



# Drools

---

David THIBAU – 2023

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- Introduction
  - BRMS
  - KIE's projects
  - Drools Rule's engine
  - Architecture alternatives
- Getting started
  - KIE APIs
  - Session stateless
  - Session stateful and inference
  - Agenda and conflicts
  - Listeners, Channels, Entry-points
- DRL syntax
  - Main structural elements
  - Rules and attributes
  - LHS / RHS
  - Queries, Agregation
- Other ways to express rules
  - Decision tables
  - Rule's template
  - DMN
- Related Projects
  - Complex Event Processing
  - Jbpm and Drools interaction
- Annexes
  - ReteOO and PHREAK



# Introduction

---

## **BRMS**

KIE's projects  
The rules Engine Drools  
IDE setup



# The problem

---

Today, the main challenge for business applications is agility

Applications must be able to adapt quickly and react to:

- Functional evolutions
- Changes in legislation
- Changes in organisation
- ...

In other words :Changes in business rules



# Where to implement business rules?

---

Stored as configuration properties in files or database ?

=> Not suited for such a rule

*If the customer lives in Paris, is over 55 years old  
and has been a customer for more than 2 years,  
give a 10% discount*

Implemented in source code

- Low maintainability
- Code spaghetti which may be unefficient



# BRMS

---

A **business rule management system (BRMS)**

identifies the notion of business rule as a resource that can be managed independently of the application code

- Rules can be edited, versionned monitored by a business expert
- Independently tested
- Independently documented, audited
- Independently deployed



# Rule's engine

---

A BRMS include a **rule's engine**

Which evaluates IF-THEN instructions based upon the business rules

When the conditions of the rules are satisfied (IF), it executes the associated actions (THEN) which generally modify the model

=> Programming becomes declarative

=> Business logic is no longer distributed in the code but centralized in the rules repository.



# Execution model

---

## Rule's engine :

- Parses and compile the set of rules. (Once at the startup of the application for instance)

## Client (code applicatif) :

- Get a reference to the rule's engine
- Insert facts
- Ask the engine to trigger rules
- Retrieve the objects updated by the rule's engine





# Required steps

---

Adopting a rule-based solution requires:

- Identification of business rules: *Business Expert*
- Implements the rule in a rule language:  
*Business / technical expert*
- Integrating the Rules Service into the Application  
(Libraries/ External service): *Technical*
- Provision of a rules management interface (BRMS  
Tool): *Business / Technical*
- Deployment of new rules procedure (Automation  
of tests / Deployment): *Technical*



# Considerations

---

- A rule engine is based on complex technologies
- It's hard to rely on a black box
- How the rules are triggered is not very intuitive
- Rules extraction is not always easy



# Benefits

---

- **Declarative programming** : Rule engines allow you to specify "What to do" and not "How to do it". They are able to solve difficult problems and in addition to providing explanations!
- **Separation of concern** : The data is in domain objects, the logic centralized in a rules file (different from the OO approach that encapsulates attributes and methods). The business logic is no longer dispersed
- **Centralization of knowledge** : Everything is centralized in the knowledge base, relatively readable and can be used as documentation
- **Understandable rules** : By defining language specific to the domain or the trade, the rules are expressed in quasi-natural language and become accessible to the business experts



# When to use a rules engine ?

---

The problem is too complex for the classical code (optimization problems for example, expert system)

The problem is not complex but no robust solution is needed.

The logic often changes, in this case the rules can be changed quickly without too much risk.

The business experts exist but are not technical. The rules then make it possible to express the business logic in their own terms.

# When not to use rules engine?



A rule engine is just a part of a complex application, you do not have to implement everything as rules

A good indicator is the degree of coupling between the rules.

- If triggering a rule invariably triggers a rule chain, then the implementation of this rule-based logic may not be appropriate, a decision tree may be sufficient



# Classical domains and use cases

---

## Domains

- Finance and Insurance
- Regulation, government rules
- Ecommerce
- Risk management

## Use cases

- Authorization of access based on several criteria (role, ownership of the entity, organization, location, etc.)
- Application customization (eg management of a personalized homepage of an e-commerce site)
- Diagnostic
- Complex validation
- Workflow / Orchestration
- Problem of routing, Optimization of planning, storage, ...



# Introduction

---

BRMS

**KIE's projects**

The rules Engine Drools

IDE setup



# KIE's projects

---

**KIE (Knowledge Is Everything)** wraps several projects that share the same API and the same building and deployment techniques (based on Maven and Git)

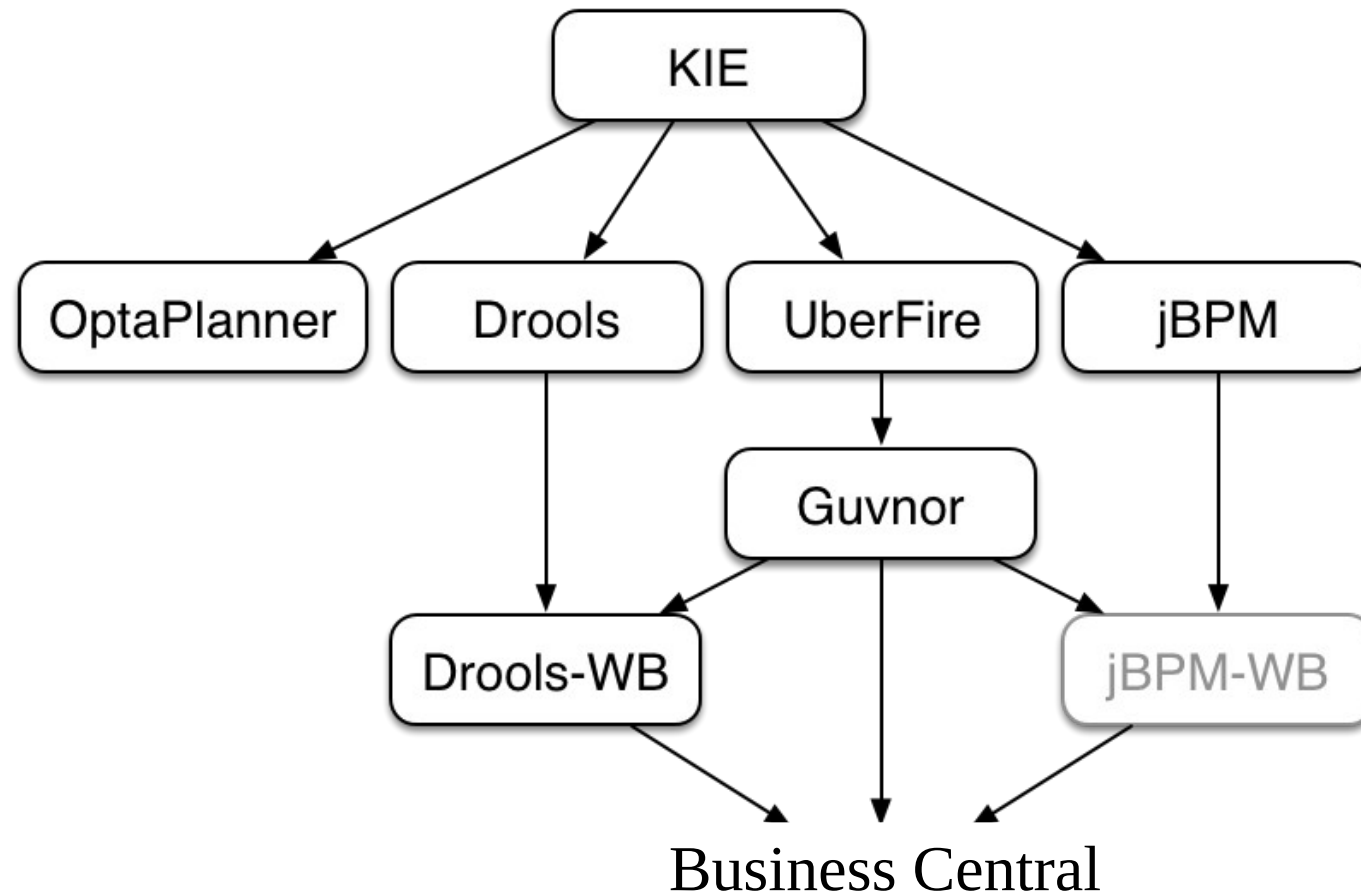
- **Drools** : Rules engine and Complex Event Processing
- **jBPM** : Workflow engine
- **OptaPlanner** optimize complex problems subject to constraints (planning, vehicle routing, etc.)
- **Business Central** : Web application for managing business rules and processes.
- **UberFire** : A framework to easily build web interfaces: workbenches





# KIE's projects

---





# Business Central Project Lifecycle

---

**Authoring** : Creation of knowledge using a specific language: DRL, BPMN2, Decision Table, ...

**Build** : Build a Deployable Artifact containing Knowledge (*kjar*)

**Test** : Test rules, processes

**Deployment** : Deployment in a repository (Typically **Maven**)

**Client Integration**: *kjar* is exposed as a **KieContainer**.  
Client application create **KieSession** to interact with the engine (*embeded or REST API* )

**Usage** : User interface or CLI for end users

**Operation** : Monitoring of session, Reporting



# Alternatives

---

## Authoring / Development

- Business Central and several *KieServer* to deploy knowledge base for testing or production
- Eclipse plugin and Java test classes. Pipeline CI/CD to deploy on *KieServer* or other

## Packaging

- Embeddeed Drools libraries and rules in a regular Java application
- Kjar deployed separately from client applications in a Maven repository

## Client integration

- Direct Java Call
- Remote Java Call (Drools Client Library)
- Regular RestFul



# Writing rules

---

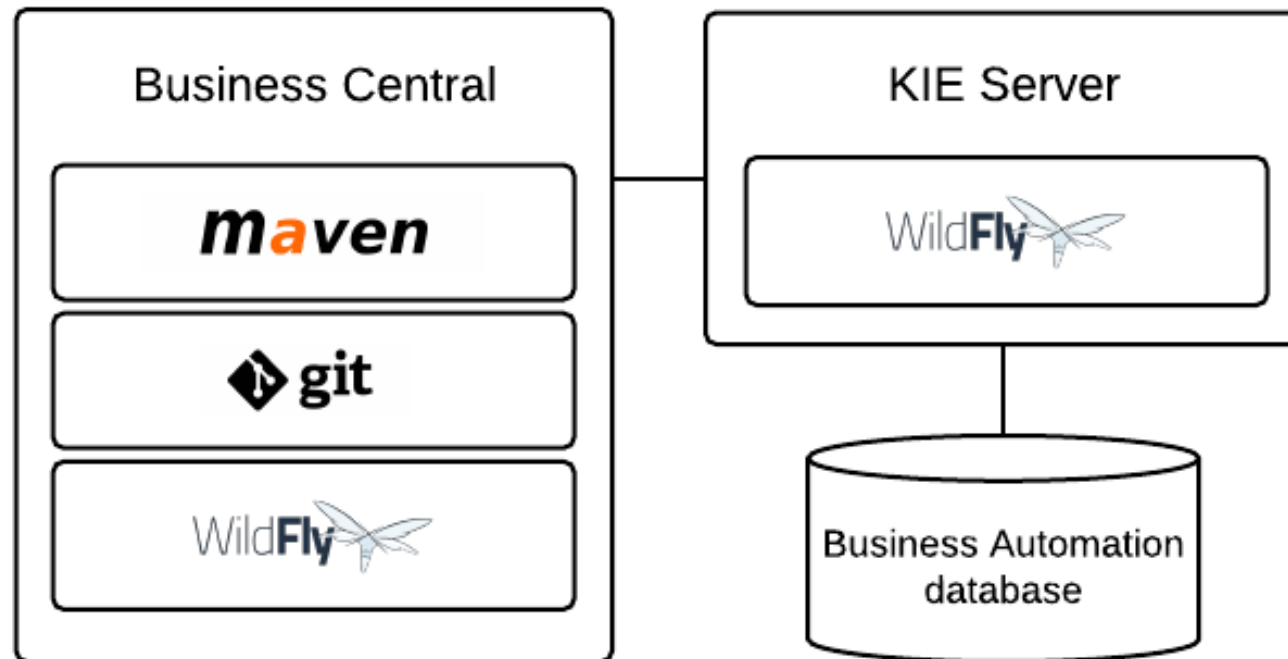
Drools supports different assets to specify rules:

- **Decision Model and Notation (DMN)** :  
Standard OMG : XML-based decision diagram
- **Decision tables** : Excel or guided decision tables of Business Central
- **Guided rules** : Simple guided rules of
- **DRL** : The most powerful
- **Predictive Model Markup Language (PMML)** : Predictive data analysis models in XML

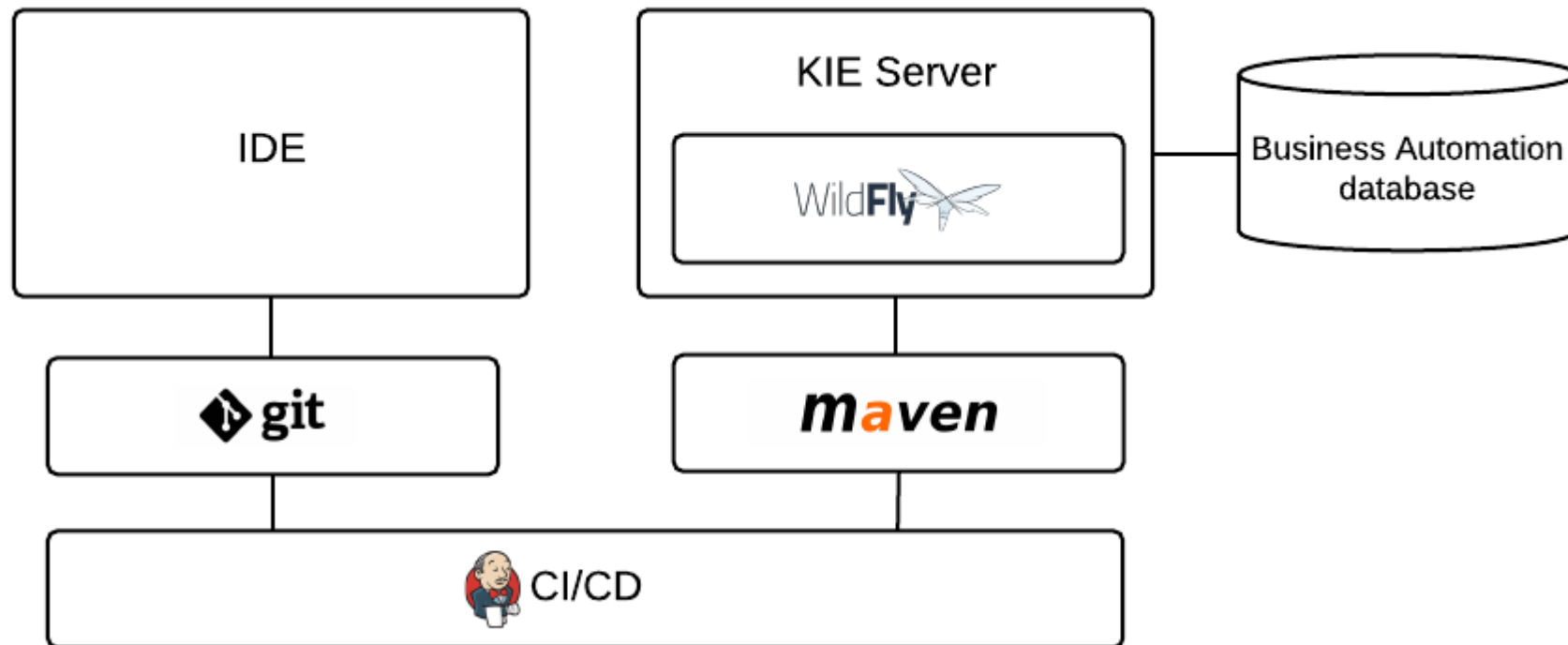


# Architecture Business Central

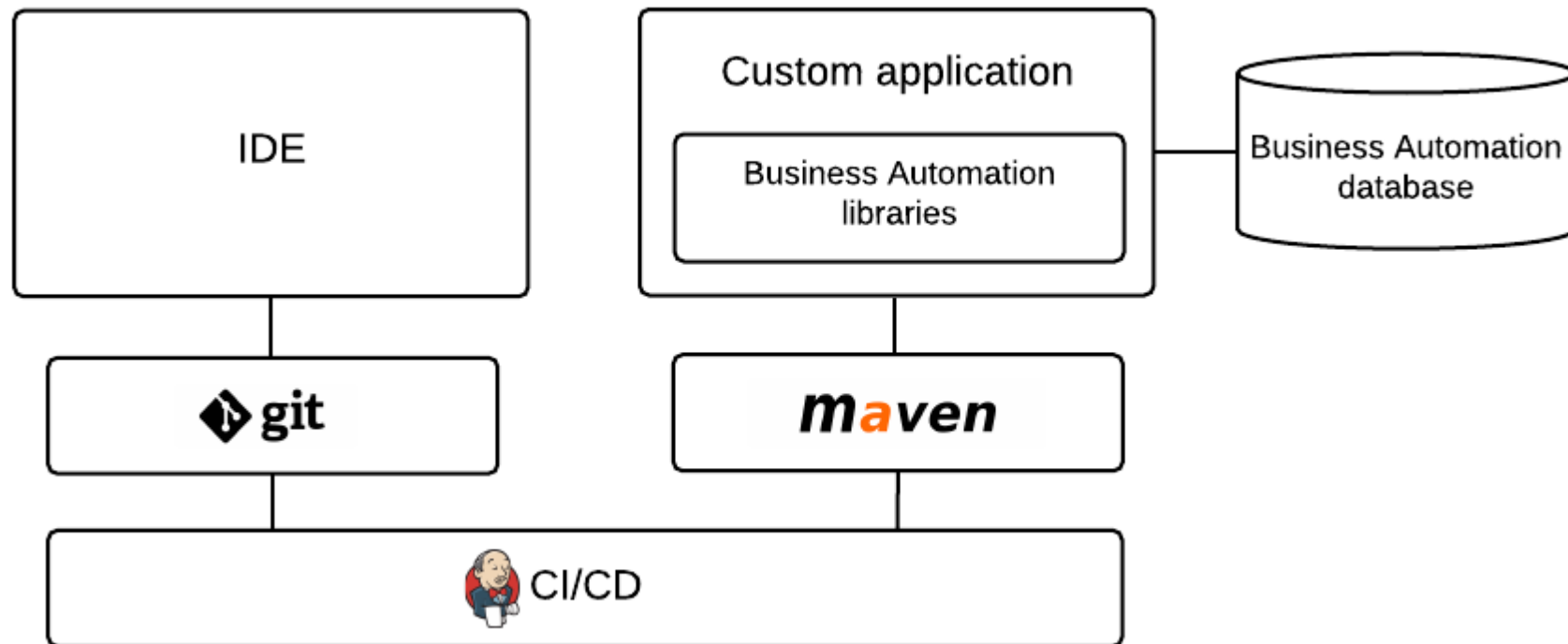
---



# IDE + KieServer



# IDE et Java embarqué





# Packaging KJAR

---

Whatever the architecture chosen, artifact produced is a JAR named **KJAR** which contains :

- The module descriptor ***META-INF/kmodule.xml***
- **Ressources** which contains business rules
- Domain **Model** classes : the facts.

The tool used for the packaging is generally Maven, a plugin allows to validate the rules files

The kjar may or may not contain the client application





# *kmodule.xml*

---

## *META-INF/kmodule.xml*

- Configures one or more knowledge bases by specifying the resources (rules files or processes)
- For each knowledge base, configure one or more types of sessions that can be created.

An empty descriptor applies a default configuration:

- all resource files found in the classpath are added to the same knowledge base.
- 2 types of sessions (stateless and stateful) are associated with the single knowledge base



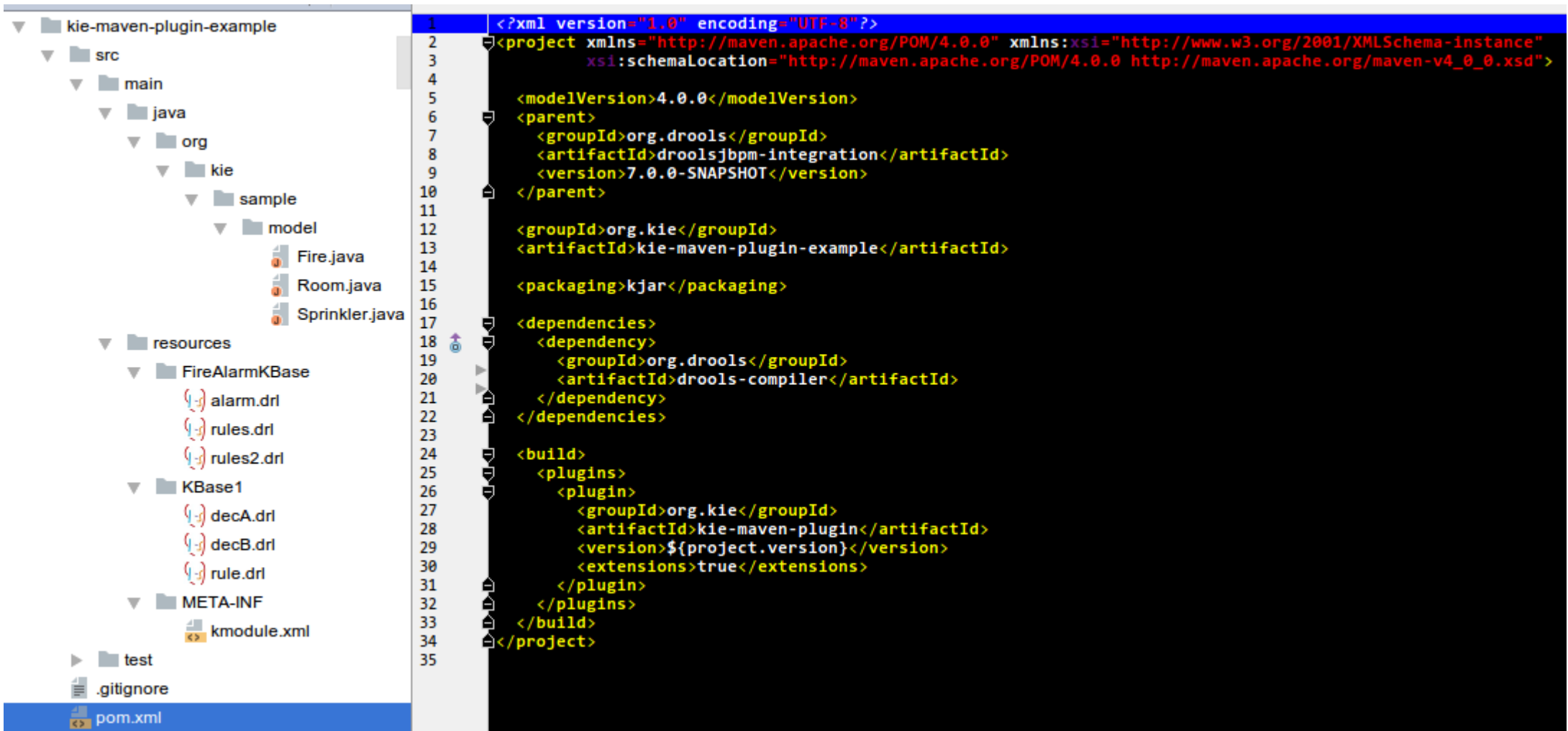
# Example *kmodule.xml*

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.drools.org/xsd/kmodule">

  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality"
  declarativeAgenda="enabled" packages="org.domain.pkg1">
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_2" type="stateless" default="false" beliefSystem="jtms"/>
  </kbase>

  <kbase name="KBase2" default="false" eventProcessingMode="stream" 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>
      <calendars>
        <calendar name="monday" type="org.domain.Monday"/>
      </calendars>
      <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>
```

# Example : Independent rules project



The screenshot displays a Maven project named 'kie-maven-plugin-example' in an IDE. The left pane shows the project's directory structure, and the right pane shows the contents of the 'pom.xml' file.

**Project Structure (Left Pane):**

- kie-maven-plugin-example
  - src
    - main
      - java
        - org
          - kie
            - sample
              - model
                - Fire.java
                - Room.java
                - Sprinkler.java
    - resources
      - FireAlarmKBase
        - alarm.drl
        - rules.drl
        - rules2.drl
      - KBase1
        - decA.drl
        - decB.drl
        - rule.drl
      - META-INF
        - kmodule.xml
    - test
    - .gitignore
    - pom.xml

**pom.xml Content (Right Pane):**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4
5     <modelVersion>4.0.0</modelVersion>
6     <parent>
7         <groupId>org.drools</groupId>
8         <artifactId>droolsjbpm-integration</artifactId>
9         <version>7.0.0-SNAPSHOT</version>
10    </parent>
11
12    <groupId>org.kie</groupId>
13    <artifactId>kie-maven-plugin-example</artifactId>
14
15    <packaging>kjar</packaging>
16
17    <dependencies>
18        <dependency>
19            <groupId>org.drools</groupId>
20            <artifactId>drools-compiler</artifactId>
21        </dependency>
22    </dependencies>
23
24    <build>
25        <plugins>
26            <plugin>
27                <groupId>org.kie</groupId>
28                <artifactId>kie-maven-plugin</artifactId>
29                <version>${project.version}</version>
30                <extensions>true</extensions>
31            </plugin>
32        </plugins>
33    </build>
34 </project>
35
```



# Introduction

---

BRMS

KIE's projects

**The rules engine Drools**

IDE setup



# Rule's engine

---

A rule's engine is composed of a **knowledge base**, an **inference engine** and a **working memory**

- The knowledge base groups the **compiled rules**
- The client application inserts facts (objects of the domain model) in the **working memory**
- The inference engine, able to handle large volume of rules and facts, has the role of comparing the facts to the conditions of the rules,  
if the **conditions of the rules are satisfied** the corresponding actions are performed.  
=> actions **modify the facts** of the working memory, which can trigger the activation of other rules.



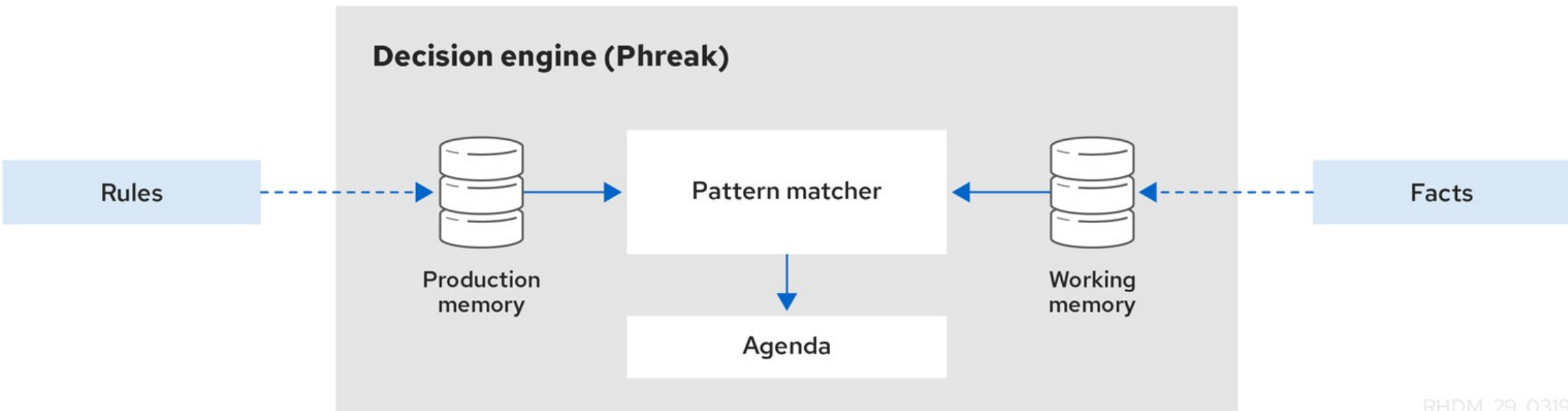
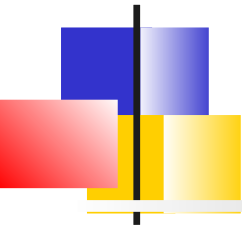
# Agenda

---

When matching the rules, it is possible that several rules are active simultaneously, it is said that they are in conflict.

The **Agenda** component is responsible for managing the execution order of the conflicting rules by using a conflict resolution strategy. (priority or other)

# Components of the engine



RHDM\_29\_0319

# Pattern matching, *ReteOO* and *PHREAK*



The treatment of comparing the facts to the rules is called the **Pattern Matching**. There are many pattern matching algorithms: *Linear*, *Rete*, *Treat*, *Leaps*.

Drools starts to implement and optimize the Rete algorithm in an object technology. (**ReteOO**)

- Eager algorithm
- Poor performance when inserting lot of facts in working memory

Since Drools6 : the new algorithm is **PHREAK** :

- Based on Rete Graph
- Lazy algorithm
- Much better performance when inserting facts





# Introduction

---

BRMS  
KIE's projects  
The rules engine Drools  
**IDE setup**



# Eclipse plugins

---

Drools provides Eclipse plugins which unfortunately will be discontinued, the current version is buggy!

It offers

- A Drools Perspective
- Project, *.drl* file, decision table and DSL creation wizards
- A *drl* file editor and validator
- Views for debugging



# Jars and dependencies

---

***knowledge-api.jar*** : Kie APIs. Required for compiling and runtime

***knowledge-internal-api.jar*** : Interfaces and factories for internal use

***drools-core.jar*** : The rule engine Required at runtime

***drools-compiler.jar*** : Rule's compiler. Generally, required at runtime rules may be pre-compiled

***drools-jsr94.jar*** : Implementation layer for JSR-94 on top of the drools-compiler.

***drools-decisiontables.jar*** : Compiler for decision tables.



# Drool's views

---

For debugging, the plugin offers several views to inspect the rules engine.

These views are available when the execution reaches a breakpoint

1. The **Working Memory View** allow to inspect the facts in memory
2. The **Agenda View** allow to see the activated rules in the agenda. For each rule, the associated variables are displayed.
3. The **Global Data View** allow to see all the global variables available in the session.



# The audit view

---

The audit view allow to display a log which can be generated with the following code :










```
KieRuntimeLogger logger =  
KieServices.Factory.get().getLoggers().newFileLogger(k  
    session, "logdir/mylogfile");  
ksession.insert(...);  
ksession.fireAllRules();  
// stop logging  
logger.close();
```

Or which has been configured via *kmodule.xml*




# Events of the logfile

---

1. Object inserted : 
2. Object updated : 
3. Object destroyed : 
4. Activation created 
5. Activation canceled : 
6. Activation executed : 
7. Sequence of starting or ending of a rule :
8. Activation/desactivation of a rules's group : 
9. Add/remove a rule's package : 
10. Add/remove a rule : 

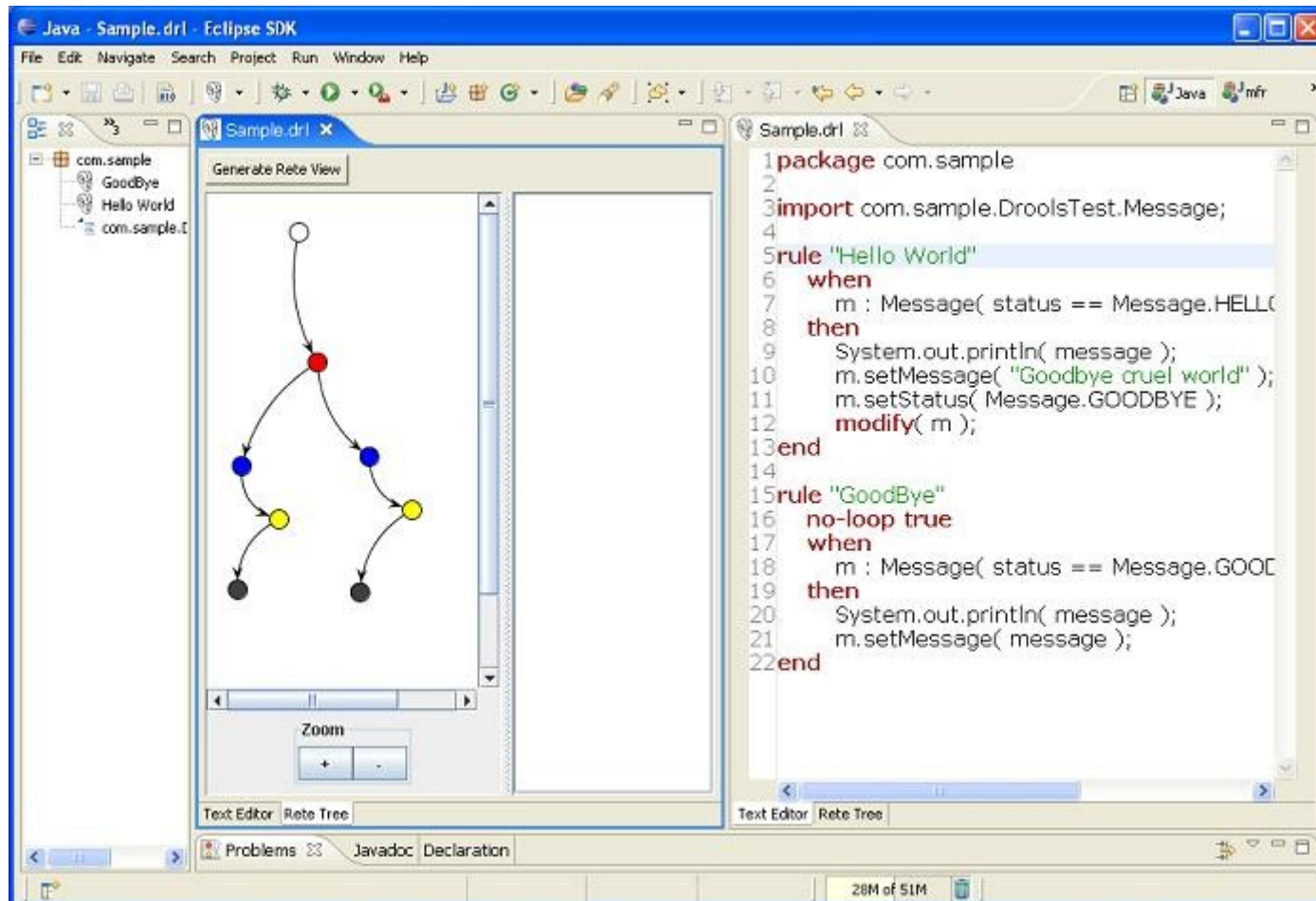
# Example



Activation executed: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:11]; guest=[fid:8:8]

- Object asserted (12): [Seating id=1, pid=0, pathDone=true, leftSeat=1, leftGuestName=n5, rightSeat=1, rightGuestName=n5]
- Object asserted (13): [Path id=1, seat=1, guest=n5]
- Object modified (11): [Count value=2]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:0:0]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:1:1]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:2:2]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:3:3]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:4:4]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:5:5]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:6:6]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:7:7]
- Activation created: Rule assignFirstSeat context=[fid:10:10]; count=[fid:11:15]; guest=[fid:8:8]
- Object modified (10): [Context state=ASSIGN\_SEATS]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:2:2]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:4:4]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:8:8]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:6:6]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:1:1]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:3:3]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:7:7]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:5:5]
- Activation cancelled: Rule assignFirstSeat context=[fid:10:16]; count=[fid:11:15]; guest=[fid:0:0]
- Activation created: Rule findSeating leftGuestName=[fid:0:0]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16];
- Activation created: Rule findSeating leftGuestName=[fid:4:4]; rightGuestSex=[fid:8:8]; seatingId=[fid:12:12]; seatingRightGuestName=[fid:12:12]; context=[fid:10:16];

# Rete view







# Rete view legend

---

- Green: Input point
- Red : *ObjectTypeNode*
- Blue : AlphaNode (Constraint)
- Yellow: Adapter of the left input
- Green : BetaNode
- Black : Rule node

Selection update the properties view



# Archetype Maven

---

An alternative to using the project creation wizard, it is possible to use a Maven archetype:

**org.kie:kie-drools-archetype**



# Getting started

---

## **KIE API**

Stateless Session

Stateful Session and inference

Agenda and conflicts

Listeners, Channels, Entry-points



# Main classes

---

**KieServices** : Singleton giving access to other Kie services (*Container, Loggers, persistence, Serializer, ...*)

**KieModule** : The configuration backed by a *kmodule.xml*

**KieContainer** : The container responsible for locating the knowledge resources defined in a module and to instantiate *KieBase* and *KieSession*

**KieBase** : A compiled knowledge base

**KieSession** : API with the working memory. 45



# *KieModule*

---

*KieModule* configure different *KieBase*  
et *KieSessions*

Configuration can be done

- Via a XML file : *META-INF/kmodule.xml*
- Or programmatically via  
*KieModuleModel*



# Exemple KieServices

---

```
// Retrieve the singleton KieServices
```

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

```
// Instanciate a container which loads ressources
```

```
// from classpath
```

```
KieContainer container =  
    KieServices.getKieClasspathContainer()
```



# *KieContainer*

---

Different *KieContainer* can be created depending the way to load resources

- From the classpath
- From a maven repository
- From a REST API

From *KieContainer*, we can instantiate :

- *KieBases* and *KieSessions* defined in the module



# Example

---

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

```
// Retrieve kieBase, kieSession from their names
```

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

```
StatelessKieSession kieSession2 =  
    kContainer.newStatelessKieSession("KSession2_2");
```





# *ReleaseId*

---

A Kie project is a Maven project

The *groupId*, *artifactId*, and release declared in the *pom.xml* file are used to generate a ***ReleaseId***.

A *KieContainer* can also be constructed with a *ReleaseId*.

It will load resources from the Maven repository<sup>1</sup>

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

1. By default, the local repository of the user running the Java process



# Types of sessions

---

Two types of Drools session are possible :

- ***stateless*** : *StatelessKieSession* which does not use inference
- ***stateful*** : *KieSession* (default)



# Getting started

---

KIE API

**Stateless Session**

Stateful Session and inference

Agenda and conflicts

Listeners, Channels, Entry-points



# *Stateless Knowledge Session*

---

Stateless sessions are the simplest case because they do not use the inference engine.

You can think about a function that we would pass arguments and that would cause a result.

Use cases of stateless sessions :

- Validation: Is a person eligible for a loan?
- The calculation: Calculate a reduction
- Routing or filtering: Filter emails, forward messages to destinations
- ...

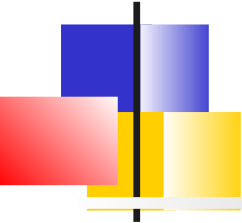


# Sample : The domain model (facts)

---

```
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  
}
```



# Sample - Rules

---

```
package com.company.license
```

```
rule "Is of valid age"
```

```
when
```

```
    Applicant( age < 18 )
```

```
    $a : Application()
```

```
then
```

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

```
end
```

```
rule "Application was made this year"
```

```
when
```

```
    $a : Application( dateApplied > "01-jan-2014" )
```

```
then
```

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

```
end
```



# Pattern matching (Rete)

---

When an instance of an *Applicant* is inserted in the engine, this fact is evaluated against the constraints of the rules.

- In this case, the 2 constraints of the first rule (constraint on the type and the age field).
- A constraint on an object type plus one or more constraints on its fields is called a pattern.

When an inserted instance matches the pattern, the consequence of the rule is executed.

The notation ***\$a*** represents a variable to reference the object matched in the consequence. Its properties can be updated in the consequence part.

The ('\$') character is optional but makes the expression of the rule more readable.



# Configuration

---

In case of a classpath container, the file *kmodule.xml* must be located in;  
***ressources/META-INF/kmodule.xml***

For the default configuration, this file is empty :

```
<?xml version="1.0" encoding="UTF-8"?>
```

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





# Container's creation

---

```
// Get the Singleton
```

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

```
// Load the module from the classpath
```

```
// Compile drl files found
```

```
// Set the compiled result
```

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



# Execution

---

**// Stateless session creation**

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

```
Applicant applicant = new Applicant( "Mr John Smith", 16 );
```

```
Application application = new Application();
```

```
assertTrue( application.isValid() );
```

**// Execution of the rules for this 2 facts**

```
ksession.execute(  
    Arrays.asList( new Object[] { application, applicant } ) );
```

```
assertFalse( application.isValid() );
```



# *BatchExecutor and CommandFactory*

---

The methods *execute(Object object)* and *execute(Iterable objects)* are shortcuts for : ***execute(Command command)*** of the *BatchExecutor* interface

For instance, the second method is equivalent to :

```
ksession.execute(  
    CommandFactory.newInsertIterable( new Object[] {  
        application, applicant } ) ) ;
```

# *BatchExecutor and CommandFactory*

*BatchExecutor* and *CommandFactory* are particularly useful for working with several commands and results avec plusieurs commandes et plusieurs identifiants de résultats :

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



# Getting started

---

KIE API

Stateless Session

**Stateful Session and inference**

Agenda and conflicts

Listeners, Channels, Entry-points



# Stateful Session

---

**Stateful** session have a longer life cycle and allow iterative update of the facts .

Use cases are :

- Monitoring : Stock market monitoring and semi-automatic purchasing
- Diagnosis: Discovery of fault, medical diagnosis
- Logistics: Delivery tracking, provisionning
- Compliance: Validation of legislation



# Stateful versus stateless

---

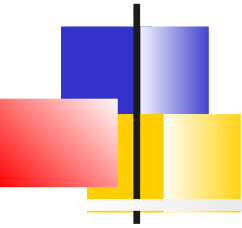
Unlike a Stateless Session, the ***dispose ()*** method must be called to avoid memory leaks (*KieBase* contains references to Stateful Sessions)

A Stateful Knowledge Session is simply called ***KieSession***.

*KieSession* supports also the interface *BatchExecutor*

- But, the method ***fireAllRules*** is not triggered automatically.

# Sample – Domain model



```
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 { }
```





# Sample – Rule

---

```
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
```



# Inference and *modify*

---

Unlike the *StatelessSession* example, which used the standard Java syntax to modify the attribute of a fact, using the ***modify*** statement can warn the engine of changes of the facts and thus allow it to make other pattern matching.

This is called **inference**



# *not* operator

---

The not operator is used to match when no instance of the object exists in the working memory:

```
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
```



# *exists* operator

---

The operator ***exist*** test the existence of a fact :

```
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
```



# *deleyte/retract* instruction

---

The instruction ***delete*** allows to remove a fact from working memory

rule "Cancel the alarm when all the fires have gone"

when

not Fire()

\$alarm : Alarm()

then

***retract***( \$alarm );

System.out.println( "Cancel the alarm" );

end



# Inference and maintainability

---

The insertion of a new fact from a previous knowledge may lead to more maintainability

```
rule "Infer Adult"
```

```
when
```

```
$p : Person( age >= 18 )
```

```
then
```

```
insert( new IsAdult( $p ) )
```

```
end
```

The other rules can be based on being an adult rather than the value 18. Adaptation to other specification will be facilitated



# *logicalInsert*

---

***insertLogical*** retract the fact as soon as the when clause becomes false again :

```
rule "Infer Child" when
```

```
$p : Person( age < 16 )
```

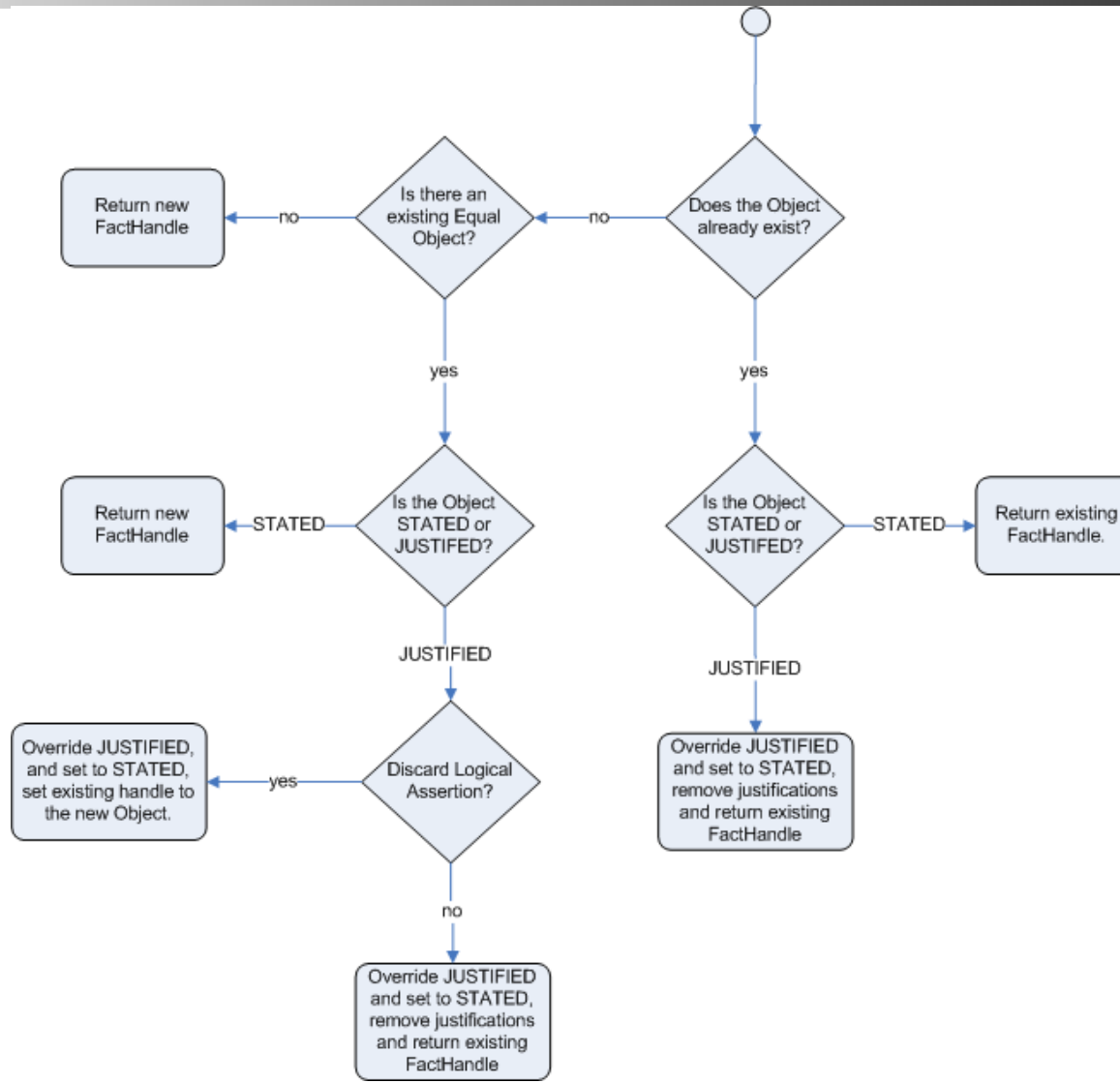
```
then
```

```
insertLogical( new IsChild( $p ) )
```

```
end
```

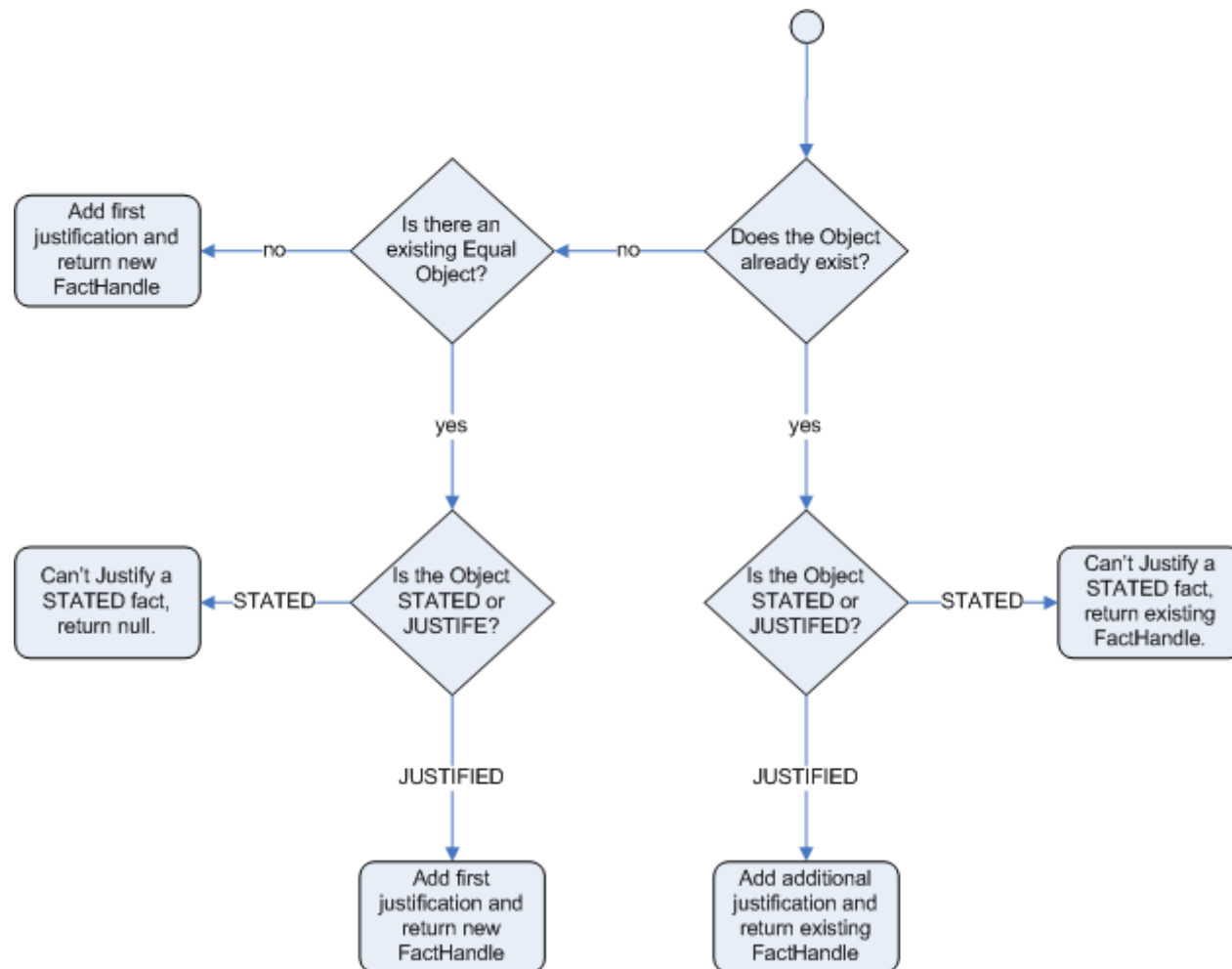
The fact *IsChild* (*\$ p*) is automatically retracted as soon as the person reaches 16

# Insertion simple (STATED)





# Insertion logique (JUSTIFIED)





# Recommendations

---

## Good practices

- Separating Knowledge Responsibilities
- Encapsulate knowledge
- Provide semantic abstractions for these encapsulations



# Equality of facts

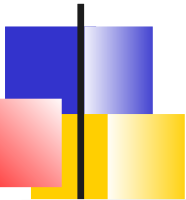
---

Determine when 2 facts are the same.

The rule engine handle 2 behaviour :

- ***identity***: (Default) Equality based on identity  
(== in Java)
- ***equality*** : Equality based on method *equals()*

```
<kbase name="KBase2" default="false"  
equalsBehavior="equality"  
packages="org.domain.pkg2, org.domain.pkg3"  
includes="KBase1">
```



# Creation of a stateful session

---

```
// Compile the drl files found in the classpath
// Set the result in the container
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
// Creation of the stateful session
KieSession ksession = kContainer.newKieSession();
```



# Execution

---

```
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
```



# *FactHandle*

---

A **FactHandle** allow to obtain a reference to an inserted fact in the working memory.

```
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

This reference allow to remove the fact later :

```
ksession.retract( kitchenFireHandle );
ksession.retract( officeFireHandle );
ksession.fireAllRules() ;
```

- > Turn on the sprinkler for room office
- > Turn on the sprinkler for room kitchen
- > Cancel the alarm
- > Everything is ok



# Getting started

---

KIE API

Stateless Session

Stateful Session and inference

**Agenda and conflicts**

Listeners, Channels, Entry-points



# Methods versus Rules

---

## Methods :

- They are explicitly called
- Specific instances are passed as arguments
- A call causes a single run

## Rules :

- They are never explicitly called
- Specific instances can not be passed as an argument
- A rule can fire once, several times, or no time.





# Activations, Agenda and conflicts (Drools6+)

---

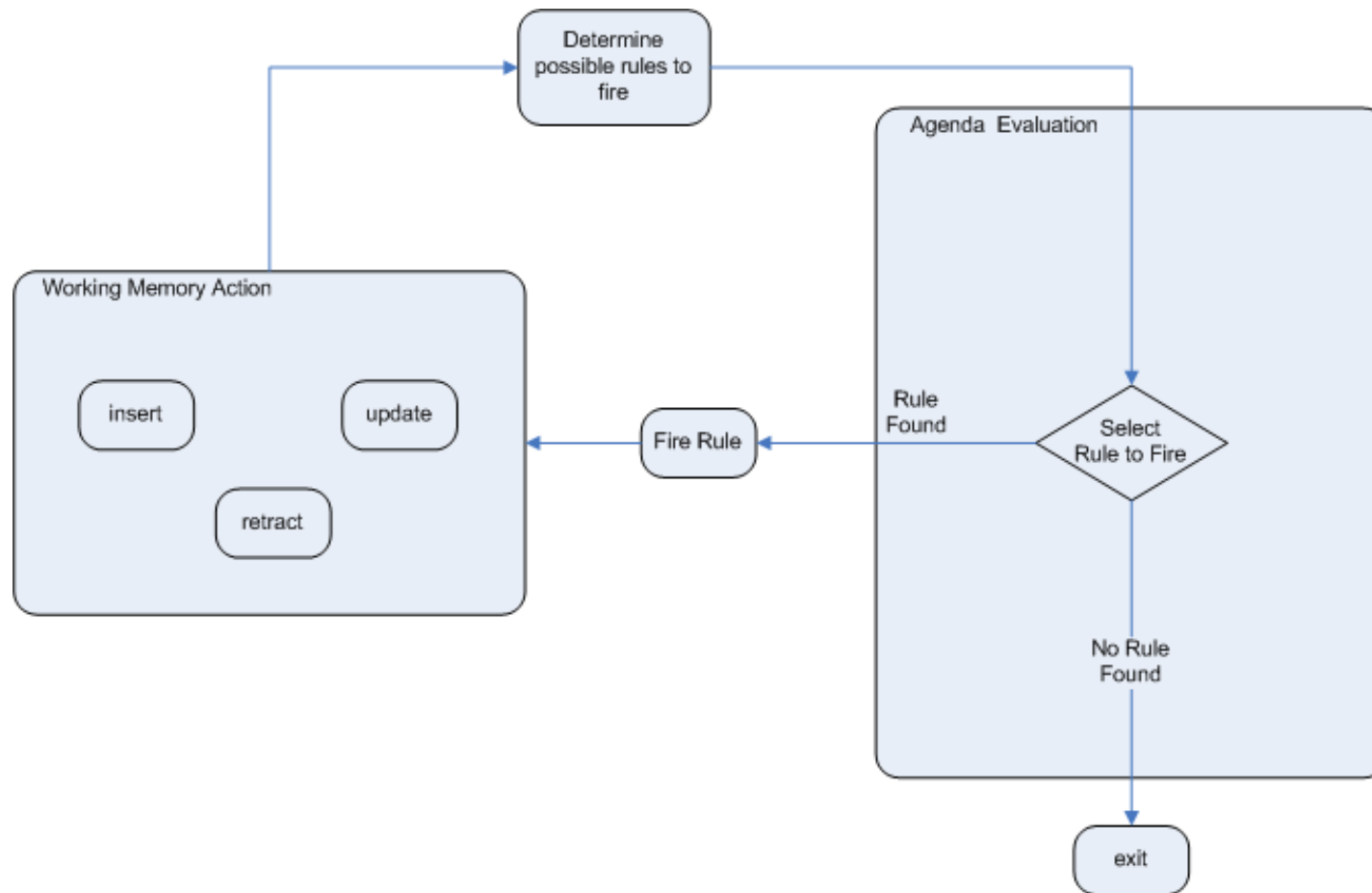
When calling *fireAllRules()*, the rules are evaluated independently of each other

The agenda chooses the first rule based on the active *activation group* and the *salience* attributes of the rule. If 2 rules have the same precedence, the declaration's order is taken in account

If the conditions of a rule are met, it is triggered immediately.

- If the consequence of the rule updates the working memory, the inference occurs

# Life cycle of the Agenda





# Agenda groups

---

**Agenda groups** allow to partition rules in groups that are themselves placed in an execution stack

The agenda executes the rules of the group placed on the top of the stack

When all the rules have been executed, the agenda pops another group from the stack.



# Execution model

---

Drools supports 2 rule execution modes :

- **Passive mode (default)** : Rules are triggered by ***fireAllRules()***
- **Active mode** : When ***fireUntilHalt()*** is called. Drools continuously evaluates rules until the ***halt()*** method is called. Application which reacts to facts insertion events, or when timers are configured



# Agenda filter

---

When executing the rules, Drools allows you to set an ***AgendaFilter*** to filter the rules that the Agenda can activate.

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```



# Getting started

---

KIE API

Stateless Session

Stateful Session and inference

Agenda and conflicts

**Listeners, Channels, Entry-points**



# Event Model

---

Kie provides an event model that allows listeners to be aware of motor events such as triggering a rule, inserting a fact, and so on.

This allows separation of trace or audit activities

The `KieRuntimeEventManager` interface is implemented by `KieRuntime` which provides 2 interfaces:

- ***RuleRuntimeEventManager***
- And ***ProcessEventManager***.



# Debug listeners

---

Drools provides 2 listeners for debugging. They display debug messages on the console :

- *DebugRuleRuntimeEventListener* and *DebugAgendaEventListener*

***KieRuntimeLogger*** also uses events to generate a trace file visible by the Audit view of Eclipse

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





# Example

---

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

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



# Configuration via *kmodule.xml*

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality">
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_1" type="stateless" default="false" beliefSystem="jtms"/>
  </kbase>
  <kbase name=""KBase2" default="false" eventProcessingMode="stream" equalsBehavior="equality"
    declarativeAgenda="enabled"
    packages="org.domain.pkg2,org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateful" default="false" clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <listeners>
        <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener"/>
        <agendaEventListener type="org.domain.FirstAgendaListener"/>
        <agendaEventListener type="org.domain.SecondAgendaListener"/>
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```



# Channels

---

A **channel** is a standardized way to transmit data from within a session to the external world.

- Alternative to globals. (see further)

Technically, *Channel* is a Java interface with a single method :

***void send(Object object)***

Channels can only be used in the RHS of our rules as a way to send data to outside



# Registration

---

Channels must be registered :

```
ksession.registerChannel("audit-channel",  
    auditChannel);
```

Then can be used in rules :

```
rule "Send Suspicious Operation to Audit  
    Channel"  
when  
    $so: SuspiciousOperation()  
then  
    channels["audit-channel"].send($so);  
end
```



# Entry points

---

**Entry points** are a way to partition working memory. The rules can then apply to certain entry points

**// Reasoning from an entry point**

```
rule "Routing transactions from small resellers"
```

```
when
```

```
  t: TransactionEvent()
```

```
    from entry-point "small resellers"
```

On insertion, the entry point can be specified:

```
ksession.getEntryPoint("myEntryPoint").insert(new Object());
```



# Rule syntax : DRL

---

## **Main elements**

Rule's attributes

LHS

RHS

Query, agregation



# *.drl* files

---

A rules file is a simple text file with the extension ***.drl***

It has the following structure :

- **package** : Namespace
- **imports** : Java types used
- **declare** : Internal declaration of new types
- **globals** : Globals variables which can accessed from outside the session
- **functions** : Reuse of logic
- **queries** : Fact queries
- **rules** : Rules

It is also possible to distribute the rules over several files which then usually have the *.rule* extension



# Structure of a rule

---

```
rule "name"  
    attributs  
    when  
        LHS  
    then  
        RHS  
end
```

Punctuation ", line breaks are optional

×Attributes are optional

×LHS is the conditional part of the rule

×RHS is a block that allows to specify in different dialects a code to execute.





# Keywords

---

## Keyword :

- Hard (*true, false, accumulate, collect, from, null, over, then, when*). Can not be used as object ID
- Soft : only recognized in their context (*package, import, attributes, rule, ...*)

The escape character is ` :  
*Holiday( `when` == "july" )*

## Comments :

- line : *#* ou *//*
- multi-lines : */\* .... \*/*



# Error message

[ERR 101] Line 6:35 no viable alternative at input ')' in rule "test rule" in pattern WorkerPerformanceContext

1st Block    2nd Block    3rd Block    4th Block    5th Block

1st block : **Error code**

2nd block : Information on the **column and the row**.

3rd block : **Description** of the error.

4th block optional : First **context** of the error.

Generally, the rule, the function, the query where the error occurred.

5th bloc optional : Identify the **pattern** where the error occurred.

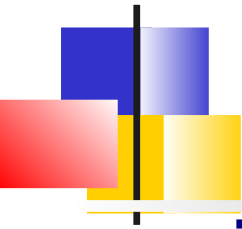


# Package

A **package** groups together a set of rules, imports, and globals that are related.

A *package* represents a namespace and must be unique in the knowledge base. It follows the naming conventions of Java packages

- If the rules of the same package are distributed over several files. Only one file contains the package configuration.
- Items declared in a package can be in any order, except for the package statement.
- The ';' are optional.



# Import

---

**import** instructions are similar to java *imports*

The full name of the Java types must be specified.

Drools automatically imports the Java classes from the package of the same name and the package *java.lang*.



# Global

---

**global** defines global variables

- Global variables can be used in the consequences of the rules.
- They are not inserted into working memory and therefore should not be used as conditions in the rules except as a constant
- The engine is not warned when a value change of a global variable



# Example

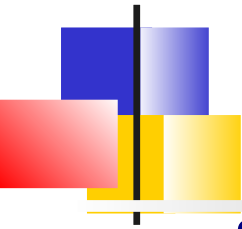
---

```
global java.util.List myGlobalList;
```

```
rule "Using a global"  
when  
  eval( true )  
then  
  myGlobalList.add( "Hello World" );  
end
```

-----

```
List list = new ArrayList();  
WorkingMemory wm = rulebase.newStatefulSession();  
wm.setGlobal( "myGlobalList", list );
```



# Functions

---

**functions** allow to insert code in rule's files.

- They are just like Helper classes.  
With functions, the logic is centralized in one place.
- They are used to invoke actions in the consequence part of the rules.



# Example

---

```
function String hello(String name) {  
    return "Hello "+name+"!";  
}
```

```
rule "using a static function"  
when  
    eval( true )  
then  
    System.out.println( hello( "Bob" ) );  
end
```





# Declarations

---

2 kinds of declarations :

- Declaration of **new types** : Drools works directly with POJOs as facts. It is therefore possible to define the business model directly in the rules or to create model objects that are useful only in reasoning.  
Drools, at compile-time, generates the Java bytecode that implements the new type
- Declaration of **meta-data** or **annotations**: The facts or their attributes can be annotated. Annotations can be used in reasoning or fact finding.



# Declaration of new types

---

```
declare Address
```

```
    number : int
```

```
    streetName : String
```

```
    city : String
```

```
end
```

```
declare Person
```

```
    name : String
```

```
    dateOfBirth : java.util.Date
```

```
    address : Address
```

```
end
```



# Access to the declared types

---

We can access to the internal declared types via the interface  
***org.drools.definition.type.FactType***

```
// Get the data type
FactType personType = kbase.getFactType( "org.drools.examples",
                                           "Person" );

// Instanciate it
Object bob = personType.newInstance();

// Set attributes
personType.set( bob, "name", "Bob" );
personType.set( bob, "age", 42 );
```



# Meta-data declaration

---

The character **@** is used

The metadata can concern a new or existing type or one of its attributes.

```
declare Person
```

```
  @author( Bob )
```

```
  @dateOfCreation( 01-Feb-2009 )
```

```
  name : String @key @maxLength( 30 )
```

```
  dateOfBirth : Date
```

```
  address : Address
```

```
end
```

On a existing type

```
declare Person
```

```
  @author( Bob )
```

```
  @dateOfCreation( 01-Feb-2009 )
```

```
end
```



# Use of meta-data

---

Drools allows the declaration of arbitrary meta-data.

Metadata can be used by queries.

Some meta-data are predefined and have a meaning for the engine. They are especially useful for *Drools-CEP*:

*@role, @timestamp, @duration, ...*



# Example

---

```
StatefulKnowledgeSession ksession= createKSession();  
ksession.fireAllRules(new AgendaFilter() {  
    public boolean accept(Activation activation) {  
        Map<String, Object> metaData =  
            activation.getRule().getMetaData();  
        if (metaData.containsKey("LegalRequirement")) {  
            System.out.println(metaData.get("LegalRequirement"));  
        }  
        return true;  
    }  
});
```



# Rule syntax : DRL

---

Main elements

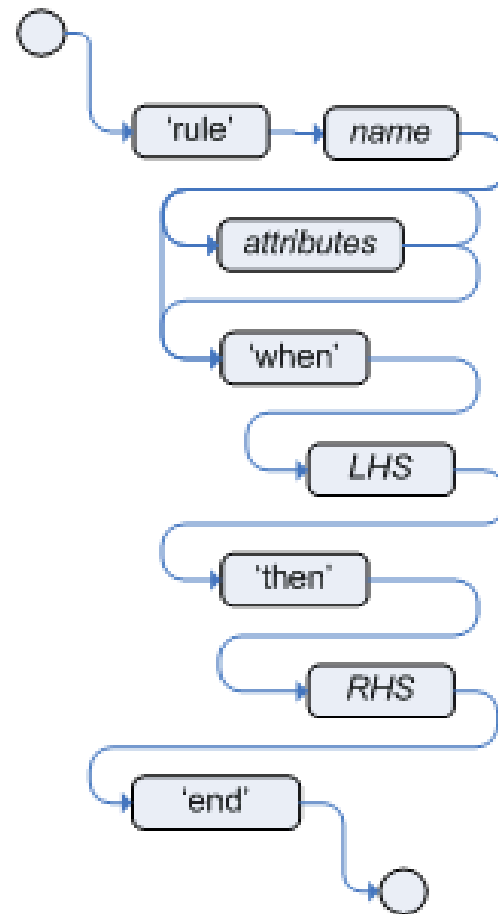
**Rule's attributes**

LHS

RHS

Query, agregation

# Rule







# Rule, syntax

---

- ✓ A rule must have a unique name inside the package.
- ✓ The name can contain spaces if it is delimited by ".
- ✓ The left side of the rule (Left Hand Side) or condition follows the keyword **when**
- ✓ The Right Hand Side or Consequence follows the keyword **then**
- ✓ The rule ends with the keyword **end.**
- ✓ Rules can not be nested.



# Attributes

---

***no-loop (boolean, false):*** When the consequence of the rule changes a fact, it can cause the rule to be activated again. Recursion can be avoided with the no-loop attribute set to *true*.

***salience (Integer, 0):*** Each rule has an integer salience attribute that determines the priority of the rule in the agenda.

***dialect:*** (String, "java" or "mvel") The dialect is usually specified at the package level. This attribute overrides the package-level definition.



# Attributes (2)

---

***agenda-group (String, MAIN)***: This attribute allows you to partition the Agenda and control the execution. Only the rules of the agenda group that has the focus are allowed to fire.

***activation-group (String)***: Rules belonging to the same activation group are exclusive. The first rule that fires cancels the others.

***ruleflow-group (String)***: Group several rules. The rules in this group will only be enabled when the process is in a particular node of an associated *jBPM* process.



# Attributes (3)

---

***auto-focus (boolean, false)***: When a rule is enabled with the auto-focus attribute set, the group indicated by one of its attributes (*agenda-group* or *activation-group*) gains the focus.

***lock-on-active (boolean, false)***: When a group (*ruleflow* or *agenda*) becomes active, all the rules in this group that have the *lock-on-active* attribute set will no longer be activated in the future whatever the origin of the update. They can be reactivated when their group is reactivated.

***effective-date (String defining a date)***: A rule can only be activated if the current date is greater than the effective date.

***date-expires (String defining a date)***: A rule can only be activated if the current date is greater than the expiration date.



# Timer

Drools supports *timers* based on intervals or expressed by cron expressions.

```
timer ( int: <initial delay> <repeat interval>? )
```

```
timer ( cron: <cron expression> )
```

## 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
```



# *fireUntilHalt()*

---

In order for rules using timers to be triggered, the engine must be active.

In this case, do not call *fireAllRules()* but ***fireUntilHalt()*** which evaluates the rules until it receives a halt signal

In this case, stop the engine can be done:

- Via a rule: ***drools.halt ()***
- Via Java: ***ksession.halt ()***

In this case, the *fireUntilHalt* method is usually started in an independent thread so that the Java code can stop it.



# Rule syntax : DRL

---

Main elements  
Rule's attributes

**LHS**

RHS

Query, agregation

# LHS



The LHS part is the conditional part of the rule.

It consists of zero or more conditions elements

- If no condition element, the LHS is set to *true* and will be activated when the working memory is created
- Conditional elements consist of *patterns* that are implicitly connected by **and**





# Pattern



A pattern consists of:

- A binding pattern to create a variable used in the rule (the \$ character is optional but recommended)
- A restriction on the type (A fact, an interface, an abstract class)
- A set of constraints linked by operators

Ex: \$ c : Cheese ( )



# Constraint's syntax

---

2 types of syntax are available:

- **Field's constraints**

- Concern only one attribute
- Combined with && , || and ()

- **Group 's constraints**

- Concern several attributes of the same fact
- Combined with ',' (which means && with a lower precedence)

# Field's constraints



A field constraint expresses a restriction on a property of the object accessible by getter / setter, it is possible to bind the field on a variable

3 types of restriction are possible:

- Unique value: the field is compared to a single value
- Multiple values: the field is compared to several values
- Multi-constraints: Several constraints are specified on the field

The value of the field can be String, numeric, date (format "dd-mmm-yyyy" by default), boolean or Enum tests on null can be carried out as well as on returns of method



# Single value constraint

---

The operators that can be used are: `<`, `<=`, `>`, `>=`, `==`, `!=`, *contains*, *not contains*, *memberof*, *not memberof*, *matches (expr. régulière)*, *not matches*

```
Cheese( quantity == 5 )
```

```
Cheese( bestBefore < "27-Oct-2009" )
```

```
Cheese( type == "camembert" )
```

```
Cheese( from == Enum.COW)
```

```
Cheese( type matches "(Buffalo)?\\S*Mozarella" )
```

```
CheeseCounter( cheeses contains "stilton" )
```

```
CheeseCounter( cheese memberOf $matureCheeses )
```

```
Person( likes : favouriteCheese ) Cheese( type == likes )
```

```
Person( girlAge : age, sex == "F" ) Person( age == ( girlAge + 2), sex ==  
    'M' )
```



# Multiple values constraint

---

Operators *in* and *not in*, allow to specify multiple values separated by ","

```
Person( $cheese : favouriteCheese )
```

```
Cheese( type in ( "stilton", "cheddar", $cheese ) )
```



# Multiples constraints

---

Multiple constraints allow you to specify several restrictions on the field related by the operators '&&' or '||' and parentheses

```
Person( age > 30 && < 40 )
```

```
Person( age ( (> 30 && < 40) ||  
              (> 20 && < 25) ) )
```

```
Person( age > 30 && < 40 || location == "london" )
```



# Group's constraint

---

The comma, allows to separate the constraints of groups and is equivalent to a less priority AND:

```
Person( age > 50, weight > 80, height > 2 )
```

The comma operator can not be nested in a composite expression:

```
Person( ( age > 50, weight > 80 ) || height > 2 )  
// => compilation ERROR
```



# Other operators

---

***not***: There is no fact in the working memory corresponding to these restrictions

***exists***: There is at least one fact in the working memory corresponding to these restrictions

***forall***: All the facts of the working memory corresponding to the first restriction satisfy the other restrictions

***from***: Used to compare data other than the entire working memory (For example a query, a channel)

***collect***: Lets reason on a collection of objects

***accumulate***: Allows you to perform an aggregate function on a collection of objects





# Examples

---

**#There is no red bus in the memory**

```
not Bus(color == "red")
```

**#There is at least one bus 42 of red color in the memory**

```
exists ( Bus(color == "red", number == 42) )
```

**#All English buses are red**

```
forall( $bus : Bus( type == 'english')  Bus( this == $bus, color =  
    'red' ) )
```

**# Addresses with the correct postal code**

**# who are detained by Person from memory**

```
Person( $personAddress : address )
```

```
Address( zipcode == "23920W") from $personAddress
```



# Examples

---

**# Build a mom list**

**# All females with children**

```
$mothers : LinkedList()
```

```
from collect( Person( gender == 'F', children > 0 ) )
```

**# All orders with total greater than 100**

```
$order : Order()
```

```
$total : Number( doubleValue > 100 )
```

```
from accumulate( OrderItem( order == $order, $value : value ),  
    sum( $value ) )
```



# Rule syntax : DRL

---

Main elements  
Rule's attributes  
LHS  
**RHS**  
Query, agregation



# RHS

---

The right part contains a list of actions to perform.

In general, no conditional code because the rule must be "atomic" (if not separate into several rules)

The operations can act on the working memory and thus trigger the inference:

- Adding facts
- Deleting facts
- Update facts



# Macro-methods

---

Drools supports several macro-methods that avoid retrieving the references of the facts that you want to update:

- ***set : set<field> ( <value> )***

Used to update a field

```
$application.setApproved ( false );  
$application.setExplanation( "has been bankrupt" );
```

- ***modify : modify ( <fact-expression> ) {  
 <expression>,  
 <expression>,  
 ... }***

Used to specify the fields to modify and notify Drools of the change

```
modify( LoanApplication ) {  
    setAmount( 100 ),  
    setApproved ( true )  
}
```



# Macro-methods (2)

---

- **update :**  
**update ( <object, <handle> )**  
**update ( <object> ) // => Find the corresponding fact**  
Used to specify the fact to set and to notify Drools of the change.  
`LoanApplication.setAmount( 100 );`  
`update( LoanApplication );`
- **insert: insert( new <object> );**  
Used to insert a new fact  
`insert( new Applicant() );`
- **insertLogical : *insertLogical(new Something())***  
the object is automatically deleted if the rule is no longer valid.  
`insertLogical( new Applicant() );`



# Macro-methods (3)

---

- **delete : delete( <object> )**  
Used to remove an object from working memory. The retract keyword is also supported  
delete( Applicant )



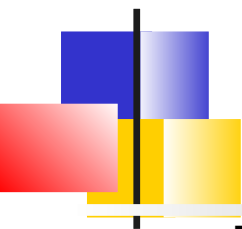
# *drools and RuleContext*

---

The predefined variable ***drools*** of type RuleContext allows to call other useful methods:

- Methods giving information about the rule
  - ***drools.getRule().getName()***: The name of the activated rule
  - ***drools.getMatch()*** : Information on why the rule was activated
- Having a reference on KieRuntime, we can also use
  - ***drools.getKieRuntime().halt()*** : Stop the engine in active mode
  - ***drools.getKieRuntime().getAgenda()*** : Access to the calendar then to the rule groups  
`drools.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp" ).setFocus();`
  - ***drools.getKieRuntime().setGlobal()*** : Set a global
  - ***drools.getKieRuntime().getQueryResults(<string> query)*** :  
Return the result of a query





# *kcontext et KieRuntime API*

Toute l'API de la base de connaissance est également exposée via la variable prédéfinie **kcontext** de type *KieContext*.

Sa méthode **getKieRuntime()** retourne un objet de type *KieRuntime* :

- **getAgenda()** retourne une référence sur l'agenda de la session
- **getQueryResults(String query)** retourne le résultat d'une requête
- **addEventListener, removeEventListener** : enregistrement de listeners pour la mémoire de travail ou l'agenda.
- **getKnowledgeBase()** retourne l'objet *KnowledgeBase* .
- Gestion des variables **globales** avec *setGlobal(...)*, *getGlobal(...)* et *getGlobals()*.
- **getEnvironment()** : l'environnement d'exécution et ses propriétés



# Rule syntax : DRL

---

Main elements  
Rule's attributes  
LHS  
RHS

**Query, agregation**



# Introduction

---

A query, in Drools, can be considered as a rule without its right section.

However, a major difference is that a query can take arguments



# Query definition

---

A query can search for facts in the knowledge base

A request can be parameterized.

The names of the queries are global to the knowledge base  
=> No identical name even in different packages

Its definition is identical to the left part of a rule:

```
query "people over the age of x"  (int x)
```

```
    person : Person( age > x )
```

```
end
```



# Query on demand

---

The result of a query is obtained by :

***ksession.getQueryResults("name")***

It is then possible to iterate on the returned lines

```
QueryResults results = ksession.getQueryResults( "people over the age of x" ,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" );  
}
```



# Live queries

---

Drools also allows you to attach a listener to a request in order to be informed of the results as soon as they are available

These are the live requests, they are executed via the method  
***openLiveQuery()***

```
public LiveQuery openLiveQuery(String query,  
                                Object[] arguments,  
                                ViewChangedEventListener listener);
```



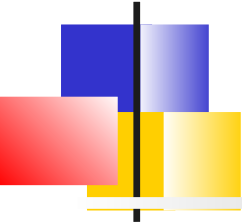
# Interface *ViewChangeListener*

---

Interface ***ViewChangeListener*** has 3 methods:

```
public interface ViewChangeListener {  
    public void rowInserted(Row row);  
    public void rowDeleted(Row row);  
    public void rowUpdated(Row row);  
}
```

The interface therefore allows you to be warned when inserting, updating and deleting facts respecting the request



# Other ways to express rules

---

## **Decision tables**

Rule's templates

DMN





# Presentation

---

Decision tables are an efficient and compact way to represent conditional logic, they are tailored to business experts

The data entered in a spreadsheet makes it possible to generate the rules.  
=> the business expert then benefits from his favorite tool: Excel

For each line of the decision table, the data is combined with a template to generate a rule.

Decision tables allow you to encapsulate rules and isolate the object model. Only the parameters of the rules that can be modified are exposed.

# Example

	B	C	D	E
7				
8				
9		RuleSet	Some business rules	
10		Import	org.drools.decisiontable.Cheese, org.drools.dec	
11		Sequential	true	
12				
13		RuleTable Cheese fans		
14		CONDITION	CONDITION	ACTION
15		Person	Cheese	list
16	(descriptions)	age	type	add(* \$param*)
17	Case	Persons age	Cheese type	Log
18	Old guy	42	stilton	Old man stilton
19	Young guy	21	cheddar	Young man cheddar
20				
21		Variables	java.util.List list	



# Template syntax

---

Decision tables have 2 types of columns:

- Condition columns  $\Leftarrow \Rightarrow$  LHS, the constraint syntax must be used
- Action columns  $\Leftarrow \Rightarrow$  RHS, the code syntax must be used

\$ param is used to indicate where cell data will be inserted (\$ 1 can be used)

If the cell contains a value list separated by commas, the symbols \$ 1, \$ 2, and so on. can be used.

The forall (DELIMITER) {SNIPPET} function can be used to loop through all available values.



# Condition columns

The rendering of a condition depends on the presence of a declaration of an object type in a line above.

If the type is specified, a type constraint is created.

If the cell contains just one attribute, the constraint will be an equality constraint, otherwise the cell will include an operator.

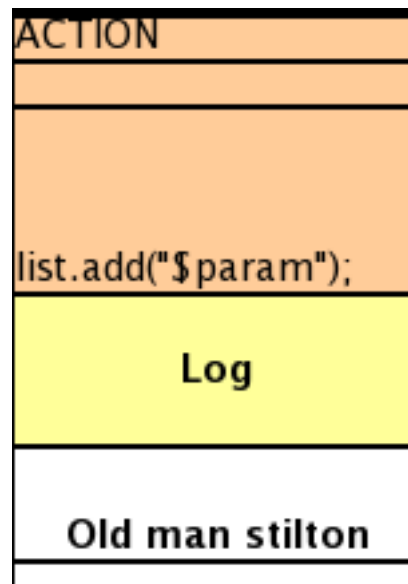
13	<b>RuleTable Cheese fans</b>	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton



# Consequences

The result of an action cell depends on the presence of an entry on the line immediately above.

- If there is nothing, the cell is interpreted as it is
- If there is a variable, the contents of the cell are added to the variable (Method Call)





# Keywords

---

Before the keyword *RuleTable*, the following keywords may be present and condition their cell immediately to the right:

- ***RuleSet*** : Specifying the name of the rule group, if empty it is the default group
- ***Sequential*** : The cell on the right contains *true* or *false*. If true, the *salience* property is used to guarantee order
- ***Import*** : List of java classes to import
- ***Functions*** : Functions declaration
- ***Variables*** : Global variables declaration
- ***Queries*** : Queries declaration



# Example

---

RuleSet	Control Cajas[1]
Import	foo.Bar, bar.Baz
Variables	Parameters parametros, RulesResult resultado, EvalDate fecha
Functions	<pre>function boolean isRango(int iValor, int iRangoInicio, int iRangoFinal) {     if (iRangoInicio &lt;= iValor &amp;&amp; iValor &lt;= iRangoFinal)         return true;     return false; }  function boolean esIgualTipo(TipoVO tipoVO, int p_tipo, boolean isNull) {     if (tipoVO == null)         return isNull;     return tipoVO.getSecuencia().intValue() == p_tipo; }</pre>



# *RuleTable*

---

A cell with ***RuleTable*** indicates the beginning of the definition of a rule table.

The table starts with the next line. It is read from left to right and from bottom to top to a white line.





# Keywords in the rules table

---

**CONDITION** : Indicates a condition column

**ACTION** : Indicates an action column

**PRIORITY** : Indicates a column used for the *salience* attribute

**DURATION** : Indicates the *duration* attribute of the rule

**NAME** : The name of the rule (optional)

**NO-LOOP** : Attribute *no-loop* (*true* or *false*)

**ACTIVATION-GROUP** : Attribute *activation-group*

**AGENDA-GROUP** : Attribute *agenda-group* of the rule

**RULEFLOW-GROUP** : Attribute *ruleflow-group* of the rule



# Integration

---

The integration of a decision table requires the library ***drools-decisiontables.jar***

The main class ***SpreadsheetCompiler*** takes as input a csv or excel file and generates the rules in DRL

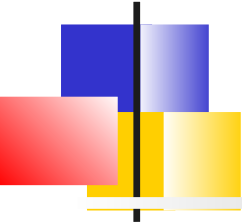
The rules can then be manipulated independently



# Steps

---

1. The business expert starts from a decision table template
2. It informs the parameters of the rules and actions with business descriptions
3. They enter the lines corresponding to the rules
4. The decision table is taken over by a technician who maps the business language to scripts
5. The business expert and the technician review together the changes made.
6. The business expert can edit the rules according to his needs.
7. The technician can write test cases that check the rules



# Other ways to express rules

---

Decision tables  
**Rule's templates**  
DMN



# Rules template

---

***Rule templates*** use tabular data sources (Spreadsheets, CSV, or others) to generate many rules.

This is a technique that is ultimately more powerful than the decision tables:

- Data can be stored in a database
- Rule generation can be conditioned by data
- The data can be used in any part of the rules (operator, name of a class, name of a property)
- Several templates can be run on the same data



# Structure of a template

---

## The text file

- starts with a template header header.
- Then the list of columns of tabular data
- A blank line to mark the end of the column definitions
- The standard DRL headers (package, import, global, functions)
- The **template** keyword marks the beginning of a rule template ; several templates can be defined in the same file.
- The template uses the syntax **@ {token\_name}** for substitutions (ex: @ {row.rowNumber})
- The keyword **end template** marks the end of the template.



# Exemple

---

**template header**

age  
type  
Log

```
package org.drools.examples.templates;
```

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

**template** "cheesefans"

```
rule "Cheese fans_{row.rowNumber}"
```

```
when
```

```
Person(age == @{age})
```

```
Cheese(type == "{@type}")
```

```
then
```

```
list.add("{@log}");
```

```
end
```

**end template**



# *kmodule*

---

The template must then be included with the associated data file in the *kmodule* definition

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://drools.org/xsd/kmodule">
  <kbase name="TemplatesKB" packages="org.drools.examples.templates">
    <ruleTemplate
      dtable="org/drools/examples/templates/ExampleCheese.xls"
      template="org/drools/examples/templates/Cheese.drt"
      row="2" col="2"/>
    <ksession name="TemplatesKS"/>
  </kbase>
</kmodule>
```

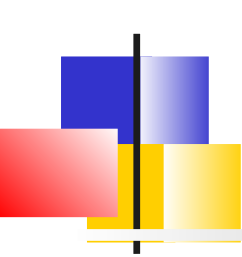




# Sample with database

---

```
// Get results from your DB query...
resultSet = preparedStmt.executeQuery();
// Generate the DRL...
resultSetGenerator = new ResultSetGenerator();
String drl = resultSetGenerator.compile(resultSet,
    new
    FileInputStream("path/to/template.drt"));
```



# Other ways to express rules

---

Decision tables  
Rule's templates  
**DMN**



# Introduction

---

Decision Model Notation is a standard published by OMG (like BPMN2)

It is supported in Drools since latest 7.x

The primary goal is to provide a standard notation that is readily understandable by :

- **Business Analysts** : They can define the initial decision requirements
- **Technical Developers** : They can create complex decision logic and automate the decisions;
- **Business Stakeholders** : They can manage and monitor the decisions.



# Main components



**Decisions** determine an output value depending on:

- their input data (input nodes or the output value from other decisions)
- their decision logic - boxed expressions that may reference functions from BKM nodes



**Input data** : input for one or more decisions. When enclosed within a Business Knowledge Model (BKM), they indicate parameters for the BKM node.

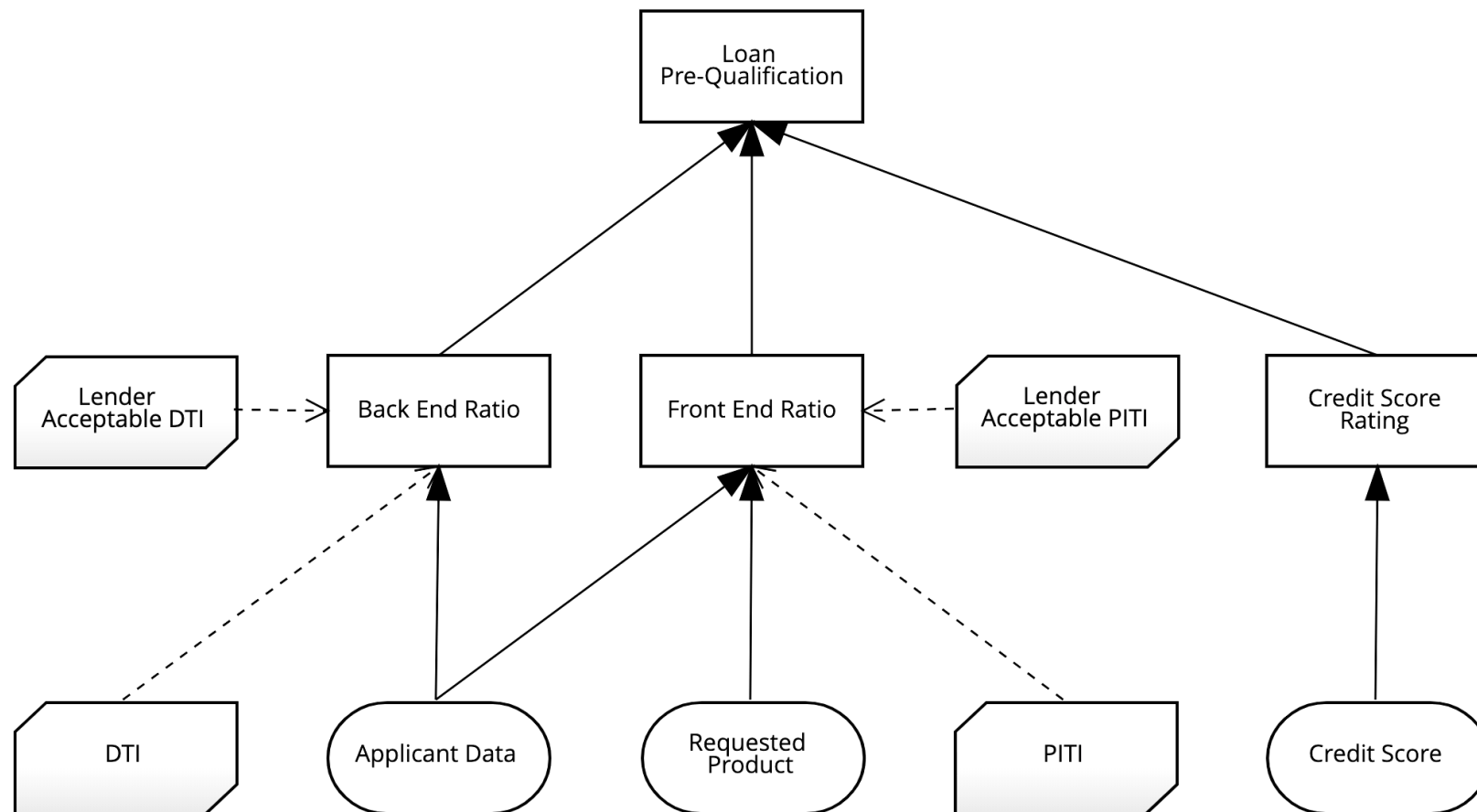


**BKM**s encapsulate business knowledge as reusable functions.



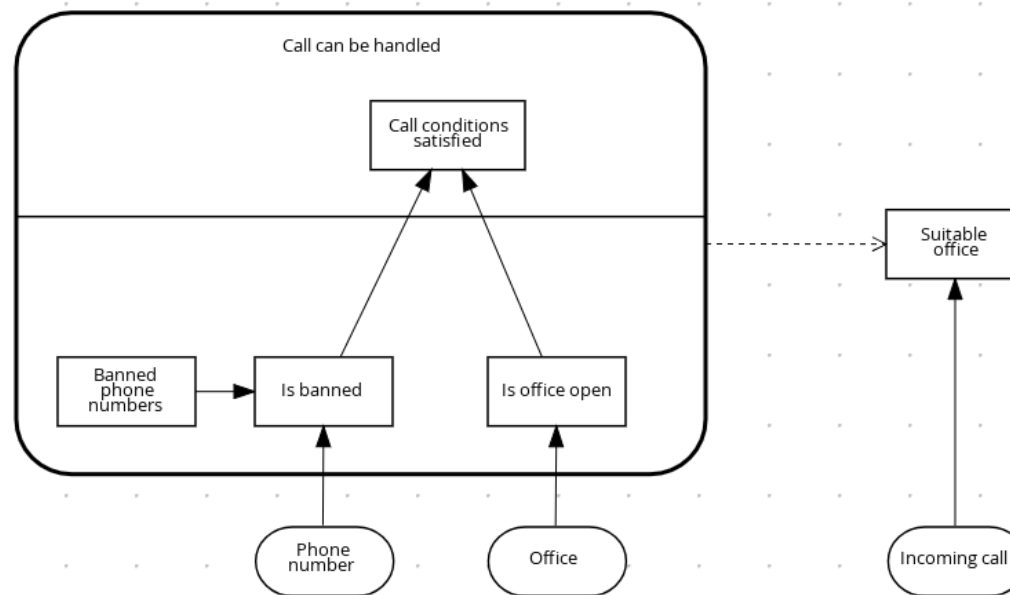
A **knowledge source** denotes an authority for a BKM or a decision node.

# Example



# Decision service

Some parts of the graph can be externalized in a *Decision Service*





# Decision node

---

Decision nodes may express their logic by a variety of boxed expressions

- FEEL expression that produces the output value
- Contexts represent a collection of one or more key-value pairs where the value is a decision logic, and the key is the respective identifier
- Decision tables
  - Input columns
  - Output columns
  - Hit policy (unique, ...)
- Relations encapsulate lists of expressions
- Functions define reusable operations into your model. They are generally associated to BKM
- Invocation : map the invocation for business knowledge model nodes
- List represent a group of FEEL expressions.



# FEEL

---

The **FEEL** (*Friendly Enough Expression Language*) is intended as a common ground between business analysts, programmers, domain experts and stakeholders.

It provides :

- Side-effect free
- Simple data model with numbers, dates, strings, lists, and contexts
- Simple syntax designed for a broad audience
- Three-valued logic (true, false, null)





# Data Types

---

- Select a data type - ▾

DEFAULT

Any

boolean

context

date

date and time


days and time duration

number

string

time

years and months duration

 Structure



# Language

---

## **if expression**

if 20 > 0 then "YES" else "NO" //→ "YES"

## **for expression**

for i in [1, 2, 3] return i \* i //→ [1, 4, 9]

## **some (name) in (list) satisfies (predicate)**

some i in [1, 2, 3] satisfies i > 2 //→ true

## **every (name) in (list) satisfies (predicate)**

every i in [1, 2, 3] satisfies i > 1 //→ false

## **in expression**

1 in [1..10] //→ true

## **Three-valued logic(and, or)**

true and true //→ true

true and false and null //→ false

true and null and true //→ null

true or false or null //→ true



# DMN support in Drools

---

Drools engine provides runtime support for DMN 1.1, 1.2, 1.3, and 1.4 models at conformance level 3.

KIE DMN Editor provides design support for DMN 1.2 models at conformance level 3.

DMN models can be integrated by :

- Design your DMN models using the KIE DMN Editor online.
- Design your DMN models using the KIE DMN Editor in VSCode.
- Import DMN files into your project by opening them in KIE DMN Editor.
- Package DMN files as part of your project knowledge JAR (KJAR) file without KIE DMN Editor.



# DMN model execution

---

The model execution differs if you use embedeed packaging or remote external service (Kogito)



# Embedded, Get the runtime and the model

---

```
// Create a KieContainer from ReleaseId
```

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

```
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "my-kjar", "1.0.0" );
```

```
KieContainer kieContainer = kieServices.newKieContainer( releaseId );
```

```
// Or from the classpath
```

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

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

```
// Obtain DMNRuntime and a reference to the DMN model to be evaluated,
```

```
// by using the model namespace and modelName
```

```
DMNRuntime dmnRuntime = KieRuntimeFactory.of(kieContainer.getKieBase()).get(DMNRuntime.class);
```

```
String namespace = "http://www.redhat.com/_c7328033-c355-43cd-b616-0aceef80e52a";
```

```
String modelName = "dmn-movieticket-ageclassification";
```

```
DMNModel dmnModel = dmnRuntime.getModel(namespace, modelName);
```



# Execution

```
// Instantiate a new DMN Context to be the input for the model evaluation
DMNContext dmnContext = dmnRuntime.newContext();

for (Integer age : Arrays.asList(1,12,13,64,65,66)) {
    // Assign input variables for the input DMN context
    dmnContext.set("Age", age);
    // Evaluate all DMN decisions defined in the DMN model.
    DMNResult dmnResult =
        dmnRuntime.evaluateAll(dmnModel, dmnContext);

    // Each evaluation may result in one or more results,
    for (DMNDecisionResult dr : dmnResult.getDecisionResults()) {
        log.info("Age: " + age + ", " +
            "Decision: '" + dr.getDecisionName() + "', " +
            "Result: " + dr.getResult());
    }
}
```



# Related Projects

---

**Complex Event Processing**  
Drools and jBPM



# Introduction

---

***Complex event processing (CEP)*** allow to make decisions based on time relationships between facts.

The main focus of CEP is to correlate small units of time-based data within an ever-changing, ever-growing data cloud in order to react to hard-to-find special situations

Reasoning is made on **events** which are facts with a time of occurrence





# Complex event

---

A **complex event** is simply an aggregation, composition, or abstraction of other events

Rules will be expressed via complex events using aggregation, composition or abstraction



# Semantic of events

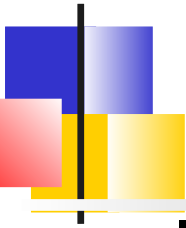
---

2 kinds of events are considered :

- **Punctual** event
- **Interval** event

All events are :

- Immutable
- A managed life cycle : when it is too old, it is removed from the session



# Declaring time-based-events

---

In order to create the CEP rules, we must inform the engine which types of objects must be treated as events

It is done by adding meta-data :

- **@Role** : fact or event
- **@Timestamp** : The attribute which gives the time of occurrence. If not present, the timestamp is the time of insertion
- **@Duration** : The attribute which gives the duration of the event. Optional
- **@Expires** : The attribute which gives the life time in the session



# Sample

---

```
@org.kie.api.definition.type.Role(Role.Type.EVENT)
@org.kie.api.definition.type.Duration("durationAttr")
@org.kie.api.definition.type.Timestamp("executionTime")
@org.kie.api.definition.type.Expires("2h30m")
public class TransactionEvent implements Serializable {
    private Date executionTime;
    private Long durationAttr;
    /* class content skipped */
}
```



# Sample (2)

---

```
declare PhoneCallEvent
  @role(event)
  @timestamp(whenDidWeReceiveTheCall)
  @duration(howLongWasTheCall)
  @expires(2h30m)
  whenDidWeReceiveTheCall: Date
  howLongWasTheCall: Long
  callInfo: String
end
```



# Temporal operators

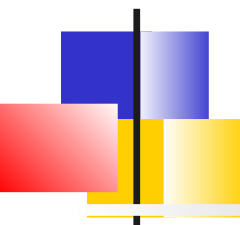
---

There are 13 temporal operators available which allow to correlate events

For example :

```
declare MyEvent
@role(event)
@timestamp(executionTime)
End
```

```
rule "my first time operators example"
when
    $e1: MyEvent()
    $e2: MyEvent(this after[5m] $e1)
Then
    System.out.println("We have two events" + " 5 minutes apart");
end
```



Operator		Point - Point	Point - Interval	Interval - Interval
A before B	A B			
A after B	A B			
A coincides B	A B			
A overlaps B	A B			
A finishes B	A B			
A includes B	A B			
A starts B	A B			
A finishedby B	A B			
A startedby B	A B			
A during B	A B			
A meets B	A B			
A metby B	A B			
A overlappedby B	A B			

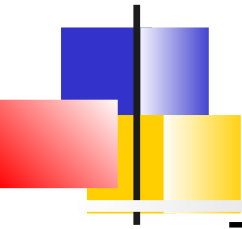


# Another sample

---

```
rule "More than 10 transactions in an hour from one client"
when
    $t1: TransactionEvent($cId: customerId)
    Number(intValue >= 10) from accumulate(
        $t2: TransactionEvent(this != $t1,
            customerId == $cId, this meets[1h] $t1),
        count($t2) )
    not (SuspiciousCustomerEvent(customerId == $cId,
        reason == "Many transactions"))
then
    insert(new SuspiciousCustomerEvent($cId,
        "Many transactions"));
end
```





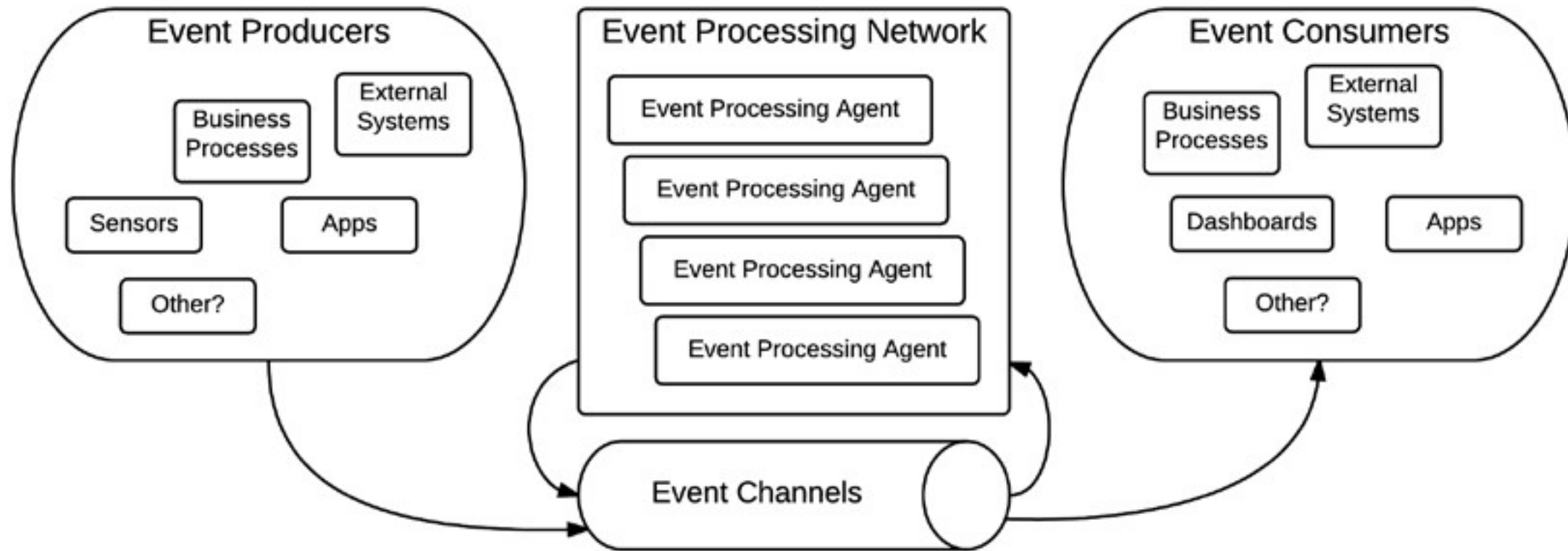
# Event-driven architecture

---

The idea of event-driven architecture (EDA) is to classify the components in the following four different categories:

- Event Producer : Creators of events, for example a sensor
- Event Consumer : Final output architecture which point the produced value. For example a dashboard
- Event Channels : Communication protocols between all the other components. For example JMS
- Event Processing Agents: Group the events to detect and process complex events. The Drools rules

# Event-driven architecture





# Entry points

---

**Entry points** are a way to partition the working memory

Rules can express conditions about events from one particular source

Entry points are declared implicitly by using them in rules

At insertion, the entry point can be specified :

```
ksession.getEntryPoint("myEntryPoint").insert(new Object());
```



# Sample

---

```
// Insert event from one entry-point to another
rule "Routing transactions from small resellers"
when
    $t: TransactionEvent() from
        entry-point "small resellers"
then
    entryPoints["Stream Y"].insert(t);
end
```



# Sliding windows

---

**Sliding windows** allow to filter the events of the working memory or any entry point

2 kinds of sliding windows :

- **Length-based** : Number of elements
- **Time-based** sliding : Elements that happened within a specific time elapsed from now

Sliding windows can be defined

- inside a rule
- or outside for reuse



# Length-based sample

---

```
rule "last 6 transactions are more than 100 dollars"
When
    Number(doubleValue > 100.00) from accumulate(
        TransactionEvent($amount: totalAmount)
        over window:length(6),
        sum($amount))
then
    //... TBD
end
```



# Time-based sample

---

```
rule "obtain last five hours of operations"
when
    $n: Number() from accumulate(
        TransactionEvent($a: totalAmount)
        over window:time(5h),
        sum($a)
    )
Then
    System.out.println("total = " + $n);
end
```



# Declared sliding windows sample

---

```
declare window Beats
```

```
  @doc("last 10 seconds heart beats")
```

```
  HeartBeat() over window:time( 10s )
```

```
    from entry-point "heart beat monitor"
```

```
end
```

```
rule "beats in the window"
```

```
  when
```

```
    accumulate(HeartBeat() from window Beats,
```

```
      $cnt : count(1))
```

```
  then
```

```
    // there has been $cnt beats over the last 10s
```

```
end
```





# Running CEP-scenarios

---

Both the Kie Base and Session that run the CEP cases need special management :

- Kie Base must be configured to support CEP
- Define the way to fire rules : *discrete* or *continous*
- The Kie Session internal clock used to evaluate temporal events must be set



# *KieBase*

---

Kie Base must be configured to use the **STREAM** event processing mode.

This configuration informs the runtime that it should manage events and keep them internally ordered by their timestamp

In *kmodule.xml*

```
<kbase name="cepKbase" eventProcessingMode="stream"
packages="chapter06.cep">
<ksession name="cepKsession"/>
</kbase>
```



# Discrete vs Continuous

---

Firing rules can be done :

- At specific point of time.  
After inserting fact, we call *fireAllRules*
- Continuously  
*fireUntilHalt*

If we have a scenario where the absence of events will trigger a rule, we have to use the continuous way

If the only thing that could trigger new rules is the insertion of new events then discrete rule firing will be enough



# Session clock

---

By default, Kie Sessions will use the clock of the machine on which is running

For testing scenarios, we can use pseudo-clock

For complex distributed scenarios, we can configured in synchronized clock



# Pseudo-clock sample

---

## Configuration :

```
<kbase name="cepKbase" eventProcessingMode="stream"  
packages="chapter06.cep">  
  <ksession name="cepKsession" clockType="pseudo"/>  
</kbase>
```

## Usage in test case for example :

```
SessionPseudoClock clock = ksession.  
getSessionClock();  
clock.advanceTime(2, TimeUnit.HOURS);  
clock.advanceTime(5, TimeUnit.MINUTES);
```



# Related Projects

---

Complex Event Processing  
**Drools and jBPM**



# Introduction

---

Drools and jBPM complement each other, allowing end users to describe business knowledge using different paradigms : Rules and processes

They shared :

- The same API
- The same integration patterns with a business application
- The same mechanisms for building and deploying



# Accessing processes from rules

---

In the action side, a rule has access to *kcontext* and *kcontext.getKieRuntime()*

With this object reference, it is possible to :

- create, abort, and signal processes
- Access the WorkItemManager to complete WorkItems





# Sample

---

```
rule "Validate OrderLine Item's cost"
```

```
  when
```

```
    $ol: OrderLine()
```

```
  then
```

```
    Map<String, Object> params = new HashMap<String, Object>();
```

```
    params.put("requested_amount", $ol.getItem().getCost());
```

```
    kcontext.getKieRuntime().startProcess("simple", params);
```

```
end
```



# Process instances as facts

---

Insertion of Process Instances as facts in the Rule Engine allow to write rules about our processes or groups of processes

The listener ***RuleAwareProcessEventLister***, provided by jBPM, automatically insert our ProcessInstances and update them whenever a variable is changed

The processes need to include Async activities



# Sample

---

```
rule "Too many orders for just our Managers"
  when
    List($managersCount:size > 0) from collect(Manager())
    List(size > ($managersCount * 3)) from
      collect(WorkflowProcessInstance(processId == "process-
order"))
  then
    //There are more than 3 Process Order Flows per manager.
    // Please hire more people :)
  end
```



# BPMN2 Business Rule Tasks

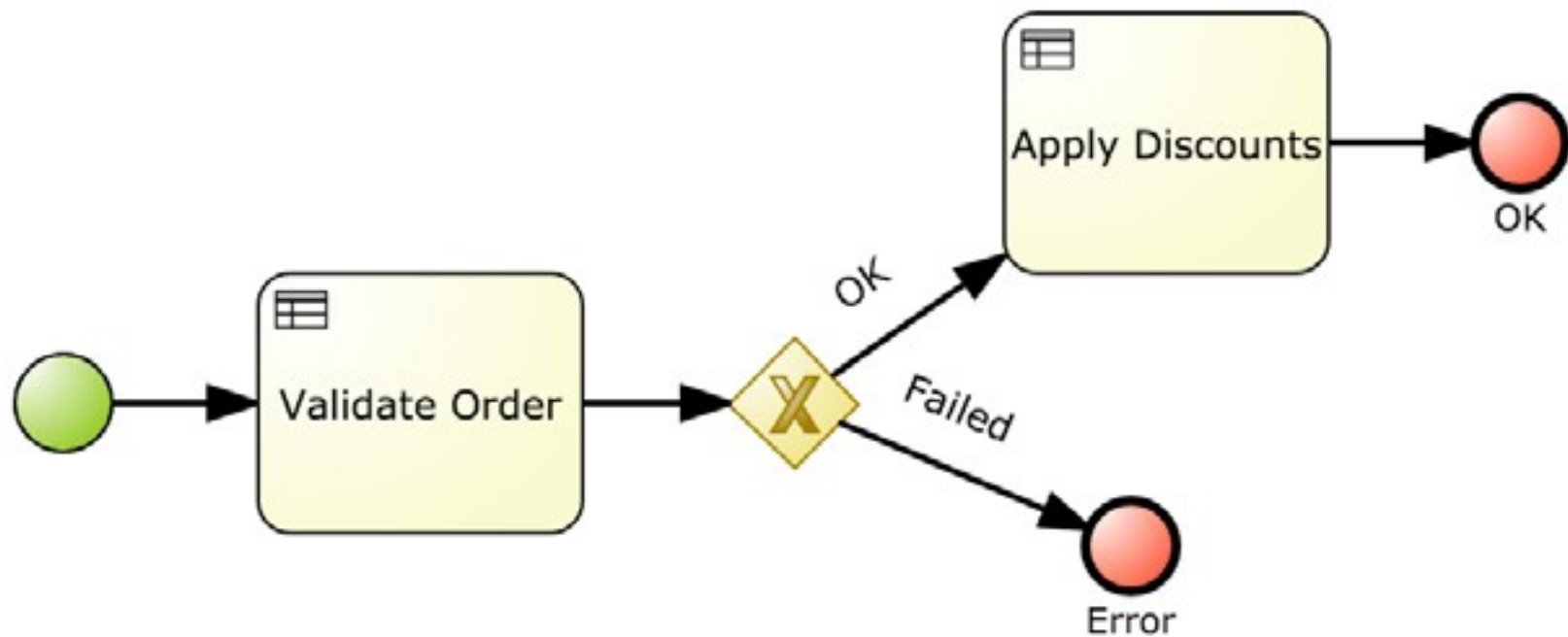
---

BPMN2 specification proposes a specific type of task called a ***Business Rule Task***.

It is used in conjunction with the Rule Property called ***ruleflow-group*** .

This property allows to specify which rules can be fired when the Business Rule Task is executed as part of a process instance

# Sample





# Integrate Kie

---

## **Considerations**

Patterns of integration

Maven knowledge repositories

Externalizing knowledge services



# Drools Usage

---

Drools can interact with any and all layers of our application, depending on what we expect to accomplish with it

- It could interact with the UI to provide complex form validations
- It could interact with data sources to load persisted data when a rule evaluation determines it is needed
- It could interact with outside services, to either load complex information into the rules or send messages to outside services about the outcome of the rules execution



# Engine/application communication

---

Some mechanisms to communicate  
between Drools and the rest of the  
application :

- Global variables may contain domain objects
- Entry points : Stream of facts and event
- Channels to send information to the outside world
- Listeners or marshallers





# Synchronous mode

---

If we need to execute our business rules in a synchronous manner

- Using stateless Kie Sessions and create as many sessions as requests
- Use global variables to store specific rule execution information



# *Asynchronous*

---

The asynchronous mode is necessary if the absence of input causes the triggering of a rule.

It can also be used if the components interact with the motor asynchronously

In this case :

- Kie Sessions may be shared between different threads, because some of them may insert new information and others might take care of firing rules (fireUntilHalt)
- Register special listeners that take care of notifying other components about special situations detected by the rules
- Entry points become a very useful component when multiple sources will be inserting information into a single Kie Session



# Integrate Kie

---

Considerations

**Patterns of integration**

Maven knowledge repositories  
Externalizing knowledge services



# Scenarios of integration

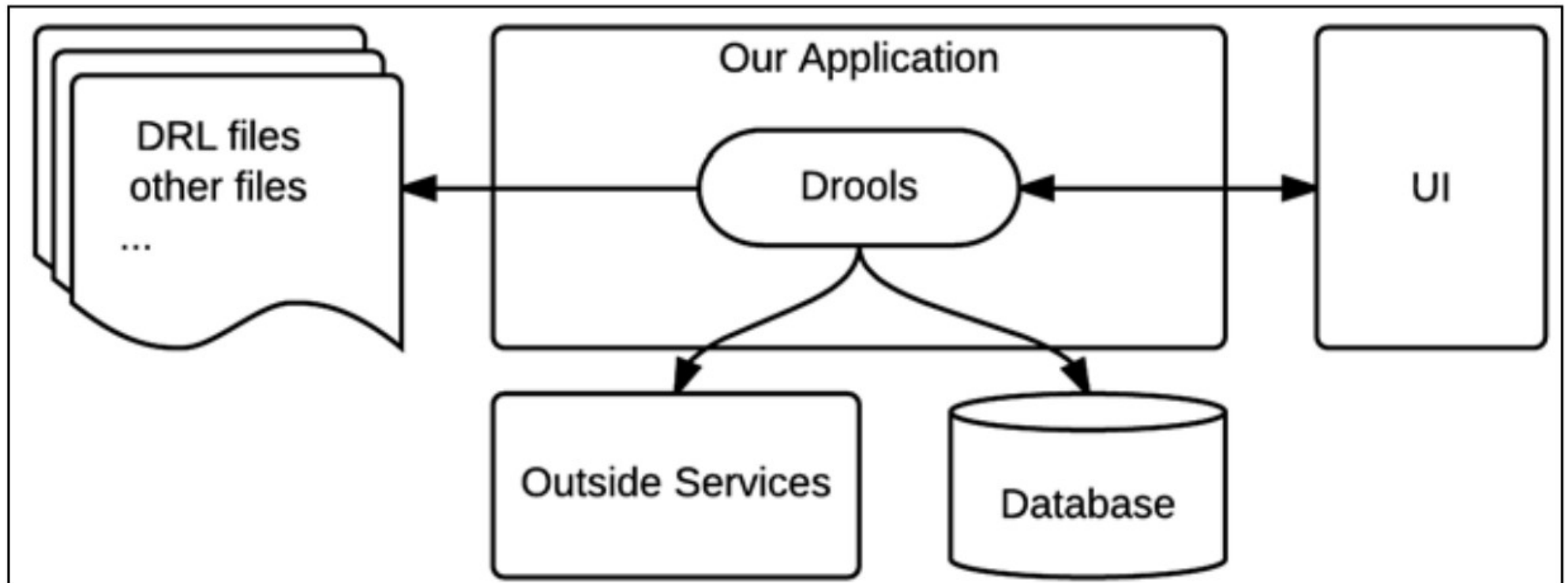
---

Different ways of integration can be considered :

- Embed Drools runtime in your application
- Externalize rules in a separate repository
- Knowledge as a service

For these scenarios, Drools offers supports for CDI, Spring and Camel

# Architecture





# Distinct Maven Repository

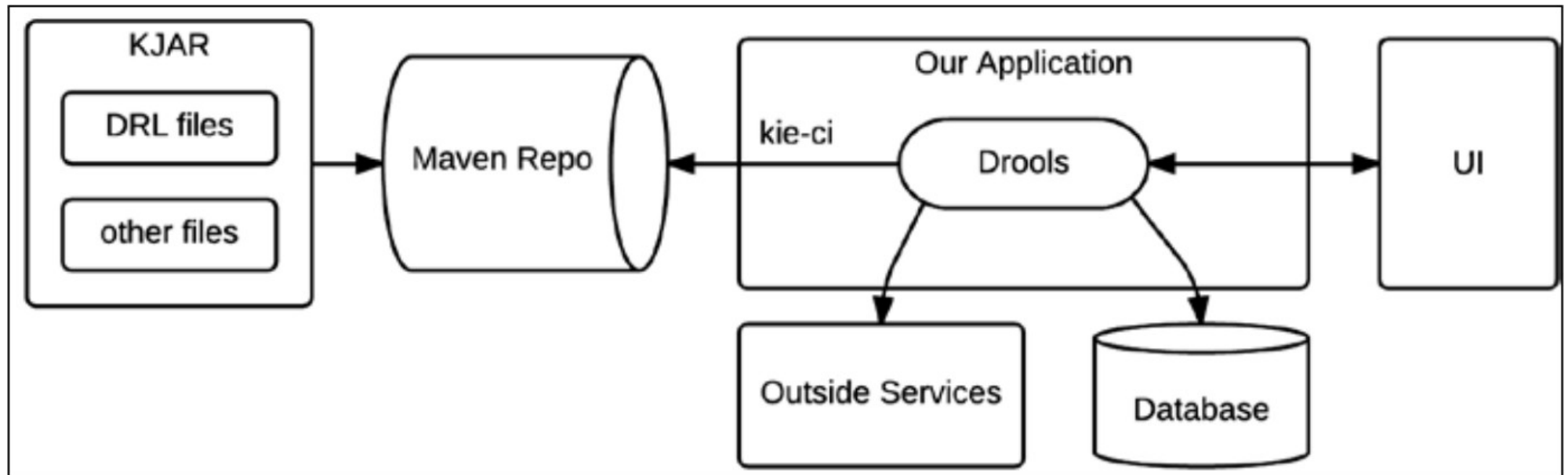
---

Rules can be deployed independently from client application by using a Maven repository :

- business rules are deployed in the Maven repository as a KJAR
- Components Drools runtime can dynamically load the rules

=> The rules can be developed, deployed, and managed as many times as needed without having to redeploy the applications.

# Architecture





# Loading the rules

---

## With CDI

```
@Inject @KSession
@KReleaseId(groupId = "org.drools.devguide", artifactId = "chapter-11-
    kjar",
version = "0.1-SNAPSHOT")
KieSession kSession;
```

## With a scanner

```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.newKieContainer(
    ks.newReleaseId("org.drools.devguide",
        "chapter-11-kjar", "0.1-SNAPSHOT"));
KieScanner kScanner = ks.newKieScanner(kContainer);
kScanner.start(10_000);
KieSession kSession = kContainer.newKieSession();
```





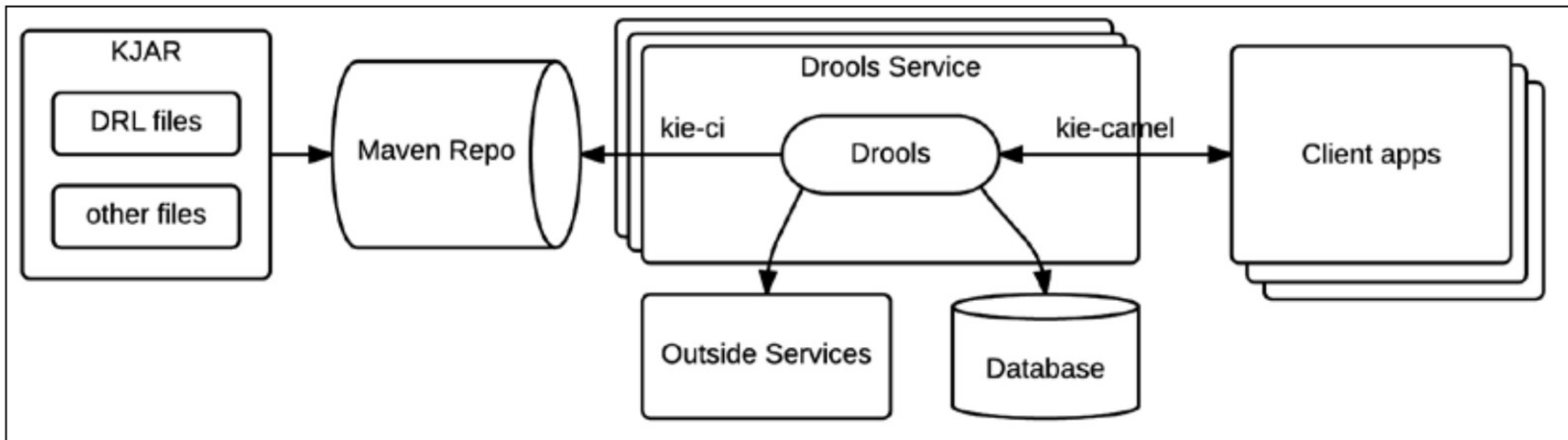
# Knowledge as a service

---

The last approach is to externalize the Drools runtime in an external services.

- This service expose une API (REST or other)
- Multiple-applications can access the service
- The service may be replicated if needed
- Life cycle of rules are completely independent

# Architecture





# Integrate Kie

---

Considerations

Patterns of integration

**Maven knowledge repositories**

Externalizing knowledge services



# Maven knowledge repositories

---

**Building a kjar**  
Business resources update



# Introduction

---

One of the primary goals when using a rule/process engine is to decorrelate the business rules/processes from the application that uses it.

Kie projects relies on Maven to deploy business rules/processes in a repository.

Applications can then:

- Either declare a Maven dependency to the artifact of containing the business resources
- The Maven repository can be local or remote



# Building *kjar* with Maven

---

A "Kie resources" project is typically a Maven project plus a *kmodule.xml* file configuring the various knowledge bases

The build cycle is enriched with the Kie plugin that precompiles the rules and ensures their validity

```
<packaging>kjar</packaging>

...

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



# Default configuration

---

*kmodule.xml* defines knowledge bases and sessions that can be used in the project

An empty file gives a default configuration:

- A single knowledge base including all rules / processes / etc. found in the *resources* directory of jar
- A stateful session and a stateless session



# Another *kmodule.xml*

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance " xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud" equalsBehavior="equality">
    <ksession name="KSession2_1" type="stateful" default="true"/>
    <ksession name="KSession2_1" type="stateless" default="false" beliefSystem="jtms"/>
  </kbase>
  <kbase name=""KBase2" default="false" eventProcessingMode="stream" equalsBehavior="equality"
    declarativeAgenda="enabled"
    packages="org.domain.pkg2,org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_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>
```





# Attributes of *kbase*

nom	défaut	autorisé	description
<b><i>name</i></b>	none		Mandatory. Used to retrieve the base from the container
<b><i>includes</i></b>	none	Liste d'item séparés par des virgules	Knowledge bases to include
<b><i>packages</i></b>	all	Liste d'item séparés par des virgules	List the resources to compile
<b><i>default</i></b>	false	true/false	You do not have to specify the name for the default knowledge base
<b><i>equalsBehaviour</i></b>	identity	identity/equality	Test of presence of a fact in the knowledge base
<b><i>eventProcessingMode</i></b>	cloud	cloud/stream	Stream for drools-fusion
<b><i>declarativeAgenda</i></b>	disabled	disabled/enabled	Declarative agenda activated or not. experimental, the rules can act directly on the activations present in the agenda



# Attributes of *ksession*

nom	default	values	description
<b><i>name</i></b>	none		Mandatory. Name used to retrieve a container session
<b><i>type</i></b>	stateful	stateless/ stateful	
<b><i>default</i></b>	false	true/false	It is not necessary to specify the name for the default session
<b><i>clockType</i></b>	realtime	realtime/ pseudo	Indicates whether the timestamp of events is provided by the system or by the application
<b><i>beliefSystem</i></b>	simple	simple/jtms/ defeasible	System of fact management



# Elements of *ksession*

---

A `<ksession>` element can also define different sub-elements:

- A **logger** is used to define a trace file that will record all Drools events in a file
- **WorkItemHandlers** allows you to define task managers that are associated with specific Drools-flow nodes (ex: Human task)
- **Event listeners**: *ruleRuntimeEventListener*, *agendaEventListener*, or *processEventListener*



# Maven and Release ID

---

If the KIE project is a Maven project, it is possible to use the Maven coordinates to instantiate the container.

```
KieServices kieServices = KieServices.Factory.get();  
ReleaseId releaseId = kieServices.newReleaseId( "org.acme", "project", "1.0" );  
KieContainer kieContainer = kieServices.newKieContainer( releaseId );  
  
KieSession kSession = kieContainer.newKieSession("ksession1");
```



# Maven knowledge repositories

---

Building a kjar  
**Business resources update**



# Deployment

---

Deployment consist of updating a Maven repository.

The client application can then :

- Be manually updated to use a new version of the *kjar*
- Be dependent of the latest version
- Be dependent of a family of version



# *KieScanner*

---

***KieScanner*** allows to constantly scan the Maven repository to check if a new release of the project is available

In this case, the new release is deployed in the *KieContainer* and the new rules are taken into account

The use of *KieScanner* requires the presence of ***kie-ci.jar*** in the classpath



# Scanner registration

---

```
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 KieScanner

// which scans the repository every 10 seconds

kScanner.start( 10000L );
```





# Upgrade

---

Automatic update is effective if:

- the version of the artifact is suffixed with SNAPSHOT, LATEST RELEASE, or a version interval
- KieScanner finds an update in the Maven repository

The update consists of downloading the new version and performing an incremental build.

*KieBases* and *KieSessions* controlled by *KieContainer* are automatically updated and all new *KieBases* or *KieSessions* use the new version of the project.



# Integrate Kie

---

Considerations  
Patterns of integration  
Maven knowledge repositories  
**Externalizing knowledge services**



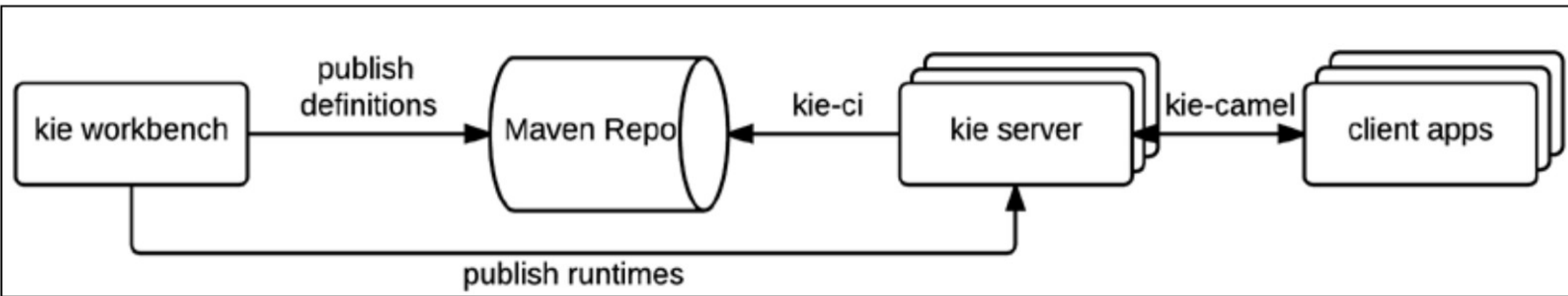
# Introduction

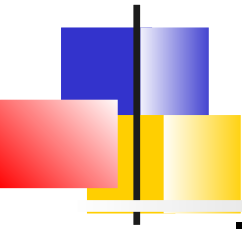
---

KIE provides a module called ***kie-server*** , which can be used to configure similar environments that are only responsible for running Drools rules and processes, taking the kJAR from outside sources

It provides also ***kie-workbench*** which can manage rules and processes : edit, build deploy

# Architecture





# Kie Server

---

It is a modular, standalone server component that can be used to execute rules and processes, configured as a WAR file

It is available for web containers and JEE6 and JEE7 application containers

It can be easily deployed in cloud environments

Each instance of the Kie Server can manage many Kie Containers

Its functionality can be extended through Kie Server Extensions



# Configuration

---

The distributions of Kie Server : archive zip or Docker image use a jBoss wildfly server in the full configuration.

Once a user with the role *kie-server* configured

The REST API is available at :

<http://localhost:8080/kie-server/services/rest/server/>

A swagger documentation is available

<http://localhost:8080/kie-server/docs/>



# Sample Response

---

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<response type="SUCCESS" msg="Kie Server info">
  <kie-server-info>
    <capabilities>KieServer</capabilities>
    <capabilities>BRM</capabilities>
    <capabilities>BPM</capabilities>
    <capabilities>CaseMgmt</capabilities>
    <capabilities>BPM-UI</capabilities>
    <capabilities>BRP</capabilities>
    <capabilities>DMN</capabilities>
    <capabilities>Swagger</capabilities>
    <location>http://localhost:8230/kie-server/services/rest/server</location>
    ...
  </kie-server-info>
</response>
```



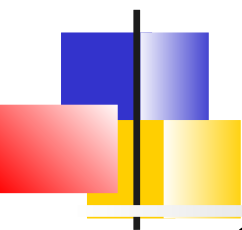
# Native REST client for Execution Server

---

Interaction with the kie-server can be performed by any REST client (curl, postman, Spring restTemplate, ...)

But KIE offers a native rest client suited for this kind of interaction. It is available with the `org.kie:kie-server-client` dependency





# Client request configuration

---

Configuration of the client includes :

- Credentials of the kie-server user
- KIE Server location, such as  
`http://localhost:8080/kie-server/services/rest/server`
- Marshalling format for API requests and responses (JSON, JAXB, or XSTREAM)
- A KieServicesConfiguration object and a KieServicesClient object, which serve as the entry point for starting the server communication using the Java client API
- A KieServicesFactory object defining REST protocol and user access
- Any other client services used, such as RuleServicesClient, ProcessServicesClient, or QueryServicesClient



# Types of client

---

There are different type of client each is specialized in an aspect of KIE :

- ***RuleServicesClient*** : Used to insert/retract facts and fire rules
- ***ProcessServicesClient*** : Used to start, signal, and abort processes or work items
- ***QueryServicesClient***: Used to query processes, process nodes, and process variables
- ***UserTaskServicesClient*** : Used to perform all user-task operations, such as starting, claiming, or canceling a tas
- ...



# Sample (1)

---

```
public static void main(String[] args) {  
    initializeKieServerClient();  
    initializeDroolsServiceClients();  
    initializeJbpmServiceClients();  
}
```



# Sample (2)

---

```
public static void initializeKieServerClient() {
    conf = KieServicesFactory.newRestConfiguration(URL, USER,
PASSWORD);
    conf.setMarshallingFormat(FORMAT);
    kieServicesClient =
KieServicesFactory.newKieServicesClient(conf);
}

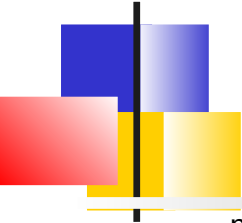
    public static void initializeDroolsServiceClients() {
        ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
        dmnClient =
kieServicesClient.getServicesClient(DMNServicesClient.class);
    }
```



# Sample (3)

---

```
public static void initializeJbpmServiceClients() {  
    caseClient =  
    kieServicesClient.getServicesClient(CaseServicesClient.class);  
    documentClient =  
    kieServicesClient.getServicesClient(DocumentServicesClient.class);  
    jobClient =  
    kieServicesClient.getServicesClient(JobServicesClient.class);  
    processClient =  
    kieServicesClient.getServicesClient(ProcessServicesClient.class);  
    queryClient =  
    kieServicesClient.getServicesClient(QueryServicesClient.class);  
    uiClient = kieServicesClient.getServicesClient(UIServicesClient.class);  
    userTaskClient =  
    kieServicesClient.getServicesClient(UserTaskServicesClient.class);  
}
```



# Sample Use of RuleService

---

```
public void executeCommands() {  
  
    String containerId = "hello";  
    System.out.println("== Sending commands to the server ==");  
    RuleServicesClient rulesClient = kieServicesClient.getServicesClient(RuleServicesClient.class);  
    KieCommands commandsFactory = KieServices.Factory.get().getCommands();  
  
    Command<?> insert = commandsFactory.newInsert("Some String OBJ");  
    Command<?> fireAllRules = commandsFactory.newFireAllRules();  
    Command<?> batchCommand = commandsFactory.newBatchExecution(Arrays.asList(insert, fireAllRules));  
  
    ServiceResponse<String> executeResponse = rulesClient.executeCommands(containerId, batchCommand);  
  
    if(executeResponse.getType() == ResponseType.SUCCESS) {  
        System.out.println("Commands executed with success! Response: ");  
        System.out.println(executeResponse.getResult());  
    } else {  
        System.out.println("Error executing rules. Message: ");  
        System.out.println(executeResponse.getMsg());  
    }  
}
```



# JBPM Commands

---

StartProcessCommand

SignalEventCommand

CompleteWorkItemCommand

AbortWorkItemCommand

QueryCommand



Kie workbenches





# Workbenches

---

Kie projects provide a set of usable workbench tools that allow to create, build, and deploy rule and process definitions in any Kie Server

These workbenches are build with the framework *UberFire* .

They are web application deployed in a application server (Jboss wildfly)



# Installation

---

Download the distribution which is a war  
***kie-wb-\*.war***

Deploy it in the application server

- Drools 7 is only available for Wildfly 14

Workbench is available at :

<http://<server>:<port>/>



# Roles and users

---

A WB use the following roles:

- **admin** : Administrator, manages users, repositories,  
...
- **developer** : Manage assets (rules, models,  
processes, forms. Create build and deploy projects
- **analyst** : Idem developer with restricted right (no  
deployment for example)
- **user** : Participate in business processes and  
perform tasks
- **manager** : Access to reporting



# Getting started

---

Typical actions to start:

- 1) Create a user with the admin rôle
- 2) Add a repository
- 3) Add a project
- 4) Provide the business model
- 5) Create the rules
- 6) Build and deploy

These actions can be done through the web interface,  
an online command tool (`kie-config-cli.sh`) or via a  
REST interface



# Packages

---

Package configuration is usually done only once by someone with expertise with rules and templates

All assets belong to a single package that acts as a namespace

A package is created by specifying its name



# Formats of rules

---

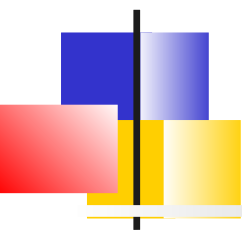
The workbench supports multiple rule formats and editors:

- BRL Format with BRL Editor (also present in Eclipse)
- DSL
- Decision tables in file to upload
- Decision tables edited directly in the web interface
- Rule Flow process to upload
- DRL
- functions
- Configuring lists of values (Enum)
- Rules template powered by data tables

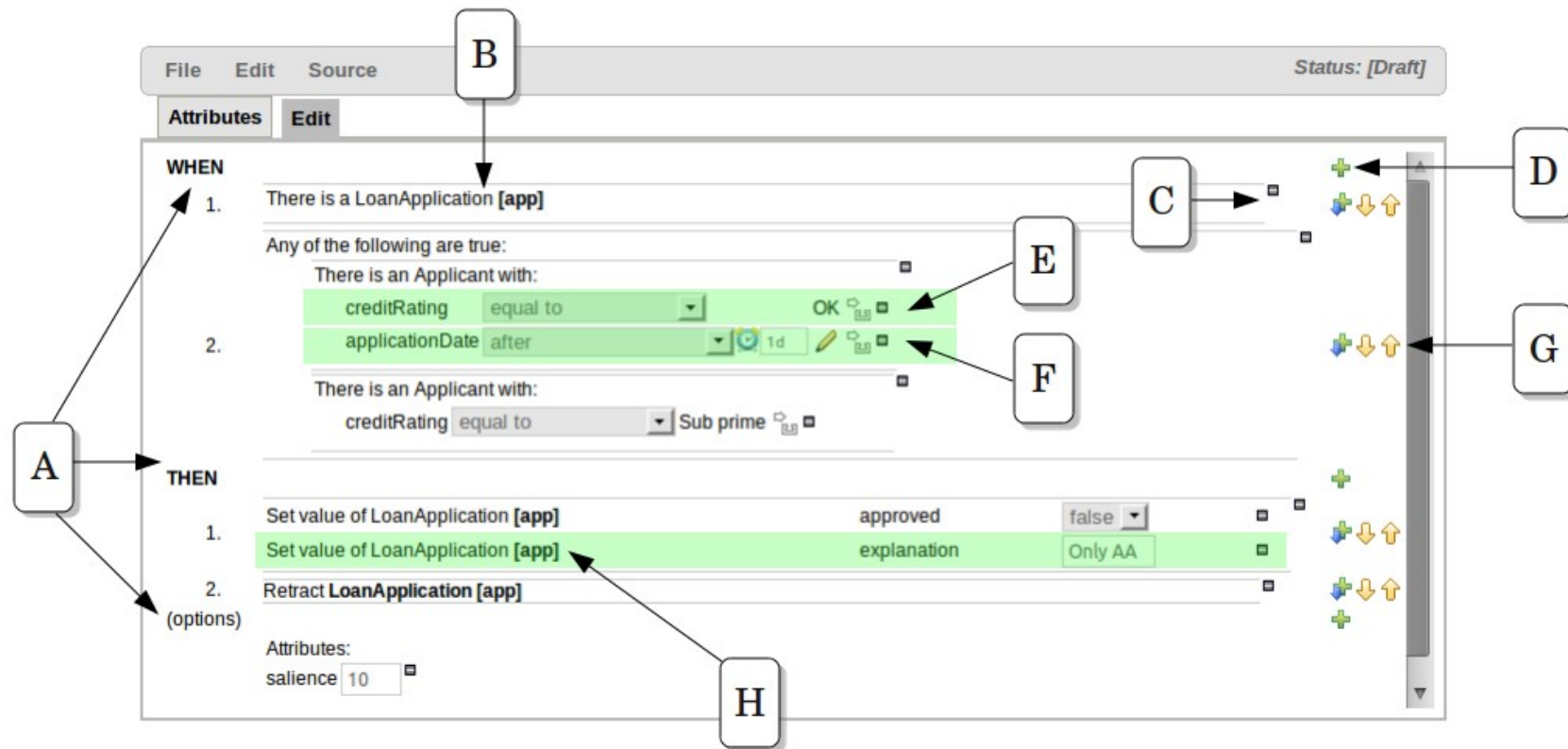
***New Rule → name, category and format***

=> A wizard starts

It is always possible to consult the source in drl



# Guided Business Rules Editor





# Guided Template Editor

Guided Template [t1]

---

**EXTENDS**      None selected ▼

---

**WHEN** +

There is an Applicant with:

	age	less than ▼	\$max_age	⊞ ⊞	
1.	age	greater than or equal to ▼	\$min_age	⊞ ⊞	⊞ + ↓ ↑
	creditRating	equal to ▼	\$scr	⊞ ⊞	

---

**THEN** +


(show options...)





# Keys of templates

Field value



Field value


Literals value: Literal value


Template key: Template key


*Advanced options:*


A formula: New formula


Expression editor: Expression editor




































258



# Template data

Guided Template [t1]

Add row...

	\$max_age	\$min_age	\$scr
			
 	 25	 20	AA
 			OK
 			Sub prime
 	 35	 25	AA
 			OK
 			Sub prime
 	 45	 35	AA
 			OK
 			Sub prime



# Decision Table Editor

---






Drools-WB offers an editor for decision tables.

The editor proposes facts and fields available in the context of the project

2 types of tables can be created:

- **Extended Entries**: Column definitions do not specify a value. The values are then indicated in the body of the table. However, they can be restricted by an interval.
- **Limited Entries**: The column definitions specify a value. The body of the table contains checkboxes

# Extended table

+ Decision table					
	#	Description	Age	Make	Premium
			Applicant [\$a]	Vehicle [\$v]	
			age [<]	make [==]	
 	1		35	BMW	1000
 	2		35	Audi	1000

# Limited table



# Test scenario

---

The test scenarios validate the operation of the rules and avoid regression bugs.

A test case defines several sections:

- **Given** list the test facts
- **Expected** lists the expected changes and actions

# Test scenario

Test Scenario [ Good credit history only ]

SaveDeleteRenameCopyx▼

Run scenario

+ GIVEN

Insert 'LoanApplication'[app]  
approved: false

Insert 'IncomeSource'[incomeSource]  
Add a field

Insert 'Applicant'[a]  
creditRating: OK

+ CALL METHOD

Add input data and expectations here.

+ EXPECT

Use real date and time

LoanApplication 'app' has values:

approved: equalsfalse

More...

(configuration)  
All rules may fire

+ (globals)

Test ScenarioConfigMetadataAll Test Scenarios

Reporting

Success

Text



# DSL editor

---

File Edit Source

Attributes Edit

```
[when]When the credit rating is {rating:ENUM:Applicant.creditRating} = applicant:Applicant(creditRating=="{rating}")  
[when]When the applicant dates is after {dos:DATE:default} = applicant:Applicant(applicationDate>"{dos}")  
[when]When the applicant approval is {bool:BOOLEAN:checked} = applicant:Applicant(approved=={bool})  
[when]When the ages is less than {num:1?[0-9]?[0-9]} = applicant:Applicant(age<{num})  
[then]Approve the loan = applicant.setApproved(true);  
[then]Set applicant name to {name} = applicant.setName("{name}");
```





# List of values

---

**Enum Editor [credit ratings]**

SaveDeleteRenameCopyValidate✕▼

Add enum

	Fact	Field	Context
<input type="checkbox"/>	Applicant	creditRating	['AA', 'OK', 'Sub prime']
<input type="checkbox"/>	Person	age	['20','25','30','35']



# Access to the Workbench database

---

```
public class MainKieTest {  
    public static void main(String[] args) {  
        // works even without -SNAPSHOT versions  
        String url = "http://localhost:8080/kie-drools/maven2/de/test/Test/1.2.3/Test-1.2.3.jar";  
  
        // make sure you use "LATEST" here!  
        ReleaseIdImpl releaseId = new ReleaseIdImpl("de.test", "Test", "LATEST");  
  
        KieServices ks = KieServices.Factory.get();  
  
        ks.getResources().newUrlResource(url);  
  
        KieContainer kieContainer = ks.newKieContainer(releaseId);  
  
        // check every 5 seconds if there is a new version at the URL  
        KieScanner kieScanner = ks.newKieScanner(kieContainer);  
        kieScanner.start(5000L);  
        // alternatively:  
        // kieScanner.scanNow();  
  
        Scanner scanner = new Scanner(System.in);  
        while (true) {  
            runRule(kieContainer);  
            System.out.println("Press enter in order to run the test again....");  
            scanner.nextLine();  
        }  
  
        private static void runRule(KieContainer kieKontainer) {  
            StatelessKieSession kSession = kieKontainer.newStatelessKieSession("testSession");  
            kSession.setGlobal("out", System.out);  
            kSession.execute("testRuleAgain");  
        }  
    }  
}
```

# Web Editor

The screenshot displays the KIE Workbench web editor interface. The top navigation bar includes links for Home, Authoring, Deploy, Process Management, Tasks, and Dashboards. The main workspace is titled 'Business Process [evaluation.bpmn2]' and shows a BPMN diagram for 'Evaluation v.1 (evaluation)'. The diagram starts with a green start node, followed by a 'Self Evaluation' task, then a split gateway (yellow diamond with a plus sign). This gateway branches into two parallel tasks: 'HR Evaluation' and 'PM Evaluation'. Both tasks converge at a join gateway (yellow diamond with a plus sign), which leads to a red end node.

On the left, the 'Project Explorer' panel shows a tree structure with 'Business' and 'Technical' tabs. Under 'Business', there are sub-items for 'demo', 'jpm-playground', 'evaluation', and 'idefault'. The 'evaluation' item is selected.

On the right, the 'Properties (BPMN Diagram)' panel displays various properties for the selected diagram:

Name	Value
Core Properties	
AdHoc	false
Executable	true
Globals	
ID	evaluation
Imports	
Package	Evaluation and main resources
Process Name	Evaluation
Variable Def...	employee.java.lang.String;reason.java.lang.String;perf...
Version	1
Extra Properties	
Documentati...	
Target Name...	http://www.org.org/bpmn20
TypeLanguage	http://www.java.com/javaTypes
Simulation Properties	
Base Currency	
Base time unit	seconds

At the bottom, the 'Problems' panel is visible, showing a table with columns for Level, Text, File, Column, and Line.

# Data modelling

Form Modeler [PerformanceEvaluation-taskform]

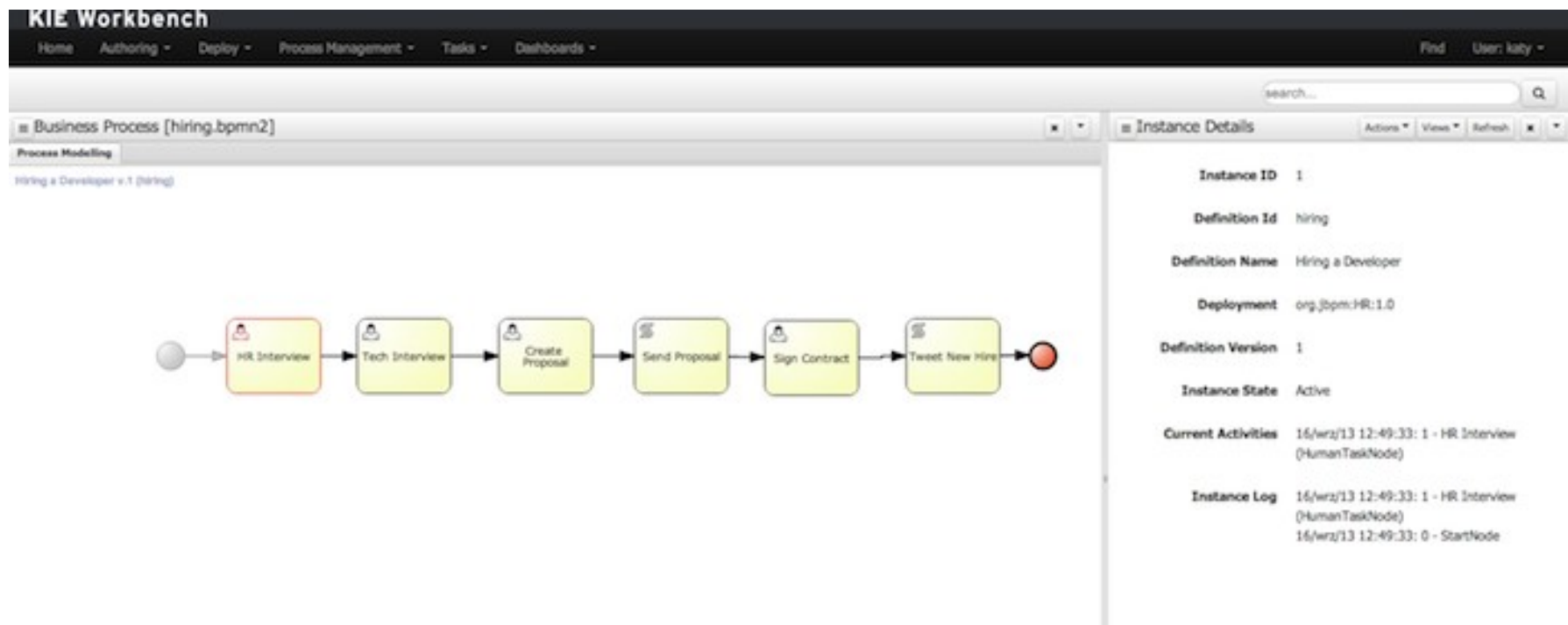
Save Delete

Form data origin Add fields by origin Add fields by type Form properties Show mode Bindings Grid & Ruler

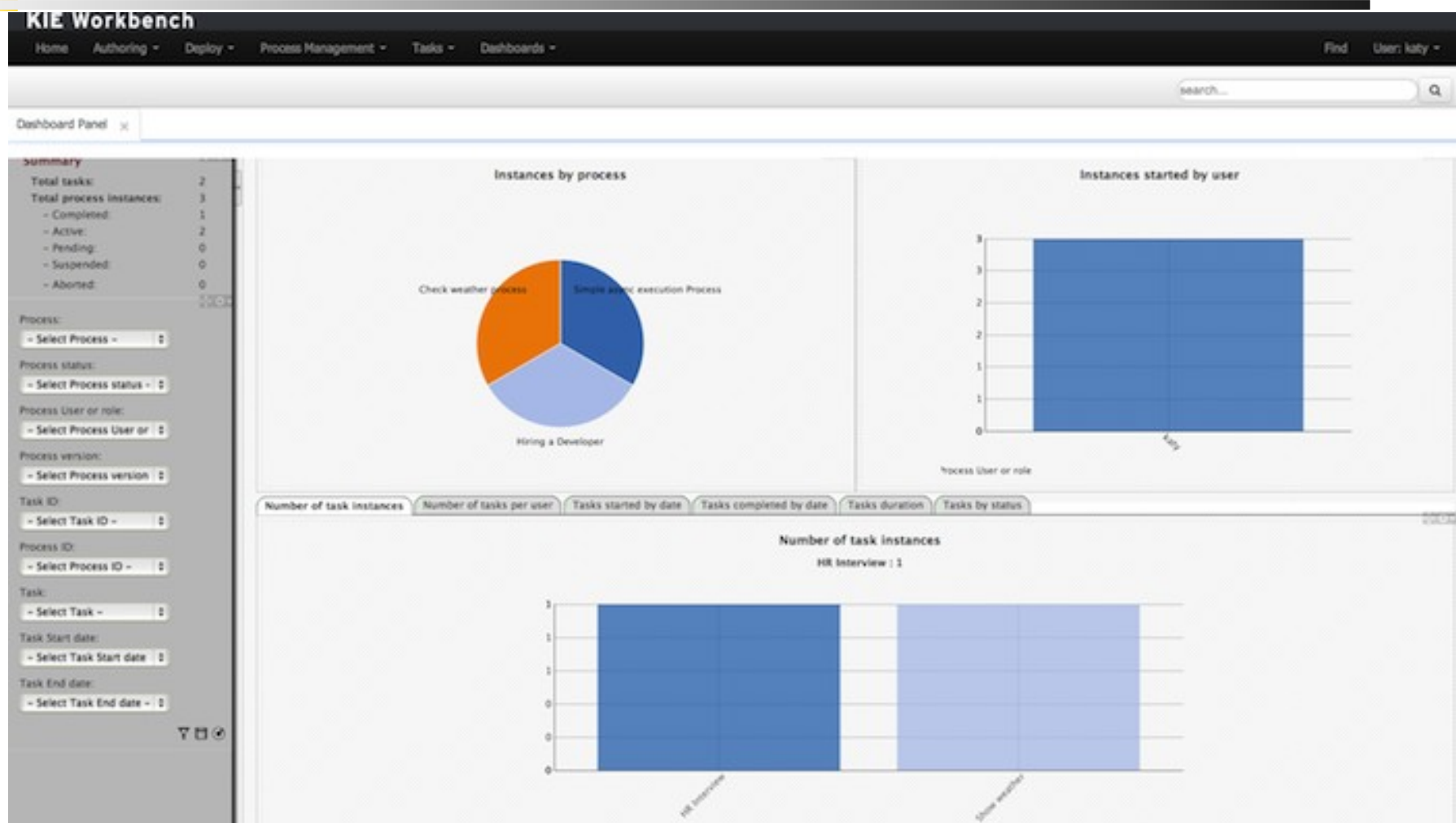
HTML label  
Separator  
Simple subform  
Multiple subform  
Short text  
Long text  
Float  
Decimal  
BigDecimal  
BigInteger  
Short  
Integer  
Long integer  
E-mail  
CheckBox  
Rich text  
Timestamp

Reason  
Performance

# Process Instances Management



# BAM Reporting (Birt)





# Thank U!!!

---

❖ THANK YOU FOR YOUR ATTENTION



# Annexes

---

## **Algorithms** DSL





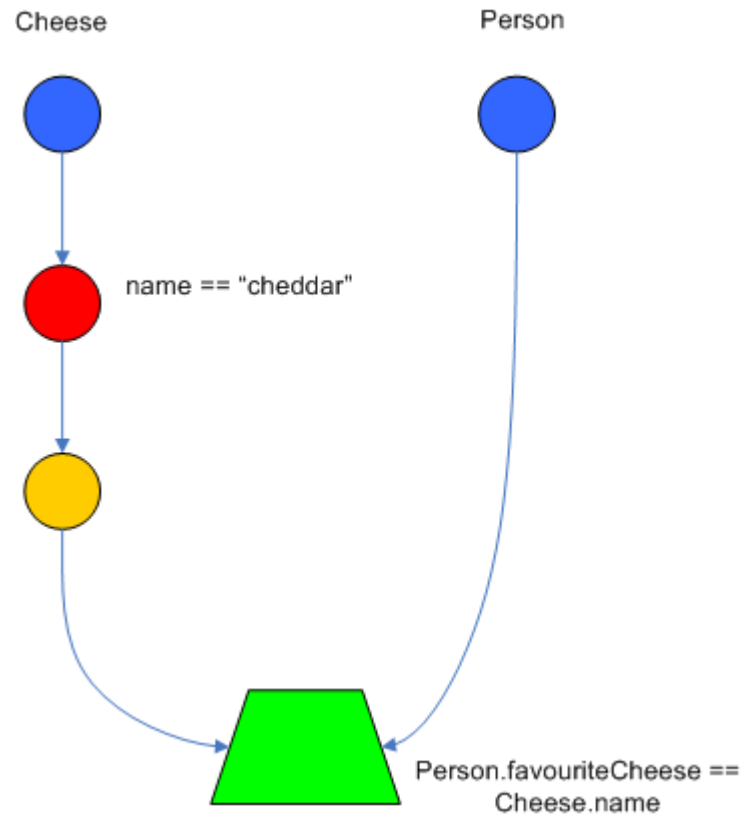
# Rete's principles

---

The algorithm is performed in 2 steps

- The compilation of rules creates a discriminant network composed of different nodes :
  - **TypeNode** : Relative to fact Type
  - **Alpha** : Constraint on a fact
  - **Beta** : Comparaison between a tuple of objects .
  - **Rule** : Activation of a specific rule
- The pattern matching. Executed whenever the working memory changes. Facts are introduced into the network and crosses the nodes if the conditions are satisfied. Some nodes (Beta) have an associated memory which stores the facts that have reached the node.

# Example





# Forward or backward chaining

---

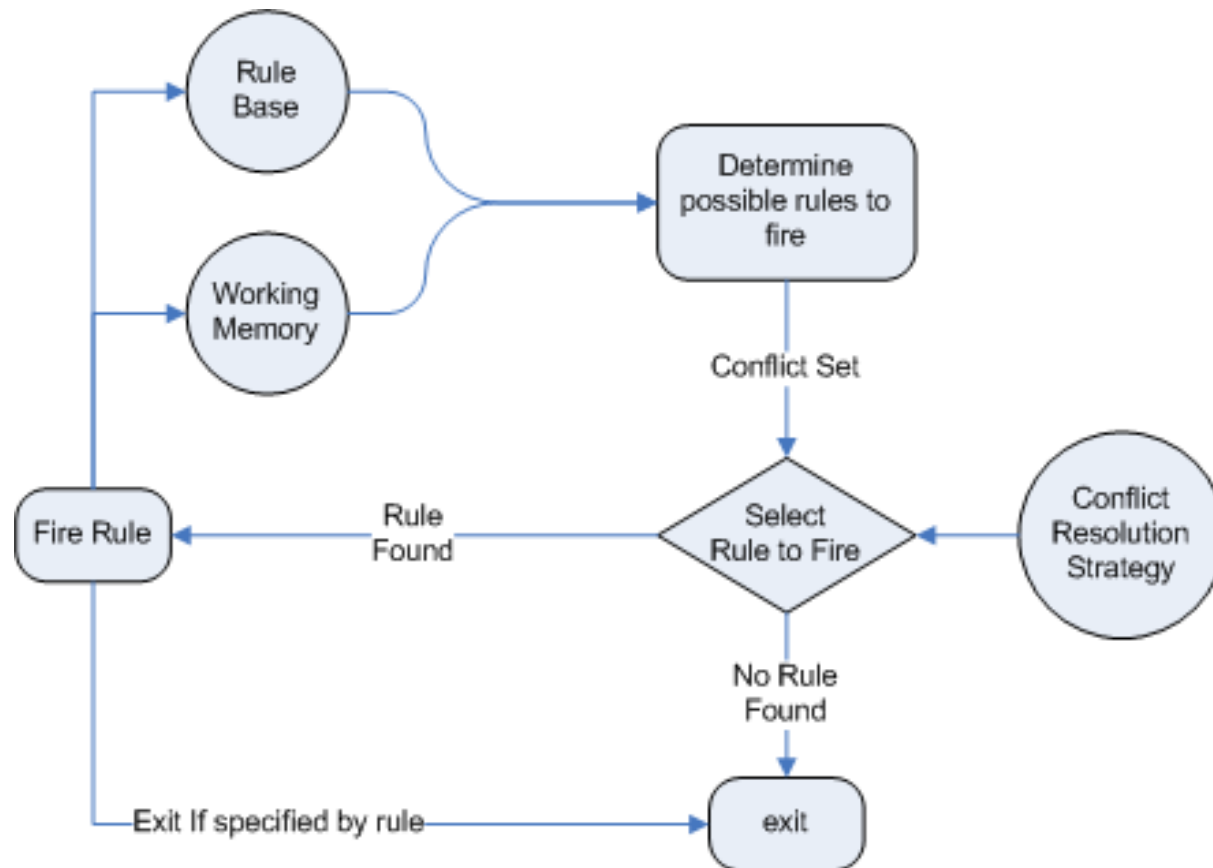
We can distinguish 2 kinds of rule's engine

- **Forward chaining** . The engine is driven by data: from a fact, rules apply, propagate and end with a conclusion.
- **Backward chaining** . The engine start from the rules to go back to the rules

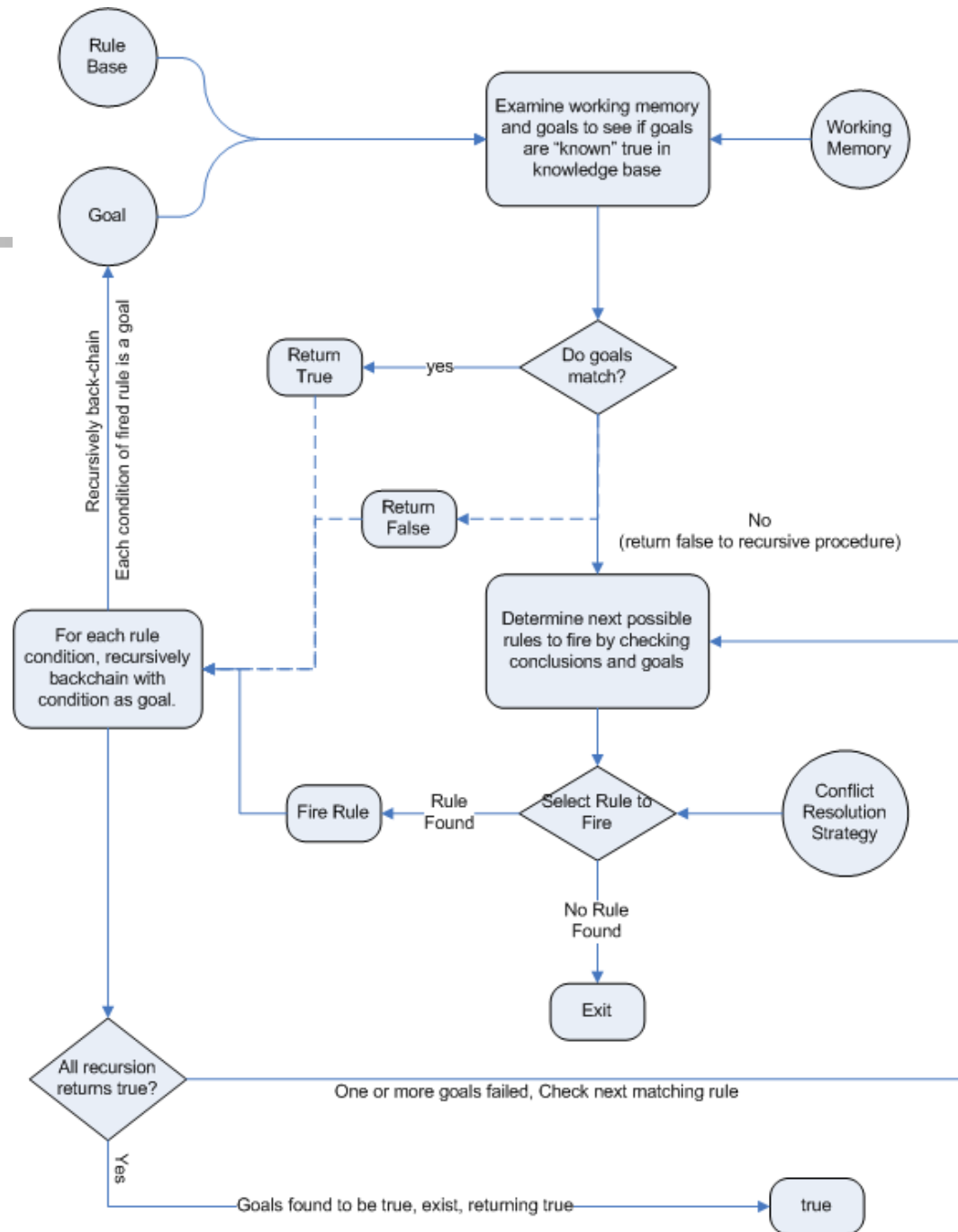
Drools 5 was a forward chaining .

Drools 6 is hybrid but default to backward chaining

# Forward chaining



# Backward chaining





# ReteOO optimizations

---

*ReteOO* is the algorithm of versions 3, 4 and 5 of Drools.

The job of pattern matching is done every time the facts in memory changes (*insert, update, delete*)

=> This can cause a lot of unnecessary work (inserting fact not causing, executing rules)



# PHREAK algorithm

---

Drools6 introduces the new PHREAK algorithm that improves ReteOO

- PHREAK is equivalent to ReteOO when the number of rules remains moderate but avoids performance losses when the number of rules grows
- PHREAK also allows performance gains when using calendar groups and priority attributes (salience) rules



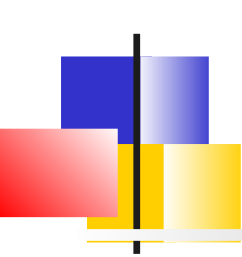
# Phreak's principles

---

The main differences with ReteOO are:

- Phreak is lazy, the pattern matching work is only done when an explicit rule is triggered (*fireAllRules()* method)
- It is chained backward, each eligible rule is evaluated independently of the others





# Other ways to express rules

---

## **DSL**

Decision tables  
Rule's templates



# Introduction

---

***Domain Specific Languages (DSL)*** allow to extend the rules language by adapting it to the business language.

This is an abstraction layer dedicated to non-technical business experts that is translated into the rule language at compile time

They can also be used as templates of conditions or actions, allowing to share certain parts of rules



# Syntax

---

The format of a DSL is simply a text file that translates "natural" language keys into drl expressions

Each line starts with a scope and then the extended language translation in the rule language

```
[when]This is
    "{something}"=Something(something=="{something}")
[then]Log
    "{message}"=System.out.println("{message}") ;
```



# Syntax

---

It is also possible to use the scope ***[keyword]*** which makes it possible to redefine a keyword:  
[keyword] quand = when

The defined sentences are actually regular expressions. Wildcards can therefore be used.



# Examples

---

```
[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}");
```

```
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");
```



# Example with regexp

---

```
[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}
```

-----

```
There is a Cheese with  
- age is less than 42  
- rating is greater than 50  
- type equals 'stilton'
```

-----

```
Cheese(age<42, type=='stilton', rating>50)
```



# Steps

---

1. Name your DRL file with the *.dslr* extension

2. Refer to the DSL file in the rule file with the keyword expander

expander your-expander.dsl

3. Put resources in the *KieContainer's* classpath: