header,
    java.lang.Boolean requiresCFOApproval)
    {
    this.total = total;
```

```
    this.description = description;
    this.lines = lines;
    this.header = header;
    this.requiresCFOApproval = requiresCFOApproval;
    }

    public PurchaseOrder(java.lang.String description,
    org.jbpm.examples.purchases.PurchaseOrderHeader header,
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
    java.lang.Double total, java.lang.Boolean requiresCFOApproval)
    {
    this.description = description;
    this.header = header;
    this.lines = lines;
    this.total = total;
    this.requiresCFOApproval = requiresCFOApproval;
    }

    public PurchaseOrder(
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
    org.jbpm.examples.purchases.PurchaseOrderHeader header)
    {
    this.lines = lines;
    this.header = header;
    }

    @Override
    public boolean equals(Object o)
    {
    if (this == o)
    return true;
    if (o == null || getClass() != o.getClass())
    return false;
    org.jbpm.examples.purchases.PurchaseOrder that = (org.jbpm.examples.purchases.PurchaseOrder) o;
    if (lines != null ? !lines.equals(that.lines) : that.lines != null)
    return false;
    if (header != null ? !header.equals(that.header) : that.header != null)
    return false;
    return true;
    }

    @Override
    public int hashCode()
    {
    int result = 17;
    result = 31 * result + (lines != null ? lines.hashCode() : 0);
    result = 31 * result + (header != null ? header.hashCode() : 0);
    return result;
    }

    }
```

18.7.7.6. Using external models

Using an external model means the ability to use a set for already defined POJOs in current project context. In order to make those POJOs available a dependency to the given JAR should be added. Once the dependency has been added the external POJOs can be referenced from current project data model.

There are two ways to add a dependency to an external JAR file:

- Dependency to a JAR file already installed in current local M2 repository (typically associated the the user home).
- Dependency to a JAR file installed in current KIE Workbench/Drools Workbench "Guvnor M2 repository". (internal to the application)

18.7.7.6.1. Dependency to a JAR file in local M2 repository

To add a dependency to a JAR file in local M2 repository follow this steps.

18.7.7.6.1.1. Open the Project Editor for current project and select the Dependencies view.

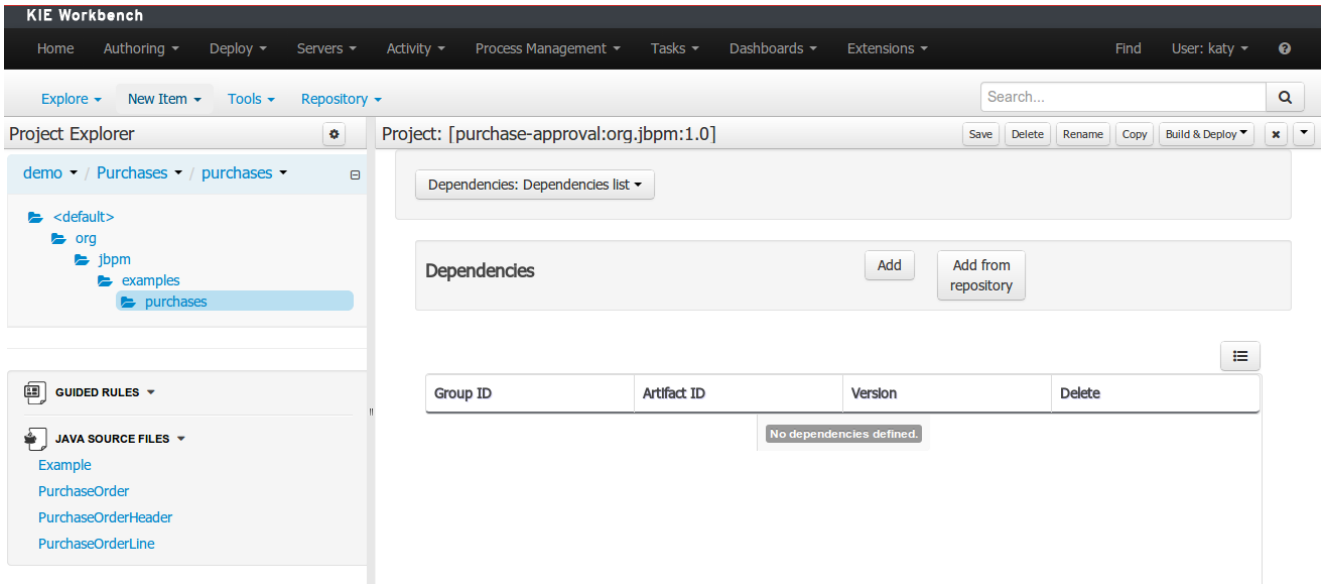


Figure 18.81. Project editor.

18.7.7.6.1.2. Click on the "Add" button to add a new dependency line.

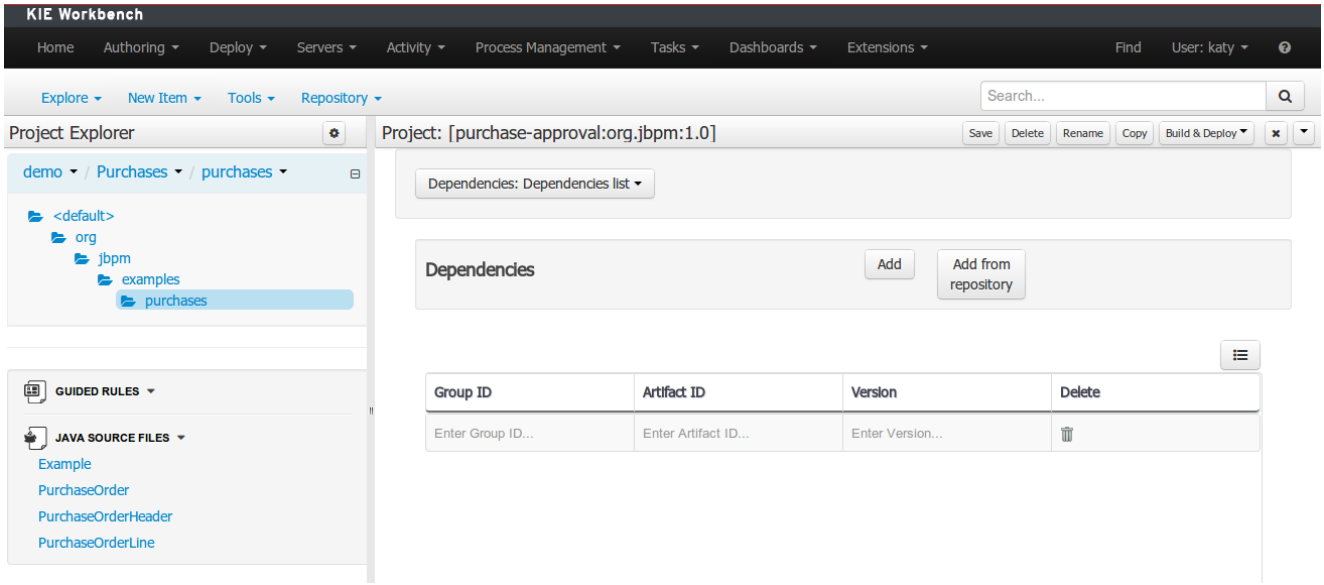


Figure 18.82. New dependency line.

18.7.7.6.1.3. Complete the GAV for the JAR file already installed in local M2 repository.

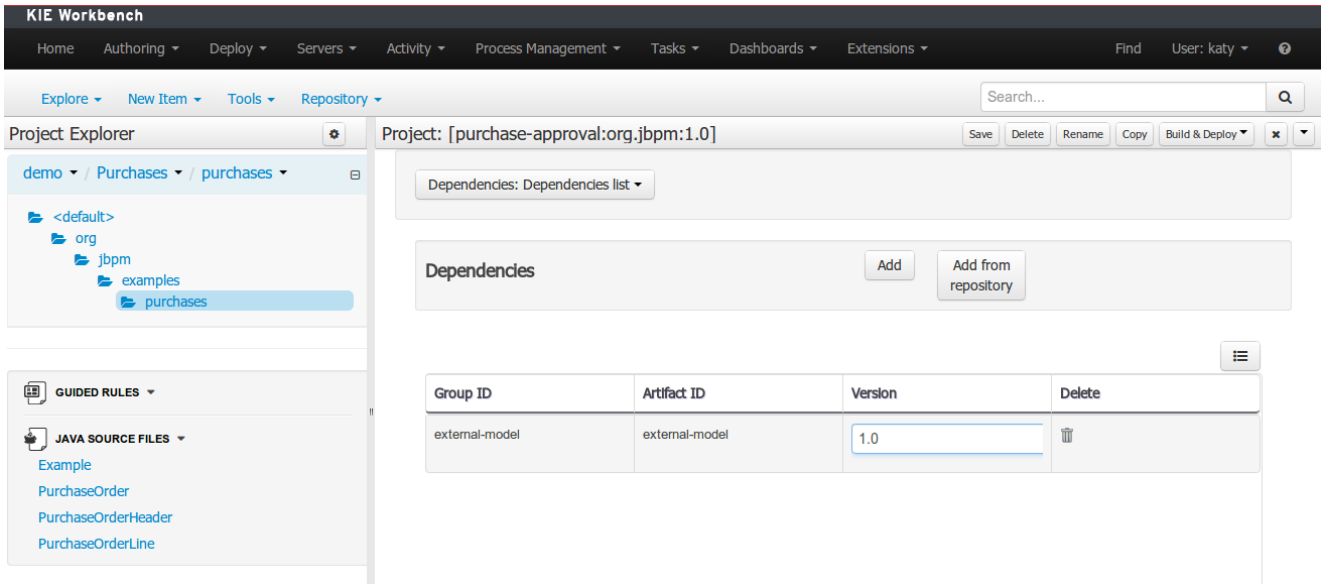


Figure 18.83. Dependency line edition.

18.7.7.6.1.4. Save the project to update its dependencies.

When project is saved the POJOs defined in the external file will be available.

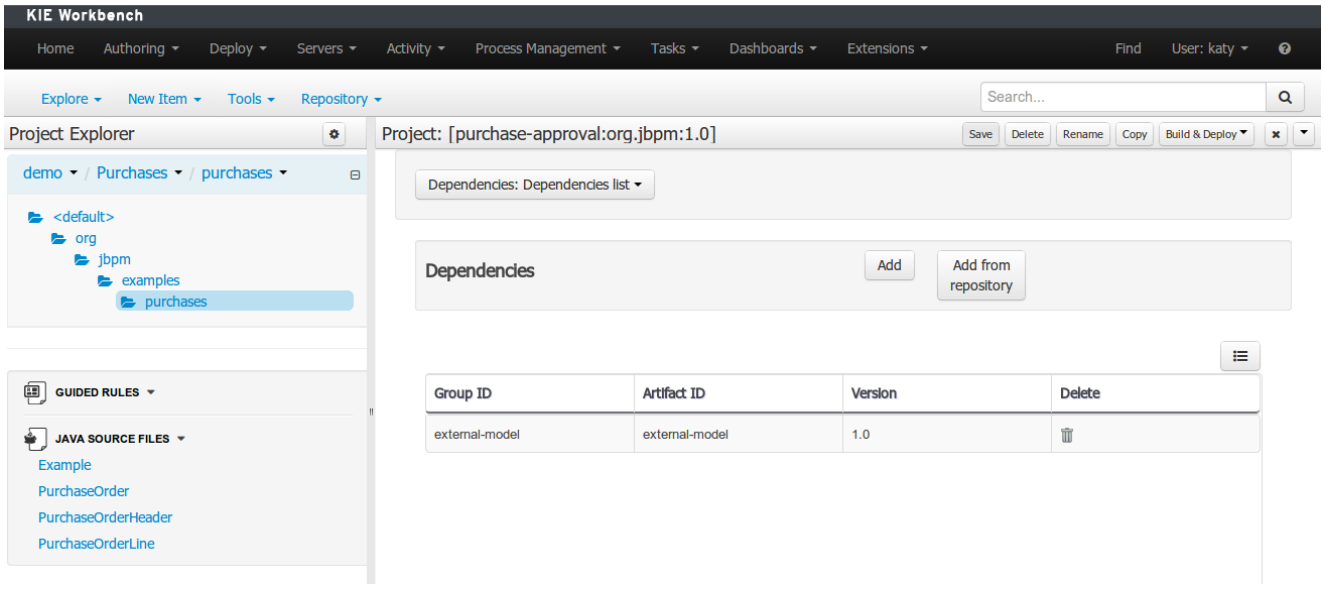


Figure 18.84. Save project.

18.7.7.6.2. Dependency to a JAR file in current "Guvnor M2 repository".

To add a dependency to a JAR file in current "Guvnor M2 repository" follow this steps.

18.7.7.6.2.1. Open the Maven Artifact Repository editor.

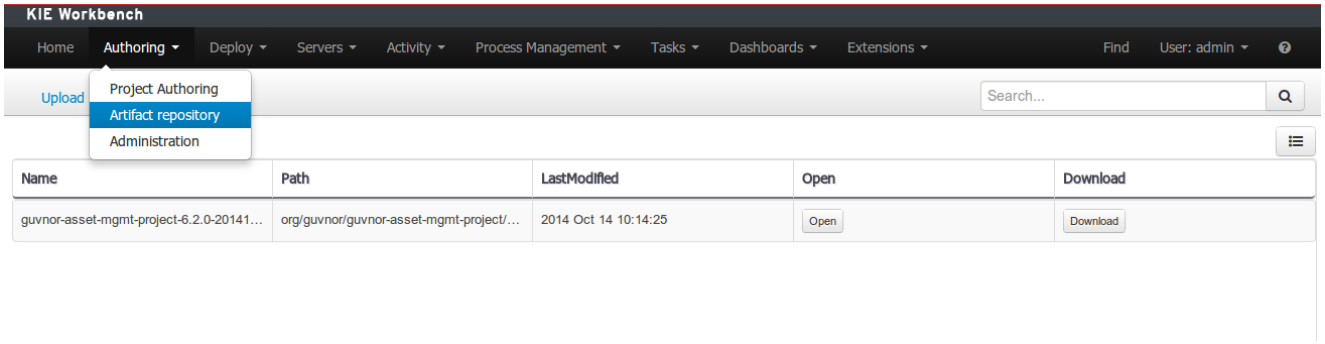


Figure 18.85. Guvnor M2 Repository editor.

18.7.7.6.2.2. Browse your local file system and select the JAR file to be uploaded using the Browse button.

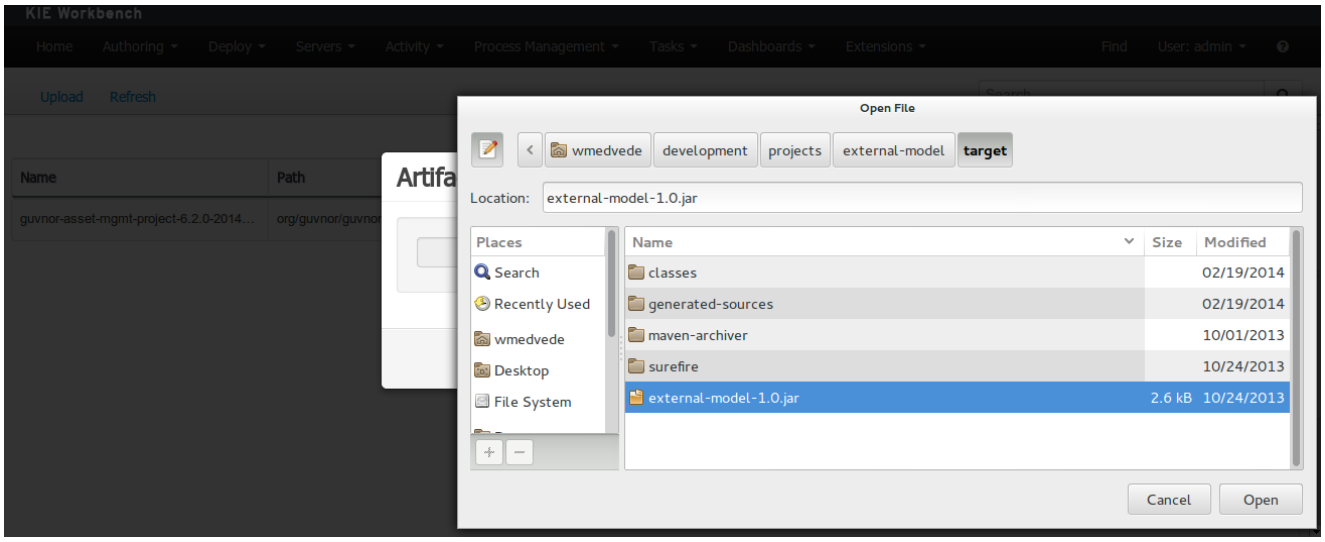


Figure 18.86. File browser.

18.7.7.6.2.3. Upload the file using the Upload button.

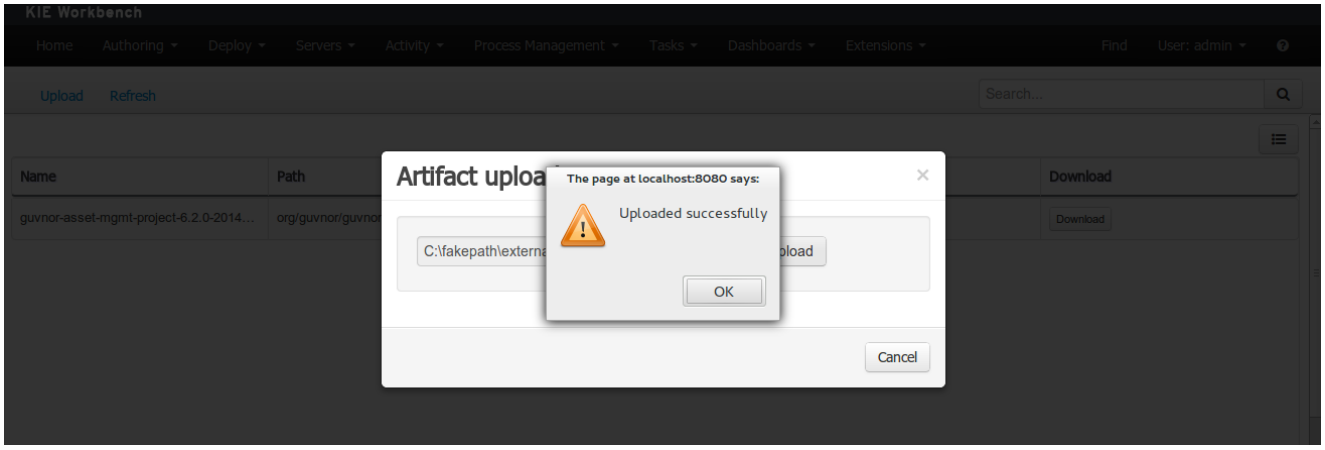


Figure 18.87. File upload success.

18.7.7.6.2.4. Guvnor M2 repository files.

Once the file has been loaded it will be displayed in the repository files list.

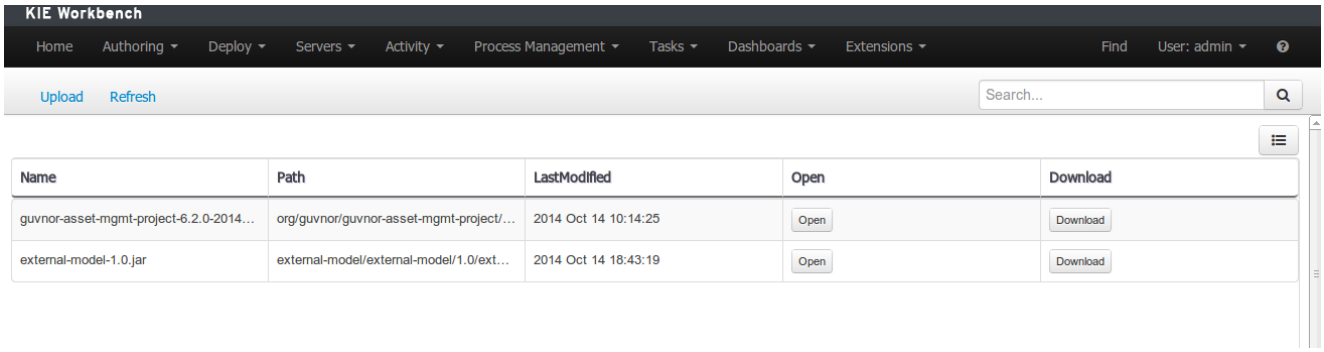


Figure 18.88. Files list.

18.7.7.6.2.5. Provide a GAV for the uploaded file (optional).

If the uploaded file is not a valid Maven JAR (don't have a pom.xml file) the system will prompt the user in order to provide a GAV for the file to be installed.

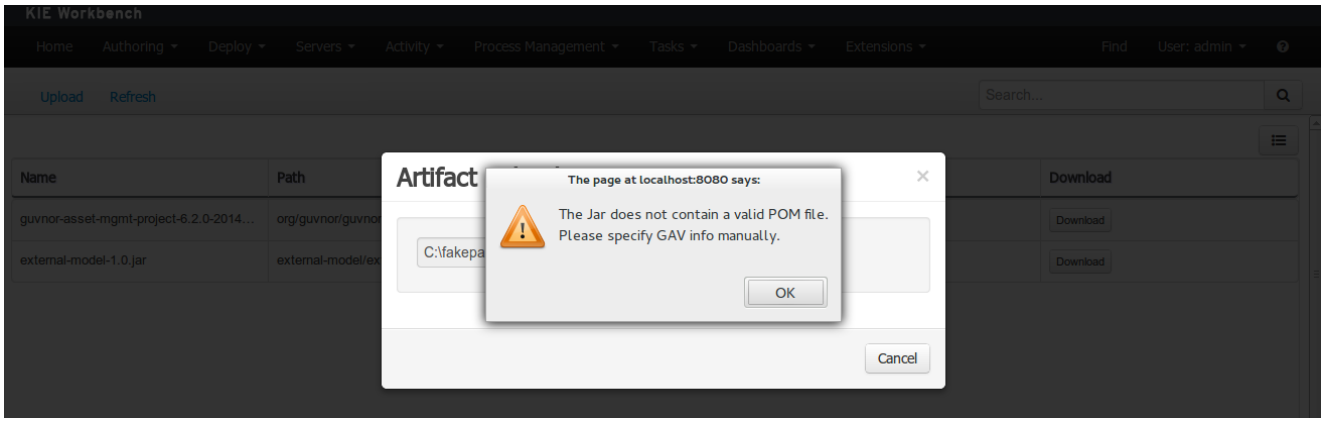


Figure 18.89. Not valid POM.

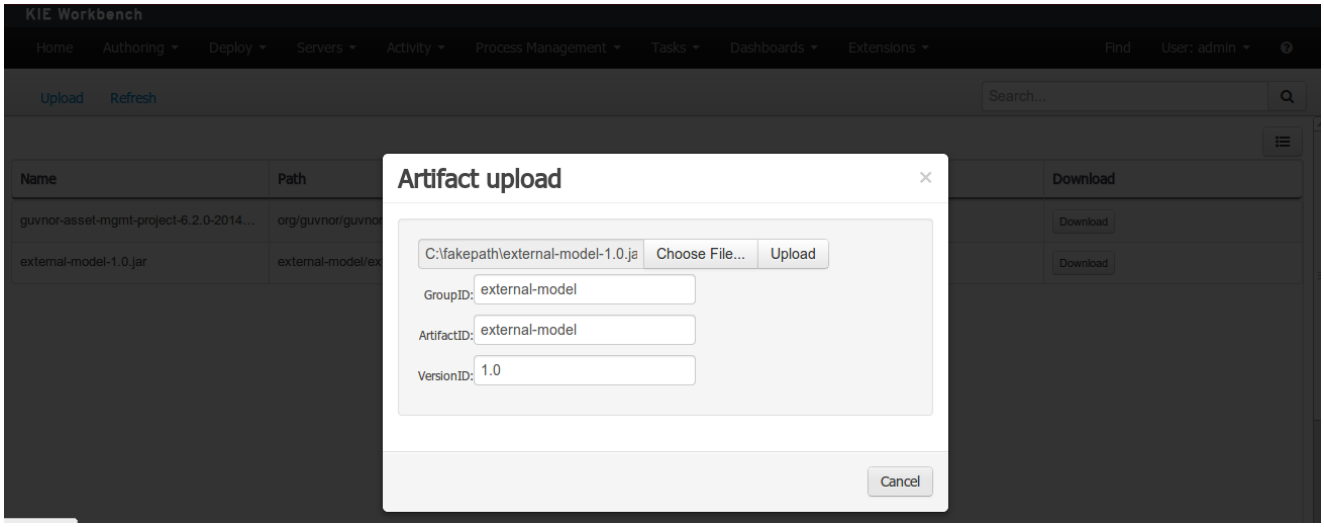


Figure 18.90. Enter GAV manually.

18.7.7.6.2.6. Add dependency from repository.

Open the project editor (see bellow) and click on the "Add from repository" button to open the JAR selector to see all the installed JAR files in current "Guvnor M2 repository". When the desired file is selected the project should be saved in order to make the new dependency available.

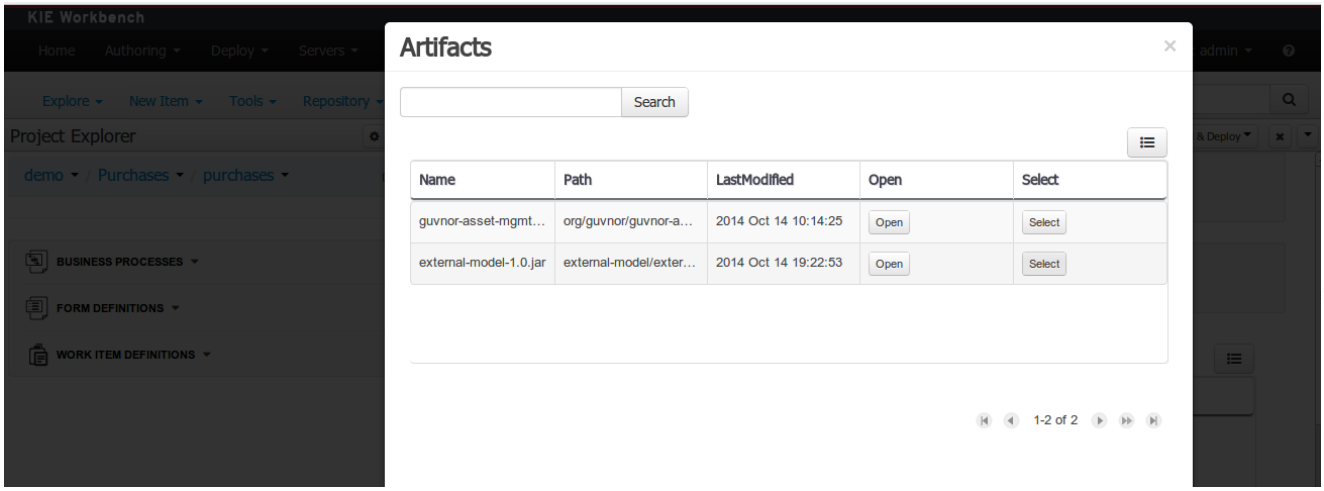


Figure 18.91. Select JAR from "Maven Artifact Repository".

18.7.7.6.3. Using the external objects

When a dependency to an external JAR has been set, the external POJOs can be used in the context of current project data model in the following ways:

- External POJOs can be extended by current model data objects.
- External POJOs can be used as field types for current model data objects.

The following screenshot shows how external objects are prefixed with the string " -ext- " in order to be quickly identified.

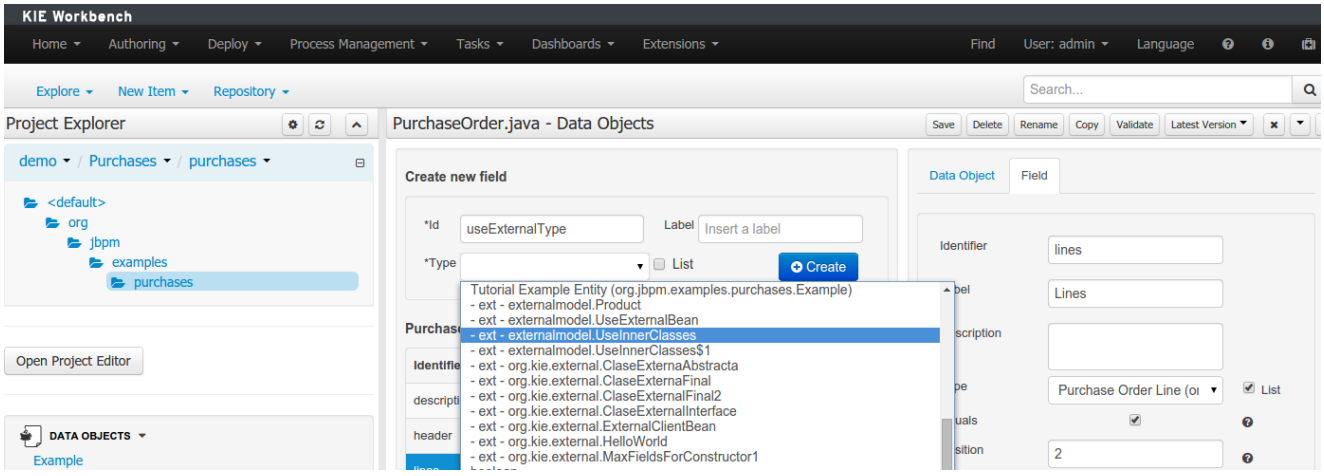


Figure 18.92. Identifying external objects.

18.7.7.7. Roundtrip and concurrency

Current version implements roundtrip and code preservation between Data modeller and Java source code. No matter where the Java code was generated (e.g. Eclipse, Data modeller), the data modeller will only create/delete/update the necessary code elements to maintain the model updated, i.e, fields, getter/setters, constructors, equals method and hashCode method. Also whatever Type or Field annotation not managed by the Data Modeler will be preserved when the Java sources are updated by the Data modeller.

Aside from code preservation, like in the other workbench editors, concurrent modification scenarios are still possible. Common scenarios are when two different users are updating the model for the same project, e.g. using the data modeller or executing a 'git push command' that modifies project sources.

From an application context's perspective, we can basically identify two different main scenarios:

18.7.7.7.1. No changes have been undertaken through the application

In this scenario the application user has basically just been navigating through the data model, without making any changes to it. Meanwhile, another user modifies the data model externally.

In this case, no immediate warning is issued to the application user. However, as soon as the user tries to make any kind of change, such as add or remove data objects or properties, or change any of the existing ones, the following pop-up will be shown:

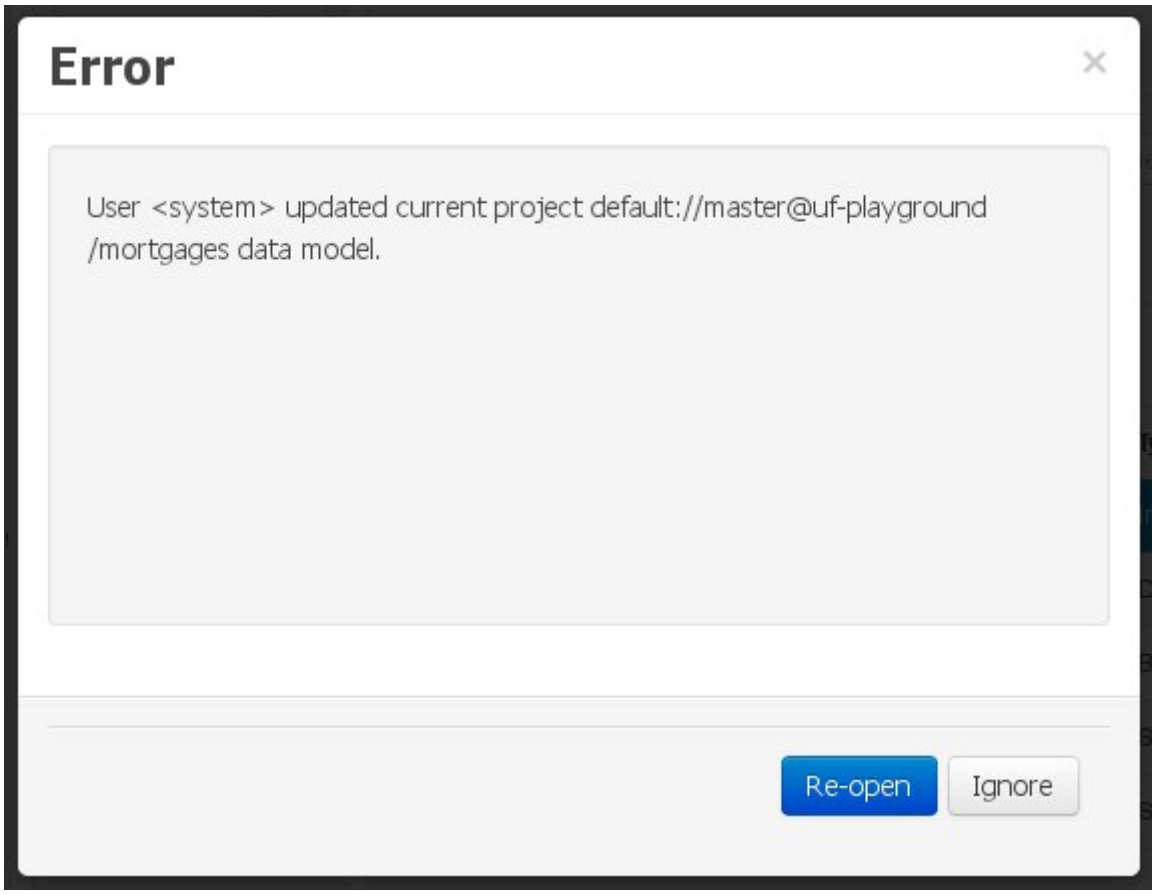


Figure 18.93. External changes warning

The user can choose to either:

- Re-open the data model, thus loading any external changes, and then perform the modification he was about to undertake, or
- Ignore any external changes, and go ahead with the modification to the model. In this case, when trying to persist these changes, another pop-up warning will be shown:

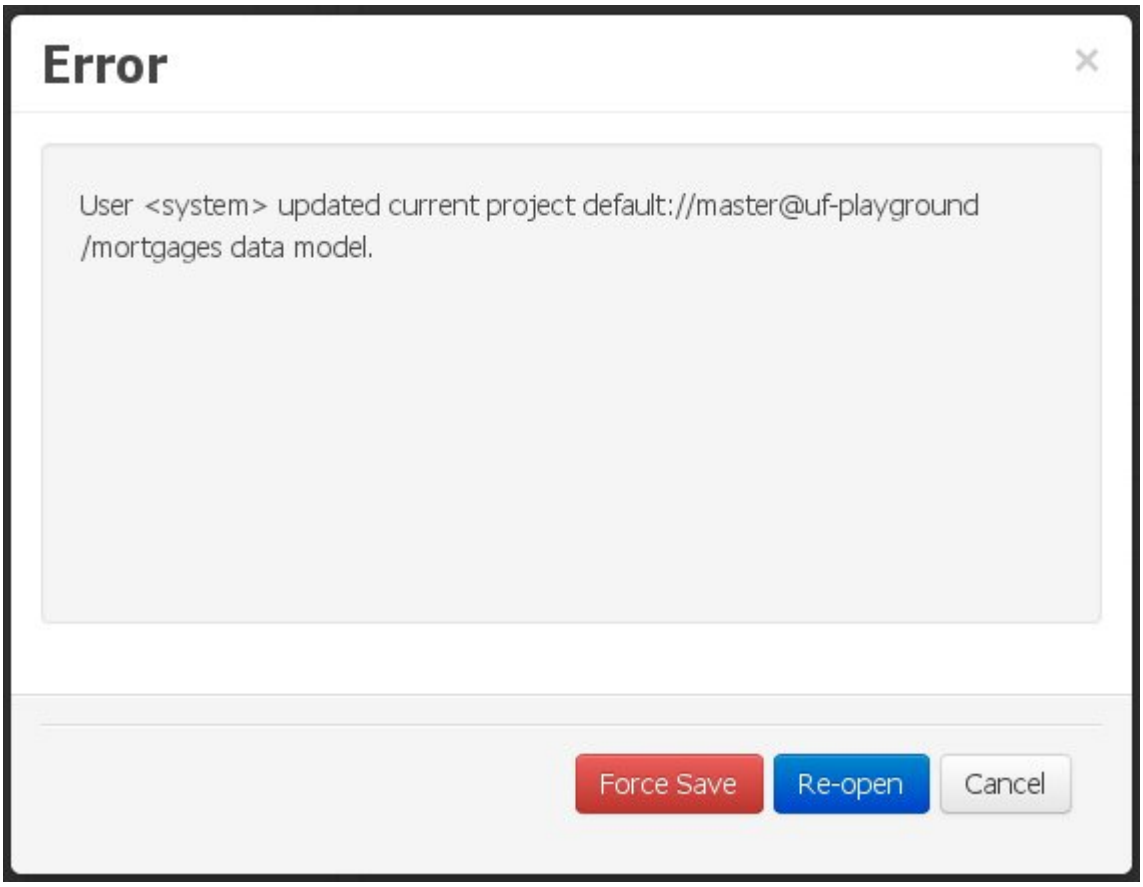


Figure 18.94. Force save / re-open

The "Force Save" option will effectively overwrite any external changes, while "Re-open" will discard any local changes and reload the model.

Warning

"Force Save" overwrites any external changes!

18.7.7.7.2. Changes have been undertaken through the application

The application user has made changes to the data model. Meanwhile, another user simultaneously modifies the data model from outside the application context.

In this alternative scenario, immediately after the external user commits his changes to the asset repository (or e.g. saves the model with the data modeller in a different session), a warning is issued to the application user:

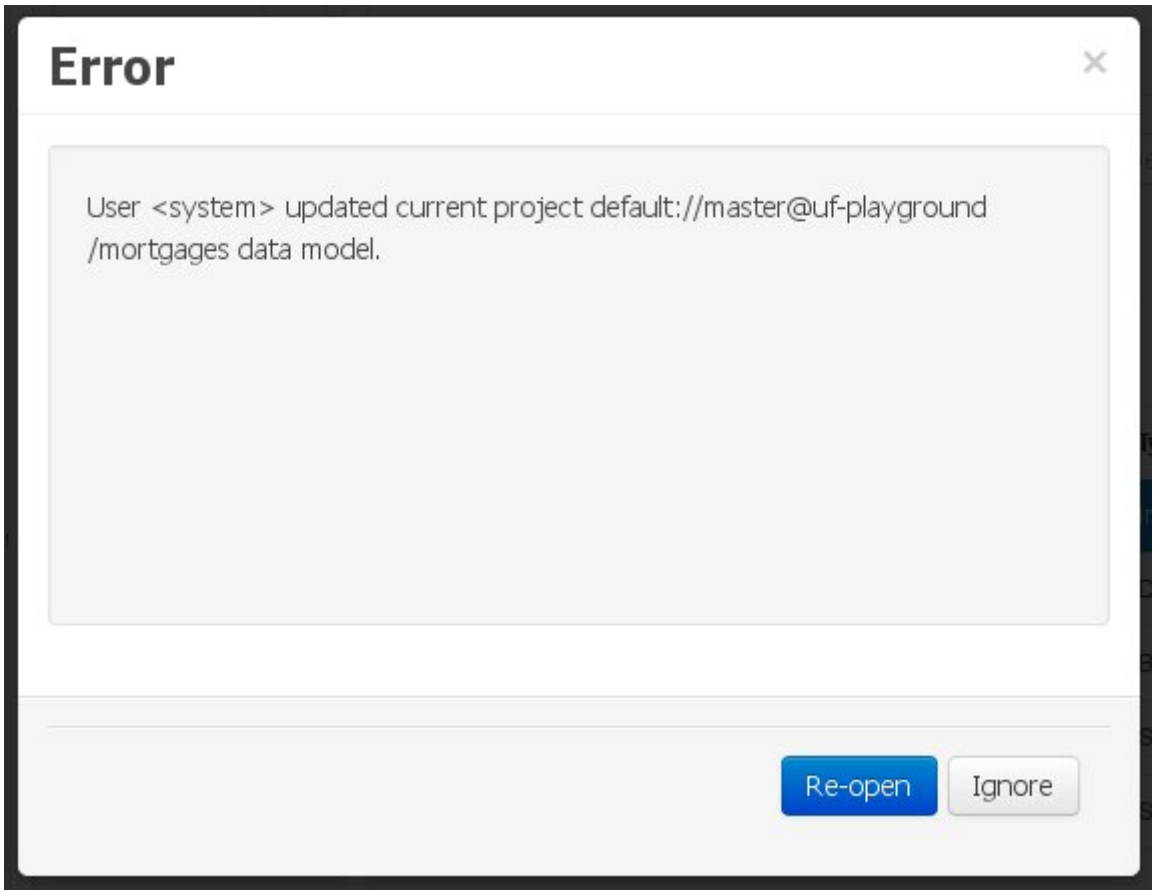


Figure 18.95. External changes warning

As with the previous scenario, the user can choose to either:

- Re-open the data model, thus losing any modifications that where made through the application, or
- Ignore any external changes, and continue working on the model.

  One of the following possibilities can now occur:

  - The user tries to persist the changes he made to the model by clicking the "Save" button in the data modeller top level menu. This leads to the following warning message:

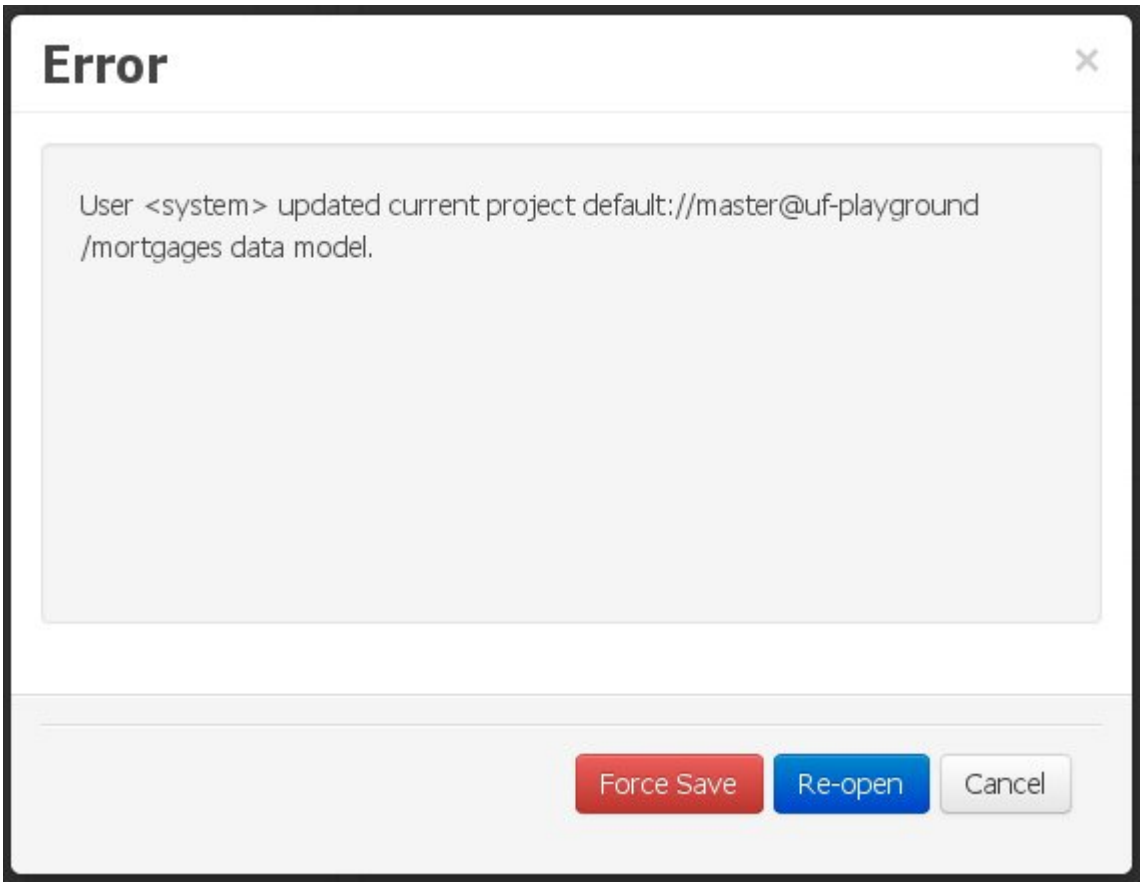


Figure 18.96. Force save / re-open

The "Force Save" option will effectively overwrite any external changes, while "Re-open" will discard any local changes and reload the model.

18.7.8. Data Sets

A **data set** is basically a set of columns populated with some rows, a matrix of data composed of timestamps, texts and numbers. A data set can be stored in different systems: a database, an excel file, in memory or in a lot of other different systems. On the other hand, a **data set definition** tells the workbench modules how such data can be accessed, read and parsed.

Notice, it's very important to make crystal clear the difference between a data set and its definition since the workbench does not take care of storing any data, it just provides an standard way to define access to those data sets regardless where the data is stored.

Let's take for instance the data stored in a remote database. A valid data set could be, for example, an entire database table or the result of an SQL query. In both cases, the database will return a bunch of columns and rows. Now, imagine we want to get access to such data to feed some charts in a new workbench perspective. First thing is to create and register a data set definition in order to indicate the following:

- where the data set is stored,
- how can be accessed, read and parsed and
- what columns contains and of which type.

This chapter introduces the available workbench tools for registering and handling data set definitions and how this definitions can be consumed in other workbench modules like, for instance, the Perspective Editor.

Note

For simplicity sake we will be using the term *data set* to refer to the actual data set definitions as *Data set* and *Data set definition* can be considered synonyms under the data set authoring context.

18.7.8.1. Data Set Authoring Perspective

Everything related to the authoring of data sets can be found under the *Data Set Authoring* perspective which is accessible from the following top level menu entry: *Extensions>Data Sets*, as shown in the following screenshot.

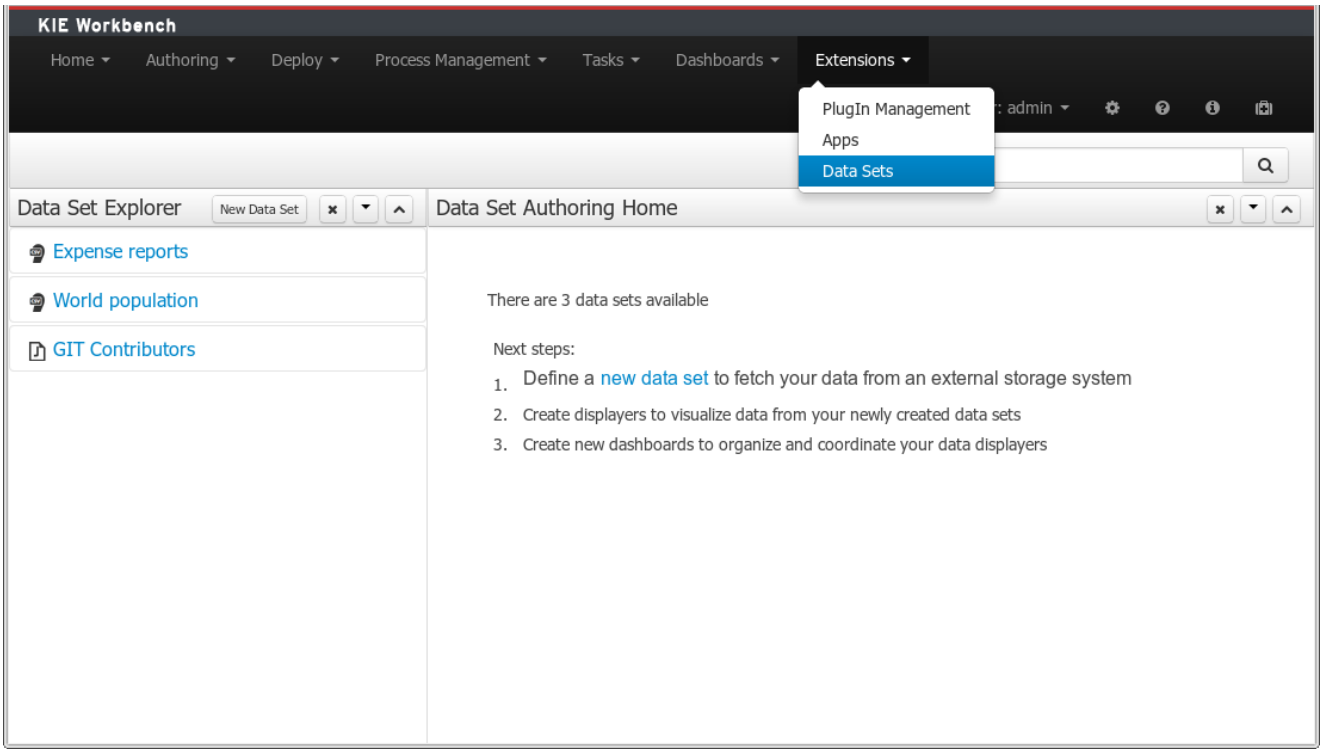


Figure 18.97. Data Set Authoring Perspective

The center panel, shows a welcome screen, whilst the left panel contains the *Data Set Explorer* listing all the data sets available

Note

This perspective is only intended to Administrator users, since defining data sets can be considered a low level task.

18.7.8.2. Data Set Explorer

The *Data Set Explorer* list the data sets present in the system. Every time the user clicks on the data set it shows a brief summary alongside the following information:

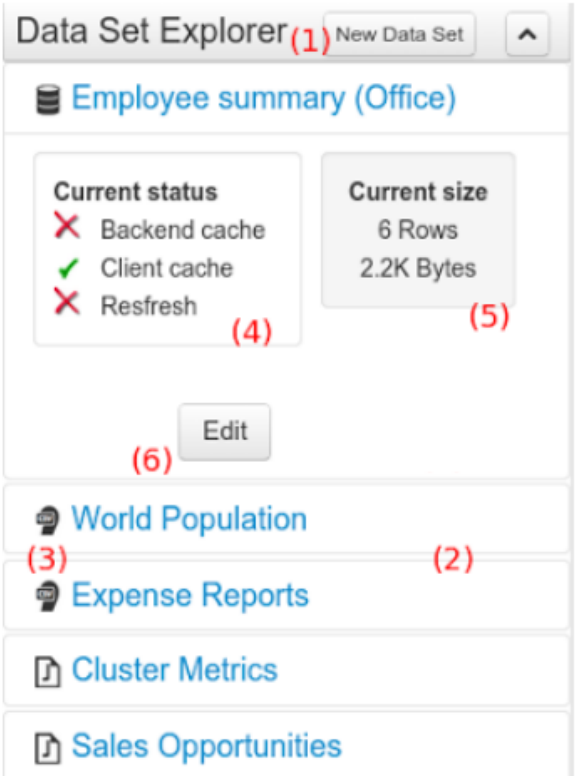


Figure 18.98. Data Set Explorer

- (1) A button for creating a new Data set
- (2) The list of currently available Data sets
- (3) An icon that represents the Data set's provider type (Bean, SQL, CSV, etc)
- (4) Details of current cache and refresh policy status
- (5) Details of current size on backend (unit as rows) and current size on client side (unit in bytes)
- (6) The button for editing the Data set. Once clicked the Data set editor screen is opened on the center panel

The next sections explains how to create, edit and fine tune data set definitions.

18.7.8.3. Data Set Creation

Clicking on the *New Data Set* button opens a new screen from which the user is able to create a new data set definition in three steps:

- *Provider type selection*

  Specify the kind of the remote storage system (BEAN, SQL, CSV, ElasticSearch)

- *Provider configuration*

  Specify the attributes for being able to look up data from the remote system. The configuration varies depending on the data provider type selected.

- *Data set columns & filter*

  Live data preview, column types and initial filter configuration.

18.7.8.3.1. Step 1: Provider type selection

Allows the user's specify the type of data provider of the data set being created.

This screen lists all the current available data provider types and helper popovers with descriptions. Each data provider is represented with a descriptive image:

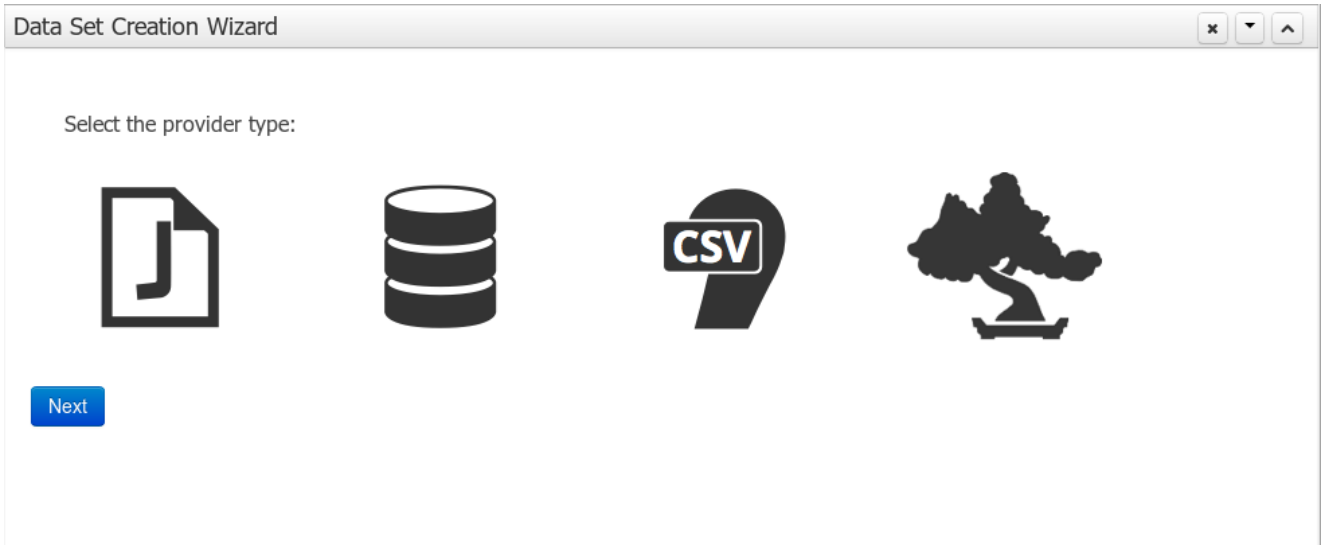


Figure 18.99. Provider type selection

Four types are currently supported:

- Bean (Java class) - To generate a data set directly from Java

- SQL - For getting data from any ANSI-SQL compliant database
- CSV - To upload the contents of a remote or local CSV file
- Elastic Search - To query and get documents stored on Elastic Search nodes as data sets

Once a type is selected, click on Next button to continue with the next workflow step.

18.7.8.3.2. Step 2: Configuration

The screenshot below shows a CSV data set configuration form. Once all the required settings are filled click on Test button. The system will try to fetch an small amount of data before moving to the next workflow step.

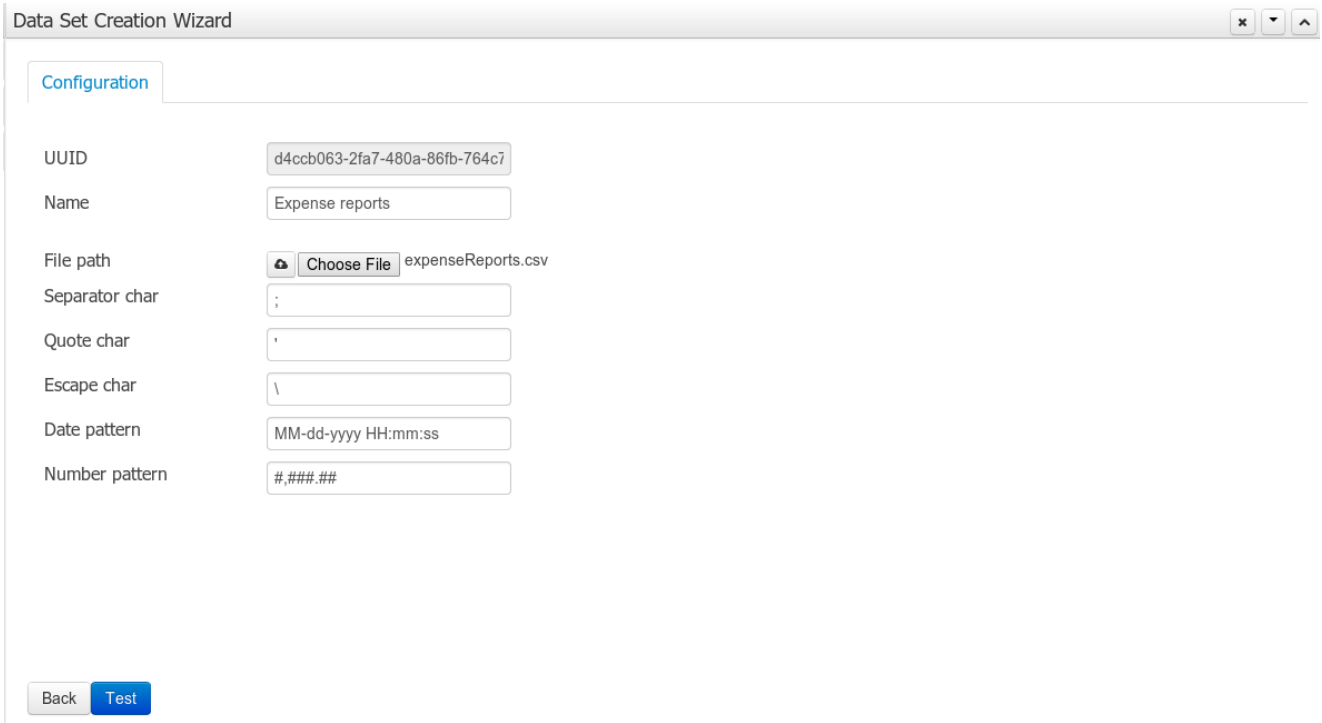


Figure 18.100. CSV Configuration

The provider type selected in the previous step will determine which configuration settings the system asks for.

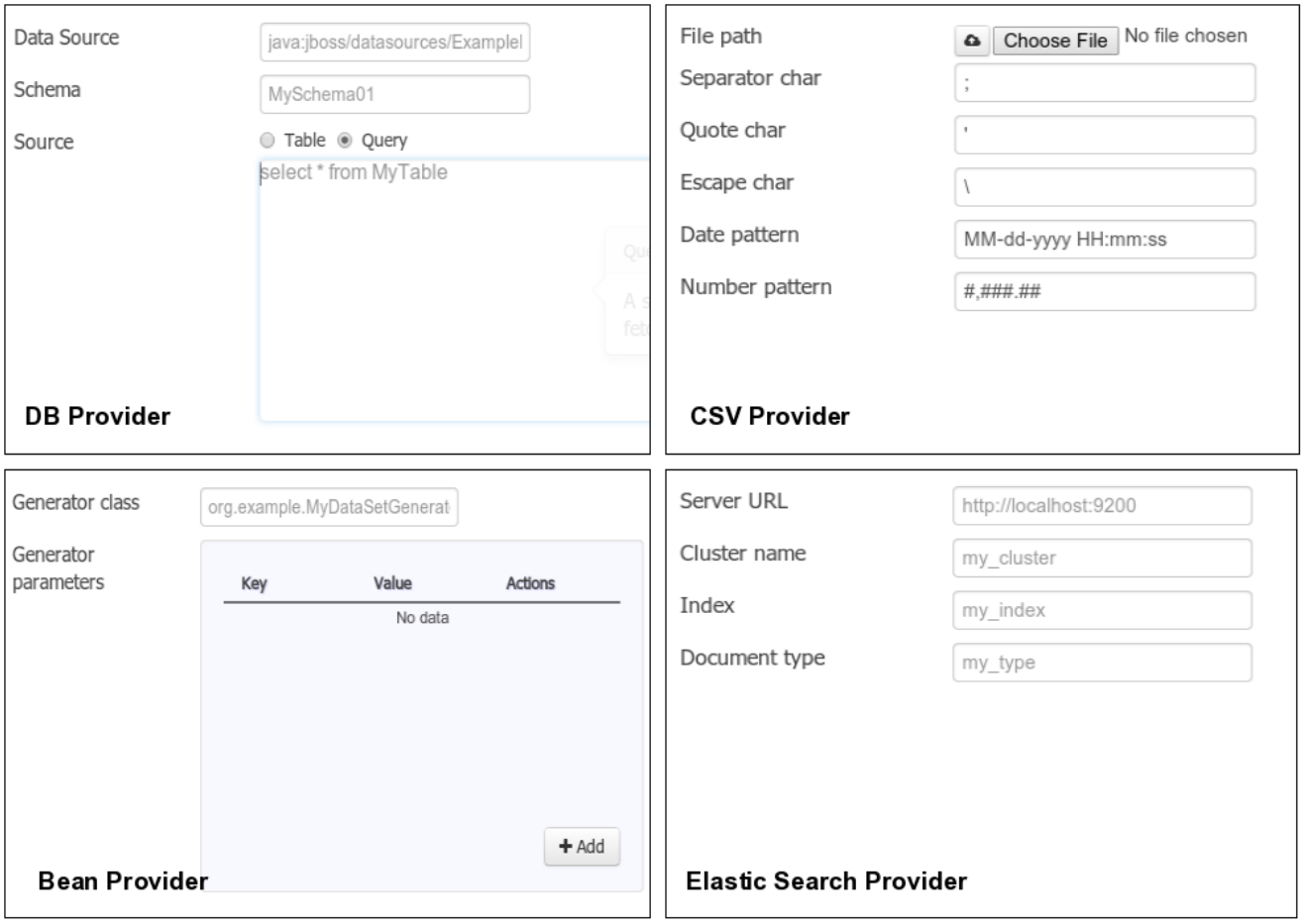


Figure 18.101. Configuration screen per data set type

Note

The UUID attribute is a read only field as it's generated by the system. It's only intended for usage in API calls or specific operations.

18.7.8.3.3. Step 3: Data set columns and preview

After clicking on the *Test* button (see previous step), the system executes a data set lookup test call in order to check if the remote system is up and the data is available. If everything goes ok the user will see the following screen:

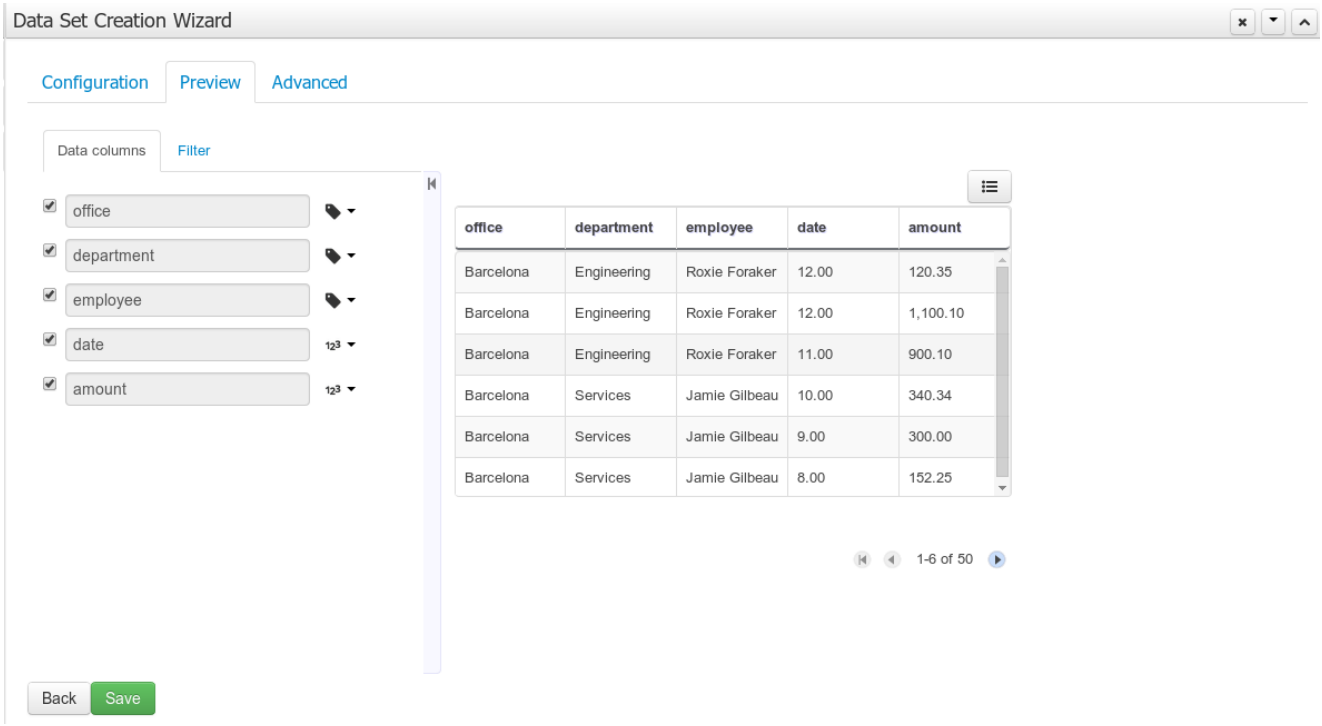


Figure 18.102. Data set preview

This screen shows a live data preview along with the columns the user wants to be part of the resulting data set. The user can also navigate through the data and apply some changes to the data set structure. Once finished, we can click on the *Save* button in order to register the new data set definition.

We can also change the configuration settings at any time just by going back to the configuration tab. We can repeat the *Configuration>Test>Preview* cycle as may times as needed until we consider it's ready to be saved.

Columns

In the *Columns* tab area the user can select what columns are part of the resulting data set definition.

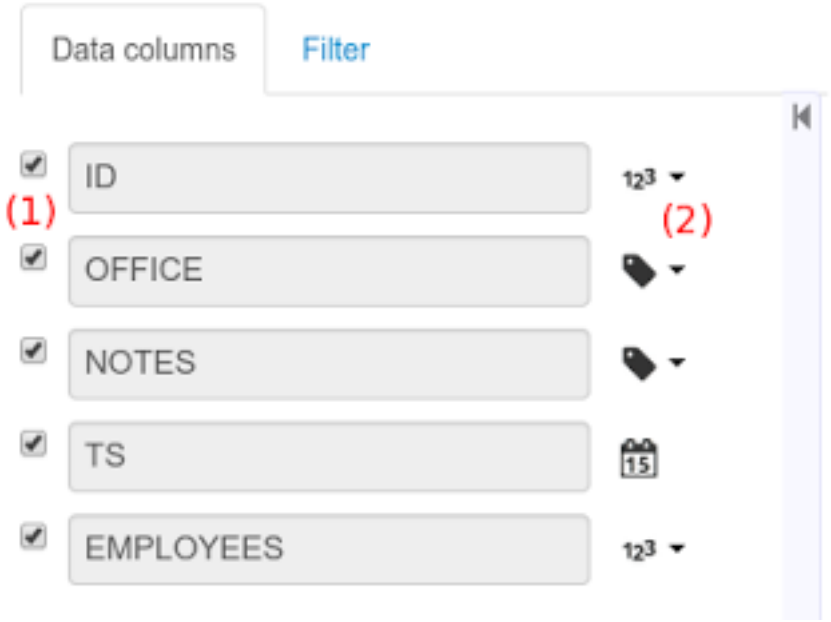


Figure 18.103. Data set columns

- (1) To add or remove columns. Select only those columns you want to be part of the resulting data set
- (2) Use the drop down image selector to change the column type

A data set may only contain columns of any of the following 4 types:

- Label - For text values supporting group operations (similar to the SQL "group by" operator) which means you can perform data lookup calls and get one row per distinct value.
- Text - For text values NOT supporting group operations. Typically for modeling large text columns such as abstracts, descriptions and the like.
- Number - For numeric values. It does support aggregation functions on data lookup calls: sum, min, max, average, count, disctinct.
- Date - For date or timestamp values. It does support time based group operations by different time intervals: minute, hour, day, month, year, ...

No matter which remote system you want to retrieve data from, the resulting data set will always return a set of columns of one of the four types above. There exists, by default, a mapping between the remote system column types and the data set types. The user is able to modify the type for some columns, depending on the data provider and the column type of the remote system. The system supports the following changes to column types:

- Label <> Text - Useful when we want to enable/disable the categorization (grouping) for the target column. For instance, imagine a database table called "document" containing a large text

column called "abstract". As we do not want the system to treat such column as a "label" we might change its column type to "text". Doing so, we are optimizing the way the system handles the data set and

- Number <> Label - Useful when we want to treat numeric columns as labels. This can be used for instance to indicate that a given numeric column is not a numeric value that can be used in aggregation functions. Despite its values are stored as numbers we want to handle the column as a "label". One example of such columns are: an item's code, an appraisal id., ...

Note

BEAN data sets do not support changing column types as it's up to the developer to decide which are the concrete types for each column.

Filter

A data set definition may define a filter. The goal of the filter is to leave out rows the user does not consider necessary. The filter feature works on any data provider type and it lets the user to apply filter operations on any of the data set columns available.

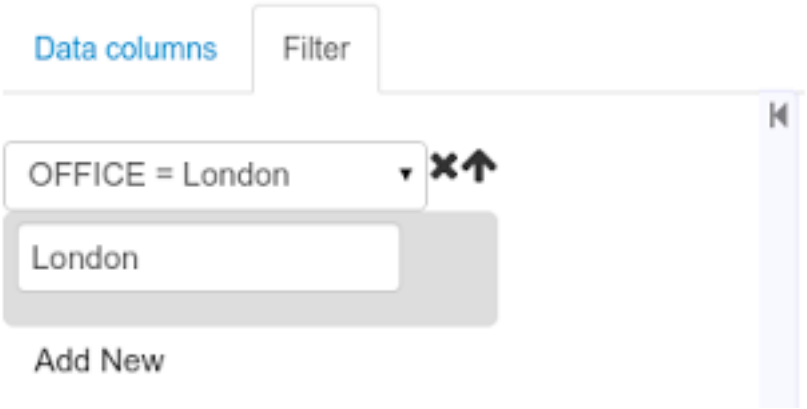


Figure 18.104. Data set filter

While adding or removing filter conditions and operations, the preview table on central area is updated with live data that reflects the current filter status.

There exists two strategies for filtering data sets and it's also important to note that choosing between the two have important implications. Imagine a dashboard with some charts feeding from a expense reports data set where such data set is built on top of an SQL table. Imagine also we only want to retrieve the expense reports from the "London" office. You may define a data set containing the filter "office=London" and then having several charts feeding from such data set. This is the recommended approach. Another option is to define a data set with no initial filter and then let the individual charts to specify their own filter. It's up to the user to decide on the best approach.

Depending on the case it might be better to define the filter at a data set level for reusing across other modules. The decision may also have impact on the performance since a filtered cached data set will have far better performance than a lot of individual non-cached data set lookup requests. (See the next section for more information about caching data sets).

Note

Notice, for SQL data sets, the user can use both the filter feature introduced or, alternatively, just add custom filter criteria to the SQL sentence. Although, the first approach is more appropriated for non technical users since they might not have the required SQL language skills.

18.7.8.4. Data set editor

To edit an existing data set definition go the data set explorer, expand the desired data set definition and click on the *Edit* button. This will cause a new editor panel to be opened and placed on the center of the screen, as shown in the next screenshot:

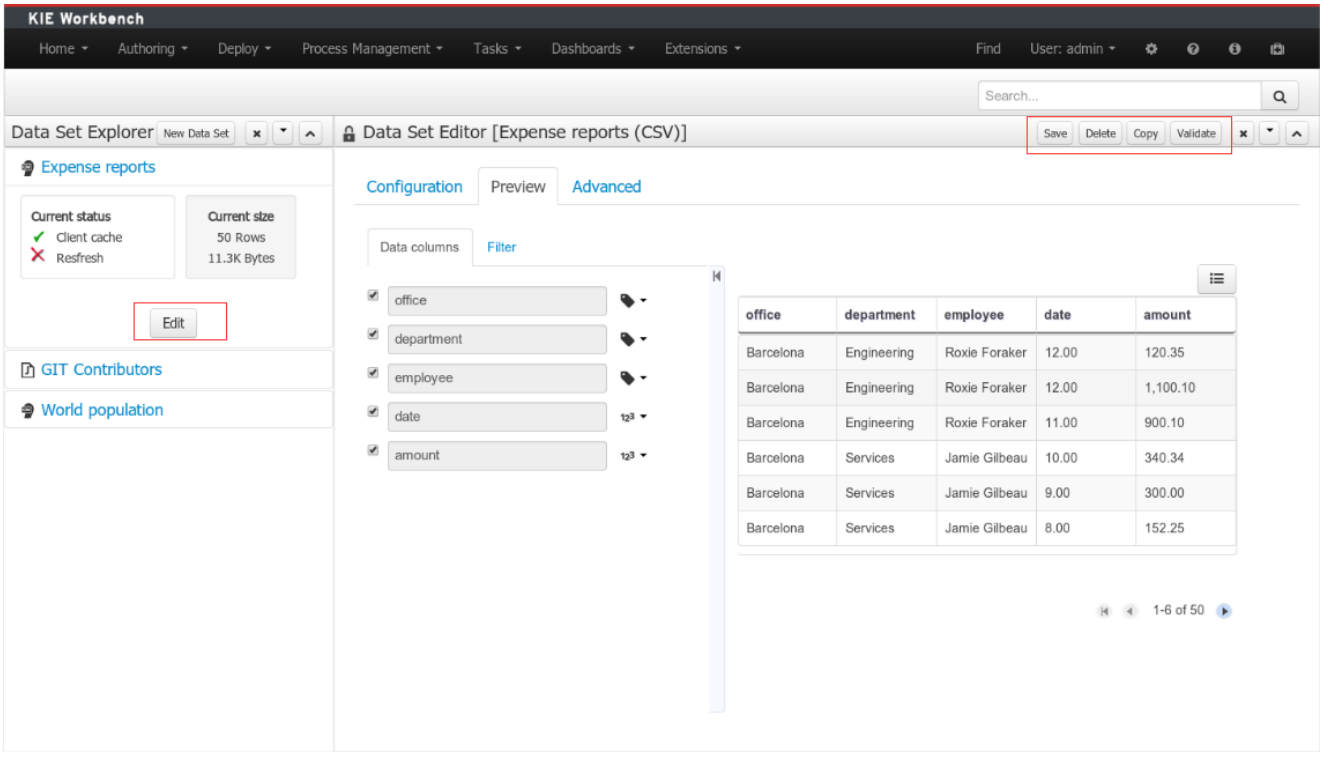


Figure 18.105. Data set definition editor

Every time we edit an item its editor is added to the center panel. We can navigate through the list of opened editors just by clicking on the down arrow icon placed at the editor's toolbar in the top right corner.

Figure 18.106. Editor selector

The editor provides all the features described in previous sections. We can change the configuration settings, test our data set definition and modify the resulting data set structure. Additionally, the editor provides some extra buttons in its toolbar:

- Save - To validate the current changes and store the data set definition.
- Delete - To remove permanently from storage the data set definition. Any client module referencing the data set may be affected.
- Validate - To check that all the required parameters exists and are correct, as well as to validate the data set can be retrieved with no issues.
- Copy - To create a brand new definition as a copy of the current one.

Note

Data set definitions are stored in the underlying GIT repository as JSON files. Any action performed is registered in the repository logs so it is possible to audit the change log later on.

18.7.8.5. Advanced settings

In the *Advanced settings* tab area the user can specify caching and refresh settings. Those are very important for making the most of the system capabilities thus improving the performance and having better application responsive levels.

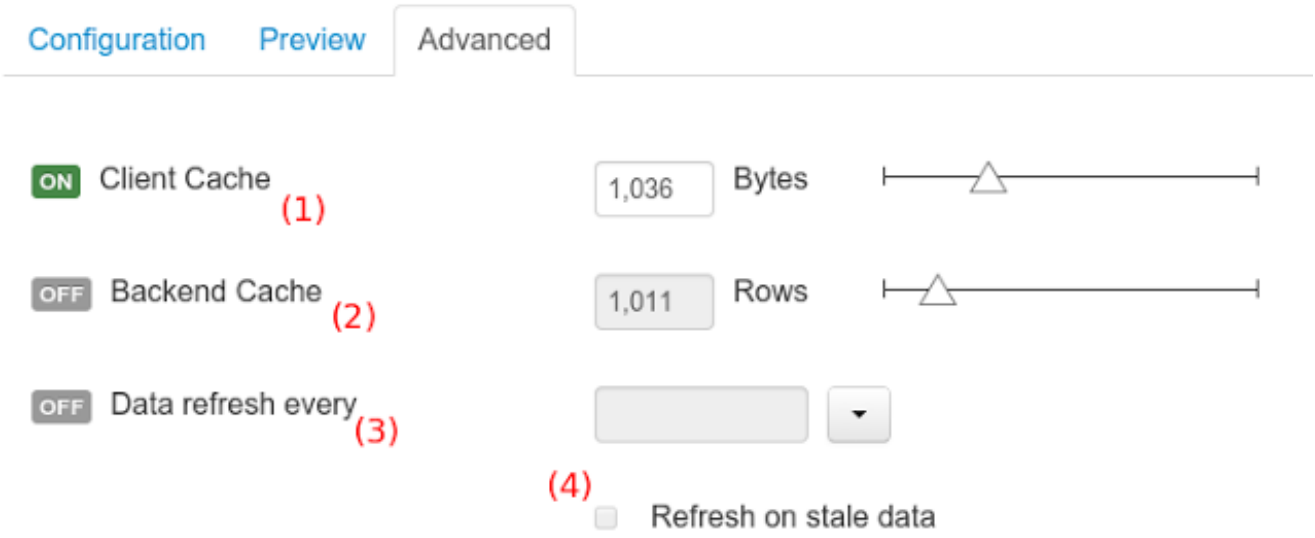


Figure 18.107. Advanced settings

- (1) To enable or disable the client cache and specify the maximum size (bytes).
- (2) To enable or disable the backend cache and specify the maximum cache size (number of rows).
- (3) To enable or disable automatic refresh for the Data set and the refresh period.
- (4) To enable or disable the refresh on stale data setting.

Let's dig into more details about the meaning of these settings.

18.7.8.6. Caching

The system provides caching mechanisms out-of-the-box for holding data sets and performing data operations using in-memory strategies. The use of these features brings a lot of advantages, like reducing the network traffic, remote system payload, processing times etc. On the other hand, it's up to the user to fine tune properly the caching settings to avoid hitting performance issues.

Two cache levels are supported:

- Client level
- Backend level

The following diagram shows how caching is involved in any data set operation:

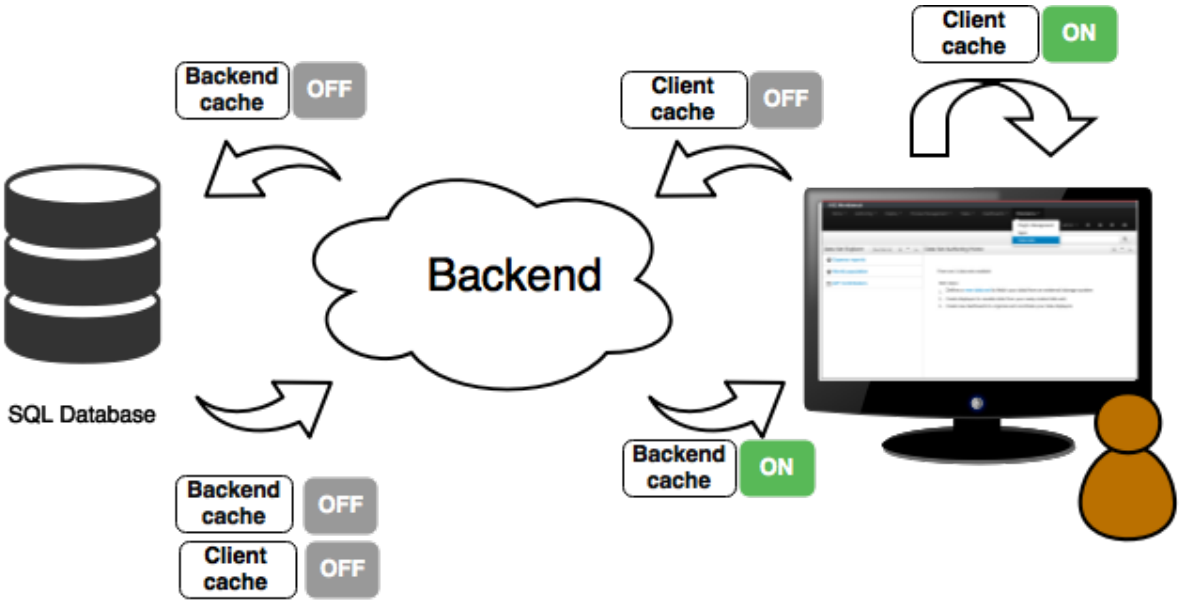


Figure 18.108. Data set caching

Any data look up call produces a resulting data set, so the use of the caching techniques determines where the data lookup calls are executed and where the resulting data set is located.

Client cache

If ON then the data set involved in a look up operation is pushed into the web browser so that all the components that feed from this data set **do not need to perform any requests to the backend** since data set operations are resolved at a client side:

- The data set is stored in the web browser's memory
- The client components feed from the data set stored in the browser
- Data set operations (grouping, aggregations, filters and sort) are processed within the web browser, by means of a Javascript data set operation engine.

If you know beforehand that your data set will remain small, you can enable the client cache. It will reduce the number of backend requests, including the requests to the storage system. On the other hand, if you consider that your data set will be quite big, disable the client cache so as to not hitting with browser issues such as slow performance or intermittent hangs.

Backend cache

Its goal is to provide a caching mechanism for data sets on backend side.

This feature allows to **reduce the number of requests to the remote storage system** , by holding the data set in memory and performing group, filter and sort operations using the in-memory engine.

It's useful for data sets that do not change very often and their size can be considered acceptable to be held and processed in memory. It can be also helpful on low latency connectivity issues with

the remote storage. On the other hand, if your data set is going to be updated frequently, it's better to disable the backend cache and perform the requests to the remote storage on each look up request, so the storage system is in charge of resolving the data set lookup request.

Note

BEAN and CSV data providers relies by default on the backend cache, as in both cases the data set must be always loaded into memory in order to resolve any data lookup operation using the in-memory engine. This is the reason why the backend settings are not visible in the Advanced settings tab.

18.7.8.7. Refresh

The refresh feature allows for the invalidation of any cached data when certain conditions are meet.

Figure 18.109. Refresh settings

- (1) To enable or disable the refresh feature.
- (2) To specify the refresh interval.
- (3) To enable or disable data set invalidation when the data is outdated.

The data set refresh policy is tightly related to data set caching, detailed in previous section. This invalidation mechanism determines the cache life-cycle.

Depending on the nature of the data there exist three main use cases:

- **Source data changes predictable** - Imagine a database being updated every night. In that case, the suggested configuration is to use a "refresh interval = 1 day" and disable "refresh on stale data". That way, the system will always invalidate the cached data set every day. This is the right configuration when we know in advance that the data is going to change.
- **Source data changes unpredictable** - On the other hand, if we do not know whether the database is updated every day, the suggested configuration is to use a "refresh interval = 1 day" and enable "refresh on stale data". If so the system, before invalidating any data, will check for modifications. On data modifications, the system will invalidate the current stale data set so that the cache is populated with fresh data on the next data set lookup call.

- **Real time scenarios** - In real time scenarios caching makes no sense as data is going to be updated constantly. In this kind of scenarios the data sent to the client has to be constantly updated, so rather than enabling the refresh settings (remember this settings affect the caching, and caching is not enabled) it's up to the clients consuming the data set to decide when to refresh. When the client is a dashboard then it's just a matter of modifying the refresh settings in the Displayer Editor configuration screen and set a proper refresh period, "refresh interval = 1 second" for example.

18.8. Embedding Workbench In Your Application

As we already know, Workbench provides a set of editors to author assets in different formats. According to asset's format a specialized editor is used.

One additional feature provided by Workbench is the ability to embed it in your own (Web) Applications thru it's **standalone** mode. So, if you want to edit rules, processes, decision tables, etc... in your own applications without switch to Workbench, you can.

In order to embed Workbench in your application all you'll need is the Workbench application deployed and running in a web/application server and, from within your own web applications, an iframe with proper HTTP query parameters as described in the following table.

Table 18.2. HTTP query parameters for standalone mode

Parameter Name	Explanation	Allow multiple values	Example
standalone	With just the presence of this parameter workbench will switch to standalone mode.	no	(none)
path	Path to the asset to be edited. Note that asset should already exists.	no	git://master@uf-playground/todo.md
perspective	Reference to an existing perspective name.	no	org.guvnor.m2repo.client.perspectives.Guv
header	Defines the name of the header that should be displayed (useful for context menu headers).	yes	ComplementNavArea

Note

Path and Perspective parameters are mutual exclusive, so can't be used together.

18.9. Asset Management

18.9.1. Asset Management Overview

This section of the documentation describes the main features included that contribute to the Asset Management functionality provided in the KIE Workbench and KIE Drools Workbench. All the features described here are entirely optional, but the usage is recommended if you are planning to have multiple projects. All the Asset Management features try to impose good practices on the repository structure that will make the maintainace, versioning and distribution of the projects simple and based on standards. All the Asset Management features are implemented using jBPM Business Processes, which means that the logic can be reused for external applications as well as adapted for domain specific requirements when needed.

Note

You must set the "kiemgmt" role to your user to be able to use the Asset Management Features

18.9.2. Managed vs Unmanaged Repositories

Since the creation of the assets management features repositories can be classified into Managed or Unmanaged.

18.9.2.1. Managed Repositories

All new assets management features are available for this type of repositories. Additionally a managed repository can be "Single Project" or "Multi Project".

A "Single Project" managed repository will contain just one Project. And a "Multi Project" managed repository can contain multiple Projects. All of them related through the same parent, and they will share the same group and version information.

18.9.2.2. Unmanaged Repositories

Assets management features are not available for this type or repositories and they basically behaves the same as the repositories created with previous workbench versions.

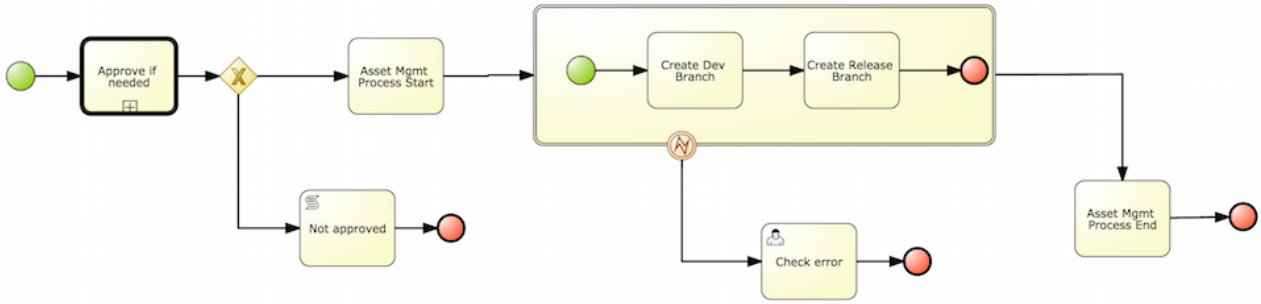
18.9.3. Asset Management Processes

There are 4 main processes which represent the stages of the Asset Management feature: Configure Repository, Promote Changes, Build and Release.

18.9.3.1. Configure Repository

The Configure Repository process is in charge of the post initialization of the repository. This process will be automatically triggered if the user selects to create a Managed Repository on the

New repository wizzard. If they decide to use the governance feature the process will kick in and as soon as the repository is created. A new development and release branches will be created. Notice that the first time that this process is called, the master branch is picked and both branches (dev and release) will be based on it.

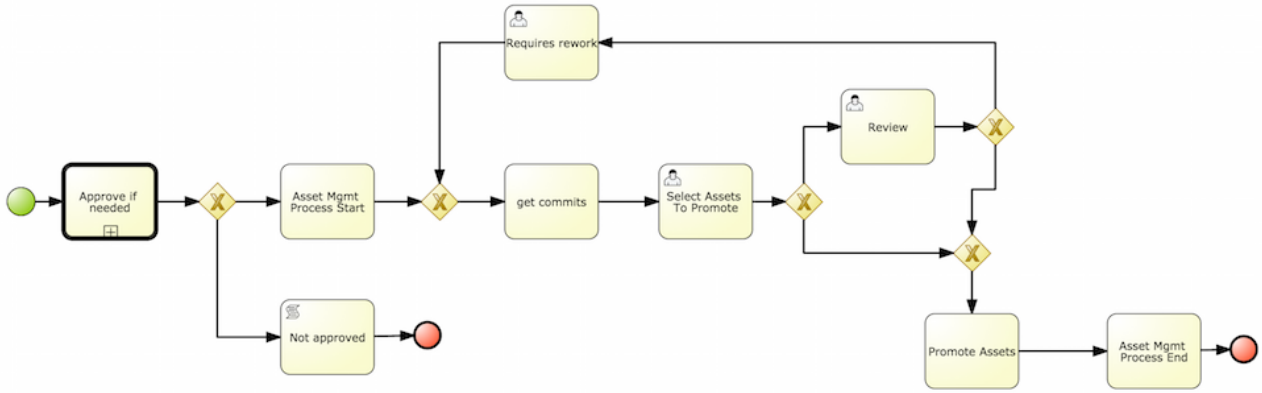


By default the asset management feature is not enabled so make sure to select Managed Repository on the New Repository Wizzard. When we work inside a managed repository, the development branch is selected for the users to work on. If multiple dev branches are created, the user will need to pick one.

18.9.3.2. Promote Changes Process

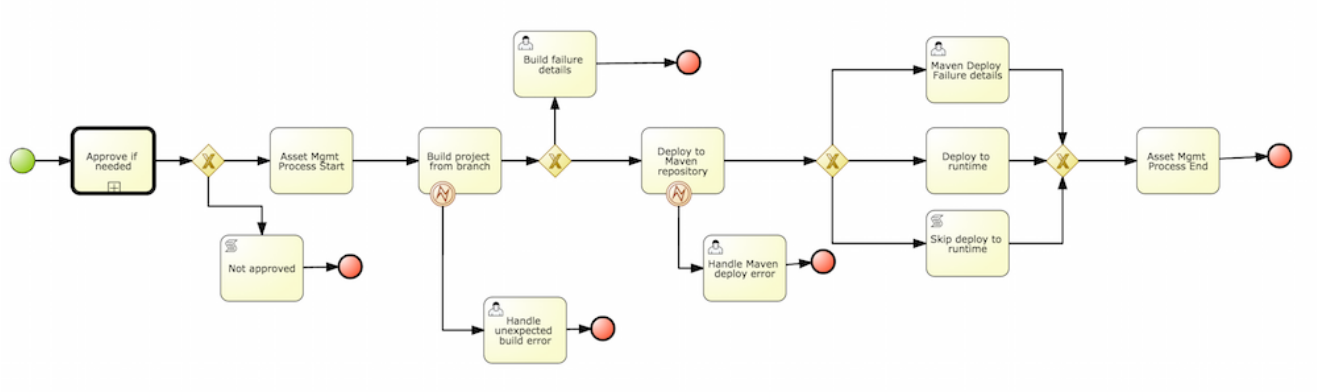
When some work is done in the developments branch and the users reach a point where the changes needs to be tested before going into production, they will start a new Promote Changes process so a more technical user can decide and review what needs to be promoted. The users belonging to the "kiemgmt" group will see a new Task in their Group Task List which will contain all the files that had being changed. The user needs to select the assets that will be promoting via the UI. The underlying process will be cherry-picking the commits selected by the user to the release branch. The user can specify that a review is needed by a more technical user.

This process can be repeated multiple times if needed before creating the artifacts for the release.



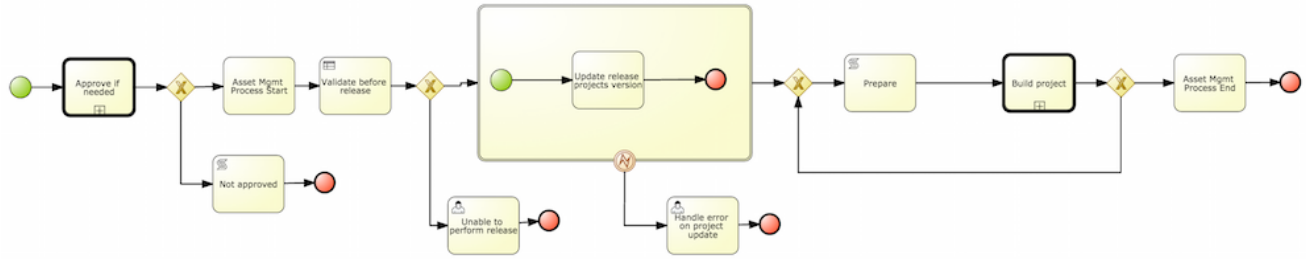
18.9.3.3. Build Process

The Build process can be triggered to build our projects from different branches. This allows us to have a more flexible way to build and deploy our projects to different runtimes.



18.9.3.4. Release Process

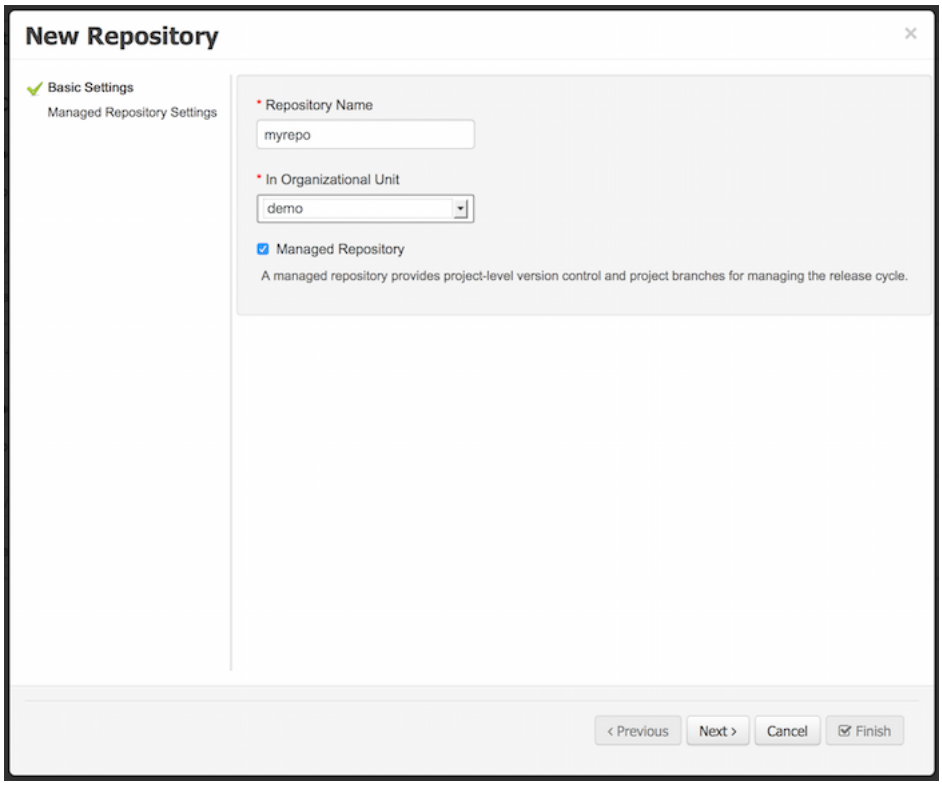
The release process is triggered at any time when the user decided that it is time to generate a release of the project that he/she is working on. This process will build the project (calling the Build Process) and it will update all the maven artifacts to the next version.



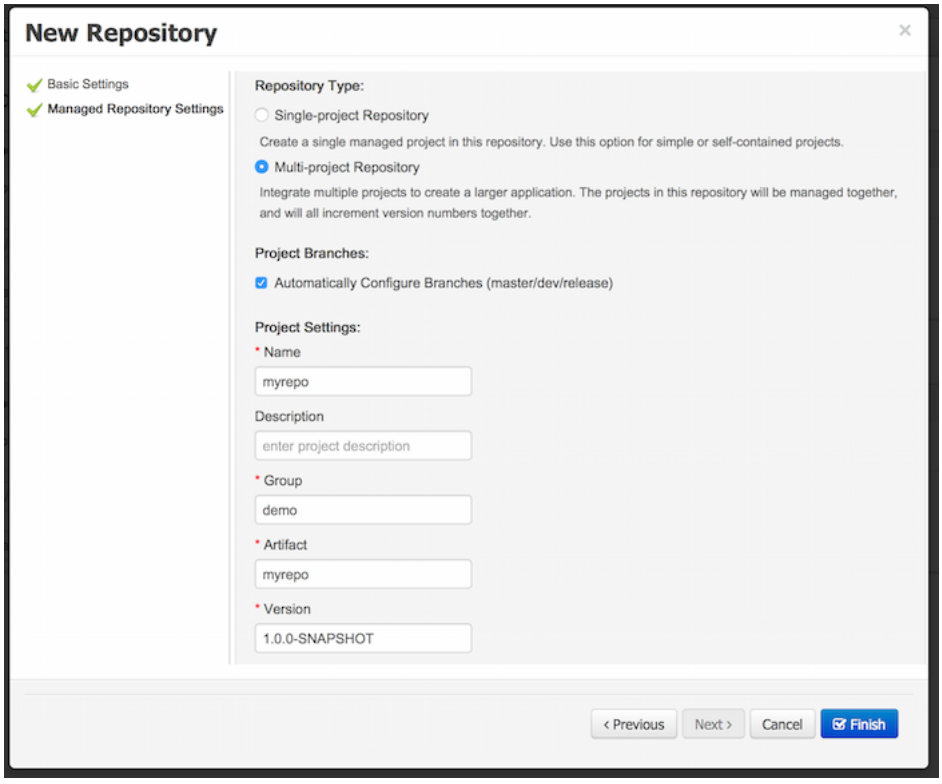
18.9.4. Usage Flow

This section describes the common usage flow for the asset management features showing all the screens involved.

The first contact with the Asset Management features starts on the Repository creation.



If the user chooses to create a Managed Respository a new page in the wizzard is enabled:



When a managed repository is created the assets management configuration process is automatically launched in order to create the repository branches, and the corresponding project structure is also created.

18.9.5. Repository Structure

Once a repository has been created it can be managed through the Repository Structure Screen.

To open the Repository Structure Screen for a given repository open the Project Authoring Perspective, browse to the given repository and select the "Repository -> Repository Structure" menu option.

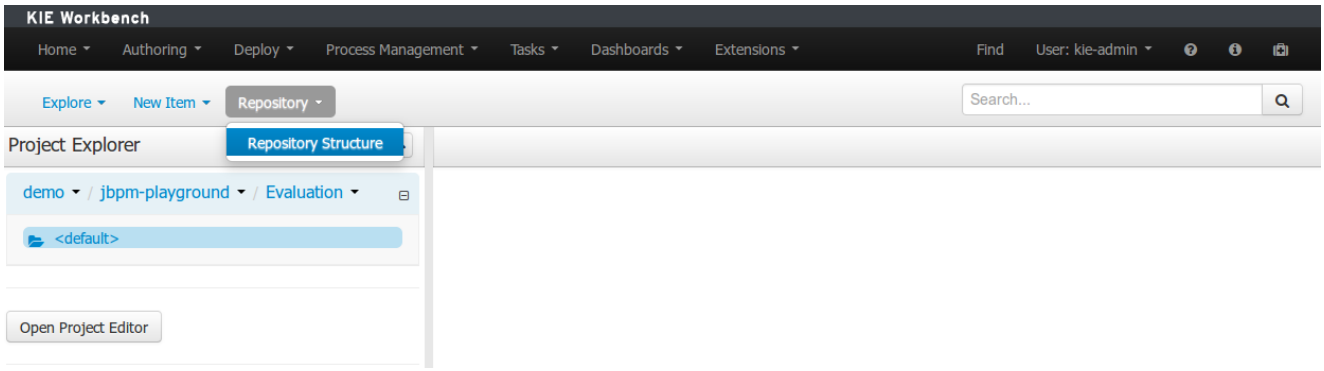


Figure 18.110. Repository Structure Menu

18.9.5.1. Single Project Managed Repository

The following picture shows an example of a single project managed repository structure.

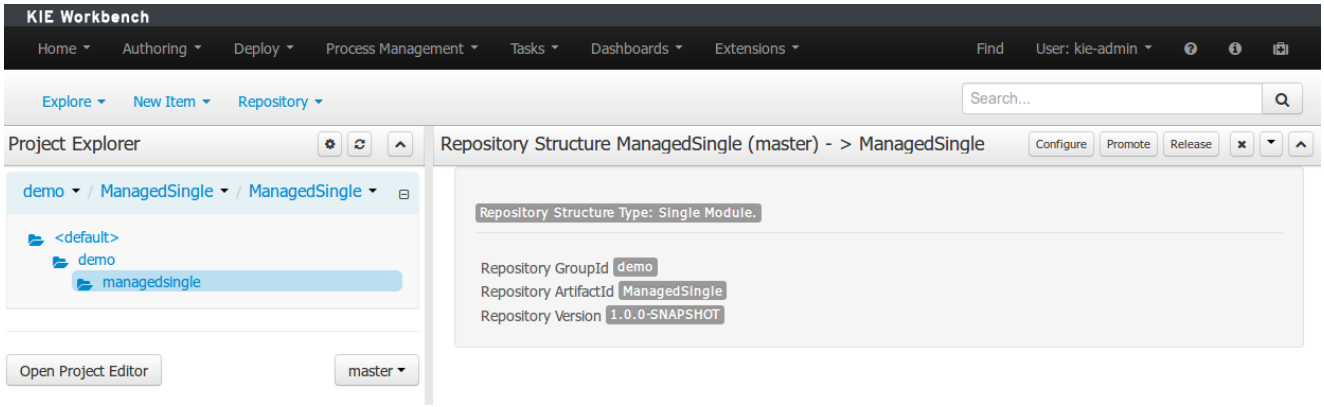


Figure 18.111. Single Project Managed Repository

18.9.5.2. Multi Project Managed Repository

The following picture shows an example of a multi project managed repository structure.

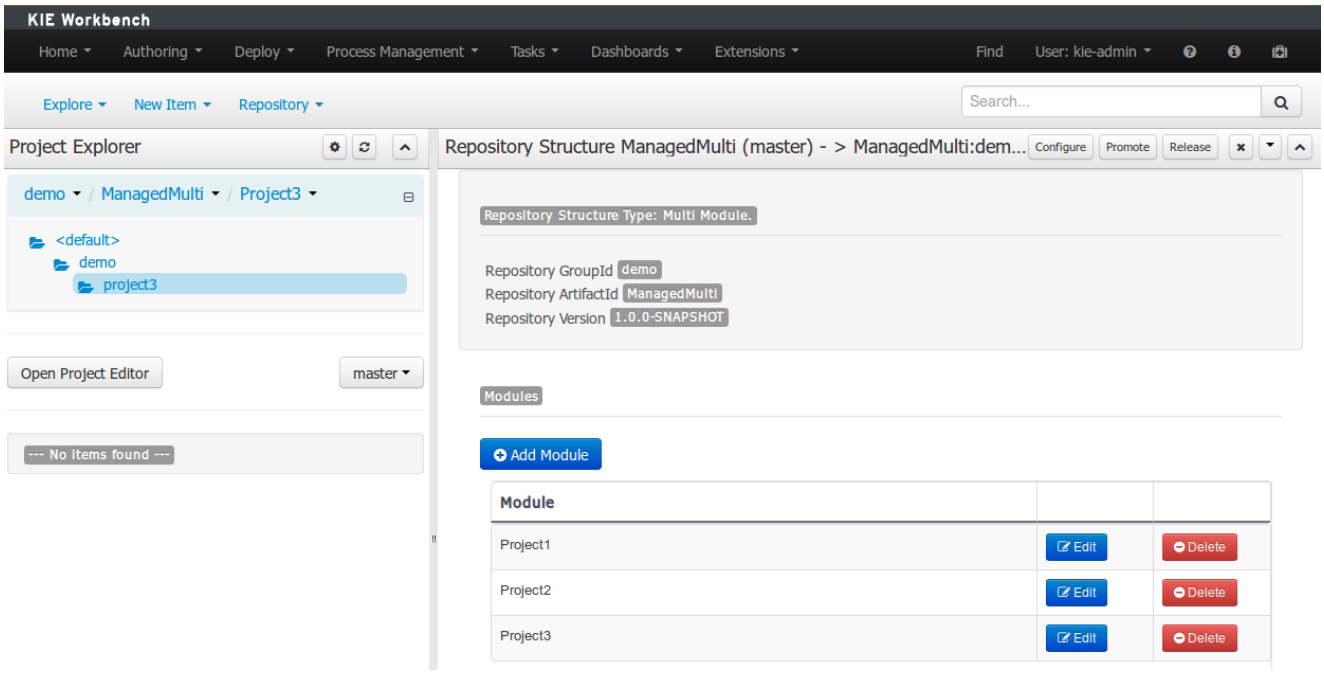


Figure 18.112. Multi Project Managed Repository

18.9.5.3. Unmanaged Repository

The following picture shows an example of an unmanaged repository structure.

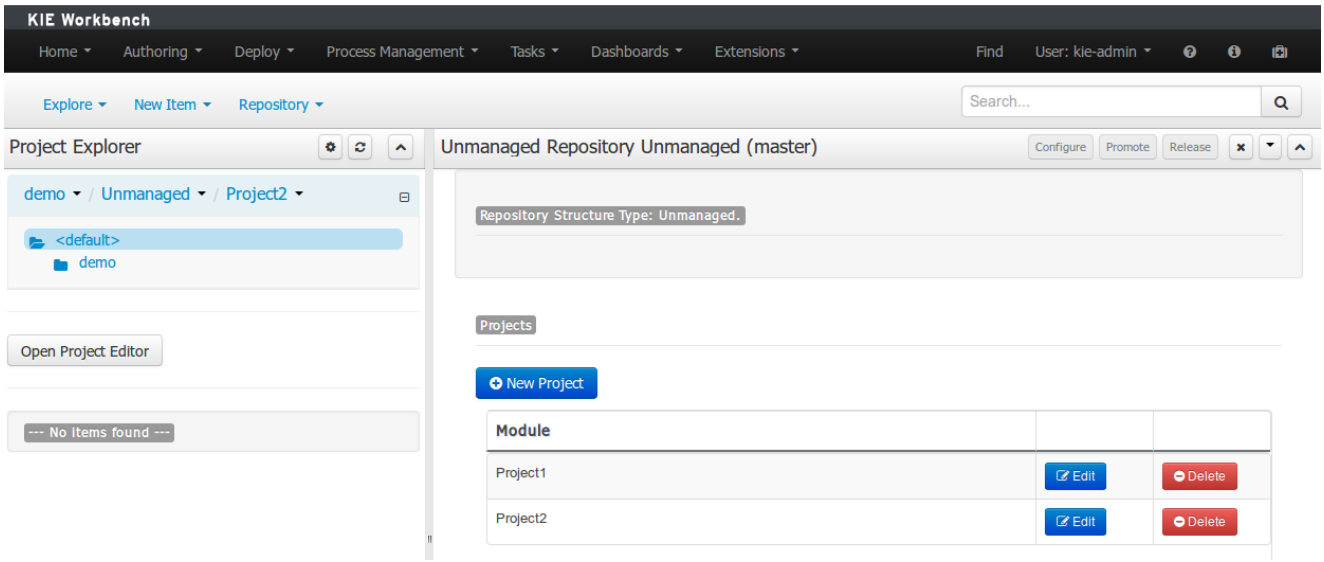


Figure 18.113. Unmanaged Repository

18.9.6. Managed Repositories Operations

The following picture shows the screen areas related to managed repositories operations.

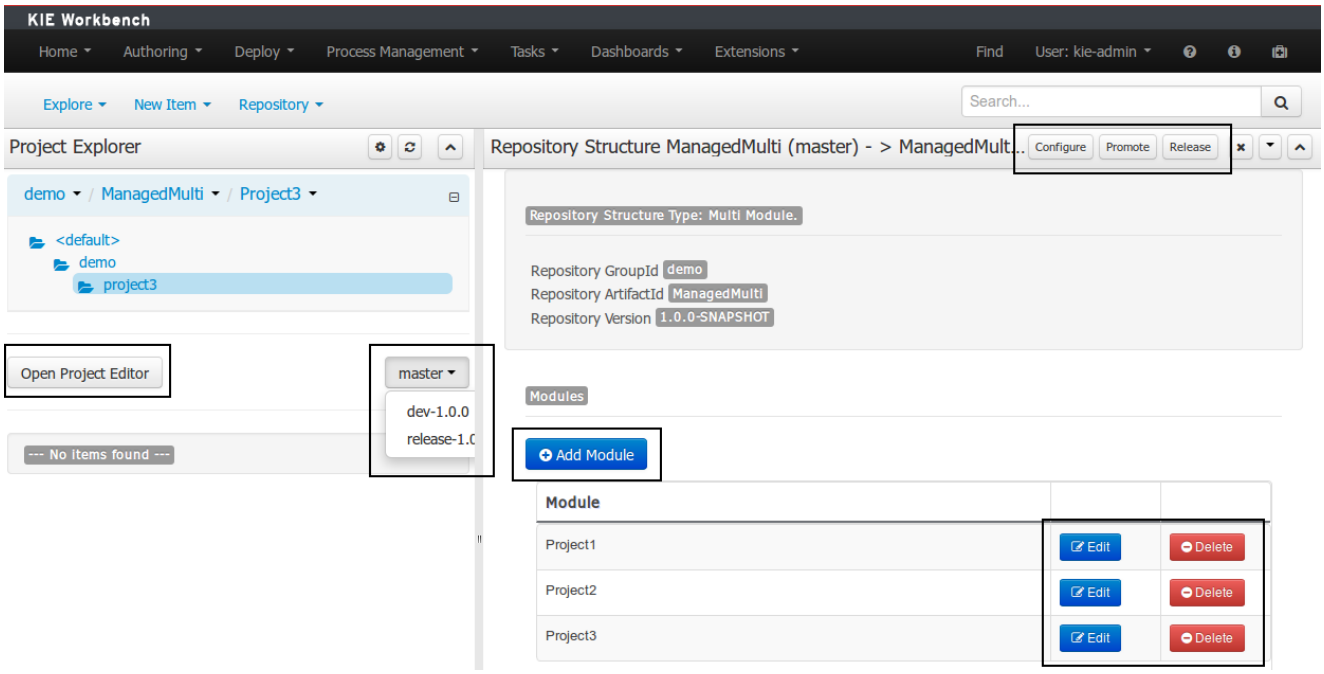


Figure 18.114. Managed Repositories Operations

18.9.6.1. Branch Selector

The branch selector lets to switch between the different branches created by the Configure Repository Process.

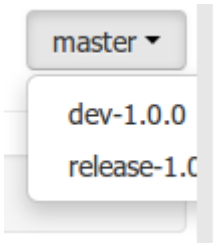


Figure 18.115. Branch Selector

18.9.6.2. Project Operations

From the repository structure screen it's also possible to create, edit or delete projects from current repository.

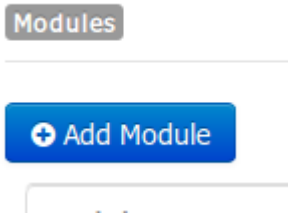


Figure 18.116. Add Project to current structure

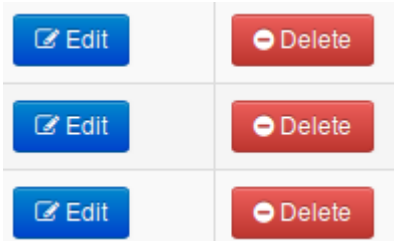


Figure 18.117. Edit/Delete projects from current structure

18.9.6.3. Launch Assets Management Processes

The assets management processes can also be launched from the Project Structure Screen.

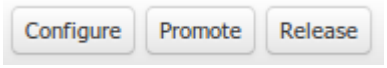


Figure 18.118. Launch Assets Management Processes

18.9.6.3.1. Launch the Configure Repository Process

Filling the parameters bellow a new instance of the Configure Repository can be started. (see Configure Repository Process)

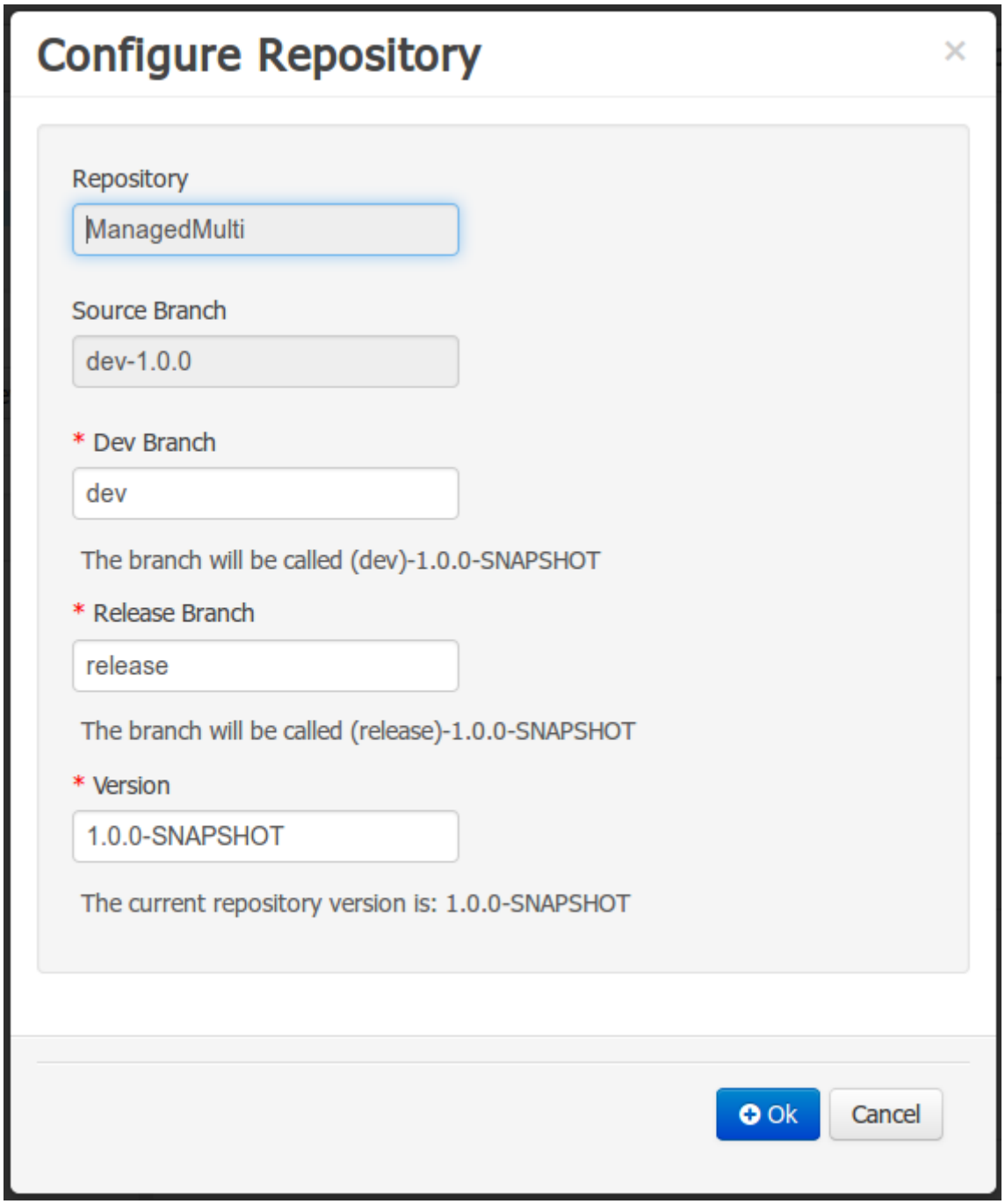


Figure 18.119. Configure Repository Process Parameters

18.9.6.3.2. Launch the Promote Changes Process

Filling the parameters bellow a new instance of the Promote Changes Process can be started. (see Promote Changes Process)

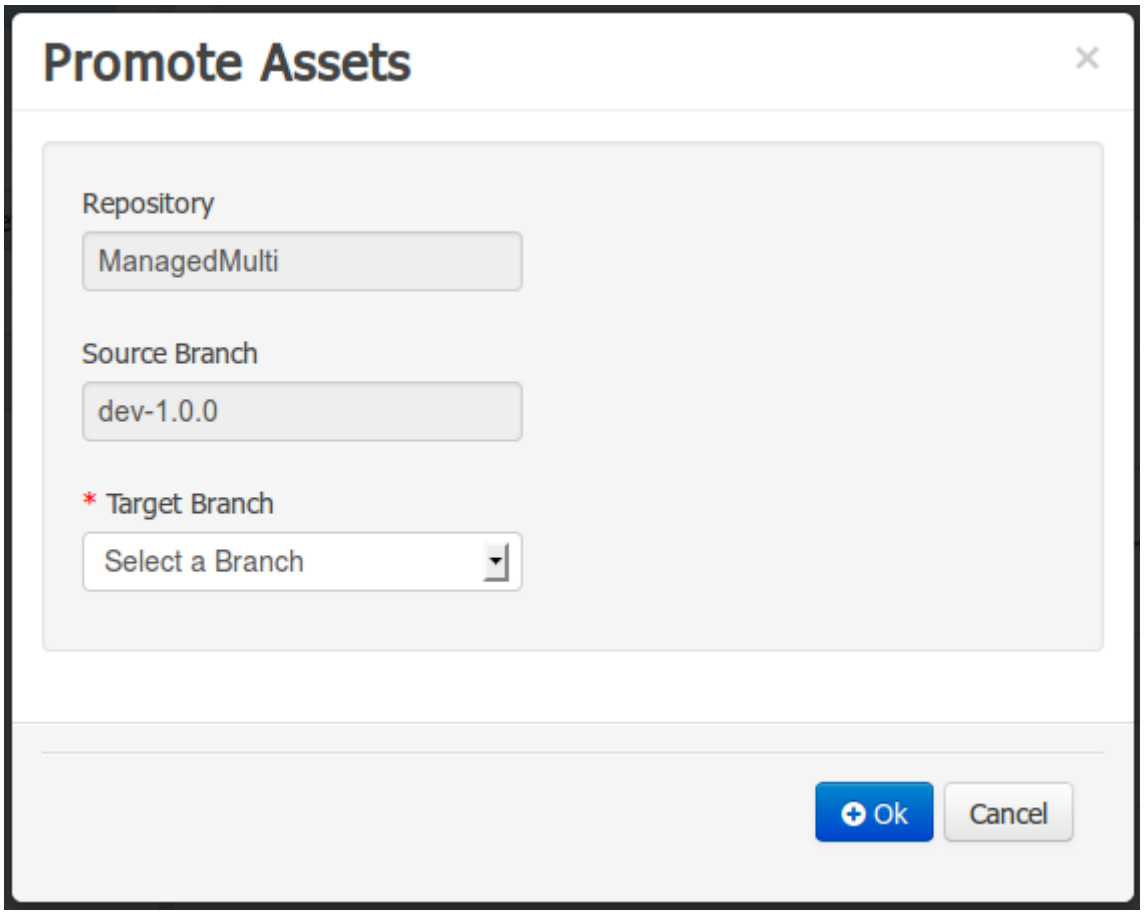


Figure 18.120. Promote Changes Process Parameters

18.9.6.3.3. Launch the Release Process

Filling the parameters bellow a new instance of the Release Process can be started. (see Release Process)

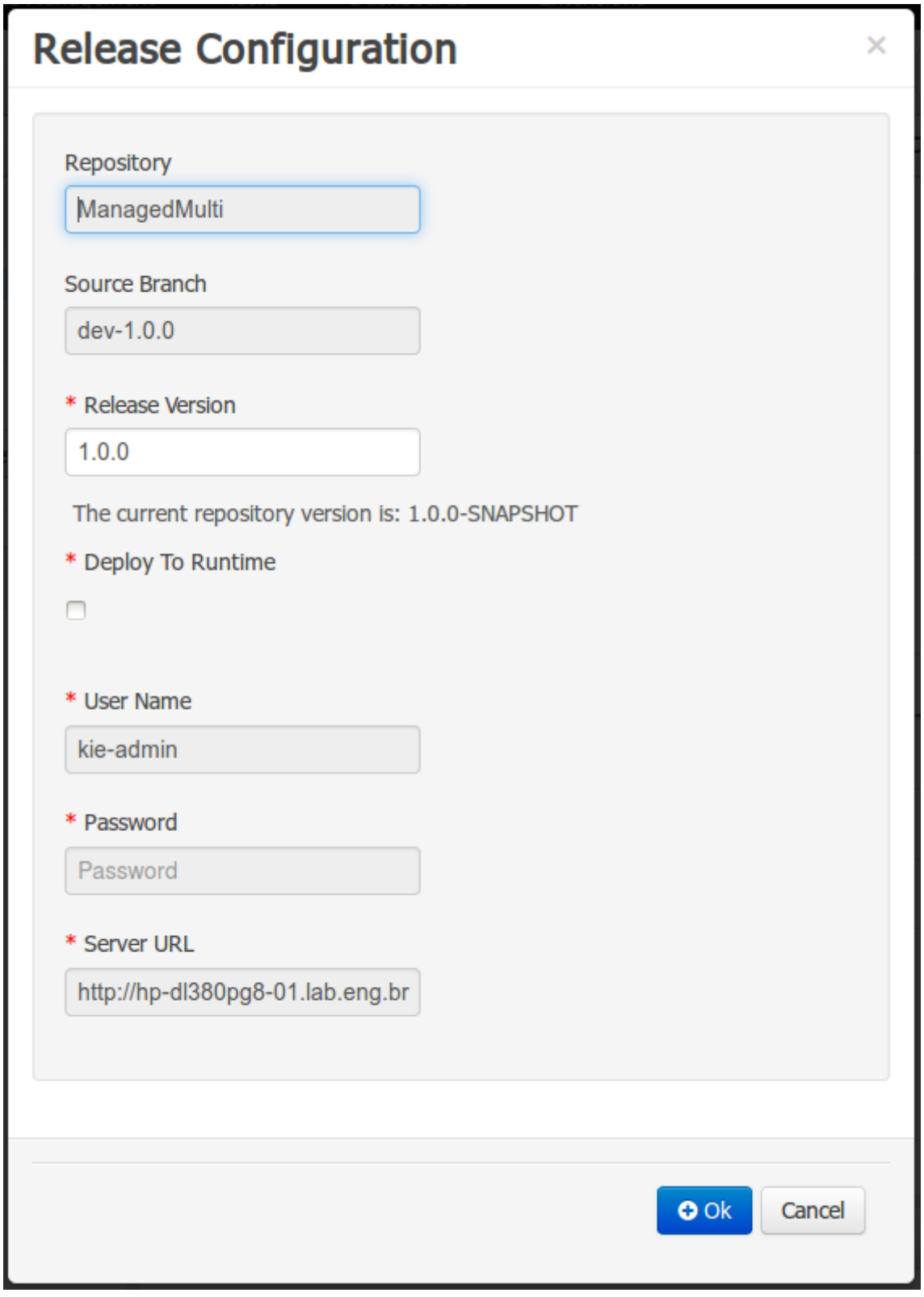


Figure 18.121. Release Process Parameters

18.9.7. Remote APIs

TBD

Chapter 19. Authoring Assets

19.1. Creating a package

Configuring packages is generally something that is done once, and by someone with some experience with rules/models. Generally speaking, very few people will need to configure packages, and once they are setup, they can be copied over and over if needed. Package configuration is most definitely a technical task that requires the appropriate expertise.

All assets live in "packages" in Drools Workbench - a package is like a folder (it also serves as a "namespace"). A home folder for rule assets to live in. Rules in particular need to know what the fact model is, what the namespace is etc.

So while rules (and assets in general) can appear in any number of categories, they only live in one package. If you think of Drools Workbench as a file system, then each package is a folder, and the assets live in that folder - as one big happy list of files.

To create an empty package select "Package" from the "New item" menu.

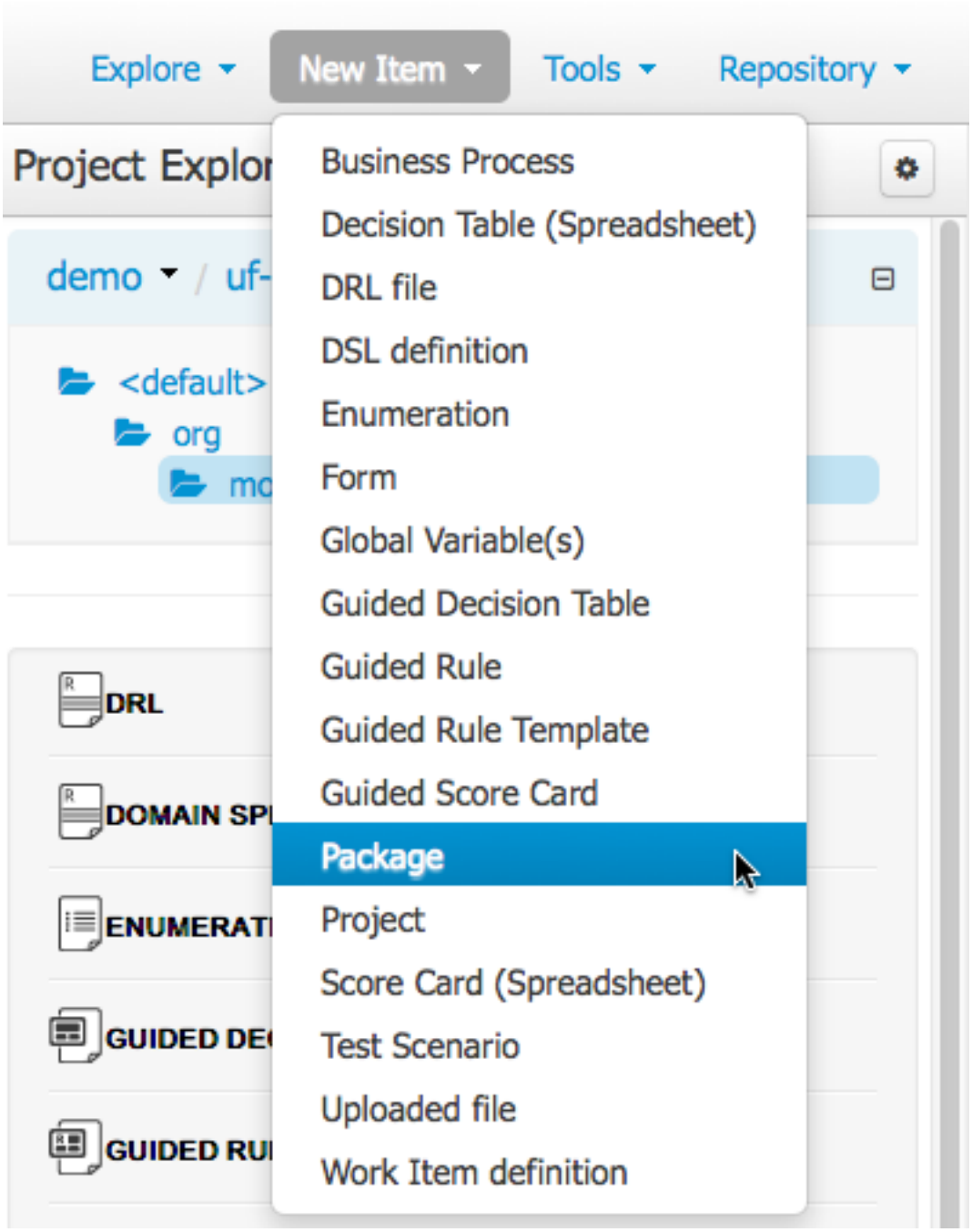


Figure 19.1. New Package

19.1.1. Empty package

An empty package can be created by simply specifying a name.

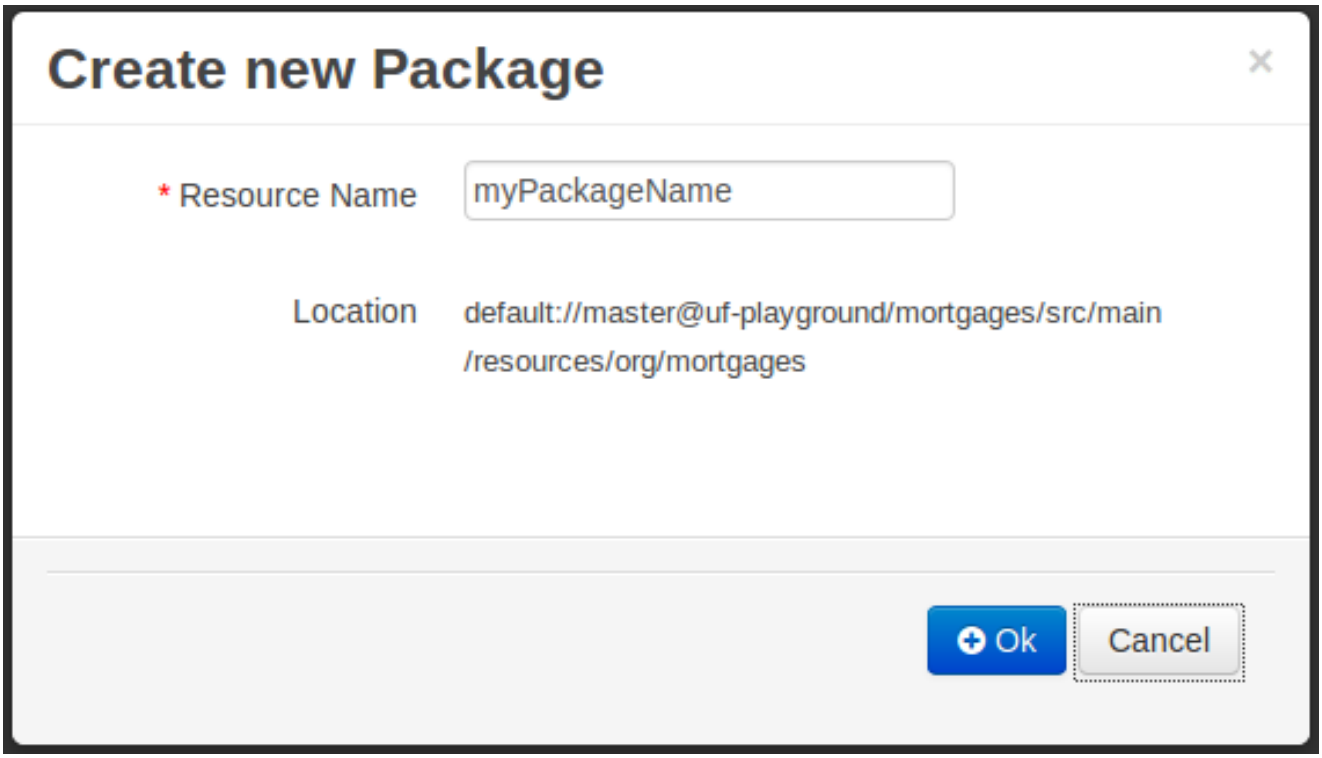


Figure 19.2. New empty Package

Once the Package has been created it will appear in the Project Explorer.

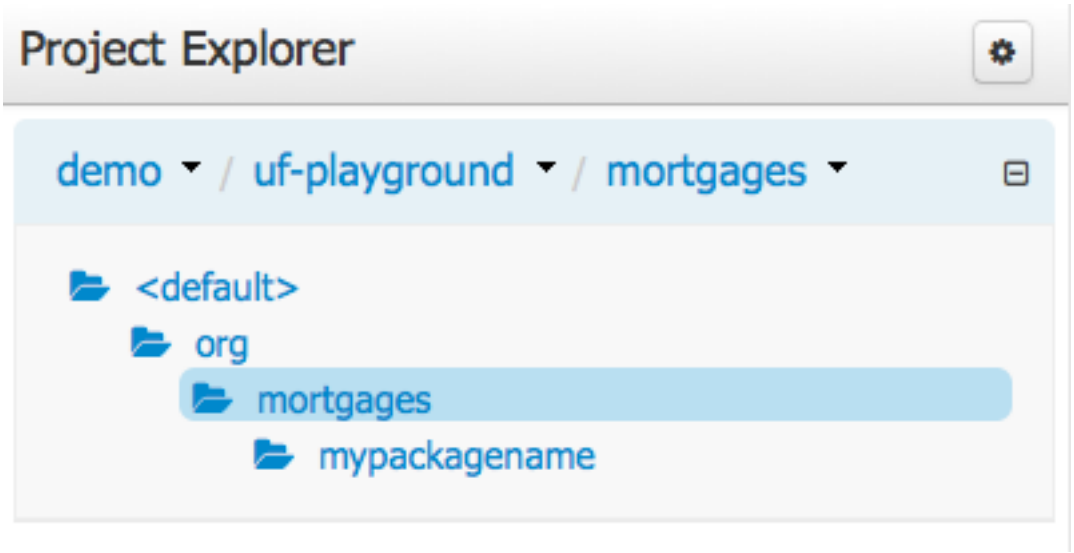


Figure 19.3. Project Explorer showing new Package

19.1.2. Copy, Rename and Delete Packages

As already mentioned on Project Explorer section, users can copy, rename or delete a package directly from Project Explorer.

As you can see in the following screenshots, those operations behaves very similar to counter part actions in most workbench editors.

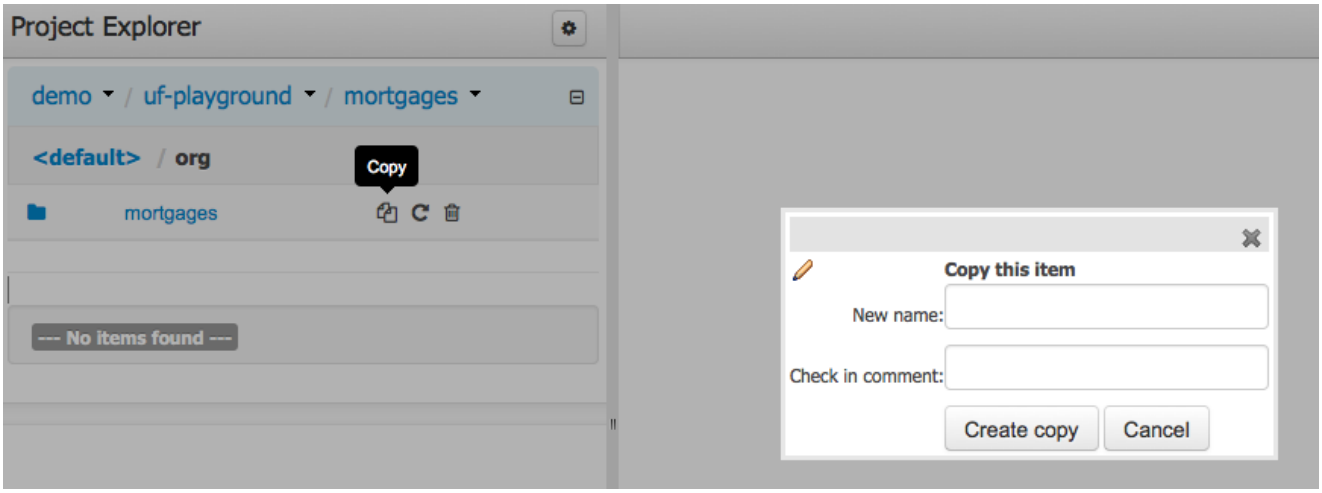


Figure 19.4. Copying a Package

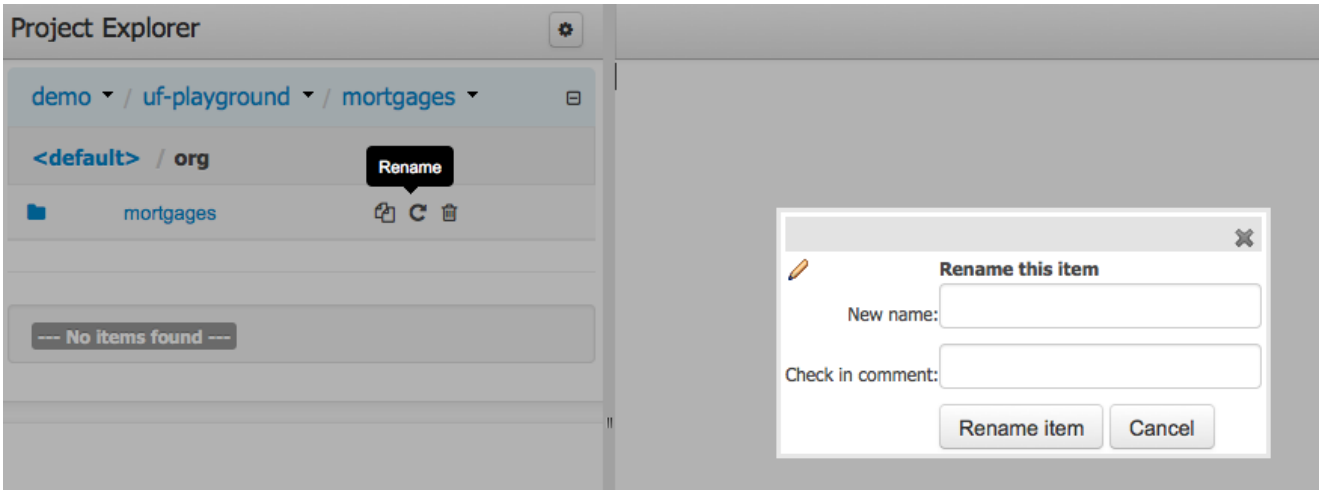


Figure 19.5. Renaming a Package

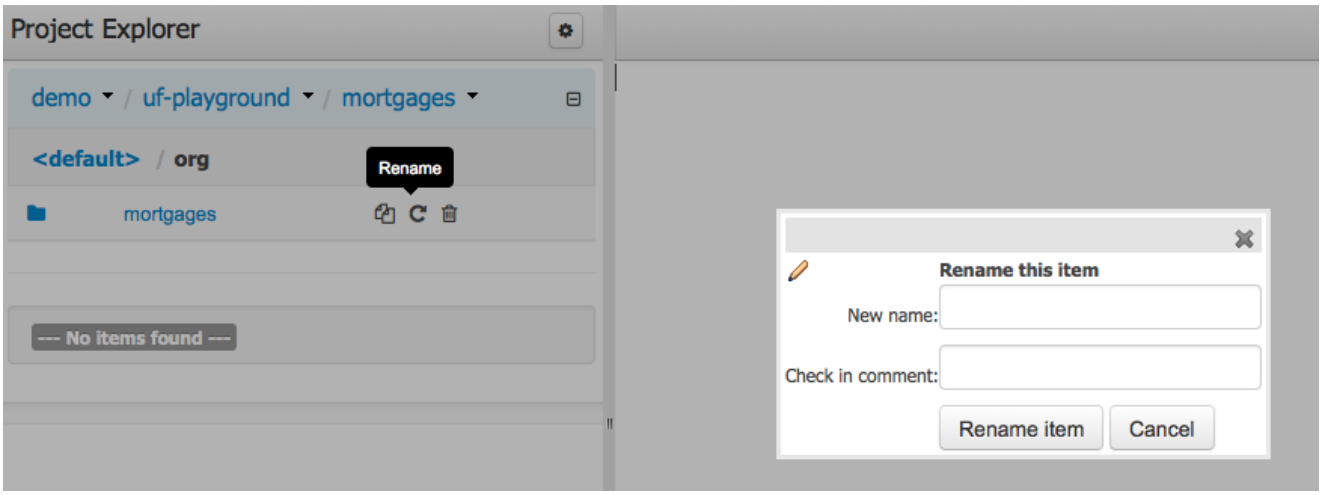


Figure 19.6. Excluding a Package

19.2. Business rules with the guided editor

Guided Rules are authored with a UI to control and prompt user input based on knowledge of the object model.

This can also be augmented with DSL sentences.

19.2.1. Parts of the Guided Rule Editor

The Guided Rule Editor is composed of three main sections.

The following diagram shows the editor in action. The following descriptions apply to the lettered boxes in the diagram:-

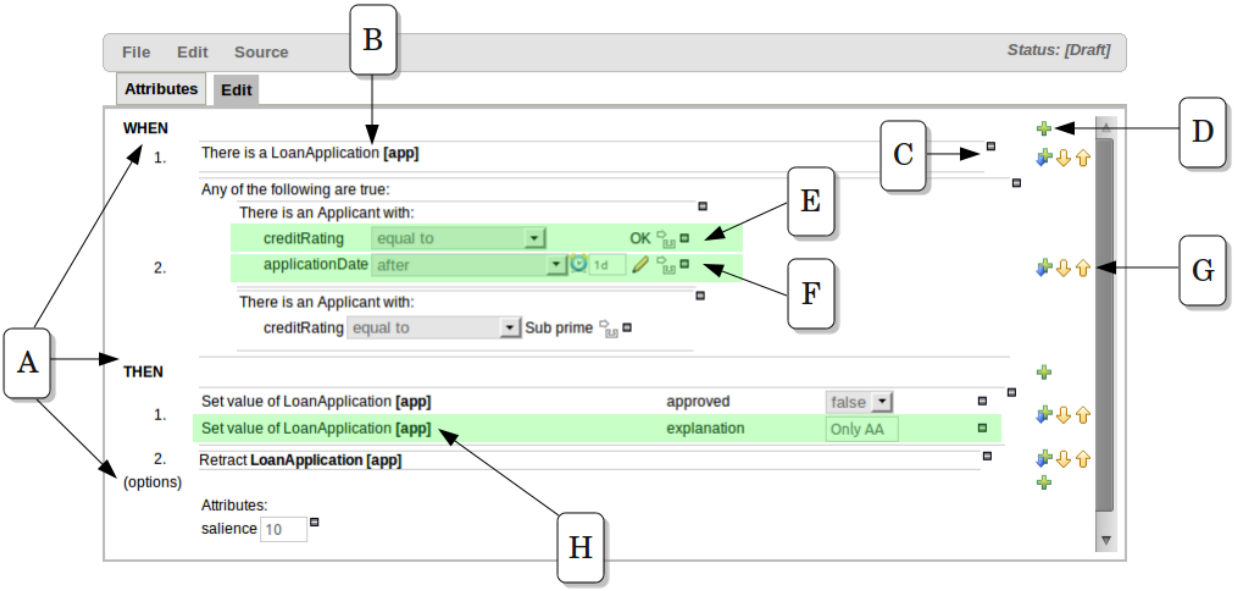


Figure 19.7. The guided BRL editor

A : The different parts of a rule:-

- The "WHEN" part, or conditions, of the rule.
- The "THEN" action part of the rule.
- Optional attributes that may effect the operation of the rule.

19.2.2. The "WHEN" (left-hand side) of a Rule

B : This shows a pattern which is declaring that the rule is looking for a "LoanApplication" fact (the fields are listed below, in this case none). Another pattern, "Applicant", is listed below "LoanApplication". Fields "creditRating" and "applicationDate" are listed. Clicking on the fact name ("LoanApplication") will pop-up a list of options to add to the fact declaration:-

- Add more fields (e.g. their "location").
- Assign a variable name to the fact (which you can use later on if needs be)
- Add "multiple field" constraints - i.e. constraints that span across fields (e.g. age > 42 or risk > 2).

C : The "minus" icon ("[-]") indicates you can remove something. In this case it would remove the whole "LoanApplication" fact declaration. Depending upon the placement of the icon different components of the rule declaration can be removed, for example a Fact Pattern, Field Constraint, other Conditional Element ("exists", "not exists", "from" etc) or an Action.

D : The "plus" icon ("+") allows you to add more patterns to the condition or the action part of the rule, or more attributes. In all cases, a popup option box is provided. For the "WHEN" part of the rule, you can choose from a list of Conditional Elements to add:

- A Constraint on a Fact: it will give you a list of facts.
- "The following does not exist": the fact plus constraints must not exist.
- "The following exists": at least one match should exist (but there only needs to be one - it will not trigger for each match).
- "Any of the following are true": any of the patterns can match (you then add patterns to these higher level patterns).
- "From": this will insert a new From Conditional Element to the rule.
- "From Accumulate": this will insert a new Accumulate Conditional Element to the rule.
- "From Collect": this will insert a new Collect Conditional Element to the rule.
- "From Entry-point": this allows you to define an Entry Point for the pattern.
- "Free Form DRL": this will let you insert a free chunk of DRL.

If you just put a fact (like is shown above) then all the patterns are combined together so they are all true ("and").

E : This shows the constraint for the "creditRating" field. Looking from left to right you find:-

- The field name: "creditRating". Clicking on it you can assign a variable name to it, or access nested properties of it.
- A list of constraint operations ("equal to" being selected): The content of this list changes depending on the field's data type.
- The value field: It could be one of the following:-

  1. A literal value: depending on the field's data type different components will be displayed:

     - String -> Textbox
     - Any numerical value -> Textbox restricting entry to values valid for the numerical sub-type (e.g. a byte can hold values from -128 to 127). BigDecimal and BigInteger data-types are also supported. Please ensure the appropriate Class has been imported in the Package configuration. The import will be added automatically if a POJO model has been uploaded that exposes an accessor or mutator for a BigDecimal or BigInteger field. BigDecimal values are automatically suffixed with "B" indicating to the underlying Engine that the literal value should be interpreted as a BigDecimal. BigIntegers are suffixed with "I". The user does not need to enter the suffix.
     - Date -> Calendar
     - Enumeration -> Listbox
     - Boolean -> Checkbox

2. A "formula": this is an expression which is calculated (this is for advanced users only)

3. An Expression - this will let you use an Expression Builder to build up a full mvel expression. (At the moment only basic expressions are supported)

F : This shows the constraint for the "applicationDate" field. Looking from left to right you find:

- The field name: "applicationDate".
- A list of constraint operations: "after" being selected.
- A "clock" icon. Since the "applicationDate" is a Date data-type the list of available operators includes those relating to Complex Event Processing (CEP). When a CEP operator is used this additional icon is displayed to allow you to enter additional CEP operator parameters. Clicking the "clock" will cycle the available combinations of CEP operator parameters.

Note

Complex Event Processing operators are also available when the Fact has been declared as an event. Refer to the "Fact Model" chapter of this user-guide for details on how to add annotations to your Fact model. Events have access to the full range of CEP operators; Date field-types are restricted to "after", "before" and "coincides".

Note

Facts annotated as Events can also have CEP sliding windows defined.

19.2.2.1. Adding Patterns

When clicking on the + button of the WHEN section, a new popup will appear letting you to add a new Pattern to the Rule. The popup will looks similar to the image below. In this popup you could select the type of Pattern to add by selecting one of the list items. In the list you will have an entry for each defined Fact Type, in addition to the already mentioned Conditional Elements like "exists", "doesn't exist", "from", "collect", "accumulate", "from entry-point" and "free form DRL". Once you have selected one of this elements, you can add a new Pattern by clicking on the "Ok" button. The new pattern will be added at the bottom of the rule's left hand side. If you want to choose a different position, you can use the combobox placed at the top of the popup.

You can also open this popup by clicking in the [+] button from a Pattern's action toolbar. If that is the case, the pop-up that appears wouldn't constraint the position combobox, because the new Pattern will be added just after the Pattern where you clicked.

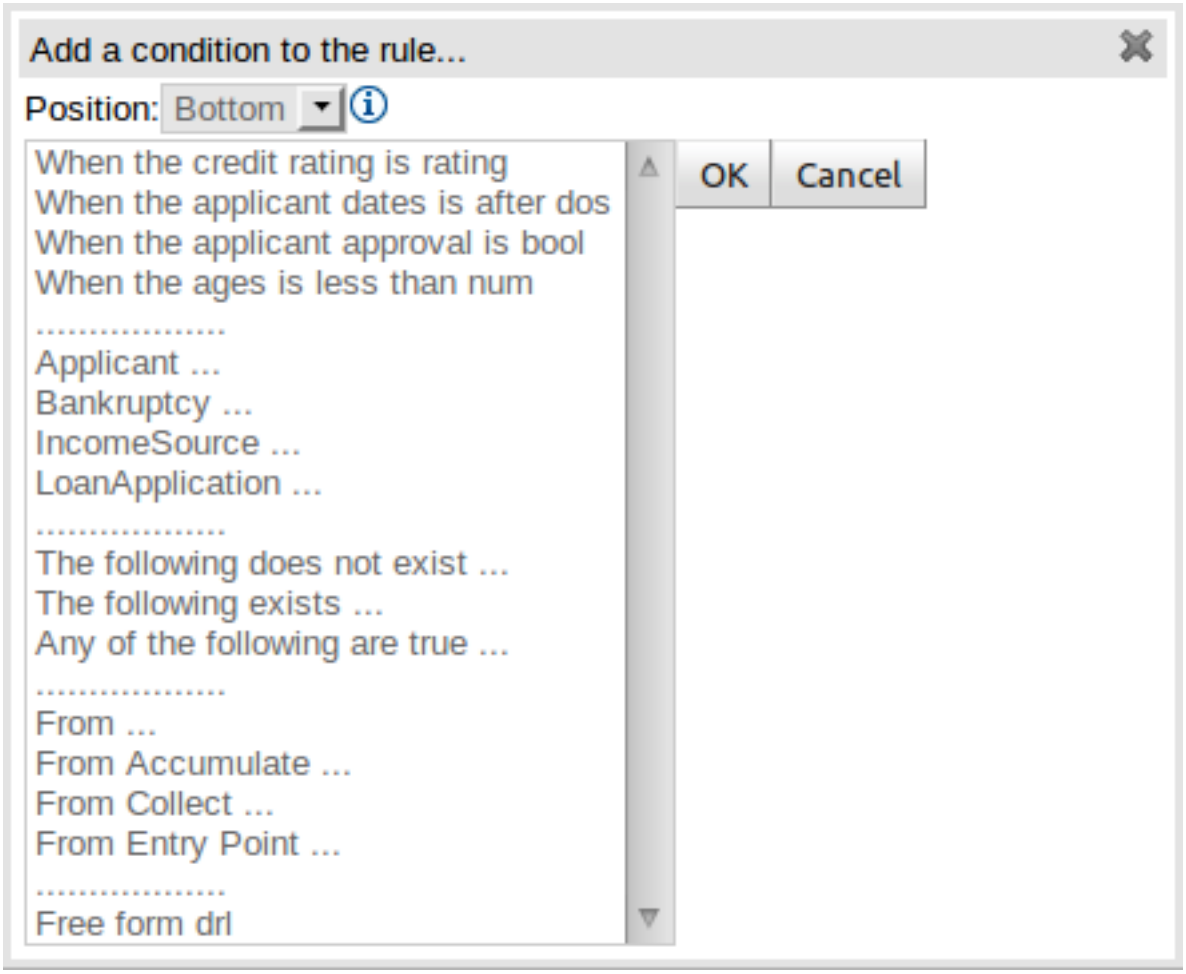


Figure 19.8. Adding Patterns

19.2.2.2. Adding constraints

The below dialog is what you will get when you want to add constraints to a fact. In the top half are the simple options: you can either add a field constraint straight away (a list of fields of the applicable fact will be shown), or you can add a "Multiple field constraint" using AND or OR operands. In the bottom half of the window you have the Advanced options: you can add a formula (which resolves to True or False - this is like in the example above: "... salary > (2500 * 4.1)". You can also assign a Variable name to the fact (which means you can then access that variable on the action part of the rule, to set a value etc).

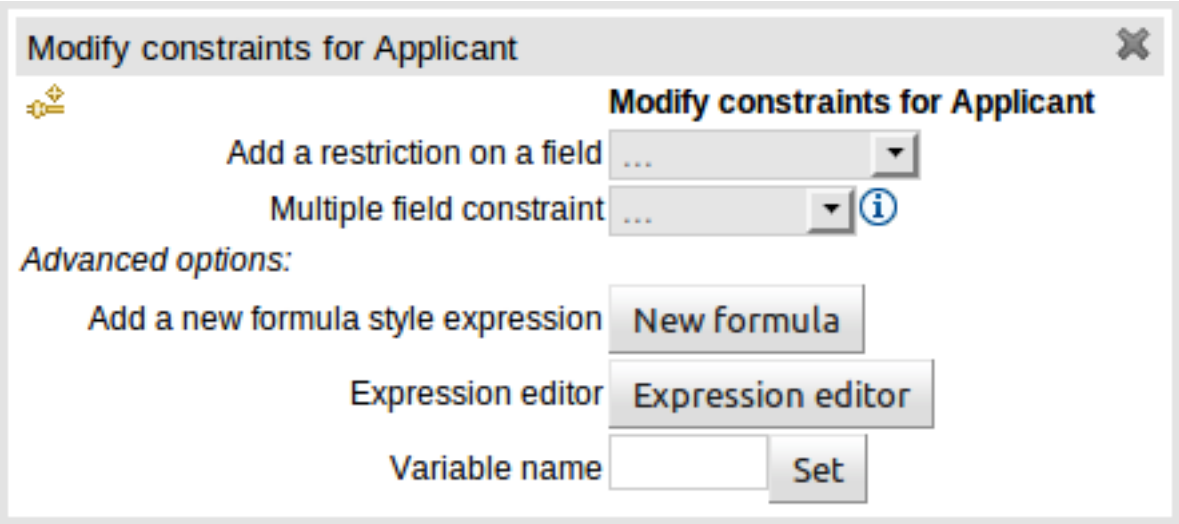


Figure 19.9. Adding constraints

19.2.3. The "THEN" (right-hand side) of a Rule

H : This shows an "action" of the rule, the Right Hand Side of a rule consists in a list of actions. In this case, we are updating the "explanation" field of the "LoanApplication" fact. There are quite a few other types of actions you can use:-

- Insert a completely new Fact and optionally set a field on the Fact.

  The value field can be one of the following:-

  1. A literal value: depending on the field's data type different components will be displayed:
     - String -> Textbox
     - Any numerical value -> Textbox restricting entry to values valid for the numerical sub-type (e.g. a byte can hold values from -128 to 127). BigDecimal and BigInteger data-types are also supported. Please ensure the appropriate Class has been imported in the Package configuration. The import will be added automatically if a POJO model has been uploaded that exposes an accessor or mutator for a BigDecimal or BigInteger field. BigDecimal values are automatically suffixed with "B" indicating to the underlying Engine that the literal value should be interpreted as a BigDecimal. BigIntegers are suffixed with "I". The user does not need to enter the suffix.
     - Date -> Calendar
     - Enumeration -> Listbox
     - Boolean -> Checkbox
  2. A variable bound to a Fact or Field in the left-hand side. The data-type of the field being set must match the data-type of the variable.

3. A "formula": this is an expression which is calculated (this is for advanced users only)

- Logically insert a completely new Fact (see "Truth Maintenance" in the Expert documentation) and optionally set a field on the Fact.

  1. A literal value: depending on the field's data type different components will be displayed:

     The value field can be one of the following:-

     a. A literal value: depending on the field's data type different components will be displayed:

        - String -> Textbox
        - Any numerical value -> Textbox restricting entry to values valid for the numerical sub-type (e.g. a byte can hold values from -128 to 127). BigDecimal and BigInteger data-types are also supported. Please ensure the appropriate Class has been imported in the Package configuration. The import will be added automatically if a POJO model has been uploaded that exposes an accessor or mutator for a BigDecimal or BigInteger field. BigDecimal values are automatically suffixed with "B" indicating to the underlying Engine that the literal value should be interpreted as a BigDecimal. BigIntegers are suffixed with "I". The user does not need to enter the suffix.
        - Date -> Calendar
        - Enumeration -> Listbox
        - Boolean -> Checkbox

     b. A variable bound to a Fact or Field in the left-hand side. The data-type of the field being set must match the data-type of the variable.

     c. A "formula": this is an expression which is calculated (this is for advanced users only)

  2. A variable bound to a Fact or Field in the left-hand side. The data-type of the field being set must match the data-type of the variable.

  3. A "formula": this is an expression which is calculated (this is for advanced users only)

- Modify a field of an existing fact (which tells the engine the fact has changed).

  The value field can be one of the following:-

  1. A literal value: depending on the field's data type different components will be displayed:

     - String -> Textbox
     - Any numerical value -> Textbox restricting entry to values valid for the numerical sub-type (e.g. a byte can hold values from -128 to 127). BigDecimal and BigInteger data-types are also supported. Please ensure the appropriate Class has been imported in the Package configuration. The import will be added automatically if a POJO model has been uploaded

that exposes an accessor or mutator for a BigDecimal or BigInteger field. BigDecimal values are automatically suffixed with "B" indicating to the underlying Engine that the literal value should be interpreted as a BigDecimal. BigIntegers are suffixed with "I". The user does not need to enter the suffix.

- Date -> Calendar
- Enumeration -> Listbox
- Boolean -> Checkbox

2. A variable bound to a Fact or Field in the left-hand side. The data-type of the field being set must match the data-type of the variable.
3. A "formula": this is an expression which is calculated (this is for advanced users only)

- Set a field on a fact (in which case the engine doesn't know about the change - normally because you are setting a result).

  The value field can be one of the following:-

  1. A literal value: depending on the field's data type different components will be displayed:
     - String -> Textbox
     - Any numerical value -> Textbox restricting entry to values valid for the numerical sub-type (e.g. a byte can hold values from -128 to 127). BigDecimal and BigInteger data-types are also supported. Please ensure the appropriate Class has been imported in the Package configuration. The import will be added automatically if a POJO model has been uploaded that exposes an accessor or mutator for a BigDecimal or BigInteger field. BigDecimal values are automatically suffixed with "B" indicating to the underlying Engine that the literal value should be interpreted as a BigDecimal. BigIntegers are suffixed with "I". The user does not need to enter the suffix.
     - Date -> Calendar
     - Enumeration -> Listbox
     - Boolean -> Checkbox
  2. A variable bound to a Fact or Field in the left-hand side. The data-type of the field being set must match the data-type of the variable.
  3. A "formula": this is an expression which is calculated (this is for advanced users only)

- Delete a fact from the Engine's Working Memory.
- Add Facts to existing global lists.
- Call a method on a variable.

- Write a chunk of free form code.

19.2.4. Optional attributes

The attributes section of a rule provides the means to define metadata and attributes (such as "salience", "no-loop" etc).

Click on the "+" icon to add a new metadata or attribute definition. Each defined will appear listed in this section.

Click on the "-" icon beside each metadata or attribute to remove it.

19.2.4.1. Salience

Each rule has a salience value which is an integer value that defaults to zero. The salience value represents the priority of the rule with higher salience values representing higher priority. Salience values can be positive or negative.

19.2.5. Pattern/Action toolbar

G : Next to each Pattern or Action you will find a toolbar containing 3 buttons.

The first "+" icon lets you insert a new Pattern/Action at an arbitrary location. The other "+" icons allow you to insert a new Pattern/Action below that you have selected.

The remaining arrow icons allow you to move the current Pattern/Action up or down.

19.2.6. User driven drop down lists

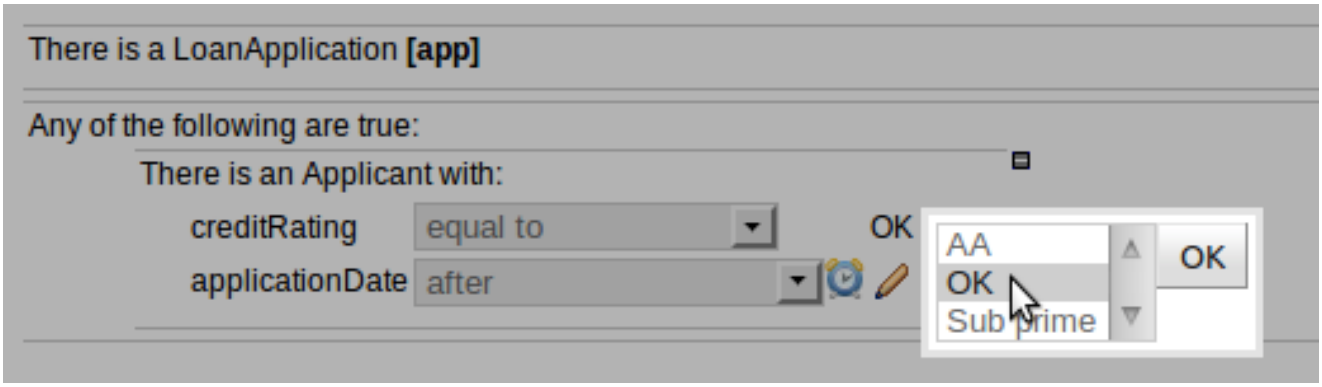


Figure 19.10. Data enumeration showing as a drop down list

Note that is it possible to limit field values to items in a pre-configured list. This list is either defined by a Java enumeration or configured as part of the package (using a data enumeration to provide values for the drop down list). These values can be a fixed list, or (for example) loaded from a database. This is useful for codes, and other fields where there are set values. It is also possible

to have what is displayed on screen, in a drop down, be different to the value (or code) used in a rule. See the section on data enumerations for how these are configured.

It is possible to define a list of values for one field that are dependent upon the value of one or more other fields, on the same Fact (e.g. a list of "Cities" depending on the selected "Country region"). Please refer to the section on "Enumerations" for more information.

19.2.7. Augmenting with DSL sentences

If the package the rule is part of has a DSL configuration, when when you add conditions or actions, then it will provide a list of "DSL Sentences" which you can choose from - when you choose one, it will add a row to the rule - where the DSL specifies values come from a user, then a edit box (text) will be shown (so it ends up looking a bit like a form). This is optional, and there is another DSL editor. Please note that the DSL capabilities in this editor are slightly less then the full set of DSL features (basically you can do [when] and [then] sections of the DSL only - which is no different to drools 3 in effect).

The following diagram shows the DSL sentences in action in the guided editor:

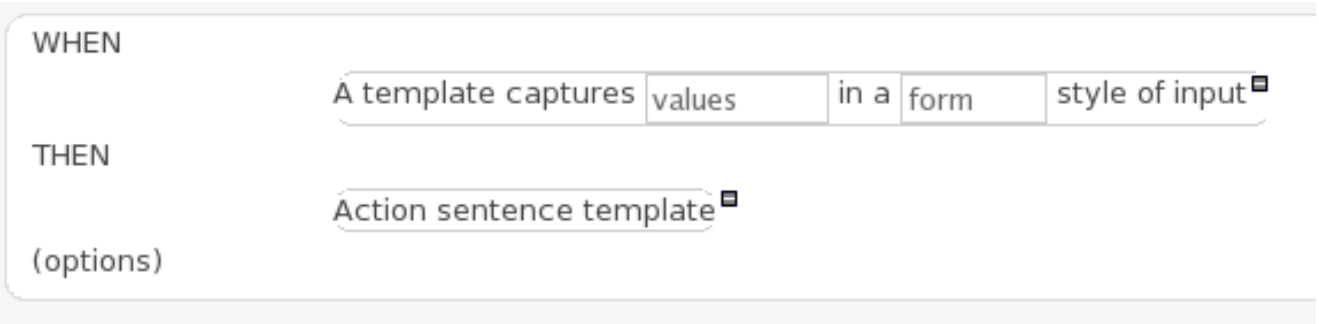


Figure 19.11. DSL in guided editor

19.2.8. A more complex example:

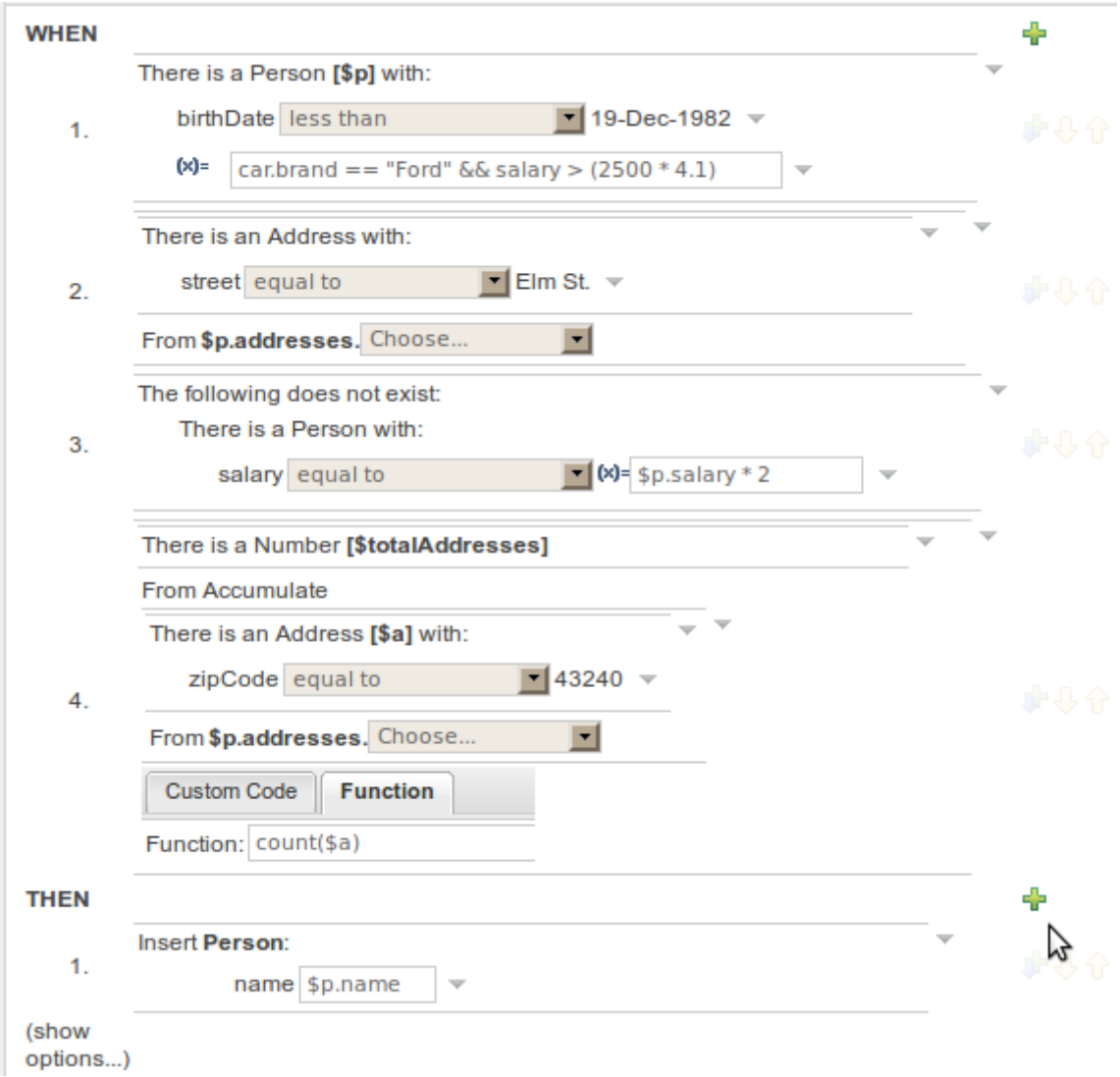


Figure 19.12. A more complex BRL example

In the above example, you can see how to use a mixture of Conditional Elements, literal values, and formulas. The rule has 4 "top level" Patterns and 1 Action. The "top level" Patterns are:

1. A Fact Pattern on Person. This Pattern contains two field constraints: one for birthDate field and the other one is a formula. Note that the value of the birthDate restriction is selected from a calendar. Another thing to note is that you can make calculations and use nested fields in the formula restriction (i.e. car.brand). Finally, we are setting a variable name ($p) to the Person Fact Type. You can then use this variable in other Patterns.

Note

The generated DRL from this Pattern will be:

```
$p : Person( birthDate < "19-Dec-1982" , eval( car.brand == "Ford" && salary
 > (2500 * 4.1) ))
```

2. A From Pattern. This condition will create a match for every Address whose street name is "Elm St." from the Person's list of addresses. The left side of the from is a regular Fact Pattern and the right side is an Expression Builder that let us inspect variable's fields.

Note

The generated DRL from this Pattern will be: `Address( street == "Elm St." ) from $p.addresses`

3. A "Not Exist" Conditional Element. This condition will match when its content doesn't create a match. In this case, its content is a regular Fact Pattern (on Person). In this Fact Pattern you can see how variables ($p) could be used inside a formula value.

Note

The generated DRL from this Pattern will be: `not Person( salary == ( $p.salary * 2 ) )`

4. A "From Accumulate" Conditional Element. This is maybe one of the most complex Patterns you can use. It consist in a Left Pattern (It must be a Fact Pattern. In this case is a Number Pattern. The Number is named $totalAddresses), a Source Pattern (Which could be a Fact Pattern, From, Collect or Accumulate conditional elements. In this case is an Address Pattern Restriction with a field restriction in its zip field) and a Formula Section where you can use any built-in or custom Accumulate Function (in this example a count() function is used). Basically, this Conditional Element will count the addresses having a zip code of 43240 from the Person's list of addresses.

Note

The generated DRL from this Pattern will be: `$totalAddresses : Number() from accumulate ($a : Address( zipCode == " 43240") from $p.addresses, count($a))`

19.3. Templates of assets/rules

The guided rule editor is great when you need to define a single rule, however if you need to define multiple rules following the same structure but with different values in field constraints or action sections a "Rule Template" is a valuable asset. Rule templates allow the user to define a

rule structure with place-holders for values that are to be interpolated from a table of data. Literal values, formulae and expressions can also continue to be used.

Rule Templates can often be used as an alternative for Decision Tables in Drools Workbench.

19.3.1. Creating a rule template

To create a template for a rule simply select the "Guided Rule Template" from the "New Item" menu.

19.3.2. Define the template

Once a rule template has been created the editor is displayed. The editor takes the form of the standard guided editor explained in more detail under the "Rule Authoring" section. As the rule is constructed you are given the ability to insert "Template Keys" as place-holders within your field constraints and action sections. Literal values, formulae and expressions can continue to be used as in the standard guided editor.

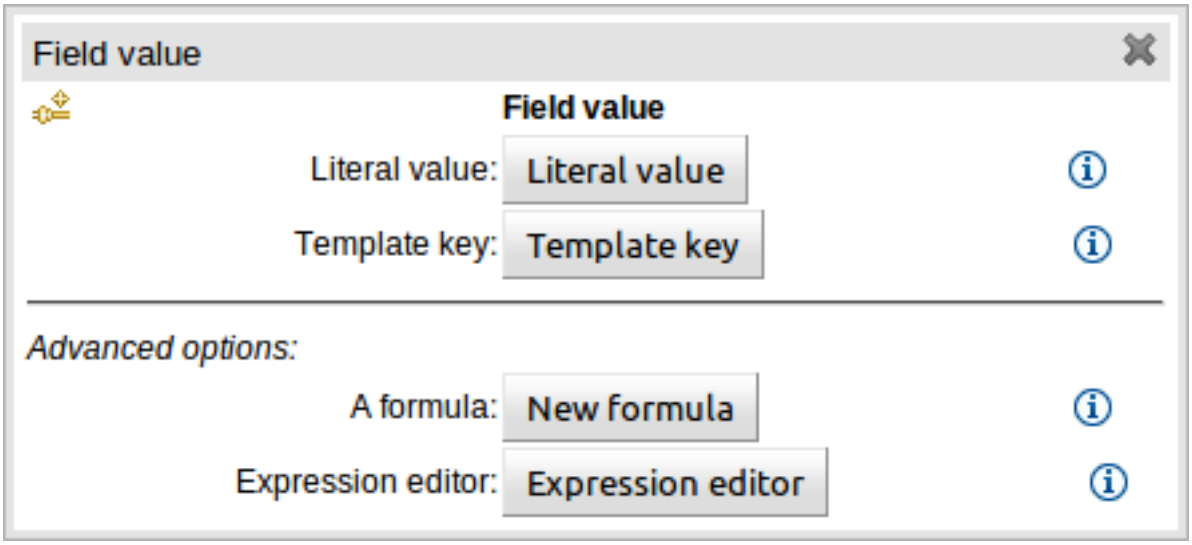


Figure 19.13. Template Key popup

The following screenshot illustrates a simple rule that has been defined with a "Template Key" for the applicants' maximum age, minimum age and credit rating. The template keys have been defined as "$max_age", "$min_age" and "$cr" respectively.

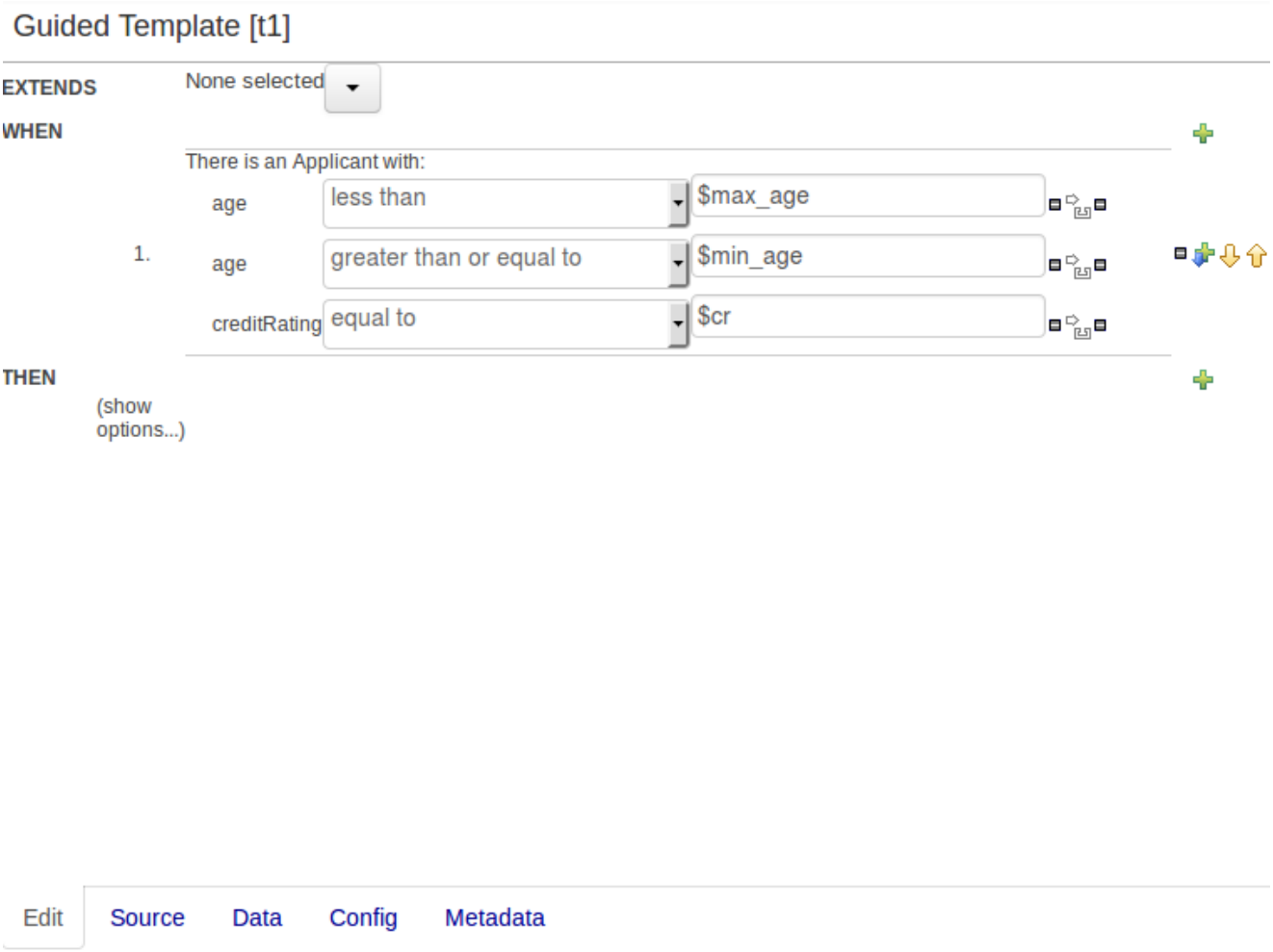


Figure 19.14. Rule template in the guided editor

19.3.3. Defining the template data

When you have completed the definition of your rule template you need to enter the data that will be used to interpolate the "Template Key" place-holders. Drools Workbench provides the facility to enter data in a flexible grid within the guided editor screen. The data entry section is located on the Data tab within the editor.

The rule template data grid is very flexible; with different pop-up editors for the underlying fields' data-types. Columns can be resized and sorted; and cells can be merged and grouped to facilitate rapid data entry.

One row of data interpolates the "Template Key" place-holders for a single rule; thus one row becomes one rule.

Note

If any cells for a row are left blank a rule for the applicable row is not generated.

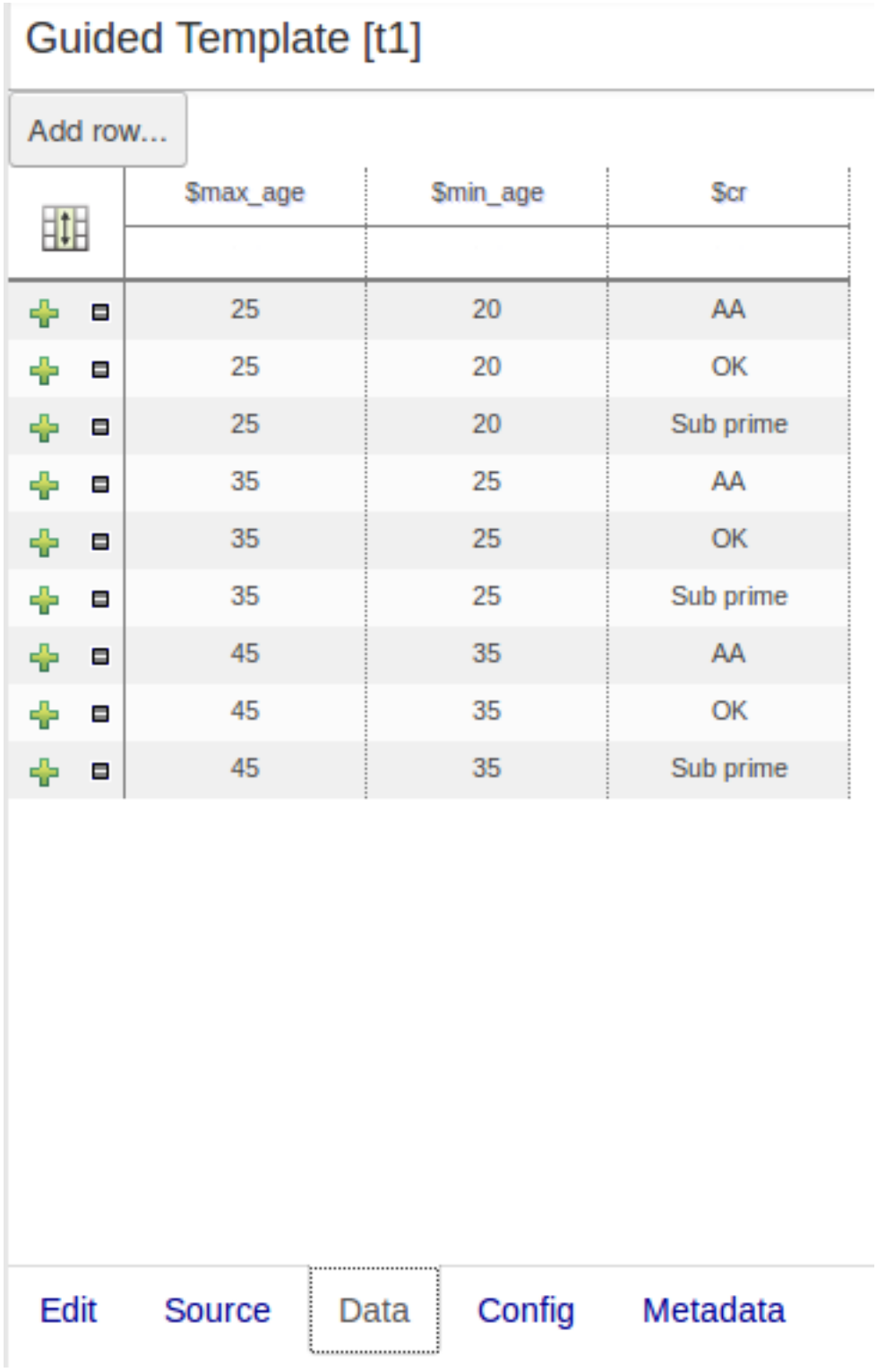


Figure 19.15. Template data grid

19.3.3.1. Cell merging

The icon in the top left of the grid toggles cell merging on and off. When cells are merged those in the same column with identical values are merged into a single cell. This simplifies changing the value of multiple cells that shared the same original value. When cells are merged they also gain an icon in the top-left of the cell that allows rows spanning the merged cell to be grouped.

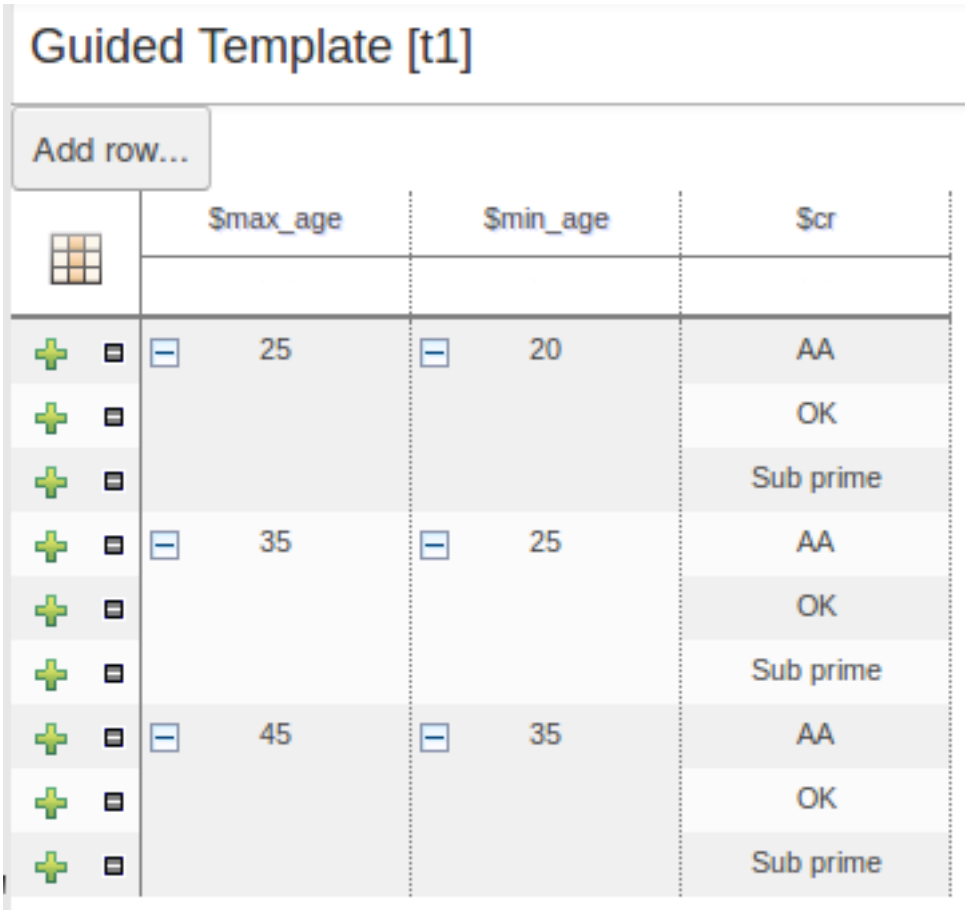


Figure 19.16. Cell merging

19.3.3.2. Cell grouping

Cells that have been merged can be further collapsed into a single row. Clicking the [+\-] icon in the top left of a merged cell collapses the corresponding rows into a single entry. Cells in other columns spanning the collapsed rows that have identical values are shown unchanged. Cells in other columns spanning the collapsed rows that have different values are highlighted and the first value displayed.

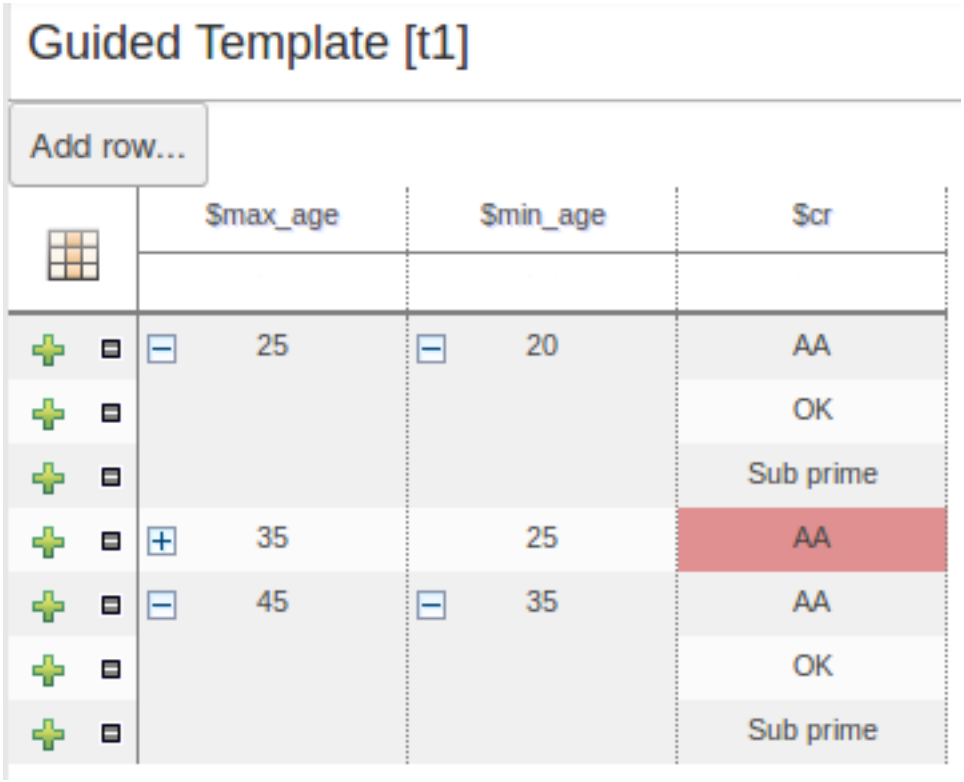


Figure 19.17. Cell grouping

When the value of a grouped cell is altered all cells that have been collapsed also have their values updated.

19.3.4. Generated DRL

Whilst not necessary, rule authors can view the DRL that will be generated for a "Rule Template" and associated data. This feature and its operation is no different to that for other assets. Select the "Source" tab from the bottom of the editor screen. The DRL for all rules will be displayed.

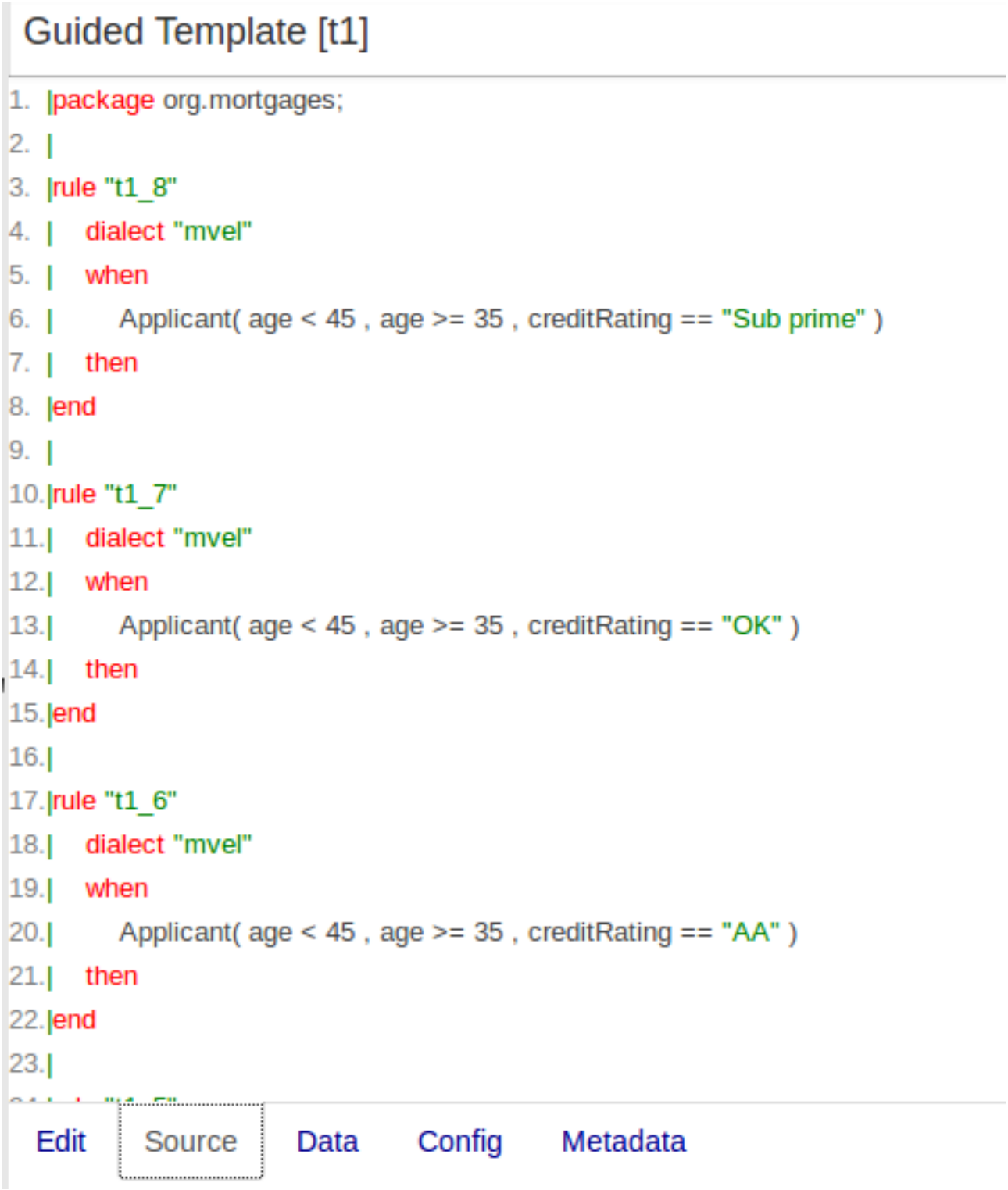


Figure 19.18. Generated DRL

19.4. Guided decision tables (web based)

The guided decision table feature allows decision tables to be edited in place on the web. This works similar to the guided editor by introspecting what facts and fields are available to guide the creation of a decision table. Rule attributes, meta-data, conditions and actions can be defined in a tabular format thus facilitating rapid entry of large sets of related rules. Web-based decision table rules are compiled into DRL like all other rule assets.

19.4.1. Types of decision table

There are broadly two different types of decision table, both of which are supported in Drools Workbench:-

- Extended Entry
- Limited Entry

19.4.1.1. Extended Entry

An Extended Entry decision table is one for which the column definitions, or stubs, specify Pattern, Field and operator but not value. The values, or states, are themselves held in the body of the decision table. It is normal, but not essential, for the range of possible values to be restricted by limiting entry to values from a list. Drools Workbench supports use of Java enumerations, Drools Workbench enumerations or decision table "optional value lists" to restrict value entry.

Decision table

#	Description	Age	Make	Premium
		Applicant [$a]	Vehicle [$v]	
		age [<]	make [==]	
1		35	BMW	1000
2		35	Audi	1000

Figure 19.19. Extended Entry Decision table

19.4.1.2. Limited Entry

A Limited Entry decision table is one for which the column definitions specify value in addition to Pattern, Field and operator. The decision table states, held in the body of the table, are boolean where a positive value (a checked tick-box) has the effect of meaning the column should apply, or be matched. A negative value (a cleared tick-box) means the column does not apply.

Decision table

#	Description	Age < 35	BMW	Audi	Premium 1000
		Applicant [$a]	Vehicle [$v]		
		age [<35]	make [==BMW]	make [==Audi]	
1		☑	☑	☑	☐
2		☐	☑	☑	☐
3		☑	☐	☑	☐
4		☐	☐	☑	☐
5		☑	☑	☐	☐
6		☐	☑	☐	☐
7		☑	☐	☐	☐
8		☐	☐	☐	☐

Figure 19.20. Limited Entry Decision table

19.4.2. Main components\concepts

The guided decision table is split into two main sections:-

- The upper section allows table columns to be defined representing rule attributes, meta-data, conditions and actions.
- The lower section contains the actual table itself; where individual rows define separate rules.

Decision table

Condition columns

Action columns

(options)

#	Description	salience	name	age	age
1		1	Bill	30	12345
2		2	Ben	<otherwise>	12345
3		3	Weed	40	12345
4		4	<otherwise>	50	12345

Figure 19.21. Main components

19.4.2.1. Navigation

Cells can be selected in a variety of ways:-

- Firstly individual cells can be double-clicked and a pop-up editor corresponding to the underlying data-type will appear. Groups of cells in the same column can be selected by either clicking in the first and dragging the mouse pointer or clicking in the first and clicking the extent of the required range with the shift key pressed.
- Secondly the keyboard cursor keys can be used to navigate around the table. Pressing the enter key will pop-up the corresponding editor. Ranges can be selected by pressing the shift key whilst extending the range with the cursor keys.

Columns can be resized by hovering over the corresponding divider in the table header. The mouse cursor will change and then the column width dragged either narrower or wider.

19.4.2.2. Cell merging

The icon in the top left of the decision table toggles cell merging on and off. When cells are merged those in the same column with identical values are merged into a single cell. This simplifies changing the value of multiple cells that shared the same original value. When cells are merged they also gain an icon in the top-left of the cell that allows rows spanning the merged cell to be grouped.

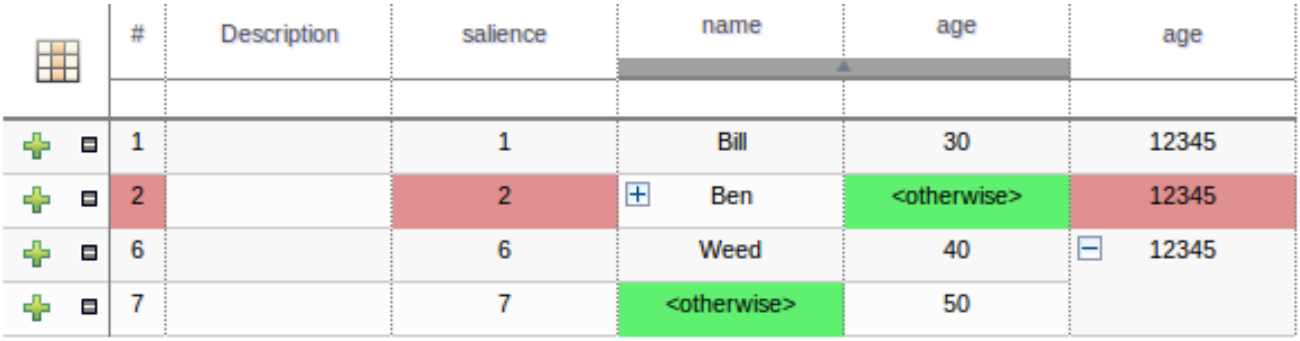
	#	Description	salience	name	age	age
	1		1	Bill	30	12345
	2		2	Ben	<otherwise>	
	3		3			
	4		4			
	5		5			
	6		6	Weed	40	12345
	7		7	<otherwise>	50	

Figure 19.22. Cell merging

19.4.2.3. Cell grouping

Cells that have been merged can be further collapsed into a single row. Clicking the [+\-] icon in the top left of a merged cell collapses the corresponding rows into a single entry. Cells in other columns spanning the collapsed rows that have identical values are shown unchanged. Cells in

other columns spanning the collapsed rows that have different values are highlighted and the first value displayed.



#	Description	salience	name	age	age
1		1	Bill	30	12345
2		2	Ben	<otherwise>	12345
6		6	Weed	40	12345
7		7	<otherwise>	50	

Figure 19.23. Cell grouping

When the value of a grouped cell is altered all cells that have been collapsed also have their values updated.

19.4.2.4. Operation of "otherwise"

Condition columns defined with literal values that use either the equality (==) or inequality (!=) operators can take advantage of a special decision table cell value of "otherwise". This special value allows a rule to be defined that matches on all values not explicitly defined in all other rules defined in the table. This is best illustrated with an example:-

```
when
  Cheese( name not in ("Cheddar", "Edam", "Brie") )
  ...
then
  ...
end
```

```
when
  Cheese( name in ("Cheddar", "Edam", "Brie") )
  ...
then
  ...
end
```

19.4.2.5. Re-arranging columns

Whole patterns and individual conditions can be re-arranged by dragging and dropping them in the configuration section of the screen. This allows constraints to be re-ordered to maximise performance of the resulting rules, by placing generalised constraints before more specific. Action columns can also be re-arranged by dragging and dropping them.

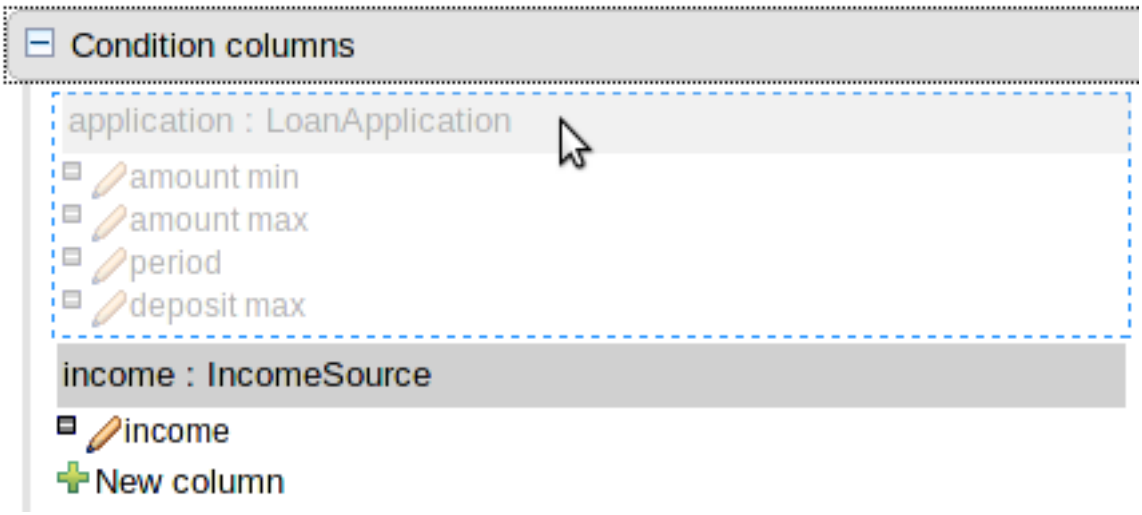


Figure 19.24. Re-arranging Condition patterns

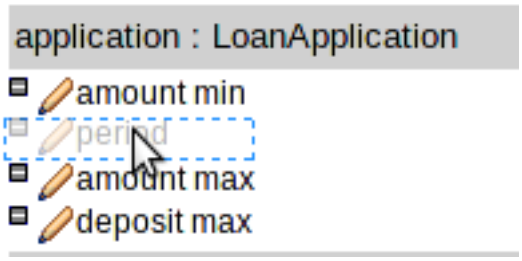


Figure 19.25. Re-arranging columns

19.4.3. Defining a web based decision table

19.4.3.1. Manual creation

When a new empty decision table has been created you need to define columns for Facts, their constraints and corresponding actions.

19.4.3.1.1. Column configuration

Expand the "Decision table" element and you will see three further sections for "Conditions", "Actions" and "Options". Expanding either the "Conditions" or "Actions" sections reveals the "New column" icon. This can be used to add new column definitions to the corresponding section. Existing columns can be removed by clicking the "-" icon beside each column name, or edited by clicking the "pencil" icon also beside each column name. The "Options" section functions slightly differently however the principle is the same: clicking the "Add Attribute/Metadata" icon allows columns for table attributes to be defined (such as "salience", "no-loop" etc) or metadata added.

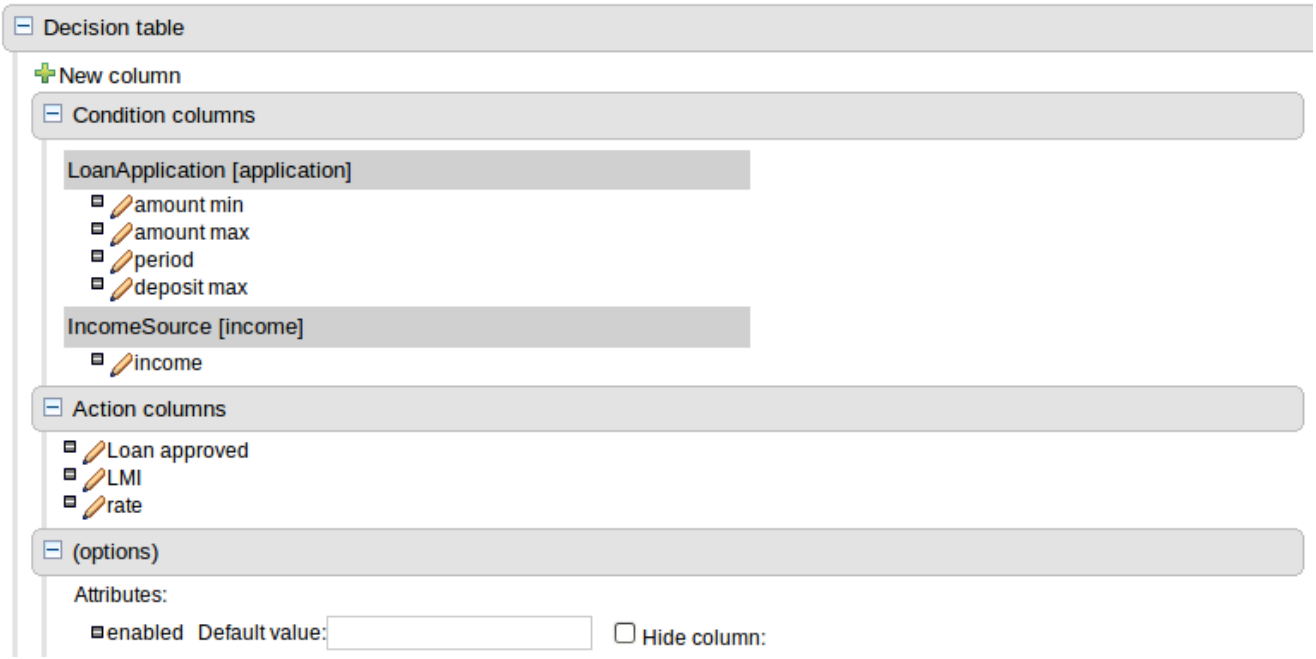


Figure 19.26. Column configuration

19.4.3.1.1.1. Utility columns

All decision table contain two utility columns containing rule number and rule description.

19.4.3.1.1.2. Adding columns

To add a column click on the "New column" icon.

You are presented with the following column type selection popup.

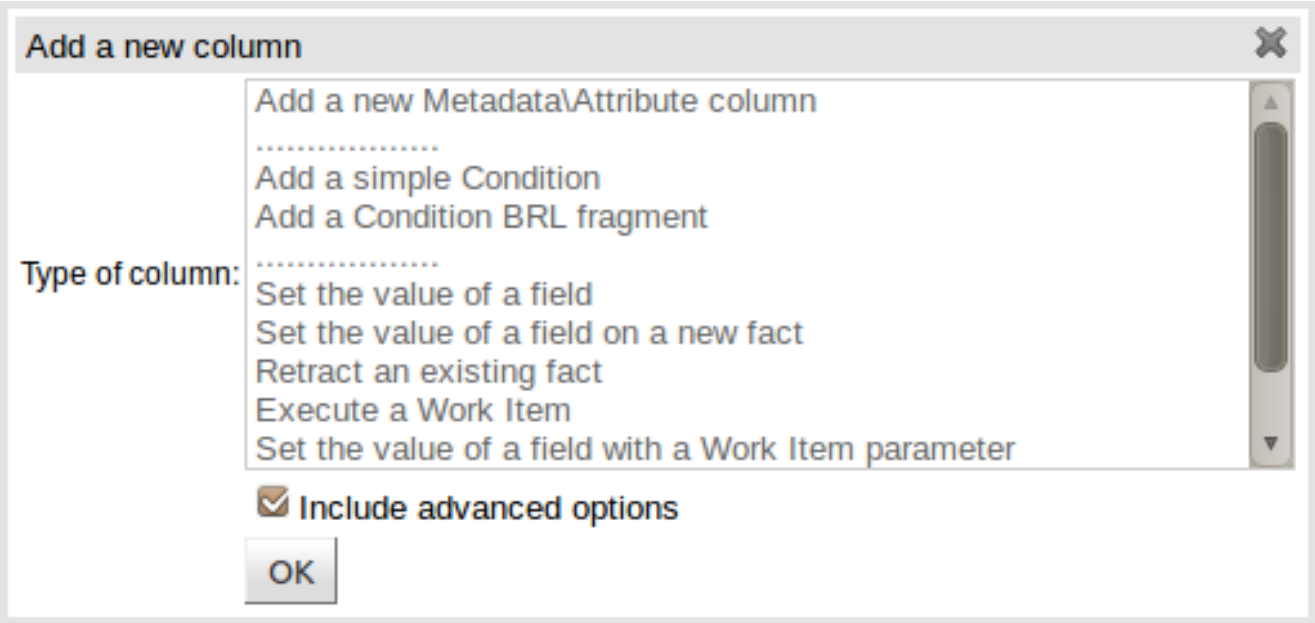


Figure 19.27. Column type popup

By default the column type popup only shows the following simple types:-

- Add a new Metadata\Attribute column
- Add a simple Condition
- Set the value of a field
- Set the value of a field on a new fact
- Delete an existing fact

Clicking on "Include advanced options" adds the following additional "advanced" column types for more advanced use cases:-

- Add a Condition BRL fragment
- Execute a Work Item
- Set the value of a field with a Work Item parameter
- Set the value of a field on a new Fact with a Work Item parameter
- Add an Action BRL fragment

19.4.3.1.1.3. Simple column types

19.4.3.1.1.3.1. Metadata

Zero or more meta-data columns can be defined, each represents the normal meta-data annotation on DRL rules.

19.4.3.1.1.3.2. Attributes

Zero or more attribute columns representing any of the DRL rule attributes (e.g. salience, timer, enabled etc) can be added. An additional pseudo attribute is provide in the guided decision table editor to "negate" a rule. Use of this attribute allows complete rules to be negated. For example the following simple rule can be negated as also shown.

```
when
  $c : Cheese( name == "Cheddar" )
then
  ...
end
```

```
when
  not Cheese( name == "Cheddar" )
then
  ...
end
```

19.4.3.1.1.3.3. Simple Condition

Conditions represent constraints on Fact Patterns defined in the left-hand side, or "when" portion, of a rule. To define a condition column you must first select or define a Fact Pattern bound to a model class. You can choose to negate the pattern. Once this has been completed you can define field constraints. If two or more columns are defined using the same fact pattern binding the field constraints become composite field constraints on the same pattern. If you define multiple bindings for a single model class each binding becomes a separate model class in the left-hand side of the rule.

When you edit or create a new column, you will be given a choice of the type of constraint:-

- Literal : The value in the cell will be compared with the field using the operator.
- Formula: The expression in the cell will be evaluated and then compared with the field.
- Predicate : No field is needed, the expression will be evaluated to true or false.

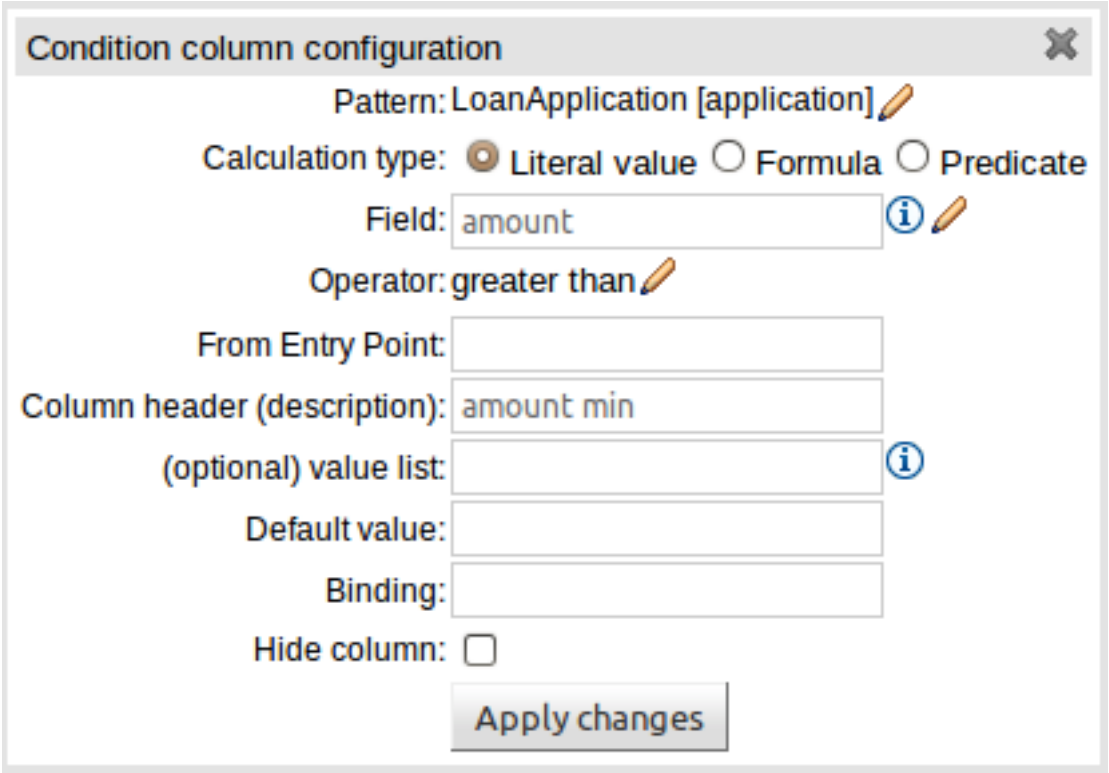


Figure 19.28. Simple Condition popup

19.4.3.1.1.3.4. Set the value of a field

An Action to set the value of a field on previously bound fact. You have the option to notify the Rule Engine of the modified values which could lead to other rules being re-activated.

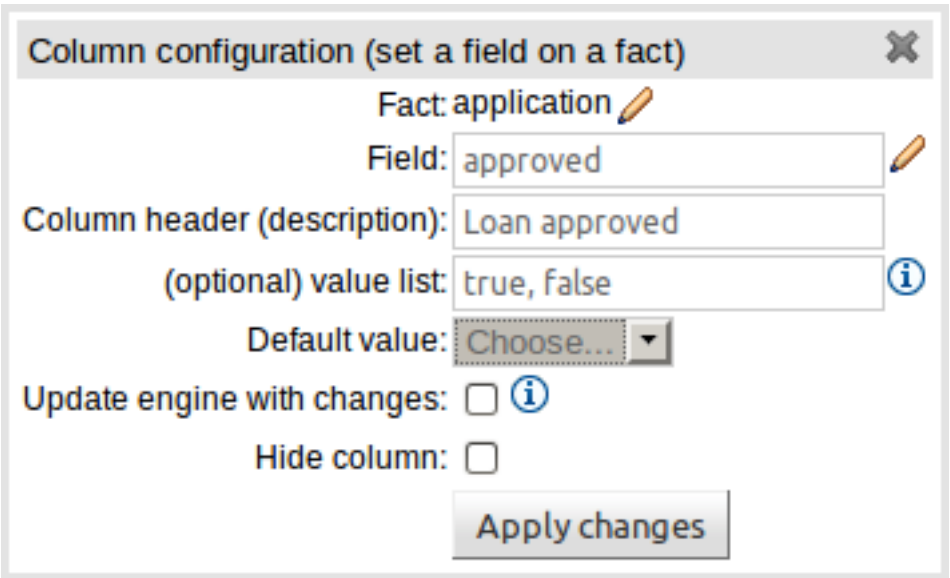


Figure 19.29. Set the value of a field popup

19.4.3.1.1.3.5. Set the value of a field on a new fact

An Action to insert a new Fact into the Rule Engine Working Memory and set the a value of one of the new Facts' fields. You can choose to have the new Fact "logically inserted" meaning it will be automatically deleted should the conditions leading to the action being invoked cease to be true. Please refer to the Drools Expert documentation for details on Truth Maintenance and Logical insertions.

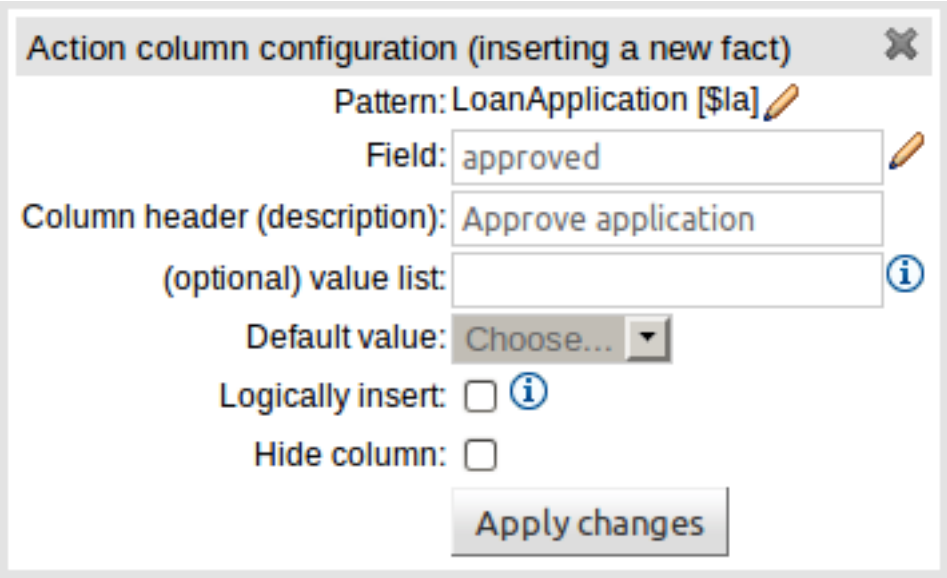


Figure 19.30. Set the value of a field on a new fact popup

19.4.3.1.1.3.6. Delete an existing fact

An Action to delete a bound Fact.

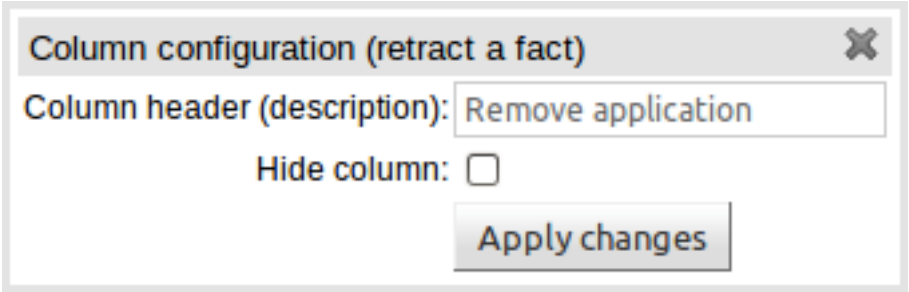


Figure 19.31. Delete an existing fact popup

19.4.3.1.1.4. Advanced column types

19.4.3.1.1.4.1. Condition BRL fragments

A construct that allows a BRL fragment to be used in the left-hand side of a rule. A BRL fragment is authored using the Guided Rule Editor and hence all features available in that editor can be used to define a decision table column; such as "from", "collect" and "accumulate" etc. When using

the embedded Guided Rule Editor field values defined as "Template Keys" will form columns in the decision table. Facts and Fact's fields bound in the BRL fragment can be referenced by the simpler column types and vice-versa.

In the following example two Template Keys have been defined and hence two columns appear in the decision table.

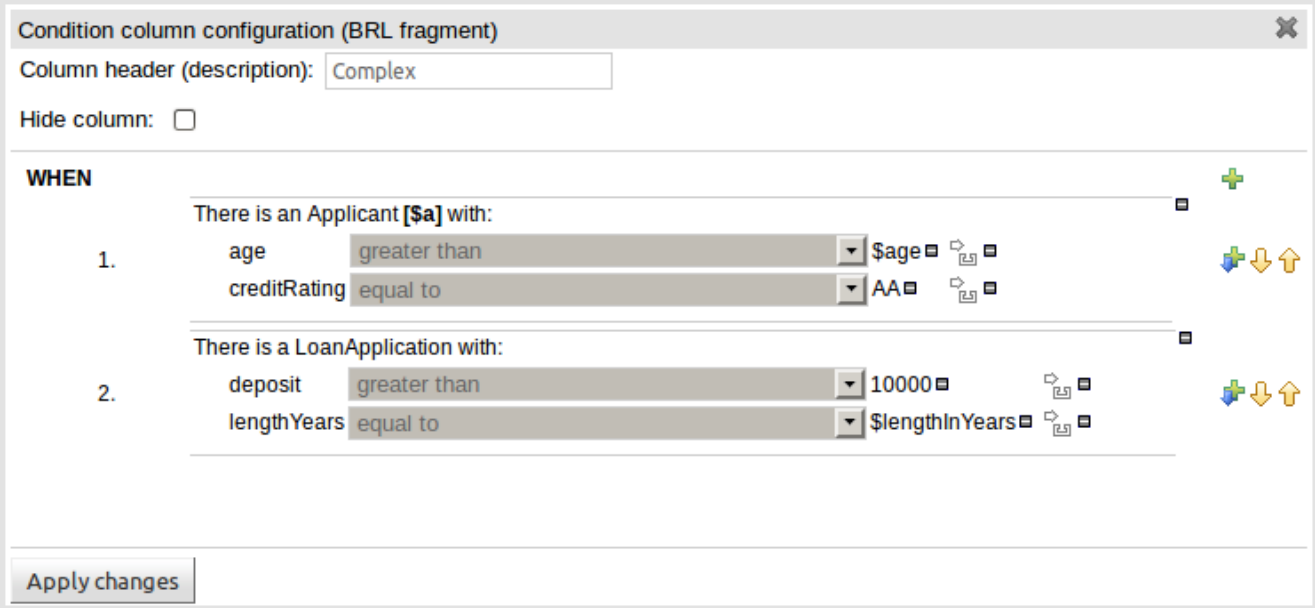
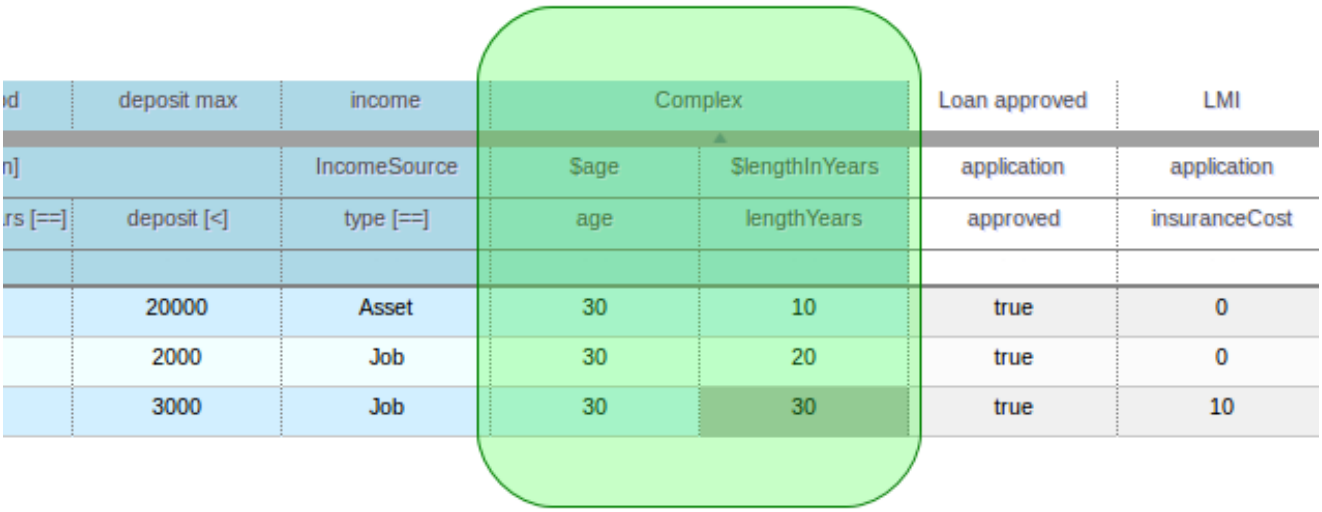


Figure 19.32. Defining a Condition with BRL



d	deposit max	income	Complex		Loan approved	LMI
n]		IncomeSource	$age	$lengthInYears	application	application
rs [==]	deposit [<]	type [==]	age	lengthYears	approved	insuranceCost
	20000	Asset	30	10	true	0
	2000	Job	30	20	true	0
	3000	Job	30	30	true	10

Figure 19.33. The resulting decision table

19.4.3.1.1.4.2. Execute a Work Item

An Action invoking a jBPM Work Item Handler setting its input parameters to bound Facts\Facts fields values.

19.4.3.1.1.4.3. Set the value of a field with a Work Item parameter

An Action setting the value of a Fact's field to that of a jBPM Work Item Handler's result parameter.

19.4.3.1.1.4.4. Set the value of a field on a new Fact with a Work Item parameter

An Action setting the value of a new Fact's field to that of a jBPM Work Item Handler's result parameter.

19.4.3.1.1.4.5. Action BRL fragment

A construct that allows a BRL fragment to be used in the right-hand side of a rule. A BRL fragment is authored using the Guided Rule Editor and hence all features available in that editor can be used to define a decision table column. When using the embedded Guided Rule Editor field values defined as "Template Keys" will form columns in the decision table. Facts bound in the BRL fragment can be referenced by the simpler column types and vice-versa.

In the following example two Template Keys have been defined and hence two columns appear in the decision table.

Figure 19.34. Defining an Action with BRL

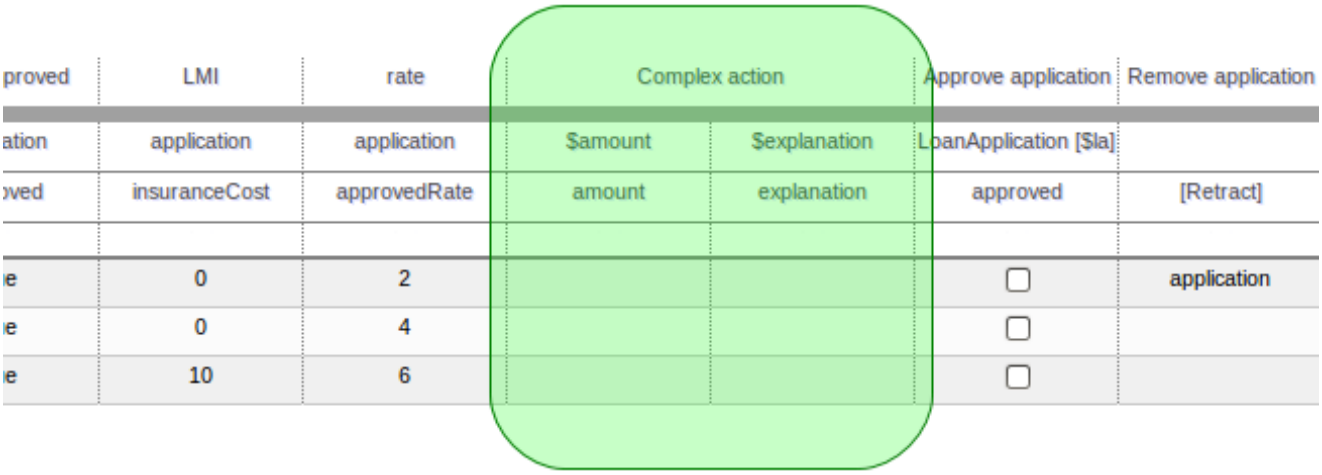


Figure 19.35. The resulting decision table

19.4.3.2. Using a Wizard

A Wizard can also be used to assist with defining the decision table columns.

The wizard can be chosen when first electing to create a new rule. The wizard provides a number of pages to define the table:-

- Summary
- Add Fact Patterns
- Add Constraints
- Add Actions to update facts
- Add Actions to insert facts
- Columns to expand

19.4.3.2.1. Selecting the wizard

The "New Wizard" dialog shows a "Use wizard" checkbox.

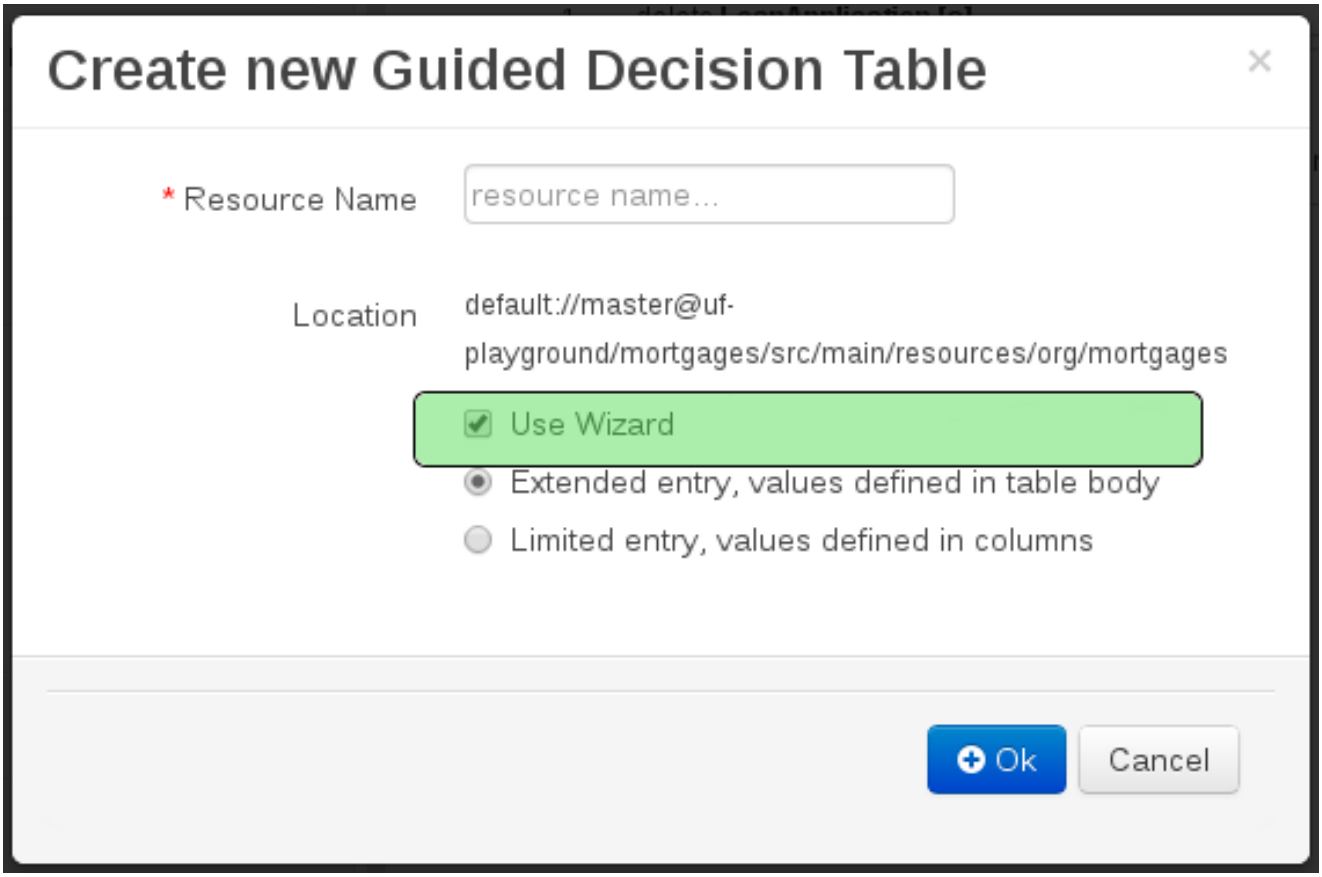


Figure 19.36. Selecting the wizard

19.4.3.2.2. Summary page

The summary page shows a few basic details about the decision table and allows the asset name to be changed.

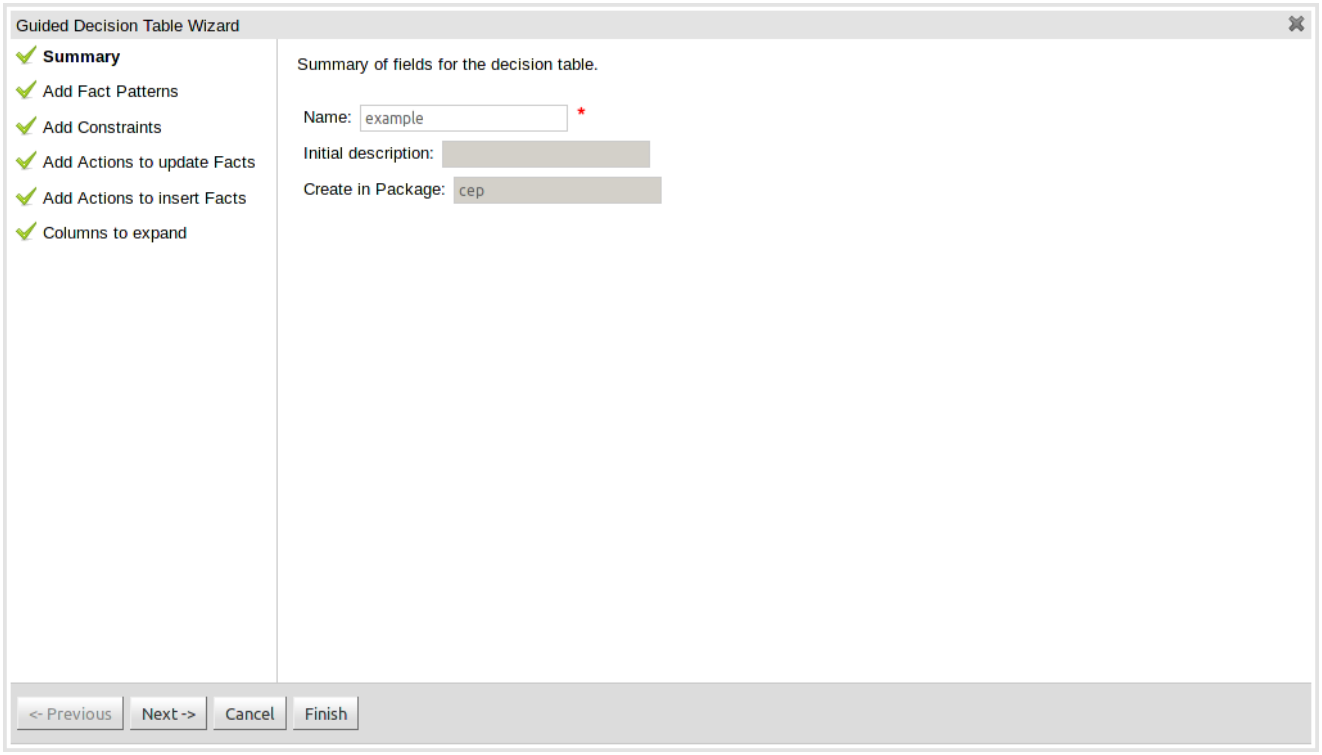


Figure 19.37. Summary page

19.4.3.2.3. Add Fact Patterns page

This page allows Fact types to be defined that will form the "When" columns of the rules. Fact types that are available in your model will be shown in the left-hand listbox. Select a Fact type and use the ">>" button to add it to your list of chosen facts on the right-hand listbox. Removal is a similar process: the Fact that is no longer required can be selected in the right-hand listbox and the "<<" button used to remove it. All Fact types need to be bound to a variable. Incomplete Fact types will be highlighted and a warning message displayed. You will be unable to finish your definition until all warnings have been resolved.

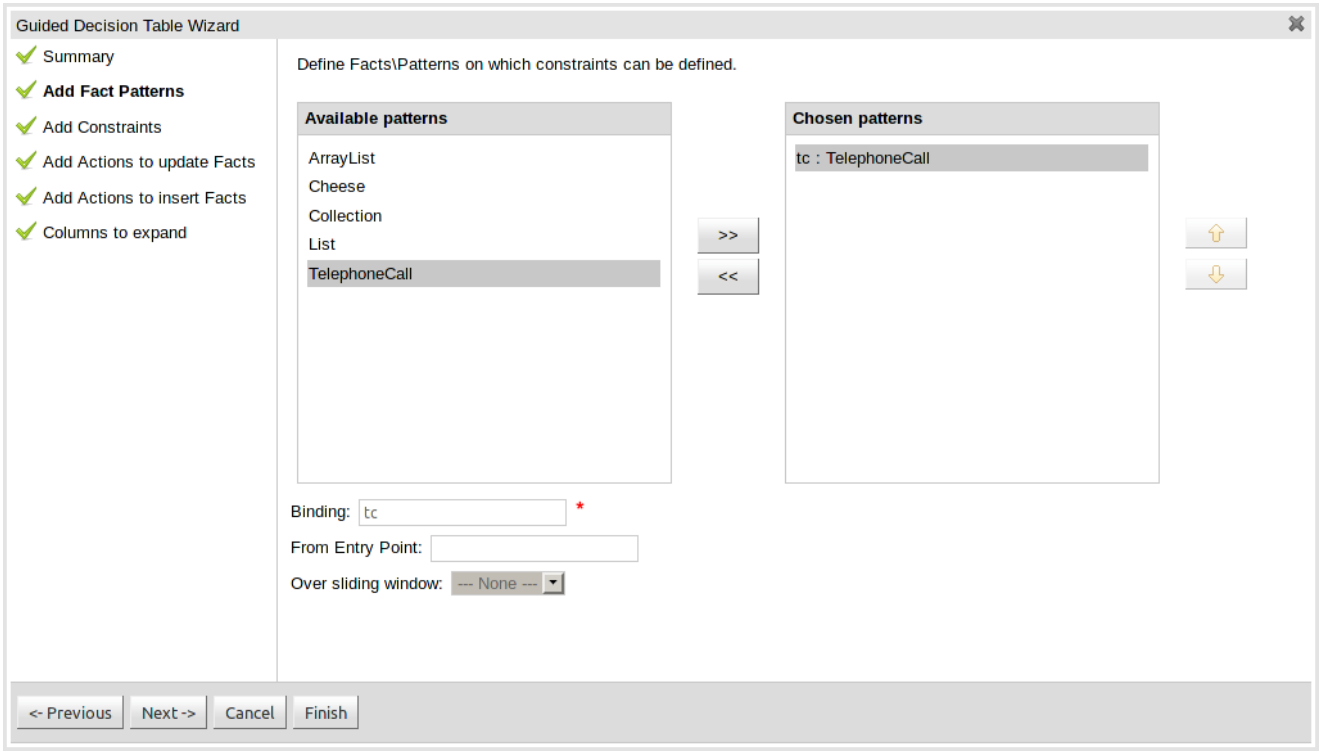


Figure 19.38. Add Fact Patterns page

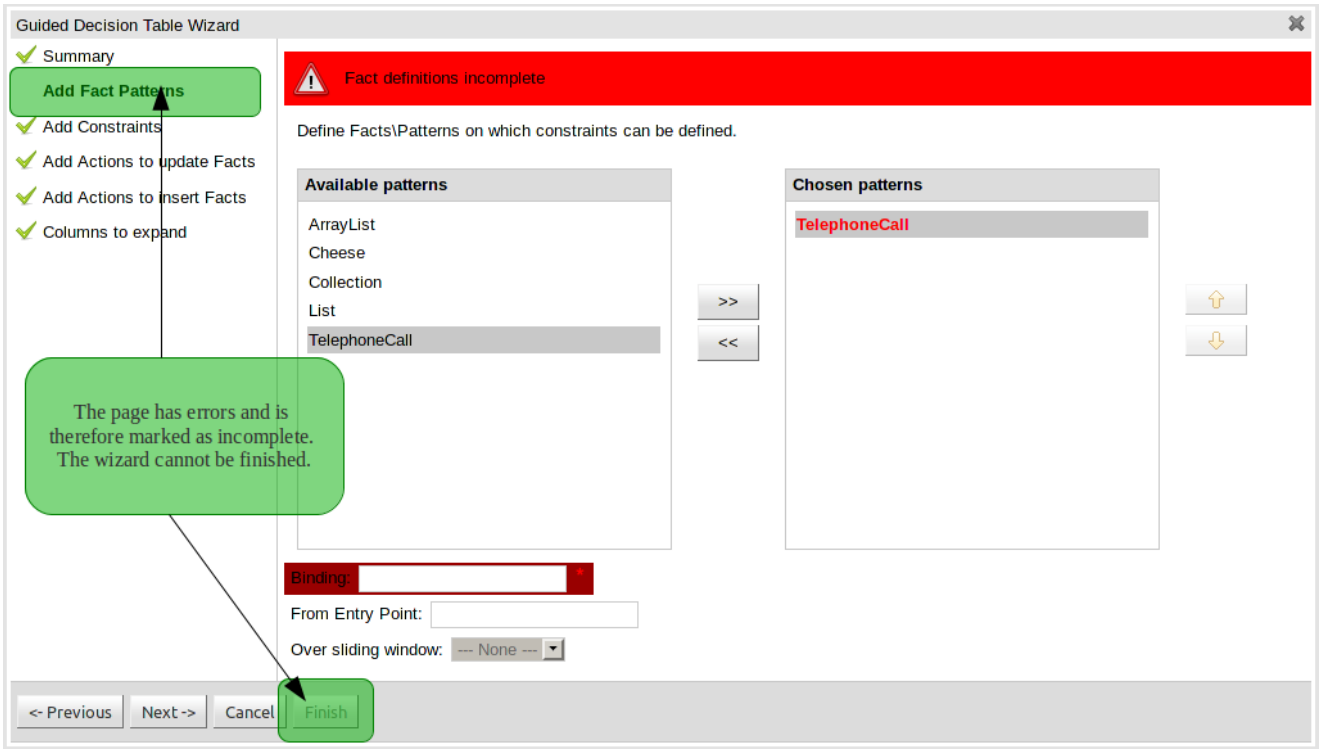


Figure 19.39. Example of an incomplete Fact definition

19.4.3.2.4. Add Constraints page

This page allows field constraints on the Fact types you have chosen to use in the decision table to be defined. Fact types chosen on the previous Wizard page are listed in the right-hand listbox. Selecting a Fact type by clicking on it will result in a list of available fields being shown in the middle listbox together with an option to create a predicate that do not require a specific field. Fields can be added to the pattern's constraints by clicking on the field and then the ">>" button. Fields can be removed from the pattern definition by clicking on the Condition in the right-hand listbox and then the "<<" button. All fields need to have a column header and operator. Incomplete fields will be highlighted and a warning message displayed. You will be unable to finish your definition until all warnings have been resolved.

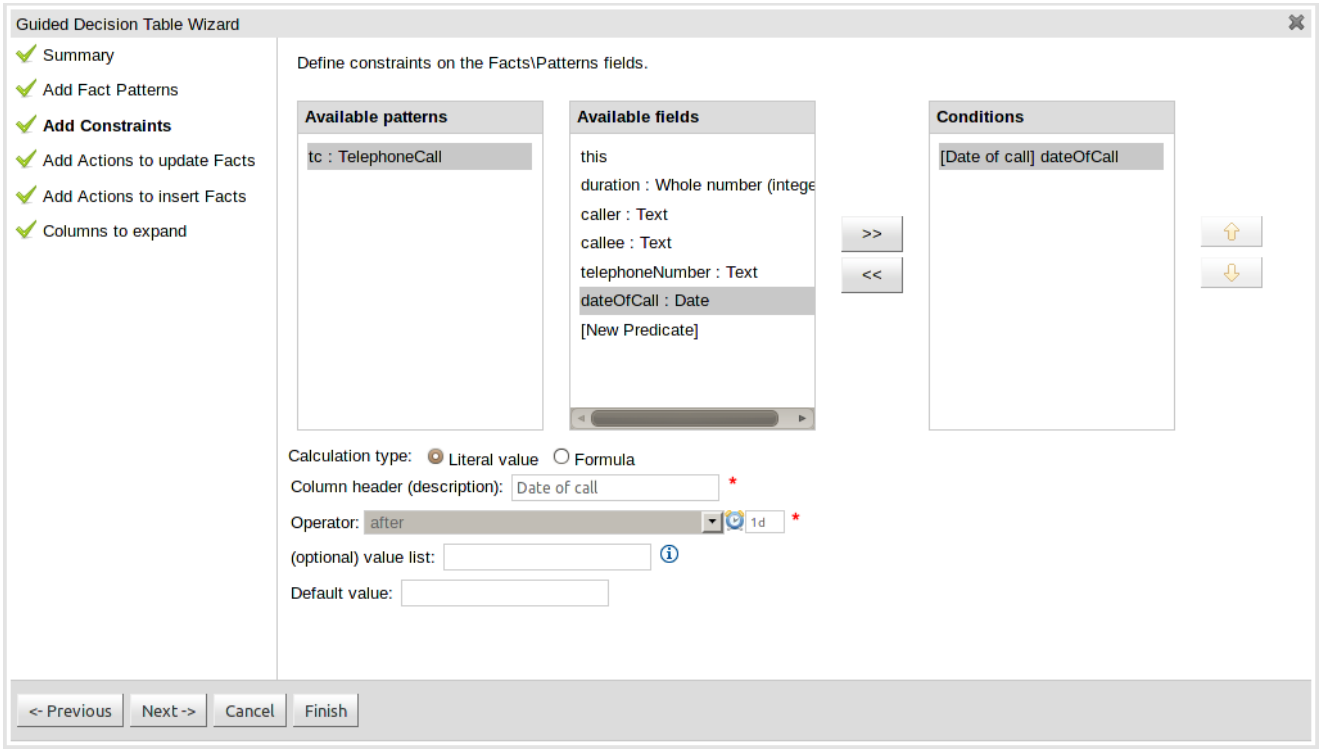


Figure 19.40. Add Constraints page

19.4.3.2.5. Add Actions to update facts page

Fact types that have been defined can be updated in the consequence, or action, part of a rule. This page allows such actions to be defined. Fact types added to the decision table definition are listed in the left-hand listbox. Selecting a Fact type by clicking on it will result in a list of available fields being shown in the middle listbox. Fields that need to be updated by the rule can be added by selecting an available field and pressing the ">>" button. Fields can be removed similarly by clicking on a chosen field and then the "<<" button. All actions require a column header. Any incomplete actions will be highlighted and a warning message displayed. You will be unable to finish your definition until all warnings have been resolved.

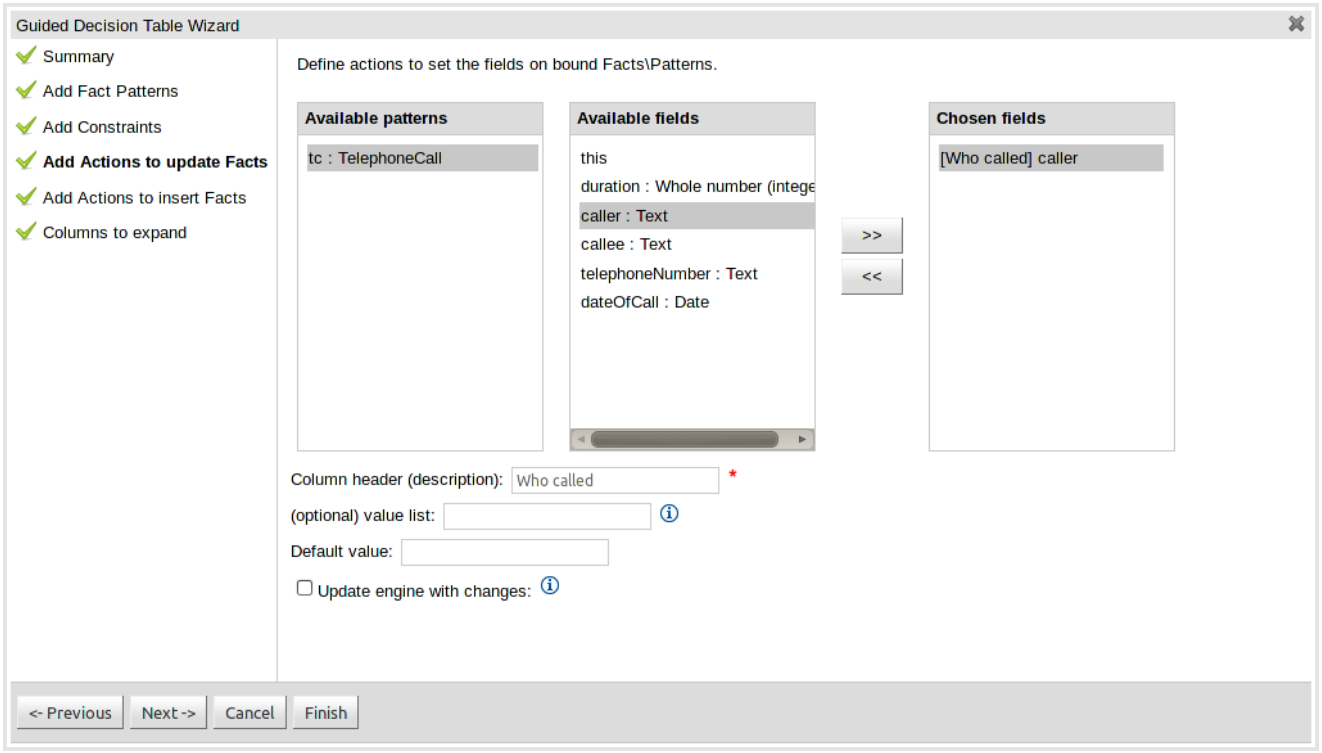


Figure 19.41. Add Actions to update facts page

19.4.3.2.6. Add Actions to insert facts page

Actions can also be defined to insert new Facts into the Rule Engine. A list of Fact types available in your model are listed in the left-hand listbox. Select those you wish to include in your decision table definition by clicking on them and pressing the ">>" button between the left most listbox and that titled "Chosen patterns". Removal is a similar process whereby a chosen pattern can be selected and removed by pressing the "<<" button. Selection of a chosen pattern presents the user with a list of available fields. Fields that need to have values set by the action can be added by selecting them and pressing the ">>" button between the "Available fields" and "Chosen fields" listbox. Removal is a similar process as already described. New Facts need to be bound to a variable and have a column heading specified. Incomplete Facts and\or fields will be highlighted and a warning message displayed. You will be unable to finish your definition until all warnings have been resolved.

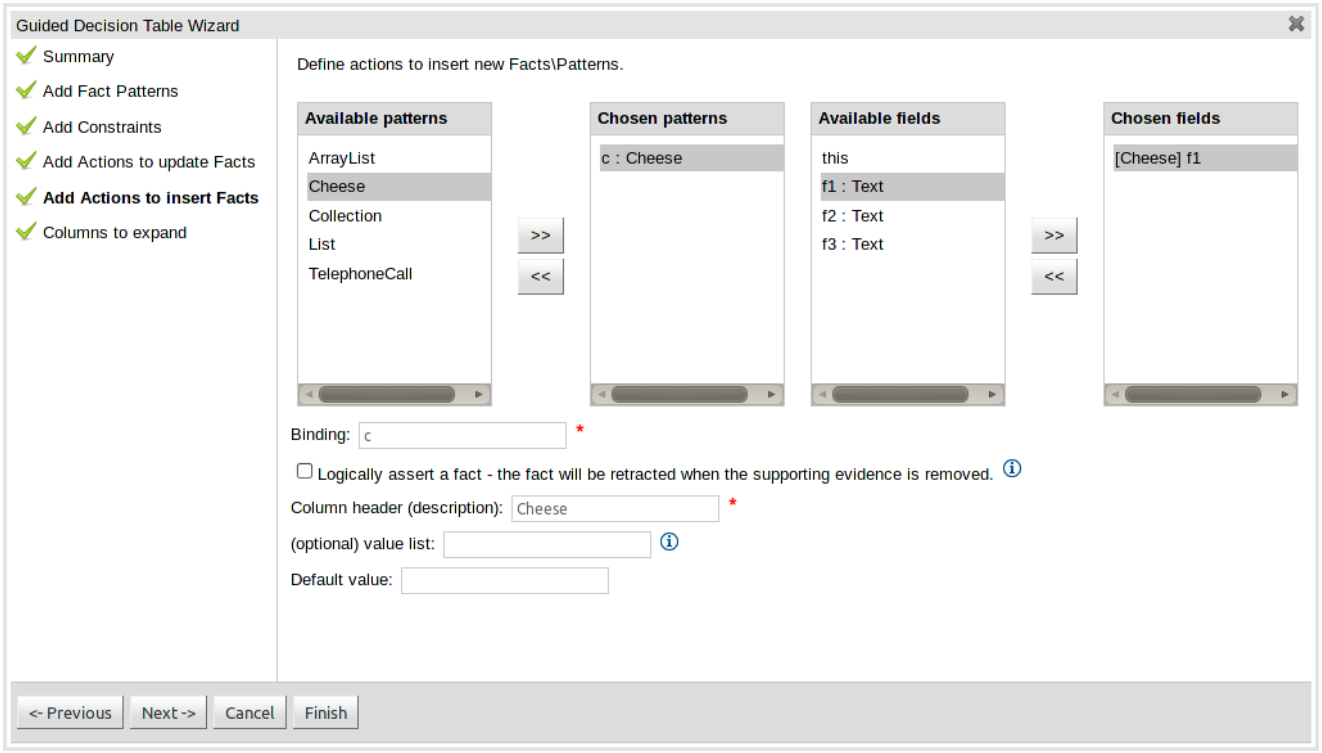


Figure 19.42. Add Actions to insert facts page

19.4.3.2.7. Columns to expand page

This page controls how the decision table, based upon Conditions defined on the prior pages, will be created. Condition columns defined with an optional list of permitted values can be used to create rows in the decision table. Where a number of Condition columns have been defined with lists of permitted values the resulting table will contain a row for every combination of values; i.e. the decision table will be in expanded form. By default all Condition columns defined with value lists will be included in the expansion however you are able to select a sub-set of columns if so required. This can be accomplished by unticking the "Fully expand" checkbox and adding columns to the right-hand listbox. If no expansion is required untick the "Fully expand" checkbox and ensure zero columns are added to the right-hand listbox.

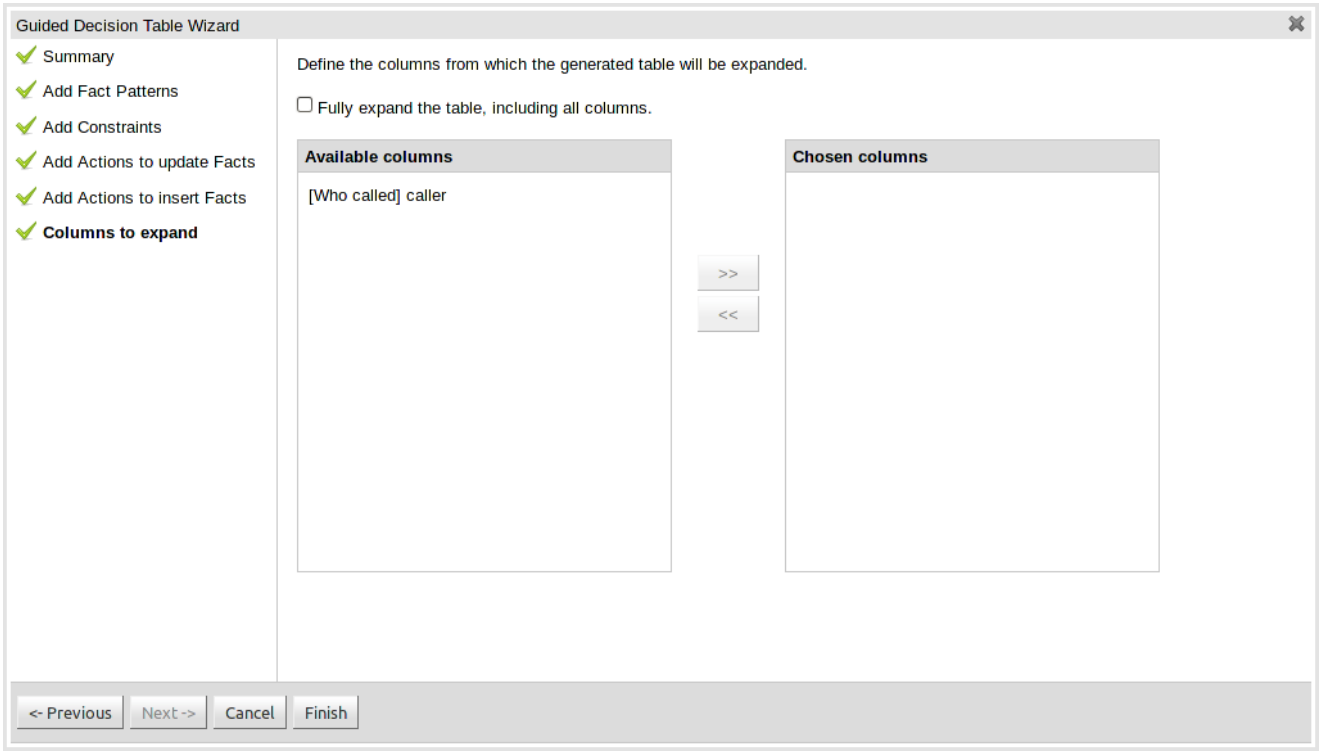


Figure 19.43. Columns to expand page

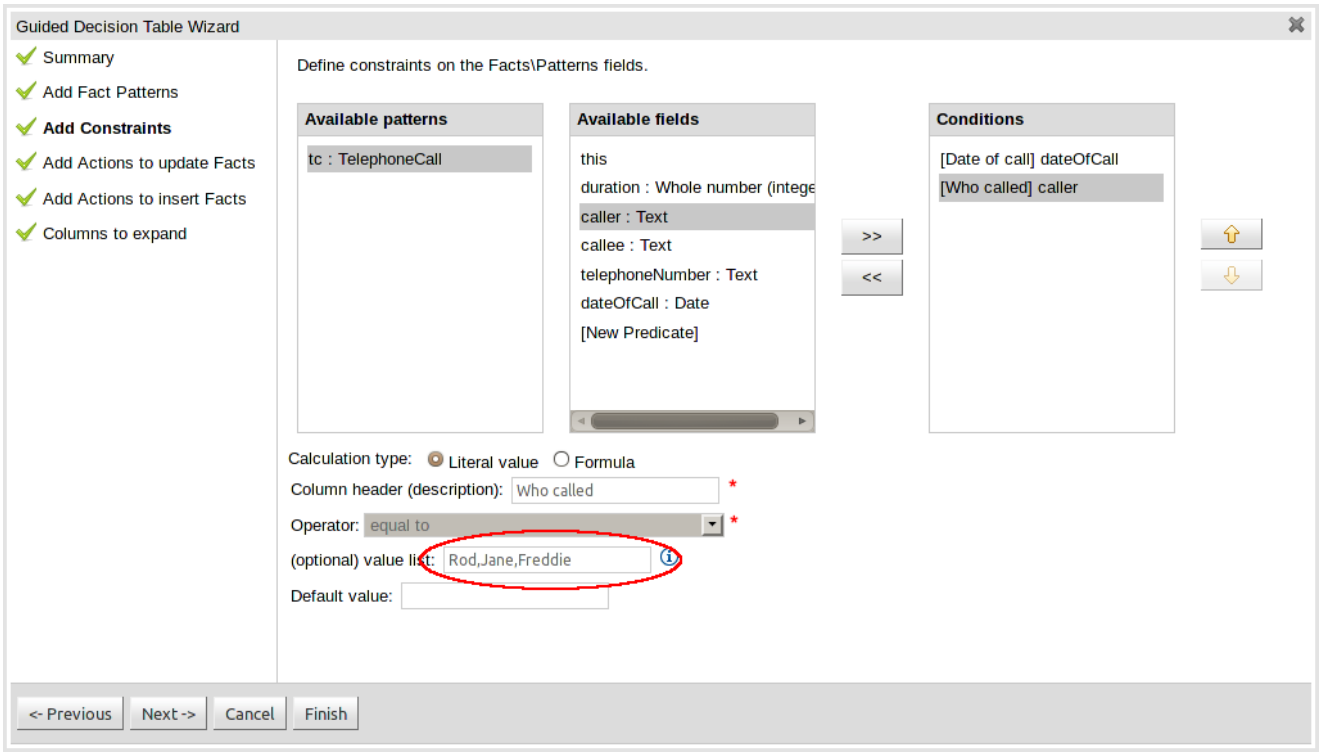


Figure 19.44. Example of a Condition column with optional values defined

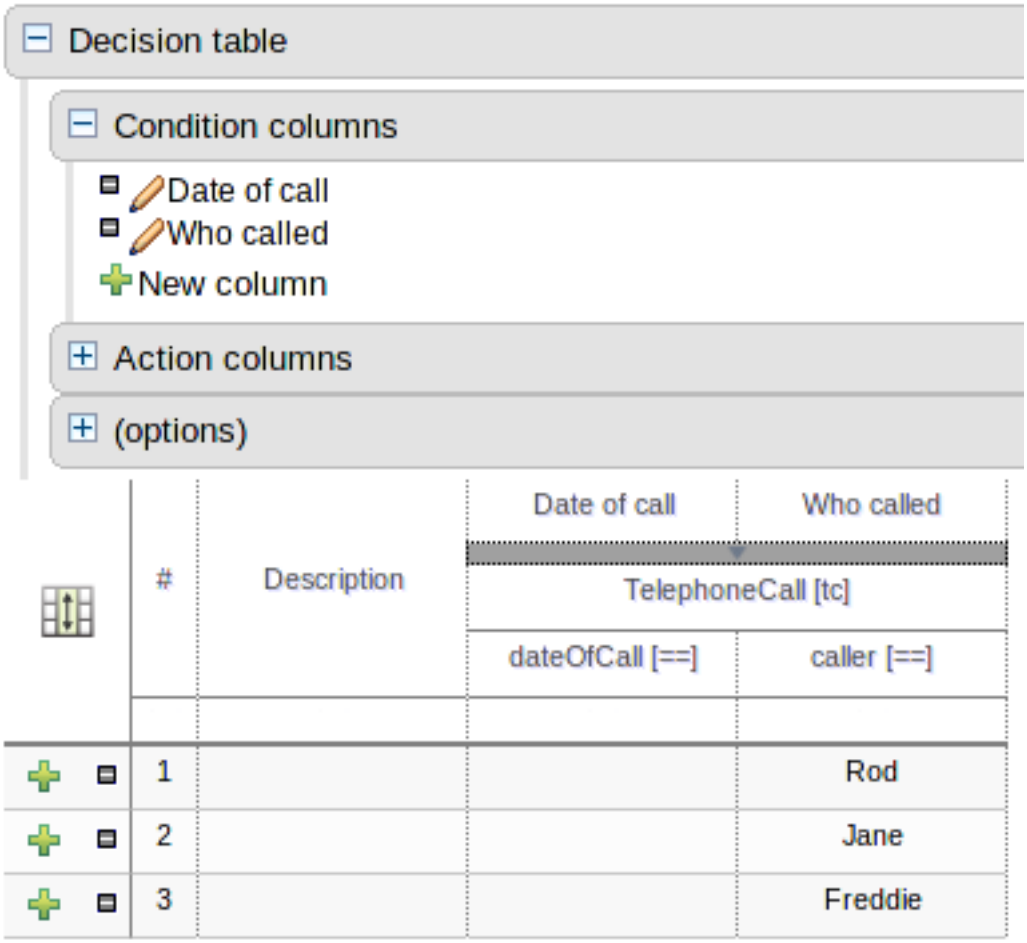


Figure 19.45. Example of a decision table generated with expanded columns

19.4.4. Rule definition

This section allows individual rules to be defined using the columns defined earlier.

Rows can be appended to the end of the table by selecting the "Add Row" button. Rows can also be inserted by clicking the "+" icon beside an existing row. The "-" icon can be used to delete rows.

	#	Description	min-age	max-age	policy type	make	model	premium
			Applicant [$a]		Policy [$p]	Vehicle [$v]		$p
			age [>=]	age [<]	type [==]	make [==]	model [==]	premium
	1		18	25	TPFT	BMW	318i	1000
	2		18	25	COMP	BMW	318i	1500
	3		18	25	TPFT	BMW	M3	2000
	4		18	25	COMP	BMW	M3	2500
	5		18	25	TPFT	Audi	A4	1500
	6		18	25	COMP	Audi	A4	2000
	7		18	25	TPFT	Audi	R8	2500
	8		18	25	COMP	Audi	R8	3000

Add row... | Otherwise | Analyze... | Audit log

Figure 19.46. Rule definition

19.4.5. Audit Log

An audit log has been added to the web-guided Decision Table editor to track additions, deletions and modifications.

By default the audit log is not configured to record any events, however, users can easily select the events in which they are interested.

The audit log is persisted whenever the asset is checked in.

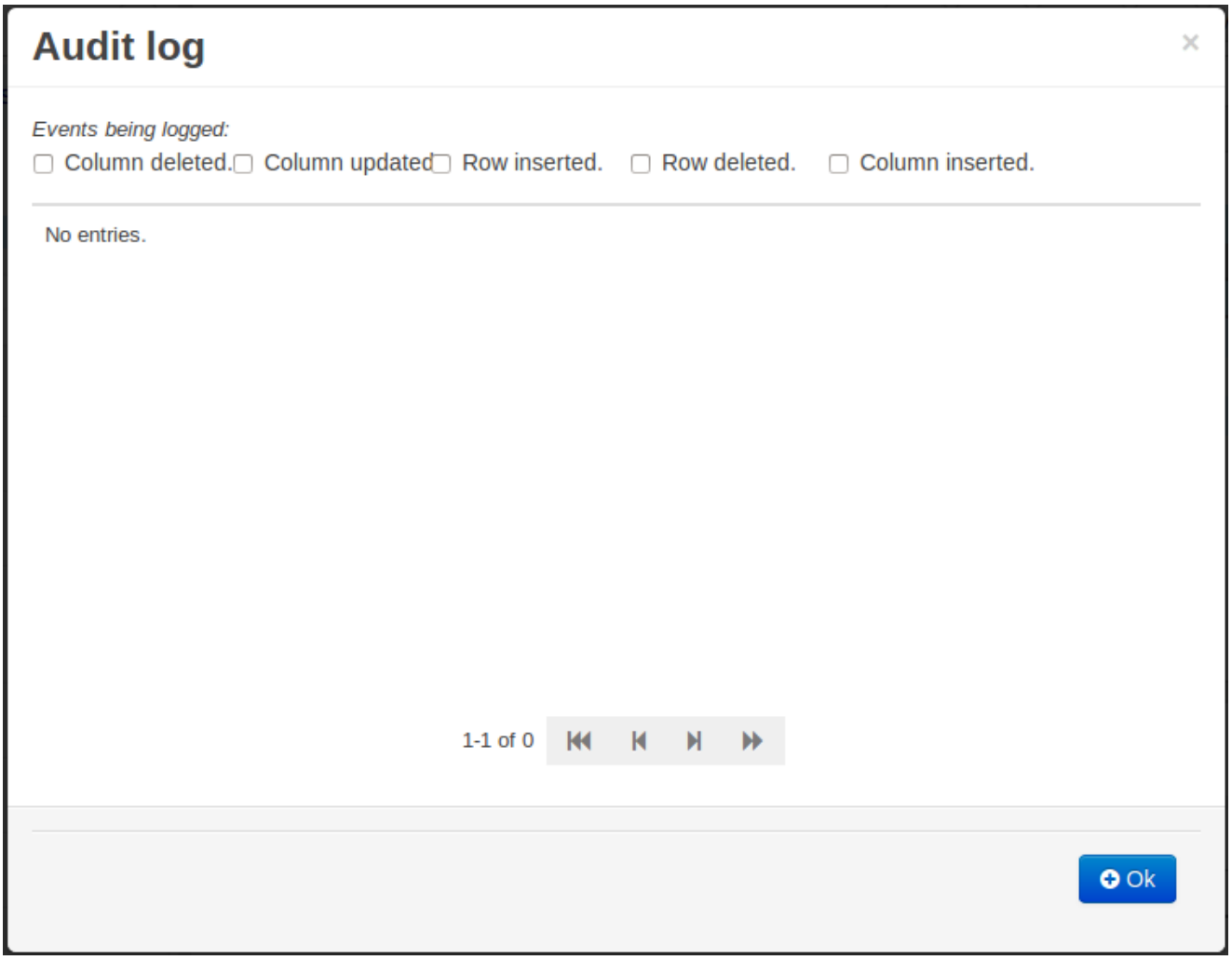


Figure 19.47. An empty audit log

Once the capture of events has been enabled all susbsequent operations are recorded. Users are able to perform the following:-

- Record an explanatory note beside each event.
- Delete an event from the log. Event details remain in the underlying repository.

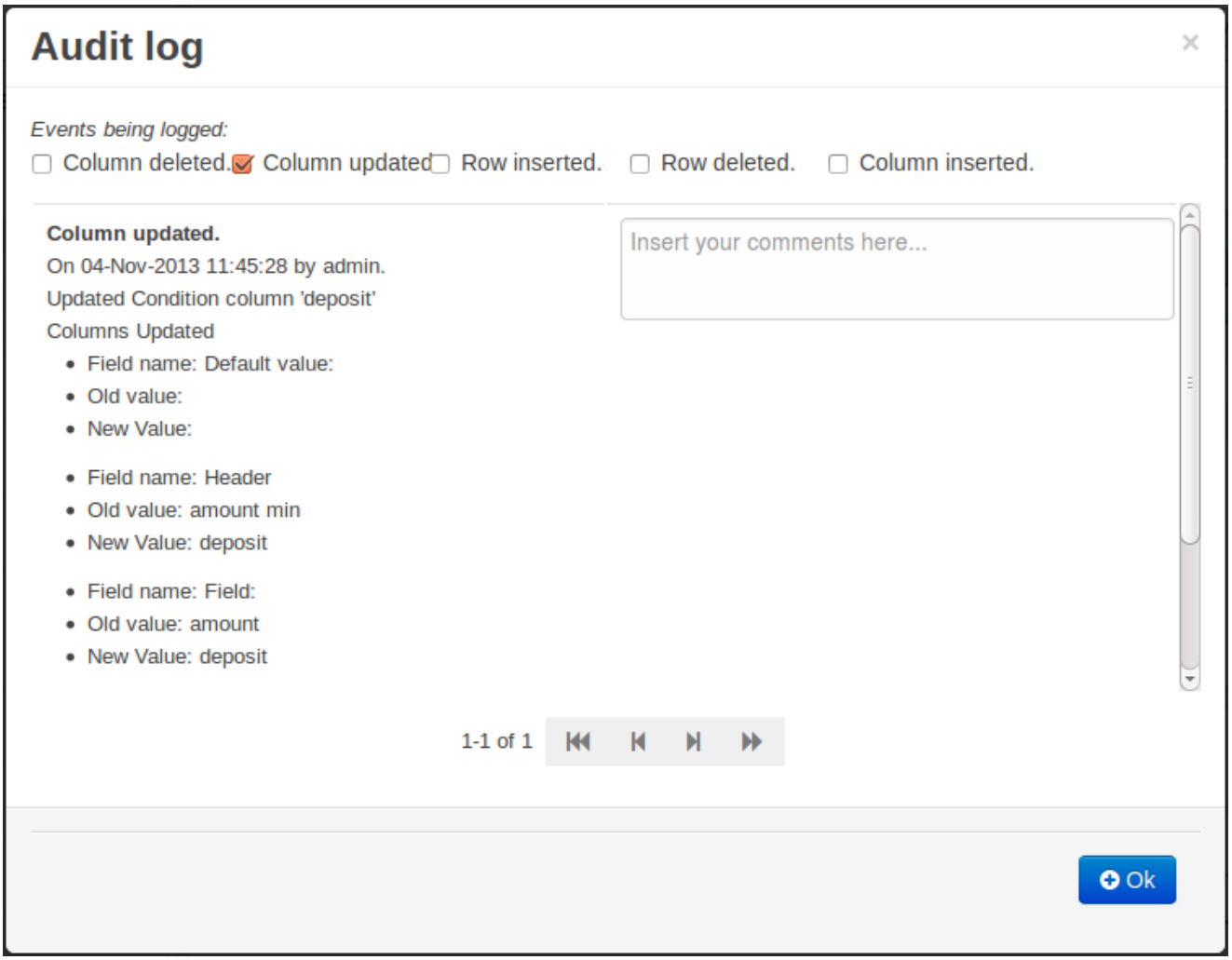


Figure 19.48. Example of audit events

19.4.6. Real Time Validation and Verification

Decision tables are validated after each cell change. If any issues are found the results will be shown in the column on the right side of the table. Validation and verification covers the following issues:

19.4.6.1. Redundancy

Redundancy exists between two rows when both rows execute the same actions when given the same set of facts.

Redundancy might not be a problem if the redundant rules are setting a value on an existing fact, this just sets the value twice. Problems occur when the two rules increase a counter or add more facts into the working memory, since this causes unwanted side effects. In both cases the other row is not needed.

19.4.6.2. Subsumption

Subsumption exists when one row executes the same action as another row when given the same set of facts. The rows are not redundant since the another row is more complicated and it can fire with a set of facts that do not fire the subsumptant row.

The problems with subsumption are similar to the case with redundancy.

19.4.6.3. Conflicts

Conflicts can exist either on a single row or between rows.

A single row conflict prevents the row actions from ever being executed.

Conflict between two rows exists when the conditions of two rules are met with a same set of facts, but the actions set fact fields to different values. The conditions can be redundant or just subsumptant. Conflicts are a problem because it is impossible to know what action is made last. Conditions might set a loan to both approbed and rejected. The end result may be different on each time the rules are ran and with each rule engine software version.

19.4.6.4. Missing Columns

In some cases, usually by accident, the user can delete all the condition or action columns. When the conditions are removed all the actions are executed and when the actions columns are missing the rows do nothing.

19.5. Guided Decision Trees

The Workbench supports authoring of simple Decision Trees.

Note

The editor does not support nested Data Objects at present. It is therefore advised to only use Guided Decision Trees with flat Data Object models.

19.5.1. The initial editor layout

When a new Guided Decision Tree is created the editor is initially blank.

The left-hand side is a palette of available Data Objects, their fields and Actions.

The right-hand side is the area where you can drag and drop Data Objects, their fields or Actions to build a tree.

The editor will show a connector between the node being dragged and applicable children to which it can be attached. When the drag is complete the new node will be attached to the applicable child.

Root nodes will not have a connector shown when being dragged to an empty tree. Completing the drag positions the root node in the centre of the editor.

There are various restrictions when composing a tree:-

1. A tree must have a Data Object at the root.
2. A tree can only have one root.
3. Data Objects can have either other Data Objects, field constraints or Actions as children.

   The field constraints must be on fields of the same Data Object as the parent node.
4. Field constraints can have either other field constraints or Actions as children.

   The field constraints must be on fields of the same Data Object as the parent node.
5. Actions can only have other Actions as children.

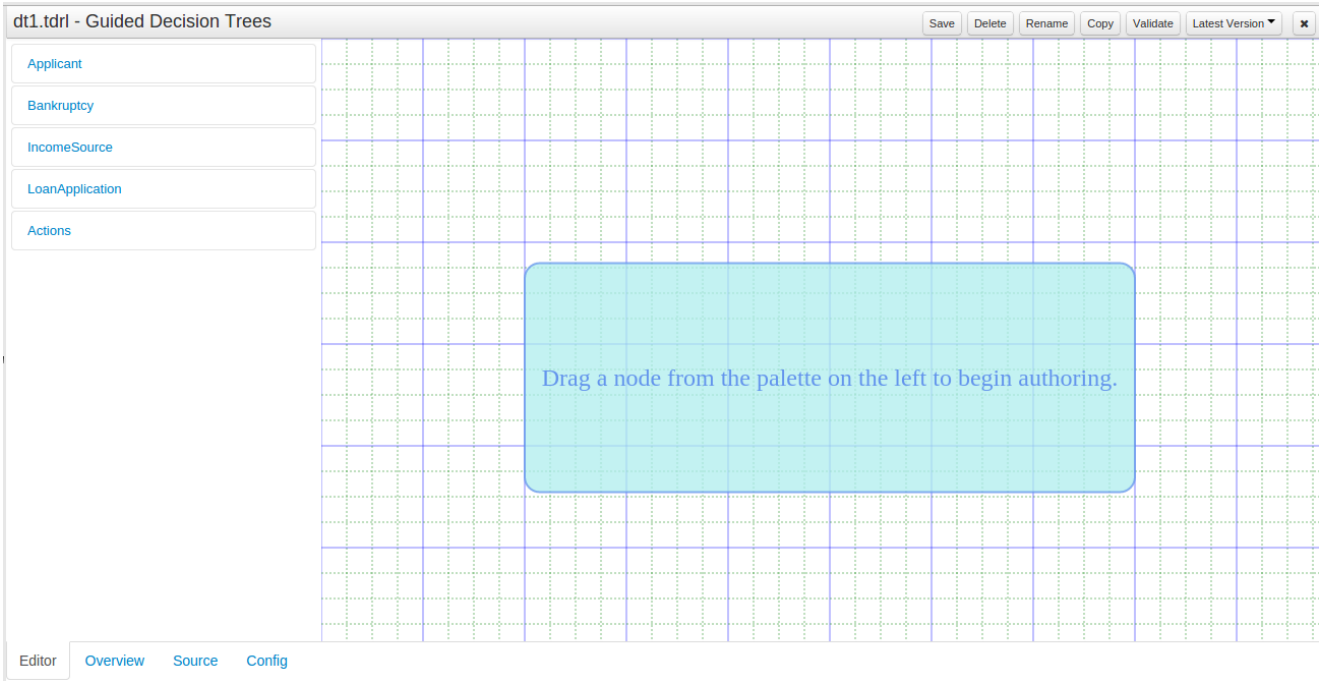


Figure 19.49. Guided Decision Trees - Empty editor

Expanding the palette reveals Tree Nodes for the Data Object and its fields.

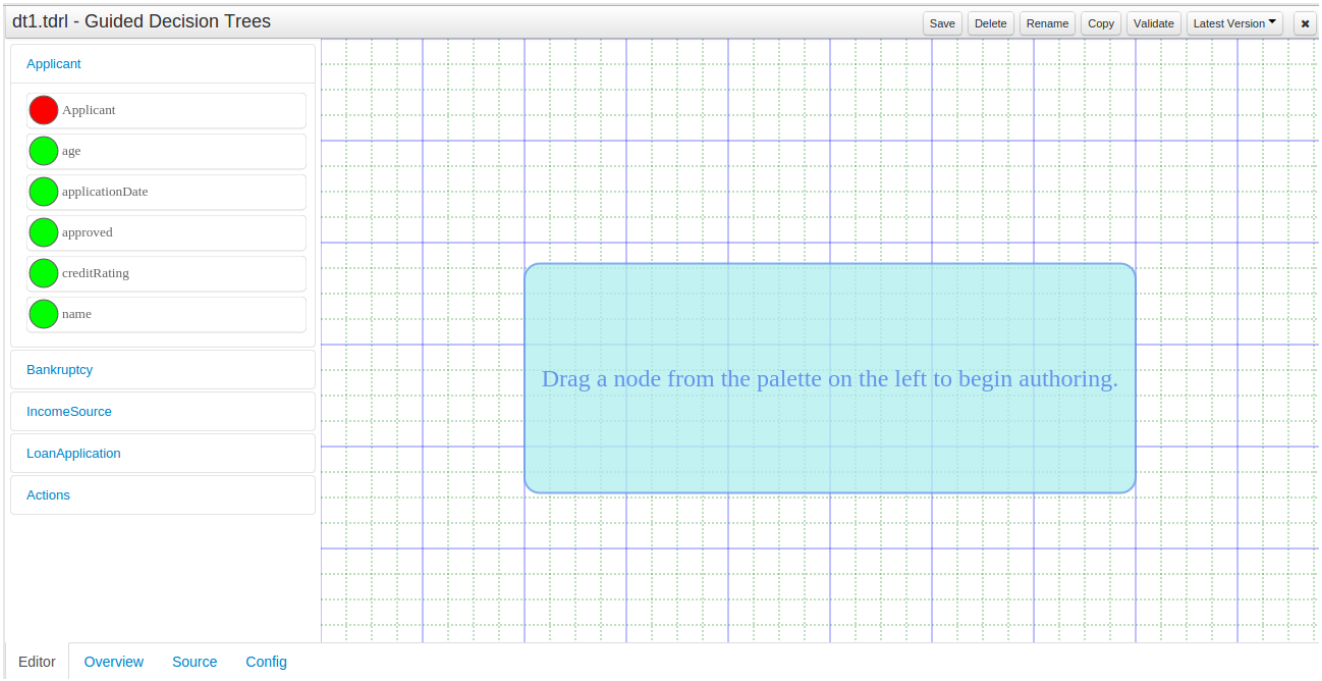


Figure 19.50. Guided Decision Trees - Expanded palette

19.5.2. First steps

Drag a Data Object on to the tree authoring area.

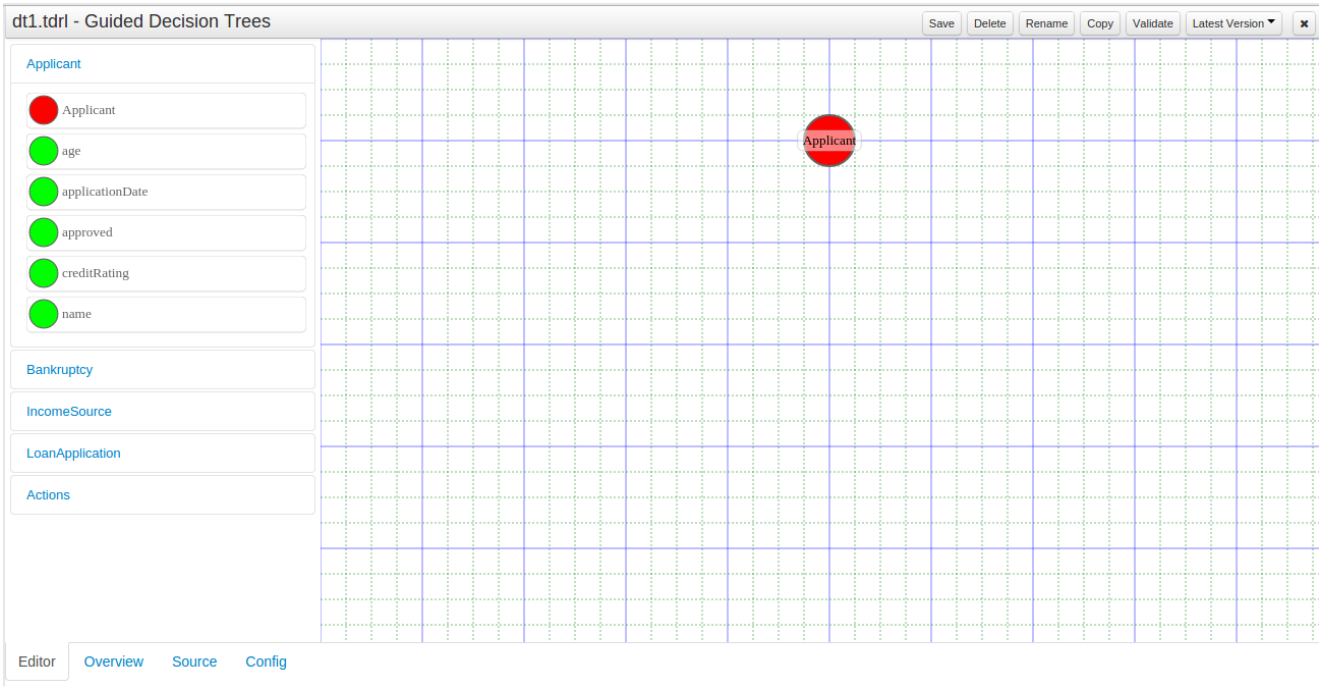


Figure 19.51. Guided Decision Trees - Data Object root node

Clicking on a node selects it.

Icons to manipulate the node appear when the node is selected.

The icons are:

1. Delete

   Deleting a node will also delete all children.

2. Edit

   Collapsed nodes cannot be edited as they contain numerous children.

3. Collapse

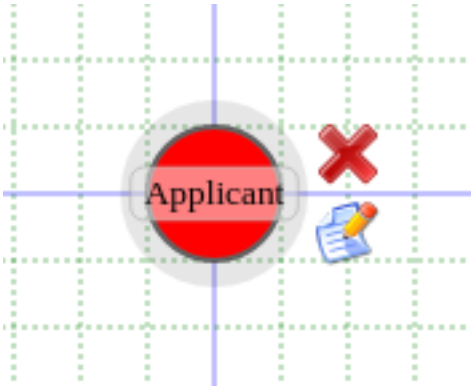


Figure 19.52. Guided Decision Trees - Selected node

19.5.3. Editing Data Object nodes

Selecting a Data Object node and clicking the edit icon shows a popup to manage the node.

The popup shows the Data Object type and allows it to be bound to a variable. Bound Data Objects can be modified or retracted by Actions.

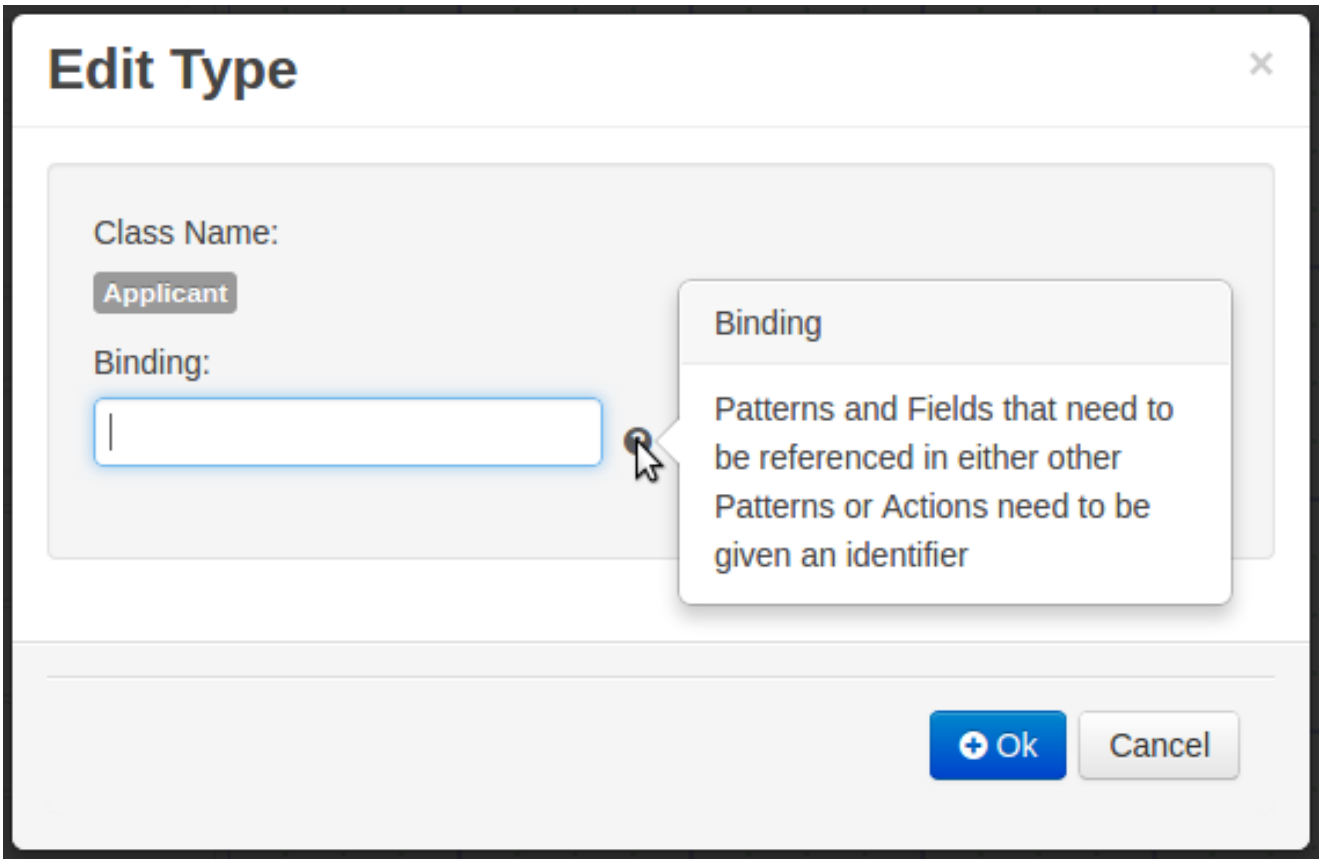


Figure 19.53. Guided Decision Trees - Data Object Editor

19.5.4. Editing Field Constraint nodes

Selecting a Field Constraint node and clicking the edit icon shows a popup to manage the node.

The popup shows the Data Object type and field and allows the field to be bound to a variable. An operator, applicable to the Data Model field's data-type, can be selected and a corresponding value entered.

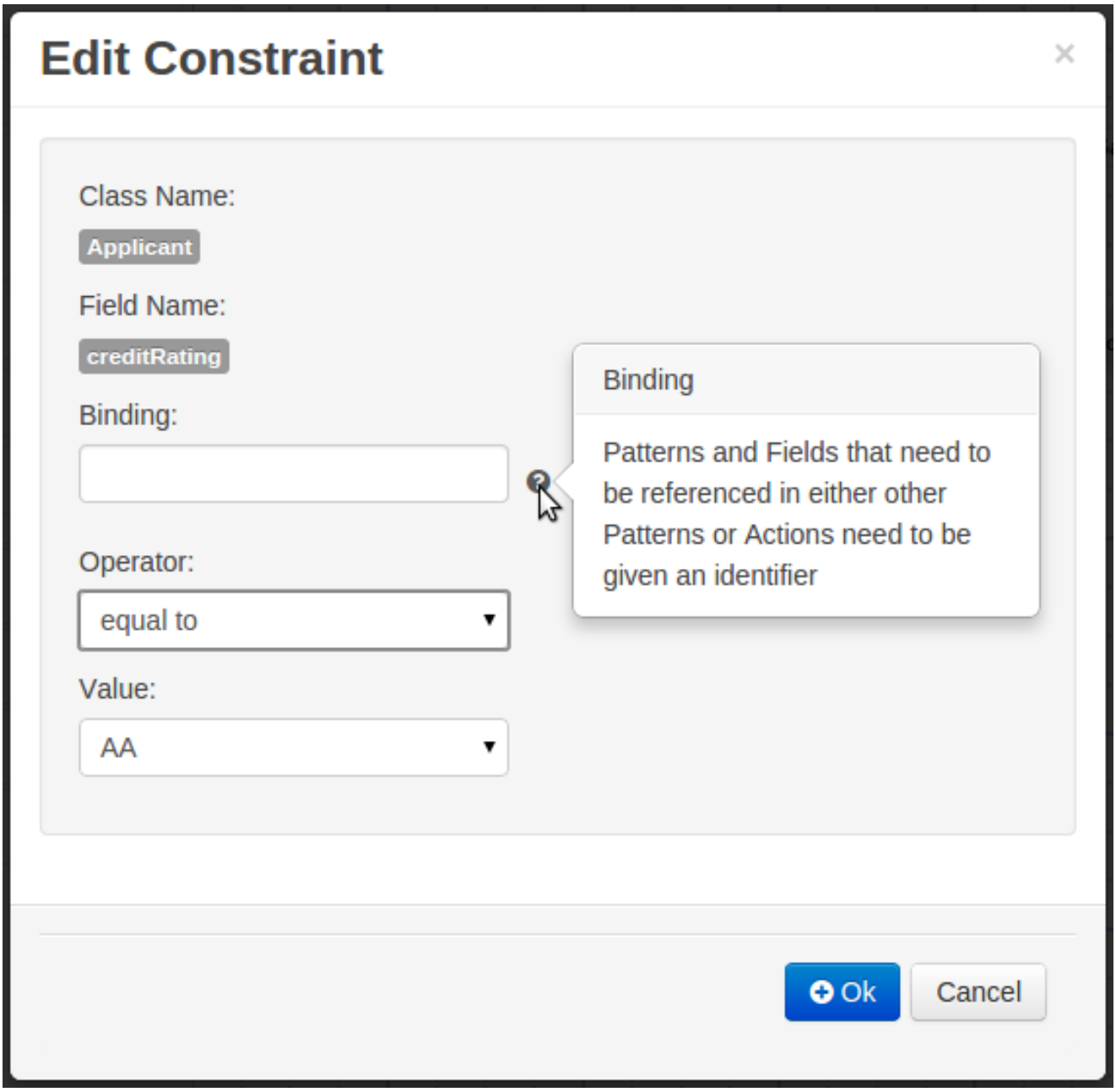


Figure 19.54. Guided Decision Trees - Field Constraint Editor

19.5.5. Editing Action nodes

Selecting an Insert Action node and clicking the edit icon shows a popup to manage the node.

The popup allows selection of the Data Object to be inserted and whether it's insertion is "logical". Please refer to Drools documentation regarding Truth Maintenance for more information. Fields for the new Data Object can have values set.

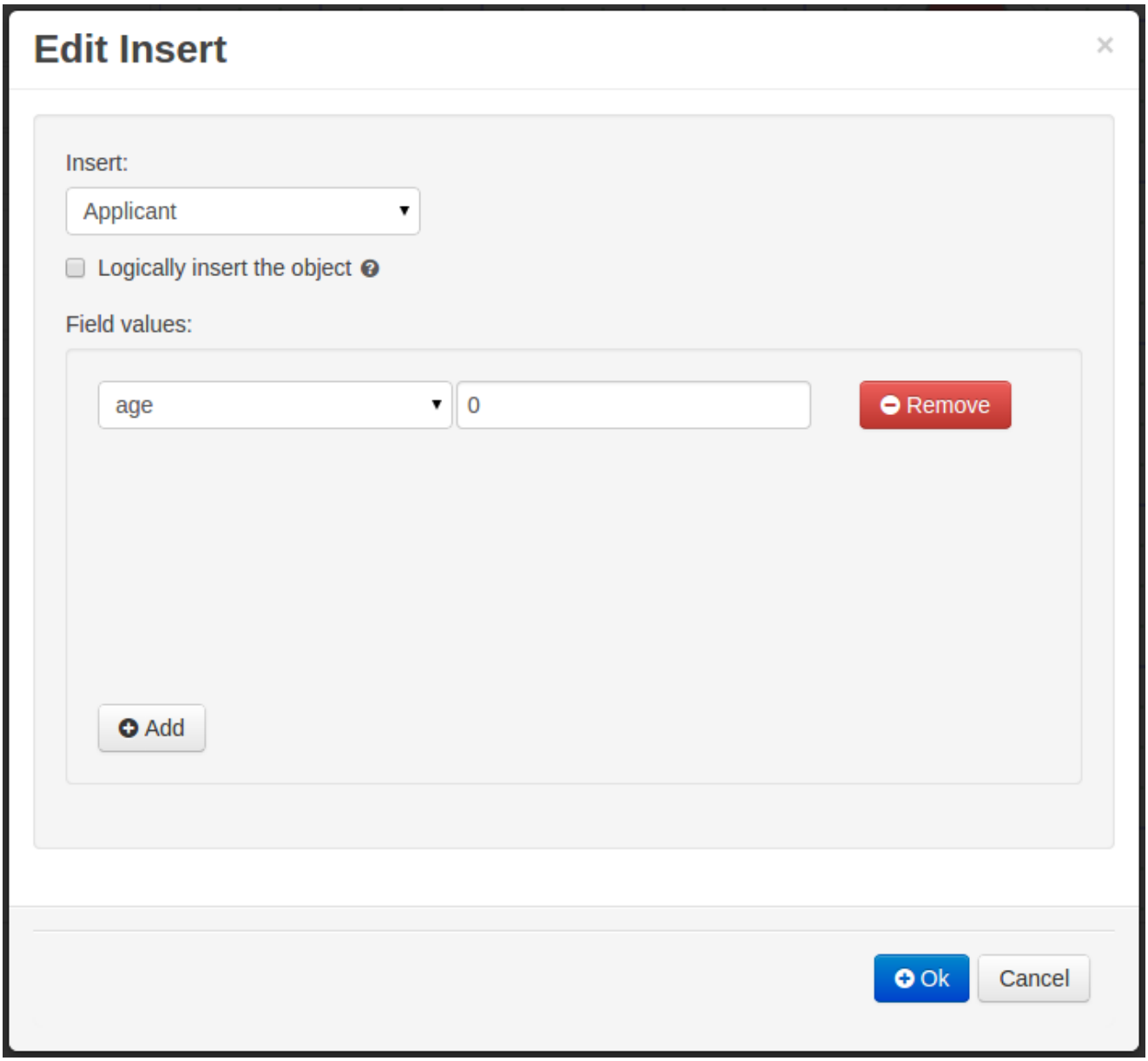


Figure 19.55. Guided Decision Trees - Action "Insert" Editor

Selecting an Insert Retract node and clicking the edit icon shows a popup to manage the node.

The popup allows any Data Object bound in the path from the selected node to the root node to be selected for retraction.

Figure 19.56. Guided Decision Trees - Action "Retract" Editor

Selecting an Insert Update node and clicking the edit icon shows a popup to manage the node.

The popup allows any Data Object bound in the path from the selected node to the root node to be modified. Fields for the modified Data Object can have values set.

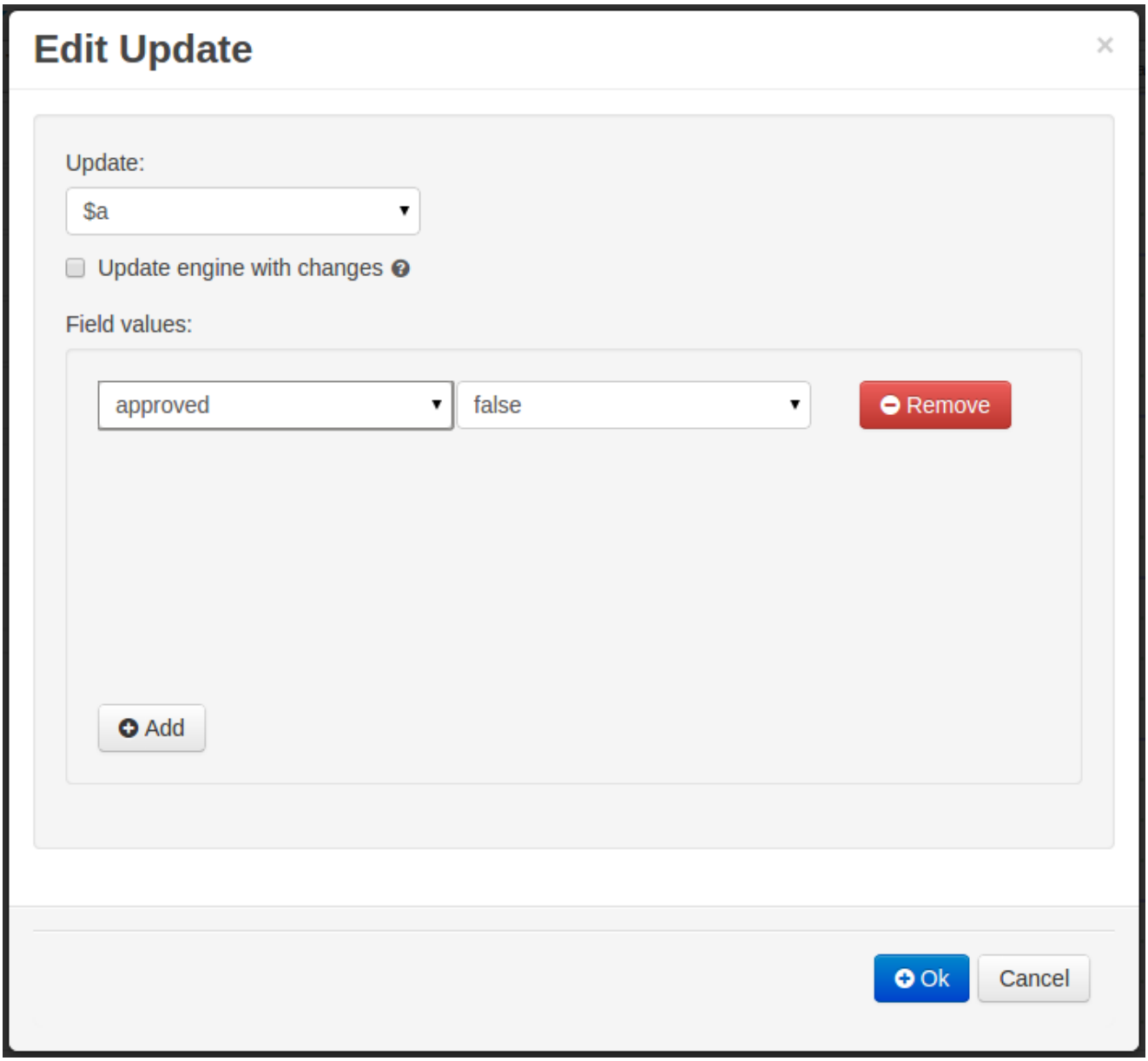


Figure 19.57. Guided Decision Trees - Action "Update" Editor

19.5.6. Managing the tree

Even simple trees can grow in size and become difficult to maintain.

It is therefore possible to collapse parts of the tree, giving more space in the user interface to maintain different parts of the tree.

If a node has children, when selected, it will have an icon to collapse the children. Clicking this icon will collapse children.

A collapsed node can equally be expanded by selecting it and clicking on the exapnd icon. A collapsed node cannot be edited as it contains numerous children. Deleting a collapsed node deletes all children too.

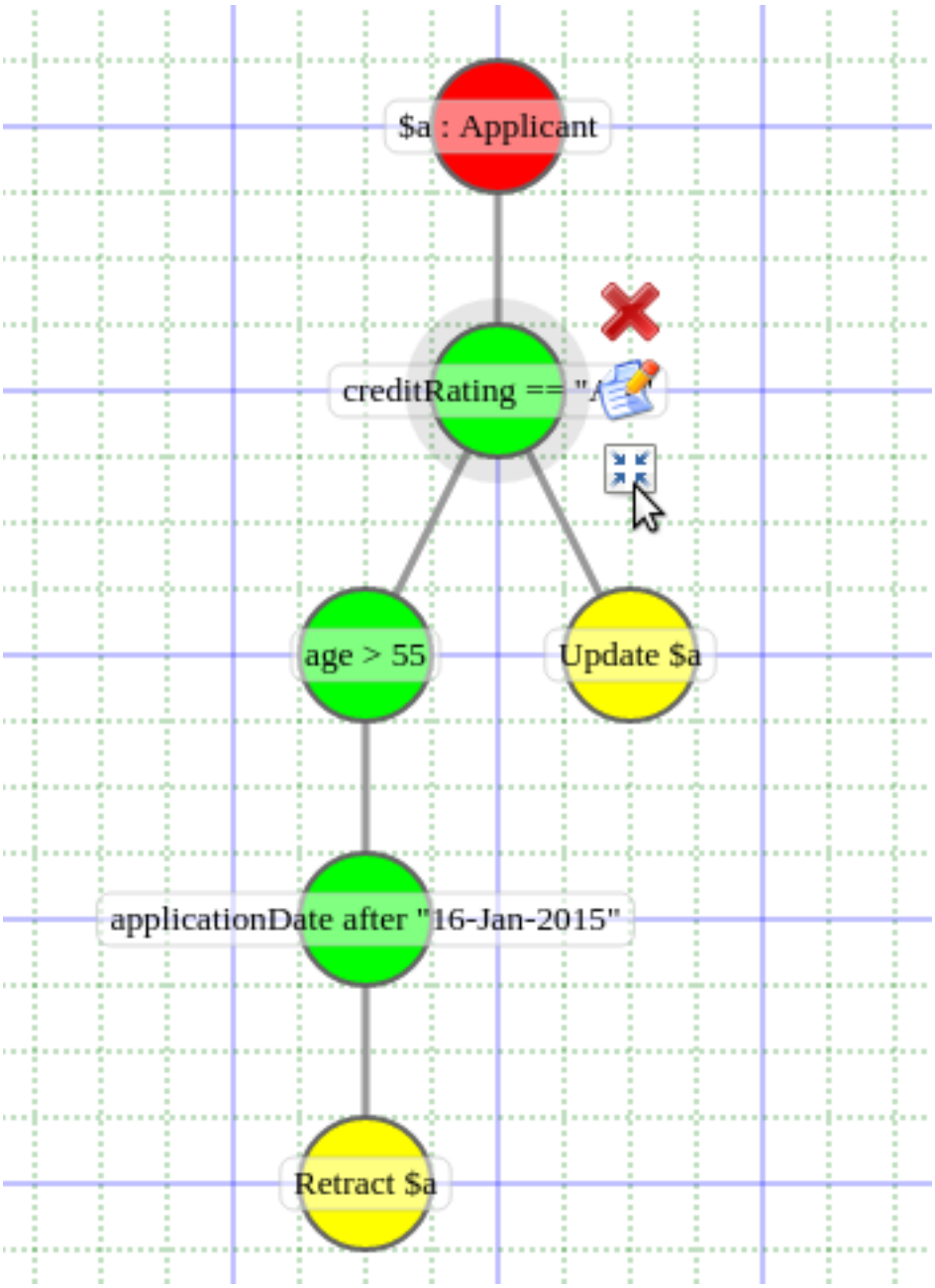


Figure 19.58. Guided Decision Trees - Collapsing nodes

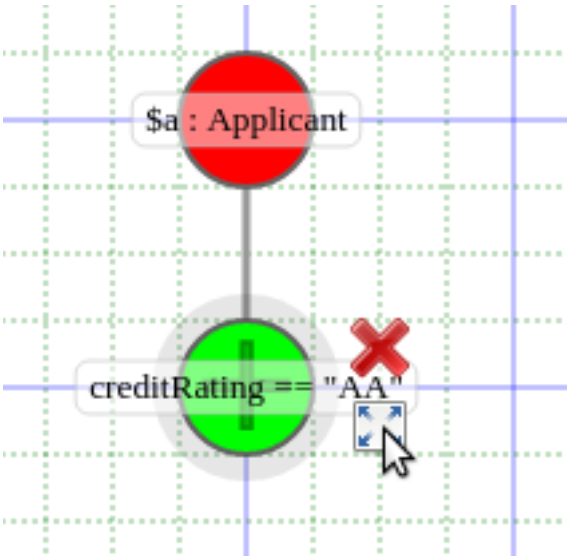


Figure 19.59. Guided Decision Trees - Collapsed node

19.6. Spreadsheet decision tables

Multiple rules can be stored in a spreadsheet. Each row in the spreadsheet is a rule, and each column is either a condition, an action, or an option. The Drools Expert section of this document discusses spreadsheet decision tables in more detail.

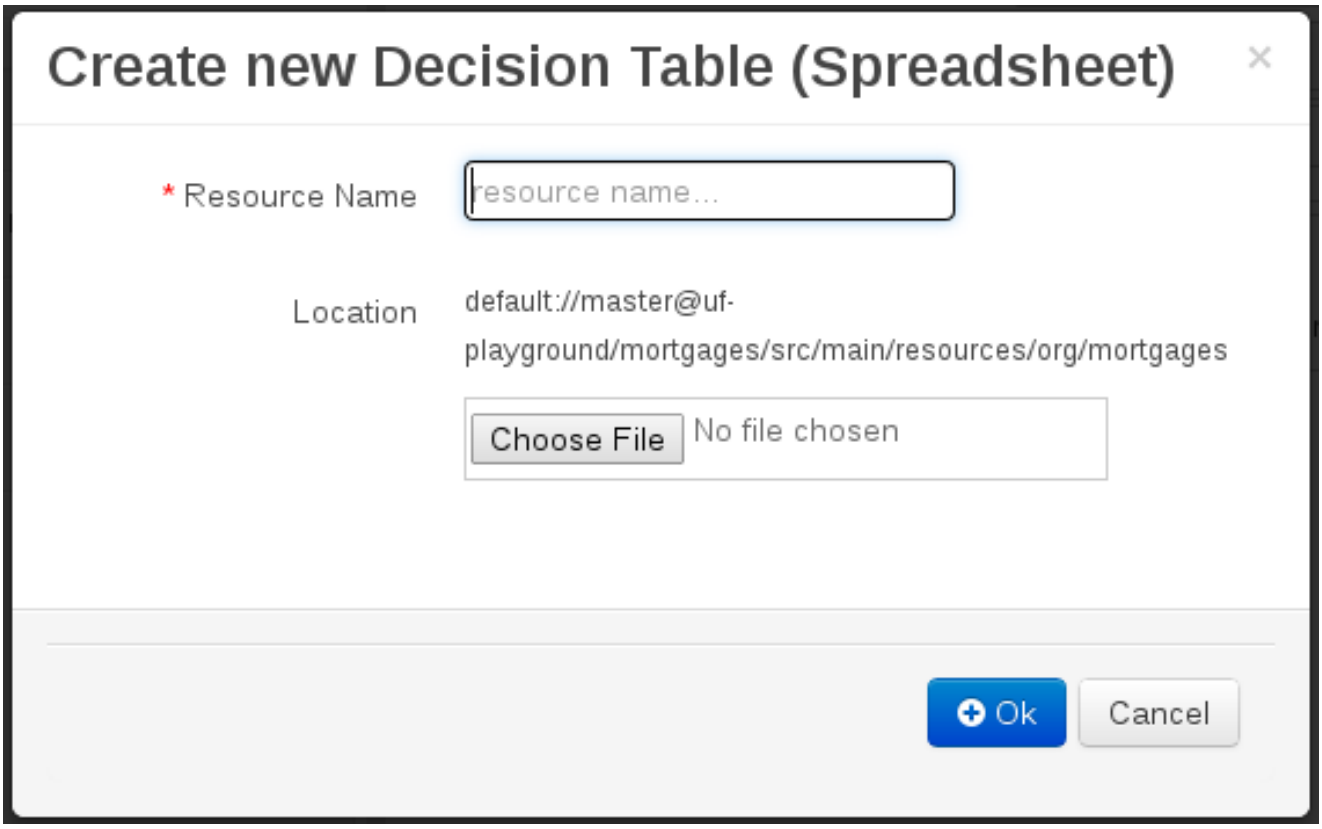


Figure 19.60. Spreadsheet decision table

To use a spreadsheet, you upload an XLS file. To create a new decision table: launch the new "Decision Table (Spreadsheet)" wizard, you will get an option to upload one.

19.7. Scorecards

A scorecard is a graphical representation of a formula used to calculate an overall score. A scorecard can be used to predict the likelihood or probability of a certain outcome. Drools now supports additive scorecards. An additive scorecard calculates an overall score by adding all partial scores assigned to individual rule conditions.

Additionally, Drools Scorecards will allows for reason codes to be set, which help in identifying the specific rules (buckets) that have contributed to the overall score. Drools Scorecards will be based on the PMML 4.1 Standard.

The New Item menu now allows for creation of scorecard assets.

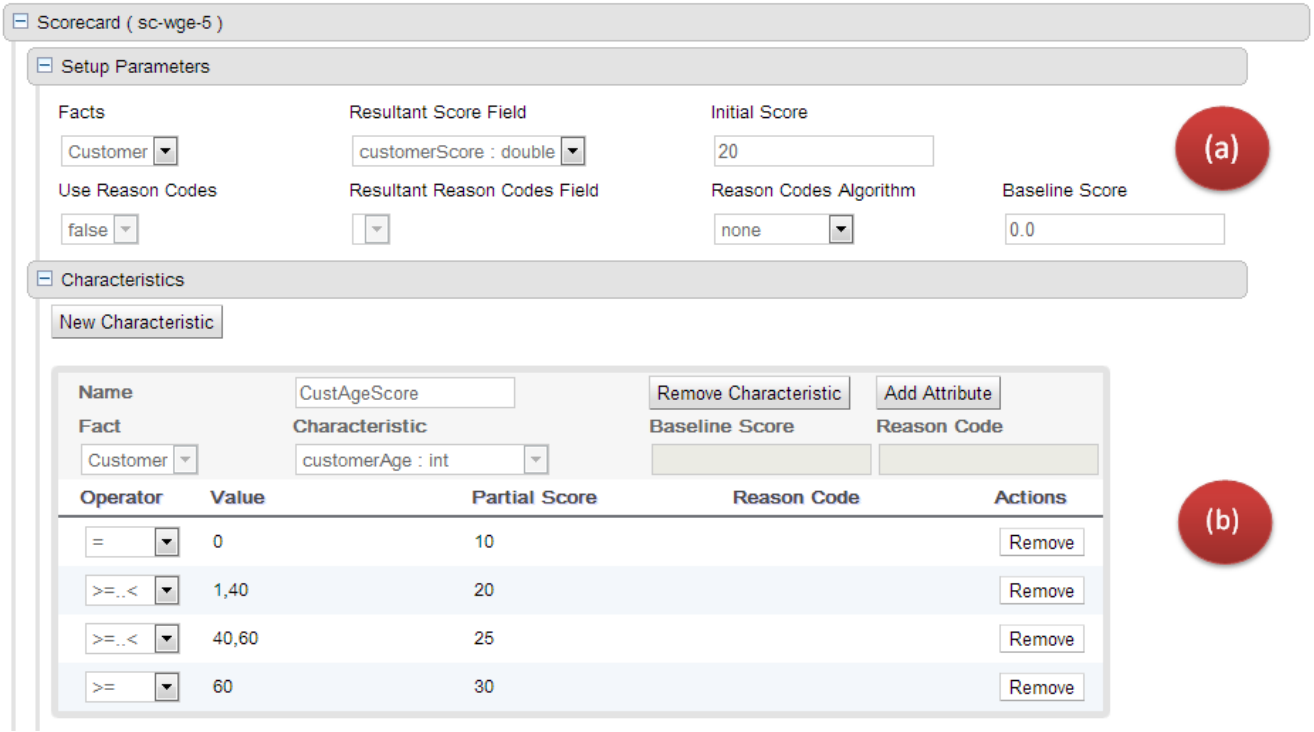


Figure 19.61. Scorecard Asset - Guided Editor

The above image shows a scorecard with one characteristic. Each scorecard consists of two sections (a) Setup Parameters (b) Characteristic Section

19.7.1. (a) Setup Parameters

The setup section consits of parameters that define the overall behaviour of this scorecard.

1. Facts: This dropdown shows a list of facts that are visible for this asset.
2. Resultant Score Field: Shows a list of fields from the selected fact. Only fields of type 'double' are shown. If this dropdown is empty double check your fact model. The final calculated score will be stored in this field.
3. Initial Score: Numeric Text Field to capture the initial score. The generated rules will initialize the 'Resultant Score Field' with this score and then is added to the overall score whenever partial scores are summed up.
4. Use Reason Codes: Boolean indicator to compute reason codes along with the final score. Selecting Yes/No in this field will enable/disable the 'Resultant Reason Codes Field', 'Reason Code Algorithm' and the 'Baseline Score' field.
5. Resultant Reason Codes Field: Shows a list of fields from the selected fact. Only fields of type 'java.util.List' are shown. This collection will hold the reason codes selected by this scorecard.
6. Reason Code Algorithm: May be "none", "pointsAbove" or "pointsBelow", describing how reason codes shall be ranked, relative to the baseline score of each Characteristic, or as set at the top-level scorecard.

7. Baseline Score: A single value to use as the baseline comparison score for all characteristics, when determining reason code ranking. Alternatively, unique baseline scores may be set for each individual Characteristic as shown below. This value is required only when UseReasonCodes is "true" and baselineScore is not given for each Characteristic.

Note

If UseReasonCodes is "true", then BaselineScore must be defined at the Scorecard level or for each Characteristic, and ReasonCode must be provided for each Characteristic or for each of its input Attributes. If UseReasonCodes is "false", then baselineScore and reasonCode are not required.

19.7.2. (b) Characteristics

On Clicking the 'New Characteristic' button, a new empty characteristic editor is added to the scorecard. Defines the point allocation strategy for each scorecard characteristic (numeric or categorical). Each scorecard characteristic is assigned a single partial score which is used to compute the overall score. The overall score is simply the sum of all partial scores. Partial scores are assumed to be continuous values of type "double".

19.7.2.1. Creating Characterstics

Every scorecard must have at least one characteristic

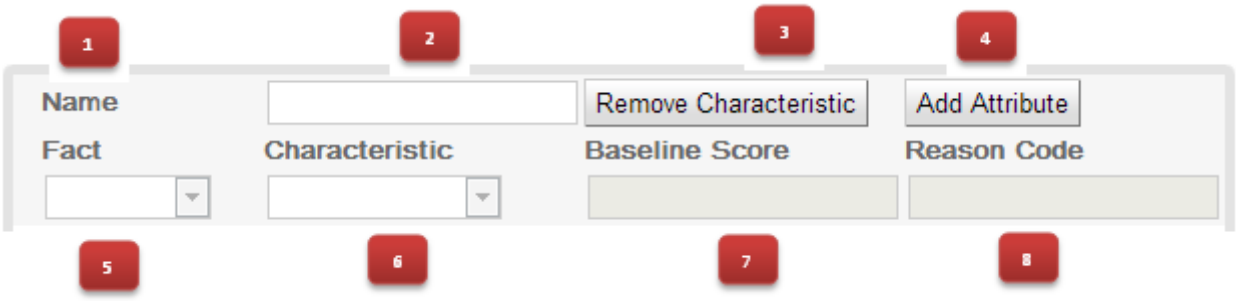


Figure 19.62. New Characteristic

1. Name: Descriptive name for this characteristic. For informational reasons only.
2. Remove Charteristic: Will remove this characteristic from the scorecard after a confirmation dialog is shown.
3. Add Attribute: Will add a line entry for an attribute (bin).
4. Fact: Select the class which will be evaluated for calculating the partial score.
5. Characteristic: Shows the list of fields from the selected Fact. Only fields of type "String", "int", "double", "boolean" are shown.

6. Baseline Score: Sets the characteristic's baseline score against which to compare the actual partial score when determining the ranking of reason codes. This value is required when useReasonCodes attribute is "true" and baselineScore is not defined in element Scorecard. Whenever baselineScore is defined for a Characteristic, it takes precedence over the baselineScore value defined in element Scorecard.
7. Reason Code: Contains the characteristic's reason code, usually associated with an adverse decision.

19.7.2.2. Creating Attributes

On Clicking the 'New Attribute' button, a new empty attribute editor. In scorecard models, all the elements defining the Attributes for a particular Characteristic must all reference a single field.

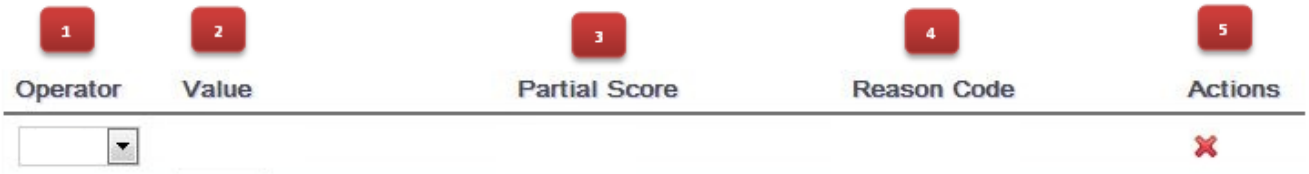


Figure 19.63. New Attribute

1. Operator: The condition upon which the mapping between input attribute and partial score takes place. The operator dropdown will show different values depending on the datatype of the selected Field.
   a. DataType Strings: "=", "in".
   b. DataType Integers: "=", ">", "<", ">=", "<=", ">..<", ">=..<", ">=..<=", ">..<=".
   c. DataType Boolean: "true", "false".
   Refer to the next sub-section (values) for more details.
2. Value: Basis the operator selected the value specified can either be a single value or a set of values separated by comma (","). The value field is disabled for operator type boolean.

Table 19.1. Operators / Values

Data Type	Operator	Value	Remarks
String	=	Single Value	will look for an exact match
String	in	Comma Separated Values (a,b,c,...)	The operator 'in' indicates an evaluation to TRUE if the field value is contained in the comma separated list of values
Boolean	is true	N/A	Value Field is uneditable (readonly)

Data Type	Operator	Value	Remarks
Boolean	is false	N/A	Value Field is uneditable (readonly)
Numeric	=	Single Value	Equals Operator
Numeric	>	Single Value	Greator Than Operator
Numeric	<	Single Value	Less Than Operator
Numeric	>=	Single Value	Greater than or equal To
Numeric	<=	Single Value	Less than or equal To
Numeric	>..<	Comma Separated Values (a,b)	(Greater than Value 'a') and (less than value 'b')
Numeric	>=..<	Comma Separated Values (a,b)	(Greater than or equal to Value 'a') and (less than value 'b')
Numeric	>=..<=	Comma Separated Values (a,b)	(Greater than or equal to Value 'a') and (less than or equal to value 'b')
Numeric	>..<=	Comma Separated Values (a,b)	(Greater than Value 'a') and (less than or equal to value 'b')

3. Partial Score: Defines the score points awarded to the Attribute.
4. Reason Code: Defines the attribute's reason code. If the reasonCode attribute is used in this level, it takes precedence over the ReasonCode associated with the Characteristic element.
5. Actions: Delete this attribute. Prompts the user for confirmation.

Note

If Use Reason Codes is "true", then Baseline Score must be defined at the Scorecard level or for each Characteristic, and Reason Code must be provided for each Characteristic or for each of its input Attributes. If Use Reason Codes is "false", then BaselineScore and ReasonCode are not required.

19.8. Test Scenario

Test Scenarios are used to validate that rules and knowledge base work as expected. When the knowledge base evolves, Test Scenarios guard against regression.

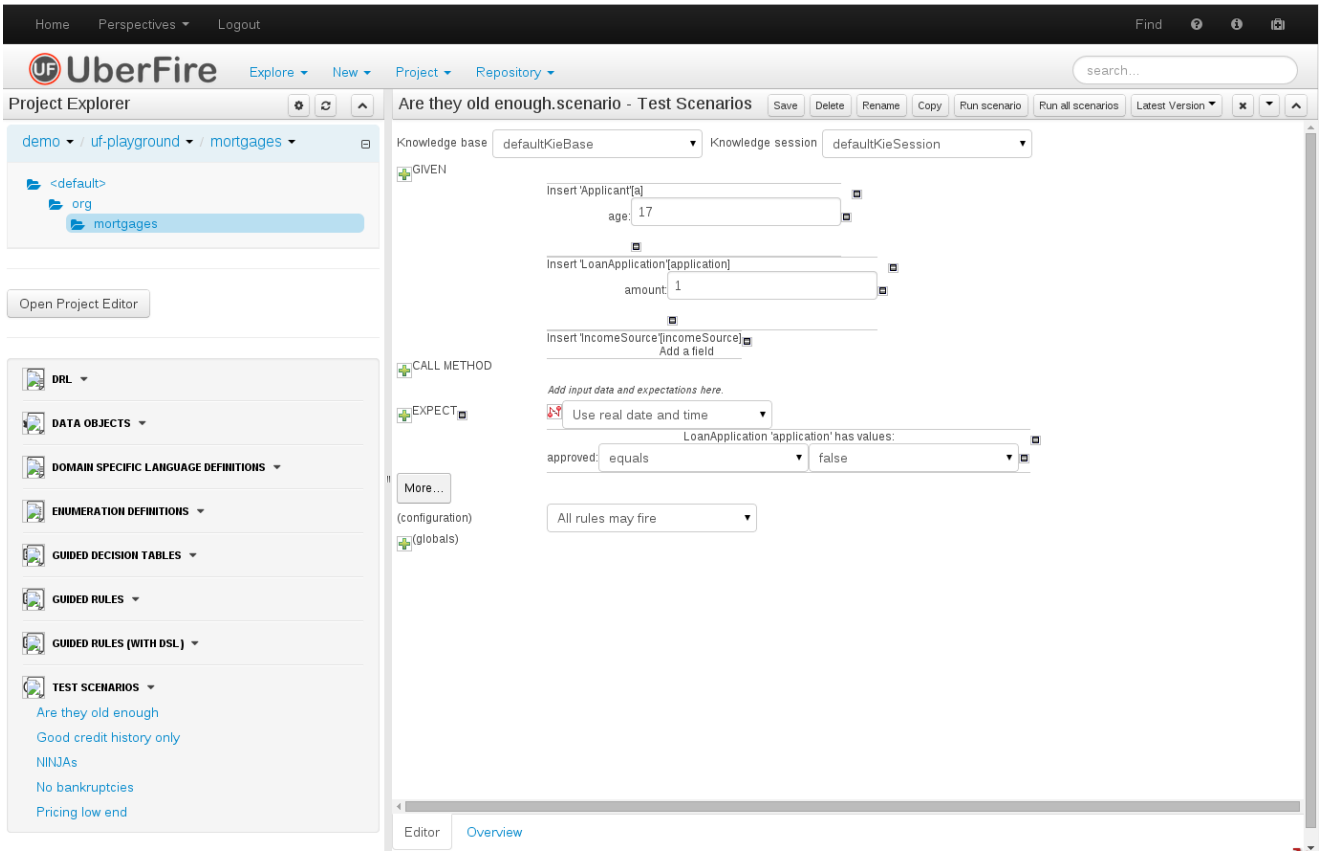


Figure 19.64. Example Test Scenario

Given section lists the facts needed for the behaviour. Expect section lists the expected changes and actions done by the behaviour. Given facts are passed for the Test Scenario before execution. During the rule execution, changes in the knowledge base are recorded. After the execution ends the recorded actions, existing facts in the knowledge base and knowledge base output is compared against the expectations.

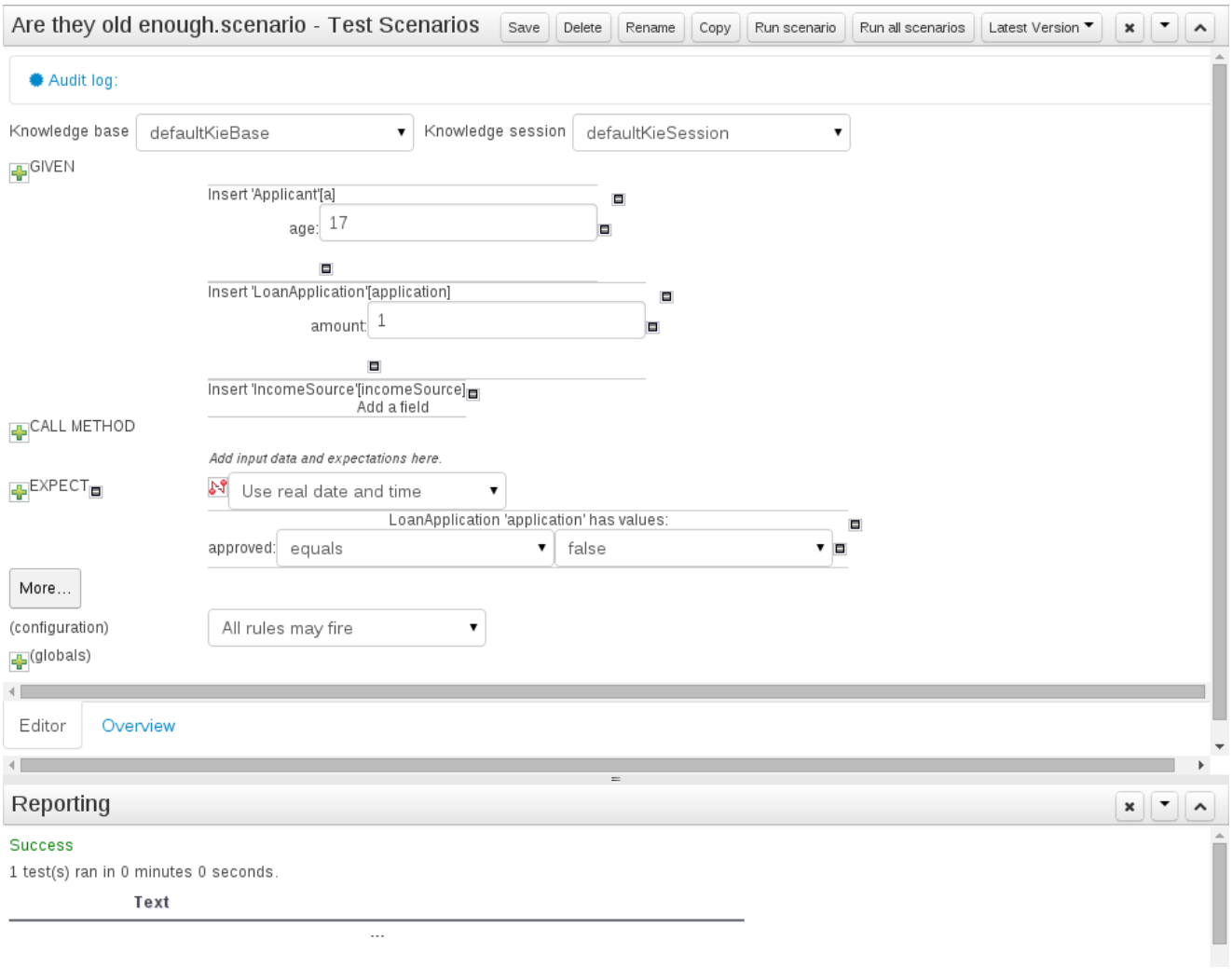


Figure 19.65. Example Test Scenario after execution

19.8.1. Knowledge Session Selector

Figure 19.66. Knowledge Session Selector

Knowledge session selector is used to select the session that the rules are ran against. The knowledge session needs to be stateful and use the pseudo clock when simulated time is used. Knowledge sessions are defined in the Project Editor, in the Knowledge bases and Sessions section.

19.8.2. Given Section

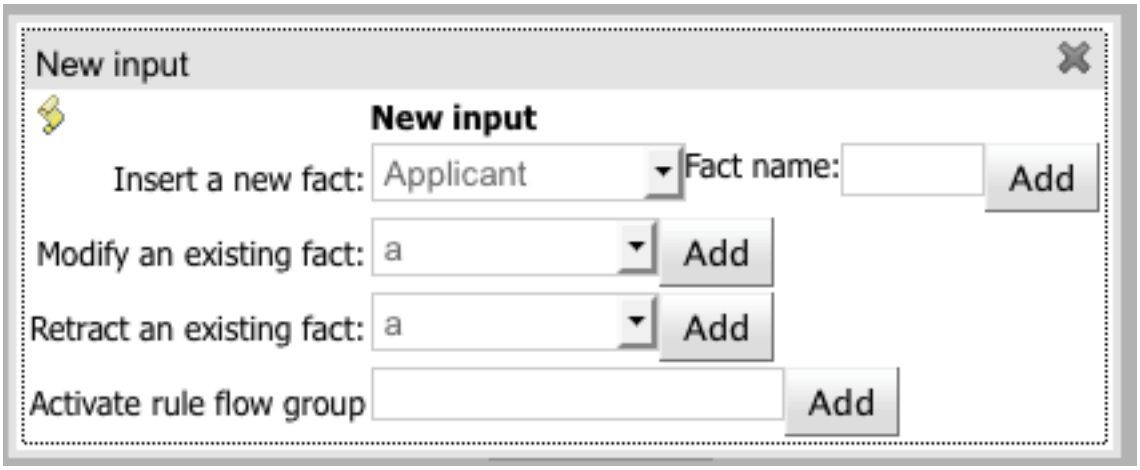


Figure 19.67. Given popup

- Insert a new fact - Adds a new fact that will be inserted into the knowledge base before execution.
- Modify an existing fact - Allows editing a fact between knowledge base executions.
- Delete an existing fact - Allows removing facts between executions.
- Activate rule flow group - Allows rules from a rule flow group to be tested, by activating the group in advance.

19.8.3. Expect Section

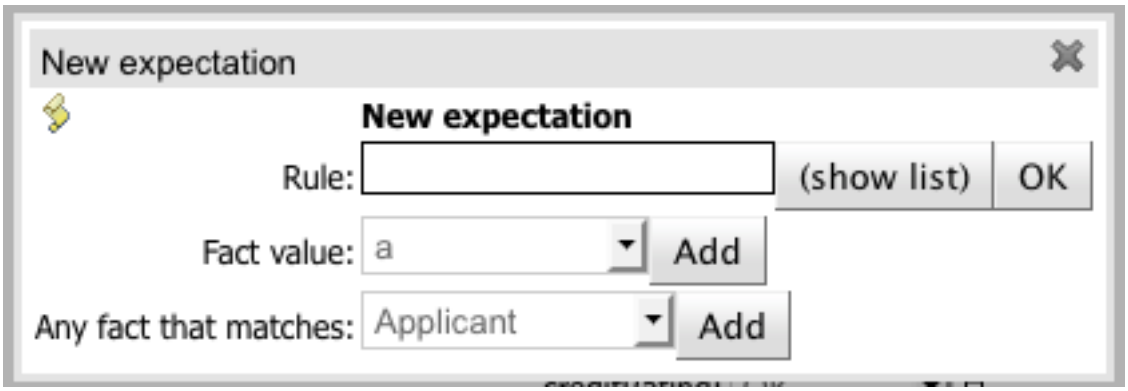


Figure 19.68. Expect popup

- Rule - Validate that a certain rule fired.
- Fact value - Validate fact values for a fact created in the Given section.
- Any fact that matches - Validate that there is at least one fact in the knowledge base with the specified field values.

19.8.4. Global Section

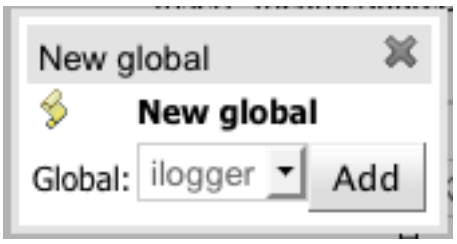


Figure 19.69. Global popup

- Global - Validate that the global field values.

19.8.5. New Input Section

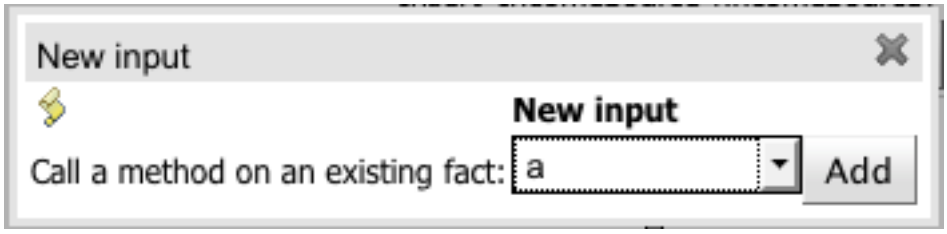


Figure 19.70. New Input popup

- Call method on an existing fact - Call a method from a fact in the beginning of the rule execution.

19.9. Functions

Functions are another asset type. They are NOT rules, and should only be used when necessary. The function editor is a textual editor. Functions

```
function <returnType> funcName(<args here>) {
  //code goes in here...

}
```

Figure 19.71. Function

19.10. DSL editor

The DSL editor allows DSL Sentences to be authored. The reader should take time to explore DSL features in the Drools Expert documentation; as the syntax in Drools Workbench's DSL Editor is identical. The normal syntax is extended to provide "hints" to control how the DSL variable is rendered and validated within the user-interface.

The following "hints" are supported:-

- {<varName>:<regular expression>}

  This will render a text field in place of the DSL variable when the DSL Sentence is used in the guided editor. The content of the text field will be validated against the regular expression.

- {<varName>:ENUM:<factType.fieldName>}

  This will render an enumeration in place of the DSL variable when the DSL Sentence is used in the guided editor. <factType.fieldName> binds the enumeration to the model Fact and Field enumeration definition. This could be either a "Drools Workbench enumeration" (i.e. defined within the Workbench) or a Java enumeration (i.e. defined in a model POJO JAR file).

- {<varName>:DATE:<dateFormat>}

  This will render a Date selector in place of the DSL variable when the DSL Sentence is used in the guided editor.

- {<varName>:BOOLEAN:<[checked | unchecked]>}

  This will render a dropdown selector in place of the DSL variable, providing boolean choices, when the DSL Sentence is used in the guided editor.

- {<varName>:CF:<factType.fieldName>}

  This will render a button that will allow you to set the value of this variable using a Custom Form. In order to use this feature, a Working-Set containing a Custom Form Configuration for factType.fieldName must be active. If there is no such Working-Set, a simple text box is used (just like a regular variable).

  For more information, please read more about Working-Sets and Custom Form Configurations.

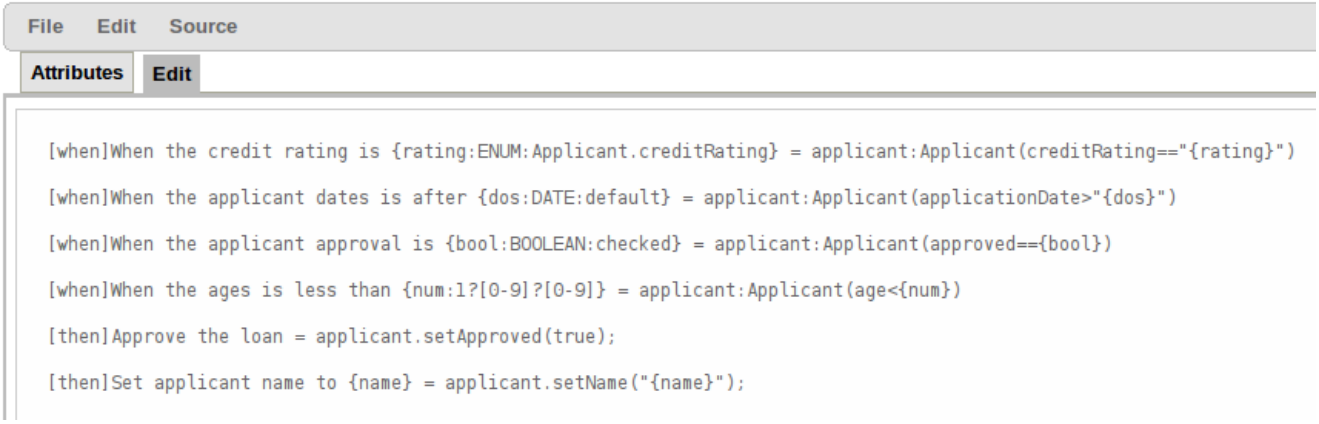


Figure 19.72. DSL rule

19.11. Data enumerations (drop down list configurations)

Data enumerations are an optional asset type that technical folk can configure to provide drop down lists for the guided editor. These are stored and edited just like any other asset, and apply to the package that they belong to.

The contents of an enum config are a mapping of `Fact.field` to a list of values to be used in a drop down. The strings are either a value to be shown in a drop down, or a mapping from the code value (what ends up used in the rule) and a display value.

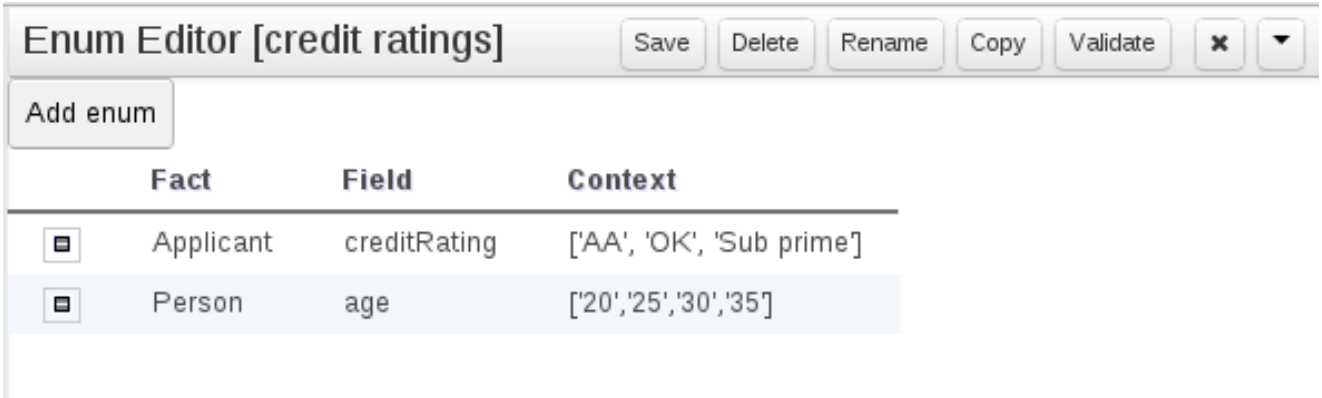


Figure 19.73. Data enumeration

If you wish to use a mapping between value used in the rule and the value shown in the UI you need to separate the code value and display value with an equals sign. For example:

```
'Person.gender' : ['M=Male','F=Female']
```

In this example `F` will be used in the rules but `Female` shown in the UI.

Drop down lists can also depend on other field values.

Lets imagine a simple fact model, we have a class called Vehicle, which has 2 fields: `engineType` and `fuelType`. We want to have a choice for the `engineType` of "Petrol" or "Diesel". Now, obviously the choice type for fuel must be dependent on the engine type (so for Petrol we have ULP and PULP, and for Diesel we have BIO and NORMAL). We can express this dependency in an enumeration as:

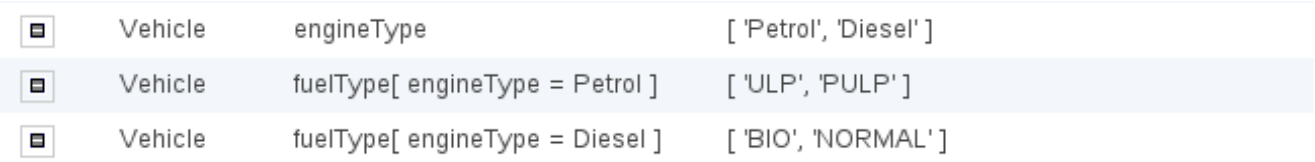


Figure 19.74.

This shows how it is possible to make the choices dependent on other field values. Note that once you pick the engineType, the choice list for the fuelType will be determined.

19.11.1. Advanced enumeration concepts

There are a few other advanced things you can do with data enumerations.

19.11.1.1. External data-sources

Instead of defining a static list you can retrieve the list from an external data-source by using a helper class.

The helper class must be on the Project's classpath; by adding a JAR containng the class as a project dependency. The helper class must be instantiable and have a non-static method that returns a `java.util.List`. Population of the list is implementation specific.

For example:

```
'Person.age' : (new org.yourco.DataHelper()).getListOfAges()
```

In some other cases you may want to load the enumeration data entirely from an external data source (such as a relational database). To do this, you can implement a class that returns a `Map<String, List<String>>`. The key of the map is the `Fact.field` and the value is a `java.util.List<String>` of values to be used.

```
public class SampleDataSource2 {

  public Map<String>, List<String> loadData() {
    Map data = new HashMap();
```

```
    List d = new ArrayList();
    d.add("value1");
    d.add("value2");
    data.put("Fact.field", d);

    return data;
 }

}
```

The enumeration definition would simply be as follows (with no reference to fact or field names):

```
=(new SampleDataSource2()).loadData()
```

The = operator informs the workbench to load all enumeration data from the helper class.

Note

The helper methods will be statically evaluated when the enumeration definition is requested for use in an editor.

19.11.1.2. Dynamic lookup

All of the above cases retrieve the list of values by statically evaluating the enumeration definition when enumerations are required by an editor.

It is also possibly to load dependent enumeration definitions dynamically from a helper class by enclosing the call to the helper class within quotation marks. For example:

```
'Country.region[countryCode]'                                :                                '(new
 org.yourco.DataHelper()).getListOfRegions("@{countryCode}")'
```

19.12. Technical rules (DRL)

Technical (DRL) rules are stored as text - they can be managed in Drools Workbench. A DRL can either be a whole chunk of rules, or an individual rule. if its an individual rule, no package statement or imports are required (in fact, you can skip the "rule" statement altogether, just use "when" and "then" to mark the condition and action sections respectively). Normally you would use the IDE to edit raw DRL files, since it has all the advanced tooling and content assistance and debugging. However, there are times when a rule may have to deal with something fairly technical in a package in Drools Workbench. In any typical package of rules, you generally have a need for some "technical rules" - you can mix and match all the rule types together of course.

```
package org.mortgages

rule 'Dummy rule'

salience 100 // This can short circuit any processing

when
  a : Approve()
  p : Policy()
then
  p.setApproved( true );
  System.out.println( "APPROVED: " + a.getReason());
end
```

Figure 19.75. DRL technical rule

Chapter 20. Workbench Integration

20.1. REST

REST API calls to Knowledge Store allow you to manage the Knowledge Store content and manipulate the static data in the repositories of the Knowledge Store. The calls are asynchronous, that is, they continue their execution after the call was performed as a job. The job ID is returned by every calls to allow after the REST API call was performed to request the job status and verify whether the job finished successfully. Parameters of these calls are provided in the form of JSON entities.

When using Java code to interface with the REST API, the classes used in POST operations or otherwise returned by various operations can be found in the `(org.kie.workbench.services:)kie-wb-common-services` JAR. All of the classes mentioned below can be found in the `org.kie.workbench.common.services.shared.rest` package in that JAR.

20.1.1. Job calls

Every Knowledge Store REST call returns its job ID after it was sent. This is necessary as the calls are asynchronous and you need to be able to reference the job to check its status as it goes through its lifecycle. During its lifecycle, a job can have the following statuses:

- `ACCEPTED`: the job was accepted and is being processed
- `BAD_REQUEST`: the request was not accepted as it contained incorrect content
- `RESOURCE_NOT_EXIST`: the requested resource (path) does not exist
- `DUPLICATE_RESOURCE`: the resource already exists
- `SERVER_ERROR`: an error on the server occurred
- `SUCCESS`: the job finished successfully
- `FAIL`: the job failed
- `DENIED`: the job was denied
- `GONE`: the job ID could not be found

  A job can be GONE in the following cases:

  - The job was explicitly removed
  - The job finished and has been deleted from the status cache (the job is removed from status cache after the cache has reached its maximum capacity)

- The job never existed

The following `job` calls are provided:

[GET] **/jobs/*{jobID}***
Returns the job status

Returns a `JobResult` instance

Example 20.1. An example (formatted) response body to the get job call on a repository clone request

```
"{
  "status":"SUCCESS",
  "jodId":"1377770574783-27",
       "result":"Alias:    testInstallAndDeployProject,    Scheme:    git,    Uri:    git://
testInstallAndDeployProject",
  "lastModified":1377770578194,"detailedResult":null
}"
```

[DELETE] **/jobs/*{jobID}***
Removes the job: If the job is not yet being processed, this will remove the job from the job queue. However, this will not cancel or stop an ongoing job

Returns a `JobResult` instance

20.1.2. Repository calls

Repository calls are calls to the Knowledge Store that allow you to manage its Git repositories and their projects.

The following `repositories` calls are provided:

[GET] **/repositories**
Gets information about the repositories in the Knowledge Store

Returns a `Collection<Map<String, String>>` or `Collection<RepositoryRequest>` instance, depending on the JSON serialization library being used. The keys used in the `Map<String, String>` instance match the fields in the `RepositoryRequest` class

Example 20.2. An example (formatted) response body to the get repositories call

```
[
  {
    "name":"wb-assets",
```

```
    "description":"generic assets",
    "userName":null,
    "password":null,
    "requestType":null,
    "gitURL":"git://bpms-assets"
 },
 {
   "name":"loanProject",
   "description":"Loan processes and rules",
   "userName":null,
   "password":null,
   "requestType":null,
   "gitURL":"git://loansProject"
 }
]
```

[POST] **/repositories**

Creates a new empty repository or a new repository cloned from an existing (git) repository

Consumes a `RepositoryRequest` instance

Returns a `CreateOrCloneRepositoryRequest` instance

Example 20.3. An example (formatted) response body to the create repositories call

```
{
  "name":"new-project-repo",
  "description":"repo for my new project",
  "userName":null,"password":null,
  "requestType":"new",
  "gitURL":null
}
```

[DELETE] **/repositories/*{repositoryName}***

Removes the repository from the Knowledge Store

Returns a `RemoveRepositoryRequest` instance

[POST] **/repositories/*{repositoryName}*/projects/**

Creates a project in the repository

Consumes an `Entity` instance

Returns a `CreateProjectRequest` instance

Example 20.4. An example (formatted) request body that defines the project to be created

```
{
```

```
  "name":"myProject",
  "description": "my project"
}
```

20.1.3. Organizational unit calls

Organizational unit calls are calls to the Knowledge Store that allow you to manage its organizational units, so as to organize the connected Git repositories.

The following `organizationalUnits` calls are provided:

[POST] **/organizationalunits**
: Creates an organizational unit in the Knowledge Store

  Consumes an `OrganizationalUnit` instance

  Returns a `CreateOrganizationalUnitRequest` instance

Example 20.5. An example (formatted) request body defining a new organizational unit to be created

```
{
  "name":"testgroup",
  "description":"",
  "owner":"tester",
  "repositories":["testGroupRepository"]
}
```

[POST] **/organizationalunits/*{organizationalUnitName}*/repositories/*{repositoryName}***
: Adds the repository to the organizational unit

  Returns a `AddRepositoryToOrganizationalUnitRequest` instance

[DELETE] **/organizationalunits/*{organizationalUnitName}*/repositories/*{repositoryName}***
: Removes the repository from the organizational unit

  Returns a `RemoveRepositoryFromOrganizationalUnitRequest` instance

20.1.4. Maven calls

Maven calls are calls to a Project in the Knowledge Store that allow you compile and deploy the Project resources.

The following `maven` calls are provided:

[POST] **/repositories/*{repositoryName}*/projects/*{projectName}*/maven/compile**
: Compiles the project (equivalent to `mvn compile`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `CompileProjectRequest` instance

[POST] **/repositories/*{repositoryName}*/projects/*{projectName}*/maven/install**
Installs the project (equivalent to `mvn install`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `InstallProjectRequest` instance

[POST] **/repositories/*{repositoryName}*/projects/*{projectName}*/maven/test**
Compiles the project runs a test as part of compilation

Consumes a `BuildConfig` instance

Returns a `TestProjectRequest` instance

[POST] **/repositories/*{repositoryName}*/projects/*{projectName}*/maven/deploy**
Deploys the project (equivalent to `mvn deploy`)

Consumes a `BuildConfig` instance. While this must be supplied, it's not needed for the operation and may be left blank.

Returns a `DeployProjectRequest` instance

20.1.5. REST summary

The URL templates in the table below are relative the following URL:

- `http://server:port/business-central/rest`

Table 20.1. Knowledge Store REST calls

URL Template	Type	Description
/jobs/{jobID}	GET	return the job status
/jobs/{jobID}	DELETE	remove the job
/organizationalunits	GET	return a list of organizational units
/organizationalunits	POST	create an organizational unit in the Knowledge Store described by the JSON `OrganizationalUnit` entity
/organizationalunits/{organizationalUnitName}/repositories/{repositoryName}	POST	add a repository to an organizational unit

URL Template	Type	Description
/organizationalunits/{organizationalUnitName}/repositories/{repositoryName}	DELETE	Remove a repository from an organizational unit
/repositories/	POST	add the repository to the organizational unit described by the JSON `RepositoryReqest` entity
/repositories	GET	return the repositories in the Knowledge Store
/repositories/{repositoryName}	DELETE	Remove the repository from the Knowledge Store
/repositories/	POST	create or clone the repository defined by the JSON `RepositoryRequest` entity
/repositories/{repositoryName}/projects/	POST	create the project defined by the JSON entity in the repository
/repositories/{repositoryName}/projects/{projectName}/maven/compile/	POST	compile the project
/repositories/{repositoryName}/projects/{projectName}/maven/install	POST	install the project
/repositories/{repositoryName}/projects/{projectName}/maven/test/	POST	compile the project and run tests as part of compilation
/repositories/{repositoryName}/projects/{projectName}/maven/deploy/	POST	deploy the project

Chapter 21. Workbench High Availability

21.1.1. VFS clustering

The VFS repositories (usually git repositories) stores all the assets (such as rules, decision tables, process definitions, forms, etc). If that VFS resides on each local server, then it must be kept in sync between all servers of a cluster.

Use Apache Zookeeper [http://zookeeper.apache.org/] and Apache Helix [http://helix.incubator.apache.org/] to accomplish this. Zookeeper glues all the parts together. Helix is the cluster management component that registers all cluster details (nodes, resources and the cluster itself). Uberfire (on top of which Workbench is build) uses those 2 components to provide VFS clustering.

To create a VFS cluster:

1. Download Apache Zookeeper [http://zookeeper.apache.org/] and Apache Helix [http://helix.incubator.apache.org/].
2. Install both:
   a. Unzip Zookeeper into a directory (`$ZOOKEEPER_HOME`).
   b. In `$ZOOKEEPER_HOME`, copy `zoo_sample.conf` to `zoo.conf`
   c. Edit `zoo.conf`. Adjust the settings if needed. Usually only these 2 properties are relevant:

```
# the directory where the snapshot is stored.dataDir=/tmp/zookeeper# the port at which the
 clients will connectclientPort=2181
 is
stored.dataDir=/tmp/zookeeper# the port at which the clients
```

   d. Unzip Helix into a directory (`$HELIX_HOME`).
3. Configure the cluster in Zookeeper:
   a. Go to its `bin` directory:

```
$ cd $ZOOKEEPER_HOME/bin
```

   b. Start the Zookeeper server:

```
$ sudo ./zkServer.sh start
```

If the server fails to start, verify that the `dataDir` (as specified in `zoo.conf`) is accessible.

c. To review Zookeeper's activities, open `zookeeper.out`:

```
$ cat $ZOOKEEPER_HOME/bin/zookeeper.out
```

4. Configure the cluster in Helix:

a. Go to its `bin` directory:

```
$ cd $HELIX_HOME/bin
```

b. Create the cluster:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --addCluster kie-cluster
```

The `zkSvr` value must match the used Zookeeper server. The cluster name (`kie-cluster`) can be changed as needed.

c. Add nodes to the cluster:

```
# Node 1
$ ./helix-admin.sh --zkSvr localhost:2181 --addNode kie-cluster nodeOne:12345
# Node 2
$ ./helix-admin.sh --zkSvr localhost:2181 --addNode kie-cluster nodeTwo:12346
...
```

Usually the number of nodes a in cluster equal the number of application servers in the cluster. The node names (`nodeOne:12345` , ...) can be changed as needed.

Note

`nodeOne:12345` is the unique identifier of the node, which will be referenced later on when configuring application servers. It is not a host and port number, but instead it is used to uniquely identify the logical node.

d. Add resources to the cluster:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --addResource kie-cluster vfs-repo 1 LeaderS
tandby AUTO_REBALANCE
```

The resource name (`vfs-repo`) can be changed as needed.

e. Rebalance the cluster to initialize it:

```
$ ./helix-admin.sh --zkSvr localhost:2181 --rebalance kie-cluster vfs-repo 2
```

f. Start the Helix controller to manage the cluster:

```
$   ./run-helix-controller.sh --zkSvr localhost:2181 --cluster kie-cluster 2>&1 > /tmp/
controller.log &
```

5. Configure the security domain correctly on the application server. For example on WildFly and JBoss EAP:

a. Edit the file `$JBOSS_HOME/domain/configuration/domain.xml`.

For simplicity sake, presume we use the default domain configuration which uses the profile `full` that defines two server nodes as part of `main-server-group`.

b. Locate the profile `full` and add a new security domain by copying the other security domain already defined there by default:

```
<security-domain name="kie-ide" cache-type="default">    <authentication>         <login-
module code="Remoting" flag="optional">           <module-option name="password-stacking"
 value="useFirstPass"/>         </login-module>         <login-module code="RealmDirect"
 flag="required">           <module-option name="password-stacking" value="useFirstPass"/
>         </login-module>    </authentication></security-domain>
ide" cache-type="default">
   <authentication>         <login-module
 code="Remoting" flag="optional">            <module-option name="password-
stacking" value="useFirstPass"/>
 </login-module>         <login-module
 code="RealmDirect" flag="required">            <module-option name="password-
stacking" value="useFirstPass"/>
 </login-module>
  </
```

Important

The security-domain name is a magic value.

6. Configure the system properties for the cluster on the application server. For example on WildFly and JBoss EAP:

   a. Edit the file `$JBOSS_HOME/domain/configuration/host.xml`.

   b. Locate the XML elements `server` that belong to the `main-server-group` and add the necessary system property.

      For example for nodeOne:

```
<system-properties>
  <property name="jboss.node.name" value="nodeOne" boot-time="false"/>
  <property name="org.uberfire.nio.git.dir" value="/tmp/kie/nodeone" boot-time="false"/>
     <property   name="org.uberfire.metadata.index.dir"   value="/tmp/kie/nodeone"   boot-
time="false"/>
  <property name="org.uberfire.cluster.id" value="kie-cluster" boot-time="false"/>
  <property name="org.uberfire.cluster.zk" value="localhost:2181" boot-time="false"/>
  <property name="org.uberfire.cluster.local.id" value="nodeOne_12345" boot-time="false"/>
  <property name="org.uberfire.cluster.vfs.lock" value="vfs-repo" boot-time="false"/>
  <!-- If you're running both nodes on the same machine: -->
  <property name="org.uberfire.nio.git.daemon.port" value="9418" boot-time="false"/>
</system-properties>
```

      And for nodeTwo:

```
<system-properties>
  <property name="jboss.node.name" value="nodeTwo" boot-time="false"/>
  <property name="org.uberfire.nio.git.dir" value="/tmp/kie/nodetwo" boot-time="false"/>
     <property   name="org.uberfire.metadata.index.dir"   value="/tmp/kie/nodetwo"   boot-
time="false"/>
  <property name="org.uberfire.cluster.id" value="kie-cluster" boot-time="false"/>
  <property name="org.uberfire.cluster.zk" value="localhost:2181" boot-time="false"/>
  <property name="org.uberfire.cluster.local.id" value="nodeTwo_12346" boot-time="false"/>
  <property name="org.uberfire.cluster.vfs.lock" value="vfs-repo" boot-time="false"/>
  <!-- If you're running both nodes on the same machine: -->
  <property name="org.uberfire.nio.git.daemon.port" value="9419" boot-time="false"/>
</system-properties>
```

      Make sure the cluster, node and resource names match those configured in Helix.

21.1.2. jBPM clustering

In addition to the information above, jBPM clustering requires additional configuration. See this blog post [http://mswiderski.blogspot.com.br/2013/06/clustering-in-jbpm-v6.html] to configure the database etc correctly.

Part VI. KIE Server

The KIE Server is a standalone execution server for rules.

Chapter 22. KIE Execution Server

22.1. Overview

The *Kie Server* is a modular, standalone server component that can be used to instantiate and execute rules and processes. It exposes this functionality via REST, JMS and Java interfaces to client application. It also provides seamless integration with the *Kie Workbench*.

At its core, the *Kie Server* is a configurable web application packaged as a WAR file. Distributions are availables for pure web containers (like Tomcat) and for JEE 6 and JEE 7 containers.

Most capabilities on the Kie Server are configurable, and based on the concepts of extensions. Each extension can be enabled/disabled independently, allowing the user to configure the server to its need.

The current version of the Kie Server ships with two default extensions:

- BRM: provides support for the execution of Business Rules using the Drools rules engine.
- BPM: provides support for the execution of Business Processes using the jBPM process engine. It supports:
  - process execution
  - task execution
  - assynchronous job execution

Both extensions enabled by default, but can be disabled by setting the corresponding property (see configuration chapter for details).

This server was designed to have a low footprint, with minimal memory consumption, and therefore, to be easily deployable on a cloud environment. Each instance of this server can open and instantiate multiple *Kie Containers* which allows you to execute multiple services in parallel.

22.1.1. Glossary

- **Kie Server**: execution server purely focusing on providing runtime environment for both rules and processes. These capabilities are provided by *Kie Server Extensions*. More capabilities can be added by further extensions (e.g. customer could add his own extensions in case of missing functionality that will then use infrastructure of the KIE Server). A Kie Server instance is a standalone Kie Server executing on a given application server/web container. A Kie Server instantiates and provides support for multiple Kie Containers.
- **Kie Server Extension**: a "plugin" for the Kie Server that adds capabilities to the server. The Kie Server ships with two default kie server extensions: BRM and BPM.

- **Kie Container**: an in-memory instantiation of a kjar, allowing for the instantiation and usage of its assets (domain models, processes, rules, etc). A Kie Server exposes Kie Containers through a standard API over transport protocols like REST and JMS.
- **Controller**: a server-backed REST endpoint that will be responsible for managing KIE Server instances. Such end point must provide following capabilities:
  - respond to connect requests
  - sync all registered containers on the corresponding *Kie Server ID*
  - respond to disconnect requests
- **Kie Server state**: currently known state of given Kie Server instance. This is a local storage (by default in file) that maintains the following information:
  - list of registered controllers
  - list of known containers
  - kie server configuration

  The server state is persisted upon receival of events like: *Kie Container* created, *Kie Container* is disposed, controller accepts registration of *Kie Server* instance, etc.
- **Kie Server ID**: an arbitrary assigned identifier to which configurations are assigned. At boot, each Kie Server Instance is assigned an ID, and that ID is matched to a configuration on the controller. The Kie Server Instance fetches and uses that configuration to setup itself.

22.2. Installing the KIE Server

The KIE Server is distributed as a web application archive (WAR) file. The WAR file comes in three different packagings:

- *webc* - WAR for ordinary Web (Servlet) containers like Tomcat
- *ee6* - WAR for JavaEE 6 containers like JBoss EAP 6.x
- *ee7* - WAR for JavaEE 7 containers like WildFly 8.x

To install the KIE Execution Server and verify it is running, complete the following steps:

1. Deploy the WAR file into your web container.
2. Create a user with the role of `kie-server` on the container.
3. Test that you can access the execution engine by navigating to the endpoint in a browser window: `http://SERVER:PORT/CONTEXT/services/rest/server/`.
4. When prompted for username/password, type in the username and password that you created in step 2.
5. Once authenticated, you will see an XML response in the form of engine status, similar to this:

Example 22.1. Sample handshaking server response

```
<response type="SUCCESS" msg="KIE Server info">
  <kie-server-info>
    <version>6.3.1-SNAPSHOT</version>
  </kie-server-info>
</response>
```

22.2.1. Bootstrap switches

The Kie Server accepts a number of bootstrap switches (system properties) to configure the behaviour of the server. The following is a table of all the supported switches.

Table 22.1. Kie Server bootstrap switches

Property	Value	Description	Required
org.drools.server.ext.disabled	boolean (default is "false")	If true, disables the BRM support (i.e. rules support).	No
org.jbpm.server.ext.disabled	boolean (default is "false")	If true, disables the BPM support (i.e. processes support)	No
org.kie.server.id	string	An arbitrary ID to be assigned to this server. If a remote controller is configured, this is the ID under which the server will connect to the controller to fetch the kie container configurations.	No. If not provided, an ID is automatically generated.
org.kie.server.user	string (default is "kieserver")	User name used to connect with the kieserver from the controller, required when running in managed mode	No
org.kie.server.pwd	string (default is "kieserver1!")	Password used to connect with the kieserver from the controller, required when running in managed mode	No
org.kie.server.controller	coma separated list of urls	List of urls to controller REST endpoint. E.g.: `http://localhost:8080/kie-wb/rest`	Yes when using a controller
org.kie.server.controller.user	string (default is "kieserver")	Username used to connect to the controller REST api	Yes when using a controller

Property	Value	Description	Required
org.kie.server.controller.pwd	string (default is "kieserver1!")	Password used to connect to the controller REST api	Yes when using a controller
org.kie.server.location	URL location of kie server instance	The URL used by the controller to call back on this server. E.g.: `http://localhost:8230/kie-server/services/rest/server`	Yes when using a controller
org.kie.server.domain	string	JAAS LoginContext domain that shall be used to authenticate users when using JMS	No
org.kie.server.bypass.auth.user	boolean (default is "false")	Allows to bypass the authenticated user for task related operations e.g. queries	No
org.kie.server.repo	valid file system path (default is ".")	Location on local file system where kie server state files will be stored	No
org.kie.server.persistence.ds	string	Datasource JNDI name	Yes when BPM support enabled
org.kie.server.persistence.tm	string	Transaction manager platform for Hibernate properties set	Yes when BPM support enabled
org.kie.server.persistence.dialect	string	Hibernate dialect to be used	Yes when BPM support enabled
org.jbpm.ht.callback	string	One of supported callbacks for Task Service (default jaas)	No
org.jbpm.ht.custom.callback	string	Custom implementation of UserGroupCallback in case org.jbpm.ht.callback was set to ‘custom’	No
kie.maven.settings.custom	valid file system path	Location of custom settings.xml for maven configuration	No
org.kie.executor.interval	integer (default is 3)	Number of time units between polls by executor	No

Property	Value	Description	Required
org.kie.executor.pool.size	Integer (default is 1)	Number of threads in the pool for async work	No
org.kie.executor.retry.count	Integer (default is 3)	Number of retries to handle errors	No
org.kie.executor.timeunit	TimeUnit (default is "SECONDS")	TimeUnit representing interval	No
org.kie.executor.disabled	Boolean (default is "false")	Disables executor completely	No
kie.server.jms.queues.response	String (default is "queue/KIE.SERVER.RESPONSE")	JNDI name of response queue for JMS	No
org.kie.server.controller.connect	Integer (default is 10000)	Waiting time in milliseconds between repeated attempts to connect kie server to controller when kie server starts up	No

Important

If you are running both KIE Server and KIE Workbench you must configure KIE Server to use a different Data Source to KIE Workbench using the **org.kie.server.persistence.ds** property. KIE Workbench uses a jBPM Executor Service that can conflict with KIE Server if they share the same Data Source.

22.2.2. Installation details for different containers

22.2.2.1. Tomcat 7.x/8.x

1. Download and unzip the Tomcat distribution. Let's call the root of the distribution `TOMCAT_HOME`. This directory is named after the Tomcat version, so for example `apache-tomcat-7.0.55`.
2. Download *kie-server-6.3.1-SNAPSHOT-webc.war* and place it into `TOMCAT_HOME/webapps`.
3. Configure user(s) and role(s). Make sure that file `TOMCAT_HOME/conf/tomcat-users.xml` contains the following username and role definition. You can of course choose different username and password, just make sure that the user has role `kie-server`:

Example 22.2. Username and role definition for Tomcat

```
<role rolename="kie-server"/>
```

```
<user username="serveruser" password="my.s3cr3t.pass" roles="kie-server"/>
```

4. Start the server by running `TOMCAT_HOME/bin/startup.[sh|bat]`. You can check out the Tomcat logs in `TOMCAT_HOME/logs` to see if the application deployed successfully. Please read the table above for the bootstrap switches that can be used to properly configure the instance. For instance:

```
./startup.sh -Dorg.kie.server.id=first-kie-server          -Dorg.kie.server.location=http://
localhost:8080/kie-server/services/rest/server
              -Dorg.kie.server.location=http://localhost:8080/kie-server/services/rest/
```

5. Verify the server is running. Go to `http://SERVER:PORT/CONTEXT/services/rest/server/` and type the specified username and password. You should see simple XML message with basic information about the server.

Important

You can not leverage the JMS interface when running on Tomcat, or any other Web container. The Web container version of the WAR contains only the REST interface.

22.2.2.2. WildFly 8.x

1. Download and unzip the WildFly distribution. Let's call the root of the distribution `WILDFLY_HOME`. This directory is named after the WildFly version, so for example `wildfly-8.2.0.Final`.

2. Download *kie-server-6.3.1-SNAPSHOT-ee7.war* and place it into `WILDFLY_HOME/standalone/deployments`.

3. Configure user(s) and role(s). Execute the following command `WILDFLY_HOME/bin/add-user.[sh|bat] -a -u 'kieserver' -p 'kieserver1!' -ro 'kie-server'`. You can of course choose different username and password, just make sure that the user has role `kie-server`.

4. Start the server by running `WILDFLY_HOME/bin/standalone.[sh|bat] -c standalone-full.xml <bootstrap_switches>`. You can check out the standard output or WildFly logs in `WILDFLY_HOME/standalone/logs` to see if the application deployed successfully. Please read the table above for the bootstrap switches that can be used to properly configure the instance. For instance:

```
./standalone.sh --server-config=standalone-full.xml            -Djboss.socket.binding.port-
offset=150                      -Dorg.kie.server.id=first-kie-server                        -
Dorg.kie.server.location=http://localhost:8230/kie-server/services/rest/server
                  -Djboss.socket.binding.port-
offset=150                 -Dorg.kie.server.id=first-kie-server
```

```
-Dorg.kie.server.location=http://localhost:8230/kie-server/services/rest/
```

5. Verify the server is running. Go to `http://SERVER:PORT/CONTEXT/services/rest/server/` and type the specified username and password. You should see simple XML message with basic information about the server.

22.3. Kie Server setup

Important

Server setup and registration changed significantly from versions 6.2 and before. The following applies only to version 6.3 and forward.

22.3.1. Managed Kie Server

A managed instance is one that requires a controller to be available to properly startup the Kie Server instance.

A Controller is a component responsible for keeping and managing a Kie Server Configuration in centralized way. Each controller can manager multiple configurations at once and there can be multiple controllers in the environment. Managed KIE Servers can be configured with a list of controllers but will connect to only one at a time.

Note

It's important to mention that even though there can be multiple controllers they should be kept in sync to make sure that regardless which one of them is contacted by KIE Server instance it will provide same set of configuration.

At startup, if a Kie Server is configured with a list of controllers, it will try succesivelly to connect to each of them until a connection is successfully stablished with one of them. If for any reason a connection can't be stablished, the server will not start, even if there is local storage available with configuration. This happens by design in order to ensure consistency. For instance, if the Kie Server was down and the configuration has changed, this restriction guarantees that it will run with up to date configuration or not at all.

Note

In order to run the Kie Server in standalone mode, without connecting to any controllers, please see "Unmanaged Kie Server".

The configuration sets, among other things:

- kie containers to be deployed and started
- configuration items - currently this is a place holder for further enhancements that will allow remotely configure KIE Execution Server components - timers, persistence, etc

The Controller, besides providing configuration management, is also responsible for overall management of Kie Servers. It provides a REST api that is divided into two parts:

- the controller itself that is exposed to interact with KIE Execution Server instances
- an administration API that allows to remotely manage Kie Server instances:
  - add/remove servers
  - add/remove containers to/from the servers
  - start/stop containers on servers

The controller deals only with the Kie Server configuration or definition to put it differently. It does not handle any runtime components of KIE Execution Server instances. They are always considered remote to controller. The controller is responsible for persisting the configuration to preserve restarts of the controller itself. It should manage the synchronization as well in case multiple controllers are configured to keep all definitions up to date on all instances of the controller.

By default controller is shipped with Kie Workbench and provides a fully featured management interface (both REST api and UI). It uses underlying git repository as persistent store and thus when GIT repositories are clustered (using Apache Zookeeper and Apache Helix) it will cover the controllers synchronization as well.

The diagram above illustrates the single controller (workbench) setup with multiple Kie Server instances managed by it.

The diagram bellow illustrates the clustered setup where there are multiple instances of controller synchronized over Zookeeper.

In the above diagram we can see that the Kie Server instances are capable of connecting to any controllers, but they will connect to only one. Each instance will attempt to connect to controller as long as it can reach one. Once connection is established with one of the controllers it will skip the others.

22.3.1.1. Working with managed servers

There are two approaches that users can take when working with managed KIE Server instances:

- *Configuration first:* with this approach, a user will start working with the controller (either UI or REST api) and create and configure Kie Server definitions. That consists basically of an

identification for the server definition (id and name + optionally version for improved readability) and the configuration for the Kie Containers to run on the server.

- *Registration first:* with this approach, the Kie Server instances are started first and auto register themselves on controller. The user then can configure the Kie Containers. This option simply skips the registration step done in the first approach and populates it with server id, name and version directly upon auto registration. There are no other differences between the two approaches.

22.3.2. Unmanaged KIE Execution Server

An unmanaged Kie Server is in turn just a standalone instance, and thus must be configured individually using REST/JMS api from the Kie Server itself. There is no controller involved. The configuration is automatically persisted by the server into a file and that is used as the internal server state, in case of restarts.

The configuration is updated during the following operations:

- deploy Kie Container
- undeploy Kie Container
- start Kie Container
- stop Kie Container

Note

if the Kie Server is restarted, it will try to restablish the same state that was persisted before shutdown. That means that Kie Containers that were running, will be started, but the ones that were stopped/disposed before, will not.

In most use cases, the Kie Server should be executed in managed mode as that provides some benefits, like a web user interface (if using the workbench as a controller) and some facilities for clustering.

22.4. Creating a Kie Container

Once your Execution Server is registered, you can start adding **Kie Containers** to it.

Kie Containers are self contained environments that have been provisioned to hold instances of your packaged and deployed rule instances.

1. Start by clicking the **+** icon next to the Execution Server where you want to deploy your Container. This will bring up the New Container screen.
2. If you know the **Group Name**, **Artifact Id** and **Version** (GAV) of your deployed package, then you can enter those details and click the Ok button to select that instance (and provide a name for the Container);

3. If you don't know these values, you can search KIE Workbench for all packages that can be deployed. Click the **Search** button without entering any value in the search field (you can narrow your search by entering any term that you know exists in the package that you want to deploy).

Important

INSERT SCREENSHOT HERE

The figure above shows that there are three deployable packages available to be used as containers on the Execution Server. Select the one that you want by clicking the Select button. This will auto-populate the **GAV** and you can then click the **Ok** button to use this deployable as the new Container.

4. Enter a name for this Container at the top and then press the Ok button.

Important

The Container name must be unique inside each execution server and must not contain any spaces.

Note

Just below the **GAV** row, you will see an uneditable row that shows you the **URL** for your Container against which you will be able to execute REST commands.

22.5. Managing Containers

Containers within the Execution Server can be started, stopped and updated from within KIE Workbench.

22.5.1. Starting a Container

Once registered, a Container is in the 'Stopped' mode. It can be started by first selecting it and then clicking the Start button. You can also select multiple Containers and start them all at the same time.

Once the Container is in the 'Running' mode, a green arrow appears next to it. If there are any errors starting the Container(s), red icons appear next to Containers and the Execution Server that they are deployed on.

You should check the logs of both the Execution Server and the current Business Central to see what the errors are before redeploying the Containers (and possibly the Execution Server).

22.5.2. Stopping and Deleting a Container

Similar to starting a Container, select the Container(s) that you want to stop (or delete) and click the Stop button (which replaces the Start button for that Container once it has entered the 'Running' mode) or the Delete button.

22.5.3. Updating a Container

You can update deployed `KieContainers` without restarting the Execution Server. This is useful in cases where the Business Rules change, creating new versions of packages to be provisioned.

You can have multiple versions of the same package provisioned and deployed, each to a different `KieContainer`.

To update deployments in a `KieContainer` dynamically, click on the icon next to the Container. This will open up the Container Info screen. An example of this screen is shown here:

Important

INSERT SCREENSHOT HERE

The Container Info screen is a useful tool because it not only allows you to see the endpoint for this `KieContainer`, but it also allows you to either manually or automatically refresh the provision if an update is available. The update can be manual or automatic:

Manual Update: To manually update a `KieContainer`, enter the new Version number in the Version box and click on the **Update** button. You can of course, update the Group Id or the Artifact Id , if these have changed as well. Once updated, the Execution server updates the container and shows you the resolved GAV attributes at the bottom of the screen in the **Resolved Release Id** section.

Automatic Update: If you want a deployed Container to always have the latest version of your deployment without manually editing it, you will need to set the Version property to the value of `LATEST` and start a `Scanner`. This will ensure that the deployed provision always contains the latest version. The Scanner can be started just once on demand by clicking the Scan Now button or you can start it in the background with scans happening at a specified interval (in seconds).You can also set this value to `LATEST` when you are first creating this deployment. The **Resolved Release Id** in this case will show you the actual, latest version number.

22.6. REST API

The Execution Server supports the following commands via the REST API.

Please note the following before using these commands:

- The base URL for these will remain as the endpoint defined earlier (for example: `http://SERVER:PORT/CONTEXT/services/rest/server/`)

- All requests require basic HTTP Authentication for the role kie-server as indicated earlier.

22.6.1. [GET] /

Returns the Execution Server information

Example 22.3. Example Server Response

```
<response type="SUCCESS" msg="KIE Server info">
  <kie-server-info>
    <version>6.2.0.redhat-1</version>
  </kie-server-info>
</response>
```

22.6.2. [POST] /

Using POST HTTP method, you can execute various commands on the Execution Server. E.g: create-container, list-containers, dispose-container and call-container.
Following is the full list of commands:

- CreateContainerCommand
- GetServerInfoCommand
- ListContainersCommand
- CallContainerCommand
- DisposeContainerCommand
- GetContainerInfoCommand
- GetScannerInfoCommand
- UpdateScannerCommand
- UpdateReleaseIdCommand

The commands itself can be found in the `org.kie.server.api.commands` package.

22.6.3. [GET] /containers

Returns a list of containers that have been created on this Execution Server.

Example 22.4. Example Server Response

```
<response type="SUCCESS" msg="List of created containers">
  <kie-containers>
    <kie-container container-id="MyProjectContainer" status="STARTED">
      <release-id>
```

```
        <artifact-id>Project1</artifact-id>
        <group-id>com.redhat</group-id>
        <version>1.0</version>
      </release-id>
      <resolved-release-id>
        <artifact-id>Project1</artifact-id>
        <group-id>com.redhat</group-id>
        <version>1.0</version>
      </resolved-release-id>
    </kie-container>
  </kie-containers>
</response>
```

22.6.4. [GET] /containers/{id}

Returns the status and information about a particular container. For example, executing `http://SERVER:PORT/CONTEXT/services/rest/server/containers/MyProjectContainer` could return the following example container info.

Example 22.5. Example Server Response

```
`<response type="SUCCESS" msg="Info for container MyProjectContainer">
  <kie-container container-id="MyProjectContainer" status="STARTED">
    <release-id>
      <artifact-id>Project1</artifact-id>
      <group-id>com.redhat</group-id>
      <version>1.0</version>
    </release-id>
    <resolved-release-id>
      <artifact-id>Project1</artifact-id>
      <group-id>com.redhat</group-id>
      <version>1.0</version>
    </resolved-release-id>
  </kie-container>
</response>
```

22.6.5. [PUT] /containers/{id}

Allows you to create a new Container in the Execution Server. For example, to create a Container with the id of **MyRESTContainer** the complete endpoint will be: `http://SERVER:PORT/CONTEXT/services/rest/server/containers/MyRESTContainer`. An example of request is:

Example 22.6. Example Request to create a container

```
<kie-container container-id="MyRESTContainer">
  <release-id>
    <artifact-id>Project1</artifact-id>
    <group-id>com.redhat</group-id>
    <version>1.0</version>
  </release-id>
```

```
</kie-container>
```

And the response from the server, if successful, would be be:

Example 22.7. Example Server Response when creating a container

```
<response type="SUCCESS" msg="Container MyRESTContainer successfully deployed with module
 com.redhat:Project1:1.0">
  <kie-container container-id="MyProjectContainer" status="STARTED">
    <release-id>
      <artifact-id>Project1</artifact-id>
      <group-id>com.redhat</group-id>
      <version>1.0</version>
    </release-id>
    <resolved-release-id>
      <artifact-id>Project1</artifact-id>
      <group-id>com.redhat</group-id>
      <version>1.0</version>
    </resolved-release-id>
  </kie-container>
</response>
```

22.6.6. [DELETE] /containers/{id}

Disposes the Container specified by the id. For example, executing `http://SERVER:PORT/CONTEXT/services/rest/server/containers/MyProjectContainer` using the DELETE HTTP method will return the following server response:

Example 22.8. Example Server Response disposing a container

```
<response type="SUCCESS" msg="Container MyProjectContainer successfully disposed."/>
```

22.6.7. [POST] /containers/{id}

Executes operations and commands against the specified Container. You can send commands to this Container in the body of the POST request. For example, to fire all rules for Container with id MyRESTContainer (`http://SERVER:PORT/CONTEXT/services/rest/server/containers/MyRESTContainer`), you would send the fire-all-rules command to it as shown below (in the body of the POST request):

Example 22.9. Example Server Request to fire all rules

```
<fire-all-rules/>
```

Following is the list of supported commands:

- AgendaGroupSetFocusCommand
- ClearActivationGroupCommand
- ClearAgendaCommand
- ClearAgendaGroupCommand
- ClearRuleFlowGroupCommand
- DeleteCommand
- InsertObjectCommand
- ModifyCommand
- GetObjectCommand
- InsertElementsCommand
- FireAllRulesCommand
- QueryCommand
- SetGlobalCommand
- GetGlobalCommand
- GetObjectsCommand
- BatchExecutionCommand

These commands can be found in the `org.drools.core.command.runtime` package.

22.6.8. [GET] /containers/{id}/release-id

Returns the full release id for the Container specified by the id.

Example 22.10. Example Server Response

```
`<response type="SUCCESS" msg="ReleaseId for container MyProjectContainer">
  <release-id>
    <artifact-id>Project1</artifact-id>
    <group-id>com.redhat</group-id>
    <version>1.0</version>
  </release-id>
</response>
```

22.6.9. [POST] /containers/{id}/release-id

Allows you to update the release id of the container deployment. Send the new complete release id to the Server.

Example 22.11. Example Server Request

```
<release-id>
  <artifact-id>Project1</artifact-id>
  <group-id>com.redhat</group-id>
  <version>1.1</version>
</release-id>
```

The Server will respond with a success or error message, similar to the one below:

Example 22.12. Example Server Response

```
<response type="SUCCESS" msg="Release id successfully updated.">  <release-id>     <artifact-
id>Project1</artifact-id>      <group-id>com.redhat</group-id>     <version>1.0</version>    </
release-id> </response>
fully up
dated.">  <release-id>
 <artifact-id>Project1</artifact-id>
  <group-id>com.redhat</group-id>
   <version>1.0</version>
```

22.6.10. [GET] /containers/{id}/scanner

Returns information about the scanner for this Container's automatic updates.

Example 22.13. Example Server Response

```
<response type="SUCCESS" msg="Scanner info successfully retrieved">
  <kie-scanner status="DISPOSED"/>
</response>
```

22.6.11. [POST] /containers/{id}/scanner

Allows you to start or stop a scanner that controls polling for updated Container deployments. To start the scanner, send a request similar to: `http://SERVER:PORT/CONTEXT/services/rest/server/containers/{container-id}/scanner` with the following POST data.

Example 22.14. Example Server Request to start the scanner

```
<kie-scanner status="STARTED" poll-interval="20"/>
```

The poll-interval attribute is in seconds. The response from the server will be similar to:

Example 22.15. Example Server Response

```
<response type="SUCCESS" msg="Kie scanner successfully created.">
  <kie-scanner status="STARTED"/>
</response>
```

To stop the Scanner, replace the status with `DISPOSED` and remove the poll-interval attribute.

22.6.12. Native REST client for Execution Server

Commands outlined in this section can be sent with any REST client, whether it is curl, RESTEasy or .NET based application. However, when sending requests from Java based application, users can utilize out of the box native client for remote communication with Execution Server. This client is part of the **org.kie:kie-server-client** project. It doesn't allow creating XML request, therefore it is necessary generate them before, for example, using Drools API.

Example 22.16. Generate XML request

```
import java.util.ArrayList;
import java.util.List;

import org.drools.core.command.impl.GenericCommand;
import org.drools.core.command.runtime.BatchExecutionCommandImpl;
import org.drools.core.command.runtime.rule.FireAllRulesCommand;
import org.drools.core.command.runtime.rule.InsertObjectCommand;
import org.kie.api.command.BatchExecutionCommand;
import org.kie.internal.runtime.helper.BatchExecutionHelper;

public class DecisionClient {

public static void main(String args[]) {
        Bean1 bean1 = new Bean1();
        bean1.setName("Robert");

        InsertObjectCommand insertObjectCommand = new InsertObjectCommand(bean1, "f1");
        FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand("myFireCommand");

        List<GenericCommand<?>> commands = new ArrayList<GenericCommand<?>>();
        commands.add(insertObjectCommand);
        commands.add(fireAllRulesCommand);
        BatchExecutionCommand command = new BatchExecutionCommandImpl(commands);

           String xStreamXml = BatchExecutionHelper.newXStreamMarshaller().toXML(command); //
 actual XML request
    }
}
```

Once the request is generated it can be sent using **kie-server-client** as follows:

Example 22.17. Sending XML request with kie-server-client

```
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;

//user "anton" must have role "kie-server" assigned
KieServicesConfiguration config =  KieServicesFactory.
        newRestConfiguration("http://localhost:8080/kie-server/services/rest/server",
        "anton",
        "password1!");
 KieServicesClient client = KieServicesFactory.newKieServicesClient(config);
// the request "xStreamXml" we generated in previous step
// "ListenerReproducer" is the name of the Container
ServiceResponse<String> response = client.executeCommands("ListenerReproducer", xStreamXml);
System.out.println(response.getResult());
```

Part VII. Drools Examples

Examples to help you learn Drools

Chapter 23. Examples

23.1. Getting the Examples

Make sure the Drools Eclipse plugin is installed, which needs the Graphical Editing Framework (GEF) dependency installed first. Then download and extract the drools-examples zip file, which includes an already created Eclipse project. Import that project into a new Eclipse workspace. The rules all have example classes that execute the rules. If you want to try the examples in another project (or another IDE) then you will need to set up the dependencies by hand, of course. Many, but not all of the examples are documented below, enjoy!

Some examples require Java 1.6 to run.

23.2. Hello World

Name: Hello World
Main class: org.drools.examples.helloworld.HelloWorldExample
Module: drools-examples
Type: Java application
Rules file: HelloWorld.drl
Objective: demonstrate basic rules in use

The "Hello World" example shows a simple application using rules, written both using the MVEL and the Java dialects.

This example demonstrates how to create and use a `KieSession`. Also, audit logging and debug outputs are shown, which is omitted from other examples as it's all very similar.

The following code snippet shows how the session is created with only 3 lines of code.

Example 23.1. HelloWorld: Creating the KieSession

```
KieServices ks = KieServices.Factory.get();
KieContainer kc = ks.getKieClasspathContainer();
KieSession ksession = kc.newKieSession("HelloWorldKS");
```

Obtains the `KieServices` factory. This is the main interface applications use to interact with the engine.
Creates a `KieContainer` from the project classpath. This will look for a `/META-INF/kmodule.xml` file to configure and instantiate the `KieModule` into the `KieContainer`.
Creates a session based on the named "HelloWorldKS" session configuration.

Drools has an event model that exposes much of what's happening internally. Two default debug listeners are supplied, `DebugAgendaEventListener` and `DebugWorkingMemoryEventListener`

which print out debug event information to the `System.err` stream displayed in the Console window. Adding listeners to a Session is trivial, as shown in the next snippet. The `KieRuntimeLogger` provides execution auditing, the result of which can be viewed in a graphical viewer. The logger is actually a specialised implementation built on the `Agenda` and `RuleRuntime` listeners. When the engine has finished executing, `logger.close()` must be called.

Most of the examples use the Audit logging features of Drools to record execution flow for later inspection.

Example 23.2. HelloWorld: Event logging and Auditing

```
        // The application can also setup listeners
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // To setup a file based audit logger, uncomment the next line
        // KieRuntimeLogger logger = ks.getLoggers().newFileLogger( ksession, "./helloworld" );

       // To setup a ThreadedFileLogger, so that the audit view reflects events whilst debugging,
        // uncomment the next line
           // KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "./
helloworld", 1000 );
```

The single class used in this example is very simple. It has two fields: the message, which is a `String` and the status which can be one of the two integers `HELLO` or `GOODBYE`.

Example 23.3. HelloWorld example: Message Class

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
    private int             status;
    ...
}
```

A single `Message` object is created with the message text "Hello World" and the status `HELLO` and then inserted into the engine, at which point `fireAllRules()` is executed.

Example 23.4. HelloWorld: Execution

```
        // The application can insert facts into the session
        final Message message = new Message();
        message.setMessage( "Hello World" );
        message.setStatus( Message.HELLO );
        ksession.insert( message );
```

```
        // and fire the rules
        ksession.fireAllRules();
```

To execute the example as a Java application:

1. Open the class `org.drools.examples.helloworld.HelloWorldExample` in your Eclipse IDE
2. Right-click the class and select "Run as..." and then "Java application"

If we put a breakpoint on the `fireAllRules()` method and select the `ksession` variable, we can see that the "Hello World" rule is already activate on the Agenda.

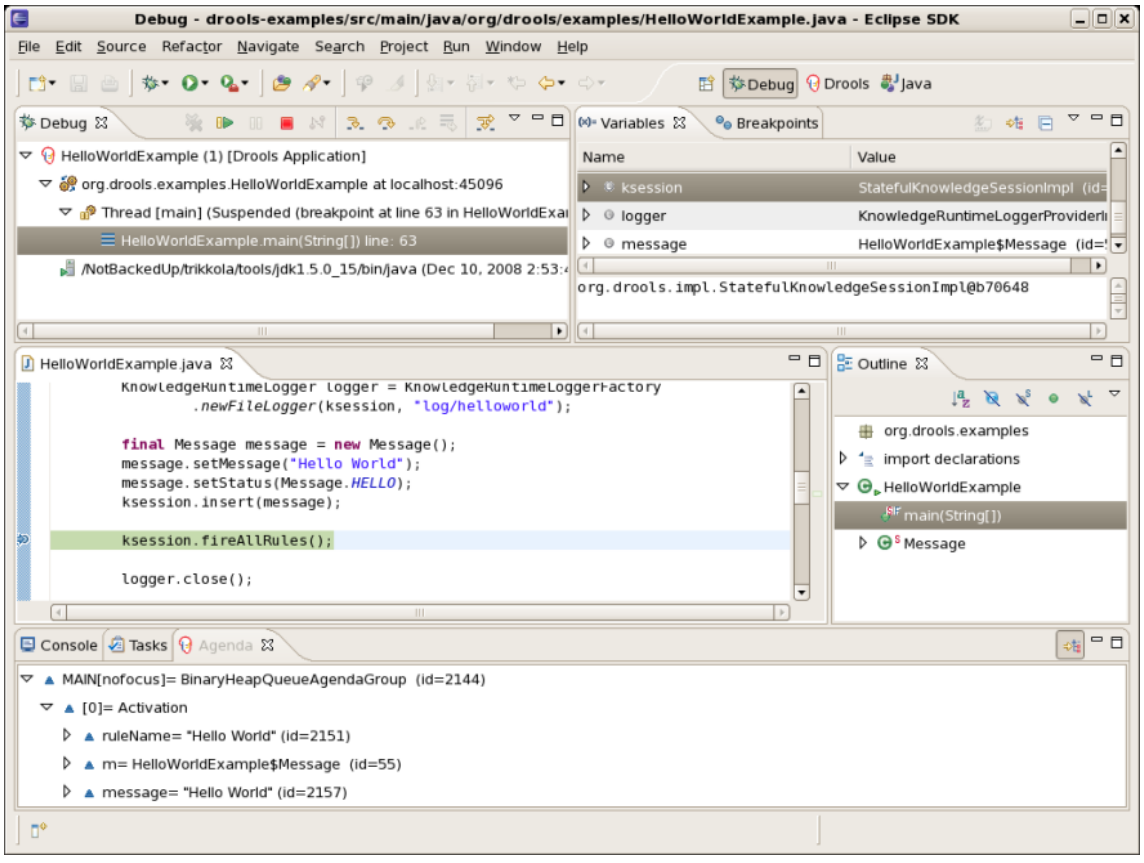


Figure 23.1. Hello World: fireAllRules Agenda View

The application print outs go to to `System.out` while the debug listener print outs go to `System.err`.

Example 23.5. HelloWorld: System.out in the Console window

```
Hello WorldGoodbye cruel world
WorldGoodbye cruel
```

Example 23.6. HelloWorld: System.err in the Console window

```
==>[ActivationCreated(0):    rule=Hello    World;
                                  tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96]][ObjectInserted:
   handle=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
                                       object=org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96][BeforeActivationFired:   rule=Hello   World;
                                  tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96]]==>[ActivationCreated(4):   rule=Good   Bye;
                                  tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96]][ObjectUpdated:
  handle=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
                   old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;
                                     new_object=org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96][AfterActivationFired(0): rule=Hello  World][BeforeActivationFired: rule=Good
 Bye;                           tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96]][AfterActivationFired(4): rule=Good Bye]
ed(0): rule=Hello World;


tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]][ObjectInserted: handle=[fid:1:1
    object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
    tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;


tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]][ObjectUpdated: handle=[fid:1:2:o
    old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;
    new_object=org.drools.examples.helloworld.HelloWorldExample
$Message@17cec96][AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
    tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]][AfterActivationFired(4):
```

The actual rules are inside the file `src/main/resources/org/drools/examples/helloworld/HelloWorld.drl`:

Example 23.7. HelloWorld: rule "Hello World"

```
rule "Hello World"     dialect "mvel"  when     m : Message( status == Message.HELLO, message :
 message )  then      System.out.println( message );        modify ( m ) { message = "Goodbye
 cruel world",                     status = Message.GOODBYE };end
World"      dialect
"mvel"
when      m : Message( status == Message.HELLO, message : message
)
then      System.out.println( message );
       modify ( m ) { message = "Goodbye cruel
world",                      status = Message.GOODBYE
};
```

The LHS (after `when`) section of the rule states that it will be activated for each `Message` object inserted into the Rule Runtime whose status is `Message.HELLO`. Besides that, two variable bindings are created: the variable `message` is bound to the `message` attribute and the variable `m` is bound to the matched `Message` object itself.

The RHS (after `then`) or consequence part of the rule is written using the MVEL expression language, as declared by the rule's attribute `dialect`. After printing the content of the bound variable `message` to `System.out`, the rule changes the values of the `message` and `status` attributes of the `Message` object bound to `m`. This is done using MVEL's `modify` statement, which allows you to apply a block of assignments in one statement, with the engine being automatically notified of the changes at the end of the block.

It is possible to set a breakpoint into the DRL, on the `modify` call, and inspect the Agenda view again during the execution of the rule's consequence. This time we start the execution via "Debug As" and "Drools application" and not by running a "Java application":

1. Open the class `org.drools.examples.HelloWorld` in your Eclipse IDE.

2. Right-click the class and select "Debug as..." and then "Drools application".

Now we can see that the other rule `"Good Bye"`, which uses the Java dialect, is activated and placed on the Agenda.

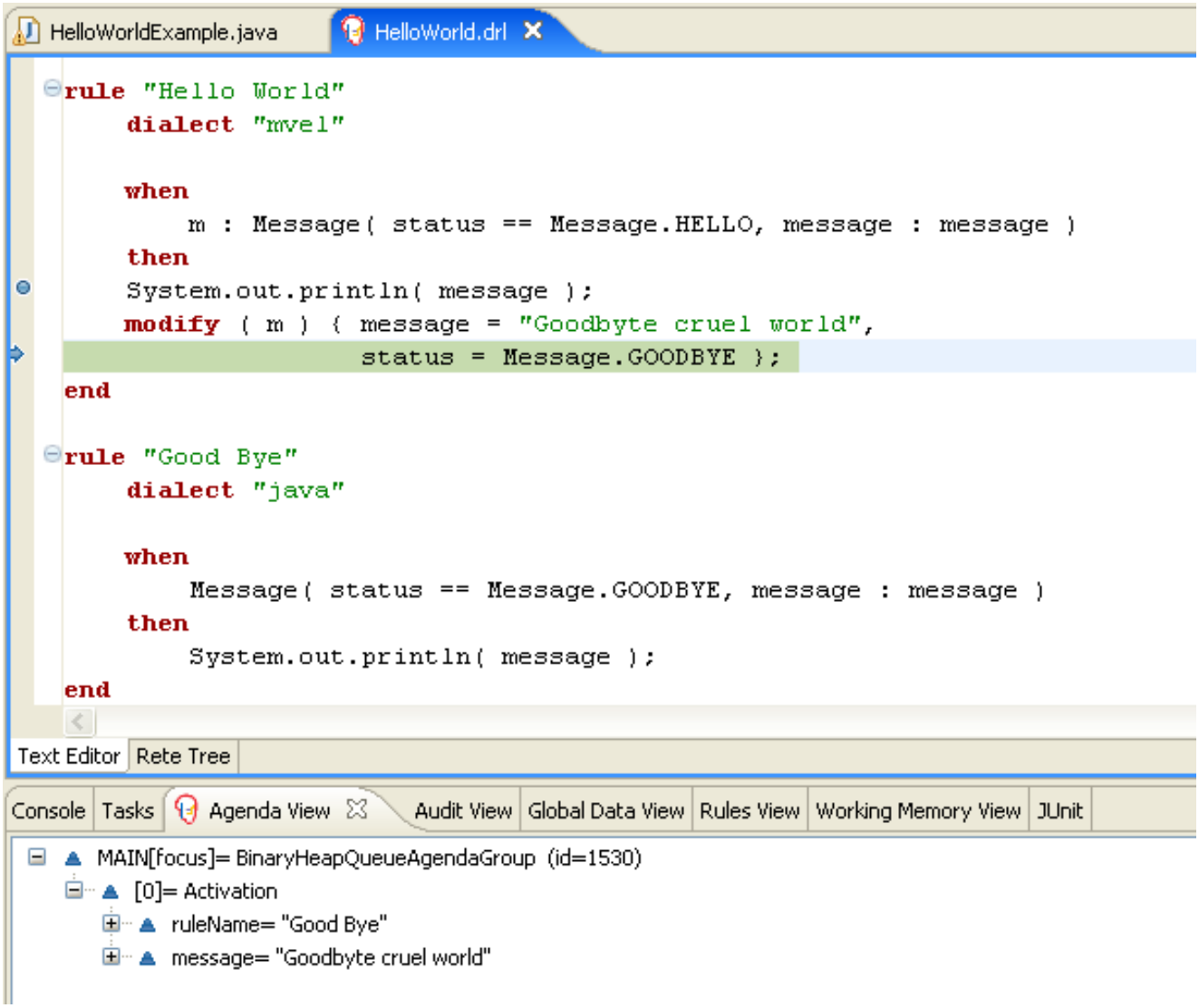


Figure 23.2. Hello World: rule "Hello World" Agenda View

The "Good Bye" rule, which specifies the "java" dialect, is similar to the "Hello World" rule except that it matches `Message` objects whose status is `Message.GOODBYE`.

Example 23.8. HelloWorld: rule "Good Bye"

```
rule "Good Bye"
      dialect "java"
  when
      Message( status == Message.GOODBYE, message : message )
  then
      System.out.println( message );
end
```

The Java code that instantiates the `KieRuntimeLogger` creates an audit log file that can be loaded into the Audit view. The Audit view is used in many of the examples to demonstrate the example execution flow. In the view screen shot below we can see that the object is inserted, which creates an activation for the "Hello World" rule; the activation is then executed which updates the `Message` object causing the "Good Bye" rule to activate; finally the "Good Bye" rule also executes. Selecting an event in the Audit view highlights the origin event in green; therefore the "Activation created" event is highlighted in green as the origin of the "Activation executed" event.

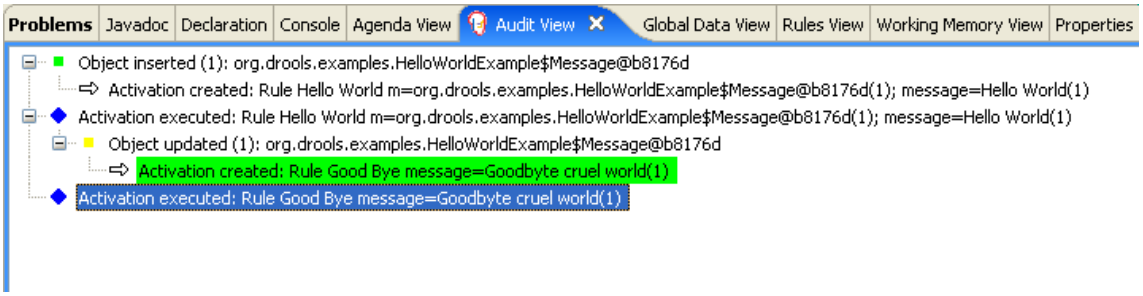


Figure 23.3. Hello World: Audit View

23.3. State Example

This example is implemented in two different versions to demonstrate different ways of implementing the same basic behavior: forward chaining, i.e., the ability the engine has to evaluate, activate and fire rules in sequence, based on changes on the facts in the Working Memory.

23.3.1. Understanding the State Example

```
Name: State Example
Main class: org.drools.examples.state.StateExampleUsingSalience
Module: drools-examples
Type: Java application
Rules file: StateExampleUsingSalience.drl
Objective: Demonstrates basic rule use and Conflict Resolution for rule
 firing priority.
```

Each `State` class has fields for its name and its current state (see the class `org.drools.examples.state.State`). The two possible states for each objects are:

- `NOTRUN`
- `FINISHED`

Example 23.9. State Class

```
public class State {
```

```
    public static final int NOTRUN   = 0;
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

Ignoring the `PropertyChangeSupport`, which will be explained later, we see the creation of four `State` objects named A, B, C and D. Initially their states are set to `NOTRUN`, which is default for the used constructor. Each instance is asserted in turn into the Session and then `fireAllRules()` is called.

Example 23.10. Salience State: Execution

```
        final State a = new State( "A" );
        final State b = new State( "B" );
        final State c = new State( "C" );
        final State d = new State( "D" );

        ksession.insert( a );
        ksession.insert( b );
        ksession.insert( c );
        ksession.insert( d );

        ksession.fireAllRules();

        ksession.dispose(); // Stateful rule session must always be disposed when finished
```

To execute the application:

1. Open the class `org.drools.examples.state.StateExampleUsingSalience` in your Eclipse IDE.
2. Right-click the class and select "Run as..." and then "Java application"

You will see the following output in the Eclipse console window:

Example 23.11. Salience State: Console Output

```
A finishedB finishedC finishedD finished
ishedB
finishedC
finishedD
```

There are four rules in total. First, the `Bootstrap` rule fires, setting A to state `FINISHED`, which then causes B to change its state to `FINISHED`. C and D are both dependent on B, causing a conflict which is resolved by the salience values. Let's look at the way this was executed.

The best way to understand what is happening is to use the Audit Logging feature to graphically see the results of each operation. To view the Audit log generated by a run of this example:

1. If the Audit View is not visible, click on "Window" and then select "Show View", then "Other..." and "Drools" and finally "Audit View".
2. In the "Audit View" click the "Open Log" button and select the file "<drools-examples-dir>/log/state.log".

After that, the "Audit view" will look like the following screenshot:

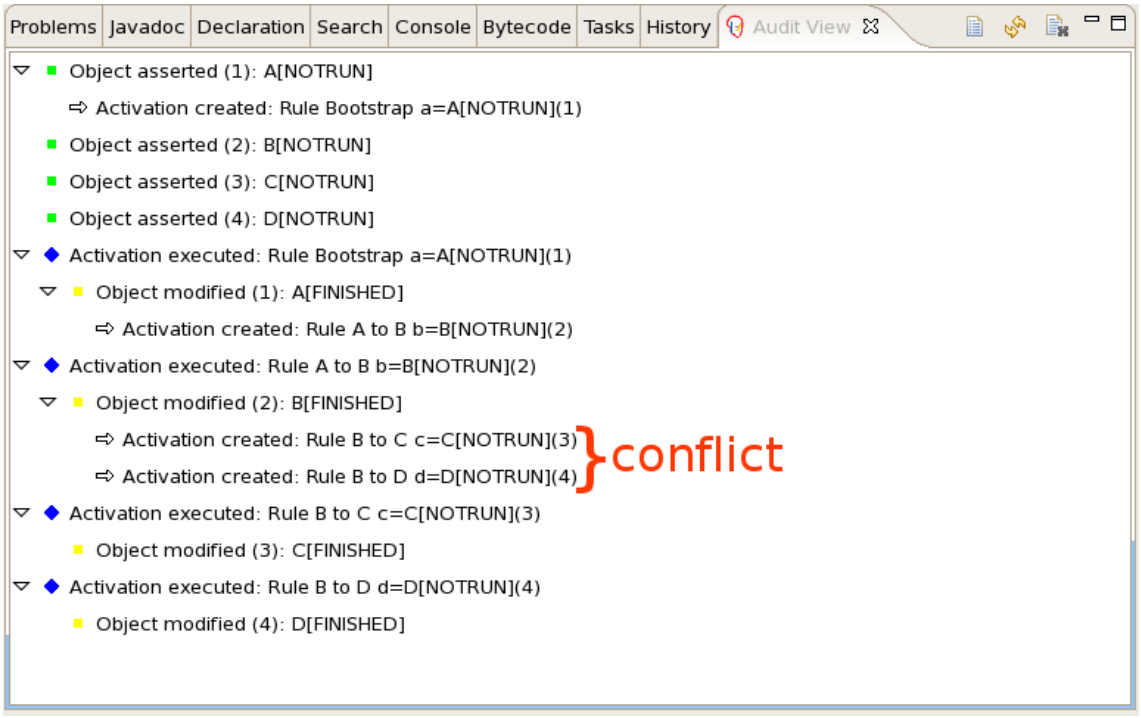


Figure 23.4. Salience State Example Audit View

Reading the log in the "Audit View", top to bottom, we see every action and the corresponding changes in the Working Memory. This way we observe that the assertion of the State object A in the state `NOTRUN` activates the `Bootstrap` rule, while the assertions of the other `State` objects have no immediate effect.

Example 23.12. Salience State: Rule "Bootstrap"

```
rule Bootstrap     when         a : State(name == "A", state == State.NOTRUN )     then
 System.out.println(a.getName() + " finished" );         a.setState( State.FINISHED );end
```

```
strap
when        a : State(name == "A", state == State.NOTRUN
)
then        System.out.println(a.getName() + " finished"
);        a.setState( State.FINISHED
);
```

The execution of rule Bootstrap changes the state of A to `FINISHED`, which, in turn, activates rule "A to B".

Example 23.13. Salience State: Rule "A to B"

```
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished" );
        b.setState( State.FINISHED );
end
```

The execution of rule "A to B" changes the state of B to `FINISHED`, which activates both, rules "B to C" and "B to D", placing their Activations onto the Agenda. From this moment on, both rules may fire and, therefore, they are said to be "in conflict". The conflict resolution strategy allows the engine's Agenda to decide which rule to fire. As rule "B to C" has the **higher salience value** (10 versus the default salience value of 0), it fires first, modifying object C to state `FINISHED`. The Audit view shown above reflects the modification of the `State` object in the rule "A to B", which results in two activations being in conflict. The Agenda view can also be used to investigate the state of the Agenda, with debug points being placed in the rules themselves and the Agenda view opened. The screen shot below shows the breakpoint in the rule "A to B" and the state of the Agenda with the two conflicting rules.

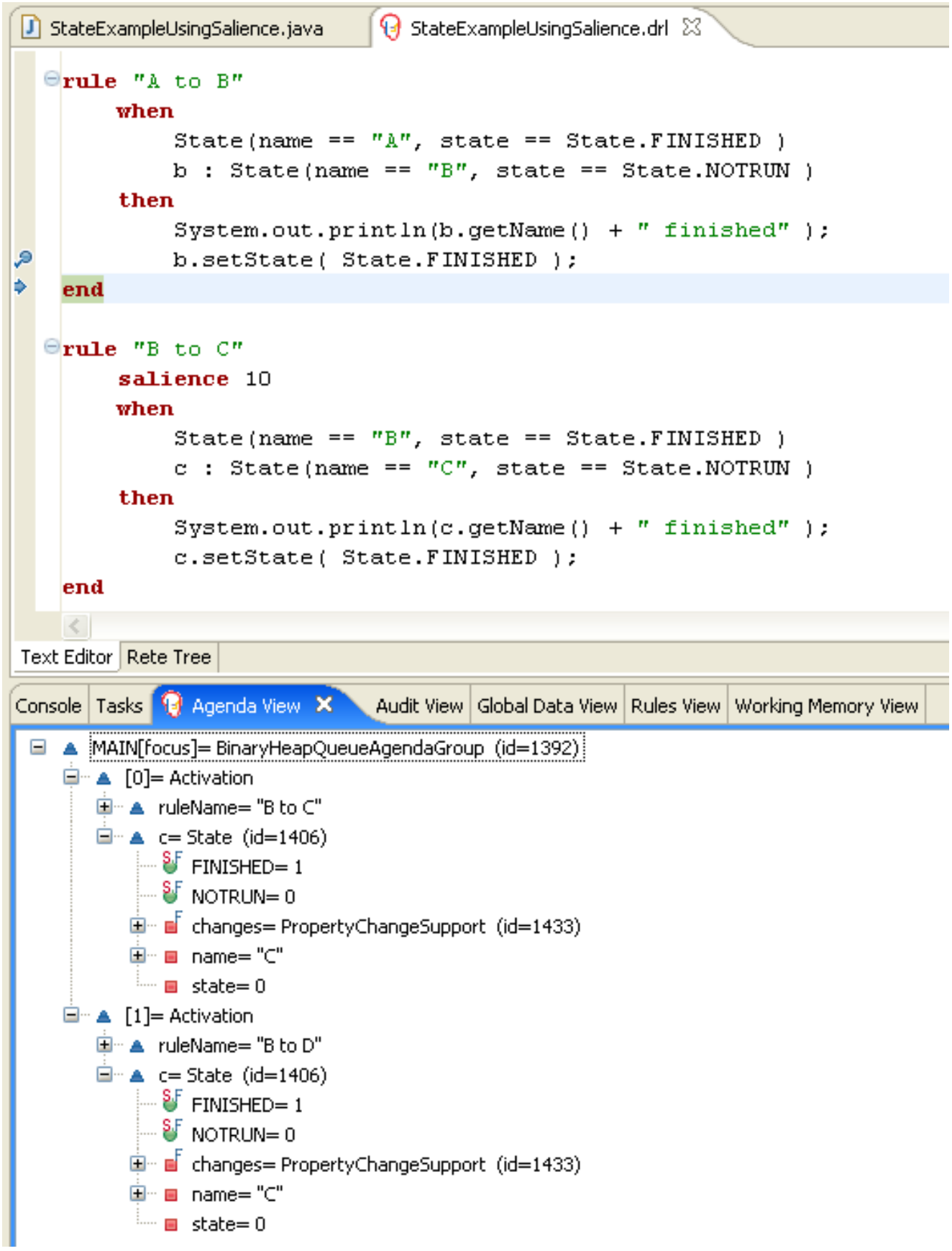


Figure 23.5. State Example Agenda View

Example 23.14. Salience State: Rule "B to C"

```
rule "B to C"
        salience 10
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished" );
        c.setState( State.FINISHED );
end
```

Rule "B to D" fires last, modifying object D to state `FINISHED`.

Example 23.15. Salience State: Rule "B to D"

```
rule "B to D"
    when
        State(name == "B", state == State.FINISHED )
        d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished" );
        d.setState( State.FINISHED );
end
```

There are no more rules to execute and so the engine stops.

Another notable concept in this example is the use of *dynamic facts*, based on `PropertyChangeListener` objects. As described in the documentation, in order for the engine to see and react to changes of fact properties, the application must tell the engine that changes occurred. This can be done explicitly in the rules by using the `modify` statement, or implicitly by letting the engine know that the facts implement `PropertyChangeSupport` as defined by the *JavaBeans specification*. This example demonstrates how to use `PropertyChangeSupport` to avoid the need for explicit `modify` statements in the rules. To make use of this feature, ensure that your facts implement `PropertyChangeSupport`, the same way the class `org.drools.example.State` does, and use the following code in the rules file to configure the engine to listen for property changes on those facts:

Example 23.16. Declaring a Dynamic Fact

```
declare type State
    @propertyChangeSupport
end
```

When using `PropertyChangeListener` objects, each setter must implement a little extra code for the notification. Here is the setter for `state` in the class `org.drools.examples`:

Example 23.17. Setter Example with PropertyChangeSupport

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                     oldState,
                                     newState );
}
```

There are another class in this example: `StateExampleUsingAgendaGroup`. It executes from A to B to C to D, as just shown, but `StateExampleUsingAgendaGroup` uses agenda-groups to control the rule conflict and which one fires first.

Agenda groups are a way to partition the Agenda into groups and to control which groups can execute. By default, all rules are in the agenda group "MAIN". The "agenda-group" attribute lets you specify a different agenda group for the rule. Initially, a Working Memory has its focus on the Agenda group "MAIN". A group's rules will only fire when the group receives the focus. This can be achieved either ny using the method by `setFocus()` or the rule attribute `auto-focus`. "auto-focus" means that the rule automatically sets the focus to its agenda group when the rule is matched and activated. It is this "auto-focus" that enables rule "B to C" to fire before "B to D".

Example 23.18. Agenda Group State Example: Rule "B to C"

```
rule "B to C"
      agenda-group "B to C"
      auto-focus true
  when
      State(name == "B", state == State.FINISHED )
      c : State(name == "C", state == State.NOTRUN )
  then
      System.out.println(c.getName() + " finished" );
      c.setState( State.FINISHED );
      kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
end
```

The rule "B to C" calls `setFocus()` on the agenda group "B to D", allowing its active rules to fire, which allows the rule "B to D" to fire.

Example 23.19. Agenda Group State Example: Rule "B to D"

```
rule "B to D"
      agenda-group "B to D"
  when
      State(name == "B", state == State.FINISHED )
      d : State(name == "D", state == State.NOTRUN )
  then
      System.out.println(d.getName() + " finished" );
```

```
      d.setState( State.FINISHED );
end
```

23.4. Fibonacci Example

Name: Fibonacci
Main class: org.drools.examples.fibonacci.FibonacciExample
Module: drools-examples
Type: Java application
Rules file: Fibonacci.drl
Objective: Demonstrates Recursion,
the CE not and cross product matching

The Fibonacci Numbers (see http://en.wikipedia.org/wiki/Fibonacci_number) discovered by Leonardo of Pisa (see http://en.wikipedia.org/wiki/Fibonacci) is a sequence that starts with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers. The Fibonacci sequence begins with 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,... The Fibonacci Example demonstrates recursion and conflict resolution with salience values.

The single fact class `Fibonacci` is used in this example. It has two fields, sequence and value. The sequence field is used to indicate the position of the object in the Fibonacci number sequence. The value field shows the value of that Fibonacci object for that sequence position, using -1 to indicate a value that still needs to be computed.

Example 23.20. Fibonacci Class

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

Execute the example:

1. Open the class `org.drools.examples.fibonacci.FibonacciExample` in your Eclipse IDE.

2. Right-click the class and select "Run as..." and then "Java application"

Eclipse shows the following output in its console window (with "...snip..." indicating lines that were removed to save space):

Example 23.21. Fibonacci Example: Console Output

```
recurse for 50recurse for 49recurse for 48recurse for 47...snip...recurse for 5recurse for
 4recurse for 3recurse for 21 == 12 == 13 == 24 == 35 == 56 == 8...snip...47 == 297121507348
 == 480752697649 == 777874204950 == 12586269025
50recurse for
49recurse for
48recurse for
47
...snip...recurse for
5recurse for
4recurse for
3recurse for
21 ==
12 ==
13 ==
24 ==
35 ==
56 ==
8
...snip...47 ==
297121507348 ==
480752697649 ==
777874204950 ==
```

To kick this off from Java we only insert a single Fibonacci object, with a sequence field of 50. A recursive rule is then used to insert the other 49 `Fibonacci` objects. This example doesn't use `PropertyChangeSupport`. It uses the MVEL dialect, which means we can use the `modify` keyword, which allows a block setter action which also notifies the engine of changes.

Example 23.22. Fibonacci Example: Execution

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

The rule Recurse is very simple. It matches each asserted `Fibonacci` object with a value of -1, creating and asserting a new `Fibonacci` object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to 1 does not exist, the rule re-matches and fires again. The `not` conditional element is used to stop the rule's matching once we have all 50 Fibonacci objects in memory. The rule also has a salience value, because we need to have all 50 `Fibonacci` objects asserted before we execute the Bootstrap rule.

Example 23.23. Fibonacci Example: Rule "Recurse"

```
rule Recurse    salience 10    when        f : Fibonacci ( value == -1 )        not ( Fibonacci
 ( sequence == 1 ) )     then          insert( new Fibonacci( f.sequence - 1 ) );
 System.out.println( "recurse for " + f.sequence );end
```

```
curse    salience
10
when        f : Fibonacci ( value == -1
)        not ( Fibonacci ( sequence == 1 )
)
then        insert( new Fibonacci( f.sequence - 1 )
);       System.out.println( "recurse for " + f.sequence
);
```

The Audit view shows the original assertion of the `Fibonacci` object with a sequence field of 50, done from Java code. From there on, the Audit view shows the continual recursion of the rule, where each asserted `Fibonacci` object causes the Recurse rule to become activated and to fire again.

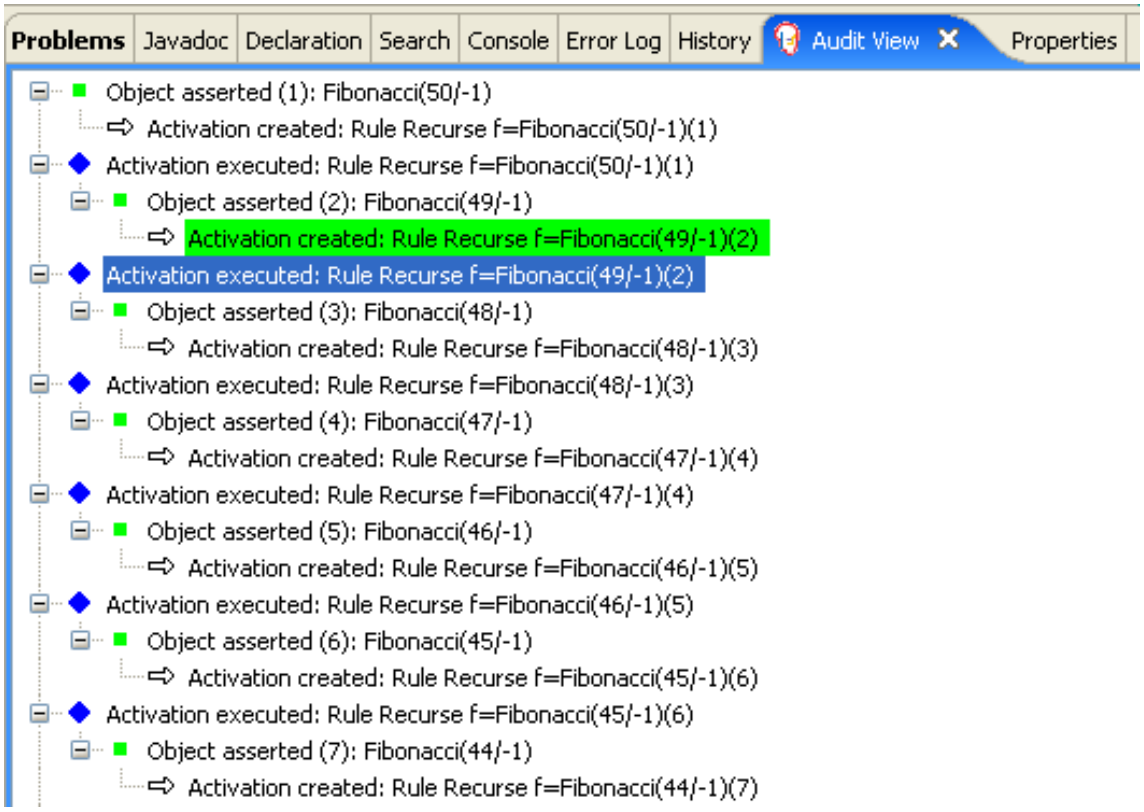


Figure 23.6. Fibonacci Example: "Recurse" Audit View 1

When a `Fibonacci` object with a sequence field of 2 is asserted the "Bootstrap" rule is matched and activated along with the "Recurse" rule. Note the multi-restriction on field `sequence`, testing for equality with 1 or 2.

Example 23.24. Fibonacci Example: Rule "Bootstrap"

```
rule Bootstrap     when         f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-
restriction     then          modify ( f ){ value = 1 };         System.out.println( f.sequence
 + " == " + f.value );end
```

```
strap
when        f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-
restriction    then
         modify ( f ){ value = 1
};        System.out.println( f.sequence + " == " + f.value
);
```

At this point the Agenda looks as shown below. However, the "Bootstrap" rule does not fire because the "Recurse" rule has a higher salience.

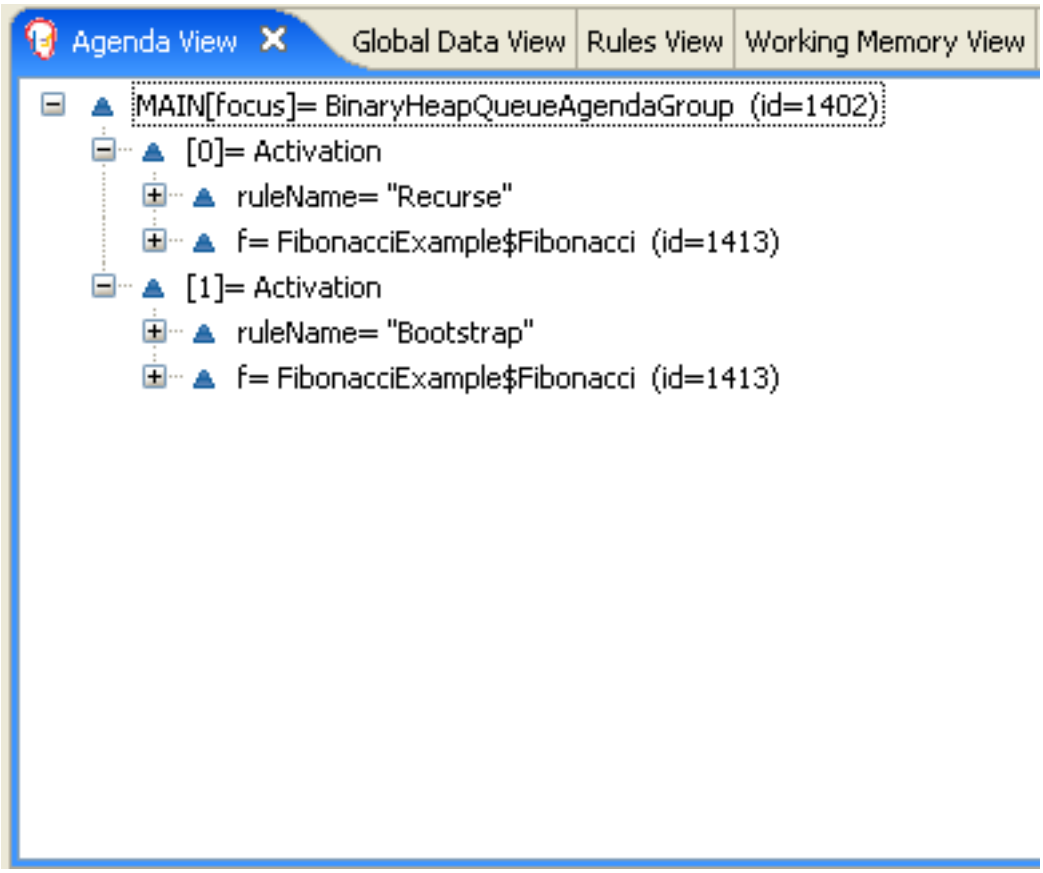


Figure 23.7. Fibonacci Example: "Recurse" Agenda View 1

When a `Fibonacci` object with a sequence of 1 is asserted the Bootstrap rule is matched again, causing two activations for this rule. Note that the "Recurse" rule does not match and activate because the `not` conditional element stops the rule's matching as soon as a `Fibonacci` object with a sequence of 1 exists.

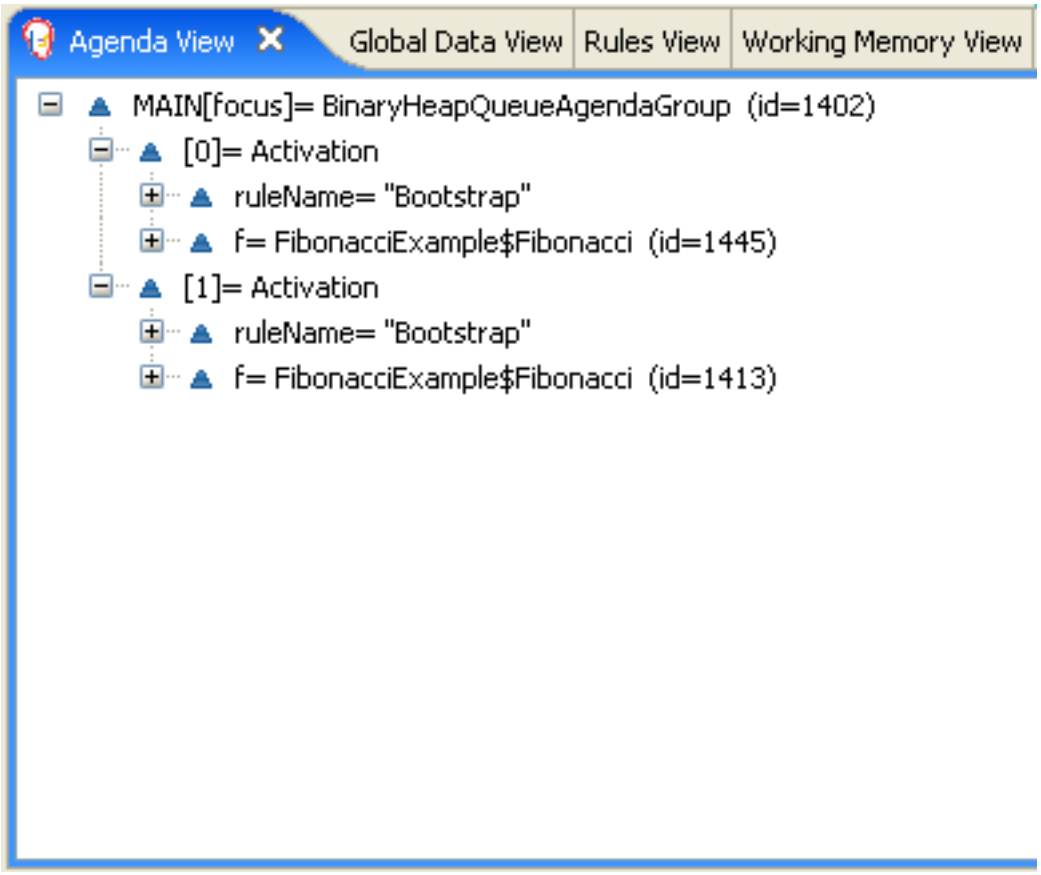


Figure 23.8. Fibonacci Example: "Recurse" Agenda View 2

Once we have two `Fibonacci` objects with values not equal to -1 the "Calculate" rule is able to match. It was the "Bootstrap" rule that set the objects with sequence 1 and 2 to values of 1. At this point we have 50 Fibonacci objects in the Working Memory. Now we need to select a suitable triple to calculate each of their values in turn. Using three Fibonacci patterns in a rule without field constraints to confine the possible cross products would result in 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect. The "Calculate" rule uses field constraints to correctly constraint the thee Fibonacci patterns in the correct order; this technique is called *cross product matching*. The first pattern finds any Fibonacci with a value != -1 and binds both the pattern and the field. The second Fibonacci does this, too, but it adds an additional field constraint to ensure that its sequence is greater by one than the Fibonacci bound to `f1`. When this rule fires for the first time, we know that only sequences 1 and 2 have values of 1, and the two constraints ensure that `f1` references sequence 1 and `f2` references sequence 2. The final pattern finds the Fibonacci with a value equal to -1 and with a sequence one greater than `f2`. At this point, we have three `Fibonacci` objects correctly selected from the available cross products, and we can calculate the value for the third `Fibonacci` object that's bound to `f3`.

Example 23.25. Fibonacci Example: Rule "Calculate"

```
rule Calculate    when        // Bind f1 and s1        f1 : Fibonacci( s1 : sequence, value !=
 -1 )        // Bind f2 and v2; refer to bound variable s1        f2 : Fibonacci( sequence ==
```

```
 (s1 + 1), v2 : value != -1 )         // Bind f3 and s3; alternative reference of f2.sequence
         f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )           then
       // Note the various referencing techniques.         modify ( f3 ) { value = f1.value +
 v2 };        System.out.println( s3 + " == " + f3.value );end
culate
 when        // Bind f1 and
 s1        f1 : Fibonacci( s1 : sequence, value != -1
 )        // Bind f2 and v2; refer to bound variable
 s1        f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1
 )        // Bind f3 and s3; alternative reference of
 f2.sequence        f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )

 then        // Note the various referencing
 techniques.        modify ( f3 ) { value = f1.value + v2
 };        System.out.println( s3 + " == " + f3.value
 );
```

The `modify` statement updated the value of the `Fibonacci` object bound to `f3`. This means we now have another new Fibonacci object with a value not equal to -1, which allows the "Calculate" rule to rematch and calculate the next Fibonacci number. The Audit view below shows how the firing of the last "Bootstrap" modifies the `Fibonacci` object, enabling the "Calculate" rule to match, which then modifies another Fibonacci object allowing the "Calculate" rule to match again. This continues till the value is set for all `Fibonacci` objects.

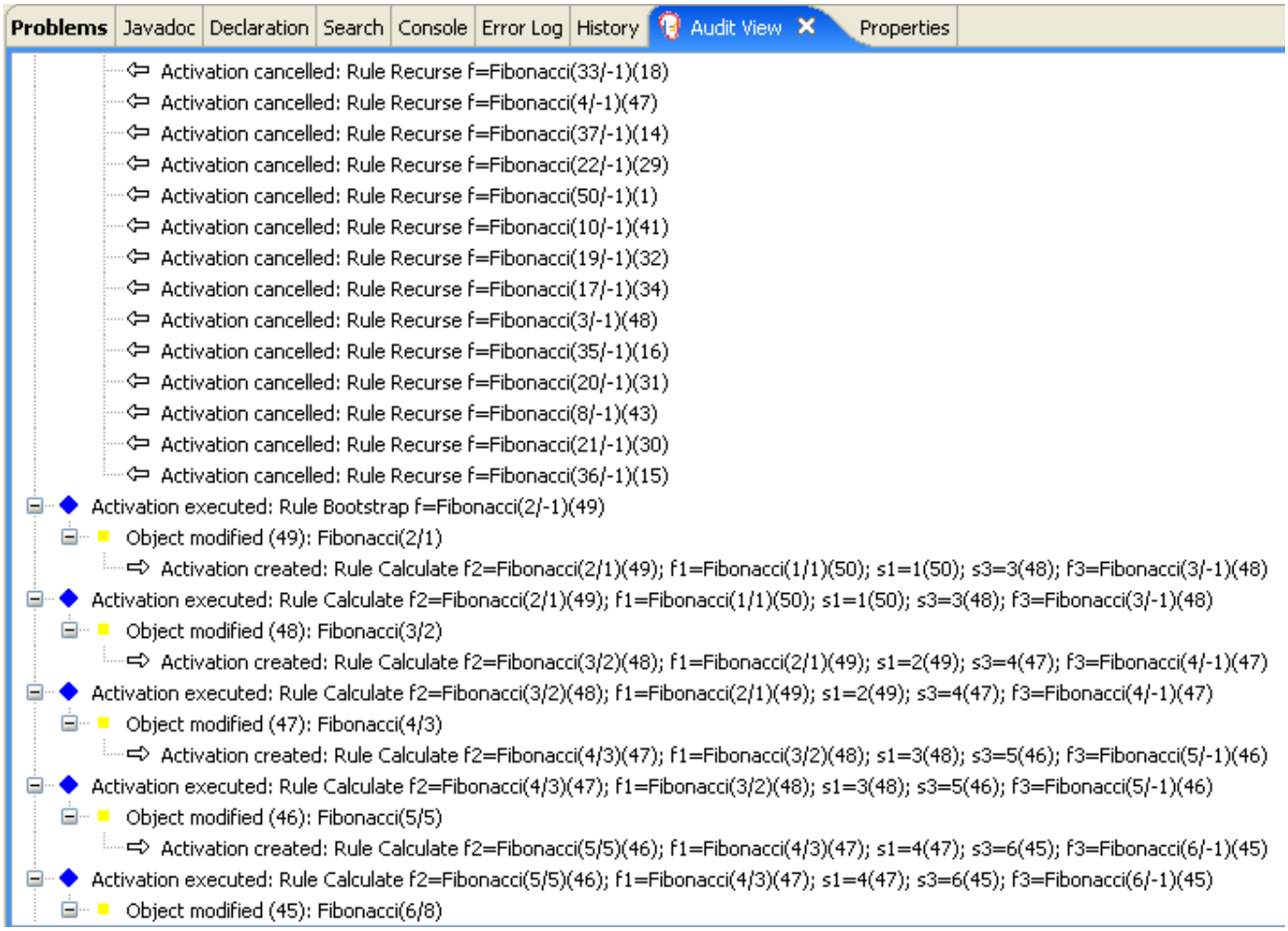


Figure 23.9. Fibonacci Example: "Bootstrap" Audit View

23.5. Banking Tutorial

```
Name: BankingTutorial
Main class: org.drools.examples.banking.BankingExamplesApp.java
Module: drools-examples
Type: Java application
Rules file: org.drools.examples.banking.*.drl
Objective: Demonstrate pattern matching, basic sorting and calculation
 rules.
```

This tutorial demonstrates the process of developing a complete personal banking application to handle credits and debits on multiple accounts. It uses a set of design patterns that have been created for the process.

The class `RuleRunner` is a simple harness to execute one or more DRL files against a set of data. It compiles the Packages and creates the Knowledge Base for each execution, allowing us to easily execute each scenario and inspect the outputs. In reality this is not a good solution for a

production system, where the Knowledge Base should be built just once and cached, but for the purposes of this tutorial it shall suffice.

Example 23.26. Banking Tutorial: RuleRunner

```
public class RuleRunner {

    public RuleRunner() {
    }

    public void runRules(String[] rules,
                         Object[] facts) throws Exception {

        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        for ( int i = 0; i < rules.length; i++ ) {
            String ruleFile = rules[i];
            System.out.println( "Loading file: " + ruleFile );
            kbuilder.add( ResourceFactory.newClassPathResource( ruleFile,
                                                                RuleRunner.class ),
                          ResourceType.DRL );
        }

        Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
        kbase.addKnowledgePackages( pkgs );
        StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();

        for ( int i = 0; i < facts.length; i++ ) {
            Object fact = facts[i];
            System.out.println( "Inserting fact: " + fact );
            ksession.insert( fact );
        }

        ksession.fireAllRules();
    }
}
```

The first of our sample Java classes loads and executes a single DRL file, `Example.drl`, but without inserting any data.

Example 23.27. Banking Tutorial : Java Example1

```
public class Example1 {
    public static void main(String[] args) throws Exception {
        new RuleRunner().runRules( new String[] { "Example1.drl" },
                                   new Object[0] );
    }
}
```

The first simple rule to execute has a single `eval` condition that will always be true, so that this rule will match and fire, once, after the start.

Example 23.28. Banking Tutorial: Rule in Example1.drl

```
rule "Rule 01"
    when
        eval( 1==1 )
    then
        System.out.println( "Rule 01 Works" );
end
```

The output for the rule is below, showing that the rule matches and executes the single print statement.

Example 23.29. Banking Tutorial: Output of Example1.java

```
Loading file: Example1.drlRule 01 Works
Example1.drlRule 01
```

The next step is to assert some simple facts and print them out.

Example 23.30. Banking Tutorial: Java Example2

```
public class Example2 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4), wrap(1), wrap(5)};
        new RuleRunner().runRules( new String[] { "Example2.drl" },
                                   numbers );
    }

    private static Integer wrap( int i ) {
        return new Integer(i);
    }
}
```

This doesn't use any specific facts but instead asserts a set of `java.lang.Integer` objects. This is not considered "best practice" as a number is not a useful fact, but we use it here to demonstrate basic techniques before more complexity is added.

Now we will create a simple rule to print out these numbers.

Example 23.31. Banking Tutorial: Rule in Example2.drl

```
rule "Rule 02"
```

```
    when
        Number( $intValue : intValue )
    then
        System.out.println( "Number found with value: " + $intValue );
end
```

Once again, this rule does nothing special. It identifies any facts that are `Number` objects and prints out the values. Notice the use of the abstract class `Number`: we inserted `Integer` objects but we now look for any kind of number. The pattern matching engine is able to match interfaces and superclasses of asserted objects.

The output shows the DRL being loaded, the facts inserted and then the matched and fired rules. We can see that each inserted number is matched and fired and thus printed.

Example 23.32. Banking Tutorial: Output of Example2.java

```
Loading file: Example2.drlInserting fact: 3Inserting fact: 1Inserting fact: 4Inserting fact:
 1Inserting fact: 5Number found with value: 5Number found with value: 1Number found with value:
 4Number found with value: 1Number found with value: 3
Example2.drlInserting fact:
3Inserting fact:
1Inserting fact:
4Inserting fact:
1Inserting fact:
5Number found with value:
5Number found with value:
1Number found with value:
4Number found with value:
1Number found with value:
```

There are certainly many better ways to sort numbers than using rules, but since we will need to apply some cashflows in date order when we start looking at banking rules we'll develop simple rule based sorting technique.

Example 23.33. Banking Tutorial: Example3.java

```
public class Example3 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4), wrap(1), wrap(5)};
        new RuleRunner().runRules( new String[] { "Example3.drl" },
                                   numbers );
    }

    private static Integer wrap(int i) {
        return new Integer(i);
    }
}
```

Again we insert our `Integer` objects, but this time the rule is slightly different:

Example 23.34. Banking Tutorial: Rule in Example3.drl

```
rule "Rule 03"
    when
        $number : Number( )
        not Number( intValue < $number.intValue )
    then
        System.out.println("Number found with value: " + $number.intValue() );
        retract( $number );
end
```

The first line of the rule identifies a `Number` and extracts the value. The second line ensures that there does not exist a smaller number than the one found by the first pattern. We might expect to match only one number - the smallest in the set. However, the retraction of the number after it has been printed means that the smallest number has been removed, revealing the next smallest number, and so on.

The resulting output shows that the numbers are now sorted numerically.

Example 23.35. Banking Tutorial: Output of Example3.java

```
Loading file: Example3.drlInserting fact: 3Inserting fact: 1Inserting fact: 4Inserting fact:
 1Inserting fact: 5Number found with value: 1Number found with value: 1Number found with value:
 3Number found with value: 4Number found with value: 5
Example3.drlInserting fact:
3Inserting fact:
1Inserting fact:
4Inserting fact:
1Inserting fact:
5Number found with value:
1Number found with value:
1Number found with value:
3Number found with value:
4Number found with value:
```

We are ready to start moving towards our personal accounting rules. The first step is to create a `Cashflow` object.

Example 23.36. Banking Tutorial: Class Cashflow

```
public class Cashflow {
    private Date   date;
    private double amount;

    public Cashflow() {
    }

    public Cashflow(Date date, double amount) {
        this.date = date;
```

```
        this.amount = amount;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public String toString() {
        return "Cashflow[date=" + date + ",amount=" + amount + "]";
    }
}
```

Class `Cashflow` has two simple attributes, a date and an amount. (Note that using the type `double` for monetary units is generally *not* a good idea because floating point numbers cannot represent most numbers accurately.) There is also an overloaded constructor to set the values, and a method `toString` to print a cashflow. The Java code of `Example4.java` inserts five Cashflow objects, with varying dates and amounts.

Example 23.37. Banking Tutorial: Example4.java

```
public class Example4 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new Cashflow(new SimpleDate("01/01/2007"), 300.00),
            new Cashflow(new SimpleDate("05/01/2007"), 100.00),
            new Cashflow(new SimpleDate("11/01/2007"), 500.00),
            new Cashflow(new SimpleDate("07/01/2007"), 800.00),
            new Cashflow(new SimpleDate("02/01/2007"), 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example4.drl" },
                                   cashflows );
    }
}
```

The convenience class `SimpleDate` extends `java.util.Date`, providing a constructor taking a String as input and defining a date format. The code is listed below

Example 23.38. Banking Tutorial: Class SimpleDate

```
public class SimpleDate extends Date {
    private static final SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");

    public SimpleDate(String datestr) throws Exception {
        setTime(format.parse(datestr).getTime());
    }
}
```

Now, let's look at `Example4.drl` to see how we print the sorted `Cashflow` objects:

Example 23.39. Banking Tutorial: Rule in Example4.drl

```
rule "Rule 04"
    when
        $cashflow : Cashflow( $date : date, $amount : amount )
        not Cashflow( date < $date)
    then
        System.out.println("Cashflow: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Here, we identify a `Cashflow` and extract the date and the amount. In the second line of the rule we ensure that there is no Cashflow with an earlier date than the one found. In the consequence, we print the `Cashflow` that satisfies the rule and then retract it, making way for the next earliest `Cashflow`. So, the output we generate is:

Example 23.40. Banking Tutorial: Output of Example4.java

```
Loading    file:    Example4.drlInserting    fact:    Cashflow[date=Mon    Jan    01    00:00:00
     GMT     2007,amount=300.0]Inserting     fact:     Cashflow[date=Fri     Jan     05     00:00:00
     GMT     2007,amount=100.0]Inserting     fact:     Cashflow[date=Thu     Jan     11     00:00:00
     GMT     2007,amount=500.0]Inserting     fact:     Cashflow[date=Sun     Jan     07     00:00:00
    GMT    2007,amount=800.0]Inserting    fact:    Cashflow[date=Tue    Jan    02    00:00:00    GMT
 2007,amount=400.0]Cashflow: Mon Jan 01 00:00:00 GMT 2007 :: 300.0Cashflow: Tue Jan 02 00:00:00
 GMT 2007 :: 400.0Cashflow: Fri Jan 05 00:00:00 GMT 2007 :: 100.0Cashflow: Sun Jan 07 00:00:00
 GMT 2007 :: 800.0Cashflow: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
Example4.drlInserting fact: Cashflow[date=Mon Jan 01 00:00:00 GMT
2007,amount=300.0]Inserting fact: Cashflow[date=Fri Jan 05 00:00:00 GMT
2007,amount=100.0]Inserting fact: Cashflow[date=Thu Jan 11 00:00:00 GMT
2007,amount=500.0]Inserting fact: Cashflow[date=Sun Jan 07 00:00:00 GMT
2007,amount=800.0]Inserting fact: Cashflow[date=Tue Jan 02 00:00:00 GMT
2007,amount=400.0]Cashflow: Mon Jan 01 00:00:00 GMT 2007 ::
300.0Cashflow: Tue Jan 02 00:00:00 GMT 2007 ::
400.0Cashflow: Fri Jan 05 00:00:00 GMT 2007 ::
100.0Cashflow: Sun Jan 07 00:00:00 GMT 2007 ::
800.0Cashflow: Thu Jan 11 00:00:00 GMT 2007 ::
```

Next, we extend our `Cashflow`, resulting in a `TypedCashflow` which can be a credit or a debit operation. (Normally, we would just add this to the `Cashflow` type, but we use extension to keep the previous version of the class intact.)

Example 23.41. Banking Tutorial: Class TypedCashflow

```
public class TypedCashflow extends Cashflow {
    public static final int CREDIT = 0;
    public static final int DEBIT  = 1;

    private int             type;

    public TypedCashflow() {
    }

    public TypedCashflow(Date date, int type, double amount) {
        super( date, amount );
        this.type = type;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public String toString() {
        return "TypedCashflow[date=" + getDate() +
               ",type=" + (type == CREDIT ? "Credit" : "Debit") +
               ",amount=" + getAmount() + "]";
    }
}
```

There are lots of ways to improve this code, but for the sake of the example this will do.

Now let's create Example5, a class for running our code.

Example 23.42. Banking Tutorial: Example5.java

```
public class Example5 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new TypedCashflow(new SimpleDate("01/01/2007"),
                              TypedCashflow.CREDIT, 300.00),
            new TypedCashflow(new SimpleDate("05/01/2007"),
                              TypedCashflow.CREDIT, 100.00),
            new TypedCashflow(new SimpleDate("11/01/2007"),
                              TypedCashflow.CREDIT, 500.00),
            new TypedCashflow(new SimpleDate("07/01/2007"),
                              TypedCashflow.DEBIT, 800.00),
            new TypedCashflow(new SimpleDate("02/01/2007"),
```

```
                                TypedCashflow.DEBIT, 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example5.drl" },
                                   cashflows );
    }
}
```

Here, we simply create a set of `Cashflow` objects which are either credit or debit operations. We supply them and `Example5.drl` to the RuleEngine.

Now, let's look at a rule printing the sorted `Cashflow` objects.

Example 23.43. Banking Tutorial: Rule in Example5.drl

```
rule "Rule 05"
    when
        $cashflow : TypedCashflow( $date : date,
                                   $amount : amount,
                                   type == TypedCashflow.CREDIT )
        not TypedCashflow( date < $date,
                           type == TypedCashflow.CREDIT )
    then
        System.out.println("Credit: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Here, we identify a `Cashflow` fact with a type of `CREDIT` and extract the date and the amount. In the second line of the rule we ensure that there is no `Cashflow` of the same type with an earlier date than the one found. In the consequence, we print the cashflow satisfying the patterns and then retract it, making way for the next earliest cashflow of type `CREDIT`.

So, the output we generate is

Example 23.44. Banking Tutorial: Output of Example5.java

```
Loading file: Example5.drl
Inserting fact: TypedCashflow[date=Mon Jan 01 00:00:00 GMT
 2007,type=Credit,amount=300.0]
Inserting fact: TypedCashflow[date=Fri Jan 05 00:00:00 GMT
 2007,type=Credit,amount=100.0]
Inserting fact: TypedCashflow[date=Thu Jan 11 00:00:00 GMT
 2007,type=Credit,amount=500.0]
Inserting fact: TypedCashflow[date=Sun Jan 07 00:00:00 GMT
 2007,type=Debit,amount=800.0]
Inserting fact: TypedCashflow[date=Tue Jan 02 00:00:00 GMT
 2007,type=Debit,amount=400.0]
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Credit: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
```

```
  Credit: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

Continuing our banking exercise, we are now going to process both credits and debits on two bank accounts, calculating the account balance. In order to do this, we create two separate `Account` objects and inject them into the `Cashflows` objects before passing them to the Rule Engine. The reason for this is to provide easy access to the correct account without having to resort to helper classes. Let's take a look at the `Account` class first. This is a simple Java object with an account number and balance:

Example 23.45. Banking Tutorial: Class Account

```
public class Account {
    private long   accountNo;
    private double balance = 0;

    public Account() {
    }

    public Account(long accountNo) {
        this.accountNo = accountNo;
    }

    public long getAccountNo() {
        return accountNo;
    }

    public void setAccountNo(long accountNo) {
        this.accountNo = accountNo;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    public String toString() {
        return "Account[" + "accountNo=" + accountNo + ",balance=" + balance + "]";
    }
}
```

Now let's extend our `TypedCashflow`, resulting in `AllocatedCashflow`, to include an `Account` reference.

Example 23.46. Banking Tutorial: Class AllocatedCashflow

```
public class AllocatedCashflow extends TypedCashflow {
    private Account account;
```

```
    public AllocatedCashflow() {
    }

    public AllocatedCashflow(Account account, Date date, int type, double amount) {
        super( date, type, amount );
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }

    public void setAccount(Account account) {
        this.account = account;
    }

    public String toString() {
        return "AllocatedCashflow[" +
               "account=" + account +
               ",date=" + getDate() +
               ",type=" + (getType() == CREDIT ? "Credit" : "Debit") +
               ",amount=" + getAmount() + "]";
    }
}
```

The Java code of `Example5.java` creates two `Account` objects and passes one of them into each cashflow, in the constructor call.

Example 23.47. Banking Tutorial: Example5.java

```
public class Example6 {
    public static void main(String[] args) throws Exception {
        Account acc1 = new Account(1);
        Account acc2 = new Account(2);

        Object[] cashflows = {
            new AllocatedCashflow(acc1,new SimpleDate("01/01/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/02/2007"),
                                  TypedCashflow.CREDIT, 100.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/03/2007"),
                                  TypedCashflow.CREDIT, 500.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/02/2007"),
                                  TypedCashflow.DEBIT,  800.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/03/2007"),
                                  TypedCashflow.DEBIT,  400.00),
            new AllocatedCashflow(acc1,new SimpleDate("01/04/2007"),
                                  TypedCashflow.CREDIT, 200.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/04/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/05/2007"),
                                  TypedCashflow.CREDIT, 700.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/05/2007"),
                                  TypedCashflow.DEBIT,  900.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/05/2007"),
                                  TypedCashflow.DEBIT,  100.00)
```

```
        };

        new RuleRunner().runRules( new String[] { "Example6.drl" },
                                   cashflows );
    }
}
```

Now, let's look at the rule in `Example6.drl` to see how we apply each cashflow in date order and calculate and print the balance.

Example 23.48. Banking Tutorial: Rule in Example6.drl

```
rule "Rule 06 - Credit"      when        $cashflow : AllocatedCashflow( $account : account,
                                        $date : date,
 $amount : amount,                                    type == TypedCashflow.CREDIT )       not
 AllocatedCashflow( account == $account, date < $date)    then       System.out.println("Credit:
 " + $date + " :: " + $amount);            $account.setBalance($account.getBalance()+$amount);
        System.out.println("Account: " + $account.getAccountNo() +                              "
 - new balance: " + $account.getBalance());                   retract($cashflow);endrule "Rule
 06 - Debit"      when        $cashflow : AllocatedCashflow( $account : account,
              $date : date,                              $amount : amount,
       type == TypedCashflow.DEBIT )        not AllocatedCashflow( account == $account, date <
 $date)    then        System.out.println("Debit: " + $date + " :: " + $amount);
 $account.setBalance($account.getBalance() - $amount);        System.out.println("Account: " +
 $account.getAccountNo() +                          " - new balance: " + $account.getBalance());
                  retract($cashflow);end

when        $cashflow : AllocatedCashflow( $account :
account,                                      $date :
date,                                      $amount :
amount,                                      type == TypedCashflow.CREDIT
)        not AllocatedCashflow( account == $account, date <
$date)
then        System.out.println("Credit: " + $date + " :: " + $amount);
         $account.setBalance($account.getBalance()+
$amount);        System.out.println("Account: " + $account.getAccountNo()
+                           " - new balance: " + $account.getBalance());

retract($cashflow);

endrule "Rule 06 - Debit"

when        $cashflow : AllocatedCashflow( $account :
account,                             $date :
date,                            $amount :
amount,                            type == TypedCashflow.DEBIT
)        not AllocatedCashflow( account == $account, date <
$date)
then        System.out.println("Debit: " + $date + " :: " + $amount);
         $account.setBalance($account.getBalance() -
$amount);        System.out.println("Account: " + $account.getAccountNo()
+                           " - new balance: " + $account.getBalance());

retract($cashflow);
```

Although we have separate rules for credits and debits, but we do not specify a type when checking for earlier cashflows. This is so that all cashflows are applied in date order, regardless of the cashflow type. In the conditions we identify the account to work with, and in the consequences we update it with the cashflow amount.

Example 23.49. Banking Tutorial: Output of Example6.java

```
Loading                  file:                  Example6.drlInserting                  fact:
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon   Jan   01   00:00:00   GMT
                                                 2007,type=Credit,amount=300.0]Inserting
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon   Feb   05   00:00:00   GMT
                                                 2007,type=Credit,amount=100.0]Inserting
   AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Sun   Mar   11   00:00:00   GMT
                                                 2007,type=Credit,amount=500.0]Inserting
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Wed   Feb   07   00:00:00   GMT
                                                  2007,type=Debit,amount=800.0]Inserting
   AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri   Mar   02   00:00:00   GMT
                                                  2007,type=Debit,amount=400.0]Inserting
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Sun   Apr   01   00:00:00   BST
                                                 2007,type=Credit,amount=200.0]Inserting
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Thu   Apr   05   00:00:00   BST
                                                 2007,type=Credit,amount=300.0]Inserting
   AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri   May   11   00:00:00   BST
                                                 2007,type=Credit,amount=700.0]Inserting
   AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon   May   07   00:00:00   BST
                                                  2007,type=Debit,amount=900.0]Inserting
   AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Wed   May   02   00:00:00   BST
  2007,type=Debit,amount=100.0]Debit:  Fri  Mar  02  00:00:00  GMT  2007  ::  400.0Account:  2  -  new
 balance: -400.0Credit: Sun Mar 11 00:00:00 GMT 2007 :: 500.0Account: 2 - new balance: 100.0Debit:
 Wed May 02 00:00:00 BST 2007 :: 100.0Account: 2 - new balance: 0.0Credit: Fri May 11 00:00:00 BST
 2007 :: 700.0Account: 2 - new balance: 700.0Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0Account:
  1 - new balance: 300.0Credit: Mon Feb 05 00:00:00 GMT 2007 :: 100.0Account: 1 - new balance:
  400.0Debit: Wed Feb 07 00:00:00 GMT 2007 :: 800.0Account: 1 - new balance: -400.0Credit: Sun
 Apr 01 00:00:00 BST 2007 :: 200.0Account: 1 - new balance: -200.0Credit: Thu Apr 05 00:00:00 BST
 2007 :: 300.0Account: 1 - new balance: 100.0Debit: Mon May 07 00:00:00 BST 2007 :: 900.0Account:
 1 - new balance: -800.0
Example6.drlInserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon  Jan
01 00:00:00 GMT
2007,type=Credit,amount=300.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=M
05 00:00:00 GMT
2007,type=Credit,amount=100.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=S
11 00:00:00 GMT
2007,type=Credit,amount=500.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=W
07 00:00:00 GMT
2007,type=Debit,amount=800.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fr
02 00:00:00 GMT
2007,type=Debit,amount=400.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Su
01 00:00:00 BST
2007,type=Credit,amount=200.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=T
05 00:00:00 BST
2007,type=Credit,amount=300.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=F
11 00:00:00 BST
2007,type=Credit,amount=700.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=M
07 00:00:00 BST
2007,type=Debit,amount=900.0]Inserting  fact:  AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=We
02 00:00:00 BST
```

```
2007,type=Debit,amount=100.0]Debit: Fri Mar 02 00:00:00 GMT 2007 ::
400.0Account: 2 - new balance:
-400.0Credit: Sun Mar 11 00:00:00 GMT 2007 ::
500.0Account: 2 - new balance:
100.0Debit: Wed May 02 00:00:00 BST 2007 ::
100.0Account: 2 - new balance:
0.0Credit: Fri May 11 00:00:00 BST 2007 ::
700.0Account: 2 - new balance:
700.0Credit: Mon Jan 01 00:00:00 GMT 2007 ::
300.0Account: 1 - new balance:
300.0Credit: Mon Feb 05 00:00:00 GMT 2007 ::
100.0Account: 1 - new balance:
400.0Debit: Wed Feb 07 00:00:00 GMT 2007 ::
800.0Account: 1 - new balance:
-400.0Credit: Sun Apr 01 00:00:00 BST 2007 ::
200.0Account: 1 - new balance:
-200.0Credit: Thu Apr 05 00:00:00 BST 2007 ::
300.0Account: 1 - new balance:
100.0Debit: Mon May 07 00:00:00 BST 2007 ::
900.0Account: 1 - new balance:
```

23.6. Pricing Rule Decision Table Example

The Pricing Rule decision table demonstrates the use of a decision table in a spreadsheet, in Excel's XLS format, in calculating the retail cost of an insurance policy. The purpose of the provide set of rules is to calculate a base price and a discount for a car driver applying for a specific policy. The driver's age, history and the policy type all contribute to what the basic premium is, and an additional chunk of rules deals with refining this with a discount percentage.

```
Name: Example Policy Pricing
Main class: org.drools.examples.decisiontable.PricingRuleDTExample
Module: drools-examples
Type: Java application
Rules file: ExamplePolicyPricing.xls
Objective: demonstrate spreadsheet-based decision tables.
```

23.6.1. Executing the example

Open the file `PricingRuleDTExample.java` and execute it as a Java application. It should produce the following output in the Console window:

```
Cheapest possibleBASE PRICE IS: 120DISCOUNT IS: 20
possibleBASE PRICE
IS: 120DISCOUNT IS: 20
```

The code to execute the example follows the usual pattern. The rules are loaded, the facts inserted and a Stateless Session is created. What is different is how the rules are added.

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
        dtableconfiguration.setInputType( DecisionTableInputType.XLS );

        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls",
                                                                getClass() );
        kbuilder.add( xlsRes,
                      ResourceType.DTABLE,
                      dtableconfiguration );
```

Note the use of the `DecisionTableConfiguration` object. Its input type is set to `DecisionTableInputType.XLS`. If you use the BRMS, all this is of course taken care of for you.

There are two fact types used in this example, `Driver` and `Policy`. Both are used with their default values. The `Driver` is 30 years old, has had no prior claims and currently has a risk profile of `LOW`. The `Policy` being applied for is `COMPREHENSIVE`, and it has not yet been approved.

23.6.2. The decision table

In this decision table, each row is a rule, and each column is a condition or an action.

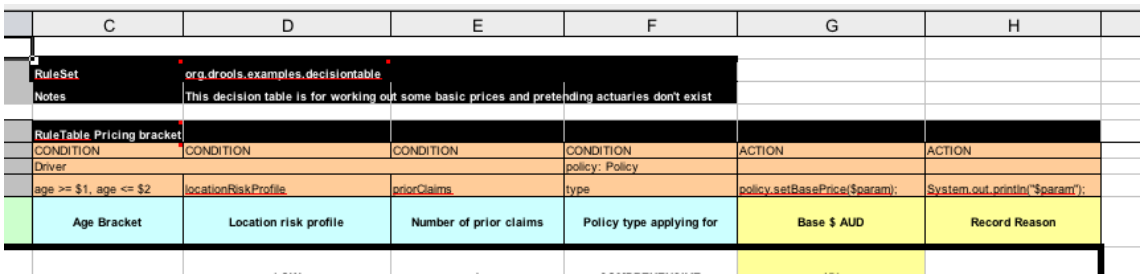
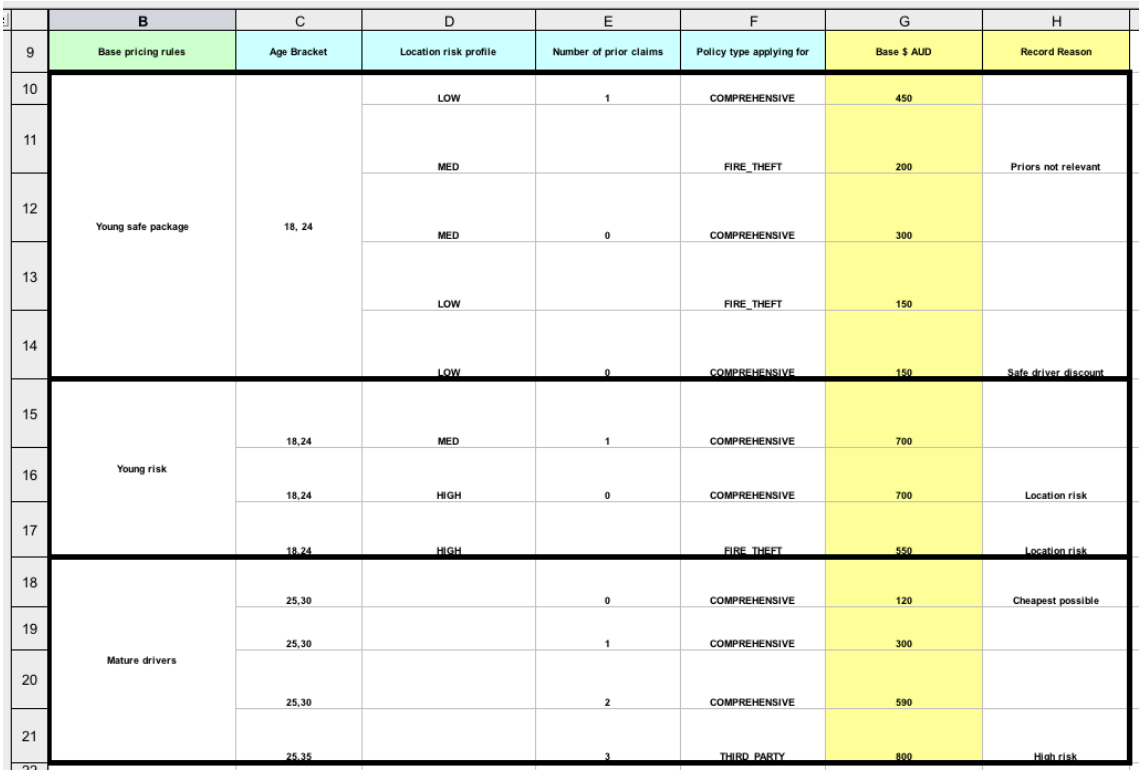


Figure 23.10. Decision table configuration

Referring to the spreadsheet show above, we have the `RuleSet` declaration, which provides the package name. There are also other optional items you can have here, such as `Variables` for global variables, and `Imports` for importing classes. In this case, the namespace of the rules is the same as the fact classes we are using, so we can omit it.

Moving further down, we can see the `RuleTable` declaration. The name after this (Pricing bracket) is used as the prefix for all the generated rules. Below that, we have "CONDITION or ACTION", indicating the purpose of the column, i.e., whether it forms part of the condition or the consequence of the rule that will be generated.

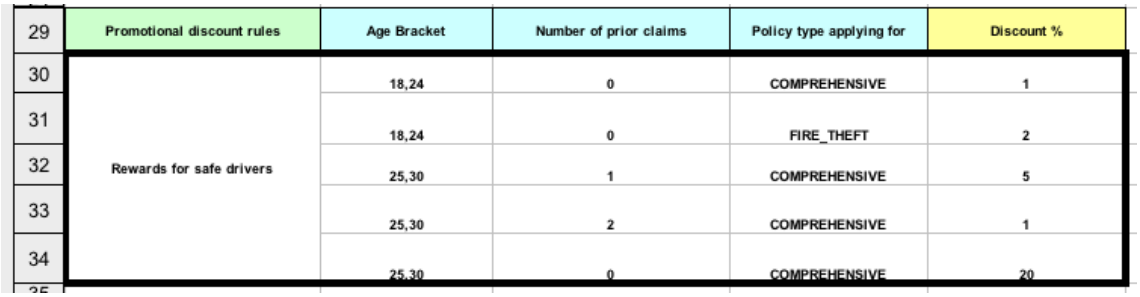
You can see that there is a driver, his data spanned across three cells, which means that the template expressions below it apply to that fact. We observe the driver's age range (which uses `$1` and `$2` with comma-separated values), `locationRiskProfile`, and `priorClaims` in the respective columns. In the action columns, we are set the policy base price and log a message.



	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason
10	Young safe package	18, 24	LOW	1	COMPREHENSIVE	450	
11			MED		FIRE_THEFT	200	Priors not relevant
12			MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	Safe driver discount
15	Young risk	18,24	MED	1	COMPREHENSIVE	700	
16		18,24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18,24	HIGH		FIRE_THEFT	550	Location risk
18	Mature drivers	25,30		0	COMPREHENSIVE	120	Cheapest possible
19		25,30		1	COMPREHENSIVE	300	
20		25,30		2	COMPREHENSIVE	590	
21		25,35		3	THIRD_PARTY	800	High risk

Figure 23.11. Base price calculation

In the preceding spreadsheet section, there are broad category brackets, indicated by the comment in the leftmost column. As we know the details of our drivers and their policies, we can tell (with a bit of thought) that they should match row number 18, as they have no prior accidents, and are 30 years old. This gives us a base price of 120.



29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32		25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20

Figure 23.12. Discount calculation

The above section contains the conditions for the discount we might grant our driver. The discount results from the `Age` bracket, the number of prior claims, and the policy type. In our case, the driver is 30, with no prior claims, and is applying for a `COMPREHENSIVE` policy, which means we can give a discount of 20%. Note that this is actually a separate table, but in the same worksheet, so that different templates apply.

It is important to note that decision tables generate rules. This means they aren't simply top-down logic, but more a means to capture data resulting in rules. This is a subtle difference that confuses

some people. The evaluation of the rules is not necessarily in the given order, since all the normal mechanics of the rule engine still apply.

23.7. Pet Store Example

```
Name: Pet Store
Main class: org.drools.examples.petstore.PetStoreExample
Module: drools-examples
Type: Java application
Rules file: PetStore.drl
Objective: Demonstrate use of Agenda Groups, Global Variables and integration with a GUI,
including callbacks from within the rules
```

The Pet Store example shows how to integrate Rules with a GUI, in this case a Swing based desktop application. Within the rules file, it demonstrates how to use Agenda groups and auto-focus to control which of a set of rules is allowed to fire at any given time. It also illustrates the mixing of the Java and MVEL dialects within the rules, the use of accumulate functions and the way of calling Java functions from within the ruleset.

All of the Java code is contained in one file, `PetStore.java`, defining the following principal classes (in addition to several classes to handle Swing Events):

- `Petstore` contains the `main()` method that we will look at shortly.
- `PetStoreUI` is responsible for creating and displaying the Swing based GUI. It contains several smaller classes, mainly for responding to various GUI events such as mouse button clicks.
- `TableModel` holds the table data. Think of it as a JavaBean that extends the Swing class `AbstractTableModel`.
- `CheckoutCallback` allows the GUI to interact with the Rules.
- `Ordershow` keeps the items that we wish to buy.
- `Purchase` stores details of the order and the products we are buying.
- `Product` is a JavaBean holding details of the product available for purchase, and its price.

Much of the Java code is either plain JavaBeans or Swing-based. Only a few Swing-related points will be discussed in this section, but a good tutorial about Swing components can be found at Sun's Swing website, in http://java.sun.com/docs/books/tutorial/uiswing/.

The pieces of Java code in `Petstore.java` that relate to rules and facts are shown below.

Example 23.50. Creating the PetStore KieContainer in PetStore.main

```
// KieServices is the factory for all KIE services
KieServices ks = KieServices.Factory.get();
```

```
// From the kie services, a container is created from the classpath
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );

// A callback is responsible for populating the
// Working Memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kc ) );
ui.createAndShowGUI();
```

The code shown above create a `KieContainer` from the classpath and based on the definitions in the `kmodule.xml` file. Unlike other examples where the facts are asserted and fired straight away, this example defers this step to later. The way it does this is via the second last line where a `PetStoreUI` object is created using a constructor accepting the `Vector` object `stock` collecting our products, and an instance of the `CheckoutCallback` class containing the Rule Base that we have just loaded.

The Java code that fires the rules is within the `CheckoutCallBack.checkout()` method. This is triggered (eventually) when the Checkout button is pressed by the user.

Example 23.51. Firing the Rules - extract from CheckoutCallBack.checkout()

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user interaction

    // From the container, a session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );
    ksession.fireAllRules();

    // Return the state of the cart
```

```
    return order.toString();
}
```

Two items get passed into this method. One is the handle to the `JFrame` Swing component surrounding the output text frame, at the bottom of the GUI. The second is a list of order items; this comes from the `TableModel` storing the information from the "Table" area at the top right section of the GUI.

The for loop transforms the list of order items coming from the GUI into the `Order` JavaBean, also contained in the file `PetStore.java`. Note that it would be possible to refer to the Swing dataset directly within the rules, but it is better coding practice to do it this way, using simple Java objects. It means that we are not tied to Swing if we wanted to transform the sample into a Web application.

It is important to note that *all state in this example is stored in the Swing components, and that the rules are effectively stateless.* Each time the "Checkout" button is pressed, this code copies the contents of the Swing `TableModel` into the Session's Working Memory.

Within this code, there are nine calls to the `KieSession`. The first of these creates a new `KieSession` from the `KieContainer`. Remember that we passed in this `KieContainer` when we created the `CheckoutCallBack` class in the `main()` method. The next two calls pass in two objects that we will hold as global variables in the rules: the Swing text area and the Swing frame used for writing messages.

More inserts put information on products into the `KieSession`, as well as the order list. The final call is the standard `fireAllRules()`. Next, we look at what this method causes to happen within the rules file.

Example 23.52. Package, Imports, Globals and Dialect: extract from PetStore.drl

```
package org.drools.examples

import org.kie.api.runtime.KieRuntime
import org.drools.examples.petstore.PetStoreExample.Order
import org.drools.examples.petstore.PetStoreExample.Purchase
import org.drools.examples.petstore.PetStoreExample.Product
import java.util.ArrayList
import javax.swing.JOptionPane;

import javax.swing.JFrame

global JFrame frame
global javax.swing.JTextArea textArea
```

The first part of file `PetStore.drl` contains the standard package and import statements to make various Java classes available to the rules. New to us are the two globals `frame` and `textArea`. They hold references to the Swing components `JFrame` and `JTextArea` components that were previously passed on by the Java code calling the `setGlobal()` method. Unlike variables in rules,

which expire as soon as the rule has fired, global variables retain their value for the lifetime of the Session.

The next extract from the file `PetStore.drl` contains two functions that are referenced by the rules that we will look at shortly.

Example 23.53. Java Functions in the Rules: extract from PetStore.drl

```
function void doCheckout(JFrame frame, KieRuntime krt) {
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                             "Would you like to checkout?",
                                             "",
                                             JOptionPane.YES_NO_OPTION,
                                             JOptionPane.QUESTION_MESSAGE,
                                             null,
                                             options,
                                             options[0]);

       if (n == 0) {
            krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
       }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total) {
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                    "Would you like to buy a tank for your " + total + " fish?",
                                             "Purchase Suggestion",
                                             JOptionPane.YES_NO_OPTION,
                                             JOptionPane.QUESTION_MESSAGE,
                                             null,
                                             options,
                                             options[0]);

       System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                           + total + " fish? - " );

       if (n == 0) {
             Purchase purchase = new Purchase( order, fishTank );
             krt.insert( purchase );
             order.addItem( purchase );
             System.out.println( "Yes" );
       } else {
            System.out.println( "No" );
       }
       return true;
}
```

Having these functions in the rules file just makes the Pet Store example more compact. In real life you probably have the functions in a file of their own, within the same rules package,

or as a static method on a standard Java class, and import them, using `import function my.package.Foo.hello`.

The purpose of these two functions is:

- `doCheckout()` displays a dialog asking users whether they wish to checkout. If they do, focus is set to the `checkout` agenda-group, allowing rules in that group to (potentially) fire.
- `requireTank()` displays a dialog asking users whether they wish to buy a tank. If so, a new fish tank `Product` is added to the order list in Working Memory.

We'll see the rules that call these functions later on. The next set of examples are from the Pet Store rules themselves. The first extract is the one that happens to fire first, partly because it has the `auto-focus` attribute set to true.

Example 23.54. Putting items into working memory: extract from PetStore.drl

```
// Insert each item in the shopping cart into the Working Memory // Insert each item in
  the shopping cart into the Working Memoryrule "Explode Cart"      agenda-group "init"
    auto-focus true      salience 10      dialect "java"when      $order : Order( grossTotal
  ==  -1  )        $item  :  Purchase()  from  $order.itemsthen        insert(  $item  );
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup(  "show  items"  ).setFocus();
 kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();end
 Memory // Insert each item in the shopping cart into
the Working Memory
rule "Explode Cart"
 agenda-group "init"
 auto-focus true
  salience 10

 dialect "java"when    $order : Order( grossTotal
== -1 )    $item :
Purchase()
 from $order.itemsthen
insert( $item );    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup(
"show items" ).setFocus();
kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup(
```

This rule matches against all orders that do not yet have their `grossTotal` calculated . It loops for each purchase item in that order. Some parts of the "Explode Cart" rule should be familiar: the rule name, the salience (suggesting the order for the rules being fired) and the dialect set to `"java"`. There are three new features:

- `agenda-group "init"` defines the name of the agenda group. In this case, there is only one rule in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the next attribute for its chance to fire.
- `auto-focus true` ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when `fireAllRules()` is called from the Java code.

- `kcontext....setFocus()` sets the focus to the `"show items"` and `"evaluate"` agenda groups in turn, permitting their rules to fire. In practice, we loop through all items on the order, inserting them into memory, then firing the other rules after each insert.

The next two listings show the rules within the `"show items"` and `evaluate` agenda groups. We look at them in the order that they are called.

Example 23.55. Show Items in the GUI - extract from PetStore.drl

```
rule "Show Items"
    agenda-group "show items"
    dialect "mvel"
when
    $order : Order( )
    $p : Purchase( order == $order )
then
   textArea.append( $p.product + "\n");
end
```

The `"show items"` agenda-group has only one rule, called "Show Items" (note the difference in case). For each purchase on the order currently in the Working Memory (or Session), it logs details to the text area at the bottom of the GUI. The `textArea` variable used to do this is one of the global variables we looked at earlier.

The `evaluate` Agenda group also gains focus from the `"Explode Cart"` rule listed previously. This Agenda group has two rules, `"Free Fish Food Sample"` and `"Suggest Tank"`, shown below.

Example 23.56. Evaluate Agenda Group: extract from PetStore.drl

```
// Free Fish Food sample when we buy a Gold Fish if we haven't already bought // Fish Food and
 don't already have a Fish Food Samplerule "Free Fish Food Sample"    agenda-group "evaluate"
     dialect "mvel"when     $order : Order()     not ( $p : Product( name == "Fish Food")
 &amp;amp;&amp;amp; Purchase( product == $p ) )     not ( $p : Product( name == "Fish Food Sample")
 &amp;amp;&amp;amp; Purchase( product == $p ) )    exists ( $p : Product( name == "Gold Fish") &amp;amp;&amp;
 Purchase( product == $p ) )    $fishFoodSample : Product( name == "Fish Food Sample" );then
 System.out.println( "Adding free Fish Food Sample to cart" );    purchase = new Purchase($order,
 $fishFoodSample);    insert( purchase );    $order.addItem( purchase ); end// Suggest a tank
 if we have bought more than 5 gold fish and don't already have onerule "Suggest Tank"
 agenda-group "evaluate"    dialect "java"when    $order : Order()    not ( $p : Product( name
 == "Fish Tank") &amp;amp;&amp;amp; Purchase( product == $p ) )     ArrayList( $total : size &amp;gt; 5 )
 from collect( Purchase( product.name == "Gold Fish" ) )    $fishTank : Product( name == "Fish
 Tank" )then    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total); end
bought // Fish Food and don't already have a Fish Food
 Samplerule "Free Fish Food
 Sample"    agenda-group
 "evaluate"    dialect

"mvel"when    $order :
 Order()    not ( $p : Product( name == "Fish Food") &amp;amp;&amp;amp; Purchase( product == $p )
 )    not ( $p : Product( name == "Fish Food Sample") &amp;amp;&amp;amp; Purchase( product == $p )
 )    exists ( $p : Product( name == "Gold Fish") &amp;amp;&amp;amp; Purchase( product == $p )
 )    $fishFoodSample : Product( name == "Fish Food Sample"
```

```
);then    System.out.println( "Adding free Fish Food Sample to cart"
 );    purchase = new Purchase($order,
 $fishFoodSample);    insert( purchase
 );    $order.addItem( purchase
);

 end// Suggest a tank if we have bought more than 5 gold fish and don't already have
 onerule "Suggest
 Tank"    agenda-group
 "evaluate"    dialect

"java"when    $order :
 Order()    not ( $p : Product( name == "Fish Tank") &amp;amp;&amp;amp; Purchase( product == $p )
 )    ArrayList( $total : size &gt; 5 ) from collect( Purchase( product.name == "Gold Fish" )
 )    $fishTank : Product( name == "Fish Tank"

)then    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank,
$total);
```

The rule `"Free Fish Food Sample"` will only fire if

- we *don't* already have any fish food, *and*
- we *don't* already have a free fish food sample, *and*
- we *do* have a Gold Fish in our order.

If the rule does fire, it creates a new product (Fish Food Sample), and adds it to the order in Working Memory.

The rule `"Suggest Tank"` will only fire if

- we *don't* already have a Fish Tank in our order, *and*
- we *do* have more than 5 Gold Fish Products in our order.

If the rule does fire, it calls the `requireTank()` function that we looked at earlier (showing a Dialog to the user, and adding a Tank to the order / working memory if confirmed). When calling the *requireTank*() function the rule passes the global *frame* variable so that the function has a handle to the Swing GUI.

The next rule we look at is `"do checkout"`.

Example 23.57. Doing the Checkout - extract (6) from PetStore.drl

```
rule  "do  checkout"       dialect  "java"       when       then             doCheckout(frame,
 kcontext.getKieRuntime());end
out"    dialect
"java"
when
then        doCheckout(frame,
kcontext.getKieRuntime());
```

The rule `"do checkout"` has **no agenda group set and no auto-focus attribute**. As such, is is deemed part of the default (MAIN) agenda group. This group gets focus by default when all the rules in agenda-groups that explicitly had focus set to them have run their course.

There is no LHS to the rule, so the RHS will always call the `doCheckout()` function. When calling the `doCheckout()` function, the rule passes the global `frame` variable to give the function a handle to the Swing GUI. As we saw earlier, the `doCheckout()` function shows a confirmation dialog to the user. If confirmed, the function sets the focus to the *checkout* agenda-group, allowing the next lot of rules to fire.

Example 23.58. Checkout Rules: extract from PetStore.drl

```
rule "Gross Total"    agenda-group "checkout"    dialect "mvel"when   $order : Order( grossTotal
 == -1)   Number( total : doubleValue )      from accumulate( Purchase( $price : product.price ),
 sum( $price ) )then    modify( $order ) { grossTotal = total };    textArea.append( "\ngross
 total=" + total + "\n" );endrule "Apply 5% Discount"   agenda-group "checkout"dialect "mvel"when
    $order : Order( grossTotal &gt;= 10 &amp;&amp; &lt; 20 )then    $order.discountedTotal =
 $order.grossTotal * 0.95;   textArea.append( "discountedTotal total=" + $order.discountedTotal
 + "\n" );endrule "Apply 10% Discount"     agenda-group "checkout"     dialect "mvel"when
 $order : Order( grossTotal &gt;= 20 )then   $order.discountedTotal = $order.grossTotal * 0.90;
   textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );end
tal"    agenda-group
"checkout"    dialect
"mvel"
when   $order : Order( grossTotal ==
-1)   Number( total : doubleValue
)       from accumulate( Purchase( $price : product.price ), sum( $price )
)
then    modify( $order ) { grossTotal = total
};    textArea.append( "\ngross total=" + total + "\n"
);

endrule "Apply 5%
Discount"    agenda-group
"checkout"dialect
"mvel"
when   $order : Order( grossTotal &gt;= 10 &amp;&amp; &lt; 20
)
then   $order.discountedTotal = $order.grossTotal *
0.95;   textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n"
);


endrule "Apply 10%
Discount"    agenda-group
"checkout"    dialect
"mvel"
when   $order : Order( grossTotal &gt;= 20
)
then   $order.discountedTotal = $order.grossTotal *
0.90;   textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n"
);
```

There are three rules in the *checkout* agenda-group:

- If we haven't already calculated the gross total, `Gross Total` accumulates the product prices into a total, puts this total into the session, and displays it via the Swing `JTextArea`, using the `textArea` global variable yet again.
- If our gross total is between 10 and 20, `"Apply 5% Discount"` calculates the discounted total and adds it to the session and displays it in the text area.
- If our gross total is not less than 20, `"Apply 10% Discount"` calculates the discounted total and adds it to the session and displays it in the text area.

Now that we've run through what happens in the code, let's have a look at what happens when we actually run the code. The file `PetStore.java` contains a `main()` method, so that it can be run as a standard Java application, either from the command line or via the IDE. This assumes you have your classpath set correctly. (See the start of the examples section for more information.)

The first screen that we see is the Pet Store Demo. It has a list of available products (top left), an empty list of selected products (top right), checkout and reset buttons (middle) and an empty system messages area (bottom).

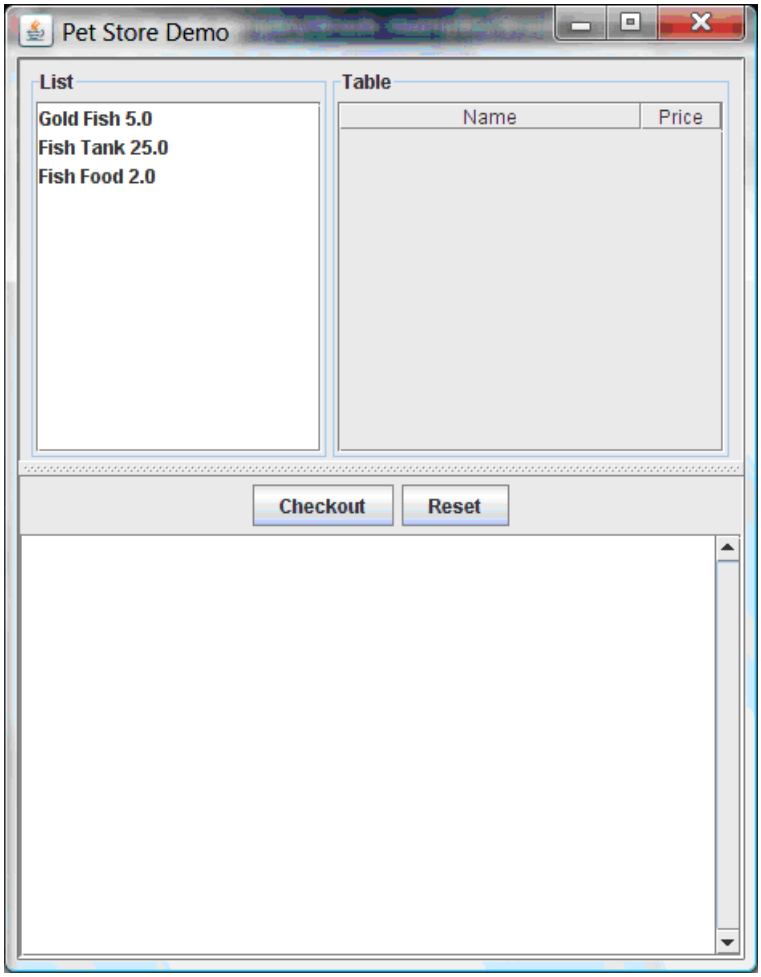


Figure 23.13. PetStore Demo just after Launch

To get to this point, the following things have happened:

1. The `main()` method has run and loaded the Rule Base *but not yet fired the rules*. So far, this is the only code in connection with rules that has been run.

2. A new `PetStoreUI` object has been created and given a handle to the Rule Base, for later use.

3. Various Swing components do their stuff, and the above screen is shown and *waits for user input*.

Clicking on various products from the list might give you a screen similar to the one below.

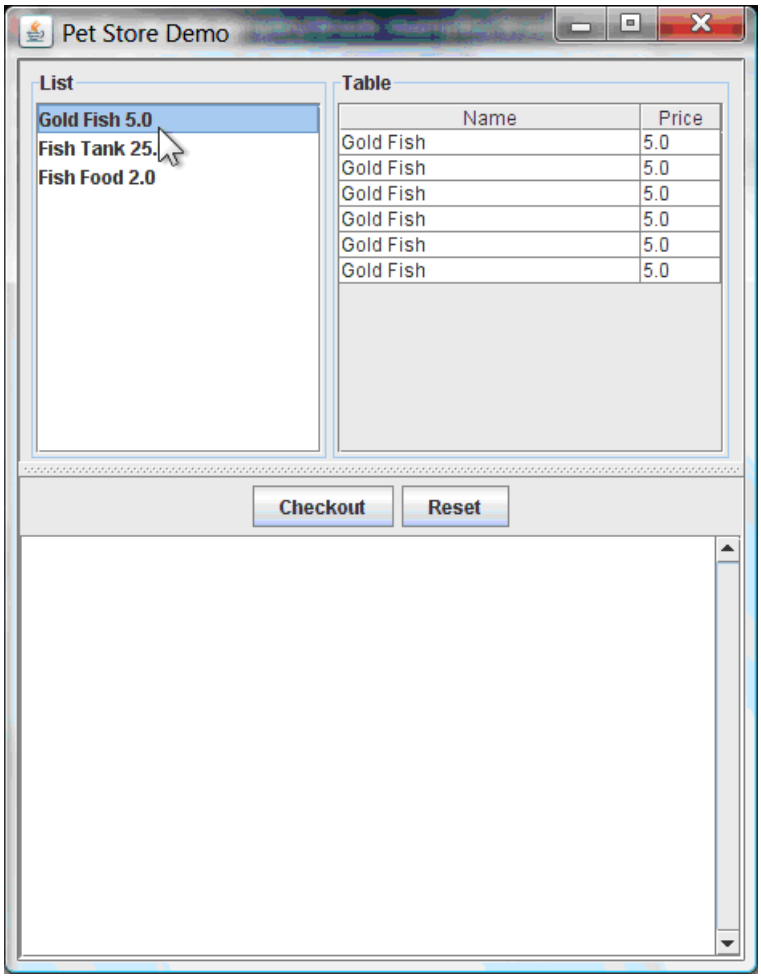


Figure 23.14. PetStore Demo with Products Selected

Note that *no rules code has been fired here*. This is only Swing code, listening for mouse click events, and adding some selected product to the `TableModel` object for display in the top right hand section. (As an aside, note that this is a classic use of the Model View Controller design pattern).

It is only when we press the "Checkout" button that we fire our business rules, in roughly the same order that we walked through the code earlier.

1. Method `CheckOutCallBack.checkout()` is called (eventually) by the Swing class waiting for the click on the "Checkout" button. This inserts the data from the `TableModel` object (top right hand side of the GUI), and inserts it into the Session's Working Memory. It then fires the rules.

2. The `"Explode Cart"` rule is the first to fire, given that it has `auto-focus` set to true. It loops through all the products in the cart, ensures that the products are in the Working Memory, and then gives the `"Show Items"` and `Evaluation` agenda groups a chance to fire. The rules in these groups add the contents of the cart to the text area (at the bottom of the window), decide whether or not to give us free fish food, and to ask us whether we want to buy a fish tank. This is shown in the figure below.

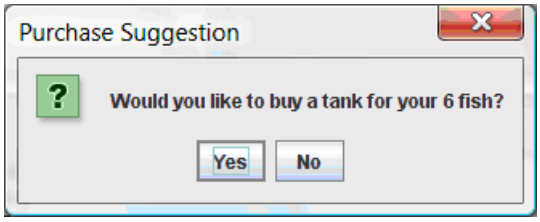


Figure 23.15. Do we want to buy a fish tank?

1. The *Do Checkout* rule is the next to fire as it (a) No other agenda group currently has focus and (b) it is part of the default (MAIN) agenda group. It always calls the *doCheckout() function* which displays a 'Would you like to Checkout?' Dialog Box.

2. The `doCheckout()` function sets the focus to the `checkout` agenda-group, giving the rules in that group the option to fire.

3. The rules in the the `checkout` agenda-group display the contents of the cart and apply the appropriate discount.

4. *Swing then waits for user input* to either checkout more products (and to cause the rules to fire again), or to close the GUI - see the figure below.

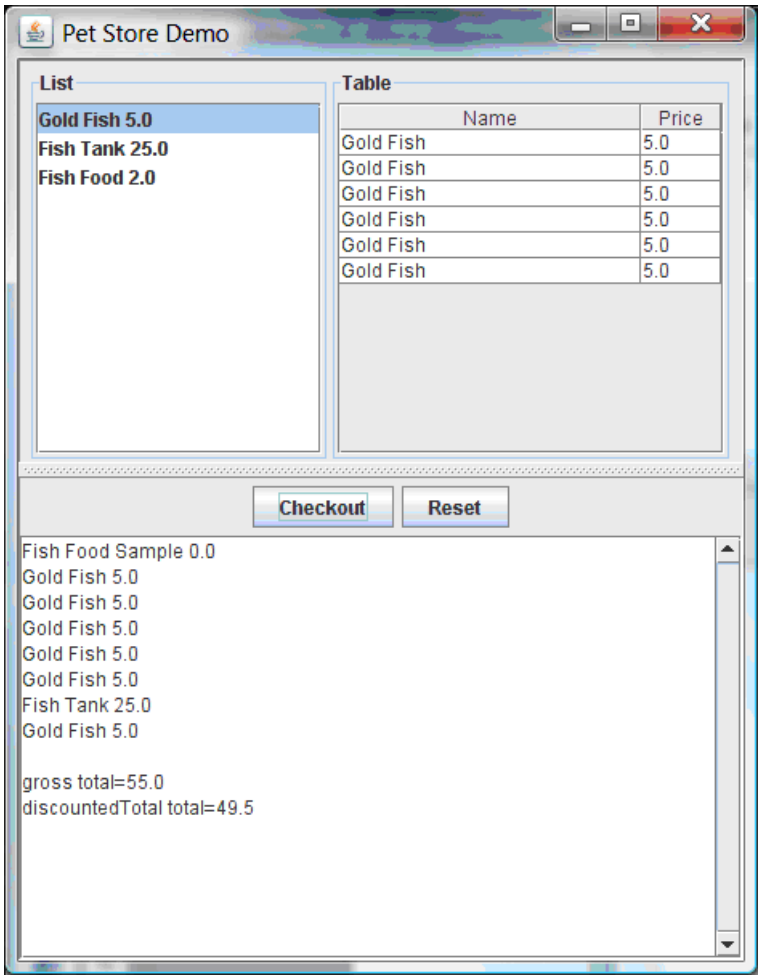


Figure 23.16. Petstore Demo after all rules have fired.

We could add more System.out calls to demonstrate this flow of events. The output, as it currently appears in the Console window, is given in the listing below.

Example 23.59. Console (System.out) from running the PetStore GUI

```
Adding free Fish Food Sample to cart SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
 SUGGESTION: Would you like to buy a tank for your 6 fish? -
```

23.8. Honest Politician Example

```
Name: Honest Politician
Main class: org.drools.examples.honestpolitician.HonestPoliticianExample
Module: drools-examples
Type: Java application
Rules file: HonestPoliticianExample.drl
Objective: Illustrate the concept of "truth maintenance" based on the logical insertion of facts
```

The Honest Politician example demonstrates truth maintenance with logical assertions. The basic premise is that an object can only exist while a statement is true. A rule's consequence can logically insert an object with the `insertLogical()` method. This means the object will only remain in the Working Memory as long as the rule that logically inserted it remains true. When the rule is no longer true the object is automatically retracted.

In this example there is the class `Politician`, with a name and a boolean value for being honest. Four politicians with honest state set to true are inserted.

Example 23.60. Class Politician

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

Example 23.61. Honest Politician: Execution

```
Politician blair = new Politician("blair", true);
Politician bush = new Politician("bush", true);
Politician chirac = new Politician("chirac", true);
Politician schroder = new Politician("schroder", true);

ksession.insert( blair );
ksession.insert( bush );
ksession.insert( chirac );
ksession.insert( schroder );

ksession.fireAllRules();
```

The Console window output shows that, while there is at least one honest politician, democracy lives. However, as each politician is in turn corrupted by an evil corporation, so that all politicians become dishonest, democracy is dead.

Example 23.62. Honest Politician: Console Output

```
Hurrah!!! Democracy LivesI'm an evil corporation and I have corrupted schroderI'm an evil
 corporation and I have corrupted chiracI'm an evil corporation and I have corrupted bushI'm an
 evil corporation and I have corrupted blairWe are all Doomed!!! Democracy is Dead
cy LivesI'm an evil corporation and I have
corrupted schroderI'm an evil corporation and I have
corrupted chiracI'm an evil corporation and I have
corrupted bushI'm an evil corporation and I have
corrupted blairWe are all Doomed!!! Democracy
```

As soon as there is at least one honest politician in the Working Memory a new `Hope` object is logically asserted. This object will only exist while there is at least one honest politician. As soon as all politicians are dishonest, the `Hope` object will be automatically retracted. This rule is given a salience of 10 to ensure that it fires before any other rule, as at this stage the "Hope is Dead" rule is actually true.

Example 23.63. Honest Politician: Rule "We have an honest politician"

```
rule "We have an honest Politician"    salience 10    when        exists( Politician( honest
 == true ) )    then        insertLogical( new Hope() );end
Politician"    salience
 10
 when        exists( Politician( honest == true )
 )
 then        insertLogical( new Hope()
```

As soon as a `Hope` object exists the "Hope Lives" rule matches and fires. It has a salience of 10 so that it takes priority over "Corrupt the Honest".

Example 23.64. Honest Politician: Rule "Hope Lives"

```
rule "Hope Lives"
    salience 10
        when
            exists( Hope() )
        then
            System.out.println("Hurrah!!! Democracy Lives");
end
```

Now that there is hope and we have, at the start, four honest politicians, we have four activations for this rule, all in conflict. They will fire in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted we have no politicians with the property `honest == true`. Thus, the rule "We have an honest Politician" is no longer true and the object it logical inserted (due to the last execution of `new Hope()`) is automatically retracted.

Example 23.65. Honest Politician: Rule "Corrupt the Honest"

```
rule "Corrupt the Honest"     when         politician : Politician( honest == true )
     exists( Hope() )     then         System.out.println( "I'm an evil corporation and I have
 corrupted " + politician.getName() );        modify ( politician ) { honest = false };end
Honest"
 when        politician : Politician( honest == true )
          exists( Hope()
 )
 then      System.out.println( "I'm an evil corporation and I have corrupted " + politician.getName()
 );        modify ( politician ) { honest = false
```

With the `Hope` object being automatically retracted, via the truth maintenance system, the conditional element `not` applied to `Hope` is no longer true so that the following rule will match and fire.

Example 23.66. Honest Politician: Rule "Hope is Dead"

```
rule "Hope is Dead"
    when
        not( Hope() )
    then
        System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

Let's take a look at the Audit trail for this application:

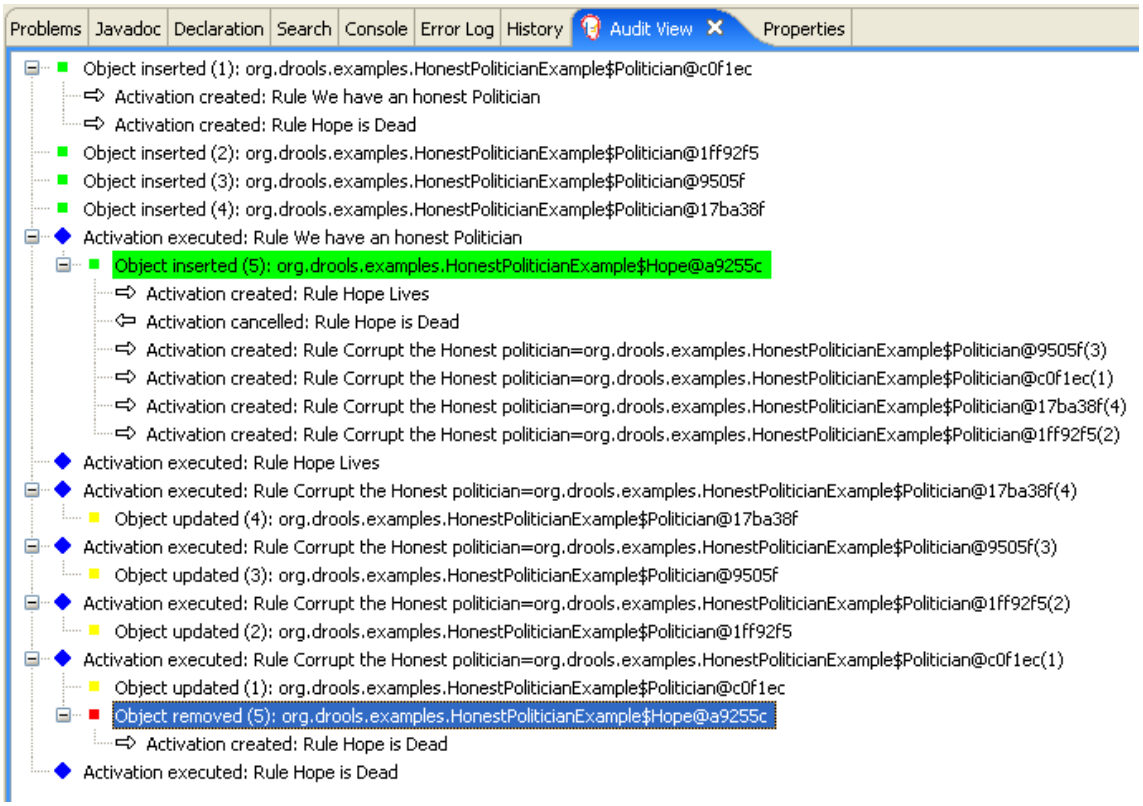


Figure 23.17. Honest Politician Example Audit View

The moment we insert the first politician we have two activations. The rule "We have an honest Politician" is activated only once for the first inserted politician because it uses an `exists` conditional element, which matches once for any number. The rule "Hope is Dead" is also activated at this stage, because we have not yet inserted the `Hope` object. Rule "We have an honest Politician" fires first, as it has a higher salience than "Hope is Dead", which inserts the `Hope` object. (That

action is highlighted green.) The insertion of the `Hope` object activates "Hope Lives" and de-activates "Hope is Dead"; it also activates "Corrupt the Honest" for each inserted honest politician. Rule "Hope Lives" executes, printing "Hurrah!!! Democracy Lives". Then, for each politician, rule "Corrupt the Honest" fires, printing "I'm an evil corporation and I have corrupted X", where X is the name of the politician, and modifies the politician's honest value to false. When the last honest politician is corrupted, `Hope` is automatically retracted, by the truth maintenance system, as shown by the blue highlighted area. The green highlighted area shows the origin of the currently selected blue highlighted area. Once the `Hope` fact is retracted, "Hope is dead" activates and fires printing "We are all Doomed!!! Democracy is Dead".

23.9. Sudoku Example

Name: Sudoku
Main class: org.drools.examples.sudoku.SudokuExample
Type: Java application
Rules file: sudoku.drl, validate.drl
Objective: Demonstrates the solving of logic problems, and complex pattern matching.

This example demonstrates how Drools can be used to find a solution in a large potential solution space based on a number of constraints. We use the popular puzzle of Sudoku. This example also shows how Drools can be integrated into a graphical interface and how callbacks can be used to interact with a running Drools rules engine in order to update the graphical interface based on changes in the Working Memory at runtime.

23.9.1. Sudoku Overview

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9, once, and only once.

The puzzle setter provides a partially completed grid and the puzzle solver's task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number it should be unique in its particular 3x3 zone, row and column.

See Wikipedia [http://en.wikipedia.org/wiki/Sudoku] for a more detailed description.

23.9.2. Running the Example

Download and install drools-examples as described above and then execute `java org.drools.examples.DroolsExamplesApp` and click on "SudokuExample".

The window contains an empty grid, but the program comes with a number of grids stored internally which can be loaded and solved. Click on "File", then "Samples" and select "Simple" to load one of the examples. Note that all buttons are disabled until a grid is loaded.

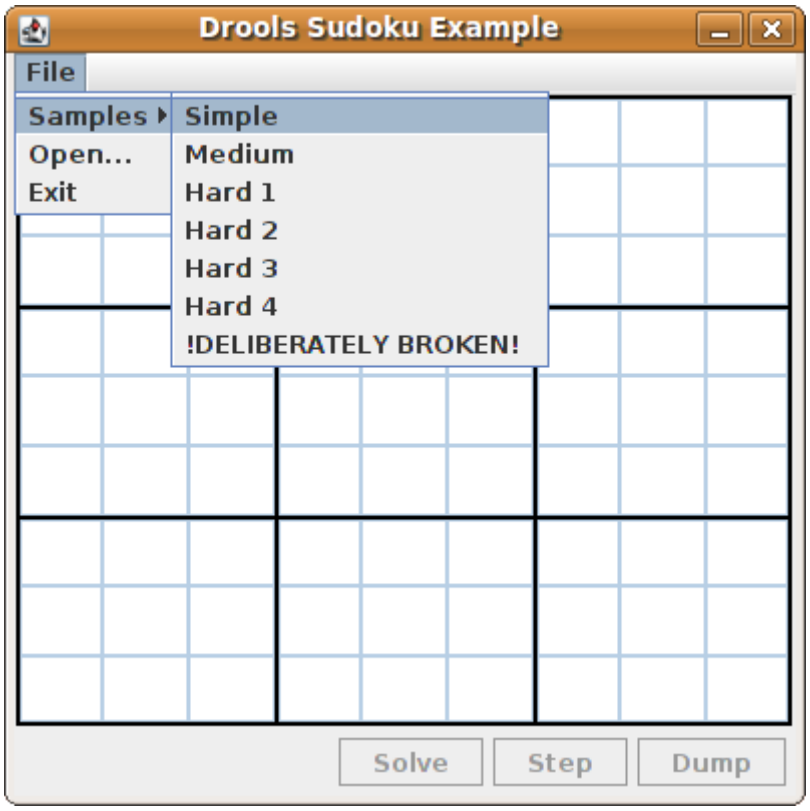


Figure 23.18. Initial screen

Loading the "Simple" example fills the grid according to the puzzle's initial state.

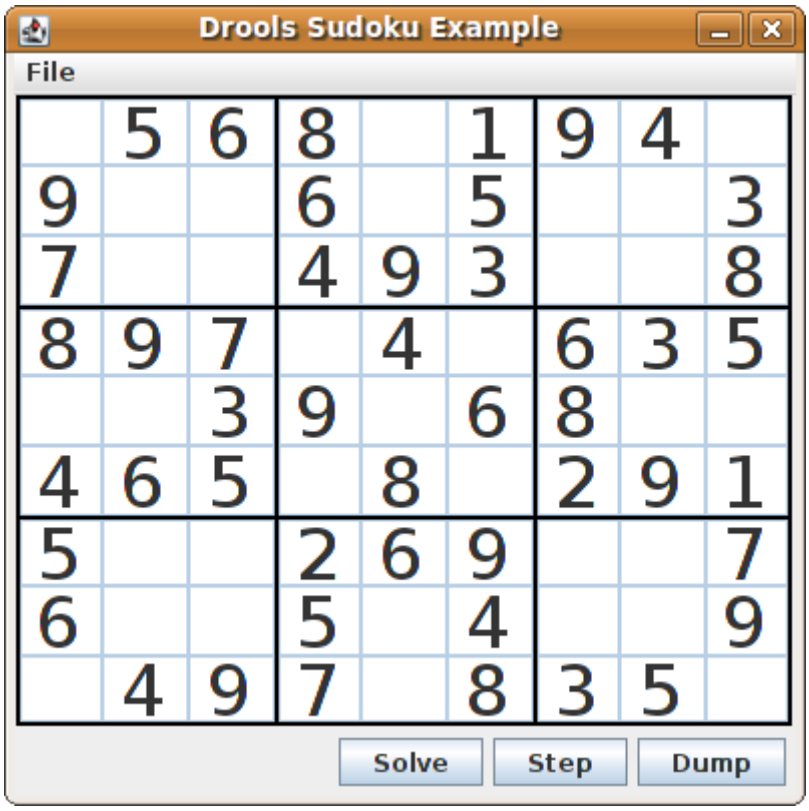


Figure 23.19. After loading "Simple"

Click on the "Solve" button and the Drools-based engine will fill out the remaining values, and the buttons are inactive once more.

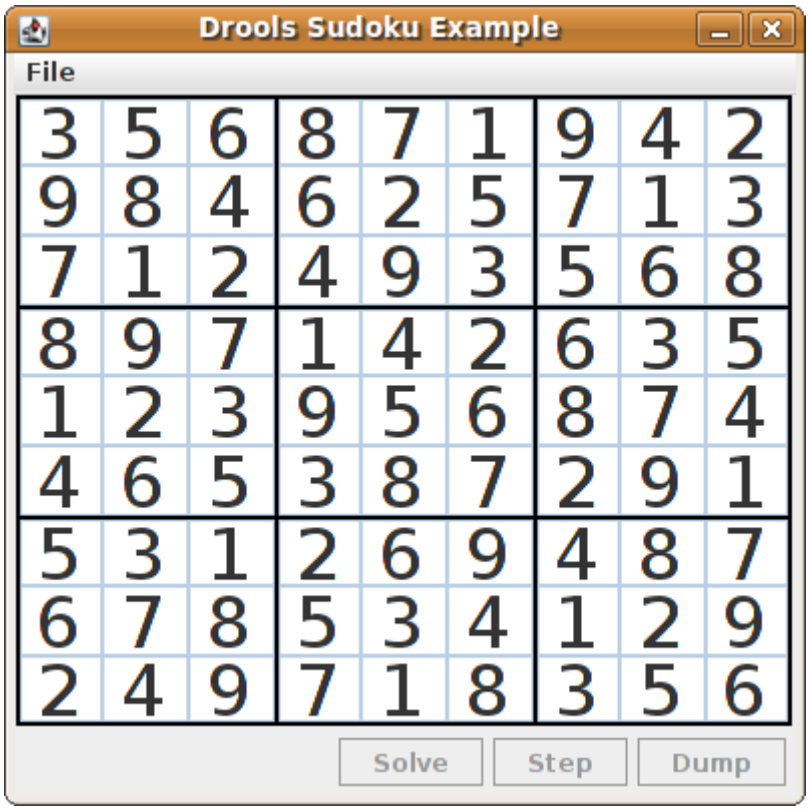


Figure 23.20. "Simple" Solved

Alternatively, you may click on the "Step" button to see the next digit found by the rule set. The Console window will display detailed information about the rules which are executing to solve the step in a human readable form. Some examples of these messages are presented below.

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```

Click on the "Dump" button to see the state of the grid, with cells showing either the established value or the remaining possibilitiescandidates.

```
       Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col: 5
 Col: 6     Col: 7     Col: 8
Row 0:   2 4  7 9   2 456        4567 9   23 56  9  --- 5 ---  --- 1 ---
 3  67 9  --- 8 ---     4 67
Row 1:  12    7 9  --- 8 ---  1    67 9   23  6  9  --- 4 ---   23  67    1
 3  67 9    3  67 9  --- 5 ---
```

```
Row 2:  1  4  7 9  1  456     --- 3 ---      56 89      5 78       5678
 --- 2 ---     4 67 9  1  4 67
Row 3:  1234       12345      1  45      12  5  8   --- 6 ---   2  5 78
  5 78      45 7    --- 9 ---
Row 4:  --- 6 ---  --- 7 ---      5      --- 4 ---   2  5  8   --- 9 ---
  5  8   --- 1 ---  --- 3 ---
Row 5:  --- 8 ---  12 45      1  45   9  12  5      --- 3 ---   2  5 7
  567       4567     2 4 67
Row 6:  1 3   7    1 3  6     --- 2 ---    3 56 8       5  8     3 56 8
 --- 4 ---    3 567 9  1    678
Row 7:  --- 5 ---  1 34 6     1  4 678     3  6 8   --- 9 ---    34 6 8   1
 3  678   --- 2 ---  1    678
Row 8:    34       --- 9 ---     4 6 8   --- 7 ---  --- 1 ---   23456 8
 3 56 8     3 56          6 8
```

Now, let us load a Sudoku grid that is deliberately invalid. Click on "File", "Samples" and "! DELIBERATELY BROKEN!". Note that this grid starts with some issues, for example the value 5 appears twice in the first row.

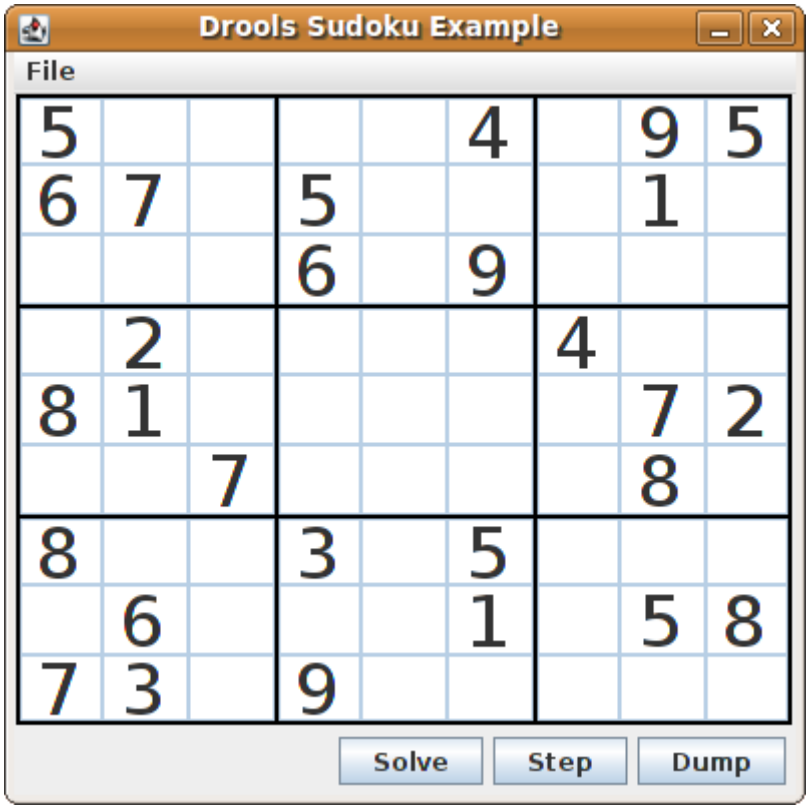


Figure 23.21. Broken initial state

A few simple rules perform a sanity check, right after loading a grid. In this case, the following messages are printed on standard output:

```
```

```
cell [0,8]: 5 has a duplicate in row 0
cell [0,0]: 5 has a duplicate in row 0
cell [6,0]: 8 has a duplicate in col 0
cell [4,0]: 8 has a duplicate in col 0
Validation complete.
```

Nevertheless, click on the "Solve" button to apply the solving rules to this invalid grid. This will not complete; some cells remain empty.

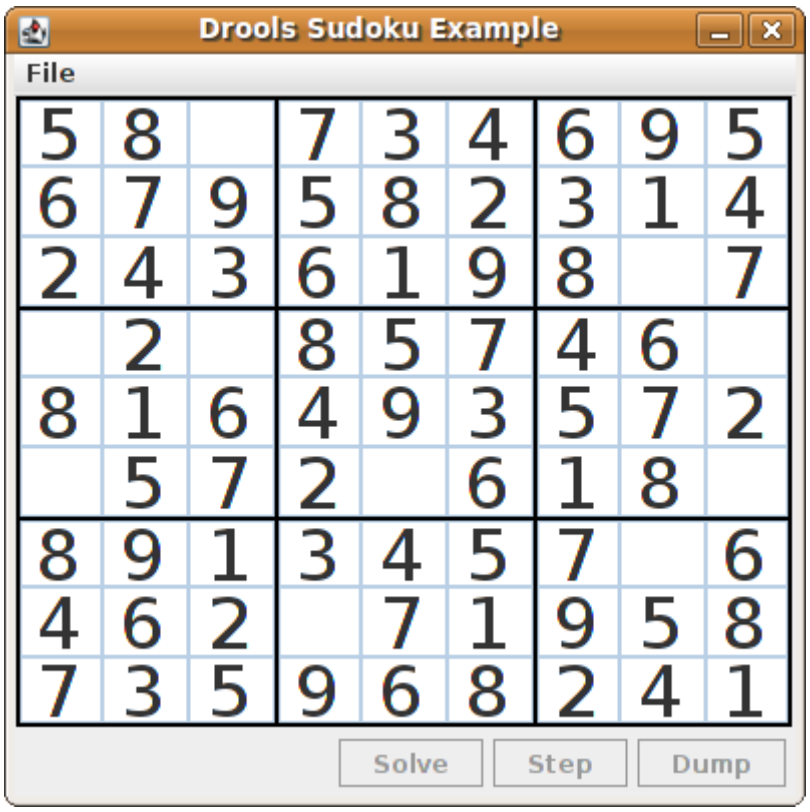


Figure 23.22. Broken "solved" state

The solving functionality has been achieved by the use of rules that implement standard solving techniques. They are based on the sets of values that are still candidates for a cell. If, for instance, such a set contains a single value, then this is the value for the cell. A little less obvious is the single occurrence of a value in one of the groups of nine cells. The rules detecting these situations insert a fact of type Setting with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to. Finally, it is retracted.

Other rules merely reduce the permissible values for some cells. Rules "naked pair", "hidden pair in row", "hidden pair in column" and "hidden pair in square" merely eliminate possibilities but do not establish solutions. More sophisticated eliminations are done by "X-wings in rows", "X-wings in columns", "intersection removal row" and "intersection removal column".

23.9.3. Java Source and Rules Overview

The Java source code can be found in the /src/main/java/org/drools/examples/sudoku directory, with the two DRL files defining the rules located in the /src/main/rules/org/drools/examples/sudoku directory.

The package `org.drools.examples.sudoku.swing` contains a set of classes which implement a framework for Sudoku puzzles. Note that this package does not have any dependencies on the Drools libraries. `SudokuGridModel` defines an interface which can be implemented to store a Sudoku puzzle as a 9x9 grid of `Cell` objects. `SudokuGridView` is a Swing component which can visualize any implementation of `SudokuGridModel`. `SudokuGridEvent` and `SudokuGridListener` are used to communicate state changes between the model and the view: events are fired when a cell's value is resolved or changed. If you are familiar with the model-view-controller patterns in other Swing components such as `JTable` then this pattern should be familiar. `SudokuGridSamples` provides a number of partially filled Sudoku puzzles for demonstration purposes.

Package `org.drools.examples.sudoku.rules` contains a utility class with a method for compiling DRL files.

The package `org.drools.examples.sudoku` contains a set of classes implementing the elementary `Cell` object and its various aggregations: the `CellFile` subtypes `CellRow` and `CellCol` as well as `CellSqr`, all of which are subtypes of `CellGroup`. It's interesting to note that `Cell` and `CellGroup` are subclasses of `SetOfNine`, which provides a property `free` with the type `Set<Integer>`. For a `Cell` it represents the individual candidate set; for a `CellGroup` the set is the union of all candidate sets of its cells, or, simply, the set of digits that still need to be allocated.

With 81 `Cell` and 27 `CellGroup` objects and the linkage provided by the `Cell` properties `cellRow`, `cellCol` and `cellSqr` and the `CellGroup` property `cells`, a list of `Cell` objects, it is possible to write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.

An object of class `Setting` is used for triggering the operations that accompany the allocation of a value: its removal from the candidate sets of sibling cells and associated cell groups. Moreover, the presence of a `Setting` fact is used in all rules that should detect a new situation; this is to avoid reactions to inconsistent intermediary states.

An object of class `Stepping` is used in a low priority rule to execute an emergency halt when a "Step" does not terminate regularly. This indicates that the puzzle cannot be solved by the program.

The class `org.drools.examples.sudoku.SudokuExample` implements a Java application combining the components described.

23.9.4. Sudoku Validator Rules (validate.drl)

Validation rules detect duplicate numbers in cell groups. They are combined in an agenda group which enables us to activate them, explicitly, after loading a puzzle.

The three rules "duplicate in cell..." are very similar. The first pattern locates a cell with an allocated value. The second pattern pulls in any of the three cell groups the cell belongs to. The final pattern would find a cell (other than the first one) with the same value as the first cell and in the same row, column or square, respectively.

Rule "terminate group" fires last. It prints a message and calls halt.

23.9.5. Sudoku Solving Rules (sudoku.drl)

There are three types of rules in this file: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

Rules "set a value", "eliminate a value from Cell" and "retract setting" depend on the presence of a `Setting` object. The first rule handles the assignment to the cell and the operations for removing the value from the "free" sets of the cell's three groups. Also, it decrements a counter that, when zero, returns control to the Java application that has called `fireUntilHalt()`. The purpose of rule "eliminate a value from Cell" is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, rule "retract setting" retracts the triggering `Setting` fact.

There are just two rules that detect a situation where an allocation of a number to a cell is possible. Rule "single" fires for a `Cell` with a candidate set containing a single number. Rule "hidden single" fires when there is no cell with a single candidate but when there is a cell containing a candidate but this candidate is absent from all other cells in one of the three groups the cell belongs to. Both rules create and insert a `Setting` fact.

Rules from the largest group of rules implement, singly or in groups of two or three, various solving techniques, as they are employed when solving Sudoku puzzles manually.

Rule "naked pair" detects identical candidate sets of size 2 in two cells of a group; these two values may be removed from all other candidate sets of that group.

A similar idea motivates the three rules "hidden pair in..."; here, the rules look for a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of this group. This, then, means that all other candidates can be eliminated from the two cells harbouring the hidden pair.

A pair of rules deals with "X-wings" in rows and columns. When there are only two possible cells for a value in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. If you follow the pattern sequence in one of these rules, you will see how the conditions that are conveniently expressed by words such as "same" or "only" result in patterns with suitable constraints or prefixed with "not".

The rule pair "intersection removal..." is based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in

one of those two or three cells of the row or column; hence it can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside the square and within the same cell file.

These rules are sufficient for many but certainly not for all Sudoku puzzles. To solve very difficult grids, the rule set would need to be extended with more complex rules. (Ultimately, there are puzzles that cannot be solved except by trial and error.)

23.10. Number Guess

```
Name: Number Guess
Main class: org.drools.examples.numberguess.NumberGuessExample
Module: droolsjbpm-integration-examples (Note: this is in a different download, the droolsjbpm-integration download.)
Type: Java application
Rules file: NumberGuess.drl
Objective: Demonstrate use of Rule Flow to organise Rules
```

The "Number Guess" example shows the use of Rule Flow, a way of controlling the order in which rules are fired. It uses widely understood workflow diagrams for defining the order in which groups of rules will be executed.

Example 23.67. Creating the Number Guess RuleBase: NumberGuessExample.main() - part 1

```
final KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.drl",
                                                    ShoppingExample.class ),
              ResourceType.DRL );
kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.rf",
                                                    ShoppingExample.class ),
              ResourceType.DRF );

final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

The creation of the package and the loading of the rules (using the `add()` method) is the same as the previous examples. There is an additional line to add the Rule Flow (`NumberGuess.rf`), which provides the option of specifying different rule flows for the same Knowledge Base. Otherwise, the Knowledge Base is created in the same manner as before.

Example 23.68. Starting the RuleFlow: NumberGuessExample.main() - part 2

```
final StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

```
KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/numberguess");

ksession.insert( new GameRules( 100, 5 ) );
ksession.insert( new RandomNumber() );
ksession.insert( new Game() );

ksession.startProcess( "Number Guess" );
ksession.fireAllRules();

logger.close();

ksession.dispose();
```

Once we have a Knowledge Base, we can use it to obtain a Stateful Session. Into our session we insert our facts, i.e., standard Java objects. (For simplicity, in this sample, these classes are all contained within our `NumberGuessExample.java` file. Class `GameRules` provides the maximum range and the number of guesses allowed. Class `RandomNumber` automatically generates a number between 0 and 100 and makes it available to our rules, by insertion via the `getValue()` method. Class `Game` keeps track of the guesses we have made before, and their number.

Note that before we call the standard `fireAllRules()` method, we also start the process that we loaded earlier, via the `startProcess()` method. We'll learn where to obtain the parameter we pass ("Number Guess", i.e., the identifier of the rule flow) when we talk about the rule flow file and the graphical Rule Flow Editor below.

Before we finish the discussion of our Java code, we note that in some real-life application we would examine the final state of the objects. (Here, we could retrieve the number of guesses, to add it to a high score table.) For this example we are content to ensure that the Working Memory session is cleared by calling the `dispose()` method.

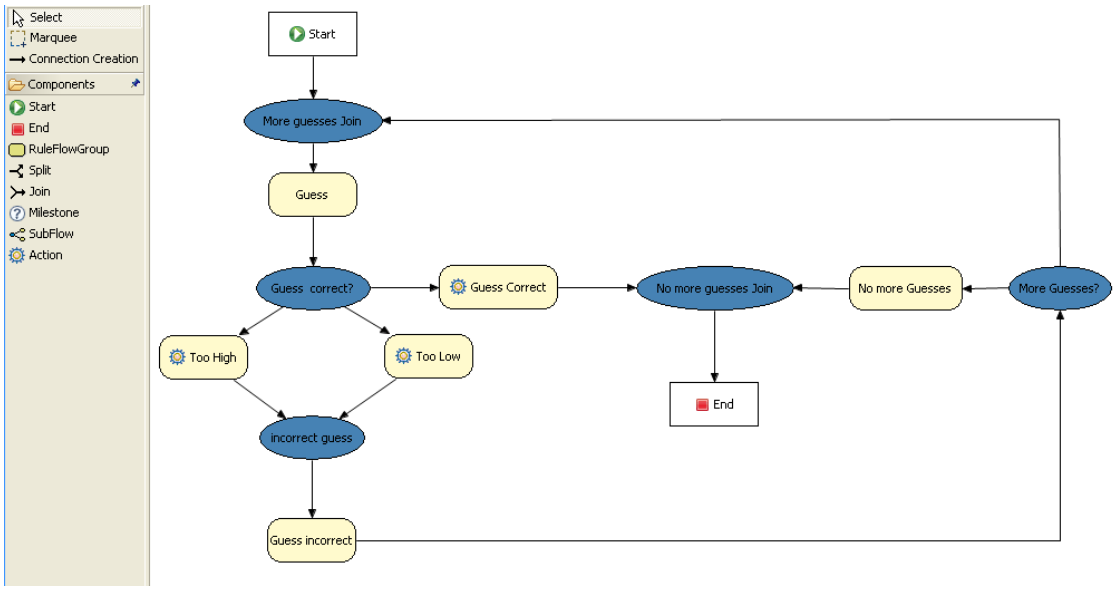


Figure 23.23. RuleFlow for the NumberGuess Example

If you open the `NumberGuess.rf` file in the Drools IDE (provided you have the JBoss Rules extensions installed correctly in Eclipse) you should see the above diagram, similar to a standard flowchart. Its icons are similar (but not exactly the same) as in the JBoss jBPM workflow product. Should you wish to edit the diagram, a menu of available components should be available to the left of the diagram in the IDE, which is called the *palette*. This diagram is saved in XML, an (almost) human readable format, using XStream.

If it is not already open, ensure that the Properties View is visible in the IDE. It can be opened by clicking "Window", then "Show View" and "Other", where you can select the "Properties" view. If you do this *before* you select any item on the rule flow (or click on the blank space in the rule flow) you should be presented with the following set of properties.

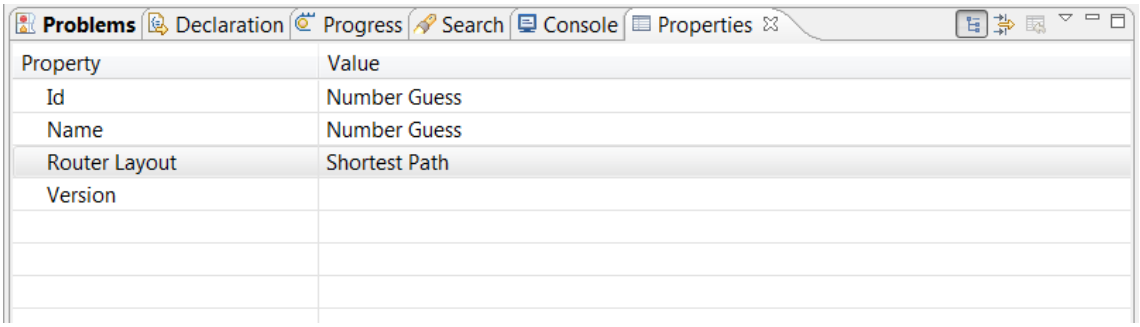


Figure 23.24. Properties for the Number Guess Rule Flow

Keep an eye on the Properties View as we progress through the example's rule flow, as it presents valuable information. In this case, it provides us with the identification of the Rule Flow Process that we used in our earlier code snippet, when we called `session.startProcess()`.

In the "Number Guess" Rule Flow we encounter several node types, many of them identified by an icon.

- The Start node (white arrow in a green circle) and the End node (red box) mark beginning and end of the rule flow.
- A Rule Flow Group box (yellow, without an icon) represents a Rule Flow Groups defined in our rules (DRL) file that we will look at later. For example, when the flow reaches the Rule Flow Group "Too High", only those rules marked with an attribute of `ruleflow-group "Too High"` can potentially fire.
- Action nodes (yellow, cog-shaped icon) perform standard Java method calls. Most action nodes in this example call `System.out.println()`, indicating the program's progress to the user.
- Split and Join Nodes (blue ovals, no icon) such as "Guess Correct?" and "More guesses Join" mark places where the flow of control can split, according to various conditions, and rejoin, respectively
- Arrows indicate the flow between the various nodes.

The various nodes in combination with the rules make the Number Guess game work. For example, the "Guess" Rule Flow Group allows only the rule "Get user Guess" to fire, because only that rule has a matching attribute of `ruleflow-group "Guess"`.

Example 23.69. A Rule firing only at a specific point in the Rule Flow: NumberGuess.drl

```
rule "Get user Guess"
    ruleflow-group "Guess"
    no-loop
    when
        $r : RandomNumber()
        rules : GameRules( allowed : allowedGuesses )
        game : Game( guessCount < allowed )
        not ( Guess() )
    then
        System.out.println( "You have " + ( rules.allowedGuesses - game.guessCount )
                            + " out of " + rules.allowedGuesses
                            + " guesses left.\nPlease enter your guess from 0 to "
                            + rules.maxRange );
        br = new BufferedReader( new InputStreamReader( System.in ) );
        i = br.readLine();
        modify ( game ) { guessCount = game.guessCount + 1 }
        insert( new Guess( i ) );
end
```

The rest of this rule is fairly standard. The LHS section (after `when`) of the rule states that it will be activated for each `RandomNumber` object inserted into the Working Memory where `guessCount` is less than `allowedGuesses` from the `GameRules` object and where the user has not guessed the correct number.

The RHS section (or consequence, after `then`) prints a message to the user and then awaits user input from `System.in`. After obtaining this input (the `readLine()` method call blocks until the return key is pressed) it modifies the guess count and inserts the new guess, making both available to the Working Memory.

The rest of the rules file is fairly standard: the package declares the dialect as MVEL, and various Java classes are imported. In total, there are five rules in this file:

1. Get User Guess, the Rule we examined above.
2. A Rule to record the highest guess.
3. A Rule to record the lowest guess.
4. A Rule to inspect the guess and retract it from memory if incorrect.
5. A Rule that notifies the user that all guesses have been used up.

One point of integration between the standard Rules and the RuleFlow is via the `ruleflow-group` attribute on the rules, as discussed above. A *second point of integration between the rules (.drl) file and the Rules Flow .rf files* is that the Split Nodes (the blue ovals) can use values in the Working Memory (as updated by the rules) to decide which flow of action to take. To see how this works, click on the "Guess Correct Node"; then within the Properties View, open the Constraints Editor by clicking the button at the right that appears once you click on the "Constraints" property line. You should see something similar to the diagram below.

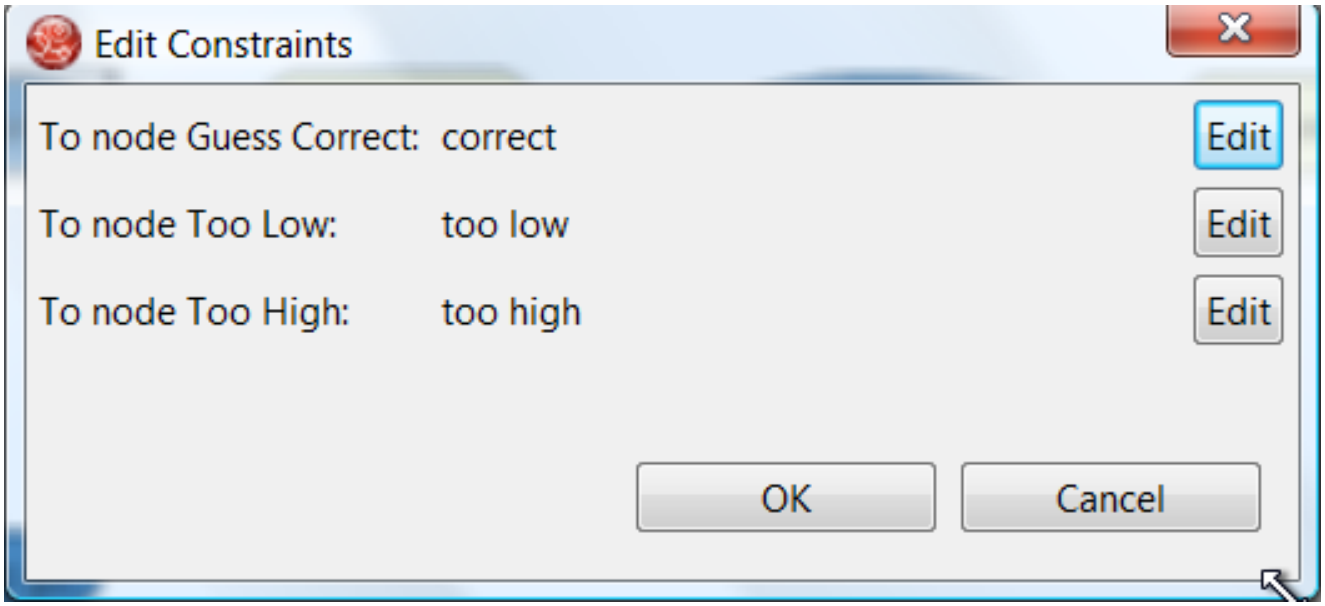


Figure 23.25. Edit Constraints for the "Guess Correct" Node

Click on the "Edit" button beside "To node Too High" and you'll see a dialog like the one below. The values in the "Textual Editor" window follow the standard rule format for the LHS and can refer to objects in Working Memory. The consequence (RHS) is that the flow of control follows this node (i.e., "To node Too High") if the LHS expression evaluates to true.

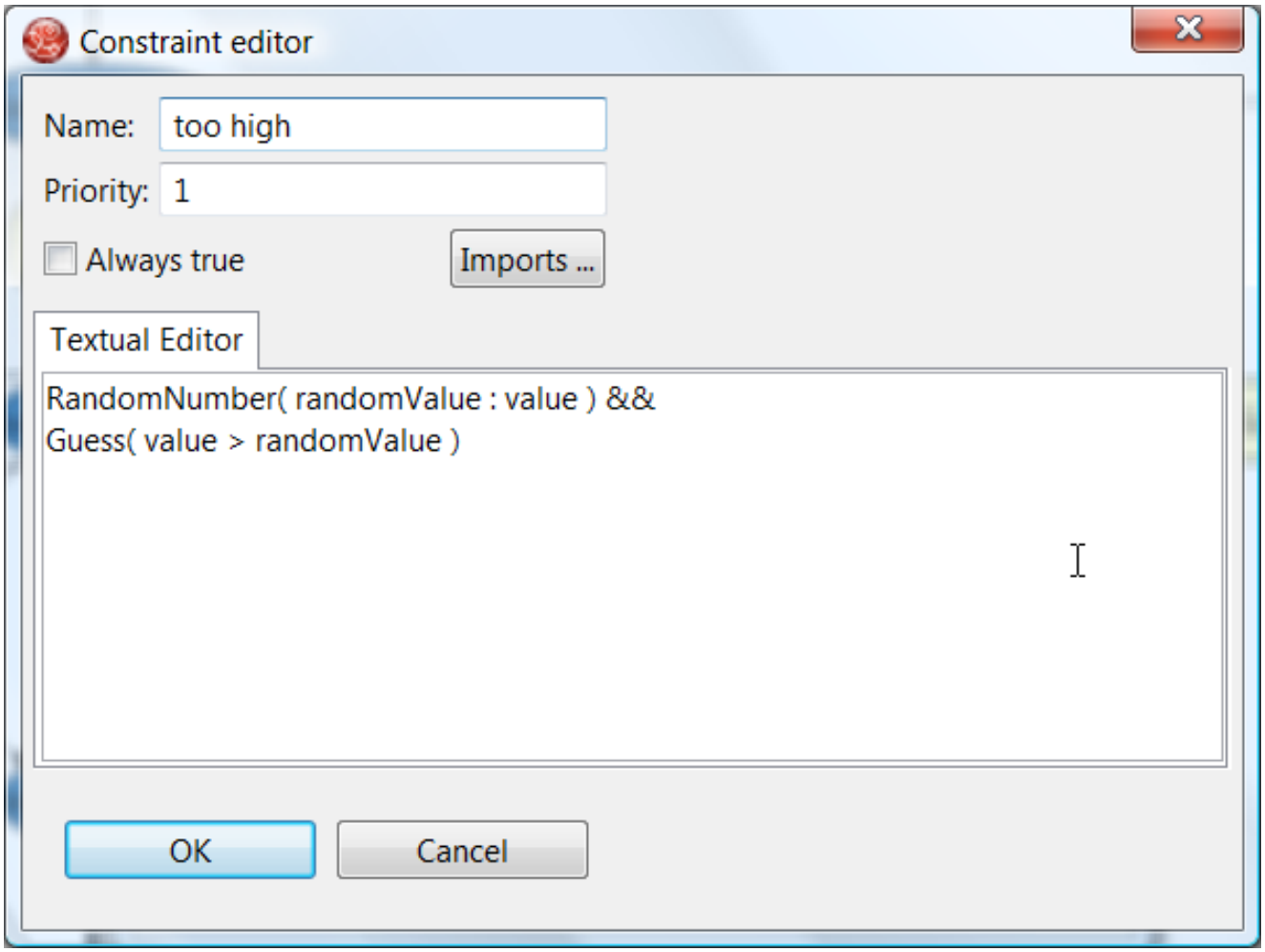


Figure 23.26. Constraint Editor for the "Guess Correct" Node: value too high

Since the file `NumberGuess.java` contains a `main()` method, it can be run as a standard Java application, either from the command line or via the IDE. A typical game might result in the interaction below. The numbers in bold are typed in by the user.

Example 23.70. Example Console output where the Number Guess Example beat the human!

```
You have 5 out of 5 guesses left.Please enter your guess from 0 to 100
left.Please enter your guess from 0 to
50
Your guess was too high
You have 4 out of 5 guesses left.
Please enter your guess from 0 to 100
25
Your guess was too low
You have 3 out of 5 guesses left.
Please enter your guess from 0 to 100
37
Your guess was too low
```

```
You have 2 out of 5 guesses left.
Please enter your guess from 0 to 100
44
Your guess was too low
You have 1 out of 5 guesses left.
Please enter your guess from 0 to 100
47
Your guess was too low
You have no more guesses
The correct guess was 48
```

A summary of what is happening in this sample is:

1. The `main()` method of `NumberGuessExample.java` loads a Rule Base, creates a Stateful Session and inserts `Game`, `GameRules` and `RandomNumber` (containing the target number) objects into it. The method also sets the process flow we are going to use, and fires all rules. Control passes to the Rule Flow.

2. File `NumberGuess.rf`, the Rule Flow, begins at the "Start" node.

3. Control passes (via the "More guesses" join node) to the Guess node.

4. At the Guess node, the appropriate Rule Flow Group ("Get user Guess") is enabled. In this case the Rule "Guess" (in the `NumberGuess.drl` file) is triggered. This rule displays a message to the user, takes the response, and puts it into Working Memory. Flow passes to the next Rule Flow Node.

5. At the next node, "Guess Correct", constraints inspect the current session and decide which path to take.

   If the guess in step 4 was too high or too low, flow proceeds along a path which has an action node with normal Java code printing a suitable message and a Rule Flow Group causing a highest guess or lowest guess rule to be triggered. Flow passes from these nodes to step 6.

   If the guess in step 4 was right, we proceed along the path towards the end of the Rule Flow. Before we get there, an action node with normal Java code prints a statement "you guessed correctly". There is a join node here (just before the Rule Flow end) so that our no-more-guesses path (step 7) can also terminate the Rule Flow.

6. Control passes as per the Rule Flow via a join node, a guess incorrect Rule Flow Group (triggering a rule to retract a guess from Working Memory) onto the "More guesses" decision node.

7. The "More guesses" decision node (on the right hand side of the rule flow) uses constraints, again looking at values that the rules have put into the working memory, to decide if we have more guesses and if so, goto step 3. If not, we proceed to the end of the rule flow, via a Rule Flow Group that triggers a rule stating "you have no more guesses".

8. The loop over steps 3 to 7 continues until the number is guessed correctly, or we run out of guesses.

23.11. Conway's Game Of Life

```
Name: Conway's Game Of Life
Main class: org.drools.examples.conway.ConwayAgendaGroupRun
            org.drools.examples.conway.ConwayRuleFlowGroupRun
Module: droolsjbpm-integration-examples (Note: this is in a different download, the droolsjbpm-
integration download.)
Type: Java application
Rules file: conway-ruleflow.drl conway-agendagroup.drl
Objective: Demonstrates 'accumulate', 'collect' and 'from'
```

Conway's Game Of Life, described in http://en.wikipedia.org/wiki/Conway's_Game_of_Life and in http://www.math.com/students/wonders/life/life.html, is a famous cellular automaton conceived in the early 1970's by the mathematician John Conway. While the system is well known as "Conway's Game Of Life", it really isn't a game at all. Conway's system is more like a simulation of a form of life. Don't be intimidated. The system is terribly simple and terribly interesting. Math and Computer Science students alike have marvelled over Conway's system for more than 30 years now. The application presented here is a Swing-based implementation of Conway's Game of Life. The rules that govern the system are implemented as business rules using Drools. This document will explain the rules that drive the simulation and discuss the Drools parts of the implementation.

We'll first introduce the grid view, shown below, designed for the visualisation of the game, showing the "arena" where the life simulation takes place. Initially the grid is empty, meaning that there are no live cells in the system. Each cell is either alive or dead, with live cells showing a green ball. Preselected patterns of live cells can be chosen from the "Pattern" drop-down list. Alternatively, individual cells can be doubled-clicked to toggle them between live and dead. It's important to understand that each cell is related to its neighboring cells, which is fundamental for the game's rules. Neighbors include not only cells to the left, right, top and bottom but also cells that are connected diagonally, so that each cell has a total of 8 neighbors. Exceptions are the four corner cells which have only three neighbors, and the cells along the four border, with five neighbors each.

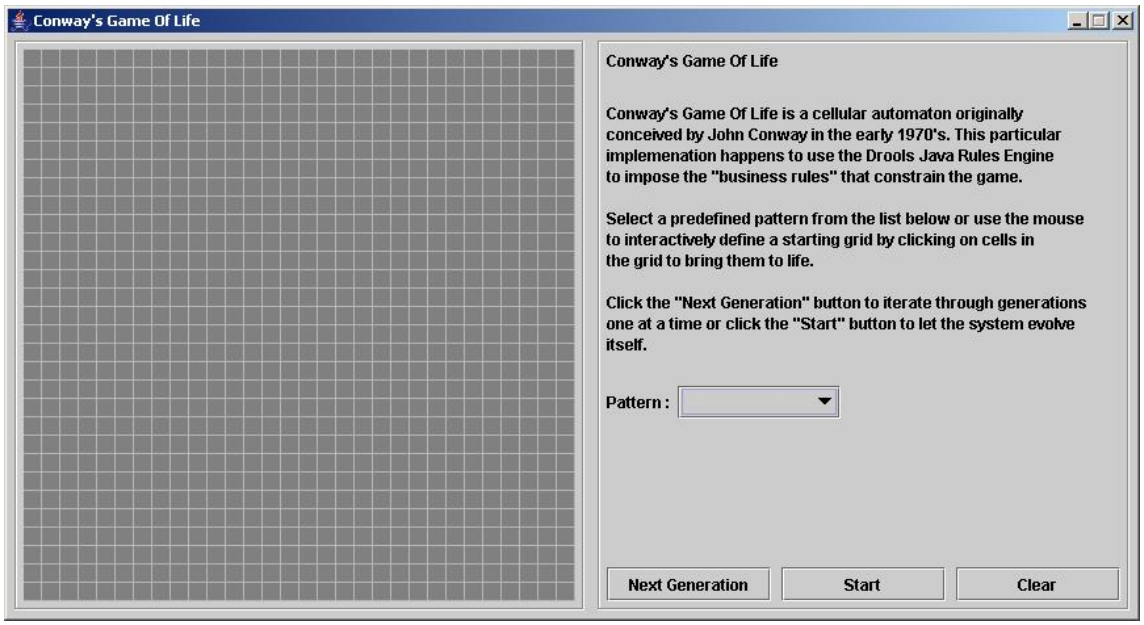


Figure 23.27. Conway's Game of Life: Starting a new game

So what are the basic rules that govern this game? Its goal is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells. This is the simple set of rules that govern what the next generation will look like:

- If a live cell has fewer than 2 live neighbors, it dies of loneliness.
- If a live cell has more than 3 live neighbors, it dies from overcrowding.
- If a dead cell has exactly 3 live neighbors, it comes to life.

That is all there is to it. Any cell that doesn't meet any of those criteria is left as is for the next generation. With those simple rules in mind, go back and play with the system a little bit more and step through some generations, one at a time, and notice these rules taking their effect.

The screenshot below shows an example generation, with a number of live cells. Don't worry about matching the exact patterns represented in the screen shot. Just get some groups of cells added to the grid. Once you have groups of live cells in the grid, or select a pre-designed pattern, click the "Next Generation" button and notice what happens. Some of the live cells are killed (the green ball disappears) and some dead cells come to life (a green ball appears). Step through several generations and see if you notice any patterns. If you click on the "Start" button, the system will evolve itself so you don't need to click the "Next Generation" button over and over. Play with the system a little and then come back here for more details of how the application works.

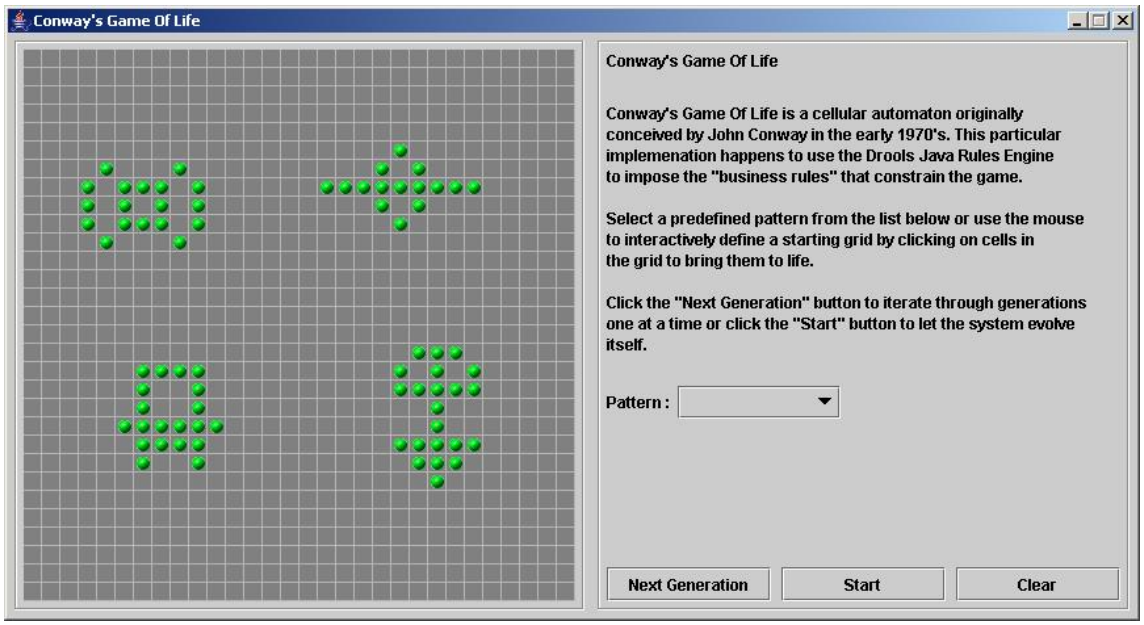


Figure 23.28. Conway's Game of Life: A running game

Now lets delve into the code. As this is an advanced example we'll assume that by now you know your way around the Drools framework and are able to connect the presented highlight, so that we'll just focus at a high level overview. The example has two ways to execute, one way uses Agenda Groups to manage execution flow, and the other one uses Rule Flow Groups to manage execution flow. These two versions are implemented in `ConwayAgendaGroupRun` and `ConwayRuleFlowGroupRun`, respectively. Here, we'll discuss the Rule Flow version, as it's what most people will use.

All the `Cell` objects are inserted into the Session and the rules in the `ruleflow-group` "register neighbor" are allowed to execute by the Rule Flow process. This group of four rules creates `Neighbor` relations between some cell and its northeastern, northern, northwestern and western neighbors. This relation is bidirectional, which takes care of the other four directions. Border cells don't need any special treatment - they simply won't be paired with neighboring cells where there isn't any. By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Example 23.71. Conway's Game of Life: Register Cell Neighbour relations

```
rule "register north east"    ruleflow-group "register neighbor"when    $cell: Cell( $row : row,
 $col : col )                   $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
   then                            insert( new Neighbor( $cell, $northEast ) );     insert( new
 Neighbor( $northEast, $cell ) );         endrule "register north"    ruleflow-group "register
 neighbor"  when    $cell: Cell( $row : row, $col : col )       $north : Cell( row  == ($row -
 1), col == $col )    then             insert( new Neighbor( $cell, $north ) );    insert( new
 Neighbor( $north, $cell ) );        endrule "register north west"    ruleflow-group "register
 neighbor"when    $cell: Cell( $row : row, $col : col )              $northWest : Cell( row  ==
 ($row - 1), col == ( $col - 1 ) )                    then          insert( new Neighbor( $cell,
 $northWest ) );    insert( new Neighbor( $northWest, $cell ) );        endrule "register west"
    ruleflow-group "register neighbor"when    $cell: Cell( $row : row, $col : col )
```

```
    $west : Cell( row  == $row, col == ( $col - 1 ) )                        then
 insert( new Neighbor( $cell, $west ) );    insert( new Neighbor( $west, $cell ) );        end
 east"    ruleflow-group "register

neighbor"when    $cell: Cell( $row : row, $col : col )
      $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
  then
      insert( new Neighbor( $cell, $northEast )
 );    insert( new Neighbor( $northEast, $cell ) );


 endrule "register
 north"    ruleflow-group "register neighbor"

 when    $cell: Cell( $row : row, $col : col )
      $north : Cell( row  == ($row - 1), col == $col )
  then
      insert( new Neighbor( $cell, $north )
 );    insert( new Neighbor( $north, $cell ) );


 endrule "register north
 west"    ruleflow-group "register

neighbor"when    $cell: Cell( $row : row, $col : col )
      $northWest : Cell( row  == ($row - 1), col == ( $col - 1 ) )
  then
      insert( new Neighbor( $cell, $northWest )
 );    insert( new Neighbor( $northWest, $cell ) );


 endrule "register
 west"    ruleflow-group "register

neighbor"when    $cell: Cell( $row : row, $col : col )
      $west : Cell( row  == $row, col == ( $col - 1 ) )
  then
      insert( new Neighbor( $cell, $west )
 );    insert( new Neighbor( $west, $cell ) );
```

Once all the cells are inserted, some Java code applies the pattern to the grid, setting certain cells to Live. Then, when the user clicks "Start" or "Next Generation", it executes the "Generation" ruleflow. This ruleflow is responsible for the management of all changes of cells in each generation cycle.

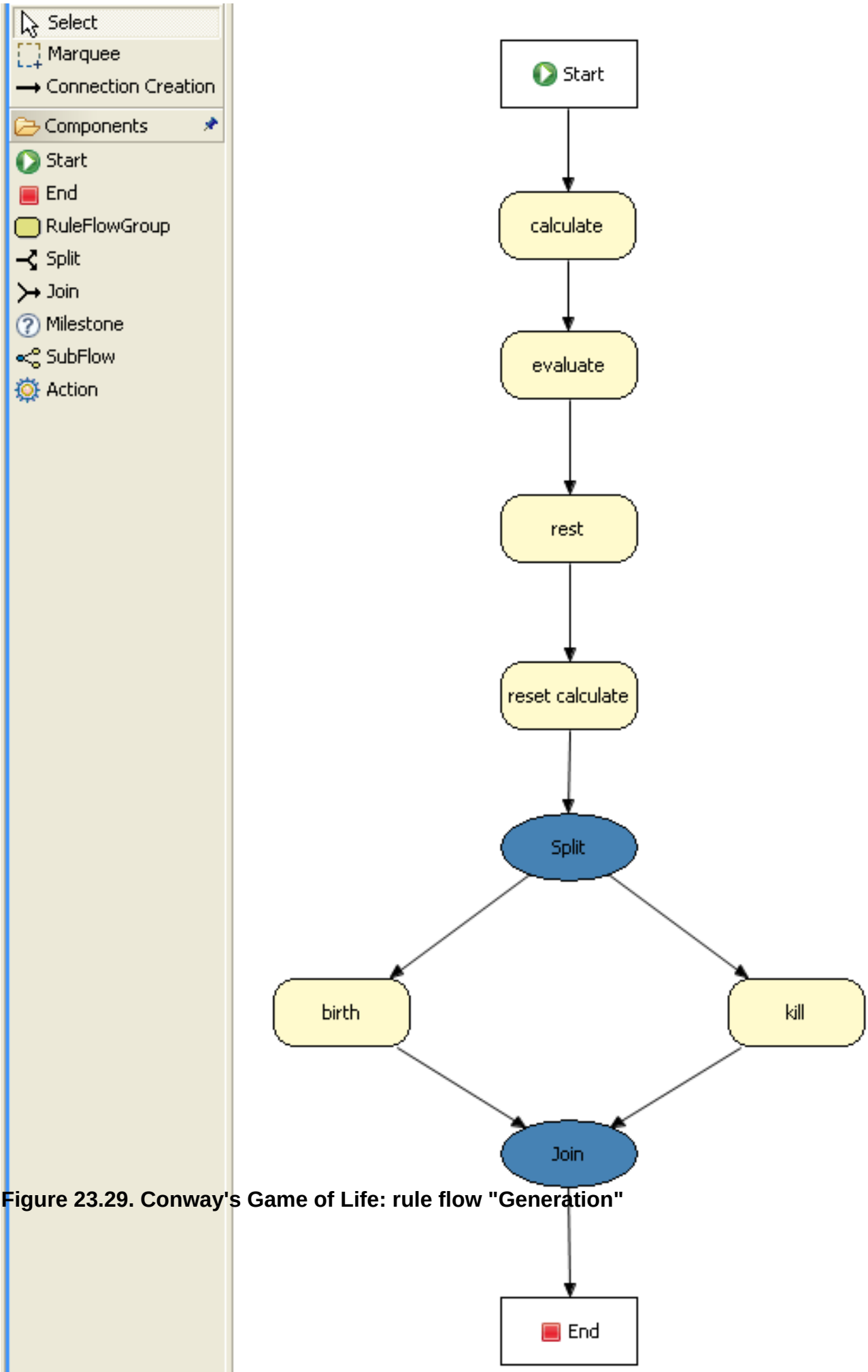


Figure 23.29. Conway's Game of Life: rule flow "Generation"

The rule flow process first enters the "evaluate" group, which means that any active rule in the group can fire. The rules in this group apply the Game-of-Life rules discussed in the beginning of the example, determining the cells to be killed and the ones to be given life. We use the "phase" attribute to drive the reasoning of the Cell by specific groups of rules; typically the phase is tied to a Rule Flow Group in the Rule Flow process definition. Notice that it doesn't actually change the state of any `Cell` objectss at this point; this is because it's evaluating the grid in turn and it must complete the full evaluation until those changes can be applied. To achieve this, it sets the cell to a "phase" which is either `Phase.KILL` or `Phase.BIRTH`, used later to control actions applied to the `Cell` object.

Example 23.72. Conway's Game of Life: Evaluate Cells with state changes

```
rule "Kill The Lonely"    ruleflow-group "evaluate"    no-loopwhen//  A live cell has fewer than
 2 live neighbors    theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE,
    phase == Phase.EVALUATE )then    modify( theCell ){       setPhase( Phase.KILL );    }endrule
 "Kill The Overcrowded"    ruleflow-group "evaluate"    no-loopwhen//  A live cell has more than
 3 live neighbors    theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE,
    phase == Phase.EVALUATE )then    modify( theCell ){       setPhase( Phase.KILL );    }endrule
 "Give Birth"    ruleflow-group "evaluate"    no-loopwhen//  A dead cell has 3 live neighbors
   theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,                    phase ==
 Phase.EVALUATE )then    modify( theCell ){        theCell.setPhase( Phase.BIRTH );    }end
ly"    ruleflow-group
"evaluate"    no-
loop
when//  A live cell has fewer than 2 live
neighbors    theCell: Cell( liveNeighbors < 2, cellState ==
CellState.LIVE,                   phase == Phase.EVALUATE
)
then    modify( theCell )
{        setPhase( Phase.KILL
);
}

endrule "Kill The
Overcrowded"    ruleflow-group
"evaluate"    no-
loop
when//  A live cell has more than 3 live
neighbors    theCell: Cell( liveNeighbors > 3, cellState ==
CellState.LIVE,                   phase == Phase.EVALUATE
)
then    modify( theCell )
{        setPhase( Phase.KILL
);
}

endrule "Give
Birth"    ruleflow-group
"evaluate"    no-
loop
when//  A dead cell has 3 live
neighbors    theCell: Cell( liveNeighbors == 3, cellState ==
CellState.DEAD,                   phase == Phase.EVALUATE
)
then    modify( theCell )
```

```
{        theCell.setPhase( Phase.BIRTH
);
}
```

Once all `Cell` objects in the grid have been evaluated, we first clear any calculation activations that occurred from any previous data changes. This is done via the "reset calculate" rule, which clears any activations in the "calculate" group. We then enter a split in the rule flow which allows any activations in both the "kill" and the "birth" group to fire. These rules are responsible for applying the state change.

Example 23.73. Conway's Game of Life: Apply the state changes

```
rule "reset calculate"      ruleflow-group "reset calculate"whenthen      WorkingMemory wm =
 drools.getWorkingMemory();    wm.clearRuleFlowGroup( "calculate" );endrule "kill"    ruleflow-
group "kill"    no-loopwhen    theCell: Cell( phase == Phase.KILL )then    modify( theCell )
{          setCellState( CellState.DEAD ),          setPhase( Phase.DONE );         }end  rule
 "birth"    ruleflow-group "birth"    no-loopwhen    theCell: Cell( phase == Phase.BIRTH )then
 modify( theCell ){       setCellState( CellState.LIVE ),       setPhase( Phase.DONE );    }end
calculate"    ruleflow-group
"reset

calculate"whenthen    WorkingMemory wm
= drools.getWorkingMemory();    wm.clearRuleFlowGroup(
"calculate"

 );end
rule "kill"    ruleflow-
group "kill"

no-loopwhen    theCell: Cell( phase ==
Phase.KILL
 )then    modify( theCell
 ){        setCellState(
CellState.DEAD ),        setPhase( Phase.DONE );


}
end
rule "birth"    ruleflow-
group "birth"

no-loopwhen    theCell: Cell( phase ==
Phase.BIRTH
 )then    modify( theCell
 ){        setCellState(
CellState.LIVE ),        setPhase(
Phase.DONE );
```

At this stage, a number of `Cell` objects have been modified with the state changed to either `LIVE` or `DEAD`. Now we get to see the power of the `Neighbor` facts defining the cell relations. When a cell becomes live or dead, we use the `Neighbor` relation to iterate over all surrounding cells,

increasing or decreasing the `liveNeighbor` count. Any cell that has its count changed is also set to to the `EVALUATE` phase, to make sure it is included in the reasoning during the evaluation stage of the Rule Flow Process. Notice that we don't have to do any iteration ourselves; simply by applying the relations in the rules we make the rule engine do all the hard work for us, with a minimal amount of code. Once the live count has been determined and set for all cells, the Rule Flow Process comes to and end. If the user has initially clicked the "Start" button, the engine will restart the rule flow; otherwise the user may request another generation.

Example 23.74. Conway's Game of Life: Evaluate cells with state changes

```
rule "Calculate Live"      ruleflow-group "calculate"      lock-on-active   when      theCell:
 Cell( cellState == CellState.LIVE )    Neighbor( cell == theCell, $neighbor : neighbor ) then
      modify( $neighbor ){            setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
          setPhase( Phase.EVALUATE );          }end rule "Calculate Dead"      ruleflow-group
  "calculate"      lock-on-active   when      theCell: Cell( cellState == CellState.DEAD )
  Neighbor( cell == theCell, $neighbor : neighbor )then      modify( $neighbor ){
 setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),       setPhase( Phase.EVALUATE );   }end
 Live"    ruleflow-group
 "calculate"    lock-on-active

 when    theCell: Cell( cellState == CellState.LIVE
 )    Neighbor( cell == theCell, $neighbor : neighbor
)
 then    modify( $neighbor
){        setLiveNeighbors( $neighbor.getLiveNeighbors() + 1
 ),        setPhase( Phase.EVALUATE );

 }

end rule "Calculate
 Dead"    ruleflow-group
 "calculate"    lock-on-active

 when    theCell: Cell( cellState == CellState.DEAD
 )    Neighbor( cell == theCell, $neighbor : neighbor

)then    modify( $neighbor
){        setLiveNeighbors( $neighbor.getLiveNeighbors() - 1
 ),        setPhase( Phase.EVALUATE
 );
 }
```

23.12. Invaders

A simplifed version of the Space Invaders game. Use the keys Z and K, to move left and right and M to fire a misile. The example is built up over 6 projects, each adding slightly more complexity to the last.

```
Name: Example Invaders
Main class: org.drools.games.invaders.Invaders1Main
Main class: org.drools.games.invaders.Invaders2Main
```

```
Main class: org.drools.games.invaders.Invaders3Main
Main class: org.drools.games.invaders.Invaders4Main
Main class: org.drools.games.invaders.Invaders5Main
Main class: org.drools.games.invaders.Invaders6Main
```

Figure 23.30. Pong Screenshot

23.12.1. Invaders1Main

Invaders1Main creates the frame and attaches the KeyListener, feeding key events into the engine. It also sets up the main game loop which can be found in "Main.drl". The typical convention used through out the example is to have one agenda group per file, and all rules in that file in the same agenda group.

The Run fact is used to drive the repeat of the Game loop. Initially there are only one groups that is evaluated, Keys. The "keys.drl" file is shared by several examples, and illustrates rule re-use across multipel projects.

Example 23.75. Game Loop

```
rule "init" when
then
    insert( new Run() );
    setFocus( "Init" );
end

rule GameLoop when
    r : Run()
then
    setFocus( "Keys" );
end


rule Draw when
    r : Run()
then
    ui.show();
    modify( r ) {} // force loop
end
```

23.12.2. Invaders2Main

Invaders2Main adds the "Draw" stage to the game loop and draws the SpaceShip

Example 23.76. Game Loop

```
rule GameLoop when
    r : Run()
then
    setFocus( "Draw" );
    setFocus( "Keys" );
end
```

23.12.3. Invaders3Main

Invaders3Main adds move controls to the spaceship, notice the ship moves out of the boundaries of the screen. KeyPressed is detected and that sets a delta of dx on the ship direction. That delta is then repeated applied to the x position of the ship

Example 23.77. Move Ship

```
rule ShipDeltaMoveLeft agenda-group "Move" when
```

```
    s : Ship()
        KeyPressed( keyText == "Z" )
then
    modify( s ) { dx = 0 - s.speed }
end

rule ShipDeltaStopLeft agenda-group "Move" when
    s : Ship()
        not KeyPressed( keyText == "Z" )
then
    modify( s ) { dx = 0 }
end

rule ShipMove agenda-group "Move" when
    s : Ship( dx != 0 )
    Run()
then
    modify( s ) { x = s.x + s.dx }
end
```

23.12.4. Invaders4Main

Invaders4Main adds boundari control to the ShipMove rule, so it doesn't move off the screen. Notice the use of "@watch(!x)", this ensures that while the rule wil modify the x property, it will not react to changes to x, which avoids recursion issues.

Example 23.78. Move Ship with Boundaries

```
rule ShipMove agenda-group "Move" when
    s : Ship( dx != 0, x + dx > 0,  x + dx + width < conf.windowWidth ) @watch( !x )
    Run()
then
    modify( s ) { x = s.x + s.dx }
end
```

23.12.5. Invaders5Main

Invaders5Main updates the "Draw" group to draw 5 Invaders.

23.12.6. Invaders6Main

Invaders6Main adds a lot more meat. Pressing the "M" key fires a missile that travels up the screen, while moving collision between the missile and the invader is checked.

23.12.7. Invaders4Main

Invaders4Main adds boundari control to the ShipMove rule, so it doesn't move off the screen. Notice the use of "@watch(!x)", this ensures that while the rule wil modify the x property, it will not react to changes to x, which avoids recursion issues.

Example 23.79. Fire Missile

```
rule InsertBullet agenda-group "Bullet" when
        KeyPressed( keyText == "M" )
    s : Ship()
    not Bullet()
then
    b = new Bullet();
    b.x = s.x + (s.width/2) - (b.width/2);
    b.y = s.y - s.height - b.height;
    b.width = conf.bulletWidth;
    b.height = conf.bulletHeight;
    b.dy = 0 - conf.bulletSpeed;
    insert( b );
end


rule BulletMove agenda-group "Bullet" when
    b : Bullet( y > 0 ) @watch( !y )
    Run()
then
    modify( b ) { y = b.y + b.dy }
end

rule Collision agenda-group "Bullet" when
    b : Bullet( ) @watch( y )
    i : Invader( x < b.x, x + width > b.x, y > b.y)
    Run()
then
    modify( i ) { alive = false }
end
```

23.13. Adventures with Drools

Based on the Adventure in Prolog, over at the Amzi website, http://www.amzi.com/AdventureInProlog/, we started to work on a text adventure game for Drools. They are ideal as they can start off simple and build in complexity and size over time, they also demonstrate key aspects of declarative relational programming.

```
Name: Example Text Adventure
Main class: org.drools.games.adventure.TextAdventure
```

The game allows you to play as the hero or the monster. If you click "New Window" you can open one window as the hero and another as the monster, and play them both at the same time. The game allows either character to move around rooms, pick up, drop or use things. Doors can be locked and unlocked, by using the key on teh exit room, and the hero can kill the monster by using the umbrella on the monster.

You can view the 8 minute demonstration and introduction for the example at http://downloads.jboss.org/drools/videos/text-adventures.swf. Be aware the video is now much older than the current improved example.

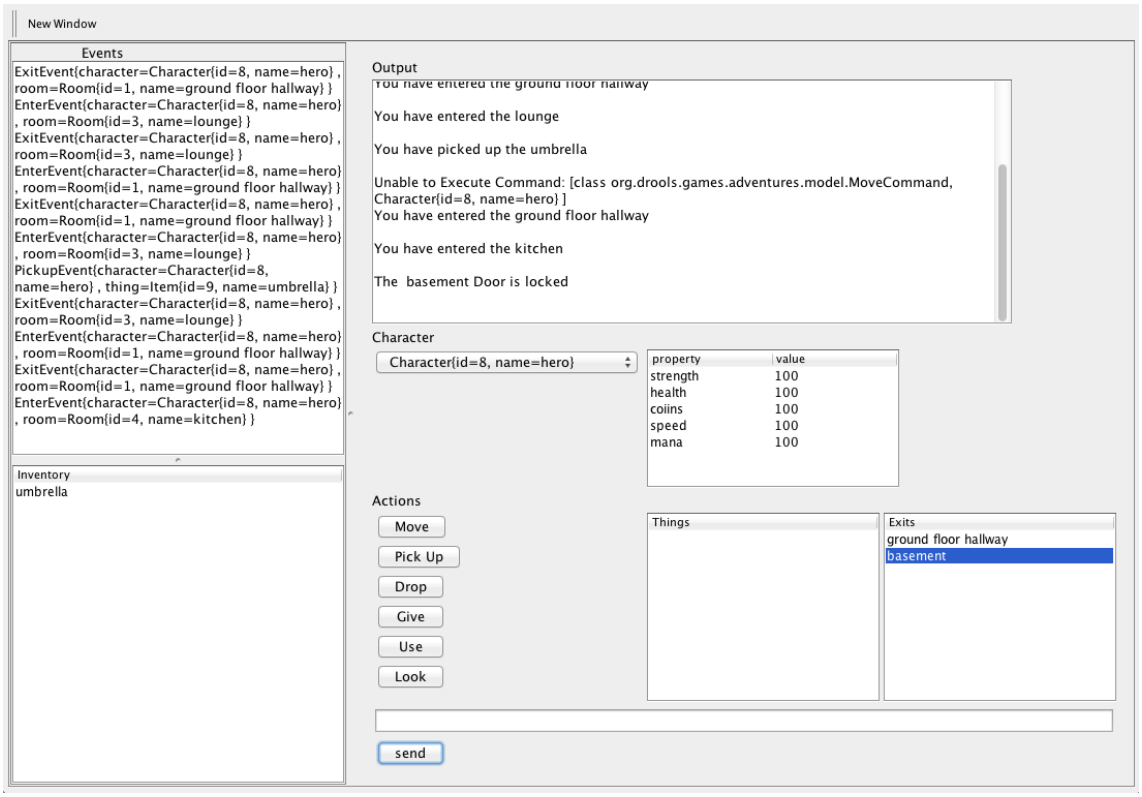


Figure 23.31. Text Adventure Screenshot

23.13.1. Using the game.

Each action follows the constructor arguments of the associated Command java class.

Example 23.80. MoveCommand

```
@PropertyReactive
public class MoveCommand extends Command {

    @Position(1)
    private Character character;

    @Position(2)
```

```
    private Room      room;

    public MoveCommand(Character character, Room room) {
        this.character = character;
        this.room = room;
    }
```

To issue a move action, select the "Move" button, then select the exit room. Notice when you press "Move" it adds the text to the white bar at the bottom. When the exit room is selected, it also is added to the white bar. Then press send and the game engine will execute the command. Internally it uses reflection to instantiate the Command and insert it into the engine. If you select incorrect arguments, such as pressing exits multiple times, the reflection will fail and you can attempt it again.

Figure 23.32. Move Action

The Things list displays anything you can see in the room, not all things can be picked up. For instance you can pick up the key and the torch, but not the monster. When something is picked up it moves from the Things list to the Inventory List. The reverse is true when something is dropped.

Figure 23.33. Pickup Action

The key is in the office, move upstairs and into the office. Then pick up the key. Move back downstairs and into the kitchen. Try and walk into the basement, notice it's locked.

Select the "Use" action, the select the key and then the basement exit. This will unlock the door and you can now walk through.

To kill the monster pick up the umbrella from the lounge and then select "Use", then select the imbrella and finally select the monster.

Don't forget to open a "New Window" to play as the monster, although you will not be able to exit the basement until the hero has opened it with the key. The monster and the hero can also give items to each other, moving items between each playsers inventory.

23.13.2. The code

The model is written in Java classes. Each classes uses @PropertyReactive and @Position. @PropertyReactive allows control of which fields patterns react to, and @Position maps a field to a argument position allowing positional as well as named arguments for patterns.

Example 23.81. Game World Data Example

```
@PropertyReactive
public class Thing {
    @Position(0)
    private long id;

    @Position(1)
    private String name;

    public Thing(long id, String name) {
        this.id = id;
        this.name = name;
    }
    ...
}
```

An MVEL data file is used to populate our world, see "data.mvel". You can edit this file to add new rooms, items and characters, as well as locks for doors.

Example 23.82. Game World Data Example

```
rooms = [
    "basement" : new Room("basement"),
    "lounge" : new Room("lounge"),
    "kitchen" : new Room("kitchen"),
    "ground floor hallway" : new Room("ground floor hallway"),
    "bedroom" : new Room("bedroom"),
    "office" : new Room("office"),
    "first floor hallway" : new Room("first floor hallway")
];

doors = [
    "d1" : new Door( rooms["kitchen"], rooms["basement"] ),
    "d2" : new Door( rooms["ground floor hallway"], rooms["lounge"]),
    "d4" : new Door( rooms["ground floor hallway"], rooms["kitchen"]),
    "d5" : new Door( rooms["ground floor hallway"], rooms[ "first floor hallway"] ),
    "d8" : new Door( rooms["first floor hallway"], rooms[ "bedroom"] ),
    "d9" : new Door( rooms["first floor hallway"], rooms[ "office"] )
];

locations = [
    "monster" :  new Location( characters["monster"], rooms["basement"] ),
```

```
    "hero" :  new Location( characters["hero"], rooms["ground floor hallway"] ),
    "umbrella" :  new Location( items["umbrella"], rooms["lounge"] ),
    "key1" :  new Location( items["key1"], rooms["office"] )
];
```

The game creates commands, which it inserts into the engine. These commands are then used to change the state of the world and that state is reflected back in the UI. The commands can be found in the "commands.drl" file. The following rule matches the MoveCommand and if it's valid it will make the move happen.

Example 23.83. Move a Characters

```
rule validMove agenda-group "commands" when
    mc : MoveCommand( c : character, r : room )
    l  : Location( thing == c, ltarget : target ) @watch( !target )
    ?connect( d, r, ltarget; )
then
    exit = new ExitEvent( c, (Room) l.target );
    enter = new EnterEvent( c, r );

    modify( l ) { target = r };

    insert( exit );
    insert( enter );

    mc.session.channels["output"].send( "You have entered the " + l.target.name + "\n" );
end
```

In the above rules notice the "connect" pattern, this is actually a query. In the MVEL data file doors are only described one way, we can use a query to check connections bi-directionally. The queries can be found in the "queries.drl" file.

Example 23.84. connect

```
query connect( Door $d, Room $x, Room $y )
    $d := Door($id, $name, $x, $y;)
    or
    $d :=Door($id, $name, $y, $x;)
end
```

The UI has its list boxes populated by rules found in "UiView.drl", those rules in turn use queries. Here is how the "Things" list box is populated, when ever the world changes.

Example 23.85. Update the UI

```
rule updateThings salience 5  when
    session : UserSession( $char : character )
    things( $char, $things; )
then
    session.channels["things"].send( $things );
end

query things(Character $char, List $things)
    $char := Character()
    Location( $char, $room; )
    $things := List() from accumulate( Location($thing, $room; thing != $char),
                                       collectList( $thing ) )
end
```

23.14. Pong

A Conversion for the classic game Pong. Use the keys A, Z and K, M. The ball should get faster after each bounce.

```
Name: Example Pong
Main class: org.drools.games.pong.PongMain
```

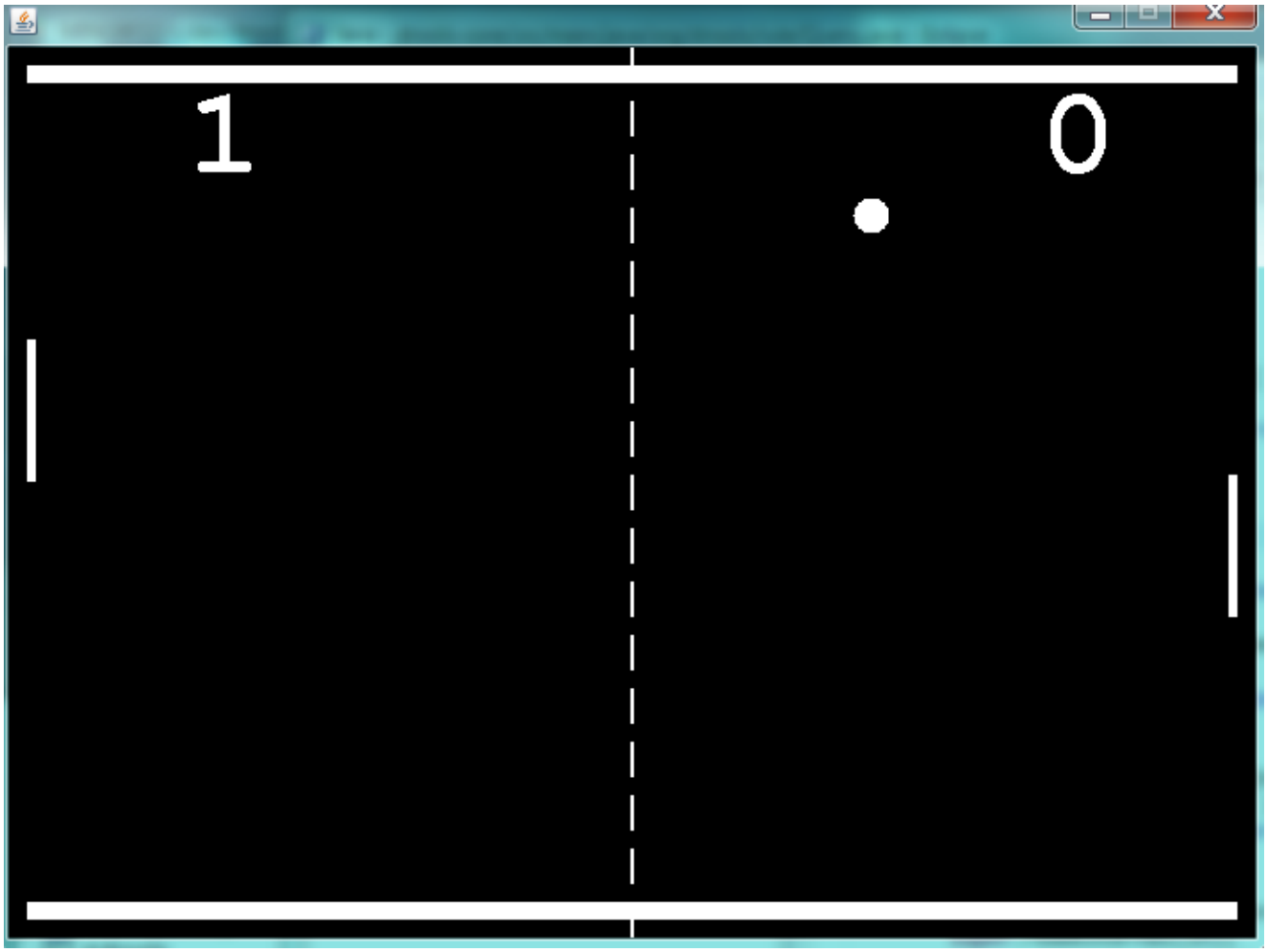
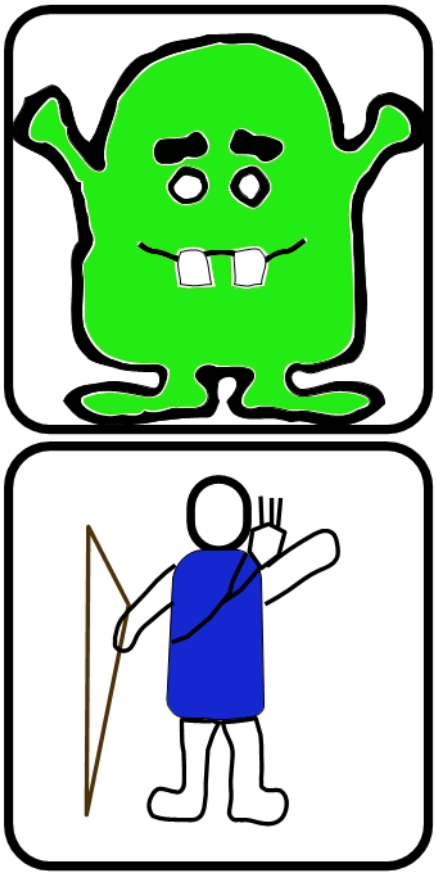


Figure 23.34. Pong Screenshot

23.15. Wumpus World

```
Name: Example Wumpus World
Main class: org.drools.games.wumpus.WumpusWorldMain
```

Wumpus World is an AI example covered in the book "Artificial Intelligence : A Modern Approach". When the game first starts all the cells are greyed out. As you walk around they become visible. The cave has pits, a wumpus and gold. When you are next to a pit you will feel a breeze, when you are next to the wumpus you will smell a stench and see glitter when next to gold. The sensor icons are shown above the move buttons. If you walk into a pit or the wumpus, you die. A more detailed overview of Wumpus World can be found at http://www.cis.temple.edu/~giorgio/cis587/readings/wumpus.shtml. A 20 minute video showing how the game is created and works is at http://www.youtube.com/watch?v=4CvjKqUOEzM. [http://www.youtube.com/watch?v=4CvjKqUOEzM]



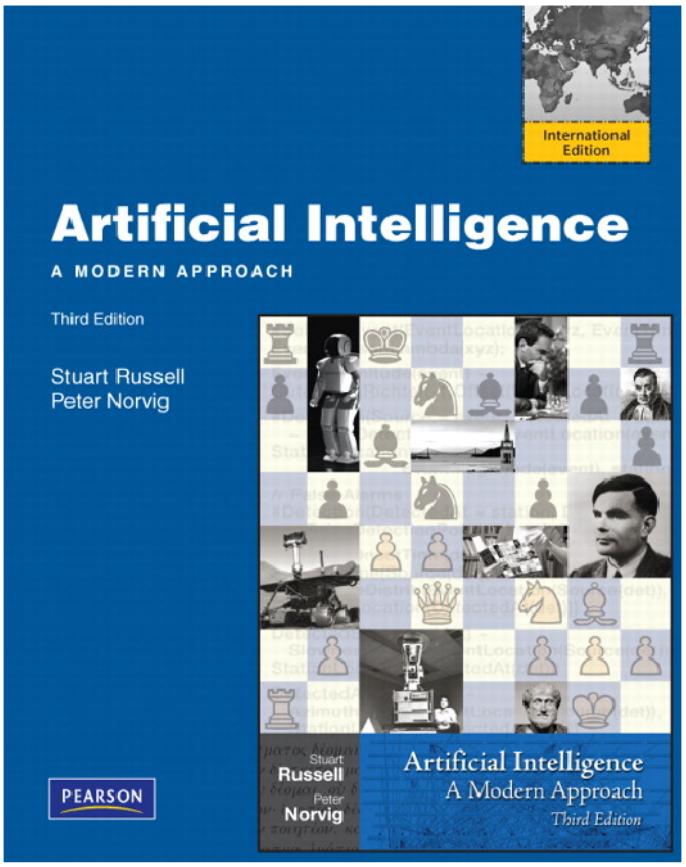


Figure 23.35. Wumpus World

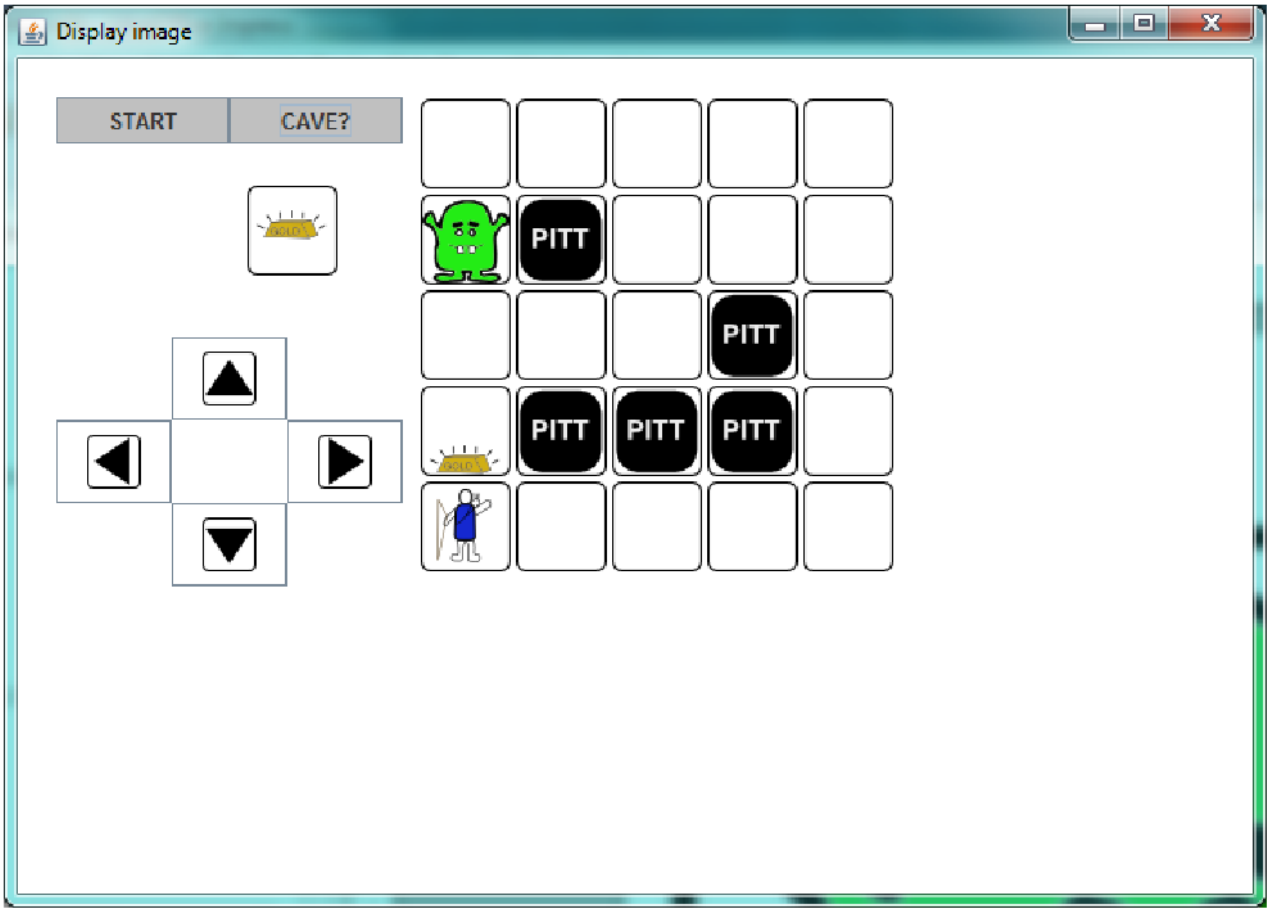


Figure 23.36. Cave Screenshot

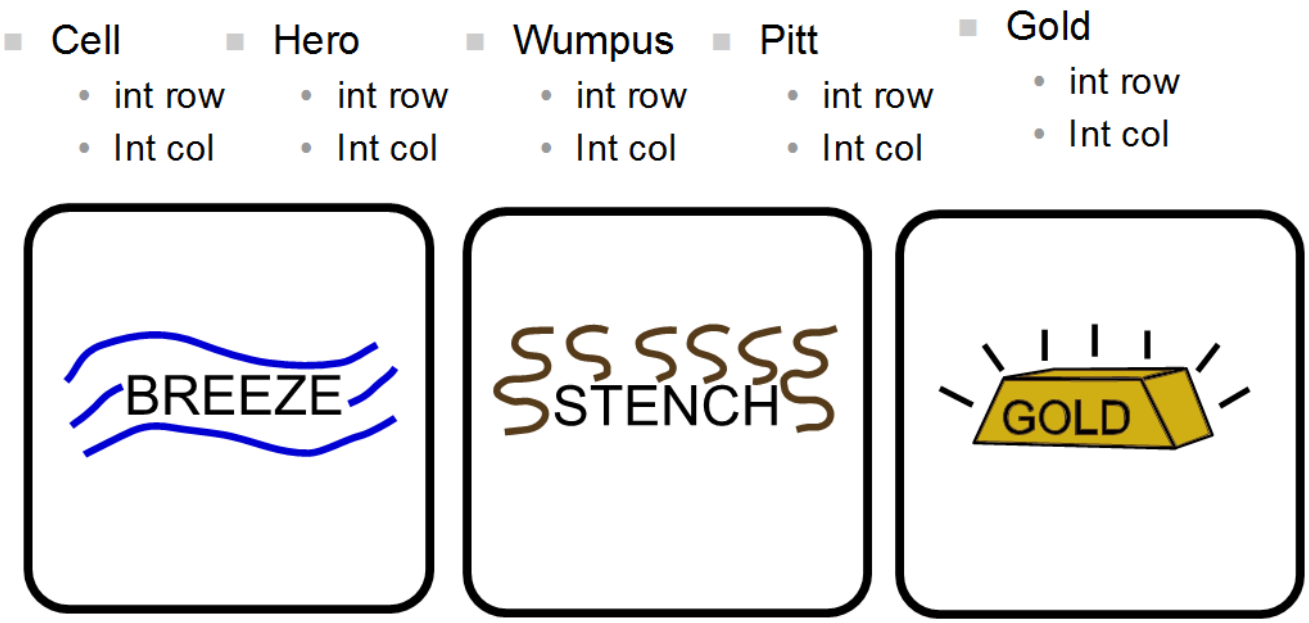


Figure 23.37. Signals Screenshot

```
rule "Smell Stench"  when
    $s  : Sensors()
    $h  : Hero()
          Wumpus(row == ($h.row + 1), col == $h.col ) or
          Wumpus(row == ($h.row - 1), col == $h.col ) or
          Wumpus(row == $h.row, col == ($h.col + 1) ) or
          Wumpus(row == $h.row, col == ($h.col - 1) )
then
    insertLogical( new SmellStench() );
    $s.smellStench = true;
end
```

Figure 23.38. Smell Stench

```
rule "Move Up"  when
    $mc : MoveCommand( move == Move.UP )
    $h  : Hero()
    $c  : Cell(row == ($h.row + 1), col == $h.col )
then
    modify( $h ) { row = $h.row + 1 };
    $c.setHidden( false );
    retract ( $mc );
end

rule "Wumpus Death"  when
    $g  : GameData()
    $h  : Hero()
          Wumpus(row == $h.row, col == $h.col )
then
    $g.wumpusDeath = true;
end
```

Figure 23.39. Move Up, Wumpus Collision

23.16. Miss Manners and Benchmarking

Name: Miss Manners
Main class: org.drools.benchmark.manners.MannersBenchmark
Module: drools-examples
Type: Java application

```
Rules file: manners.drl
Objective: Advanced walkthrough on the Manners benchmark, covers Depth conflict resolution in
 depth.
```

23.16.1. Introduction

Miss Manners is throwing a party and, being a good host, she wants to arrange good seating. Her initial design arranges everyone in male-female pairs, but then she worries about people have things to talk about. What is a good host to do? She decides to note the hobby of each guest so she can then arrange guests not only pairing them according to alternating sex but also ensuring that a guest has someone with a common hobby, at least on one side.

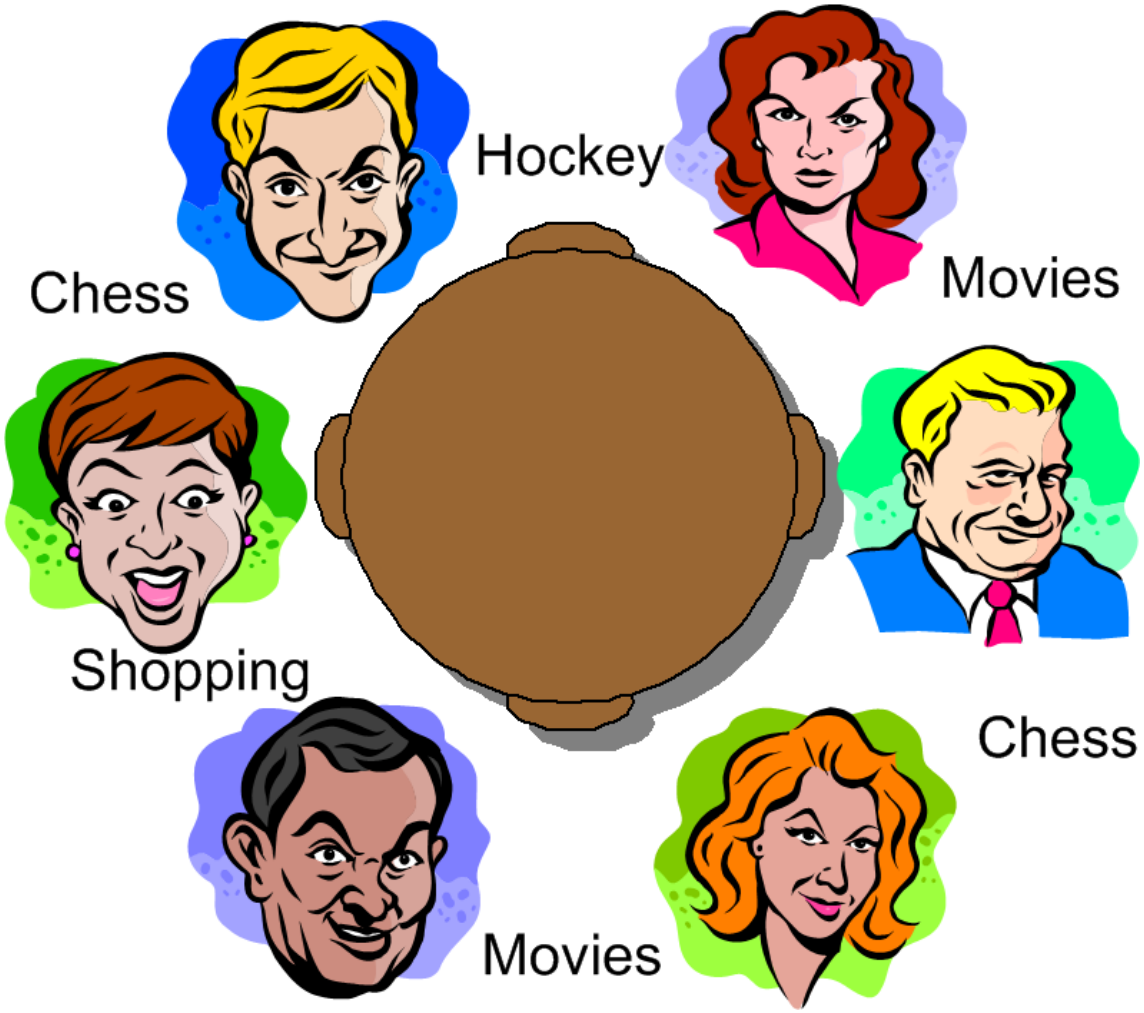


Figure 23.40. Miss Manners' Guests

23.16.1.1. BenchMarking

Five benchmarks were established in the 1991 paper "Effects of Database Size on Rule System Performance: Five Case Studies" by David Brant, Timothy Grose, Bernie Lofaso and Daniel P. Miranker:

- **Manners** uses a depth-first search approach to determine the seating arrangements alternating women and men and ensuring one common hobby for neighbors.
- **Waltz** establishes a three-dimensional interpretation of a line drawing by line labeling by constraint propagation.
- **WaltzDB** is a more general version of Waltz, supporting junctions of more than three lines and using a database.
- **ARP** is a route planner for a robotic air vehicle using the A* search algorithm to achieve minimal cost.
- **Weaver** VLSI router for channels and boxes using a black-board technique.

Manners has become the de facto rule engine benchmark. Its behavior, however, is now well known and many engines optimize for this, thus negating its usefulness as a benchmark which is why Waltz is becoming more favorable. These five benchmarks are also published at the University of Texas http://www.cs.utexas.edu/ftp/pub/ops5-benchmark-suite/.

23.16.1.2. Miss Manners Execution Flow

After the first seating arrangement has been assigned, a depth-first recursion occurs which repeatedly assigns correct seating arrangements until the last seat is assigned. Manners uses a `Context` instance to control execution flow. The activity diagram is partitioned to show the relation of the rule execution to the current `Context` state.

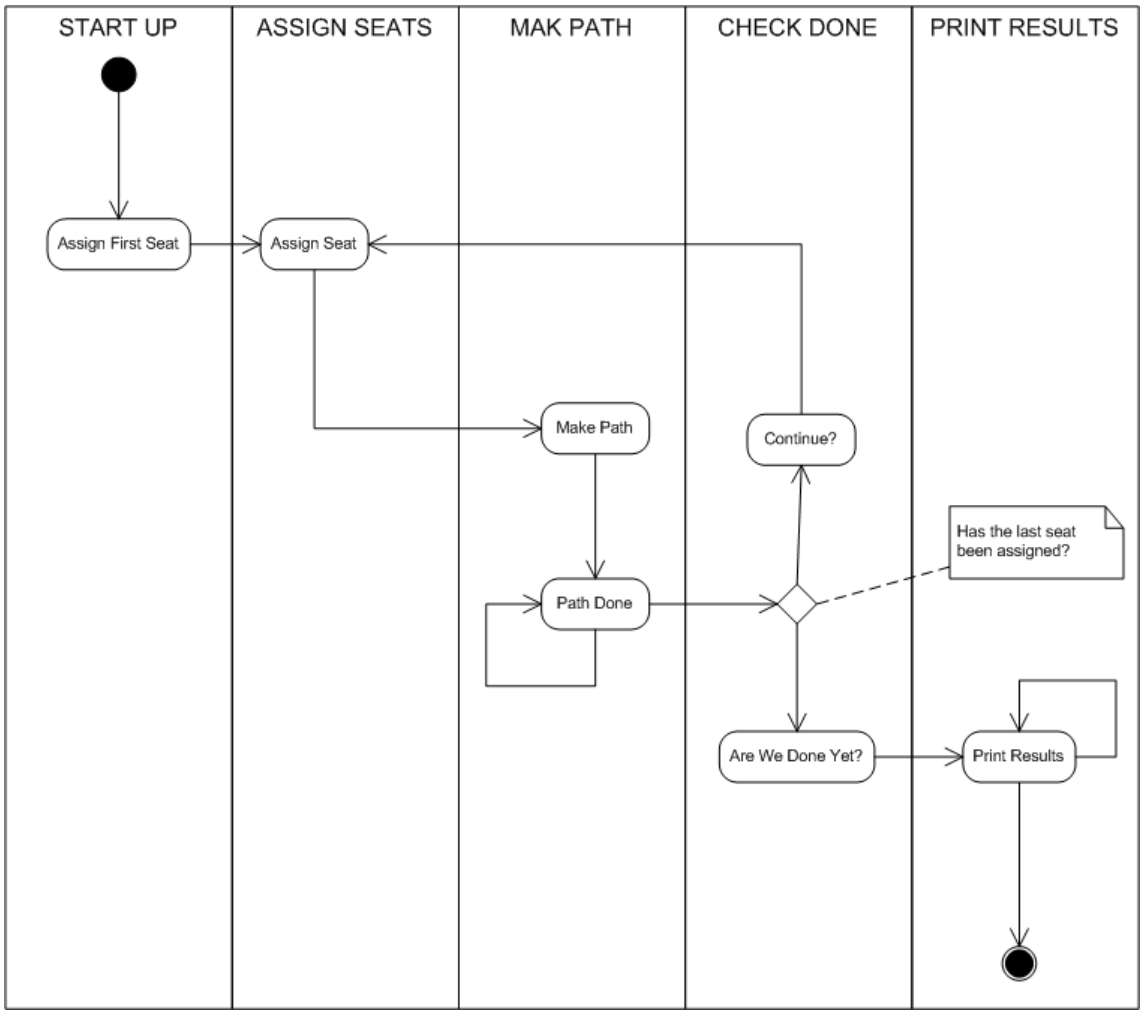


Figure 23.41. Manners Activity Diagram

23.16.1.3. The Data and Results

Before going deeper into the rules, let's first take a look at the asserted data and the resulting seating arrangement. The data is a simple set of five guests who should be arranged so that sexes alternate and neighbors have a common hobby.

The Data

The data is given in OPS5 syntax, with a parenthesized list of name and value pairs for each attribute. Each person has only one hobby.

```
(guest (name n1) (sex m) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h1)  )
(guest (name n2) (sex f) (hobby  h3)  )
(guest (name n3) (sex m) (hobby  h3)  )
(guest (name n4) (sex m) (hobby  h1)  )
(guest (name n4) (sex f) (hobby  h2)  )
```

```
(guest (name n4) (sex f) (hobby  h3)  )
(guest (name n5) (sex f) (hobby  h2)  )
(guest (name n5) (sex f) (hobby  h1)  )
(last_seat (seat 5)  )
```

The Results

Each line of the results list is printed per execution of the "Assign Seat" rule. They key bit to notice is that each line has a "pid" value one greater than the last. (The significance of this will be explained in the discussion of the rule "Assign Seating".) The "ls", "rs", "ln" and "rn" refer to the left and right seat and neighbor's name, respectively. The actual implementation uses longer attribute names (e.g., `leftGuestName`, but here we'll stick to the notation from the original implementation.

```
[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[Seating id=4, pid=3, done=false, ls=3, rn=n3, rs=4, rn=n2]
[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
```

23.16.2. In depth Discussion

23.16.2.1. Cheating

Manners has been designed to exercise cross product joins and Agenda activities. Many people not understanding this tweak the example to achieve better performance, making their port of the Manners benchmark pointless. Known cheats or porting errors for Miss Manners are:

- Using arrays for a guests hobbies, instead of asserting each one as a single fact massively reduces the cross products.
- Altering the sequence of data can also reduce the amount of matching, increasing execution speed.
- It's possible to change the `not` Conditional Element so that the test algorithm only uses the "first-best-match", which is, basically, transforming the test algorithm to backward chaining. The results are only comparable to other backward chaining rule engines or ports of Manners.
- Removing the context so the rule engine matches the guests and seats prematurely. A proper port will prevent facts from matching using the context start.
- It's possible to prevent the rule engine from performing combinatorial pattern matching.
- If no facts are retracted in the reasoning cycle, as a result of the `not` CE, the port is incorrect.

23.16.2.2. Conflict Resolution

The Manners benchmark was written for OPS5 which has two conflict resolution strategies, LEX and MEA. LEX is a chain of several strategies including salience, recency and complexity. The

recency part of the strategy drives the depth first (LIFO) firing order. The CLIPS manual documents the Recency strategy as follows:

> Every fact and instance is marked internally with a "time tag" to indicate its relative recency with respect to every other fact and instance in the system. The pattern entities associated with each rule activation are sorted in descending order for determining placement. An activation with a more recent pattern entity is placed before activations with less recent pattern entities. To determine the placement order of two activations, compare the sorted time tags of the two activations one by one starting with the largest time tags. The comparison should continue until one activation's time tag is greater than the other activation's corresponding time tag. The activation with the greater time tag is placed before the other activation on the agenda. If one activation has more pattern entities than the other activation and the compared time tags are all identical, then the activation with more time tags is placed before the other activation on the agenda.
>
> —CLIPS Reference Manual

However Jess and CLIPS both use the Depth strategy, which is simpler and lighter, which Drools also adopted. The CLIPS manual documents the Depth strategy as:

> Newly activated rules are placed above all rules of the same salience. For example, given that fact-a activates rule-1 and rule-2 and fact-b activates rule-3 and rule-4, then if fact-a is asserted before fact-b, rule-3 and rule-4 will be above rule-1 and rule-2 on the agenda. However, the position of rule-1 relative to rule-2 and rule-3 relative to rule-4 will be arbitrary.
>
> —CLIPS Reference Manual

The initial Drools implementation for the Depth strategy would not work for Manners without the use of salience on the "make_path" rule. The CLIPS support team had this to say:

> The default conflict resolution strategy for CLIPS, Depth, is different than the default conflict resolution strategy used by OPS5. Therefore if you directly translate an OPS5 program to CLIPS, but use the default depth conflict resolution strategy, you're only likely to get the correct behavior by coincidence. The LEX and MEA conflict resolution strategies are provided in CLIPS to allow you to quickly convert and correctly run an OPS5 program in CLIPS.
>
> —Clips Support Forum

Investigation into the CLIPS code reveals there is undocumented functionality in the Depth strategy. There is an accumulated time tag used in this strategy; it's not an extensively fact by fact comparison as in the recency strategy, it simply adds the total of all the time tags for each activation and compares.

23.16.2.3. Rule "assignFirstSeat"

Once the context is changed to `START_UP`, activations are created for all asserted guest. Because all activations are created as the result of a single Working Memory action, they all have the same

Activation time tag. The last asserted `Guest` object would have a higher fact time tag, and its Activation would fire because it has the highest accumulated fact time tag. The execution order in this rule has little importance, but has a big impact in the rule "Assign Seat". The activation fires and asserts the first `Seating` arrangement and a `Path`, and then sets the `Context` attribute `state` to create an activation for rule `findSeating`.

```
rule assignFirstSeat    when       context : Context( state == Context.START_UP )       guest :
 Guest()         count : Count()     then         String guestName = guest.getName();
         Seating seating =           new Seating( count.getValue(), 1, true, 1, guestName, 1,
 guestName);        insert( seating );               Path path = new Path( count.getValue(), 1,
 guestName );       insert( path );               modify( count ) { setValue ( count.getValue()
 + 1 )  }    System.out.println( "assign first seat :  " + seating + " : " + path );
 modify( context ) {            setState( Context.ASSIGN_SEATS )        } end
signFirstSeat
 when        context : Context( state == Context.START_UP
 )        guest :
 Guest()        count :
 Count()
 then        String guestName =
 guest.getName();
         Seating seating
 =          new Seating( count.getValue(), 1, true, 1, guestName, 1,
 guestName);        insert( seating
 );
         Path path = new Path( count.getValue(), 1, guestName
 );        insert( path
 );
         modify( count ) { setValue ( count.getValue() + 1 )

 }    System.out.println( "assign first seat :  " + seating + " : " + path

 );       modify( context )
 {           setState( Context.ASSIGN_SEATS
 )
}
```

23.16.2.4. Rule "findSeating"

This rule determines each of the `Seating` arrangements. The rule creates cross product solutions for *all* asserted `Seating` arrangements against *all* the asserted guests except against itself or any already assigned chosen solutions.

```
rule findSeating   when         context : Context( state == Context.ASSIGN_SEATS )        $s
      : Seating( pathDone == true )        $g1      : Guest( name == $s.rightGuestName )
  $g2      : Guest( sex != $g1.sex, hobby == $g1.hobby )        count    : Count()        not
 ( Path( id == $s.id, guestName == $g2.name) )        not ( Chosen( id == $s.id, guestName ==
 $g2.name, hobby == $g1.hobby) )   then      int rightSeat = $s.getRightSeat();      int seatId
 = $s.getId();       int countValue = count.getValue();              Seating seating =
 new Seating( countValue, seatId, false, rightSeat,                    $s.getRightGuestName(),
 rightSeat + 1, $g2.getName() );       insert( seating );                                  Path
 path = new Path( countValue, rightSeat + 1, $g2.getName()  );       insert( path );
     Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby() );       insert( chosen
```

```
  );        System.err.println( "find seating : " + seating + " : " + path +
               " : " + chosen);        modify( count ) {setValue(  countValue + 1 )}
 modify( context ) {setState( Context.MAKE_PATH )}end
Seating
when        context : Context( state == Context.ASSIGN_SEATS
 )       $s      : Seating( pathDone == true
 )       $g1     : Guest( name == $s.rightGuestName
 )       $g2     : Guest( sex != $g1.sex, hobby == $g1.hobby

 )       count   :

 Count()       not ( Path( id == $s.id, guestName == $g2.name)
 )       not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby)
 )
 then       int rightSeat =
 $s.getRightSeat();       int seatId =
 $s.getId();       int countValue =
 count.getValue();
        Seating seating
 =         new Seating( countValue, seatId, false,
 rightSeat,                      $s.getRightGuestName(), rightSeat + 1, $g2.getName()
 );       insert( seating
 );
        Path path = new Path( countValue, rightSeat + 1, $g2.getName()
 );       insert( path
 );
        Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby()
 );       insert( chosen

 );       System.err.println( "find seating : " + seating + " : " + path
 +                          " : " +

 chosen);       modify( count ) {setValue(  countValue + 1
 )}       modify( context ) {setState( Context.MAKE_PATH
```

However, as can be seen from the printed results shown earlier, it is essential that only the `Seating` with the highest `pid` cross product be chosen. How can this be possible if we have activations, of the same time tag, for nearly all existing `Seating` and `Guest` objects? For example, on the third iteration of `findSeating` the produced activations will be as shown below. Remember, this is from a very small data set, and with larger data sets there would be many more possible activated `Seating` solutions, with multiple solutions per `pid`:

```
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3]

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]
```

```
=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]
```

The creation of all these redundant activations might seem pointless, but it must be remembered that Manners is not about good rule design; it's purposefully designed as a bad ruleset to fully stress-test the cross product matching process and the Agenda, which this clearly does. Notice that each activation has the same time tag of 35, as they were all activated by the change in the `Context` object to `ASSIGN_SEATS`. With OPS5 and LEX it would correctly fire the activation with the `Seating` asserted last. With Depth, the accumulated fact time tag ensures that the activation with the last asserted `Seating` fires.

23.16.2.5. Rules "makePath" and "pathDone"

Rule `makePath` must always fire before `pathDone`. A `Path` object is asserted for each `Seating` arrangement, up to the last asserted `Seating`. Notice that the conditions in `pathDone` are a subset of those in `makePath` - so how do we ensure that `makePath` fires first?

```
rule makePath
    when
        Context( state == Context.MAKE_PATH )
        Seating( seatingId:id, seatingPid:pid, pathDone == false )
        Path( id == seatingPid, pathGuestName:guestName, pathSeat:seat )
        not Path( id == seatingId, guestName == pathGuestName )
    then
        insert( new Path( seatingId, pathSeat, pathGuestName ) );
end
```

```
rule pathDone    when        context : Context( state == Context.MAKE_PATH )         seating :
 Seating( pathDone == false )      then         modify( seating ) {setPathDone( true )}
       modify( context ) {setState( Context.CHECK_DONE)}end
Done
when        context : Context( state == Context.MAKE_PATH )
         seating : Seating( pathDone == false )

then        modify( seating ) {setPathDone( true )}

     modify( context ) {setState(
Context.CHECK_DONE)}
```

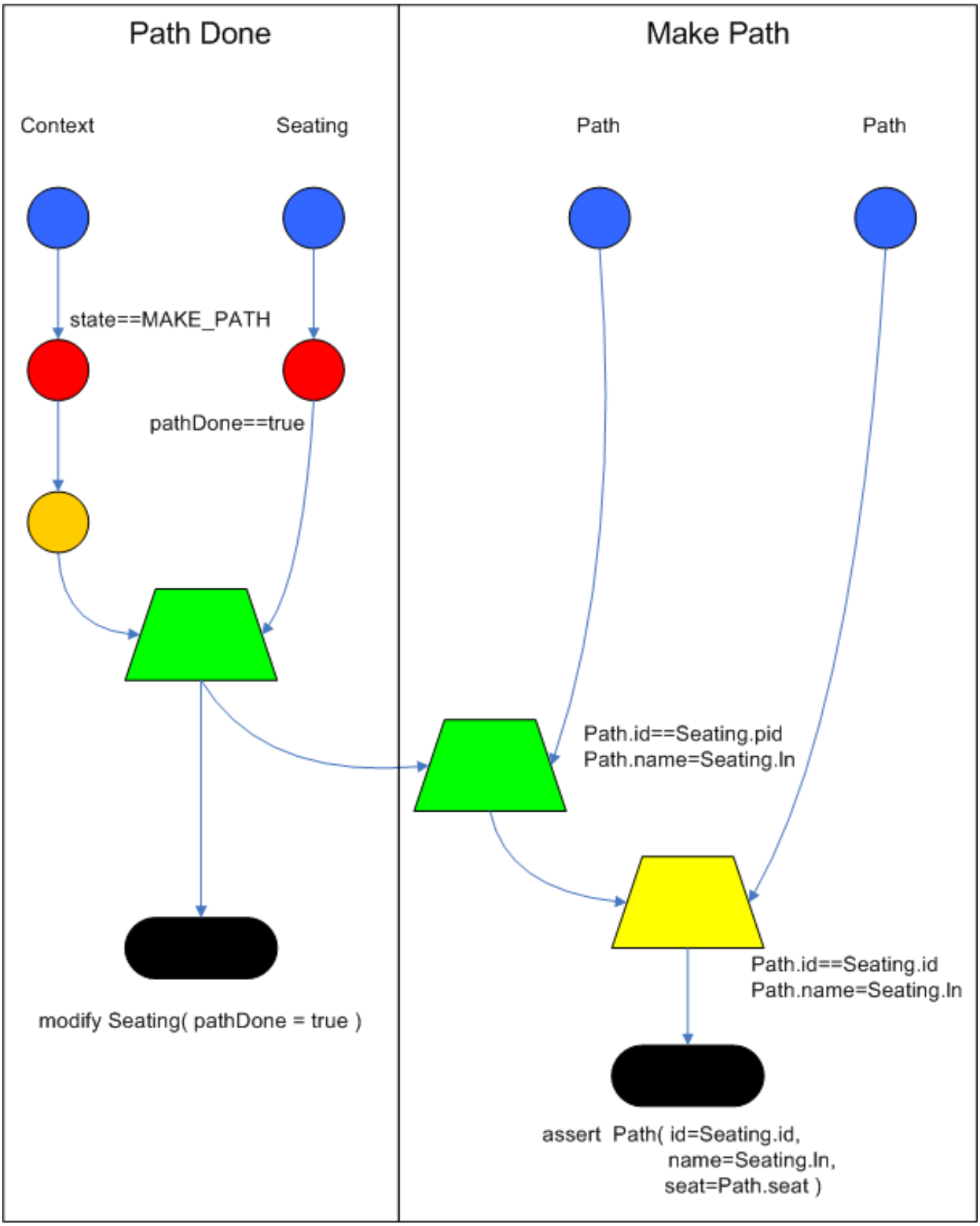


Figure 23.42. Rete Diagram

Both rules end up on the Agenda in conflict and with identical activation time tags. However, the accumulate fact time tag is greater for "Make Path" so it gets priority.

23.16.2.6. Rules "continue" and "areWeDone"

Rule `areWeDone` only activates when the last seat is assigned, at which point both rules will be activated. For the same reason that `makePath` always wins over `path Done`, `areWeDone` will take priority over rule `continue`.

```
rule areWeDone     when          context : Context( state == Context.CHECK_DONE )
 LastSeat( lastSeat: seat )      Seating( rightSeat == lastSeat )    then      modify( context )
 {setState(Context.PRINT_RESULTS )}end
Done
when        context : Context( state == Context.CHECK_DONE )
         LastSeat( lastSeat: seat
)        Seating( rightSeat == lastSeat )

then        modify( context ) {setState(Context.PRINT_RESULTS
)}
```

```
rule continue    when         context : Context( state == Context.CHECK_DONE )     then
   modify( context ) {setState( Context.ASSIGN_SEATS )}end
tinue
 when        context : Context( state == Context.CHECK_DONE
)
 then        modify( context ) {setState( Context.ASSIGN_SEATS
```

23.16.3. Output Summary

Assign First seat
=>[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
=>[fid:14:14]:[Path id=1, seat=1, guest=n5]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1 , pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]*

Assign Seating
=>[fid:15:17] :[Seating id=2 , pid=1 , done=false, ls=1, lg=n5, rs=2, rn=n4]

=>[fid:16:18]:[Path id=2, seat=2, guest=n4]
=>[fid:17:19]:[Chosen id=1, name=n4, hobbies=h1]

=>[ActivationCreated(21): rule=makePath
[fid:15:17] : [Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[fid:14:14] : [Path id=1, seat=1, guest=n5]*

==>[ActivationCreated(21): rule=pathDone
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]*

Make Path
=>[fid:18:22:[Path id=2, seat=1, guest=n5]]

Path Done

Continue Process
=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:7:7]:[Guest name=n4, sex=f, hobbies=h3]
[fid:4:4] : [Guest name=n3, sex=m, hobbies=h3]*

=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1], [fid:12:20] : [Count value=3]

=>[ActivationCreated(25): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating
=>[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, lnn4, rs=3, rn=n3]]
=>[fid:20:27]:[Path id=3, seat=3, guest=n3]]
=>[fid:21:28]:[Chosen id=2, name=n3, hobbies=h3}]

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:18:22]:[Path id=2, seat=1, guest=n5]*

=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:16:18]:[Path id=2, seat=2, guest=n4]*

=>[ActivationCreated(30): rule=done
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]*

Make Path
=>[fid:22:31]:[Path id=3, seat=1, guest=n5]

Make Path
=>[fid:23:32] [Path id=3, seat=2, guest=n4]

Path Done

Continue Processing
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3], [fid:12:29]*

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1], [fid:1:1] : [Guest name=n1, sex=m,
hobbies=h1]

Assign Seating
=>[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]]
=>[fid:25:37]:[Path id=4, seat=4, guest=n2]]
=>[fid:26:38]:[Chosen id=3, name=n2, hobbies=h3]

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:23:32]:[Path id=3, seat=2, guest=n4]*

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:20:27]:[Path id=3, seat=3, guest=n3]*

=>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:22:31]:[Path id=3, seat=1, guest=n5]*

```
=>[ActivationCreated(40): rule=done
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]*

Make Path
=>fid:27:41:[Path id=4, seat=2, guest=n4]

Make Path
=>fid:28:42]:[Path id=4, seat=1, guest=n5]]

Make Path
=>fid:29:43]:[Path id=4, seat=3, guest=n3]]

Path Done

Continue  Processing
=>[ActivationCreated(46): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1], [fid:2:2]
[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(46): rule=findSeating
[fid:24:44]:[Seating id=4, pid=3, done=true, ls=3, ln=n3, rs=4, rn=n2]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]*

=>[ActivationCreated(46): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating
=>[fid:30:47]:[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
=>[fid:31:48]:[Path id=5, seat=5, guest=n1]
=>[fid:32:49]:[Chosen id=4, name=n1, hobbies=h1]
```

23.17. Backward-Chaining

A backward-chaining rule system is goal-driven. This means the system starts with a conclusion which the engine tries to satisfy. If it cannot do so it searches for sub-goals, that is, conclusions that will complete part of the current goal. It continues this process until either the initial conclusion is satisfied or there are no more unsatisfied sub-goals. **Prolog** is an example of a backward-chaining engine.

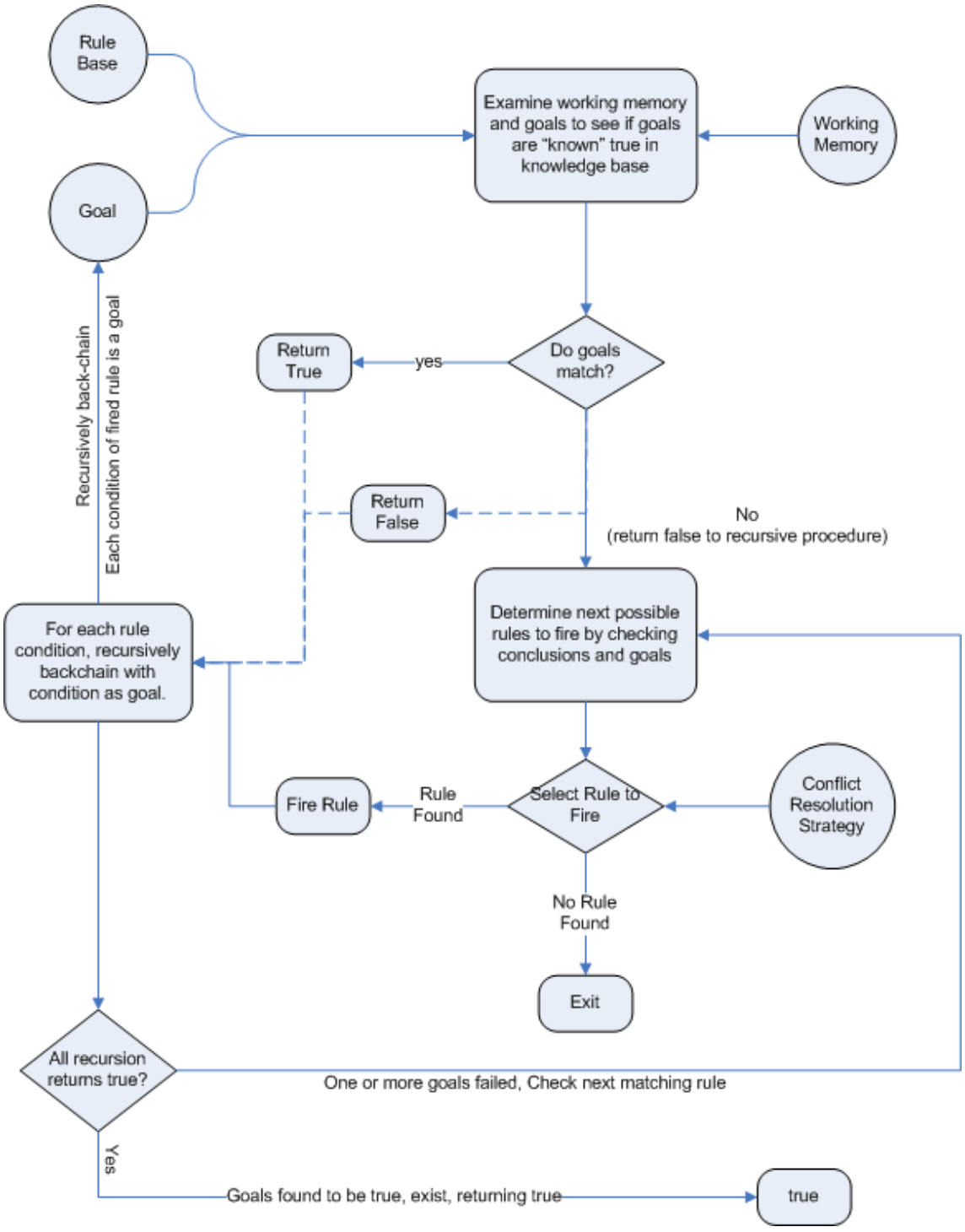


Figure 23.43. Backward Chaining Chart

23.17.1. Backward-Chaining Systems

Backward-Chaining is a feature recently added to the JBoss Rules Engine. This process is often referred to as derivation queries, and it is not as common compared to reactive systems since

JBoss Rules is primarily reactive forward chaining. That is, it responds to changes in your data. The backward-chaining added to the engine is for product-like derivations.

23.17.2. Cloning Transitive Closures

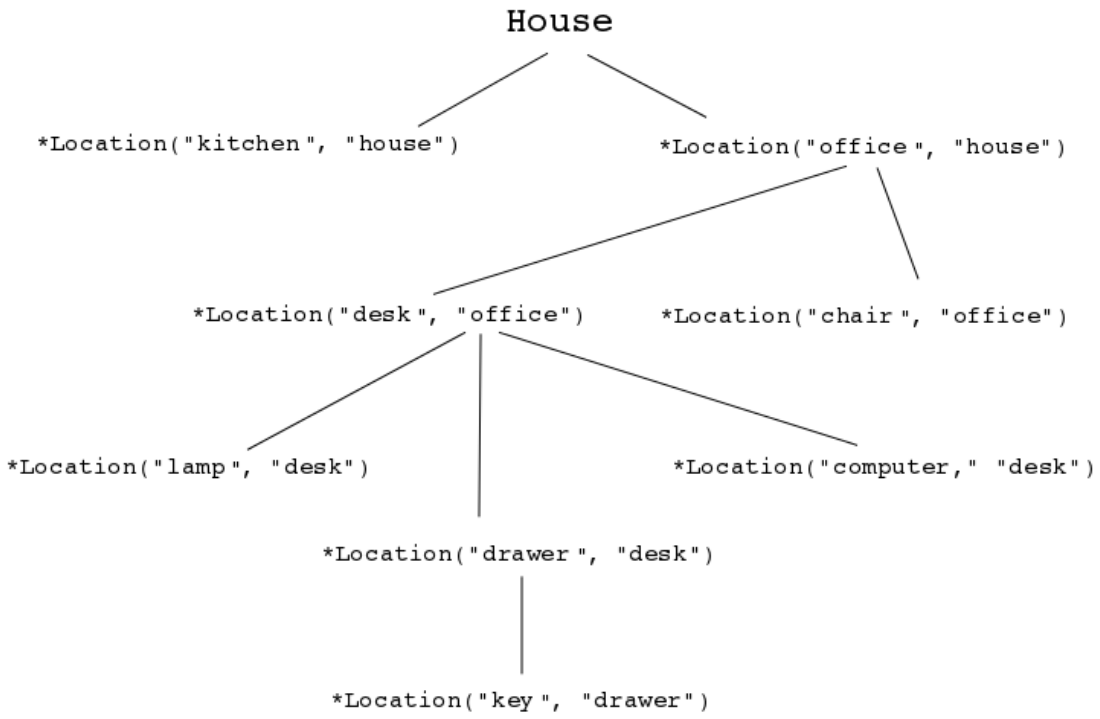


Figure 23.44. Reasoning Graph

The previous chart demonstrates a House example of transitive items. A similar reasoning chart can be created by implementing the following rules:

1. First, create some java rules to develop reasoning for transitive items. It inserts each of the locations.

2. Next, create the **Location** class; it has the item and where it is located.

3. Type the rules for the House example as depicted below:

```
ksession.insert( new Location("office", "house") );
ksession.insert( new Location("kitchen", "house") );
ksession.insert( new Location("knife", "kitchen") );
ksession.insert( new Location("cheese", "kitchen") );
ksession.insert( new Location("desk", "office") );
ksession.insert( new Location("chair", "office") );
ksession.insert( new Location("computer", "desk") );
ksession.insert( new Location("drawer", "desk") );
```

4. A transitive design is created in which the item is in its designated location such as a "desk" located in an "office."

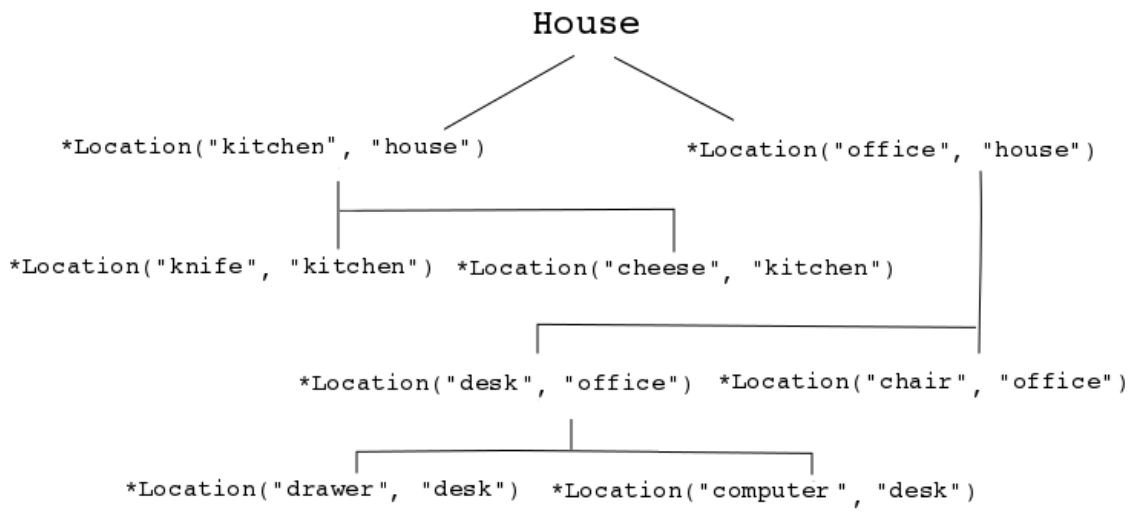


Figure 23.45. Transitive Reasoning Graph of a House

23.17.3. Defining a Query

1. Create a query to look at the data inserted into the rules engine:

```
query isContainedIn( String x, String y )  Location( x, y; )  or  ( Location( z, y; ) and
 isContainedIn( x, z; ) )end
 )  Location( x, y;
 )
 or  ( Location( z, y; ) and isContainedIn( x, z; )
```

Notice how the query is recursive and is calling "isContainedIn."

2. Create a rule to print out every string inserted into the system to see how things are implemented. The rule should resemble the following format:

```
rule "go" salience 10
when
  $s : String( )
then
  System.out.println( $s );
end
```

3. Using Step 2 as a model, create a rule that calls upon the Step 1 query "isContainedIn."

```
rule "go1"
when
  String( this == "go1" )
  isContainedIn("office", "house"; )
then
  System.out.println( "office is in the house" );
end
```

The "go1" rule will fire when the first string is inserted into the engine. That is, it asks if the item "office" is in the location "house." Therefore, the Step 1 query is evoked by the previous rule when the "go1" String is inserted.

4. Create the "go1," insert it into the engine, and call the fireAllRules.

```
         ksession.insert( "go1" );
ksession.fireAllRules();
---
go1
office is in the house
```

The --- line indicates the separation of the output of the engine from the firing of the "go" rule and the "go1" rule.

23.17.4. Transitive Closure Example

1. Create a Transitive Closure by implementing the following rule:

```
rule "go2"
when
  String( this == "go2" )
  isContainedIn("drawer", "house"; )
then
  System.out.println( "Drawer in the House" );
end
```

2. Recall from the Cloning Transitive Closure's topic, there was no instance of "drawer" in "house." "drawer" was located in "desk."

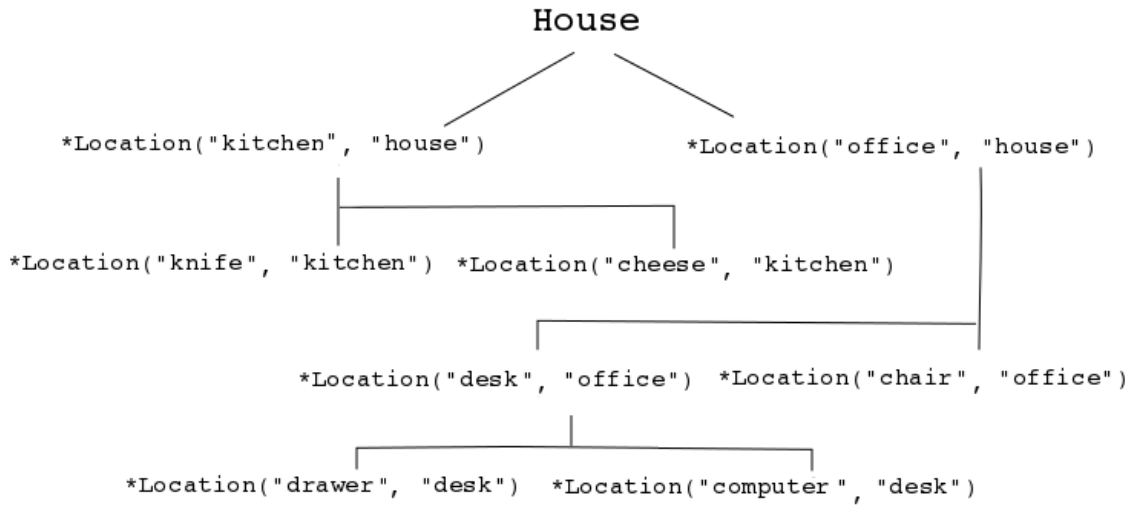


Figure 23.46. Transitive Reasoning Graph of a Drawer

3. Use the previous query for this recursive information.

```
query isContainedIn( String x, String y )  Location( x, y; )  or  ( Location( z, y; ) and
 isContainedIn( x, z; ) )end
 )  Location( x, y;
 )
 or  ( Location( z, y; ) and isContainedIn( x, z; )
```

4. Create the "go2," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go2" );
ksession.fireAllRules();
---
go2
Drawer in the House
```

When the rule is fired, it correctly tells you "go2" has been inserted and that the "drawer" is in the "house."

5. Check how the engine determined this outcome

- The query has to recurse down several levels to determine this.
- Instead of using **Location(x, y;)**, The query uses the value of **(z, y;)** since "drawer" is not in "house."

- The **z** is currently unbound which means it has no value and will return everything that is in the argument.
- **y** is currently bound to "house," so **z** will return "office" and "kitchen."
- Information is gathered from "office" and checks recursively if the "drawer" is in the "office." The following query line is being called for these parameters: **isContainedIn (x ,z;)**

There is no instance of "drawer" in "office;" therefore, it does not match. With z being unbound, it will return data that is within the "office," and it will gather that **z == desk**.

```
isContainedIn(x==drawer, z==desk)
```

isContainedIn recurses three times. On the final recurse, an instance triggers of "drawer" in the "desk."

```
 Location(x==drawer, y==desk)
```

This matches on the first location and recurses back up, so we know that "drawer" is in the "desk," the "desk" is in the "office," and the "office" is in the "house;" therefore, the "drawer" is in the "house" and returns true.

23.17.5. Reactive Transitive Queries

1. Create a Reactive Transitive Query by implementing the following rule:

```
rule "go3"
when
  String( this == "go3" )
  isContainedIn("key", "office"; )
then
  System.out.println( "Key in the Office" );
end
```

Reactive Transitive Queries can ask a question even if the answer can not be satisfied. Later, if it is satisfied, it will return an answer.

2. Use the same query for this reactive information.

```
query isContainedIn( String x, String y )  Location( x, y; )  or  ( Location( z, y; ) and
 isContainedIn( x, z; ) )end
 )  Location( x, y;
 )
 or  ( Location( z, y; ) and isContainedIn( x, z; )
```

3. Create the "go3," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go3" );
ksession.fireAllRules();
---
go3
```

The first rule that matches any String returns "go3" but nothing else is returned because there is no answer; however, while "go3" is inserted in the system, it will continuously wait until it is satisfied.

4. Insert a new location of "key" in the "drawer":

```
ksession.insert( new Location("key", "drawer") );ksession.fireAllRules();---Key in the Office
);
ksession.fireAllRules();
---Key in the
```

This new location satisfies the transitive closure because it is monitoring the entire graph. In addition, this process now has four recursive levels in which it goes through to match and fire the rule.

23.17.6. Queries with Unbound Arguments

1. Create a Query with Unbound Arguments by implementing the following rule:

```
rule "go4"
when
  String( this == "go4" )
  isContainedIn(thing, "office"; )
then
  System.out.println( "thing" + thing + "is in the Office" );
end
```

This rule is asking for everything in the "office," and it will tell everything in all the rows below. The unbound argument (out variable thing) in this example will return every possible value; accordingly, it is very similar to the z value used in the Reactive Transitive Query example.

2. Use the query for the unbound arguments.

```
query isContainedIn( String x, String y )  Location( x, y; )  or  ( Location( z, y; ) and
 isContainedIn( x, z; ) )end
 )  Location( x, y;
 )
 or  ( Location( z, y; ) and isContainedIn( x, z; )
```

3. Create the "go4," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go4" );
ksession.fireAllRules();
---
go4
thing Key is in the Office
thing Computer is in the Office
thing Drawer is in the Office
thing Desk is in the Office
thing Chair is in the Office
```

When "go4" is inserted, it returns all the previous information that is transitively below "Office."

23.17.7. Multiple Unbound Arguments

1. Create a query with Mulitple Unbound Arguments by implementing the following rule:

```
rule "go5"
when
  String( this == "go5" )
  isContainedIn(thing, location; )
then
  System.out.println( "thing" + thing + "is in" + location );
end
```

This rule is asking for everything in the "office," and it will tell everything in all the rows below. The unbound argument (out variable thing) in this example will return every possible value; accordingly, it is very similar to the z value used in the Reactive Transitive Query example.

Both **thing** and **location** are unbound out variables, and without bound arguments, everything is called upon.

2. Use the query for multiple unbound arguments.

```
query isContainedIn( String x, String y )  Location( x, y; )  or  ( Location( z, y; ) and
 isContainedIn( x, z; ) )end
 )  Location( x, y;
 )
 or  ( Location( z, y; ) and isContainedIn( x, z; )
```

3. Create the "go5," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go5" );
ksession.fireAllRules();
---
go5
thing Knife is in House
thing Cheese is in House
```

```
thing Key is in House
thing Computer is in House
thing Drawer is in House
thing Desk is in House
thing Chair is in House
thing Key is in Office
thing Computer is in Office
thing Drawer is in Office
thing Key is in Desk
thing Office is in House
thing Computer is in Desk
thing Knife is in Kitchen
thing Cheese is in Kitchen
thing Kitchen is in House
thing Key is in Drawer
thing Drawer is in Desk
thing Desk is in Office
thing Chair is in Office
```

When "go5" is called, it returns everything within everything.