



# **JBoss Enterprise SOA Platform 5**

## **JBoss Rules 5 リファレンスガイド**

JBoss のプログラマーおよび JBoss Rules の開発者を対象  
エディション 5.2.0



# JBoss Enterprise SOA Platform 5 JBoss Rules 5 リファレンスガイド

---

JBoss のプログラマーおよび JBoss Rules の開発者を対象  
エディション 5.2.0

Red Hat Documentation Team

## 法律上の通知

Copyright © 2011 Red Hat, Inc..

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## 概要

所属企業のビジネスルールを開発する際に、本書を参考ガイドとしてご利用ください。

---

## 目次

はじめに .....	2
第1章 はじめに .....	3
第2章 クイックスタート .....	8
第3章 ユーザーガイド .....	27
第4章 ルール言語 .....	76
第5章 スプレッドシートのデシジョンテーブルの使用 .....	143
第6章 JAVA ルールエンジンアプリケーションプログラミングインターフェース .....	157
第7章 JBOSS DEVELOPER STUDIO .....	162
第8章 例 .....	181
付録A © 2011 .....	260
付録B 改訂履歴 .....	261

## はじめに

### 1. 謝辞

本書の一部は、<http://www.jboss.org/drools> より入手可能な Mark Proctor、Michael Neale、Edson Tirelli 著の『Drools Expert』（著作 © 2010 JBoss Inc）に初めて公開されました。

『JBoss Enterprise BRMS Platform JBoss Rules 5 Reference Guide』は Darrin Mison (Red Hat) および David Le Sage (Red Hat) によって編集されました。

# 第1章 はじめに

## 1.1. ルールエンジンとは

### 1.1.1. 説明と背景

**JBoss Rules** は、チューリングが完全な *Rete* アルゴリズム を使用して プロダクションルール を作成および解釈する高度な人工知能システムです。本書では、このシステムを使用して時間と共に進化するビジネスルールとプロシージャを記述および変更する方法について説明します。ルールが記述されたら、ソフトウェアを使用してルールを管理、デプロイ、および分析します。

最初の項を読んで、ソフトウェアの仕組みを理解しましょう。この概要では、基本的な用語や理論を紹介し、システムの主な機能について説明します。

ルールシステムの「頭脳」は大量のプロダクションルールとファクトをスケールできる *推論エンジン* です。

#### 推論エンジン

推論エンジンはファクトとデータをルールに対して一致し、アクションになる結果を推論します。

#### プロダクションルール

プロダクションルールはナレッジを表すために 1 階論理を使用する 2 つ部分で構成される構造です。

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

#### パターンマッチング

パターンマッチングは、ファクトをルールに対して一致するプロセスです。Linear、Rete、Treat および Leaps アルゴリズムを使用して推論エンジンによって実行されます。

#### ReteOO

使用される Rete 実装は **ReteOO** と呼ばれます。これは、オブジェクト指向システム向けに向上され最適化された Rete アルゴリズムの実装です。

#### 競合解決ストラテジー

システムに大量のルールがある場合、ファクトのアサーションによっては複数のルールが true になることがあります。このような場合、これらのルールは競合状態にあると言えます。**agenda** は、競合解決ストラテジーを使用してルールが実行される順番を指示し、このような状態を管理します。

ルールはプロダクションメモリーに格納され、ファクトはワーキングメモリーへ アサート されます。ファクトがワーキングメモリーに格納されると、ファクトの変更または取り消しが可能になります。

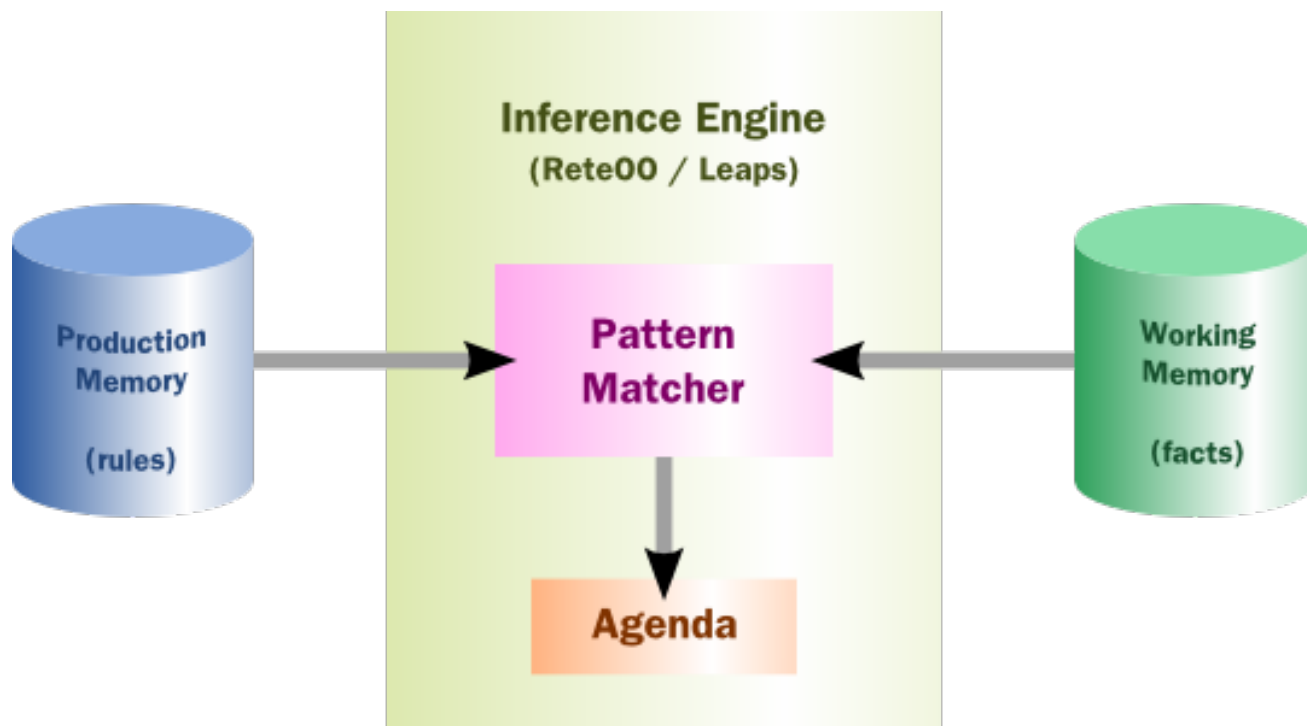


図1.1 Rules エンジンの概要

プロダクションルールシステムの推論エンジンは **ステートフル** で、**真理維持** を行います。

#### 真理維持

真理を強制する推論エンジンの機能。

アクションを使用して **論理関係** を宣言します。

#### 論理関係

論理関係は、アクションの状態が **true** である推論に依存する場合に存在します。推論が **true** でなくなると、それに依存するアクションは元に戻されます。

プロダクションルールシステムには、**前向き連鎖**、**後向き連鎖**、この2つを組み合わせた **ハイブリッド** の3種類があります。

#### 前向き連鎖

前向き連鎖はデータ駆動型で、提供されたデータに反応します。ファクトはワーキングメモリーに挿入され、1つまたは複数のルールが **true** になります。そして、アジェンダによって実行されるようスケジュールに置かれます。



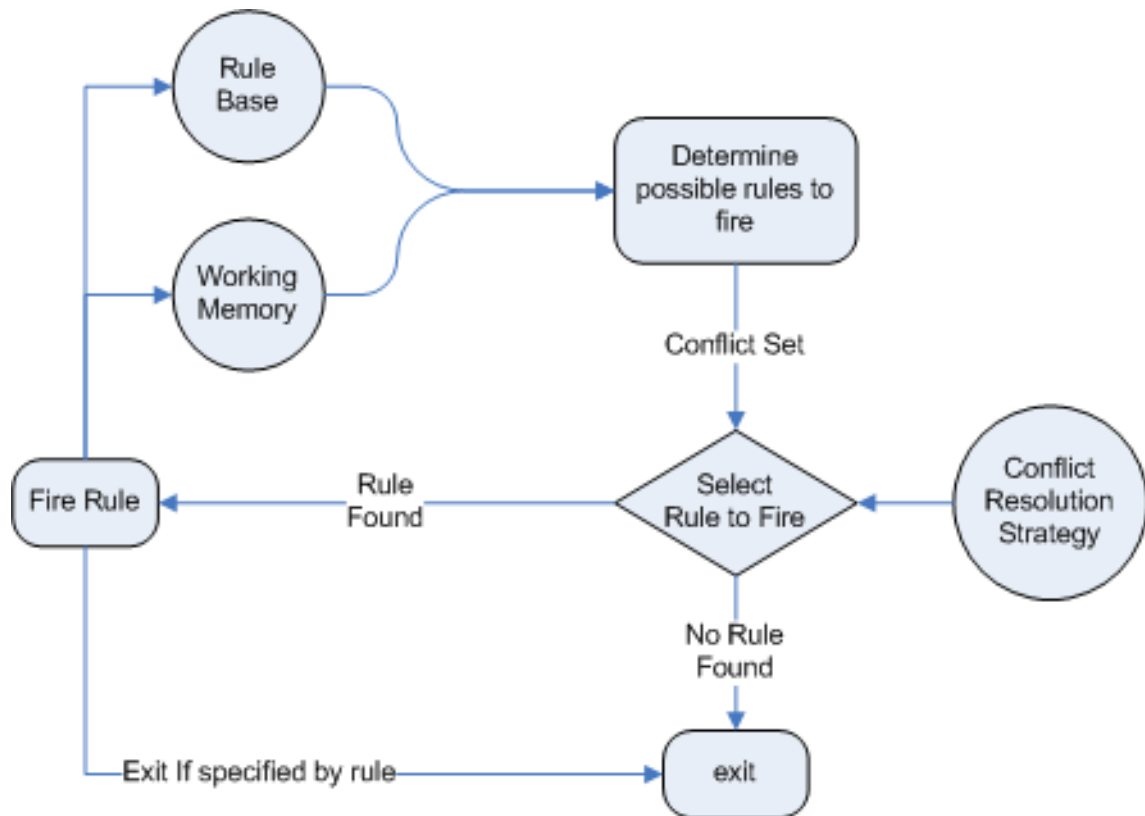


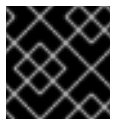
図1.2 前向き連鎖

**重要**

**JBoss Rules** は前向き連鎖のエンジンです。

**後向き連鎖**

後向き連鎖はゴール駆動型で、システムはエンジンが満たそうとする **結論** より開始します。結論を満たすことができない場合、現在のゴールの一部を満たせるようにする **サブゴール** と呼ばれる結論を探します。最初の結論が満たされるか、満たされないサブゴールがなくなるまでこの処理が継続されます。**Prolog** は後向き連鎖エンジンの一例になります。

**重要**

**JBoss Rules** の次回のメジャーリリースに、後向き連鎖のサポートが追加されます。

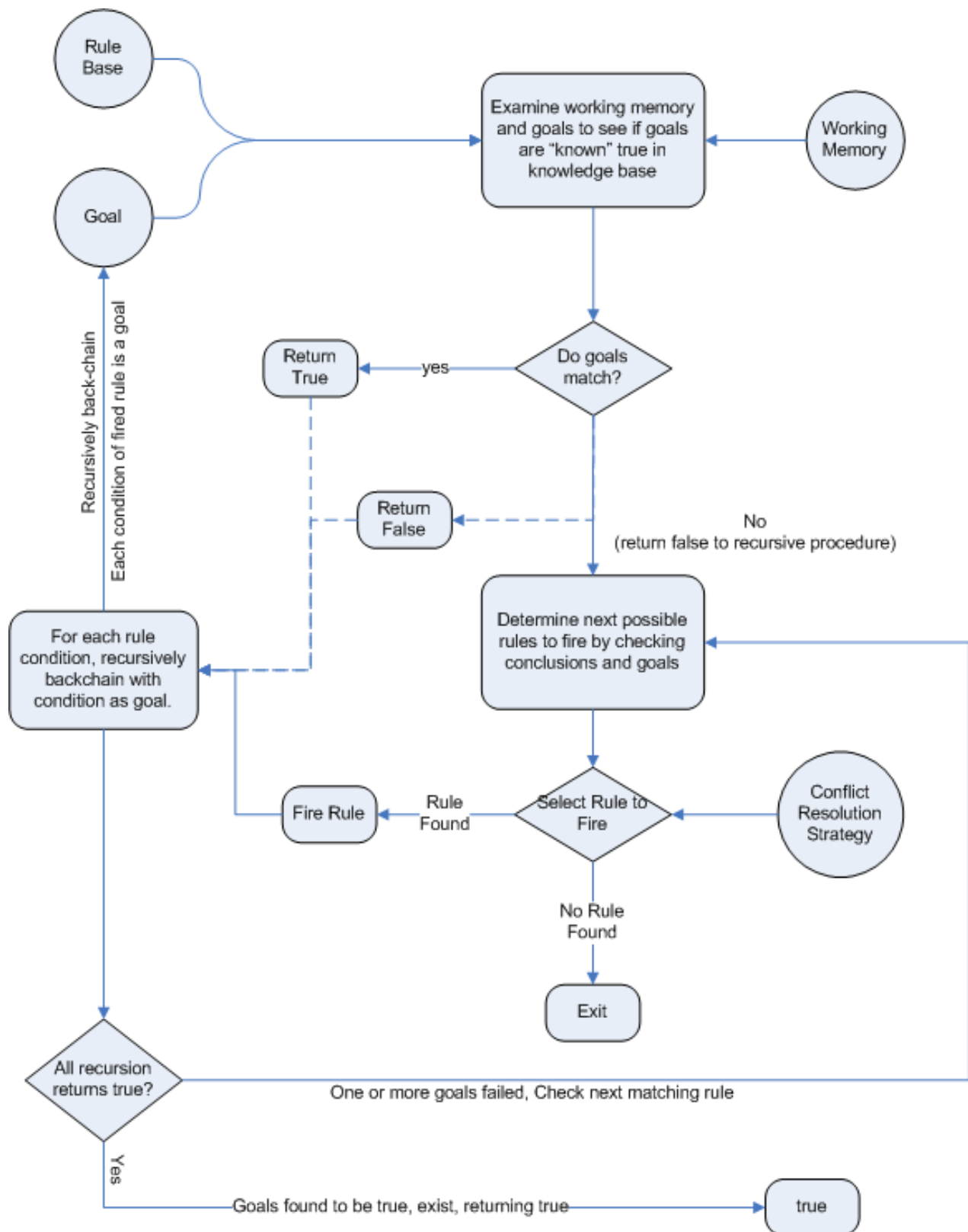


図1.3 後向き連鎖

**重要**

システムの違いや、システムの最適化に最も適した方法を理解するには、これらの動作モードを理解することが重要となります。

## 1.2. 強結合と疎結合

柔軟性が非常に高いため、一般的には **疎結合** を示す設計が好まれます。ルールがすべて強結合されている場合、ルールの柔軟性がなくなる可能性が高くなります。さらに重要な点として、このような状況でルールエンジンをデプロイするのは過剰であると言えます。

### 疎結合

あるルールの実行によって他のルールが実行されることがない設計。

### 強結合

ルールが強結合である場合、あるルールが実行されると直接的に他のルールが実行されます。言い換えると、論理の明確なチェーンが存在することになります。

明確なチェーンは **decision tree** を使用してハードコードまたは実装することができます。



### 注記

強結合は本質的に悪いわけではありませんが、強結合の難点を念頭に置いてルールがキャプチャーされる方法を設計するようにしてください。

疎結合のシステムは柔軟性がより高く、影響を出さずにルールを追加、変更および削除できます。

## 第2章 クイックスタート

### 2.1. クイックスタートの基本

JBoss Rules は、多くのユースケースに対応するようソフトウェアが設計されているため、非常に多くの機能があります。そのため、新規ユーザーは JBoss Rules に圧倒されてしまうことがあります。本章の目的はこれらの機能を少しずつ説明することです。理解しやすくするため、非常に簡単な例をいくつか使用します。

#### 2.1.1. ステートレスナレッジセッション

最も簡単なユースケースは ステートレスセッション です。

##### ステートレスセッション

推論のないセッション。

ステートレスセッションでは、データを渡した後にその結果を受け取るため、関数のようと言えます。ステートレスセッションの一般的なユースケースは多くありますが、その一部は次の通りです。

- 検証。例：「この申請者は住宅ローンの資格があるか」
- 計算。例：「住宅ローンの保険料を計算する」
- ルーティングとフィルタリング。例：「受信メッセージをフォルダーへフィルターする」または「受信メッセージを宛先へ送信する」

運転免許証の申請に関する簡単な例は次の通りです。

1. 必要なデータを収集します。これはルールに渡される **ファクト** のセットを形成します。この例では、1 つのデータのみが存在します。

```
package com.company.license;

public class Applicant
{
    private String name;
    private int age;
    private boolean valid;

    public Applicant (String name, int age, boolean valid)
    {
        this.name = name;
        this.age = age;
        this.valid = valid;
    }

    //add getters & setters here
}
```

2. この時点でデータモデルが存在するため、最初のルールを記述します。このルールの目的は、18 歳未満の申請者を拒否することです。

```
package com.company.license;

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

**Applicant** オブジェクトがルールエンジンに挿入されると、各ルールの制約が評価し、一致するものを探します (常に「オブジェクトタイプ」の暗黙的な制約があり、それ以降は明示的なフィールド制約がいくつでも存在できます)。

### パターン

制約のコレクションはパターンと呼ばれます。

### パターンマッチング

各ルールの制約グループがオブジェクトを評価し、一致するものを探すプロセスです。

### 一致

挿入されたオブジェクトがルールのすべての制約を満たす場合、一致したと見なされます。

たとえば、**Is of valid age** ルールでは次の 2 つの制約があります。

1. 一致するファクトのタイプは **Applicant** でなければならない。
2. **Age** の値は 18 未満でなければならない。

**\$a** はバインディング変数で、ルールの結果 (ここからオブジェクトのプロパティを更新可能) の一致したオブジェクトの参照を可能にします。



### 注記

ドル記号 (\$) の使用は任意です。使用すると変数名とフィールド名を区別できます。



### 注記

ここでは、ルールがクラスと同じフォルダーにあり、最初の ナレッジベース を構築するために クラスパスリソースローダー を使用できると仮定します。

### ナレッジベース

ナレッジベースは **KnowledgeBuilder** によってコンパイルされたルールのコレクションです。

3. 

```
KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource(
    "licenseApplication.drl", getClass() ), ResourceType.DRL );
```

```
if ( kbuilder.hasErrors() ) {
    System.err.println( kbuilder.getErrors().toString() );
}
```

上記で引用符で囲まれたコードは **newClassPathResource()** メソッドを使用して **licenseApplication.drl** ファイルのクラスパスを検索します。**ResourceType** は *Drools* ルール言語で記述されます。

### Drools ルール言語

Drools ルール言語 (DRL) は JBoss Rules のネイティブルール言語です。

4. エラーは **KnowledgeBuilder** をチェックします。エラーがない場合、セッションを構築できます。
5. ルールに対してデータを実行します (申請者は 18 歳未満であるため、申請は「無効」とマーク付けされます)。

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
StatelessKnowledgeSession ksession =
kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant( "Mr John Smith", 16, true );

assertTrue( applicant.isValid() );
ksession.execute( applicant );
assertFalse( applicant.isValid() );
```

ここまで、データは単一のオブジェクトで構成されていました。複数のオブジェクトを使用したい場合はどうするのでしょうか。この場合、**collection** など、オブジェクトを実装する **iterable** に対して実行することが可能です。次の例では、運転免許申請日が含まれる **Application** という名前の別のクラスを追加する方法を説明します。この他に、**valid** というブール値フィールドを **Application** に移動する方法についても説明します。

1. コードは次の通りです。

```
public class Applicant {
    private String name;
    private int age;

    public Applicant (String name, int age)
    {
        this.name = name;
        this.age = age;
    }
    // getter and setter methods here
}

public class Application {
    private Date dateApplied;
    private boolean valid;

    public Application (boolean valid)
    {
        this.valid = valid;
    }
}
```

```

    }
    // getter and setter methods here
}

```

2. 申請が正規の期間内に行われたことを確認するために、以下のルールを追加します。

```

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-2009" )
then
    $a.setValid( false );
end

```

3. Java アレイは **iterable** インターフェースを実装できないため、*JDK コンバーター* メソッドを代わりに使用します (このメソッドは **Arrays.asList(...)** という行で始まります)。

以下のコードは **iterable** リストに対して実行されます。各コレクション要素は、一致したルールが実行される前に挿入されます。

```

StatelessKnowledgeSession ksession =
    kbase.newStatelessKnowledgeSession();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
Applicant applicant = new Applicant( "Mr John Smith", 16 );
Application application = new Application(true);
assertTrue( application.isValid() );
ksession.execute( Arrays.asList( new Object[] {application,
    applicant} ));
assertFalse( application.isValid() );

```



#### 注記

**execute(Object object)** および **execute(Iterable objects)** メソッドは、実際は **BatchExecutor** インターフェースからの **execute(Command command)** というメソッドに対するラッパーです。

4. **CommandFactory** を使用して命令を作成し、以下の内容が **execute( Iterable it )** と同等になるようにします。

```

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

```

5. 多くのコマンドや結果出力識別子を使用する場合は、特に **BatchExecutor** と **CommandFactory** が便利です。

```
List<Command> cmds = new ArrayList<Command>();
cmds.add(
    CommandFactory.newInsertObject(new Person("Mr John Smith"),
    "mrSmith"));
cmds.add(
    CommandFactory.newInsertObject(new Person( "Mr John Doe" ),
    "mrDoe" ));

ExecutionResults results =
    ksession.execute( CommandFactory.newBatchExecution(cmds) );

assertEquals( new Person("Mr John Smith"),
    results.getValue("mrSmith" ) );
```



#### 注記

**CommandFactory** は **BatchExecutor** で使用できる他のコマンドを多くサポートします。 **StartProcess**、**Query**、**SetGlobal** などがその一例です。

### 2.1.2. ステートフルナレッジセッション

ステートフルセッションの一般的なユースケースの一部は次の通りです。

#### ステートフルセッション

ステートフルセッションでは、経時的にファクトに繰り返し変更を行うことが可能です。

- 監視: 例として、株式市場を監視し、購入処理を自動化できます。
- 診断: 例として、これを使用して障害検出処理を実行できます。また、医療診断処理にも使用できます。
- 物流: 例として、配達物追跡および配送提供に関連する問題に適用できます。
- 法令遵守: 例として、市場売買の合法性を検証するために使用できます。



#### 警告

メモリーリークの発生を防ぐため、**dispose()** メソッドは、必ずステートフルセッションの実行後に呼び出されるようにしてください。これは、ナレッジベースは作成時にステートフルナレッジセッションへの参照を取得するためです。

**StatelessKnowledgeSession** の場合と同様に、**StatefulKnowledgeSession** は **BatchExecutor** インターフェースをサポートします。唯一の違いは、**FireAllRules** コマンドが最後に自動的に呼び出されないことです。



以下の火災報知器システムの開発に関する例は、「監視」のユースケースを表しています。

1. 各部屋にスプリンクラーが設定されている家のモデルを作成します。いずれかの部屋で火事が発生します。

```
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
{
}
```

2. ルールは複数のオブジェクト間の関係を表す必要があります (特定の部屋にスプリンクラーが存在することなどを定義するため)。

これには、パターンでバインディング変数を制約として使用します。これは クロス積 になります。

3. **Fire** クラスのインスタンスを作成します。このインスタンスをセッションに挿入します。

以下のルールは、**Fire** オブジェクトの room フィールドへのバインディングを制約の一致へ追加します。これにより、その部屋のスプリンクラーのみがチェックされます。このルールが実行され、結果が実行されると、スプリンクラーが有効になります。

```
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
```

ステートレスセッションは標準的な Java 構文を使用してフィールドを変更しますが、上記のルールは **modify** ステートメントを使用します (「with」ステートメントのように動作します)。

これには、一連のコンマ区切りの Java 式が含まれています。これらはあらゆる意味

で、**modify** ステートメントの *制御式* によって選択されたオブジェクト "**setters**" への呼び出しです。これらの **setters** はデータを変更し、変更内容の「理由付け」を再度行えるよう、**engine** がその変更を認識するようにします。このプロセスが *推論* と呼ばれ、ステートフルセッションがどのように操作するかを理解するために重要になります (ステートレスセッションは推論を使用しないため、**engine** がデータの変更を認識する必要はありません)。



### 注記

推論を無効にするには、シーケンシャルモードを使用します。

ここまで、一致が存在する時に操作するルールを見てきましたが、一致が存在しない場合はどうするのでしょうか。火災が鎮火したことをどのように決定するのでしょうか。これまでの制約はエンジンが個別のインスタンスを制約する、*命題論理* に基づいた「文」でした。しかし、**JBoss Rules** はデータのセットを閲覧できる *1 階論理* もサポートします。キーワードが **not** のパターンは何かが存在しない場合のみ一致します。

次のルールは火事が鎮火した時にスプリンクラーを停止します。

```
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
```

各部屋に 1 つのスプリンクラーがありますが、建物全体には火災報知器が 1 つしかありません。火災がある時には **Alarm** オブジェクトが作成されますが、火元がいくつあっても建物全体で 1 つの **Alarm** のみが必要になります。ここで、**not** の補語である **exists** を導入できます。**exists** はカテゴリの 1 つまたは複数のインスタンスに一致します。

```
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
```

火災が鎮火したら火災報知器を止める必要があります。火災報知器を停止するには再度 **not** を使用します。

```
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
```

最後に、次のコードはアプリケーションが最初に起動した時と火災報知器とすべてのスプリンクラーが停止した時に、全体的な状態のメッセージを出力します。

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

ルールを **fireAlarm.drl** という名前のファイルに格納します。このファイルはクラスパスのサブディレクトリに保存します。次に、**fireAlarm.drl** という新しい名前で **ナレッジベース** を構築します。

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "fireAlarm.drl",
    getClass() ), ResourceType.DRL );

if ( kbuilder.hasErrors() )
    System.err.println( kbuilder.getErrors().toString() );

StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

次の例では、4つの部屋オブジェクトが作成され、1つのスプリンクラーオブジェクトと共に各部屋に対し挿入されます（一致するプロセスは終了しますが、ルールはまだ実行されません）。

1. **ksession.fireAllRules()** を呼び出します。これにより、一致するルールに実行権限が与えられますが、実行されないため健全性に関するメッセージのみが生成されます。

```
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 Okay
```

2. 次に、2つのファイルを作成し、挿入します（ファクトハンドルは保持されます）。

### ファクトハンドル

ファクトハンドルは挿入されたインスタンスへの内部参照です。これにより、後でインスタンスを取り消したり変更したりすることができます。

3. **fires** がエンジンにあるため、**fireAllRules()** を呼び出します。火災報知器が作動し、該当するスプリンクラーがオンになります。

```
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
```

4. 鎮火したら、火災オブジェクトが取り消され、スプリンクラーがオフになります。この時点で火災報知器はキャンセルされ、状態メッセージが再度表示されます。

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

ksession.fireAllRules();
```

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

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

これらの例は、ルールシステムの一部機能を実演する簡単な例であり、プログラミングの仕方が大まかに理解できたと思います。

## 2.2. 理論の概要

### 2.2.1. メソッドとルール

新しいユーザーはメソッドとルールを混同することがよくあります。メソッドの概要は次の通りです。

- 直接呼び出される
- 特定のインスタンスが渡される
- 1つの呼び出しによって1つの実行が行われる

```
public void helloWorld(Person person)
{
    if ( person.getName().equals( "Chuck" ) )
    {
```

```

        System.out.println( "Hello Chuck" );
    }
}

```

```

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

```

ルールの概要は次の通りです。

- **engine** に挿入されたデータに一致することにより実行される
- 直接呼び出すことができない
- 特定のインスタンスをルールに渡すことはできない
- 一致によって、ルールは一度または複数回実行され、全く実行されないこともある

## 2.2.2. クロス積

### クロス積

2 つ以上のデータセットが組み合わせられた時、その結果はクロス積と呼ばれます。

火災報知器の例で、以下のルールについて考えてみましょう。

```

rule "show sprinklers in rooms"
when
    $room : Room()
    $sprinkler : Sprinkler()
then
    System.out.println( "room:" + $room.getName() +
        " sprinkler:" + $sprinkler.getRoom().getName() );
end

```

これは、**select \* from Room, Sprinkler** への SQL (Structured Query Language) コマンドと同じで、**Room** テーブル内の各行が **Sprinkler** テーブル内の各行と結合するよう指示します。結果として、以下が出力されます。

```

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

```

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

クロス積は巨大になることがあり、パフォーマンス上問題となることがあります。このような問題を防ぐには、変数制約を使用して意味のない結果を除外します。

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

これにより、4つの行のデータのみになり、正しい **Sprinkler** が各 **Room** に割り当てられます。SQLで書かれた通り、対応するクエリは **select \* from Room, Sprinkler where Room == Sprinkler.room** になります。

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

### 2.2.3. アクティベーション、アジェンダおよび競合セット

これまでの例は、データと一致プロセスが小さく、比較的単純でした。しかし、徐々に多くのファクトやルールを挿入することになります。現時点では、**rule engine** には結果の実行を管理する方法が必要となります。JBoss Rules は **アクティベーション、アジェンダおよび競合解決戦略** を使用してこれを実現します。

次の例はより複雑で、複数の期間にわたるキャッシュフロー計算の処理を実証します。



#### 注記

ここでは、**knowledge bases** の作成に必要な Java コードの知識があることを前提とし、コードが繰り返されないように **StatefulKnowledgeSession** にファクトを投入します。

図は主要な段階での **rule engine** の状態を示します。

データモデルは **Cashflow**、**Account**、および **AccountPeriod** の3つのクラスから構成されます。

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

```

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

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

```

**knowledge bases** の作成方法と、ファクトをインスタンス化して **StatefulKnowledgeSession** に投入する方法は理解されているはずです。したがって、説明を明確にするため、表を使用して挿入されたデータの状態を示します。下表は単一のファクトが **Account** に対して挿入されたことを表しています。2 四半期にわたる入出金も **Cashflow** オブジェクトとして **Account** に挿入されています。

図2.1 「Cash Flow および Account」 は 4 つの **Cashflow** ファクトと共に挿入された単一の **Account** を表しています。

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

Account	
accountNo	balance
1	0

図2.1 Cash Flow および Account

以下の 2 つのルールは、指定された期間の入出金の合計を判断するために使用されます。さらに、口座残高を更新するために使用されます。(&& 演算子を使用するとフィールド名を繰り返し使用する必要がなくなります)。

```

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

```

```

rule "decrease balance for debits"
when

```

```

    ap : AccountPeriod()
    acc : Account( $accountNo : accountNo )
    CashFlow( type == DEBIT,
               accountNo == $accountNo,
               date >= ap.start && <= ap.end,
               $amount : amount )
then
    acc.setBalance(acc.getBalance() - $amount);
end

```

図2.2「Cash Flow および Account」の通り、会計期間開始日は1月1日に設定され、終了日は3月31日に設定されます。これにより、データは入金と出金に対する2つの **Cashflow** オブジェクトに制約されます。

AccountingPeriod		
start	end	
01-Jan-07	31-Mar-07	

CashFlow		
date	amount	type
12-Jan-07	100	CREDIT
9-Mar-07	75	CREDIT

CashFlow		
date	amount	type
2-Feb-07	200	DEBIT

図2.2 Cash Flow および Account

1. データは挿入段階で照合されますが、これはステートフルセッションであるため、ルールの結果は即座に実行されません。一致したルールと対応するデータは **アクティベーション** と呼ばれます。
2. 各アクティベーションは **アジェンダ** と呼ばれるリストへ追加されます。
3. アジェンダ上の各アクティベーションは、**fireAllRules()** メソッドが呼び出された時に実行されます。指定のない限り、アクティベーションは任意の順番で実行されます。

Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	

図2.3 Cash Flow および Account

上記すべてのアクティベーションが実行されると、口座の残高は -25 になります。

Account	
accountNo	balance
1	-25

図2.4 Cash Flow および Account



会計期間が第2四半期に更新された場合、一致したデータ行は1つのみになるため、**アジェンダ**のアクティベーションも1つになります。

AccountingPeriod		
start	end	
01-Apr-07	30-Jun-07	

CashFlow		
date	amount	type
18-May-07	50	CREDIT

図2.5 Cash Flow および Account

アクティベーションが実行されると、残高は 25 になります。

accountNo	balance
1	25

図2.6 Cash Flow および Account

**agenda** に 1 つまたは複数のアクティベーションがある場合、アクティベーションは「競合」状態であると言えます。この場合、実行の順序を決定するため、**競合解決戦略**が使用されます。最も単純なレベルでは、デフォルトの戦略は *salience* を使用してルール of 優先度を決定します。各ルールのデフォルトの *salience* 値はゼロで、この値が大きいほど優先度が高くなります。*salience* に負の値を使用することも可能です。これにより、相対的にルールの実行順序を適用することができます。



#### 注記

同じ *salience* 値を持つルールの実行順序は任意です。

これを示すため、次に口座残高を出力するルールを追加します。このルールはすべての入出金がすべての口座に適用された後に実行されます。*salience* が負の値であるため、デフォルトの *salience* 値 (ゼロ) を持つルールの後に実行されます。

```
rule "Print balance for AccountPeriod"
    salience -50
when
    ap : AccountPeriod()
    acc : Account()
then
    System.out.println( acc.getAccountNo() + " : " + acc.getBalance() );
end
```

下表は結果となる **agenda** を表しています。3 つの入出金ルールが任意の順番で、出力ルールは最後であるため、入出金ルールの後に実行されます。



#### 重要

**JBoss Rules** には *ruleflow-group* 属性が含まれています。ルールを実行できるタイミングを指定するために、この属性を使用してワークフロー図を宣言します。以下は、**JBoss Developer Studio** によって撮られたスクリーンショットで、2 つの *ruleflow-group* ノードがあります。これにより、レポートルールの前に計算ルールが実行されます。

Agenda		
1	increase balance	arbitrary
2	decrease balance	
3	increase balance	
4	print balance	

図2.7 Cash-Flow および Account

## 2.2.4. 推論

推論とは、ある一部のデータを使用して他のことを推論する行為のことです。たとえば、**age** フィールドと年齢ポリシー制御を提供するルールを持つ **Person** ファクトの場合、Person が成人または子供であるかを推論し、その推論に基づいたアクションを実行できます。

### 例2.1 成人の推論

```
rule "Infer Adult"
when
    $p : Person( age >= 18 )
then
    insert( new IsAdult( $p ) )
end
```

18 歳以上の **Person** には、**IsAdult** のインスタンスが挿入されます。このようなファクトは関係と呼ばれます。この推論された関係はすべてのルールで使用できます。

```
$p : Person()
IsAdult( person == $p )
```

### 2.2.4.1. 使用する推論

次の例では、ある政府部門が成人に ID カードを発行します。ID 部門は、ロンドンに居住している人が 18 歳以上の場合にカードを発行することを示す論理が含まれる決定テーブルを使用します。

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person		
	location	age >= \$1	issueIdCard( \$1 )
	Select Person	Select Adults	Issue ID Card
Issue ID Card to Adults	London	18	p

図2.8 モノリシック決定テーブル

ID 部門は、誰が成人であるかについてポリシーを設定しません。中央政府が成人年齢を 21 歳に変更した場合は、変更管理プロセスが存在します。これを ID 部門に伝え、変更を反映してシステムが確実に更新されるようにする必要があります。

変更管理プロセスは高価で、エラーを引き起こすことがあります。ID 部門は、成人年齢を保存することにより 図2.8「モノリシック決定テーブル」の決定テーブルに必要以上の情報を保持するため、この情報を最新の状態に保つ必要があります。

各部門が独自のルールを保持することにより、作成責任を分割または分離することが可能です。つまり、中央政府が成人年齢を変更した場合、他の機関 (ID 部門) が使用する新しいルールを用いて中央政府は中央リポジトリを更新します。

	RuleTable Age Policy	
	CONDITION	ACTION
	p : Person	
	age >= \$1	insert( \$1 )
	Adult Age Policy	Add Adult Relation
Infer Adult	18	new IsAdult( p )

図2.9 ルールテーブルの年齢ポリシー

これまで説明した **IsAdult** ファクトはポリシールールから推論されます。中央政府が **IsAdult** ファクトを保持するため、ID 部門は成人であるかどうかのみを知る必要があります、ルールが現在のポリシーに適合するよう維持する必要はありません。

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person	IsAdult	
	location	person == \$1	issueIdCard( \$1 )
	Select Person	Select Adults	Issue ID Card
Issue ID Card to Adults	London	p	p

図2.10 ルールテーブルの ID カード

## 2.2.5. 推論および TruthMaintenance

真理維持 (Truth Maintenance) および 論理的挿入 (**logicalInsert**) は、関心の分離を提供するために使用されます。以下の例は、子供または成人用のバス乗車パスを発行します。

```
rule "Issue Child Bus Pass" when
    $p : Person( age < 16 )
then
    insert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
    $p : Person( age >= 16 )
then
    insert(new AdultBusPass( $p ) );
end
```

関心の分離は **logicalInsert** を使用して実現できます。

```

rule "Infer Child" when
    $p : Person( age < 16 )
then
    logicalInsert( new IsChild( $p ) )
end
rule "Infer Adult" when
    $p : Person( age >= 16 )
then
    logicalInsert( new IsAdult( $p ) )
end

```

ファクトは、**when** 節の真偽に基づいて論理的に挿入されます。**when** 節の真偽が `false` に変更された場合、ファクトは自動的に取り消されます。これは、相互排他的なルールには適切です。たとえば、年齢が 15 から 16 に変わった場合、**IsChild** ファクトは自動的に取り消され、**IsAdult** ファクトが挿入されます。

ここで、コードに戻り、乗車パスを発行できます。取り消しセットのカスケードに対する論理的挿入の連鎖を TMS がサポートするため、論理的に挿入することもできます。

```

rule "Issue Child Bus Pass" when
    $p : Person( )
        IsChild( person =$p )
then
    logicalInsert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
    $p : Person( age >= 16 )
        IsAdult( person =$p )
then
    logicalInsert(new AdultBusPass( $p ) );
end

```

**LogicalInsert** を **not** 条件要素と組み合わせて通知を処理できます。この場合、バス乗車パスの返却に対する要求を送信することができます。ChildBusPass オブジェクトが取り消されると、ルールがトリガーされ、要求が対象者に送信されます。

```

rule "Return ChildBusPass Request "when
    $p : Person( )
        not( ChildBusPass( person == $p ) )
then
    requestChildBusPass( $p );
end

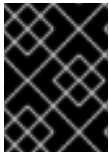
```

## 2.3. 構築およびデプロイに関するその他のコメント

### 2.3.1. Change-Sets を使用したルールの追加

これまでの例では、JBoss Rules API を使用して **knowledge bases** を構築してきました。各ルールは手作業で追加しました。JBoss Rules では、リソースを **knowledge bases** に追加するよう XML ファイルより宣言することもできます。この機能は *change-set* と呼ばれます。

change-set の XML ファイルには、**knowledge base** に追加できるルールリソースのリストが含まれています。このファイルを他のファイルに示すことも可能です。



### 重要

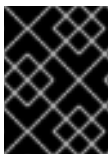
現時点では、change-sets は <add> 要素のみをサポートしています。Red Hat は将来的に <remove> および <modify> 要素のサポートを追加する予定です。

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set
    drools-change-set-5.0.xsd' >

  <add>
    <resource source='http://hostname/myrules.drl' type='DRL' />
  </add>

</change-set>
```

URL は各リソースの場所を指定します。java.net.URL が提供するすべてのプロトコルがサポートされます。**classpath** と呼ばれるプロトコルの使用も可能です。このプロトコルは、リソースの **current processes** クラスパスを参照します。



### 重要

リソースに対して常に type 属性を指定する必要がありますが、ファイル名の拡張子からは推論されません。



### 注記

上記の XML を使用する場合、ResourceType が **CHANGE\_SET** に変更されていること以外は、前述のコードとほぼ同じであることに注意してください。

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClasspathResource( "myChangeSet.xml",
  getClass() ), ResourceType.CHANGE_SET );

if ( kbuilder.hasErrors() ) {
  System.err.println( kbuilder.getErrors().toString() );
}
```

Change-sets には任意数のリソースを含めることが可能です。決定テーブルの設定情報を追加することもできます。次の例では、クラスパスプロトコルを介して、HTTP URL と決定テーブルのスプレッドシートの両方からルールをロードします。

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='http://hostname/myrules.drl' type='DRL' />
    <resource source='classpath:data/IntegrationTest.xls' type="DTABLE">
      <decisiontable-conf input-type="XLS" worksheet-name="Tables_2" />
    </resource>
  </add>
</change-set>
```

```

    </resource>
  </add>
</change-set>

```

ディレクトリにあるすべてのファイルを追加するには、ディレクトリの名前をリソースソースとして使用します (すべてのファイルが指定された型である必要があります)。

```

<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='file://rules/' type='DRL' />
  </add>
</change-set>

```

### 2.3.2. ナレッジエージェント

**KnowledgeAgent** は自動的にルールリソースをロード、再ロード、およびキャッシュします。**properties** ファイルより設定します。

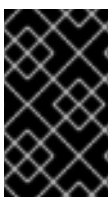
**knowledge base** によって使用されるリソースに変更があった場合、**KnowledgeAgent** は **knowledge base** を更新または再構築できます。これらの更新にストラテジーを設定するには、**KnowledgeAgentFactory** を再設定します。

```

KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent("MyAgent");
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();

```

**KnowledgeAgent** は、追加された各リソースをスキャンします。デフォルトのポーリング間隔は60 秒です。リソースの「最終変更」日が変わった場合、**KnowledgeAgent** は **knowledge base** を再構築します (ディレクトリがリソースの 1 つとして設定された場合は、そのディレクトリのすべての内容が変更に対してスキャンされます)。



#### 重要

以前の **knowledge base** の参照は変更後も存在するため、新たに構築された **knowledge base** へアクセスするには **getKnowledgeBase()** を呼び出す必要があります。

本章ではメソッドとルールの違いについて学習し、アジェンダ、アクティベーションおよび競合のセットがどのように動作し、**KnowledgeAgent** と **knowledge bases** がどのように対話するか説明しました。これにより、本ソフトウェアの仕組みをより深く理解していただけたと思います。また、ステートレスセッションとステートフルセッションについてより包括的な知識を得られたことと思います。

## 第3章 ユーザーガイド

### 3.1. 構築

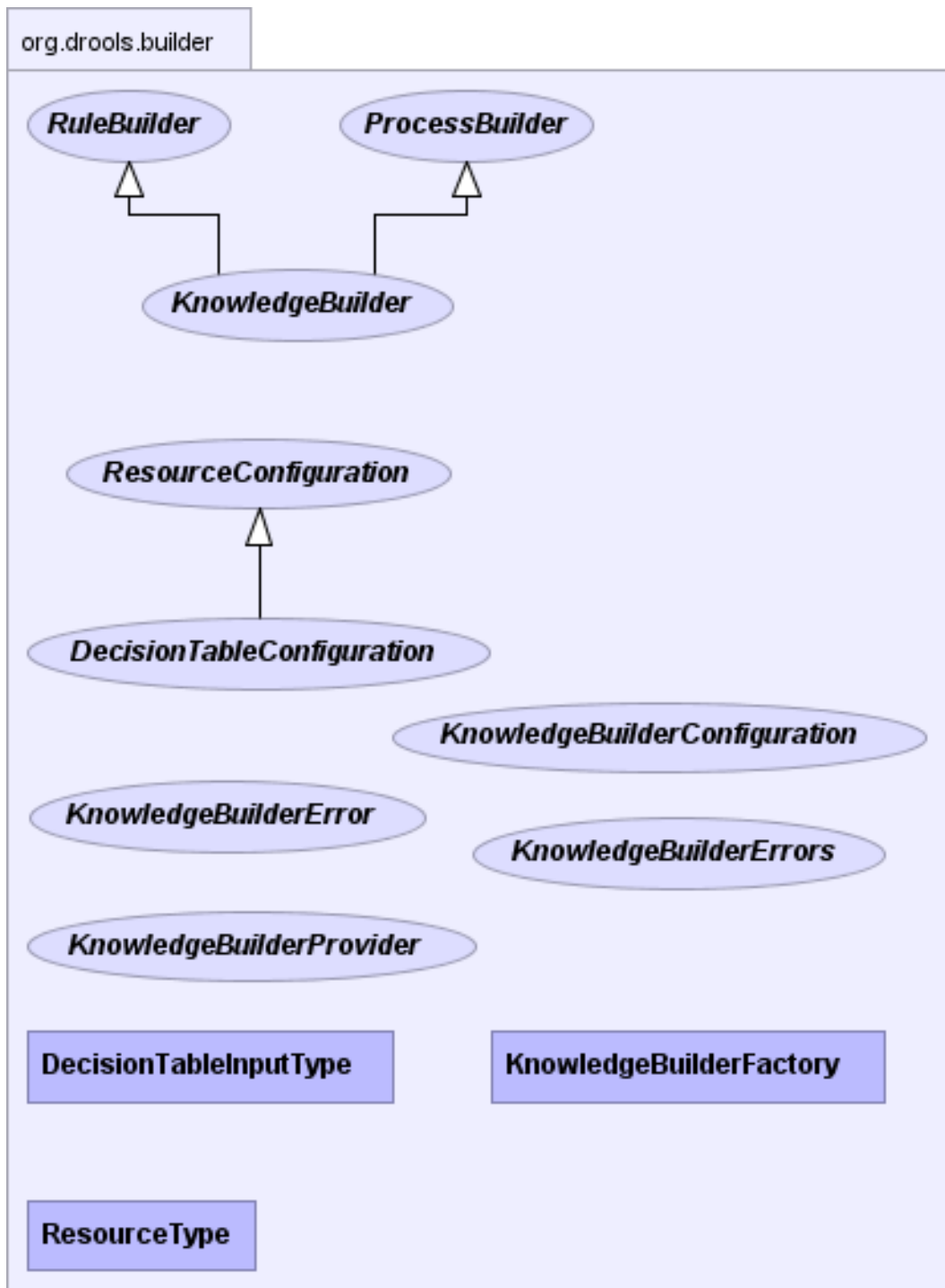


図3.1 org.drools.builder

#### 3.1.1. コードによる構築

**Knowledge Builder** はソースデータを取得し、*knowledge package* に変換します。*knowledge package* には **Knowledge Base** が消費するルールおよびプロセス定義が含まれています。



### 注記

名前の通り、**ResourceType** オブジェクトクラスは構築されるリソースのタイプを示します。

**ResourceFactory** は、**reader**、クラスパス、URI、ファイルまたは **ByteArray** など含む複数のソースよりリソースをロードする機能を提供します。



### 重要

バイナリ (決定テーブルなど) を扱う場合、**Reader** ベースのリソースハンドラーは使用しないでください。これらのハンドラーはプレーンテキストでの使用のみに適していません。

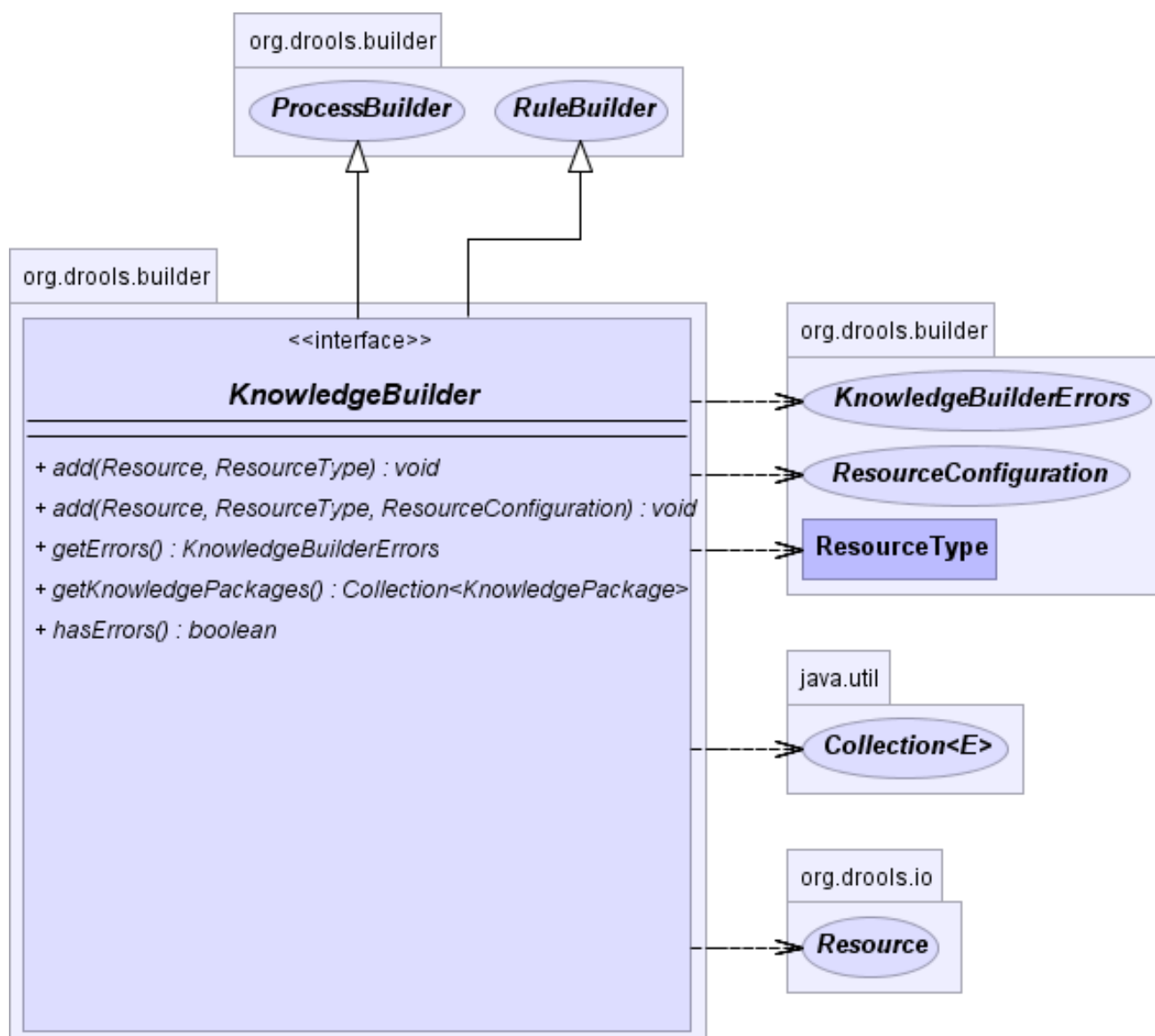


図3.2 KnowledgeBuilder



### 注記

**Knowledge Builder** は **KnowledgeBuilderFactory** によって作成されます。



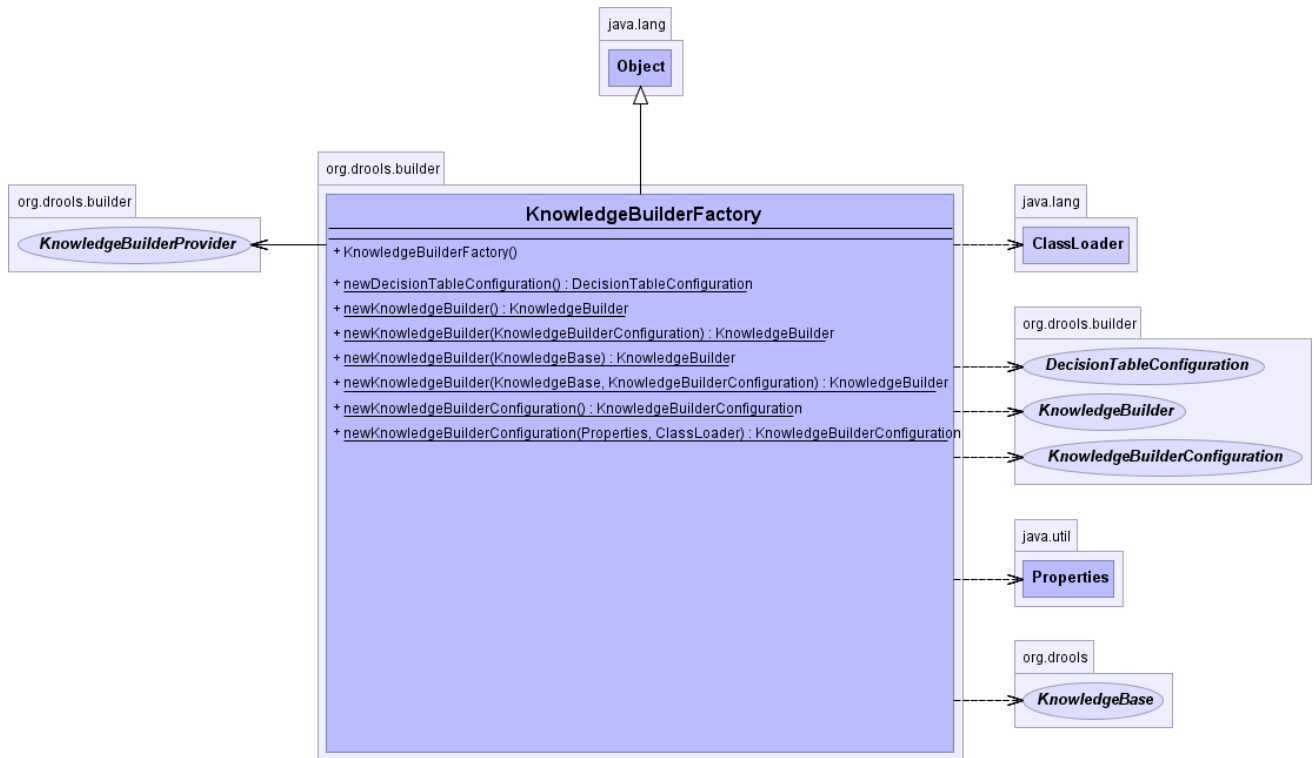


図3.3 KnowledgeBuilderFactory

デフォルトの設定を使用して **Knowledge Builder** を作成します。

### 例3.1 新しい Knowledge Builder の作成

```
KnowledgeBuilder kbuilder =
    KnowledgeBuilderFactory.newKnowledgeBuilder();
```

**KnowledgeBuilderFactory** を使用して設定を作成します。このような設定により **Knowledge Builder** の挙動を変更できるようになります。



#### 注記

ユーザーの多くは、デフォルトパスにないクラスを **Knowledge Builder** オブジェクトが解決できるようにするカスタム クラスローダーを提供するために、この設定の作成を行います。

最初のパラメーターは properties のパラメーターです。これは任意のパラメーターで、null のままにしておくことが可能です。この場合、デフォルトのオプションが使用されます。options パラメーターは、方言の変更や新しい accumulator 関数の登録などのタスクに使用できます。

### 例3.2 カスタムクラスローダーを使用した新しい Knowledge Builder の作成

```
KnowledgeBuilderConfiguration kbuilderConf =
    KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration(
        null, classLoader );

KnowledgeBuilder kbuilder =
    KnowledgeBuilderFactory.newKnowledgeBuilder(kbuilderConf);
```

全タイプのリソースを繰り返し追加することが可能です。次の例では、**.drl** ファイルが追加されます。



#### 注記

**Knowledge Builder** は、JBoss Rules 4.0 Package Builder では不可能であった複数の名前空間の処理を行います。そのため、名前空間に関係なく、リソースの追加を継続することが可能です。

### 例3.3 DRL リソースの追加

```
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules.drl" ),
    ResourceType.DRL );
```



#### 重要

必ず、追加後に **hasErrors()** メソッドをチェックしてください。エラーがある時は、リソースを追加したり **Knowledge Package** を読み出したりしないでください (エラーがある場合、**getKnowledgePackages()** は空のリストを返します)。

### 例3.4 検証

```
if( kbuilder.hasErrors() )
{
    System.out.println( kbuilder.getErrors() );
    return;
}
```

すべてのリソースが追加され、エラーがなくなると **Knowledge Package** のコレクションを取得します (「コレクション」と表現したのは、パッケージの名前空間ごとに1つの **Knowledge Package** があるからです)。これらの **Knowledge Package** は **シリアル化可能** で、頻繁にデプロイメントの単位として使用されます。

### 例3.5 Knowledge Package の取得

```
Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
```

最後の例はこれらすべての要素を組み合わせています。

### 例3.6 全要素の組み合わせ

```
KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
if( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
}
```

```

        return;
    }

    KnowledgeBuilder kbuilder =
        KnowledgeBuilderFactory.newKnowledgeBuilder();
    kbuilder.add( ResourceFactory.newFileResource( "/project/myrules1.drl"
    ),
        ResourceType.DRL);
    kbuilder.add( ResourceFactory.newFileResource( "/project/myrules2.drl"
    ),
        ResourceType.DRL);

    if( kbuilder.hasErrors() )
    {
        System.out.println( kbuilder.getErrors() );
        return;
    }

    Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();

```

### 3.1.2. 設定および Change-Set XML を用いた構築

リソースを追加してプログラミングを行う代わりに、設定を用いて定義を作成することが可能です。これには Change-Set XML を使用します。簡単な XML ファイルは、add、remove、および modify の 3 つの要素をサポートします。これらの各要素は、設定エンティティを定義する resource サブ要素のシーケンスを持ちます。



#### 警告

以下の XML スキーマは *規範的* ではなく、説明する目的でのみ記載されています。

#### 例3.7 Change-Set XML のスキーマ (非「規範的」)

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://drools.org/drools-5.0/change-set"
    targetNamespace="http://drools.org/drools-5.0/change-set">

    <xs:element name="change-set" type="ChangeSet"/>

    <xs:complexType name="ChangeSet">
        <xs:choice maxOccurs="unbounded">
            <xs:element name="add" type="Operation"/>
            <xs:element name="remove" type="Operation"/>
            <xs:element name="modify" type="Operation"/>
        </xs:choice>
    </xs:complexType>

    <xs:complexType name="Operation">

```

```
<xs:sequence>
  <xs:element name="resource" type="Resource"
    maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="Resource">
  <xs:sequence>
    <!-- To be used with <resource type="DTABLE"...> -->
    <xs:element name="decisiontable-conf" type="DecTabConf"
      minOccurs="0"/>
  </xs:sequence>
  <!-- java.net.URL, plus "classpath" protocol -->
  <xs:attribute name="source" type="xs:string"/>
  <xs:attribute name="type" type="ResourceType"/>
  <xs:attribute name="basicAuthentication" type="xs:string"/>
  <xs:attribute name="username" type="xs:string"/>
  <xs:attribute name="password" type="xs:string"/>
</xs:complexType>

<xs:complexType name="DecTabConf">
  <xs:attribute name="input-type" type="DecTabInpType"/>
  <xs:attribute name="worksheet-name" type="xs:string"
    use="optional"/>
</xs:complexType>

<!-- according to org.drools.builder.ResourceType -->
<xs:simpleType name="ResourceType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="DRL"/>
    <xs:enumeration value="XDRL"/>
    <xs:enumeration value="DSL"/>
    <xs:enumeration value="DSLX"/>
    <xs:enumeration value="DRF"/>
    <xs:enumeration value="DTABLE"/>
    <xs:enumeration value="PKG"/>
    <xs:enumeration value="BRL"/>
    <xs:enumeration value="CHANGE_SET"/>
  </xs:restriction>
</xs:simpleType>

<!-- according to org.drools.builder.DecisionTableInputType -->
<xs:simpleType name="DecTabInpType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="XLS"/>
    <xs:enumeration value="CSV"/>
  </xs:restriction>
</xs:simpleType>

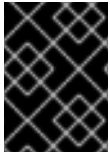
</xs:schema>
```



## 注記

BRMS 5.1 以降、ユーザーは HTTP 上で認証されます。例3.7「Change-Set XML のスキーマ (非「規範的」)」に記載されている次の3つの行は、BRMS 5.1 およびそれ以降のバージョンで必要となりますが、BRMS 5.0 では含まれないようにしてください。

```
<xs:attribute name="basicAuthentication" type="xs:string"/>
<xs:attribute name="username" type="xs:string"/>
<xs:attribute name="password" type="xs:string"/>
```



## 重要

現在、add 要素のみがサポートされています。繰り返しの変更をサポートするため、他の要素も後日実装される予定です。

次の例は単一の **.dr1** ファイルをロードします。

### 例3.8 単純な Change-Set XML

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd' >
  <add>
    <resource source='file:/project/myrules.dr1' type='DRL' />
  </add>
</change-set>
```

リソースのプロトコルを示す **file:** プレフィックスに注目してください。Change-Set は、**file** や **http** など **java.net.URL** によって提供されるすべてのプロトコルをサポートし、**classpath** の追加バージョンもサポートします。



## 重要

ファイル名の拡張子から推論されないため、必ずリソースに **type** 属性を指定するようにしてください。

Java で **ClassPath resource loader** を使用して、リソースの特定に使用される **class loader** を指定します (XML では不可能です)。デフォルトでは、**Knowledge Builder** によって使用される **class loader** が使用されます (Change-Set XML が **ClassPath** リソースによってロードされる場合を除きます。この場合、リソースに対して指定された **class loader** が使用されます)。

### 例3.9 Change-Set XML のロード

```
kbuilder.add(ResourceFactory.newUrlResource(url), ResourceType.CHANGE_SET);
```

change-set には任意数のリソースを含めることができます。結果的に、これらのリソースは追加の設定情報をサポートします (現在、この使用は決定テーブルのみに制限されています)。例3.10「リソース設

定を用いた [Change-Set XML](#) は HTTP URL とクラスパス上の Excel 決定テーブルの両方よりルールをロードします。

### 例3.10 リソース設定を用いた Change-Set XML

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='http:org/domain/myrules.drl' type='DRL' />
    <resource source='classpath:data/IntegrationExampleTest.xls'
      type="DTABLE">
      <decisiontable-conf input-type="XLS" worksheet-
name="Tables_2" />
    </resource>
  </add>
</change-set>
```

変更通知機能を提供し、自動的に **Knowledge Base** を再構築するため、Change-Set は **Knowledge Agent** を使用する時に便利です (これらの機能の詳細は **Knowledge Agent** の項の「デプロイメント」に記載されています)。

ディレクトリ内のリソースをすべて追加するため、ディレクトリを指定することも可能です (ソフトウェアはすべてのリソースが同じタイプであることを想定します)。 **Knowledge Agent** を使用する場合、リソースへの変更を継続的にスキャンします。また、キャッシュされた **Knowledge Base** も再構築します。



#### 注記

change-set を **Knowledge Agent** と併用することも可能です。詳細は「[KnowledgeAgent](#)」を参照してください。

### 例3.11 ディレクトリの内容を追加するための Change-Set XML コード

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='file:/projects/myproject/myrules' type='DRL'
  />
  </add>
</change-set>
```

## 3.2. デプロイメント

### 3.2.1. KnowledgePackage とナレッジ定義

*KnowledgePackage* は ナレッジ定義のコレクションです。ナレッジ定義とは、ルールとプロセスの別の呼び方です。「構築」に説明がある通り、**KnowledgePackage** は **KnowledgeBuilder** によって

作成されます。**KnowledgePackage** は自己充足的で、シリアル化可能です。現在の基本的なデプロイメントユニットを形成します。

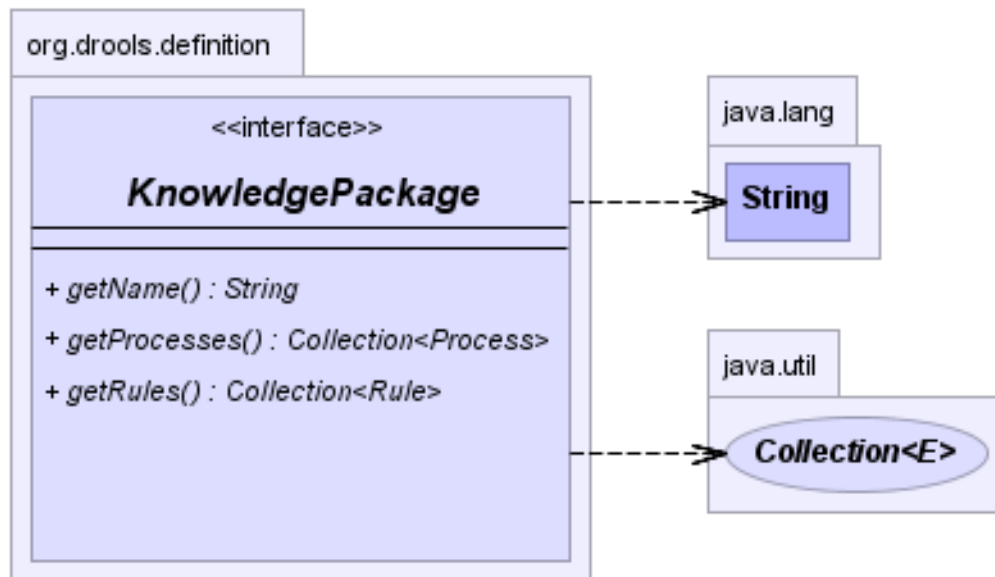


図3.4 KnowledgePackage



#### 重要

**KnowledgePackage** は **Knowledge Base** に追加されますが、**KnowledgePackage** インスタンスは追加されると再使用できないことに注意してください。別の **knowledge base** に追加するには、最初に **シリアル化**を行い、「クローン」された結果を使用します。**JBoss Rules** の今後のバージョンでは、この制限がなくなる予定です。

### 3.2.2. ナレッジベース

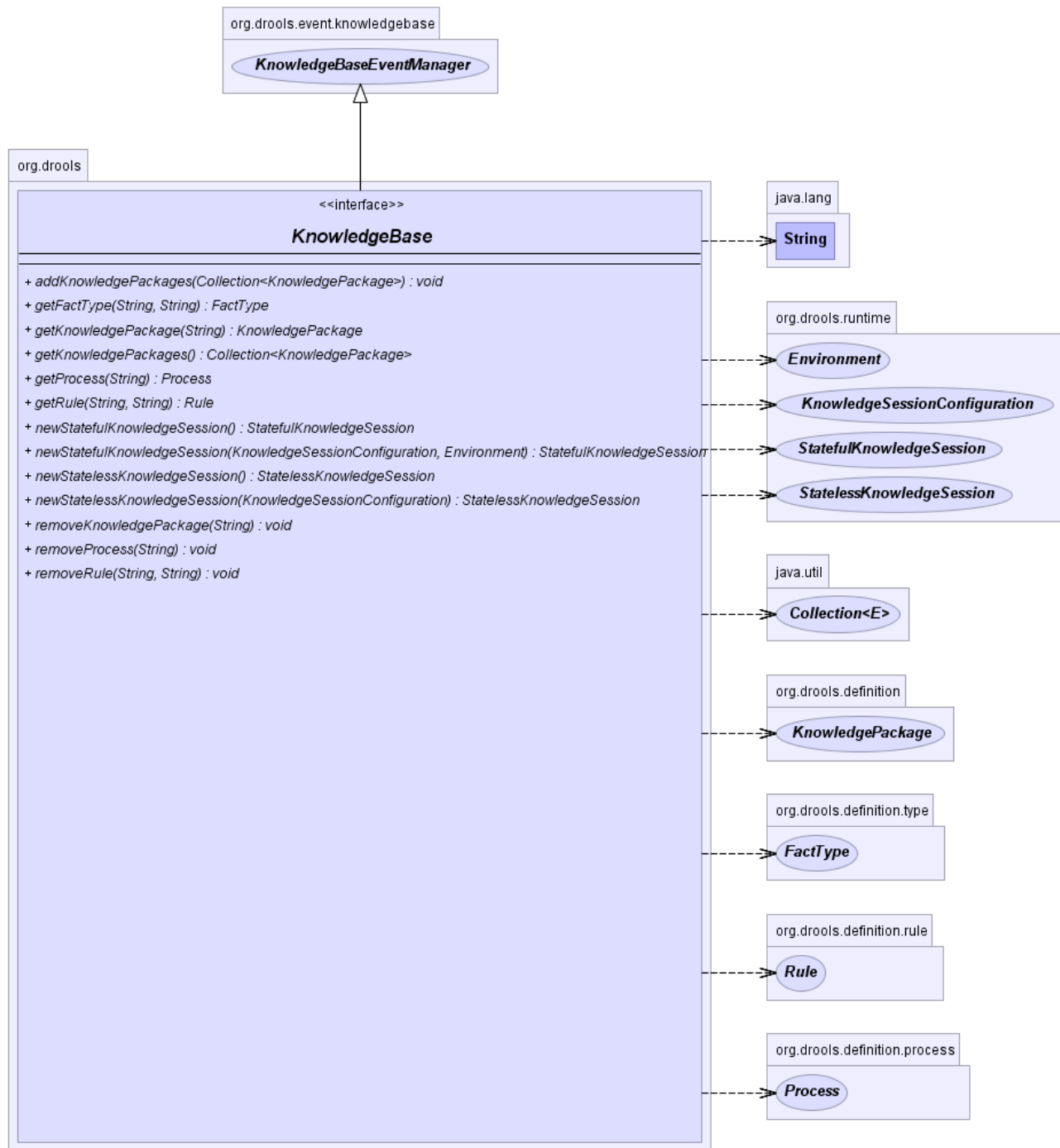


図3.5 ナレッジベース

**knowledge base** は、アプリケーションすべての ナレッジ定義が格納されるレポジトリです。これには、ルール、プロセス、関数およびタイプモデルが含まれます。**knowledge base** 自体には「インスタンス」データ (ファクト) は含まれません。この代わりに、セッションは **Knowledge Base** より作成されます。**facts** は **Knowledge Base** へ挿入され、**process instances** は **Knowledge Base** より開始します。



### 重要

**knowledge base** の作成は比較的资源を多く使用するプロセスですが、セッションの作成はそうではありません。よって、セッションを繰り返し作成できるようにするため、Red Hat は可能な限り **knowledge bases** をキャッシュすることを推奨します。



**knowledge bases** オブジェクトも **シリアル化可能** であるため、構築して保存する方がよいでしょう。こうすることで、**knowledge packages** ではなくデプロイメントの単位として取り扱うことが可能です。

**knowledge base** を作成する方法の 1 つが **KnowledgeBaseFactory** クラスを使用する方法です。

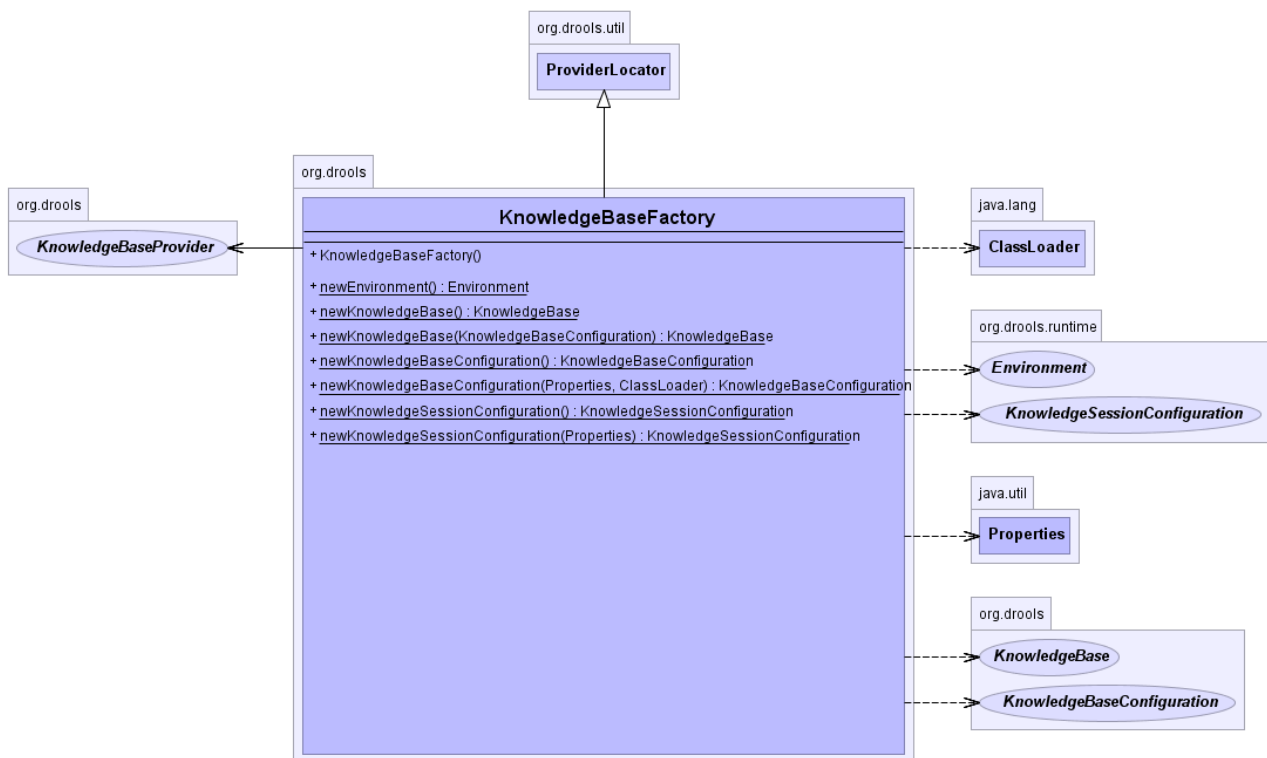


図3.6 KnowledgeBaseFactory

デフォルト設定を使用して作成する方法もあります。

### 例3.12 新しいナレッジベースの作成

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
```

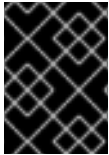
**Knowledge Builder** と共にカスタマイズされた **class-loader** を使用してデフォルトの **loader** がない **types** を解決したい場合、これを **Knowledge Base** に設定します (このテクニックは **Knowledge Builder** に適用されるものと同じです)。

### 例3.13 カスタム Class-Loader を用いた新しいナレッジベースの作成

```
KnowledgeBaseConfiguration kbaseConf =
    KnowledgeBaseFactory.newKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase( kbaseConf
);
```

## 3.2.3. インプロセス構築およびデプロイメント

デプロイメントの最も簡単な形式は **インプロセス構築** と呼ばれます。この場合、ナレッジ定義がコンパイルされ、同じ Java 仮想マシンにある **knowledge base** に追加されます。



## 重要

この方法を使用する際、**drools-`{module}`-`{version}`.jar** ファイルが必ずクラスパス上にあるようにしてください。

### 例3.14 Knowledge Packages の Knowledge Base への追加

```
Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kpkgs );
```



## 注記

**addKnowledgePackages(kpkgs)** メソッドは繰り返し呼び出せることを理解してください。これはナレッジを追加するために行います。

### 3.2.4. 別プロセスとしての構築およびデプロイメント

**Knowledge Base** と **KnowledgePackage** は両方ともデプロイメントの単位であるため、シリアル化することが可能です。そのため、**drools-compiler.jar** に必要な構築を実行するため 1 つのマシンを割り当て、すべてをデプロイおよび実行するために他のマシンを確保することができます。2 つ目のマシンは **drools-core.jar** のみを必要とします。

シリアル化は標準的な Java の用法ですが、以下の例は、あるマシンがデプロイメントユニットを書き込むと別のマシンがどのようにそれを読み取って使用するかを表しています。

### 例3.15 KnowledgePackage を出力ストリームへ書き込む

```
ObjectOutputStream out =
    new ObjectOutputStream( new FileOutputStream( fileName ) );
out.writeObject( kpkgs );
out.close();
```

### 例3.16 入力ストリームから KnowledgePackage を読み取る

```
ObjectInputStream in = new ObjectInputStream( new FileInputStream(
    fileName ) );
// The input stream might contain an individual
// package or a collection.
@SuppressWarnings( "unchecked" )
Collection<KnowledgePackage> kpkgs =
    (Collection<KnowledgePackage>) in.readObject();
in.close();

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kpkgs );
```

実際の **knowledge base** 自体もシリアル化可能であるため、ナレッジパッケージではなく構築と保存を行うことが推奨されます。



#### 注記

Red Hat のサーバー側管理システムである Drools Guvnor はこのデプロイメント方法を使用します。シリアル化された **knowledge packages** を URL へコンパイルおよびパブリッシュした後、このアドレスリソースタイプを使用してこれらをロードできます。

### 3.2.5. ステートフルナレッジセッションとナレッジベースの変更

ステートフルナレッジセッションについては「[StatefulKnowledgeSession](#)」で詳細に説明します。ステートフルナレッジセッションは **Knowledge Base** によって作成され、返されます。また、任意で参照を保持することも可能です。**Knowledge Base** が変更されると、変更はセッションのデータに適用されます。これは弱い任意の参照で、ブール値フラグによって制御されます。

### 3.2.6. KnowledgeAgent

**KnowledgeAgent** は、リソースの自動ロード、キャッシュ、および再ロードを提供するクラスで、プロパティファイルより設定されます。使用するリソースが変更されると、**KnowledgeAgent** は **Knowledge Base** を更新または再構築することができます。**factory** の設定は使用されるストラテジーを決定します (通常はプルベースで、標準のポーリングを使用します)。



#### 注記

ブッシュベースの更新および再構築を行う機能は、今後のバージョンで追加される予定です。

**KnowledgeAgent** は、デフォルトのポーリング間隔 (60 秒) で追加されたリソースをすべて継続してスキャンします。最後の変更の日付が更新されると、キャッシュされた **Knowledge Base** は新しいリソースを使用して自動的に再構築されます。

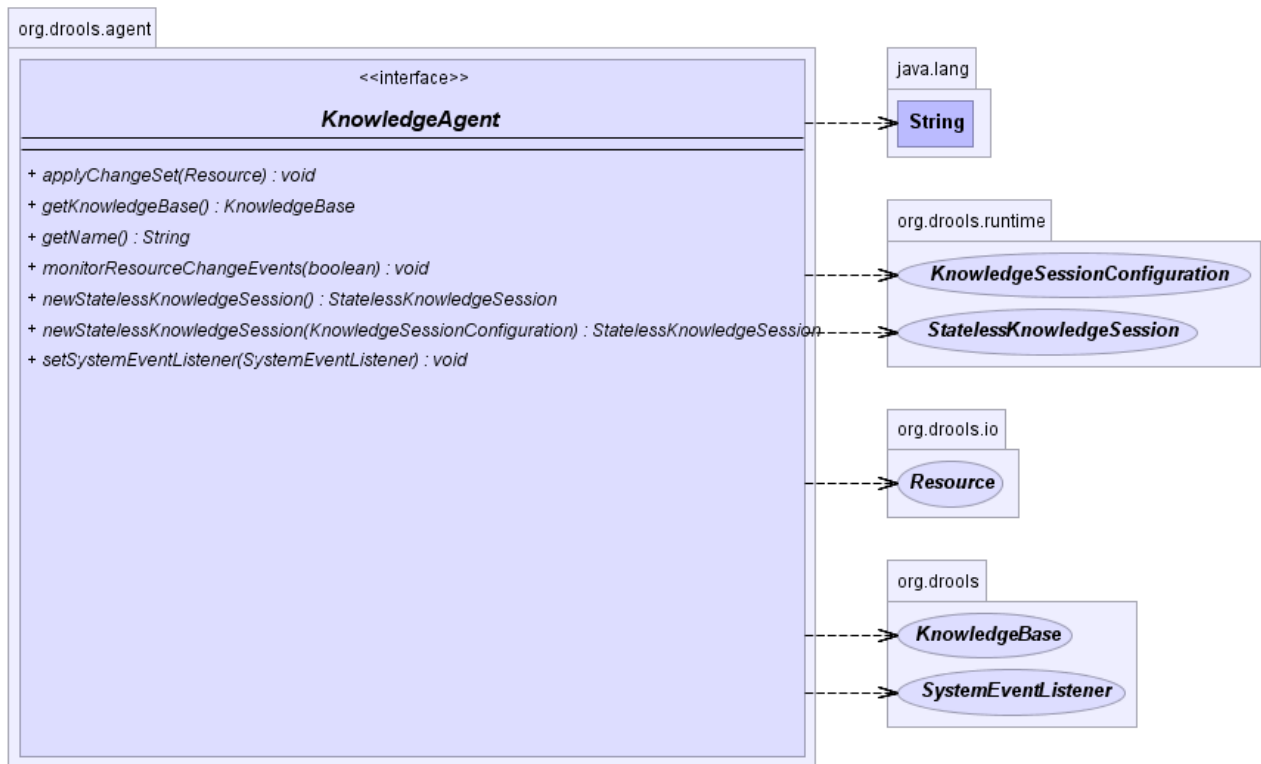


図3.7 KnowledgeAgent

**KnowledgeBuilderFactory** オブジェクトは **Knowledge Builder** を作成するために使用されます。ログファイルが必要とするため、エージェントは名前を指定する必要があります (ログエントリを正しいエージェントへ関連付けできるようにするため)。

### 例3.17 KnowledgeAgent の作成

```
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent" );
```

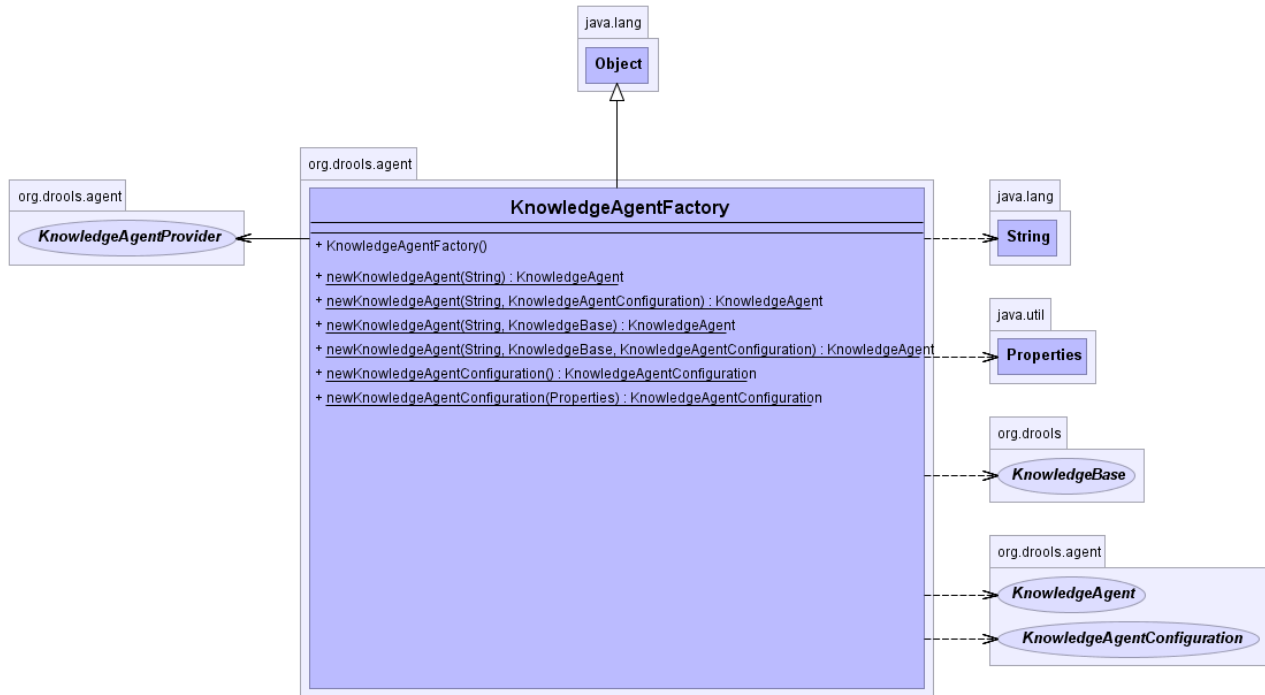


図3.8 KnowledgeAgentFactory

以下の例は、指定された change-set から新しい **knowledge base** を構築するエージェントを構築します。



#### 注記

change-set の詳細情報は「[設定および Change-Set XML を用いた構築](#)」を参照してください。



#### 注記

メソッドは繰り返し呼び出すことが可能です。これにより、徐々に新しいリソースを追加することができます。

**KnowledgeAgent** は、**change set** から追加されたリソースを 60 秒ごと (デフォルトの間隔) にポーリングして、更新されたか確認します。変更が検出されると、新しい **Knowledge Base** を構築します。また、ディレクトリがリソースとして指定された場合は、その内容がスキャンされます。

#### 例3.18 KnowledgePackage を出力ストリームへ書き込む

```
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent" );
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

リソースのスキャンはデフォルトで無効になっています。これはサービスであるため、特別に開始する必要があります。通知に関しても同様です。スキャンと通知の両方を **ResourceFactory** を用いてアクティベートします。

#### 例3.19 スキャンおよび通知サービスの開始

```
ResourceFactory.getResourceChangeNotifierService().start();
ResourceFactory.getResourceChangeScannerService().start();
```

**ResourceChangeScannerService** クラスより、デフォルトのリソーススキャン期間を変更します (更新された **ResourceChangeScannerConfiguration** オブジェクトはサービスの **configure()** メソッドに渡されるため、サービスは要求に応じて再設定できるようになります)。

### 例3.20 スキャン間隔の変更

```
ResourceChangeScannerConfiguration sconf =
    ResourceFactory.getResourceChangeScannerService().
        newResourceChangeScannerConfiguration();
// Set the disk scanning interval to 30s, default is 60s.
sconf.setProperty( "drools.resource.scanner.interval", "30" );
ResourceFactory.getResourceChangeScannerService().configure( sconf );
```

**KnowledgeAgents** は、空の **Knowledge Bases** と値が入力された **Knowledge Bases** の両方を処理できます。値が入力された **Knowledge Bases** が提供された場合、**KnowledgeAgent** が内部から **イテレーター** を実行し、見つかった各リソースをサブスクライブします。



#### 警告

**KnowledgeBuilder** がディレクトリのリソースをすべて構築するようにすることは可能ですが、この情報は失われます。つまり、これらのディレクトリは連続してスキャンされません。**applyChangeSet(Resource)** メソッドによって指定されたディレクトリのみが監視されます。



#### 注記

**Knowledge Base** を土台として使用する利点の1つは、**KnowledgeBaseConfiguration** クラスで **Knowledge Base** を提供できることです。リソースの変更が検出され、新しい **Knowledge Base** がインスタンス化されると、以前の **Knowledge Base** オブジェクトに属する **KnowledgeBaseConfiguration** クラスが使用されます。

### 例3.21 既存のナレッジベースの使用

```
KnowledgeBaseConfiguration kbaseConf =
    KnowledgeBaseFactory.newKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase( kbaseConf
);
// Populate kbase with resources here.
```

```
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent", kbase );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

上記の例では、リソースの変更が検出され、新しい **Knowledge Base** が構築されるまで **getKnowledgeBase()** メソッドは同じ **Knowledge Base** インスタンスを返します。これは、以前の **Knowledge Base** へ提供された **KnowledgeBaseConfiguration** を用いて行われます。

### 例3.22 ディレクトリの内容を追加する Change-Set XML

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
  xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
  xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd' >
  <add>
    <resource source='file:/projects/myproject/myrules' type='PKG' />
  </add>
</change-set>
```



#### 注記

**drools-compiler** 依存関係は、**PKG** という名前のリソースタイプには必要ありません。**KnowledgeAgent** は **drools-core** のみを用いてこのようなリソースタイプに対応できます。

**KnowledgeAgentConfiguration** を使用して **KnowledgeAgent** のデフォルトの動作を変更します。これは、変更に対するディレクトリの連続スキャンを抑制しながらディレクトリよりリソースをロードするために行います。

### 例3.23 スキャンの挙動変更

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

KnowledgeAgentConfiguration kaconf =
    KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
// Do not scan directories, just files.
kaconf.setProperty( "drools.agent.scanDirectories", "false" );
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "test agent", kaconf );
```

これまで、**JBoss Enterprise BRMS Platform** が URL よりシリアル化された **Knowledge Packages** を構築およびパブリッシュする方法と、Change-Set XML が URL とパッケージの両方を処理する方法を見てきました。これらの方法は、**Knowledge Agent** の重要なデプロイメントのシナリオを形成します。

## 3.3. 実行

### 3.3.1. ナレッジベース



**KnowledgeBase** はアプリケーションの ナレッジ定義 がすべて含まれるレポジトリで、ルール、プロセス、関数、およびタイプモデルが含まれることがあります。**Knowledge Base** 自体にはインスタンスデータ (ファクトと呼ばれます) は含まれません。ファクトが挿入でき、プロセスインスタンスが起動できる **KnowledgeBase** よりセッションが作成されます。



#### 注記

**Knowledge Base** の作成はリソースを大量に消費するプロセスですが、セッションの作成はリソースを大量に消費しません。繰り返しセッションを作成できるようにするため、可能な限り **Knowledge Bases** をキャッシュするようにしてください。

#### 例3.24 新しいナレッジベースの作成

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
```

### 3.3.2. StatefulKnowledgeSession

**StatefulKnowledgeSession** はランタイムデータを格納し、実行します。**StatefulKnowledgeSession** は **KnowledgeBase** より作成されます。

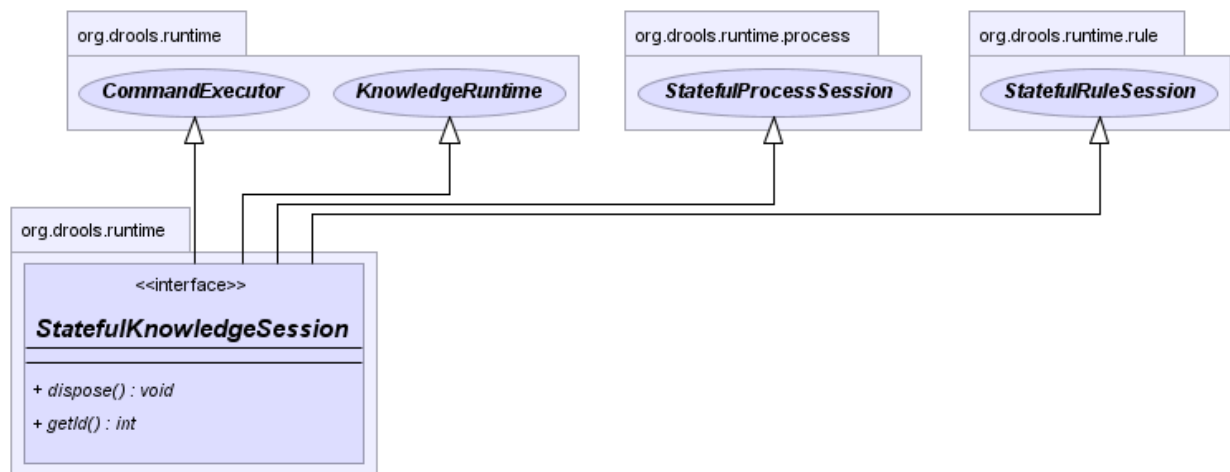


図3.9 StatefulKnowledgeSession

#### 例3.25 KnowledgeBase より StatefulKnowledgeSession を作成

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

### 3.3.3. KnowledgeRuntime

#### 3.3.3.1. WorkingMemoryEntryPoint

**WorkingMemoryEntryPoint** は、ファクトを挿入、更新、および読み出しするメソッドを提供します。





## 注記

エントリーポイントという用語は、**working memory**に複数のパーティションが存在し、ファクトが挿入されるパーティションを選択できることに関連しています。しかし、このユースケースはイベント処理を対象とし、ほとんどのルールベースのアプリケーションはデフォルトのエントリーポイントのみを使用します。

**KnowledgeRuntime** インターフェースは **engine** と主な対話を行い、ルールの結果とプロセスのアクションで使用可能です。メソッドとインターフェースに関連するルールに焦点を置きますが、**KnowledgeRuntime** は **WorkingMemory** と **ProcessRuntime** の両方よりメソッドを継承します。これは、プロセスとルールに対応する統合 API を提供します。ルールを用いて作業する時、**WorkingMemoryEntryPoint**、**WorkingMemory**、および **KnowledgeRuntime** の 3 つのインターフェースが **KnowledgeRuntime** を形成します。

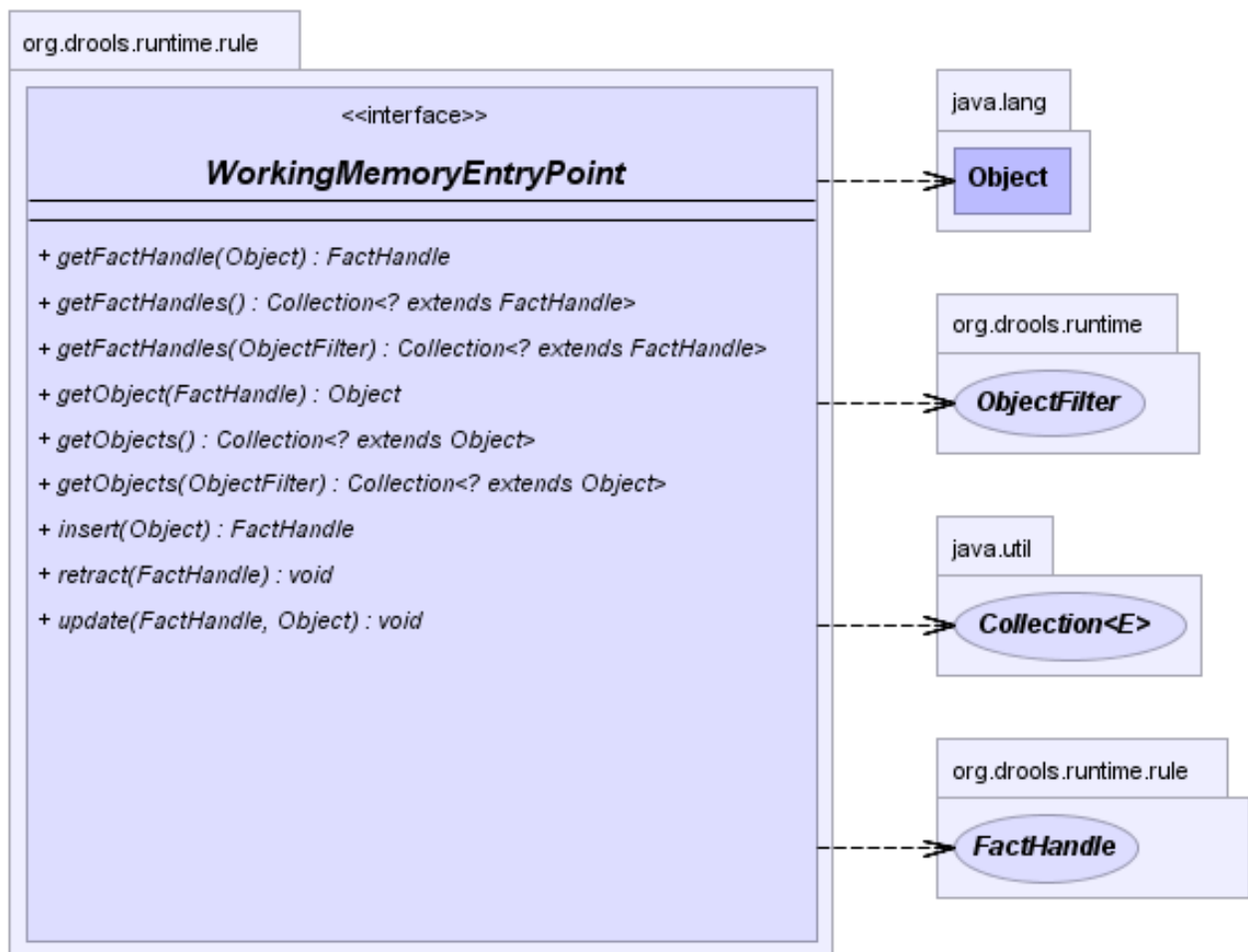


図3.10 WorkingMemoryEntryPoint

### 3.3.3.1.1. 挿入 (insert)

挿入 (*insert*) は **WorkingMemory** にファクトについて通知する行為です (`ksession.insert(yourObject)` など)。挿入が行われると、システムはルールに対する一致があるか各ファクトを調べます。ルール実行の有無に関する決定はすべて挿入時に行われます。ただし、`fireAllRules()` が呼び出されるまでルールは実行されません。必ずすべてのファクトが挿入された後に行います。



## 注記

**fireAllRules()** が呼び出された時に条件評価が行われるという誤った考えを持ったユーザーが過去に存在しました。



## 注記

*assert* (アサート) または *assertion* という用語は、利用可能なファクトを示すため、通常は専門システムと関連して使用されます。しかし、ほとんどの言語で「assert」がキーワードとして使用されるため、混乱を防ぐために Red Hat は *insert* (挿入) というキーワードを使用することにしました。そのため、*assert* と *insert* は多くの場合、同じ意味で使用されます。

オブジェクトが挿入されると、ファクトハンドルを返します。**FactHandle** は **working memory** 内で挿入されたオブジェクトを示すために使用されるトークンです。オブジェクトが変更または取り消された時に、**working memory** との対話にも使用されます。

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );
```

**working memory** は、*equality* と *identity* の 2 つのアサーションモードのいずれかで操作します (デフォルトは *identity* です)。

- **Identity** が使用される場合、**working memory** は **IdentityHashMap** を使用してアサートされたオブジェクトをすべて格納します。インスタンスが新たにアサートされると、常に新しい **FactHandle** が返されます。同じインスタンスを繰り返し挿入すると、元のファクトハンドルが返されます。
- **Equality** が使用される場合、**working memory** は **HashMap** を使用してアサートされたオブジェクトをすべて格納します。同等のオブジェクトがアサートされていない場合、インスタンスが新たにアサートされると、新しい **FactHandle** のみが返されます。

### 3.3.3.1.2. 取り消し

取り消しは、**working memory** よりファクトを削除することを意味します。ファクトの追跡やルール的一致は行われなくなります。さらに、そのファクトに依存するアクティベートされたルールは、キャンセルされます。取り消しは、アサート時に返された **FactHandle** を使用して実行されます。



## 注記

特定のファクトが存在しない時に実行されるルールを作成することは可能です (**not** および **exist** キーワードを使用)。このような場合にファクトを取り消すと、ルールがアクティベートされる原因となることがあります。

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = ksession.insert( stilton );

ksession.retract( stiltonHandle );
```

### 3.3.3.1.3. 更新

ファクトの変更を **rule engine** に通知し、変更されたファクトを再処理できるようにする必要があります。ファクトが更新済みであると見なされる場合、**working memory** より自動的に取り消され、再度挿入されます。

変更されたオブジェクトが **working memory** 自体に通知できない場合、**update** メソッドを使用して通知します。**update** メソッドは常に変更されたオブジェクトをセカンダリパラメーターとして取ります。これにより、新しいインスタンスを 不変オブジェクト に対して指定できます。



#### 注記

**update** メソッドは、シャドウプロキシが有効になったオブジェクトのみに使用できません。



#### 重要

**update** メソッドは Java コードと併用する場合のみ使用できます。オブジェクトの setter メソッドへの呼び出しを提供するため、ルール内で **modify** キーワードを使用します。

```
Cheese stilton = new Cheese("stilton");
FactHandle stiltonHandle = workingMemory.insert( stilton );
...
stilton.setPrice( 100 );
workingMemory.update( stiltonHandle, stilton );
```

#### 3.3.3.2. ワーキングメモリー

**working memory** は **agenda** へのアクセスを提供します。また、クエリの実行を許可し、名前付きの **entry points** へのアクセスを許可します。

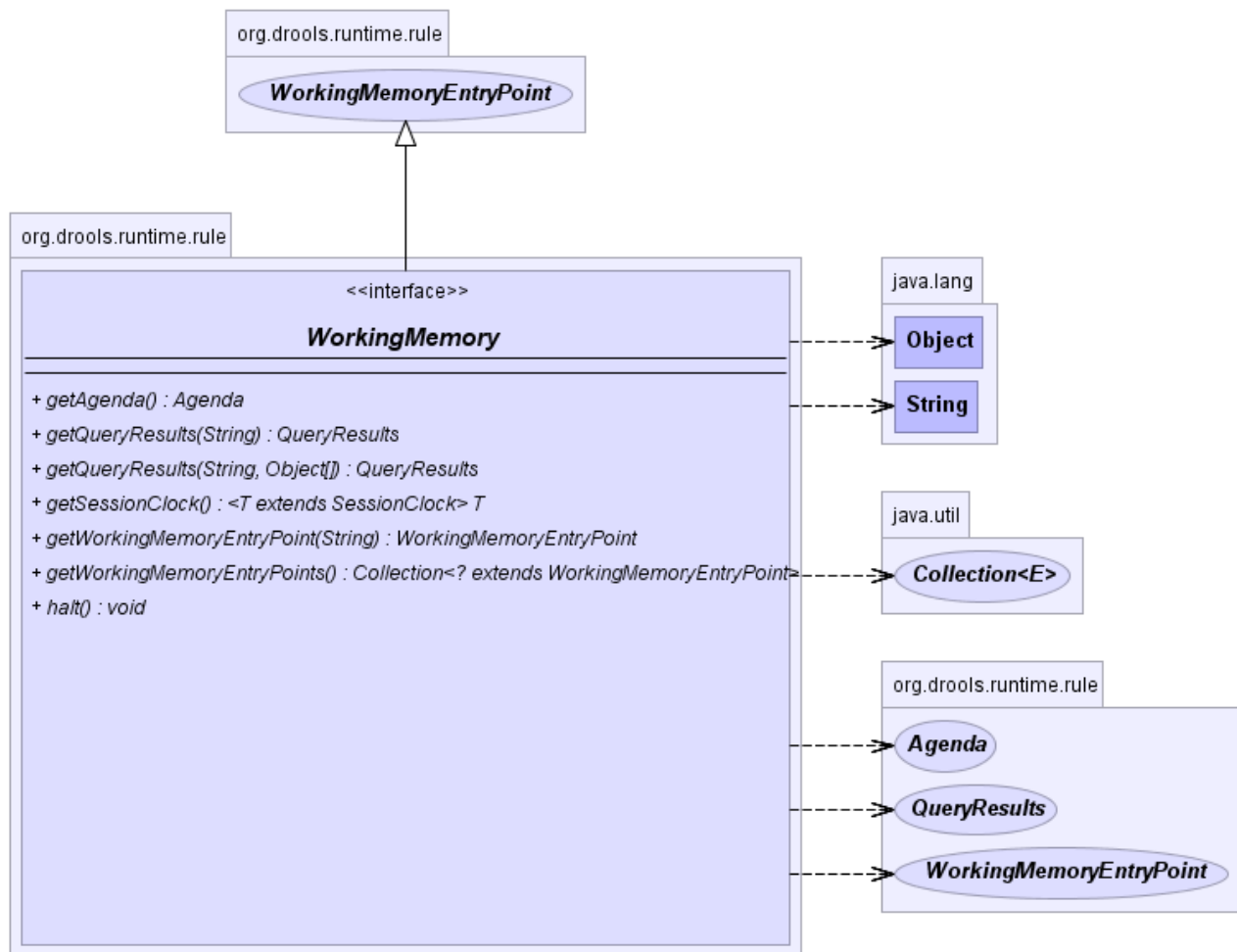


図3.11 ワーキングメモリー

### 3.3.3.2.1. クエリ

クエリを使用してファクトセットを読み出します。ルールで使用されるためパターンが基になります。これらのパターンはオプションのパラメーターを使用することもあります。

**Knowledge Base** にクエリを定義します。**Knowledge Base** よりクエリを呼び出して一致する結果を返します。結果コレクション上で繰り返し替えされる間、**get(String identifier)** メソッドを使用してクエリのバインド識別子へアクセスできます。**getFactHandle(String identifier)** を使用すると、その識別子の **FactHandle** を読み出すことができます。

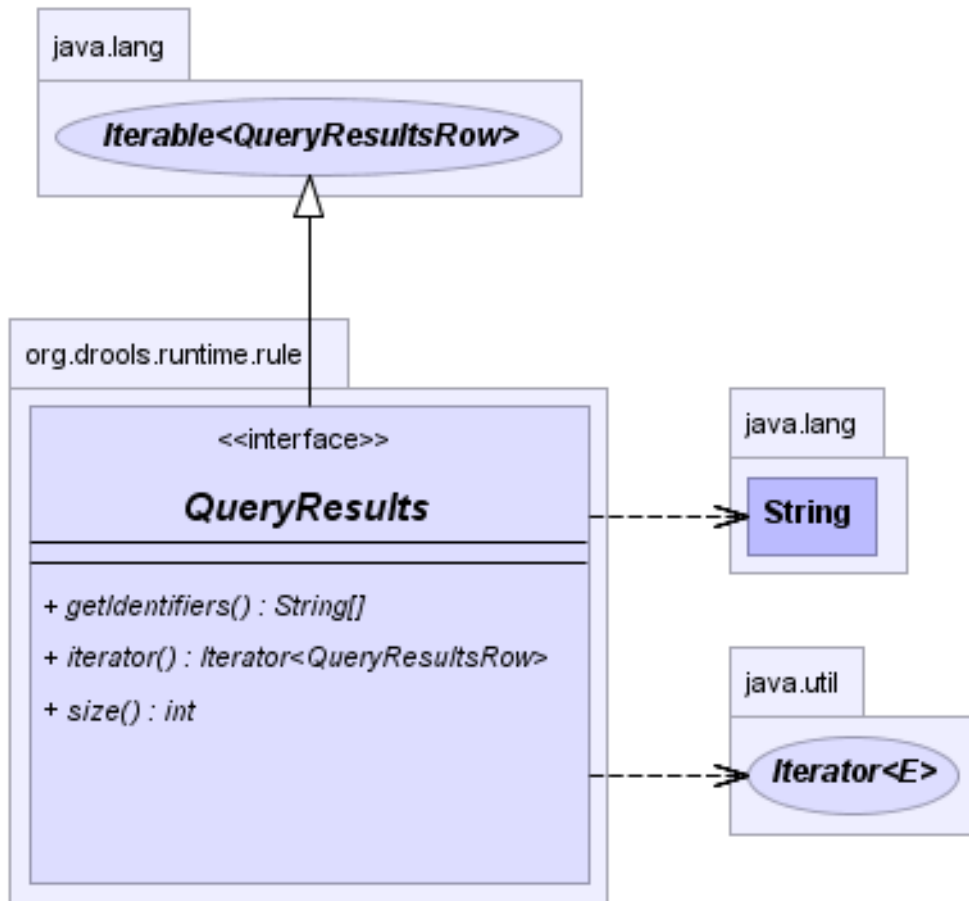


図3.12 クエリ結果

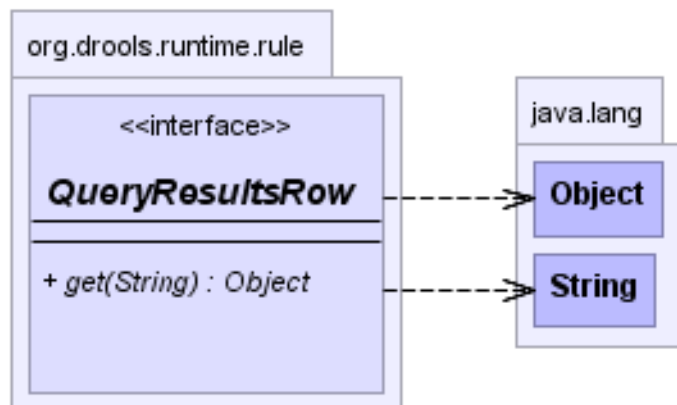


図3.13 QueryResultsRow

## 例3.26 簡単なクエリの例

```

QueryResults results =
    ksession.getQueryResults( "my query", new Object[] { "string" } );
for ( QueryResultsRow row : results ) {
    System.out.println( row.get( "varName" ) );
}

```

### 3.3.3.3. ライブクエリ

**JBoss Enterprise BRMS 5.2** はライブクエリをサポートします。

ライブクエリはアタッチされたリスナーを使用し、ビューとして開かれます。また、このビューの内容に対する変更イベントをパブリッシュします。これにより、パラメーターを用いてクエリを実行することが可能になり、結果ビューで変更をリッスンできるようになります。

#### 例3.27 ViewChangedEventListener の実装

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
    public void rowUpdated(Row row) {
        updated.add( row.get( "$price" ) );
    }

    public void rowRemoved(Row row) {
        removed.add( row.get( "$price" ) );
    }

    public void rowAdded(Row row) {
        added.add( row.get( "$price" ) );
    }
};

// Open the LiveQuery
LiveQuery query = ksession.openLiveQuery( "cheeses",
                                           new Object[] { "cheddar",
                                                           "stilton" },
                                           listener );

...
...
query.dispose() // make sure you call dispose when you want the query to
close
```

### 3.3.3.4. KnowledgeRuntime

**KnowledgeRuntime** は、ルールとプロセスの両方へ適用可能なその他のメソッドを提供します。グローバルを設定したり、**ExitPoints** を登録するメソッドがこの一例となります。

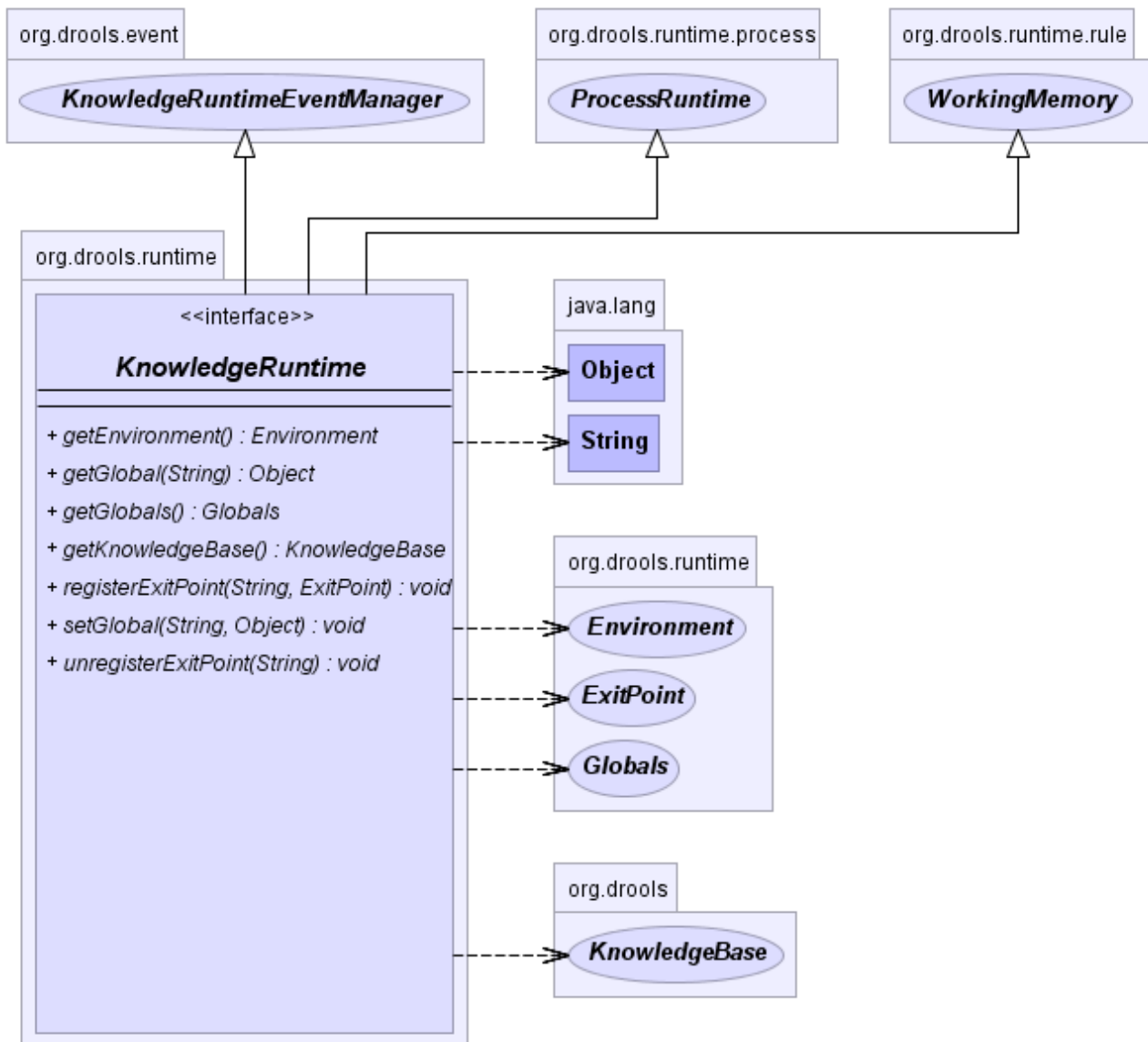


図3.14 KnowledgeRuntime

### 3.3.3.4.1. グローバル

グローバルは **rule engine** へ渡すことができる名前付きオブジェクトです。挿入する必要はありません。統計情報やルールで右側で使われるサービスに対して最も頻繁に使用されます。また、**rule engine** よりオブジェクトを返す手段としても使用されます。

ルールの左側でグローバルを使用するには、次の手順に従います。

1. 不変であることを確認します。
2. セッションに設定する前に **rules** ファイルで宣言します。

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

3. これで、**Knowledge Base** がグローバル識別子とそのタイプを認識するようになったため、任意セッションの **ksession.setGlobal** を呼び出します。

**警告**

最初にグローバルタイプと識別子を宣言しないと、例外がスローされます。

- セッションにグローバルを設定するには、`ksession.setGlobal(identifier, value)` を使用します。

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

**警告**

設定される前にルールがグローバルを評価すると、**NullPointerException** 例外がスローされます。

### 3.3.3.5. StatefulRuleSession

**NullPointerException** は **StatefulKnowledgeSession** によって継承されます。これは、**engine** 外部に適用可能なルール関連のメソッドを提供します。

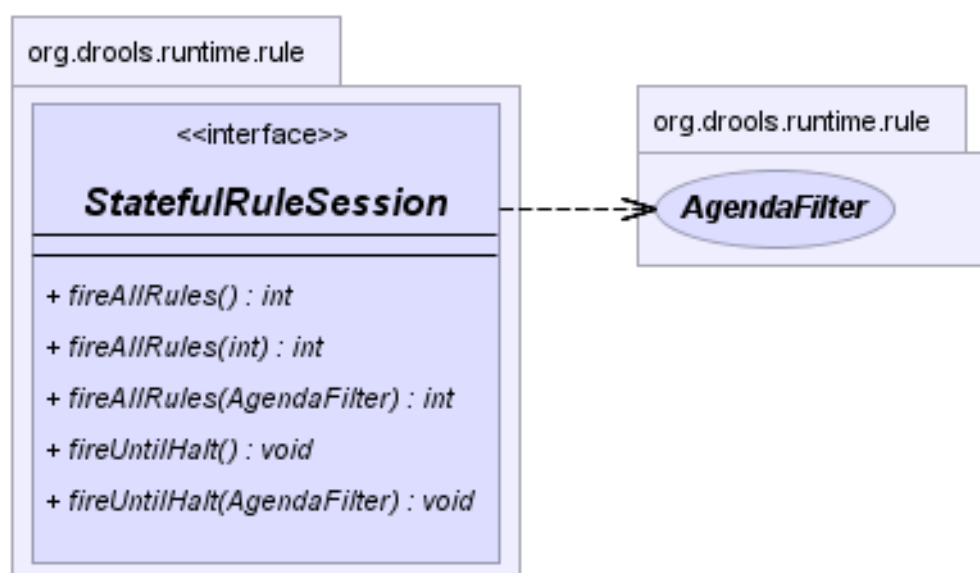


図3.15 StatefulRuleSession

#### 3.3.3.5.1. アジェンダ フィルター



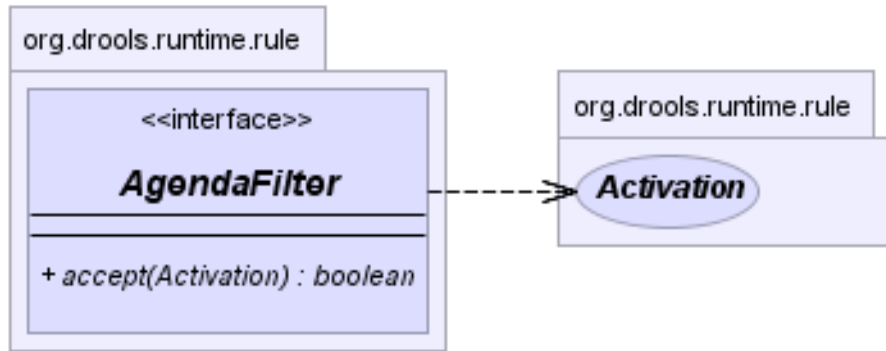


図3.16 AgendaFilters

アジェンダフィルターは **filter** インターフェースの実装です。アジェンダフィルターを使用して実行権利の有効化を許可または拒否します (フィルターできるかは実装に完全依存します)。



#### 注記

バージョン 5.0 では提供されないフィルターで、以前のバージョンの **JBoss Rules** に含まれているフィルターが複数あります。これらのフィルターは簡単に実装できます。実装方法は **JBoss Rules 4** のコードベースを参照してください。

フィルターを使用するには、**fireAllRules()** を呼び出す時に指定します。次の例では、**Test** という文字列で終わるルールのみ実行が許可されます。他のルールはフィルターによって除外されます。

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

### 3.3.4. アジェンダ

アジェンダは RETE 機能です。**working memory** でアクションが実行される時、ルールが完全一致すると実行可能になります。単一の **working memory** アクションによって複数のルールを実行可能にできます。ルールが完全一致すると、アクティベーションが作成されます。これはルールと、一致するファクトの両方を参照し、**Agenda** 上に置かれます。次に **Agenda** は、競合解決ストラテジを介してアクティベーションの順序を決定します。

**engine** は 2 つの段階を繰り返します。

1. 最初の段階は **ワーキングメモリアクション段階** と呼ばれます。ほとんどの作業はこの段階で行われ、**結果** (右側) または主要な Java アプリケーションプロセスのいずれかになります。結果が終了したり、主要な Java アプリケーションが **fireAllRules()** を呼び出すと、**engine** がアジェンダの第 2 段階へ切り替えられます。
2. 第 2 段階は **アジェンダ評価段階** と呼ばれます。この段階でシステムは実行するルールを検索します。何も検出されないと終了します。検出されたルールがある場合はそのルールを実行し、その後ワーキングメモリアクション段階へ切り替えます。

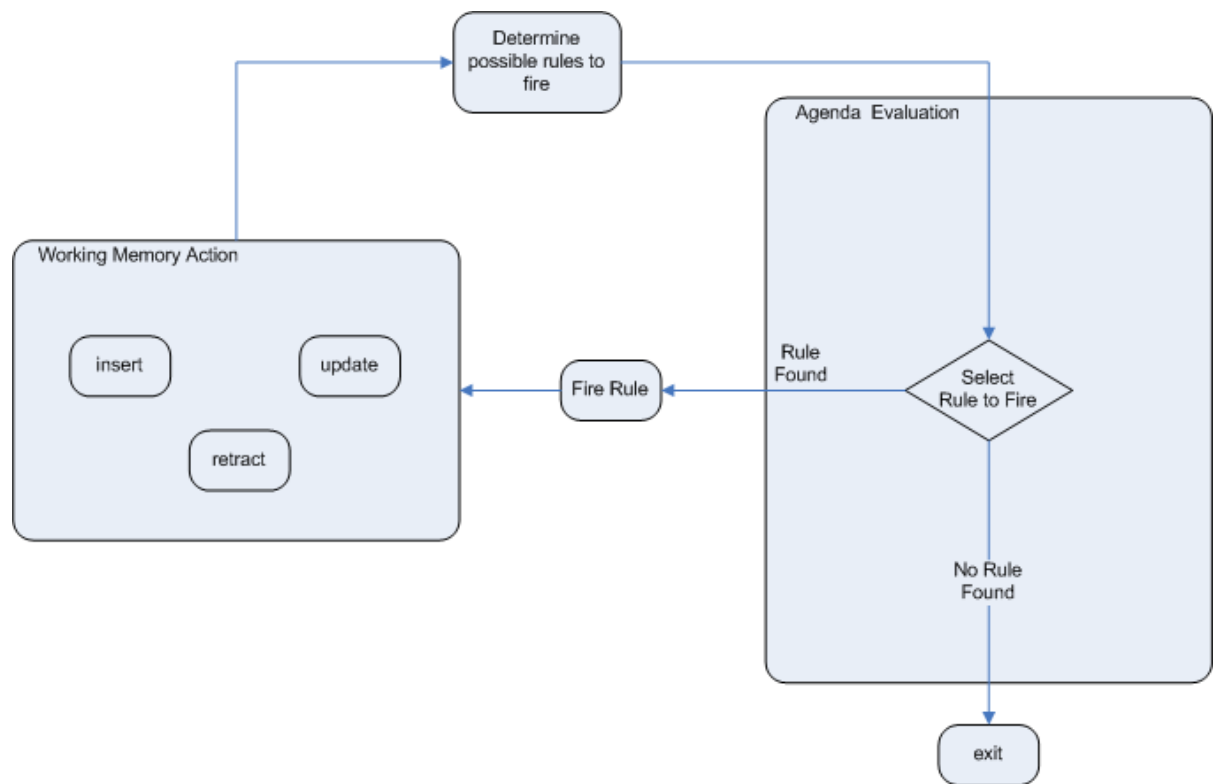
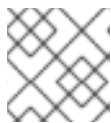


図3.17 2段階の実行

- プロセスは **agenda** が消去されるまで繰り返し実行され、消去された時点で時間制御が呼び出しアプリケーションへ返されます。



#### 注記

ワーキングメモリアクションの実行中、ルールは実行されません。

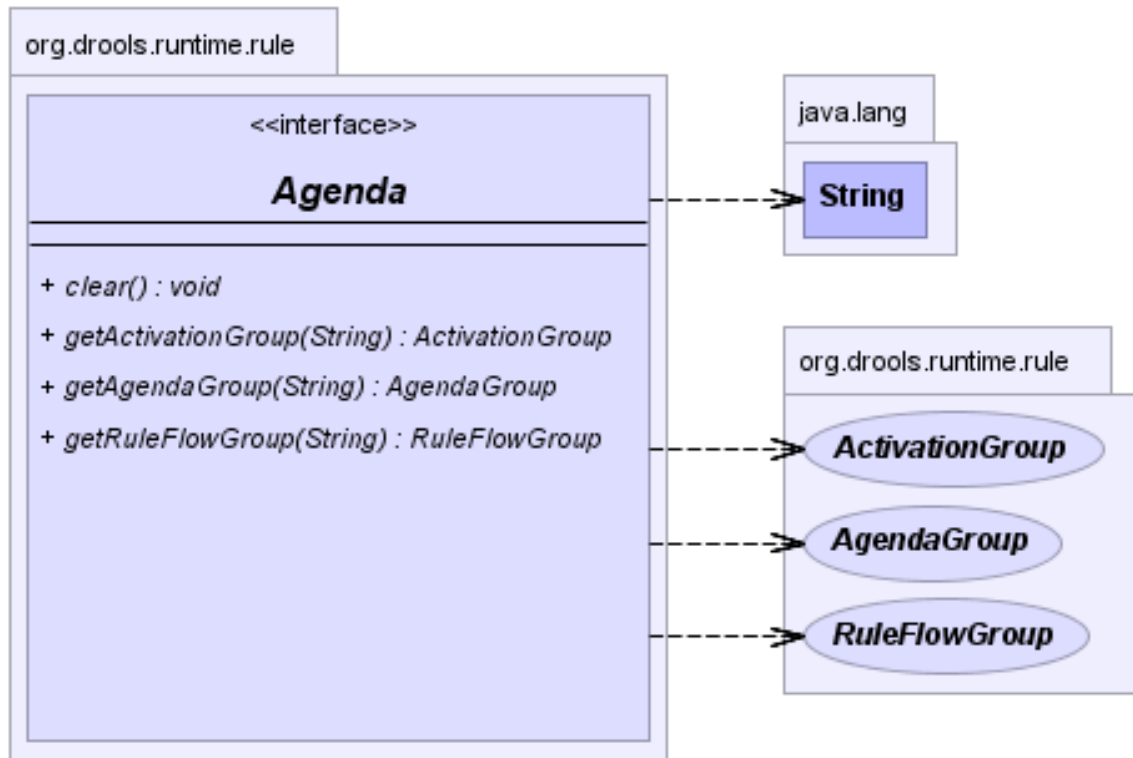


図3.18 アジェンダ

### 3.3.4.1. 競合の解決

**agenda** に複数のルールがある場合、競合解決ストラテジが必要となります。ルールの実行はワーキングメモリに影響を与えることがあるため、**rule engine** はルールが実行される順序を認識する必要があります (たとえば、**ruleA** を実行すると **ruleB** がアジェンダより削除される原因となります)。

**JBoss Rules** は次の 2 つの競合解決ストラテジーを使用します。

- Saliency
- LIFO (後入れ先出し)

*saliency* ストラテジーを使用すると特定ルールが他のルールよりも優先度が高いことを指定できます (大きい数字を割り当てます)。この場合、高い *saliency* を持つルールが優先されます。

LIFO ストラテジーは、割り当てられた **working memory** の **action counter** 値を基に優先度を決定します。同じアクションが同じ値を受け取る間に各ルールが作成されます (実行のセットが同じ優先度値を持つ場合、実行順序は任意になります)。



#### 重要

場合によっては回避不可能ですが、正しく動作させるために特定順序の実行に依存するルールは作成しないようにしてください。ルールを必須プロセスの手順として考慮しないでください。



## 注記

以前のバージョンの **JBoss Rules** はカスタムの競合解決ストラテジーをサポートしていました。この機能はバージョン 5 でも存在しますが、アプリケーションプログラミングインターフェースが公開されないようになりました。

### 3.3.4.2. AgendaGroup

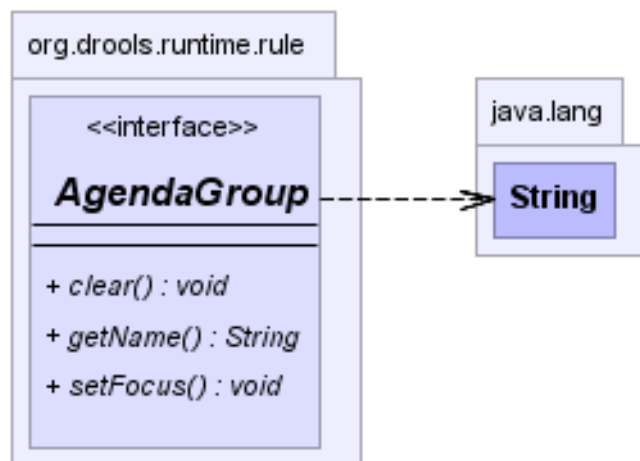


図3.19 AgendaGroup

アジェンダグループ (CLIPS の用語では「モジュール」と呼ばれます) を使用して **agenda** 上のアクティベーションを分割します。常に 1 つのグループのみが「フォーカス」を持つことができ、そのグループに属するアクティベーションのみを有効にできます。



## 注記

特定状況 (処理の段階など) に適用するルールのサブセットを 1 つ以上定義し、これらのルールセットがいつ適用されるかを制御するために、**Agenda groups** は最も一般的に使用されます。

ルール内部または **JBoss Rules** アプリケーションプログラミングインターフェースを介してフォーカスを設定します (*auto-focus* を使用するようルールを設定する方法もあります。この方法では、**agenda group** が一致するとフォーカスされます)。

**setFocus()** が呼び出されるたびに、**agenda group** がスタックにプッシュされます。フォーカスグループが空である場合、スタックから削除され、次のフォーカスグループ (この時点で一番上のグループ) を評価することが許可されます。



## 注記

**agenda group** はスタックの複数の場所に表示できます。

```
ksession.getAgenda().getAgendaGroup( "Group A" ).setFocus();
```

デフォルトの **agenda group** グループは **MAIN** と呼ばれます。これがスタックの最初のグループで、最初にフォーカスを持ちます。**agenda group** のないルールは自動的にこのグループに置かれます。

### 3.3.4.3. アクティベーショングループ

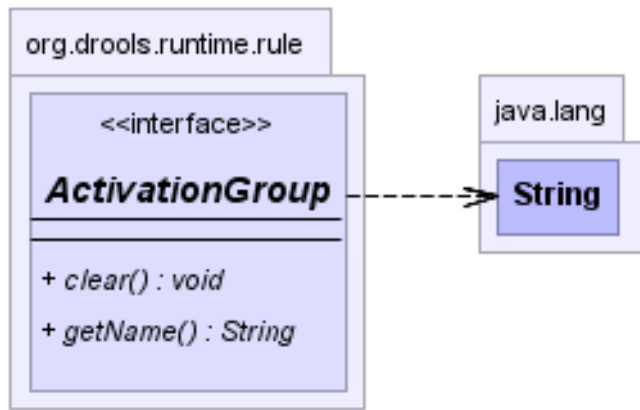
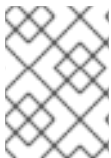


図3.20 ActivationGroup

**activation group** は `activation-group` ルール属性によってバインドされるルールのセットです。このグループでは 1 つのルールのみが実行できます。そのルールが実行した後、他のルールはすべてキャンセルされます。



#### 注記

任意のタイミングで **clear()** メソッドを呼び出し、アクティベーションが実行する前にすべてのアクティベーションをキャンセルします。

```
ksession.getAgenda().getActivationGroup( "Group B" ).clear();
```

### 3.3.5. イベントモデル

**event package** は **rule engine** イベントの 1 つに通知します。これを使用して、アプリケーションの主な部分やルールから、ロギングおよび監査のアクティビティを切り離します。

**KnowledgeRuntimeEventManager** インターフェースは **KnowledgeRuntime** クラスによって実装されます。このクラスは、**WorkingMemoryEventManager** と **ProcessEventManager** の 2 つのインターフェースを提供します。



#### 注記

本書では **WorkingMemoryEventManager** のみ取り上げます。

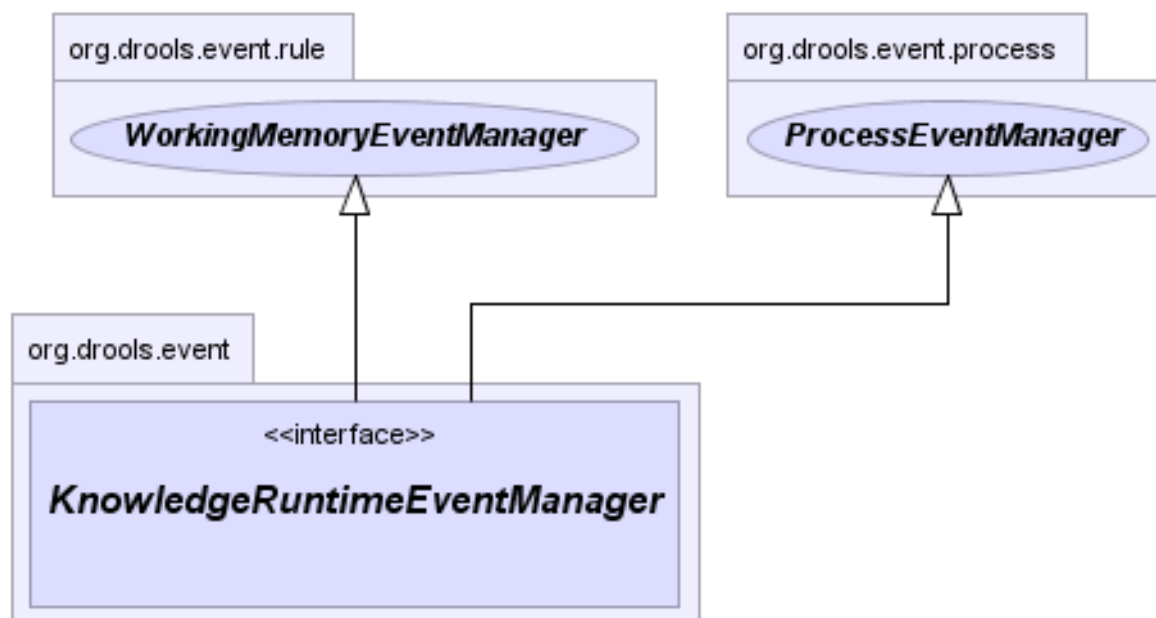


図3.21 KnowledgeRuntimeEventManager

**WorkingMemoryEventManager** を使用してリスナーを追加および削除します。リスナーを追加すると、**working memory** および **agenda** に影響するイベントを「リッスン」できます。

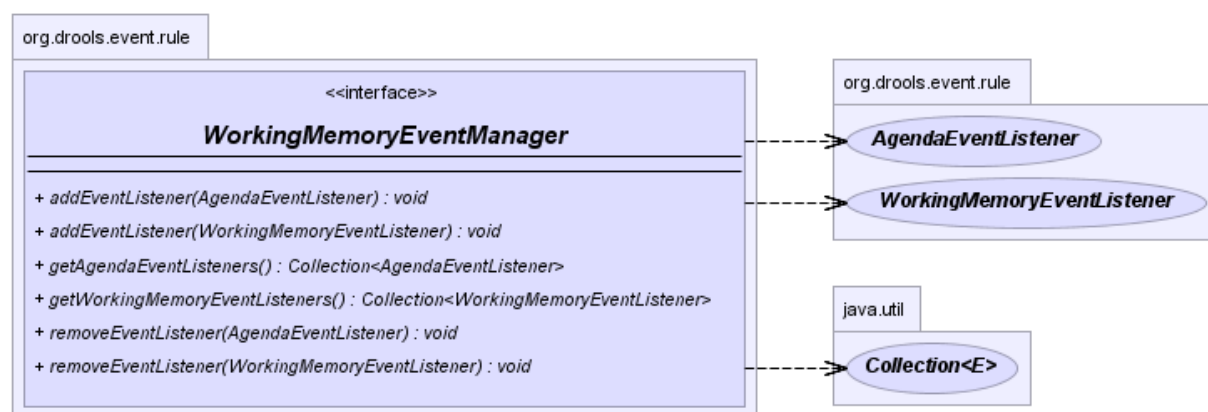


図3.22 WorkingMemoryEventManager

次のコードは、簡単な **agenda listener** を宣言し、セッションにアタッチする方法を表しています。アクティベーションが実行された後に、アクティベーションを出力します。

### 例3.28 AgendaEventListener の追加

```

ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterActivationFired(AfterActivationFiredEvent event) {
        super.afterActivationFired( event );
        System.out.println( event );
    }
});

```

**JBoss Rules** は、デバッグ出力ステートメントと共に各メソッドを実装する **DebugWorkingMemoryEventListener** および **DebugAgendaEventListener** と呼ばれる 2 つのク

ラスも提供します。すべての **working memory** イベントを出力するには、これらリスナーの 1 つを追加します。

### 例3.29 新しい KnowledgeBuilder の作成

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

イベント発生元より **KnowledgeRuntime** を読み出すには、**KnowledgeRuntimeEvent** インターフェイスを使用します。

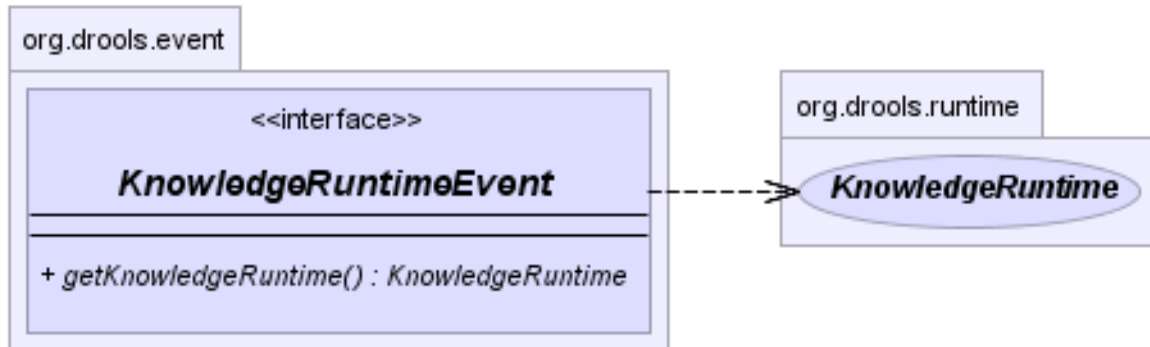


図3.23 KnowledgeRuntimeEvent

サポートされるイベントは次の通りです。

ActivationCreatedEvent	ActivationCancelledEvent
BeforeActivationFiredEvent	AfterActivationFiredEvent
AgendaGroupPushedEvent	AgendaGroupPoppedEvent
ObjectInsertEvent	ObjectRetractedEvent
ObjectUpdatedEvent	ProcessCompletedEvent
ProcessNodeLeftEvent	ProcessNodeTriggeredEvent
ProcessStartEvent	

### 3.3.6. KnowledgeRuntimeLogger

アプリケーションが実行されると、毎回 **KnowledgeRuntimeLogger** は **JBoss Rules** の **event system** を使用して監査ログを作成します。**JBoss Rules IDE** の **Audit Viewer** などのツールを使用してこのログを調査します。

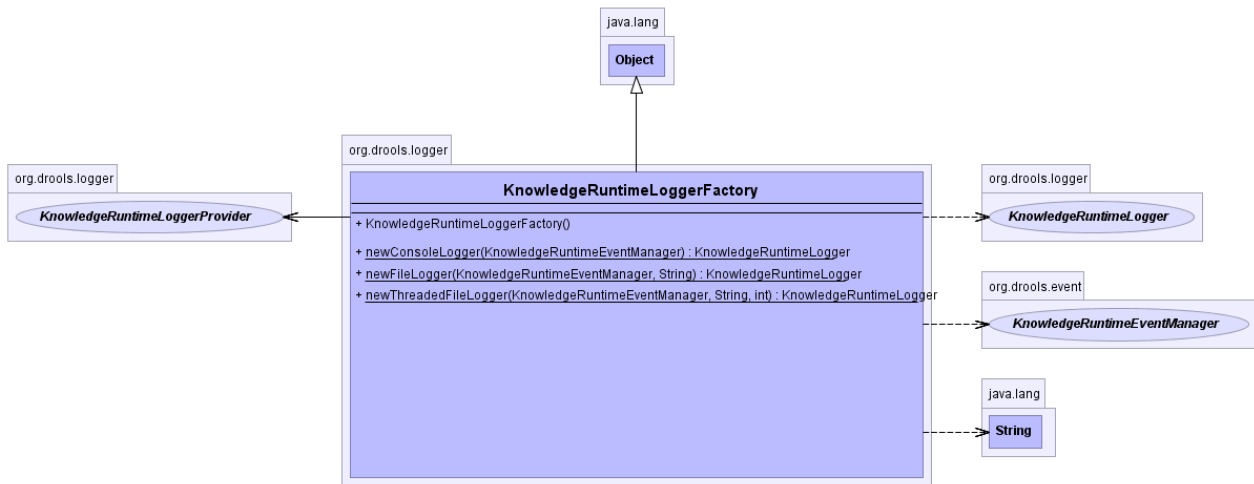


図3.24 KnowledgeRuntimeLoggerFactory

## 例3.30 FileLogger

```

KnowledgeRuntimeLogger logger =
    KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
        "logdir/mylogfile");
...
logger.close();

```

`newFileLogger()` メソッドを使用して、自動的にファイル拡張子 `.log` をファイルへ追加します。

## 3.3.7. StatelessKnowledgeSession

**StatelessKnowledgeSession** は **StatefulKnowledgeSession** をラッピングし、決定サービスタイプのシナリオに関連して使用されます。この存在により、`dispose()` の呼び出しが軽減されます。

ステートレスセッションの使用時に、繰り返し挿入を実行したり Java コードから `fireAllRules()` メソッドを呼び出したりすることはできません。`execute()` メソッドは内部で **StatefulKnowledgeSession** をインスタンス化し、ユーザーデータをすべて追加してユーザーコマンドを実行します。その後、`fireAllRules()` および `dispose()` メソッドを呼び出します。

通常、**BatchExecution** コマンドよりこのクラスを使用します (**CommandExecutor** インターフェースによってサポートされます)。しかし、2つの *簡便性 (convenience)* メソッドも提供されています。これらのメソッドは、簡単なオブジェクト挿入が必要な場合のみ使用します (**CommandExecutor** および **BatchExecution** は独自の項で詳細に説明されています)。



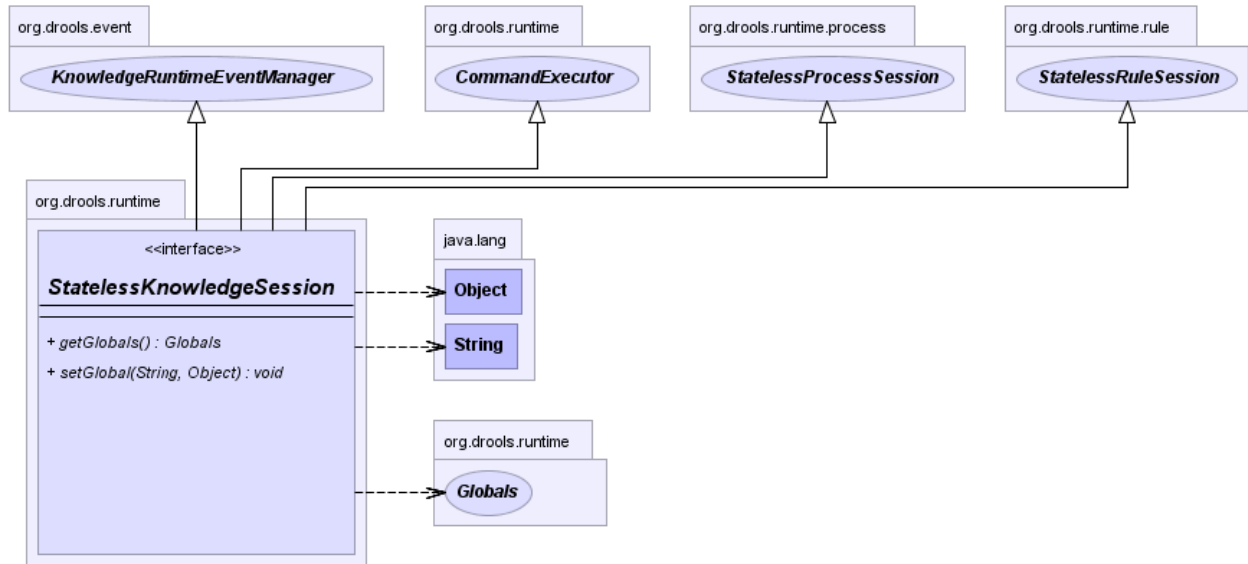


図3.25 StatelessKnowledgeSession

次の例は、Java オブジェクトのコレクションを実行するため、**Convenience API** を使用して **stateless session** を実行することを表しています。コレクションを繰り返し処理し、各要素を順に挿入します。

#### 例3.31 コレクションを用いた簡単な StatelessKnowledgeSession の実行

```

KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileResource( fileName ),
ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
} else {
    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
    StatelessKnowledgeSession ksession =
kbase.newStatelessKnowledgeSession();
    ksession.execute( collection );
}
  
```

単一コマンドとして同じことを実行するには、次のコードを使用します。

#### 例3.32 InsertElements コマンドを用いた簡単な StatelessKnowledgeSession の実行

```

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

コレクションの繰り返し処理や要素の挿入を行わずにコレクション自体を挿入するには、**CommandFactory.newInsert(collection)** を使用します。

**CommandFactory** にはサポートされるコマンドの詳細が含まれています。これらのコマンドのいずれかをマーシャリングするには **XStream** および **BatchExecutionHelper** を使用します。また、**BatchExecutionHelper** を使用して、使用される XML 形式の詳細について学びます。JBoss

**Rules Pipeline** を使用して、自動的に **BatchExecution** および **ExecutionResults** をマーシャリングします。

**StatelessKnowledgeSession** はさまざまなやり方でグローバルをスコープ指定できるようにします。最初はコマンドではない方法です。コマンドは特定の実行呼び出しへスコープ指定されます (グローバルは 3 つの方法で解決されます)。

- **StatelessKnowledgeSession** の **getGlobals()** メソッドは **Globals** インスタンスを返します。名前の通り、このメソッドはセッションのグローバルへのアクセスを提供します。セッションのグローバルは **すべての** 実行呼び出しによって共有されます。



#### 警告

実行呼び出しは異なるスレッドで同時に実行できるため、**可変グローバル** を扱う場合は注意が必要です。

### 例3.33 セッションスコープグローバル

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

- 委譲を使用してグローバル解決を実行する方法もあります。グローバルに値を割り当てると (**setGlobal(String, Object)** を使用)、値は内部コレクションに格納されます。これは、識別子を値へマッピングすることが目的です。これらの識別子は提供される委譲よりも優先されます。識別子が見つからない場合のみ委譲グローバル (存在する場合) が使用されます。
- グローバルを解決する 3 つ目の方法は、**実行スコープグローバル** を使用することです。この場合、グローバルを設定するコマンドは **CommandExecutor** へ渡されます。

また、**CommandExecutor** インターフェースは out パラメーターを用いてデータをエクスポートする機能も提供します。挿入されたファクト、グローバル、およびクエリの結果はすべて返すことが可能です。

### 例3.34 out 識別子

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

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

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

### 3.3.7.1. シーケンシャルモード

Rete はオブジェクトを徐々にアサートでき、ルールも追加および削除できるステートフルセッションを提供します。ただし、ステートフルセッションでは最初のデータセットが提供された後に、データをアサートしたり変更することはできず、ルールを追加したり削除することもできません。この場合、ルールを再評価する必要はなく、**engine** は簡素化された方法で稼働できます。以下の手順に従ってください。

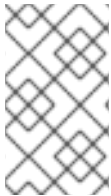
1. ルールセットの **salience** と **position** によりルールの順番を決定します (**rule terminal** ノードの **sequence** 属性を設定します)。
2. 可能なルールアクティベーションごとに 1 つの要素を持つアレイを作成します。要素のポジションが実行順序を表します。
3. **right-input** オブジェクトメモリー以外の ノードメモリー をすべてオフにします。
4. **Left Input Adapter Node** 伝播の接続を切断し、コマンドオブジェクトがオブジェクトとノードを参照できるようにします。後で実行するために、このコマンドオブジェクトは **working memory** のリストに追加されます。
5. すべてのオブジェクトをアサートします。アサートが実行され、**right-input** ノードメモリーにデータが投入されると、コマンドリストを確認し、各項目を順番に実行します。
6. ルールに対して決定されたシーケンス番号に従って、結果となるすべてのアクティベーションをアレイに格納します。繰り返し処理の範囲を縮小するため、最初と最後に投入された要素を記録します。
7. アクティベーションのアレイを繰り返し処理し、投入された要素を順番に実行します。
8. 許可される最大ルール実行数が存在する場合、アレイのルールをすべて実行するため、ネットワーク評価を早期に終了します。

#### 注記

タプルの作成、オブジェクトの追加、およびタプルの伝播は **LeftInputAdapterNode** によって実行されないようになりました。代わりに、コマンドオブジェクトが作成され、**working memory** のリストに追加されるようになりました。このオブジェクトには、**LeftInputAdapterNode** と伝播されたオブジェクト両方への参照が含まれます。これにより、挿入時に **left-input** 伝播が発生しないようにするため、**left-input** で結合を実行しようとする **right-input** 伝播はありません (よって、**left-input** メモリーが不必要になります)。

left-input タプルメモリを含むノードのメモリはほぼすべてオフになりますが、right-input オブジェクトメモリは除外されます。そのため、挿入の伝播を記憶するノードは right-input オブジェクトメモリのみになります。

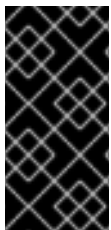
すべてのアサートが終了し、その結果 right-input メモリがすべて投入されたら、**LeftInputAdapterNode** コマンドオブジェクトを順に呼び出し、**LeftInputAdapterNode** コマンドオブジェクトのリストを繰り返し処理します。これらのコマンドオブジェクトはネットワークへ渡され、right-input オブジェクトと結合しようとはしますが、right-input メモリへアサートまたは伝播されるオブジェクトはこれ以上ないため、left-input には記録されません。



#### 注記

タプルをスケジュールする優先度キューを持つ **agenda** はなくなりました。代わりに、ルールの数に対する簡単なアレイが存在します。**RuleTerminalNode** のシーケンス番号はアクティベーションを格納するアレイ内の要素を表します。

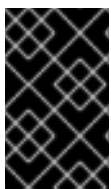
コマンドオブジェクトがすべて処理されると、各要素を順番にチェックし、アクティベーションを実行して (存在する場合) アレイを繰り返し処理します。



#### 重要

パフォーマンスを向上するため、アレイの最初と最後に投入されたセルを記憶するようにしてください。各 **RuleTerminalNode** にシーケンス番号が割り当てられ、ネットワークが構築されます。この番号は、salience 番号とネットワークに追加された順序に基づいています。

オブジェクトを迅速に取り消しできるようにするため、right-input ノードメモリは通常ハッシュマップになります。この場合、オブジェクトの取り消しがなく、オブジェクト値がインデックス化されないため、リストを使用します。



#### 重要

インデックス化されたオブジェクトの多くは、ハッシュマップによってパフォーマンスが改善されます。しかし、オブジェクトタイプに少数のインスタンスしかない場合、インデックス化の利点はないため、リストが使用されます。

シーケンシャルモードは、ステートレスセッションでのみ使用可能で、デフォルトでは無効になっています。有効にするには、**RuleBaseConfiguration.setSequential(true)** を呼び出すか、ルールベース設定の drools.sequential プロパティを **true** に設定します。



#### 注記

**SequentialAgenda.DYNAMIC** を用いて **setSequentialAgenda** を呼び出し、シーケンシャルモードが動的アジェンダヘフォールバックするようにします。また、drools.sequential.agenda プロパティを **sequential** または **dynamic** に設定することもできます。

### 3.3.8. コマンドと CommandExecutor

**JBoss Rules** はステートフルセッションおよびステートレスセッションを使用します。ステートフルセッションは、徐々に 繰り返し 作業できる標準的な **working memory** を使用します。ステートレスセッションは、提供されたデータセットを用いて **working memory** を一度だけ実行します。結果がい

くつか返される可能性があります。対話が繰り返し行われないようにするためセッションは最後に破棄されます。ステートレスセッションは、任意の結果を返す関数として **rule engine** を処理する方法であると考えてください。

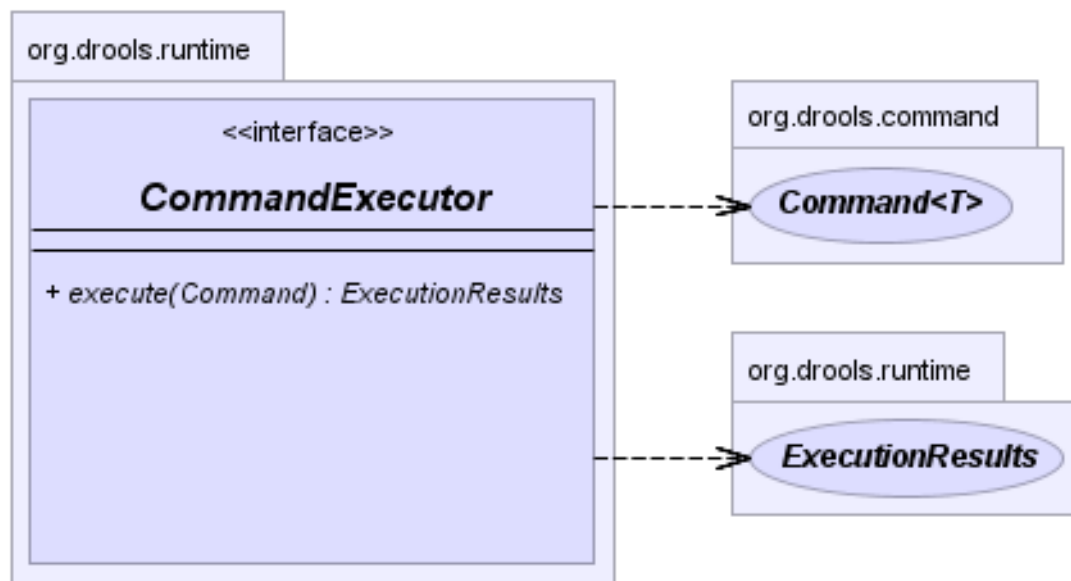


図3.26 CommandExecutor

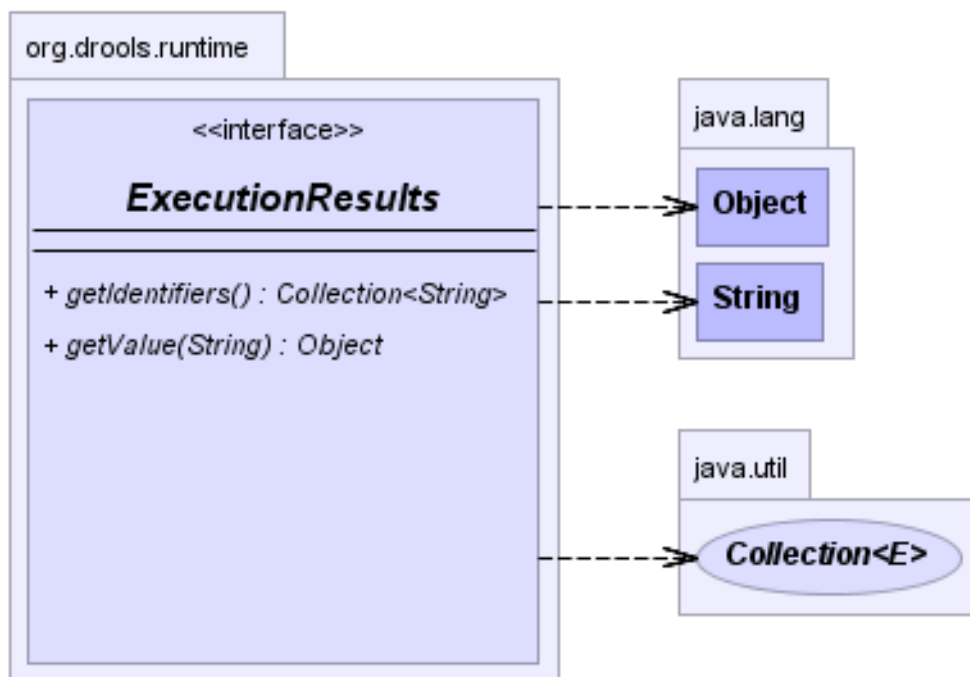


図3.27 ExecutionResults

**CommandFactory** はコマンドをステートフルおよびステートレスセッションで実行できるようにします (ステートレスナレッジセッションは破棄される前に **fireAllRules()** を最後に実行することが唯一の違いとなります)。現在サポートされているコマンドは次の通りです。

FireAllRules	GetGlobal
SetGlobal	InsertObject

InsertElements	クエリ
StartProcess	BatchExecution

名前の通り、**InsertObject** は任意の out 識別子を用いて単一オブジェクトを挿入します。**InsertElements** は繰り返し処理が可能なオブジェクトを確認し、各要素を挿入します。その結果、ステートレスナレッジセッションへオブジェクトを挿入するだけでなく、プロセスを開始したりクエリを実行したりして任意の順番でこれを実行できるようになります。

### 例3.35 insert コマンド

```
StatelessKnowledgeSession ksession =
    kbase.newStatelessKnowledgeSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.newInsert( new Cheese( "stilton" ),
        "stilton_id" ) );
Stilton stilton = bresults.getValue( "stilton_id" );
```

execute メソッドは常に **ExecutionResults** インスタンスを返します。これにより、上記の stilton\_id などの out 識別子が指定されていると、すべてのコマンドの結果にアクセスできます。

### 例3.36 InsertElements コマンド

```
StatelessKnowledgeSession ksession =
    kbase.newStatelessKnowledgeSession();
Command cmd = CommandFactory.newInsertElements(
    Arrays.asList(new Object[] {
        new Cheese("stilton"), new Cheese("brie"), new
        Cheese("cheddar")}
    ));
ExecutionResults bresults = ksession.execute( cmd );
```

## 重要

このメソッドは単一のコマンドのみを許可します。**BatchExecution** は命令リストを取る複合コマンドで、これらの命令を順番に繰り返し処理し、実行します。そのため、オブジェクトをいくつか挿入してプロセスを開始し、**fireAllRules** を呼び出して単一の **execute(...)** 呼び出しでクエリを実行できるため、大変強力なコマンドになります。

ステートレスナレッジセッションは処理を終えると **fireAllRules()** メソッドを自動的に実行しますが、**FireAllRules** コマンドも許可されます。このコマンドを使用すると、最後に自動実行が無効になります。これは手動のオーバーライドです。

コマンドは out 識別子をサポートします。設定されるコマンドは、返される **ExecutionResults** インスタンスにその結果を追加します。次の例はこの仕組みを表しています。

**例3.37 BatchExecution コマンド**

```

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

```

この例では、複数のコマンドが実行され、2つのコマンドが **ExecutionResults** にデータを投入します。query コマンドはデフォルトではクエリ名と同じ識別子を使用しますが、異なる識別子へマッピングすることも可能です。

カスタマイズされた **XStream** マーシャラーを **JBoss Rules Pipeline** と共に使用すると XML スクリプトを提供できるため、サービスに最適です。以下は、**BatchExecution** と **ExecutionResults** 向けの2つの簡単な XML の例になります。

**例3.38 簡単な BatchExecution XML**

```

<batch-execution>
  <insert out-identifier='outStilton'>
    <org.drools.Cheese>
      <type>stilton</type>
      <price>25</price>
      <oldPrice>0</oldPrice>
    </org.drools.Cheese>
  </insert>
</batch-execution>

```

**例3.39 簡単な ExecutionResults XML**

```

<execution-results>
  <result identifier='outStilton'>
    <org.drools.Cheese>
      <type>stilton</type>
      <oldPrice>25</oldPrice>
      <price>30</price>
    </org.drools.Cheese>
  </result>
</execution-results>

```

パイプラインにより、複数の **stage** オブジェクトを使用できます。これらを組み合わせると、より簡単にデータをセッション内やセッション外で移動できます。

**CommandExecutor** インターフェースを実装する **stage** があります。これを使用して、パイプラインスクリプトをステートフルまたはステートレスセッションにします。次のように設定を行います。

### 例3.40 CommandExecutor のパイプライン

```

Action executeResultHandler = PipelineFactory.newExecuteResultHandler();

Action assignResult = PipelineFactory.newAssignObjectAsResult();

assignResult.setReceiver( executeResultHandler );

Transformer outTransformer =
    PipelineFactory.newXStreamToXmlTransformer(
        BatchExecutionHelper.newXStreamMarshaller() );
outTransformer.setReceiver( assignResult );

KnowledgeRuntimeCommand cmdExecution =
    PipelineFactory.newCommandExecutor();
batchExecution.setReceiver( cmdExecution );

Transformer inTransformer =
    PipelineFactory.newXStreamFromXmlTransformer(
        BatchExecutionHelper.newXStreamMarshaller() );
inTransformer.setReceiver( batchExecution );

Pipeline pipeline =
    PipelineFactory.newStatelessKnowledgeSessionPipeline( ksession );
pipeline.setReceiver( inTransformer );

```

**BatchExecutionHelper** を使用して、**command** のカスタムコンバーターと新しい **BatchExecutor** ステージを持つ、特別に設定された **XStream** を提供します。

**pipeline** を使用するには、**ResultHandler** の実装を提供します。これは、**pipeline** が **ExecuteResultHandler** ステージを実行する時に呼び出されます。

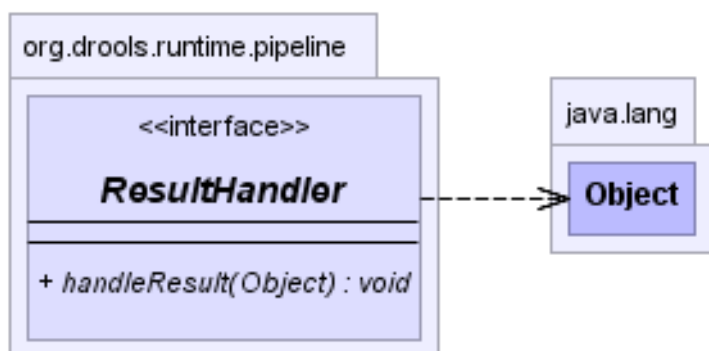


図3.28 パイプライン **ResultHandler**

### 例3.41 簡単なパイプライン **ResultHandler**

```

public static class ResultHandlerImpl implements ResultHandler {
    Object object;
}

```



```

    public void handleResult(Object object) {
        this.object = object;
    }

    public Object getObject() {
        return this.object;
    }
}

```

### 例3.42 パイプラインの使用

```

ResultHandler resultHandler = new ResultHandlerImpl();
pipeline.insert( inXml, resultHandler );

```

ここで、作成した **BatchExecution** を使用してオブジェクトを挿入し、クエリを実行します。以下の **pipeline** の例では XML 表現が使用されます。パラメーターはクエリに追加されています。

### 例3.43 XML ヘマーシャリングされた BatchExecution

```

<batch-execution>
  <insert out-identifier="stilton">
    <org.drools.Cheese>
      <type>stilton</type>
      <price>1</price>
      <oldPrice>0</oldPrice>
    </org.drools.Cheese>
  </insert>
  <query out-identifier='cheeses2' name='cheesesWithParams'>
    <string>stilton</string>
    <string>cheddar</string>
  </query>
</batch-execution>

```

**CommandExecutor** は **ExecutionResults** を返し、**pipeline** コードスニペットによって処理されます。

以下と似ている出力が <batch-execution> XML の例で生成されます。

### 例3.44 XML ヘマーシャリングされた ExecutionResults

```

<execution-results>
  <result identifier="stilton">
    <org.drools.Cheese>
      <type>stilton</type>
      <price>2</price>
    </org.drools.Cheese>
  </result>
  <result identifier='cheeses2'>
    <query-results>

```

```

<identifiers>
  <identifier>cheese</identifier>
</identifiers>
<row>
  <org.drools.Cheese>
    <type>cheddar</type>
    <price>2</price>
    <oldPrice>0</oldPrice>
  </org.drools.Cheese>
</row>
<row>
  <org.drools.Cheese>
    <type>cheddar</type>
    <price>1</price>
    <oldPrice>0</oldPrice>
  </org.drools.Cheese>
</row>
</query-results>
</result>
</execution-results>

```

**BatchExecutionHelper** は事前設定された **XStream** を提供します。これを使用して一括実行のマージングをサポートします (結果となる XML は上記の通り、メッセージ形式として使用できます)。 **Command Factory** よりサポートされるコマンドのみに事前設定されたコンバーターが存在します。ユーザーオブジェクトに他のコンバーターを追加することもできます (特にサービスが関与する場合、ステートレスまたはステートフルナレッジセッションのスクリプティングに大変便利です)。

現在、検証をサポートする XML スキーマはありません。基本形式はここで説明されていますが、**drools-transformer-xstream** モジュールには **drools-transformer-xstream** と呼ばれる単体テストがあります。ルート要素は `<batch-execution>` と命名され、任意の数の `command` 要素を含めることが可能です。

#### 例3.45 ルート XML 要素

```

<batch-execution>
...
</batch-execution>

```

これには、コマンドを表現する要素のリストが含まれます。サポートされるコマンドは **Command Factory** によって提供されるコマンドに限定されます。最も基本的なものが `<insert>` 要素で、オブジェクトを挿入します。insert 要素の内容はユーザーオブジェクトで、**XStream** によって決まります。

#### 例3.46 Insert

```

<batch-execution>
  <insert>
    ...<!-- any user object -->
  </insert>
</batch-execution>

```

**insert** 要素は `out-identifier` と呼ばれる属性を特徴とします。これは、挿入されたオブジェクトが結果ペイロードの一部として返されることを要求します。

#### 例3.47 out 識別子コマンドを用いた挿入

```
<batch-execution>
  <insert out-identifier='userVar'>
    ...
  </insert>
</batch-execution>
```

`<insert-elements>` 要素を使用してオブジェクトのコレクションを挿入することも可能です。このコマンドは **out-identifier** をサポートしません (`org.domain.UserClass` は `XStream` によるシリアル化が可能な例示のユーザーオブジェクトです)。

#### 例3.48 Insert Elements コマンド

```
<batch-execution>
  <insert-elements>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
  </insert-elements>
</batch-execution>
```

名前の通り、`<set-global>` 要素はセッションのグローバルを設定するために使用されます。

#### 例3.49 Insert Elements コマンド

```
<batch-execution>
  <set-global identifier='userVar'>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
  </set-global>
</batch-execution>
```

`<set-global>` は任意の属性である `out` と `out-identifier` もサポートします。identifier 属性からの名前を使用して、ブール値 `out` の真値はグローバルを `<batch-execution-results>` ペイロードに追加します。`out-identifier` は `out` のように挙動しますが、`<batch-execution-results>` ペイロードで使用される識別子をオーバーライドできます。

#### 例3.50 Set Global コマンド

```

<batch-execution>
  <set-global identifier='userVar1' out='true'>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
  </set-global>
  <set-global identifier='userVar2' out-
identifier='alternativeUserVar2'>
    <org.domain.UserClass>
      ...
    </org.domain.UserClass>
  </set-global>
</batch-execution>

```

また、out-identifier 属性のみがあり、内容のない <get-global> 要素も存在します (<get-global> の唯一の目的は値を読み出すことであるため、out 属性は必要ありません)。

### 例3.51 Get Global コマンド

```

<batch-execution>
  <get-global identifier='userVar1' />
  <get-global identifier='userVar2' out-
identifier='alternativeUserVar2' />
</batch-execution>

```

out 属性は、特定のインスタンスを結果ペイロードとして返すためだけに使用できます。実際のクエリの実行には他の方法が必要になります。クエリはパラメーターの有無に関係なくサポートされます。name 属性は呼び出されるクエリの名前で、out-identifier は <execution-results> ペイロードのクエリ結果に使用される識別子になります。

### 例3.52 Query コマンド

```

<batch-execution>
  <query out-identifier='cheeses' name='cheeses' />
  <query out-identifier='cheeses2' name='cheesesWithParams'>
    <string>stilton</string>
    <string>cheddar</string>
  </query>
</batch-execution>

```



#### 注記

<start-process> コマンドは任意のパラメーターも許可します。

### 例3.53 Start Process コマンド

```

<batch-execution>
  <startProcess processId='org.drools.actions'>
    <parameter identifier='person'>

```

```

        <org.drools.TestVariable>
            <name>John Doe</name>
        </org.drools.TestVariable>
    </parameter>
</startProcess>
</batch-execution>

```

#### 例3.54 Signal Event コマンド

```

<signal-event process-instance-id='1' event-type='MyEvent'>
    <string>MyValue</string>
</signal-event>

```

#### 例3.55 Complete Work Item コマンド

```

<complete-work-item id='" + workItem.getId() + "' >
    <result identifier='Result'>
        <string>SomeOtherString</string>
    </result>
</complete-work-item>

```

#### 例3.56 About Work Item コマンド

```

<abort-work-item id='21' />

```



#### 注記

コマンドは徐々に追加される予定です。

### 3.3.9. マーシャリング

**MarshalerFactory** を使用して **stateful knowledge sessions** のマーシャリングおよびアンマーシャリングを行います。

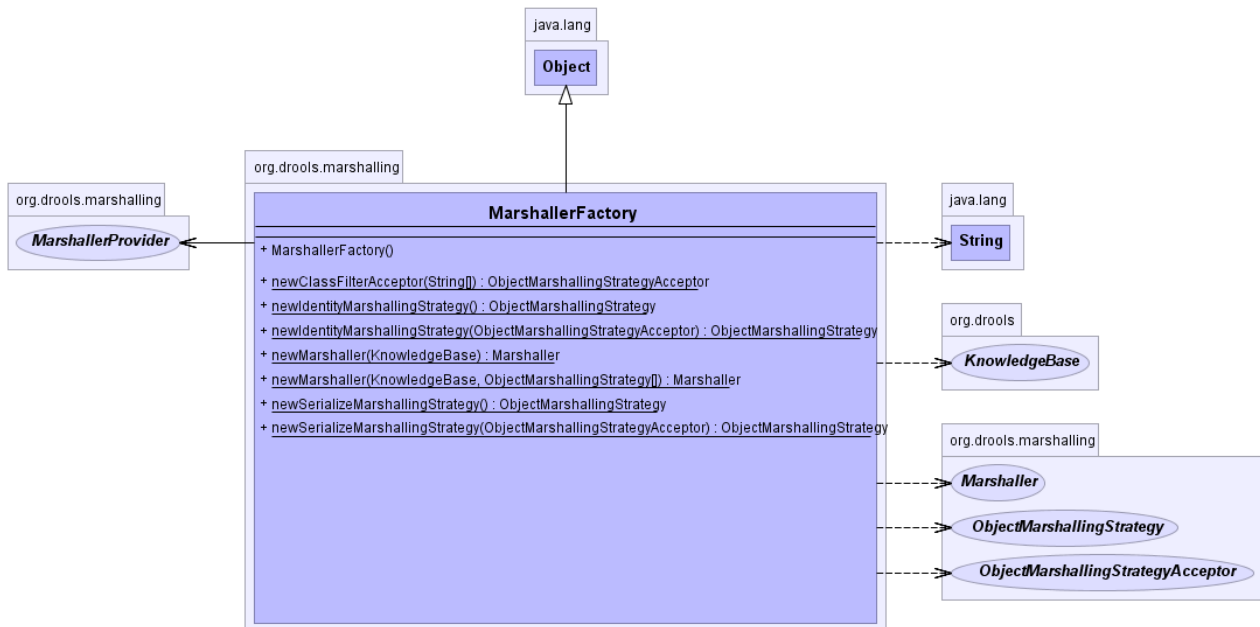


図3.29 MarshalerFactory

以下は **MarshalerFactory** を使用する最も簡単な方法になります。

#### 例3.57 簡単なマーシャラーの例

```
// ksession is the StatefulKnowledgeSession
// kbase is the KnowledgeBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = MarshallerFactory.newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

参照されたユーザーデータをマーシャリングする場合には柔軟性が必要となります。柔軟性を提供するため、**ObjectMarshallingStrategy** インターフェースが追加されています。このインターフェースの2つの実装が提供され、ユーザーは独自の実装を追加できます。提供されている2つの実装は次の通りです。

- IdentityMarshalingStrategy
- SerializeMarshalingStrategy

デフォルトは **SerializeMarshalingStrategy** です (上記の例で使用されています)。これは、ユーザーインスタンス上で **Serializable** または **Externalizable** メソッドを呼び出します。

反対に、**IdentityMarshalingStrategy** は ID がストリームに書き込まれる間に、各ユーザーオブジェクトに対して整数識別番号を作成し、マップに格納します。アンマーシャリングが行われている間、**IdentityMarshalingStrategy** マップへアクセスし、インスタンスを読み出します (そのため、**IdentityMarshalingStrategy** が使用されると、Marshaller インスタンスが生存している間はステートフルになり、識別子を作成し、マーシャリングを行いたい各オブジェクトへの参照を保持します)。**IdentityMarshalingStrategy** に使用するコードは次の通りです。

#### 例3.58 IdentityMarshallingStrategy

```

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectMarshallingStrategy oms =
MarshallerFactory.newIdentityMarshallingStrategy()
Marshaller marshaller =
    MarshallerFactory.newMarshaller( kbase, new
ObjectMarshallingStrategy[] { oms } );
marshaller.marshall( baos, ksession );
baos.close();

```

柔軟性を向上するため、**ObjectMarshalingStrategyAcceptor** インターフェースも提供されています。各 **Object Marshaling Strategy** にはこのインターフェースが含まれています。マーシャラーは一連のストラテジーを持ち、ユーザーオブジェクトへ読み書きしようとする、ストラテジーを繰り返し、ユーザーオブジェクトをマーシャリングする責任を受け入れるかどうか「依頼」します。提供される実装の1つは **ClassFilterAcceptor** と呼ばれます。これは、文字列とワイルドカードを使用してクラス名を照合できるようにします。デフォルトは `*.*` であるため、上記の例では使用される **IdentityMarshalingStrategy** にはデフォルトの `*.*` が含まれます。

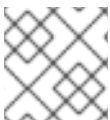
各クラスパー1をシリアライズするには (ID ルックアップが使用されます)、以下を実行します。

### 例3.59 アクセプターを用いた IdentityMarshalingStrategy

```

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

```



#### 注記

許可を確認する順序は提供されるアレイの自然な順序になります。

## 第4章 ルール言語

### 4.1. 概要

JBoss Rules は「ネイティブ」ルール言語を持ちます。この形式は句読点の扱いが非常に簡単で、問題ドメインへ言語を適合できるようにする「エクspander」を用いて自然言語およびドメイン固有の言語をサポートします。本章ではこのネイティブルール形式について説明します。

構文を示すために使用される図は *railroad* 図と呼ばれ、言語的な用語のフローチャートに似ています。興味がある方は **DRL.g** 内のルール言語に対する Antlr3 グラマーを参照することもできますが、必須ではありません。Rule Workbench を使用する場合、コンテンツアシスタントはルール構造の大部分に対応します。例えば、「ru」を入力し、ctrl キーとスペースキーを同時に押すと、ルール構造が構築されます。

#### 4.1.1. ルールファイル

ルールファイルとは通常、**.drl** 拡張子を持つファイルです。DRL ファイルには複数のルール、クエリ、および関数を含めることができ、またルールやクエリによって割り当てられ使用されるインポート、グローバルおよび属性などの一部のリソース宣言も含めることができます。複数のルールファイルにまたがってルールを分散することもできます。この場合、拡張子 **.rule** が推奨されますが必須ではありません。ファイルにまたがってルールを分散すると、大量のルールを管理する場合に役立ちます。DRL ファイルは簡単なテキストファイルです。

ルールファイルの全体的な構造は以下のようになります。

```
package package-name

imports

globals

functions

queries

rules
```

要素が宣言される順番は重要ではありません。ただし、パッケージ名は例外で、宣言される場合はルールファイルの最初の要素でなければなりません。すべての要素は任意であるため、必要な要素のみを使用できます。以降の項でこれらの要素について 1 つずつ説明します。

#### 4.1.2. ルールの構造

ルールの大まかな構造は次のようになります。

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



ほとんどの部分で句読点は必要ありません。"name" の二重引用符や改行も任意です。属性は簡単で (常に任意です)、ルールが取るべき挙動について示します。LHS はルールの条件的な部分で、特定の構文に従います (説明は以下を参照)。RHS は基本的にダイアレクト固有のセマンティックコードを実行できるようにするブロックです。

空白文字は重要ではありませんが、ドメイン固有の言語の場合のみ重要となることに注意してください。ドメイン固有の言語を使用する場合、各行は後続の行の前に処理され、空白文字はドメイン言語では重要であることがあります。

## 4.2. キーワード

本項では ハードキーワードと ソフトキーワードの概念について説明します。

ハードキーワードは予約されたキーワードであるため、ルールテキストで使用されるドメインオブジェクト、プロパティ、メソッド、関数またはその他の要素を命名する時に使用できません。

ハードキーワードの一覧は次の通りです。ルールを書くときにこれらのキーワードを識別子として使用しないでください。

true	false	accumulate
collect	from	null
over	then	when

ソフトキーワードは直接のコンテキストでのみ認識されるため、これらの言葉は他の場所でも使用できます。

ソフトキーワードの一覧は次の通りです。

lock-on-active	date-effective	date-expires
no-loop	auto-focus	activation-group
agenda-group	ruleflow-group	entry-point
duration	package	import
dialect	salience	enabled
attributes	rule	extend
template	query	declare
function	global	eval
not	in	or
and	exists	forall

action	reverse	result
end	init	

ハードキーワードおよびソフトキーワードは、**notSomething()** や **accumulateSomething()** のようにメソッド名に使用することが可能です。

DRL 言語では、ルールテキスト上でハードキーワードをエスケープすることも可能です。この機能により、キーワードの「衝突」を懸念せずに既存のドメインオブジェクトを使用することができます。言葉をエスケープするには、次のようにアクサングラフで文字を囲みます。

```
Holiday( `when` == "july" )
```

エスケープは、左側と右側のコードブロックにあるコード式内以外の場所であればどこでも使用できます。適切な使用方法の例は次の通りです。

```
rule "validate holiday by eval"
dialect "mvel"
when
    h1 : Holiday( )
    eval( h1.when == "july" )
then
    System.out.println(h1.name + ":" + h1.when);
end
```

```
rule "validate holiday"
dialect "mvel"
when
    h1 : Holiday( `when` == "july" )
then
    System.out.println(h1.name + ":" + h1.when);
end
```

### 4.3. コメント

コメントは **rule engine** によって無視されるテキストの部分のことです。セマンティックコードブロック内にコメントがある場合以外は、コメントが検出されると、ルールの右側のように無視されます。コメントには *単一行コメント* と *複数行コメント* の 2 種類があります。

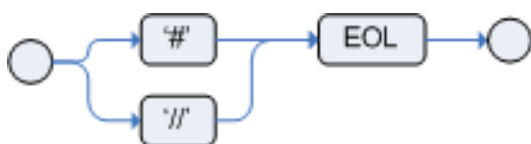


図4.1 単一行コメント

# または // のいずれかを使用して単一行コメントを作成します。パーサーは行のコメント記号後の内容をすべて無視します。

例は次の通りです。

```
rule "Testing Comments"
```

```

when
    # this is a single line comment
    // this is also a single line comment
    eval( true ) # this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
    # this is another comment in a semantic code block
end

```



図4.2 複数行コメント

複数行コメントは、セマンティックコードブロックの内外両方のテキストブロックを示すために使用されます。

例は次の通りです。

```

rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end

```

## 4.4. エラーメッセージ

**JBoss Rules** には標準化されたエラーメッセージの機能があります。この機能は、問題の検索や解決を迅速および容易にします。本項を読んで、エラーメッセージの識別および解釈方法を学び、報告される問題の一部を解決する方法について確認しましょう。

エラーメッセージの形式は次のようになります。

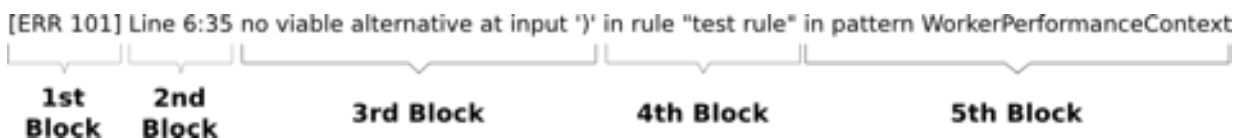


図4.3 エラーメッセージの形式

### 最初のブロック

この領域はエラーコードを特定します。

### 2 番目のブロック

この領域には行および列の情報が表示されます。

### 3 番目のブロック

この行行には問題を説明するテキストが表示されます。

### 4 番目のブロック

最初のコンテキストです。通常、エラーが発生したルール、関数、テンプレート、またはクエリを示します (このブロックは必須ではありません)。

## 5 番目のブロック

エラーが発生したパターンを特定します (このブロックは必須ではありません)。

以下にすべてのエラーメッセージについて説明します。

### 4.4.1. 101: No viable alternative

最も一般的なエラーに対して表示されるメッセージです。**parser** が決定ポイントに到達し、代替を特定できない場合に表示されます。例は次の通りです。

```
rule one
when
    exists Foo()
    exits Bar()
then
end
```

この例は次のメッセージを生成します。

```
[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one
```

一見、有効な構文に見えるかもしれませんが、**exits != exists** であるため、実際は有効ではありません。

別の例を見てみましょう。

```
package org.drools;
rule
when
    Object()
then
    System.out.println("A RHS");
end
```

上記のコードは次のエラーメッセージを生成します。

```
[ERR 101] Line 3:2 no viable alternative at input 'WHEN'
```

このメッセージは、**parser** が WHEN トークンを検出し(実際、これはハードキーワードです)、ルール名がないためこれが誤った場所にあることを意味します。

単純に語彙を間違えた時も同じエラーメッセージが表示されます。この問題の例は次の通りです。

```
rule simple_rule
when
    Student( name == "Andy )
then
end
```

閉じ引用符がないため、**parser** が次のようなエラーメッセージを生成します。

```
[ERR 101] Line 0:-1 no viable alternative at input
'<eof>' in rule simple_rule in pattern Student
```



#### 注記

通常、行と列の情報は正確ですが、**parser** が **0:-1** の位置を生成する場合があります。この場合、引用符、アポストロフィ、および括弧がすべて閉じられているか確認してください。

### 4.4.2. 102: Mismatched input

このエラーは、**parser** が特定の記号を探していたものの、現在の入力場所になかったことを示しています。例は次の通りです。

```
rule simple_rule
when
    foo3 : Bar(
```

このコードは次のメッセージを生成します。

```
[ERR 102] Line 0:-1 mismatched input '<eof>' expecting
')' in rule simple_rule in pattern Bar
```

この問題を修正するには、ルールステートメントを完成させます。



#### 注記

**0:-1** 位置は **parser** がソースの最後に到達したことを意味します。

次のコードは複数のエラーメッセージを生成します。

```
package org.drools;

rule "Avoid NPE on wrong syntax"
when
    not(Cheese((type=="stilton",price==10)|| (type=="brie",price==15))
        from $cheeseList)
then
    System.out.println("OK");
end
```

以下はこのソースに関連するエラーです。

```
[ERR 102] Line 5:36 mismatched input ', ' expecting ')' in rule
"Avoid NPE on wrong syntax" in pattern Cheese
```

```
[ERR 101] Line 5:57 no viable alternative at input 'type' in
rule "Avoid NPE on wrong syntax"
```

```
[ERR 102] Line 5:106 mismatched input ')' expecting 'then' in
rule "Avoid NPE on wrong syntax"
```

2 つ目の問題が最初の問題に関連していることに注意してください。この問題を修正するには、コンマ(,) を **&&** 演算子に置き換えます。



#### 注記

エラーメッセージが複数ある場合、1 つずつ修正するのがよいでしょう。一部のメッセージは他のメッセージの結果としてのみ生成されます。

### 4.4.3. 103: Failed predicate

これは、**検証するセマンティック述語**が **false** として評価されたことを意味します。通常、これらのセマンティック述語は「ソフト」キーワードを識別するために使用されます。次の例でも同様です。

```
package nesting;
dialect "mvel"

import org.drools.Person
import org.drools.Address

fdsfdsfds

rule "test something"
when
    p: Person( name=="Michael" )
then
    p.name = "other";
    System.out.println(p.name);
end
```

この例によって生成されるメッセージは次の通りです。

```
[ERR 103] Line 7:0 rule 'rule_key' failed predicate:
{(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule
```

**fdsfdsfds** は無効なテキストで、**parser** はそれをソフトキーワードの rule として識別できません。



#### 注記

このエラーは **102: Mismatched input** と大変よく似ていますが、通常はソフトキーワードが関係します。

### 4.4.4. 104: Trailing semi-colon not allowed

このエラーは eval 節と関係しています。式が誤ってセミコロンで終了すると発生します。例は次の通りです。

```
rule simple_rule
when
    eval(abc();)
```

```
then
end
```

このエラーは eval 節内の末尾にあるセミコロンによって発生します。

```
[ERR 104] Line 3:4 trailing semi-colon not allowed in rule
simple_rule
```

この問題を修正するのは簡単です。セミコロンを削除します。

#### 4.4.5. 105: Early Exit

これは、**recognizer** がどの代替とも一致しない文法 (grammar) のサブルールを検出すると発生します。これは、**parser** が出口のないブランチに入ったことを意味します。この状態を表す例は次のようになります。

```
template test_error
    aa s 11;
end
```

結果となるエラーメッセージ全体は次の通りです。

```
[ERR 105] Line 2:2 required (...) + loop did not match anything
at input 'aa' in template test_error
```

この問題を修正するには、数値を削除します (新しいテンプレートスロットを開始する可能性がある有効なデータタイプではなく、他のルールファイルコンストラクトの可能な開始でもないため)。

本項の内容を読んだ後、エラーメッセージの意味とエラーメッセージが示す問題の修正方法について理解していただけたと思います。

### 4.5. パッケージ

パッケージはインポートやグローバルなど、ルールやその他関連するコンストラクトのコレクションです。通常、パッケージメンバーは相互に関連しています。その例が HR ルールなどになります。パッケージは名前空間を表し、名前空間はルールのグループ化に対して一意であることが理想的です。パッケージ名自体が名前空間ですが、ファイルやフォルダーには関連しません。

複数のルールソースからルールをアセンブルすることが可能で、すべてのルールが対象となる (ルールがアセンブルされた時)、トップレベルのパッケージ設定を持つことが可能です。しかし、異なる名前宣言された同じパッケージリソースへマージすることは不可能です。単一の Rulebase には含まれる複数のナレッジパッケージを構築することが可能です。パッケージのルールをすべてパッケージ宣言と同じファイルに格納し、自己充足にすることが一般的です。

次の syntax diagram は、パッケージを構成するすべてのコンポーネントを表しています。パッケージには名前空間がある必要があり、パッケージ名の標準的な Java の慣例を使用して宣言する必要があることに注意してください (空白文字を使用できるルール名とは異なり、名前空間には空白文字を使用できません)。ルールの順序に関しては、ファイルの上にある必要がある package ステートメント以外は、任意の順序でルールファイルに記載されます。すべての場合で、セミコロンの使用は任意となります。

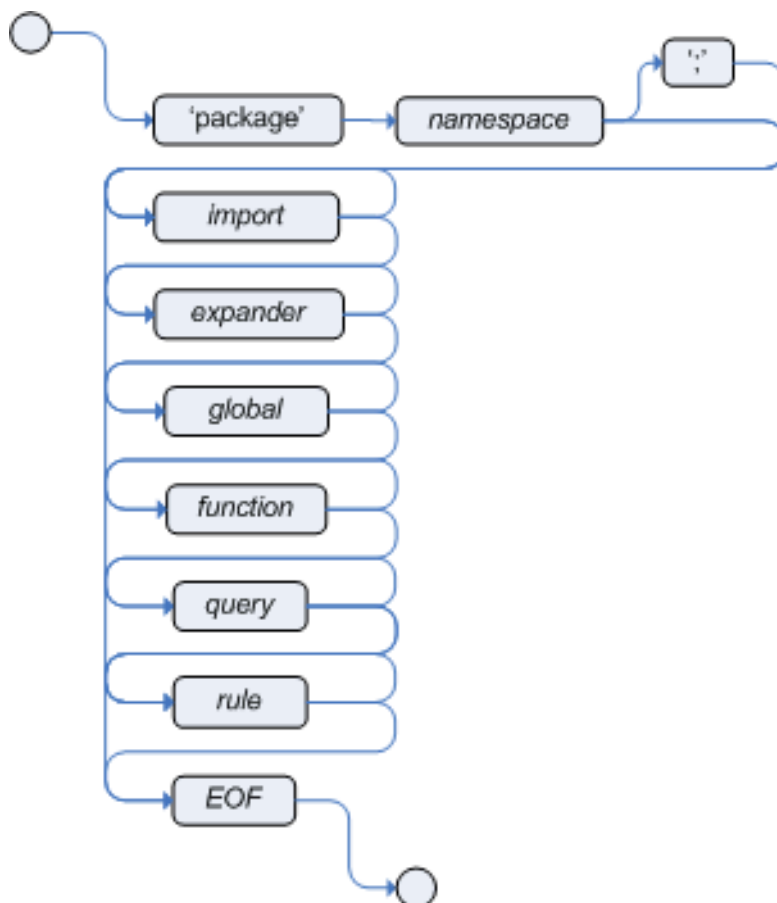


図4.4 package



## 注記

ルール属性もすべてパッケージレベルで書き込みされ、属性のデフォルト値が置き換えられることに注意してください (ルール属性の項を参照)。変更されたデフォルトもルール内の属性設定によって置き換えられることがあります。

## 4.5.1. import



図4.5 import

import ステートメントは Java の import ステートメントと同様に挙動します。ルールで使いたいオブジェクトの完全修飾パスとタイプ名を指定する必要があります。JBoss Rules は、同名の Java パッケージおよび `java.lang` パッケージより自動的にクラスをインポートします。

## 4.5.2. global



図4.6 global

global を用いてグローバル変数を定義します。グローバル変数は、アプリケーションオブジェクトを



ルールで使用できるようにするため使用されます。一般的に、ルールが使用するデータまたはサービス(特にルール結果で使用するアプリケーションサービス)を提供し、ルール結果に追加されるログや値などのデータをルールから返すためにグローバル変数が使用されます。ルールがアプリケーションと対話し、コールバックを実行するために使用されることもあります。グローバルはワーキングメモリーには挿入されないため、一定の不変の値を持つ場合以外はルールの条件を確立するためにグローバルを使用すべきではありません。グローバルの値変更についてエンジンは通知されず、変更を追跡しません。制約でグローバルを適切に使用しないと、予期しない悪い結果がもたらされることがあります。

複数のナレッジパッケージが同じ識別子を用いてグローバルを宣言する場合、すべてが同じタイプで、同じグローバル値を参照する必要があります。

グローバルを使用するには、以下を実行する必要があります。

1. ルールファイルにグローバル変数を宣言し、ルールで使用します。例は次の通りです。

```
global java.util.List myGlobalList;

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

2. ワーキングメモリーにグローバル値を設定します。ワーキングメモリーにファクトをアサートする前にすべてのグローバル値を設定することが推奨されます。例は次の通りです。

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

これらは、アプリケーションからワーキングメモリーへ渡すオブジェクトの名前付きインスタンスにすぎないことに注意してください。そのため、サービスロケーターやサービス自体など、好きなオブジェクトを渡すことができます。from 要素が新たに導入されたため、Hibernate セッションをグローバルとして渡し、from によって名前付きの Hibernate クエリよりデータをプルすることが一般的になりました。

電子メールサービスのインスタンスがこの一例となります。ルールエンジン呼び出す統合コードで、emailService オブジェクトを取得し、ワーキングメモリーに設定します。タイプが EmailService のグローバルがあることを DRLで宣言し、名前を「email」にします。その後、ルール結果にて email.sendSMS(number, message) などを使用できます。

グローバルはルール間でデータを共有するためのものではないため、このような目的で使用すべきではありません。ルールは常にワーキングメモリーの状態を理由付けし、反応するため、ルール間でデータを渡したい場合はデータをファクトとしてワーキングメモリーにアサートします。

ルール内よりグローバルの値を設定または変更することは推奨されません。常にワーキングメモリーインターフェースを使用してアプリケーションより値を設定することが推奨されます。

## 4.6. 関数

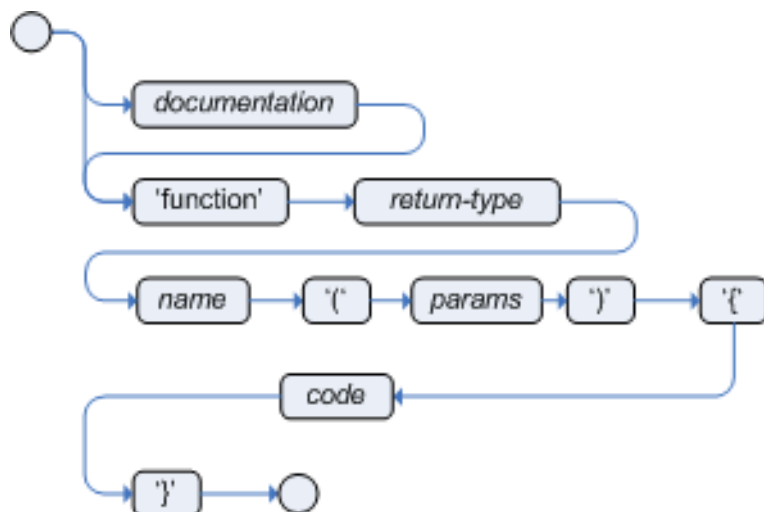


図4.7 関数

関数を使用して、セマンティックコードをルールソースファイル (普通の Java クラスではなく) に格納します。**helper** クラス以上のことは実行できませんが (実際にコンパイラが「背後」で **helper** クラスを生成します)、ロジックをすべて 1 つの場所に集められることが主な利点となります。また、必要に応じて関数を変更することもできます (これには良い面と悪い面の両方があります)。

関数はルールの結果でアクションを呼び出す場合に最も便利です。特定のアクションが繰り返し使用される場合 (電子メールのメッセージの内容など、各ルールの異なるパラメーターのみ)、特に有用です。

以下は標準的な関数の宣言になります。

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



### 注記

実際には Java の一部ではなくても、**function** キーワードが使用されます。関数にとってパラメーターは通常のメソッドと同様です (また、必要でない場合はパラメーターを使用する必要はありません)。戻り値は通常のメソッドと同様です。

代わりに、**Foo.hello()** のように **helper** クラスで静的メソッドを使用することもできます。**JBoss Rules** では **関数インポート** を使用できます。次のコード例は使用方法を表しています。

```
import function my.package.Foo.hello
```

上記両方の場合、関数を使用するには、結果またはセマンティックコードブロック内のいずれかで関数の名前を呼び出します。以下は最終的な例になります。

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

## 4.7. タイプ宣言

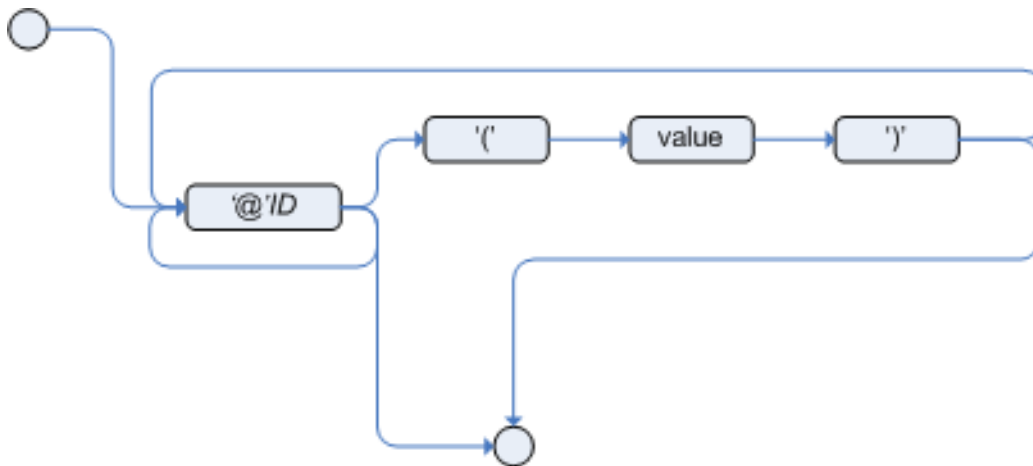


図4.8 meta\_data

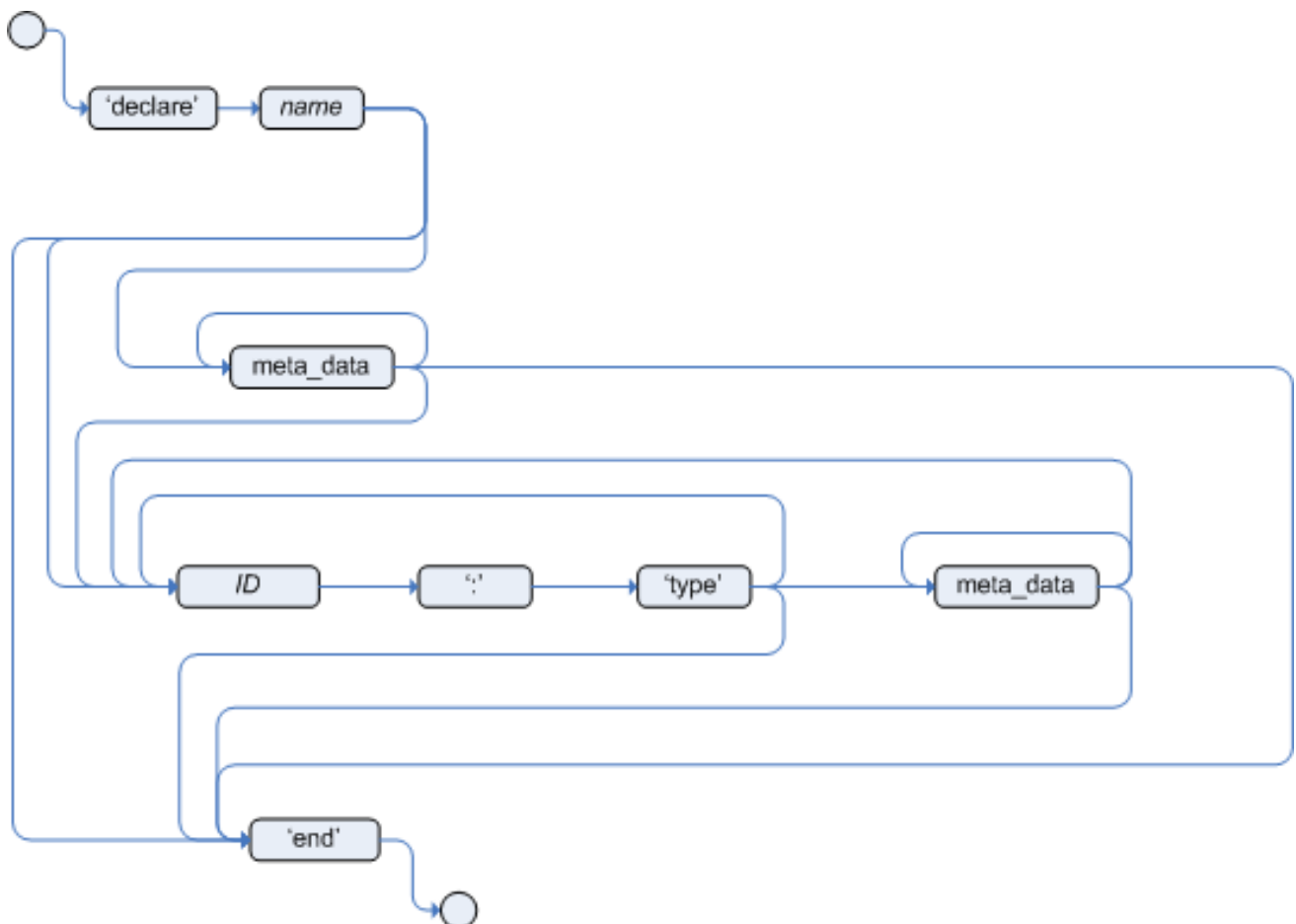


図4.9 type\_declaration

タイプ宣言のルールエンジンに関する主な2つの目的は、新しいタイプの宣言を許可することと、タイプのメタデータの宣言を許可することです。

- **新しいタイプの宣言:** JBoss Rules はそのままの状態ではプレーン POJO をファクトとして動作します。Java などの下位言語でモデルを作成することを懸念せずに直接モデルをルールエンジンへ定義したい時があります。また、ドメインモデルが既に構築されている状態で、主に理由付け処理中に使用される追加エンティティを用いてこのモデルを補完したい場合や補完する必要がある場合もあります。

- **メタデータの宣言:** ファクトに、関連するメタ情報があることがあります。メタ情報の例には、ファクト属性によって表されず、そのファクトタイプのすべてのインスタンス間で一貫するあらゆる種類のデータが含まれます。メタ情報はエンジンによってランタイム時にクエリされ、理由付け処理で使用されることがあります。

#### 4.7.1. 新しいタイプの宣言

新しいタイプを宣言するには、**declare** をキーワードとして使用し、その後にフィールドのリストとキーワード **end** を指定します。

##### 例4.1 新しいファクトタイプの宣言: Address

```
declare Address
  number : int
  streetName : String
  city : String
end
```

上記の例は **Address** という新しいファクトタイプを宣言します。このファクトタイプには、**number**、**streetName**、および **city** の3つの属性があります。各属性は有効な Java タイプであるタイプを持ちます。これには、ユーザーや以前宣言された他のファクトタイプによって作成される他のクラスが含まれます。

たとえば、別のファクトタイプである **Person** を宣言できます。

##### 例4.2 新しいファクトタイプ Person の宣言

```
declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end
```

上記の例の通り、**dateOfBirth** は Java API からの **java.util.Date** タイプで、**address** は以前に定義されたファクトタイプ Address です。

以前に説明した **import** を使用すると、書き込みごとにクラスの完全修飾名を記述する必要がなくなります。

##### 例4.3 import を使用して完全修飾クラス名の使用を省略する

```
import java.util.Date

declare Person
  name : String
  dateOfBirth : Date
  address : Address
end
```

新しいファクトタイプを宣言する場合、JBoss Rules はコンパイル時にファクトタイプを表す POJO を実装するバイトコードを生成します。生成された Java クラスは、タイプ定義の 1 対 1 の Java Bean マッピングになります。そのため、前の例の場合、生成される Java クラスは次のようになります。

#### 例4.4 以前の Person ファクトタイプ宣言に対して生成された Java クラス

```
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // getters and setters
    // equals/hashCode
    // toString
}
```

これは単純な POJO であるため、生成されたクラスは他のファクト同様にルールで透過的に使用できます。

#### 例4.5 ルールで宣言されたタイプの使用

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    System.out.println( "The name of the person is "+ )
    // lets insert Mark, that is Bob's mate
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

### 4.7.2. メタデータの宣言

JBoss Rules では、メタデータはファクトタイプ、ファクト属性、およびルールなど、さまざまなコンストラクションに割り当てることができます。JBoss Rules は @ 記号を使用してメタデータを導入し、常に次の形式を使用します。

```
@metadata_key( metadata_value )
```

括弧と metadata\_value は任意です。

たとえば、値が **Bob** である **author** のようなメタデータ属性を宣言したい場合、次のように記述します。

#### 例4.6 任意のメタデータ属性に宣言

```
@author( Bob )
```

JBoss Rules では、任意のメタデータ属性を宣言できますが、一部はエンジンに対して特別な意味を持っており、その他のメタデータ属性は実行時のクエリに対して使用できます。また、JBoss Rules ではファクトタイプとファクト属性の両方に対してメタデータを宣言できます。ファクトタイプのフィールドの前に宣言されたメタデータはファクトタイプへ割り当てられ、属性の後に宣言されたメタデータは特定の属性に割り当てられます。

#### 例4.7 ファクトタイプと属性に対するメタデータ属性の宣言

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

上記の例では、ファクトタイプに対して2つのメタデータが宣言され (@author と @dateOfCreation)、さらに2つのメタデータが2つの名前属性 (@key と @maxLength) に対して定義されています。@key メタデータは値を持たないため、括弧と値が省略されていることに注意してください。

#### 4.7.3. 既存タイプに対するメタデータの宣言

JBoss Rules では、新しいファクトタイプにメタデータ属性を宣言する場合と同様に既存のタイプのメタデータ属性を宣言することができます。唯一の違いは宣言にフィールドがないことです。

たとえば、クラス org.drools.examples.Person があり、このクラスに対してメタデータを宣言したい場合、次のコードを作成します。

#### 例4.8 既存のタイプに対するメタデータの宣言

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

インポートを使用する代わりに完全修飾名でクラスを参照することも可能ですが、クラスはルールでも参照されるため、通常はインポートを追加し、短いクラス名を使用した方が効率的です。

#### 例4.9 完全修飾クラス名を使用したメタデータの宣言

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

#### 4.7.4. アプリケーションコードからの宣言されたタイプへのアクセス

宣言されたタイプは通常ルールファイルの内部で使用され、Java モデルはルールとアプリケーション間でモデルを共有する時に使用されます。しかし、アプリケーションがルールエンジンをラッピングし、ルール管理に対して高レベルでドメイン固有のユーザーインターフェースを提供する場合、アプリケーションが宣言されたタイプよりファクトにアクセスし、処理する必要があることがあります。

このような場合、生成されたクラスは Java Reflection API を用いて通常通り処理できますが、通常は小さな結果のために大量の作業が必要になります。JBoss Rules はアプリケーションが行う最も一般的なファクトの処理に対して簡易 API を提供します。

宣言されたファクトは宣言が行われたパッケージに属することが最初の重要点になります。たとえば、下記の例では **Person** は **org.drools.examples** パッケージに属するため、生成されるクラスの完全修飾名は **org.drools.examples.Person** になります。

##### 例4.10 org.drools.examples パッケージでのタイプの宣言

```
package org.drools.examples

import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

前述の通り、宣言されたタイプはナレッジベースのコンパイル時に生成されます。アプリケーションはアプリケーションの実行時のみアクセスできます。そのため、アプリケーションからの直接参照にこれらのクラスは使用できません。

JBoss Rules は、アプリケーションコードから宣言されたタイプを処理できるインターフェースを提供します (org.drools.definition.type.FactType)。このインターフェースより、宣言されたファクトタイプのフィールドをインスタンス化および読み書きすることができます。

##### 例4.11 API を介した宣言されたファクトタイプの処理

```
// get a reference to a knowledge base with a declared type:
KnowledgeBase kbase = ...

// get the declared FactType
FactType personType = kbase.getFactType( "org.drools.examples",
                                           "Person" );

// handle the type as necessary:
// create instances:
Object bob = personType.newInstance();

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

```
42 );
```

```
// insert fact into a session
StatefulKnowledgeSession ksession = ...
ksession.insert( bob );
ksession.fireAllRules();

// read attributes
String name = personType.get( bob, "name" );
int age = personType.get( bob, "age" );
```

API には他にも役に立つメソッドが含まれています。このようなメソッドには一度にすべての属性を設定するメソッド、マップから値を読み取るメソッド、すべての属性を一度に読み取るメソッド、マップにデータを投入するメソッドなどがあります。

API は Java リフレクションと似ていますが、リフレクションは使用しません。その代わりに、バイトコードによって生成されたより高速なアクセッサに依存します。

## 4.8. ルール

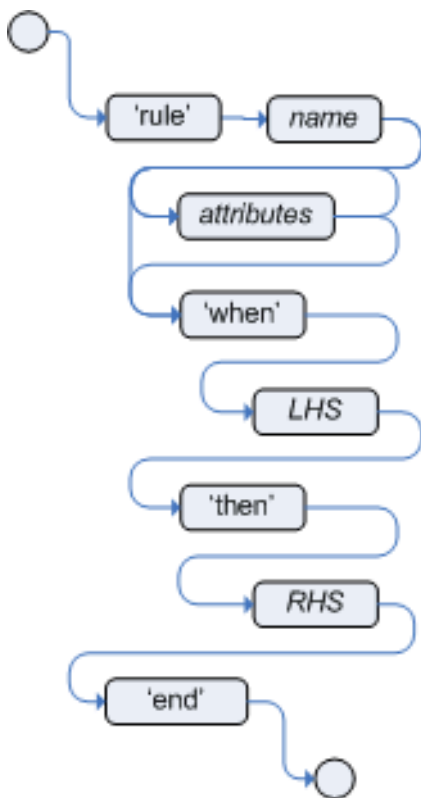


図4.10 ルール

ルールでは、特定の条件セットが発生した時に、左側に指定され、右側のアクションのリストとして指定されたものを実行するよう指定します。ユーザーからの一般的な疑問としては、「なぜ when を if の代わりに使用するのか」などがあります。when が if の代わりに選択される理由は、if は通常、特定時に条件がチェックされるプロシージャールール実行フローの一部であるためです。反対に、when は条件評価が特定の評価結果または時点に関係しませんが、エンジンが生存する間はいつでも連続して実行されます。条件が満たされると、アクションが実行されます。

ルールには、ルールパッケージ内で一意となる名前を付ける必要があります。ルールが単一の DRL で



2 回定義された場合は、ロード時にエラーが発生します。パッケージ内に既存するルール名が含まれる DRL が追加された場合は、以前のルールが置き換えられます。ルール名に空白文字が存在する場合、ルール名を二重引用符で囲む必要があります (常に二重引用符を使用してください)。

属性は任意です。必ず 1 行に 1 つずつ記述してください。

ルールの左側は when キーワードの後に指定し (新しい行にするのが理想的です)、同様に右側は when キーワードの後に指定します (この場合も新しい行にするのが理想的です)。ルールはキーワード end で終了します。ルールをネストすることはできません。

#### 例4.12 ルール構文の概要

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

#### 例4.13 単純なルール

```
rule "Approve if not rejected"
    salience -100
    agenda-group "approval"
when
    not Rejection()
    p : Policy(approved == false, policyState:status)
    exists Driver(age > 25)
    Process(status == policyState)
then
    log("APPROVED: due to no objections.");
    p.setApproved(true);
end
```

#### 注記

**JBoss Rules** は、プリミティブまたはオブジェクトラッパー形式で番号を保持しようとするため、int プリミティブにバインドされた変数がコードブロックまたは式で使用されると、手作業でボックスを解除する必要がなくなります。この点が、すべてのプリミティブが「自動的にボックス化」され、手作業による「ボックス化の解除」が必要な JBoss Rules 3.0 とは異なります。オブジェクトラッパーにバインドされた変数はオブジェクトのままになります。この場合は、自動ボックス化およびボックス化の解除を処理する既存の JDK 1.5 および JDK 5 のルールが適用されます。フィールド制約を評価する場合は、システムが値の 1 つを同等の形式に強制変換します。そのため、プリミティブはオブジェクトラッパーに相当します。

#### 4.8.1. ルール属性

ルール属性は、宣言によってルールの動作に影響を与えることができます。単純な属性もありますが、**Ruleflow** など複雑なサブシステムの一部であるものもあります。**JBoss Rules** を活用するには、本項を読んで各属性をよく理解するとよいでしょう。

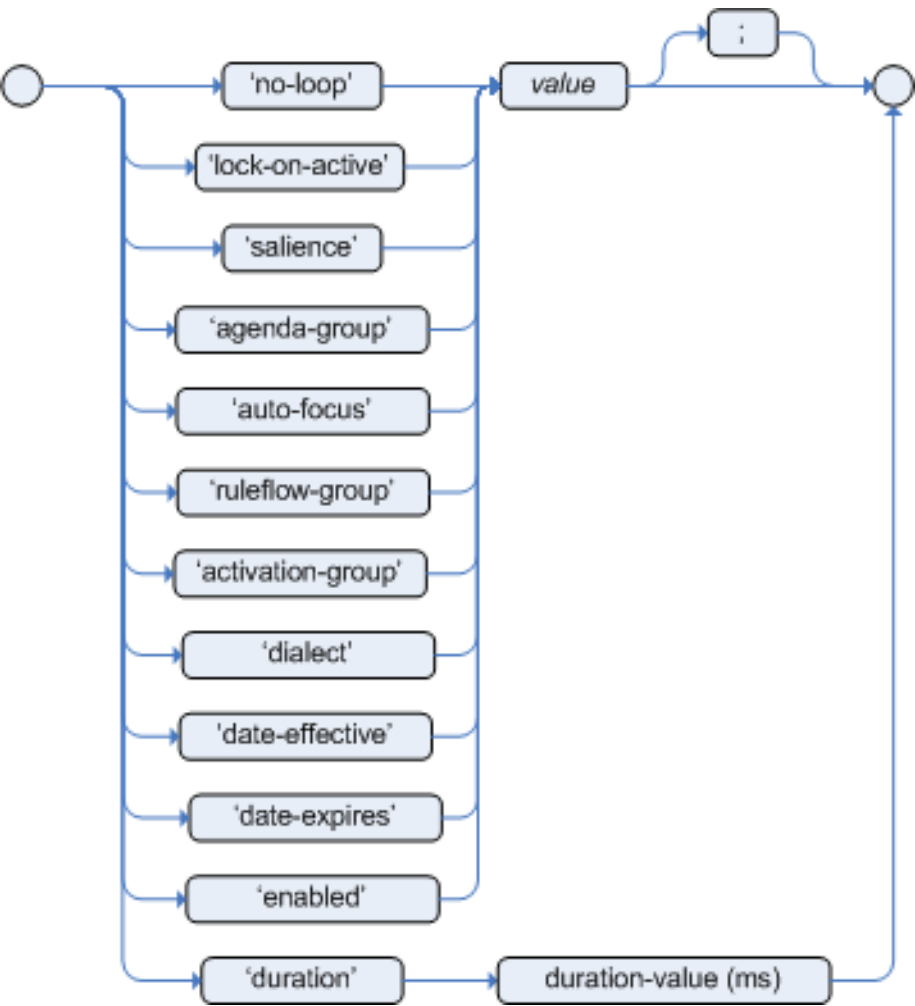



図4.11 ルール属性

表4.1 ルール属性

属性	デフォルト値	タイプ	コメント
no-loop	false	ブール値	ルールの結果がファクトを変更すると、ルールが再度アクティベートされることがあり、再帰が行われる原因となります。no-loop を <b>true</b> に設定して、現在のデータセットに対する <b>activation</b> の作成が無視されるようにします。
ruleflow-group	N/A	文字列	ルールフロー機能を使用して、ルールの実行を制御します (同じ ruleflow-group 識別子によってアセンブルされたルールはグループがアクティブな時のみ実行されます)。

属性	デフォルト 値	タイプ	コメント
lock-on-active	false	ブール値	uleflow-group がアクティブになるか、agenda-group がフォーカスを受けると、lock-on-active が true に設定されたそのグループ内のルールがアクティブでなくなります。更新元に関係なく、一致するルールのアクティベーションは破棄されます。これは、変更がルール自体のみによって引き起こされるわけではないため、no-loop の強いバージョンになります。これは、ファクトを変更する複数のルールが存在し、ルールの再照合や再実行を希望しない場合の計算ルールに最適です。ruleflow-group がアクティブでなくなるか、agenda-group がフォーカスを失った時のみ、lock-on-active が true に設定されたルールがアジェンダに配置されるアクティベーションに対して再び有効になります。
salience	0	整数	各ルールは整数を割り当てることができる salience 属性を持ちます。デフォルトはゼロです。salience は優先度の形式で、 <b>activation queue</b> で順序付けされる時に大きい値を持つルールに高い優先度が割り当てられます。
agenda-group	MAIN	文字列	アジェンダグループを用いるとアジェンダを分割できるため、実行制御を強化できます。フォーカスを取得したアジェンダグループのルールのみ実行が許可されます。
auto-focus	false	ブール値	<b>auto-focus</b> 値が <b>true</b> のルールがアクティベートされた場合や、ルールの <b>agenda</b> グループがフォーカスを取得していない場合、フォーカスがグループに与えられ、ルールを実行できるようになります。
activation-group	N/A	文字列	<p>この文字列は同じ <b>activation</b> グループに属するルールを識別します。このようなグループのルールはお互いを排他的に実行します。つまり、実行する <b>activation</b> グループの最初のルールは他のルールのアクティベーションをキャンセルし、実行を阻止します。</p> <div>  <div> <p><b>注記</b></p> <p>これは以前 Xor グループと呼ばれていましたが、厳密には Xor の定義を満たしていません。</p> </div> </div>

属性	デフォルト値	タイプ	コメント
dialect	パッケージの指定通り	文字列で、可能な値は <b>java</b> と <b>mvel</b>	この属性を使用して、左側または右側のいずれかにあるコード式に使用される言語を指定します。現在、Java と MVFLEX 式言語 の 2 つのダイアレクトを使用できます (ダイアレクトはパッケージレベルで指定できますが、この属性を使用すると、ルールのパッケージ定義を上書きできます)。
date-effective	N/A	日付と時間の定義が含まれる文字列	この属性で設定されたタイムスタンプよりも現在の日時が後であることを示す場合にのみルールをアクティベートできます。
date-expires	N/A	日付と時間の定義が含まれる文字列	この属性で設定されたタイムスタンプよりも現在の日時が後であるとルールをアクティベートできません。
duration	デフォルト値なし	long	この属性を使用すると、指定された期間後に属性が <b>true</b> のままであった場合にルールが実行されます。

次のコード例は、一般的な属性の一部を使用する方法について表しています。

```
rule "my rule"
salience 42
agenda-group "number 1"
when ...
```

#### 4.8.2. タイマーとカレンダー

JBoss Rules は、間隔および cron ベース両方のタイマーをサポートするようになりました。これらのタイマーは、廃止された duration 属性の代わりとなります。

##### 例4.14 timer 属性の使用例

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron: * 0/15 * * * ? )
```

間隔「int:」タイマーは、初期遅延の JDK セマンティックに従い、任意で繰り返し間隔が続きます。Cron「cron:」タイマーは標準的な cron 式に従います。

##### 例4.15 cron の例

```
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

```

ルールの実行時、カレンダーが制御するようになりました。カレンダー API は <http://www.quartz-scheduler.org/> にモデル化されています。

#### 例4.16 クォーツ式カレンダーの適応

```

Calendar weekDayCal =
  QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)

```

カレンダーは StatefulKnowledgeSession で登録されます。

#### 例4.17 カレンダーの登録

```

ksession.getCalendars().set( "week day", weekDayCal );

```

これらは、通常のルールやタイマーが含まれるルールと共に使用できます。ルールカレンダー属性は 1 つまたは複数のコンマカレンダー名を持つことが可能です。

#### 例4.18 カレンダーとタイマーを一緒に使用

```

rule "weekdays are high priority"
  calendars "weekday"
  timer (int:0 1h)
when
  Alarm()
then
  send( "priority high - we have an alarm" );
end

rule "weekend are low priority"
  calendars "weekend"
  timer (int:0 4h)
when
  Alarm()
then
  send( "priority low - we have an alarm" );
end

```

### 4.8.3. 左側条件要素

左側 (Left-Hand Side, LHS) は、ルールの 条件 (when) 部分の共通名で、条件要素で構成されます (何も指定しなくても問題ありません。左側が空白である場合、**eval(true)** と書き換えられます。そのため、ルールの条件は常に true のままになります)。

左側は、新しい **working memory** セッションが作成された時に一度だけアクティベートされます。



図4.12 左側

以下は **conditional element** のないルールになります。

```
rule "no CEs"
when
then
    <action>*
end
```

これは内部で次のように書き換えられます。

```
rule "no CEs"
when
    eval( true )
then
    <action>*
end
```

条件要素は 1 つまたは複数の パターン で動作します (詳細は次に説明します)。最も一般的な条件要素は **and** で、ルール の左側に複数の完全に無関係なパターンがある場合に暗示されます。



#### 注記

**or** パターンとは異なり、**and** は *主導宣言* バインディングを持つことができません。これは、宣言は 1 つのファクトへのみ参照でき、**and** が満たされると複数のファクトを照合するため、どのファクトへバインドするのか分からなくなるためです。

#### 4.8.3.1. パターン

パターンは **conditional element** の最も重要な型です。以下のエンティティ関係図はパターンの制約を構成するさまざまな部分の概要と、これらの部分がどのように連携するかを表しています。各部分の詳細は、図やコード例と共に本項の後半で取り上げます。

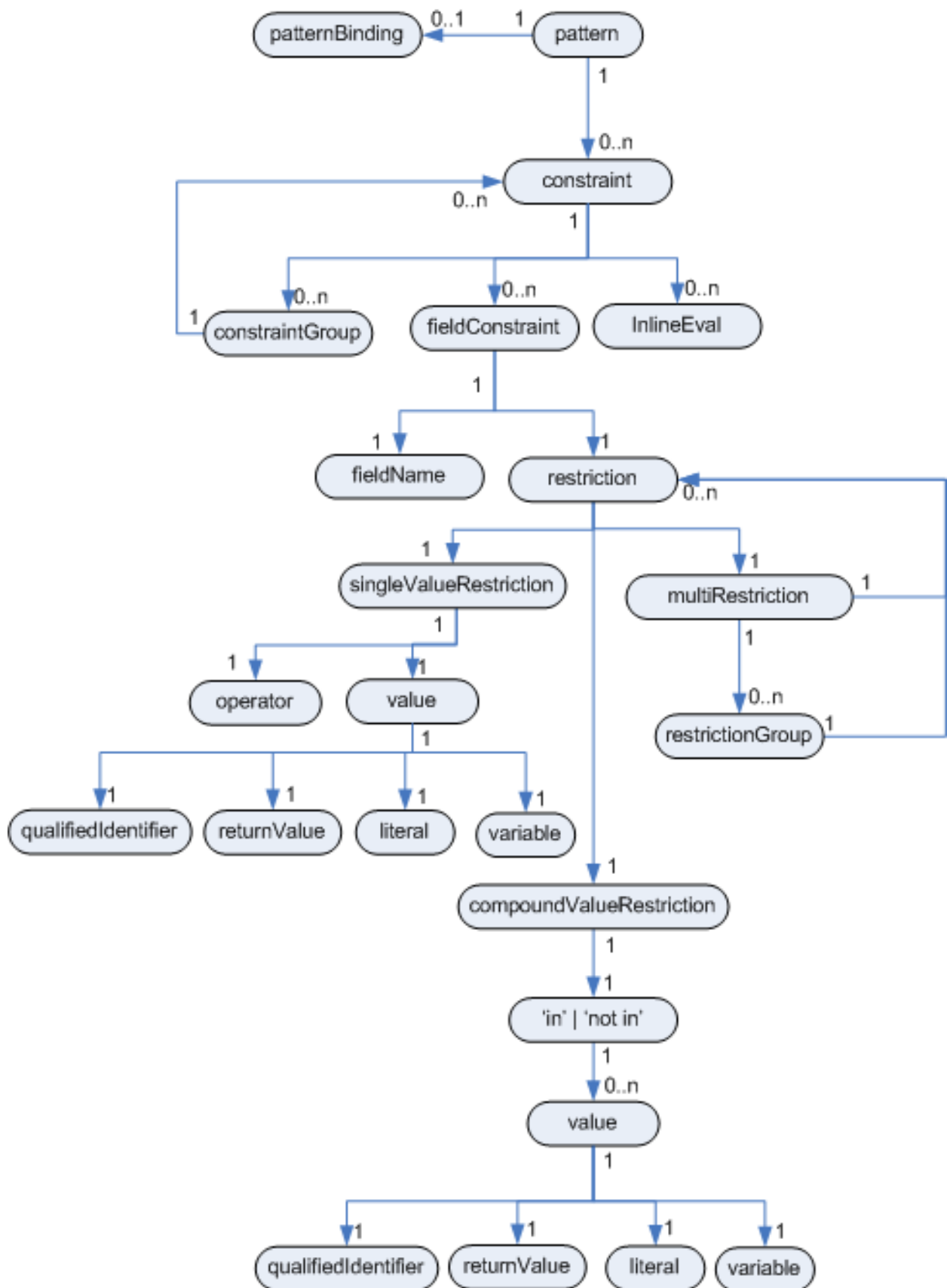


図4.13 パターンエンティティ関係図

エンティティ関係図の上部を見ると、パターンはゼロ以上の制約で構成され、パターンバイディングは任意であることが分かります。次の図は構文の形式を表しています。



図4.14 パターン

最も簡単な形式 (制約のない状態) では、パターンは該当する型のファクトに対して照合されます。以下の場合、型は **Cheese** であるため、パターンは **working memory** の各 **Cheese** オブジェクトに対して照合されます。



#### 注記

「型」はファクトオブジェクトの実際のクラスである必要はありません。パターンはスーパークラスやインターフェースも参照する場合があるため、多数のクラスからのファクトと一致する可能性があります。

**Cheese( )**

**\$c** などのパターンバインディング変数を使用して一致するオブジェクトを参照します。**\$** プレフィックスの使用は任意ですが、このプレフィックスを使用すると変数をフィールドを区別しやすくなるため、複雑なルールに対処するする場合に便利です。

**\$c : Cheese( )**

構文の主要要素は括弧です。括弧内に制約を置きます。主要な型は **フィールド制約**、**インライン評価**、および **制約グループ** になります。

**,**、**&&**、**||** のいずれかを使用して制約を区切ります。ただし、これらの記号は若干ことなる機能を持つことに注意してください。

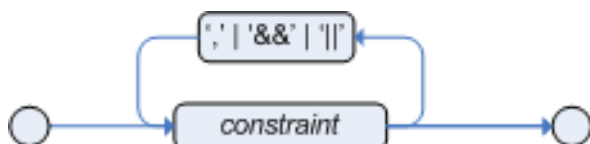


図4.15 制約

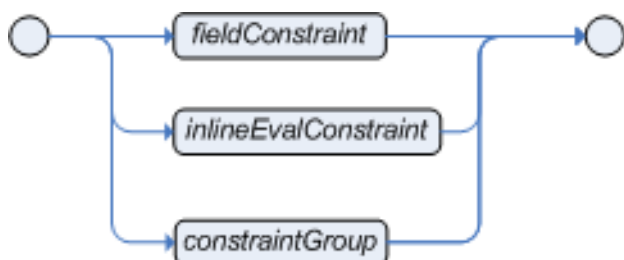


図4.16 制約



図4.17 constraintGroup

コンマ (,) は制約グループを区切るために使用されます。これは暗黙的で接続的な意味論を持ちます。



```
# Cheese type is stilton and price < 10 and age is mature.
Cheese( type == "stilton", price < 10, age == "mature" )
```

この例では、3つの制約グループがあり、各制約グループは単一の制約を持っています。

1. `type == "stilton"` の通り、型は stilton です。
2. `price < 10` の通り、価格 (price) は 10 未満です。
3. `age == "mature"` の通り、熟成度 (age) は mature です。

`&&` と `||` を用いると、グループは複数の制約を持つことができます。例は次の通りです。

```
// Cheese type is "stilton" and price < 10, and age is mature
Cheese( type == "stilton" && price < 10, age == "mature" )
// Cheese type is "stilton" or price < 10, and age is mature
Cheese( type == "stilton" || price < 10, age == "mature" )
```

この場合、2つの制約グループが存在します。最初の制約グループは2つの制約を持ち、2つ目の制約グループは制約を1つ持ちます。

結合記号は次の順で最初から最後まで評価されます。

1. `&&`
2. `||`
3. `,`

評価の優先度を変更するには、論理や数式のように括弧を使用します。例は次の通りです。

```
# Cheese type is stilton and ( price is less than 20 or age is mature ).
Cheese( type == "stilton" && ( price < 20 || age == "mature" ) )
```

この場合、括弧を使用すると `||` は `&&` の前に評価されるようになります。

`&&` と `,` は同じ意味論を持ちますが、異なる優先度で解決されることに注意してください。そのため、複合制約式に `,` を組み込むことはできません。

```
// invalid as ',' cannot be embedded in an expression:
Cheese( ( type == "stilton", price < 10 ) || age == "mature" )
// valid as '&&' can be embedded in an expression:
Cheese( ( type == "stilton" && price < 10 ) || age == "mature" )
```

#### 4.8.3.1.1. フィールド制約

フィールド制約を使用して名前付きフィールドを制限します。任意でフィールド名に変数バンディングを使用することができます。



図4.18 fieldConstraint

制限には次の 3 つの形式があります。

- 単一値制限
- 複合値制限
- 複数制限

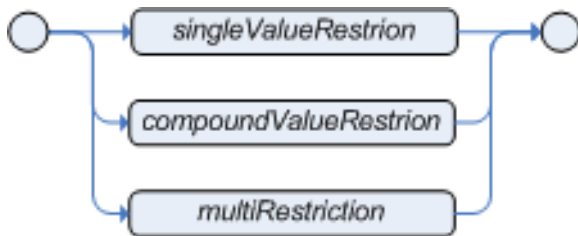


図4.19 制限

#### 4.8.3.1.2. ファクトとしての Java Bean

フィールドはオブジェクトの *アクセス可能* メソッドです。モデルオブジェクトが Java *Bean* パターンに従う場合、**getXXX** メソッドか **isXXX** メソッドを使用してフィールドが公開されます。この場合、これらのメソッドは引数を取りませんが、何らかの値を返します。

Bean 命名規則を使用してパターン内のフィールドにアクセスします (たとえば、**getType** は **type** としてアクセスされ、**JBoss Rules** は標準的な Java Development Kit の **Introspector** クラスを使用してこのマッピング処理を実行します)。

**Cheese** クラスの例では、**Cheese(type == "brie")** パターンは **getType()** メソッドを **Cheese** インスタンスへ適用します。フィールド名が見つからない場合、コンパイラーは最終的に引数のないメソッドとして名前を使用します。そのため、**Cheese(toString == "cheddar")** 制約により **toString()** が呼び出されます。この場合、大文字と小文字を正しく区別したメソッドの完全名を使用しますが、括弧は省略します。必ず、アクセスされるメソッドがパラメーターを取らず、これらのメソッドはルールに影響する方法でオブジェクトの状態を変更しない **accessors** であるようにしてください (パフォーマンス上の理由で、呼び出し間の照合の結果を **rule engine** がキャッシュすることに注意してください)。

#### 4.8.3.1.3. 値

フィールド制約は次の値を含むさまざまな値を取ります。

- リテラル
- qualifiedIdentifiers (列挙値)
- 変数
- returnValues

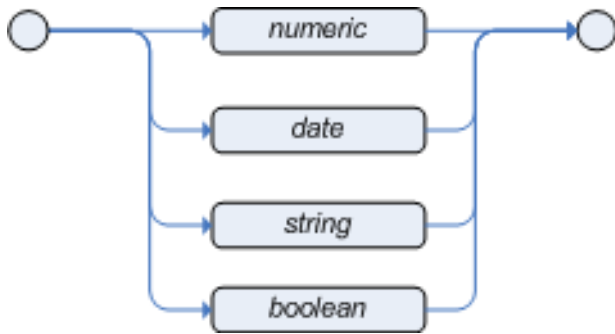


図4.20 リテラル



図4.21 qualifiedIdentifier



図4.22 変数



図4.23 returnValue

null であるフィールドに対してチェックを行うには、通常通り `==` と `!=` を使用します。リテラルの **null** キーワード (**Cheese(type != null)** のように使用) は、**evaluator** が例外をスローしない場合、値が null であると **true** を返します。

フィールドと値が異なる型である場合、システムは常に **型強制** を実行しようとします。「悪質な」強制を行おうとすると、例外が発生します。たとえば、数値 **evaluator** に **ten** を文字列として提供すると、例外が発生しますが、**10** を提供すると数値の 10 に強制します (強制は常に値型ではなくフィールド型を優先します)。

#### 4.8.3.1.4. 単一値制限

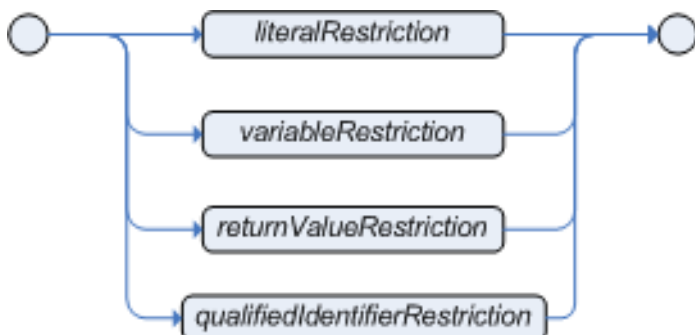


図4.24 singleValueRestriction

#### 4.8.3.1.5. 演算子

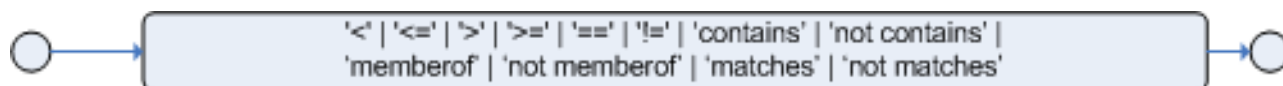


図4.25 演算子

**==** および **!=** 演算子はすべての型で有効です。型値が順序付けされている時は他のリレーショナル演算子を使用することもできます。データフィールドでは **<** は「前」を意味します。**matches** と **not matches** のペアは文字列フィールドへのみ適用し、**contains** と **not contains** には **Collection** 型のフィールドが必要になります (**evaluator** とフィールドに対して正しい値を強制しようとしません)。

### matches 演算子

これは有効な Java 正規表現があるフィールドと一致します。正規表現は通常は文字列リテラルですが、有効な正規表現を解決する変数を使用することもできます。



#### 重要

Java とは異なり、文字列リテラルとして書かれる正規表現内にエスケープは必要ありません。

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

### not matches 演算子

この演算子は文字列が正規表現と一致しない時に **true** を返します。**matches** 演算子でも同じルールが適用されます。使用例は次の通りです。

```
Cheese( type not matches "(Buffulo)?\S*Mozerella" )
```

### contains 演算子

この演算子を使用して、collection または array フィールドに指定された値が含まれていることを確認します。

```
CheeseCounter( cheeses contains "stilton" ) // contains with a String literal
CheeseCounter( cheeses contains $var ) // contains with a variable
```

### not contains 演算子

この演算子を使用して、collection または array フィールドに指定された値がないことを確認します。

```
CheeseCounter( cheeses not contains "cheddar" ) // not contains with a String literal
CheeseCounter( cheeses not contains $var ) // not contains with a variable
```



#### 注記

後方互換性を維持するため、**excludes** 演算子もサポートされています。この演算子は **not contains** と同等です。

### memberOf 演算子

この演算子を使用してフィールドが collection または array のメンバーであるかどうかを確認します。コレクションは変数である必要があります。

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

### not memberOf 演算子

この演算子を使用して、フィールドが collection または array のメンバーでないかどうかを確認します。コレクションは変数である必要があります。

```
CheeseCounter( cheese not memberOf $matureCheeses )
```

### soundlike 演算子

この演算子は **matches** と似ていますが、言葉が指定値と似た発音であるかどうかを確認します。**Soundex** アルゴリズムを使用し、英語の発音を基にして言葉の発音を確認します。

```
// match cheese "fubar" or "foobar"
Cheese( name soundlike 'foobar' )
```

#### 4.8.3.1.6. リテラル制限

最も単純な形式の制限です。フィールドを指定されたリテラルに対して評価します。リテラルは、数値、日付、文字列、またはブール値になります。



図4.26 literalRestriction

**==** 演算子を使用するリテラル制限は、ハッシングを使用してインデックスを作成しパフォーマンスを改善できるため、高速な実行を実現します。

#### 数値

標準的な Java 数値プリミティブがすべてサポートされます。

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

#### 日付

デフォルトの日付形式は **dd-mm-yyyy** です。これを変更するには、drools.dateformat プロパティの別の日付形式マスクを提供します (制御を強化する必要がある場合は、**inline-eval** 制約を使用します)。

```
Cheese( bestBefore < "27-Oct-2013" )
```

#### 文字列

有効な Java 文字列を使用します。

```
Cheese( type == "stilton" )
```

## ブール値

**true** か **false** のみを使用できます。0 と 1 は使用できません。単一のブール値フィールド (例 `Cheese( smelly )`) は許可されません。ブール値リテラルの比較対象となる必要があります。

```
Cheese( smelly == true )
```

## 修飾識別子

Java Development Kit 1.4 および 1.5 形式の **enums** はサポートされますが、1.5 は JDK 5 の環境でのみ使用可能です。

```
Cheese( smelly == SomeClass.TRUE )
```

### 4.8.3.1.7. バインドされた変数制限



図4.27 variableRestriction

変数をファクトとファクトのフィールドにバインドし、後続の **fied constraints** で使用することが可能です。バインドされた変数は **宣言** と呼ばれます。制約されるフィールドの型によって有効な演算子が決定されます。可能な場合、強制が試行されます。パフォーマンスを高速化するには、**==** 演算子を使用して変数制限をバインドします。

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

この例では、一致する各 **Person** インスタンスの **favouriteCheese** フィールドへ **likes** 変数がバインドされます (次のパターンで **Cheese** の型を制約します)。有効な Java 変数名を使用することができ、フィールドと宣言を区別するために使用される **\$** プレフィックスを使用できます。

次の例は、最初のパターンと一致するオブジェクトにバインドされる **\$stilton** の宣言を表しています。**contains** 演算子と共に使用されます (任意の **\$** の使用方法に注目してください)。

```
$stilton : Cheese( type == "stilton" )
Cheesery( cheeses contains $stilton )
```

### 4.8.3.1.8. 戻り値制限



図4.28 returnValueRestriction

**戻り値制限** は、リテラル、有効な Java プリミティブまたはオブジェクト、以前バインドされた変数、関数呼び出し、および演算子によって構成される括弧で囲まれている式です。使用される関数が時間に依存する結果を返すことは禁止されています。

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

## 4.8.3.1.9. 複合値制限

一致する可能性がある値が複数ある時に **複合値制限** を使用します（現在、**in** および **not in evaluators** のみがサポートしています）。

2 つ目のオペランドは、括弧で囲まれている値のコンマ区切りリストである必要があります。値は変数、リテラル、戻り値、または修飾識別子として渡すことができます。両方の **evaluators** は実際は「シンタクティックシュガー」で、**!=** および **==** 演算子を使用して複数制限のリストとして内部で書き直されます。

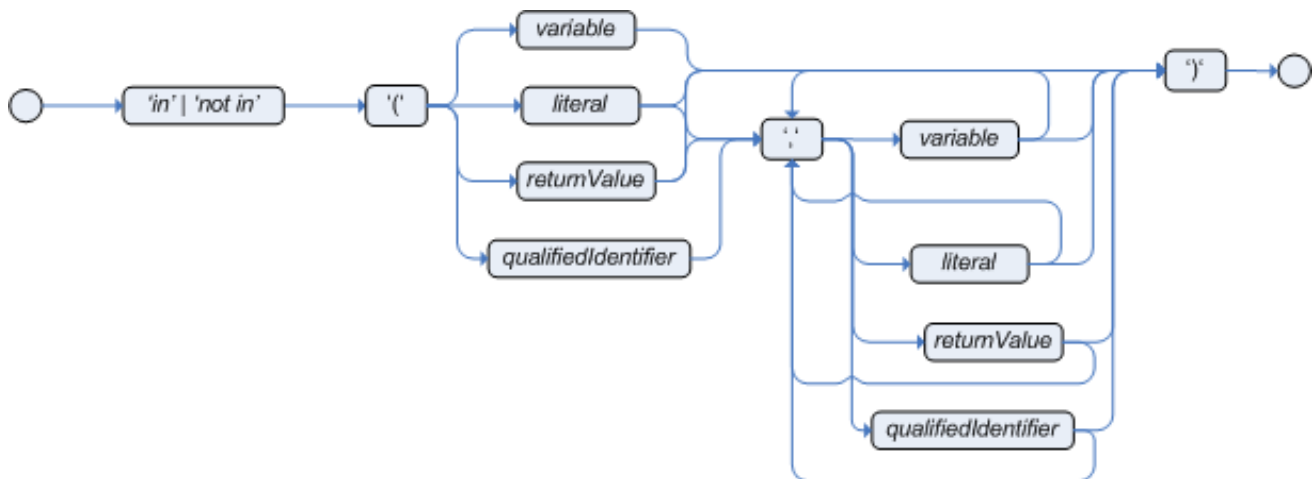


図4.29 compoundValueRestriction

```

Person( $cheese : favouriteCheese )
Cheese( type in ( "stilton", "cheddar", $cheese )

```

## 4.8.3.1.10. 複数制限

**複数制限** 制約を使用してフィールドに複数の制限を割り当てます (**&&** または **||** 分離記号を使用)。括弧を用いたグループ化は許可され、結果的に再帰的な構文パターンになります。

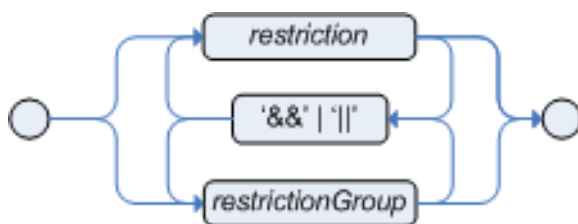


図4.30 multiRestriction



図4.31 restrictionGroup

```

// Simple multi restriction using a single &&
Person( age > 30 && < 40 )
// Complex multi restriction using groupings of multi restrictions
Person( age ( (> 30 && < 40) ||
               (> 20 && < 25) ) )
// Mixing multi restrictions with constraint connectives
Person( age > 30 && < 40 || location == "london" )

```



## 4.8.3.1.11. インライン評価制約



図4.32 インライン評価式

インライン評価 (inline eval) 制約はプリミティブなブール値へ解決する有効なダイレクト式を使用できます。この式は時間が経過しても一定である必要があります。現在または以前のパターンより以前バインドされた変数を使用できます。フィールドバインディング変数を自動的に作成するため、*auto-vivification* (自動有効化) も使用されます。



## 注記

現在の変数でない識別子が発見された場合、識別子が現在のオブジェクト型のフィールドであるかを判断するためにビルダーによってチェックされます。識別子がこれに該当する場合、フィールドバインディングが同じ名前の変数として自動的に作成されます。これは、フィールド変数の *auto-vivification* (自動有効化) と呼ばれます。

この例では、男性が女性よりも 2 歳年上となる可能な男女のペアをすべて探します。**age** 変数は *auto-vivification* によって 2 つ目のパターンで自動作成されます。

```

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

## 4.8.3.1.12. ネストされたアクセサー

**JBoss Rules** では、フィールド制約で ネストされたアクセサーを使用できます。これには、MVFLEX 式言語のアクセサーグラフ表記法を使用します (ネストされたアクセサーを持つフィールド制約は、実際には MVFLEX 式言語の **inline-eval** 制約として書き換えられます)。



## 警告

**working memory** はネストされた値を認識せず、ネストされた値がいつ変更するかも認識しないため、**nested accessors** を使用する時は十分注意してください。親参照が **working memory** に存在する間は常に不変の値として見なしてください。

ネストされた値を変更するには、最初に親オブジェクトを削除し、その後再度アサートします。グラフのルートに親が 1 つのみ存在する場合は、MVEL ダイアレクトの **modify** とその **ブロックセッター** を使用し、ネストされたアクセサーの割り当てを書くことができます。この場合、ルートの親オブジェクトを必要に応じて取り消したり挿入したりすることができます (**Nested accessors** は演算子記号のどちらの側でも使用できます)。

```

// Find a pet older than its owners first-born child
$p : Person( )
Pet( owner == $p, age > $p.children[0].age )
  
```



これは内部で MVEL **inline eval** として書き換えられます。

```
// Find a pet older than its owners first-born child
$p : Person( )
Pet( owner == $p, eval( age > $p.children[0].age ) )
```



#### 警告

**nested accessors** は **direct field accesses** よりもパフォーマンスへの影響がかなり大きいので、注意して使用してください。

### 4.8.3.2. and 条件要素

**and conditional element** を使用して他の **conditional elements** を論理積にグループ化します。

左側の **root element** は暗黙的な **and** プレフィックスです。これは指定する必要はありません。**JBoss Rules** は **and** をプレフィックスとインフィックスの両方としてサポートしますが、暗黙的なグループ化により混乱を避けることができるため、プレフィックスの使用が推奨されます。



図4.33 prefixAnd

```
(and Cheese( cheeseType : type )
Person( favouriteCheese == cheeseType ) )
```

```
when
Cheese( cheeseType : type )
Person( favouriteCheese == cheeseType )
```

必要な場合は、**and** インフィックスと括弧による明示的なグループ化がサポートされます。



#### 注記

**and** の代わりに **&&** 記号を使用できます。この記号は廃止されましたが、レガシーサポートの理由で現在でも使用可能です。



図4.34 infixAnd

```
//infixAnd
Cheese( cheeseType : type ) and Person( favouriteCheese == cheeseType )
```

```
//infixAnd with grouping
( Cheese( cheeseType : type ) and
  ( Person( favouriteCheese == cheeseType ) or
    Person( favouriteCheese == cheeseType ) ) )
```

#### 4.8.3.3. or 条件要素

**or conditional element** を使用して他の **conditional element** を論理和にグループ化します。



##### 注記

**JBoss Rules** では **or** をプレフィックスまたはインフィックスとして使用できますが、暗黙的なグループ化により混乱を避けることができるため、プレフィックスの使用が推奨されます。

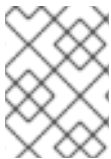
この **conditional element** の挙動は、フィールド制約や制限に対する結合記号 **||** とは異なります。**engine** は実際には **or** を理解しません。複数の異なる論理変換を使用することにより、**or** を使用するルールは複数のサブルールとして書き換えられます。この結果、ルールは単一の **root node** である **or** と、各 **conditional elements** に対する 1 つのサブルールを持ちます。各サブルールは通常のルールと同様にアクティベートおよび実行できます。特別な挙動や対話はありますが、それが新しい開発者が混乱する原因となることがあります。



図4.35 prefixOr

```
(or Person( sex == "f", age > 60 )
  Person( sex == "m", age > 65 ) )
```

必要な場合、**or** インフィックスと括弧による暗示的なグループ化がサポートされます。



##### 注記

**or** の代わりに **||** 記号を使用できます。この記号は廃止されましたが、レガシーサポートの理由で現在でも使用可能です。



図4.36 infixOr

```
//infixOr
Cheese( cheeseType : type ) or Person( favouriteCheese == cheeseType )
//infixOr with grouping
( Cheese( cheeseType : type ) or
  ( Person( favouriteCheese == cheeseType ) and
    Person( favouriteCheese == cheeseType ) ) )
```

任意で、**or** を用いて *pattern binding* を使用することもできます。これは、結果となる各サブルールがパターンにバインドされることを意味します。次の例のように、eponymous 変数を使用して各パターンを別々にバインドする必要があります。

```
(or pensioner : Person( sex == "f", age > 60 )
    pensioner : Person( sex == "m", age > 65 ) )
```

**or** を使用すると、可能な結果ごとに 1 つサブルールが作成されます。上記の単純な例では、2 つのルールが生成されます。これらのルールは **working memory** 内で独立して機能するため、両方とも照合、アクティベートおよび実行することが可能です。ショートカットはありません。

**or** を 2 つ以上の同様のルールを生成する方法と考えると分かりやすいかもしれません。2 つ以上の項の論理和が true の場合、単一のルールが複数のアクティベーションを持つことがあります。

#### 4.8.3.4. eval 条件要素



図4.37 eval

**eval conditional element** はプリミティブなブール値を返す意味論コードを実行できるようにする「catch-all」です。このコードはルールの左側にバインドされた変数か、ルールパッケージの関数を参照できます。



#### 警告

**eval** を使用しすぎないようにしてください。ルールの宣言性が低下し、**engine** のパフォーマンスが低下することがあります。**eval** はパターンのどこにでも使用できますが、ルールの左側で最後の条件要素として追加することが推奨されます。

**Evals** に索引を付けることはできません。そのため、フィールド制約ほど効率的ではありません。しかし、時間とともに変化する値を返す関数として使用することに適しています (フィールド制約にはこの機能はありません)。

```
p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey(p2.getItem()) )
// call function isValid in the LHS
eval( isValid(p1, p2) )
```

#### 4.8.3.5. not 条件要素

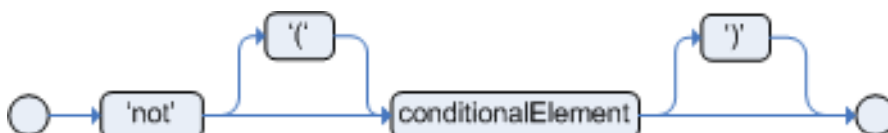


図4.38 not

**not conditional element** は、1 階論理の *非存在数量詞* (non-existential quantifier) です。この目的は、**working memory** に存在しないものを確認することです。

**not** キーワードは、括弧で囲まれた **conditional element** の前に指定する必要があります (最も単純なユースケースでは、括弧を省略することが可能です)。

```
not Bus()
```

```
// Brackets are optional:
not Bus(color == "red")
```

```
// Brackets are optional:
not ( Bus(color == "red", number == 42) )
```

```
// "not" with nested infix and - two patterns,
// brackets are required:
not ( Bus(color == "red") and Bus(color == "blue") )
```

#### 4.8.3.6. exists 条件要素

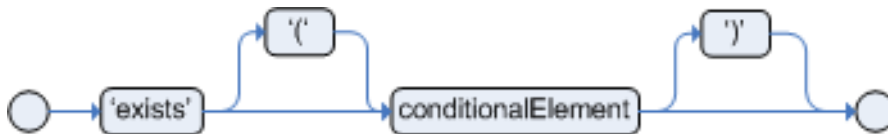


図4.39 exists

**exists conditional element** は 1 階論理の *存在数量詞* (existential quantifier) です。この目的は、**working memory** に存在するものを確認することです。**exists** を「少なくとも 1 つ存在する」という意味で考えるプログラマーもいます (「それぞれに対する」のように独自のパターンを持つこととは異なります)。

パターンで **exists** が使用されると、その条件と一致する **working memory** のデータの量に関係なく、ルールは一度だけアクティベートされます。存在のみが非常に重要となるため、バインディングは確立されません。

**exists** キーワードは、適用する **conditional elements** の前に指定する必要があります。**conditional elements** は括弧で囲まれている必要があります (以下のような最も単純な単一のパターンでは、括弧を省略することもできます)。

```
exists Bus()
```

```
exists Bus(color == "red")
// brackets are optional:
exists ( Bus(color == "red", number == 42) )
// "exists" with nested infix and,
// brackets are required:
exists ( Bus(color == "red") and
Bus(color == "blue") )
```

#### 4.8.3.7. conditional elements



図4.40 forall

**forall conditional element** は **JBoss Rules** の 1 階論理のサポートを完全にします。最初のパターンに一致するすべてのファクトが、残りの各パターンにも一致する場合、**forall** は **true** として評価します。例は次の通りです。

```

rule "All English buses are red"
when
    forall( $bus : Bus( type == 'english')
           Bus( this == $bus, color = 'red' ) )
then
    # all english buses are red
end
  
```

このルールは、タイプが **english** の各 **Bus** オブジェクトを選択します。そして、このパターンに一致する各ファクトに対し、後続のパターンが評価されます。後続のパターンにも一致する場合、**forall conditional element** は **true** として評価します。

該当するタイプの各ファクトが制約セットと一致するよう指定するには、以下のような単純な単一パターンを記述します。

```

rule "All Buses are Red"
when
    forall( Bus( color == 'red' ) )
then
    # all asserted Bus facts are red
end
  
```

反対に、複数パターンの場合は次のようになります。

```

rule "all employees have health and dental care programs"
when
    forall( $emp : Employee()
           HealthCare( employee == $emp )
           DentalCare( employee == $emp )
           )
then
    # all employees have health and dental care
end
  
```

**forall** は **not** などの他の **conditional elements** 内にネストできます。



### 重要

単一パターンの場合のみ括弧の使用が任意になります。そのため、ネストされる場合は括弧を使用する必要があります。

```

rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
  
```

```

        HealthCare( employee == $emp )
        DentalCare( employee == $emp ) )
    then
        # not all employees have health and dental care
    end

```

### 注記

`not( forall( p1 p2 p3... ) )` は次のコードと同等です。

```
not(p1 and not(and p2 p3...))
```

### 重要

**forall** は **範囲**の区切り文字であることに注意してください。そのため、以前にバインドされた変数を使用できますが、内部でバインドされた変数は外部で使用することはできません。

#### 4.8.3.8. from 条件要素



図4.41 from

**from conditional element** を使用して、左側のパターンによって一致されるデータの任意のソースを指定します。これを行うことにより、**engine** が **working memory** で発見されなかったデータを推論できます。データソースはバインドされた変数上のサブフィールドや、メソッド呼び出しの結果であることがあります。

これは、他のアプリケーションコンポーネントやフレームワークとそのまま統合できるようにする強力なコンストラクトです。一般的な例の1つは、**Hibernate** の名前が付けられたクエリを使用してデータベースから要求に応じて読み出されるデータとの統合です。

通常の MVFLEX 式言語の構文に従う式を使用してオブジェクトソースを定義します。これにより、メソッドコールの実行、マップや collections 要素へのアクセス、オブジェクトプロパティナビゲーションの使用を簡単に行うことができます。

別のパターンサブフィールドへの推論とバインディングを表す簡単な例は次の通りです。

```

rule "validate zipcode"
when
    Person( $personAddress : address )
    Address( zipcode == "23920W" ) from $personAddress
then
    # zip code is ok
end

```

グラフ表記法を使用して同じことを行う方法は次の通りです。

```

rule "validate zipcode"
when
    $p : Person( )

```

```

    $a : Address( zipcode == "23920W") from $p.address
  then
    # zip code is ok
  end

```

前述の例は単一パターンの評価でしたが、オブジェクトソース上で **from** を使用してオブジェクトのコレクションを返すこともできます。この場合、**from** がコレクションの各オブジェクトを繰り返し、個別に照合しようとします。以下は、注文の各商品を 10 % 割引するためのルールが含まれる例になります。

```

rule "apply 10% discount to all items over $ 100,00 in an order"
when
    $order : Order()
    $item : OrderItem( value > 100 ) from $order.items
then
    # apply discount to $item
end

```

各注文の値が 100 を越える各商品に対して、このルールが一度実行されます。

**from** を lock-on-active ルール属性と共に使用する場合、予期しない結果がもたらされることがあるため特に注意してください。次のように若干変更が加えられている前述の例を考慮してください。

```

rule "Assign people in Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person( )
  $a : Address( state == "QLD") from $p.address
then
  modify ($p) {} #Assign person to sales region 1 in a modify block
end

rule "Apply a discount to people in the city of Brisbane"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person( )
  $a : Address( city == "Brisbane") from $p.address
then
  modify ($p) {} #Apply discount to person in a modify block
end

```

この例では、QLD 州の Brisbane に在住する人は Sales Region 1 に割り当てられ、割引が適用されます (つまり、両方のルールがアクティベートおよび実行されることを想定しますが、2 つ目のルールのみが実行されます)。

監査ログを有効にすると、2 つ目のルールが実行された時に最初のルールを非アクティベートします。lock-on-active ルール属性により、ファクトのセットが変更された時にルールによって新しいアクティベーションが作成されないようにするため、最初のルールは再アクティベートできません (ファクトのセットは変更されませんが、**from** を使用すると目的に関係なく、評価される度に新しいファクトが返されます)。

次の手順に従います。

## 1. 前述のパターンを使用する必要性について検討します。

これは異なるルールフローグループ全体で多くのルールが存在するためです。ルールが **working memory** および該当するルールフローのダウンストリームにある他のルールを変更し、必要性を再評価しなければならない場合、**modify** を使用することが重要となります。しかし、同じルールフローグループの他のルールがお互いにアクティベーションを再帰的に配置しないようにしてください。

この場合、ルールがルール自体を再帰的にアクティベートできないようになるため、no-loop 属性では効果がありません。したがって、lock-on-active を使用してください。

## 2. この問題に対応する方法はいくつかあります。

- すべてのファクトを **working memory** にアサートできる場合に **from** を使用しないようにするか、制約式でネストされたオブジェクト参照を使用します (下記を参照)。
- **modify block** で使用するために割り当てられた変数を、左側の条件の最後にある文として配置します。
- 同じルールフローグループ内のルールがお互いにアクティベーションを配置する方法を明示的に管理できる場合、lock-on-active を使用しないようにします。

この中で推奨される解決法は、**working memory** へすべてのファクトを直接アサートできる場合に **from** の使用を最小限に抑える方法になります。

上記の例では、**Person** および **Address** インスタンスの両方を **working memory** にアサートすることが可能です。グラフは比較的単純であるため、次のようにルールを変更するとさらに簡単です。

```
rule "Assign people in Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person(address.state == "QLD" )
then
  modify ($p) {} #Assign person to sales region 1 in a modify
block
end

rule "Apply a discount to people in the city of Brisbane"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person(address.city == "Brisbane" )
then
  modify ($p) {} #Apply discount to person in a modify block
end
```

## 3. 両方のルールは想定通り実行されるようになりますが、この方法で常にネストされたファクトへアクセスできるわけではありません。**Person** が1つまたは複数の **Addresses** を保持し、*存在数量詞* を使用して特定条件を満たす住所を最低でも1つ person に一致させたい例を考えてみましょう。この場合、コレクションを推論するため **from** を使用する必要があります。



これを実現する方法は複数ありますが、すべてで lock-on-active 使用時の問題が発生するわけではありません。たとえば、次のように **from** を使用すると、両方のルールが想定通りに実行されます。

```
rule "Assign people in Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person($addresses : addresses)
  exists (Address(state == "QLD") from $addresses)
then
  modify ($p) {} #Assign person to sales region 1 in a modify
block
end

rule "Apply a discount to people in the city of Brisbane"
  ruleflow-group "test"
  lock-on-active true
when
  $p : Person($addresses : addresses)
  exists (Address(city == "Brisbane") from $addresses)
then
  modify ($p) {} #Apply discount to person in a modify block
end
```

しかし、若干異なる方法では問題が発生します。

```
rule "Assign people in Queensland (QLD) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
when
  $assessment : Assessment()
  $p : Person()
  $addresses : List() from $p.addresses
  exists (Address( state == "QLD") from $addresses)
then
  modify ($assessment) {} #Modify assessment in a modify block
end

rule "Apply a discount to people in the city of Brisbane"
  ruleflow-group "test"
  lock-on-active true
when
  $assessment : Assessment()
  $p : Person()
  $addresses : List() from $p.addresses
  exists (Address( city == "Brisbane") from $addresses)
then
  modify ($assessment) {} #Modify assessment in a modify block
end
```

この場合、**from** を使用すると \$addresses が返されます。またこの例では、可能な解決法を示す新しいオブジェクト **assessment** が導入されます。\$addresses 変数が移動され、各ルールの最後の条件となると、両方のルールは想定通り実行されます。

上記の例は、ルールのアクティベーションを損失せずに **from** を lock-on-active と組み合わせて使用する方法を表していますが、左側の条件の配置順序に依存するという欠点があります。またこれにより、問題を起こす可能性のある条件を追跡する必要があるため、ルールの作成が極めて複雑になります。

4. 代わりに、ファクトをさらに **working memory** へアサートする方がよいでしょう。この場合、人の住所が **working memory** へアサートされると、**from** を使用する必要がなくなります。

#### 注記

すべてのデータを **working memory** にアサートするのは現実的ではなく、他の方法を使用する必要がある場合があります。

lock-on-active の必要性を再評価することが 1 つの方法です。**working memory** が変更された時にお互いを再帰的にアクティベートしないようにする条件が各ルールに含まれるようにして、同じルールフローグループ内のルールがお互いをアクティベートするよう、直接管理する方法もあります。たとえば、前述の例の場合、割引が既に適用されているか確認し、適用されている場合はルールがアクティベートしないようにする条件をルールに追加できます。

#### 4.8.3.9. collect 条件要素

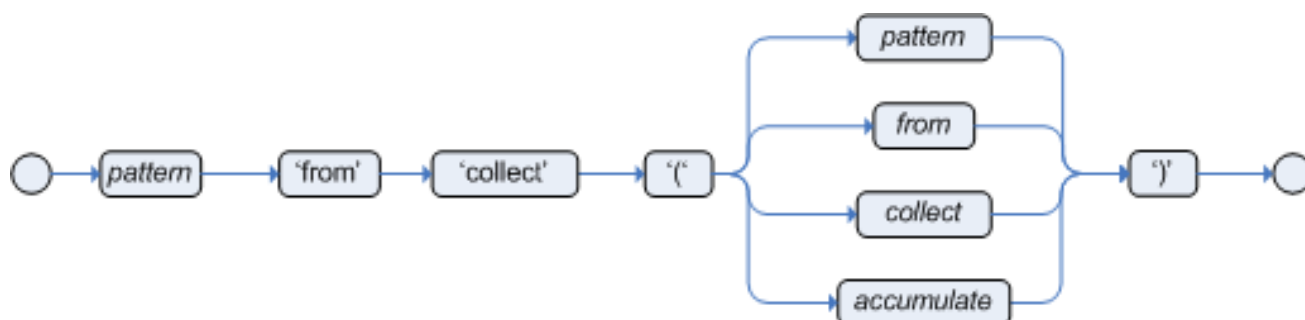


図4.42 collect

**collect conditional element** を使用して、指定のソースまたは **working memory** のいずれかより取得されたオブジェクトのコレクションをルールが推定するようにします。

#### 注記

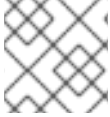
1 階論理では *基数数量詞* と呼ばれます。

```

import java.util.ArrayList
rule "Raise priority if system has more than 3 pending alarms"
when
    $system : System()
    $alarms : ArrayList( size >= 3 )
    from collect( Alarm( system == $system, status == 'pending' ) )
then
    # Raise priority, because system $system has
    # 3 or more alarms pending. The pending alarms
    # are $alarms.
end
  
```

この場合、ルールは **working memory** で該当する各システムの保留のアラームを探し、**ArrayLists** でグループ化します。1 つのシステムに対して 3 つ以上のアラームが見つかった場合にルールが実行されます。

**collect** の結果パターンは、**java.util.Collection** インターフェースを実装し、引数のないデフォルトの公開コンストラクターを提供する「具体的」なクラスになります。これは **java.util.Collection** を実装し、これらの条件を満たす限り、**ArrayList**、**LinkedList**、**HashSet**、またはカスタムクラスなどの Java 使用できることを意味します。



#### 注記

ソースと結果パターンの両方を他のパターンとして制約できます。

**collect conditional element** の前にバインドされた変数は、ソースと結果パターン両方のスコープ内にあります。このような変数を使用してこれらのパターンを制約します。しかし、**collect** はバインディングの **scope delimiter** であるため、内部のバインディングは外部では使用できません。

**collect** はネストされた **from conditional elements** を許可します。そのため、以下は **collect** の有効な使用例になります。

```
import java.util.LinkedList;
rule "Send a message to all mothers"
when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
    from collect(
        Person( gender == 'F', children > 0 )
        from $town.getPeople()
    )
then
    # send a message to all mothers
end
```

#### 4.8.3.10. accumulate 条件要素

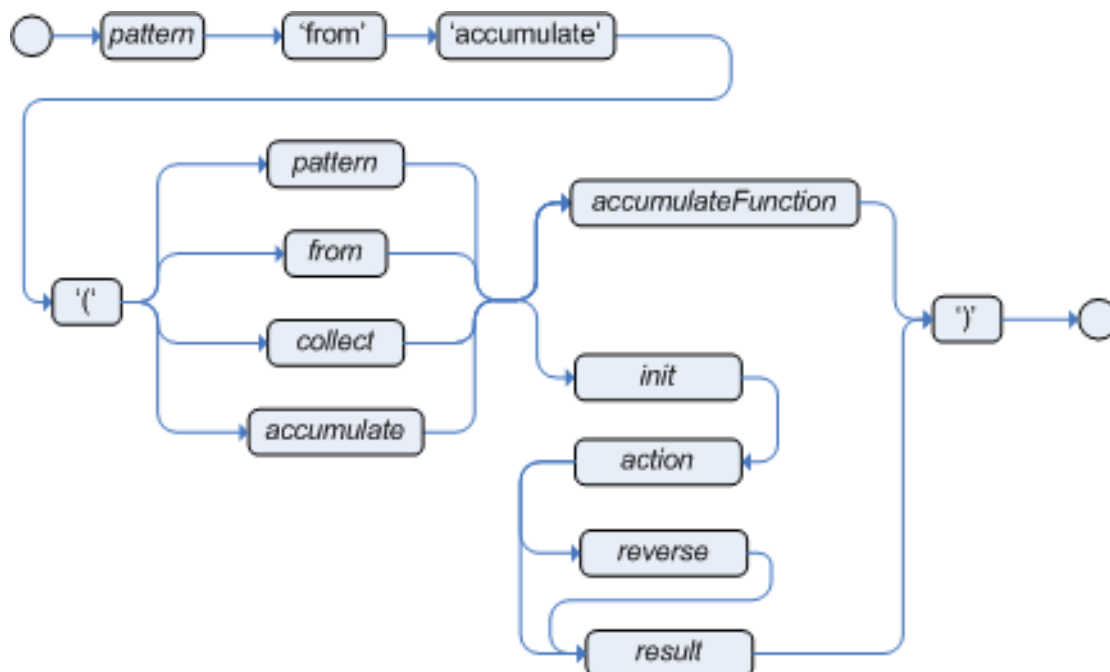


図4.43 accumulate

**accumulate conditional element** は **collect** よりも柔軟で強力な条件要素です。ルールはオブジェクトのコレクションにまたがって繰り返しでき、各要素に対してカスタムのアクションを実行できます。終了すると結果オブジェクトを返します。

**accumulate conditional element** の一般的な構文は次の通りです。

```
<result pattern> from accumulate(<source pattern>,
    init( <init code> ),
    action( <action code> ),
    reverse( <reverse code> ),
    result( <result expression> ) )
```

各要素の意味は次のようになります。

- **<source pattern>**: **engine** が各ソースオブジェクトとの一致を試みる通常のパターンです。
- **<init code>**: 選択されたダイレクトのコードにある意味論ブロックです。ソースオブジェクトで繰り返しする前に各タプルに対して一度実行されます。
- **<action code>**: 各ソースオブジェクトに対して実行される選択されたオブジェクトにあるコードの意味論ブロックです。
- **<reverse code>**: 選択されたダイレクトにあるコードの任意の意味論ブロックです。存在する場合、ソースパターンと一致しなくなった各ソースオブジェクトに対して実行されます。ソースオブジェクトが変更または取り消された時に **engine** が減分計算を行えるよう、**<action code>** ブロックで実行された計算を元に戻すことが目的です。このような操作のパフォーマンスを大幅に向上します。
- **<result expression>**: すべてのソースオブジェクトが繰り返し処理された後に実行される選択されたダイレクトの意味論の式です。

- **<result pattern>**: **engine** が **<result expression>** から返されたオブジェクトと一致しようとする通常のパターンです。一致する場合、**accumulate** 条件要素が **true** として評価し、**engine** はルールにある次の条件要素を評価します。

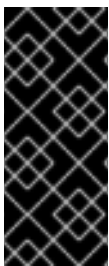
一致しない場合、**accumulate** 条件要素は **false** として評価し、ルールの条件要素を評価することを停止します。

例は次の通りです。

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
    from accumulate( OrderItem( order == $order, $value : value ),
        init( double total = 0; ),
        action( total += $value; ),
        reverse( total -= $value; ),
        result( total ) )
then
    # apply discount to $order
end
```

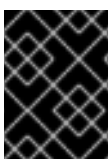
この場合、以下が発生します。

1. **engine** は **working memory** の各 **order** に対する **init** のコードを実行します。これは、**total** 変数をゼロに初期化します。
2. 次に、該当する注文に対する **OrderItem** オブジェクトすべてで繰り返し処理を行い、各オブジェクトに対してアクションを実行します (この場合、全商品の合計値が計算され、**total** 変数に置かれます)。
3. **result expression** に対応する値が返されます (**total** 変数の値)。
4. エンジン結果を **Number** パターンに一致しようとします。double の値が 100 よりも大きい場合にルールが実行されます。



### 重要

この例では Java が意味論のダイアレクトとして使用されています。そのため、**init**、**action**、および **reverse** コードブロックではセミコロンをステートメントの区切り文字として使用しなければならないことに注意してください。結果は式であり、**;** は許可されません。他のダイアレクトを使用する場合は、常に特定の構文に従ってください。



### 重要

**reverse code** は任意ですが、**update** および **retract** の使用時にパフォーマンスが向上するため、Red Hat は **reverse code** の使用を強く推奨します。

**accumulate conditional element** を使用してソースオブジェクト上のアクションを実行します。次項の例は、カスタムオブジェクトをインスタンス化し、値を投入します。

#### 4.8.3.10.1. Accumulate 関数

**accumulate conditional element** は非常に強力な CE ですが、*累積関数* と呼ばれる事前定義された関数を使用する場合、特に簡単に使用することができます。累積関数は **accumulate** とほぼ同様に動作しますが、**accumulate conditional element** すべてにカスタムコードを明示的に記述する代わりに、一般的な操作に対して事前定義されたコードを使用できることが異なります。

例は次の通りです。これは、**accumulate** 関数を用いて、次のように注文に割引を適用するルールをプログラミングできることを表しています。

```
rule "Apply 10% discount to orders over US$ 100,00"
when
  $order : Order()
  $total : Number( doubleValue > 100 )
  from accumulate( OrderItem( order == $order, $value : value ),
    sum( $value ) )
then
  # apply discount to $order
end
```

この場合、**sum** は **accumulate** 関数です。名前の通り、各 **OrderItem** の **\$value** の合計を計算し、結果を返します。

**JBoss Rules** には以下の **accumulate** 関数が内蔵されています。

- **average**
- **min**
- **max**
- **count**
- **sum**

このような一般的な関数は、式を入力として受け入れます。たとえば、注文の全商品の平均利益を計算するには、次のように **average** 関数を使用してルールを記述します。

```
rule "Average profit"
when
  $order : Order()
  $profit : Number()
  from accumulate( OrderItem( order == $order, $cost : cost, $price :
    price )
    average( 1 - $cost / $price ) )
then
  # average profit for $order is $profit
end
```

**accumulate** 関数はすべて *プラグ可能* です。そのため、必要な場合はカスタマイズされたドメイン固有の関数を比較的簡単に **engine** へ追加することが可能です。追加後、ルールは制限なしでこれらの関数を使用できます。新しい **accumulate** 関数を実装するには、以下の手順に従います。

1. **org.drools.base.accumulators.AccumulateFunction** インターフェースを実装する Java クラスを作成します。

2. 設定ファイルに行を追加するか、システムプロパティを設定し、**engine** に新しい関数について知らせます。

次の例は、**average** 関数の実装を示しています。

```

/*
 * Copyright 2007 JBoss Inc
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * Created on Jun 21, 2007
 */
package org.drools.base.accumulators;

/**
 * An implementation of an accumulator capable of calculating average
 * values
 *
 * @author etirelli
 */
public class AverageAccumulateFunction implements AccumulateFunction
{
    protected static class AverageData {
        public int    count = 0;
        public double total = 0;
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Object createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Object context) throws Exception {

```

```

AverageData data = (AverageData) context;
data.count = 0;
data.total = 0;
}

/* (non-Javadoc)
 * @see
 * org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang
 * .Object,
 * java.lang.Object)
 */
public void accumulate(Object context,
                       Object value) {
    AverageData data = (AverageData) context;
    data.count++;
    data.total += ((Number) value).doubleValue();
}

/* (non-Javadoc)
 * @see
 * org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Ob
 * ject,
 * java.lang.Object)
 */
public void reverse(Object context,
                    Object value) throws Exception {
    AverageData data = (AverageData) context;
    data.count--;
    data.total -= ((Number) value).doubleValue();
}

/* (non-Javadoc)
 * @see
 * org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.
 * Object)
 */
public Object getResult(Object context) throws Exception {
    AverageData data = (AverageData) context;
    return new Double( data.count == 0 ? 0 : data.total / data.count );
}

/* (non-Javadoc)
 * @see
 * org.drools.base.accumulators.AccumulateFunction#supportsReverse()
 */
public boolean supportsReverse() {
    return true;
}

}

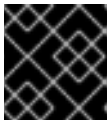
```

統合の作業はすべて **engine** によって実行されるため、コードは非常に簡単です。

- 機能を **engine** ヘプラグするには、**configuration** ファイルへ追加します。



```
drools.accumulate.function.average =
    org.drools.base.accumulators.AverageAccumulateFunction
```



### 重要

必ず `drools.accumulate.function.` プレフィックスを使用してください。

**average** は関数がルールファイルでどのように使用されるかを制御します。**org.drools.base.accumulators.AverageAccumulateFunction** は関数の動作を実装するクラスの完全修飾名です。

## 4.8.4. 右側

### 4.8.4.1. 使用法

右側は、結果またはルールのアクション部分の共通名です。ここに、実行するアクションのリストを置くことができます。



### 重要

ルールは本質的に アトミックである必要があるため、右側で命令または条件コードを使用するのは適切ではありません (「これである場合にこれをやるかもしれない」ではなく、「これである場合にこれをやる」)。

宣言的で解釈できるようにするため、ルールの右側は小さくする必要もあります。右側で命令または条件コードが必要になりそうな場合、1 つのルールを小さい複数のルールに分割することを考慮してください。

右側を使用して **working memory** のデータを挿入、取り消し、または変更します。これを行うには、最初に **working memory** を参照せずに **working memory** を変更する次の 便宜的なメソッド (convenience method) を利用します。

- **update(object, handle)** を使用して、オブジェクト (左側にあるものにバインドされている) が変更されたため、ルールの「再検討」が必要になる可能性があることを **engine** に伝えます。
- **update(object)** を使用して、ナレッジヘルパーが必要な **facthandle** をルックアップするようにします。これは、渡されたオブジェクトの ID チェックを使用して行われます (Java bean に **property change listeners** を提供する場合、**engine** に挿入するため、オブジェクト変更時に **update()** を呼び出す必要はありません)。
- **insert(new Something ())** を使用して、新しく作成されたオブジェクトを **working memory** に置きます。
- **insertLogical(new Something())** は **insert** と似ていますが、現在実行されているルールの真実をサポートするファクトがこれ以上存在しない場合に、オブジェクトが自動的に取り消される点が異なります。
- **retract(handle)** を使用して **working memory** からオブジェクトを削除します。

**convenience methods** は、実際には Knowledge Helper インスタンスヘショートカットを提供するマクロにすぎません。これを行うことにより、**rules** ファイルから **working memory** へのアクセスが可能になります。

事前定義された KnowledgeHelper 変数によって、便利なメソッドを複数呼び出すことが可能です。

- **drools.halt()** を使用して、ルールの実行を即座に終了します。これは、現在のセッションが **fireUntilHalt()** で開始された時点へ制御を戻すために行います。
- **insert(Object o)**、**update(Object o)**、および **retract(Object o)** メソッドを呼び出すこともできます (頻繁に使用されるため、オブジェクト参照なしで呼び出すことが可能です)。
- **drools.getWorkingMemory()** を使用して **working memory** オブジェクトを返します。
- **drools.setFocus( String s)** を使用して、指定された **agenda group** にフォーカスを設定します。
- **drools.getRule().getName()** を使用してルール名を返します。
- **drools.getTuple()** を使用して現在実行しているルールに一致する **タプル** を返します。**drools.getActivation()** は対応する **activation** を返します (これらの呼び出しはデバッグの処理に便利です)。

完全な **Knowledge Runtime** アプリケーションプログラミングインターフェース

は、**KnowledgeContext** 型の事前定義された変数 **kcontext** を介して公開されます。この **getKnowledgeRuntime()** メソッドが **KnowledgeRuntime** 型のオブジェクトを送信し、多数のメソッドへのアクセスを提供します。これらのメソッドは、右側の論理をコーディングする時に便利です。

- **kcontext.getKnowledgeRuntime().halt()** 呼び出しを使用して、ルールの実行を即座に終了します。
- **getAgenda()** アクセサーを使用して、このセッションの **agenda** への参照を返します。これは、さまざまなアクティベーション、アジェンダ、およびルールフローグループへのアクセスを提供します。以下のようなアジェンダグループのアクティベーションは比較的一般的な使用例になります。

```
// give focus to the agenda group Cleanup
kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "Cleanup"
).setFocus();
```



#### 注記

**drools.setFocus( "Cleanup" )** を使用して上記を実現することも可能です。

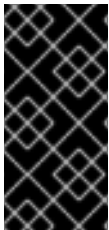
- クエリを実行するには、**getQueryResults(String query)** を呼び出します。その後、「[クエリ](#)」の通り、結果を処理することが可能です。
- **working memory** と **agenda event listeners** を追加および削除できる **event management** に対応するためのメソッドがあります。

- **getKnowledgeBase()** メソッドを使用して **KnowledgeBase** オブジェクトを返します。このオブジェクトはシステムのバックボーンで、現在のセッションの発信元です。
- **setGlobal(...)**、**getGlobal(...)**、および **getGlobals()** を用いてグローバルを管理します。
- **getEnvironment()** を使用して **run-time** の **環境** を返します (これはオペレーティングシステムの環境と同様です)。

#### 4.8.4.2. modify ステートメント

これは、**fact** の更新を実行する構造化された方法を提供する言語拡張です。更新操作と、オブジェクトのフィールドを変更する複数の **setter** 呼び出しを組み合わせます。構文スキーマは次の通りです。

```
modify ( <fact-expression> ) {
    <expression> [ , <expression> ]*
}
```



#### 重要

括弧で囲まれた **<fact-expression>** は **ファクトオブジェクト参照** を生成しなければなりません。必ず、ブロックの式リストが該当するオブジェクトの **setter** 呼び出しで構成されるようにしてください (これらは、コンパイラーによって自動的に事前終了される通常のオブジェクト参照を用いずに記述されます)。

以下は、ファクトの変更の簡単な例になります。

```
rule "modify stilton"
when
    $stilton : Cheese(type == "stilton")
then
    modify( $stilton ){
        setPrice( 20 ),
        setAge( "overripe" )
    }
end
```

## 4.9. クエリ

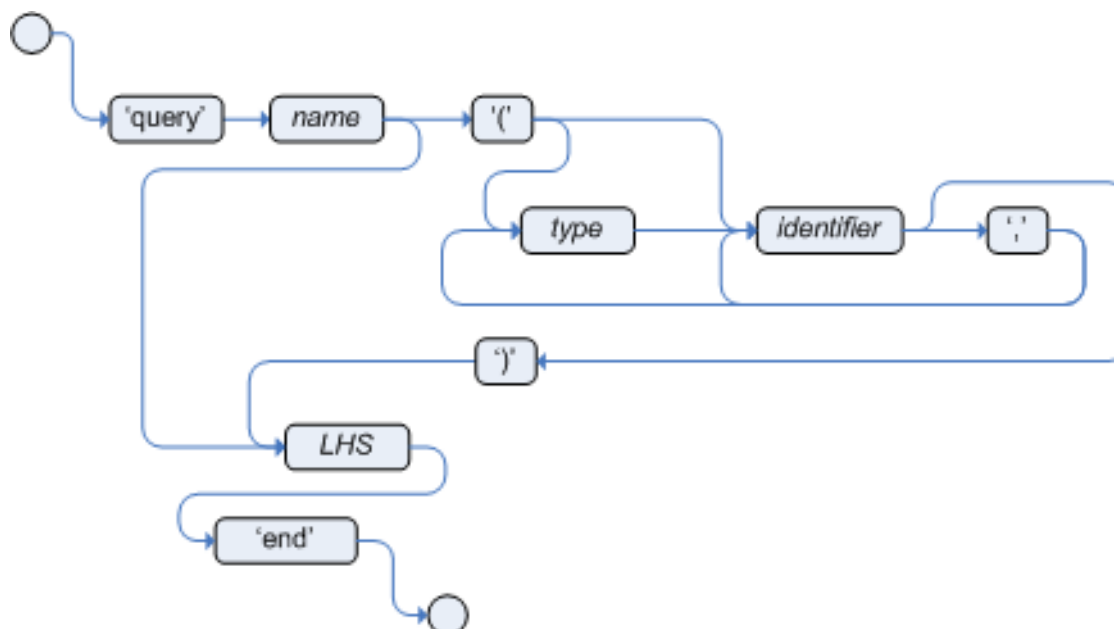


図4.44 query

クエリは、示された条件に一致するファクトをワーキングメモリーで検索する最も単純な方法です。そのため、ルール の LHS 構造のみが含まれ、「when」や「then」は指定しません。クエリには任意のパラメーターセットがあり、各パラメーターを任意に型付けすることも可能です。型が指定されていない場合、Object 型と見なされます。エンジンは必要に応じて値を強制しようとします。クエリ名は **KnowledgeBase** に対してグローバルであるため、同じ名前のクエリを同じ **RuleBase** の異なるナレッジパッケージに追加しないでください。

結果を返すには `ksession.getQueryResults("name")` を使用し、"name" はクエリ名に置き換えます。これは、クエリと一致したオブジェクトを読み出せるクエリ結果のリストを返します。

最初の例は、年齢が 30 を超える人を問い合わせする単純なクエリです。2 つ目の例はパラメーターを使用して年齢制限と地域を組み合わせます。

#### 例4.19 年齢が 30 を超える人を問い合わせるクエリー

```

query "people over the age of 30"
  person : Person( age > 30 )
end

```

#### 例4.20 年齢が x を超え、y に住む人を問い合わせるクエリー

```

query "people over the age of x" (int x, String y)
  person : Person( age > x, location == y )
end

```

標準的な for ループを使用して返された `QueryResults` を繰り返し処理します。各要素は **QueryResultsRow** で、タプルの各列へアクセスするために使用します。これらの列は、バインドされた宣言名または索引の位置よりアクセスできます。

#### 例4.21 年齢が 30 を超える人を問い合わせるクエリー

```

QueryResults results = ksession.getQueryResults( "people over the age of
30" );
System.out.println( "we have " + results.size() + " people over the age
of 30" );

System.out.println( "These people are are over 30:" );

for ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( "person" );
    System.out.println( person.getName() + "
" );
}

```

## 4.10. ドメイン固有言語

ドメイン固有言語 (DSL) は、問題があるドメイン専用のルール言語を作成する方法です。DSL 定義のセットは、DSL 「センテンス」 から DRL コンストラクトへの変換で構成され、基礎となるすべてのルール言語とエンジン機能の使用を可能にします。DSL では、ルールを DSL ルール (DSLRL) ファイルに記述し、これらのファイルは DRL ファイルに変換されます。

DSL ファイルと DSLRL ファイルはプレーンテキストファイルで、テキストエディターで作成または変更できます。DSL と DSLRL は、統合開発環境と **Business Rules Management System** の Web ユーザーインターフェースの両方で使用できます。

### 4.10.1. ドメイン固有言語の使用

ドメイン固有言語には以下の利点があります。

- ルールの作成と、**engine** が操作するドメインオブジェクトを区別するレイヤーとして機能します。これは、ドメインの専門家 (ビジネスアナリストなど) によってルールが読まれ、検証される必要がある場合に便利です。DSL は実装の詳細を隠し、ルール論理に集中します。
- DSL センテンスは、ルールで繰り返し使用される条件要素と結果アクションのテンプレートとしても機能します。
- DSL の実行時、ルールエンジンは影響を受けません。DSL はコンパイル時の機能であり、特別なパーサーとトランスフォーマーを必要とします。

### 4.10.2. ドメイン固有言語の作成

ドメイン固有言語の開発を開始する時に、次の点を考慮してください。

- ドメイン固有言語の作成には、技術者とドメインの専門家の協力が必要となります。
- 最初に、大きさと複雑さを把握するために、アプリケーションが必要なルールの例を記述します。
- ルールにおける類似のステートメントと繰り返しのステートメントを特定し、変数の部分をパラメーターとしてマーク付けします。
- 言語が開発されたら、ルールをテストします。

- アプリケーションのデータモデルがデータ型をファクトとして表す場合、ルールの記述は通常容易になります。
- 行の最初に大なり記号 (">") を付けて条件要素とアクションを DRL 形式のままにすると、初期設計段階で条件とアクションに関する実装の決定を延期することができます (これはデバッグのステートメントを挿入する際にも便利です)。
- 既存の DSL 定義を再使用したり、既存の条件または結果エンタリにパラメーターを追加したりしてルールを記述することが可能です。
- DSL エンタリの数をできるだけ少なくするようにしてください。パラメーターを使用すると、似たようなルールパターンや制約に同じ DSL センテンスを適用することができます。

### 4.10.3. ドメイン固有言語の管理

ドメイン固有言語の設定はプレーンテキストファイルに保存されます。

DSL のメカニズムにより、条件式と結果アクションをカスタマイズできます。グローバル置換メカニズムの **キーワード** も使用できます。

```
[when]Something is {color}=Something(color=="{color}")
```

上記の例には以下が適用されます。

- **[when]** キーワードは、式のスコープを示します (ルールの LHS (左側) または RHS (右側) に有効であるかどうかなど)。
- 括弧で囲まれたキーワードの後の部分は、ルールで使用する式になります。通常は自然言語の式ですが、自然言語以外の式でも可能です。
- 最初の等号 = の右部分は、式のルール言語へのマッピングになります。この文字列の形式は、宛先、RHS、または LHS によって異なります。LHS の場合、通常の LHS 構文に基づいた用語となり、RHS の場合は Java ステートメントとなることがあります。

DSL パーサーが DSL で記述されたルールファイルの行と DSL 定義の式を照合する場合、文字列操作の 3 つの手順が実行されます。最初に、式の括弧で囲まれた変数名が含まれる場所に表示される文字列値が抽出されます (たとえば **{color}** など)。次に、括弧で囲まれた名前がマッピングの右側にある場合に、これらのキャプチャーから取得された値が補間されます。最後に、DSL ルールファイルの行にある式全体と一致する値を、補間された文字列が置き換えます。

式 (等号の左側にある文字列など) は、DSL ルールファイルの行に対するパターンマッチング操作の正規表現として使用され、行のすべてまたは一部と一致します。つまり、**?** を使用して先行する文字が任意であることを示すことができます。これにより、DSL の自然言語のフレーズの変異を解消できます。ただし、これらの式は正規表現のパターンであるため、Java のパターン構文の **マジック** 文字はすべてバックスラッシュ (\) でエスケープする必要があります。



#### 注記

コンパイラーは DSL ルールファイルを行ごとに変換することに注意してください。上記の例では、「something is」の後から行の最後までテキストがすべて {colour} の置換値としてキャプチャーされ、ターゲット文字列を補間するために使用されます。

異なる DSL 式をマージし、複合の DRL パターンを生成するには、複数の独立した操作で DSLR 行を変換する必要があります。キャプチャーされた値が必ず文字テキスト (単語や単一の文字) で囲まれるようにしてこれを行います。パーサーによって実行されたマッチング操作は、行からサブ文字列を抽出しま

す。次の例では、引用符が示差的な文字として使用されています。キャプチャーされた値を囲むために使用する文字は補間中には含まれず、文字間の内容のみが含まれます。

ルールエディターが入力するテキストデータに対して引用符を使用します。キャプチャーされた値を単語で囲んで、テキストが適切に一致するようにすることも可能です。

例は次の通りです。

```
[when]This is "{something}" and "{another}"=Something(something=="{something}", another=="{another}")
[when]This is {also} valid=Another(something=="{also}")
```

ドメイン固有言語の式で句読点 (引用符以外) を使用しないようにしてください。DSL を使用するルール作成者は句読点を簡単に忘れてしまうことがあり、一部の句読点 (括弧、ピリオド、および疑問符) は DSL 定義をエスケープするために必要になります。

DSL マッピングでは、キャプチャーの結果となる変数定義や参照を囲むためのみに波括弧 ( { および } ) を使用する必要があります。波括弧をそのまま式や右側の置換テキスト内で使用したい場合は、前にバックスラッシュ ( \ ) を付けてエスケープする必要があります。

```
[then]do something= if (foo) \{ doSomething(); \}
```



#### 注記

波括弧 ( { および } ) を DSL 定義の置換文字列で表示する場合は、バックスラッシュ ( \ ) でエスケープする必要があります。

DSL ルールの行からプレーンテキストをキャプチャーし、拡張で文字列リテラルとして使用したい場合は、マッピングの右側で引用符を使用する必要があります。

#### 例4.22 DSL マッピングエントリー

```
#This is a comment to be ignored.
[when]There is a Person with name of "{name}"=Person(name=="{name}")
[when]Person is at least {age} years old and lives in "
{location}"=Person(age >= {age}, location=="{location}")
[then]Log "{message}"=System.out.println("{message}");
[when]And = and
```

例4.22 「DSL マッピングエントリー」を前提とすると、次の例はさまざまな DSLR スニペットの拡張を表しています。

#### 例4.23 DSL マッピングエントリーの拡張

```
There is a Person with name of "kitty"
==> Person(name="kitty")
Person is at least 42 years old and lives in "Atlanta"
==> Person(age > 42, location="Atlanta")
Log "boo"
==> System.out.println("boo");
```

```
There is a Person with name of "Bob" and Person is at least 30 years old
and lives in "Atlanta"
==> Person(name="kitty") and Person(age > 30, location="Atlanta")
```

#### 4.10.4. ファクトへの制約の追加

ルール条件を記述する場合、制約の任意の組み合わせをパターンへ追加できることが一般的な要件になります。ファクト型に多くのフィールドが存在する場合があるため、組み合わせごとに個別の DSL ステートメントを提供することは非常に困難になります。

DSL 機能を使用すると、ハイフン (-) を DSL 式の先頭に追加することでパターンに制約を追加することができます。式がハイフンで始まる場合、式はフィールド制約と見なされ、その前にある最後のパターン行に追加されます。

たとえば、**Cheese** クラスに type、price、age、および country のフィールドがある場合、次のように左側の条件の一部を通常の **DRL** ファイルに表現することが可能です。

##### 例4.24 DRL の LHS 条件

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

[例4.25「制約の追加」](#) の DSL 定義により 3 つの DSL フェーズが生成され、これらのフィールドに関係する制約の組み合わせを作成するために使用することができます。

##### 例4.25 制約の追加

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

次のように条件を持つルールを記述することができます。

##### 例4.26 制約の記述

```
There is a Cheese with
- age is less than 42
- type is 'stilton'
```

**parser** は - で始まる行を見つけ、制約として先行のパターンに追加し、必要な場合にコンマを挿入します。[例4.26「制約の記述」](#) の例では、結果となる DRL は次のようになります。

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

すべての数値フィールドをすべての関係演算子と組み合わせると (前の例の DSL 「age is less than」に従う)、多くの DSL エントリーが生成されます。下記のように、DSL フレーズはさまざまな演算子と、フィールド制約を処理する汎用的な式に対して定義することができます (式の定義には変数名以外に正規表現が含まれることに注意してください)。



```
[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()
```

これらの定義は条件をテキストで記述することが可能であること意味します (is less than など)。



#### 注記

個別の DSL 式が同じ行を順に照合する場合、DSL のエントリーの順序が重要になります。



#### 注記

フィールド制約のフィルタリングされたリストを **Context Assistant** で表示させるには、**-** を押した後に **Ctrl+Space** を押してこのリストから項目を選択します。

最初の項目のドメイン固有コードを `[when][org.drools.Cheese]- age is less than {age}` のように変更します。同じことを、上記の例の全項目に対して行います。

追加の `[org.drools.Cheese]` コードは、文が直上の主要な制約にのみ適用されることを表しています (この場合は **There is a Cheese with**)。

たとえば、**Cheese** というクラスが存在し、**Content Assistance** を用いた方法で追加される場合、**com.yourcompany.Something** オブジェクトスコープでマーク付けされた項目のみが有効となり、リストに表示されます。これは完全に任意となります。

### 4.10.5. DSL および DSLR 参照

DSL ファイルは行試行形式のテキストファイルです。エントリーは、DSLR ファイルを DRL 構文のファイルへ変換するために使用されます。

- `#` または `//` 始まる行はコメントとして扱われます (先頭に空白文字がある場合とない場合の両方)。`#/` で始まるコメント行は、デバッグオプションを要求する単語に対してスキャンされません。
- 左角括弧 `[` で始まる行は DSL エントリー定義の最初の行と見なされます。
- 他の行はすべて先行する DSL エントリー定義に付加され、行端が空白文字に置き換えられます。

DSL エントリーは次の 4 つの部分で構成されます。

- スコープの定義
  - `[condition]` または `[when]`
  - `[consequence]` または `[then]`
  - `[keyword]` (`rule` または `end` など)。

キーワードは、グローバルであるかどうか (DSL ファイル全体で認識されるなど)、エンタリーのスコープを示します。

- Java クラス名として記述されたタイプ定義は括弧で囲まれます。次の部分が左括弧で始まる場合を除き、この部分は任意となります。括弧の中が空の場合は無効になります。
- DSL 式は (Java) 正規表現で構成され、任意数の組み込み **点数定義** を持ち、等号 = で終了します。変数定義は波括弧 { および } で囲まれます。変数定義は変数名と 2 つの任意のアタッチメントで構成され、コロン : で区切られます。アタッチメントが 1 つある場合、変数に割り当てられるテキストを一致する正規表現となります。2 つのアタッチメントがある場合、最初のアタッチメントは GUI エディターのヒントで、2 つ目が正規表現になります。

正規表現で magic である文字を式の中で文字通り使用する場合、その文字の前にバックスラッシュ \ を付けてエスケープする必要があることに注意してください。

- 区切り等号の後にある行の残りの部分は、正規表現に一致するすべての DSLR テキストの置換テキストになります。これには、波括弧で囲まれた変数名など、変数参照が含まれるようにすることもできます。任意で、変数名の後に感嘆符 (!) と変換関数を使用することもできます。

置換文字列内で波括弧 { および } を文字通り使用する場合、括弧の前にバックスラッシュ \ を付けてエスケープする必要があることに注意してください。

DSL 拡張のデバッグは #/ で始まるコメント行を使用して選択的に有効にすることができます。このコメント行には下表の単語が 1 つ以上含まれることがあります。結果となる出力は標準出力に書き込まれます。

表4.2 DSL 拡張のデバッグオプション

単語	説明
result	結果となる DRL テキストを行番号と共に出力します。
steps	条件および結果行の各拡張ステップを出力します。
keyword	スコープ <b>keyword</b> を持つすべての DSL エントリーの内部表現をダンプします。
when	スコープ <b>when</b> または * を持つすべての DSL エントリーの内部表現をダンプします。
then	スコープ <b>then</b> または * を持つすべての DSL エントリーの内部表現をダンプします。
usage	すべての DSL エントリーの使用統計を表示します。

以下は DSL 定義の例になります。コメントは言語機能の説明になります。

```
# Comment: DSL examples

#/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
```

```
[keyword][]regula=rule

# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
    ${entity!lc}: ${entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][]update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

#### 4.10.6. DSLR ファイルのトランスフォーメーション

DSLR ファイルのトランスフォーメーションの順序は次の通りです。

1. テキストがメモリーへ読み込まれます。
2. 各 **keyword** エントリーがテキスト全体に適用されます。最初に、空白文字シーケンスを任意数の空白文字と一致するパターンに置き換え、変数定義をデフォルトの ".\*?" または定義で提供される正規表現より作成されたキャプチャーに置き換え、キーワード定義からの正規表現が変更されます。次に、DSLR テキストにて、変更された正規表現と一致する文字列が徹底的に検索されます。さらに、変数キャプチャーに対応する一致する文字列のサブ文字列が抽出され、対応する置換テキストの変数参照を置き換えます。DSLR テキストの一致する文字列はこのテキストによって置き換えられます。
3. **when** と **then** の間にある DSLR テキストは、均一に行ごとに検索されます。**then** と **end** の間にある DSLR テキストは、均一に行ごとに処理されます。

行では、行のセクションに関する各 DSL エントリーが、DSL ファイルに記述されている順番通りに取得されます。正規表現の部分は変更されます。空白文字は任意数の空白文字と一致するパターンによって置き換えられ、正規表現を持つ変数定義はこの正規表現を持つキャプチャーによって置き換えられます (デフォルトは \*.?\* になります)。結果となる正規表現が行のすべてまたは一部と一致すると、一致した部分が適切に変更された置換テキストによって置き換えられます。

変数参照を正規表現のキャプチャーに対応するテキストに置き換えることで置換テキストが変更されます。このテキストは、変数参照に提供される文字列変換関数に従って変更することができます。

同じエントリーで定義されない変数を命名する変数参照がある場合、その名前の変数にバインドされた値をエキスパンダーが置換します (現在のルールの前にある行の 1 つに定義されている場合)。

4. 条件の DSLR 行がハイフンで始まる場合、拡張された結果は最後の行に挿入されます。最後の行にはパターン CE (例: 型名の後に左右の括弧が続く) が含まれる必要があります。括弧内が空白である場合、拡張された行 (有効な制約が含まれている) が挿入されます。これ以外の場合、コンマ (,) がその前に挿入されます。

結果の DSLR 行がハイフンで始まる場合、拡張された結果は最後の行に挿入されます。最後の行には **modify** ステートメントが含まれるはずで、左右の波括弧 { および } で終わります。括弧内が空白である場合、拡張された行 (有効なメソッド呼び出しが含まれる) が挿入されます。これ以外の場合、コンマ (,) がその前に挿入されます。



## 注記

現在、ハイフンで始まる行を使用して他の条件要素形式 (**accumulate** など) にテキストを挿入することは **できません**。最初の挿入のみ実行されることはあります (**eval** など)。

### 4.10.7. 文字列トランスフォーメーション関数

表4.3「文字列トランスフォーメーション関数」 にすべての文字列トランスフォーメーション関数が記載されています。

表4.3 文字列トランスフォーメーション関数

名前	説明
uc	すべての文字を大文字に変換します。
lc	すべての文字を小文字に変換します。
ucfirst	最初の文字を大文字に変換し、他の文字はすべて小文字に変換します。
num	すべての数字と - を文字列から抽出します。元の文字列の最後にある 2 つの数字の前に . または , がある場合、少数点に対応する場所に挿入されます。
a?b/c	文字列と文字列 <b>a</b> を比較し、同等であれば <b>b</b> と置き換え、同等でなければ <b>c</b> と置き換えます。ただし、 <b>c</b> は別の <b>a</b> 、 <b>b</b> 、 <b>c</b> の 3 つ組である可能性もあるため、構造全体が実際にはトランスレーションテーブルになります。

次の DSL の例は、文字列トランスフォーメーション関数の使用方法を表しています。

#### 例4.27 DSL 文字列トランスフォーメーション関数

```
# definitions for conditions
[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()
[when][]- with an? {attr} greater than {amount}={attr} <= {amount!num}
[when][]- with a {what} {attr}={attr} {what!positive?>0/negative?
%lt;0/zero?==0/ERROR}
```

DSL 定義が含まれるファイルには、慣例的に **.dsl** が拡張子として使用されます。これは、**ResourceType.DSL** で Knowledge Builder へ渡されます。DSL 定義を使用するファイルでは、**.dslr** を拡張子として使用すべきです。Knowledge Builder は **ResourceType.DSLR** を想定します。しかし、IDE はファイル拡張子に依存してルールファイルを正しく認識し、ルールファイルと動作します。

DSL は、DSL を使用するルールファイルよりも前に Knowledge Builder へ渡す必要があります。

```
KnowledgeBuilder kBuilder = new KnowledgeBuilder();
Resource dsl = ResourceFactory.newClassPathResource( dslPath, getClass()
);
kBuilder.add( dsl, ResourceType.DSL );
Resource dslr = ResourceFactory.newClassPathResource( dslrPath, getClass()
);
kBuilder.add( dslr, ResourceType.DSLR );
```

DSL ファイルの解析および拡張では、DSL 設定が読み取られ、パーサーへ提供されます。パーサーは DSL 式を認識し、ネイティブのルール言語式に変換します。

#### 4.10.8. BRMS および IDE におけるドメイン固有言語

**Guided Editor** を使用してルールを開発する場合でも、ドメイン固有言語を使用することができます。



##### 重要

**Guided Editor** は複雑な式を処理できないことがあるため、できるだけ簡単な式を使用するようにしてください。

**Guided Editor** を使用すると、小さなデータキャプチャーのテキストフィールド「形式」を定義できます (ドメイン固有言語式を選択すると、GUI に項目が追加され、**{token}** へのみデータの入力が可能です)。

パッケージが BRMS でビルドされる時にドメイン固有言語は自動的に追加されます。

等号開発環境にドメイン固有言語を追加するには、**drools-ant** タスクを使用するか、[「XML ルール言語」](#) のようにコードを組み込みます。

### 4.11. XML ルール言語

JBoss Rules は任意で「ネイティブ」XML ルール言語を DRL の代替としてサポートします。これにより、ルールを XML データとしてキャプチャーおよび管理することができます。非 XML の DRL 形式と同様に、XML 形式は内部「AST」表現へできるだけ早く (SAX パーサーを使用) 解析されます。外部のトランスフォーメーション手順は必要ありません。すべての機能は DRL が使用できる XML で使用可能です。

#### 4.11.1. XML を使用する場合

XML の使用が推奨されるケースは複数あります。ただし、XML は人間が判読することが難しく、視覚的に肥大化したルールを作成できるため、デフォルトとして選択しないことが推奨されます。

他にも、入力 (プログラミングで生成されたルール) よりルールを生成するツールがある場合や、他のルール言語または XML を出力する他のツールから交換するツールがある場合に XML を使用したいことがあります (XSLT を使用すると XML 形式同士を簡単に変換できます)。通常の DRL は常に生成することができます。

また、すでに設定で XML を使用する製品に JBoss Rules を組み込むこともできるため、ルールを XML 形式にしたいこともあります。XML で独自のルール言語を作成することができます。独自のルール言語を作成するため、常に AST オブジェクトを直接使用することに注意してください (オープンアーキテクチャーであるため多数のオプションがあります)。

#### 4.11.2. XML 形式

XML 言語を定義する完全な W3C 標準 (XML スキーマ) に準拠する XSD が提供されます (ここでは繰り返し説明しません)。言語の概要は次の通りです。

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

<package name="com.sample"
  xmlns="http://drools.org/drools-4.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/drools-4.0 drools-4.0.xsd">

  <import name="java.util.HashMap" />
  <import name="org.drools.*" />

  <global identifier="x" type="com.sample.X" />
  <global identifier="yada" type="com.sample.Yada" />

  <function return-type="void" name="myFunc">
    <parameter identifier="foo" type="Bar" />
    <parameter identifier="bada" type="Bing" />
    <body>System.out.println("hello world");</body>
  </function>

  <rule name="simple_rule">
    <rule-attribute name="salience" value="10" />
    <rule-attribute name="no-loop" value="true" />
    <rule-attribute name="agenda-group" value="agenda-group" />
    <rule-attribute name="activation-group" value="activation-group" />

    <lhs>
      <pattern identifier="foo2" object-type="Bar" >
        <or-constraint-connective>
          <and-constraint-connective>
            <field-constraint field-name="a">
              <or-restriction-connective>
                <and-restriction-connective>
                  <literal-restriction evaluator=">" value="60" />
                  <literal-restriction evaluator="<" value="70" />
                </and-restriction-connective>
                <and-restriction-connective>
                  <literal-restriction evaluator="<" value="50" />
                  <literal-restriction evaluator=">" value="55" />
                </and-restriction-connective>
              </or-restriction-connective>
            </field-constraint>

            <field-constraint field-name="a3">
              <literal-restriction evaluator="==" value="black" />
            </field-constraint>
          </and-constraint-connective>

          <and-constraint-connective>
            <field-constraint field-name="a">
              <literal-restriction evaluator="==" value="40" />
            </field-constraint>

            <field-constraint field-name="a3">
```

```

        <literal-restriction evaluator="==" value="pink" />
      </field-constraint>
    </and-constraint-connective>

    <and-constraint-connective>
      <field-constraint field-name="a">
        <literal-restriction evaluator="==" value="12"/>
      </field-constraint>

      <field-constraint field-name="a3">
        <or-restriction-connective>
          <literal-restriction evaluator="==" value="yellow"/>
          <literal-restriction evaluator="==" value="blue" />
        </or-restriction-connective>
      </field-constraint>
    </and-constraint-connective>
  </or-constraint-connective>
</pattern>

<not>
  <pattern object-type="Person">
    <field-constraint field-name="likes">
      <variable-restriction evaluator="==" identifier="type"/>
    </field-constraint>
  </pattern>

  <exists>
    <pattern object-type="Person">
      <field-constraint field-name="likes">
        <variable-restriction evaluator="==" identifier="type"/>
      </field-constraint>
    </pattern>
  </exists>
</not>

<or-conditional-element>
  <pattern identifier="foo3" object-type="Bar" >
    <field-constraint field-name="a">
      <or-restriction-connective>
        <literal-restriction evaluator="==" value="3" />
        <literal-restriction evaluator="==" value="4" />
      </or-restriction-connective>
    </field-constraint>
    <field-constraint field-name="a3">
      <literal-restriction evaluator="==" value="hello" />
    </field-constraint>
    <field-constraint field-name="a4">
      <literal-restriction evaluator="==" value="null" />
    </field-constraint>
  </pattern>

  <pattern identifier="foo4" object-type="Bar" >
    <field-binding field-name="a" identifier="a4" />
    <field-constraint field-name="a">
      <literal-restriction evaluator="!=" value="4" />
      <literal-restriction evaluator="!=" value="5" />
    </field-constraint>
  </pattern>

```

```

        </field-constraint>
      </pattern>
    </or-conditional-element>

    <pattern identifier="foo5" object-type="Bar" >
      <field-constraint field-name="b">
        <or-restriction-connective>
          <return-value-restriction evaluator="==" >
            a4 + 1
          </return-value-restriction>
          <variable-restriction evaluator=">" identifier="a4" />
          <qualified-identifier-restriction evaluator="==">
            org.drools.Bar.BAR_ENUM_VALUE
          </qualified-identifier-restriction>
        </or-restriction-connective>
      </field-constraint>
    </pattern>

    <pattern identifier="foo6" object-type="Bar" >
      <field-binding field-name="a" identifier="a4" />
      <field-constraint field-name="b">
        <literal-restriction evaluator="==" value="6" />
      </field-constraint>
    </pattern>
  </lhs>
  <rhs>
    if ( a == b ) {
      assert( foo3 );
    } else {
      retract( foo4 );
    }
    System.out.println( a4 );
  </rhs>
</rule>

</package>

```

最初の XML テキストには一般的な XML 要素、パッケージ宣言、インポート、グローバル、関数、およびルール自体があります。JBoss Rules の機能についてある程度理解している場合は、説明がなくてもほとんどの要素について理解できるはずです。

**import** 要素はルールで使用したい型をインポートします。

**global** 要素はルールで参照できるグローバルオブジェクトを定義します。

**function** にはルールで使用される関数のための関数宣言が含まれます。戻り型、一意の名前、およびパラメーターをコードのスニペットのボディーに指定する必要があります。

ルールは以下で説明します。

#### 例4.28 rule 要素の詳細

```

<rule name="simple_rule">
  <rule-attribute name="salience" value="10" />
  <rule-attribute name="no-loop" value="true" />
  <rule-attribute name="agenda-group" value="agenda-group" />

```



```

<rule-attribute name="activation-group" value="activation-group" />

<lhs>
  <pattern identifier="cheese" object-type="Cheese">
    <from>
      <accumulate>
        <pattern object-type="Person"></pattern>
        <init>
          int total = 0;
        </init>
        <action>
          total += $cheese.getPrice();
        </action>
        <result>
          new Integer( total );
        </result>
      </accumulate>
    </from>
  </pattern>

  <pattern identifier="max" object-type="Number">
    <from>
      <accumulate>
        <pattern identifier="cheese" object-
type="Cheese"></pattern>
        <external-function evaluator="max" expression="$price"/>
      </accumulate>
    </from>
  </pattern>
</lhs>
<rhs>
  list1.add( $cheese );
</rhs>
</rule>

```

上記の規則の詳細では、規則には LHS と RHS (条件と結果) のセクションがあります。RHS は単純で、規則がアクティベートされた時に実行される意味論コードのブロックになります。LHS は若干複雑で、条件要素、制約、および制限のネストされた要素が含まれます。

LHS の主要な要素は Pattern 要素です。これにより、型 (クラス) を指定したり、そのクラスのインスタンスへ変数をバインドすることができます。満たされなければならない制約や制限はパターンオブジェクト下でネストされます。述部や戻り値の制約により、Java 式を組み込むことができます。

これにより、not、exists、and などの条件要素が残ります。これらは DRL のように挙動します。以下にネストされた要素と「and」要素は論理的に「and」によって結合されます。「or」も同様です (さらにネストすることができます)。「Exists」と「Not」はパターンを回避して、パターンの制約を満たすファクトが存在するかどうかをチェックします。

Eval 要素を使用すると、ブール値を評価する限り、Java コードの有効なスニペットを実行することが可能です (断片であるため、ブール値をセミコロンで終了しないでください)。これに、関数の呼び出しが含まれるようにすることも可能です。ルールエンジンが毎回評価を行うため、Eval は列よりも非効率的ですが、列の制約で何を行う必要があるかを表現できる時の「キャッチオール」機能です。

#### 4.11.3. 形式 (XML と DRL) 間の自動転換

JBoss Rules には、ある形式から他の形式へ変換するためのユーティリティークラスがいくつか含まれています。これは、ルールをソース形式から AST へ解析し、適切なターゲット形式へ「ダンプ」することによって機能します。これにより、ルールを DRL で記述し、必要に応じて将来的に XML にエクスポートすることが可能です。

この作業を行う必要がある時に注意すべきクラスは次の通りです。

- XmlDumper - XML のエクスポート。
- Dr1Dumper - DRL のエクスポート
- Dr1Parser - DRL の読み取り。
- XmlPackageReader - XML の読み取り。

上記のクラスを組み合わせると、任意の形式間で変換を行うことが可能です (ラウンドトリップを含む)。DSL は保持されませんが (DSL を使用する DRL から)、変換できることに注意してください。

XSLT を自由に使用して、さまざまな XML の可能性を提供してください。XSLT やその種類は XML を強化します。

## 第5章 スプレッドシートのデシジョンテーブルの使用

本章を読んで、デシジョンテーブルの使用方法について学びましょう。

デシジョンテーブルは *条件論理* を表現する手段で、ビジネス レベルのルールを表現するタスクに適しています。

**JBoss Rules** は、**CSV** や **.XLS** などのスプレッドシート形式でルールを保存し、ルールを管理できるようにします。

**JBoss Rules** はデシジョンテーブルを使用して、スプレッドシートに入力されたデータから生じるルールを生成します。データキャプチャやデータ操作など、スプレッドシートの通常機能をすべて利用して、このようなデータセットを構築できます。

### 5.1. デシジョンテーブルを使用する場合

テンプレートやデータとして表現できるルールが存在する場合、デシジョンテーブルの使用を考慮してください。デシジョンテーブルの各行でデータが収集され、収集されたデータとテンプレートを組み合わせてルールが生成されます。

該当のルールセットがテンプレートに従わない場合や、ルールが少ない場合はデシジョンテーブルを使用しないでください。また、これは個人的な好みの問題でもあります。スプレッドシートアプリケーションを使用したいユーザーもいれば、使用したくないユーザーもいます。

また、デシジョンテーブルはユーザー基盤のオブジェクトモデルを隔離できますが、これが望ましい場合と望ましくない場合があります。

### 5.2. 概要

デシジョンテーブルの一例は次の通りです。

B17    Catastrophic Claim				
1	2			
12		B	C	D
16		Type of New Claim	Is case catastrophic	Allocation code
17		Catastrophic Claim	Y	
18		New Claim with previous Accident num		2
19		Previous Open claim		1
20		Dependency Claim		8
21		Dependency Claim		9
22		Interstate Claim		A
23		Interstate Claim		D
24		Interstate Claim		N
25		Interstate Claim		S

図5.1 Excel を使用してデシジョンテーブルを編集

	J	K	L
ner	Allocate to Team	Stop processing	Log reason
	Team Red	Stop processing	The claim was catastrophic

図5.2 ルール行の複数のアクション

The screenshot shows the OpenOffice.org Calc interface with a spreadsheet titled 'Catastrophic Claim'. The spreadsheet has columns B through G. Row 16 is the header row with the following content:

	B	C	D	E	F	G
16	Type of New Claim	Is case catastrophic	Allocation code	Claim Type	Insurance Class	Date of accident is after
17	Catastrophic Claim	Y				
18	New Claim with previous Accident num		2			
19	Previous Open claim		1	P		
20	Dependency Claim			8		
21	Dependency Claim			9		
22	Interstate Claim			A		
23	Interstate Claim			D		
24	Interstate Claim			N		
25	Interstate Claim			S		
26	Interstate Claim			T		

The status bar at the bottom shows 'Sheet 1 / 2', 'PageStyle\_Tables', '100%', 'STD', and 'Sum=0'.

図5.3 OpenOffice.org Calc の使用

**注記**

上記の例では、デシジョンテーブルの技術的な部分は隠されています (標準的なスプレッドシートの機能)。

ルールは 17 行から始まります (各行は 1 つのルールになります)。条件は C、D、E 列などにあります (アクションは画面外になります)。セルにある値は比較的単純で、16 行のヘッダーを確認すると意味が分かります (B 列は説明です)。

**注記**

色を使用してテーブルの異なる領域の意味を示すと便利な場合があります。

**重要**

デシジョンテーブルは上から下へデータを処理するように見えますが、必ずしもそうであるとは限りません。順番が関係ないようにルールを記述することが推奨されます (これにより維持が簡単になり、行を頻繁に移動する必要がなくなるため)。

各行は 1 つのルールを表し、同じ原則が適用されます。ルールエンジンがファクトを処理すると、一致するルールが実行されます。これに戸惑うユーザーもいます。ルールが実行された時に **agenda** を消去し、最初の一致が存在する場所に変な単純なデシジョンテーブルをシミュレートすることが可能です。デシジョンテーブルは基本的に DRL パッケージを自動的に生成するツールです。

**注記**

1 つのスプレッドシートに複数のテーブルを使用することも可能です。共通のテンプレートを共有する場合にルールをグループ化でき、最終的にすべてが 1 つのルールパッケージにまとめられるため便利です。

1	2	3	4	5	6
1					
2	Module	PRSC[02]			
3	RuleSet	Control Cajas[1]			
8					
9	1.ValidarAperturaCaja (Caja, Registro Estado Sucursal, Transaccion)				
	ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas
13			1	2000	ValidarAperturaCajaSucursal Abierta
14			2	2000	ValidarAperturaCajaMismaFecha
15					
16					
17					
18	2.ValidarCierreCajasSucursal(Registro Estado Sucursal, TransaccionCaja)				
	ID_Caso de Uso	Caso de Uso	Identificadores de las Reglas	Prioridades de las Reglas	Nombres de las Reglas
22	C_PRSC_503 C_PRSC_504 C_PRSC_513		1	1000	ValidarCierreCajasSucursal
23					
24					
25	3.ValidarTransaccionCaja(Caja, Transaccion_Caja)				
26	RuleTable[3] ValidarTransaccionCaja(CajaVO caja, MovimientoCajaVO movimientoCaja)				
27	ID_Caso de Uso	Caso de Uso	Identificador	Prioridad	Nombre
28					

図5.4 複数のテーブルを使用して同様のルールをグループ化する実例

### 5.3. デシジョンテーブルの仕組み



#### 重要

デシジョンテーブルでは各行が1つのルールであり、その行の各列はそのルールの条件またはアクションになることに留意することが重要となります。

1	2						
12		B	C	D	E	F	G
16		Type of New Claim	Is case catastrophic	Allocation code	Each column may be a condition, or action etc.	Insurance Class	Date of accident is after
17		Catastrophic Claim	Y				
		New Claim with previous Accident num		2			
Each row results in a rule							
20		Dependency Claim					
21		Dependency Claim					
22		Interstate Claim					
23		Interstate Claim					
24		Interstate Claim					
25		Interstate Claim					

図5.5 行と列

スプレッドシートは、ルールテーブルの開始行と開始列を示す *RuleTable* キーワードを探します。(他のパッケージレベル属性を定義するために使用される他のキーワードは本章で後ほど説明します)。キーワードを1つの列に収めることが重要になります。慣習的に、これには2つ目の列("B")が使用されますが、どの列でも使用できます(慣習としては左側に注意書き用の余白を残します)。下図では、Cが実際の開始列になります。この列の左側はすべて無視されます。



## 注記

分かりやすい場合は隠れたセクションを展開します。

C 列にキーワードがあることに注意してください。

	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	Cheese	ACTION
15		Person		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	
22				

図5.6 ルールテンプレートの展開

**RuleSet** キーワードは、ルールがすべてグループ化される **rule package** で使用される名前を示します (名前は任意です。デフォルト値もありますが、**RuleSet** キーワードが存在する必要があります)。C 列に表示される他のキーワードは **Import** と **Sequential** です。これらについては本章で後ほど説明します。この段階では、通常、キーワードが名前と値のペアを作成します。

従うルールのグループを示し、これらのルールが一部のルールテンプレートのベースとなるため、**RuleTable** は重要になります。

**RuleTable** キーワードの後には名前があります。この名前は生成されたルール名の先頭に追加するために使用されます (一意のルール名を作成するために行番号が付加されます)。**RuleTable** の列はルールが開始する列を示します (左側の列は無視されます)。

14 行の **CONDITION** および **ACTION** キーワードは、下の列にあるデータがルールの LHS または RHS 部分であることを示しています (このように任意に設定できる他の属性もあります)。

15 行には **ObjectTypes** の宣言が含まれます。この行の内容は任意で、内容を使用したくない場合は行を空白のままにします。この行が使用されると、下のセル (16 行) の値がそのオブジェクトタイプの制約になります。上記の例では、**Person(age=="42")** (42 は 18 行から取得) が生成されます。この場合、**==** は暗黙的で、フィールド名のみを指定すると完全一致の検索と見なされます。



## 注記

**ObjectType** の宣言は複数の列にまたがることができます (マージされたセルにより)。そのため、マージされた範囲の下にあるすべての列が 1 つの制約セットにまとめられます。

16 行には、ルールテンプレート自体が含まれます。**\$para** プレースホルダーを使用して、セルからのデータが投入される場所を示します。**\$param** または **\$1**、**\$2** などを使用して下のセルにあるコンマ区切りリストからのパラメーターを示します。

17 行はルールテンプレートの説明テキストであるため、無視されます。

18 行から 19 行は、実際のルールを生成するために 15 行のテンプレートと組み合わせられる (補間される) データを示します。セルにデータがない場合は、テンプレートが無視されます。ルール行は空白行の前まで読み取られます。1 つのシートに複数の **RuleTables** を使用できます。

20 行には別のキーワードと値が含まれます。このようなキーワードの行位置は重要ではありませんが、上部に配置することが推奨されます。しかし、列は **RuleTable** または **RuleSet** キーワードが表示される場所と同じである必要があります (この例では、C 列を選択しましたが、代わりに A 列を選択することも可能です)。

上記の例では、ルールは以下のようにレンダリングされます (**ObjectType** 行が使用されるため)。

```
//row 18
rule "Cheese_fans_18"
  when
    Person(age=="42")
    Cheese(type=="stilton")
  then
    list.add("Old man stilton");
  end
```

[age=="42"] と [type=="stilton"] は上のセルにある対象の **ObjectType** に追加される単一の制約として解釈されます (上のセルが複数にまたがる場合は、1 つの「列」に対して複数の制約が存在することがあります)。

## 5.4. キーワードと構文

### 5.4.1. テンプレートの構文

テンプレートで使用する構文は、列が **CONDITION** または **ACTION** であるかによって異なります。ほとんどの場合、LHS (**CONDITION** の場合) または RHS (**ACTION** の場合) の「vanilla」DRL と同じです。そのため、LHS では制約言語を使用する必要があり、RHS は実行目的のコードのスニペットになります。

セルのデータが補間される場所を示すため、テンプレートで **\$param** プレースホルダーが使用されます。**\$1** を使用することも可能です。セルに値のコンマ区切りリストが含まれる場合、\$1 や \$2 などのシンボルを使用して、セルの値リストよりどの位置パラメーターを使用するか示すことができます。**forall(DELIMITER){SNIPPET}** 関数を使用して、使用できるコンマ区切りの値をすべてループすることが可能です。

例は次の通りです。



テンプレートが `[Foo(bar == $param)]` で、セルが `[ 42 ]` である場合、結果は `[Foo(bar == 42)]` になります。

テンプレートが `[Foo(bar < $1, baz == $2)]` で、セルが `[42,43]` である場合、結果は `[Foo(bar > 42, baz ==43)]` になります。

条件の場合、スニペットのレンダリングは上の行に **ObjectType** 宣言が存在するかどうかによって異なります。存在する場合、スニペットは **ObjectType** の個別の制約としてレンダリングされます。存在しない場合、そのままレンダリングされます (値は置換されます)。上記の例のようにプレーンフィールドのみが入力された場合、相等であると見なされます。スニペットの最後に別の演算子がある場合、制約の最後で値が補間されます。スニペットの最後に別の演算子がない場合は、前述の通り **\$param** を探します。

結果の場合、スニペットのレンダリングはすぐ上の行に存在するものがあるかどうかによって異なります。エントリーがない場合、補間されたスニペットが出力されます。バインドされた変数やグローバルなど (上記の例のように) がある場合、オブジェクトのメソッド呼び出しとして追加されます。

例は次の通りです。

13	RuleTable Cheese fans	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton
19	21	cheddar

図5.7 またがった列

上記の例は、**Person ObjectType** 宣言がどのように2つのスプレッドシート列にまたがっているかを表しています。そのため、両方の制約は **Person(age == ... , type == ...)** として表示されます。これまでと同様に、フィールド名のみがスニペットに存在するため、等価テストを意味します。

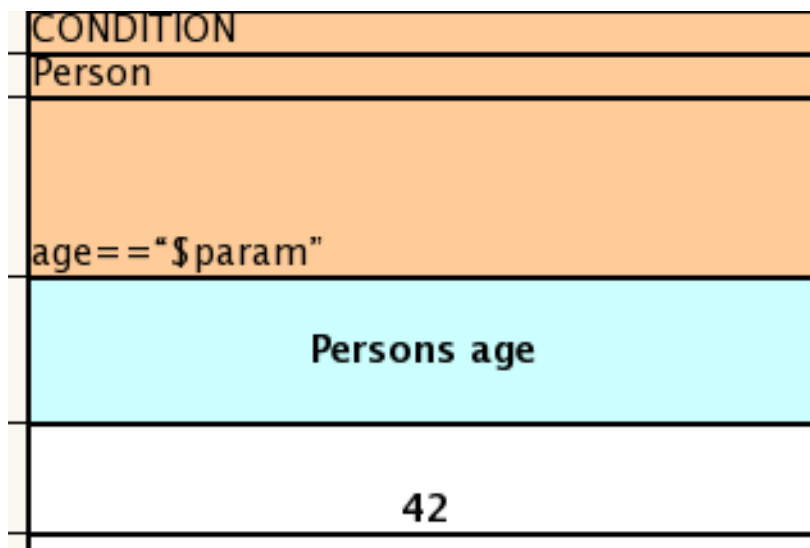


図5.8 パラメーターの使用

この例では、値をスニペットに指定するために補間が使用されています (結果は **Person(age == "42")** になります)。

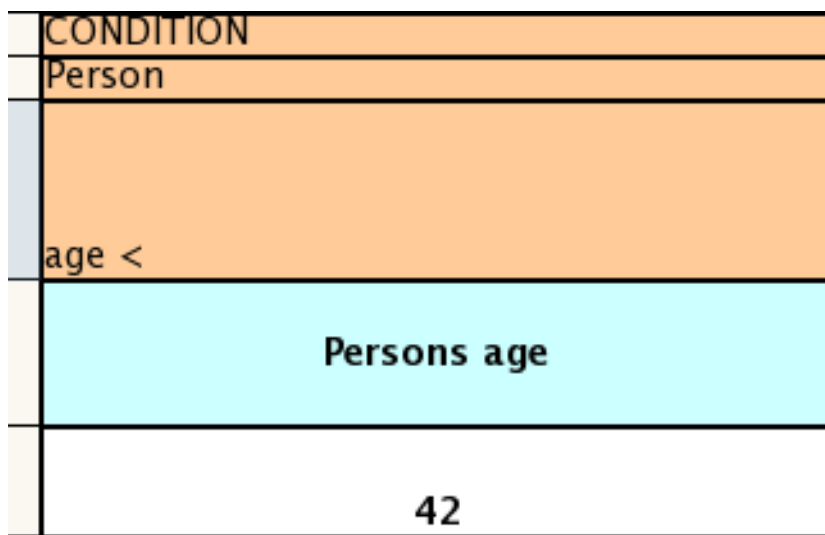


図5.9 演算子の補完

上記の条件の例は、演算子が最後に配置されると、演算子の後に値が自動的に配置されることを表しています。

CONDITION
c: Cheese
type
Cheese type
stilton

図5.10 バインディングの使用

列の前にバインディングを指定できます (下のセルから制約が追加されます)。 **ObjectType** 行にはどのような値でも入力することができます (たとえば、後続するスプレッドシートの列に対する事前条件など)。

ACTION
list.add("\$param");
Log
Old man stilton

図5.11 結果

最後の例は、結果がどのように単純な補間によって行われるかを表しています (上のセルを空白のままにします。条件の列も同様です)。 このようにして、 結果には 1 つのメソッド呼び出しだけでなく、どのような値でも入力することができます。

5.4.2. キーワード

以下の表は、ルールテーブル構造に必要なキーワードについて説明しています。

表5.1 キーワード

キーワード	説明	含まれる状態
RuleSet	この右側のセルにはルールセット名が含まれます。	1 つのみ (指定しない場合はデフォルト値が使用されます)

キーワード	説明	含まれる状態
Sequential	この右側のセルは <b>true</b> または <b>false</b> になります。true の場合は salience が使用され、ルールが上から順に実行されるようにします。	任意
Import	右側のセルには、インポートする Java クラスのコンマ区切りリストが含まれます。	任意
RuleTable	<b>RuleTable</b> はルールテーブルの定義の始まりを示します (実際のルールテーブルは次の行で開始します)。ルールテーブルは次の空白行まで、左から右、そして上から下の順に読み取られます。	最低 1 つ。複数ある場合は、1 つのルールセットにすべてが追加されます。
CONDITION	この列がルール条件用であることを示します。	ルールテーブルごとに最低 1 つ。
ACTION	この列がルール結果用であることを示します。	ルールテーブルごとに最低 1 つ。
PRIORITY	この列の値がルール行に対して salience 値を設定することを示します。Sequential フラグを上書きします。	任意
DURATION	この列の値がルール行に対して期間の値を設定することを示します。	任意
NAME	この列の値がその行から生成されたルールに対して名前を設定することを示します。	任意
Functions	すぐ右側のセルには、ルールスニペットで使用できる関数を指定できます。JBoss Rules は DRL で定義された関数をサポートし、ハードコーディングせずに論理をルールに組み込んだり、論理を変更したりすることができます。使用の際は注意して使用してください。構文は通常の DRL と同じです。	任意

キーワード	説明	含まれる状態
Variables	すぐ右側のセルには JBoss Rules がサポートするグローバル宣言を指定できます。タイプの後に変数名を指定したものです。複数の変数が必要な場合はコンマで区切ります。	任意
No-loop または Unloop	no-loop または unloop は、テーブルのヘッダーに指定され、ルール(行)のループを許可しない同じ関数を補完します。このオプションが適切に機能するには、セルに値(true または false)が存在しなければなりません。セルが空白のままであると、このオプションは行に対して設定されません。	任意
XOR-GROUP	この列のセル値は、ルール行が指定のアクティベーショングループに属することを意味します。アクティベーショングループとは、名前付きグループの 1 つのルールのみが実行されることを意味します(最初に実行されるルールが他のルールのアクティベーションをキャンセルします)。	任意
AGENDA-GROUP	この列のセル値は、ルール行が指定のアジェンダグループに属することを意味します(ルールのグループ間のフローを制御する方法の 1 つです。「ルールフロー」を参照してください)。	任意
RULEFLOW-GROUP	この列のセル値は、ルール行が指定のルールフローグループに属することを意味します。	任意
Worksheet	デフォルトでは、デシジョンテーブルで最初のワークシートだけが参照されます。	該当なし

各行に対して生成されたルールに影響する **HEADER** キーワードのユースケースは次の通りです。ほとんどの場合でヘッダー名が最も重要になります。下のセルに値がないと、特定の行に属性が適用されません。

B	C	D	E	F	G	H
1						
2	RuleSet	org.acme.insurance.base				
3	import	import org.acme.insurance.base.Approve, import org.acme.insurance.base.Driver				
4	Package	org.acme.insurance.base				
5						
6	RuleTable Old Driver					
7	CONDITION	CONDITION	RULEFLOW-GROUP	NO-LOOP	ACTION	ACTION
8	\$driver: Driver					
9	licenceYears	priorClaims			insert(new Approve("\$param"));	system.out.println( "Spa
10	base	Persons age	Prior Claims		Inserting approval	Log
11	d guy	30	1	risk assessment	Safe and mature	Old driver Approved
12						
13						
14						
15						
16						

図5.12 キーワードの使用例

以下は、**Import** (コンマ区切り)、**Variables** (グローバル、コンマ区切り)、および **function block** (複数の関数で構成されることもあり、普通の DRL 構文を使用) の例になります。**RuleSet** キーワードと同じ列に指定したり、すべてのルール行の下に指定することもできます。

<b>RuleSet</b>	<b>Control Cajas[ 1 ]</b>
<b>Import</b>	foo.Bar, bar.Baz
<b>Variables</b>	Parameters parametros, RulesResult resultado, EvalDate fecha
<b>Functions</b>	<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>

図5.13 関数などのキーワードの使用例

## 5.5. スプレッドシートベースのデシジョンテーブルの作成と統合

**drools-decisiontables** モジュールのスプレッドシートベースのデシジョンテーブルと共に使用されるアプリケーションプログラミングインターフェースを探します。1つのクラスのみが関連しますが、そのクラスが **SpreadsheetCompiler** です。このクラスはさまざまな形式のスプレッドシートを取得し、通常通りに使用できる DRL ルールを生成します。

また、**SpreadsheetCompiler** を使用して部分的なルールファイルを生成し、後で完全なルールパッケージへアセンブルすることができます (これを使用してルールの技術的な側面と非技術的な側面を区別します)。

サンプルのスプレッドシートをベースとします。または、**Rule Workbench IDE** プラグインを使用する場合はインビルドの **Wizard** を使用してテンプレートよりスプレッドシートを生成し、**XLS** 対応のスプレッドシートアプリケーションを用いて編集します。

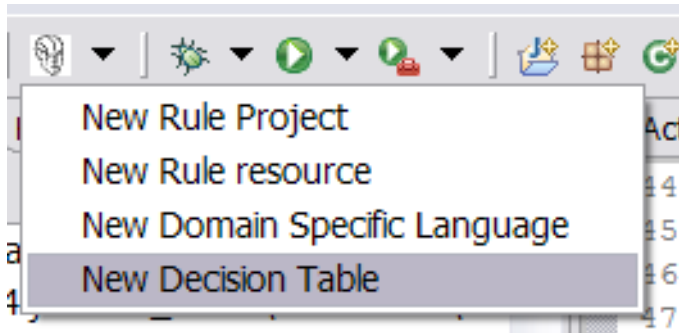


図5.14 統合開発環境の使用

## 5.6. デシジョンテーブルにおけるビジネスルールの管理

### 5.6.1. ワークフローとコラボレーション

デシジョンテーブルは、IT の専門家とドメインの専門家が密接に協力する必要がある場合に適しています。デシジョンテーブルを使用すると、ビジネスルールがアナリストにとって明確になります。

ビジネスルールを作成するには、次の手順に従います。

1. ビジネスアナリストがテンプレートのデシジョンテーブルを取得します (リポジトリまたは IT 部門から)。
2. ビジネスアナリストがデシジョンテーブルのビジネス言語の説明をテンプレートに入力します。
3. デシジョンテーブルルール (行) が入力されます (下書きとして)。
4. デシジョンテーブルが、ビジネス言語 (説明) をスクリプトへマッピングするプログラマーへ渡されます (新しいアプリケーションやデータモデルの場合、ソフトウェアの開発が関係することがあります)。
5. プログラマーがビジネスアナリストと共に変更を確認します。
6. ビジネスアナリストは必要に応じてルール行の編集を継続して行えます (列の移動など)。
7. プログラマーは、システム稼働後にルールの変更を検証するために使用するルールのテストケースを開発します。

### 5.6.2. スプレッドシート機能の使用

LibreOffice **Calc** の機能を使用すると、スプレッドシートへのデータの入力を容易にすることができます。また、他のワークシートに保存されたリストを使用して、セルの値の有効なリストを提供することができます。

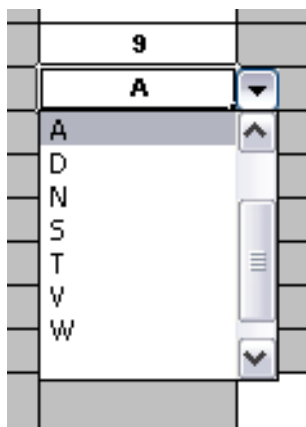


図5.15 スプレッドシートリストの使用



### 重要

Red Hat はバージョン制御システムを使用して変更履歴を維持することを推奨します。



## 第6章 JAVA ルールエンジンアプリケーションプログラミングインターフェース

### 6.1. はじめに

**JBoss Rules** は、*JSR94* の実装である **Java rule engine** アプリケーションプログラミングインターフェース (API) を提供します。複数のルールエンジンをこの単一の API で実行することができます。本章を読んで、この API の機能について詳しく学びましょう。



#### 注記

JSR94 はルール言語自体とは全く対話しません。

ルールエンジン全体の機能としては、JSR94 標準は「最小公分母」を表すことに注意してください。そのため、JSR94 API の機能は標準の **JBoss Rules** API の機能よりも少なくなります。そのため、JSR94 を使用すると **JBoss Rules** の **rule engine** によって提供される機能の一部が無効になります。

完全な機能 (**globals**、**DRL**、**DSL**、**XML** ファイルなど) へアクセスするには、**property maps** を使用します。これにより、移植不可能な機能が導入されることに注意してください。さらに、JSR94 はルール言語を提供しないため、ルールエンジンを切り替えても複雑さはほとんど解消されません。よって、ほとんど利点がありません。そのため、JSR94 の使用が要求される場合に Red Hat はサポートを行いますが、プログラマーは **JBoss Rules** API を使用することが強く推奨されます。

### 6.2. API の使用方法

JSR94 は 2 つの部分で構成されます。1 つ目は **Administrative API** で、**RuleExecutionSets** のビルドおよび登録に使用されます。2 つ目は **run-time session** で、同じ **RuleExecutionSets** を実行します。

#### 6.2.1. RuleExecutionSets のビルドと登録

**RuleServiceProviderManager** は **RuleExecutionSets** の登録と読み出しを管理します。**JBoss Rules RuleServiceProvider** 実装は、**Class.forName** を使用してクラスがロードされた時に 静的ブロックによって自動的に登録されます (これは JDBC ドライバーとほぼ同様です)。

##### 例6.1 RuleServiceProvider の自動登録

```
// RuleServiceProviderImpl is registered to "http://drools.org/"
// via a static initialization block
Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");

// Get the rule service provider from the provider manager.
RuleServiceProvider ruleServiceProvider =

RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/");
```

**RuleServiceProvider** は、**RuleRuntime** および **RuleAdministration** API へのアクセスを提供します。**RuleAdministration** は、**RuleExecutionSets** を管理するための管理 API を提供します。これにより、**RuleRuntime** より読み出しできる **RuleExecutionSet** の登録が可能になります。

**RuleExecutionSet** を作成し登録するには、次の手順に従います。

1. **RuleExecutionSet** を作成します。**RuleAdministrator** は、空の **LocalRuleExecutionSetProvider** または **RuleExecutionSetProvider** を返すファクトリメソッドを提供します。
2. **LocalRuleExecutionSetProvider** を使用して、**stream** などのローカルのシリアル化不可能なソースより **LocalRuleExecutionSetProvider** をロードします。

**RuleExecutionSetProvider** を使用して、DOM 要素やナレッジパッケージなどのシリアル化可能なソースより **RuleExecutionSets** をロードできます。



#### 注記

**ruleAdministrator.getLocalRuleExecutionSetProvider( null );** および **ruleAdministrator.getRuleExecutionSetProvider( null );** メソッドは両方 **null** をパラメーターとして許可します。これは、これらのメソッドの **properties map** は現在使用されていないからです。

#### 例6.2 RuleAdministrator API を用いた LocalRuleExecutionSet の登録

```
// Get the RuleAdministration
RuleAdministrator ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader( drlUrl.openStream() );

// Create the RuleExecutionSet for the drl
RuleExecutionSet ruleExecutionSet =
ruleExecutionSetProvider.createRuleExecutionSet( drlReader, null );
```

上記の例では、**ruleExecutionSetProvider.createRuleExecutionSet( reader, null )** が **properties map** に対して **null** のパラメーターを取ります (しかし、実際は受信ソースの設定情報を提供するために使用されます)。**null** が渡されると、**DRL** ファイルより入力を読み込むためデフォルトが使用されます。マップに使用できるキーは **source** と **dsl** です。**source** は **drl** または **xml** を値として取ります。**source** を **drl** に設定して **DRL** ファイルを読み込みます。また同様に、**source** を **xml** に設定して **XML** ファイルを読み込みます (**xml** は **dsl** のキーと値の設定をすべて無視します)。**dsl** キーは **reader** または **string** (ドメイン固有言語の内容) を値として使用します。

#### 例6.3 LocalRuleExecutionSet の登録時におけるドメイン固有言語の指定

```
// Get the RuleAdministration
RuleAdministration ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
```

```

URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader( drlUrl.openStream() );

// Create a Reader for the dsl and a put in the properties map
URL dslUrl = new URL("http://mydomain.org/sources/myrules.dsl");
Reader dslReader = new InputStreamReader( dslUrl.openStream() );
Map properties = new HashMap();
properties.put( "source", "drl" );
properties.put( "dsl", dslReader );

// Create the RuleExecutionSet for the drl and dsl
RuleExecutionSet ruleExecutionSet =
    ruleExecutionSetProvider.createRuleExecutionSet( reader, properties );

```

### 重要

**RuleExecutionSet** を読み出すために使用する名前は、登録時に指定する必要があります (プロパティを渡せるようにするためのフィールドもありますが、現在使用されていないため、**null** を渡します)。

#### 例6.4 RuleExecutionSet の登録

```

// Register the RuleExecutionSet with the RuleAdministrator
String uri = ruleExecutionSet.getName();
ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet, null);

```

### 6.2.2. 「ステートフル」 および 「ステートレス」 ルールセッションの使用

**run-time** は **RuleServiceProvider** より取得できます。これは、ステートフル および ステートレス **rule engine** セッションを作成するために使用されます。

#### 例6.5 RuleRunTime の取得

```

RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();

```

ルールセッションを作成するには、次の手順に従います。

1. **RuleRuntime** の 2 つのパブリック定数である **RuleRuntime.STATEFUL\_SESSION\_TYPE** および **RuleRuntime.STATELESS\_SESSION\_TYPE** のいずれかを使用します。
2. **RuleSession** をインスタンス化するために使用される **RuleExecutionSet** の URI (Uniform Resource Indicator) を提供します。
3. **properties map** を **null** に設定するか、**properties map** を使用してグローバルを指定します (次項に記載されています)。
4. **createRuleSession(...)** メソッドは **RuleSession** インスタンスを返します。これを **StatefulRuleSession** または **StatelessRuleSession** へキャストします。

### 例6.6 ステートフルルール

```
(StatefulRuleSession) session =
    ruleRuntime.createRuleSession( uri,
                                   null,
                                   RuleRuntime.STATEFUL_SESSION_TYPE );
session.addObject( new PurchaseOrder( "lots of cheese" ) );
session.executeRules();
```

**StatelessRuleSession** は非常に単純な API を持ちます。これを使用して **executeRules(List list)** (オブジェクトのリストを渡す) を呼び出し、任意でフィルター呼び出します。その後、結果となるオブジェクトが返されます。

### 例6.7 ステートレスルール

```
(StatelessRuleSession) session =
    ruleRuntime.createRuleSession( uri,
                                   null,
                                   RuleRuntime.STATELESS_SESSION_TYPE );

List list = new ArrayList();
list.add( new PurchaseOrder( "even more cheese" ) );

List results = new ArrayList();
results = session.executeRules( list );
```

## 6.2.3. グローバル

JSR94 でグローバルをサポートすることは可能ですが、移植は不可能になります。JSR94でグローバルをサポートするには、**properties map** を渡すメソッドを **RuleSession** ファクトリに渡します。最初に、**DRL** または **XML** ファイルにグローバルを定義し、例外がスローされないようにします。

キーは **DRL** または **XML** ファイルに宣言された識別子を表します。このキーの値は、実行で使用するインスタンスになります。次の例では、結果はグローバル **java.util.List** リストに収集されます。

### 例6.8 グローバル

```
java.util.List globalList = new java.util.ArrayList( );
java.util.Map map = new java.util.HashMap( );
map.put( "list", globalList );
//Open a stateless Session
StatelessRuleSession srs =
    (StatelessRuleSession) runtime.createRuleSession( "SistersRules",
                                                       map,

    RuleRuntime.STATELESS_SESSION_TYPE );
...
// Persons added to List
// call executeRules( ) giving a List of Objects as parameter
// There are rules which will put Objects in the List
// fetch the list from the map
List list = (java.util.List) map.get("list");
```

必ず **DRL** ファイルにグローバルリストを宣言してください。グローバルリストの宣言は次のように行います。

#### 例6.9 グローバルリスト

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list
rule FindSisters
when
    $person1 : Person ( $name1:name )
    $person2 : Person ( $name2:name )
    eval( $person1.hasSister($person2) )
then
    list.add($person1.getName() + " and " + $person2.getName() + " are
sisters");
    assert( $person1.getName() + " and " + $person2.getName() + " are
sisters");
end
```

### 6.3. 参考文献

JSR94 の詳細については、次の文献を参照してください。

- **Java Rule Engine API (JSR 94) の公式の JCP 仕様**

<http://www.jcp.org/en/jsr/detail?id=94>

- **Java Rule Engine API ドキュメント**

[http://www.javarules.org/api\\_doc/api/index.html](http://www.javarules.org/api_doc/api/index.html)

- **Jess and the javax.rules API**, E Friedman-Hill 著、TheServerSide.com 発行、2003 年

<http://www.theserverside.com/articles/article.tss?!=Jess>

- **Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications**, Q. H. Mahmoud 著、Sun Developer Network 発行、2005 年

<http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html>

- **The Logic From The Bottom Line: An Introduction to The Drools Project**, N. A. Rupp 著、TheServiceSide.com 発行、2004 年

<http://www.theserverside.com/articles/article.tss?!=Drools>

## 第7章 JBOSS DEVELOPER STUDIO

**JBoss Developer Studio** アプリケーションは **JBoss Rules** で唯一サポートされる *統合開発環境* (IDE) で、多くのプログラマーにとって便利な機能を複数提供します。本章を読んで、これらの機能の使用方法について学びましょう。



### 注記

**JBoss Rules IDE** のコンポーネントは **Eclipse** プラグインとして個別に使用することも可能です。



### 注記

**JBoss Developer Studio** はルールの記述に必要ではなく、**JBoss Rules** エンジンも **Eclipse** 環境に全く依存しません。

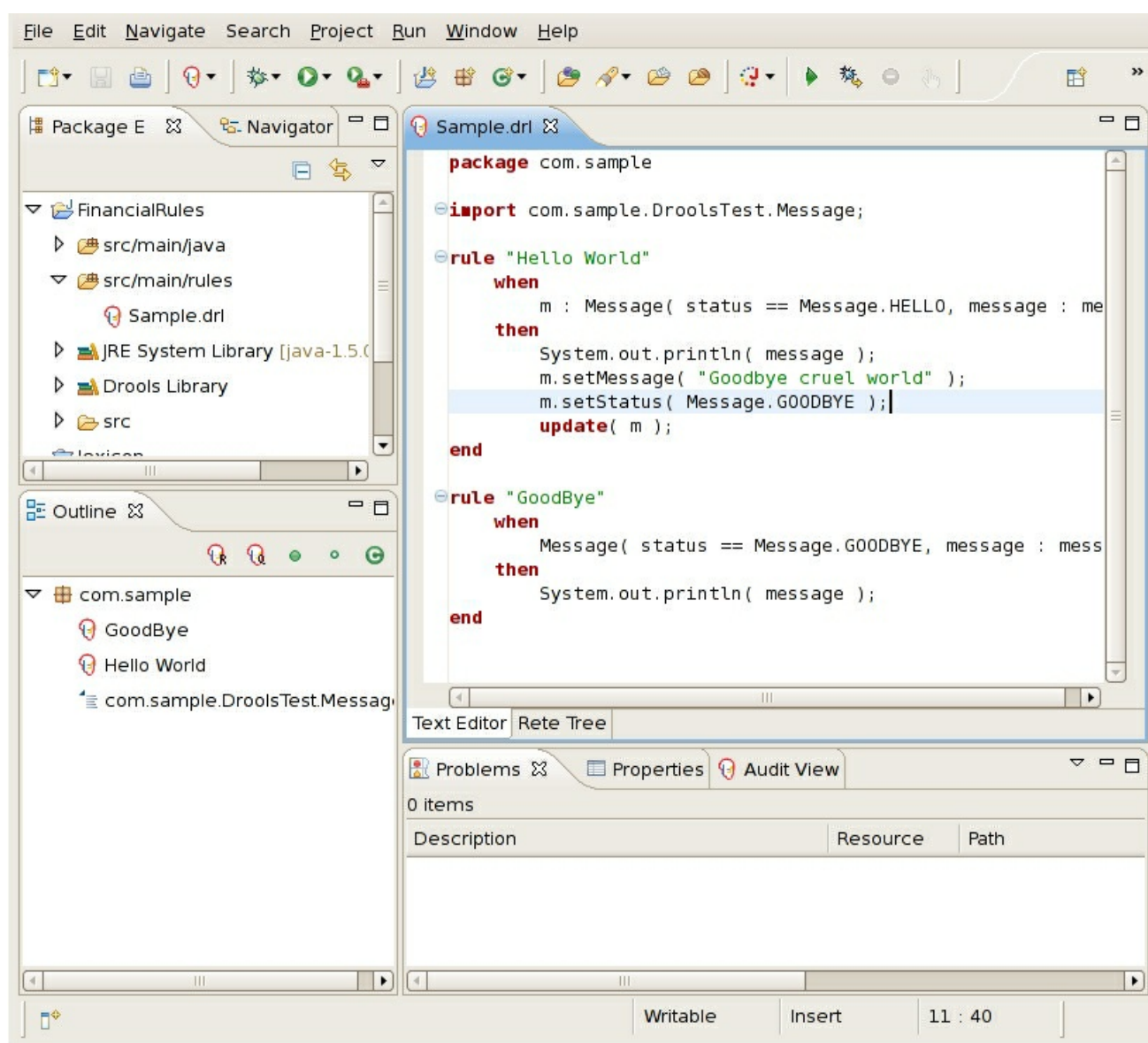


図7.1 概要

### 7.1. 概要

**JBoss Developer Studio** には以下の機能があります。

- コンテンツアシスタント 機能 (**outline view** を含む) を提供する **DRL** 構文認識エディター
- コンテンツアシスタント 機能を提供する ドメイン固有言語 拡張認識エディター
- ルールフローグラフ (プロセスを表す) を編集するための **Rule-Flow Graphical Editor**。ルールフローグラフをルールパッケージに適用し、**命令型制御** を許可できます。
- 以下を作成するウィザード
  - 「rules」プロジェクト
  - ルールリソース (**DRL** または **BRL** ファイルの形式)
  - ドメイン固有言語
  - ディシジョンテーブル
  - ルールフロー
- カスタム言語とルール言語間のマッピングを作成および管理するための **domain-specific language editor**
- ルール検証は、変更が行われるたびにルールを再構築します。**Problem View** よりエラーを報告します。

## 7.2. DROOLS ランタイム

Drools ランタイムは、Drools プロジェクト jar の 特定のリリースを 1 つ表す jar ファイルのコレクションです。ランタイムを作成するには、IDE を選択するリリースへ示す必要があります。プラグイン自体に含まれる最新の Drools プロジェクト jar に基づいて新しいランタイムを作成する必要がある場合も、簡単に行うことができます。Eclipse ワークスペースに対してデフォルトの Drools ランタイムを指定する必要がありますが、各プロジェクトはデフォルトを上書きすることができ、プロジェクトに対して適切なランタイムを選択することができます。

### 7.2.1. Drools ランタイムの定義

Eclipse 設定ビューを使用して 1 つ以上の Drools ランタイムを定義するには、「Window」メニューで「Preferences」メニュー項目を選択し、Preferences を開きます。「Preferences」ダイアログに設定がすべて表示されるはずです。このダイアログの左側にある Drools カテゴリー下で「Installed Drools runtimes」を選択します。右側のパネルに、現在定義されている Drools ランタイムが表示されるはずです。

新しい Drools ランタイムを定義するには、追加ボタンをクリックします。ダイアログが表示され、ランタイムの名前とランタイムが存在するファイルシステムの場所を入力するよう要求されます。

通常、2 つのオプションがあります。

1. Drools Eclipse プラグインに含まれるようにデフォルトの jar ファイルを使用するには、「Create a new Drools 5 runtime ...」ボタンをクリックすると新しい Drools ランタイムを自動的に作成することができます。ファイルブラウザーが表示され、このランタイムを作成するファイルシステム上のフォルダーを選択するよう求められます。プラグインが必要な依存関係を自動的に指定されたフォルダーへコピーします。このフォルダーの選択後、ダイアログは下図のようになります。

2. 特定リリースの Drools プロジェクトを使用したい場合は、必要なすべての Drools ライブラリおよび依存関係が含まれるファイルシステム上にフォルダーを作成する必要があります。上記のように、新しい Drools ランタイムを作成する代わりにランタイムに名前を割り当て、必要な jar がすべて含まれるこのフォルダーの場所を選択します。

OK ボタンをクリックした後、新規作成されたランタイムの前にあるチェックボックスをクリックします。プロジェクト固有のランタイムが選択されなかった Drools プロジェクトのランタイムとしてデフォルトの Drools ランタイムは使用されます。

Drools ランタイムは必要な数だけ追加することができます。

デフォルトのランタイムを変更した場合、デフォルトのランタイムを使用しているプロジェクトがすべてクラスパスを適切に更新するようにするには、Eclipse を再起動する必要があることに注意してください。

### 7.2.2. Drools プロジェクトに対するランタイムの選択

Drools プロジェクトを作成するたびに (New Drools Project ウィザードを使用するか、Drools で既存の Java プロジェクトを右クリックし「Convert to Drools Project」アクションを使用して既存の Java プロジェクトを変換する)、プラグインは自動的に必要な jar をすべてプロジェクトのクラスパスに追加します。

新しい Drools プロジェクトを作成する時、プロジェクト固有の Drools ランタイムが指定されている場合以外は、プラグインはそのプロジェクトのデフォルトの Drools ランタイムを自動的に使用します。これを行うには、New Drools Project ウィザードの最後の手順で「Use default Drools runtime」チェックボックスを選択解除し、ドロップダウンボックスで適切なランタイムを選択します。「Configure workspace settings ...」リンクをクリックすると、現在インストールされている Drools ランタイムを表示するワークスペース設定が開かれるため、そこに新しいランタイムを追加することができます。

プロジェクトプロパティーを開き、Drools カテゴリーを選択すると、いつでも Drools プロジェクトのランタイムを変更できます。「Enable project specific settings」チェックボックスを選択し、ドロップダウンボックスより適切なランタイムを選択します。「Configure workspace settings ...」リンクをクリックすると、現在インストールされている Drools ランタイムを表示するワークスペース設定が開かれるため、ここに新しいランタイムを追加することができます。「Enable project specific settings」チェックボックスの選択を解除すると、グローバル設定に定義された通りデフォルトのランタイムが使用されます。

## 7.3. ルールプロジェクトの作成

新しいプロジェクトウィザードの目的は、ルールを即座に使用するために実行可能なプロジェクトを設定することです。これにより、基本的な構造、クラスパス、サンプルルール、およびテストケースが設定されます。

新しいルールオブジェクトを作成する時、ルール、デシジョンテーブルおよびルールフローなどデフォルトのアーティファクトを追加するかどうかを選択します。これが土台となり、即座に実行できます。これをカスタマイズの土台として扱います。簡単な **Hello World** ルールを学習してください。



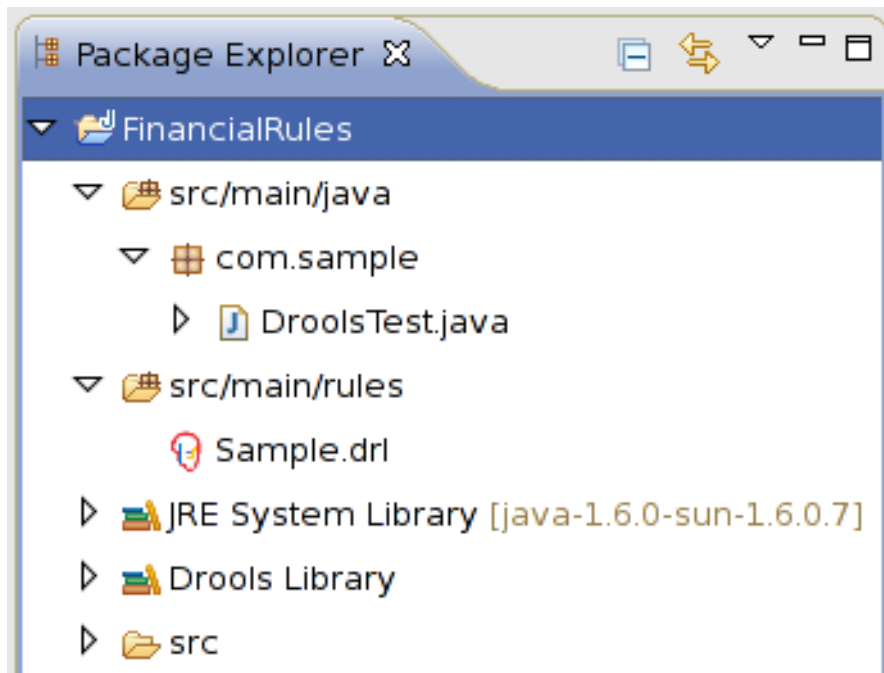


図7.2 新しいルールプロジェクトの結果

新しく作成されたプロジェクトには、サンプルルールファイル (Sample.drl) が src/rules ディレクトリに含まれ、Drools エンジンでルールを実行するために使用できるサンプル Java ファイル (DroolsTest.java) も含まれます。これは、com.sample パッケージの src/java フォルダーに含まれます。実行時に必要な他の jar はすべて Drools Library と呼ばれるカスタムクラスパスコンテナにも追加されます。ルールを「Java」プロジェクトに保持する必要はまったくありません。これは、Eclipse を Java IDE としてすでに使用しているユーザーの利便性を考慮しています。



#### 注記

厳密には、ルールを Java プロジェクトに保持する必要はありません。ここでは、**JBoss Developer Studio** を Java IDE としてすでに使用しているユーザーの利便性を考慮しています。



#### 重要

**JBoss Developer Studio** は、使用するリソースが変更されるたびに自動的にルールの再構築および検証を行う **JBoss Rules Builder** と呼ばれる機能を提供します。**Rule Project Wizard** を使用してプロジェクトが作成されると、この機能はデフォルトで有効になります。また、他のプロジェクトに対してこの機能を手作業で有効にすることも可能です。



#### 重要

ファイルに大量のルール (通常 500 個以上) が含まれる場合は、大量の処理が発生します。これは、ファイル変更のたびに、各ルールが再構築されるためです。これが問題になる場合、2つの対処法があります。最も簡単な方法は一時的にビルダーを無効にすることです。これ以外に、大きなルールを **.rule** ファイルに移動する方法もあります。これらのファイルはビルダーによって無視されますが、ファイルに含まれるルールを検証するために単体テストで実行する必要があります。

## 7.4. 新しいルールの作成とウィザード

ルールを作成するには、**.drl** ファイル拡張子を持つ空のテキストファイルを生成するか、**Wizard** を使用します。**Control+N** を押すか、ツールバーの **JBoss Rules** アイコンをクリックして **Wizard** のメニューを起動します。

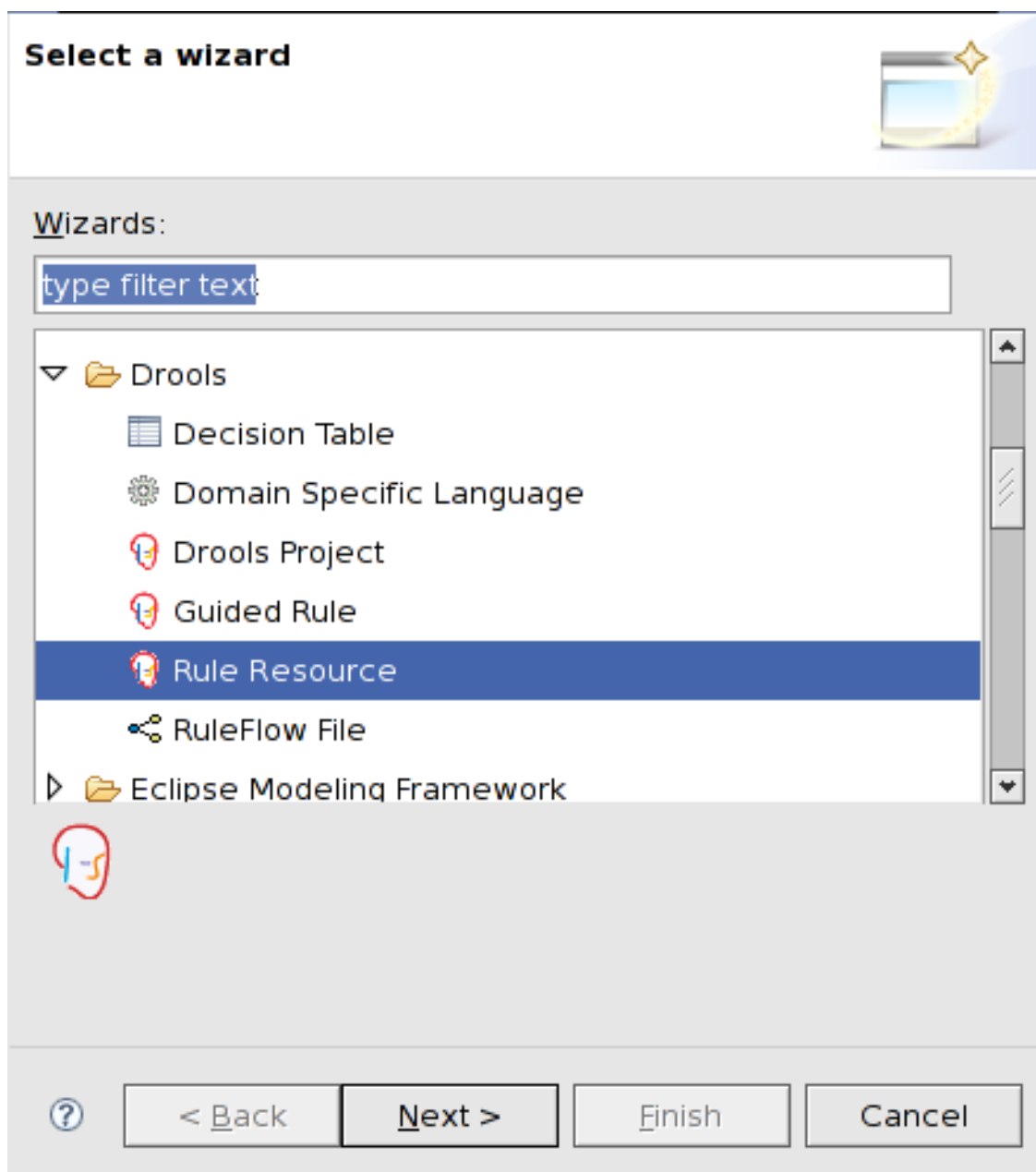


図7.3 ウィザードメニュー

**Wizard** は、ルールリソースの生成に関するオプションを提示することでユーザーに入力を求めます (何を入力すればよいか分からない場合、入力した内容を後で変更できます)。

ルールファイルを格納するため、**src/rules** というディレクトリを作成し、適切に命名されたサブディレクトリを追加します。パッケージ名は必須で、Java のパッケージ名と似ていることに注意してください (関係するルールをグループ化する名前空間を確立します)。

**New Rules File**

Hint: Press CTRL+SPACE when editing rules to get content sensitive assistance/popups.

Enter or select the parent folder:

FinancialRules

FinancialRules

File name:

Type of rule resource: New DRL (rule package) ▼

Use a DSL: ☐

Use functions: ☐

Rule package name:

Advanced >>

? < Back Next > Finish Cancel

図7.4 新しいルールウィザード

**Wizard** を実行すると、骨格またはスケルトンが作成され、「肉づけ」することが可能になります。このウィザードも他のウィザードと同様に任意のヘルパーであるため、使用したくない場合は使用する必要はありません。

## 7.5. テキストルールエディター

**Rule Editor** は、ルールマネージャーおよび開発者が最も使用するツールです。**Rule Editor** は、通常の **JBoss Developer Studio** テキストエディターの標準的な機能を備えています。その他に、「ポップアップ」コンテキストアシスタントも提供します。この機能にアクセスするには、Control キーと Space キーを同時に押します。

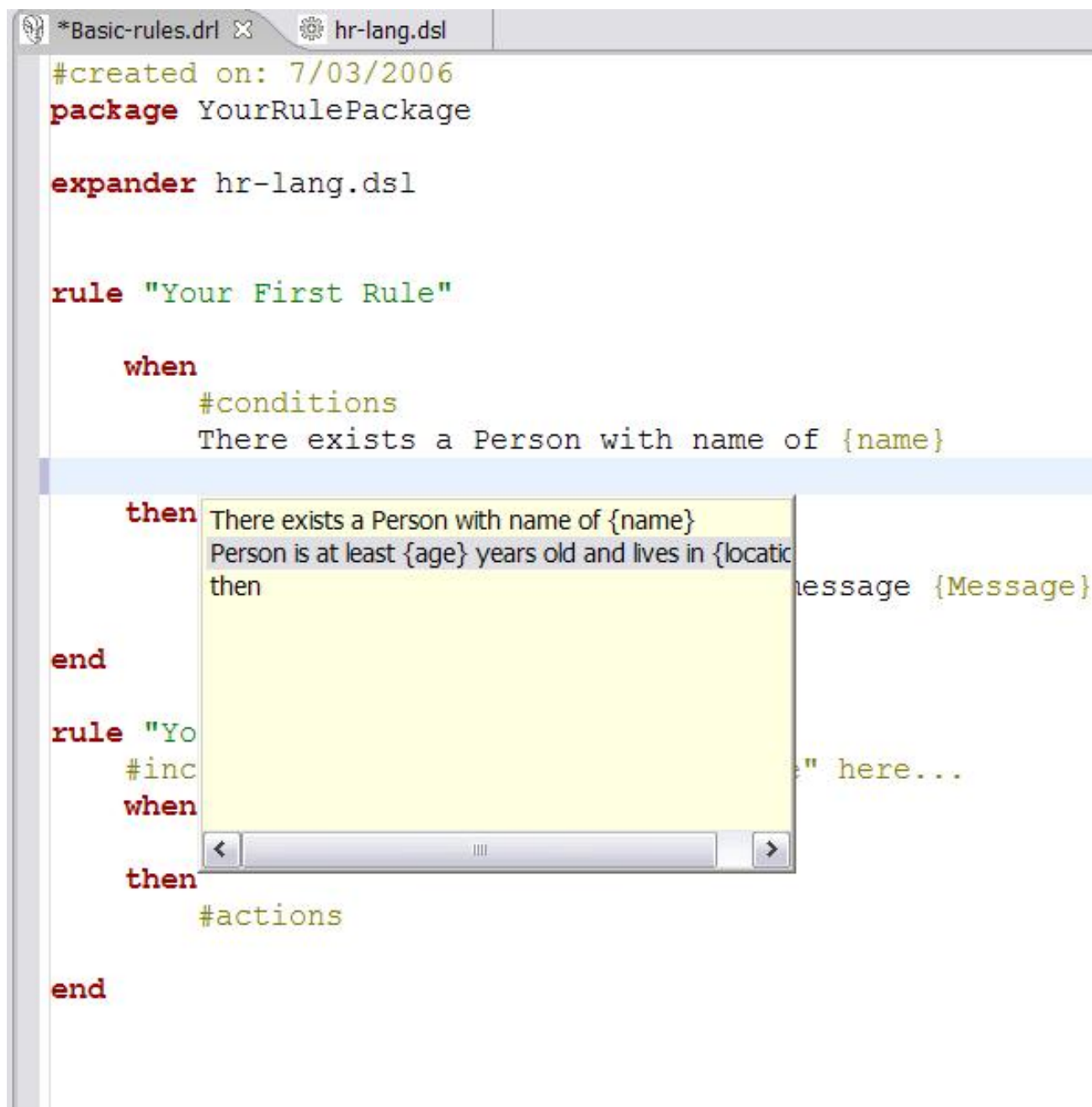


図7.5 使用中のルールエディター

**Rule Editor**は、**.drl** または **.rule** 拡張子を持つファイルを開くことができます。通常、これらのファイルには関連する複数のルールが含まれますが、個別のファイルに各ルールが含まれるようにし、同じパッケージ名前空間に存在する利点を生かしてグループ化することも可能です。



#### 注記

**DRL** ファイルのデータはプレーンテキスト形式で保存されます。

上記の例では、ルールグループはドメイン固有言語を使用しています。**expander** キーワードが存在することに注意してください。これは、ルール言語を解決するため、その名前の **.dsl** ファイルを検索するようルールコンパイラーに指示します。ドメイン固有言語が使用可能であっても、ルールはプレーンテキストとして格納され、画面上で表示できる内容を反映します。これにより、バージョンを比較する場合など、ルールの管理が非常に簡単になります。

**editor** は **outline view** を提供します。これは、ルール構造と同期化されます (ファイルが保存され

るたびに更新されます)。この機能を使用して、迅速および効率的に名前でルールを見つけることができます。これは、数百ものルールが含まれる大型のファイルで使用すると大変便利です。デフォルトでは、項目がアルファベット順に表示されます。

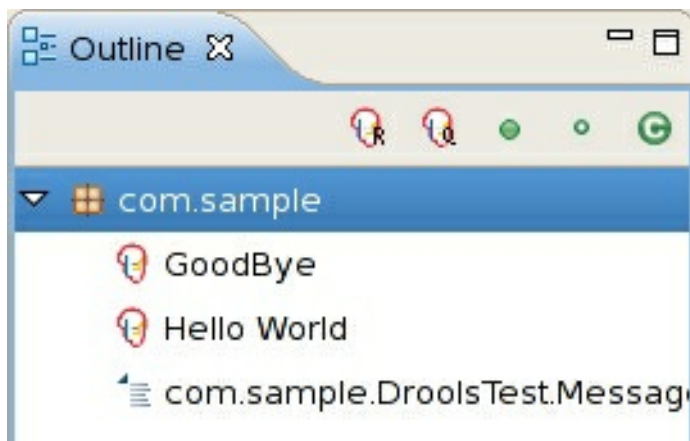


図7.6 ルールアウトラインビュー

## 7.6. ガイド付きエディター

JBoss Developer Studio には **Guided Editor** と呼ばれる機能も含まれています。この機能は BRMS で使用できる Web ベースのエディターと似ています。この機能により、ルールを視覚的に構築することができます。

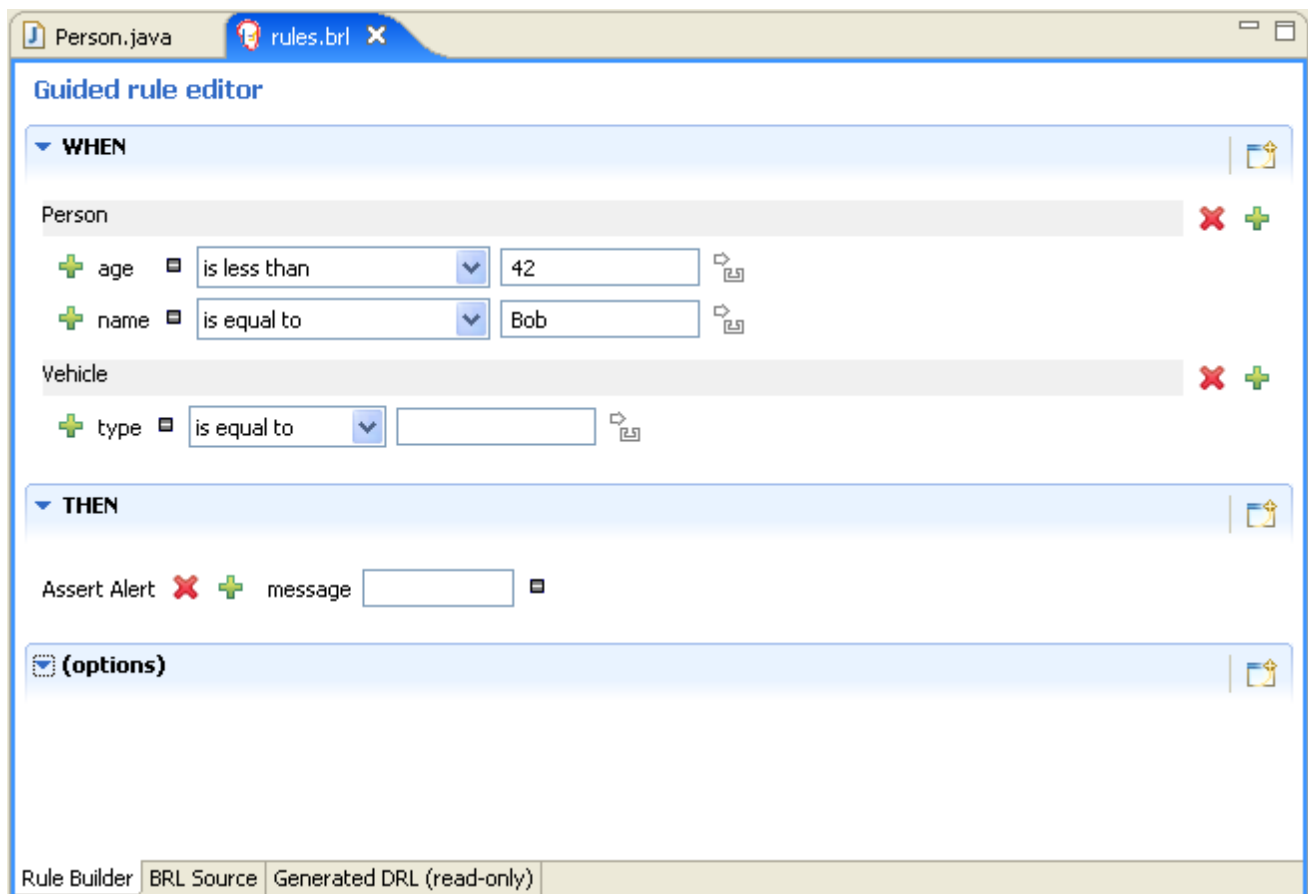


図7.7 ガイド付きエディター

このツールを使用してルールを作成するには、次の手順に従います。

1. **Wizard** メニューをクリックします。

2. **.brl** ファイルを作成し、**Guided Editor** で開きます。



#### 注記

**Editor** は、**.brl** ファイルと同じディレクトリにある **.package** を使用して動作します。このファイルには、通常の **.drl** ファイルの上部にあるようなパッケージ名とインポートステートメントが含まれます。

3. パッケージファイルに必要な **fact** クラスを追加します。
4. この情報を追加したら、**Guided Editor** によって示されたプロンプトに従います (ファクトと関連するフィールドが表示されます)。

**model** または **fact** クラスが提供されたら、**Guided Editor** はルールの視覚的な表現をレンダリングすることができます。また、これを使用し、ビジネスルール言語を直接使用してルールを構築することもできます。これを実現する方法の 1 つが **drools-ant** モジュールを使用する方法です。このモジュールは、ルールアセットをディレクトリのルールパッケージとして作成するため、バイナリファイルとしてデプロイすることが可能になります。また、以下のコードスニペットを使用して、**BRL** ファイルを **.drl** ルールに変換する方法もあります。

#### 例7.1 変換コード

```
BRXMLPersistence read = BRXMLPersistence.getInstance();
BRDRLPersistence write = BRDRLPersistence.getInstance();
String brl = ... // read from the .brl file as needed...
String outputDRL = write.marshall(read.unmarshal(brl));
// Pass the outputDRL to the PackageBuilder, as usual
```

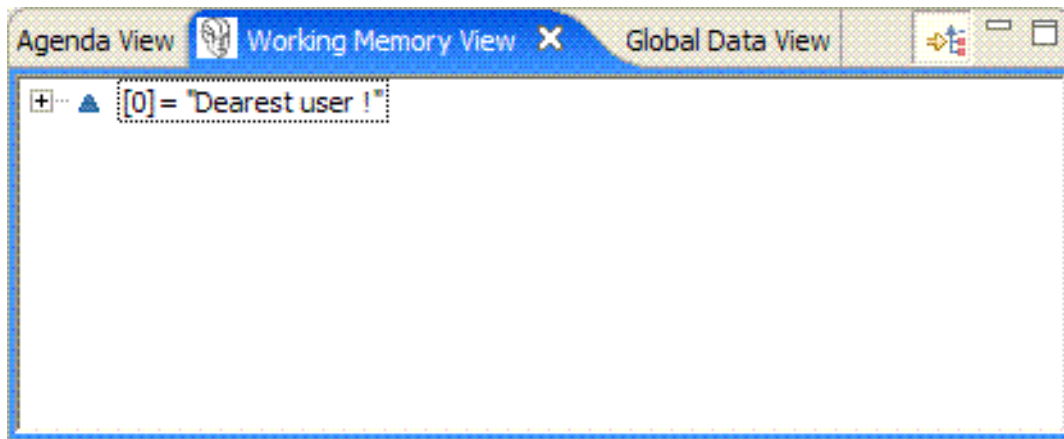
## 7.7. JBOSS RULES ビュー

ビューを使用して、アプリケーションのデバッグ時に **JBoss Rules** エンジンの状態をチェックします。**Working Memory View**、**Agenda View** および **Global Data View** の 3 つのビューが提供されます (**Audit View** もあります)。これらのビューを使用するには、**working memory** を起動するコードでブレークポイントを作成します (**workingMemory.fireAllRules()** を呼び出す行が適しています)。デバッガーが **joinpoint** で停止した場合、**Debugging Variables** ビューで **working memory** 変数を選択します。次に、次の機能を使用して選択された **working memory** の詳細を表示します。

- **Working Memory View** には **JBoss Rules** の **working memory** にある要素がすべて表示されます。
- 名前の通り、**Agenda View** には **agenda** の要素がすべて表示されます (各ルールの名前とバインドされた変数が表示されます)。
- **Global Data View** には、現在 **JBoss Rules** の **working memory** に定義されているグローバルデータがすべて表示されます。
- **Audit View** には、**rules engine** が実行された時に生成された監査ログがツリー形式で表示されます。

### 7.7.1. Working Memory View



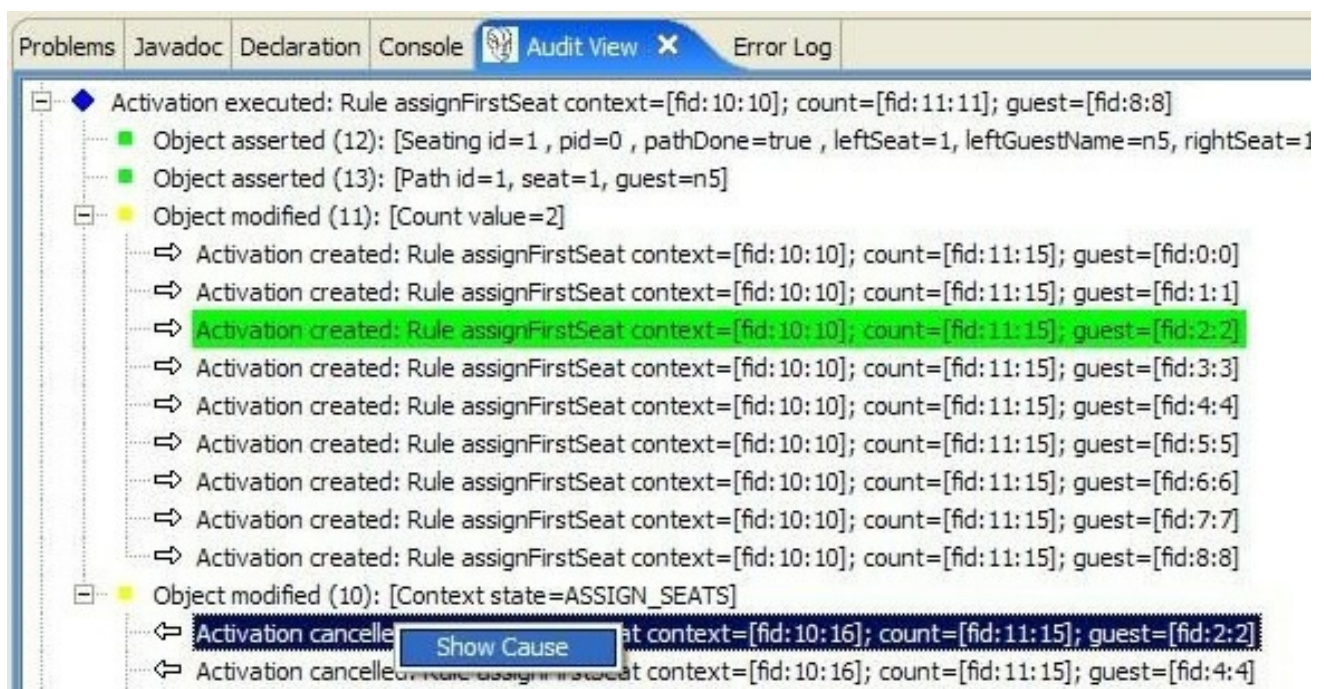


Working Memory View には、Drools エンジンのワーキングメモリーの要素がすべて表示されます。

表示内容をカスタマイズするため、ビューの右側にアクションが追加されます。

**Show Logical Structure** アイコンをクリックして、**working memory**にある各要素の **論理構造**の表示と、要素の詳細表示を切り替えます。論理構造では要素のセットを簡単に視覚化できます。**AgendaItems** の論理構造は、ルールとそのルールによって使用される全パラメーターの値を表示します。

## 7.7.2. Audit View



次のコードを使用して監査ログを作成します。

### 例7.2 監査ログの設定

```
WorkingMemory workingMemory = ruleBase.newWorkingMemory();
// Create a new Working Memory Logger, that logs to file.
WorkingMemoryFileLogger logger = new
WorkingMemoryFileLogger(workingMemory);
// An event.log file is created in the subdirectory log (which must
exist)
// of the working directory.
```

```

logger.setFileName( "log/event" );

workingMemory.assertObject(...);
workingMemory.fireAllRules();

// stop logging
logger.writeToDisk();

```

Audit View の最初のアイコンである Open Log アクションをクリックしてログを開きます。すると、Audit View にルール実行中に記録された全イベントが表示されます。イベントのタイプによってアイコンが異なります。

1. 挿入されたオブジェクト (緑色の正方形)
2. 更新されたオブジェクト (黄色の正方形)
3. 削除されたオブジェクト (赤色の正方形)
4. 作成されたアクティベーション (右矢印)
5. キャンセルされたアクティベーション (左矢印)
6. 実行されたアクティベーション (青色の菱形)
7. 開始または終了したルールフロー (「プロセス」アイコン)
8. アクティベートまたはアクティベート解除されたルールフローグループ (「アクティビティ」アイコン)
9. 追加または削除されたルールパッケージ (「JBoss Rules」アイコン)
10. 追加または削除されたルール (「JBoss Rules」アイコン)

すべてのイベント記録は、イベント発生時の追加情報を提供します。ワーキングメモリのイベントでは (挿入、変更、取り消しなど)、詳細にオブジェクトの **id** と **toString** 表現が含まれます。アクティベーションイベントでは (作成済み、キャンセル済み、実行済みなど)、詳細にルール名とアクティベーションにバインドされたすべての変数が含まれます。



### 注記

アクティベーションの実行中にイベントが発生した場合、その実行の子として表示されます。

イベントの原因を調査するにはイベントを選択します。原因が判明している場合、緑色で表示されます。または、アクションを右クリックして **Show Cause** メニューエントリを選択します。これにより、ログで原因が記録された場所にカーソルが移動します。





## 注記

オブジェクト「modification」または「retraction」の原因は、そのオブジェクトの最後のイベントとして記録されます。これは、同じオブジェクトに対する「object asserted」または最後の「object modified」イベントのいずれかです。

「activation canceled」または「executed」イベントの原因は、対応する「activation created」イベントです。

## 7.8. ドメイン固有言語

ドメイン固有言語では機能的に英語のルールを記述できるカスタム言語を作成できます。つまり、ドメイン固有言語は自然言語のように読み取ります。使用するには、次の手順に従います。

1. ビジネスアナリストが独自の言葉でどのように記述するか注意してください。
2. ルールコンストラクトよりこれをオブジェクトモデルへマッピングします (ドメインオブジェクトとルール自体の間で隔離レイヤーを提供できることも利点となります)。



## 注記

ルールが増えるとドメイン固有言語も大きくなります。一般的な用語を異なるパラメーターで繰り返し使用すると最も効率的です。



## 注記

**Rule Workbench** はドメイン固有言語のエディターを提供します (言語はプレーンテキスト形式で格納されるため、好きなエディターを使用することができますが、**Rule Workbench** ツールは **Properties** ファイル形式の若干改良されたバージョンを提供する利点があります)。

**Editor** は **.dsl** 拡張子を持つ任意のファイル上で起動されます (**.dsl** ファイルのサンプルを作成する **wizard** もあります)。

### 7.8.1. 言語の編集

Editing Domain specific language: [/FinancialRules/hr-lang.dsl]

Description:

Language Expression	Rule Language Mapping	Object	Scope
There is an Person with name of {name}	Person(name == "{name}")		[condition]
Person is at least {age} and lives in {location}	Person(age > {age},location == "{loc		[condition]
Log : "{message}"	System.out.println("{message}");		[consequence]
Send a message to {Person} with message {Me EmailUtil.sendEmail("{Person			[consequence]

Expression:

Mapping:

Object:

Sort by:

図7.8 ドメイン固有言語エディター

**Domain-Specific Language Editor** は言語のルール形式へのマッピングのタブ形式ビューを提供します (「Language Expressions」はルールで使用されるものです)。**Domain-Specific Language Editor** は **Rule Editor** の コンテンツアシスタント もフィードします。そのため、言語式をドメイン固有言語設定へ提示できます (**rule resource** ファイルが開かれた時に **Rule Editor** はこの設定をロードします)。ルールの言語マッピングは、**rule engine** によって言語式がコンパイルされる「コード」を定義します。

ルール言語式で使用される形式は、ルールの「条件」または「アクション」部分を対象にするかどうかによって異なります (右側の場合、Java の断片になることがあります)。**scope** 項目は式がどこに属するかを示します。**when** は左側を示し、**then** は右側を示します。**\*** は「どこでも」という意味です。



#### 注記

キーワードのエイリアスを作成することも可能です。

マッピングアイテム (テーブルの行) を選択し、テーブルの下にあるテキストフィールドの式とマッピングを確認します。これをダブルクリックするか、**edit** ボタンを押すと、**Edit** ダイアログボックスが表示されます。ここで項目を削除したり追加したりすることが可能です。



#### 警告

必ず、式が使用されていないことを確認してから項目を削除してください。

**Edit an existing language mapping item.**

Language expression:

Rule mapping:

Object:

Scope:

図7.9 言語マッピングエディターのダイアログ

変換処理は次のように行われます。

1. パーサーが **DSL** ファイルのルールテキストを読み取り、スコープに応じて言語式に対して行ごとに一致しようとします。
2. 一致した場合、括弧内のプレースホルダー (**{age}** など) に対応する値がルールソースより抽出されます。
3. 「Rule Expression」のプレースホルダーは、対応する値に置き換えられます (上記の例では、自然言語式は「age」および「location」フィールドと、元のルールテキストより抽出された **{age}** および **{location}** 値に基づいて、「Person」型のファクトの2つの制約へマッピングします)。



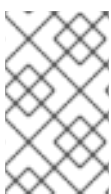
### 注記

**.dr1** ファイルの特定のルールに対して言語マッピングを使用したくない場合は、式の先頭に **>** を付けるとコンパイラによって無視されます。また、ドメイン固有言語は任意であることにも注意してください。

ルールがコンパイルされると、**.dsl** ファイルも使用可能である必要があります。

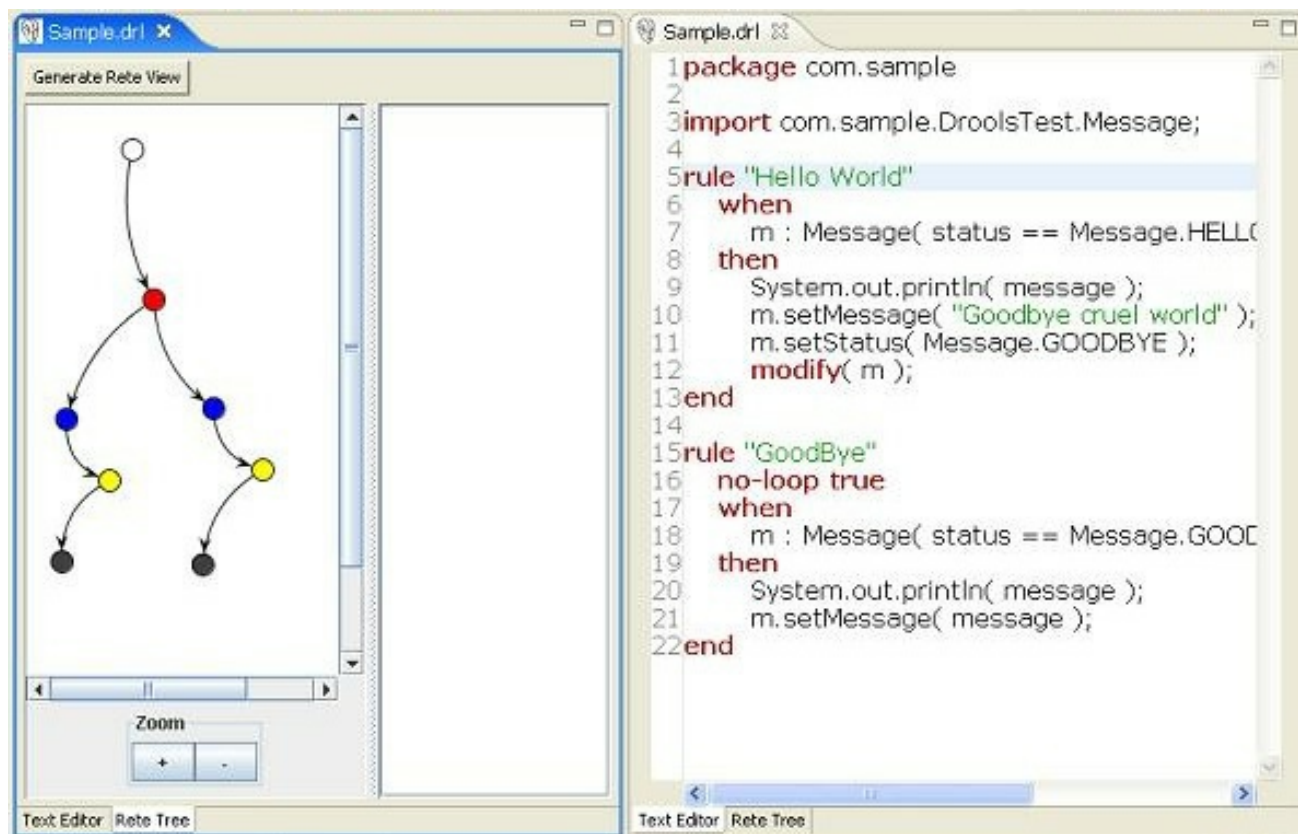
## 7.9. RETE VIEW

**Rete Tree View** は現在の **.dr1** ファイルに対する **Rete Network** を表示します。表示するには、**DRL Editor** ウィンドウの下部にある **Rete Tree** というタブをクリックします。表示されると、個別のノードを「ドラッグアンドドロップ」し、最適な概観を調整します (これらのノード上で長方形をドラッグし、複数のノードを選択することもできます。こうすることで、グループ全体を移動することができます)。JBoss Rules IDE ツールバーの拡大アイコンは、慣習的なやり方で使用できます。



### 注記

今後のバージョンでは、**Rete Tree** をイメージとしてエクスポートできるようになる予定です。この機能が導入されるまで、スクリーンショットを回避策として使用してください。



**Rete View** は、JBoss Developer Studio の グラフィカル編集フレームワークを完全活用する高度な機能です。



#### 重要

この機能は **JBoss Rules** プロジェクトでのみ使用可能です。**JBoss Rules Builder** はプロジェクトのプロパティーで設定されます。

## 7.10. 大型の .DRL ファイル

使用される **Java Development Kit** によっては、permanent generation 設定の最大サイズを増やす必要がある場合があります。SUN および IBM の JDK には permanent generation 設定がありますが、BEA の JRockit にはありません。

permanent generation のサイズを増やすには、**-XX:MaxPermSize=###m** を用いて **JBoss Rules IDE** を起動します。例は次の通りです。

例: **c:\Eclipse\Eclipse.exe -XX:MaxPermSize=128m**

4000 以上のルールがある場合に備えて、permanent generation を最低でも **128 Mb** に設定してください。



#### 注記

これは、一般的に大量のルールをコンパイルする場合でも同様です。通常、1 つのルールに 1 つ以上のクラスがあるためです。

この代わりに、**.rule** 拡張子を持つファイルにルールを格納することもできます。これを行うと、**background builder** は変更のたびにルールをコンパイルしないようになります。これにより、パフォーマンスが向上されます (特に、非常に大量のルールを処理する時に **IDE** の動作が遅い場合)。

## 7.11. ルールのデバッグ

**JBoss Rules** の実行中にルールをデバッグすることが可能です。ルールの **consequences** にブレークポイントを追加できます。ルール実行中にこのようなブレークポイントが検出されると処理が停止されるため、そのポイントで既知の変数をチェックし、デフォルトのデバッグアクションを使用して次に行う処理を決定することができます。また、**Debugging View** を使用して **working memory** と **agenda** の内容を調査することも可能です。

### 7.11.1. ブレークポイントの作成

次の 2 つの方法のいずれかを用いて、ルールのブレークポイントを追加または削除します。

1. ブレークポイントを追加する行で **DRL Editor** の **Ruler** をダブルクリックします。



#### 重要

このようなブレークポイントは、ルールの **consequence** のみで作成できます。ブレークポイントが許可されない行をダブルクリックしても何も行われません。

ブレークポイントを削除するには、再度 **Ruler** をダブルクリックします。

2. ルーラーを右クリックすると、「Toggle breakpoint」アクションが含まれるポップアップメニューが表示されます。ルールブレークポイントはルールの結果でのみ作成できることに注意してください。ルールブレークポイントがその行で許可されない場合、アクションは自動的に無効になります。アクションをクリックすると、選択された行にブレークポイントが追加されますが、ブレークポイントがすでに存在する場合はブレークポイントが削除されます。

**Debug Perspective** には **Breakpoint View** が含まれます。これを使用して、定義されたブレークポイントをすべて確認し、プロパティを取得して有効、無効、または削除します。

### 7.11.2. ルールのデバッグ

ブレークポイントを有効にするには、プログラムを **JBoss Rules Application** としてデバッグする必要があります。

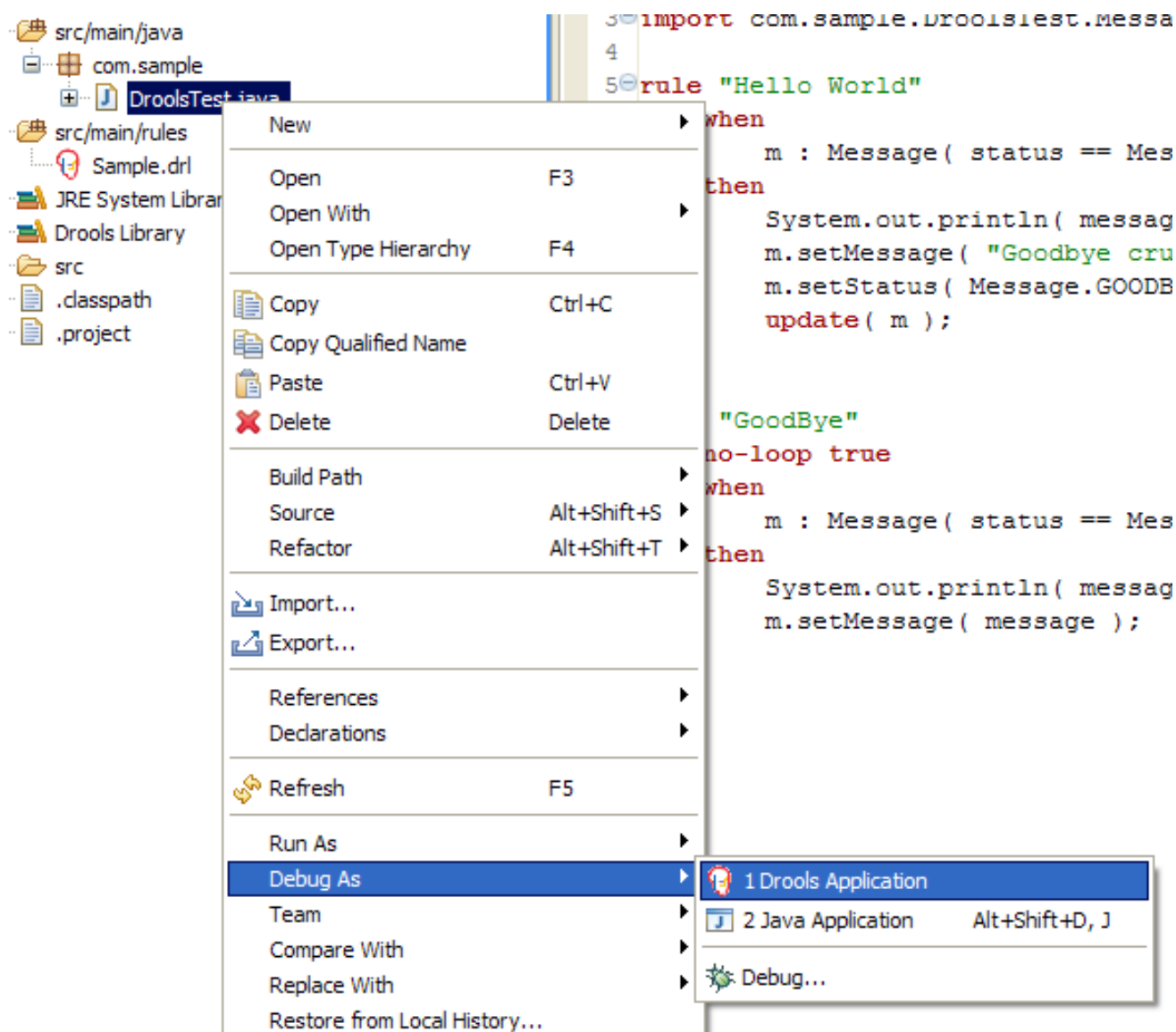


図7.10 JBoss Rules アプリケーションとしてのデバッグ

1. アプリケーションのメインクラスを選択し、右クリックします。**Debug As** サブメニューを選択した後に **JBoss Rules Application** を選択します。

この代わりに **Debug ...** メニューアイテムを選択することもできます。この場合、デバッグ設定を作成、管理、および実行するための新しいダイアログボックスが表示されます (以下のスクリーンショットを参照)。

2. 左側の **tree** で **JBoss Rules Application** アイテムを選択し、**New Launch Configuration** ボタン (**tree** の上にあるツールバーの一番左にあるアイコン) をクリックします。これにより、すでに一部のプロパティが設定された (プロジェクトやメインクラスなど) 新しい設定が作成されます (最初に選択したメインクラスに基づきます)。これらすべてのプロパティは標準的な Java プログラムのプロパティと同じです。
3. デバッグの設定の名前を意味のある名前に変更します。
4. アプリケーションのデバッグは開始するには、ウインドウの下にある **Debug** ボタンをクリックします。

デバッグ設定は一度だけ設定する必要があります。次回デバッグが必要な時は同じ設定を使用します。





## 注記

**JBoss Rules IDE** ツールバーには、以前の設定の 1 つを即座に再実行するショートカットボタンも含まれています (少なくとも Java、Java Debug、または JBoss Rules の 1 つが選択された時)。

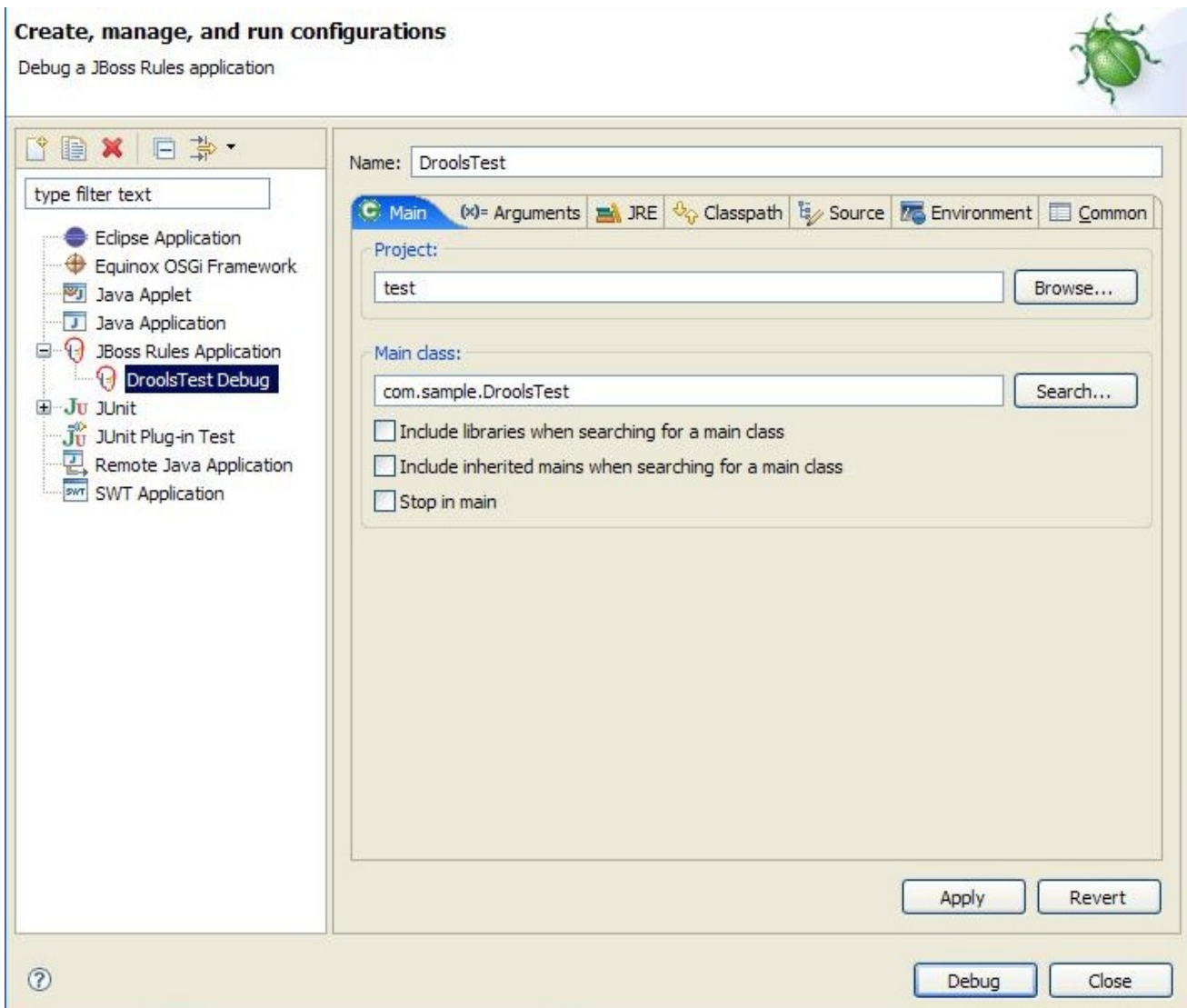


図7.11 「Debug as JBoss Rules Application」の設定

**Debug** ボタンをクリックすると、アプリケーションが実行を開始し、ブレークポイント (**JBoss Rules** のルールブレークポイントまたは他の標準的な Java ブレークポイント) が検出されると停止します。**JBoss Rules** のルールブレークポイントが検出されると、対応する DRL ファイルが開かれ、アクティブな行が強調表示されます。**Variables View** には、ルールパラメーターすべてと、それらのパラメーターに関連する値が含まれています。

デフォルトの Java デバッグアクションを使用して、次に行う処理 (再開、終了、または行のステップオーバー) を決定します。その時点で、**Debug View** を使用して **working memory** と **agenda** の内容を調査することも可能です (現在実行されているものが自動的に表示されるため、この時点で **working memory** を選択する必要はありません)。

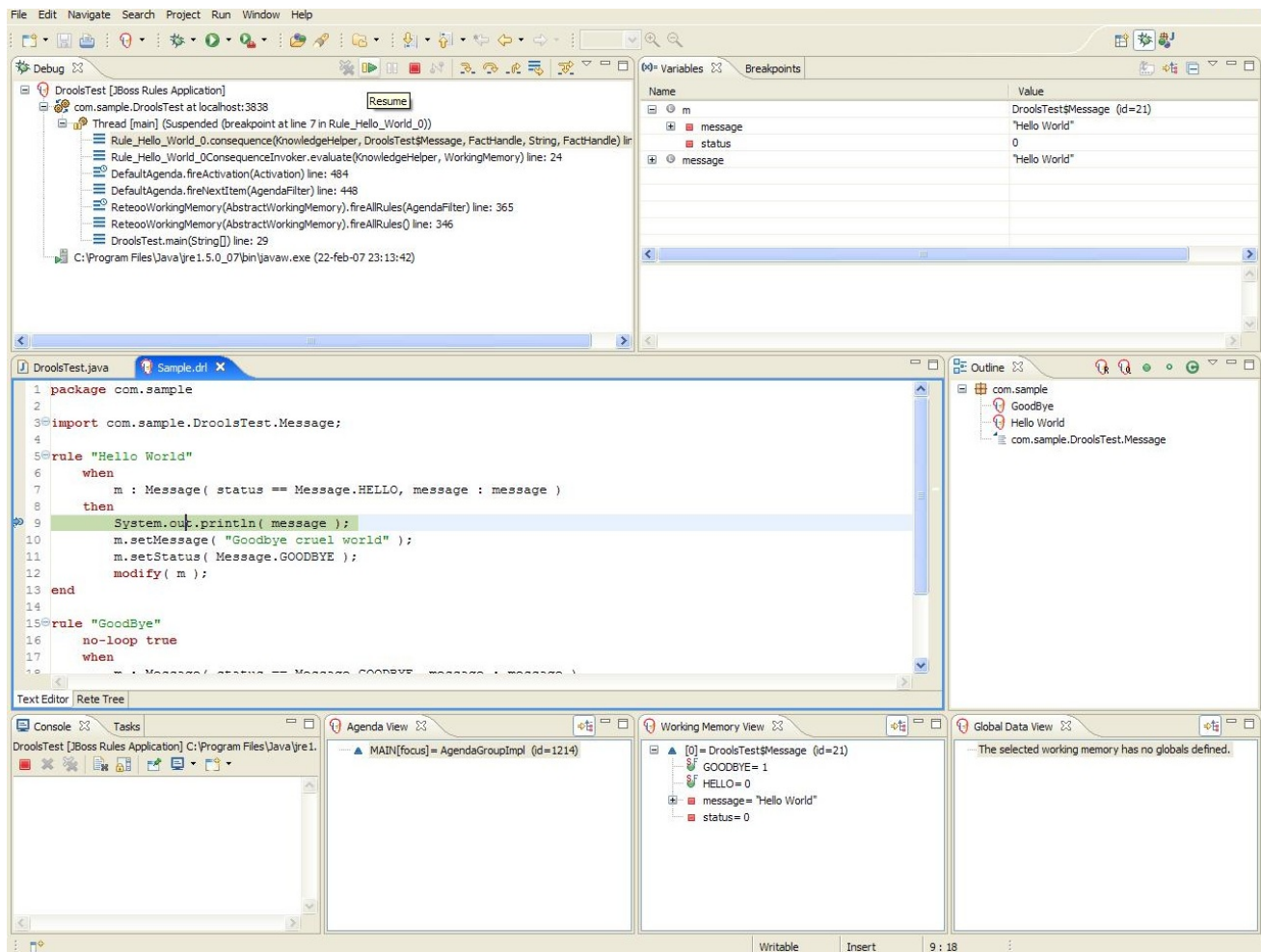


図7.12 デバッグ



# 第8章 例

本書の最後には、本書で説明した一部機能の使用例を提供するチュートリアルがあります。本章の例を確認するには、<http://download.jboss.org/drools/release/5.0.1.26597.FINAL/drools-5.0-examples.zip> より **Examples ZIP** アーカイブファイルをダウンロードしてください。

## 8.1. HELLOWORLD の例

名前:	HelloWorld の例
メインクラス:	<b>org.drools.examples.helloworld.HelloWorldExample</b>
タイプ:	Java アプリケーション
目的:	簡単な Rules の使用方法を示すチュートリアル

このチュートリアルは、ルール使用の例を提供し、MVFLEX 式言語と Java ダイアレクトを実証します。また、**knowledge bases** および **sessions** をビルドする方法を説明し、監査のロギングとデバッグの出力を実証します (両方とも他の例では省略されています)。

**Knowledge Builder** を使用して、*Drools* ルール言語 (DRL) ソースファイルを **knowledge base** が消費できる複数の **Package** オブジェクトへ変換します。

**add** メソッドは、**Resource** インターフェイスと **Resource Type** の両方をパラメーターとして取ります。**Resource** インターフェイスを使用して、**ResourceFactory** を介して **DRL** ソースファイルをクラスパスより読み出します。



### 注記

ここでは実証されていませんが、**Resource** を使用して **DRL** ファイルを URL アドレスなどの他の場所から読み出すことも可能です。必要に応じて複数のファイルを追加できます。

また、異なる名前空間を持つ **DRL** ファイルを追加することも可能です (この場合、**Knowledge Builder** は各名前空間に対して 1 つのパッケージを作成します)。異なる名前空間を持つ複数のナレッジパッケージを同じ **knowledge base** に追加することができます。

**DRL** をすべて追加したら、**Knowledge Builder** のエラーをチェックします (**knowledge base** がパッケージを検証しますが、文字列形式のエラー情報にしかアクセスできません。そのため、**Knowledge Builder** インスタンスよりデバッグを行う必要があります)。

エラーが修正されたら、**Knowledge Builder** コレクションを取得し、**KnowledgeBuilderFactory** より **Knowledge Builder** をインスタンス化してナレッジパッケージのコレクションを追加します。

### 例8.1 HelloWorld の例: Knowledge Base および Session の作成

```
final KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
```

```
// this will parse and compile in one step
kbuilder.add(ResourceFactory.newClassPathResource
    ("HelloWorld.drl", HelloWorldExample.class), ResourceType.DRL);

// Check the builder for errors
if (kbuilder.hasErrors())
{
    System.out.println(kbuilder.getErrors().toString());
    throw new RuntimeException("Unable to compile \"HelloWorld.drl\".");
}

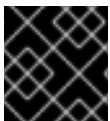
// get the compiled packages (which are serializable)
final Collection<KnowledgePackage> pkgs =
kbuilder.getKnowledgePackages();

// add the packages to a knowledgebase (deploy the knowledge packages).
final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(pkgs);

final StatefulKnowledgeSession ksession =
    kbase.newStatefulKnowledgeSession();
```

**JBoss Rules** の イベントモデル は、独自の内部プロセスのほとんどを公開します。 **DebugAgendaEventListener** と **DebugWorkingMemoryEventListener** の 2 つのデフォルトデバッグ **listeners** が提供されます。これらのリスナーは、デバッグ情報を **Error Console** へ出力します (**listeners** をセッションに追加するのは簡単です。この処理については、後で説明します)。

**KnowledgeRuntimeLogger** は、**Agenda** と **Working Memory listeners** から特別に派生されたものです。 **実行監査** を提供し、この出力はグラフィック表示されます。



### 重要

エンジンが実行を終了した時に **logger.close()** を呼び出す必要があります。



### 注記

本書の例では主に **JBoss Rules** の監査ロギング機能を使用し、今後の検査のために実行フローを記録します。

## 例8.2 HelloWorld の例: イベントのロギングと監査

```
// setup the debug listeners
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugWorkingMemoryEventListener() );

// setup the audit logging
KnowledgeRuntimeLogger logger =
    KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/helloworld");
```

これは、2つのフィールドのみを持つ (文字列の `message` と、整数である **HELLO** または **GOODBYE** のどちらかになる `status`) 単一クラスの単純な例です。

### 例8.3 HelloWorld の例: メッセージクラス

```
public static class Message
{
    public static final int HELLO    = 0;
    public static final int GOODBYE = 1;

    private String      message;
    private int         status;
    ...
}
```

この例は、**Hello World** という言葉が含まれ、**HELLO** の状態を持つ単一の **Message** オブジェクトを作成します。作成後、このオブジェクトはエンジンへ挿入され、その時点で **fireAllRules()** が実行されます。



#### 注記

ネットワーク評価はすべて挿入中に実行されます。そのため、実行しているプログラムが **fireAllRules()** メソッド呼び出しに到達するまでに、完全一致し、適切に実行できるルールをエンジンが認識します。

### 例8.4 実行

```
final Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
ksession.insert(message);

ksession.fireAllRules();

logger.close();

ksession.dispose();
```

この例を Java アプリケーションとして実行するには、次の手順に従います。

1. JBoss Rules IDE で `org.drools.examples.helloworld.HelloWorldExample` クラスを開きます。
2. クラスを右クリックし、`[Run as...]` を選択した後、`[context menu]` から `[Java application]` を選択します。



### 注記

`fireAllRules()` にブレイクポイントを追加し、`ksession` 変数を選択すると、**Hello World** がすでにアクティベートされ、**Agenda** に追加されたことが分かるはずです（これにより、パターン一致の作業がすべて挿入中に実行されたことが確認されます）。

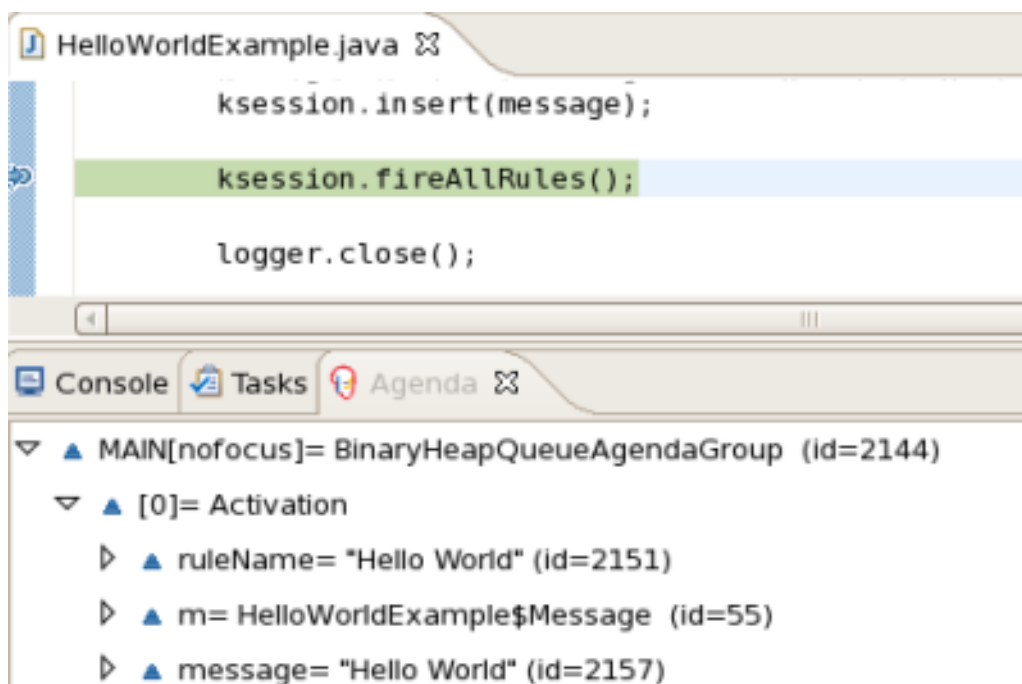


図8.1 fireAllRules Agenda ビュー

アプリケーションの出力は `System.out` へ送られ、`debug listener` の出力は `debug listener` へ送られます。

#### 例8.5 コンソールウィンドウの `System.out`

```
Hello World
Goodbye cruel world
```

#### 例8.6 コンソールウィンドウの `System.err`

```
==>[ActivationCreated(0): rule=Hello World;
tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]
```

```

[ObjectInserted: handle=
[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96];
object=org.drools.examples.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;

tuple=[fid:1:1:org.drools.examples.HelloWorldExample$Message@17cec96]]

==>[ActivationCreated(4): rule=Good Bye;
tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96];
old_object=org.drools.examples.HelloWorldExample$Message@17cec96;

new_object=org.drools.examples.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;

tuple=[fid:1:2:org.drools.examples.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]

```

ルールの「左側」の部分 (**when** 以降) は、**working memory** への挿入時に、**Message.HELLO** 状態の各 **Message** オブジェクトに対してアクティベートされることを示しています。

また、コードの左側の部分は、**message** (**message** 属性へバインドされる) と **m** (一致した **Message** オブジェクト自体にバインドされる) の 2 つの変数バインディングが作成されるよう指示します。

右側 (**then** 以降) はルールの「結果」部分です。ルールの **dialect** 属性で宣言された通り、MVEL で書かれていることを確認してください。ルールのこの部分は、**bound variable message** の内容を **System.out** へ送信します。この後、MVEL の **modify** ステートメントより、**m** にバインドされた **message object** に含まれる **message** および **status** 属性を変更します。このステートメントにより、割り当てのブロックを一度に適用できます (ブロックが完了すると、エンジンは自動的に変更を通知されます)。

### 例8.7 Hello World のルール

```

rule "Hello World"
  dialect "mvel"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify (m) { message="Goodbye cruel world", status=Message.GOODBYE };
  end

```

ルール結果の実行中に再度 **Agenda** ビューを確認するには、次の手順に従います。

1. **DRL** ファイルの **modify** 呼び出し上にブレイクポイントを設定します。
2. **JBoss Rules IDE** の **org.drools.examples.HelloWorld** クラスを開きます。
3. コンテキストメニューへ移動し、[ **Debug As...** ] をクリックした後に [ **JBoss Rules Application** ] を選択し、実行を開始します。

**Good Bye** という別のルールは **Java** を使用します。このルールはここでアクティベートされ、アジェンダに置かれます。

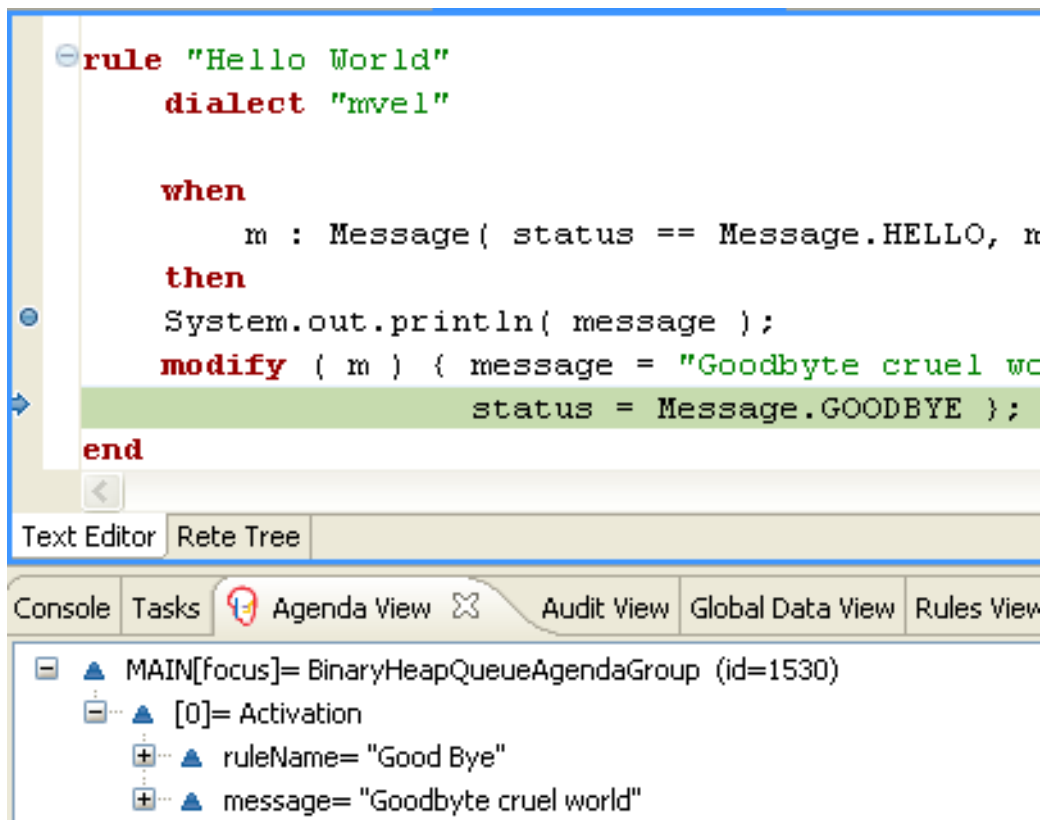


図8.2 Hello World ルールの Agenda ビュー

**Good Bye** ルールは **Hello World** ルールと似ていますが、**Message.GOODBYE** 状態の **Message** オブジェクトと一致します。

#### 例8.8 Good Bye ルール

```
rule "Good Bye"
  dialect "java"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
  end
```

`KnowledgeRuntimeLoggerFactory` メソッドの `newFileLogger` を使用して `KnowledgeRuntimeLogger` を作成した Java コードをもう一度思い出してください。このコードは最後に `logger.close()` を呼び出しました。このようにして、**Audit** に表示される監査ログファイルが作成されました。



注記

**Audit** ビューは実行フローを表示するため多くの例で使用されます。

以下のスクリーンショットを見てください。これは以下を表しています。

- 1. オブジェクトが挿入され、**Hello World** ルールのアクティベーションが作成されます。
- 2. アクティベーションが実行され、**Message** オブジェクトが更新されます。その結果、**Good Bye** ルールがアクティベートされ、実行されます。
- 3. **Audit** ビューでイベントを選択すると、元のイベントが緑色で強調表示されます (この例では、**Activation executed** イベントの元となる **Activation created** イベントが緑色で強調表示されます)。

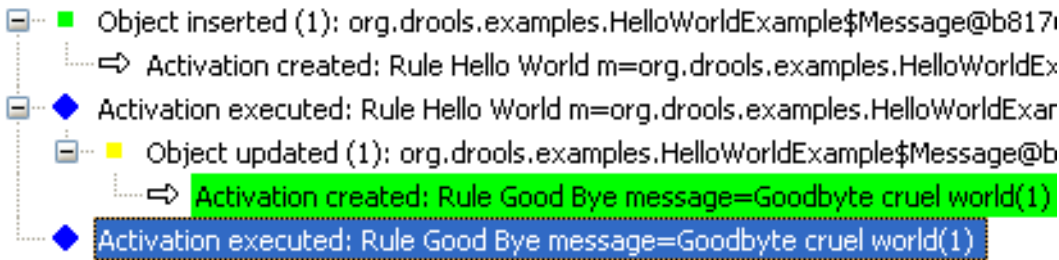


図8.3 Audit ビュー

8.2. 状態の例

名前:	状態の例
状態クラス:	<code>org.drools.examples.state.StateExampleUsingSalience</code>
タイプ:	Java アプリケーション
Rules ファイル:	<code>StateExampleUsingSalience.drl</code>
目的:	基本的なルールの使用とルール実行優先度における競合の解決方法を実証します。

この例には 3 つの異なる実装があり、各実装は *前向き連鎖* と呼ばれる同じ基本的な挙動を実装するための代替方法を実証します。前向き連鎖とは、**working memory** のファクトへの変更に基づいてルールを順に評価、アクティベート、および実行する **engine** の機能のことです。

8.2.1. 状態の例について

`org.drools.examples.state.State` クラスには、名前と現在の状態の 2 つのフィールドがあります。現在の状態は以下のいずれかになります。

- **NOTRUN**
- **FINISHED**

#### 例8.9 状態クラス

```
public class State {
    public static final int NOTRUN    = 0;
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

上記の例を見てください。`org.drools.examples.state.State` を無視すると (理由は後で説明します)、**A**、**B**、**C**、および **D** という名前の 4 つの **State** オブジェクトが作成されたことが分かります。当初これらの状態は、使用されるコンストラクターのデフォルト値である **NOTRUN** に設定されます。各インスタンスが順に **session** にアサートされ、**fireAllRules()** が呼び出されます。

#### 例8.10 Saliency 状態の実行例

```
State a = new State( "A" );
State b = new State( "B" );
State c = new State( "C" );
final State d = new State( "D" );

// By setting dynamic to TRUE, JBoss Rules will use JavaBean
// PropertyChangeListeners so you do not have to call modify or
// update().
boolean dynamic = true;

session.insert( a, dynamic );
session.insert( b, dynamic );
session.insert( c, dynamic );
session.insert( d, dynamic );

session.fireAllRules();
session.dispose(); // Stateful rule session must always be disposed when
finished</programlisting>
```

アプリケーションを実行するには、次の手順に従います。



1. JBoss Rules IDE で `org.drools.examples.state.StateExampleUsingSalience` クラスを開きます。
2. `class` を右クリックし、`[Run as...]` を選択した後、`[Java Application]` を選択します。

JBoss Rules IDE コンソールウィンドウに次の出力が表示されます。

#### 例8.11 Salience 状態のコンソール出力

```
A finished
B finished
C finished
D finished
```

計 4 つのルールがあります。Bootstrap ルールが最初に実行され、A が **FINISHED** 状態に設定されると、B の状態も **FINISHED** になります。(C と D は両方とも B に依存するため、一時的に競合が発生しますが、salience 値によって解決されます)。

次に、この処理が実行された方法を分析します。これには、**監査ロギング**機能を使用します。この機能により、各操作の結果がグラフィック表示されます。次の手順に従って監査ログを取得します。

1. **Audit View** が表示されていない場合、`[Window]` をクリックし、`[Show View]`、`[Other...]`、`[JBoss Rules]`、`[Audit View]` と順番に選択します。
2. **Audit View** で **Open Log** ボタンをクリックし、`drools-examples-drl-dir>/log/state.log` というファイルを選択します。

この時点で、画面の **Audit View** は次のようになるはずです。

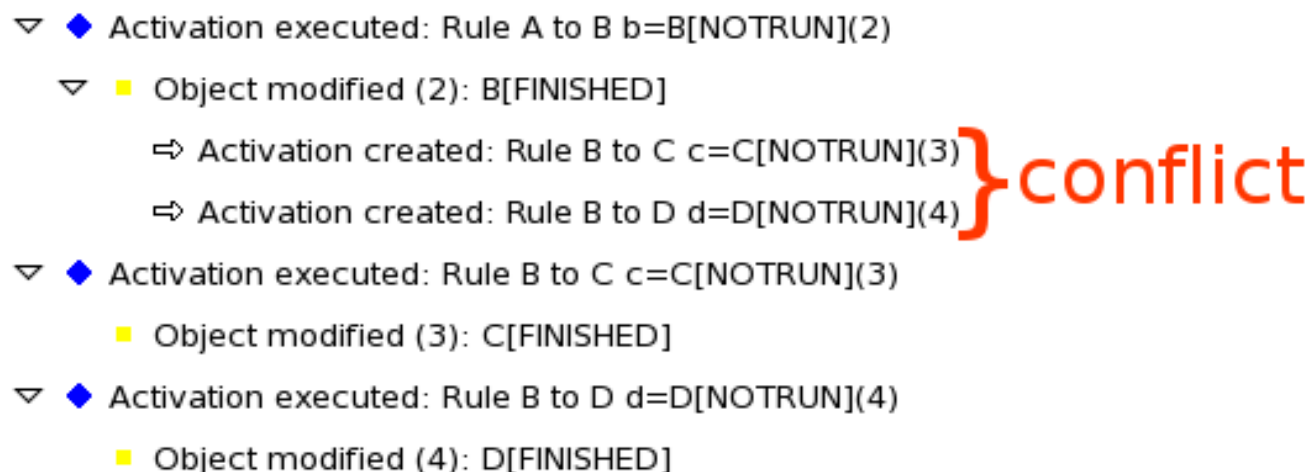


図8.4 Salience 状態の Audit View 例

**Audit View** のログを上から下まで読みます。各アクションと、**working memory** に書き込まれた変更が記録されているのが分かります。そのため、状態が **NOTRUN** の **State** オブジェクト A をアサートすると、**Bootstrap** ルールがアクティベートされますが、他の **State** オブジェクトをアサートしてもすぐに影響はありません。

#### 例8.12 Salience 状態: Bootstrap ルール

```
rule Bootstrap
```

```
when
  a : State(name == "A", state == State.NOTRUN )
then
  System.out.println(a.getName() + " finished" );
  a.setState( State.FINISHED );
end
```

**Bootstrap** ルールを実行すると、**A** の状態が **FINISHED** に変わり、**A to B** ルールがアクティベートされます。

#### 例8.13 A to B ルール

```
rule "A to B"
when
  State(name == "A", state == State.FINISHED )
  b : State(name == "B", state == State.NOTRUN )
then
  System.out.println(b.getName() + " finished" );
  b.setState( State.FINISHED );
end
```

**A to B** ルールを実行すると **B** の状態が **FINISHED** に変わり、**B to C** ルールと **B to D** ルールの両方がアクティベートされ、アクティベーションが **agenda** に置かれます。

これ以降は、両方のルールを実行できるため、「競合」の状態にあると言えます。競合解決戦略により、**engine** の **agenda** はどちらのルールを実行するか決定することができます。**B to C** ルールの **salience** 値の方が大きいため (10、デフォルト値は 0) 最初に実行され、オブジェクト **C** が **FINISHED** 状態に設定されます。

上記の **Audit View** は、結果として 2 つのアクティベーションが競合状態になる **A to B** ルールの **State** オブジェクトへの変更を表しています。**Audit View** が開かれている間にルール内にデバッグポイントを配置することができるため、**Agenda View** を使用してアジェンダの状態を調査することも可能です。以下のスクリーンショットは、**A to B** ルールのブレイクポイントを表しています。また、2 つのルールが競合状態にある時の **agenda** の状態も表しています。

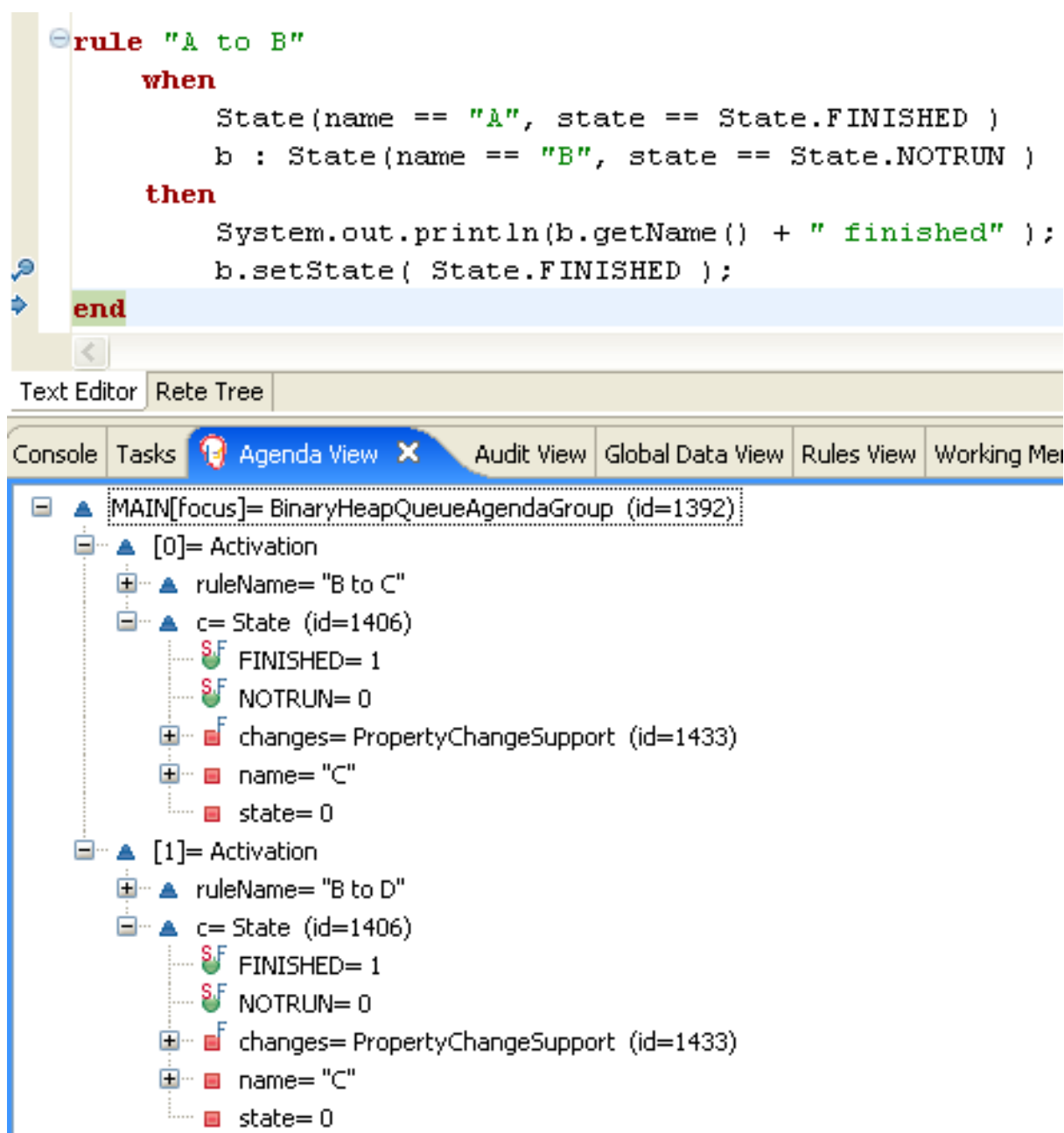


図8.5 状態の例: Agenda View

## 例8.14 B to C ルール

```

rule "B to C"
  salience 10
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
  end

```

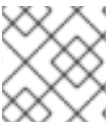
B to D ルールは最後に実行され、オブジェクト D の状態を **FINISHED** に変更します。

## 例8.15 B to D ルール

```
rule "B to D"
when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

この時点で、実行するルールがないため **engine** が停止します。

この例で注意する点は、**PropertyChangeListener** オブジェクトに基づいた *動的* ファクトです。エンジンが fact プロパティの変更を「確認」し、反応するには、アプリケーションがこれらの情報をエンジンに通知する必要があります。これは、ルール (**modify** ステートメント) を介して明示的に行うか、暗黙的 (ファクトが実装された **PropertyChangeSupport** よりエンジンに通知) に行うことが可能です。



### 注記

**PropertyChangeSupport** は **Java Beans Specification** に定義されます。

次の例では、**PropertyChangeSupport** を使用方法について学びましょう (これを使用すると、ルールが **modify** ステートメントだらけにならないようにすることが可能です)。この機能を使用するには、**org.drools.example.State** クラスと同じ方法で、最初にファクトが **PropertyChangeSupport** を実装するようにします。その後、次のコードを使用してファクトを **working memory** に挿入します。

#### 例8.16 動的ファクトの挿入

```
// By setting dynamic to TRUE, JBoss Rules will use JavaBean
// PropertyChangeListeners so that one does not have to call modify or
// update().
final boolean dynamic = true;

session.insert( fact, dynamic );
```

**PropertyChangeListener** オブジェクトが使用される場合、各 セッター は追加のコード (通知のため) を実装する必要があります。**org.drools.examples** クラスの **State** のセッターは次の通りです。

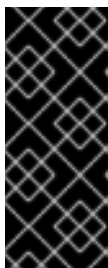
#### 例8.17 「セッター」の **PropertyChangeListener** サポート

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",oldState,newState );
}
```

この例には、他のクラスが2つ存在します。これらのクラスは **StateExampleUsingAgendGroup** と **StateExampleWithDynamicRules** と呼ばれます。上記の通り、これらのクラスは **A to B** から **C to D** へ実行されます。

- **StateExampleUsingAgendGroup** クラスは **agenda groups** クラスを使用してルールの競合を制御し、最初に実行するルールを決定します。
- **StateExampleWithDynamicRules** クラスは、実行している **working memory** セッションへルールを追加する方法を表します。

アジェンダグループを使用してアジェンダをグループに分割し、実行パーミッションを持つグループを決定します。デフォルトでは、すべてのグループが **MAIN** というアジェンダグループに含まれます。agenda-group 属性により、ルールの異なる **agenda group** を指定することができます。最初に、**MAIN agenda group** は **working memory** によって使用されます。



### 重要

グループのルールは、グループがフォーカスを受け取る場合のみ実行されます。これには、**setFocus()** メソッドまたは auto-focus ルール属性を使用します (auto-focus ルール属性を使用する場合、ルールが一致し、アクティベートされると自動的にフォーカスが **agenda group** に設定されるため、*auto-focus* という名前になっています。このメソッドは、**B to D** ルールの前に **B to C** ルールを実行できるようにします)。

#### 例8.18 アジェンダグループの状態の例: B to C ルール

```
rule "B to C"
  agenda-group "B to C"
  auto-focus true
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to
D" ).setFocus();
  end
```

**B to C** は **B to D agenda group** 上で **setFocus()** を呼び出します。これにより、アクティブなルールを実行でき、**B to D** に属するルールをトリガーします。

#### 例8.19 アジェンダグループの状態の例: B to D ルール

```
rule "B to D"
  agenda-group "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
  end
```

`StateExampleWithDynamicRules` の例は、`fireAllRules()` の実行後、他のルールをベースに追加します。この新しいルールは他の 状態遷移 です。

例8.20 動的状態の例: D to E ルール

```
rule "D to E"
when
    State(name == "D", state == State.FINISHED )
    e : State(name == "E", state == State.NOTRUN )
then
    System.out.println(e.getName() + " finished" );
    e.setState( State.FINISHED );
end
```

これは、出力の最後の部分を生成します。

例8.21 動的状態の出力例

```
A finished
B finished
C finished
D finished
E finished
```

8.3. フィボナッチの例

名前:	フィボナッチ
メインクラス:	<code>org.drools.examples.fibonacci.FibonacciExample</code>
タイプ:	Java アプリケーション
Rules ファイル:	<code>Fibonacci.drl</code>
目的:	再帰およびクロス積の一致を実証します。

フィボナッチ数 ([http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)) は、レオナルド・ダ・ピサが発見したゼロと1で始まる数列です。その後のフィボナッチ数は、2つの先行する数を足して取得します。よって、フィボナッチ数列は 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584 で始まり、無限に続きます。この例では、数列を使用して 再帰 を実証し、*salience* 値 使用して競合を解決する方法も実証します。

この例では `Fibonacci` 単一ファクトクラスを使用します。このクラスには `sequence` と `value` の2つのフィールドがあります。`sequence` フィールドは、フィボナッチ数列のオブジェクトの位置を示すために使用されます。`value` フィールドは、特定の数列位置に対する `Fibonacci` オブジェクトの値を示し

ます (算出する必要がある値を示すために **-1** を使用)。

### 例8.22 フィボナッチクラス

```
public static class Fibonacci {
    private int sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

次の手順に従って、例を実行します。

1. **JBoss Rules** の統合開発環境で **org.drools.examples.FibonacciExample** クラスを開きます。
2. このクラスを右クリックし、**[Run as...]** を選択した後に **[Java application]** を選択します。

次の出力が **JBoss Rules IDE Console** ウィンドウに表示されます ( **...snip...** は行が省略されていることを示しています)。

### 例8.23 フィボナッチの例: コンソールの出力

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...snip...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...snip...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

Java から実行するには、数列フィールド **fifty** を持つ単一の **Fibonacci** オブジェクトを挿入します。その後、再帰ルールが実行され、他の 49 個のオブジェクトが自動的に挿入されます。



## 注記

この例では、**PropertyChangeSupport** を使用しません。この代わりに、MVFLEX 式言語を使用するため、**modify** キーワードを使用できます。このキーワードにより、ブロックセッターアクションを使用できます (エンジンに変更も通知します)。

### 例8.24 フィボナッチの例: 実行

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

**Recurse** ルールは非常に単純です。このルールは、**-1** を値として持つアサートされた Fibonacci オブジェクトを照合するため、現在よりも 1 つ前の値を持つ新しい Fibonacci オブジェクトが作成され、アサートされます。数列フィールドが **1** である Fibonacci オブジェクトが存在しない限り、Fibonacci オブジェクトが追加されるたびにルールが実行されます。

50 個のフィボナッチオブジェクトがすべてメモリーに格納されると、ルールの照合を停止するために **not** 条件要素が使用されます。**Bootstrap** ルールを実行する前に 50 個の **Fibonacci** オブジェクトをすべてアサートする必要があるため、ルールは **salience** 値も持っています。

### 例8.25 フィボナッチの例: 「再帰」ルール

```
rule Recurse
  salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
  end
```

**Audit** ビューは元のアサート (数列フィールドは **50**) を表示します。その後、アサートされた各 Fibonacci オブジェクトによって **Recurse** ルールが何度も実行される、ルールの連続的な再帰を表します。



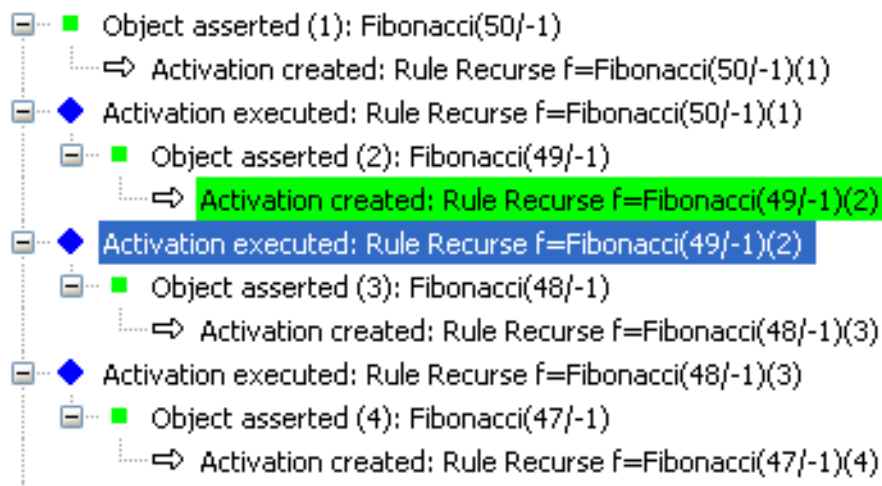


図8.6 フィボナッチの例: Recurse Audit ビュー 1

sequence フィールドの値が 2 である Fibonacci オブジェクトがアサートされると、**Bootstrap** ルールと一致し、**Bootstrap** ルールと共にアクティベートされます。



### 注記

**Bootstrap** フィールドには複数の制限があります。これらの制限は、フィールドの値が 1 または 2 と等しいか確認します。

### 例8.26 フィボナッチの例: Bootstrap ルール

```
rule Bootstrap
when
  f : Fibonacci( sequence == 1 || == 2, value == -1 )
  // this is a multi-restriction || on a single field
then
  modify ( f ){ value = 1 };
  System.out.println( f.sequence + " == " + f.value );
end
```

この時点で、アジェンダは下図のようになります。しかし、**Recurse** ルールの salience 値の方が大きいため、**Bootstrap** ルールは実行されません。

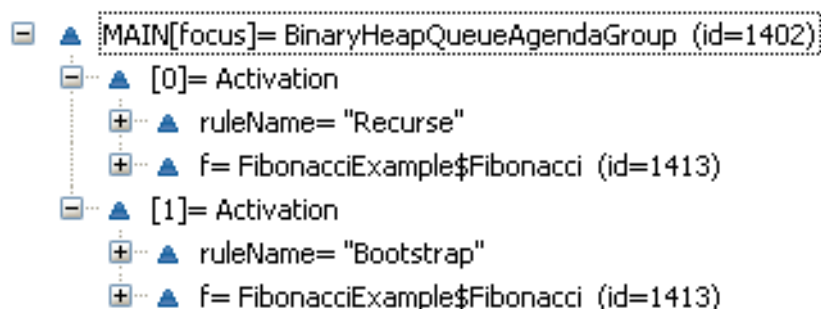


図8.7 フィボナッチの例: Recurse Agenda ビュー 1

sequence 値が 1 の Fibonacci オブジェクトがアサートされると、再度 **Recurse** ルールに対して一致し、2 度アクティベートされます。



## 注記

not 条件要素により、sequence 値が **1** である Fibonacci オブジェクトが存在すると即座にルール的一致が行われなくなるため、**Recurse** は一致せず、アクティベートされません。

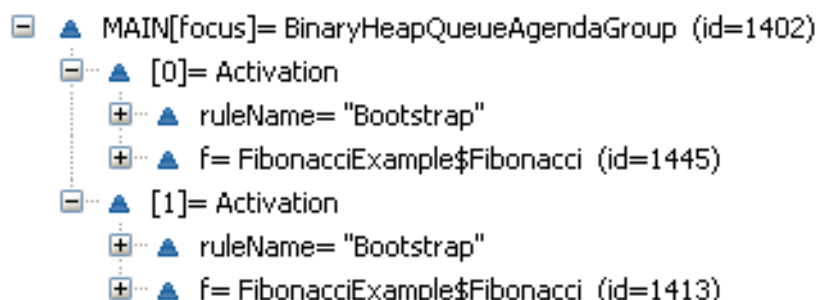


図8.8 フィボナッチの例: Recurse Agenda ビュー 2

-1 と等しくない値を持つ 2 つのオブジェクトが存在すると、**Calculate** ルールが実行されます (**Bootstrap** ルールがオブジェクトに **1** と **2** から **1** の数列値を設定したことに注意してください)。

この時点で、**working memory** に 50 個の Fibonacci オブジェクトが存在します。適切な「3 つ組」を選択して順に値を計算します。

可能なクロス積を制限するフィールド制約を使用しないルールに 3 つのフィボナッチパターンが存在する場合、50x49x48 の可能な組み合わせが存在し、125,000 のルール実行が発生する可能性があります。これらのほとんどは正しくありません。この問題が発生しないようにするため、**Calculate** ルールはフィールド制約を使用して、3 つのフィボナッチパターンを正しい順番に制限します。この方法はクロス積の一致と呼ばれます。この仕組みは次の通りです。

1. 最初のパターンは、値が **!= -1** の Fibonacci オブジェクトを見つけ、パターンとフィールドの両方をバインドします。
2. 2 つ目の Fibonacci オブジェクトも同じ事を行いますが、別のフィールド制約を追加します。これは、f1 にバインドされた Fibonacci オブジェクトよりも数列が大きくなるようにするためです。このルールが最初に実行される時、最初と 2 番目の数列のみ値が **1** になります。2 つの制約は、f1 が最初の数列を参照し、f2 が 2 番目の数列を参照するようにします。
3. 最後のパターンは、-1 と等しい値を持ち、f2 に含まれる値よりも 1 大きい数列値を持つ Fibonacci オブジェクトを見つけます。

この時点で、使用可能なクロス積より 3 つの Fibonacci オブジェクトが適切に選択されます。そのため、3 つ目のオブジェクトの値 (f3 へバインドされる) を計算できます。

### 例8.27 フィボナッチの例: Calculate ルール

```
rule Calculate
  when
    // Bind f1 and s1
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2; refer to bound variable s1
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3; alternative reference of f2.sequence
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1), value ==
-1 )
```

```

    then
        // Note the various referencing techniques.
        modify ( f3 ) { value = f1.value + v2 };
        System.out.println( s3 + " == " + f3.value );
    end

```

**Modify** ステートメントは **f3** にバインドされるオブジェクトの値を更新します。その結果、値が **-1** と等しくない別の新しいオブジェクトが存在することになります。このオブジェクトが作成されると、**Calculate** ルールが再度一致します。その後、次のフィボナッチ数を処理します。次図の **Audit** ビューはこの処理の概要を表しています。これは、最後に実行される **Bootstrap** ルールがどのように Fibonacci オブジェクトを変更し、**Calculate** ルールをトリガーするかを表しています。その後、別の Fibonacci オブジェクトが変更され、**Calculate** ルールが再実行されます。このサイクルは、オブジェクトすべてに値が設定されるまで継続されます。

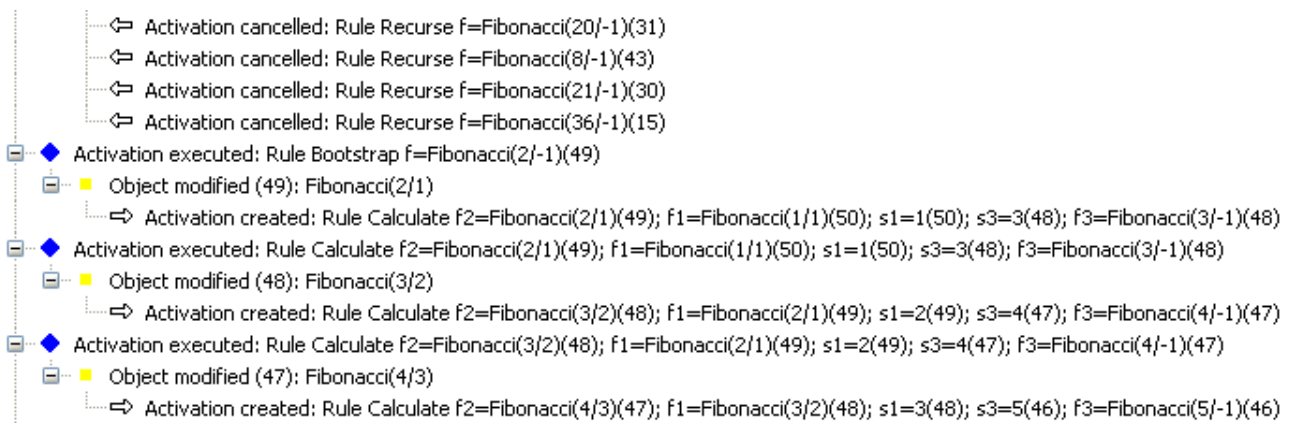


図8.9 フィボナッチの例: Bootstrap Audit ビュー

## 8.4. 銀行取引のチュートリアル

名前:	銀行取引のチュートリアル
メインクラス:	<b>org.drools.tutorials.banking.BankingExamplesApp.java</b>
タイプ:	Java アプリケーション
Rules ファイル:	<b>org.drools.tutorials.banking.*.dr1</b>
目的:	パターンの一致、基本的なソート、および計算ルールを実証します。

本チュートリアルは、複数の口座で入出金を処理する完全な個人銀行取引アプリケーションの開発プロセスを実証します。このプロセスのために作成された複数の設計パターンを使用します。



## 重要

**RuleRunner** クラスは、データのセットに対して 1 つまたは複数の DRL ファイルを実行して、ナレッジパッケージをコンパイルし、各実行に対して **knowledge base** を作成します。これは、テストおよびチュートリアルの目的で使用できますが、**knowledge base** を一度だけ構築してキャッシュする必要がある実稼働システムでは適切でないことに注意してください。

### 例8.28 銀行取引のチュートリアル: RuleRunner

```
public class RuleRunner {

    public RuleRunner() {}

    public void runRules(String[] rules, Object[] facts) throws Exception
    {
        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        KnowledgeBuilder kbuilder =
            KnowledgeBuilderFactory.newKnowledgeBuilder();

        for ( int i = 0; i < rules.length; i++ ) {
            String ruleFile = rules[i];
            System.out.println( "Loading file: " + ruleFile );
            kbuilder.add( ResourceFactory.newClassPathResource
                ( ruleFile, RuleRunner.class ), ResourceType.DRL );
        }

        Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();
        kbase.addKnowledgePackages( pkgs );
        StatefulKnowledgeSession ksession =
            kbase.newStatefulKnowledgeSession();

        for ( int i = 0; i < facts.length; i++ ) {
            Object fact = facts[i];
            System.out.println( "Inserting fact: " + fact );
            ksession.insert( fact );
        }
        ksession.fireAllRules();
    }
}
```

最初の Java クラスは、**Example.dr1** という単一の DRL ファイルをロードし、実行しますが、データは挿入しません。

### 例8.29 銀行取引のチュートリアル: Java の例 1

```
public class Example1
{
    public static void main(String[] args) throws Exception
    {
        new RuleRunner().runRules(new String[]{"Example1.dr1"},
            new Object[0] );
    }
}
```

```
}

```

これは最初に実行する簡単なルールです。これは単一の eval 条件を持ち、常に **true** になります。そのため、常に一致し、起動後に一度「実行」します。

### 例8.30 銀行取引のチュートリアル: Example1.dr1 のルール

```
rule "Rule 01"
when
  eval( 1==1 )
then
  System.out.println( "Rule 01 Works" );
end

```

出力は次の通りです。ルールは単一の **print** ステートメントに一致し、実行することを表しています。

### 例8.31 銀行取引のチュートリアル: Example1.java の出力

```
Loading file: Example1.dr1
Rule 01 Works

```

次に、簡単なファクトをいくつかアサートし、出力します。

### 例8.32 銀行取引のチュートリアル: Java の例 2

```
public class Example2
{
  private static Integer wrap(int i) {return new Integer(i);}

  public static void main(String[] args) throws Exception
  {
    Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
    wrap(1), wrap(5)};
    new RuleRunner().runRules(new String[] { "Example2.dr1" },numbers);
  }
}

```

```
}

```

これは、特定のファクトを使用しませんが、その代わりに `java.lang.Integer` クラスのセットを使用します (コレクション番号が「fact」でも「thing」でもないため、最良の方法ではありません。銀行の口座には残高を表す数字があります。そのため、この場合、口座が「fact」になります。しかし、最初の実証の目的を満たすには整数をアサートすれば十分です)。

次に、これらの数字を出力する簡単なルールを作成します。

### 例8.33 銀行取引のチュートリアル: `Example2.drl` のルール

```
rule "Rule 02"
when
    Number( $intValue : intValue )
then
    System.out.println("Number found with value: " + $intValue);
end

```

繰り返しになりますが、これは大変単純なルールです。数字である「ファクト」を識別し、出力します。ここでは、インターフェースの使用に注意してください。整数は挿入されますが、パターン一致エンジンは、アサートされたオブジェクトのインターフェースとスーパークラスを一致できます。

出力の最初にはロードされた DRL が表示されます。次に挿入されたファクト、その後的一致して実行されたルールが表示されます (挿入された数字は一致および実行されるため、出力されます)。

### 例8.34 銀行取引のチュートリアル: `Example2.drl` の出力

```
Loading file: Example2.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 5
Number found with value: 1
Number found with value: 4
Number found with value: 1
Number found with value: 3

```

数字のソートには、ルールを使用するよりも優れた方法がいくつかありますが、この例の後半で **Cashflow** クラスを日付順に適用するタスクがあるため、ここでプロセスについて説明します。

### 例8.35 銀行取引のチュートリアル: Example3.java

```
public class Example3
{
    private static Integer wrap(int i) {return new Integer(i);}

    public static void main(String[] args) throws Exception
    {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
            wrap(1), wrap(5)};

        new RuleRunner().runRules(new String[]{ "Example3.drl"}, numbers);
    }
}
```

ここでは、ルールが若干異なります。

### 例8.36 銀行取引のチュートリアル: Example3.drl のルール

```
rule "Rule 03"
when
    $number : Number( )
    not Number( intValue < $number.intValue )
then
    System.out.println("Number found with value: "+$number.intValue() );
    retract( $number );
end
```

ルールの最初の行は数字を識別し、値を抽出します。2 番目の行は、最初のパターンで検出された数字よりも小さい数字が存在しないようにします。セットで最も小さな数字のみ一致することを想定するかもしれませんが、出力後に数字が取り消されると、最も小さい数字が削除され、2 番目に小さい数字が公開されます。

生成された出力は次の通りです (数字は数値的にソートされることに注目してください)。

### 例8.37 銀行取引のチュートリアル: Example3.java の出力

```
Loading file: Example3.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
```

```
Inserting fact: 1
Inserting fact: 5
Number found with value: 1
Number found with value: 1
Number found with value: 3
Number found with value: 4
Number found with value: 5
```

次に、個人口座のルールを作成しましょう。最初に **Cashflow** クラスオブジェクトを作成します。

#### 例8.38 銀行取引のチュートリアル: **Cashflow** クラス

```
public class Cashflow
{
    private Date    date;
    private double  amount;

    public Cashflow() {}

    public Cashflow(Date date, double amount)
    {
        this.date = date; this.amount = amount;
    }

    public Date getDate()    { return date; }
    public void setDate(Date date) { this.date = date; }

    public double getAmount()    { return amount; }
    public void setAmount(double amount) { this.amount = amount; }

    public String toString()
    {
        return "Cashflow[date=" + date + ", amount=" + amount + "]";
    }
}
```

**Cashflow** クラスには、日付と金額の2つの簡単な属性があります。**toString** メソッドが追加されているため、出力することができます (浮動小数点形式は多くの数型を正確に表せないため、通貨単位に **double** 型を使用することは通常推奨されません)。

値を設定するために使用できる「オーバーロードされた」コンストラクターもあります。次の例は、それぞれ日付と金額が異なる5つの **Cashflow** オブジェクトを挿入します。



## 例8.39 銀行取引のチュートリアル: Example4.java

```

public class Example4
{
    public static void main(String[] args) throws Exception
    {
        Object[] cashflows = {
            new Cashflow(new SimpleDate("01/01/2007"), 300.00),
            new Cashflow(new SimpleDate("05/01/2007"), 100.00),
            new Cashflow(new SimpleDate("11/01/2007"), 500.00),
            new Cashflow(new SimpleDate("07/01/2007"), 800.00),
            new Cashflow(new SimpleDate("02/01/2007"), 400.00),
        };

        new RuleRunner().runRules(new String[]
{"Example4.dr1"}, cashflows;
    }
}

```

**SimpleDate** 「コンビニエンス」クラスは、入力文字列を取り日付形式を定義するコンストラクターを提供し、**java.util.Date** クラスを拡張します。コードは次の通りです。

## 例8.40 銀行取引のチュートリアル: クラス SimpleDate

```

public class SimpleDate extends Date
{
    private static final SimpleDateFormat format =
        new SimpleDateFormat("dd/MM/yyyy");

    public SimpleDate(String datestr) throws Exception
    {
        setTime(format.parse(datestr).getTime());
    }
}

```

ここで、ソートされた **cashflow** がどのように出力されるかを学ぶため、**cashflow** ファイルを検証します。

## 例8.41 銀行取引のチュートリアル: Example4.dr1 のルール

```

rule "Rule 04"
when
    $cashflow : Cashflow( $date : date, $amount : amount )

```

```

    not Cashflow( date &lt; $date)
then
    System.out.println("Cashflow: "+$date+" :: "+$amount);
    retract($cashflow);
end

```

この時点で **cashflow** を識別し、日付と金額を抽出できます。ルール of 2 行目では、見つかった **cashflow** よりも日付が古い **cashflow** が存在しないようにします。結果として、ルールを満たす **cashflow** を出力して取り消し、日付が次に新しい **cashflow** を処理できるようにします。

#### 例8.42 銀行取引のチュートリアル: Example4.java の出力

```

Loading file: Example4.drl
Inserting fact: Cashflow[date=Mon Jan 01 00:00:00 GMT 2007,amount=300.0]
Inserting fact: Cashflow[date=Fri Jan 05 00:00:00 GMT 2007,amount=100.0]
Inserting fact: Cashflow[date=Thu Jan 11 00:00:00 GMT 2007,amount=500.0]
Inserting fact: Cashflow[date=Sun Jan 07 00:00:00 GMT 2007,amount=800.0]
Inserting fact: Cashflow[date=Tue Jan 02 00:00:00 GMT 2007,amount=400.0]
Cashflow: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Cashflow: Tue Jan 02 00:00:00 GMT 2007 :: 400.0
Cashflow: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Cashflow: Sun Jan 07 00:00:00 GMT 2007 :: 800.0
Cashflow: Thu Jan 11 00:00:00 GMT 2007 :: 500.0

```

次に、**TypedCashflow** を作成するため **Cashflow** クラスを拡張します (入金または出金操作のいずれかになります)。通常、**Cashflow** に追加するのが最も簡単な方法ですが、ここでは以前のバージョンのクラスを保持するために拡張が使用されます。

```

public class TypedCashflow extends Cashflow {
    public static final int CREDIT = 0;
    public static final int DEBIT  = 1;

    private int type;

    public TypedCashflow() { }

    public TypedCashflow(Date date, int type, double amount)
    {
        super( date, amount );
        this.type = type;
    }

    public int getType()

```

```

{
    return type;
}

public void setType(int type)
{
    this.type = type;
}

public String toString()
{
    return "TypedCashflow[date=" + getDate()
+ ",type=" + (type == CREDIT ? "Credit" : "Debit")
+ ",amount=" + getAmount()
+ "]\n";
}
}

```

このコードはさまざまな方法で改善することが可能ですが、この例ではこのまま使用します。

次に、コードを実行するクラスを作成します。

#### 例8.43 銀行取引のチュートリアル: Example5.java

```

public class Example5
{
    public static void main(String[] args) throws Exception
    {
        Object[] cashflows = {
            new TypedCashflow(new SimpleDate("01/01/2007"),
TypedCashflow.CREDIT, 300.00),
            new TypedCashflow(new SimpleDate("05/01/2007"),
TypedCashflow.CREDIT, 100.00),
            new TypedCashflow(new SimpleDate("11/01/2007"),
TypedCashflow.CREDIT, 500.00),
            new TypedCashflow(new SimpleDate("07/01/2007"),
TypedCashflow.DEBIT, 800.00),
            new TypedCashflow(new SimpleDate("02/01/2007"),
TypedCashflow.DEBIT, 400.00),
        };

        new RuleRunner().runRules(
new String[] { "Example5.dr1" }, cashflows );
    }
}

```

**Cashflow** オブジェクトのセットが作成され、各オブジェクトは入金または出金操作のいずれかになります。これらのオブジェクトは **Example5.drl** とともに **RuleEngine** へ提供されます。

次に、このルールを検証します。このルールはソートされた **Cashflow** オブジェクトを出力します。

#### 例8.44 銀行取引のチュートリアル: Example5.drl のルール

```
rule "Rule 05"
when
    $cashflow : TypedCashflow( $date : date, $amount : amount,
        type == TypedCashflow.CREDIT )
    not TypedCashflow( date < $date, type == TypedCashflow.CREDIT )
then
    System.out.println("Credit: "+$date+" :: "+$amount);
    retract($cashflow);
end
```

この時点で、CREDIT 型を持つ **Cashflow** ファクトを識別し、日付と金額を抽出することができます。ルールの 2 行目では、見つかった **Cashflow** よりも日付が古い同じ型の **Cashflow** が存在しないようにします。結果として、ルールを満たす **Cashflow** を出力して取り消し、日付が次に新しい CREDIT 型の **Cashflow** を処理できるようにします。

生成される出力は次の例の通りです。

#### 例8.45 銀行取引のチュートリアル: Example5.java の出力

```
Loading file: Example5.drl
Inserting fact: TypedCashflow[date=Mon Jan 01 00:00:00 GMT
2007,type=Credit,amount=300.0]
Inserting fact: TypedCashflow[date=Fri Jan 05 00:00:00 GMT
2007,type=Credit,amount=100.0]
Inserting fact: TypedCashflow[date=Thu Jan 11 00:00:00 GMT
2007,type=Credit,amount=500.0]
Inserting fact: TypedCashflow[date=Sun Jan 07 00:00:00 GMT
2007,type=Debit,amount=800.0]
Inserting fact: TypedCashflow[date=Tue Jan 02 00:00:00 GMT
2007,type=Debit,amount=400.0]
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Credit: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Credit: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

次に、2つの銀行口座の入金と出金の両方を処理し、各口座の残高を計算します。最初に、**Account** オブジェクトを2つ作成し、**RuleEngine**へ渡す前に**Cashflow**クラスへインジェクトします(**helper**を用いずに正しい口座へ簡単にアクセスできるようにするため、この作業を行います)。

最初に **Account** クラスを見てみましょう。これは単純な Java オブジェクトで、口座番号と残高の両方を持ちます。

```
public class Account
{
    private long    accountNo;
    private double  balance = 0;

    public Account() { }

    public Account(long accountNo)
    {
        this.accountNo = accountNo;
    }

    public long getAccountNo()
    {
        return accountNo;
    }

    public void setAccountNo(long accountNo)
    {
        this.accountNo = accountNo;
    }

    public double getBalance()
    {
        return balance;
    }

    public void setBalance(double balance)
    {
        this.balance = balance;
    }

    public String toString()
    {
        return "Account[" + "accountNo=" + accountNo
        + ",balance=" + balance + "];"
    }
}
```

次に、**AllocatedCashflow** クラスを作成するため **TypedCashflow** を拡張し、**Account** 参照が含まれるようにします。

#### 例8.46 AllocatedCashflow クラス

```

public class AllocatedCashflow extends TypedCashflow
{
    private Account account;

    public AllocatedCashflow() {}

    public AllocatedCashflow(Account account, Date date,
        int type, double amount)
    {
        super( date, type, amount );
        this.account = account;
    }

    public Account getAccount()
    {
        return account;
    }

    public void setAccount(Account account)
    {
        this.account = account;
    }

    public String toString()
    {
        return "AllocatedCashflow["
            + "account=" + account
            + ",date=" + getDate()
            + ",type=" + (getType() == CREDIT ? "Credit" : "Debit")
            + ",amount=" + getAmount()
            + "];"
    }
}

```

**Example5.java** は2つの **Account** オブジェクトを作成します。コンストラクターの呼び出しにより **Account** オブジェクトの1つが各 **Cashflow** に渡されます。

#### 例8.47 銀行取引のチュートリアル: Example5.java

```

public class Example6
{
    public static void main(String[] args) throws Exception
    {
        Account acc1 = new Account(1);
        Account acc2 = new Account(2);

        Object[] cashflows =
        {
            new AllocatedCashflow(acc1,new SimpleDate("01/01/2007"),
                TypedCashflow.CREDIT, 300.00),

```

```

    new AllocatedCashflow(acc1,new SimpleDate("05/02/2007"),
        TypedCashflow.CREDIT, 100.00),
    new AllocatedCashflow(acc2,new SimpleDate("11/03/2007"),
        TypedCashflow.CREDIT, 500.00),
    new AllocatedCashflow(acc1,new SimpleDate("07/02/2007"),
        TypedCashflow.DEBIT, 800.00),
    new AllocatedCashflow(acc2,new SimpleDate("02/03/2007"),
        TypedCashflow.DEBIT, 400.00),
    new AllocatedCashflow(acc1,new SimpleDate("01/04/2007"),
        TypedCashflow.CREDIT, 200.00),
    new AllocatedCashflow(acc1,new SimpleDate("05/04/2007"),
        TypedCashflow.CREDIT, 300.00),
    new AllocatedCashflow(acc2,new SimpleDate("11/05/2007"),
        TypedCashflow.CREDIT, 700.00),
    new AllocatedCashflow(acc1,new SimpleDate("07/05/2007"),
        TypedCashflow.DEBIT, 900.00),
    new AllocatedCashflow(acc2,new SimpleDate("02/05/2007"),
        TypedCashflow.DEBIT, 100.00)
};

new RuleRunner().runRules(new String[]{"Example6.dr1"},cashflows);
}
}

```

ここで、**Example6.dr1** ファイルのルールを確認して、各 **Cashflow** がどのように日付順で適用されるか確認し、残高を計算および出力します。

```

rule "Rule 06 - Credit"
when
    $cashflow : AllocatedCashflow( $account : account,
        $date : date, $amount : amount, type==TypedCashflow.CREDIT )
    not AllocatedCashflow( account == $account, date < $date)
then
    System.out.println("Credit: " + $date + " :: " + $amount);
    $account.setBalance($account.getBalance()+$amount);
    System.out.println("Account: " + $account.getAccountNo() +
        " - new balance: " + $account.getBalance());
    retract($cashflow);
end

rule "Rule 06 - Debit"
when
    $cashflow : AllocatedCashflow( $account : account,
        $date : date, $amount : amount, type==TypedCashflow.DEBIT )
    not AllocatedCashflow( account == $account, date < $date)
then
    System.out.println("Debit: " + $date + " :: " + $amount);
    $account.setBalance($account.getBalance() - $amount);
    System.out.println("Account: " + $account.getAccountNo() +
        " - new balance: " + $account.getBalance());

```

```

    retract($cashflow);
end

```

これにより、入金と出金に対して異なるルールが存在することになりますが、以前の **Cashflow** をチェックする時に型を指定しないでください。これは、型に関係なくすべての **Cashflow** が日付順にチェックされるようにするためです。作業する口座を識別するために **conditions** が使用され、**Cashflow** の金額で更新するために **consequences** が使用されます。

```

Loading file: Example6.drl
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Jan 01
00:00:00 GMT 2007,type=Credit,amount=300.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Feb 05
00:00:00 GMT 2007,type=Credit,amount=100.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Sun Mar 11
00:00:00 GMT 2007,type=Credit,amount=500.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Wed Feb 07
00:00:00 GMT 2007,type=Debit,amount=800.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri Mar 02
00:00:00 GMT 2007,type=Debit,amount=400.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Sun Apr 01
00:00:00 BST 2007,type=Credit,amount=200.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Thu Apr 05
00:00:00 BST 2007,type=Credit,amount=300.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri May 11
00:00:00 BST 2007,type=Credit,amount=700.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon May 07
00:00:00 BST 2007,type=Debit,amount=900.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Wed May 02
00:00:00 BST 2007,type=Debit,amount=100.0]
Debit: Fri Mar 02 00:00:00 GMT 2007 :: 400.0
Account: 2 - new balance: -400.0
Credit: Sun Mar 11 00:00:00 GMT 2007 :: 500.0
Account: 2 - new balance: 100.0
Debit: Wed May 02 00:00:00 BST 2007 :: 100.0
Account: 2 - new balance: 0.0
Credit: Fri May 11 00:00:00 BST 2007 :: 700.0
Account: 2 - new balance: 700.0
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Account: 1 - new balance: 300.0
Credit: Mon Feb 05 00:00:00 GMT 2007 :: 100.0
Account: 1 - new balance: 400.0
Debit: Wed Feb 07 00:00:00 GMT 2007 :: 800.0

```



```

Account: 1 - new balance: -400.0
Credit: Sun Apr 01 00:00:00 BST 2007 :: 200.0
Account: 1 - new balance: -200.0
Credit: Thu Apr 05 00:00:00 BST 2007 :: 300.0
Account: 1 - new balance: 100.0
Debit: Mon May 07 00:00:00 BST 2007 :: 900.0
Account: 1 - new balance: -800.0

```

## 8.5. 価格ルールのデシジョンテーブルの例

名前:	価格決定のポリシー例
メインクラス:	<b>org.drools.examples.decisiontable.PricingRuleDTEExample</b>
タイプ:	Java アプリケーション
Rules ファイル:	<b>ExamplePolicyPricing.xls</b>
目的:	スプレッドシートベースのデシジョンテーブルの実証。

本チュートリアルは、スプレッドシートベースの デシジョンテーブル を使用して保険契約の小売価格を計算する方法を実証します。提供されているルールのセットは、特定の保険契約を申請するドライバーに対する基本価格と割引を計算します。基本保険料を決定するには、ドライバーの年齢、履歴、および契約タイプを考慮する必要があります。複数の追加ルールにより割引率を計算し、結果を絞り込みます。

### 8.5.1. 例の実行

**PricingRuleDTEExample.java** ファイルを開き、Java アプリケーションとして実行します。次の出力が **Console** ウィンドウに表示されます。

```

Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20

```

実行コードは標準パターンに準拠します。ルールがロードされ、ファクトが挿入され、さらに **stateless session** が作成されます。違いはルールが追加される方法にあります。

```

DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
    dtableconfiguration.setInputType( DecisionTableInputType.XLS );

    KnowledgeBuilder kbuilder =
    KnowledgeBuilderFactory.newKnowledgeBuilder();

```

```

        Resource xlsRes = ResourceFactory.newClassPathResource(
            "ExamplePolicyPricing.xls",
            getClass() );
        kbuilder.add( xlsRes,
            ResourceType.DTABLE,
            dtableconfiguration );

```

**DecisionTableConfiguration** オブジェクトの使用に注目してください。この入力型は **DecisionTableInputType.XLS** に設定されます。



### 注記

**Business Rules Management System** を使用している場合、すべて自動的に設定されます。

この例では、**Driver** と **Policy** の 2 つの **fact types** が使用されます。両方のファクトタイプのデフォルト値が使用されます。**Driver** の年齢は 30 歳で、以前に保険の請求がなく、リスクプロファイルは **LOW** になります。ドライバーが申請している **Policy** は **COMPREHENSIVE** で、まだ承認されていません。

## 8.5.2. デシジョンテーブル

このデシジョンテーブルでは、各行が 1 つのルールを表し、各列は条件またはアクションのいずれかを表します。

	C	D	E	F	G	H
<b>RuleSet</b>	org.drools.examples.decisiontable					
<b>Notes</b>	This decision table is for working out some basic prices and pretending actuaries don't exist					
<b>RuleTable Pricing bracket</b>						
<b>CONDITION</b>	CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION
<b>Driver</b>				policy: Policy		
age >= \$1, age <= \$2	locationRiskProfile	priorClaims	type	policy.setBasePrice(\$param);	System.out.println("\$param");	
<b>Age Bracket</b>	<b>Location risk profile</b>	<b>Number of prior claims</b>	<b>Policy type applying for</b>	<b>Base \$ AUD</b>	<b>Record Reason</b>	

図8.10 デシジョンテーブルの設定

上記のスプレッドシートを見てください。**RuleSet** 宣言がパッケージ名を提供することに注意してください。**Variables** (グローバル変数用) や **Imports** (クラスのインポートに使用) など、他のオプション項目をここに追加することも可能です。この例では、ルールの名前空間は **fact classes** の名前空間と同じであるため、省略されています。

下の方に **RuleTable** 宣言があります。後続の **Pricing Bracket** が、生成されるルールの名前のプレフィックスとして割り当てられます。

その下に **CONDITION** または **ACTION** があります。これは、列の目的を示しています。列が条件の一部または生成されたルールの結果を形成するかどうかを決定します。

ドライバーのデータが 3 つのセルにまたがることを確認してください。各ファクトの下にあるテンプレート式はこのデータに適用されます。ドライバーの年齢範囲のデータは **\$1** および **\$2** (コンマ区切りの値で入力)、**locationRiskProfile** および **priorClaims** (それぞれの列に存在) を使用します。

基本保険料は **action** 列に設定されます。ここにメッセージのログを記録することも可能です。

	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base \$ AUD	Record Reason
10	Young safe package	18, 24	LOW	1	COMPREHENSIVE	450	Priors not relevant
11			MED		FIRE_THEFT	200	
12			MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	
15	Young risk	18, 24	MED	1	COMPREHENSIVE	700	
16		18, 24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18, 24	HIGH		FIRE_THEFT	550	Location risk
18	Mature drivers	25, 30		0	COMPREHENSIVE	120	Cheapest possible
19		25, 30		1	COMPREHENSIVE	300	
20		25, 30		2	COMPREHENSIVE	590	
21		25, 35		3	THIRD PARTY	800	High risk

図8.11 基本保険料の計算

大まかなカテゴリー範囲は最も左にある列のコメントによって示されます。ドライバーと保険に関する既知の事実を用いて、手作業で基本コストを判断してみましょう。



## 注記

正解は Row Eighteen (このドライバーは過去に事故歴がないため) です。年齢は 30 歳で、基本価格は **120** になります。

29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18, 24	0	COMPREHENSIVE	1
31		18, 24	0	FIRE_THEFT	2
32		25, 30	1	COMPREHENSIVE	5
33		25, 30	2	COMPREHENSIVE	1
34		25, 30	0	COMPREHENSIVE	20

図8.12 割引の計算

次に、上記の条件を基に割引を算出します。割引は、Age の範囲、過去の請求数、契約タイプなどの、要素の組み合わせによって判断されます。この例では、ドライバーは 30 歳で、過去の請求歴がなく、**COMPREHENSIVE** 契約を申請しています。これにより、20 パーセントの割引が適応されます。



注記

割引情報は同じワークシートの別のテーブルに保存されます。そのため、異なるテンプレートを適用できます。



重要

デシジョンテーブルがルールを生成することを理解することが重要となります。単に「トップダウン」論理は使用されません。ルールが作成されるデータをキャプチャする方法として考えてみてください。この小さな違いがユーザーを混乱させることがあります。ルールの評価は、ルールエンジンの通常の仕組みが適用されるため、指定の順番で行われるとは限りません。

8.6. ペットストアの例

名前:	ペットストア
メインクラス:	<code>org.drools.examples.petstore.PetStoreExample</code>
タイプ:	Java アプリケーション
Rules ファイル:	<code>PetStore.drl</code>
目的:	アジェンダグループ、グローバル変数、およびグラフィカルユーザーインターフェースの統合 (ルール内からのコールバックを含む) の使用について実証します。

この例は、グラフィカルユーザーインターフェース (この例では **Swing** ベースのデスクトップアプリケーション) を備えたプログラムとともにルールを使用する方法を示します。

指定時に実行を許可されるルールセットのメンバーを決定するため、**Rules** ファイル内には **アジェンダグループ**および **自動フォーカス** 機能の使用方法を説明する例があります。さらに、この例は Java と MVFLEX 式言語のダイアレクトを組み合わせる方法も説明し、**accumulate** の使用やルールセット内から Java 関数を読み出す方法についても実証します。

Java コードはすべて **PetStore.java** ファイルに格納されます。このコードは次のプリンシパルクラスを定義します (このコードには、**Swing** イベントを処理するために使用されるマイナークラスも複数含まれます)。

- **Petstore - main()** メソッドが含まれます。
- **PetStoreUI - Swing** ベースのグラフィカルユーザーインターフェースを作成および表示します。マウスやボタンのクリックなど、主にさまざまな GUI イベントに応答する小型のクラスが複数含まれています。
- **TableModel** - テーブルデータを保持します。**Swing AbstractTableModel** クラスを拡張する Java Bean として考慮してください。
- **CheckoutCallback** - このクラスにより、グラフィカルユーザーインターフェースがルールと対話できます。
- **Ordershow** - ユーザーが購入を希望するアイテムを保持します。

- **Purchase** - 注文と購入済み製品の詳細を保存します。
- **Product** - 購入可能な製品の価格情報やその他の情報を保持する *Java Bean* です。



### 注記

ほとんどのコードが **Swing** ベースまたはプレーン **Java Beans** 形式になります。ここでは **Swing** について詳しく説明しませんが、Sun Microsystems の Web サイト <http://java.sun.com/docs/books/tutorial/uiswing/> に使用方法のチュートリアルがあります。

ルールとファクトに関連する **Petstore.java** ファイルの Java コードの一部は次の通りです。

#### 例8.48 PetStore.main にペットストアの RuleBase を作成

```
KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource( "PetStore.drl",
                                                    PetStore.class ),
              ResourceType.DRL );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );

// A callback is responsible for populating the
// Working Memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kbase ) );
ui.createAndShowGUI();
```

上記のコードは、クラスパス上に存在する **DRL** ファイルからルールをロードします。ファクトが即座にアサートされ実行される他の例とは異なり、この例ではファクトのアサートと実行が遅延されます。これは、**Vector** オブジェクトである **stock** を許可するコンストラクターを使用して **PetStoreUI** オブジェクトが作成される最後から2つ目の行によって決定されます。これは製品と、この直前にロードされた **rule-base** が含まれる **CheckoutCallback** クラスのインスタンスを収集します。

ルールを実行する実際の Java コードは **CheckoutCallBack.checkout()** メソッドによって呼び出されます。これは、ユーザーが **Checkout** ボタンをクリックするとトリガーされます。

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();
```

```

// Iterate through list and add to cart
for ( Product p: items ) {
    order.addItem( new Purchase( order, p ) );
}

// Add the JFrame to the ApplicationData to allow for user interaction

StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();
ksession.setGlobal( "frame", frame );
ksession.setGlobal( "textArea", this.output );

ksession.insert( new Product( "Gold Fish", 5 ) );
ksession.insert( new Product( "Fish Tank", 25 ) );
ksession.insert( new Product( "Fish Food", 2 ) );

ksession.insert( new Product( "Fish Food Sample", 0 ) );

ksession.insert( order );
ksession.fireAllRules();

// Return the state of the cart
return order.toString();
}

```

2つのアイテムがこのメソッドを介して渡されます。1つは出力されたテキストフレーム (グラフィカルユーザーインターフェースの最下部) を囲む **JFrame Swing** コンポーネントで、もう1つは注文商品のリストです。画面の右上にある **Table** エリアに表示される情報を保持する **TableModel** からこのリストが取得されます。

**for** ループは注文商品のリストを **Order** Java Bean に変換します (**PetStore.java** にあります)。



#### 注記

ルールの **Swing** データセットを直接参照することはできますが、このように単純な Java オブジェクトを使用する方法が推奨されます。このメソッドを使用すると、例を web アプリケーションに変換したい場合に **Swing** にバインドされなくなります。



#### 注記

この例に示されている ステート はすべて **Swing** コンポーネントに格納されます。ルール自体は実質的に「ステートレス」です。**Checkout** ボタンがクリックされるたびに **Swing TableModel** の内容がセッションの **working memory** にコピーされます。

このコード内には、**working memory** への呼び出しが9つあります。最初の呼び出しは新しい **working memory** ステートフルナレッジセッション (**Knowledge Base** 内) を作成します (この **Knowledge Base** は、**main()** メソッドで作成された **CheckoutCallBack** クラスで渡されることに注意してください)。

これに続く 2 つの呼び出しは、ルールによってグローバル変数として保持される 2 つのオブジェクトを渡します。これらのオブジェクトは **Swing** テキストエリアと **Swing** フレームで、メッセージの書き込みに使用されます。

さらなる 挿入 によって、製品自体の情報が **working memory** と注文リストの両方に格納されます。最後の呼び出しは標準の **fireAllRules()** メソッドです。次に、ルールが実行された時にこのメソッドが何を行うか見てみましょう。

#### 例8.49 パッケージ、インポート、グローバル、およびダイアレクト - **PetStore.dr1** より抜粋

```
package org.drools.examples

import org.drools.WorkingMemory
import org.drools.examples.PetStore.Order
import org.drools.examples.PetStore.Purchase
import org.drools.examples.PetStore.Product
import java.util.ArrayList
import javax.swing.JOptionPane;

import javax.swing.JFrame

global JFrame frame
global javax.swing.JTextArea textArea
```

**PetStore.dr1** ファイルの最初の部分には標準の **package** および **import** ステートメント (さまざまな Java クラスをルールが使用できるようにする) が含まれています。この他に、2 つの グローバル変数、**frame** および **textArea** があります。これらのグローバル変数は、**Swing** の **JFrame** コンポーネントおよび **Textarea** コンポーネントへの参照を保持します (Rules 変数は実行後すぐに期限切れとなりますが、グローバル変数はセッションの有効期間中、値が保持されます)。

次の例は **PetStore.dr1** ファイルの最後の部分から抜粋したものです。これには、ルールによって参照される 2 つの関数が含まれています (これら 2 つの関数については次の項で取り上げます)。

```
function void doCheckout(JFrame frame, WorkingMemory workingMemory)
{
    Object[] options = {"Yes", "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to checkout?", "",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, options, options[0]);

    if (n == 0) {workingMemory.setFocus( "checkout" );}
}

function boolean requireTank(JFrame frame, WorkingMemory workingMemory,
    Order order, Product fishTank, int total)
{
```

```

Object[] options = {"Yes", "No"};

int n = JOptionPane.showOptionDialog(frame,
    "Would you like to buy a tank for your " +
    total + " fish?",
    "Purchase Suggestion",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null, options, options[0]);

System.out.print( "SUGGESTION: Would you like to buy a tank for your "
    + total + " fish? - " );

if (n == 0) {
    Purchase purchase = new Purchase( order, fishTank );
    workingMemory.insert( purchase );
    order.addItem( purchase );
    System.out.println( "Yes" );
} else {
    System.out.println( "No" );
}
return true;
}

```



## 注記

これらの関数を **Rules** ファイルに格納すると、ペットストアの例がコンパクトになります。実際には、同じルールパッケージ内の個別のファイルに関数を格納するか、標準の Java クラスの静的メソッドとして個別のファイルに関数を格納する必要があります。これらの関数は **import function my.package.Foo.hello** でインポートします。

これら 2 つの関数の目的は次の通りです。

- **doCheckout()** は、ユーザーに支払い (チェックアウト) するかどうかを尋ねるダイアログボックスを表示します。支払いしたい場合、フォーカスが **checkOut** アジェンダグループに設定され、そのグループ内のルールに潜在的な実行パーミッションが与えられます。
- **requireTank()** は、水槽を購入したいかどうかをユーザーに尋ねるダイアログボックスを表示します。購入したい場合、**working memory** の注文リストに追加されます。

これらの関数を呼び出すルールについては本チュートリアルの後半で説明します。今後の例は、ペットストアルールから派生します。最初の抽出は最初に実行されます (**auto-focus** 属性が **true** に設定されていることが部分的に関係します)。

### 例8.50 アイテムをワーキングメモリーに格納 - PetStore.drl ファイルより抜粋

```

/// Insert each item in the shopping cart into the Working Memory
// Insert each item in the shopping cart into the Working Memory
rule "Explode Cart"
    agenda-group "init"

```



```

    auto-focus true
    salience 10
    dialect "java"
when
    $order : Order( grossTotal == -1 )
    $item : Purchase() from $order.items
then
    insert( $item );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show
items" ).setFocus();
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup(
"evaluate" ).setFocus();
end

```

このルールは、総合計 (**PetStore.dr1**) が算出されていないすべての注文に対して一致します。順番に各購入アイテムをループします。ルール名、*salience* (ルールが実行される順番を提示)、ダイアレクト (Java に設定) など、**Explode Cart** ルールにはすでに説明した項目があります。また、ルールには新しいアイテムが3つあります。

- **agenda-group "init"** - アジェンダグループの名前です。この例では、グループに1つのルールしか存在しません。しかし、Java コードとルール結果はどちらもフォーカスをこのグループに設定しないため、実行する機会を得られるかどうかは次の属性に依存します。
- **auto-focus true** は、Java コードから **fireAllRules()** が呼び出された時に、アジェンダグループの唯一のルールが実行する機会を得られるようにします。
- **drools.setFocus()** は **show items** および **evaluate** アジェンダグループに順番にフォーカスを提供し、ルールの実行を許可します (実際には、確実に注文アイテムがメモリーに挿入されるよう、注文アイテムはすべてループされます。他のルールはその後順次実行されます)。

次の2つのリストは、**show items** および **evaluate** アジェンダグループのルールを表しています。

#### 例8.51 GUI のアイテムを表示 - PetStore.dr1 ファイルより抜粋

```

rule "Show Items"
    agenda-group "show items"
    dialect "mvel"
when
    $order : Order( )
    $p : Purchase( order == $order )
then
    textArea.append( $p.product + "\n");
end

```

**show items** アジェンダグループが最初に呼び出されます。これには、**Show Items** (小文字と大文字の違いに注意してください) と呼ばれるルールのみがあります。このルールは、各購入のログ詳細をテキストエリア (GUI 画面の下部) へ移動します (この操作に使用される **textArea** 変数は先に説明したグローバル変数の 1 つです)。

**evaluate** アジェンダグループも先にリストされた **explode cart** よりフォーカスを取得します。このアジェンダグループには、**Free Fish Food Sample** と **Suggest Tank** の 2 つのルールがあります。

#### 例8.52 Evaluate アジェンダグループ: PetStore.dr1 ファイルより抜粋

```
// Free Fish Food sample when we buy a Gold Fish if we have not already
//bought Fish Food and dont already have a Fish Food Sample
rule "Free Fish Food Sample"
  agenda-group "evaluate"
  dialect "mvel"
when
  $order : Order()
  not ( $p : Product( name == "Fish Food") &&
    Purchase( product == $p ) )
  not ( $p : Product( name == "Fish Food Sample") &&
    Purchase( product == $p ) )
  exists ( $p : Product( name == "Gold Fish") &&
    Purchase( product == $p ) )
  $fishFoodSample : Product( name == "Fish Food Sample" );
then
  System.out.println( "Adding free Fish Food Sample to cart" );
  purchase = new Purchase($order, $fishFoodSample);
  insert( purchase );
  $order.addItem( purchase );
end

// Suggest a tank if we have bought more than 5 gold fish and do not
// already have one
rule "Suggest Tank"
  agenda-group "evaluate"
  dialect "java"
when
  $order : Order()
  not ( $p : Product( name == "Fish Tank") &&
    Purchase( product == $p ) )
  ArrayList( $total : size > 5 ) from collect( Purchase
    ( product.name == "Gold Fish" ) )
  $fishTank : Product( name == "Fish Tank" )
then
  requireTank(frame, drools.getWorkingMemory(),
    $order, $fishTank, $total);
end
```

**Free Fish Food Sample** ルールは以下の場合のみ実行されます。

- ユーザーが魚の餌を持っていない場合。
- ユーザーが魚の餌の無料サンプルを持っていない場合。
- ユーザーの注文に金魚が含まれる場合。

これらの条件が満たされると、ルールが実行され、新しい商品 (**Fish Food Sample**) が生成され、**Fish Food Sample** の注文に追加されます。

同様に、**Suggest Tank** ルールは次の 2 つの条件が満たされた場合のみ実行されます。

- ユーザーが水槽を注文していない場合。
- ユーザーが金魚関係の製品を 6 つ以上注文した場合。

これらの条件が満たされると、ルールが実行され、ユーザーにダイアログボックスを順番に表示する **requireTank()** 関数が呼び出されます。確認後、水槽が注文に追加されます。ルールが **requireTank()** 関数を呼び出す時、ルールがグローバル **frame** 変数を渡すことに注意してください。これは、関数が **Swing** グラフィカルユーザーインターフェースのハンドルを持つからです。

次のルールは **do checkout** と呼ばれます。

#### 例8.53 チェックアウトの実行: PetStore.dr1 ファイルより抜粋

```
rule "do checkout"
  dialect "java"
  when
  then
    doCheckout(frame, drools.getWorkingMemory());
  end
```

**do checkout** ルールには設定されたアジェンダグループや **auto-focus** 属性がありません。そのため、デフォルトのアジェンダグループの一部と見なされ、明示的なフォーカスを受け取るよう設定されたルールがすべて完了したときに、フォーカスを自動的に受け取ります。

ルールには左側がないため、常に右側が **doCheckout()** 関数を呼び出します。この場合、ルールはグローバル **frame** 変数を渡し、**Swing** グラフィカルユーザーインターフェースのハンドルを関数に付与します (上記の通り、**doCheckout()** 関数はユーザーに確認ダイアログボックスを表示します。確認後、関数はフォーカスを **checkout** アジェンダグループに渡し、次のルールセットを実行できるようにします)。

#### 例8.54 チェックアウトのルール: PetStore.dr1 ファイルより抜粋

```
rule "Gross Total"
  agenda-group "checkout"
  dialect "mvel"
  when
    $order : Order( grossTotal == -1)
```

```

    Number( total : doubleValue ) from accumulate( Purchase ( $price :
product.price ),sum( $price ) )
then
    modify( $order ) { grossTotal = total };
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
    agenda-group "checkout"
    dialect "mvel"
when
    $order : Order( grossTotal >= 10 && < 20 )
then
    $order.discountedTotal = $order.grossTotal * 0.95;
    textArea.append("discountedTotal total="+$order.discountedTotal+"\n");
end

rule "Apply 10% Discount"
    agenda-group "checkout"
    dialect "mvel"
when
    $order : Order( grossTotal >= 20 )
then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append("discountedTotal total="+$order.discountedTotal+"\n");
end

```

**checkout** アジェンダグループには 3 つのルールがあります。

- **Gross Total** - まだ実行されていない場合、製品価格を累積し、合計を **working memory** に保存して、**Swing Text Area** に表示します (**textArea** グローバル変数を使用)。
- 総合計が 10 と 20 の間になる場合、**Apply 5% Discount** が割引後の合計を計算し、**working memory** へ追加してテキストエリアに表示します。
- 総合計が 20 を越える場合、**Apply 10% Discount** が割引後の合計を計算し、**working memory** へ追加してテキストエリアに表示します。

これが理論的なコードの仕組みの概要になります。ここで、実際に何が起こるか確認する必要があります。**PetStore.java** というファイルには **main()** メソッドが含まれているため、コマンドラインまたは IDE 内より標準の Java アプリケーションとして実行することができます (クラスパスが適切に設定されていることを前提とします)。

最初の画面には **Pet Store Demo** が含まれます。これには使用可能な製品 (左上) のリスト、選択された製品の空白のリスト (右上)、チェックアウトおよびリセットボタン (真ん中)、空白のシステムメッセージエリア (最下部) が含まれます。

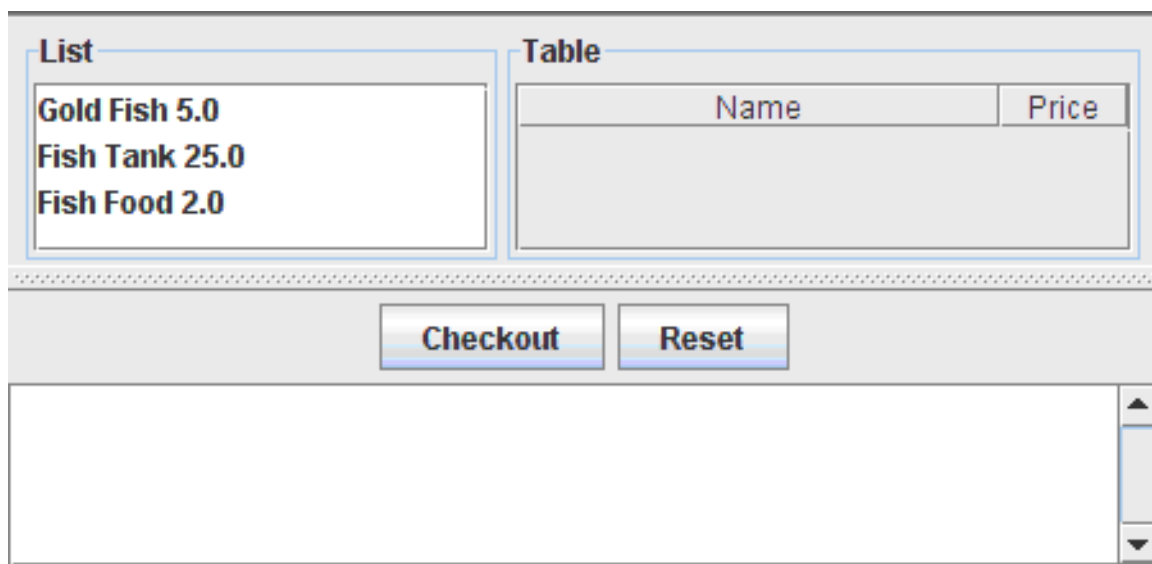


図8.13 起動直後の Pet Store Demo

この時点に到達するまで、次のイベントが発生します。

1. `main()` メソッドが実行され、ルールベースがロードされますが、ルールはまだ実行されません (ここまで、このコードがルールと何らかの関係がある唯一のコードになります)。
2. 新しい **PetStoreUI** オブジェクトが作成され、ルールベースのハンドルが付与されます。このハンドルは後で使用します。
3. さまざまな **Swing** コンポーネントが操作を実行します。この時点で初めて上記の画面が表示され、ユーザーによる入力を待ちます。

リストでさまざまな製品をクリックし、以下に似た画面が表示されるか確認します。

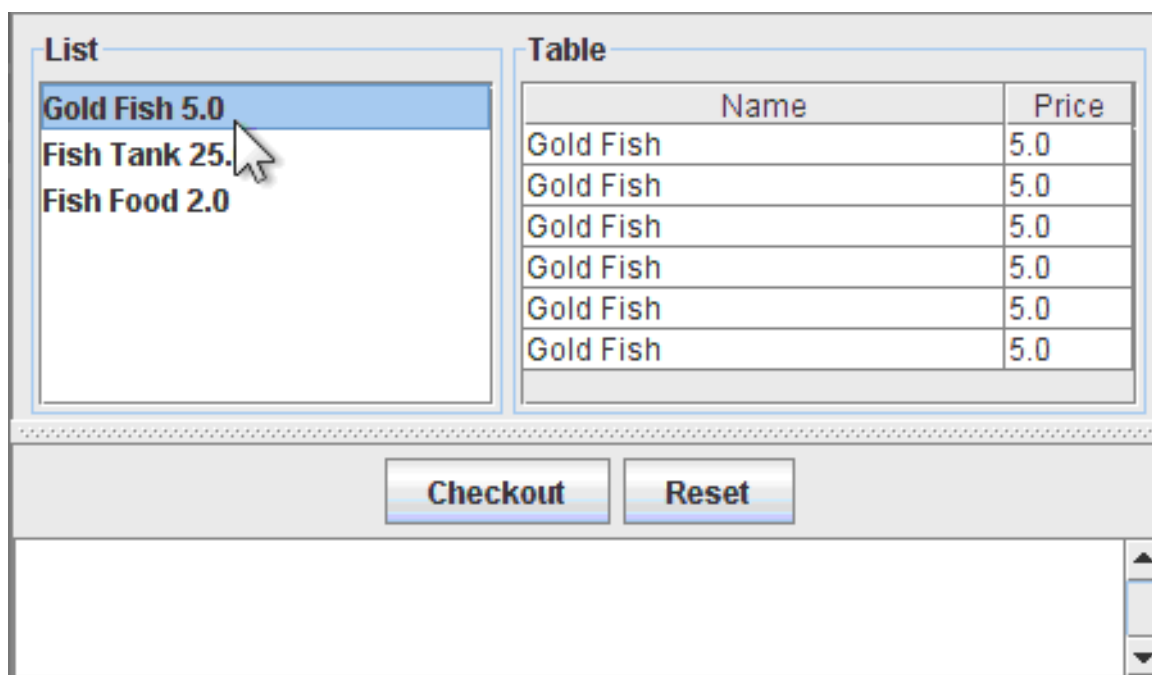


図8.14 製品が選択された Pet Store Demo



## 注記

繰り返しになりますが、ルールコードはまだ実行されていません。これは **Swing** コードで、マウスクリックイベントを「リッスン」し、選択された製品を **TableModel** オブジェクトへ追加して右上の部分に表示します (これは *Model View Controller* 設計パターンの従来の実装であることに注意してください)。

**Checkout** がクリックされた時のみビジネスルールが実行され、実行順は前述の順番とほぼ同じになります。

1. **Checkout** ボタンがクリックされると、**Swing** クラスによって **CheckoutCallback.checkout()** メソッドが呼び出されます。このメソッドは **TableModel** オブジェクト (グラフィカルユーザーインターフェースの右上に表示されます) よりデータを挿入し、さらにセッションの **working memory** に置きます。その後、ルールが実行されます。
2. **Explode Cart** ルールは、自動フォーカス設定が **true** であるため、最初に実行されます。カート内のすべての製品をループし、確実にこれらの製品が **working memory** に存在するようにします。次に、**Show Items** および **Evaluation** アジェンダグループに実行パーミッションを与えます。これらのグループにあるルールは、カートの内容をテキストエリア (ウィンドウの下部) に追加します。さらに、魚の餌をユーザーに無料で提供するか決定し、水槽を購入するかどうかをユーザーに尋ねます。水槽購入の画面は次のようになります。



図8.15 水槽を購入しますか？

3. **Do Checkout** ルールが次に実行されます。これは、現在フォーカスを持つアジェンダグループが他になく、デフォルトのアジェンダグループの一部であるためです。このルールは、次の質問が含まれるダイアログボックスを表示する **doCheckout()** 関数を常に呼び出します。

```
"Would you like to Checkout?"
```

**doCheckout()** 関数はフォーカスを **checkout** アジェンダグループに設定し、このグループのルールが実行する選択肢を与えます。

4. **checkout** アジェンダグループのルールは「カート」の内容を表示し、適切な割引を適用します。
5. 下図の通り、さらに製品をチェックアウトするか (これによりルールが再度実行されます) またはグラフィカルユーザーインターフェースが閉じられるかに応じて、**Swing** はユーザーの入力を待ちます。

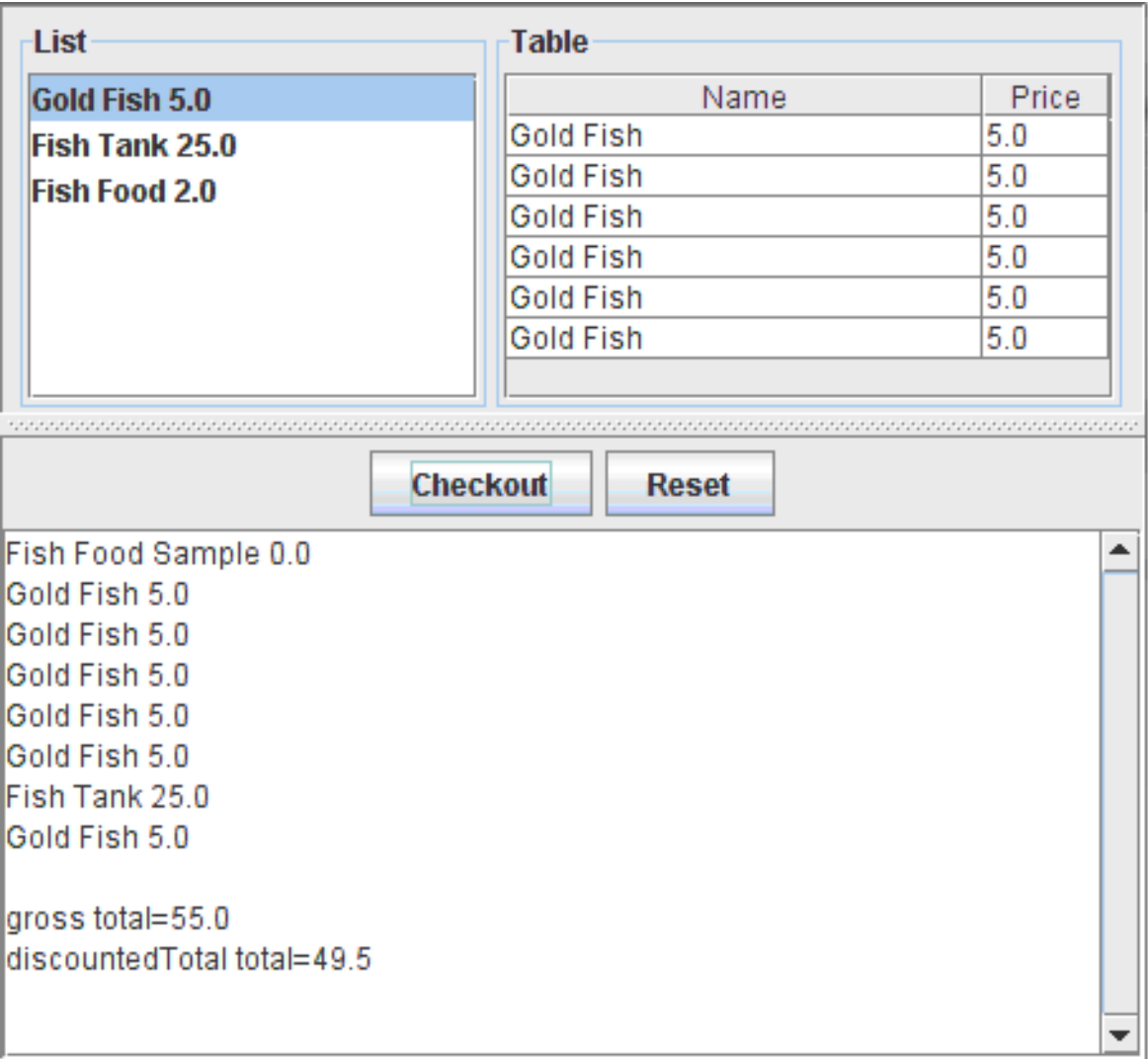


図8.16 ルールがすべて実行され、アプリケーションが終了

**System.out** 呼び出しをさらに追加して、イベントの流れを実証することもできます。上記の例に表された **Console** の出力は次のようになります。

**例8.55 ペットストア GUI 実行後のコンソール出力**

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

**8.7. 数独の例**

名前:	数独
メインクラス:	org.drools.examples.sudoku.Main

タイプ:	Java アプリケーション
Rules ファイル:	<b>sudokuSolver.drl, sudokuValidator.drl</b>
目的:	論理的な問題を解決する方法を実証し、複雑なパターンマッチングを使用する方法を説明します。

この例では、**JBoss Rules** を使用して、複数の制約から派生される潜在的に大きな ソリューションスペースで回答を見つける方法を実証します。また、*callbacks* を使用して、実行時の **working memory** への変更に基づいて画面を更新し、**JBoss Rules** とグラフィカルユーザーインターフェースベースのアプリケーションを統合する方法についても説明します。

### 8.7.1. 数独の概要

数独は、日本で生まれた論理ベースの数字配置パズルです。このパズルの目的は、各列、行、および 3x3 の「ゾーン」に 1 から 9 までの数字が一度だけ含まれるよう、9x9 の正方形グリッドを埋めることです。

プレイヤーには、一部が完成されたグリッドと、ルールを守りながら完成させるタスクが与えられます。

プレイヤーが追加する新しい数字は、特定の行、列、および 3x3 のブロックで同時に一意である必要があります。

### 8.7.2. 例の実行

前述の手順に従って **drools-examples** ファイルをダウンロードし、インストールします。その後、**java org.drools.examples.sudoku.Main** を実行します (この例には **Java 5** が必要です)。比較的簡単な一部が埋められたグリッドがウインドウに表示されます。

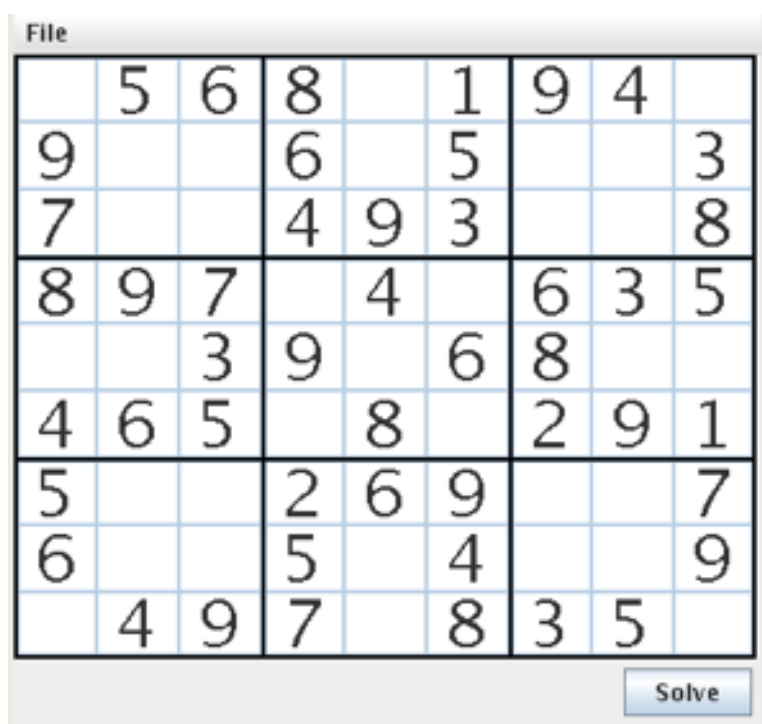


図8.17 一部が埋められたグリッド



**Solve** ボタンをクリックすると **rules engine** が残りの値を埋めます。**Console** はルールが実行されるとルールの詳細情報を表示し、パズルの解き方を実証します。

```
Rule #3 determined the value at (4,1) could not be 4 as this value already
exists in the same column at (8,1)
Rule #3 determined the value at (5,5) could not be 2 as this value already
exists in the same row at (5,6)
Rule #7 determined (3,5) is 2 as this is the only possible cell in the
column that can have this value
Rule #1 cleared the other PossibleCellValues for (3,5) as a
ResolvedCellValue of 2 exists for this cell.
Rule #1 cleared the other PossibleCellValues for (3,5) as a
ResolvedCellValue of 2 exists for this cell.
...
Rule #3 determined the value at (1,1) could not be 1 as this value already
exists in the same zone at (2,1)
Rule #6 determined (1,7) is 1 as this is the only possible cell in the row
that can have this value
Rule #1 cleared the other PossibleCellValues for (1,7) as a
ResolvedCellValue of 1 exists for this cell.
Rule #6 determined (1,1) is 8 as this is the only possible cell in the row
that can have this value
```

「ロジックの解決」ルールがすべてアクティベートおよび実行されると、**engine** は2番目の **rule base** を処理します。これにより、解答が完全で有効であることがチェックされます。この例では、すべてが完全で有効であるため、結果として **Solve** ボタンが無効になり、以下のメッセージが表示されます。

Solved (1052ms)

File								
3	5	6	8	7	1	9	4	2
9	8	4	6	2	5	7	1	3
7	1	2	4	9	3	5	6	8
8	9	7	1	4	2	6	3	5
1	2	3	9	5	6	8	7	4
4	6	5	3	8	7	2	9	1
5	3	1	2	6	9	4	8	7
6	7	8	5	3	4	1	2	9
2	4	9	7	1	8	3	5	6

Solved (954 ms)

図8.18 解決済みのグリッド

この例にはロードおよび解決可能な複数のグリッドが含まれています。[File]、[Samples]、[Medium]の順にクリックし、より難易度が高いグリッドをロードします (新しいグリッドがロードされると、**Solve** ボタンが再度有効になることに注目してください)。

このグリッドを少し試した後、[File]、[Samples]、[!DELIBERATELY BROKEN!]の順にクリックし、意図的に無効なグリッドをロードします。間違いを見つけてみてください (たとえば、最初の行に **5** が 2 つあります)。

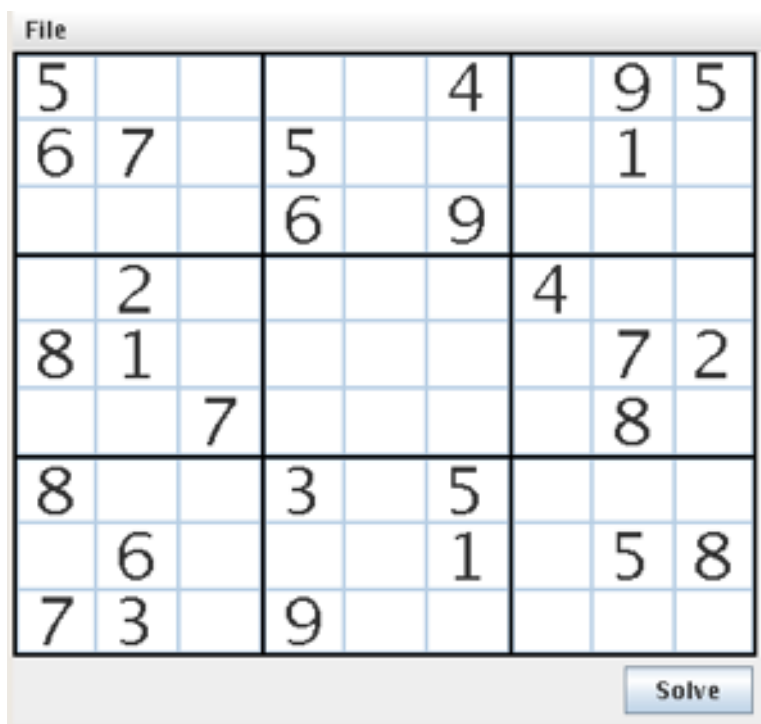


図8.19 無効なグリッド

無効であることを確認した後、**Solve** ボタンをクリックし、無効なグリッドに「ルールの解決」が適用された場合に何が起こるか見てみます。

解答が無効であることを示すために **Solve** ボタンのラベルが変更され、**Validation Rule Set** が発見した問題をすべて **Console** に出力することを確認してください。

```
There are two cells on the same column with the same value at (6,0) and
(4,0)
There are two cells on the same column with the same value at (4,0) and
(6,0)
There are two cells on the same row with the same value at (2,4) and (2,2)
There are two cells on the same row with the same value at (2,2) and (2,4)
There are two cells on the same row with the same value at (6,3) and (6,8)
There are two cells on the same row with the same value at (6,8) and (6,3)
There are two cells on the same column with the same value at (7,4) and
(0,4)
There are two cells on the same column with the same value at (0,4) and
(7,4)
There are two cells on the same row with the same value at (0,8) and (0,0)
There are two cells on the same row with the same value at (0,0) and (0,8)
There are two cells on the same column with the same value at (1,2) and
(3,2)
There are two cells on the same column with the same value at (3,2) and
(1,2)
There are two cells in the same zone with the same value at (6,3) and
```

```
(7,3)
There are two cells in the same zone with the same value at (7,3) and
(6,3)
There are two cells on the same column with the same value at (7,3) and
(6,3)
There are two cells on the same column with the same value at (6,3) and
(7,3)
```

理論的に解決可能なパズルの解答によっては、現状態で **engine** が実際に見つけられないものがあります。例を確認するには、[File]、[Samples]、[Hard 3] の順にクリックします。一部が埋められたグリッドがロードされます。

ここで、**Solve** ボタンをクリックし、人間が解答を見つけられる可能性があっても、現在のルールがグリッドを完成できないことを確認してください。

これまで、プログラムはルールが 10 個のセットを使用して解答を推測しました。このルールセットを拡張し、このような難解度の高いパズルを解くために必要なロジックを **engine** に提供する必要があります。

### 8.7.3. Java ソースとルールの概要

Java ソースは `/src/main/java/org/drools/examples/sudoku` ディレクトリにあります。2 つの **DRL** ルール定義ファイルは `/src/main/rules/org/drools/examples/sudoku` ディレクトリにあります。

**org.drools.examples.sudoku.swing** パッケージには、数独パズルのフレームワークを実装するクラスセットが含まれています。



#### 注記

このパッケージは **JBoss Rules** ライブラリに依存しません。

数独パズルを整数値の 9x9 グリッドとして保存するため、**SudokuGridModel** インターフェースを実装します (値はセルの値が決定していないことを示す `null` である場合があります)。

**SudokuGridView** は **Swing** コンポーネントです。**SudokuGridModel** の実装をグラフィカルに表すことができます。

**SudokuGridEvent** および **SudokuGridListener** はモデルのステート変更をビューに通知するため使用されます。セルの値が解決または変更されると、イベントが実行されます。**JTable** など、他の **Swing** コンポーネントで使用されるモデルビューコントローラーパターンの知識があるユーザーは、この挙動を理解できるはずです (このような概念を実証するため、**SudokuGridSamples** は部分的に完成されたパズルを複数提供します)。

**org.drools.examples.sudoku.rules** パッケージには、**JBoss Rules** に基づいた **SudokuGridModel** の実装が含まれています。**AbstractCellValue** を拡張し、グリッドの特定セルの値を表す 2 つの Java オブジェクトが使用されます。この値には、セルの行と列の場所、値が含まれる 3x3 ゾーンのインデックス番号、およびセルに保持される実際の番号が含まれます。

**PossibleCellValue** はセルの値が現在不明であることを示しています (どのセルにも 2 から 9 個の可能な値があります)。

**ResolvedCellValue** は、セルの値が決定されたことを示します。セルの解決された値は 1 つのみです。

**DroolsSudokuGridModel** は **SudokuGridModel** を実装します。これは、一部指定されたセルのレイ (最初は 2 次元) を **CellValue** Java オブジェクトのセットに変換します。これにより、**solverSudoku.drl** ルールファイルに基づいて **working memory** セッションが作成されます。また、**CellValue** オブジェクトも **working memory** に挿入します。

**solve()** メソッドが呼び出されると、パズルを解こうとする **fireAllRules()** が呼び出されます。

**DroolsSudokuGridModel** は **WorkingMemoryListener** を **working memory** にアタッチします。これにより、パズルが解決された時に挿入および取り消しイベントで呼び戻すことが可能です。新しい **ResolvedCellValue** が **working memory** に挿入されると、このコールバックにより実装は **SudokuGridEvent** を **SudokuGridListener** クライアントに対して実行することができ、リアルタイムで更新します。

"**solver**" **working memory** によってルールがすべて実行されると、**DroolsSudokuGridModel** が 2 番目のルールセットを実行します (これらのルールは **validatorSudoku.drl** ファイルから取得されます)。これらのルールは同じ Java オブジェクトのセットと動作し、結果となるグリッドが有効で完全な解答であるかを判断することが目的です。



#### 注記

**org.drools.examples.sudoku.Main** クラスは上記のコンポーネントを組み合わせる Java アプリケーションを実装します。

**org.drools.examples.sudoku** パッケージには、**solverSudoku.drl** と **validator.drl** の 2 つの **DRL** ファイルが含まれています。**solverSudoku.drl** は数独パズルを解くために使用されるルールを定義します。**validator.drl** は、**working memory** の現在の状態が有効な解答を表すかどうかをテストするルールを定義します。これらのルールセットは **PossibleCellValue** および **ResolvedCellValue** オブジェクトをファクトとして使用します。また、両方とも実行時に **Console** ウィンドウヘッダーを出力します。



#### 注記

実際には、このように **Console** を使用するのではなく、ロギング情報を挿入し、**WorkingMemoryListener** を使用して出力をユーザーに表示します。

### 8.7.4. 検証ルール

**validatorSudoku.drl** ファイルにある検証ルールが従うプロセスは次の通りです。

1. 最初のルールは、**working memory** に **PossibleCellValue** オブジェクトがないことを確認します (パズルが解かれると、各セルに **ResolvedCellValue** オブジェクト 1 つのみが存在するはずです)。
2. 他の 3 つのルールは **ResolvedCellValue** オブジェクトすべてと一致し、**\$resolved1** という変数へバインドされます。そして、同じ値が含まれ、同じ行、列および 3x3 ゾーンにある **ResolvedCellValues** を探します。
3. これらのルールが実行されると、解答が無効である理由を示すメッセージが文字列のグローバルリストに追加されます。**DroolsSudokuGridModel** はルールセットを実行する前にこのリストをインジェクトします (また、**fireAllRules()** を呼び出した後にこのリストが空であるかどうかを確認します。空でない場合、リストに全文字列を出力します。空でない場合、すべての文字列がリストに出力され、グリッドが解決していないことを示すフラグが設定されます)。

### 8.7.5. ルールの解決

これは、**solverSudoku.dr1** ファイルにある「解決」ルールの前に実行されるさらに複雑なプロセスです。

1. ルール #1 は、無効な解答が存在する行と列に対応する **PossibleCellValues** の **working memory** を消去します。他の複数のルールは、セルに値が必要であると判断した後、特定行および列の **working memory** に **ResolvedCellValues** を挿入します。

このルールは重要であるため、他のルールよりも大きな **salience** の値が指定されます。そのため、左側が **true** になると、即座に **activations** がアジェンダの最上部に移動され、実行されます。これにより、他のルールが誤って実行されないようにします。

このルールは **ResolvedCellValue** 上で **update()** も実行します (これは、**working memory** にアタッチされた **WorkingMemoryListeners** へ **JBoss Rules** がイベントを送信するようにルールが変更されていなくても発生します。このようなイベントを実行すると、ルールがルール自体を更新できます)。これにより、グラフィカルユーザーインターフェースがグリッドの新しい状態を表示します。

2. ルール #2 は、可能な値が 1 つのみであるグリッドのセルを識別します。**when** 句の最初の行は、**working memory** の **PossibleCellValue** オブジェクトすべてと一致します。2 行目は **not** キーワードの使用法を表しています。



#### 注記

このルールは、異なる値を持つ同じ行と列の **PossibleCellValue** が他にない場合のみ実行されます。

このルールが実行されると、その行と列の単一の **PossibleCellValue** が **working memory** から取り消され、同じ値の **ResolvedCellValue** に置き換えられます。

3. ルール #3 は、**ResolvedCellValue** と同じ値がある場合に行から **PossibleCellValues** を削除します。つまり、解決された値がセルに含まれる場合、パズルのルールに従うため同じ値が含まれる同じ行の他のセルを消去する必要があります。**when** 句の最初の行は、**working memory** の **ResolvedCellValue** オブジェクトをすべて見つけます。2 番目の行は、これらの **ResolvedCellValue** オブジェクトと同じ行と値を持つ **PossibleCellValues** を見つけます。このような **PossibleCellValues** が見つかり、ルールがアクティベートされ、実行された時に取り消されます。これは、これらのセルに対する適切な解答ではないためです。
4. ルール #4 と #5 はルール #3 と同じように動作しますが、**ResolvedCellValues** と競合する冗長な **PossibleCellValues** がないか列とゾーンをチェックします。
5. ルール #6 は、セルの可能な値が該当する行に 1 つだけある場合をチェックします。左側の最初の行は **working memory** の **PossibleCellValue** ファクトすべてと一致し、結果を複数のローカル変数に格納します。2 行目は、同じ値を持つ他の **PossibleCellValue** オブジェクトが同じ行に存在しないことをチェックします。3 - 5 行目は、同じ値を持つ **ResolvedCellValue** が同じゾーン、行、または列に存在しないことをチェックします (これは、このルールが不完全な状態で実行されないようにするためです)。



## 注記

また、3 - 5 行目を行を削除し、ルール #3、#4、および #5 の **salience** の値を大きくしてこれらのルールが常にルール #6、#7、および #8 の前に実行されるようにすることも可能です。

ルールが実行された場合、**\$possible** はセルの値を表す必要があります。そのため、これを取り消し、同等の **ResolvedCellValue** に置き換えます (これはルール #2 のプロセスと同じです)。

6. ルール #7 と #8 はルール #2 と同じように動作しますが、#7 はグリッドの該当列にある単一の **PossibleCellValues** をチェックし、#8 はグリッドの該当ゾーンにある **PossibleCellValues** をチェックします。
7. ルール #9 は最も複雑です。これは「該当する値のペアが特定行の 2 つのセルにのみあり (たとえば、4 と 6 の値はセル [0,3] および [0,5] の最初の行のみに存在できることを判断しました)、このセルのペアが他の値を保持できない場合、4 が含まれるペアと 6 が含まれるペアがどれであるは分かりませんが、これらの値がこれらの 2 つのセルになければならないことはわかっています。そのため、これらの値が同じ行の他の場所で発生する可能性を排除することができます」ということです。
8. ルール #10 と #11 はルール #9 と同じように動作しますが、#10 は該当列に可能な値が 2 つのみ存在することをチェックし、#8 は該当ゾーンに可能な値が 2 つのみ存在することをチェックします。

さらに難解なパズルを解くため、さらに複雑な理由付けをカプセル化するカスタムルールをコーディングし、ルールセットを拡張します。次項では、この作業を行う方法について提案します。

### 8.7.6. 将来的な開発に関する提言

この例を開発する方法は複数あります。技術を身に付けるため、次の提言を演習として行うようにしてください。

- **Agenda Groups**: 段階的な実行に **宣言ツール** を使用します。この例では、「解決」と「検証」の 2 つの段階があることが分かるはずですが。現在、これらは 2 つのルールベースファイルより実行されます。「解決」と「検証」のカテゴリーに分類し、すべてのルールに対してアジェンダグループを定義することが推奨されます。その後、すべてが単一の **rule-base** よりロードされます。**engine** はこれらを順番に即座に実行します。
- **自動フォーカス**: このメソッドを使用し、通常のルール実行の例外に対応します。現時点では、入力データまたは解決ルールのどちらかに不整合があった場合、そのまま無駄にルールを実行せずに、即座に報告した方がよいでしょう。すでに単一の **rule-base** が作成されているため、パズルの一貫性の検証に使用される各ルールに対して **auto-focus** 属性を定義します。
- **論理挿入**: 不整合は **working memory** に誤ったデータがある場合のみ発生します。そのため、したがって、検証ルールが不整合を **理論的に挿入** と言えます (誤ったデータが取り消されると、すぐに不整合はなくなります)。
- **session.iterateObjects()**: 現時点では、グローバルリストは問題を記録するために使用されますが、**session.iterateObjects( new ClassObjectFilter( Inconsistency.class ) )** を介して必要な問題と呼び出すよう **stateful session** に要求すると、より興味深いでしょう。**inconsistency** を使用すると、不正なセルに色 (赤など) を付けて見やすくすることもできます。
- **kcontext.getKnowledgeRuntime().halt()**: ソフトウェアが検出したエラーを即座に報

告した場合でも、ルールの評価を停止するようエンジンに伝える方法が必要です。これは、不整合が存在する時に `halt()` コードを呼び出すルールをプログラミングして作成します。

- **Queries: DroolsSudokuGridModel** の `getPossibleCellValues(int row, int col)` メソッドを見てください。実際に必要な **CellValue** オブジェクトを探しながらすべての **CellValue** オブジェクトを繰り返すことが分かります。**JBoss Rules** クエリを使用すると、この処理の効率を向上することができます。必要なオブジェクトを返し、きれいに繰り返すクエリを定義します (必要に応じて他のクエリを定義します)。
- **サービスとしてのグローバル**: この変更の主な目的は、これに続くものを容易にすることですが、これ自体にも便利です。**サービスとしてのグローバル** の使用法について学ぶには、最初に **callback** を設定することが理想的です。この変更は、該当のセルに対して **ResolvedCellValue** を見つける各ルールがグラフィカルユーザーインターフェースを「呼び出し」て、更新することができ、ユーザーにフィードバックを即座に提供することを意味します。また、最後に見つかったセルの番号を違う色に「塗り替え」、異なるルールの結果を素早く識別できるようにすることもできます。
- **ステップバイステップ実行**: 即座にユーザーフィードバックを提供できるようコールバックを設定すると、**JBoss Rules** の **制約実行機能** を使用できます。グラフィカルユーザーインターフェースにボタンを追加し、クリックした時に `fireAllRules( 1 )` を呼び出して1つのルールを実行します。このようにして、ユーザーはエンジンが行っていることをステップごとに確認できます。

## 8.8. 数字当て

名前:	数字当て
メインクラス:	<code>org.drools.examples.numberguess.NumberGuessExample</code>
タイプ:	Java アプリケーション
Rules ファイル:	<code>NumberGuess.drl</code>
目的:	ルールを整理するためルールフローを使用する方法を実証します。

本項では、**Number Guess** の例を用いて、ルールの実行順序を制御するための **rule-flow** の使用方法について学びましょう。ルールのグループを実行する順序を明確に示すため、標準的なワークフロー図が使用されます。

### 例8.56 数字当てルールベースの作成

```
final KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.drl",
ShoppingExample.class ), ResourceType.DRL );

kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.rf",
ShoppingExample.class ), ResourceType.DRF );

final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
```

```
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

このパッケージを作成し、**add()** メソッドを用いてルールをロードするプロセスは、前述の例と同じですが、任意の手順が 1 つ追加されます。この追加手順では、1 つの **knowledge base** でさまざまな **rule-flow** を使用する機能を指定するため、現在の **rule-flow** に **NumberGuess.rf** 行を追加します (この手順を省略すると、**knowledge base** が前述の例と同じ方法で作成されます)。

#### 例8.57 ルールフローの開始

```
final StatefulKnowledgeSession ksession =
    kbase.newStatefulKnowledgeSession();
KnowledgeRuntimeLogger logger =
    KnowledgeRuntimeLoggerFactory.newFileLogger
        (ksession, "log/numberguess");

ksession.insert( new GameRules( 100, 5 ) );
ksession.insert( new RandomNumber() );
ksession.insert( new Game() );

ksession.startProcess( "Number Guess" );
ksession.fireAllRules();

logger.close();
ksession.dispose();
```

生成後、**knowledge base** を使用して「ステートフル」セッションを取得します。このセッションにファクト (標準的な Java オブジェクトのこと) を挿入します。簡単にするため、この例ではすべてのクラスが **NumberGuessExample.java** という 1 つのファイルに含まれています。

- **GameRules** クラスは最大範囲と回答可能な回数を提供します。
- **RandomNumber** クラスは、自動的にゼロから 100 までの数字を生成し、**getValue()** メソッドを用いて挿入した後にルールで使用できるようにします。
- **Game** クラスは以前回答した答えと、現在の回答数を追跡します。



#### 重要

標準の **fireAllRules()** メソッドを呼び出す前に、**startProcess()** メソッドを用いて以前ロードされたプロセスを開始します。





注記

実際は、オブジェクトをさらに最終状態で使用します (たとえば、この数字を最大スコアのテーブルに追加できるようにする回答数)。しかし、本チュートリアルでは、**working memory** セッションが **dispose()** メソッドへの呼び出しによって消去されるようにすれば十分です。

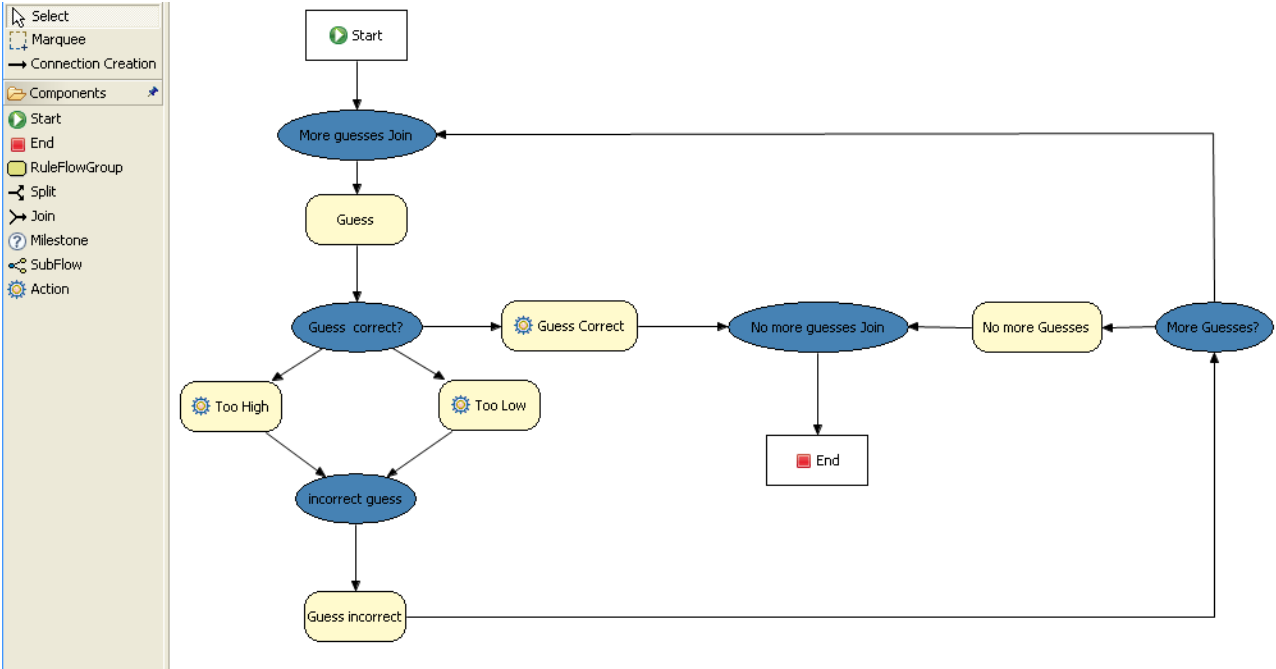


図8.20 数字当ての例のルールフロー

上図を表示するには、JBoss Rules IDE の `NumberGuess.rf` ファイルを開きます。外見上、アイコンは JBoss jBPM Workflow のアイコンと大変似ていて (同じではない)、このイメージは標準的なフローチャートと非常に似ています。

この図を編集するには、画面の左にあるコンポーネントのメニューを使用します。このエリアは **パレット** と呼ばれます。図は、X-Stream より、人言が判読できる XML 形式で保存されます。

**Properties** ビューが表示されていない場合は、**[Window] > [Show View] > [Other]** の順に選択し、**Properties** ビューをクリックして表示します。これは、**rule-flow** 上のアイテムを選択する **前** に行います (または **rule-flow** の空白スペースをクリックした後)。その後、次のプロパティーが使用できるようになります。

Property	Value
Id	Number Guess
Name	Number Guess
Router Layout	Shortest Path
Version	

図8.21 文字当てルールフローのプロパティー



## 注記

意味のある情報を得るため、本チュートリアルに残りの作業を行う間、**Properties** ビューを継続的に確認することが推奨されます。現時点では、**session.startprocess()** メソッドが最初の実行した時に開始された **rule-flow** プロセスの ID 番号が表示されています。

**Number Guess** の **rule-flow** のノード型は次の通りです。

- **start** (緑の矢印) および **end** ノード (赤いボックス) は **rule-flow** が開始および終了する場所を示します。
- **RuleFlowGroup** (シンプルな黄色のボックスによって示されます) は、本項の後半で説明する **DRL** ファイルの **RuleFlowGroups** へマップします。たとえば、フローが **Too High RuleFlowGroup** に達すると、補足的な **ruleflow-group Too High** 属性でマーク付けされたルールのみが実行を許可されます。
- **action nodes** (歯車のようなエンブレムを持つ黄色のボックスで示されます) は、標準的な Java メソッドの呼び出しを実行できます。この例では、ほとんどの **action nodes** が **System.out.println** を呼び出し、ユーザーに何が起きているか伝えます。
- **Guess Correct** や **More Guesses Join** などの **split** および **join nodes** (青の楕円) は、制御のフローを分割 (さまざまな条件による) または再結合できる場所を示します。
- **arrows** はさまざまなノード間のフローの方向を示します。

これらのノードは、ルールとともに動作し、文字当てゲームを形成します。たとえば、**Guess RuleFlowGroup** は、実行する **Get User Guess** ルールのみを許可します。これは、一致する **ruleflow-group "Guess"** の属性を持つ唯一のルールであるからです。

### 例8.58 ルールフローの特定場所でのみ実行されるルールの例

```
rule "Get user Guess"
  ruleflow-group "Guess"
  no-loop
  when
    $r : RandomNumber()
    rules : GameRules( allowed : allowedGuesses )
    game : Game( guessCount < allowed )
    not ( Guess() )
  then
    System.out.println( "You have " + ( rules.allowedGuesses -
                                game.guessCount )
                        + " out of " + rules.allowedGuesses
                        + " guesses left.\nPlease enter your guess
from 0 to "
                                + rules.maxRange );
    br = new BufferedReader( new InputStreamReader( System.in ) );
    i = br.readLine();
    modify ( game ) { guessCount = game.guessCount + 1 }
    insert( new Guess( i ) );
  end
```

このルールの残りの部分は比較的簡単です。**guessCount** が **allowedGuesses** の数よりも小さく (**GameRules** クラスより読み取り)、ユーザーが正しい数字を回答していない場合に、左側 (**when** 句) は **RandomNumber** オブジェクトが **working memory** に挿入されるたびにアクティベートされるよう指示します。

右側 (結果と呼ばれ、キーワード **then** が使用されます) はユーザーにメッセージを出力し、**System.in** よりユーザーの応答が到達するのを待ちます。この入力を受け取った後 (**return** キーが押されるまで **System.in** がブロックします)、回答数と実際の回答を更新または変更し、これらが **working memory** で使用可能になります。

**Rules** の残りの部分は比較的簡単です。ダイレクトが MVEL に設定され、さまざまな Java クラスがインポートされることをパッケージが宣言します。このファイルには合わせて 5 つのルールが存在します。

1. **Get User Guess** (上記で検証されたルール)
2. 最大の回答を記録するルール
3. 最小の回答を記録するルール
4. 回答を検査し、不正解の場合にメモリーから取り消すルール。
5. すべての回答が使用されたことをユーザーに通知するルール。

標準ルールと **rule-flow** との間を統合する 1 つのポイントが **ruleflow-group** 属性です。2 つ目のポイントは **DRL** ルールファイルと **rule-flow.rf** ファイルの間です。**split nodes** (青の楕円) はワーキングメモリーの値を使用して (ルールによって更新される) 実行するアクションのフローを決定します。この挙動を確認するには、**Guess Correct Node** をクリックします。**Properties** ビュー内より **Constraints Editor** を開きます (**Constraints** プロパティ行をクリックした後に表示される右側のボタンを押します)。以下が表示されます。

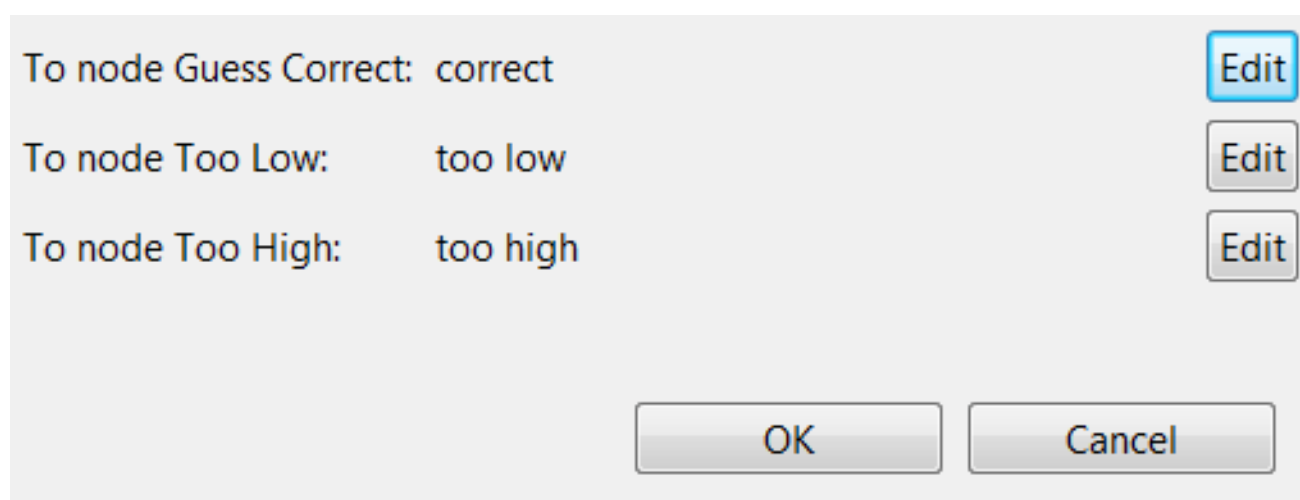


図8.22 GuessCorrect ノードの制約を編集

**Edit** をクリックすると、以下のダイアログボックスが表示されます。**Textual Editor** の値は標準の左側のルール形式に従い、**working memory** のオブジェクトを参照できます。**consequence** (右側にコード化) として、左側の式が **true** である場合に制御のフローがこのノードに従います。

Name:

Priority:

☐ Always true

Textual Editor

```
RandomNumber( randomValue : value ) &&
Guess( value > randomValue )
```

図8.23 GuessCorrect の Constraints Editor

`NumberGuess.java` の例には `main()` メソッドが含まれるため、標準の Java アプリケーション (**BASH** から、または **IDE** 経由) として実行できます。典型的数字当てゲームでは次の対話が行われます (数字はユーザーによって入力されます)。

**例8.59 数字当てプログラムが人間に勝った場合のコンソールの出力例**

```
You have 5 out of 5 guesses left.
Please enter your guess from 0 to 100
50
Your guess was too high
You have 4 out of 5 guesses left.
Please enter your guess from 0 to 100
25
Your guess was too low
You have 3 out of 5 guesses left.
Please enter your guess from 0 to 100
37
Your guess was too low
You have 2 out of 5 guesses left.
Please enter your guess from 0 to 100
44
Your guess was too low
You have 1 out of 5 guesses left.
Please enter your guess from 0 to 100
47
Your guess was too low
You have no more guesses
```

```
The correct guess was 48
```

重要なポイントは次のようになります。

1. **NumberGuessExample.java** の **Main()** メソッドは **RuleBase** をロードして **StatefulSession** を取得し、**Game**、**GameRules**、および **RandomNumber** (ターゲット番号が含まれる) オブジェクトを挿入します。このメソッドは使用されるプロセスフローを設定し、すべてのルールを実行します。そして、制御が **RuleFlow** へ移ります。
2. **NumberGuessExample.rf RuleFlow** が **Start** ノードで開始されます。
3. 制御が **Guess** ノードに移り、**More Guesses** 結合ノードより移行します。
4. **Guess** ノードで **Get User Guess RuleFlowGroup** が有効になります。この場合、**Guess** ルール (**NumberGuess.drl** ファイルに存在する) がトリガーされます。ユーザーにメッセージを表示し、応答を受け取って、メモリーに格納します。制御が次のノードである **Guess Correct** に移ります。
5. このノードでは、制約が現在のセッションを検証し、取るべきパスを決定します。

4. で回答した数字が大きすぎるまたは小さすぎる場合、フローは以下の両方を持つパスに移ります。

- 通常の Java コードを使用して下記のステートメントのいずれかを出力するアクションノード。

```
Too high
```

```
Too low
```

- **Rules** ファイル内より最大または最小の回答ルールがトリガーされるようにする **RuleFlowGroup**。

フローはこれらのノードから 6. で指定されたノードに移ります。

4. の回答が正しい場合、通常の Java コードを持つアクションコードが次のステートメントを出力します。

```
You guessed correctly.
```

ここには、結合ノードがあるため (ルールフロー終了の直前)、**no-more-guesses** パス (7.) が **RuleFlow** も終了します。

6. 制御が結合ノードを介して **RuleFlow** へ移り、**Guess Incorrect RuleFlowGroup** (**working memory** より回答を取り消すようルールをトリガーします) へ移動します。その後、**More Guesses** 決定ノードへ移動します。

7. **More Guesses** 決定ノード (ルールフローの右側) は制約を使用して (ルールが **working memory** に格納した値を確認して)、これ以上回答できるかどうかを決定します。回答可能な場合は 3. に戻ります。これ以上回答できない場合は、以下を表示するルールをトリガーする **RuleFlowGroup** を介して、制御がワークフローの最後に移動します。

You have no more guesses

8. 正しい数字が回答されるか、回答回数を使い果たすまで、3. から 7. のループが継続します。

## 8.9. MISS MANNERS とベンチマーキング

名前:	Miss Manners
メインクラス:	<b>org.drools.benchmark.manners.MannersBenchmark</b>
タイプ:	Java アプリケーション
Rules ファイル:	<b>manners.dr1</b>
目的:	Manners ベンチマークについて説明し、深さの競合解決について取り上げます。

### 8.9.1. はじめに

Miss Manners はパーティーを開催する予定で、ホストとして座席を適切に配置したいと思っています。当初の計画では、ゲストを男女のペアで配置する予定でしたが、共通する話題がない可能性を心配しています。ホストとしてどうすべきでしょうか。Miss Manners は各ゲストの趣味に注目することにしました。性別が交互になるよう全員の座席を配置し、同じ趣味を持つ人が最低でも 1 名隣に座るようにします。

#### 8.9.1.1. ベンチマーキングのスクリプト

- *Miss Manners* は 深さ優先 探索を使用して座席の配置を決定します。男性と女性を交互に配置し、共通する趣味を持つ人が隣同士になるようにします。
- *Waltz* は線画を 3 次元にします。3 次元にするため、線をラベル付けし、*制約伝播* を使用します。
- *WaltzDB* は *Waltz* の汎用バージョンで、4 本以上の線による接合点をサポートし、データベースを使用します。
- *自動ルートプランナー (ARP)* は、ロボット車両向けに設計されたルートプランナーです。**A\*** 探索アルゴリズムを使用します。
- *Weavera* は、チャンネルとボックス用の *VLSI* (Very Large Scale Integration) ルーターです。ブラックボード技術を使用します。



## 注記

**Miss Manners** は、ルールエンジン業界の実質上の業界標準となるベンチマーキングテストです。この挙動は現在では広く知られるようになり、このテストを効率的に実行するよう多くのエンジンが最適化されたため、有用性は低くなっています。そのため、**Waltz** の人気が以前よりも高くなっています。

### 8.9.1.2. Miss Manners の実行フロー

最初の座席が割り当てられた後、システムは深さ優先再帰コードを実行します。これにより、最後の座席が割り当てられるまで、正しい座席の配置が繰り返し処理されます。**Miss Manners** は **context** インスタンスを使用し実行フローを制御します。次のアクティビティ図は、ルール実行と **context** の現状との関係を示すために分割されています。

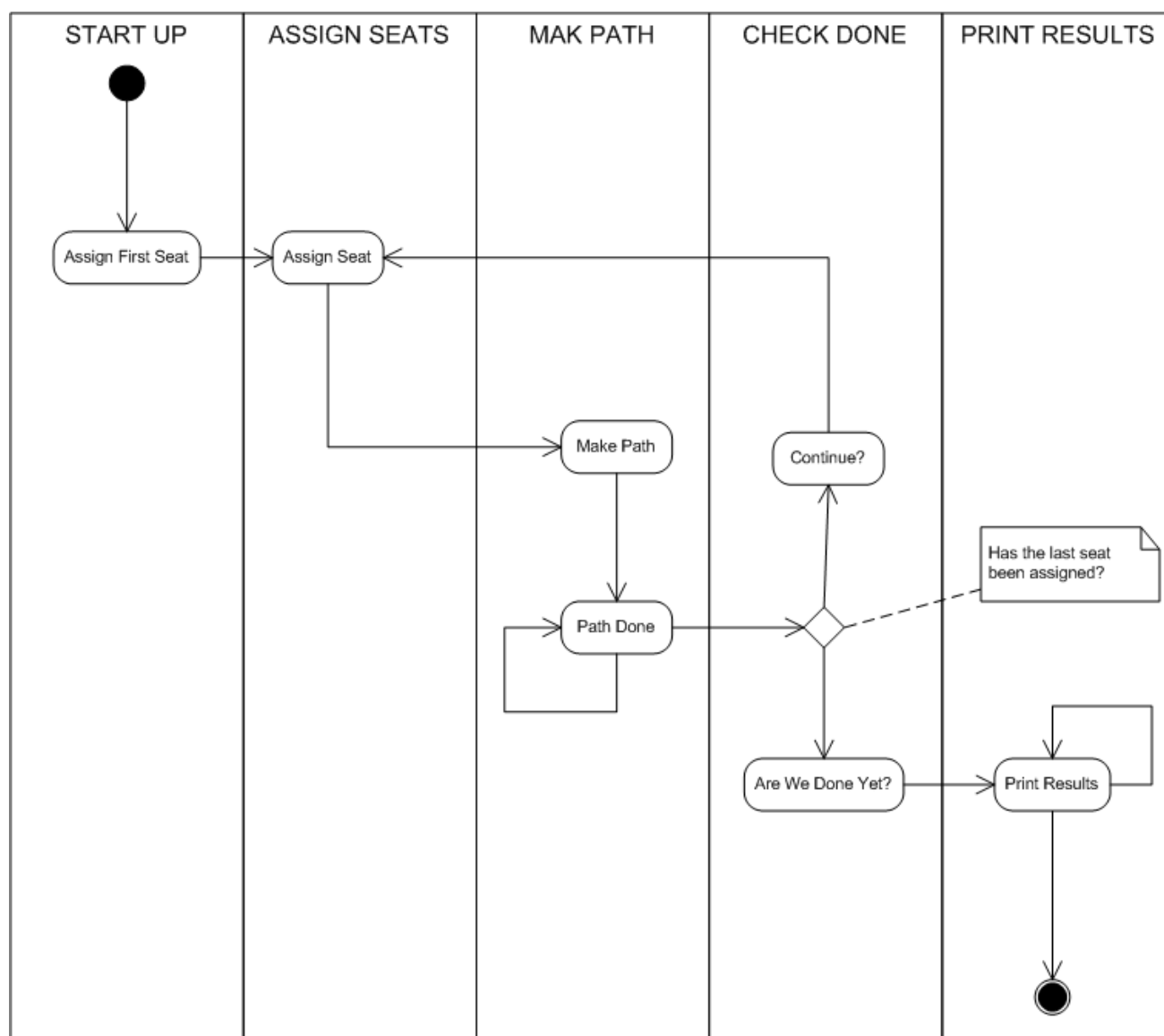


図8.24 Miss Manners のアクティビティ図

### 8.9.1.3. データと結果

ルールの詳細を説明する前に、アサートされたデータと結果となる座席配置を確認してください。データは、性別が交互になるようにし、隣に共通の趣味を持つ人が座るよう配置された5人のゲストの簡単なセットです。

### 8.9.1.4. データ

データは OPS5 (*Official Production System 5*) 構文で提供されます。各属性は、名前と値のペアの括弧で囲まれたリストを持ちます。趣味は 1 人 1 つのみとなります。

```
(guest (name n1) (sex m) (hobby h1) )
(guest (name n2) (sex f) (hobby h1) )
(guest (name n2) (sex f) (hobby h3) )
(guest (name n3) (sex m) (hobby h3) )
(guest (name n4) (sex m) (hobby h1) )
(guest (name n4) (sex f) (hobby h2) )
(guest (name n4) (sex f) (hobby h3) )
(guest (name n5) (sex f) (hobby h2) )
(guest (name n5) (sex f) (hobby h1) )
(last_seat (seat 5) )
```

### 8.9.1.5. 結果

結果リストの各行は、**Assign Seat** ルールが実行されると出力されます。各行に前の行よりも 1 大きい *pid* 値があることに注意してください (この重要性については、本項の後半の **Assign Seat** ルールの説明で取り上げます)。ls、rs、ln、および rn の略語は、左および右側の座席とこれらの座席に割り当てられたゲストの名前を表します。実際の実装では、長い属性名 (**leftGuestName** など) を使用しますが、本ガイドでは元の実装で使用された表記法が保持されます。

```
[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[Seating id=4, pid=3, done=false, ls=3, rn=n3, rs=4, rn=n2]
[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
```

## 8.9.2. 深い分析

### 8.9.2.1. 不正行為

**Miss Manners** は、クロス積の結合と **Agenda** のアクティビティを実行するよう設計されています。これを理解していないユーザーが、パフォーマンスを改善するために例を「微調整」してしまうことがあります。これにより **Manners** ベンチマークのポートは意味がなくなります。**Miss Manners** における既知の不正行為とポートエラーのリストは次の通りです。

- ゲストの趣味を 1 つずつ 1 つのファクトとしてアサートせずに、アレイを使用。これにより、**cross products** が大幅に減少します。
- データシーケンスの変更。これにより、一致する量が減少し、実行速度が向上します。
- テストアルゴリズムが **first-best-match** コードのみを使用するよう **not** 条件要素を変更。これは、基本的に 後向き連鎖 を実行するようテストテストアルゴリズムを変換します。このエリアの結果は、他の後向き連鎖ルールエンジンまたは **Miss Manners** のポートのみに相当します。
- **rule engine** がゲストと座席を時期尚早に一致するよう、**context** を削除。適切なポートは、**context start** によりファクトが一致しないようにします。
- **rule engine** が 組み合わせ パターンマッチングを実行しないようにする。



- **NOT CE** の結果として *推論サイクル* で取り消されるファクトがない場合、ポートが不完全。

### 8.9.2.2. 最初の座席の割り当て

コンテキストが **START\_UP** に変わった後、アサートされたゲストごとに **activations** が作成されます。各 **activation** は単一の **working memory** アクションの結果として作成されるため、すべて同じ **activation time** タグを持ちます (アサートされる最後のゲストは、さらに大きな **fact time** タグを持ちます)。



#### 注記

実行順序は、このルールにはほとんど影響しませんが、**Assign Seat** ルールには大きく影響します。

アクティベーションが実行され、最初の座席の配置と **path** がアサートされます。次に、**findSeating** ルールのアクティベーションを作成するため、Context 属性のステートが設定されます。

```
rule assignFirstSeat
when
  context : Context( state == Context.START_UP )
  guest : Guest()
  count : Count()
then
  String guestName = guest.getName();

  Seating seating = new Seating( count.getValue(),
    1,
    true,
    1,
    guestName,
    1,
    guestName);
  insert( seating );

  Path path = new Path( count.getValue(), 1, guestName );
  insert( path );

  modify( count ) { setValue ( count.getValue() + 1 ) }

  System.out.println( "assign first seat : "+seating+" : "+path);

  modify( context ) { setState( Context.ASSIGN_SEATS ) }
end
```

### 8.9.2.3. "findSeating" ルール

**findSeating** ルールは座席の配置を決定します。実行されると、アサートされた各ゲストに対して、アサートされた各座席配置の **cross-product** ソリューションを生成します (ルール自体とすでに割り当てられた選択済みソリューションを除く)。

```
rule findSeating
when
```

```

context : Context( state == Context.ASSIGN_SEATS )
$s      : Seating( pathDone == true )
$g1     : Guest( name == $s.rightGuestName )
$g2     : Guest( sex != $g1.sex, hobby == $g1.hobby )
count   : Count()
not ( Path( id == $s.id, guestName == $g2.name ) )
not ( Chosen( id == $s.id, guestName == $g2.name, hobby == $g1.hobby ) )
then
  int rightSeat = $s.getRightSeat();
  int seatId = $s.getId();
  int countValue = count.getValue();

  Seating seating = new Seating( countValue,
    seatId,
    false,
    rightSeat,
    $s.getRightGuestName(),
    rightSeat + 1,
    $g2.getName()
  );
  insert( seating );

  Path path = new Path( countValue, rightSeat + 1, $g2.getName() );
  insert( path );

  Chosen chosen = new Chosen( seatId, $g2.getName(), $g1.getHobby() );
  insert( chosen );

  System.err.println( "find seating : "+seating : "+path : "+chosen);

  modify( count ) {setValue( countValue + 1 )}
  modify( context ) {setState( Context.MAKE_PATH )}
end

```

```

=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3]

```

```

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

```

```

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

```



## 注記

冗長な **activations** を作成することは意味がないように見えますが、**Miss Manners** の作成では優良なルール設計を目的としていないことを思い出してください。これは、**cross-product matching** 処理と **Agenda** 機能に対して完全にストレステストを行うために、意図的にルールセットが下手に記述されています。



## 注記

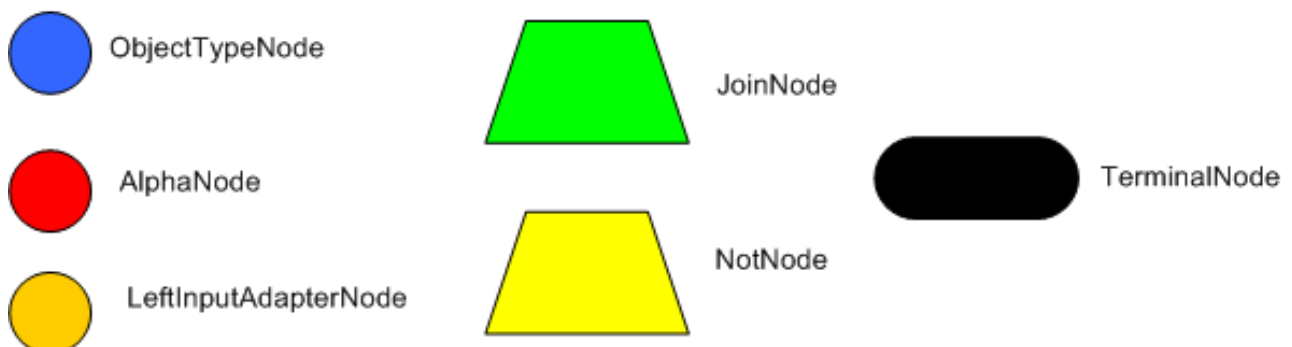
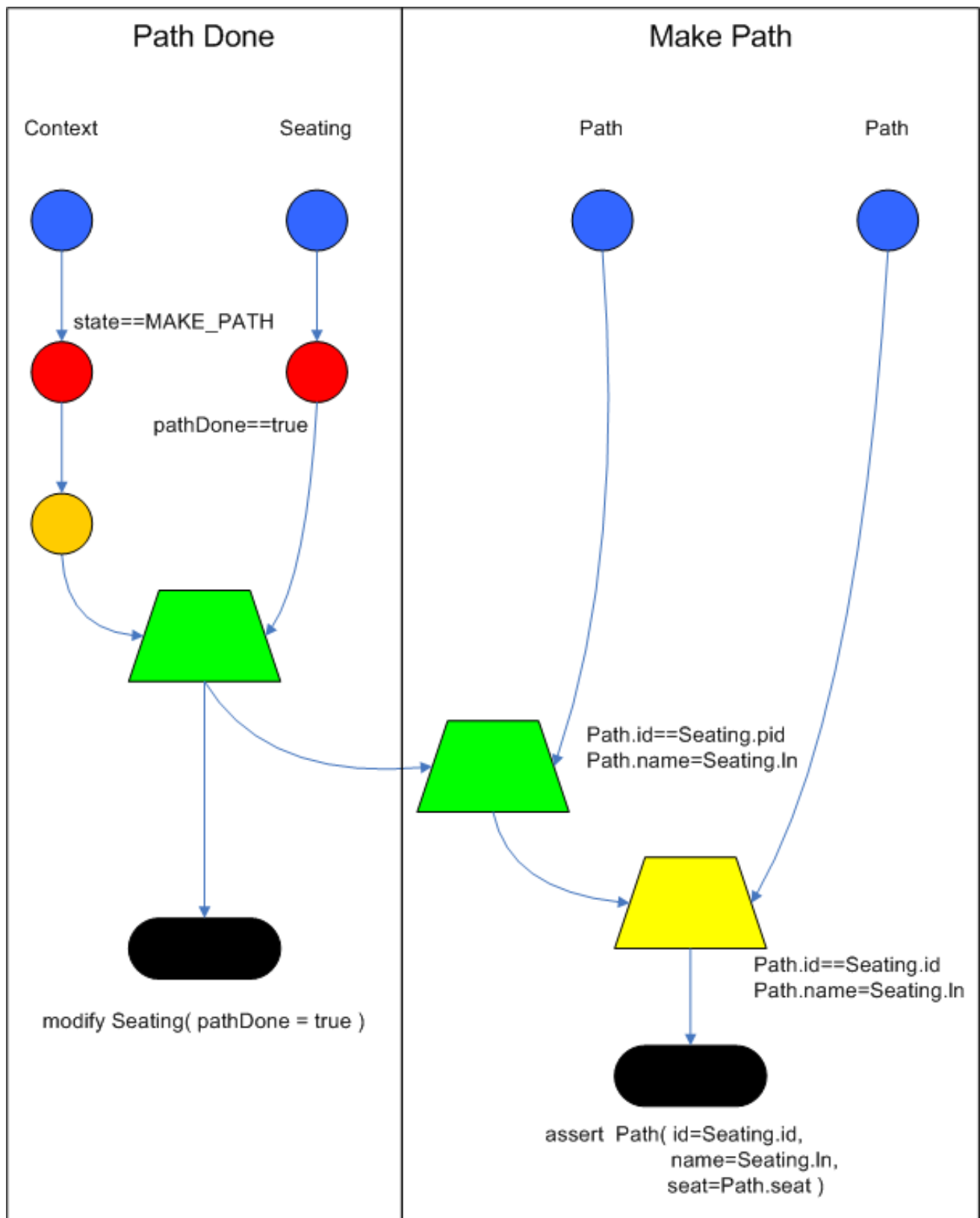
各アクティベーションの **time** タグ値は同じ 35 であることに注意してください。これは、**ASSIGN\_SEATS** へ **Context** オブジェクトが変更になり、すべてアクティベートされたためです。OPS5 および LEX では、最後にアサートされた座席のアクティベーションが適切に実行されます。**Depth** では、**accumulated fact time** により、最後にアサートされる座席のアクティベーションが確実に実行されます。

### 8.9.2.4. "makePath" および "pathDone" ルール

**makePath** ルールは必ず **pathDone** の前に実行する必要があります。**path** オブジェクトは各座席の配置に対して最後までアサートされます (**pathDone** の条件は **makePath** のサブセットであることに注意してください。そのため、どのように **makePath** を最初に行わせるのか疑問に思う人がいるかもしれません)。

```
rule makePath
when
  Context( state == Context.MAKE_PATH )
  Seating( seatingId:id, seatingPid:pid, pathDone == false )
  Path( id == seatingPid, pathGuestName:guestName, pathSeat:seat )
  not Path( id == seatingId, guestName == pathGuestName )
then
  insert( new Path( seatingId, pathSeat, pathGuestName ) );
end
```

```
rule pathDone
when
  context : Context( state == Context.MAKE_PATH )
  seating : Seating( pathDone == false )
then
  modify( seating ) {setPathDone( true )}
  modify( context ) {setState( Context.CHECK_DONE)}
end
```



## 図8.25 Rete 図

最終的に、ルールは両方 **Agenda** と競合します。両方のルールは同一の **activation time** タグを持ちますが、**makePath** ルールの **accumulate fact time** タグの方が大きいため、**makePath** ルールが優先されます。

### 8.9.2.5. "Continue" および "Are We Done?" ルール

**Are We Done** は、最後の座席が割り当てられた時のみアクティベートします。この時点で、両方のルールが実行されます。**makePath** は常に **pathDone** よりも優先されますが、同じ理由で **Are We Done** は **Continue** よりも優先されます。

```
rule areWeDone
when
  context : Context( state == Context.CHECK_DONE )
  LastSeat( lastSeat: seat )
  Seating( rightSeat == lastSeat )
then
  modify( context ) {setState(Context.PRINT_RESULTS )}
end
```

```
rule continue
when
  context : Context( state == Context.CHECK_DONE )
then
  context.setState( Context.ASSIGN_SEATS );
  update( context );
end
```

### 8.9.3. 出力の概要

Assign First Seat

```
=>[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
=>[fid:14:14]:[Path id=1, seat=1, guest=n5]
```

```
==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]
```

```
==>[ActivationCreated(16): rule=findSeating
[fid:13:13]:[Seating id=1 , pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]*
```

Assign Seating

```
=>[fid:15:17] :[Seating id=2 , pid=1 , done=false, ls=1, lg=n5, rs=2,
rn=n4]
=>[fid:16:18]:[Path id=2, seat=2, guest=n4]
=>[fid:17:19]:[Chosen id=1, name=n4, hobbies=h1]
```

```
=>[ActivationCreated(21): rule=makePath
```

```
[fid:15:17] : [Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]
[fid:14:14] : [Path id=1, seat=1, guest=n5]*
```

```
==>[ActivationCreated(21): rule=pathDone
[Seating id=2, pid=1, done=false, ls=1, ln=n5, rs=2, rn=n4]*
```

Make Path

```
=>[fid:18:22:[Path id=2, seat=1, guest=n5]]
```

Path Done

Continue Process

```
=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:7:7]:[Guest name=n4, sex=f, hobbies=h3]
[fid:4:4] : [Guest name=n3, sex=m, hobbies=h3]*
```

```
=>[ActivationCreated(25): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1], [fid:12:20] : [Count
value=3]
```

```
=>[ActivationCreated(25): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]
```

Assign Seating

```
=>[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]]
=>[fid:20:27]:[Path id=3, seat=3, guest=n3]]
=>[fid:21:28]:[Chosen id=2, name=n3, hobbies=h3]]
```

```
=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:18:22]:[Path id=2, seat=1, guest=n5]*
```

```
=>[ActivationCreated(30): rule=makePath
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]
[fid:16:18]:[Path id=2, seat=2, guest=n4]*
```

```
=>[ActivationCreated(30): rule=done
[fid:19:26]:[Seating id=3, pid=2, done=false, ls=2, ln=n4, rs=3, rn=n3]*
```

Make Path

```
=>[fid:22:31]:[Path id=3, seat=1, guest=n5]
```

Make Path

```
=>[fid:23:32] [Path id=3, seat=2, guest=n4]
```

Path Done

Continue Processing

```
=>[ActivationCreated(35): rule=findSeating
[fid:19:33]:[Seating id=3, pid=2, done=true, ls=2, ln=n4, rs=3, rn=n3]
[fid:4:4]:[Guest name=n3, sex=m, hobbies=h3]
[fid:3:3]:[Guest name=n2, sex=f, hobbies=h3], [fid:12:29]*

=>[ActivationCreated(35): rule=findSeating
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(35): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1], [fid:1:1] : [Guest name=n1,
sex=m, hobbies=h1]
```

Assign Seating

```
=>[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]]
=>[fid:25:37]:[Path id=4, seat=4, guest=n2]]
=>[fid:26:38]:[Chosen id=3, name=n2, hobbies=h3]

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:23:32]:[Path id=3, seat=2, guest=n4]*

==>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:20:27]:[Path id=3, seat=3, guest=n3]*

=>[ActivationCreated(40): rule=makePath
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]
[fid:22:31]:[Path id=3, seat=1, guest=n5]*

=>[ActivationCreated(40): rule=done
[fid:24:36]:[Seating id=4, pid=3, done=false, ls=3, ln=n3, rs=4, rn=n2]*
```

Make Path

```
=>fid:27:41:[Path id=4, seat=2, guest=n4]
```

Make Path

```
=>fid:28:42:[Path id=4, seat=1, guest=n5]]
```

Make Path

```
=>fid:29:43:[Path id=4, seat=3, guest=n3]]
```

Path Done

Continue Processing

```
=>[ActivationCreated(46): rule=findSeating
```

```
[fid:15:23]:[Seating id=2, pid=1, done=true, ls=1, ln=n5, rs=2, rn=n4]
[fid:5:5]:[Guest name=n4, sex=m, hobbies=h1], [fid:2:2]
[Guest name=n2, sex=f, hobbies=h1]

=>[ActivationCreated(46): rule=findSeating
[fid:24:44]:[Seating id=4, pid=3, done=true, ls=3, ln=n3, rs=4, rn=n2]
[fid:2:2]:[Guest name=n2, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]*

=>[ActivationCreated(46): rule=findSeating
[fid:13:13]:[Seating id=1, pid=0, done=true, ls=1, ln=n5, rs=1, rn=n5]
[fid:9:9]:[Guest name=n5, sex=f, hobbies=h1]
[fid:1:1]:[Guest name=n1, sex=m, hobbies=h1]

Assign Seating

=>[fid:30:47]:[Seating id=5, pid=4, done=false, ls=4, ln=n2, rs=5, rn=n1]
=>[fid:31:48]:[Path id=5, seat=5, guest=n1]
=>[fid:32:49]:[Chosen id=4, name=n1, hobbies=h1]
```

本項の内容を学習し、**Miss Manners** ベンチマーキングスクリプトの仕組みや、注意すべき警告や落とし穴について理解できたと思います。

## 8.10. ライフゲーム (CONWAY'S GAME OF LIFE) の例

名前:	Conway's Game Of Life
メインクラス:	<b>org.drools.examples.conway.ConwayAgendaGroupRun</b> <b>org.drools.examples.conway.ConwayRuleFlowGroupRun</b>
タイプ:	Java アプリケーション
Rules ファイル:	<b>conway-ruleflow.drl</b> <b>conway-agendagroup.drl</b>
目的:	accumulate、collect、および from を実証します。

**ライフゲーム** (Conway's Game Of Life) はよく知られているシミュレーションモデルです。ここで使用するアプリケーションは **Swing** ベースの実装になります。この **ゲーム** を規定するルールは、**JBoss Rules** を使用して実装されます。ここでは、この実装の仕組みについて学びましょう。

最初に、以下のグリッドを見てください。ライフのシミュレーションが行われる「領域」を表すことで、ゲームをイメージしやすくなります。最初、グリッドは空であるため、システムには生きているセルは存在しません。各セルは「生」と「死」のいずれかとなり、生きているセルは緑色の丸で表されます。生きているセルの事前選択パターンは、**Pattern** ドロップダウンリストより選択できます (また、個々のセルをダブルクリックすると、「生」と「死」の状態を切り替えることができます)。

各セルは隣接するセルと関係していることを理解するのが重要になります。これは **ゲーム** のルールの基本となります。「隣接するセル」には、上下左右のセルだけでなく、対角線上に隣接するセルも含まれます。そのため、1 つセルに隣接するセルは 8 つあります。例外は、隣接するセルが 3 つしかない四隅のセルと、隣接するセルが 5 つのみの 4 辺上のセルになります。



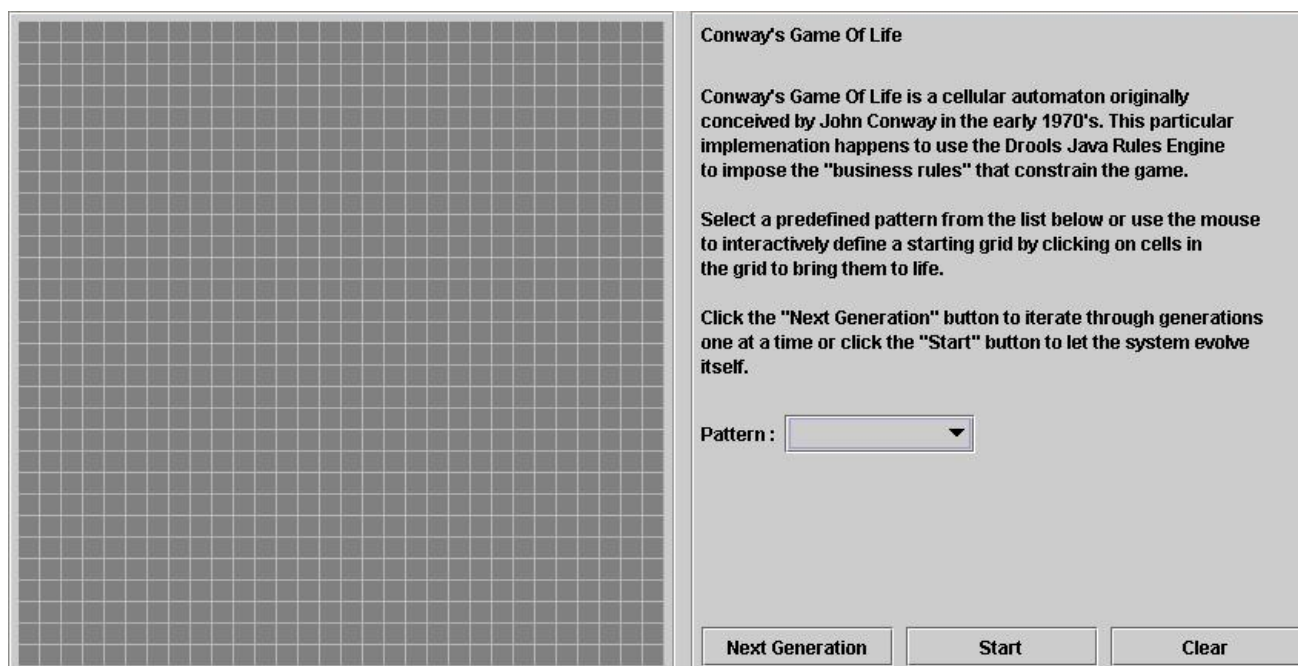


図8.26 新しいゲームの開始

各世代ごとに (セルすべての完全な繰り返しと評価)、システムは進化し、セルが誕生したり死滅したりします。次世代がどのようなようになるかを規定する簡単なルールは次の通りです。

- 生きているセルに隣接する生きたセルが 1 つ以下の場合、孤独により死滅します。
- 生きているセルに隣接する生きたセルが 4 つ以上の場合、過密により死滅します。
- 死んでいるセルに隣接する生きたセルがちょうど 3 つある場合、死んでいるセルが生き返ります。

これら基準のいずれかを満たさないセルは、次の繰り返しまでそのままの状態になります。これらの簡単なルールを考慮し、しばらくシステムでゲームを試してみてください。1 つずつ繰り返しを行い、ルールの適用を観察します。

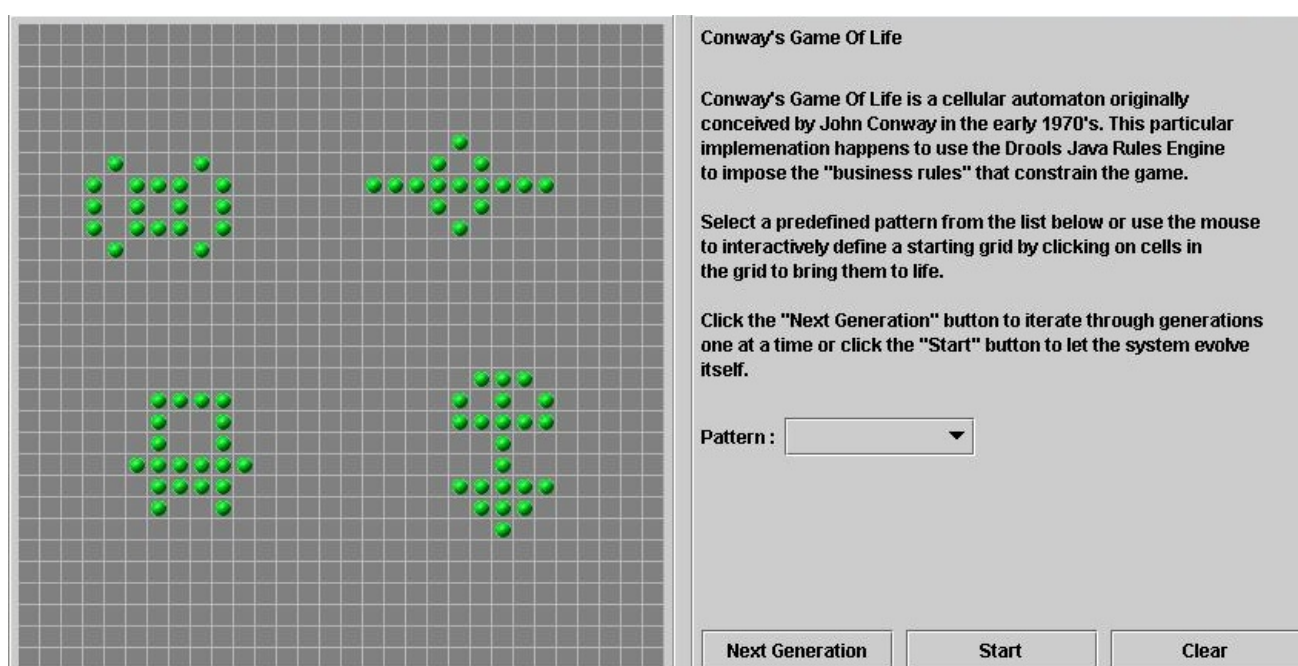


図8.27 進行中のゲーム

ここで、コードについて学びましょう (これは高度な例であるため、JBoss Rules フレームワークを理解していることが前提になります)。この例は、実行フローを管理する、**AgendaGroups (ConwayAgendaGroupRun)** と **RuleFlowGroups (ConwayRuleFlowGroupRun)** による 2 つの方法を表しています。これら方法を比較することは大変有益です。本章では、読者のほとんどが使用するルールフローバージョンについて説明します。

セルはすべて **session** に挿入されます。ルールフロープロセスは、**register neighbour** と呼ばれるグループのルールに実行パーミッションを与えます。ルールグループは **Neighbour Relation** クラスを使用して、北東、北、北西、西に隣接するセルを登録します。この関係は両方向であるため、南方向のセルにルールを作成する必要がないことに注意してください。また制約により、セルが最後から 1 つ前の列と一番上から 1 つ前の行に置かれることにも注意してください。アクティベーションがすべて実行されるまでに、すべてのセルが隣接するすべてのセルに関係します。

#### 例8.60 隣接するセルの関係をすべて登録

```

    rule "register north east"
    ruleflow-group "register neighbour"
when
    CellGrid( $numberOfColumns : numberOfColumns )
    $cell: Cell( $row : row > 0, $col : col <=
        ( $numberOfColumns - 1 ) )
    $northEast : Cell( row == ($row - 1), col == ( $col + 1 ) )
then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
    ruleflow-group "register neighbour"
when
    $cell: Cell( $row : row > 0, $col : col )
    $north : Cell( row == ($row - 1), col == $col )
then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

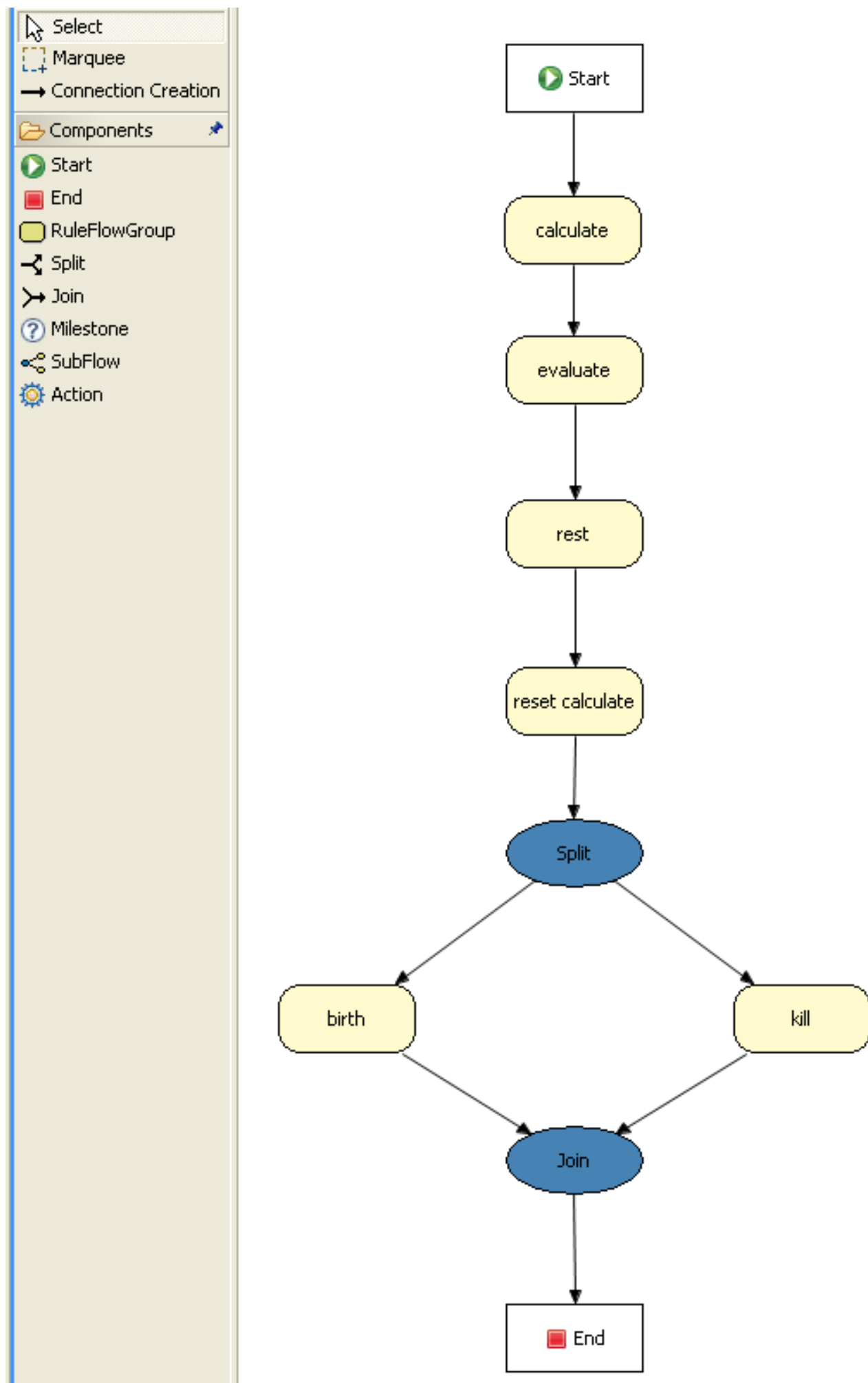
rule "register north west"
    ruleflow-group "register neighbour"
when
    $cell: Cell( $row : row > 0, $col : col > 0 )
    $northWest : Cell( row == ($row - 1), col == ( $col - 1 ) )
then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
    ruleflow-group "register neighbour"
when
    $cell: Cell( $row : row >= 0, $col : col > 0 )
    $west : Cell( row == $row, col == ( $col - 1 ) )
then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end

```



すべてのセルが挿入されると、Java コードが実行されます。さらに、パターンがグリッドに適用され、特定のセルが「生」の状態に設定されます。**[Start]** または **[Next Generation]** をクリックすると、**Generation** ルールフローが実行されます。このルールフローは繰り返しサイクルごとにセルの変更をすべて管理します。



## 図8.28 生成ルールフロー

ルールフロープロセスは最初にルールの **evaluate** グループより実行されます。その結果、このグループのアクティブなルールはすべて実行可能になります (このグループのルールは、本項の最初に説明したメイン **Game** ルールを適用します。これらのルールは、死滅するセルと生き返るセルを決定します)。

phase 属性は、各セルがルールの特定グループへ応答する方法を指示します。通常、この属性はルールフロープロセスで定義された **RuleFlowGroup** の 1 つと関係します。

この属性は、現時点では実際にセルの状態を変更しません。これは、グリッドをすべて完全に評価してから編集内容を適用する必要があるからです。セルは一時的に Phase.KILL または Phase.BIRTH フェーズのいずれかに設定されます。これらのフェーズは後で変更を適用する時に使用されます。

## 例8.61 ステート変更でセルを評価

```

    rule "Kill The Lonely"
    ruleflow-group "evaluate"
    no-loop
when
#   A live cell has fewer than 2 live neighbors
theCell: Cell(liveNeighbors < 2, cellState ==
CellState.LIVE, phase == Phase.EVALUATE)then
theCell.setPhase(Phase.KILL);
update( theCell );
end

rule "Kill The Overcrowded"
    ruleflow-group "evaluate"
    no-loop
when
#   A live cell has more than 3 live neighbors
theCell: Cell(liveNeighbors > 3, cellState ==
CellState.LIVE, phase == Phase.EVALUATE)then
theCell.setPhase(Phase.KILL);
update( theCell );
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
when
#   A dead cell has 3 live neighbors
theCell: Cell(liveNeighbors == 3, cellState ==
CellState.DEAD, phase == Phase.EVALUATE)then
theCell.setPhase(Phase.BIRTH);
update( theCell );
end

```

セルがすべて評価された後、**reset calculate** ルールは、以前のデータ変更によって発生した計算アクティベーションを **calculate** グループから消去します。次に、「split」が入力されます。これにより、「kill」および「birth」グループの両方にあるアクティベーションを実行できます (これらのルールは状態の変更を適用します)。

#### 例8.62 変更の適用

```

    rule "reset calculate"
    ruleflow-group "reset calculate"
    when
    then
        WorkingMemory wm = drools.getWorkingMemory();
        wm.clearRuleFlowGroup( "calculate" );
    end

    rule "kill"
    ruleflow-group "kill"
    no-loop
    when
        theCell: Cell( phase == Phase.KILL )
    then
        modify( theCell ){
            setCellState( CellState.DEAD ),
            setPhase( Phase.DONE );
        }
    end

    rule "birth"
    ruleflow-group "birth"
    no-loop
    when
        theCell: Cell( phase == Phase.BIRTH )
    then
        modify( theCell ){
            setCellState( CellState.LIVE ),
            setPhase( Phase.DONE );
        }
    end

```

この時点で、複数のセルが変更されています。これは、複数のセルの状態が「live」または「dead」に変更されたためです。次に、隣接するすべてのセルを繰り返し処理するため **neighbour relation** が使用されるため、生きている隣接するセルの数が増加または減少します。セルの数が変わったセルはすべて **evaluate** フェーズに設定され、ルールフロープロセスの次の段階で評価されます。

手作業で繰り返し処理を実行する必要がないことに注意してください。ルールに関係を適用するだけで、必要なコードが最小限であってもルールエンジンはすべての処理を行います。「live」セルの数が決定され、設定されると、ルールプロセスは終了します。ルールフロープロセスに他の世代を評価するよう指示できますが、**evaluate** がクリックされた場合はエンジンがルールフロープロセスを再度実行します。

## 例8.63 ステート変更があったセルの評価

```
rule "Calculate Live"
ruleflow-group "calculate"
lock-on-active
when
  theCell: Cell( cellState == CellState.LIVE )
  Neighbor( cell == theCell, $neighbor : neighbor )
then
  modify( $neighbor ){
    setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
    setPhase( Phase.EVALUATE );
  }
end

rule "Calculate Dead"
ruleflow-group "calculate"
lock-on-active
when
  theCell: Cell( cellState == CellState.DEAD )
  Neighbor( cell == theCell, $neighbor : neighbor )
then
  modify( $neighbor ){
    setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
    setPhase( Phase.EVALUATE );
  }
end
```

## 付録A © 2011

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.



## 付録B 改訂履歴

<b>改訂 5.2.0-0.1.400</b> Rebuild with publican 4.0.0	<b>2013-10-31</b>	<b>Rüdiger Landmann</b>
<b>改訂 5.2.0-0.1</b> Translation files synchronised with XML sources 5.2.0-0	<b>Mon Jun 10 2013</b>	<b>Junko Ito</b>
<b>改訂 5.2.0-0.1</b> Translation files synchronised with XML sources 5.2.0-0	<b>Thu May 9 2013</b>	<b>Junko Ito</b>
<b>改訂 5.2.0-0</b> 5.2.0 向けに更新	<b>Wed Dec 15 2010</b>	<b>L Carlon</b>
<b>改訂 5.1.0-0</b> 5.1.0 向けに更新	<b>Wed Dec 15 2010</b>	<b>David Le Sage</b>
<b>改訂 5.0.2-0</b> BRMS 341 - 古い情報を削除。3.3.8 および 3.3.11。	<b>Tue Jun 29 2010</b>	<b>David Le Sage</b>
<b>改訂 5.0.2-0</b> 5.0.2 向けに更新	<b>Wed May 5 2010</b>	<b>Darrin Mison</b>
<b>改訂 5.0.1-0</b> 5.0.1 の更新。本書における文法訂正の第 1 段階。	<b>Tue Oct 6 2009</b>	<b>David Le Sage</b>
<b>改訂 5.0.0-0</b> 公開済み	<b>Mon May 18 2009</b>	<b>Darrin Mison</b>