



Red Hat Process Automation Manager 7.7

DRL ルールを使用したデシジョンサービスの作
成

Red Hat Process Automation Manager 7.7 DRL ルールを使用したデシジョンサービスの作成

Red Hat Customer Content Services
brms-docs@redhat.com

法律上の通知

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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, the Red Hat 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 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

本書は、Red Hat Process Automation Manager 7.7 で、DRL ルールを使用してデシジョンサービスを作成する方法を説明します。

目次

前書き	4
第1章 RED HAT PROCESS AUTOMATION MANAGER におけるデシジョン作成アセット	5
第2章 DRL (DROOLS RULE LANGUAGE) ルール言語	9
2.1. DRL のパッケージ	10
2.2. DRL のインポートステートメント	10
2.3. DRL の機能	10
2.4. DRL のクエリー	11
2.5. DRL でのタイプ宣言とメタデータ	12
2.5.1. DRL のメタデータを使用しないタイプ宣言	12
2.5.2. DRL の列挙タイプの宣言	14
2.5.3. DRL の拡張タイプ宣言	14
2.5.4. DRL のメタデータが含まれるタイプ宣言	15
2.5.5. DRL でのファクトタイプと属性宣言のメタデータタグ	15
2.5.6. ファクトタイプに対するプロパティ変更の設定およびリスナー	21
2.5.7. アプリケーションコード内の DRL で宣言されたタイプへのアクセス	23
2.6. DRL のグローバル変数	24
2.7. DRL でのルール属性	25
2.7.1. DRL でのタイマーおよびカレンダールール属性	28
2.8. DRL のルール条件 (WHEN)	31
2.8.1. パターンと制約	32
2.8.2. パターンと制約でバインドされた変数	36
2.8.3. ネストされた制約とインラインキャスト	37
2.8.4. 制約内の日付リテラル	38
2.8.5. DRL のパターン制約でサポートされている演算子	39
2.8.6. DRL のパターン制約における演算子の優先順位	43
2.8.7. DRL でサポートされるルール条件要素 (キーワード)	44
2.8.8. DRL ルール条件内でオブジェクトのグラフが使用される OOPath 構文	54
2.9. DRL におけるルールアクション (THEN)	57
2.9.1. DRL でサポートされるルールアクションメソッド	57
2.9.2. drools および kcontext 変数のその他のルールアクションメソッド	59
2.9.3. 条件付きおよび名前付きの結果を伴う高度なルールアクション	61
2.10. DRL ファイルのコメント	63
2.11. DRL トラブルシューティングのエラーメッセージ	63
2.12. DRL ルールセットのルールユニット	67
2.12.1. ルールユニットのデータソース	71
2.12.2. ルールユニットの実行制御	72
2.12.3. ルールユニットのアイデンティティの競合	76
第3章 データオブジェクト	80
3.1. データオブジェクトの作成	80
第4章 BUSINESS CENTRAL における DRL ルールの作成	82
4.1. DRL ルールへの WHEN 条件の追加	86
4.2. DRL ルールへの THEN アクションの追加	90
第5章 ルールの実行	92
第6章 その他の DRL ルールの作成および実行方法	98
6.1. RED HAT CODEREADY STUDIO での DRL ルールの作成および実行	98
6.2. JAVA を使用した DRL ルールの作成および実行	102
6.3. MAVEN を使用した DRL ルールの作成および実行	105

第7章 RED HAT PROCESS AUTOMATION MANAGER の IDE 向けのデシジョン例	111
7.1. IDE での RED HAT PROCESS AUTOMATION MANAGER のデシジョン例のインポートと実行	111
7.2. HELLO WORLD のデシジョン例 (基本ルールおよびデバッグ)	114
7.3. 状態のデシジョン例 (前向き連鎖および競合解決)	117
顕著性を使用した状態の例	120
アジェンダグループを使用した状態の例	123
状態の例に含まれる動的なファクト	124
7.4. フィボナッチのデシジョン例 (再帰および競合解決)	125
7.5. 価格設定のデシジョン例 (デシジョンテーブル)	131
スプレッドシートのデシジョンテーブルの設定	132
基本価格のルール	135
プロモーション割引ルール	136
7.6. ペットショップのデシジョン例 (アジェンダグループ、グローバル変数、コールバック、GUI 統合)	136
ペットショップの例でのルール実行動作	137
ペットショップのルールファイルのインポート、グローバル変数、Java 関数	139
アジェンダグループを使用したペットショップルール	140
ペットショップ例の実行	144
7.7. 誠実な政治家のデシジョン例 (真理維持および顕著性)	148
Politician および Hope クラス	149
政治家の誠実性に関するルール定義	150
実行と監査証跡	151
7.8. 数独のデシジョン例 (複雑なパターン一致、コールバック、GUI 統合)	154
数独例の実行および対話	154
数独例のクラス	160
数独の検証ルール (validate.drl)	160
数独の解決ルール (sudoku.drl)	161
7.9. CONWAY の GAME OF LIFE のデシジョン例 (ルールフローグループおよび GUI 統合)	168
Conway 例の実行および対話	169
ルールグループを使用する Conway 例のルール	170
7.10. HOUSE OF DOOM のデシジョン例 (後向き連鎖および再帰)	174
再帰クエリーおよび関連のルール	178
推移閉包ルール	179
リアクティブクエリールール	180
ルールにバインドなしの引数が含まれたクエリー	181
第8章 DRL 使用時のパフォーマンスチューニングに関する考慮点	183
第9章 次のステップ	185
付録A バージョン情報	186

前書き

ビジネスルール開発者は、Business Central で DRL (Drools Rule Language) デザイナーを使用してビジネスルールを定義できます。DRL ルールは、Business Central におけるその他のルールアセットとは異なり、ガイド付きまたは表形式ではなく、フリーフォームの **.drl** テキストファイルで直接定義できます。このような DRL ファイルは、プロジェクトのデシジョンサービスの中心となります。



注記

ルールベースやテーブルベースのアセットではなく、Decision Model and Notation (DMN) モデルを使用してデシジョンサービスを設計することもできます。Red Hat Process Automation Manager 7.7 の DMN サポートに関する詳細は、以下の資料を参照してください。

- 『[デシジョンサービスのスタートガイド](#)』 (DMN デシジョンサービスの例を使用したステップバイステップのチュートリアル)
- 『[DMN モデルを使用したデシジョンサービスの作成](#)』 (Red Hat Process Automation Manager の DMN サポートおよび機能の概要)

前提条件

- DRL ルールのチームおよびプロジェクトが Business Central に作成されていて、各アセットが、スペースに割り当てられたプロジェクトに関連付けられている。詳細は『[デシジョンサービスのスタートガイド](#)』を参照してください。

第1章 RED HAT PROCESS AUTOMATION MANAGER におけるデシジョン作成アセット

Red Hat Process Automation Manager は、デシジョンサービスにビジネスデシジョンを定義するのに使用可能なアセットを複数サポートします。デシジョン作成アセットはそれぞれ長所が異なるため、ゴールおよびニーズに合わせて、アセットを1つ、または複数を組み合わせて使用できます。

以下の表では、デシジョンサービスでデシジョンを定義する最適な方法を選択できるように、Red Hat Process Automation Manager プロジェクトでサポートされている主要なデシジョン作成アセットを紹介します。

表1.1 Red Hat Process Automation Manager でサポートされるデシジョン作成アセット

アセット	主な特徴	オーサリングツール	ドキュメンテーション
DMN (Decision Model and Notation) モデル	<ul style="list-style-type: none"> Object Management Group (OMG) が定義する標準記法に基づくデシジョンモデルである 1つまたは複数の意思決定要件グラフ (DRG: decision requirements graphs) を含むグラフィカルな意思決定要件ダイアグラム (DRD: decision requirements diagrams) を使用してビジネスの意思決定フローを追跡する DMN モデルが DMN 準拠プラットフォーム間で共有できるようにする XML スキーマを使用する DMN デシジョンテーブルおよび他の DMN ボックス式 (Boxed Expression) でデシジョンロジックを定義する Friendly Enough Expression Language (FEEL) をサポートする Business Process Model and Notation (BPMN) プロセスモデルと効率的に統合できる 包括性、具体性および安定性のあるデシジョンフローの作成に最適である 	Business Central または DMN 準拠のエディター	『 DMN モデルを使用したデシジョンサービスの作成 』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
ガイド付きデシジョンテーブル	<ul style="list-style-type: none"> ● Business Central の UI ベースのテーブルデザイナーで作成するルールのテーブル ● デシジョンテーブルにスプレッドシートで対応する代わりにウィザードで対応する ● 条件を満たした入力に、フィールドとオプションを指定する ● ルールテンプレートを作成するためのテンプレートキーと値をサポートする ● その他のアセットではサポートされていない、ヒットポリシー、リアルタイム検証などの追加機能をサポートする ● コンパイルエラーを最小限に抑えるため、制限されているテーブル形式でルールを作成するのに最適 	Business Central	『 ガイド付きデシジョンテーブルを使用したデシジョンサービスの作成 』
スプレッドシートのデシジョンテーブル	<ul style="list-style-type: none"> ● Business Central にアップロード可能な XLS または XLSX スプレッドシート形式のデシジョンテーブルである ● ルールテンプレートを作成するためのテンプレートキーと値をサポートする ● Business Central 外で管理しているデシジョンテーブルでルールを作成するのに最適 ● アップロード時に適切にルールをコンパイルするために厳密な構文要件がある 	スプレッドシートエディター	『 スプレッドシート形式のデシジョンテーブルを使用したデシジョンサービスの作成 』
ガイド付きルール	<ul style="list-style-type: none"> ● Business Central の UI ベースのルールデザイナーで作成する個別ルール ● 条件を満たした入力に、フィールドとオプションを指定する ● コンパイルエラーを最小限に抑えるため、制御されている形式で単独のルールを作成するのに最適 	Business Central	『 ガイド付きルールを使用したデシジョンサービスの作成 』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
ガイド付きルールテンプレート	<ul style="list-style-type: none"> ● Business Central の UI ベースのテンプレートデザイナーで作成する再利用可能なルール構造 ● 条件を満たした入力に、フィールドとオプションを指定する ● (このアセットの基本となる) ルールテンプレートを作成するためのテンプレートキーと値をサポートする ● ルール構造が同じで、定義したフィールド値が異なるルールを多数作成するのに最適 	Business Central	『 ガイド付きルールテンプレートを使用したデシジョンサービスの作成 』
DRL ルール	<ul style="list-style-type: none"> ● .drl テキストファイルに直接定義する個別ルール ● 最も柔軟性が高く、ルールと、ルール動作に関するその他の技術を定義できる ● スタンドアロン環境で作成し、Red Hat Process Automation Manager に統合可能 ● 詳細にわたる DRL オプションが必要なルールを作成するのに最適 ● ルールを適切にコンパイルするために厳密な構文要件がある 	Business Central または統合開発環境 (IDE)	『 DRL ルールを使用したデシジョンサービスの作成 』

アセット	主な特徴	オーサリングツール	ドキュメンテーション
予測モデルマークアップ言語 (PMML: Predictive Model Markup Language) モデル	<ul style="list-style-type: none"> ● Data Mining Group (DMG) が定義する標準記法に基づく予測データ分析モデルである ● PMML モデルを PMML 準拠プラットフォーム間で共有できるようにする XML スキーマを使用する ● 回帰、スコアカード、ツリー、マイニングなどのモデルタイプをサポートする ● スタンドアロンの Red Hat Process Automation Manager プロジェクトに追加したり、Business Central のプロジェクトにインポートしたりできる ● Red Hat Process Automation Manager のデシジョンサービスに予測データを統合するのに最適である 	PMML または XML エディター	『 PMML モデルでのデシジョンサービスの作成 』

第2章 DRL (DROOLS RULE LANGUAGE) ルール言語

DRL (Drools Rule Language) ルールは、**.drl** テキストファイルに直接定義するビジネスルールです。これらの DRL ファイルは、Business Central 内の他のすべてのルールアセットが最終的にレンダリングされるソースとなります。Business Central インターフェースで DRL ファイルを作成して管理するか、または Red Hat CodeReady Studio や別の統合開発環境 (IDE) を使用して Maven または Java プロジェクトの一部として外部で作成することができます。DRL ファイルには、最低でもルールの条件 (**when**) およびアクション (**then**) を定義するルールを1つ以上追加できます。Business Central の DRL デザイナーは、Java、DRL、および XML の構文を強調表示します。

DRL ファイルは、以下のコンポーネントで構成されます。

DRL ファイル内のコンポーネント

```
package

import

function // Optional

query // Optional

declare // Optional

global // Optional

rule "rule name"
  // Attributes
  when
    // Conditions
  then
    // Actions
end

rule "rule2 name"

...
```

以下の DRL ルールの例では、ローン申し込みのデシジョンサービスで年齢制限を指定します。

申込者の年齢制限に関するルールの例

```
rule "Underage"
  salience 15
  agenda-group "applicationGroup"
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
  end
```

DRL ファイルには、ルール、クエリー、関数が1つまたは複数含まれており、このファイルで、ルールやクエリーで割り当て、使用するインポート、グローバル、属性などのリソース宣言を定義できます。

DRL パッケージは、DRL ファイルの一番上に表示し、ルールは通常最後に表示されます。他の DRL コンポーネントはどのような順番でも構いません。

ルールごとに、ルールパッケージ内で一意の名前を指定する必要があります。パッケージ内の DRL ファイルで、同じルール名を複数回使用すると、ルールのコンパイルに失敗します。特にルール名にスペースを使用する場合など、ルール名には必ず二重引用符 (**rule "rule name"**) を使用して、コンパイルエラーが発生しないようにしてください。

DRL ルールに関連するデータオブジェクトはすべて、DRL ファイルと同じ Business Central のプロジェクトパッケージに置く必要があります。同じパッケージのアセットはデフォルトでインポートされます。その他のパッケージの既存アセットは、DRL ルールを使用してインポートできます。

2.1. DRL のパッケージ

パッケージは、データオブジェクト、DRL ファイル、デシジョンテーブル、他のアセットタイプなど、Red Hat Process Automation Manager に含まれる関連アセットをまとめたフォルダーです。また、パッケージは、各ルールグループに固有の namespace としても機能します。1つのルールベースには、複数のパッケージを含めることができます。通常、パッケージだけで自己完結できるように、パッケージのすべてのルールはパッケージ宣言と同じファイルに保存します。ただし、対象のルールで使用するのに、他のパッケージからオブジェクトをインポートできます。

以下は、ローン申請デシジョンサービスの DRL ファイルのパッケージ名と namespace の例です。

DRL ファイルのパッケージ定義例

```
package org.mortgages;
```

2.2. DRL のインポートステートメント

Java のインポートステートメントと同様に、DRL ファイルのインポートで、ルールで使用するオブジェクトの完全修飾パスとタイプ名を識別します。**packageName.objectName** の形式でパッケージとデータオブジェクトを指定し、複数のインポートを指定する場合には別の行に指定します。デシジョンエンジンは自動的に、DRL パッケージと同じ名前の Java パッケージおよび、**java.lang** パッケージからクラスを自動的にインポートします。

以下の例は、住宅ローン申請デシジョンテーブルに含まれるローン申請オブジェクトのインポートステートメントです。

DRL ファイルのインポートステートメント例

```
import org.mortgages.LoanApplication;
```

2.3. DRL の機能

DRL ファイルの関数は、Java クラスではなく、ルールのソースファイルにセマンティックコードを追加します。関数は、特に、ルールのアクション (**then**) 部分が繰り返し使用され、パラメーターだけがルールごとに異なる場合に便利です。DRL ファイルのルールで、関数を宣言したり、ヘルパークラスから静的メソッドを関数としてインポートしたりすることで、ルールのアクション (**then**) 部分で、名前を指定して関数を使用できます。

以下は、DRL ファイルで宣言またはインポートされる関数の例です。

ルールを含む関数宣言の例 (オプション 1)

-

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

rule "Using a function"
when
    // Empty
then
    System.out.println( hello( "James" ) );
end
```

ルールを含む関数インポートの例 (オプション 2)

```
import function my.package.applicant.hello;

rule "Using a function"
when
    // Empty
then
    System.out.println( hello( "James" ) );
end
```

2.4. DRL のクエリー

DRL ファイルのクエリーは、デシジョンエンジンのワーキングメモリーで DRL ファイル内のルールに関連するファクトを検索します。DRL ファイルにクエリー定義を追加してから、アプリケーションコードで一致する結果を取得します。クエリーは、定義された条件セットを検索するため、**when** または **then** を指定する必要はありません。クエリー名は KIE ベースでグローバルとなるため、プロジェクト内の他のすべてのルールクエリーと重複しないように固有の名前にする必要があります。クエリーの結果を返すには、**ksession.getQueryResults("name")** を使用して **QueryResults** 定義を構成します ("name" はクエリー名)。これにより、クエリーの結果の一覧が返され、クエリーに一致したオブジェクトを取得できるようになります。DRL ファイルのルールに、クエリーとクエリー結果パラメーターを定義します。

以下は、ローン申請デシジョンサービスの未成年の申請者に関する DRL ファイルのクエリー定義と、付属のアプリケーションコードの例です。

DRL ファイルにおけるクエリー定義の例

```
query "people under the age of 21"
    $person : Person( age < 21 )
end
```

クエリー結果を取得するためのアプリケーションコードの例

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age of 21" );
```

標準的な **for** ループを使用して、返される **QueryResults** を反復処理できます。各要素は **QueryResultsRow** で、これを使用してタプルの各列にアクセスできます。

クエリー結果を取得し、反復処理するためのアプリケーションのコード例

```

QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age of 21" );

System.out.println( "These people are under the age of 21:" );

for ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( "person" );
    System.out.println( person.getName() + "\n" );
}

```

2.5. DRL でのタイプ宣言とメタデータ

DRL ファイルの宣言は、DRL ファイルのルールで使用するファクトタイプまたはメタデータを新たに定義します。

- 新規ファクトタイプ:** Red Hat Process Automation Manager に含まれる **java.lang** パッケージのデフォルトのファクトタイプは **Object** ですが、必要に応じて DRL ファイルで他のタイプを宣言できます。DRL ファイルでファクトタイプを宣言すると、Java などの低級言語でモデルを作成せず、デシジョンエンジンで新しいファクトモデルを直接定義することができます。また、ドメインモデルがすでにビルドされていて、推論 (reasoning) のプロセスで主に使用する追加のエンティティーでこのモデルを補完する場合に、新規タイプを宣言することもできます。
- ファクトタイプのメタデータ:** **@key(value)** 形式のメタデータを新規または既存のファクトに関連付けることができます。メタデータには、ファクト属性で表現されない、該当のファクトタイプのすべてのインスタンス間で一貫性のあるすべての種類のデータを使用できます。メタデータはランタイム時に、デシジョンエンジンでクエリーでき、推論 (reasoning) のプロセスで使用できます。

2.5.1. DRL のメタデータを使用しないタイプ宣言

新規ファクトの宣言にメタデータは必要ありませんが、属性またはフィールドの一覧を含める必要があります。タイプ宣言に指定属性が含まれていない場合には、デシジョンエンジンはクラスパス内で既存のファクトクラスを検索し、クラスが見つからない場合はエラーを出します。

以下の例は、DRL ファイルにメタデータがない新規ファクトタイプ **Person** の宣言です。

ルールが1つ含まれる新規ファクトタイプの宣言の例

```

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

rule "Using a declared type"
when
    $p : Person( name == "James" )
then // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName( "Mark" );
    insert( mark );
end

```

この例では、新規ファクトタイプ **Person** には **name**、**dateOfBirth**、および **address** の3つの属性が

含まれます。属性ごとにタイプがあり、このタイプには作成する別のクラスや以前に宣言したファクトタイプなどの、有効な Java タイプを指定できます。**dateOfBirth** 属性には、Java API からの **java.util.Date** タイプがあり、**address** 属性には、以前に定義したファクトタイプ **Address** が含まれます。

宣言するたびにクラスの完全修飾名が記述されないように、完全なクラス名を **import** 句の一部として定義できます。

インポートの完全修飾クラス名でのタイプ宣言例

```
import java.util.Date

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

新規のファクトタイプを宣言すると、デシジョンエンジンはコンパイル時にファクトタイプを表す Java クラスを生成します。生成される Java クラスはタイプ定義の一対一の JavaBeans マッピングとなります。

たとえば、以下の Java クラスは **Person** タイプ宣言例から生成されます。

Person ファクトタイプの宣言用に生成された Java クラス

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

    // Empty constructor
    public Person() {...}

    // Constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // If keys are defined, constructor with keys
    public Person( ...keys... ) {...}

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

Person タイプ宣言が含まれる以前のルールの例が示すように、他のファクトと同様にルールで生成したクラスを使用できます。

宣言された Person ファクトタイプを使用するルールの例

```
rule "Using a declared type"
when
    $p : Person( name == "James" )
then // Insert Mark, who is a customer of James.
    Person mark = new Person();
```

```
mark.setName( "Mark" );
insert( mark );
end
```

2.5.2. DRL の列挙タイプの宣言

DRL は、**declare enum <factType>** 形式の列挙タイプの宣言をサポートします。列挙タイプの後にはコンマ区切りの値の一覧が続き、最後はセミコロンで終了します。これにより、DRL ファイルのルールで列挙リストを使用できます。

たとえば、以下の列挙タイプの宣言では、従業員スケジュールルールの曜日を定義します。

スケジュールルールを使用した列挙タイプ宣言の例

```
declare enum DaysOfWeek

SUN("Sunday"),MON("Monday"),TUE("Tuesday"),WED("Wednesday"),THU("Thursday"),FRI("Friday"),SAT("Saturday");

    fullName : String
end

rule "Using a declared Enum"
when
    $emp : Employee( dayOff == DaysOfWeek.MONDAY )
then
    ...
end
```

2.5.3. DRL の拡張タイプ宣言

DRL は、**declare <factType1> extends <factType2>** 形式のタイプ宣言の継承をサポートします。DRL で宣言したサブタイプで Java で宣言したタイプを拡張するには、フィールドなしの宣言ステートメントで親タイプを繰り返し使用します。

たとえば、以下のタイプ宣言はトップレベルの **Person** タイプから **Student** タイプを拡張し、さらに **Student** サブタイプから **LongTermStudent** タイプを拡張します。

拡張タイプ宣言の例

```
import org.people.Person

declare Person end

declare Student extends Person
    school : String
end

declare LongTermStudent extends Student
    years : int
    course : String
end
```

2.5.4. DRL のメタデータが含まれるタイプ宣言

@key(value) (値は任意) 形式のメタデータをファクトタイプまたはファクト属性に関連付けることができます。メタデータには、ファクト属性で表現されていない、該当のファクトタイプのすべてのインスタンス間で一貫性のあるすべての種類のデータを使用できます。メタデータはランタイム時に、デシジョンエンジンでクエリーでき、推論 (reasoning) のプロセスで使用できます。ファクトタイプの属性の前に宣言するメタデータはファクトタイプに割り当てられ、属性の後に宣言するメタデータはこの特定の属性に割り当てられます。

以下の例では、**@author** と **@dateOfCreation** のメタデータ属性 2 つが **Person** ファクトタイプに、**@key** と **@maxLength** のメタデータアイテム 2 つが **name** 属性に割り当てられています。**@key** メタデータ属性には必須の値がないので、括弧と値は省略されます。

ファクトタイプおよび属性のメタデータ宣言例

```
import java.util.Date

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

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

既存タイプのメタデータ属性を宣言する場合には、すべての宣言の **import** 句の一部として、または個別の **declare** 句の一部として完全修飾クラス名を特定できます。

インポートタイプのメタデータ宣言の例

```
import org.drools.examples.Person

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

宣言タイプのメタデータ宣言例

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

2.5.5. DRL でのファクトタイプと属性宣言のメタデータタグ

DRL 宣言でカスタムメタデータ属性を定義することはできますが、デシジョンエンジンはファクトタイプまたはファクトタイプ属性の宣言についての、以下の事前定義メタデータタグもサポートします。



注記

VoiceCall クラスを参照する本セクションの例では、サンプルアプリケーションドメインモデルに以下のクラスの詳細が含まれていることを前提としています。

Telecom ドメインモデルの例における VoiceCall ファクトクラス

```
public class VoiceCall {
    private String originNumber;
    private String destinationNumber;
    private Date callDateTime;
    private long callDuration; // in milliseconds

    // Constructors, getters, and setters
}
```

@role

このタグは、指定のファクトタイプが複雑なイベントの処理時にデシジョンエンジンにて通常のファクトまたはイベントとして処理されるかどうかを決定します。

デフォルトパラメーター: **fact**

サポート対象のパラメーター: **fact**、**event**

```
@role( fact | event )
```

例: イベントタイプとして VoiceCall の宣言

```
declare VoiceCall
    @role( event )
end
```

@timestamp

このタグは、デシジョンエンジンのすべてのイベントに自動的に割り当てられます。デフォルトでは、タイムはセッションクロックにより提供され、デシジョンエンジンのワーキングメモリーへの挿入時にイベントに割り当てられます。セッションクロックが追加するデフォルトのタイムスタンプの代わりに、カスタムのタイムスタンプ属性を指定することができます。

デフォルトパラメーター: デシジョンエンジンのセッションクロックが追加する時間

サポート対象のパラメーター: セッションクロックタイムまたはカスタムのタイムスタンプ属性

```
@timestamp( <attributeName> )
```

例: VoiceCall のタイムスタンプ属性の宣言

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

このタグは、デシジョンエンジンのイベントの持続期間を決定します。イベントは、interval-based

イベントまたは point-in-time イベントのいずれかになります。interval-based イベントには持続期間があり、この持続期間が経過するまでデシジョンエンジンのワーキングメモリーで持続します。point-in-time イベントには持続期間はなく、基本的には期間の時間単位が 0 の interval-based イベントと同じです。デフォルトでは、デシジョンエンジンのすべてのイベントの持続期間は 0 です。デフォルトの代わりに、カスタムの持続期間属性を指定することができます。
デフォルトパラメーター: null (ゼロ)

サポート対象のパラメーター: カスタムの持続期間属性

```
@duration( <attributeName> )
```

例: VoiceCall の持続期間属性の宣言

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
end
```

@expires

このタグは、デシジョンエンジンのワーキングメモリーでイベントの有効期限が切れるまでの時間を決定します。デフォルトでは、イベントは現在のルール of のいずれにも一致せず、それらのいずれもアクティベートできなくなった時点で失効します。イベント失効後の期間を定義できます。また、このタグの定義は、KIE ベースの一時的な制約やスライディングウィンドウから算出した暗黙的な有効期限のオフセットもオーバーライドします。デシジョンエンジンがストリームモードで実行中の場合にのみ、このタグを使用できます。
デフォルトパラメーター: null (イベントがルールに一致せず、ルールをアクティブにできなくなるとイベントの有効期限が切れる)

サポート対象のパラメーター: `[#d][#h][#m][#s][[ms]]` 形式のカスタムの **timeOffset** 属性

```
@expires( <timeOffset> )
```

例: VoiceCall イベントに対する有効期限のオフセットの宣言

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
  @expires( 1h35m )
end
```

@typesafe

このタグは、型安全性を有効して/有効にせずに指定のファクトタイプをコンパイルするかどうかを決定します。デフォルトでは、すべてのタイプ宣言は型安全性が有効な状態でコンパイルされます。この動作を type-unsafe の評価にオーバーライドすることもできます。type-unsafe の評価の場合、すべての制約は MVEL 制約として生成され、動的に実行されます。これは、一般的なコレクションではない場合や、タイプが混同されているコレクションを処理する場合に便利です。
デフォルトパラメーター: **true**

サポート対象のパラメーター: **true**、**false**

```
@typesafe( <boolean> )
```

例: type-unsafe 評価の VoiceCall の宣言

```
declare VoiceCall
  @role( fact )
  @typesafe( false )
end
```

@serialVersionUID

このタグは、ファクト宣言でシリアル化可能なクラスの **serialVersionUID** の指定値を定義します。シリアル化可能なクラスで明示的に **serialVersionUID** が宣言されていない場合には、[Java Object Serialization Specification](#) に記載されているように、シリアル化のランタイムが、各種のクラスのさまざまな側面に基づいてそのクラスのデフォルトの **serialVersionUID** 値を計算します。ただし、デシリアル化の結果を最適化し、シリアル化した KIE セッションの互換性を向上するには、関連のクラスや DRL 宣言の要件に応じて **serialVersionUID** を設定します。

デフォルトパラメーター: Null

サポート対象のパラメーター: カスタムの **serialVersionUID** 整数

```
@serialVersionUID( <integer> )
```

例: VoiceCall クラスの serialVersionUID の宣言

```
declare VoiceCall
  @serialVersionUID( 42 )
end
```

@key

このタグを指定すると、ファクトタイプ属性をファクトタイプのキー識別子として使用できるようになります。生成されたクラスは **equals()** と **hashCode()** メソッドを実装できるようになり、該当タイプの2つのインスタンスが同等であるかを判別できるようになります。デシジョンエンジンは、すべてのキー属性をパラメーターとして使用してコンストラクターを生成することもできます。

デフォルトパラメーター: なし

サポート対象のパラメーター: なし

```
<attributeDefinition> @key
```

例: Person タイプ属性をキーとして宣言する

```
declare Person
  firstName : String @key
  lastName : String @key
  age : int
end
```

この例では、デシジョンエンジンは **firstName** と **lastName** 属性をチェックして、**Person** の2つのインスタンスが同等であるかどうかを判断しますが、**age** 属性はチェックしません。また、このデシジョンエンジンは暗黙的に3つのコンストラクターを生成します。1つはパラメーターなし、も

う1つ目は **@key** フィールドのあるコンストラクター、3つ目はすべてのフィールドのあるコンストラクターです。

キー宣言に基づくコンストラクターの例

```
Person() // Empty constructor

Person( String firstName, String lastName )

Person( String firstName, String lastName, int age )
```

以下の例のように、キーコンストラクターに基づいてタイプのインスタンスを作成できます。

キーコンストラクターを使用したインスタンスの例

```
Person person = new Person( "John", "Doe" );
```

@position

このタグは、位置引数で宣言されたファクトタイプ属性またはフィールドの位置を決定し、デフォルトで宣言した属性の順番をオーバーライドします。このタグを使用して、タイプ宣言または位置引数で制約の形式を維持しつつ、パターンの位置制約を変更できます。このタグは、クラスパスのクラスにあるフィールドに対してのみ使用できます。1つのクラスのフィールドでこのタグを使用する場合も、使用しない場合もありますが、このタグのない属性は、宣言の順番では最後になります。クラスの継承はサポートされますが、メソッドのインターフェースはサポートされません。デフォルトパラメーター: なし

サポート対象のパラメーター: 整数

```
<attributeDefinition> @position ( <integer> )
```

例: ファクトタイプを宣言し、宣言した順番をオーバーライドする

```
declare Person
  firstName : String @position( 1 )
  lastName : String @position( 0 )
  age : int @position( 2 )
  occupation: String
end
```

この例では、位置引数に含まれる属性の優先順位は以下のとおりです。

1. **lastName**
2. **firstName**
3. **age**
4. **occupation**

位置引数では、位置が既知の名前付きフィールドにマッピングされるので、フィールド名を指定する必要はありません。たとえば、**Person(lastName == "Doe")** の引数は **Person("Doe";)** と同じです。ここでは、**lastName** フィールドには、DRL 宣言の最上位の位置アノテーションが含まれます。セミコロン ; は、その前にあるものはすべて位置引数であることを示します。セミコロンを使

用して引数を区切ることで、パターンで位置引数と名前付き引数を混せて使用できます。バインドされていない位置引数の変数は、その位置にマッピングされているフィールドにバインドされます。

以下のパターン例では、位置引数と名前付き引数をさまざまな方法で構築する方法を示します。このパターンには、制約が2つ、バインディングが1つ含まれており、セミコロンで位置引数のセクションと名前付き引数のセクションを分けています。位置引数では、変数とリテラル、およびリテラルのみを使用する式がサポートされていますが、変数だけの使用はサポートされていません。

位置引数および名前付き引数を含むパターン例

```
Person( "Doe", "John", $a; )

Person( "Doe", "John"; $a : age )

Person( "Doe"; firstName == "John", $a : age )

Person( lastName == "Doe"; firstName == "John", $a : age )
```

位置引数は、**入力引数** または **出力引数** とに分類できます。入力引数は、以前に宣言したバインディングと、ユニフィケーション (unification) を使用したそのバインディングに対する制約が含まれます。出力引数は、この宣言を生成して、バインディングがまだ存在しない場合には、これを位置引数で表現するフィールドにバインディングします。

拡張タイプの宣言で **@position** アノテーションを定義する場合は、属性の位置がサブタイプに継承されるので注意が必要です。このように継承されることで、属性の順番が混同し、混乱を生じさせる可能性があります。2つのフィールドに同じ **@position** の値を指定でき、連続する値は宣言する必要がありません。位置が繰り返し使用される場合には、継承を使用することで競合が発生しないように解決されます。この場合、親タイプの位置の値の優先順位が高く、その後は最初から最後の順番で宣言を使用します。

たとえば、以下の拡張タイプ宣言では、位置の優先順位が混合します。

位置アノテーションが混合した拡張ファクトタイプの例

```
declare Person
  firstName : String @position( 1 )
  lastName : String @position( 0 )
  age : int @position( 2 )
  occupation: String
end

declare Student extends Person
  degree : String @position( 1 )
  school : String @position( 0 )
  graduationDate : Date
end
```

この例では、位置引数に含まれる属性の優先順位は以下のとおりです。

1. **lastName** (親タイプでの位置 0)
2. **school** (サブタイプでの位置 0)
3. **firstName** (親タイプでの位置 1)

4. **degree** (サブタイプでの位置 1)
5. **age** (親タイプでの位置 2)
6. **occupation** (位置アノテーションがない最初のフィールド)
7. **graduationDate** (位置アノテーションがない 2 番目のフィールド)

2.5.6. ファクトタイプに対するプロパティ変更の設定およびリスナー

デフォルトでは、デシジョンエンジンは、ルールがトリガーされるたびに、ファクトタイプに対するすべてのファクトパターンを再評価しません。代わりに、指定のパターン内に制約またはバインドされている変更されたプロパティのみに対応します。たとえば、ルールが、ルールアクションの一環として **modify()** を呼び出すものの、アクションが KIE ベースで新しいデータを生成しない場合、データが変更されないため、デシジョンエンジンはすべてのファクトパターンを自動的に再評価しません。このプロパティのリアクティビティ動作は、KIE ベースでの不要な再帰を阻止し、より効率的なルール評価をもたらします。また、この動作は無限再帰を回避するために **no-loop** ルール属性を必ずしも使用する必要がないことを意味します。

以下の **KnowledgeBuilderConfiguration** オプションを使用して、このプロパティリアクティビティ動作を変更または無効にできます。次に、Java クラスまたは DRL ファイルでプロパティ変更設定を使用し、必要に応じてプロパティリアクティビティを調整します。

- **ALWAYS:** (デフォルト) すべてのタイプはプロパティリアクティブです。ただし、**@classReactive** プロパティ変更設定を使用して、特定タイプのプロパティリアクティビティを無効にできます。
- **ALLOWED:** すべてのタイプはプロパティリアクティブではありません。ただし、**@propertyReactive** プロパティ変更設定を使用して、特定タイプのプロパティリアクティビティを有効にできます。
- **DISABLED:** すべてのタイプはプロパティリアクティブではありません。すべてのプロパティ変更リスナーは無視されます。

KnowledgeBuilderConfiguration におけるプロパティリアクティビティ設定の例

```
KnowledgeBuilderConfiguration config =
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();
config.setOption(PropertySpecificOption.ALLOWED);
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
```

または、Red Hat Process Automation Manager ディストリビューションにおける **standalone.xml** ファイルの **drools.propertySpecific** システムプロパティを更新できます。

システムプロパティにおけるプロパティリアクティビティ設定の例

```
<system-properties>
...
<property name="drools.propertySpecific" value="ALLOWED"/>
...
</system-properties>
```

デシジョンエンジンは、ファクトクラスまたは宣言された DRL ファクトタイプに対して、以下のプロパティ変更の設定およびリスナーをサポートします。

@classReactive

デシジョンエンジンでプロパティリアクティビティが **ALWAYS** に設定されている場合 (すべてのタイプはプロパティリアクティブ)、このタグは特定の Java クラスまたは宣言された DRL ファクトタイプに対してデフォルトのプロパティリアクティビティ動作を無効にします。このタグは、特定パターン内に制約またはバインドされる変更されたプロパティのみに対応するのではなく、ルールがトリガーされるたびに指定されたファクトタイプのすべてのファクトパターンをデシジョンエンジンが再評価する必要がある場合に使用できます。

例: DRL タイプの宣言におけるデフォルトのプロパティリアクティビティの無効化

```
declare Person
  @classReactive
  firstName : String
  lastName : String
end
```

例: Java クラスにおけるデフォルトのプロパティリアクティビティの無効化

```
@classReactive
public static class Person {
  private String firstName;
  private String lastName;
}
```

@propertyReactive

プロパティリアクティビティがデシジョンエンジンで **ALLOWED** に設定されている場合 (指定されていない場合、すべてのタイプはプロパティリアクティブではない)、このタグは特定の Java クラスまたは宣言された DRL ファクトタイプに対してプロパティリアクティビティを有効にします。デシジョンエンジンが指定されたファクトタイプに対して指定のパターン内に制約またはバインドされている変更されたプロパティのみに対応するようにする場合に、このタグを使用できます。

例: DRL タイプの宣言におけるプロパティリアクティビティの有効化 (リアクティビティがグローバルに無効にされる場合)

```
declare Person
  @propertyReactive
  firstName : String
  lastName : String
end
```

例: Java クラスでのプロパティのリアクティビティの有効化 (リアクティビティがグローバルに無効にされる場合)

```
@propertyReactive
public static class Person {
  private String firstName;
  private String lastName;
}
```

@watch

このタグは、DRL ルールのファクトパターンにインラインで指定する追加のプロパティに対する

プロパティリアクティビティを有効にします。このタグがサポートされるのは、デシジョンエンジンでプロパティリアクティビティが **ALWAYS** に設定されている場合か、またはプロパティリアクティビティが **ALLOWED** に設定され、関連するファクトタイプが **@propertyReactive** タグを使用する場合に限られます。DRL ルールでこのタグを使用して、ファクトプロパティリアクティビティ論理で指定されたプロパティを追加または除外できます。
デフォルトパラメーター: なし

サポート対象のパラメーター: プロパティ名、* (all)、! (not)、!* (no properties)

```
<factPattern> @watch ( <property> )
```

例: ファクトパターンにおけるプロパティリアクティビティの有効化または無効化

```
// Listens for changes in both `firstName` (inferred) and `lastName`:
Person(firstName == $expectedFirstName) @watch( lastName )

// Listens for changes in all properties of the `Person` fact:
Person(firstName == $expectedFirstName) @watch( * )

// Listens for changes in `lastName` and explicitly excludes changes in `firstName`:
Person(firstName == $expectedFirstName) @watch( lastName, !firstName )

// Listens for changes in all properties of the `Person` fact except `age`:
Person(firstName == $expectedFirstName) @watch( *, !age )

// Excludes changes in all properties of the `Person` fact (equivalent to using `@classReactivity` tag):
Person(firstName == $expectedFirstName) @watch( !* )
```

デシジョンエンジンは、**@classReactive** タグ (プロパティリアクティビティを無効にする) を使用するファクトタイプのプロパティに対して **@watch** タグを使用する場合に、またはデシジョンエンジンでプロパティリアクティビティが **ALLOWED** に設定され、関連するファクトタイプが **@propertyReactive** タグを使用しない場合に、コンパイルエラーを生成します。また、**@watch(firstName, ! firstName)** などのリスナーアノテーションでプロパティを複製する場合でも、コンパイルエラーが生じます。

@propertyChangeSupport

[JavaBeans Specification](#) で定義されたプロパティ変更のサポートを実装するファクトの場合、このタグによりデシジョンエンジンがファクトプロパティの変更を監視できるようになります。

例: JavaBeans オブジェクトでのプロパティ変更のサポートの宣言

```
declare Person
    @propertyChangeSupport
end
```

2.5.7. アプリケーションコード内の DRL で宣言されたタイプへのアクセス

DRL の宣言タイプは通常 DRL ファイル内で使用され、Java モデルは通常モデルをルールとアプリケーション間で共有する場合に使用されます。宣言タイプは KIE ベースのコンパイル時に生成されるため、アプリケーションはアプリケーションのランタイムまでこの宣言タイプにアクセスできません。状

況によっては、アプリケーションが宣言タイプからファクトに直接アクセスし、処理する必要があります (とくにアプリケーションがデシジョンエンジンをラップして、ルール管理用によりレベルの高い、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)。

アプリケーションコードから宣言タイプを直接処理するには、Red Hat Process Automation Manager で **org.drools.definition.type.FactType** API を使用します。この API を使用して、宣言ファクトタイプでフィールドのインスタンス化、読み取り、書き込みを行います。

以下のコード例では、アプリケーションから **Person** ファクトタイプを直接変更します。

FactType API を使用した宣言されたファクトタイプを処理するアプリケーションコード例

```
import java.util.Date;

import org.kie.api.definition.type.FactType;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;

...

// Get a reference to a KIE base with the declared type:
KieBase kbase = ...

// Get the declared fact type:
FactType personType = kbase.getFactType("org.drools.examples", "Person");

// Create instances:
Object bob = personType.newInstance();

// Set attribute values:
personType.set(bob, "name", "Bob");
personType.set(bob, "dateOfBirth", new Date());
personType.set(bob, "address", new Address("King's Road", "London", "404"));

// Insert the fact into a KIE session:
KieSession ksession = ...
ksession.insert(bob);
ksession.fireAllRules();

// Read attributes:
String name = (String) personType.get(bob, "name");
Date date = (Date) personType.get(bob, "dateOfBirth");
```

API には、一度にすべての属性を設定したり、**Map** コレクションから値を読み取ったり、すべての属性を一度に **Map** コレクションに読み込んだりするなどの、他の便利なメソッドも含まれます。

API の動作は Java リフレクションと似ていますが、API はリフレクションを使用せず、生成されるバイトコードで実装されるさらに高性能のアクセサーに依存します。

2.6. DRL のグローバル変数

DRL ファイルのグローバル変数は通常、ルールの結果で使用するアプリケーションサービスなど、ルールのデータやサービスを提供し、ルールの結果で追加されるログや値など、ルールからのデータを返します。KIE セッション設定や REST 操作を使用したデシジョンエンジンのワーキングメモリーにグロー

バル値を設定し、DRL ファイルのルールの上にグローバル変数を宣言してから、ルールのアクション部分 (**then**) でこれを使用します。グローバル変数が複数ある場合には、DRL ファイルで行を分けて使用してください。

以下は、DRL ファイルにおけるデシジョンエンジンのグローバル変数一覧の設定と、対応するグローバル変数定義の例です。

デシジョンエンジンに対するグローバルリストの設定例

```
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

ルールを使用したグローバル変数定義の例

```
global java.util.List myGlobalList;

rule "Using a global"
when
    // Empty
then
    myGlobalList.add( "My global list" );
end
```



警告

グローバル変数に定数イミュータブル値がない場合には、ルールの条件設定にグローバル変数を使用しないでください。グローバル変数はデシジョンエンジンのワーキングメモリーに挿入されないため、デシジョンエンジンでは変数の値の変更を追跡できません。

グローバル変数を使用してルール間でデータを共有しないでください。ルールは常に、ワーキングメモリーの状態に関して推論し、これに対応するので、ルールからルールにデータを渡す必要がある場合には、データをファクトとしてデシジョンエンジンのワーキングメモリーにアサートしてください。

グローバル変数のユースケースとして、メールサービスの例があります。デシジョンエンジンを呼び出す統合コードで、**emailService** オブジェクトを取得してから、デシジョンエンジンのワーキングメモリーでそのオブジェクトを設定します。DRL ファイルで、**emailService** のグローバルタイプがあり、名前が **"email"** であることを宣言すると、ルールの結果で **email.sendSMS(number, message)** などのアクションを使用できるようになります。

複数のパッケージで同じ ID のグローバル変数を宣言した場合には、すべてのパッケージを同じタイプで設定し、すべてが同じグローバル値を参照するようにする必要があります。

2.7. DRL でのルール属性

ルール属性は、ルールの動作を変更するためにビジネスルールに追加できる追加の仕様です。DRL ファイルでは、ルール属性は、以下の形式で、通常ルールの条件とアクションの上に定義します。複数の属性がある場合には、別々の行に指定します。

```
rule "rule_name"
  // Attribute
  // Attribute
  when
    // Conditions
  then
    // Actions
end
```

以下の表には、ルールに割り当て可能な属性のサポートされる値と名前が一覧でまとめられています。

表2.1 ルールの属性

属性	値
salience	<p>ルールの優先順位を定義する整数。ルールの salience 値を高くすると、アクティベーションキューに追加したときの優先順位が高くなります。</p> <p>例: salience 10</p>
enabled	<p>ブール値。このオプションを選択すると、ルールが有効になります。このオプションを選択しないと、ルールは無効になります。</p> <p>例: enabled true</p>
date-effective	<p>日付定義および時間定義を含む文字列。現在の日時が date-effective 属性よりも後の場合は、このルールがアクティブになります。</p> <p>例: date-effective "4-Sep-2018"</p>
date-expires	<p>日時定義を含む文字列。現在の日時が date-expires 属性よりも後になると、このルールをアクティブにすることはできません。</p> <p>例: date-expires "4-Oct-2018"</p>
no-loop	<p>ブール値。このオプションを選択すると、以前一致した条件がこのルールにより再トリガーとなる場合に、このルールを再度アクティブにする (ループする) ことができません。条件を選択しないと、この状況でルールがループされます。</p> <p>例: no-loop true</p>
agenda-group	<p>ルールを割り当てるアジェンダグループを指定する文字列。アジェンダグループを使用すると、アジェンダをパーティションで区切り、ルールのグループに対する実行をさらに制御できます。フォーカスを取得したアジェンダグループのルールだけがアクティブになります。</p> <p>例: agenda-group "GroupName"</p>

属性	値
activation-group	<p>ルールを割り当てるアクティベーション (または XOR) グループを指定する文字列。アクティベーショングループでアクティブにできるルールは1つだけです。最初のルールが実行されると、アクティベーショングループのすべてのルールの保留中のアクティベーションはすべてキャンセルされます。</p> <p>例: activation-group "GroupName"</p>
duration	<p>ルールの条件が一致している場合に、ルールがアクティブになってからの時間をミリ秒で定義する長整数値。</p> <p>例: duration 10000</p>
timer	<p>ルールのスケジュールに対する int (間隔) または cron タイマー定義を指定する文字列。</p> <p>例: timer (cron:* 0/15 * * * ?) (15 分ごと)</p>
calendar	<p>ルールをスケジュールするための Quartz カレンダー定義。</p> <p>例: calendars "" * 0-7,18-23 ? * "" (営業時間外を除く)</p>
auto-focus	<p>アジェンダグループ内のルールにのみ適用可能なブール値。このオプションが選択されている場合は、次にルールがアクティブになると、そのルールが割り当てられたアジェンダグループに自動的にフォーカスに移ります。</p> <p>例: auto-focus true</p>
lock-on-active	<p>ルールフローグループまたはアジェンダグループ内のルールにのみ適用可能なブール値。このオプションを選択すると、次回ルールのルールフローグループがアクティブになるか、ルールのアジェンダグループがフォーカスを受け取ると、(ルールフローグループがアクティブでなくなるか、アジェンダグループがフォーカスを失うまで) ルールをアクティブにすることができません。これは、no-loop 属性を強力にしたものです。一致するルールのアクティベーションが、(ルール自体によるものだけでなく) アップデートが何に基づいて行われるかにかかわらず破棄されるためです。この属性は、ファクトを修正するルールが多数あり、ルールの再一致と再発行を希望しない計算ルールに適しています。</p> <p>例: lock-on-active true</p>
ruleflow-group	<p>ルールフローグループを指定する文字列。ルールフローグループで、グループが関連するルールフローによってアクティブにされる場合に限りルールを発行できます。</p> <p>例: ruleflow-group "GroupName"</p>

属性	値
dialect	<p>ルールのコード表記に使用される言語を指定する文字列 (JAVA または MVEL)。デフォルトでは、ルールは、パッケージレベルに指定される言語を使用します。ここで指定される言語は、ルールのパッケージ言語設定を上書きします。</p> <p>例: dialect "JAVA"</p>

2.7.1. DRL でのタイマーおよびカレンダールール属性

タイマーおよびカレンダーは、DRL ルール属性であり、これを使用してスケジュールと時間の制約を DRL ルールに適用できます。これらの属性を使用するには、ユースケースによって追加の設定が必要になります。

DRL ルールの **timer** 属性は、ルールのスケジュール設定を行うための **int** (間隔) または **cron** タイマー定義を指定する文字列で、以下の形式をサポートします。

タイマー属性の形式

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

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

間隔タイマー属性の例

```
// Run after a 30-second delay
timer ( int: 30s )
```

```
// Run every 5 minutes after a 30-second delay each time
timer ( int: 30s 5m )
```

cron タイマー属性の例

```
// Run every 15 minutes
timer ( cron: * 0/15 * * * ? )
```

間隔タイマーは、最初に遅延があり、オプションで間隔が繰り返されるという **java.util.Timer** オブジェクトのセマンティクスに従います。Cron タイマーは標準の Unix cron 式に準拠します。

以下の DRL ルール例では、cron タイマーを使用して 15 分ごとに SMS テキストメッセージを送信します。

cron タイマーを使用した DRL ルールの例

```
rule "Send SMS message 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
```


一般的に、タイマーが制御するルールは、ルールがトリガーされた時点でアクティブになり、タイマーの設定に合わせてルールの結果が繰り返し実行されます。実行は、ルールの条件が受信ファクトに一致しなくなる時点で停止します。ただし、デシジョンエンジンがタイマー付きのルールを処理する方法は、デシジョンエンジンが **アクティブモード** か、**パッシブモード** によって異なります。

デフォルトでは、ユーザーまたはアプリケーションが明示的に **fireAllRules()** を呼び出した場合に、デシジョンエンジンは **パッシブモード** で実行され、定義されたタイマー設定によりルールが評価されます。一方、ユーザーまたはアプリケーションが **fireUntilHalt()** を呼び出す場合には、デシジョンエンジンは **アクティブモード** で実行され、ユーザーまたはアプリケーションが **halt()** を明示的に呼び出すまでルールを継続的に評価します。

デシジョンエンジンがアクティブモードの場合には、**fireUntilHalt()** への呼び出しからコントロールが戻った後もルールの結果が実行され、デシジョンエンジンは、ワーキングメモリーに加えられた変更に対して **リアクティブ** のままになります。たとえば、タイマールールの実行のトリガーに関連したファクトを削除すると、反復実行が停止になり、ファクトが挿入されるので、ルールが一致するとそのルールが実行されます。ただし、デシジョンエンジンは継続して **アクティブ** な訳ではなく、ルールの実行後にだけアクティブになります。そのため、タイマーで制御されるルールが次回に実行されるまで、デシジョンエンジンは非同期のファクト挿入には反応しません。KIE セッションを破棄するとタイマーアクティビティはすべて中断されます。

デシジョンエンジンがパッシブモードの場合は、タイマー付きのルールの結果は、**fireAllRules()** が再度呼び出される場合のみ評価されます。ただし、以下の例にあるように、パッシブモードでは **TimedRuleExecutionOption** オプションで KIE セッションを設定することで、デフォルトのタイマー実行動作を変更できます。

パッシブモードでタイマー付きルールを自動的に実行するための KIE セッションの設定

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

以下の例のように、**TimedRuleExecutionOption** オプションに追加で **FILTERED** 仕様を設定することで、対象のルールをフィルターするコールバックを定義できるようになります。

どのタイマー付きのルールを自動的に実行するかをフィルターするための KIE セッション

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

間隔タイマーの場合は、**int** ではなく、**expr** を指定した式タイマーを使用して、固定値の代わりに、式として遅延と間隔の両方を定義できます。

以下の DRL ファイルの例では、遅延と期間を含むファクトタイプを宣言し、その後に後続のルールの式タイマーでこの遅延と期間を使用します。

式タイマーが含まれるルールの例

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

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

この例の **\$d** と **\$p** などの式は、ルールのパターンが一致する部分に定義した変数を使用できます。この変数には **String** の値を使用でき、これは期間や (ミリ秒単位の) 期間の **long** 値に内部で変換される数値に解析できます。

間隔と式のタイマーはいずれも、以下のオプションのパラメーターを使用できます。

- **start** および **end**: **Date**、または **Date** または **long** 値を表す **String**。この値には、**new Date((Number) n).longValue()** の形式の Java の **Date** に変換される **Number** を指定することも可能です。
- **repeat-limit**: タイマーが許可する最大反復回数を定義する整数。**end** および **repeat-limit** の両パラメーターが設定されている場合には、2つのどちらかに先に到達した時点でタイマーが停止します。

オプションの **start**、**end**、および **repeat-limit** パラメーターが使用されるタイマー属性の例

```
timer (int: 30s 1h; start=3-JAN-2020, end=4-JAN-2020, repeat-limit=50)
```

この例では、ルールは1時間ごとに30秒の遅延の後に実行されるようにスケジュールされ、2020年1月3日に開始し、2020年1月4日またはサイクルが50回繰り返された後に終了するようにスケジュールされています。

システムが一時停止すると (セッションがシリアルライズされた後にデシリアルライズされる場合など)、ルールは、一時停止中に実施されなかったアクティベーションの数にかかわらず、実施されなかったアクティベーションからの回復を1回のみ実施するようにスケジュールされており、ルールはその後にタイマー設定と同期して再度スケジュールされます。

DRL ルールの **calendar** 属性は、ルールのスケジュールのための **Quartz** カレンダー定義であり、以下の形式をサポートします。

カレンダー属性の形式

```
calendars "<definition or registered name>"
```

カレンダー属性の例

```
// Exclude non-business hours
calendars "*** 0-7,18-23 ? * *"

// Weekdays only, as registered in the KIE session
calendars "weekday"
```

以下の例のように、Quartz カレンダー API に基づいて Quartz カレンダーを適用し、そのカレンダーを KIE セッションに登録することができます。

Quartz カレンダーの適用

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

KIE セッションでのカレンダーの登録

```
ksession.getCalendars().set( "weekday", weekDayCal );
```

カレンダーは、標準のルールや、タイマーを使用するルールと共に使用できます。カレンダー属性には、**String** リテラルとして記述された1つ以上のコンマ区切りのカレンダー名を含めることができます。

以下のルールの例では、カレンダーとタイマーの両方を使用してルールをスケジュールします。

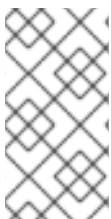
カレンダーとタイマーを使用したルールの例

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

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

2.8. DRL のルール条件 (WHEN)

DRL ルールの **when** 部分 (ルールの左辺 (LHS)とも言う) には、アクションを実行するのに満たされなければならない条件が含まれます。条件は、パッケージ内で使用可能なデータオブジェクトに基づいて、指定された一連の **パターン** および**制約**、オプションの **バインディング** およびサポートされるルール条件要素 (キーワード) で構成されます。たとえば、銀行のローンの申し込みに年齢制限 (21 歳以上) が必要な場合、**Underage** ルールの **when** 条件は **Applicant(age < 21)** となります。



注記

DRL は **if** ではなく **when** を使用します。これは、**if** が一般に手続き型の実行フローの一部であり、条件が特定の時点でチェックされるためです。一方、**when** は、条件の評価が特定の評価シーケンスや時点に限定されず、いつでも継続的に行われることを示しています。条件が満たされるたびに、これらのアクションが実施されます。

when セクションを空にすると、条件は true であると見なされ、デシジョンエンジンで **fireAllRules()** 呼び出しが最初に実施された場合に、**then** セクションのアクションが実行されます。これは、デシジョンエンジンの状態を設定するルールを使用する場合に便利です。

以下のルール例では、空の条件を使用して、ルールが実行されるたびにファクトを挿入します。

条件のないルール例

```
rule "Always insert applicant"
  when
    // Empty
  then // Actions to be executed once
    insert( new Applicant() );
  end

// The rule is internally rewritten in the following way:

rule "Always insert applicant"
  when
    eval( true )
  then
    insert( new Applicant() );
  end
```

ルールの条件が、定義されたキーワード接続詞 (**and**、**or**、または **not** など) なしで複数のパターンを使用する場合、デフォルトの接続詞は **and** になります。

キーワード接続詞なしのルールの例

```
rule "Underage"
  when
    application : LoanApplication()
    Applicant( age < 21 )
  then
    // Actions
  end

// The rule is internally rewritten in the following way:

rule "Underage"
  when
    application : LoanApplication()
    and Applicant( age < 21 )
  then
    // Actions
  end
```

2.8.1. パターンと制約

DRL ルール条件の **パターン** は、デシジョンエンジンによって照合されるセグメントです。パターンは、デシジョンエンジンのワーキングメモリーに挿入される各ファクトと一致する可能性があります。パターンには **制約** を含めることもでき、これにより一致するファクトをより詳細に定義できます。

制約のない最も単純なフォームでは、パターンは指定されたタイプのファクトに一致します。以下の例ではタイプが **Person** であるため、このパターンはデシジョンエンジンのワーキングメモリーのすべての **Person** オブジェクトに一致します。

ファクトタイプが1つの場合のパターン例

■

```
Person()
```

このタイプは、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、複数の異なるクラスのファクトと一致する可能性のあるスーパーユーザーやインターフェースも参照できます。たとえば、以下のパターンは、デシジョンエンジンのワーキングメモリーにあるすべてのオブジェクトと一致します。

すべてのオブジェクトの場合のパターン例

```
Object() // Matches all objects in the working memory
```

パターンの括弧は制約を囲みます (以下のユーザーの年齢に関する制約など)。

制約のあるパターンの例

```
Person( age == 50 )
```

制約は、**true** または **false** を返す式です。DRL 内のパターンの制約は、基本的にはプロパティのアクセスなどの拡張が設定された Java の式ですが、**==** および **!=** に対する **equals()** および **!equals()** セマンティクスなど、若干の違いがあります (通常の **same** および **not same** セマンティクスではありません)。

JavaBean プロパティはパターンの制約から直接アクセスできます。Bean プロパティは、引数を使用せずに何かを返す標準の JavaBeans の getter を使用して内部的に公開されます。たとえば、**age** プロパティは、DRL で getter の **getAge()** ではなく、**age** として記述されます。

JavaBeans プロパティを使用した DRL 制約構文

```
Person( age == 50 )
```

```
// This is the same as the following getter format:
```

```
Person( getAge() == 50 )
```

Red Hat Process Automation Manager は標準の JDK **leavingspector** クラスを使用してこのマッピングを行うため、標準の JavaBeans 仕様に従います。デシジョンエンジンのパフォーマンスの最適化には、**getAge()** のように getter を明示的に使用するのではなく、**age** のようなプロパティアクセスの形式を使用します。



警告

デシジョンエンジンは効率化のために呼び出し間で一致した結果をキャッシュするため、プロパティアクセサーを使用してルールに影響を与える可能性がある仕方ではオブジェクトの状態を変更しないでください。

たとえば、プロパティアクセサーを以下のように使用しないでください。

```
public int getAge() {
    age++; // Do not do this.
    return age;
}
```

```
public int getAge() {
    Date now = DateUtil.now(); // Do not do this.
    return DateUtil.differenceInYears(now, birthday);
}
```

2 番目の例に従う代わりに、ワーキングメモリーに現在の日付をラップするファクトを挿入し、必要に応じてそのファクトを **fireAllRules()** の間で更新します。

ただし、プロパティの getter が見つからなかった場合、コンパイラーは、以下のようにこのプロパティ名をフォールバックメソッド名として引数なしで使用します。

オブジェクトが見つからない場合のフォールバックメソッド

```
Person( age == 50 )

// If `Person.getAge()` does not exist, the compiler uses the following syntax:

Person( age() == 50 )
```

以下の例のように、パターンでアクセスプロパティをネストすることもできます。ネストされたプロパティにはデシジョンエンジンでインデックス化されます。

ネストされたプロパティアクセスを使用するパターンの例

```
Person( address.houseNumber == 50 )

// This is the same as the following format:

Person( getAddress().getHouseNumber() == 50 )
```



警告

ステートフルな KIE セッションでは、ネストされたアクセサーの使用に注意が必要です。デシジョンエンジンのワーキングメモリーではネストされた値は認識されず、これらの値の変更は検出されません。ネストされた値の親参照がワーキングメモリーに挿入されている場合はこれらの値を不変と見なすか、ネストされた値を変更する必要がある場合は、すべての外部ファクトを更新済みとしてマークします。前の例では、**houseNumber** プロパティーが変更された場合は、この **Address** が指定された **Person** は更新済みとしてマークされる必要があります。

パターンの括弧内では **boolean** 値を制約として返す任意の Java 式を使用できます。Java 式は、プロパティーアクセスなどの他の式の拡張機能と組み合わせることができます。

プロパティーアクセスと Java 式を使用する制約が設定されたパターンの例

```
Person( age == 50 )
```

評価の優先度は、論理式や数式のように括弧を使用して変更できます。

制約の評価順序の例

```
Person( age > 100 && ( age % 10 == 0 ) )
```

以下の例のように、制約で Java メソッドを再利用することもできます。

再利用される Java メソッドによる制約の例

```
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```



警告

デシジョンエンジンは効率化を図るために呼び出し間で一致の結果をキャッシュするため、ルールに影響を与える可能性のある方法でオブジェクトの状態を変更するために制約を使用しないでください。ルール条件のファクトで実行されるメソッドは、読み取り専用のメソッドである必要があります。また、ファクトの状態は、ファクトがワーキングメモリーで更新済みとしてマークされているのでない限り、毎回変更されるたびにルールの呼び出し間で変更されません。

たとえば、以下のような方法でパターンの制約を使用しないでください。

```
Person( incrementAndGetAge() == 10 ) // Do not do this.
```

```
Person( System.currentTimeMillis() % 1000 == 0 ) // Do not do this.
```

DRL 内の制約演算子には、標準の Java 演算子の優先順位が適用されます。DRL 演算子は、`==` および `!=` 演算子を除き、標準の Java セマンティクスに従います。

`==` 演算子は、通常の `same` セマンティクスではなく、null 安全な `equals()` セマンティクスを使用します。たとえば、`Person(firstName == "John")` というパターンは `java.util.Objects.equals(person.getFirstName(), "John")` と同様であり、`"John"` は null でないため、このパターンは `"John".equals(person.getFirstName())` にも似ています。

`!=` 演算子は、通常の `not same` セマンティクスではなく null 安全な `!equals()` セマンティクスを使用します。たとえば、`Person(firstName != "John")` というパターンは、`!java.util.Objects.equals(person.getFirstName(), "John")` に似ています。

フィールドと制約の値が異なるタイプの場合、デシジョンエンジンは型強制 (type coercion) を使用して競合を解決し、コンパイルエラーを減らします。たとえば、`"ten"` が数値エバリュエーターで文字列として指定される場合、コンパイルエラーが発生しますが、`"10"` は数値 10 に型強制されます。型強制では、フィールドのタイプは常に値のタイプより優先されます。

型強制された値を使用する制約の例

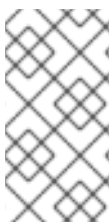
```
Person( age == "10" ) // "10" is coerced to 10
```

制約のグループの場合は、コンマ区切り (,) を使って、暗黙的な `and` の接続的なセマンティクスを使用することができます。

複数の制約があるパターンの例

```
// Person is at least 50 years old and weighs at least 80 kilograms:
Person( age > 50, weight > 80 )
```

```
// Person is at least 50 years old, weighs at least 80 kilograms, and is taller than 2 meters:
Person( age > 50, weight > 80, height > 2 )
```



注記

`&&` および `||` 演算子のセマンティクスは同じですが、これらは異なる優先順位で解決されます。`&&` 演算子は `||` 演算子より優先され、`&&` および `||` 演算子はどちらも、演算子より優先されます。デシジョンエンジンのパフォーマンスと人による可読性を最適化するために、コンマ演算子は最上位レベルの制約で使用してください。

括弧内など、複合制約式にコンマ演算子を埋め込むことはできません。

複合制約式での不適切なコンマの例

```
// Do not use the following format:
Person( ( age > 50, weight > 80 ) || height > 2 )
```

```
// Use the following format instead:
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

2.8.2. パターンと制約でバインドされた変数

パターンおよび制約に変数をバインドして、ルールその他の部分の一致するオブジェクトを参照することができます。バインドされた変数は、ルールをより効率的に、かつデータモデルでのファクトへのアノ

テーションの付け方と一貫した方法で定義するのに役立ちます。(とくに複雑なルールの場合に) ルール内で変数とフィールドを簡単に区別するには、変数に対して標準の形式である **\$variable** を使用します。この規則は便利ですが、DRL で必須ではありません。

たとえば、以下の DRL ルールでは、**Person** ファクトが指定されたパターンに対して変数 **\$p** が使用されています。

バインドされた変数が使用されているパターン

```
rule "simple rule"
  when
    $p : Person()
  then
    System.out.println( "Person " + $p );
  end
```

同様に、以下の例のように、パターンの制約で変数をプロパティにバインドすることもできます。

```
// Two persons of the same age:
Person( $firstAge : age ) // Binding
Person( age == $firstAge ) // Constraint expression
```

注記

より明確で効率的なルールを定義するには、制約のバインディングと制約式を必ず分離します。バインディングと式の組み合わせはサポートされますが、パターンが複雑になり、評価の効率に影響が及ぶ可能性があります。

```
// Do not use the following format:
Person( $age : age * 2 < 100 )

// Use the following format instead:
Person( age * 2 < 100, $age : age )
```

デジジョンエンジンは同じ宣言に対するバインディングをサポートしませんが、複数のプロパティ間での引数の **ユニフィケーション** をサポートします。位置引数は、常にユニフィケーションで常に処理され、名前付き引数の場合はユニフィケーション記号 **:=** が使用されます。

以下のパターンの例では、2つの **Person** ファクト間で **age** プロパティを統合します。

ユニフィケーションが使用されるパターンの例

```
Person( $age := age )
Person( $age := age )
```

ユニフィケーションは、シーケンスオカレンスのバインドされたフィールドの同じ値に対して、最初のオカレンスと制約のバインディングを宣言します。

2.8.3. ネストされた制約とインラインキャスト

以下の例のように、ネストされたオブジェクトの複数のプロパティにアクセスしなければならない場合があります。

複数のプロパティにアクセスするパターンの例

```
Person( name == "mark", address.city == "london", address.country == "uk" )
```

以下の例のように、これらのプロパティのアクセサーを、`.(<constraints>)` という構文を使用してネストされたオブジェクトに対してグループ化することで、ルールを読みやすくすることができます。

グループ化された制約を使用したパターンの例

```
Person( name == "mark", address.( city == "london", country == "uk" ) )
```



注記

ピリオドの接頭辞 `.` は、メソッド呼び出しとネストされたオブジェクト制約とを区別します。

パターンでネストされたオブジェクトを使用する場合は、構文 `<type>#<subtype>` を使用してサブタイプにキャストし、親タイプの getter をサブタイプに対して利用可能にします。以下の例のように、オブジェクト名または完全修飾クラス名のいずれかを使用して、1つまたは複数のサブタイプにキャストできます。

サブタイプへのインラインキャストを使用したパターンの例

```
// Inline casting with subtype name:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000 )
```

これらのパターン例では、**Address** を **LongAddress** に、さらに最後の例にある **DetailedCountry** にキャストし、各ケースのサブタイプで親の getter を利用可能にします。

以下の例のように、**instanceof** 演算子を使用して、パターンを使用した後続のフィールドで指定されたタイプの結果を推測できます。

```
Person( name == "mark", address instanceof LongAddress, address.country == "uk" )
```

インラインキャストが使用できない場合 (たとえば **instanceof** が **false** を返す場合)、評価は **false** と見なされます。

2.8.4. 制約内の日付リテラル

デフォルトで、デシジョンエンジンは **dd-mmm-yyyy** という日付形式をサポートします。この日付形式 (必要に応じて時間形式マスクを含む) は、システムプロパティ **drools.dateformat="dd-mmm-yyyy hh:mm"** を使用して、別の形式マスクを指定することによってカスタマイズすることができます。日付形式は、**drools.defaultlanguage** および **drools.defaultcountry** システムプロパティを使用し、言語ロケールを変更することによってカスタマイズすることもできます (たとえば、タイのロケールは **drools.defaultlanguage=th** および **drools.defaultcountry=TH** と設定します)。

日付のリテラル制限を使用したパターンの例

```
Person( bornBefore < "27-Oct-2009" )
```

2.8.5. DRL のパターン制約でサポートされている演算子

DRL では、パターン制約の演算子で標準の Java セマンティクスがサポートされていますが、いくつかの例外があり、追加となる DRL 固有の演算子もいくつかあります。以下の一覧は、標準の Java セマンティクスとは異なる方法で処理される DRL の制約の演算子や DRL の制約に固有の演算子をまとめています。

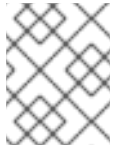
.(),

.() 演算子を使用すると、プロパティのアクセサーをネストされたオブジェクトにグループ化でき、# 演算子を使用すると、ネストされたオブジェクトのサブタイプにキャストできます。サブタイプにキャストすることで、親タイプの getter をサブタイプに対して使用可能にできます。オブジェクト名または完全修飾クラス名のいずれかを使用することができ、1つまたは複数のサブタイプにキャストすることができます。

ネストされたオブジェクトが使用されるパターンの例

```
// Ungrouped property accessors:
Person( name == "mark", address.city == "london", address.country == "uk" )

// Grouped property accessors:
Person( name == "mark", address.( city == "london", country == "uk") )
```



注記

ピリオドの接頭辞 . は、メソッド呼び出しとネストされたオブジェクト制約とを区別します。

サブタイプへのインラインキャストを使用したパターンの例

```
// Inline casting with subtype name:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000 )
```

!.

この演算子を使用すると、null 安全な方法でプロパティを逆参照します。パターンマッチングの適切な結果を得るには、!. 演算子の左側の値を null にすることはできません (!= null と解釈される)。

null 安全な逆参照を使用した制約の例

```
Person( $streetName : address!.street )

// This is internally rewritten in the following way:

Person( address != null, $streetName : address.street )
```

-

[]

この演算子を使用して、インデックスで **List** 値にアクセスするか、またはキーで **Map** 値にアクセスします。

List および Map アクセスを使用する制約の例

```
// The following format is the same as `childList(0).getAge() == 18`:
Person(childList[0].age == 18)

// The following format is the same as `credentialMap.get("jdoe").isValid()`:
Person(credentialMap["jdoe"].valid)
```

<, <=, >, >=

これらの演算子は、自然順序付けのあるプロパティに使用されます。たとえば、< 演算子は、**Date** フィールドでは **前** を意味し、**String** フィールドでは **アルファベット順で前** であることを意味します。これらのプロパティは、比較可能なプロパティにのみ適用されます。

before 演算子を使用した制約の例

```
Person( birthDate < $otherBirthDate )

Person( firstName < $otherFirstName )
```

==, !=

制約では、これらの演算子を、通常の **same** および **not same** セマンティクスではなく、**equals()** および **!equals()** メソッドとして使用します。

null 安全な等価性を使用する制約の例

```
Person( firstName == "John" )

// This is similar to the following formats:

java.util.Objects.equals(person.getFirstName(), "John")
"John".equals(person.getFirstName())
```

null 安全な非等価性を使用する制約の例

```
Person( firstName != "John" )

// This is similar to the following format:

!java.util.Objects.equals(person.getFirstName(), "John")
```

&&, ||

これらの演算子を使用して、フィールドに複数の制約を追加する略記組合せ比較条件を作成します。再帰的な構文パターンを作成するには、括弧 () を使用して制約をグループ化します。

略記組合せ比較を使用する制約の例

```
// Simple abbreviated combined relation condition using a single `&&`:
Person(age > 30 && < 40)

// Complex abbreviated combined relation using groupings:
Person(age ((> 30 && < 40) || (> 20 && < 25)))

// Mixing abbreviated combined relation with constraint connectives:
Person(age > 30 && < 40 || location == "london")
```

matches, not matches

これらの演算子を使用して、指定された Java 正規表現にフィールドが一致するかしないかを示します。一般に、正規表現は **String** リテラルですが、有効な正規表現に解決される変数もサポートされます。これらの演算子は **String** プロパティのみに適用されます。**null** 値に対して **matches** を使用する場合、結果の評価は常に **false** になります。**null** 値に対して **not matches** を使用する場合、結果の評価は常に **true** になります。Java の場合のように、**String** リテラルとして記述された正規表現は二重のバックスラッシュ `\\` を使用してエスケープする必要があります。

正規表現と一致する制約または一致しない制約の例

```
Person( country matches "(USA)?\\S*UK" )

Person( country not matches "(USA)?\\S*UK" )
```

contains, not contains

これらの演算子を使用して、フィールドの **Array** または **Collection** が指定された値を含むか、または含まないかを検証します。これらの演算子は **Array** または **Collection** プロパティに適用されますが、これらの演算子を **String.contains()** および **!String.contains()** の制約チェックの代わりとして使用することもできます。

コレクションに対して contains および not contains が使用された制約の例

```
// Collection with a specified field:
FamilyTree( countries contains "UK" )

FamilyTree( countries not contains "UK" )

// Collection with a variable:
FamilyTree( countries contains $var )

FamilyTree( countries not contains $var )
```

String リテラルに対して contains および not contains が使用された制約の例

```
// Sting literal with a specified field:
Person( fullName contains "Jr" )

Person( fullName not contains "Jr" )

// String literal with a variable:
```

```
Person( fullName contains $var )
```

```
Person( fullName not contains $var )
```



注記

下位互換性を確保するため、**excludes** 演算子は **not contains** の同義語としてサポートされます。

memberOf, not memberOf

これらの演算子を使用して、フィールドが変数として定義されている **Array** または **Collection** のメンバーであるかどうかを検証します。**Array** または **Collection** は変数でなければなりません。

コレクションと memberOf および not memberOf を使用する制約の例

```
FamilyTree( person memberOf $europeanDescendants )
```

```
FamilyTree( person not memberOf $europeanDescendants )
```

soundlike

この演算子を使用して、ある単語を英語で発音した場合に、指定された値と発音がほぼ同じであるかどうかを検証します (**matches** 演算子に類似)。この演算子は Soundex アルゴリズムを使用します。

soundlike を使用した制約の例

```
// Match firstName "Jon" or "John":
Person( firstName soundlike "John" )
```

str

この演算子を使用して、**String** であるフィールドが指定された値で開始されているか、または終了されているかを検証します。この演算子を使用して、**String** の長さを検証することもできます。

str を使用する制約の例

```
// Verify what the String starts with:
Message( routingValue str[startsWith] "R1" )

// Verify what the String ends with:
Message( routingValue str[endsWith] "R2" )

// Verify the length of the String:
Message( routingValue str[length] 17 )
```

in, notin

これらの演算子を使用して、制約の中で一致する可能性がある複数の値を指定します (複合値の制約)。複合値の制約についての機能をサポートするのは **in** 演算子および **not in** 演算子のみです。これらの演算子の 2 番目のオペランドは括弧で囲まれたコンマ区切りの値のリストでなければなりません。値は変数、リテラル、戻り値、または修飾された識別子として指定できます。これらの演算子は、**==** または **!=** 演算子を使用し、複数の制約のリストとして内部に再書き込みされます。

in および notin を使用した制約の例

```
Person( $color : favoriteColor )
Color( type in ( "red", "blue", $color ) )
```

```
Person( $color : favoriteColor )
Color( type notin ( "red", "blue", $color ) )
```

2.8.6. DRL のパターン制約における演算子の優先順位

DRL では、適用可能な制約演算子の場合には標準的な Java 演算子の優先順位をサポートしていますが、一部の例外と、DRL に固有の追加の演算子があります。以下の表には、適用可能な DRL 演算子を優先順位の高いものから低いものの順で記載しています。

表2.2 DRL のパターン制約における演算子の優先順位

演算子のタイプ	演算子	注記
ネストされているか、null 安全なプロパティアクセス	.(), !.	標準の Java セマンティクスではない
List または Map アクセス	[]	標準の Java セマンティクスではない
制約のバインディング	:	標準の Java セマンティクスではない
乗法	*, /%	
加法	+, -	
シフト	>>, >>>, <<	
リレーショナル	<, <=, >, >=, instanceof	
等価性	== !=	標準の Java の same および not same セマンティクスではなく、 equals() および !equals() を使用
非短絡 (Non-short-circuiting) AND	&	
非短絡の排他的 OR	^	
非短絡の包含的 OR		
論理 AND	&&	
論理 OR		

演算子のタイプ	演算子	注記
三項	?:	
コンマ区切り AND	,	標準の Java セマンティクスではない

2.8.7. DRL でサポートされるルール条件要素 (キーワード)

DRL では、DRL のルール条件で定義するパターンで利用できる以下のルール条件要素 (キーワード) がサポートされます。

and

条件コンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。括弧 **()** を使用することにより、パターンを明示的にグループ化できます。デフォルトでは、結合演算子を指定しない場合、リストされているパターンはすべて **and** で結合されます。

and を使用したパターンの例

```
//Infix `and`:
Color( colorType : type ) and Person( favoriteColor == colorType )

//Infix `and` with grouping:
(Color( colorType : type ) and (Person( favoriteColor == colorType ) or Person( favoriteColor == colorType )))

// Prefix `and`:
(and Color( colorType : type ) Person( favoriteColor == colorType ))

// Default implicit `and`:
Color( colorType : type )
Person( favoriteColor == colorType )
```

注記

先頭の宣言のバインディングには **and** キーワードを使用しないでください (**or** などは使用できます)。宣言が参照できるのは一度に1つのファクトのみであり、**and** と宣言のバインディングを使用すると、**and** が満たされた場合にこの要素が両方のファクトと一致してしまうため、エラーが発生します。

and の誤った使用例

```
// Causes compile error:
$person : (Person( name == "Romeo" ) and Person( name == "Juliet"))
```

or

条件コンポーネントを論理和にグループ化します。接中辞および接頭辞の **or** がサポートされます。括弧 **()** を使用することにより、パターンを明示的にグループ化できます。**or** と共にパターンバインディングを使用することもできますが、各パターンは個別にバインディングする必要があります。

or を使用したパターンの例


```
//Infix `or`:
Color( colorType : type ) or Person( favoriteColor == colorType )

//Infix `or` with grouping:
(Color( colorType : type ) or (Person( favoriteColor == colorType ) and Person( favoriteColor ==
colorType )))

// Prefix `or`:
(or Color( colorType : type ) Person( favoriteColor == colorType ))
```

or とパターンのバインディングを使用したパターンの例

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

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

or 条件要素の動作は、フィールド制約の制約および制限を対象とした接続的な **||** 演算子とは異なります。デジジョンエンジンは **or** 要素を直接解釈しませんが、論理変換を使用して、**or** が使用されているルールを複数のサブルールに書き換えます。このプロセスにより、最終的には、ルートノードおよび各条件要素のサブルールとして、1つの **or** を使用するルールが生成されます。各サブルールは、通常のルールと同様に有効にされ、実行されます。その場合、サブルール間で動作や対話はとくに行われません。

したがって、**or** 条件要素については複数の類似したルールを生成するための近道であり、複数の論理和が true である場合には、複数のアクティベーションが作成される可能性があることに留意してください。

exists

存在している必要のあるファクトおよび制約を指定します。このオプションは、初回の一致についてのみトリガーされ、後続の一致については無視されます。この要素を複数のパターンで使用する場合は、これらのパターンを括弧 **()** で囲みます。

exists を使用したパターンの例

```
exists Person( firstName == "John")

exists (Person( firstName == "John", age == 42 ))

exists (Person( firstName == "John" ) and
        Person( lastName == "Doe" ))
```

not

存在してはならないファクトと制約を指定します。この要素を複数のパターンで使用する場合は、これらのパターンを括弧 **()** で囲みます。

not を使用したパターンの例

```
not Person( firstName == "John")

not (Person( firstName == "John", age == 42 ))
```

```
not (Person( firstName == "John" ) and
    Person( lastName == "Doe" ))
```

forall

最初のパターンと一致するすべてのファクトが残りのパターンのすべてと一致するかどうかを検証します。**forall** 構成が満たされると、ルールは **true** と評価されます。この要素は範囲の区切りであるため、以前にバインドされたすべての変数を使用できますが、内部でバインドされた変数を外部で使用することはできません。

forall を使用したルールの例

```
rule "All full-time employees have red ID badges"
when
    forall( $emp : Employee( type == "fulltime" )
        Employee( this == $emp, badgeColor = "red" ) )
then
    // True, all full-time employees have red ID badges.
end
```

この例では、ルールによってタイプが **"fulltime"** であるすべての **Employee** オブジェクトが選択されます。このパターンに一致するそれぞれのファクトに対して、ルールは、従うパターン (バッジの色) を評価し、一致すると、ルールは **true** と評価されます。

デシジョンエンジンのワーキングメモリー内の指定されたタイプのファクトすべてが制約セットに一致するように指定するには、**forall** を単一パターンで使用し、単純化を図ることができます。

forall と1つのパターンを使用するルールの例

```
rule "All full-time employees have red ID badges"
when
    forall( Employee( badgeColor = "red" ) )
then
    // True, all full-time employees have red ID badges.
end
```

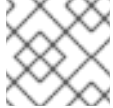
複数のパターンと **forall** 構成を使用するか、または **not** 要素構成内など、他の条件要素でネスト化することができます。

forall と複数のパターンを使用するルールの例

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

forall と not を使用するルールの例

```
rule "Not all employees have health and dental care"
  when
    not ( forall( $emp : Employee()
      HealthCare( employee == $emp )
      DentalCare( employee == $emp ) )
  )
  then
    // True, not all employees have health and dental care.
  end
```



注記

forall(p1 p2 p3 ...) の形式は **not(p1 and not(and p2 p3 ...))** と等価です。

from

これを使用してパターンのデータソースを指定します。これにより、デシジョンエンジンがワーキングメモリーにないデータに対して推論できるようになります。データソースには、バインドされた変数のサブフィールド、またはメソッド呼び出しの結果を指定できます。オブジェクトソースの定義に使用される式として、通常の MVEL 構文に準拠する任意の式を使用できます。このため、**from** 要素により、オブジェクトプロパティのナビゲーションを使用して、メソッド呼び出しを実行し、マップとコレクション要素にアクセスすることが簡単にできます。

from およびパターンのバインディングを使用するルールの例

```
rule "Validate zipcode"
  when
    Person( $personAddress : address )
    Address( zipcode == "23920W" ) from $personAddress
  then
    // Zip code is okay.
  end
```

from とグラフ表記を使用するルールの例

```
rule "Validate zipcode"
  when
    $p : Person()
    $a : Address( zipcode == "23920W" ) from $p.address
  then
    // Zip code is okay.
  end
```

すべてのオブジェクトに対して反復処理される from のルールの例

```
rule "Apply 10% discount to all items over US$ 100 in an order"
  when
    $order : Order()
    $item : OrderItem( value > 100 ) from $order.items
  then
    // Apply discount to ` $item `.
  end
```

注記

オブジェクトコレクションの規模が大きい場合には、以下の例のように、サイズの大きいグラフが含まれるオブジェクトを追加して、デシジョンエンジンが頻繁に反復作業を行うのではなく、コレクションを KIE セッションに直接追加して、条件内でコレクションを結合します。

```
when
  $order : Order()
  OrderItem( value > 100, order == $order )
```

from および lock-on-active ルール属性を使用するルールの例

```
rule "Assign people in North Carolina (NC) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
  when
    $p : Person()
    $a : Address( state == "NC" ) from $p.address
  then
    modify ($p) {} // Assign the person to sales region 1.
  end
```

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

重要

from と **lock-on-active** のルール属性を同時に使用すると、ルールが実行されなくなります。この問題に対しては、以下のいずれかの方法で対処できます。

- すべてのファクトをデシジョンエンジンのワーキングメモリーに挿入したり、制約式でネストされたオブジェクト参照を使用したりする場合は、**from** 要素は使用しないでください。
- ルール条件の最後の文として、**modify()** ブロックで使用される変数を配置します。
- 同じルールフローグループ内のルールがアクティベーションを相互に組み込む方法を明示的に管理できる場合、**lock-on-active** ルール属性は使用しないでください。

from 句を含むパターンの後に、括弧から始まる別のパターンを使用することはできません。この制限がある理由は、DRL パーサーが **from** 式を "**from \$l (String() or Number())**" として読み取り、この式を関数呼び出しと区別できないためです。この最も単純な回避策は、以下の例に示すように、**from** 句を括弧でラップする方法です。

from が適切に使用されていないルールと適切に使用されているルールの例

```
// Do not use `from` in this way:
rule R
  when
    $l : List()
    String() from $l
    (String() or Number())
  then
    // Actions
  end

// Use `from` in this way instead:
rule R
  when
    $l : List()
    (String() from $l)
    (String() or Number())
  then
    // Actions
  end
```

entry-point

エントリーポイントまたはパターンのデータソースに対応した **イベントストリーム** を定義します。この要素は通常、**from** 条件要素と共に使用します。イベントのエントリーポイントを宣言し、デシジョンエンジンがそのエントリーポイントからのデータのみを使用してルールを評価することが可能です。エントリーポイントは、DRL ルールで参照することによって暗黙的に宣言することも、Java アプリケーションで明示的に宣言することもできます。

from entry-point を使用したルールの例

```
rule "Authorize withdrawal"
  when
    WithdrawRequest( $ai : accountId, $am : amount ) from entry-point "ATM Stream"
    CheckingAccount( accountId == $ai, balance > $am )
  then
    // Authorize withdrawal.
  end
```

EntryPoint オブジェクトが使用され、ファクトが挿入された Java アプリケーションコードの例

```
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.rule.EntryPoint;

// Create your KIE base and KIE session as usual:
KieSession session = ...

// Create a reference to the entry point:
EntryPoint atmStream = session.getEntryPoint("ATM Stream");

// Start inserting your facts into the entry point:
atmStream.insert(aWithdrawRequest);
```

collect

ルールで条件の一部として使用できるオブジェクトのコレクションを定義します。このルールは、指定されたソースまたはデシジョンエンジンのワーキングメモリーのいずれかよりコレクションを取得します。**collect** 要素の結果パターンには、**java.util.Collection** インターフェースを実装し、デフォルトの引数を持たないパブリックコンストラクターを指定する任意の具象クラスを使用できます。**List**、**LinkedList**、および **HashSet** のような Java コレクションを使用することも、独自のクラスを使用することもできます。条件内で **collect** 要素の前に変数がバインドされている場合は、その変数を使用してソースおよび結果パターンの両方を制限することができます。ただし、**collect** 要素内で作成されるバインディングをその外部で使用することはできません。

collect を使用するルールの例

```
import java.util.List

rule "Raise priority when system has more than three pending alarms"
when
    $system : System()
    $alarms : List( size >= 3 )
        from collect( Alarm( system == $system, status == 'pending' ) )
then
    // Raise priority because `$system` has three or more `$alarms` pending.
end
```

この例では、ルールは指定された各システムのデシジョンエンジンのワーキングメモリーの保留中のすべてのアラームを評価し、それらを **List** にグループ化します。指定されたシステムについての3つ以上のアラームが見つかった場合には、ルールが実行されます。

以下の例のように、ネストされた **from** 要素と共に **collect** 要素を使用することもできます。

collect とネストされた from を使用するルールの例

```
import java.util.LinkedList;

rule "Send a message to all parents"
when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
        from collect( Person( children > 0 )
            from $town.getPeople()
        )
then
    // Send a message to all parents.
end
```

accumulate

オブジェクトのコレクションを反復処理し、各要素に対してカスタムアクションを実行し、結果オブジェクトを返します(制約が **true** に評価される場合)。この要素は、**collect** 条件要素のより柔軟性が高い、強化された形式です。**accumulate** 条件で事前に定義した関数を使用するか、必要に応じてカスタム関数を実装できます。また、ルール条件で **accumulate** の短縮形である **acc** を使用することもできます。

以下の形式を使用して、ルールに **accumulate** 条件を定義します。

accumulate の推奨形式

■

```
accumulate( <source pattern>; <functions> [;<constraints>] )
```



注記

デシジョンエンジンは下位互換性を確保するために **accumulate** 要素の代替形式をサポートしますが、この形式は、ルールとアプリケーションの最適なパフォーマンスという点ではより適しています。

デシジョンエンジンは、以下の事前に定義された **accumulate** 関数をサポートします。これらの関数は、任意の式を入力として受け入れます。

- **average**
- **min**
- **max**
- **count**
- **sum**
- **collectList**
- **collectSet**

以下のルールの例では、**min**、**max**、および **average** は **accumulate** 関数であり、各センサーの読み取り値での最低、最高、および平均の温度値を算出します。

温度値を算出する **accumulate** を使用したルールの例

```
rule "Raise alarm"
when
  $s : Sensor()
  accumulate( Reading( sensor == $s, $temp : temperature );
    $min : min( $temp ),
    $max : max( $temp ),
    $avg : average( $temp );
    $min < 20, $avg > 70 )
then
  // Raise the alarm.
end
```

以下のルールの例では、**accumulate** を指定した **average** 関数を使用して、ある注文のすべてのアイテムの平均収益を計算します。

平均収益を計算する **accumulate** を使用したルールの例

```
rule "Average profit"
when
  $order : Order()
  accumulate( OrderItem( order == $order, $cost : cost, $price : price );
    $avgProfit : average( 1 - $cost / $price ) )
then
  // Average profit for ` $order ` is ` $avgProfit `.
end
```

■

accumulate の条件でカスタムかつドメイン固有の関数を使用するには、**org.kie.api.runtime.rule.AccumulateFunction** インターフェースを実装する Java クラスを作成します。たとえば、以下の Java クラスは **AverageData** 関数のカスタム実装を定義します。

average 関数がカスタムで実装された Java クラスの例

```
// An implementation of an accumulator capable of calculating average values

public class AverageAccumulateFunction implements
org.kie.api.runtime.rule.AccumulateFunction<AverageAccumulateFunction.AverageData> {

    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

    }

    public void writeExternal(ObjectOutput out) throws IOException {

    }

    public static class AverageData implements Externalizable {
        public int    count = 0;
        public double total = 0;

        public AverageData() {}

        public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
            count = in.readInt();
            total = in.readDouble();
        }

        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeInt(count);
            out.writeDouble(total);
        }

    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#createContext()
     */
    public AverageData createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#init(java.io.Serializable)
     */
    public void init(AverageData context) {
        context.count = 0;
        context.total = 0;
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#accumulate(java.io.Serializable,
     java.lang.Object)

```



```

    */
    public void accumulate(AverageData context,
                          Object value) {
        context.count++;
        context.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#reverse(java.io.Serializable,
     java.lang.Object)
     */
    public void reverse(AverageData context, Object value) {
        context.count--;
        context.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#getResult(java.io.Serializable)
     */
    public Object getResult(AverageData context) {
        return new Double( context.count == 0 ? 0 : context.total / context.count );
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#getResultType()
     */
    public Class< ? > getResultType() {
        return Number.class;
    }
}

```

DRL ルールでカスタム関数を使用するため、**import accumulate** ステートメントを使用してその関数をインポートします。

カスタム関数をインポートするための形式

```
import accumulate <class_name> <function_name>
```

インポートされた **average** 関数を使用するルールの例

```

import accumulate AverageAccumulateFunction.AverageData average

rule "Average profit"
when
    $order : Order()
    accumulate( OrderItem( order == $order, $cost : cost, $price : price );
                $avgProfit : average( 1 - $cost / $price ) )

```

```

then
    // Average profit for `$order` is `$avgProfit`.
end

```

2.8.8. DRL ルール条件内でオブジェクトのグラフが使用される OOPath 構文

OOPath は、DRL ルールの条件の制約でオブジェクトのグラフを参照するために設計された XPath のオブジェクト指向構文の拡張です。OOPath は、コレクションおよびフィルター制約を処理する間に XPath からのコンパクト表記を使用して関連要素を移動します。また、OOPath はとくにオブジェクトグラフの場合に役に立ちます。

ファクトのフィールドがコレクションである場合、**from** 条件要素 (キーワード) を使用してバインドし、そのコレクションのすべての項目を 1 つずつ判断することができます。ルール条件制約でオブジェクトのグラフを参照する必要がある場合、**from** 条件要素を過度に使用すると、以下の例のように冗長かつ繰り返しの多い構文になります。

from を使用してグラフオブジェクトを参照するルールの例

```

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

```

この例では、ドメインモデルには **Student** オブジェクトと学習の **Plan** が含まれています。**Plan** には、ゼロ以上の **Exam** インスタンスを指定でき、**Exam** にはゼロ以上の **Grade** インスタンスを指定できます。このルール設定が機能するためにデシジョンエンジンのワーキングメモリーに存在する必要があるのは、グラフのルートオブジェクト (この例では **Student**) のみです。

from ステートメントを使用するより効率的な別の方法として、以下の例のように短縮された OOPath 構文を使用できます。

OOPath 構文でオブジェクトのグラフを参照するルールの例

```

rule "Find all grades for Big Data exam"
when
    Student( $grade: /plan/exams[course == "Big Data"]/grades )
then
    // Actions
end

```

通常、OOPath 式の中核となる文法は、以下のように拡張された Backus-Naur form (EBNF) 表記で定義されます。

OOPath 式の EBNF 表記

```

OOPEXpr = [ID ( ":" | "!=" )] ( "/" | "?/" ) OOPSegment { ( "/" | "?/" | "." ) OOPSegment } ;
OOPSegment = ID ["#" ID] ["[" ( Number | Constraints ) "]" ]

```

OOPath 式の実際の特性および機能は以下のとおりです。

- 非リアクティブな OOPath 式の場合は、スラッシュ / または疑問符とスラッシュ ?/ で始まります (このセクションで後述します)。
- オブジェクトの単一プロパティを、ピリオド . 演算子を使用して逆参照できます。
- オブジェクトの複数のプロパティをスラッシュ / 演算子を使用して逆参照できます。コレクションが返される場合、この式はコレクションの値を反復処理します。
- 1つまたは複数の制約を満たさない、走査されたオブジェクトをフィルターで除外できます。この制約は、以下の例のように、角括弧内の述語式として記述されます。

述語式としての制約

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

- 汎用コレクションで宣言されたクラスのサブクラスに走査されたオブジェクトをダウンキャストできます。以下の例に示すように、後続の制約も、このサブクラスで宣言されたプロパティにのみ安全にアクセスすることができます。このインラインキャストで指定されたクラスのインスタンスではないオブジェクトは、自動的にフィルターで除外されます。

ダウンキャストオブジェクトが使用される制約

```
Student( $grade: /plan/exams#AdvancedExam[ course == "Big Data", level > 3 ]/grades )
```

- 現在の反復処理されたグラフの前に走査されたグラフのオブジェクトを前方参照できます。たとえば、以下の OOPath 式は、合格した試験の平均を上回るグレードにのみ一致します。

前方参照オブジェクトを使用する制約

```
Student( $grade: /plan/exams/grades[ result > ../averageResult ] )
```

- 以下の例のように、別の OOPath 式を再帰的に使用できます。

再帰的な制約式

```
Student( $exam: /plan/exams[ /grades[ result > 20 ] ] )
```

- 以下の例のように、角括弧 [] 内のオブジェクトのインデックスを使用することにより、そのオブジェクトにアクセスすることができます。Java の規則に従うために、OOPath のインデックスは 0 をベースとし、XPath のインデックスは 1 をベースとします。

インデックスによるオブジェクトへのアクセスが設定された制約

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

OOPath 式は、リアクティブまたは非リアクティブにできます。デシジョンエンジンは、OOPath 式の評価中に走査された深くネストされたオブジェクトを含む更新には反応しません。

これらのオブジェクトが変更に応答するようにするには、**org.drools.core.phreak.ReactiveObject** クラスを拡張するようにオブジェクトを変更します。オブジェクトを変更して **ReactiveObject** クラスを拡張すると、ドメインオブジェクトはいずれかのフィールドが更新されている場合に、以下のように継承されたメソッド **notifyModification** を呼び出して、デシジョンエンジンに通知します。

試験が別のコースに移動したことをデシジョンエンジンに通知するオブジェクトメソッドの例

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

次の対応する OOPath 式を使うと、試験が別のコースに移動された場合にルールが再実行され、そのルールに一致するグレードのリストが再計算されます。

「Big Data」ルールの OOPath 式の例

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

以下の例に示すように、/ セパレーターの代わりに ?/ セパレーターを使用して、OOPath 式の一部のみについてリアクティビティーを無効にすることもできます。

部分的に非リアクティブである OOPath 式の例

```
Student( $grade: /plan/exams[ course == "Big Data" ]?/grades )
```

この例では、デシジョンエンジンは試験に対して実施された変更や、計画に試験が追加された場合に反応しますが、既存の試験に新しいグレードが追加された場合には反応しません。

OOPath の一部が非リアクティブである場合は、OOPath 式の残りの部分も非リアクティブになります。たとえば、以下の OOPath 式は完全に非リアクティブです。

完全に非リアクティブである OOPath 式の例

```
Student( $grade: ?/plan/exams[ course == "Big Data" ]?/grades )
```

こうした理由から、同じ OOPath 式内で ?/ セパレーターを複数回使用することはできません。たとえば、以下の式はコンパイルエラーの原因となります。

重複した非リアクティビティーマーカースを使用する OOPath 式の例

```
Student( $grade: /plan?/exams[ course == "Big Data" ]?/grades )
```

OOPath 式のリアクティビティーを有効にするもう 1 つの方法は、Red Hat Process Automation Manager で **List** および **Set** インターフェースの専用の実装を使用することです。これらの実装は、**ReactiveList** および **ReactiveSet** クラスです。また、**ReactiveCollection** クラスも使用できます。これらの実装により、**Iterator** および **ListIterator** クラスを使用した可変操作の実行もリアクティブにサポートされます。

以下のクラスの例では、これらのクラスを使用して OOPath 式のリアクティビティーを設定します。

OOPath 式のリアクティビティーを設定する Java クラスの例

```
public class School extends AbstractReactiveObject {
    private String name;
    private final List<Child> children = new ReactiveList<Child>(); ❶

    public void setName(String name) {
        this.name = name;
        notifyModification(); ❷
    }
}
```

```

    }

    public void addChild(Child child) {
        children.add(child); ❸
        // No need to call `notifyModification()` here
    }
}

```

- ❶ 標準の Java **List** インスタンスに対するリアクティブサポートのために、**ReactiveList** インスタンスを使用します。
- ❷ フィールドがリアクティブなサポートで変更された場合は、必須の **notifyModification()** メソッドを使用します。
- ❸ **children** フィールドは **ReactiveList** のインスタンスであるため、**notifyModification()** メソッド呼び出しは必要ありません。通知は、**children** フィールドで実行される他のすべての変更操作と同様に、自動的に処理されます。

2.9. DRL におけるルールアクション (THEN)

ルールの **then** 部分 (または、ルールの 右辺 (RHS)) には、ルールの条件部分が満たされる場合に実行されるアクションが含まれます。アクションは、1つ以上の **メソッド** で構成されます。これらのメソッドは、ルール条件とパッケージ内で使用できる利用可能なデータオブジェクトに応じて結果を実行します。たとえば、銀行がローン申請者が 21 歳を超えていることを要件としているが (ルール条件が **Applicant(age < 21)**)、ローン申請者が 21 歳未満の場合、**"Underage"** ルールの **then** アクションは **setApproved(false)** となり、年齢が基準に達していないためローンの申し込みは承認されません。

ルールアクションの主な目的は、デジジョンエンジンのワーキングメモリー内でデータの挿入、削除、または変更を行うことです。有効なルールアクションは小規模かつ宣言的で、可読性があるものです。ルールアクションで必須または条件付きコードを使用する必要がある場合は、ルールを小規模かつより宣言的な複数のルールに分割します。

申込者の年齢制限に関するルールの例

```

rule "Underage"
when
    application : LoanApplication()
    Applicant( age < 21 )
then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end

```

2.9.1. DRL でサポートされるルールアクションメソッド

DRL では DRL ルールアクションで使用する以下のルールアクションメソッドがサポートされています。これらのメソッドを使用することで、最初にワーキングメモリーインスタンスを参照せずに、デジジョンエンジンのワーキングメモリーを変更できます。これらのメソッドは Red Hat Process Automation Manager ディストリビューションの **KnowledgeHelper** クラスで提供されるメソッドへのショートカットとして機能します。

すべてのルールアクションメソッドについては、[Red Hat カスタマーポータル](#) から Red Hat Process Automation Manager 7.7.0 Source Distribution ZIP ファイルをダウンロードし、`~/rhpam-7.7.0-sources/src/drools-$VERSION/drools-`

`core/src/main/java/org/drools/core/spi/KnowledgeHelper.java` に移動してください。

set

これを使用してフィールドの値を設定します。

```
set<field> ( <value> )
```

ローン申し込みの承認の値を設定するルールアクションの例

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

modify

ファクトの変更するフィールドを指定し、デシジョンエンジンに変更を通知します。このメソッドは、ファクトの更新に対する構造化されたアプローチを提供します。このメソッドは、**update** 操作とオブジェクトフィールドを変更する setter 呼び出しを組み合わせたものです。

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

ローン申し込み件数および承認を変更するルールアクションの例

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

update

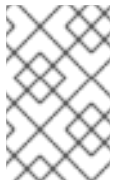
フィールドと更新される関連ファクト全体を指定して、その変更をデシジョンエンジンに通知します。ファクトが変更されたら、更新された値の影響を受ける可能性がある別のファクトを変更する前に、**update** を呼び出す必要があります。この追加設定を回避するには、**modify** メソッドを代わりに使用します。

```
update ( <object, <handle> ) // Informs the decision engine that an object has changed

update ( <object> ) // Causes `KieSession` to search for a fact handle of the object
```

ローンの申し込み件数および承認を更新するルールアクションの例

```
LoanApplication.setAmount( 100 );
update( LoanApplication );
```



注記

property-change リスナーを指定する場合は、オブジェクトの変更時にこのメソッドを呼び出す必要はありません。property-change リスナーの詳細については、「[Red Hat Process Automation Manager のデシジョンエンジン](#)」を参照してください。

insert

new ファクトをデシジョンエンジンのワーキングメモリーに挿入し、ファクトに必要な結果として生成されるフィールドと値を定義します。

```
insert( new <object> );
```

新しいローン申請者オブジェクトを挿入するルールアクションの例

```
insert( new Applicant() );
```

insertLogical

デシジョンエンジンに **new** ファクトを論理挿入する場合に使用します。デシジョンエンジンはファクトの挿入、取り消しに対して論理的な決定を行う役割を担います。通常の挿入または記述による挿入の後には、ファクトは明示的に取り消される必要があります。論理挿入後にファクトを挿入したルールの条件が true でなくなった場合、挿入されたファクトは自動的に取り消されます。

```
insertLogical( new <object> );
```

新しいローン申請者オブジェクトを論理的に挿入するルールアクションの例

```
insertLogical( new Applicant() );
```

delete

デシジョンエンジンからオブジェクトを削除します。キーワード **retract** も DRL でサポートされており、同じアクションを実行しますが、DRL のコードでは、キーワード **insert** との整合性を考慮して **delete** が通常推奨されます。

```
delete( <object> );
```

ローン申請者オブジェクトを削除するルールアクションの例

```
delete( Applicant );
```

2.9.2. drools および kcontext 変数のその他のルールアクションメソッド

標準のルールアクションメソッドに加えて、デシジョンエンジンでは、ルールアクションで使用できる事前定義された **drools** および **kcontext** 変数と組み合わせて使用できるメソッドもサポートしています。

drools 変数を使用して、Red Hat Process Automation Manager ディストリビューションの **KnowledgeHelper** クラスからメソッドを呼び出すことができます。これは、標準のアクションメソッドのベースとなるクラスでもあります。すべての **drools** ルールアクションのオプションについては、[Red Hat カスタマーポータル](#) から Red Hat Process Automation Manager 7.7.0 Source Distribution の ZIP ファイルをダウンロードし、`~/rhpam-7.7.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/spi/KnowledgeHelper.java` に移動してください。

以下の例は、**drools** 変数と共に使用できる一般的なメソッドです。

- **drools.halt()**: ユーザーまたはアプリケーションで以前に **fireUntilHalt()** が呼び出されている場

合は、ルールの実行を終了します。ユーザーまたはアプリケーションが **fireUntilHalt()** を呼び出す場合、デシジョンエンジンは **アクティブモード** で開始され、ユーザーまたはアプリケーションが **halt()** を明示的に呼び出すまでルールの評価を継続します。それ以外の場合、デシジョンエンジンはデフォルトで **パッシブモード** で実行され、ユーザーまたはアプリケーションが明示的に **fireAllRules()** を呼び出す場合にのみルールを評価します。

- **drools.getWorkingMemory(): WorkingMemory** オブジェクトを返します。
- **drools.setFocus("<agenda_group>")**: ルールが属する指定されたアジェンダグループにフォーカスを設定します。
- **drools.getRule().getName()**: ルールの名前を返します。
- **drools.getTuple()**、**drools.getActivation()**: 現在実行されているルールに一致する **Tuple** を返し、対応する **Activation** を提供します。これらの呼び出しは、ロギングやデバッグを行う場合に役立ちます。

kcontext 変数を **getKieRuntime()** メソッドと共に使用して、**KieContext** クラスや、拡張によりご使用の Red Hat Process Automation Manager ディストリビューションの **RuleContext** クラスから別のメソッドを呼び出すことができます。完全な Knowledge Runtime API は **kcontext** 変数を通じて公開され、詳細なルールアクションメソッドを提供します。**kcontext** ルールアクションのすべてのオプションについては、[Red Hat カスタマーポータル](#) から Red Hat Process Automation Manager 7.7.0 Source Distribution の ZIP ファイルをダウンロードし、`~/rhpam-7.7.0-sources/src/kie-api-parent-$VERSION/kie-api/src/main/java/org/kie/api/runtime/rule/RuleContext.java` に移動してください。

以下の例は **kcontext.getKieRuntime()** の変数とメソッドの組み合わせで使用できる一般的なメソッドです。

- **kcontext.getKieRuntime().halt()**: ユーザーまたはアプリケーションで以前に **fireUntilHalt()** が呼び出されている場合は、ルールの実行を終了します。このメソッドは、**drools.halt()** メソッドと同等です。ユーザーまたはアプリケーションが **fireUntilHalt()** を呼び出す場合、デシジョンエンジンは **アクティブモード** で開始され、ユーザーまたはアプリケーションが **halt()** を明示的に呼び出すまでルールの評価を継続します。それ以外の場合、デシジョンエンジンはデフォルトで **パッシブモード** で実行され、ユーザーまたはアプリケーションが明示的に **fireAllRules()** を呼び出す場合にのみルールを評価します。
- **kcontext.getKieRuntime().getAgenda()**: KIE セッション **Agenda** への参照を返し、次にルールアクティベーショングループ、ルールアジェンダグループ、およびルールフローグループにアクセスできるようにします。

アジェンダグループ「CleanUp」にアクセスしてフォーカスを設定する呼び出しの例

```
kcontext.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp" ).setFocus();
```

この例は、**drools.setFocus("CleanUp")** と同等です。

- **kcontext.getKieRuntime().getQueryResults(<string> query)**: クエリーを実行し、結果を返します。このメソッドは **drools.getKieRuntime().getQueryResults()** と同等です。
- **kcontext.getKieRuntime().getKieBase()**: **KieBase** オブジェクトを返します。KIE ベースはルールシステムのすべてのナレッジのソースであり、現在の KIE セッションの開始元です。
- **kcontext.getKieRuntime().setGlobal()**、**~.getGlobal()**、**~.getGlobals()**: グローバル変数を設定するか、または取得します。

- `kcontext.getKieRuntime().getEnvironment()`: 使用するオペレーティングシステム環境と類似したランタイムの **Environment** を返します。

2.9.3. 条件付きおよび名前付きの結果を伴う高度なルールアクション

一般的に、有効なルールアクションは小規模かつ宣言的であり、可読性があります。ただし、場合によっては、ルールごとに結果を1つに制限することが困難であり、以下のルールの例のように、ルールの構文が冗長になったり、繰り返しが多くなる可能性があります。

冗長で繰り返しの構文の多いルールの例

```
rule "Give 10% discount to customers older than 60"
  when
    $customer : Customer( age > 60 )
  then
    modify($customer) { setDiscount( 0.1 ) };
  end

rule "Give free parking to customers older than 60"
  when
    $customer : Customer( age > 60 )
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
  end
```

繰り返しを部分的に解決する手段として、以下の変更例にあるように、2 番目のルールで最初のルールを拡張します。

拡張された条件を使用して部分的に強化されたルールの例

```
rule "Give 10% discount to customers older than 60"
  when
    $customer : Customer( age > 60 )
  then
    modify($customer) { setDiscount( 0.1 ) };
  end

rule "Give free parking to customers older than 60"
  extends "Give 10% discount to customers older than 60"
  when
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
  end
```

より効率的な代替方法として、以下の例に示すように、変更した条件およびラベルが付与された対応するルールアクションを使用して、2 つのルールを1つのルールに統合することができます。

条件および名前付きの結果を使用した統合されたルールの例

```
rule "Give 10% discount and free parking to customers older than 60"
  when
    $customer : Customer( age > 60 )
  do[giveDiscount]
```

```

$car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
then[giveDiscount]
  modify($customer) { setDiscount( 0.1 ) };
end

```

このルールの例では、通常のデフォルトアクションと、**giveDiscount** という名前のもう1つの別のアクションが使用されています。**giveDiscount** アクションは、KIE ベースで年齢が 60 歳を超えた顧客が見つかったと、その顧客が車を所有しているかどうかにかかわらず、キーワード **do** でアクティブにされます。

名前付きの結果のアクティベーションは、次の例の **if** ステートメントのように、追加の条件を使用して設定できます。**if** ステートメント内の条件は、その直前にあるパターンで常に評価されます。

追加の条件が指定される統合されたルールの例

```

rule "Give free parking to customers older than 60 and 10% discount to golden ones among them"
when
  $customer : Customer( age > 60 )
  if ( type == "Golden" ) do[giveDiscount]
  $car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
then[giveDiscount]
  modify($customer) { setDiscount( 0.1 ) };
end

```

以下のより複雑な例のように、ネストされた **if** および **else if** 構成を使用して、さまざまなルール条件を評価することもできます。

より複雑な条件を使用して統合されたルールの例

```

rule "Give free parking and 10% discount to over 60 Golden customer and 5% to Silver ones"
when
  $customer : Customer( age > 60 )
  if ( type == "Golden" ) do[giveDiscount10]
  else if ( type == "Silver" ) break[giveDiscount5]
  $car : Car( owner == $customer )
then
  modify($car) { setFreeParking( true ) };
then[giveDiscount10]
  modify($customer) { setDiscount( 0.1 ) };
then[giveDiscount5]
  modify($customer) { setDiscount( 0.05 ) };
end

```

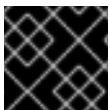
このルールの例では、60 歳を超えた Golden 顧客には 10% の割引きと無料駐車サービスが提供されますが、Silver 顧客に提供されるのは 5% の割引きのみで、無料駐車サービスは提供されません。このルールでは、**do** ではなく **break** というキーワードによって、**giveDiscount5** という名前の結果がアクティブにされます。キーワード **do** は、デシジョンエンジンのアジェンダで結果をスケジュール設定し、ルール条件の残りの部分が引き続き評価されるようにします。一方、**break** は追加の条件の評価を行いません。名前付きの結果が **do** のいずれの条件にも一致しないが、**break** でアクティブにされる場合、ルールの条件部分に到達することはないため、ルールはコンパイルされません。

2.10. DRL ファイルのコメント

DRL では、二重のスラッシュ (//) が先頭に付けられた単一行コメントと、スラッシュおよびアスタリスク (* ... *) で囲まれた複数行のコメントがサポートされます。DRL のコメントを使用すると、DRL ファイル内のルールまたは関連するコンポーネントに対して注釈を付けることができます。DRL ファイルが処理される際、デシジョンエンジンはこれらのコメントを無視します。

コメントが使用されるルールの例

```
rule "Underage"
// This is a single-line comment.
when
    $application : LoanApplication() // This is an in-line comment.
    Applicant( age < 21 )
then
    /* This is a multi-line comment
    in the rule actions. */
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```



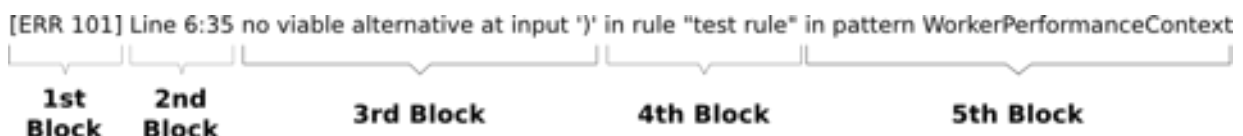
重要

DRL コメントでは、ハッシュ記号 # はサポートされていません。

2.11. DRL トラブルシューティングのエラーメッセージ

Red Hat Process Automation Manager は、DRL エラーに関する標準メッセージを提供します。これらのメッセージは DRL ファイルで問題をトラブルシューティングして解決するのに役立ちます。エラーメッセージでは、以下の形式が使用されています。

図2.1 DRL ファイルの問題に関するエラーメッセージの形式



- 最初のブロック: エラーコード
- 2 番目のブロック: DRL ソースのエラーが発生している行および列
- 3 番目のブロック: 問題の説明
- 4 番目のブロック: DRL ソース内のエラーが発生しているコンポーネント (ルール、関数、クエリー)
- 5 番目のブロック: DRL ソース内のエラーが発生しているパターン (該当する場合)

Red Hat Process Automation Manager では、以下の標準化されたエラーメッセージがサポートされます。

101: no viable alternative

パーサーが決定ポイントに到達したが、選択肢を特定できなかったことを示します。

誤ったスペリングを含むルールの例

```

1: rule "simple rule"
2:  when
3:    exists Person()
4:    exists Student() // Must be `exists`
5:  then
6:  end

```

エラーメッセージ

```
[ERR 101] Line 4:4 no viable alternative at input 'exists' in rule "simple rule"
```

ルール名がないルールの例

```

1: package org.drools.examples;
2: rule // Must be `rule "rule name"` (or `rule rule_name` if no spacing)
3:  when
4:    Object()
5:  then
6:    System.out.println("A RHS");
7:  end

```

エラーメッセージ

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

この例では、パーサーがキーワード **when** を検出しましたが、予想していたのはルール名であったため、パーサーは **when** に予想外の不適切なトークンというフラグを付けます。

誤った構文が使用されるルールの例

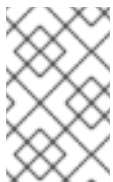
```

1: rule "simple rule"
2:  when
3:    Student( name == "Andy" ) // Must be `"Andy"`
4:  then
5:  end

```

エラーメッセージ

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



注記

列と行の値が **0:-1** の場合、パーサーがソースファイルの終わり (**<eof>**) に到達したが、不完全な構成を検出したことを示します。よくある原因として、引用符 "...", アポストロフィー '...', または括弧 (...) が欠落して場合があります。

102: mismatched input

パーサーが特定の記号を予想していたが、これが現在の入力位置で欠落していることを示します。

不完全なルールステートメントが使用されるルールの例

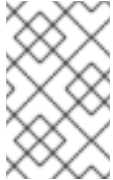
```

1: rule simple_rule
2: when
3:   $p : Person(
      // Must be a complete rule statement

```

エラーメッセージ

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule "simple rule" in pattern Person



注記

列と行の値が **0:-1** の場合、パーサーがソースファイルの終わり (<eof>) に到達したが、不完全な構成を検出したことを示します。よくある原因として、引用符 "...", アポストロフィー '...', または括弧 (...) が欠落して場合があります。

誤った構文が使用されるルールの例

```

1: package org.drools.examples;
2:
3: rule "Wrong syntax"
4: when
5:   not( Car( ( type == "tesla", price == 10000 ) || ( type == "kia", price == 1000 ) ) from $carList )
      // Must use `&&` operators instead of commas `,`
6: then
7:   System.out.println("OK");
8: end

```

エラーメッセージ

[ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Wrong syntax" in pattern Car
 [ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Wrong syntax"
 [ERR 102] Line 5:106 mismatched input ')' expecting 'then' in rule "Wrong syntax"

この例では、構文に関する問題が原因で相互に関連する複数のエラーメッセージが発生しています。**&&** 演算子で, を置き換えるという1つの解決策により、すべてのエラーが解消されます。複数のエラーが発生した場合に、エラーが前のエラーの結果である場合があるため、一度に1つずつ解決します。

103: failed predicate

セマンティクスの述語の検証が **false** と評価されたことを示します。これらのセマンティクスの述語は一般に **declare**、**rule**、**exists**、**not** など、DRL ファイルのコンポーネントのキーワードを特定するために使用されます。

無効なキーワードが使用されているルールの例

```

1: package nesting;
2:
3: import org.drools.compiler.Person
4: import org.drools.compiler.Address
5:
6: Some text // Must be a valid DRL keyword
7:

```

```

8: rule "test something"
9:  when
10:    $p: Person( name=="Michael" )
11:  then
12:    $p.name = "other";
13:    System.out.println(p.name);
14: end

```

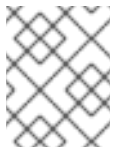
エラーメッセージ

```

[ERR 103] Line 6:0 rule 'rule_key' failed predicate:
{(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule

```

Some text の行は、DRL キーワード構成で始まっていないか、または DRL キーワード構成の一部ではないため、パーサーが DRL ファイルの残りの部分の評価に失敗します。



注記

このエラーは **102: mismatched input** に似ていますが、通常は DRL キーワードが関係しています。

104: trailing semi-colon not allowed

ルール条件の **eval()** 句でセミコロン ; が使用されていますが、セミコロンは使用できません。

eval() と末尾のセミコロンが使用されているルールの例

```

1: rule "simple rule"
2:  when
3:    eval( abc(); ) // Must not use semicolon `;`
4:  then
5: end

```

エラーメッセージ

```

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

```

105: did not match anything

パーサーが文法内で、少なくとも 1 回は代替の選択肢に一致する必要があるサブルールに到達したが、サブルールがいずれにも一致しなかったことを示します。パーサーは出口のないブランチに入ります。

空の条件内に無効なテキストがあるルールの例

```

1: rule "empty condition"
2:  when
3:    None // Must remove `None` if condition is empty
4:  then
5:    insert( new Person() );
6: end

```

エラーメッセージ

[ERR 105] Line 2:2 required (...) + loop did not match anything at input 'WHEN' in rule "empty condition"

この例では、条件は空であることが意図されていますが、**None** という単語が使用されています。このエラーは、DRL の有効でないキーワード、データタイプ、またはパターン構成である **None** を削除することによって解決できます。



注記

解決できないその他の DRL エラーメッセージが発生した場合は、Red Hat のテクニカルアカウントマネージャーにお問い合わせください。

2.12. DRL ルールセットのルールユニット

ルールユニットは、データソース、グローバル変数、および DRL ルールのグループで、特定の目的に向けて互いに機能し合います。ルールユニットを使用して、ルールセットを小さなユニットに分割し、それらのユニットにさまざまなデータソースをバインドしてから、個別のユニットを実行します。ルールユニットは、実行制御用のルールアジェンダグループまたはアクティブ化グループなどの、ルールをグループ化する DRL 属性に代わるものとして、強化されています。

ルールユニットは、ルールの実行を調整することで、あるルールユニットが完全に実行されると別のルールユニットの開始をトリガーする場合などに便利です。たとえば、データ強化用の一連のルール、そのデータを処理する別の一連のルール、および処理されたデータを抽出して出力する別の一連のルールがあるとします。これらのルールセットを 3 つの異なるルールユニットに追加する場合、これらのルールユニットを調整することで、1 つ目のユニットが完全に実行されると 2 つ目のユニットの開始をトリガーし、2 つ目のユニットが完全に実行されると 3 つ目のユニットの開始をトリガーすることができます。

ルールユニットを定義するには、以下の例に示すように **RuleUnit** インターフェースを実装します。

ルールユニットクラスの例

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    private int adultAge;
    private DataSource<Person> persons;

    public AdultUnit( ) {}

    public AdultUnit( DataSource<Person> persons, int age ) {
        this.persons = persons;
        this.age = age;
    }

    // A data source of `Persons` in this rule unit:
    public DataSource<Person> getPersons() {
        return persons;
    }

    // A global variable in this rule unit:
    public int getAdultAge() {
        return adultAge;
    }
}
```

```
// Life-cycle methods:
@Override
public void onStart() {
    System.out.println("AdultUnit started.");
}

@Override
public void onEnd() {
    System.out.println("AdultUnit ended.");
}
}
```

この例では、**persons** はタイプ **Person** のファクトのソースです。ルールユニットのデータソースは、指定のルールユニットで処理されるデータのソースで、デシジョンエンジンがルールユニットの評価に使用するエン트리ポイントを表します。**adultAge** グローバル変数は、このルールユニットに属するすべてのルールからアクセスできます。最後の2つのメソッドは、ルールユニットのライフサイクルの一部で、デシジョンエンジンによって呼び出されます。

デシジョンエンジンは、以下のようなルールユニットのオプションのライフサイクルメソッドをサポートします。

表2.3 ルールユニットのライフサイクルメソッド

メソッド	呼び出されるタイミング
onStart()	ルールユニット実行開始時
onEnd()	ルールユニット実行終了時
onSuspend()	ルールユニット実行の一時停止時 (runUntilHalt() でのみを使用される)
onResume()	ルールユニット実行の再開時 (runUntilHalt() でのみ使用される)
onYield(RuleUnit other)	ルールユニットにおけるルールの結果が異なるルールユニットの実行をトリガー

ルールユニットに、ルールを1つ以上追加することができます。デフォルトでは、DRL ファイルのすべてのルールは、DRL ファイル名の命名規則に従うルールユニットに自動的に関連付けられます。DRL ファイルが同じパッケージにあり、**RuleUnit** インターフェースを実装するクラスと同じ名前を持つ場合、その DRL ファイルのすべてのルールは、そのルールユニットに暗黙的に属します。たとえば、**org.mypackage.myunit** パッケージの **AdultUnit.drl** ファイルにあるすべてのルールは、自動的にルールユニット **org.mypackage.myunit.AdultUnit** の一部となります。

この命名規則をオーバーライドし、DRL ファイル内のルールが属するルールユニットを明示的に宣言するには、DRL ファイル内でキーワード **unit** を使用します。**unit** 宣言は、すぐに **package** 宣言に従い、DRL ファイルのルールが一部となっているパッケージ内のクラス名を含む必要があります。

DRL ファイルのルールユニット宣言の例

```
package org.mypackage.myunit
```



```
unit AdultUnit

rule Adult
  when
    $p : Person(age >= adultAge) from persons
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
  end
```



警告

同じ KIE ベースで、ルールユニットありのルールとルールユニットなしのルールを混在させないでください。KIE ベースで 2 つのルールのパラダイムを混在させると、コンパイルエラーが発生します。

以下の例のように OOPath 表記を使用して、より便利な方法で同じパターンを書き換えることもできます。

OOPath 表記を使用する DRL ファイルのルールユニット宣言の例

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : /persons[age >= adultAge]
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
  end
```



注記

OOPath は、DRL ルールの条件の制約でオブジェクトのグラフを参照するために設計された XPath のオブジェクト指向構文の拡張です。OOPath は、コレクションおよびフィルター制約を処理する間に XPath からのコンパクト表記を使用して関連要素を移動します。また、OOPath はとくにオブジェクトグラフの場合に役に立ちます。

この例では、ルール条件で一致するファクトはすべて、ルールユニットクラスの **DataSource** 定義で定義される **persons** のデータソースから取得されます。ルール条件およびアクションは、グローバル変数が DRL ファイルレベルで定義されるのと同じ方法で **adultAge** 変数を使用します。

KIE ベースに定義されたルールユニットを 1 つ以上実行するには、KIE ベースにバインドされている新規の **RuleUnitExecutor** クラスを作成し、関連するデータソースからルールユニットを作成して、ルールユニットエグゼキューターを実行します。

ルールユニット実行の例

```
// Create a `RuleUnitExecutor` class and bind it to the KIE base:
KieBase kbase = kieContainer.getKieBase();
```

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

// Create the `AdultUnit` rule unit using the `persons` data source and run the executor:
RuleUnit adultUnit = new AdultUnit(persons, 18);
executor.run( adultUnit );
```

ルールは **RuleUnitExecutor** クラスによって実行されます。**RuleUnitExecutor** クラスは KIE セッションを作成し、必要な **DataSource** オブジェクトをこれらのセッションに追加してから、**run()** メソッドにパラメーターとして渡される **RuleUnit** に基づいてルールを実行します。

例の実行コードは、関連する **Person** ファクトが **persons** データソースに挿入されると、以下の出力を生成します。

ルールユニット実行出力の例

```
org.mypackage.myunit.AdultUnit started.
Jane is adult and greater than 18
John is adult and greater than 18
org.mypackage.myunit.AdultUnit ended.
```

ルールユニットインスタンスを明示的に作成するのではなく、エグゼキューターにルールユニット変数を登録し、実行するルールユニットクラスをエグゼキューターに渡すと、エグゼキューターがルールユニットのインスタンスを作成します。続いて、ルールユニットを実行する前に **DataSource** 定義および他の変数を設定できます。

登録変数を含む別のルールユニット実行オプション

```
executor.bindVariable( "persons", persons );
        .bindVariable( "adultAge", 18 );
executor.run( AdultUnit.class );
```

RuleUnitExecutor.bindVariable() メソッドに渡す名前は、実行時に、同じ名前のルールユニットクラスのフィールドに変数をバインドするために使用されます。前述の例では、**RuleUnitExecutor** は、新しいルールユニットに **"persons"** の名前にバインドされているデータソースを挿入します。また、**AdultUnit** クラス内の対応する名前のフィールドに、文字列 **"adultAge"** にバインドされている値 **18** を挿入します。

このデフォルトの変数バインディング動作をオーバーライドするには、**@UnitVar** アノテーションを使用してルールユニットクラスの各フィールドに対して論理バインディング名を明示的に定義します。たとえば、以下のクラスのフィールドバインディングは、代替名で再度定義されます。

@UnitVar を使用した変数バインディング名を変更するコード例

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    @UnitVar("minAge")
    private int adultAge = 18;

    @UnitVar("data")
    private DataSource<Person> persons;
}
```

次に、これらの代替名を使用して、変数をエグゼキューターにバインドし、ルールユニットを実行できます。

変更した変数名を使用したルールユニット実行の例

```
executor.bindVariable( "data", persons );
    .bindVariable( "minAge", 18 );
executor.run( AdultUnit.class );
```

ルールユニットは、**run()** メソッド (KIE セッションで **fireAllRules()** を呼び出す場合と同じ) を使用してパッシブモードで、または **runUntilHalt()** メソッド (KIE セッションで **fireUntilHalt()** を呼び出す場合と同じ) を使用してアクティブモードで実行できます。デフォルトでは、デシジョンエンジンはパッシブモードで実行され、ユーザーまたはアプリケーションが明示的に **run()** (標準ルールでは **fireAllRules()**) を呼び出す場合にのみルールユニットを評価します。ユーザーまたはアプリケーションがルールユニットに **runUntilHalt()** (標準ルールでは **fireAllRules()**) を呼び出す場合、デシジョンエンジンはアクティブモードで開始し、ユーザーまたはアプリケーションが明示的に **halt()** を呼び出すまで、継続的にルールユニットを評価します。

runUntilHalt() メソッドを使用する場合は、メインスレッドをブロックしないように、別の実行スレッド上でメソッドを呼び出します。

別のスレッド上の **runUntilHalt()** を使用したルールユニットの実行例

```
new Thread( () -> executor.runUntilHalt( adultUnit ) ).start();
```

2.12.1. ルールユニットのデータソース

ルールユニットのデータソースは、指定のルールユニットが処理したデータのソースで、デシジョンエンジンがルールユニットの評価に使用するエン트리ポイントを表します。ルールユニットは、ゼロまたは複数のデータソースを持つことができ、ルールユニット内で宣言された各 **DataSource** の定義は、ルールユニットエグゼキューターへの異なるエン트리ポイントに対応することができます。複数のルールユニットは、単一データソースを共有できます。ただし、各ルールユニットは、別々のエン트리ポイントを使用しなければなりません。このエン트리ポイントを介して同じオブジェクトが挿入されます。

以下の例で示すように、ルールユニットクラスの固定されたデータセットを使用して **DataSource** 定義を作成できます。

データソース定義の例

```
DataSource<Person> persons = DataSource.create( new Person( "John", 42 ),
    new Person( "Jane", 44 ),
    new Person( "Sally", 4 ) );
```

データソースはルールユニットのエン트리ポイントを表すため、ルールユニットでファクトを挿入、更新、または削除できます。

ルールユニットでファクトを挿入、更新、削除するコード例

```
// Insert a fact:
Person john = new Person( "John", 42 );
FactHandle johnFh = persons.insert( john );

// Modify the fact and optionally specify modified properties (for property reactivity):
```

```
john.setAge( 43 );
persons.update( johnFh, john, "age" );

// Delete the fact:
persons.delete( johnFh );
```

2.12.2. ルールユニットの実行制御

一方のルールユニットの実行により、もう一方のルールユニットの開始がトリガーされるようにルールの実行を調整する必要がある場合に、ルールユニットは役に立ちます。

ルールユニットの実行制御を容易にするために、デシジョンエンジンは以下のルールユニットメソッドをサポートします。このメソッドは、DRL ルールアクションでを使用して、ルールユニットの実行を調整することができます。

- **drools.run()**: 指定されたルールユニットクラスの実行をトリガーします。このメソッドでは、ルールユニットの実行を命令的に中断し、他の指定されたルールユニットを有効化します。
- **drools.guard()**: 関連付けられたルール条件が満たされるまで、指定されたルールユニットクラスが実行されないようにします (保護します)。このメソッドは、他の指定されたルールユニットの実行を宣言的にスケジュールします。デシジョンエンジンが、保護ルールの条件に対して少なくとも1つの一致をもたらす場合、保護されたルールユニットは有効とみなされます。ルールユニットには、複数の保護ルールを含めることができます。

drools.run() メソッドの例として、それぞれが指定されたルールユニットに属す以下の DRL ルールを検討してください。 **NotAdult** ルールは **drools.run(AdultUnit.class)** メソッドを使用して **AdultUnit** ルールユニットの実行をトリガーします。

drools.run() を使用した制御された実行を含む DRL ルールの例

```
package org.mypackage.mynunit
unit AdultUnit

rule Adult
when
    Person(age >= 18, $name : name) from persons
then
    System.out.println($name + " is adult");
end
```

```
package org.mypackage.mynunit
unit NotAdultUnit

rule NotAdult
when
    $p : Person(age < 18, $name : name) from persons
then
    System.out.println($name + " is NOT adult");
    modify($p) { setAge(18); }
    drools.run( AdultUnit.class );
end
```

この例では、これらのルールからビルドされた KIE ベースから作成された **RuleUnitExecutor** クラスと、これにバインドされている **persons** の **DataSource** 定義も使用します。

ルールエグゼキューターとデータソース定義の例

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Person> persons = executor.newDataSource( "persons",
    new Person( "John", 42 ),
    new Person( "Jane", 44 ),
    new Person( "Sally", 4 ) );
```

この例では、**RuleUnitExecutor** クラスから **DataSource** 定義を直接作成し、これを単一ステートメントで **"persons"** 変数にバインドします。

例の実行コードは、関連する **Person** ファクトが **persons** データソースに挿入されると、以下の出力を生成します。

ルールユニット実行出力の例

```
Sally is NOT adult
John is adult
Jane is adult
Sally is adult
```

NotAdult ルールは、**"Sally"** という人物の評価時に一致を検出します。この人物は 18 歳未満です。続いてこのルールは、この人物の年齢を **18** に変更し、**drools.run(AdultUnit.class)** メソッドを使用して **AdultUnit** ルールユニットの実行をトリガーします。**AdultUnit** ルールユニットには、**DataSource** 定義の 3 人の **persons** 全員に対して実行可能となったルールが含まれています。

drools.guard() メソッドの例として、以下の **BoxOffice** クラスと **BoxOfficeUnit** ルールユニットクラスを検討してください。

BoxOffice クラスの例

```
public class BoxOffice {
    private boolean open;

    public BoxOffice( boolean open ) {
        this.open = open;
    }

    public boolean isOpen() {
        return open;
    }

    public void setOpen( boolean open ) {
        this.open = open;
    }
}
```

BoxOfficeUnit ルールユニットクラスの例

```
public class BoxOfficeUnit implements RuleUnit {
    private DataSource<BoxOffice> boxOffices;

    public DataSource<BoxOffice> getBoxOffices() {
```

```

        return boxOffices;
    }
}

```

また、この例では、以下の **TicketIssuerUnit** ルールユニットクラスを使用して、少なくとも1つのボックスオフィス (チケット売り場) が営業中である限り、ボックスオフィスでのイベントチケットの販売を続行します。このルールユニットは **persons** および **tickets** の **DataSource** 定義を使用します。

TicketIssuerUnit ルールユニットクラスの例

```

public class TicketIssuerUnit implements RuleUnit {
    private DataSource<Person> persons;
    private DataSource<AdultTicket> tickets;

    private List<String> results;

    public TicketIssuerUnit() { }

    public TicketIssuerUnit( DataSource<Person> persons, DataSource<AdultTicket> tickets ) {
        this.persons = persons;
        this.tickets = tickets;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public DataSource<AdultTicket> getTickets() {
        return tickets;
    }

    public List<String> getResults() {
        return results;
    }
}

```

BoxOfficeUnit ルールユニットには、**BoxOfficeIsOpen** DRL ルールが含まれます。これは、**drools.guard(TicketIssuerUnit.class)** メソッドを使用して、イベントチケットを配布する **TicketIssuerUnit** ルールユニットの実行を保護します。以下に DRL ルールの例を示します。

drools.guard() を使用した制御された実行を含む DRL ルールの例

```

package org.mypackage.myunit;
unit TicketIssuerUnit;

rule IssueAdultTicket when
    $p: /persons[ age >= 18 ]
then
    tickets.insert(new AdultTicket($p));
end
rule RegisterAdultTicket when
    $t: /tickets
then
    results.add( $t.getPerson().getName() );
end

```

```

package org.mypackage.myunit;
unit BoxOfficeUnit;

rule BoxOfficelsOpen
  when
    $box: /boxOffices[ open ]
  then
    drools.guard( TicketIssuerUnit.class );
  end

```

この例では、少なくとも1つのボックスオフィスが **open** である限り、保護された **TicketIssuerUnit** ルールユニットが有効なため、イベントチケットは配布されます。**open** 状態のボックスオフィスがなくなると、保護された **TicketIssuerUnit** ルールユニットは実行されなくなります。

以下のクラスの例は、より完全なボックスオフィスのシナリオを説明しています。

ボックスオフィスシナリオのクラスの例

```

DataSource<Person> persons = executor.newDataSource( "persons" );
DataSource<BoxOffice> boxOffices = executor.newDataSource( "boxOffices" );
DataSource<AdultTicket> tickets = executor.newDataSource( "tickets" );

List<String> list = new ArrayList<>();
executor.bindVariable( "results", list );

// Two box offices are open:
BoxOffice office1 = new BoxOffice(true);
FactHandle officeFH1 = boxOffices.insert( office1 );
BoxOffice office2 = new BoxOffice(true);
FactHandle officeFH2 = boxOffices.insert( office2 );

persons.insert(new Person("John", 40));

// Execute `BoxOfficelsOpen` rule, run `TicketIssuerUnit` rule unit, and execute `RegisterAdultTicket`
// rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "John", list.get(0) );
list.clear();

persons.insert(new Person("Matteo", 30));

// Execute `RegisterAdultTicket` rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "Matteo", list.get(0) );
list.clear();

// One box office is closed, the other is open:
office1.setOpen(false);
boxOffices.update(officeFH1, office1);
persons.insert(new Person("Mark", 35));
executor.run(BoxOfficeUnit.class);

```



```

assertEquals( 1, list.size() );
assertEquals( "Mark", list.get(0) );
list.clear();

// All box offices are closed:
office2.setOpen(false);
boxOffices.update(officeFH2, office2); // Guarding rule is no longer true.
persons.insert(new Person("Edson", 35));
executor.run(BoxOfficeUnit.class); // No execution

assertEquals( 0, list.size() );

```

2.12.3. ルールユニットのアイデンティティの競合

保護されたルールユニットを使用したルール実行のシナリオでは、1つのルールが複数のルールユニットを保護することができます。同時に、複数のルールが1つのルールユニットを保護してから有効化することもできます。このような2通りの保護シナリオでは、ルールユニットには、アイデンティティの競合を避けるための明確に定義されたアイデンティティが必要です。

デフォルトでは、ルールユニットのアイデンティティはルールユニットクラス名で、**RuleUnitExecutor** によりシングルトンクラスとして処理されます。この識別動作は、**RuleUnit** インターフェースの **getUnitIdentity()** のデフォルトメソッドにエンコードされています。

RuleUnit インターフェースのデフォルトのアイデンティティメソッド

```

default Identity getUnitIdentity() {
    return new Identity( getClass() );
}

```

場合によっては、ルールユニット間のアイデンティティの競合を避けるために、このデフォルトの識別動作をオーバーライドする必要があります。

たとえば、以下の **RuleUnit** クラスには、あらゆる種類のオブジェクトを許可する **DataSource** 定義が含まれています。

Unit0 ルールユニットクラスの例

```

public class Unit0 implements RuleUnit {
    private DataSource<Object> input;

    public DataSource<Object> getInput() {
        return input;
    }
}

```

このルールユニットには、2つの条件 (OOPath 表記) に基づいて別のルールユニットを保護する、以下の DRL ルールが含まれています。

ルールユニットの GuardAgeCheck DRL ルールの例

```

package org.mypackage.myunit
unit Unit0

```



```
rule GuardAgeCheck
  when
    $i: /input#Integer
    $s: /input#String
  then
    drools.guard( new AgeCheckUnit($i) );
    drools.guard( new AgeCheckUnit($s.length()) );
  end
```

保護された **AgeCheckUnit** ルールユニットは、一連の **persons** の年齢を検証します。**AgeCheckUnit** には、確認用の **persons** の **DataSource** の定義、検証用の **minAge** 変数、および結果を集計する **List** が含まれます。

AgeCheckUnit ルールユニットの例

```
public class AgeCheckUnit implements RuleUnit {
    private final int minAge;
    private DataSource<Person> persons;
    private List<String> results;

    public AgeCheckUnit( int minAge ) {
        this.minAge = minAge;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public int getMinAge() {
        return minAge;
    }

    public List<String> getResults() {
        return results;
    }
}
```

AgeCheckUnit ルールユニットには、データソースの **persons** の検証を実行する以下の DRL ルールが含まれます。

ルールユニットの CheckAge DRL ルールの例

```
package org.mypackage.myunit
unit AgeCheckUnit

rule CheckAge
  when
    $p : /persons{ age > minAge }
  then
    results.add($p.getName() + ">" + minAge);
  end
```

この例では、**RuleUnitExecutor** クラスを作成し、これらの2つのルールユニットが含まれる KIE ベースにクラスをバインドして、同じルールユニットの **DataSource** 定義を2つ作成します。

executor 定義とデータソース定義の例

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

DataSource<Object> input = executor.newDataSource( "input" );
DataSource<Person> persons = executor.newDataSource( "persons",
    new Person( "John", 42 ),
    new Person( "Sally", 4 ) );

List<String> results = new ArrayList<>();
executor.bindVariable( "results", results );
```

一部のオブジェクトを入力データソースに挿入し、**Unit0** ルールユニットを実行できるようになりました。

挿入されたオブジェクトを使用したルールユニット実行の例

```
ds.insert("test");
ds.insert(3);
ds.insert(4);
executor.run(Unit0.class);
```

実行結果一覧の例

```
[Sally>3, John>3]
```

この例では、**AgeCheckUnit** という名前のルールユニットはシングルトンクラスと見なされ、1回のみ実行されます。この時、**minAge** 変数は **3** に設定されます。入力データソースに挿入された文字列 **"test"** および整数 **4** の両方は、**minAge** 変数が **4** に設定された 2 回目の実行をトリガーする可能性もあります。しかし、同じアイデンティティを持つ別のルールユニットがすでに評価されているため、2 回目の実行はありません。

このルールユニットのアイデンティティの競合を解決するには、**AgeCheckUnit** クラスの **getUnitIdentity()** メソッドをオーバーライドして、ルールユニットアイデンティティに **minAge** 変数も含めます。

getUnitIdentity() メソッドをオーバーライドする変更された AgeCheckUnit ルールユニット

```
public class AgeCheckUnit implements RuleUnit {

    ...

    @Override
    public Identity getUnitIdentity() {
        return new Identity(getClass(), minAge);
    }
}
```

このオーバーライドにより、以前のルールユニットの実行例は、以下の出力を生成します。

変更したルールユニットの実行結果一覧の例

```
[John>4, Sally>3, John>3]
```

minAge が **3** と **4** に設定されたルールユニットは、2つの異なるルールユニットと見なされるようになり、両方とも実行されます。

第3章 データオブジェクト

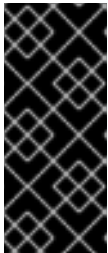
データオブジェクトは、作成するルールアセットの構成要素です。データオブジェクトは、プロジェクトで指定したパッケージに Java オブジェクトとして実装されているカスタムのデータ型です。たとえば、データフィールド **Name**、**Address**、および **DateOfBirth** を使用して **Person** オブジェクトを作成し、ローン申し込みルールに詳細な個人情報を指定できます。このカスタムのデータ型は、アセットとデシジョンサービスがどのデータに基づいているかを指定します。

3.1. データオブジェクトの作成

以下の手順は、データオブジェクトを作成する際の一般的な概要で、特定のビジネスアセットに固有のものではありません。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset → Data Object** をクリックします。
3. 一意の **データオブジェクト** 名を入力し、**パッケージ** を選択します。これにより、その他のルールアセットでもデータオブジェクトを利用できるようになります。同じパッケージに、同じ名前のデータオブジェクトを複数作成することはできません。指定の DRL ファイルで、どのパッケージからでもデータオブジェクトをインポートできます。



別のパッケージからのデータオブジェクトのインポート

別のパッケージから直接、ガイド付きルールやガイド付きデシジョンテーブルデザイナーなどのアセットデザイナーに、既存のデータオブジェクトをインポートすることができます。プロジェクトに関連するルールアセットを選択し、アセットデザイナーで **Data Objects → New item** に移動して、インポートするオブジェクトを選択します。

4. データオブジェクトを永続化するには、**Persistable** チェックボックスを選択します。永続型データオブジェクトは、JPA 仕様に準じてデータベースに保存できます。デフォルトの JPA は Hibernate です。
5. **OK** をクリックします。
6. データオブジェクトデザイナーで **add field** をクリックして、**Id** 属性、**Label** 属性、**Type** 属性を使用するオブジェクトにフィールドを追加します。必須属性にはアスタリスク (*) マークが付いています。
 - **Id**: フィールドの一意の ID を入力します。
 - **Label**: (任意) フィールドのラベルを入力します。
 - **Type**: フィールドのデータタイプ\を入力します。
 - **List**: (任意) このチェックボックスを選択すると、このフィールドで、指定したタイプのアイテムを複数保持できるようになります。

図3.1 データオブジェクトへのデータフィールドの追加

New Field

×

Id *

salary

Label

Salary

Type *

BigInteger

▼

List ⓘ

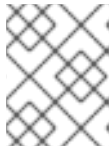
☐

Cancel

Create

Create and continue

7. **Create** をクリックして新規フィールドを追加します。**Create and continue** をクリックすると、新しいフィールドが追加され、別のフィールドを引き続き追加できます。



注記

フィールドを編集するには、フィールド行を選択し、画面右側の **general properties** を使用します。

第4章 BUSINESS CENTRAL における DRL ルールの作成

Business Central で、プロジェクトに対して DRL ルールを作成して管理できます。パッケージに作成またはインポートするデータオブジェクトに基づいて、各 DRL ファイルで、ルールの条件、アクション、そしてルールに関連するその他のコンポーネントを定義します。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. **Add Asset → DRL file** をクリックします。
3. 参考となる **DRL ファイル** 名を入力し、適切な **パッケージ** を選択します。指定するパッケージは、必要なデータオブジェクトが割り当てられているか、またはこれから割り当てるパッケージにする必要があります。
ドメイン固有言語 (DSL) アセットがプロジェクトに定義されている場合は、**Show declared DSL sentences** を選択することもできます。これらの DSL アセットは、DRL デザイナーで定義する条件およびアクションに使用できるオブジェクトです。
4. **OK** をクリックして、ルールアセットを作成します。
新しい DRL ファイルが、**Project Explorer** の **DRL** パネルに追加されます。**Show declared DSL sentences** オプションを選択した場合は、**DSL** パネルに追加されます。この DRL ファイルを割り当てたパッケージは、ファイルの上部にリストされます。
5. DRL デザイナーの左パネルの **Fact types** リストで、ルールに必要なすべてのデータオブジェクトとデータオブジェクトフィールドがリストされていることを確認します (それぞれを展開します)。リストされていない場合は、DRL ファイルの **import** ステートメントを使用してその他のパッケージから関連するデータオブジェクトをインポートするか、またはパッケージに **データオブジェクトを作成** します。
6. データオブジェクトをすべて配置したら、DRL デザイナーの **Model** タブに戻り、以下のいずれかのコンポーネントで DRL ファイルを定義します。

DRL ファイル内のコンポーネント

```
package

import

function // Optional

query // Optional

declare // Optional

global // Optional

rule "rule name"
  // Attributes
  when
    // Conditions
  then
    // Actions
end
```

```
rule "rule2 name"
```

```
...
```

- **package:** (自動) これは、DRL ファイルを作成し、パッケージを選択すると定義されます。
- **import:** このパッケージ、または DRL ファイルで使用する別のパッケージのデータオブジェクトを指定します。パッケージとデータオブジェクトは **packageName.objectName** の形式で指定し、複数のインポートは別々の行に指定します。

データオブジェクトのインポート

```
import org.mortgages.LoanApplication;
```

- **function:** (任意) DRL ファイルのルールが使用する関数を指定します。DRL ファイルの関数は、Java クラスにではなくルールのソースファイルにセマンティックコードを追加します。関数は、特に、ルールのアクション (**then**) 部分が繰り返し使用され、パラメーターだけがルールごとに異なる場合に便利です。DRL ファイルのルールで、関数を宣言したり、静的メソッドを関数としてインポートしたりして、ルールの アクション (**then**) 部分に、名前を指定して関数を使用します。

ルールに関数を宣言して使用 (オプション 1)

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

rule "Using a function"
when
    // Empty
then
    System.out.println( hello( "James" ) );
end
```

ルールに関数をインポートして使用 (オプション 2)

```
import function my.package.applicant.hello;

rule "Using a function"
when
    // Empty
then
    System.out.println( hello( "James" ) );
end
```

- **query:** (オプション) DRL ファイル内のルールに関連するファクトのデシジョンエンジンのワーキングメモリーを検索します。DRL ファイルにクエリー定義を追加してから、アプリケーションコードで合致する結果を取得します。クエリーは、定義した条件セットを検索するため、**when** または **then** を指定する必要はありません。クエリー名は KIE ベースでグローバルとなるため、プロジェクトにあるその他のすべてのルールクエリーと重複しないようにする必要があります。クエリーの結果に戻るには、**ksession.getQueryResults("name")** を使用して従来の、**QueryResults** 定義を構成し

ます ("**name**" はクエリー名)。これにより、クエリーの結果が返り、クエリーに一致したオブジェクトを取得できるようになります。DRL ファイルのルールに、クエリーと、クエリー結果パラメーターを定義します。

DRL ファイルにおけるクエリー定義の例

```
query "people under the age of 21"
    $person : Person( age < 21 )
end
```

クエリー結果を取得するためのアプリケーションコードの例

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age of 21" );
```

- **declare:** (任意) DRL ファイルのルールが使用する新しいファクトタイプを宣言します。Red Hat Process Automation Manager の **java.lang** パッケージのデフォルトは **Object** ですが、必要に応じて DRL ファイルに別のタイプを宣言することもできます。DRL ファイルにファクトタイプを宣言すると、Java などの低級言語でモデルを作成せず、デシジョンエンジンに直接新しいファクトモデルを定義できるようになります。

新しいファクトタイプの宣言および使用

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

rule "Using a declared type"
    when
        $p : Person( name == "James" )
    then // Insert Mark, who is a customer of James.
        Person mark = new Person();
        mark.setName( "Mark" );
        insert( mark );
    end
```

- **global:** (オプション) DRL ファイルのルールで使用するグローバル変数を組み込みます。グローバル変数は通常、ルールの結果で使用するアプリケーションサービスなど、ルールのデータやサービスを提供し、ルールの結果で追加されるログや値など、ルールからのデータを返します。KIE セッション設定や REST 操作を使用してデシジョンエンジンのワーキングメモリーにグローバル値を設定し、DRL ファイルのルールの上にグローバル変数を宣言してから、これをルールのアクション部分 (**then**) で使用します。グローバル変数が複数ある場合には、DRL ファイルで別々の行を使用してください。

デシジョンエンジンに対するグローバルリストの設定

```
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

ルールでのグローバルリストの定義

■


```
global java.util.List myGlobalList;

rule "Using a global"
when
    // Empty
then
    myGlobalList.add( "My global list" );
end
```



警告

グローバル変数に定数イミュータブル値がない場合には、ルールの条件設定にグローバル変数を使用しないでください。グローバル変数はデシジョンエンジンのワーキングメモリーに挿入されないため、デシジョンエンジンでは変数の値の変更を追跡できません。

グローバル変数を使用してルール間でデータを共有しないでください。ルールは常に、ワーキングメモリーの状態に関して推論し、これに対応するので、ルールからルールにデータを渡す必要がある場合には、データをファクトとしてデシジョンエンジンのワーキングメモリーにアサートしてください。

- **rule:** DRL ファイルで各ルールを定義します。ルールは、**rule "name"** 形式のルール名に、ルールの動作 (**salience**、**no-loop** など) を定義する任意の属性、**when** および **then** 定義が続きます。ルールのパッケージ内の各ルールには一意の名前が必要です。ルールの **when** 部分には、アクションを実行するために満たす必要のある条件が含まれます。たとえば、銀行が、ローンの申し込みを 21 歳以上に限定した場合、**"Underage"** ルールの **when** 条件は **Applicant(age < 21)** になります。ルールの **then** 部分には、ルールの条件部分が満たされる場合に実行するアクションが含まれます。たとえば、ローンの申請者が 21 歳に満たない場合、**then** アクションは **setApproved(false)** になり、申込者が年齢条件を満たしていないためにローンの申し込みは承認されません。

申込者の年齢制限に関するルール

```
rule "Underage"
    salience 15
    when
        $application : LoanApplication()
        Applicant( age < 21 )
    then
        $application.setApproved( false );
        $application.setExplanation( "Underage" );
    end
```

少なくとも、各 DRL ファイルは **package** コンポーネント、**import** コンポーネント、**rule** コンポーネントを指定する必要があります。他のすべてのコンポーネントは任意です。

以下は、ローン申し込みのデシジョンサービスの DRL ファイルの例です。

ローン申し込みの DRL ファイルの例

```

package org.mortgages;

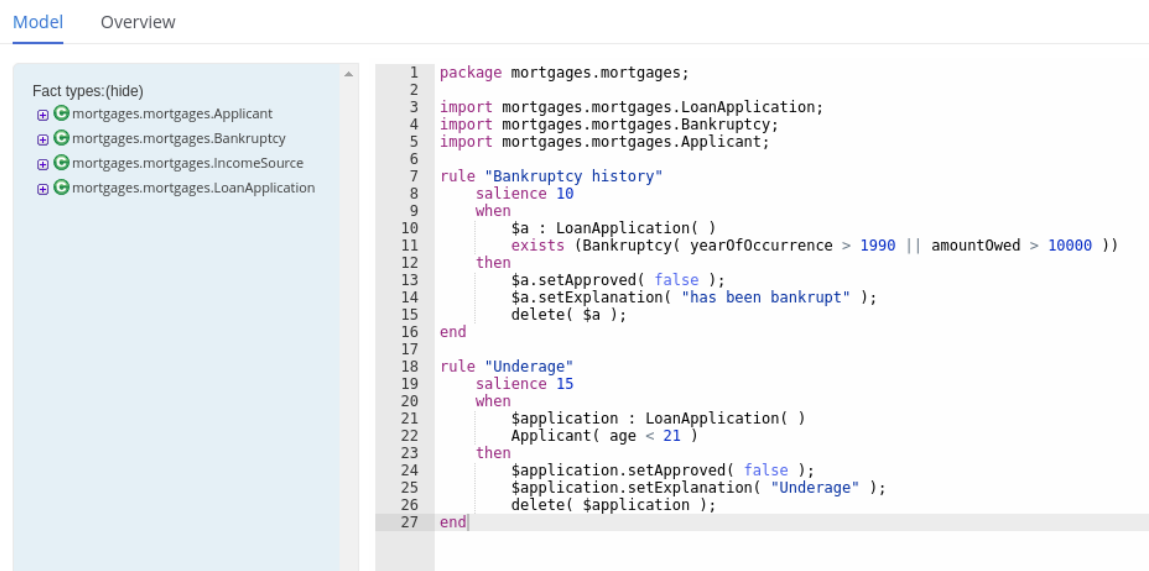
import org.mortgages.LoanApplication;
import org.mortgages.Bankruptcy;
import org.mortgages.Applicant;

rule "Bankruptcy history"
  salience 10
  when
    $a : LoanApplication()
    exists (Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000 ))
  then
    $a.setApproved( false );
    $a.setExplanation( "has been bankrupt" );
    delete( $a );
  end

rule "Underage"
  salience 15
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
    delete( $application );
  end

```

図4.1 Business Central のルール申し込み用の DRL ファイルの例



7. ルールのすべてのコンポーネントを定義したら、DRL デザイナーの右上ツールバーで **Validate** をクリックし、DRL ファイルを検証します。検証に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまでファイルを検証します。

8. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

4.1. DRL ルールへの WHEN 条件の追加

ルールの **when** 部分には、アクションを実行するのに必要な条件が含まれます。たとえば、銀行のローン申し込みに年齢制限 (21 歳以上) を満たす必要がある場合、**"Underage"** ルールの **when** 条件は **Applicant(age < 21)** となります。条件は、パッケージで利用可能なデータオブジェクトに基づいて、指定した一連のパターンおよび制約と、任意のバインディングその他のサポートされる DRL 要素で構成されます。

前提条件

- **package** は DRL ファイルに上部に定義されます。これはファイルの作成時に実行されているはずです。
- ルールで使ったデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージか、または別の Business Central のパッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に **rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作を定義する任意のルール属性 (**salience**、**no-loop** など) は、ルール名の下 **when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールに **when** を入力して、条件ステートメントを追加します。**when** セクションは、ルールの条件を定義するファクトパターンで構成されますが、ファクトパターンが1つも追加されない場合もあります。

when セクションを空にすると、条件は true であると見なされ、デシジョンエンジンで **fireAllRules()** 呼び出しが最初の実施された場合に、**then** セクションのアクションが実行されます。これは、デシジョンエンジンの状態を設定するルールを使用する場合に便利です。

条件のないルール例

```
rule "Always insert applicant"
  when
    // Empty
  then // Actions to be executed once
    insert( new Applicant() );
  end

// The rule is internally rewritten in the following way:

rule "Always insert applicant"
  when
    eval( true )
  then
    insert( new Applicant() );
  end
```

2. 一致させる最初の条件のパターンを入力し、任意で制約、バインディング、およびサポートされる DRL 要素を入力します。基本的なパターンフォーマットは **<patternBinding> : <patternType> (<constraints>)** です。パターンは、パッケージで利用可能なデータオブジェクトに基づいており、**then** セクションのアクションを発生させるのに必要な条件を定義します。
 - **単純なパターン:** 制約のない単純なパターンは、指定したタイプのファクトに一致します。たとえば、次は、申込者が存在することだけが条件になります。

-

```
when
  Applicant()
```

- **制約のあるパターン:** 制約を持つパターンは、指定したタイプのファクトと、追加制限を括弧で指定したパターン (true または false) に一致します。たとえば、次は、申込者が 21 歳に満たないことを条件としています。

```
when
  Applicant( age < 21 )
```

- **バインディングのあるパターン:** パターンのバインディングは簡単な参照となり、ルールのその他のコンポーネントが、定義したパターンに戻って参照します。たとえば、次の例では、**LoanApplication** のバインディング **a** が、underage の申込者に関連するアクションとして使用されます。

```
when
  $a : LoanApplication()
  Applicant( age < 21 )
then
  $a.setApproved( false );
  $a.setExplanation( "Underage" )
```

3. このルールに適用するすべての条件パターンの定義を継続します。以下は、DRL 条件を定義するいくつかのキーワードオプションです。

- **and:** 条件コンポーネントを論理積に分類します。接中辞および接頭辞の **and** がサポートされます。デフォルトでは、結合演算子を指定しないと、リストされているパターンがすべて **and** で結合されます。

```
// All of the following examples are interpreted the same way:
$a : LoanApplication() and Applicant( age < 21 )

$a : LoanApplication()
and Applicant( age < 21 )

$a : LoanApplication()
Applicant( age < 21 )

(and $a : LoanApplication() Applicant( age < 21 ))
```

- **or:** 条件コンポーネントを論理和に分類します。接中辞および接頭辞の **or** がサポートされます。

```
// All of the following examples are interpreted the same way:
Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount == 20000 )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )

(or Bankruptcy( amountOwed == 100000 ) IncomeSource( amount == 20000 ))
```

- **exists:** 存在している必要のあるファクトおよび制約を指定します。このオプションは、初回の一致についてのみトリガーされ、後続の一致については無視されます。この要素を複数のパターンで使用する場合は、これらのパターンを括弧 () で囲みます。

```
exists ( Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000 ) )
```

- **not:** 存在するべきでないファクトおよび制約を指定します。

```
not ( Applicant( age < 21 ) )
```

- **forall:** 最初のパターンに一致するすべてのファクトが残りのすべてのパターンに一致するかどうかを検証します。**forall** 構成が満たされると、このルールが **true** と評価されます。

```
forall( $app : Applicant( age < 21 )
        Applicant( this == $app, status = 'underage' ) )
```

- **from:** パターンのデータソースを指定します。

```
Applicant( ApplicantAddress : address )
Address( zipcode == "23920W" ) from ApplicantAddress
```

- **entry-point:** パターンのデータソースに対応する エントリーポイント を定義します。通常は **from** と共に使用されます。

```
Applicant() from entry-point "LoanApplication"
```

- **collect:** ルールを条件の一部として使用できる、オブジェクトのコレクションを定義します。この例では、指定したそれぞれのローンについてデシジョンエンジンで保留されているすべての申し込みが **List** に分類されます。保留中の申し込みが3つ以上ある場合は、このルールが実行されます。

```
$m : Mortgage()
$a : List( size >= 3 )
from collect( LoanApplication( Mortgage == $m, status == 'pending' ) )
```

- **accumulate:** オブジェクトのコレクションを処理し、各要素のカスタムアクションを実行し、(制約が **true** と評価されると) 結果オブジェクトを1つ以上返します。このオプションは、**collect** よりも強力で、柔軟性が高いオプションです。**accumulate(<source pattern>; <functions> [;<constraints>])** 形式を使用します。この例では、**min**、**max**、および **average** は累積関数で、各センサーのすべての測定値から、最低気温、最高気温、平均気温の値を計算します。その他のサポートされる関数には、**count**、**sum**、**variance**、**standardDeviation**、**collectList**、および **collectSet** があります。

```
$s : Sensor()
accumulate( Reading( sensor == $s, $temp : temperature );
            $min : min( $temp ),
            $max : max( $temp ),
            $avg : average( $temp );
            $min < 20, $avg > 70 )
```



注記

DRL ルール条件の詳細については、「[DRL のルール条件 \(WHEN\)](#)」を参照してください。

4. ルールの条件コンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイルの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまで再度、ファイルを検証します。
5. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

4.2. DRL ルールへの THEN アクションの追加

ルールの **then** 部分には、ルールの条件部分が満たされる場合に実行するアクションが含まれます。たとえば、ローンの申請者が 21 歳に満たない場合は、**"Underage"** ルールの **then** アクションが **setApproved(false)** となり、年齢が基準に達していないためローンの申し込みは承認されません。アクションは、ルールの条件とパッケージで利用可能オブジェクトに基づいて結果を実行する 1 つ以上のメソッドで構成されます。ルールアクションの主な目的は、デシジョンエンジンのワーキングメモリーでデータの挿入、削除、または変更を行うことです。

前提条件

- **package** は DRL ファイルに上部に定義されます。これはファイルの作成時に実行されているはずです。
- ルールで使ったデータオブジェクトの **import** リストが、DRL ファイルの **package** 行の下に定義されます。データオブジェクトは、このパッケージか、または別の Business Central のパッケージから使用できます。
- **rule** 名は、**package**、**import**、または DRL ファイル全体に適用されるその他の行の下に **rule "name"** という形式で定義されます。同じパッケージでルール名を重複させることはできません。ルールの動作を定義する任意のルール属性 (**salience**、**no-loop** など) は、ルール名の下に **when** セクションの前に定義します。

手順

1. DRL デザイナーで、ルールの **when** セクションの後に **then** を入力して、アクションステートメントを追加します。
2. ルールの条件に基づいて、ファクトパターンに対して実行するアクションを 1 つ以上入力します。
以下は、DRL アクションを定義するためのキーワードオプションの例です。

- **set**: これを使用してフィールドの値を設定します。

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

- **modify**: ファクトの変更するフィールドを指定し、デシジョンエンジンに変更を通知します。このメソッドを使用することで、ファクトの更新に対する構造化されたアプローチを提供します。このメソッドは、**update** 操作とオブジェクトフィールドを変更する setter 呼び出しを組み合わせたものです。

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

- **update**: フィールドと、更新される関連ファクト全体を指定して、その変更をデシジョンエ

ンジンに通知します。ファクトが変更されたら、更新された値の影響を受ける可能性がある別のファクトを変更する前に **update** を呼び出す必要があります。この追加設定を回避するには、代わりに **modify** メソッドを使用します。

```
LoanApplication.setAmount( 100 );
update( LoanApplication );
```

- **insert: new** ファクトをデシジョンエンジンに挿入します。

```
insert( new Applicant() );
```

- **insertLogical**: デシジョンエンジンに **new** ファクトを論理挿入する場合に使用します。デシジョンエンジンはファクトの挿入、取り消しに対して論理的な決定を行う役割を担います。通常の挿入または記述による挿入の後には、ファクトは明示的に取り消される必要があります。論理挿入後にファクトを挿入したルールの条件が true でなくなった場合、挿入されたファクトは自動的に取り消されます。

```
insertLogical( new Applicant() );
```

- **delete**: デシジョンエンジンからオブジェクトを削除します。キーワード **retract** も DRL でサポートされており、同じアクションを実行しますが、DRL のコードでは、キーワード **insert** との整合性を考慮して **delete** が通常推奨されます。

```
delete( Applicant );
```



注記

DRL ルールアクションの詳細については、[「DRL におけるルールアクション \(THEN\)」](#) を参照してください。

3. ルールのアクションコンポーネントをすべて定義したら、DRL デザイナーの右上のツールバーの **Validate** をクリックして、DRL ファイルの妥当性を確認します。ファイルの妥当性確認に失敗したら、エラーメッセージに記載された問題に対応し、DRL ファイルの構文およびコンポーネントをすべて見直し、エラーが表示されなくなるまでファイルを検証します。
4. DRL デザイナーで **Save** をクリックして、設定した内容を保存します。

第5章 ルールの実行

ルールの例を特定するか、Business Central でルールを作成したら、関連のプロジェクトをビルドしてデプロイし、ローカルまたは KIE Server でルールを実行してテストできます。

前提条件

- Business Central および KIE Server がインストールされ、実行されている。インストールオプションは、『[Red Hat Process Automation Manager インストールの計画](#)』を参照してください。

手順

1. Business Central で、**Menu → Design → Projects** に移動して、プロジェクト名をクリックします。
2. プロジェクトの **Assets** ページの右上にある **Deploy** をクリックして、プロジェクトをビルドして KIE Server にデプロイします。ビルドに失敗したら、画面下部の **Alerts** パネルに記載されている問題に対処します。

プロジェクトデプロイメントのオプションに関する詳細は、『[Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ](#)』を参照してください。

注記

デフォルトでプロジェクト内のルールアセットが実行可能なルールモデルからビルドされていない場合には、以下の依存関係がプロジェクトの **pom.xml** ファイルに含まれているか確認して、プロジェクトを再構築してください。

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhpam.version}</version>
</dependency>
```

この依存関係は、デフォルトで Red Hat Process Automation Manager のルールアセットが実行可能なルールモデルからビルドされるようにするために必要です。Red Hat Process Automation Manager のコアパッケージに、この依存関係は同梱されていますが、Red Hat Process Automation Manager のアップグレード履歴によっては、この依存関係を手動で追加して、実行可能なルールモデルの動作を有効にする必要がある場合があります。

実行可能なルールモデルに関する詳細は、『[Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ](#)』を参照してください。

3. ローカルでのルール実行に使用するか、KIE Server でルールを実行するクライアントアプリケーションとして使用できるように、まだ作成されていない場合には、Business Central 外に Maven または Java プロジェクトを作成します。プロジェクトには、**pom.xml** ファイルと、プロジェクトリソースの実行に必要なその他のコンポーネントを含める必要があります。テストプロジェクトの例については、『[その他の DRL ルールの作成および実行方法](#)』を参照してください。
4. テストプロジェクトまたはクライアントアプリケーションの **pom.xml** ファイルを開き、以下の依存関係が追加されていない場合は追加します。

- **kie-ci**: クライアントアプリケーションで、**ReleasesId** を使用して、Business Central プロ

ジェクトデータをローカルにロードします。

- **kie-server-client**: クライアントアプリケーションで、KIE Server のアセットを使用してリモートに接続します。
- **slf4j**: (オプション) クライアントアプリケーションで、KIE Server に接続した後に、SLF4J (Simple Logging Facade for Java) を使用して、デバッグのログ情報を返します。

クライアントアプリケーションの **pom.xml** ファイルにおける、Red Hat Process Automation Manager 7.7 の依存関係の例

```
<!-- For local execution -->
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-ci</artifactId>
  <version>7.33.0.Final-redhat-00002</version>
</dependency>

<!-- For remote execution on KIE Server -->
<dependency>
  <groupId>org.kie.server</groupId>
  <artifactId>kie-server-client</artifactId>
  <version>7.33.0.Final-redhat-00002</version>
</dependency>

<!-- For debug logging (optional) -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-simple</artifactId>
  <version>1.7.25</version>
</dependency>
```

このアーティファクトで利用可能なバージョンについては、オンラインの [Nexus Repository Manager](#) でグループ ID とアーティファクト ID を検索してください。

注記

個別の依存関係に対して Red Hat Process Automation Manager **<version>** を指定するのではなく、Red Hat Business Automation 部品表 (BOM) の依存関係をプロジェクトの **pom.xml** ファイルに追加することを検討してください。Red Hat Business Automation BOM は、Red Hat Process Decision Manager と Red Hat Process Automation Manager の両方に適用します。BOM ファイルを追加すると、指定の Maven リポジトリからの一時的な依存関係の内、正しいバージョンが、このプロジェクトに追加されます。

BOM 依存関係の例:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.7.0.redhat-00002</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

Red Hat Business Automation BOM (Bill of Materials) についての詳細情報は、「[What is the mapping between Red Hat Process Automation Manager and the Maven library version?](#)」を参照してください。

5. モジュールクラスを含むアーティファクトの依存関係が、クライアントアプリケーションの **pom.xml** ファイルに定義されていて、デプロイしたプロジェクトの **pom.xml** ファイルに記載されているのと同じであることを確認します。モデルクラスの依存関係が、クライアントアプリケーションとプロジェクトで異なると、実行エラーが発生します。
Business Central でプロジェクトの **pom.xml** ファイルを利用するには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

たとえば、以下の **Person** クラスの依存関係は、クライアントと、デプロイしたプロジェクトの **pom.xml** ファイル両方に表示されます。

```
<dependency>
  <groupId>com.sample</groupId>
  <artifactId>Person</artifactId>
  <version>1.0.0</version>
</dependency>
```

6. デバッグ向けロギングを行うために、**slf4j** 依存関係を、クライアントアプリケーションの **pom.xml** ファイルに追加した場合は、関連するクラスパス (Maven の **src/main/resources/META-INF** 内など) に **simplelogger.properties** ファイルを作成し、以下の内容を記載します。

```
org.slf4j.simpleLogger.defaultLogLevel=debug
```

7. クライアントアプリケーションに、必要なインポートを含む **.java** メインクラスと、KIE ベースをロードする **main()** メソッドを作成し、ファクトを挿入し、ルールを実行します。
たとえば、プロジェクトの **Person** オブジェクトには、名、姓、時給、賃金を設定および取得する getter メソッドおよび setter メソッドが含まれます。プロジェクトにある以下の **Wage** ルールでは、賃金と時給を計算し、その結果に基づいてメッセージを表示します。

```

package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
  end

```

(必要に応じて) KIE Server の外でローカルにこのルールをテストするには、**.java** クラスで、KIE サービス、KIE コンテナ、および KIE セッションをインポートするように設定し、その後、**main()** メソッドを使用して、定義したファクトモデルに対してすべてのルールを実行するようにします。

ローカルでのルールの実行

```

import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseIdImpl("com.myspace", "MyProject", "1.0.0");

      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.newKieContainer(rid);
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);

      // Fire all rules:
      kSession.fireAllRules();
      kSession.dispose();
    }
  }
}

```

```

    catch (Throwable t) {
        t.printStackTrace();
    }
}
}

```

KIE Server でこのルールをテストするには、ローカルの例と同じように、インポートとルール実行情報で **.java** クラスを設定し、KIE サービス設定および KIE サービスクライアントの詳細を指定します。

KIE Server でのルールの実行

```

package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

    private static final String containerName = "testProject";
    private static final String sessionName = "myStatelessSession";

    public static final void main(String[] args) {
        try {
            // Define KIE services configuration and client:
            Set<Class<?>> allClasses = new HashSet<Class<?>>();
            allClasses.add(Person.class);
            String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
            String username = "$USERNAME";
            String password = "$PASSWORD";
            KieServicesConfiguration config =
                KieServicesFactory.newRestConfiguration(serverUrl,
                                                         username,
                                                         password);
            config.setMarshallingFormat(MarshallingFormat.JAXB);
            config.addExtraClasses(allClasses);
            KieServicesClient kieServicesClient =
                KieServicesFactory.newKieServicesClient(config);

```

```

// Set up the fact model:
Person p = new Person();
p.setWage(12);
p.setFirstName("Tom");
p.setLastName("Summers");
p.setHourlyRate(10);

// Insert Person into the session:
KieCommands kieCommands = KieServices.Factory.get().getCommands();
List<Command> commandList = new ArrayList<Command>();
commandList.add(kieCommands.newInsert(p, "personReturnId"));

// Fire all rules:
commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
BatchExecutionCommand batch = kieCommands.newBatchExecution(commandList,
sessionName);

// Use rule services client to send request:
RuleServicesClient ruleClient =
kieServicesClient.getServicesClient(RuleServicesClient.class);
ServiceResponse<ExecutionResults> executeResponse =
ruleClient.executeCommandsWithResults(containerName, batch);
System.out.println("number of fired rules:" +
executeResponse.getResult().getValue("numberOfFiredRules"));
}

catch (Throwable t) {
t.printStackTrace();
}
}
}

```

8. 設定した **.java** クラスをプロジェクトディレクトリーから実行します。(Red Hat CodeReady Studio などの) 開発プラットフォーム、またはコマンドラインでファイルを実行できます。(プロジェクトディレクトリー内の) Maven の実行例:

```
mvn clean install exec:java -Dexec.mainClass="com.sample.app.RulesTest"
```

(プロジェクトディレクトリー内の) Java の実行例

```
javac -classpath ".*$DEPENDENCIES/*:." RulesTest.java
java -classpath ".*$DEPENDENCIES/*:." RulesTest
```

9. コマンドラインおよびサーバーログで、ルール実行のステータスを確認します。ルールが期待通りに実行しない場合は、プロジェクトに設定したルールと、メインのクラス設定を確認して、提供されるデータを検証します。

第6章 その他の DRL ルールの作成および実行方法

Business Central インターフェースで DRL ルールを作成し、管理する代わりに、Red Hat CodeReady Studio やその他の統合開発環境 (IDE) を使用して、Maven または Java プロジェクトの一部として DRL ルールファイルを作成できます。こうしたスタンドアロンプロジェクトは、ナレッジ JAR (KJAR) 依存関係として、Business Central の既存の Red Hat Process Automation Manager プロジェクトに統合できます。スタンドアロンプロジェクトの DRL ファイルには、少なくとも必要な **package** 仕様、**import** リスト、および **rule** 定義が含まれる必要があります。グローバル変数や関数など、その他の DRL コンポーネントは任意です。DRL ルールに関連するすべてのデータオブジェクトは、スタンドアロンの DRL プロジェクトまたはデプロイメントに含まれる必要があります。

Maven または Java プロジェクトで実行可能なルールモデルを使用して、ビルド時に実行するルールセットの Java ベース表記を提供します。実行可能モデルは Red Hat Process Automation Manager の標準アセットパッケージングの代わりとなるもので、より効率的です。KIE コンテナと KIE ベースの作成がより迅速にでき、DRL (Drools Rule Language) ファイルリストや他の Red Hat Process Automation Manager アセットが多い場合は、特に有効です。

6.1. RED HAT CODEREADY STUDIO での DRL ルールの作成および実行

Red Hat JBoss CodeReady Studio を使用して、ルールが含まれる DRL ファイルを作成し、Red Hat Process Automation Manager デシジョンサービスにファイルを統合します。DRL ルールを作成する方法は、デシジョンサービスに Red Hat CodeReady Studio を使用する場合は、同じワークフローを継続する場合に便利です。この方法を使用していない場合は、Red Hat Process Automation Manager の代わりに Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

前提条件

- [Red Hat カスタマーポータル](#) から Red Hat CodeReady Studio をインストールしている。

手順

1. Red Hat CodeReady Studio で、**File → New → Project** をクリックします。
2. 開いた **New Project** ウィンドウで、**Drools → Drools Project** を選択し、**Next** をクリックします。
3. **Create a project and populate it with some example files to help you get started quickly** の 2 番目のアイコンをクリックして、**Next** をクリックします。
4. **Project name** を入力し、プロジェクトのビルドオプションで **Maven** ラジオボタンを選択します。GAV 値が自動的に生成されます。必要に応じて、プロジェクトについてこの値を更新できます。
 - **Group ID: com.sample**
 - **Artifact ID: my-project**
 - **Version: 1.0.0-SNAPSHOT**
5. **Finish** をクリックしてプロジェクトを作成します。
この設定は、基本的なプロジェクト構造、クラスパス、サンプルルールを設定します。以下は、プロジェクト構造の概要です。

my-project

```

|-- src/main/java
|   |-- com.sample
|       |-- DecisionTableTest.java
|       |-- DroolsTest.java
|       |-- ProcessTest.java
|
|-- src/main/resources
|   |-- dtables
|       |-- Sample.xls
|   |-- process
|       |-- sample.bpmn
|   |-- rules
|       |-- Sample.drl
|       |-- META-INF
|
|-- JRE System Library
|
|-- Maven Dependencies
|
|-- Drools Library
|
|-- src
|
|-- target
|
|-- pom.xml

```

以下の要素に注目してください。

- **src/main/resources** ディレクトリーの **Sample.drl** ルールファイル。これには、サンプルの **Hello World** ルールおよび **GoodBye** ルールが含まれます。
- **com.sample** パッケージの **src/main/java** ディレクトリーにある **DroolsTest.java** ファイル。**Sample.drl** ルールの実行には、**DroolsTest** クラスを使用できます。
- 実行するのに必要な JAR ファイルを含むカスタムのクラスパスとして機能する **Drools Library** ディレクトリー。

既存の **Sample.drl** ファイルおよび **DroolsTest.java** ファイルを必要に応じて新しい設定に変更するか、ルールファイルをオブジェクトファイルを新たに作成します。この手順では、ルールと Java オブジェクトを新たに作成します。

6. ルールが機能する Java オブジェクトを作成します。

この例では、**my-project/src/main/java/com.sample** に **Person.java** ファイルが作成されます。**Person** クラスには、名、姓、時給、賃金を設定および取得する getter メソッドおよび setter メソッドが含まれます。

```

public class Person {
    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }
}

```

```

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

public Integer getHourlyRate() {
    return hourlyRate;
}

public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
}

public Integer getWage(){
    return wage;
}

public void setWage(Integer wage){
    this.wage = wage;
}
}

```

7. **File** → **Save** をクリックして、ファイルを保存します。

8. **my-project/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1つまたは複数の) ルールで使用するデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ1つ以上のルールが含まれます。

以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

```

package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello" + " " + name + " " + surname + "!");
        System.out.println("You are rich!");
    end

```

9. **File** → **Save** をクリックして、ファイルを保存します。

10. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。



注記

また、**DroolsTest.java** サンプルファイルと同様に、**main()** メソッドと **Person** クラスを1つの Java オブジェクトファイルに追加できます。

11. メインクラスに、KIE サービス、KIE コンテナ、および KIE セッションをインポートするのに必要な **import** 命令文を追加します。次に、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
この例では、必要なインポートと **main()** メソッドを使用して、**my-project/src/main/java/com.sample** に **RulesTest.java** ファイルを作成します。

```
package com.sample;

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {
    public static final void main(String[] args) {
        try {
            // Load the KIE base:
            KieServices ks = KieServices.Factory.get();
            KieContainer kContainer = ks.getKieClasspathContainer();
            KieSession kSession = kContainer.newKieSession();

            // Set up the fact model:
            Person p = new Person();
            p.setWage(12);
            p.setFirstName("Tom");
            p.setLastName("Summers");
            p.setHourlyRate(10);

            // Insert the person into the session:
            kSession.insert(p);

            // Fire all rules:
            kSession.fireAllRules();
            kSession.dispose();
        }

        catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

12. **File → Save** をクリックして、ファイルを保存します。
13. プロジェクトで DRL アセットをすべて作成して保存した後に、プロジェクトフォルダーを右クリックして、**Run As → Java Application** を選択してプロジェクトをビルドします。プロジェクトのビルドに失敗したら、CodeReady Studio の下部ウィンドウの **Problems** タブに記載されている問題に対応し、プロジェクトがビルドされるまでプロジェクトの検証を行います。



RUN AS → JAVA APPLICATION オプションが利用できない場合

プロジェクトを右クリックして、**Run As** を選択した場合に **Java Application** が選択肢にない場合は、**Run As → Run Configurations** に移動して **Java Application** を右クリックし、**New** をクリックします。次に、**Main** タブで、**Project** と、関連する **Main class** を参照して選択します。**Apply** をクリックし、**Run** をクリックしてプロジェクトをテストします。再度プロジェクトフォルダーを右クリックすると、**Java Application** オプションが表示されます。

Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジJAR (KJAR) として新規プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View → pom.xml** を選択します。

6.2. JAVA を使用した DRL ルールの作成および実行

Java オブジェクトを使用して、ルールが含まれる DRL ファイルを作成し、オブジェクトを Red Hat Process Automation Manager デシジョンサービスに統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Java オブジェクトを使用している場合や、同じワークフローを継続する場合に便利です。この方法を使用しなくなった場合は、Red Hat Process Automation Manager の Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. ルールが機能する Java オブジェクトを作成します。

この例では、**my-project** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、名前、苗字、時給、賃金を設定および取得するゲッターメソッドおよびセッターメソッドが含まれます。

```
public class Person {
    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Integer getHourlyRate() {
```

```

        return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
        this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
        return wage;
    }

    public void setWage(Integer wage){
        this.wage = wage;
    }
}

```

2. **my-project** ディレクトリーに、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定 (該当する場合) と、(1 つまたは複数の) ルールで使用されるデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ 1 つ以上のルールが含まれます。

以下の **Wage.drl** ファイルには、賃金と時給の値を計算し、その結果に基づいてメッセージを表示する **Wage** ルールが含まれます。

```

package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello" + " " + name + " " + surname + "!");
        System.out.println("You are rich!");
    end

```

3. メインクラスを作成し、Java オブジェクトを作成したディレクトリーに保存します。メインクラスは KIE ベースをロードし、ルールを実行します。
4. メインクラスに、KIE サービス、KIE コンテナ、および KIE セッションをインポートするのに必要な **import** 命令文を追加します。次に、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **main()** メソッドからルールを実行します。
この例では、必要なインポートと **main()** メソッドを使用して、**my-project** に **RulesTest.java** ファイルを作成します。

```

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {
    public static final void main(String[] args) {
        try {
            // Load the KIE base:

```

```

KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieSession kSession = kContainer.newKieSession();

// Set up the fact model:
Person p = new Person();
p.setWage(12);
p.setFirstName("Tom");
p.setLastName("Summers");
p.setHourlyRate(10);

// Insert the person into the session:
kSession.insert(p);

// Fire all rules:
kSession.fireAllRules();
kSession.dispose();
}

catch (Throwable t) {
    t.printStackTrace();
}
}
}

```

5. [Red Hat カスタマーポータル](#) から **Red Hat Process Automation Manager 7.7.0 Source Distribution** の ZIP ファイルをダウンロードし、**my-project/pam-engine-jars/** で展開します。
6. **my-project/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>

```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを1つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベースから KIE セッションを1つ以上作成することもできます。

以下の例では、より高度な **kmodule.xml** ファイルを示します。

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
    <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
        <ksession name="KSession1_1" type="stateful" default="true" />
        <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtsms" />
    </kbase>
    <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
        <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
            <fileLogger file="debugInfo" threaded="true" interval="10" />
        </ksession>
    </kbase>
</kmodule>

```

```

<workItemHandlers>
  <workItemHandler name="name" type="new org.domain.WorkItemHandler()" />
</workItemHandlers>
<listeners>
  <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener" />
  <agendaEventListener type="org.domain.FirstAgendaListener" />
  <agendaEventListener type="org.domain.SecondAgendaListener" />
  <processEventListener type="org.domain.ProcessListener" />
</listeners>
</ksession>
</kbase>
</kmodule>

```

この例は、KIE ベースを 2 つ定義します。KIE ベース **KBase1** から 2 つの KIE セッションをインスタンス化し、**KBase2** から 1 つの KIE セッションをインスタンス化します。**KBase2** の KIE セッションは、**ステートレス** な KIE セッションですが、これは 1 つ前の KIE セッションで呼び出されたデータ (1 つ前のセッションの状態) が、セッションの呼び出し間で破棄されることを示しています。ルールアセットの特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

7. Java オブジェクトですべての DRL アセットを作成して保存したあと、コマンドラインで **my-project** ディレクトリーに移動し、以下のコマンドを実行して Java ファイルをビルドします。**RulesTest.java** を、Java のメインクラスの名前に置き換えます。

```
javac -classpath "./pam-engine-jars/*:." RulesTest.java
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、エラーが表示されなくなるまで Java オブジェクトの妥当性確認を行います。

8. Java ファイルが問題なくビルトできたら、以下のコマンドを実行してローカルでルールを実行します。**RulesTest** を、Java のメインクラスの接頭辞に置き換えます。

```
java -classpath "./pam-engine-jars/*:." RulesTest
```

9. ルールを見直して、適切に実行したことを確認し、Java ファイルで必要な変更を加えます。

Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジ JAR (KJAR) として新規 Java プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

6.3. MAVEN を使用した DRL ルールの作成および実行

Maven アーキタイプを使用して、ルールを含めて DRL ファイルを作成し、アーキタイプを Red Hat Process Automation Manager デシジョンサービスに統合します。DRL ルールを作成する方法は、デシジョンサービスに外部 Maven アーキタイプを使用している場合や、同じワークフローを継続する場合に便利です。この方法を使用しなくなった場合は、Red Hat Process Automation Manager の Business Central インターフェースを使用して、DRL ファイルや、その他のルールアセットを作成することが推奨されます。

手順

1. Maven アーキタイプを作成するディレクトリーに移動して、次のコマンドを実行します。

```
mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

これにより、**my-app** という名前のディレクトリーが、以下の構造で作成されます。

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   |-- com
    |   |   |   |-- sample
    |   |   |   |   |-- app
    |   |   |   |   |   |-- App.java
    |   |-- test
    |   |   |-- java
    |   |   |   |-- com
    |   |   |   |   |-- sample
    |   |   |   |   |   |-- app
    |   |   |   |   |   |   |-- AppTest.java
```

my-app ディレクトリーには、以下の重要なコンポーネントが含まれます。

- アプリケーションソースを保存する **src/main** ディレクトリー
 - テストソースを保存する **src/test** ディレクトリー
 - プロジェクト設定を含む **pom.xml** ファイル
2. Maven アーキタイプに、ルールが機能する Java オブジェクトを作成します。
この例では、**my-app/src/main/java/com/sample/app** ディレクトリーに **Person.java** ファイルが作成されます。**Person** クラスには、名、姓、時給、賃金を設定し、取得する getter メソッドおよび setter メソッドが含まれます。

```
package com.sample.app;

public class Person {

    private String firstName;
    private String lastName;
    private Integer hourlyRate;
    private Integer wage;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
}
```

```

    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Integer getHourlyRate() {
        return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
        this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
        return wage;
    }

    public void setWage(Integer wage){
        this.wage = wage;
    }
}

```

3. **my-app/src/main/resources/rules** に、**.drl** 形式のルールファイルを作成します。DRL ファイルには、少なくともパッケージの指定と、(1つまたは複数の) ルールで使用するデータオブジェクトのインポートリストと、**when** 条件および **then** アクションを持つ1つ以上のルールが含まれます。

以下の **Wage.drl** ファイルには、**Person** クラスをインポートする **Wage** ルールが含まれ、賃金および時給の値を計算し、その結果に基づいてメッセージを表示します。

```

package com.sample.app;

import com.sample.app.Person;

dialect "java"

rule "Wage"
    when
        Person(hourlyRate * wage > 100)
        Person(name : firstName, surname : lastName)
    then
        System.out.println("Hello " + name + " " + surname + "!");
        System.out.println("You are rich!");
    end

```

4. **my-app/src/main/resources/META-INF** ディレクトリーに、以下の内容の **kmodule.xml** メタデータファイルを作成します。

```

<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>

```

この **kmodule.xml** ファイルは、KIE ベースへのリソースを選択し、セッションを設定する KIE モジュールの記述子です。このファイルを使用すると、KIE ベースを1つ以上定義して設定し、特定の KIE ベースの特定の **packages** から DRL ファイルを含めることができます。各 KIE ベー

スから KIE セッションを1つ以上作成することもできます。

以下の例では、より高度な **kmodule.xml** ファイルを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtsms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener" />
        <agendaEventListener type="org.domain.FirstAgendaListener" />
        <agendaEventListener type="org.domain.SecondAgendaListener" />
        <processEventListener type="org.domain.ProcessListener" />
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

この例は、KIE ベースを2つ定義します。KIE ベース **KBase1** から2つの KIE セッションをインスタンス化し、**KBase2** から1つの KIE セッションをインスタンス化します。**KBase2** の KIE セッションは、**ステートレス** な KIE セッションですが、これは1つ前の KIE セッションで呼び出されたデータ (1つ前のセッションの状態) が、セッションの呼び出し間で破棄されることを示しています。ルールアセットの特定の **packages** が両方の KIE ベースに含まれます。この方法でパッケージを指定した場合は、指定したパッケージを反映するフォルダー構造で DRL ファイルを整理する必要があります。

5. **my-app/pom.xml** 設定ファイルで、アプリケーションが要求するライブラリーを指定します。Red Hat Process Automation Manager の依存関係と、アプリケーションの **group ID**、**artifact ID**、および **version** (GAV) を提供します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0</version>
  <repositories>
    <repository>
      <id>jboss-ga-repository</id>
      <url>http://maven.repository.redhat.com/ga/</url>
    </repository>
```



```

</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>org.kie</groupId>
    <artifactId>kie-api</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Red Hat Process Automation Manager における Maven 依存関係および BOM (Bill of Materials) については、[「What is the mapping between Red Hat Process Automation Manager and Maven library version?」](#) を参照してください。

6. **my-app/src/test/java/com/sample/app/AppTest.java** の **testApp** メソッドを使用してルールをテストします。Maven によって、**AppTest.java** ファイルがデフォルトで作成されます。
7. **AppTest.java** ファイルで、KIE サービス、KIE コンテナ、および KIE セッションをインポートするのに必要な **import** ステートメントを追加します。次に、KIE ベースをロードし、ファクトを挿入し、ファクトモデルをルールに渡す **testApp()** メソッドからルールを実行します。

```

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public void testApp() {

    // Load the KIE base:
    KieServices ks = KieServices.Factory.get();
    KieContainer kContainer = ks.getKieClasspathContainer();
    KieSession kSession = kContainer.newKieSession();

    // Set up the fact model:
    Person p = new Person();
    p.setWage(12);
    p.setFirstName("Tom");
    p.setLastName("Summers");
    p.setHourlyRate(10);

    // Insert the person into the session:
    kSession.insert(p);

    // Fire all rules:

```

```
kSession.fireAllRules();  
kSession.dispose();  
}
```

8. Maven アーキタイプにすべての DRL アセットを作成して保存した後に、コマンドラインで **my-app** ディレクトリーに移動し、以下のコマンドを実行してファイルを作成します。

```
mvn clean install
```

ビルドに失敗したら、コマンドラインのエラーメッセージに記載されている問題に対応し、ビルドに成功するまでファイルの検証を行います。

9. ファイルが問題なくビルドできたら、以下のコマンドを実行してルールをローカルに実行します。 **com.sample.app** をパッケージ名に置き換えます。

```
mvn exec:java -Dexec.mainClass="com.sample.app"
```

10. ルールを見直して、適切に実行されたことを確認し、ファイルで必要な変更を加えます。

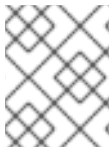
Red Hat Process Automation Manager で既存のプロジェクトと新しいルールアセットを統合するには、ナレッジ JAR (KJAR) として新規 Maven プロジェクトをコンパイルし、Business Central でプロジェクトの **pom.xml** ファイルに、依存関係としてこのプロジェクトを追加します。Business Central でプロジェクト **pom.xml** にアクセスするには、プロジェクトで既存のアセットを選択し、画面左側の **Project Explorer** メニューで **Customize View** ギアアイコンをクリックし、**Repository View** → **pom.xml** を選択します。

第7章 RED HAT PROCESS AUTOMATION MANAGER の IDE 向けのデシジョン例

Red Hat Process Automation Manager は、統合開発環境 (IDE: integrated development environment) にインポートできるように Java クラスとして配信される、デシジョン例を提供します。これらの例は、デシジョンエンジン機能をさらに理解するために使用する目的か、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

以下のデシジョンセットの例は、Red Hat Process Automation Manager で利用可能な例の一部です。

- **Hello World の例:** 基本的なルール実行や、デバッグ出力の使用方法を例示します。
- **状態の例:** ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。
- **フィボナッチの例:** ルールの顕著性を使用した再帰や競合解決を例示します。
- **銀行の例:** パターン一致、基本的なソート、計算を例示します。
- **ペットショップの例:** ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。
- **数独の例:** 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。
- **House of Doom の例:** 後向き連鎖と再帰を例示します。



注記

Red Hat Business Optimizer で提供される最適化の例については、『[Red Hat Business Optimizer のスタートガイド](#)』を参照してください。

7.1. IDE での RED HAT PROCESS AUTOMATION MANAGER のデシジョン例のインポートと実行

Red Hat Process Automation Manager のデシジョン例を統合開発環境 (IDE) にインポートして実行し、ルールとコードがどのように機能するかチェックできます。これらの例は、デシジョンエンジン機能をさらに理解するために使用するか、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

前提条件

- Java 8 以降をインストールしていること
- Maven 3.5.x 以降をインストールしていること
- Red Hat CodeReady Studio などの IDE がインストールされている。

手順

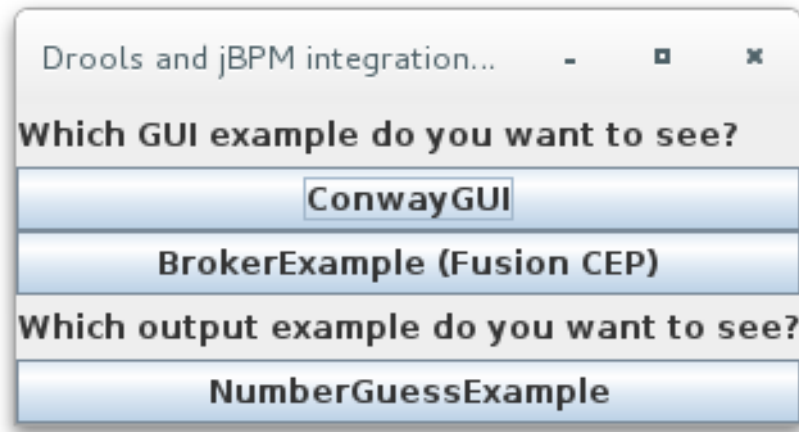
1. [Red Hat カスタマーポータル](#) から **Red Hat Process Automation Manager 7.7.0 Source Distribution** を **/rhpam-7.7.0-sources** などの一時的なディレクトリーにダウンロードして展開します。

2. IDE を開き、**File** → **Import** → **Maven** → **Existing Maven Projects** を選択するか、同等のオプションを選択して、Maven プロジェクトをインポートします。
3. **Browse** をクリックして、`~/rhpam-7.7.0-sources/src/drools-$VERSION/drools-examples` (または、Conway の Game of Life の例の場合は、`~/rhpam-7.7.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples`) に移動して、プロジェクトをインポートします。
4. 実行するパッケージ例に移動して、**main** メソッドが含まれる Java クラスを検索します。
5. Java クラスを右クリックし、**Run As** → **Java Application** を選択して例を実行します。
基本的なユーザーインターフェースですべての例を実行するには、**org.drools.examples** Main クラスの **DroolsExamplesApp.java** クラス (または Conway の Game of Life の場合は **DroolsJbpmIntegrationExamplesApp.java** クラス) を実行します。

図7.1 drools-examples (DroolsExamplesApp.java) 内のすべての例のインターフェース



図7.2 droolsjbpm-integration-examples (DroolsJbpmIntegrationExamplesApp.java) のすべての例のインターフェース



7.2. HELLO WORLD のデシジョン例 (基本ルールおよびデバッグ)

Hello World のデシジョンセットの例では、オブジェクトをデシジョンエンジンのワーキングメモリーに挿入する方法、ルールを使用してオブジェクトを照合する方法、エンジンの内部アクティビティを追跡するロギングの設定方法を例示します。

以下は、Hello World の例の概要です。

- 名前: **helloworld**
- Main クラス: (src/main/java 内の) **org.drools.examples.helloworld.HelloWorldExample**
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の) **org.drools.examples.helloworld.HelloWorld.drl**
- 目的: 基本的なルール実行とデバッグ出力の使用方法を例示します。

Hello World の例では、KIE セッションが生成されて、ルールの実行が可能になります。すべてのルールは、実行できるように KIE セッションが必要です。

ルール実行の KIE セッション

```
KieServices ks = KieServices.Factory.get(); ①
KieContainer kc = ks.getKieClasspathContainer(); ②
KieSession ksession = kc.newKieSession("HelloWorldKS"); ③
```

- ① **KieServices** ファクトリーを取得します。これは、アプリケーションがデシジョンエンジンとの対話に使用する主なインターフェースです。
- ② プロジェクトクラスパスから **KieContainer** を作成します。これで、/META-INF/kmodule.xml ファイルを検出し、このファイルをもとに設定して **KieModule** で **KieContainer** をインスタンス化します。
- ③ /META-INF/kmodule.xml ファイルに定義された **"HelloWorldKS"** KIE セッション設定をもとに **KieSession** を作成します。



注記

Red Hat Process Automation Manager プロジェクトのパッケージ化に関する詳細は、『[Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ](#)』を参照してください。

Red Hat Process Automation Manager には、内部エンジンアクティビティを公開するイベントモデルがあります。**DebugAgendaEventListener** と **DebugRuleRuntimeEventListener** のデフォルトのデバッグリスナー 2 つにより、デバッグイベント情報が **System.err** の出力に表示されます。**KieRuntimeLogger** では実行監査が提供され、その結果はグラフィックビューワーで確認できます。

リスナーと監査ロガーのデバッグ

```
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession,
    ".target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, ".target/helloworld",
    1000 );
```

ロガーは、**Agenda** と **RuleRuntime** リスナーにビルドされる特別な実装です。デシジョンエンジンが実行を終えると、**logger.close()** が呼び出されます。

この例では、**"Hello World"** というメッセージを含む **Message** オブジェクトを作成し、ステータス **HELLO** を **KieSession** に挿入して、**fireAllRules()** でルールを実行します。

データの挿入および実行

```
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

ルール実行は、データモデルを使用して、**KieSession** への出入力としてデータを渡します。この例のデータモデルには **message (String)** と **status (HELLO または GOODBYE)** の2つのフィールドが含まれます。

データモデルクラス

```
public static class Message {
    public static final int HELLO = 0;
    public static final int GOODBYE = 1;

    private String message;
```

```
private int      status;
...
}
```

この2つのルールは、**src/main/resources/org/drools/examples/helloworld/HelloWorld.drl** ファイルに配置されます。

"Hello World" ルールの **when** 条件では、ステータスが **Message.HELLO** の KIE セッションに、**Message** オブジェクトが挿入されるたびに、このルールを有効化すると記述しています。さらに、変数のバインドが2つ作成されます (**message** 変数を **message** 属性に、**m** 変数を一致する **Message** オブジェクト自体にバインド)。

ルールの **then** アクションは、バインドされた変数 **message** のコンテンツを **System.out** に出力するよう指定し、続いて **m** にバインドされている **Message** オブジェクトの **message** と **status** 属性値を変更します。このルールは **modify** ステートメントを使用して、1つのステートメントに割り当てブロックを適用し、ブロックの最後にデシジョンエンジンにこの変更について通知します。

"Hello World" rule

```
rule "Hello World"
when
  m : Message( status == Message.HELLO, message : message )
then
  System.out.println( message );
  modify ( m ) { message = "Goodbye cruel world",
                  status = Message.GOODBYE };
end
```

"Good Bye" ルールは、ステータスが **Message.GOODBYE** の **Message** オブジェクトと一致する点を除き、"Hello World" ルールによく似ています。

"Good Bye" rule

```
rule "Good Bye"
when
  Message( status == Message.GOODBYE, message : message )
then
  System.out.println( message );
end
```

この例を実行するには、**org.drools.examples.helloworld.HelloWorldExample** クラスを IDE で Java アプリケーションとして実行します。このルールは **System.out** に、デバッグリスナーは **System.err** に書き込み、監査ログは **target/helloworld.log** のログファイルを作成します。

IDE コンソールの System.out 出力

```
Hello World
Goodbye cruel world
```

IDE コンソールでの System.err の出力

```
==>[ActivationCreated(0): rule=Hello World;
      tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```

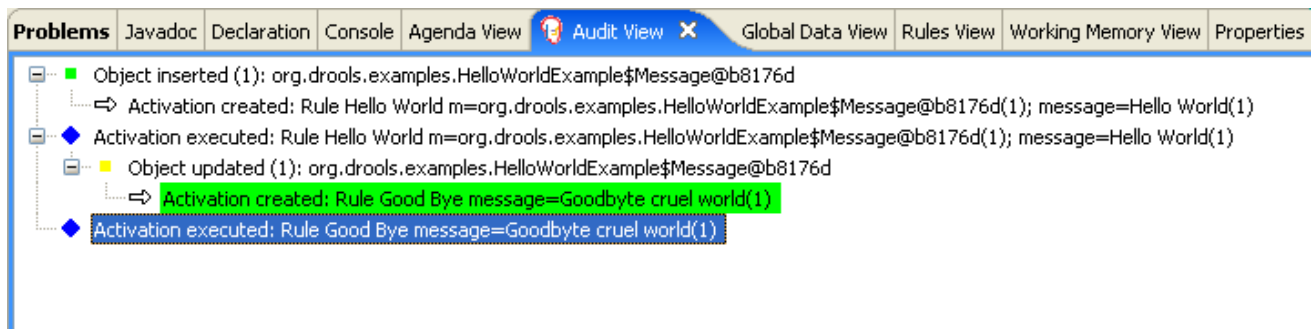


```
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
    object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
    tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;
    tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
    old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;
    new_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
    tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

この例の実行フローをさらに理解するには、**target/helloworld.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**Audit view** で、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。次に、このアクティベーションが実行され、**Message** オブジェクトを更新して、**"Good Bye"** ルールのアクティベーションをトリガーします。最後に、**"Good Bye"** ルールが実行されます。**Audit View** でインベントが選択されると、この例の **"Activation created"** イベントである元のイベントが緑色にハイライトされます。

図7.3 Hello World の例の監査ビュー



7.3. 状態のデシジョン例 (前向き連鎖および競合解決)

状態のデシジョンセットの例では、デシジョンエンジンが、前向き連鎖およびワーキングメモリー内のファクトへの変更を使用して、ルールの実行競合を順番に解決していく方法を説明しています。この例では、ルールで定義可能な顕著性の値またはアジェンダグループを使用して競合を解決することにフォーカスしています。

以下は、状態の例の概要です。

- 名前: **state**
- Main クラス: (src/main/java 内の)
org.drools.examples.state.StateExampleUsingSaliency、**org.drools.examples.state.StateExampleUsingAgendaGroup**
- モジュール: **drools-examples**
- タイプ: Java アプリケーション

- **ルールファイル:** (`src/main/resources` 内の) `org.drools.examples.state.*.drl`
- **目的:** ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。

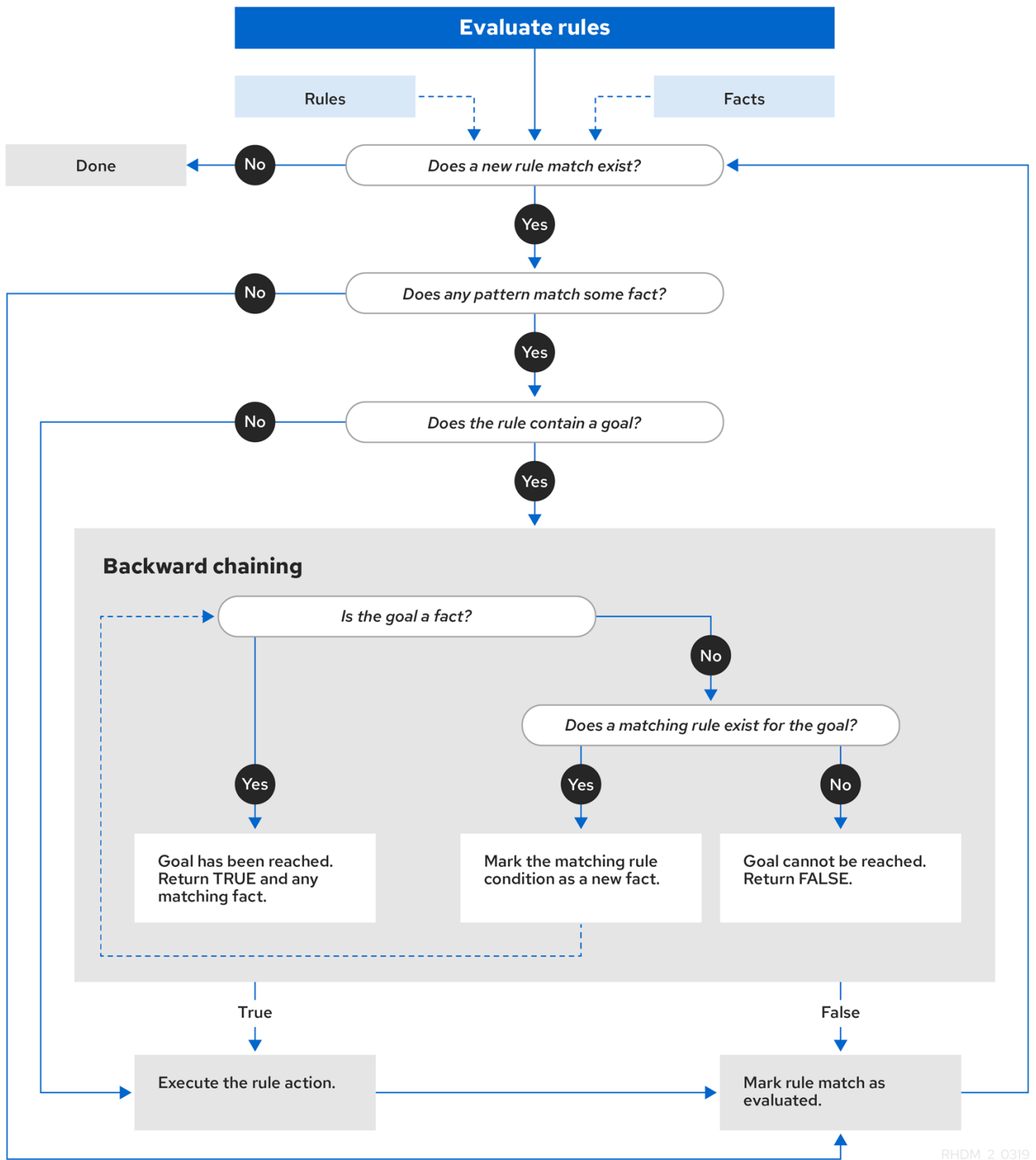
前向き連鎖のルールシステムは、デシジョンエンジンのワーキングメモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトがワーキングメモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

反対に、後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

Red Hat Process Automation Manager のデシジョンエンジンは、前向き連鎖と後向き連鎖の両方を使用してルールを評価します。

以下の図は、デシジョンエンジンが、ロジックフローで後向き連鎖のセグメントと、前向き連鎖全体とを使用してルールを評価する方法を例示します。

図7.4 前向き連鎖と後向き連鎖を使用したルール評価のロジック



状態の例では、**State** クラスごとに、名前や現在の状態のフィールドが含まれます (`org.drools.examples.state.State` のクラス参照)。以下の状態は、各プロジェクトで考えられる状態 2 つです。

- NOTRUN
- FINISHED

State 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...
}

```

状態の例には、同じ例が 2 つのバージョンとして提供されており、それぞれルール実行の競合を解決します。

- ルールの顕著性を使用して競合を解決する **StateExampleUsingSalience** バージョン
- ルールアジェンダグループを使用して競合を解決する **StateExampleUsingAgendaGroups** バージョン

状態の例のバージョンはいずれも、**A**、**B**、**C**、**D** の 4 つの **State** オブジェクトを使用します。最初に、それぞれの状態は、**NOTRUN** に設定されます。NOTRUN は、例が使用するコンストラクターのデフォルト値です。

顕著性を使用した状態の例

状態の例の **StateExampleUsingSalience** バージョンでは、ルールで顕著性の値を使用し、ルール実行の競合を解決します。顕著性の値が高いルールは、アクティベーションキューの順番で、優先度が高くなります。

この例では、各 **State** インスタンスを KIE セッションに挿入して、**fireAllRules()** を呼び出します。

顕著性の状態例の実行

```

final State a = new State( "A" );
final State b = new State( "B" );
final State c = new State( "C" );
final State d = new State( "D" );

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();

```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingSalience** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの顕著性の状態例の出力

```

A finished
B finished
C finished
D finished

```

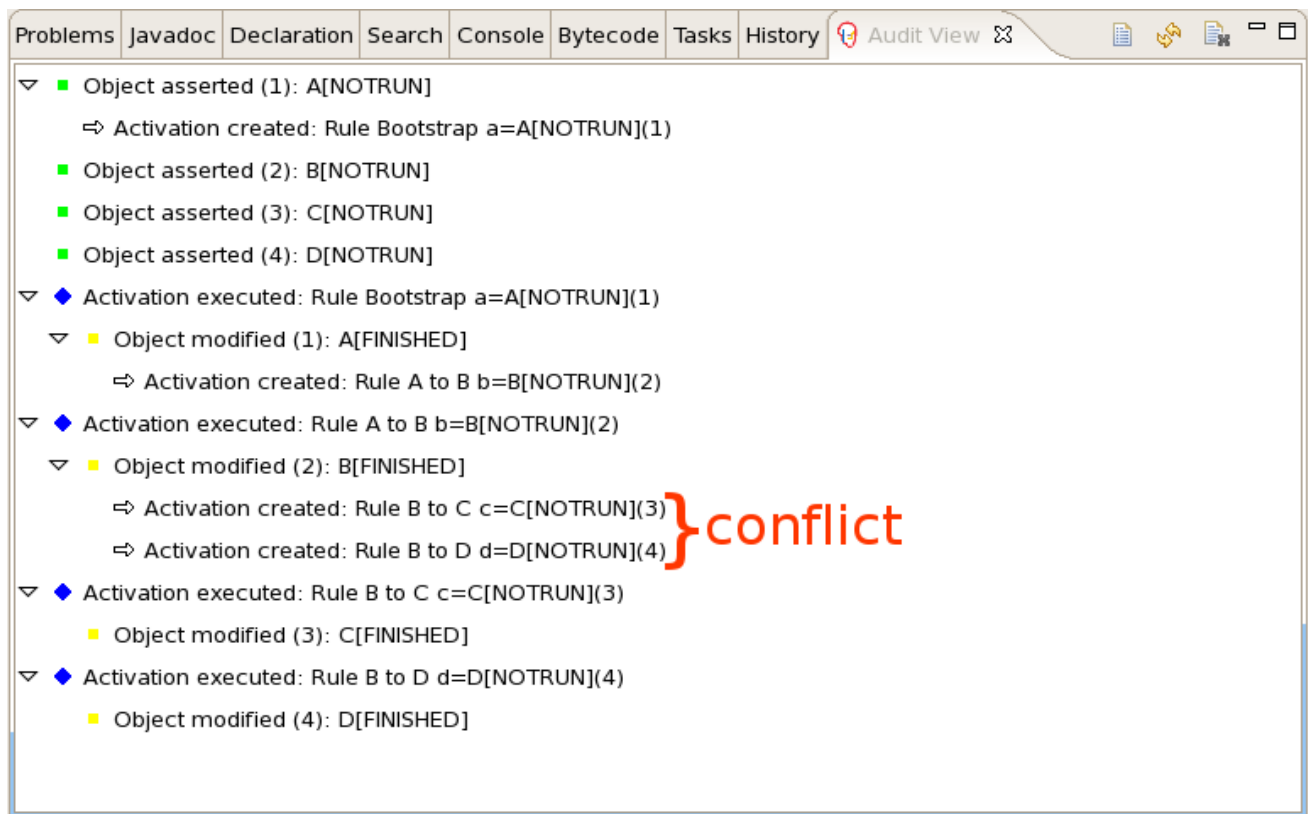
4 つのルールが存在します。

まず、**"Bootstrap"** ルールが実行され、**A** の状態が **FINISHED** に設定されます。次に、**B** の状態が **FINISHED** に変更され、オブジェクト **C** と **D** はいずれも **B** に依存するため、競合が発生しますが、顕著性の値で解決されます。

この例の実行フローをさらに理解するには、**target/state.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**Audit View** は、状態が **NOTRUN** のオブジェクト **A** のアサーションが **"Bootstrap"** ルールをアクティベートしますが、他のオブジェクトのアサーションはすぐに有効になりません。

図7.5 顕著性の状態例の監査ビュー



顕著性の状態例の "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、ルール **"A to B"** をアクティベートします。

顕著性の状態例の "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** の状態を **FINISHED** に変更し、"**B to C**" と "**B to D**" 両方のルールをアクティベートして、これらのアクティベーションをデシジョンエンジンアジェンダに配置します。

顕著性の状態例の "B to C" および "B to D" 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

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
```

この時点から、両方のルールが実行される可能性があるため、これらのルールは競合しています。競合解決戦略を使用すると、デシジョンエンジンアジェンダがどのルールを実行するかを決定できます。**"B to C"** は、顕著性の値が高い (**10** と、デフォルトの顕著性の値 **0**) ので、先に実行され、オブジェクト **C** の状態が **FINISHED** に変更されます。

IDE の **Audit View** では、ルール "**A to B**" の **State** オブジェクトが変更され、2 つのアクティベーションが競合する結果になることが分かります。

IDE で **Agenda View** を使用して、デシジョンエンジンアジェンダの状態を調査できます。この例では **Agenda View** で、ルール "**A to B**" のブレイクポイントと、2 つの競合するルールを持つアジェンダの状態が分かります。最後にルール "**B to D**" が実行され、オブジェクト **D** の状態が **FINISHED** に変更されます。

図7.6 顕著性の状態例のアジェンダビュー

The screenshot shows the IDE interface with two tabs: `StateExampleUsingSaliience.java` and `StateExampleUsingSaliience.drl`. The `StateExampleUsingSaliience.drl` tab is active, showing the following rules:

```

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

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
end

```

The bottom pane shows the Agenda View with the following structure:

- MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1392)
 - [0]= Activation
 - ruleName= "B to C"
 - c= State (id=1406)
 - FINISHED= 1
 - NOTRUN= 0
 - changes= PropertyChangeSupport (id=1433)
 - name= "C"
 - state= 0
 - [1]= Activation
 - ruleName= "B to D"
 - c= State (id=1406)
 - FINISHED= 1
 - NOTRUN= 0
 - changes= PropertyChangeSupport (id=1433)
 - name= "C"
 - state= 0

アジェンダグループを使用した状態の例

状態の例の **StateExampleUsingAgendaGroups** バージョンでは、ルールでアジェンダグループを使用し、ルール実行における競合を解決します。アジェンダグループを使用すると、デシジョンエンジンアジェンダが分割され、ルールのグループの実行に対してこれまで以上に制御ができるようになります。デフォルトでは、ルールはすべてアジェンダグループ **MAIN** に含まれています。**agenda-group** 属性を使用して、ルールに異なるアジェンダグループを指定できます。

最初は、ワーキングメモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

この例では、**auto-focus** 属性を使用すると **"B to D"** の前に **"B to C"** ルールを実行できます。

アジェンダグループの状態例のルール "B to C"

```
rule "B to C"
  agenda-group "B to C"
  auto-focus true
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
  end
```

ルール **"B to C"** は、アジェンダグループ **"B to D"** の **setFocus()** を呼び出し、有効なルールを実行できるようにします。その後にルール **"B to D"** が実行できるようになります。

アジェンダグループの状態例のルール "B to D"

```
rule "B to D"
  agenda-group "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
  end
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingAgendaGroups** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます (状態の例の顕著性バージョンと同じ)。

IDE コンソールでのアジェンダグループの状態例の出力

```
A finished
B finished
C finished
D finished
```

状態の例に含まれる動的なファクト

状態の例に含まれる主なコンセプトとして他には、**PropertyChangeListener** オブジェクトを実装するオブジェクトをもとに **動的ファクト** を使用するというものがあります。デシジョンエンジンがファクトプロパティーへの変更を確認し、対応するためには、アプリケーションがデシジョンエンジンに対し

て、変更があったことを通知する必要があります。**modify** ステートメントを使用して、このコミュニケーションをルールで明示的に設定するか、JavaBeans 使用で定義されているようにファクトが **PropertyChangeSupport** インターフェースを実装するように指定することで暗黙的に設定できます。

この例は、ルールで **modify** ステートメントを明示的に指定しなくても良いように **PropertyChangeSupport** インターフェースを使用する方法が示されています。このインターフェースを使用するには、**org.drools.example.State** クラスと同じ方法で、ファクトに **PropertyChangeSupport** が実装されていることを確認し、DRL ルールファイルで以下のコードを使用して、これらのファクトでプロパティ変更がないかをリッスンするようにデシジョンエンジンを設定してください。

動的ファクトの宣言

```
declare type State
    @propertyChangeSupport
end
```

PropertyChangeListener オブジェクトを使用する場合に、各セッターは通知用に追加のコードを実装する必要があります。たとえば、**state** の以下のセッターは **org.drools.examples** のクラスに含まれます。

PropertyChangeSupport のセッター例

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                     oldState,
                                     newState );
}
```

7.4. フィボナッチのデシジョン例 (再帰および競合解決)

フィボナッチのデシジョンセットの例では、デシジョンエンジンが再帰を使用してルールの実行競合を順番に解決していく方法を説明しています。この例では、ルールで定義可能な顕著性の値を使用して競合を解決することにフォーカスしています。

以下は、フィボナッチの例の概要です。

- 名前: フィボナッチ
- Main クラス: (src/main/java 内の) **org.drools.examples.fibonacci.FibonacciExample**
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の) **org.drools.examples.fibonacci.Fibonacci.drl**
- 目的: ルールの顕著性を使用した再帰や競合解決を例示します。

フィボナッチ数は、0 または 1 で開始する数列です。0、1、1、2、3、5、8、13、21、34、55、89、144、233、377、610、987、1597、2584、4181、6765、10946 などのように、2 つの先行する数を足すことにより、次にくるフィボナッチ数が求められます。

フィボナッチの例では、**Fibonacci** のファクトクラスを1つ使用し、このクラスに以下のフィールド2つが含まれています。

- **sequence**
- **value**

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します。**value** フィールドは、その数列の位置のフィボナッチオブジェクトの値を示します。**-1** は、計算する必要がある値という意味です。

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...
}
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.fibonacci.FibonacciExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでのフィボナッチの例の出力

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

Java でこの動作を実現するには、sequence フィールドに **50** を指定して、**Fibonacci** オブジェクトを挿入します。この例では、次に再帰ルールを使用して、他の 49 個の **Fibonacci** オブジェクトを挿入します。

PropertyChangeSupport インターフェースを実装して動的ファクトを使用する代わりに、この例では MVEL 方言の **modify** キーワードを使用して、ブロックセッターアクションを有効にしてデシジョンエンジンに変更を通知しています。

フィボナッチの例の実行

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

この例では、以下の 3 つのルールを使用します。

- "Recurse"
- "Bootstrap"
- "Calculate"

"Recurse" ルールは、値が **-1** の、アサートされた各 **Fibonacci** オブジェクトを照合して、現在の値よりも数列が 1 つ小さい **Fibonacci** オブジェクトを新たに作成し、アサートします。数列フィールドが **1** に相当するオブジェクトが存在しない場合に、フィボナッチオブジェクトが追加されると毎回、このルールは再度照合、実行されます。メモリーにフィボナッチオブジェクト 50 個すべてが存在する場合には、**not** 条件要素を使用して、ルールの合致を停止します。また、"**Bootstrap**" ルールを実行する前に **Fibonacci** オブジェクト 50 個すべてをアサートする必要があるので、このルールには **salience** の値も含まれます。

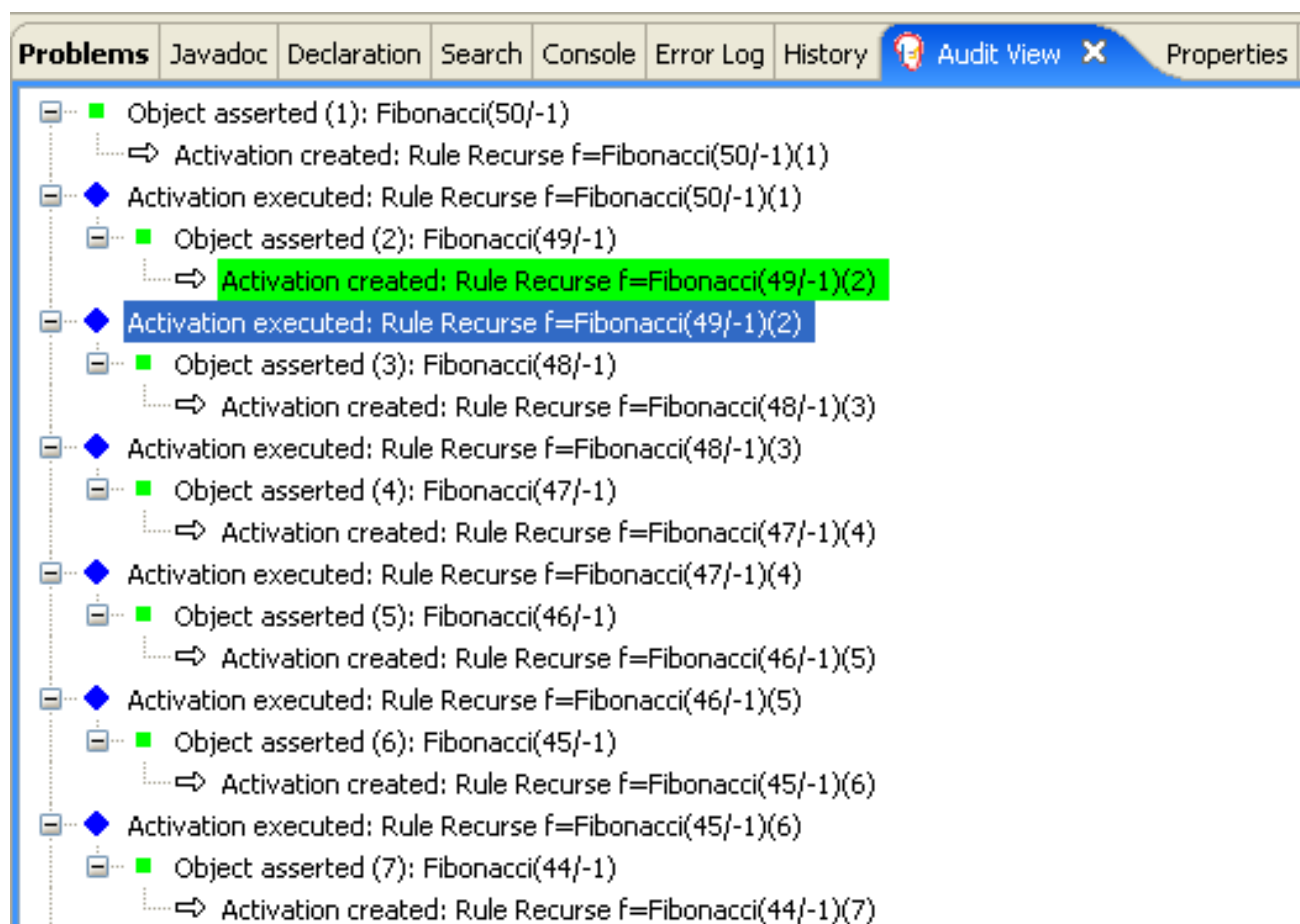
ルール "Recurse"

```
rule "Recurse"
    salience 10
    when
        f : Fibonacci ( value == -1 )
        not ( Fibonacci ( sequence == 1 ) )
    then
        insert( new Fibonacci( f.sequence - 1 ) );
        System.out.println( "recurse for " + f.sequence );
    end
```

この例の実行フローをさらに理解するには、**target/fibonacci.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window** → **Show View**) にロードします。

この例では、**監査ビュー** に、**sequence** フィールドが **50** に指定された、**Fibonacci** の元のアサーションが表示されます。これは Java コードで実行されています。これ以降、**監査ビュー** で、ルールの再帰が継続して行われ、アサートされた **Fibonacci** オブジェクトにより、"**Recurse**" ルールがアクティベートされて、再度実行されます。

図7.7 監査ビューでのルール "Recurse"



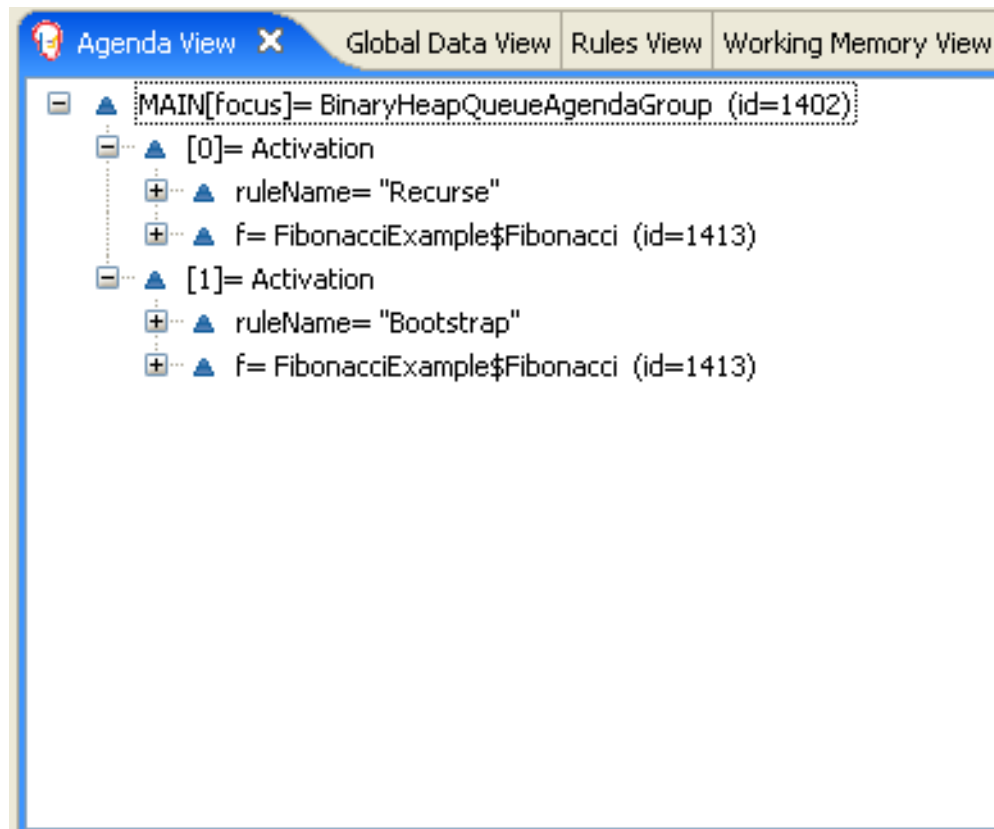
sequence フィールドが 2 の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが一致し、**"Recurse"** ルールとともにアクティベートされます。フィールド **sequence** には、複数の制約があり、1 または 2 と同等かをテストしている点に注目してください。

ルール "Bootstrap"

```
rule "Bootstrap"
when
  f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction
then
  modify ( f ){ value = 1 };
  System.out.println( f.sequence + " == " + f.value );
end
```

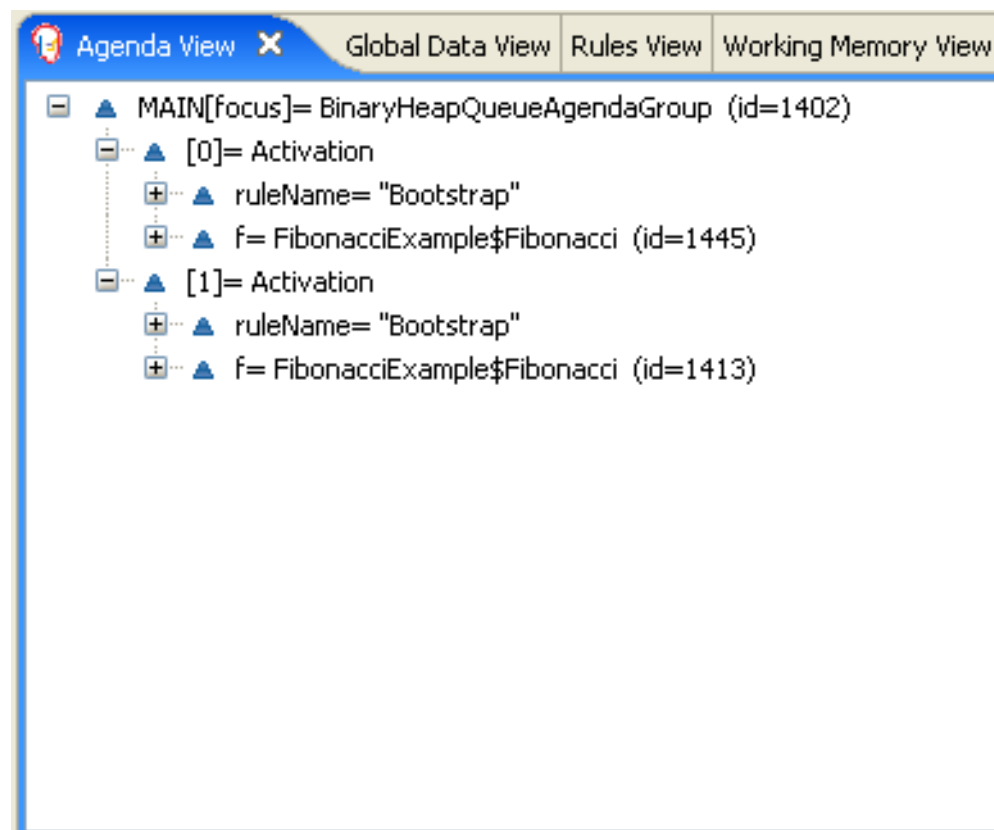
IDE で **Agenda View** を使用して、デシジョンエンジンアジェンダの状態を調査できます。**"Recurse"** の顕著性の値のほうが高いので、**"Bootstrap"** ルールはまだ実行されません。

図7.8 アジェンダビュー 1 でのルール "Recurse" および "Bootstrap"



sequence が 1 の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが再度一致し、このルールに含まれる 2 つのルールがアクティベートされます。**sequence** が 1 の **Fibonacci** オブジェクトが存在すると、すぐに **not** 条件要素で、ルールが一致しなくなるので、**"Recurse"** ルールの照合やアクティベーションはされません。

図7.9 アジェンダビュー 2 でのルール "Recurse" および "Bootstrap"



"Bootstrap" ルールは、**sequence** が **1** と **2** のオブジェクトの値を **1** に設定します。値が **-1** でない **Fibonacci** オブジェクトが 2 つあるので、**"Calculate"** ルールの照合が可能になります。

この例のある時点で、ワーキングメモリーに 50 近くの **Fibonacci** オブジェクトが存在します。3 つ選択してそれぞれを乗算し、順番に各値を計算する必要があります。フィールドの制約なしに、ルールで **Fibonacci** パターン 3 つを使用してクラス積候補を絞り込む場合に、考えられる組み合わせとして 50x49x48 通りあり、約 12 万 5000 のルールを実行できるにもかかわらず、その大半が誤っていることになります。

"Calculate" ルールは、フィールドの制約を使用して正しい順番にフィボナッチパターン 3 つを評価します。この手法は **cross-product matching** と呼ばれます。

最初のパターンでは、値が **!= -1** の **Fibonacci** オブジェクトを検索して、このパターンとフィールド両方をバインドします。2 番目の **Fibonacci** オブジェクトの実行する内容は同じですが、別のフィールド制約を追加して、シーケンスが **f1** にバインドされている **Fibonacci** オブジェクトより 1 つ大きくなるようにします。このルールが初めて実行されると、シーケンスが **1** と **2** にだけ、値 **1** が割り当てられていることが分かります。また、この 2 つの制約で、**f1** がシーケンス **1** を、**f2** がシーケンス **2** を参照するようにします。

最後のパターンでは、値が **-1** と等しく、シーケンスが **f2** よりも大きい **Fibonacci** オブジェクトを検索します。

フィボナッチの例のこの時点で、3 つの **Fibonacci** オブジェクトが利用可能なクロス積から正しく選択され、**f3** にバインドされている 3 番目の **Fibonacci** オブジェクトの値を計算できます。

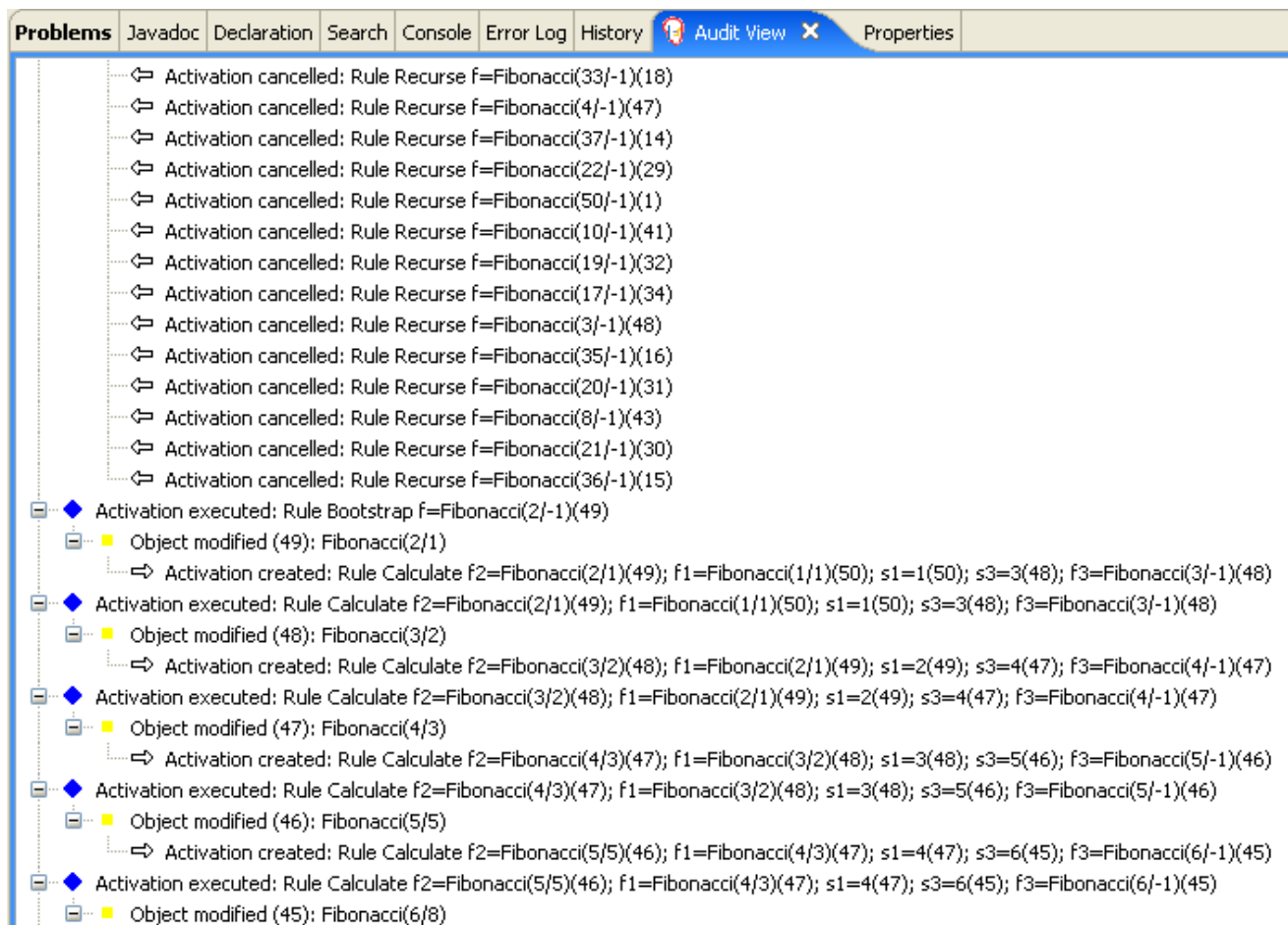
ルール "Calculate"

```
rule "Calculate"
when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

modify ステートメントにより、**f3** にバインドされた **Fibonacci** オブジェクトの値が更新されます。つまり、値が **-1** 以外の **Fibonacci** オブジェクトが新たに存在するということで、**"Calculate"** ルールにより、再度合致があるか検索して次のフィボナッチ番号を算出することができます。

IDE のデバッグビューまたは 監査ビュー では、最後の **"Bootstrap"** ルールが実行されることで **Fibonacci** オブジェクトが変更され、**"Calculate"** ルールに合致し、次に、別の **Fibonacci** オブジェクトが変更され、この **"Calculate"** ルールに再度合致できていることが分かります。このプロセスは、すべての **Fibonacci** オブジェクトに値が設定されるまで継続されます。

図7.10 監査ビューのルール



7.5. 価格設定のデシジョン例 (デシジョンテーブル)

価格設定のデシジョン例では、スプレッドシートのデシジョンテーブルを使用して、DRL ファイルに直接ではなく、表形式で保険料金の価格を計算する方法が説明されます。

以下は価格設定の例の概要です。

- 名前: **decisiontable**
- Main クラス: **org.drools.examples.decisiontable.PricingRuleDTExample** (in **src/main/java**)
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: **org.drools.examples.decisiontable.ExamplePolicyPricing.xls** (**src/main/resources** 内)
- 目的: スプレッドシートのデシジョンテーブルを使用してルールを定義する方法を示します。

スプレッドシートのデシジョンテーブルは、表形式でビジネスルールを定義する XLS または XLSX 形式のスプレッドシートです。スプレッドシートのデシジョンテーブルは、スタンドアロンの Red Hat Process Automation Manager プロジェクトに追加したり、Business Central のプロジェクトにアップロードしたりできます。スプレッドシートの各行がルールになり、各列が条件、アクション、または別のルール属性になります。最初に Red Hat Process Automation Manager でデシジョンテーブルを作成してアップロードします。次に、その他のすべてのルールアセットと同じように、定義したルールを Drools Rule Language (DRL) ルールにコンパイルします。

この価格設定の例では、特定タイプの保険を申請するドライバーに対して基本価格と割引を計算するビジネスルールセットを提供します。保険の種類とともにドライバーの年齢と履歴が、基本保険料の算定で考慮され、別のルールで適用可能な割引が計算されます。

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.decisiontable.PricingRuleDTExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

この例を実行するコードは、標準の実行パターンに準拠しています。ルールが読み込まれ、ファクトが挿入されて、ステートレスな KIE セッションが作成されます。この例における違いは、DRL ファイルや他のソースではなく、**ExamplePolicyPricing.xls** ファイルでルールが定義されるという点です。このスプレッドシートファイルは、テンプレートと DRL ルールを使ってデシジョンエンジンに読み込まれます。

スプレッドシートのデシジョンテーブルの設定

ExamplePolicyPricing.xls スプレッドシートには、以下の 2 つのデシジョンテーブルが含まれています。

- **Base pricing rules**
- **Promotional discount rules**

この例のスプレッドシートで分かるように、デシジョンテーブルの作成にはスプレッドシートの最初のタブしか使用できませんが、単一のタブ内に複数のテーブルが作成できます。デシジョンテーブルは必ずしもトップダウンの論理に従うものではなく、ルールとなるデータを補足する手段です。ルールの評価は、ルールエンジンのすべての通常のメカニクが適用されるため、必ずしも特定の順序で行われるわけではありません。このために、スプレッドシートの同一タブ内に複数のデシジョンテーブルが作成可能となります。

デシジョンテーブルは、対応するルールテンプレートファイルである **BasePricing.drt** と **PromotionalPricing.drt** で実行されます。これらのテンプレートファイルはテンプレートパラメーターによってデシジョンテーブルを参照し、デシジョンテーブルの条件およびアクションの各種ヘッダーを直接参照します。

BasePricing.drt ルールテンプレートファイル

```
template header
age[]
profile
priorClaims
policyType
base
reason

package org.drools.examples.decisiontable;

template "Pricing bracket"
age
policyType
base
```



```

rule "Pricing bracket_{row.rowNumber}"
when
  Driver(age >= @{age0}, age <= @{age1}
    , priorClaims == "@{priorClaims}"
    , locationRiskProfile == "@{profile}"
  )
  policy: Policy(type == "@{policyType}")
then
  policy.setBasePrice(@{base});
  System.out.println("@{reason}");
end
end template

```

PromotionalPricing.drt ルールテンプレートファイル

```

template header
age[]
priorClaims
policyType
discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_{row.rowNumber}"
when
  Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}")
  policy: Policy(type == "@{policyType}")
then
  policy.applyDiscount(@{discount});
end
end template

```

ルールは、KIE セッション **DTableWithTemplateKB** の **kmodule.xml** 参照によって実行されます。これは **ExamplePolicyPricing.xls** スプレッドシートを特定して参照するもので、ルールの実行の成功には必要なものです。この実行方法により、ルールをスタンドアロンユニットとして実行したり (ここでの例) パッケージ化されたナレッジ JAR (KJAR) ファイルにルールを含めたりすることができるので、スプレッドシートはルール実行とともにパッケージ化されます。

kmodule.xml ファイルの以下のセクションは、ルールが正常に実行され、スプレッドシートが機能するために必要になります。

```

<kbase name="DecisionTableKB" packages="org.drools.examples.decisiontable">
  <ksession name="DecisionTableKS" type="stateless"/>
</kbase>

<kbase name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
  <ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
    template="org/drools/examples/decisiontable-template/BasePricing.drt"

```

```

        row="3" col="3"/>
<ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
    template="org/drools/examples/decisiontable-template/PromotionalPricing.drt"
    row="18" col="3"/>
<ksession name="DTableWithTemplateKS"/>
</kbase>

```

ルールテンプレートファイルを使ったデシジョンテーブルの実行方法とは別に、**DecisionTableConfiguration** オブジェクトを使用して、**DecisionTableInputType.xls** のような入力スプレッドシートを入力タイプとして指定することもできます。

```

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 );

```

価格設定の例では以下の 2 つのファクトタイプを使用します。

- **Driver**
- **Policy**.

この例では、これらのファクトのデフォルト値をそれぞれの Java クラス **Driver.java** と **Policy.java** に設定します。**Driver** は 30 歳で、これまでに保険の請求をしたことがなく、現在のリスクプロファイル **LOW** となっています。申請している **Policy** は **COMPREHENSIVE** です。

デシジョンテーブルでは、各行は異なるルール。各列は条件またはアクションとみなされます。実行時にアジェンダがクリアされなければ、デシジョンテーブルの各行が評価されます。

デシジョンテーブルのスプレッドシート (XLS または XLSX) には、ルールデータを定義する以下の 2 つの主要なエリアが必要です。

- **RuleSet** エリア
- **RuleTable** エリア

RuleSet エリアでは、ルールセット名、ユニバーサルルール属性など、(このスプレッドシートだけでなく) すべてのルールをパッケージ全体に、グローバルに適用する要素を定義します。**RuleTable** エリアでは、実際のルール (行) と、指定したルールセットのルールテーブルを構成する条件、アクション、その他のルール属性 (列) を定義します。デシジョンテーブルのスプレッドシートには複数の **RuleTable** エリアを追加できますが、**RuleSet** エリアは 1 つのみとなります。

図7.11 デシジョンテーブルの設定

	C	D	E	F	G	H
RuleSet	org.drools.examples.decisiontable					
Notes	This decision table is for working out some basic prices and pretending actuaries don't exist					
RuleTable Pricing bracket						
CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION	
Driver	policy: Policy					
age >= \$1, age <= \$2	locationRiskProfile	priorClaims	type	policy.setBasePrice(\$param);	System.out.println("\$param");	
Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base \$ AUD	Record Reason	

RuleTable エリアでは、ルール属性を適用するオブジェクトも定義します。この例では、**Driver** と **Policy**、さらにオブジェクトの制限です。たとえば、**Driver** オブジェクトの制限では、**Age Bracket** コラムが **age >= \$1, age <= \$2** と定義されています。ここでのコンマ区切りの範囲は、**18,24** などのテーブルのコラム値で定義されます。

基本価格のルール

価格設定例での **Base pricing rules** デシジョンテーブルでは、ドライバーの年齢、リスクプロファイル、請求数、ポリシータイプを評価し、これらの条件をベースにしたポリシーの基本価格を算定します。

図7.12 基本価格の計算

	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

Driver 属性は、以下のテーブルコラムで定義されます。

- Age Bracket:** この年齢層には、ドライバー年齢の条件範囲を定義する条件 **age >=\$1, age <=\$2** の定義があります。この条件コラムでは **\$1 and \$2** を使用しており、スプレッドシートではコンマ区切りになります。ここでの値入力は **18,24** や **18, 24** の形式となり、両方ともビジネスルールの実行で機能する形式です。

- **Location risk profile:** リスクプロファイルは、この例のプロバラムでは常に **LOW** として渡す文字列です。ただし、**MED** または **HIGH** に変更することが可能です。
- **Number of prior claims:** これまでの請求数は整数で定義し、アクションをトリガーするには条件コラムのものと同一である必要があります。この値は範囲ではなく、完全一致のみになります。

デシジョンテーブルの **Policy** は、ルールの条件とアクションの両方で使用され、属性は以下のテーブルコラムで定義されます。

- **Policy type applying for:** ポリシータイプは文字列として渡される条件で、適用される以下のいずれかのポリシータイプを定義します: **COMPREHENSIVE**、**FIRE_THEFT**、または **THIRD_PARTY**。
- **Base \$ AUD:** **basePrice** は **ACTION** として定義され、これはこの値に対応するスプレッドシートのセルをベースに制限 **policy.setBasePrice(\$param);** で価格を設定します。このデシジョンテーブルの対応する DRL ルールを実行する際には、ファクトに合致する **true** 条件でルールの **then** 部分がこのアクションステートメントを実行し、基本価格に対応する値に設定します。
- **Record Reason:** ルールが正常に実行されると、アクションは出力メッセージを **System.out** コンソールに生成し、どのルールが適用されたかが反映されます。これは後でアプリケーションにキャプチャーされ、プリントされます。

この例では、左側の最初のコラムでルールをカテゴリ分けしています。このコラムは注釈目的で、ルール実行には影響がありません。

プロモーション割引ルール

価格設定例での **Promotional discount rules** デシジョンテーブルでは、ドライバーの年齢、請求数、ポリシータイプを評価し、ポリシー価格の割引を算定します。

図7.13 割引計算

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

このデシジョンテーブルには、ドライバーに適用可能な割引条件が含まれています。基本価格の算定と同様に、このテーブルではドライバーの **Age**、**Number of prior claims**、および **Policy type applying for** を評価して、適用する **Discount %** 率を判定します。たとえば、ドライバーは 30 歳であり、請求履歴がなく、**COMPREHENSIVE** ポリシーを申請している場合、**20** パーセントの割引率が導き出されます。

7.6. ペットショップのデシジョン例 (アジェンダグループ、グローバル変数、コールバック、GUI 統合)

ペットショップのデシジョンセットの例では、ルールでのアジェンダグループとグローバル変数の使用方法および Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法を説明しています。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられたワーキングメモリー内の変更をもとに GUI を更新する方法も説明しています。

以下は、ペットショップの例の概要です。

- 名前: **petstore**
- Main クラス: (src/main/java 内の) **org.drools.examples.petstore.PetStoreExample**
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の) **org.drools.examples.petstore.PetStore.drl**
- 目的: ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。

ペットショップの例では、**PetStoreExample.java** クラス例を使用して (Swing イベントを処理する複数のクラスに加え)、以下のクラスを主に定義しています。

- **Petstore** には **main()** メソッドが含まれます。
- **PetStoreUI** は Swing ベースの GUI を作成して表示します。このクラスには複数の小さいクラスが含まれており、マウスボタンのクリックなど、さまざまな GUI イベントに主に対応します。
- **TableModel** には表データが含まれています。このクラスは基本的に **AbstractTableModel** を継承する **JavaBean** です。
- **CheckoutCallback** により、GUI がルールと対話できるようになります。
- **Ordershow** は購入するアイテムを保持します。
- **Purchase** には、顧客が購入する商品と商品の詳細が保存されます。
- **Product** は、販売可能な商品と価格の詳細を含む **JavaBean** です。

この例の Java コードはほぼ、プレーンな **JavaBean** か **Swing** ベースとなっています。Swing コンポーネントの詳細は、[「Creating a GUI with JFC/Swing」](#) の Java チュートリアルを参照してください。

ペットショップの例でのルール実行動作

他のデシジョンセットの例では、ファクトがすぐにアサートされて実行されるのに対し、ペットショップの例では、ユーザーの対話をもとに他のファクトが収集されるまでルールは実行されます。このルールでは、コンストラクターで作成される **PetStoreUI** オブジェクトを使用してルールを実行し、**Vector** オブジェクトの **stock** を受け入れ入れて商品を収集します。次に、この例では、以前の読み込まれたルールベースを含む **CheckoutCallback** クラスのインスタンスを使用します。

ペットショップの KIE コンテナおよびファクト実行の設定

```
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );
```

```
// A callback is responsible for populating the working memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kc ) );
ui.createAndShowGUI();
```

ルールを実行する Java コードは **CheckoutCallBack.checkout()** メソッドに含まれます。このメソッドは、ユーザーが UI で **チェックアウト** をクリックするとトリガーされます。

CheckoutCallBack.checkout() からのルール実行

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user interaction.

    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

このコード例では、2つの要素を **CheckoutCallBack.checkout()** メソッドに渡します。1つ目の要素は、GUI の一番下にある出力テキストのフレームを囲む **JFrame** Swing コンポーネントのハンドルです。2つ目の要素は注文アイテムのリストで、GUI の右上のセクションにある **Table** エリアからの情報を保存する **TableModel** から取得します。

for ループは GUI からの注文アイテム一覧を **Order** JavaBean に変換します。これは、**PetStoreExample.java** ファイルにも含まれています。

今回の例では、データはすべて Swing コンポーネントに含まれており、ユーザーが UI の **チェックアウト** をクリックしない限り実行されないため、ルールはステートレスな KIE セッションで実行します。ユーザーが **チェックアウト** をクリックするたびに、リストの内容を Swing **TableModel** から KIE セッションのワーキングメモリーに移動し、**ksession.fireAllRules()** メソッドで実行します。

このコード内には、**KieSession** への呼び出しが 9 個あります。1 つ目は、**KieContainer** から新しい **KieSession** を作成します (この例では、**main()** メソッドの **CheckoutCallBack** クラスから **KieContainer** に渡されます)。次の 2 つの呼び出しは、ルールでグローバル変数として保持されるオブジェクトを 2 つ渡します (メッセージの書き込みに使用する Swing テキストエリアと Swing フレーム)。他に挿入することで、商品の情報を **KieSession** と注文リストに配置します。最後の呼び出しは、標準の **fireAllRules()** です。

ペットショップのルールファイルのインポート、グローバル変数、Java 関数

PetStore.drl ファイルには、さまざまな Java クラスをルールで利用できるように、標準のパッケージとインポートステートメントが含まれています。このルールファイルには、**frame** および **textArea** などのように、ルール内で使用する **グローバル変数** が含まれています。グローバル変数では、Swing コンポーネント **JFrame** と、**setGlobal()** メソッドを呼び出した Java コードにより以前に渡された **JTextArea** コンポーネントへの参照を保持します。ルールが実行されるとすぐに失効するルールの標準変数とは異なり、グローバル変数は KIE セッションの有効期間中この値を保持します。つまり、この後に続く全ルールを評価するのに、これらの変数の内容を使用できます。

PetStore.drl パッケージ、インポートおよびグローバル変数

```
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;

import javax.swing.JFrame;

global JFrame frame
global javax.swing.JTextArea textArea
```

PetStore.drl ファイルには、このファイル内のルールが使用する関数 2 つも含まれています。

PetStore.drl Java 関数

```
function void doCheckout(JFrame frame, KieRuntime krt) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
                                         "Would you like to checkout?",
                                         "",
                                         JOptionPane.YES_NO_OPTION,
                                         JOptionPane.QUESTION_MESSAGE,
                                         null,
                                         options,
                                         options[0]);

    if (n == 0) {
        krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
    }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total)
```

```

{
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to buy a tank for your " + total + " fish?",
        "Purchase Suggestion",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        options[0]);

    System.out.print( "SUGGESTION: Would you like to buy a tank for your "
        + total + " fish? - " );

    if (n == 0) {
        Purchase purchase = new Purchase( order, fishTank );
        krt.insert( purchase );
        order.addItem( purchase );
        System.out.println( "Yes" );
    } else {
        System.out.println( "No" );
    }
    return true;
}

```

この2つの関数は以下のアクションを実行します。

- **doCheckout()** は、チェックアウトするかどうかユーザーに尋ねるダイアログボックスを表示します。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。
- **requireTank()** は、水槽を購入するかどうかを確認するダイアリーを表示します。購入する場合は、新しい水槽の **Product** がワーキングメモリーの注文リストに追加されます。



注記

この例では、効率化を図るため、すべてのルールと関数が同じルールファイルで実行しています。実稼働環境では、通常、ルールと関数を別のファイルに分けるか、静的な Java メソッドを構築して、**import function my.package.name.hello** などのインポート関数を使用し、ファイルをインポートします。

アジェンダグループを使用したペットショップルール

ペットショップの例のルールはほぼ、アジェンダグループを使用してルールの実行を制御しています。アジェンダグループを使用すると、デシジョンエンジンアジェンダを分割し、ルールのグループの実行を、詳細にわたり制御できるようになります。デフォルトでは、全ルールはアジェンダグループ **MAIN** に含まれます。**agenda-group** 属性を使用してルールに異なるアジェンダグループを指定できます。

最初は、ワーキングメモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

ペットショップの例では、ルールに以下のアジェンダグループを使用します。

- "init"
- "evaluate"
- "show items"
- "checkout"

たとえば、同じルール **"Explode Cart"** は **"init"** のアジェンダグループを使用して、ショッピングカードのアイテムを実行して、KIE セッションのワーキングメモリーに挿入するオプションが提供されるようにします。

ルール "Explode Cart"

```
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
  agenda-group "init"
  auto-focus true
  salience 10
  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、**grossTotal** がまだ計算されていない全注文に対してマッチングされます。購入アイテムごとに、順番に実行がループされます。

ルールは、アジェンダグループに関連する以下の機能を使用します。

- **agenda-group "init"** はアジェンダグループの名前を定義します。この例では、グループにはルールが1つしかありませんが、Java コードもルール結果もこのグループにフォーカスされていないため、**auto-focus** の属性により、ルールが実行されるかが決まります。
- このルールはアジェンダグループで唯一のルールですが、**auto-focus true** を使用して、**fireAllRules()** が Java コードから呼び出されると、必ず実行されるようにします。
- **kcontext....setFocus()** で、フォーカスを **"show items"** と **"evaluate"** のアジェンダグループに設定して、これらのルールが実行されるようにします。実際は、注文に含まれる全アイテムをループでチェックし、メモリーに挿入してから、挿入ごとに他のルールを実行します。

"show items" アジェンダグループには **"Show Items"** というルール1つだけが含まれます。KIE セッションのワーキングメモリーに現在含まれる注文で購入があるたびに、このルールを使用して、ルールファイルに定義した **textArea** 変数をもとに、GUI の下の部分にあるテキストエリアに詳細がロギングされます。

ルール "Show Items"

```
rule "Show Items"
  agenda-group "show items"
  when
```

```

$order : Order()
$p : Purchase( order == $order )
then
  textArea.append( $p.product + "\n");
end

```

また、**"evaluate"** アジェンダグループにより、**"Explode Cart"** ルールから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"** のルールが2つ含まれます。**"Free Fish Food Sample"**、**"Suggest Tank"** の順番に実行されます。

ルール "Free Fish Food Sample"

```

// Free fish food sample when users buy a goldfish if they did not already buy
// fish food and do not already have a fish food sample.
rule "Free Fish Food Sample"
  agenda-group "evaluate" ❶
  when
    $order : Order()
    not ( $p : Product( name == "Fish Food" ) && Purchase( product == $p ) ) ❷
    not ( $p : Product( name == "Fish Food Sample" ) && Purchase( product == $p ) ) ❸
    exists ( $p : Product( name == "Gold Fish" ) && Purchase( product == $p ) ) ❹
    $fishFoodSample : Product( name == "Fish Food Sample" );
  then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
  end

```

ルール **"Free Fish Food Sample"** は、以下の条件がすべて該当する場合のみ実行されます。

- ❶ アジェンダグループ **"evaluate"** がルール実行で評価している
- ❷ ユーザーが魚の餌をまだ持っていない
- ❸ ユーザーが無料の魚の餌サンプルをまだ持っていない
- ❹ ユーザーが金魚を注文している

この注文ファクトが上記の要件すべてを満たす場合には、新しい商品 (Fish Food Sample) が作成され、ワーキングメモリーの注文に追加されます。

ルール "Suggest Tank"

```

// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.
rule "Suggest Tank"
  agenda-group "evaluate"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Tank" ) && Purchase( product == $p ) ) ❶
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) ) ❷
    $fishTank : Product( name == "Fish Tank" )
  end

```

```

then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
end

```

ルール **"Suggest Tank"** は以下の条件がすべて該当する場合のみ実行されます。

- ① ユーザーが水槽を注文していない
- ② ユーザーが 6 匹以上注文した

このルールが実行されると、ルールファイルに定義されている **requireTank()** 関数が呼び出されます。この関数により、水槽を購入するかどうかを尋ねるダイアログが表示されます。新しい水槽の **Product** がワーキングメモリの注文リストに追加されます。ルールが **requireTank()** 関数を呼び出した場合には、このルールを使用して、関数に Swing GUI のハンドルが含まれるように、**frame** のグローバル変数を渡します。

ペットショップの例の **"do checkout"** ルールにはアジェンダルールや **when** 条件がないので、ルールは常に実行されて、デフォルトの **MAIN** のアジェンダグループの一部とみなされます。

ルール "do checkout"

```

rule "do checkout"
    when
    then
        doCheckout(frame, kcontext.getKieRuntime());
    end

```

このルールが実行されると、ルールファイルで定義されている **doCheckout()** 関数を呼び出します。この関数により、チェックアウトするかどうかユーザーに尋ねるダイアログボックスが表示されます。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。このルールで **doCheckout()** 関数を呼び出し、この変数に Swing GUI のハンドルが含まれるように **frame** グローバル変数を渡します。



注記

この例では、結果が想定どおりに実行されない場合のトラブルシューティングの方法を例示します。ルールの **when** ステートメントから条件を削除して、**then** ステートメントのアクションをテストし、アクションが正しく実行されることを検証します。

"checkout" アジェンダグループには、**"Gross Total"**、**"Apply 5% Discount"** および **"Apply 10% Discount"** の注文のチェックアウト処理、割引の適用のルールが 3 つ含まれています。

ルール "Gross Total"、"Apply 5% Discount" および "Apply 10% Discount"

```

rule "Gross Total"
    agenda-group "checkout"
    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"
  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"
  when
    $order : Order( grossTotal >= 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
  end

```

ユーザーがまだ総計を算出していない場合には、**Gross Total** で、商品の価格を累積して合計を出し、この合計を KIE セッションに渡して、**textArea** のグローバル変数を使用し、Swing **JTextArea** で合計を表示します。

総計が **10** から **20** (通貨単位) の場合には、**"Apply 5% Discount"** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

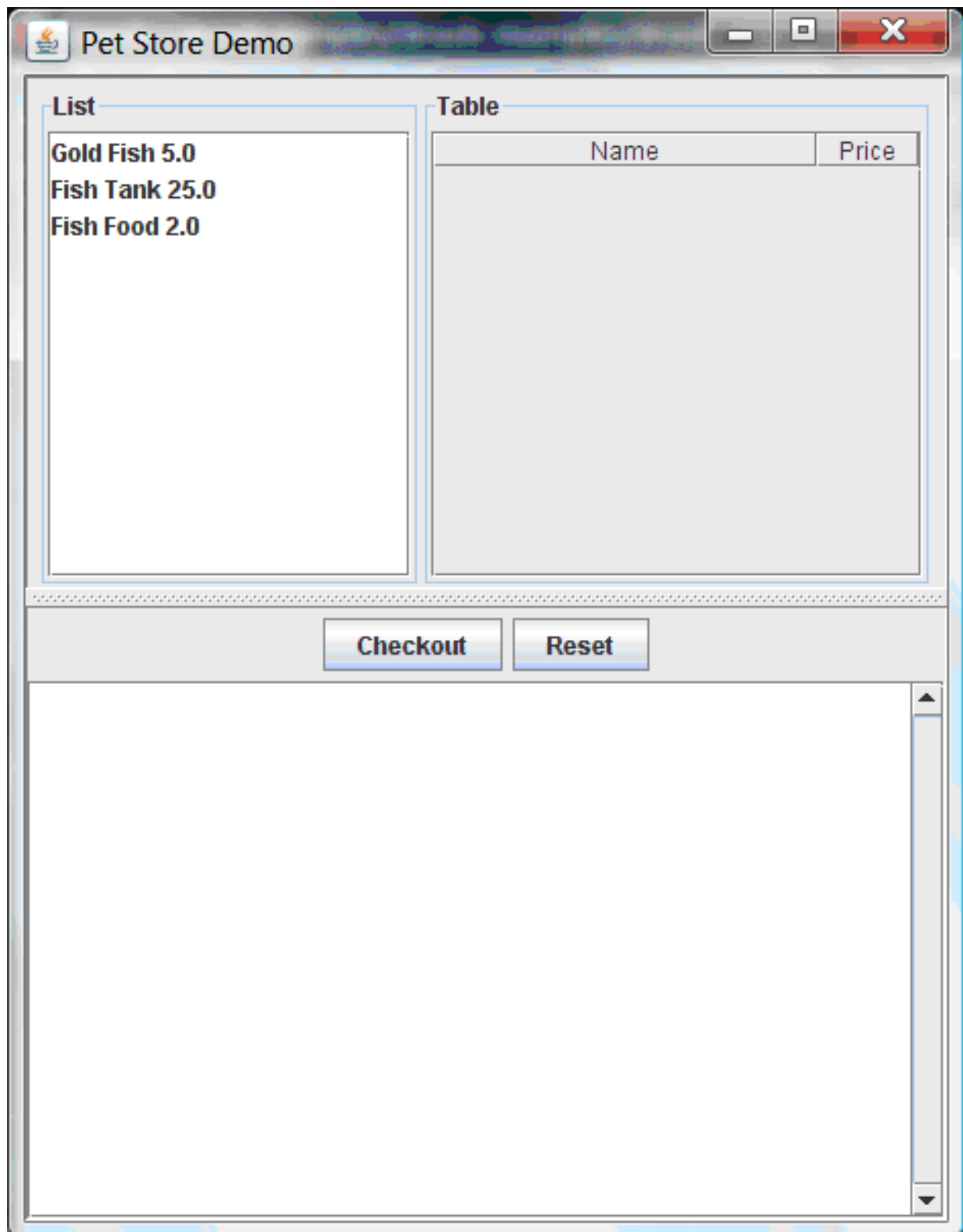
総計が **20** 未満の場合には、**"Apply 10% Discount"** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

ペットショップ例の実行

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.petstore.PetStoreExample** クラスを Java アプリケーションとして実行し、ペットショップの例を実行します。

ペットショップの例を実行すると、**Pet Store Demo** GUI ウィンドウが表示されます。このウィンドウでは、購入可能な商品 (左上)、選択済み商品の空白のリスト (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 (下) が表示されます。

図7.14 起動後のペットショップ例の GUI

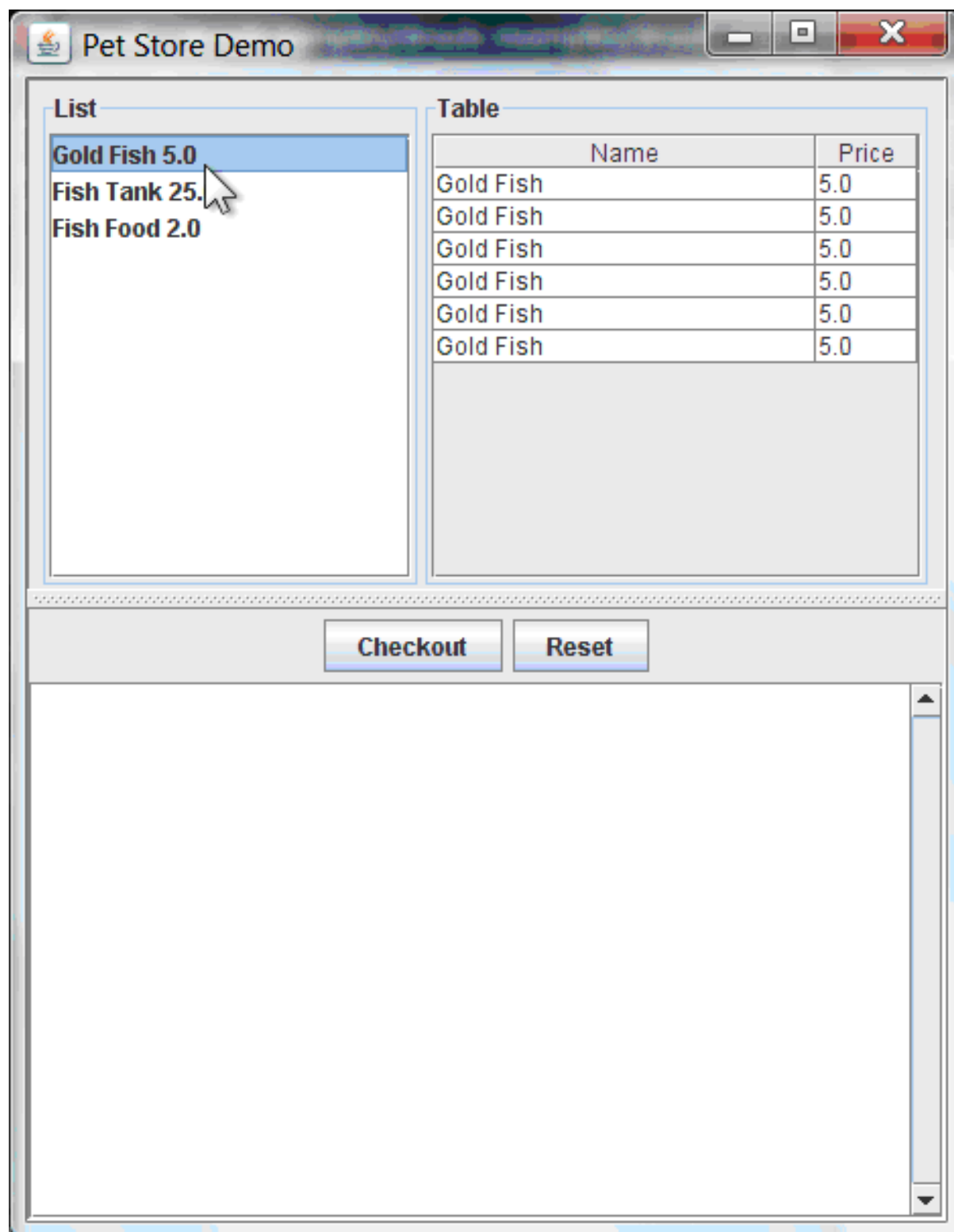


この例では、以下のイベントが発生して、この実行動作を確立します。

1. **main()** メソッドがルールベースの実行と読み込みを終えているが、ルールが実行されていないこと。今のところ、実行されたルールに関連する唯一のコードがこれです。
2. 新しい **PetStoreUI** オブジェクトが作成され、後でできるようにルールベースにハンドルを渡すこと。
3. さまざまな Swing コンポーネントが関数を実行し、最初の UI 画面が表示され、ユーザーの入力を待ちます。

リストからさまざまな商品をクリックして、UI 設定をチェックできます。

図7.15 ペットショップ例の GUI のチェック



ルールコードはまだ実行されていません。UI は Swing コードを使用してユーザーによるマウスクリックを検出し、選択済みの商品を **TableModel** オブジェクトに追加して、UI の右上隅に表示します。この例では、Model-View-Controller 設計パターンを紹介しています。

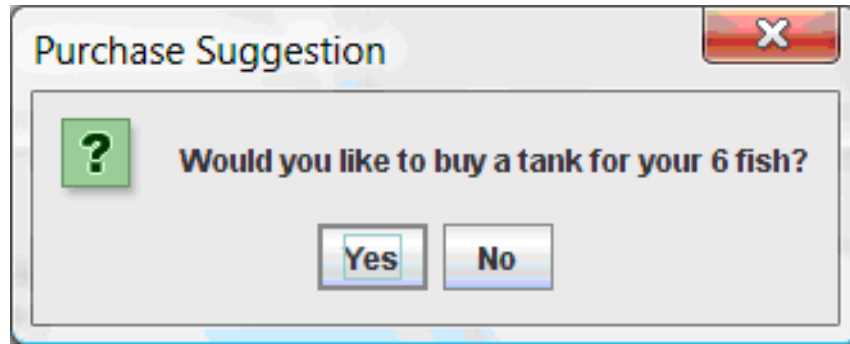
Checkout をクリックすると、ルールが以下の方法で実行されます。

1. Swing クラスは **Checkout** がクリックされるまで待機して、(最終的に) **CheckOutCallBack.checkout()** メソッドを呼び出します。これにより、**TableModel** オブジェ

クト (UI の右上隅) から KIE セッションのワーキングメモリーにデータを挿入します。その後、メソッドによりルールが実行されます。

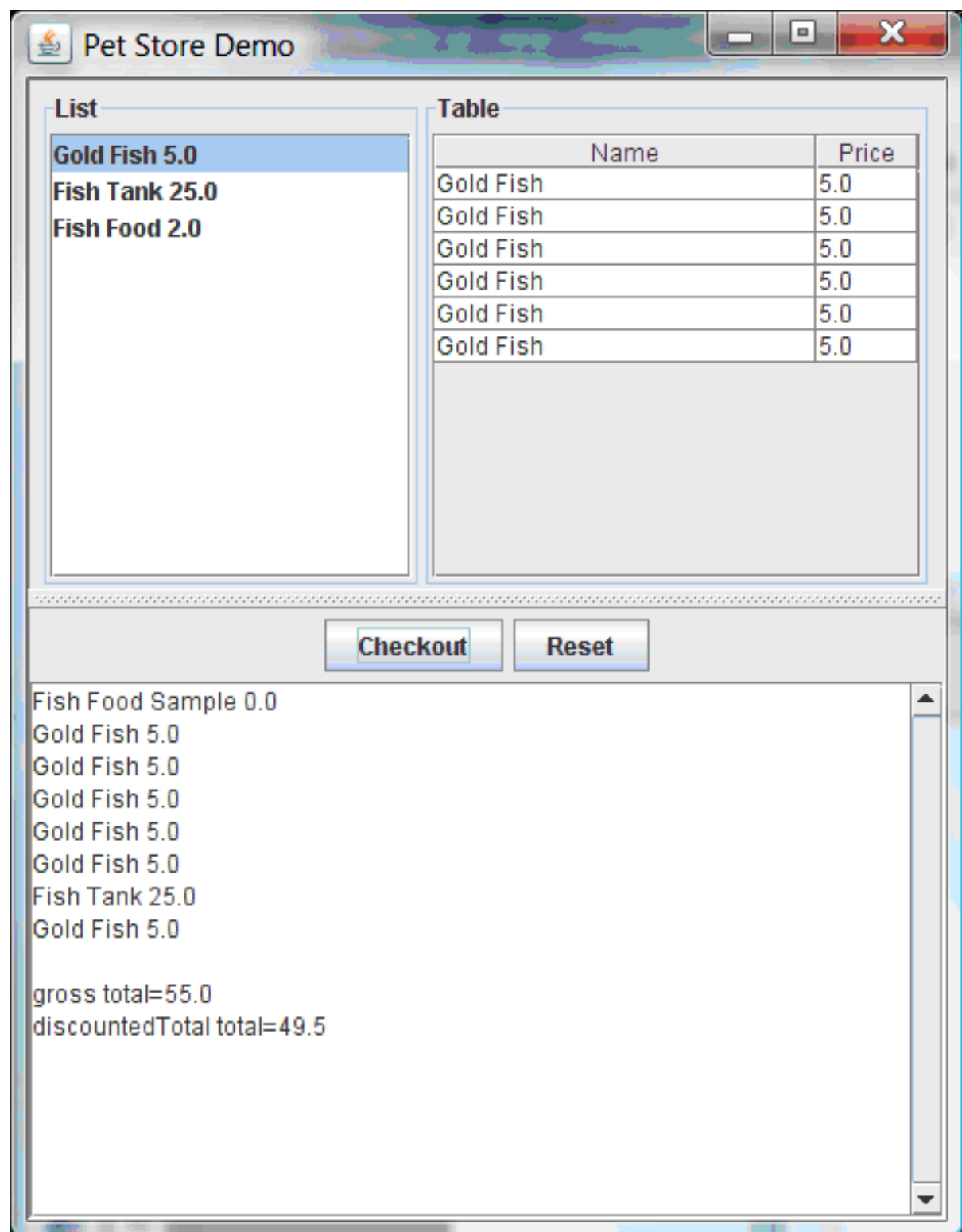
2. **"Explode Cart"** ルールは、**auto-focus** 属性を **true** に設定して最初に実行します。このルールは、カートの商品すべてを順にループしていき、商品がワーキングメモリーに含まれていることを確認し、**"show Items"** と **"evaluate"** アジェンダグループに実行するオプションを提供します。このグループのルールは、カートのコンテンツをテキストエリア (UI の下) に追加して、魚の餌を無料で受け取る資格があるかどうかを評価し、また水槽購入の有無を尋ねるかどうかを決定します。

図7.16 水槽の資格



3. 現在、他のアジェンダグループにフォーカスが当たっておらず、**"do checkout"** ルールは、デフォルトの **MAIN** アジェンダグループに含まれているので、次に実行されます。このルールは常に **doCheckout()** 関数を呼び出し、この関数によりチェックアウトをするかどうかを尋ねられます。
4. **doCheckout()** 関数は、フォーカスを **"checkout"** アジェンダグループに設定し、そのグループ内のルールに、実行するオプションを提供します。
5. **"checkout"** アジェンダグループ内のルールは、「カート」内の内容を表示し、適切な割引を適用します。
6. Swing は、別の商品の選択 (およびもう一度ルールを実行させる) または GUI の終了のいずれかのユーザー入力を待ちます。

図7.17 全ルールが実行された後のペットショップ例の GUI



IDE コンソールでイベントのこのフローを例示するには、他の **System.out** 呼び出しを追加します。

IDE コンソールの System.out 出力

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

7.7. 誠実な政治家のデシジョン例 (真理維持および顕著性)

誠実な政治家のデシジョンセットの例では、論理挿入を使用した真理維持の概念およびルールでの顕著性の使用方法を説明しています。

以下は、誠実な政治家の例の概要です。

- 名前: **honestpolitician**
- Main クラス: (src/main/java 内の)
org.drools.examples.honestpolitician.HonestPoliticianExample
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の)
org.drools.examples.honestpolitician.HonestPolitician.drl
- 目的: ファクトの論理挿入をもとにした真理維持の概念およびルールでの顕著性の使用方法を紹介しします。

誠実な政治家の例の前提として基本的に、ステートメントが True の場合にのみ、オブジェクトが存在できます。**insertLogical()** メソッドを使用して、ルールの結果により、オブジェクトを論理的に挿入します。つまり、論理的に挿入されたルールが True の状態であれば、オブジェクトは KIE セッションのワーキングメモリー内に留まります。ルールが True でなくなると、オブジェクトは自動的に取り消されます。

この例では、ルールを実行することで、企業による政治家の買収が原因で、政治家グループが「誠実」から「不誠実」に変わります。各政治家が評価されるにつれ、最初は **honesty** 属性を **true** に設定して開始しますが、ルールが実行されると政治家は「誠実」ではなくなります。状態が「誠実」から「不誠実」に切り替わると、ワーキングメモリーから削除されます。ルールの顕著性により、顕著性が定義されているルールをどのように優先付けするかを、デシジョンエンジンに通知します。そうでない場合には、デフォルトの顕著性の値 **0** を使用します。アクティベーションキューで順番待ちをしている場合には、顕著性の高いルールの優先順位が高くなります。

Politician および Hope クラス

この例の **Politician** クラス例は、誠実な政治家として設定されています。**Politician** クラスは、文字列アイテム **name** とブール値アイテム **honest** で構成されています。

Politician クラス

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

Hope クラスは、**Hope** オブジェクトが存在するかどうかを判断します。このクラスには意味を持つメンバーは存在しませんが、社会に希望がある限り、ワーキングメモリーに存在します。

Hope クラス

```
public class Hope {

    public Hope() {
```

```
}
}
```

政治家の誠実性に関するルール定義

誠実な政治家の例では、ワーキングメモリーに最低でも 1 名誠実な政治家が存在する場合には、**"We have an honest Politician"** ルールで論理的に新しい **Hope** オブジェクトを挿入します。すべての政治家が不誠実になると、**Hope** オブジェクトは自動的に取り除かれます。このルールでは、**salience** 属性の値が **10** となっており、他のルールより先に実行されます。理由は、この時点では **"Hope is Dead"** ルールが True となっているためです。

ルール "We have an honest politician"

```
rule "We have an honest Politician"
    salience 10
    when
        exists( Politician( honest == true ) )
    then
        insertLogical( new Hope() );
    end
```

Hope オブジェクトが存在すると、すぐに **"Hope Lives"** ルールが一致して実行されます。**"Corrupt the Honest"** ルールよりも優先されるように、このルールにも **salience** 値を **10** に指定しています。

ルール "Hope Lives"

```
rule "Hope Lives"
    salience 10
    when
        exists( Hope() )
    then
        System.out.println("Hurrah!!! Democracy Lives");
    end
```

最初は、誠実な政治家が 4 人いるので、このルールには 4 つのアクティベーションが存在し、すべてが競合しています。各ルールが順番に実行され、政治家が誠実でなくなるように、企業により各政治家を買収させていきます。政治家 4 人すべてが買収されたら、プロパティが **honest == true** の政治家はいなくなります。**"We have an honest Politician"** のルールは True でなくなり、論理的に挿入されるオブジェクト (最後に実行された **new Hope()** による) は自動的に取り除かれます。

ルール "Corrupt the Honest"

```
rule "Corrupt the Honest"
    when
        politician : Politician( honest == true )
        exists( Hope() )
    then
        System.out.println( "I'm an evil corporation and I have corrupted " + politician.getName() );
        modify ( politician ) { honest = false };
    end
```

真理維持システムにより **Hope** オブジェクトが自動的に取り除かれると、**Hope** に適用された条件付き要素 **not** は True でなくなり、**"Hope is Dead"** ルールが一致して実行されます。

ルール "Hope is Dead"

```
rule "Hope is Dead"
  when
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
  end
```

実行と監査証跡

HonestPoliticianExample.java クラスでは、honest の状態が **true** に設定されている政治家 4 人が挿入され、定義したビジネスルールに対して評価します。

HonestPoliticianExample.java クラスの実行

```
public static void execute( KieContainer kc ) {
    KieSession ksession = kc.newKieSession("HonestPoliticianKS");

    final Politician p1 = new Politician( "President of Umpa Lumpa", true );
    final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
    final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
    final Politician p4 = new Politician( "Omnipotence Om", true );

    ksession.insert( p1 );
    ksession.insert( p2 );
    ksession.insert( p3 );
    ksession.insert( p4 );

    ksession.fireAllRules();

    ksession.dispose();
}
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.honestpolitician.HonestPoliticianExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

この出力では、democracy lives に誠実な政治家が最低でも 1 人いることが分かります。ただし、各政治家は企業に買収されているので、全政治家は不誠実になり、民主性がなくなります。

この例の実行フローをさらに理解するために、**HonestPoliticianExample.java** クラスを変更し、**DebugRuleRuntimeEventListener** リスナーと監査ロガーを追加して実行の詳細を表示することができます。

監査ロガーを含む HonestPoliticianExample.java クラス

```

package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
import org.kie.api.event.rule.DebugAgendaEventListener; ❶
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
        KieServices ks = KieServices.Factory.get(); ❷
        //ks = KieServices.Factory.get();
        KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        //execute( kc );
        execute( ks, kc); ❸
    }

    public static void execute( KieServices ks, KieContainer kc ) { ❹
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        // The application can also setup listeners ❺
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger( ksession, "./target/honestpolitician" ); ❻

        ksession.fireAllRules();

        ksession.dispose();
    }
}

```

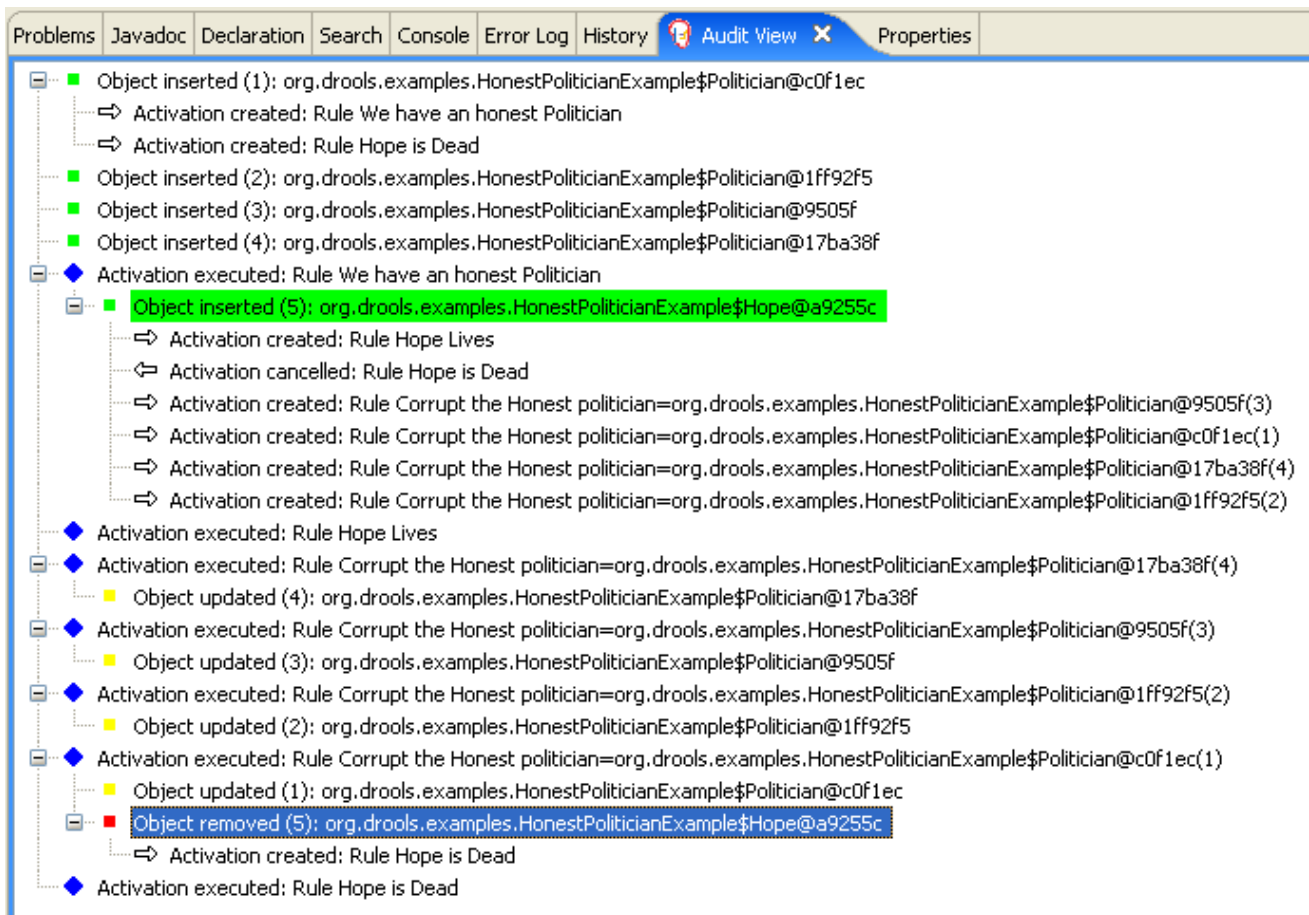
- ❶ **DebugAgendaEventListener** と **DebugRuleRuntimeEventListener** を処理するインポートパッケージに追加します。

- 2 この監査ログは **KieContainer** レベルでは利用できないので、**KieServices Factory** および **ks** 要素を作成してログを生成します。
- 3 **execute** メソッドを変更して **KieServices** と **KieContainer** 両方を使用します。
- 4 **execute** メソッドを変更して **KieContainer** に加えて **KieServices** で渡します。
- 5 リスナーを作成します。
- 6 ルールの実行後にデバッグビュー、監査ビュー または IDE に渡すことが可能なログを構築します。

ロギング機能を変更して、「誠実な政治家」のサンプルを実行すると、**target/honestpolitician.log** から IDE デバッグビューまたは 利用可能な場合には (IDE の一部では **Window → Show View**) **監査ビュー** に、監査ログファイルを読み込むことができます。

この例では、**監査ビュー** では、クラスやルールのサンプルで定義されているように、実行フロー、挿入、取り消しが示されています。

図7.18 誠実な政治家例の監査ビュー



最初の政治家が挿入されると、2つのアクティベーションが発生します。**"We have an honest Politician"** のルールは、**exists** の条件付き要素を使用するので、最初に挿入された政治家に対してのみ一度だけアクティベートされます。この条件付き要素は、政治家が最低でも1人挿入されると一致します。**Hope** オブジェクトがまだ挿入されていないので、ルール **"Hope is Dead"** もこの時点でアクティベートされます。**"We have an honest Politician"** ルールは、**"Hope is Dead"** ルールより、**salience** の値が高いので先に実行され、**Hope** オブジェクト (緑にハイライト) を挿入します。**Hope** オブジェクトを挿入すると、ルール **"Hope Lives"** がアクティベートされ、ルール **"Hope is Dead"** が無効になり

ます。この挿入により、挿入された誠実な各政治家に対して **"Corrupt the Honest"** ルールがアクティベートされます。**"Hope Lives"** のルールが実行されて、**"Hurrah!!! Democracy Lives"** が出力されます。

次に、政治家ごとに **"Corrupt the Honest"** ルールを実行して **"I'm an evil corporation and I have corrupted X"** と出力します。X には政治家の名前が入り、政治家の誠実性の値を **false** に変更します。最後の政治家が買収された時点で、真理維持システム (青でハイライト) により **Hope** が自動的に取り消されます。緑でハイライトされたエリアは、現在選択されている青のハイライトエリアの出元です。**Hope** ファクトが取り消されると、**"Hope is dead"** ルールが実行されて **"We are all Doomed!!! Democracy is Dead"** が出力されます。

7.8. 数独のデシジョン例 (複雑なパターン一致、コールバック、GUI 統合)

数独のデシジョンセットの例では、人気の数字パズルゲーム「数独」をもとにしています。このセットでは、Red Hat Process Automation Manager のルールを使用してさまざまな制約をもとに、多数の考えられる回答スペースの中で回答を導き出す方法を説明しています。また、この例では、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法も説明しています。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられたワーキングメモリ内の変更をもとに GUI を更新する方法も説明しています。

以下は数独の例の概要です。

- 名前: **sudoku**
- Main クラス: (src/main/java 内の) **org.drools.examples.sudoku.SudokuExample**
- モジュール: **drools-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の) **org.drools.examples.sudoku.*.drl**
- 目的: 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。

数独は、ロジックベースの数字配置パズルです。目的は、各列、行、および 3x3 ゾーンに 1 から 9 の数字が一度だけ含まれるように 9x9 のグリッドを埋めることです。パズルセッターでは、グリッド内の一部だけ記入されており、上記の制約ですべての空白を埋めるのがパズルの回答者のタスクです。

問題解決の一般的な戦略として、新しい番号の挿入時に、特定の 3x3 ゾーン、行、列で同じ番号がないことを確認します。この数独のデシジョンセットの例では、Red Hat Process Automation Manager ルールを使用して、さまざまな難易度の数独パズルを解き、無効なエントリーが含まれ、不備のあるパズルの解決を試みます。

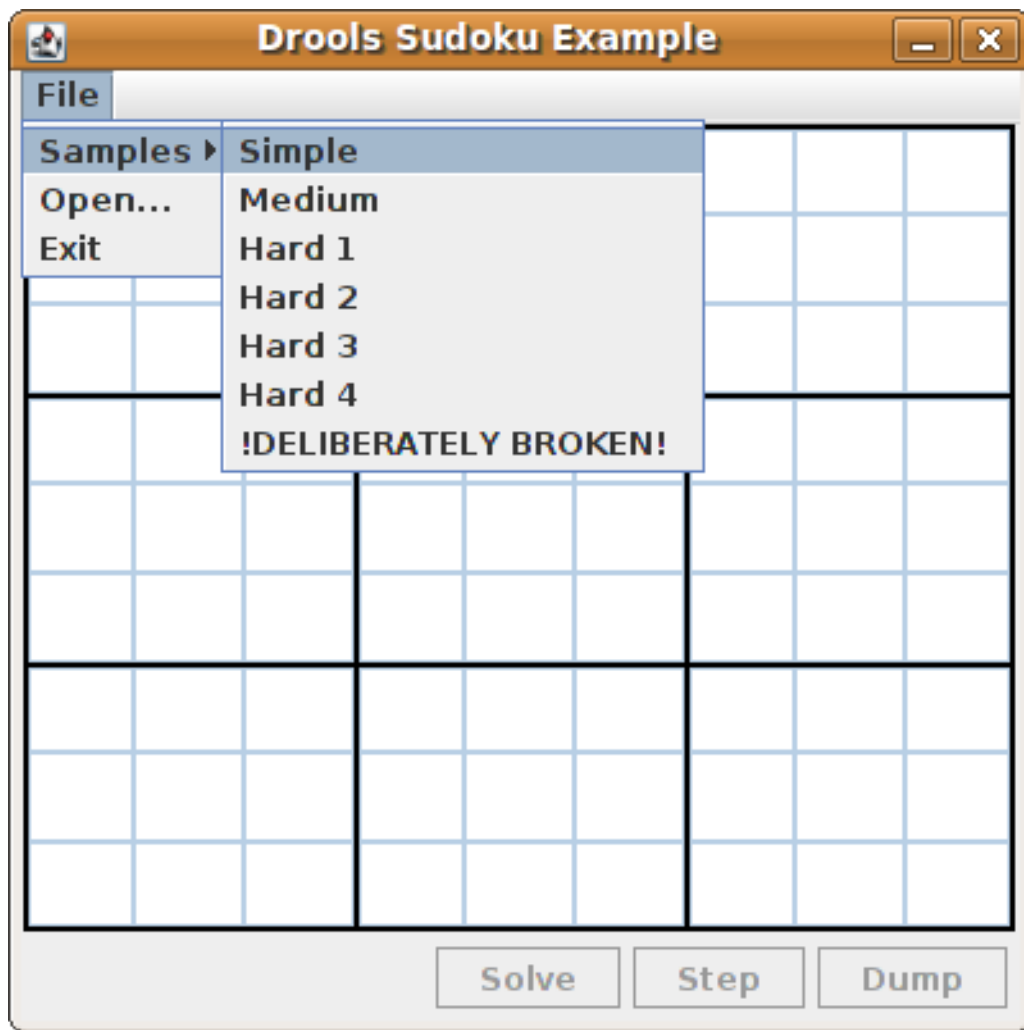
数独例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.sudoku.SudokuExample** クラスを Java アプリケーションとして実行し、数独の例を実行します。

数独の例を実行すると、**Drools Sudoku Example** GUI ウィンドウが表示されますこのウィンドウには、空白のグリッドが含まれますが、プログラムには、読み込みや解決が可能なグリッドが複数、内部に格納されています。

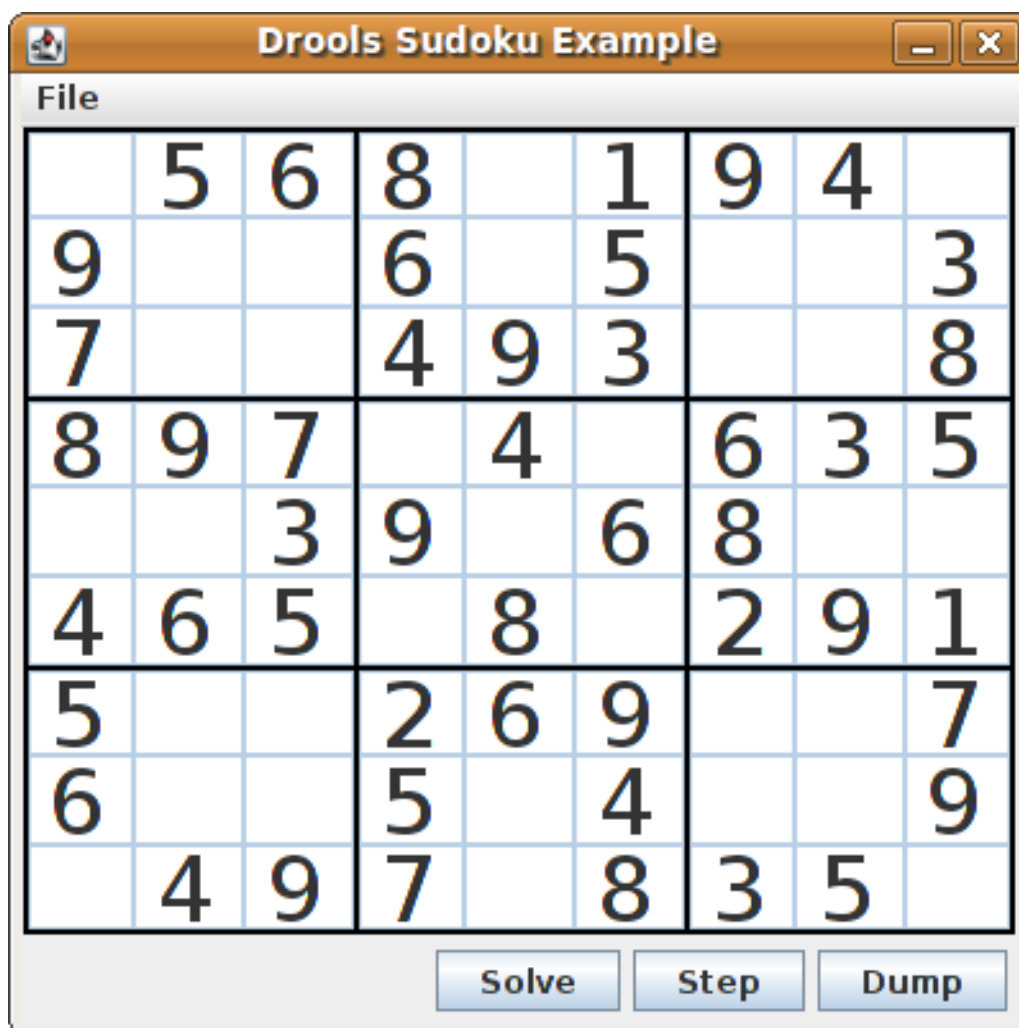
File → **Samples** → **Simple** をクリックして、例の 1 つを読み込みます。グリッドが読み込まれるまで、すべてのボタンが無効になっている点に注目してください。

図7.19 起動後の数独例の GUI



Simple サンプルを読み込むと、パズルの最初の状態に合わせて、グリッドが埋められます。

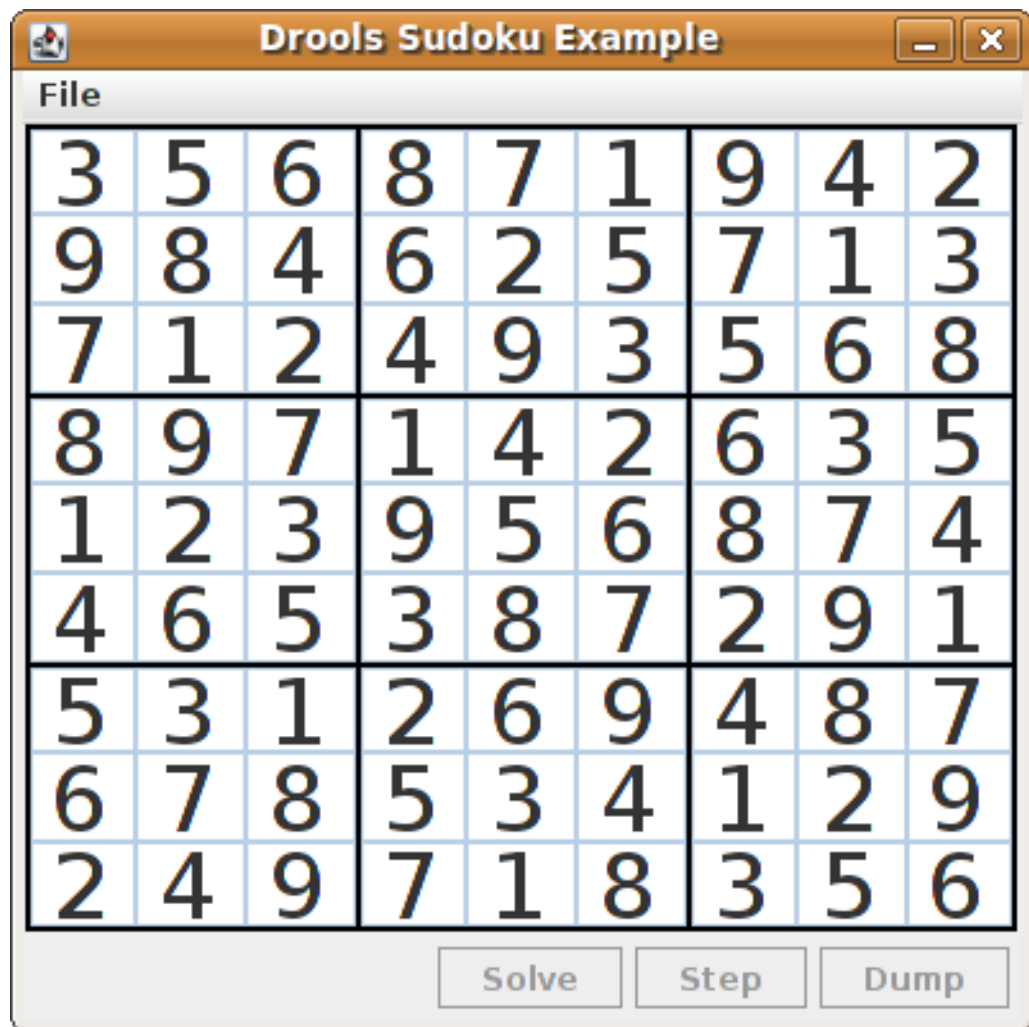
図7.20 Simple サンプルを読み込んだ後の数独例の GUI



以下のオプションから選択します。

- **Solve** をクリックして、数独の例に定義されているルールを実行し、残りの値を埋めていき、このボタンを再度無効にします。

図7.21 Simple サンプルの解決



- **Step** をクリックして、ルールセットに含まれる次の数字を表示します。IDE のコンソールウィンドウでは、解決手順を実行するルールに関する情報が詳細に表示されます。

IDE コンソールでの手順実行の出力

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```

- **Dump** をクリックしてグリッドの状態を表示します。セルには、解決済みの値か、残りの候補値が表示されます。

IDE コンソールでのダンプ実行の出力

```
Col: 0  Col: 1  Col: 2  Col: 3  Col: 4  Col: 5  Col: 6  Col: 7  Col: 8
Row 0: 123456789 --- 5 --- --- 6 --- --- 8 --- 123456789 --- 1 --- --- 9 --- --- 4 ---
123456789
Row 1: --- 9 --- 123456789 123456789 --- 6 --- 123456789 --- 5 --- 123456789
123456789 --- 3 ---
Row 2: --- 7 --- 123456789 123456789 --- 4 --- --- 9 --- --- 3 --- 123456789 123456789
--- 8 ---
```

```

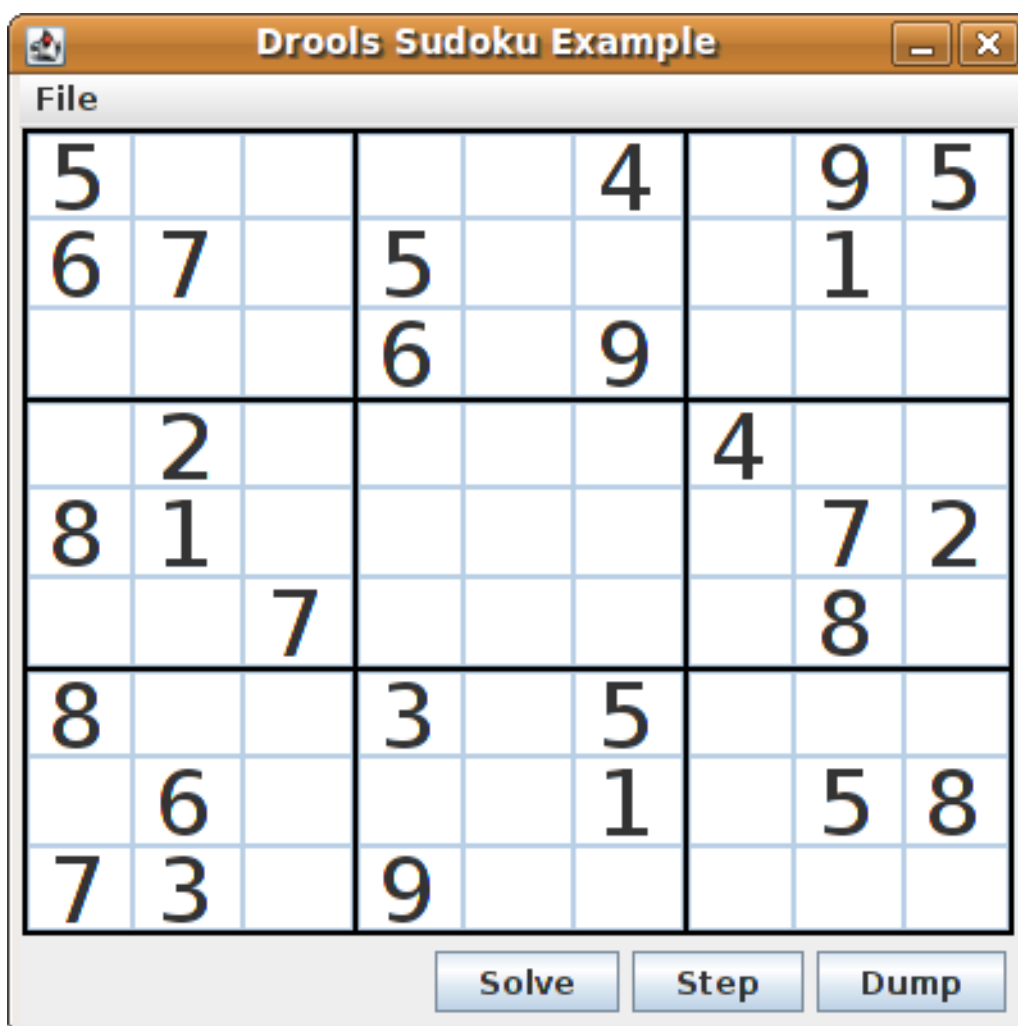
Row 3: --- 8 --- --- 9 --- --- 7 --- 123456789 --- 4 --- 123456789 --- 6 --- --- 3 --- --- 5 ---
Row 4: 123456789 123456789 --- 3 --- --- 9 --- 123456789 --- 6 --- --- 8 --- 123456789
123456789
Row 5: --- 4 --- --- 6 --- --- 5 --- 123456789 --- 8 --- 123456789 --- 2 --- --- 9 --- --- 1 ---
Row 6: --- 5 --- 123456789 123456789 --- 2 --- --- 6 --- --- 9 --- 123456789 123456789
--- 7 ---
Row 7: --- 6 --- 123456789 123456789 --- 5 --- 123456789 --- 4 --- 123456789
123456789 --- 9 ---
Row 8: 123456789 --- 4 --- --- 9 --- --- 7 --- 123456789 --- 8 --- --- 3 --- --- 5 ---
123456789

```

数独の例には、不備のあるサンプルファイルが意図的に含まれています。このファイルは、例で定義したルールを使用して解決できます。

File → **Samples** → **!DELIBERATELY BROKEN!** をクリックして、不備のあるサンプルを読み込みます。グリッドは、最初の行に **5** の値を 2 回表示できないにもかかわらず、表示されるなど、問題が含まれた状態で表示されます。

図7.22 不備のある数独例の最初の状態



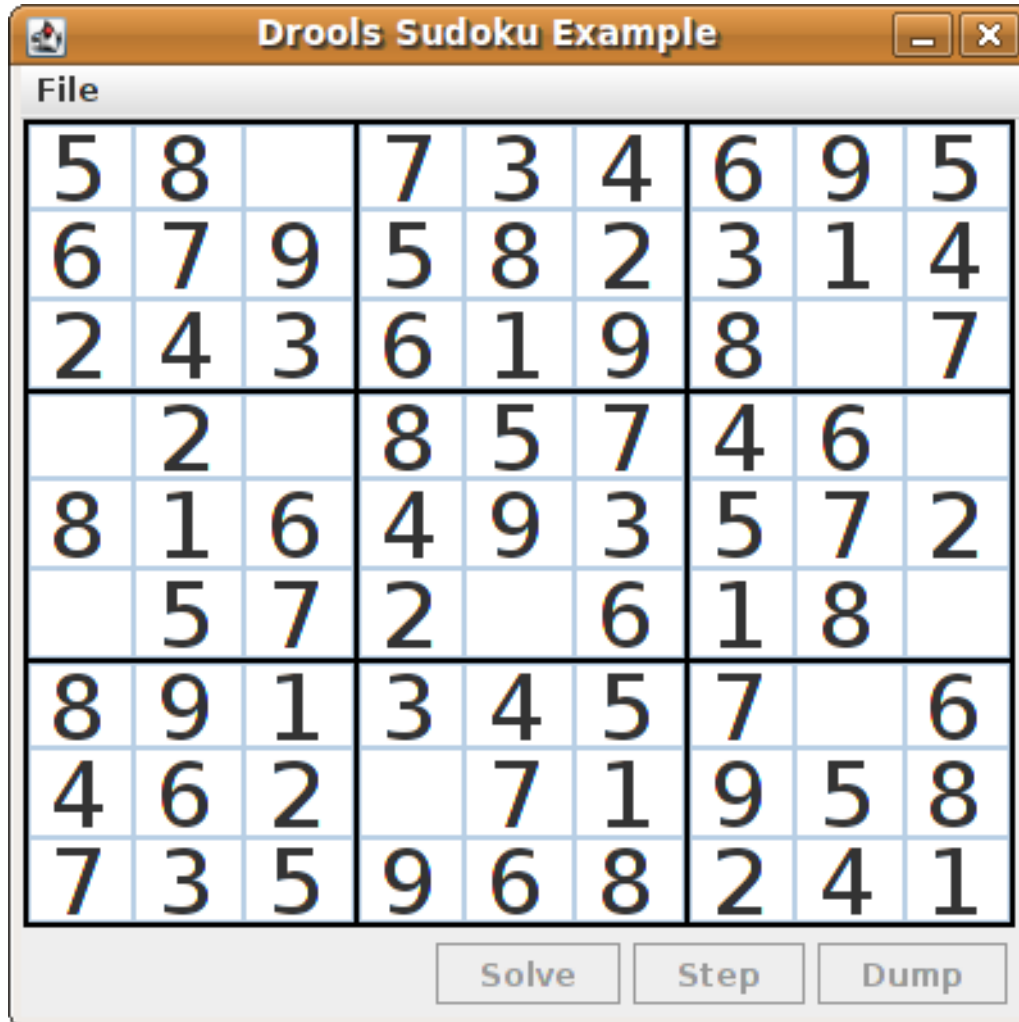
Solve をクリックしてこの無効なグリッドに解決ルールを適用します。数独の例に含まれる関連の解決ルールにより、サンプルの問題が検出され、できる限りパズル解決します。このプロセスでは、すべてを完了させず、空白のセルをいくつか残します。

解決ルールのアクティビティが IDE コンソールウィンドウに表示されます。

不備のあるサンプルでの問題検出

cell [0,8]: 5 has a duplicate in row 0
 cell [0,0]: 5 has a duplicate in row 0
 cell [6,0]: 8 has a duplicate in col 0
 cell [4,0]: 8 has a duplicate in col 0
 Validation complete.

図7.23 不備のあるサンプルの解決試行



Hard のラベルの付いた数独サンプルファイルはより複雑で、解決ルールを使用しても解決できない可能性があります。解決をしようとして失敗した場合には、IDE コンソールウィンドウに表示されます。

解決不可の Hard サンプル

Validation complete.

...

Sorry - can't solve this grid.

不備のあるサンプルを解決するためのルールでは、セルの候補となりえる値をもとにした標準の解決手法を実装します。たとえば、セットに値が1つ含まれる場合には、これが値になります。セルが9個あるグループの1つに値が1度挿入された場合に、ルールを使用して、特定のセルに対する値を持ち、タイプが **Setting** のファクトを挿入します。このファクトは、セルが含まれるグループの他のセルすべてからこの値を取り除き、この値を(選択肢から)取り消します。

この例の他のルールで、セルに入力可能な値を減らしていきます。**"naked pair"**、**"hidden pair in row"**、**"hidden pair in column"** および **"hidden pair in square"** のルールでは、候補の絞り込みはできますが、回答を得ることはできません。**"X-wings in rows"**、**"X-wings in columns"**、**"intersection**

removal row" および **"intersection removal column"** のルールは、より精緻な絞り込みを実行します。

数独例のクラス

org.drools.examples.sudoku.swing パッケージには、以下のように、数独パズルのフレームワークを実装する主なクラスセットが含まれます。

- **SudokuGridModel** は、9x9 グリッドの **Cell** オブジェクトとして数独パズルを格納するために実装可能なインターフェースを定義しています。
- **SudokuGridView** クラスは Swing コンポーネントで **SudokuGridModel** クラス実装の視覚化が可能です。
- **SudokuGridEvent** および **SudokuGridListener** クラスは、モデルとビュー間のステータスの変化をやり取りするために使用します。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。
- **SudokuGridSamples** クラスは、デモ目的に一部入力されている数独パズルを複数提供します。



注記

このパッケージには、Red Hat Process Automation Manager ライブラリーの依存関係は含まれません。

org.drools.examples.sudoku パッケージには、以下のように、基本的な **Cell** オブジェクトと各種アグリゲーションを実装する主なクラスセットが含まれます。

- **CellRow**、**CellCol** および **CellSqr** のサブクラスを含む **CellFile** クラス。これらすべては、**CellGroup** クラスのサブタイプになります。
- **Cell** と **CellGroup** は **SetOfNine** のサブクラスで、**Set<Integer>** 型の **free** プロパティを提供します。**Cell** クラスは、個別の候補セットを表します。**CellGroup** は、セルの全候補セットの統合 (割り当ての必要のある数値セット) です。
数独の例には、81 個の **Cell** と 27 個の **CellGroup** オブジェクト、**Cell** プロパティの **cellRow**、**cellCol** および **cellSqr** が提供するリンク、**CellGroup** プロパティ **cells** (**Cell** オブジェクトリスト) が提供するリストが含まれます。これらのコンポーネントを使用して、セルに値を割り当てたり、候補セットから値を取り除いたりできるように、特定の状態を検出するルールを記述できます。
- **Setting** クラスを使用して、値の割り当てに伴うオペレーションをトリガーします。**Setting** ファクトは、整合性の取れない中間の状態に対して反応しないように、新しい状況を検出する全ルールに配置して使用します。
- **Stepping** クラスは、優先順位が低いルールに使用して、**"Step"** が予期なく中断された場合に緊急停止を行います。この動作は、プログラムでパズルを解決できないということです。
- Main クラス **org.drools.examples.sudoku.SudokuExample** は、全コンポーネントを統合する Java アプリケーションを実装します。

数独の検証ルール (validate.drl)

数独の例の **validate.drl** ファイルには、セルグループで数が重複している状況を検出する検証ルールが含まれます。このグループは、**"validate"** アジェンダグループに統合され、ユーザーがパズルを読み込むと、明示的にルールをアクティベートできます。

"duplicate in cell ..." の 3 つのルールの **when** 条件はすべて以下の方法で機能します。

- このルールで最初の条件で、割り当てられた値でセルを特定します。
- このルールの 2 番目の条件では、3 つのセルグループのどれかを所属先にプルします。
- 最終条件は、ルールに従い、最初のセル、同じ行、列、または四角に入る値と同じセル (上記のセル以外) を検索します。

ルール "duplicate in cell ..."

```
rule "duplicate in cell row"
when
  $c: Cell( $v: value != null )
  $cr: CellRow( cells contains $c )
  exists Cell( this != $c, value == $v, cellRow == $cr )
then
  System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
end

rule "duplicate in cell col"
when
  $c: Cell( $v: value != null )
  $cc: CellCol( cells contains $c )
  exists Cell( this != $c, value == $v, cellCol == $cc )
then
  System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
end

rule "duplicate in cell sqr"
when
  $c: Cell( $v: value != null )
  $cs: CellSqr( cells contains $c )
  exists Cell( this != $c, value == $v, cellSqr == $cs )
then
  System.out.println( "cell " + $c.toString() + " has duplicate in its square of nine." );
end
```

ルール **"terminate group"** は最後に実行されます。このルールは、メッセージを出力して、シーケンスを停止します。

ルール "terminate group"

```
rule "terminate group"
  salience -100
when
then
  System.out.println( "Validation complete." );
  drools.halt();
end
```

数独の解決ルール (sudoku.drl)

数独の例の **sudoku.drl** ファイルには、3 種類の ルールタイプが含まれます。1 つ目のグループは、セルへの数値の割り当てを処理して、もう 1 つは実行可能な割り当てを検出して、3 つ目は候補セットからの値を削除します。

"set a value"、**"eliminate a value from Cell"** および **"retract setting"** のルールは、**Setting** オブジェ

クトの有無により左右されます。最初のルールは、セルへの割り当てと、3つのセルグループの **free** セットから値を削除する操作を処理します。また、ゼロの場合には、このグループはカウンターを1つ減らし、**fireUntilHalt()** を呼び出した Java アプリケーションに制御を戻します。

"eliminate a value from Cell" ルールの目的は、新たに割り当てられたセルに関連する全セルの候補リストを絞り込むことです。最後に、すべての除外が完了したら、**"retract setting"** ルールにより、トリガーされている **Setting** ファクトを取り消します。

ルール "set a value"、"eliminate a value from a Cell" および "retract setting"

```
// A Setting object is inserted to define the value of a Cell.
// Rule for updating the cell and all cell groups that contain it
rule "set a value"
when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // A matching Cell, with no value set
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,
              $cr: cellRow, $cc: cellCol, $cs: cellSqr )

    // Count down
    $ctr: Counter( $count: count )
then
    // Modify the Cell by setting its value.
    modify( $c ){ setValue( $v ) }
    // System.out.println( "set cell " + $c.toString() );
    modify( $cr ){ blockValue( $v ) }
    modify( $cc ){ blockValue( $v ) }
    modify( $cs ){ blockValue( $v ) }
    modify( $ctr ){ setCount( $count - 1 ) }
end

// Rule for removing a value from all cells that are siblings
// in one of the three cell groups
rule "eliminate a value from Cell"
when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )

    // For all Cells that are associated with the updated cell
    $c: Cell( free contains $v ) from $exCells
then
    // System.out.println( "clear " + $v + " from cell " + $c.posAsString() );
    // Modify a related Cell by blocking the assigned value.
    modify( $c ){ blockValue( $v ) }
end

// Rule for eliminating the Setting fact
rule "retract setting"
when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )
```

```

// The matching Cell, with the value already set
$c: Cell( rowNo == $rn, colNo == $cn, value == $v )

// This is the negation of the last pattern in the previous rule.
// Now the Setting fact can be safely retracted.
not( $x: Cell( free contains $v )
    and
    Cell( this == $c, exCells contains $x ) )
then
// System.out.println( "done setting cell " + $c.toString() );
// Discard the Setter fact.
delete( $s );
// Sudoku.sudoku.consistencyCheck();
end

```

解決ルール 2 つを使用して、セルに数字を割り当てることができる状況を検出します。**"single"** のルールは、**Cell** に、数字が 1 つだけの候補セットが含まれる場合に実行されます。**"hidden single"** ルールは、候補が 1 つだけのセルが存在しない場合に実行されますが、セルに候補が含まれる場合には、セルが所属する 3 つのグループの 1 つに含まれるその他すべてのセルに、この候補が存在しないということです。いずれのルールも **Setting** ファクトを作成して、挿入します。

ルール "single" および "hidden single"

```

// Detect a set of candidate values with cardinality 1 for some Cell.
// This is the value to be set.
rule "single"
when
    // Currently no setting underway
    not Setting()

    // One element in the "free" set
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )
then
    Integer i = $c.getFreeValue();
    if (explain) System.out.println( "single " + i + " at " + $c.posAsString() );
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, i ) );
end

// Detect a set of candidate values with a value that is the only one
// in one of its groups. This is the value to be set.
rule "hidden single"
when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Some integer
    $i: Integer()

    // The "free" set contains this number
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )

    // A cell group contains this cell $c.
    $cg: CellGroup( cells contains $c )

```

```
// No other cell from that group contains $i.
not ( Cell( this != $c, free contains $i ) from $cg.getCells() )
then
  if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
  // Insert another Setter fact.
  insert( new Setting( $rn, $cn, $i ) );
end
```

最大グループからのルール (個別または 2 - 3 のグループ単位) は、数独パズルを手作業で解決するのに使用する、さまざまな解決手法を実装します。

"naked pair" ルールは、グループの 2 つのセルで、全く同じ候補セットでサイズ **2** のものを検出します。これらの 2 つの値は、対象グループの他の候補セットすべてから削除することができます。

ルール "naked pair"

```
// A "naked pair" is two cells in some cell group with their sets of
// permissible values being equal with cardinality 2. These two values
// can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )

    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )

    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 /** , rowNo >= $rn1, colNo >= $cn1 **/ ) from $cg.cells

    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()

    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
  then
    if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair
    at " + $c1.posAsString() + " and " + $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v ) }
  end
```

3 番目のルール **"hidden pair in ..."** は、ルール **"naked pair"** と同様に機能します。ルールはグループの 2 つのセルで 2 つの数字を検出します。どの値もこのグループの他のセルには入りません。つまり、他の候補はすべて、隠れたペアを持つ 2 つのセルから削除します。

ルール "hidden pair in ..."

```
// If two cells within the same cell group contain candidate sets with more than
// two values, with two values being in both of them but in none of the other
// cells, then we have a "hidden pair". We can remove all other candidates from
```



```

// these two cells.
rule "hidden pair in row"
when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Establish a pair of Integer facts.
    $i1: Integer()
    $i2: Integer( this > $i1 )

    // Look for a Cell with these two among its candidates. (The upper bound on
    // the number of candidates avoids a lot of useless work during startup.)
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
    $cellRow: cellRow )

    // Get another one from the same row, with the same pair among its candidates.
    $c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free contains $i1 && contains $i2 )

    // Ascertain that no other cell in the group has one of these two values.
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellRow.getCells() )
then
    if( explain) System.out.println( "hidden pair in row at " + $c1.posAsString() + " and " +
    $c2.posAsString() );
    // Set the candidate lists of these two Cells to the "hidden pair".
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
    $cellCol: cellCol )
    $c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellCol.getCells() )
then
    if( explain) System.out.println( "hidden pair in column at " + $c1.posAsString() + " and " +
    $c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
    $cellSqr: cellSqr )

```

```

    $c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellSqr.getCells() )
    then
        if (explain) System.out.println( "hidden pair in square " + $c1.posAsString() + " and " +
            $c2.posAsString() );
        modify( $c1 ){ blockExcept( $i1, $i2 ) }
        modify( $c2 ){ blockExcept( $i1, $i2 ) }
    end

```

2つのルールは行と列で **"X-wings"** を処理します。2つの異なる行 (または列) で、ある値を入力できるセルが2つしかなく、これらの候補が同じ列 (または行) に入る場合に、この列 (または行) のこの値に対する他の候補は除外できます。これらのルールの1つに含まれるパターンシーケンスに従うと、**same** または **only** などの用語で都合よく表現されている条件は、適切な制約が付けられたパターンになるか、**not** の接頭辞が付きます。

ルール "X-wings in ..."

```

rule "X-wings in rows"
when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
        $ra: cellRow, $rano: rowNo,    $c1: cellCol,    $c1no: colNo )
    $cb1: Cell( freeCount > 1, free contains $i,
        $rb: cellRow, $rbno: rowNo > $rano,    cellCol == $c1 )
    not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )

    $ca2: Cell( freeCount > 1, free contains $i,
        cellRow == $ra, $c2: cellCol,    $c2no: colNo > $c1no )
    $cb2: Cell( freeCount > 1, free contains $i,
        cellRow == $rb,    cellCol == $c2 )
    not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )

    $cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
        freeCount > 1, free contains $i )
then
    if (explain) {
        System.out.println( "X-wing with " + $i + " in rows " +
            $ca1.posAsString() + " - " + $cb1.posAsString() +
            $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
        $c1: cellCol, $c1no: colNo,    $ra: cellRow,    $rano: rowNo )
    $ca2: Cell( freeCount > 1, free contains $i,
        $c2: cellCol, $c2no: colNo > $c1no,    cellRow == $ra )

```

```

not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

$cb1: Cell( freeCount > 1, free contains $i,
           cellCol == $c1, $rb: cellRow, $rbno: rowNo > $rano )
$cb2: Cell( freeCount > 1, free contains $i,
           cellCol == $c2,   cellRow == $rb )
not( Cell( this != $cb1 && != $cb2, free contains $i ) from $rb.getCells() )

$cx: Cell( colNo == $c1no || == $c2no, rowNo != $rano && != $rbno,
           freeCount > 1, free contains $i )
then
  if (explain) {
    System.out.println( "X-wing with " + $i + " in columns " +
                        $ca1.posAsString() + " - " + $ca2.posAsString() +
                        $cb1.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
  }
  modify( $cx ){ blockValue( $i ) }
end

```

"intersection removal ..." の 2 つのルールは、1 つの四角の中に (1 つの行または列) 使用できる数字を制限するというルールに基づいています。つまり、この番号は行または列の中の 2-3 セルの 1 つに入っていないといけません。グループの別のセルすべての中にある候補セットから削除できます。このパターンは、発生制限を確立して、同じセルファイルの中かつ、四角の外のセルそれぞれに対して実行されます。

ルール "intersection removal ..."

```

rule "intersection removal column"
when
  not Setting()
  not Cell( freeCount == 1 )

  $i: Integer()
  // Occurs in a Cell
  $c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
  // Does not occur in another cell of the same square and a different column
  not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc )

  // A cell exists in the same column and another square containing this value.
  $cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr != $cs )
then
  // Remove the value from that other cell.
  if (explain) {
    System.out.println( "column elimination due to " + $c.posAsString() +
                        ": remove " + $i + " from " + $cx.posAsString() );
  }
  modify( $cx ){ blockValue( $i ) }
end

rule "intersection removal row"
when
  not Setting()
  not Cell( freeCount == 1 )

  $i: Integer()
  // Occurs in a Cell

```

```

$c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
// Does not occur in another cell of the same square and a different row.
not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr )

// A cell exists in the same row and another square containing this value.
$cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr != $cs )
then
// Remove the value from that other cell.
if (explain) {
    System.out.println( "row elimination due to " + $c.posAsString() +
        ": remove " + $i + " from " + $cx.posAsString() );
}
modify( $cx ){ blockValue( $i ) }
end

```

これらのルールは、すべてではありませんが、多くの数独パズルでは十分です。非常に難度の高いグリッドを解決するには、ルールセットにはさらに複雑なルールが必要です (最終的には、パズルは試行錯誤でしか解決できません)。

7.9. CONWAY の GAME OF LIFE のデシジョン例 (ルールフローグループおよび GUI 統合)

John Conway による有名なセルオートマトン (CA: Cellular automation) をベースにした Conway の Game of Life のデシジョンセットの例では、ルールでルールフローグループを使用してルール実行を制御する方法を説明しています。また、この例では、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法も説明しています。今回は、Conway の Game of Life を Swing ベースで実装しています。

以下は、Conway の Game of Life の例の概要です。

- 名前: **conway**
- Main クラス: (src/main/java 内の) **org.drools.examples.conway.ConwayRuleFlowGroupRun**、**org.drools.examples.conway.ConwayAgendaGroupRun**
- モジュール: **droolsjbpm-integration-examples**
- タイプ: Java アプリケーション
- ルールファイル: (src/main/resources 内の) **org.drools.examples.conway.*.drl**
- 目的: ルールフローグループと GUI 統合を例示します。



注記

Conway の Game of Life の例は、Red Hat Process Automation Manager に含まれる他のデシジョンセットの例の多くとは異なり、[Red Hat カスタマーポータル](#) から取得する **Red Hat Process Automation Manager 7.7.0 Source Distribution** の `~/rhpam-7.7.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples` に配置されています。

Conway の Game of Life では、初期設定または定義済みのプロパティで高度なパターンを作成して、初期状態からどのように進化していくかを観察することで、ユーザーはゲームと対話します。ゲームの目的は、世代ごとに人口の成長を表示します。各世代は、すべてのセル (細胞) が同時に進化していき、

前の世代をもとにして生み出されます。

以下の基本的なルールで、次の世代がどのようなになるかを制御していきます。

- 生きているセルの近傍に、生きているセルが 2 個未満の場合は、孤独で死んでしまう。
- 生きているセルの近傍に、生きているセルが 4 個以上ある場合は、過密で死んでしまう。
- 死亡したセルの近傍に、生きているセルがちょうど 3 つある場合には、このセルは生き返る。

この基準のいずれも満たさないセルは、そのまま次の世代に残ります。

Conway の Game of Life の例は、**ruleflow-group** 属性が含まれる Red Hat Process Automation Manager ルールで、ゲームに実装されているパターンを定義します。この例には、アジェンダグループを使用して同じ動作を行うデシジョンセットのバージョンも含まれています。アジェンダグループは、デシジョンエンジンアジェンダをパーティションして、ルールのグループを実行制御できるようにします。デフォルトでは、すべてのルールがアジェンダグループ **MAIN** に含まれています。ルールに異なるアジェンダグループを指定するには、**agenda-group** 属性を使用できます。

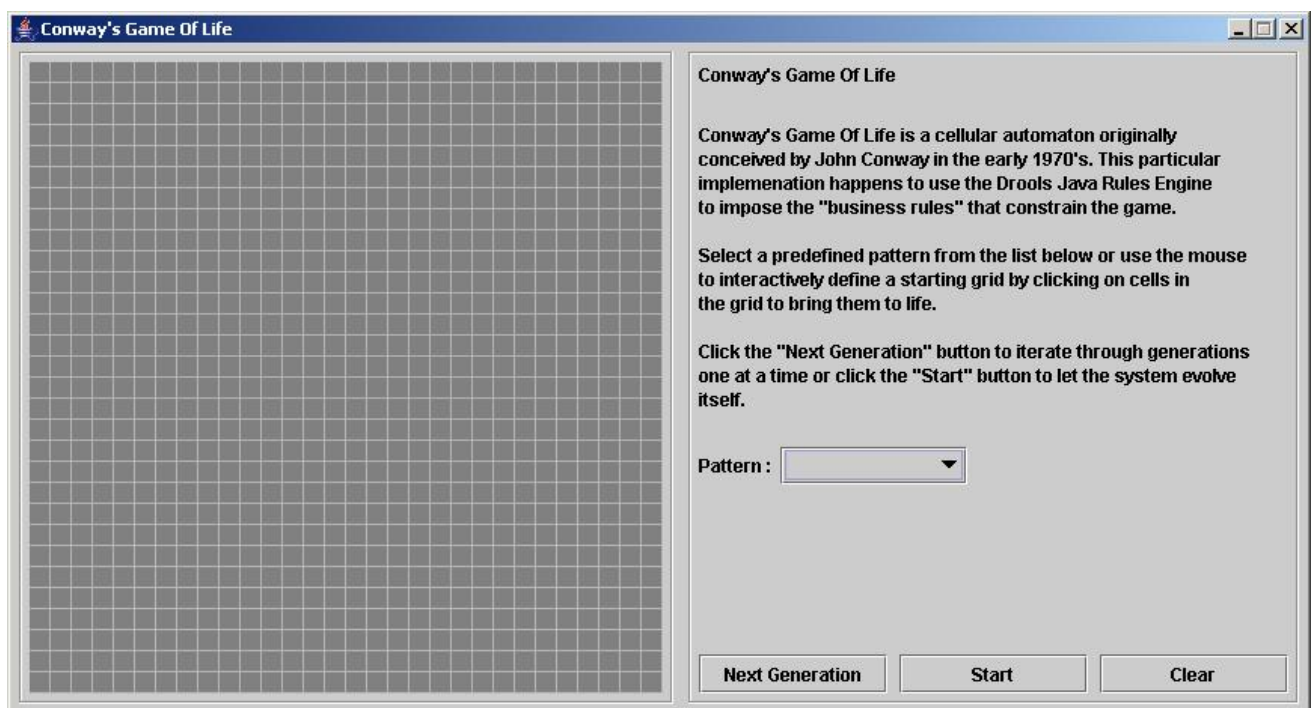
この概要では、Conway の例でアジェンダグループを使用したバージョンには触れません。アジェンダグループの詳細情報は、特にアジェンダグループについて対応している Red Hat Process Automation Manager のデシジョンセットの例を参照してください。

Conway 例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.conway.ConwayRuleFlowGroupRun** クラスを Java アプリケーションとして実行し、Conway の例を実行します。

Conway の例を実行すると、**Conway's Game of Life** GUI ウィンドウが表示されます。このウィンドウには、空のグリッドまたは "アリーナ" が含まれており、ここで生命のシミュレーションが行われます。システムにまだ生きているセルが含まれていないので、グリッドは最初は空白です。

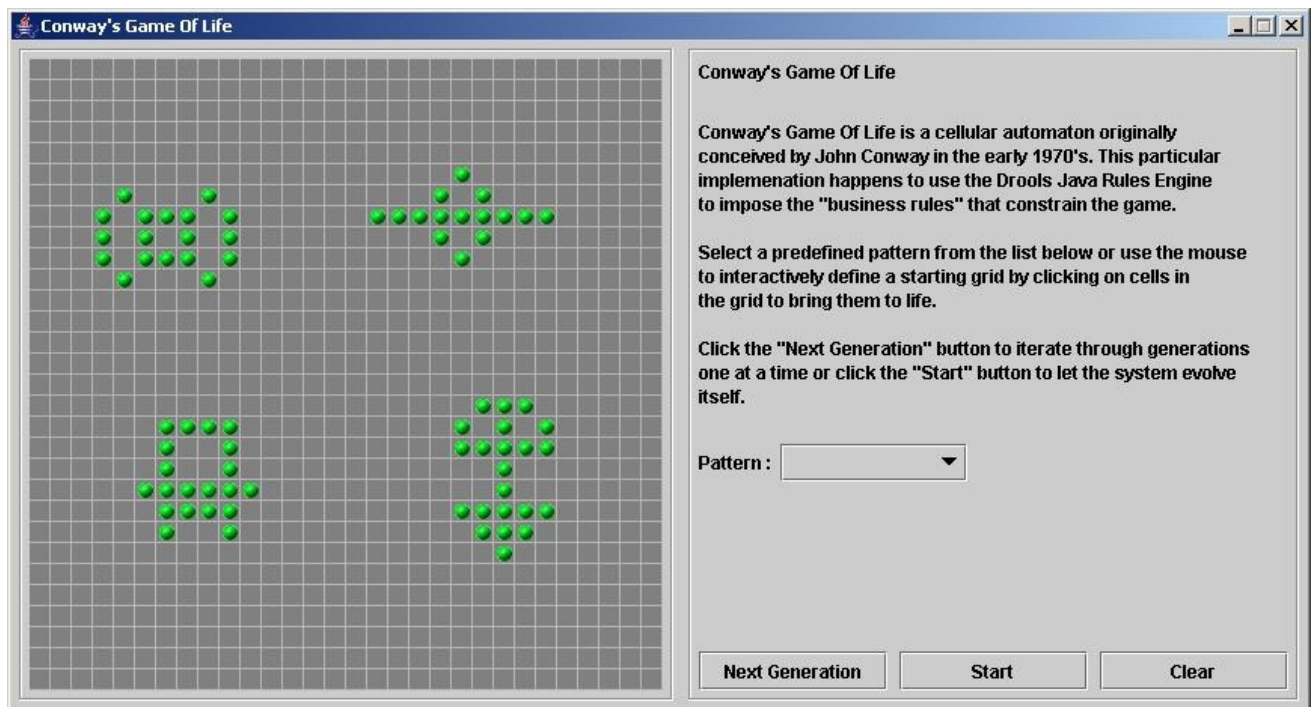
図7.24 起動後における Conway の例の GUI



パターンのドロップダウンメニューから事前定義済みのパターンを選択して、**次の世代** をクリックし、各人口の世代をクリックしていきます。セルは生きているか、死んでいるかのどちらかで、生きて

いるセルには緑のボールが含まれます。最初のパターンから人口が進化するにつれ、ゲームのルールをもとに、セルが近傍のセルに合わせて、生存するか、死亡していきます。

図7.25 Conway の例の世代進化



近傍には、上下左右のセルだけでなく対角線上につながっているセルも含まれるので、各セルには合計 8 つの近傍があります。例外は、角のセルと 4 辺上にあるセルで、それぞれ順に近傍が 3 つだけと、5 つだけになります。

セルをクリックすることで手動で介入して、セルを作成することも、死亡させることもできます。

最初のパターンから自動的に進化を実行するには、**スタート** をクリックします。

ルールグループを使用する Conway 例のルール

ConwayRuleFlowGroupRun の例のルールは、ルールフローグループを使用して、ルール実行を制御します。ルールフローグループは、**ruleflow-group** ルール属性に関連付けられたルールのグループです。これらのルールは、このグループがアクティベートされたときにしか実行されません。グループ自体は、ルールフローの図の詳細がグループを表すノードに到達してからでないと、有効化されません。

Conway の例では、ルールに以下のルールフローグループを使用します。

- "register neighbor"
- "evaluate"
- "calculate"
- "reset calculate"
- "birth"
- "kill"
- "kill all"

Cell オブジェクトはすべて、KIE セッションに挿入され、**"register neighbor"** ルールフローグループの **"register ..."** ルールがルールフロープロセスにより実行できるようになります。4 つのルールが含まれ

るこのグループは、セル同士の **Neighbor** の関係と、北東、北、北西、西の近傍との Neighbour の関係を作り出します。

この関係は双方向で、他の 4 方向を処理します。各辺上のセルは、特別な対応は必要ありません。これらのセルは、近傍のセルがなければペアは作成されません。

これらのルールに対して、すべてのアクティベーションが実行されるまで、全セルは、近傍の全セルと関係があります。

ルール "register ..."

```
rule "register north east"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row == ($row - 1), col == ( $col + 1 ) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
  end

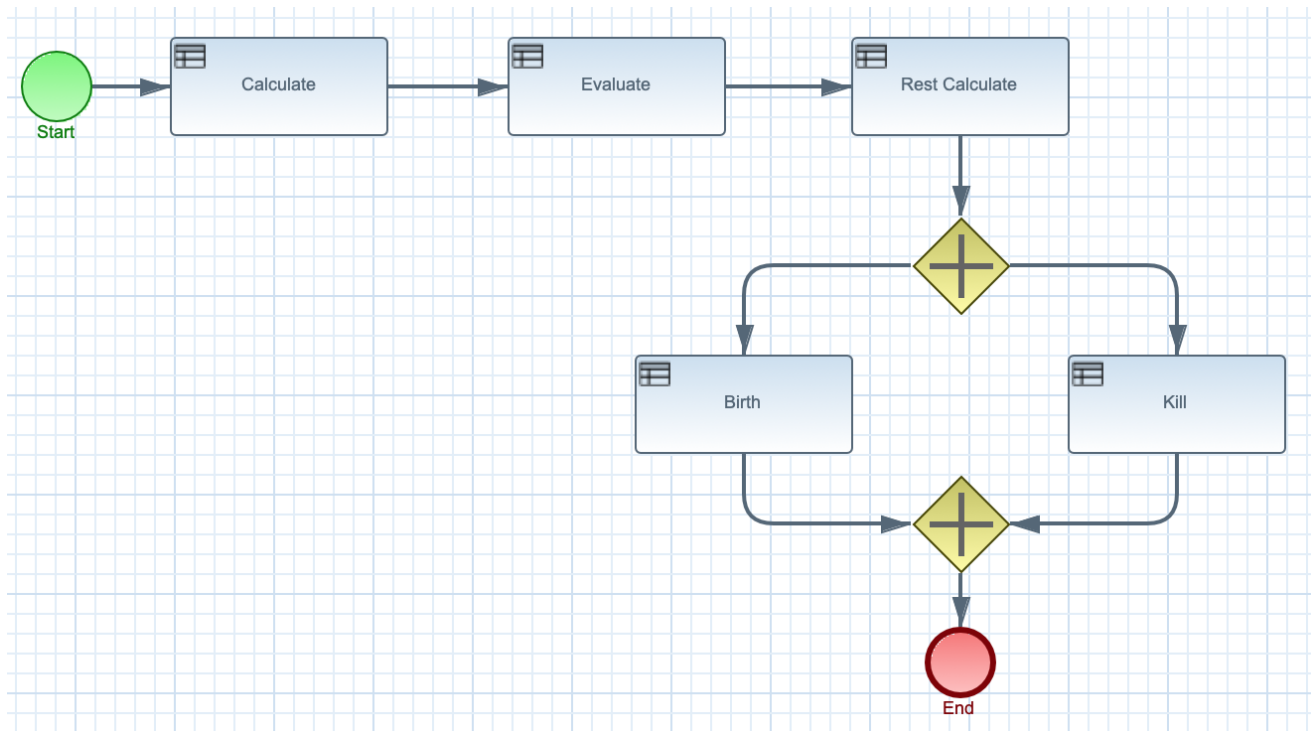
rule "register north"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
  end

rule "register north west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
  end

rule "register west"
  ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
  end
```

全セルが挿入されたら、Java コードはグリッドにパターンを適用し、特定のセルを **Live** に設定します。次に、ユーザーが **スタート** または **次の世代** をクリックすると、**Generation** のルールフローが実行されます。このルールフローは、世代のサイクルごとにセルの変更をすべて管理します。

図7.26 世代のルールフロー



ルールフロープロセスは、実行可能なグループに **"evaluate"** ルールフローグループおよび有効なルールを追加します。このグループの **"Kill the ..."** と **"Give Birth"** ルールを使用して、細胞の誕生または死亡セルにゲームのルールを適用します。この例では、**phase** 属性を使用して、特定のルールグループで **Cell** オブジェクトの理由付けをトリガーします。通常は、**phase** はルールフロープロセスの定義に含まれるルールフローグループに紐づけされています。

この例では、変更の適用前に評価を完全に完了しておく必要があるため、この時点では **Cell** オブジェクトの状態は変更されません。細胞の **phase** を **Phase.KILL** または **Phase.BIRTH** に適用し、後ほど **Cell** オブジェクトに適用されたアクションを制御するのに使用します。

ルール "Kill the ..." および "Give Birth"

```

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
    modify( theCell ){
      setPhase( Phase.KILL );
    }
  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

```



```

    modify( theCell ){
        setPhase( Phase.KILL );
    }
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
        modify( theCell ){
            theCell.setPhase( Phase.BIRTH );
        }
    end
end

```

グリッド内の全 **Cell** オブジェクトが評価されると、この例では **"reset calculate"** ルールを使用して **"calculate"** ルールフローグループのアクティベーションを消去します。次に、ルールグループがアクティベートされると、**"kill"** と **"birth"** のルールを有効にするルールフローに分岐を挿入します。これらのルールにより状態の変更が適用されます。

ルール "reset calculate"、"kill" および "birth"

```

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
end

```

この段階では、複数の **Cell** オブジェクトの状態が **LIVE** または **DEAD** のいずれかに変更されています。この例では、細胞が生存または死亡すると、**"Calculate ..."** ルールの **Neighbor** 関係を使用して、周辺の細胞すべてに繰り返し実行し、**liveNeighbor** の数が増減します。数が変更された細胞は、**EVALUATE** フェーズに設定され、ルールフロー処理の評価段階の理由付けに含まれるようになります。

生存数が判断され、全細胞に設定されると、ルールフロー処理が終了します。ユーザーが最初に **スタート** をクリックすると、その時点でデシジョンエンジンにより、ルールフローが再起動され、ユーザーが最初に **次の世代** をクリックした場合には、ユーザーは別の世代を要求することができます。

ルール "Calculate ..."

```
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
```

7.10. HOUSE OF DOOM のデシジョン例 (後向き連鎖および再帰)

The House of Doom のデシジョンセットの例では、デシジョンエンジンが後ろ向き連鎖と再帰を使用して、階層システムで定義した目的やサブゴールに到達する方法を説明しています。

以下は House of Doom の例の概要です。

- 名前: **backwardchaining**
- Main クラス: (src/main/java 内の)
org.drools.examples.backwardchaining.HouseOfDoomMain
- モジュール: **drools-examples**
- タイプ: Java アプリケーション

- **ルールファイル:** (src/main/resources 内の) **org.drools.examples.backwardchaining.BC-Example.drl**
- **目的:** 後向き連鎖と再帰を例示します。

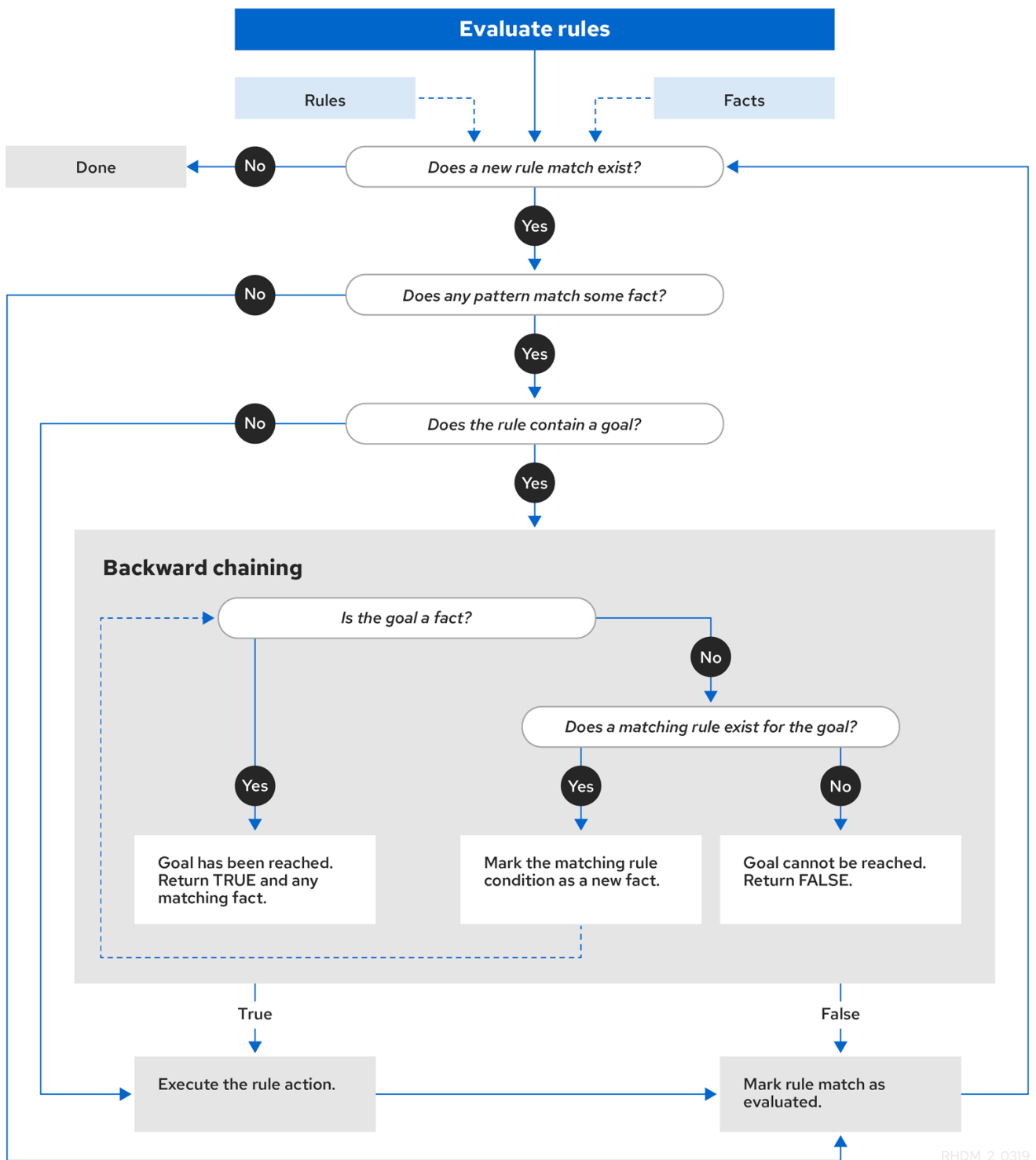
後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

反対に、前向き連鎖のルールシステムは、デシジョンエンジンのワーキングメモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトがワーキングメモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

Red Hat Process Automation Manager のデシジョンエンジンは、前向き連鎖と後向き連鎖の両方を使用してルールを評価します。

以下の図は、デシジョンエンジンが、ロジックフローで後向き連鎖のセグメントと、前向き連鎖全体とを使用してルールを評価する方法を例示します。

図7.27 前向き連鎖と後向き連鎖を使用したルール評価のロジック



RHDM_2_0319

House of Doom の例は、さまざまなクエリタイプが含まれるルールを使用し、部屋の場所と家の中のアイテムを探し出します。**Location.java** のサンプルクラスには、この例で使用する **item** と **location** 要素が含まれます。**HouseOfDoomMain.java** のサンプルクラスで、家の該当の場所にアイテムまたは部屋を挿入して、ルールを実行します。

HouseOfDoomMain.java クラスでのアイテムと場所

```

ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );

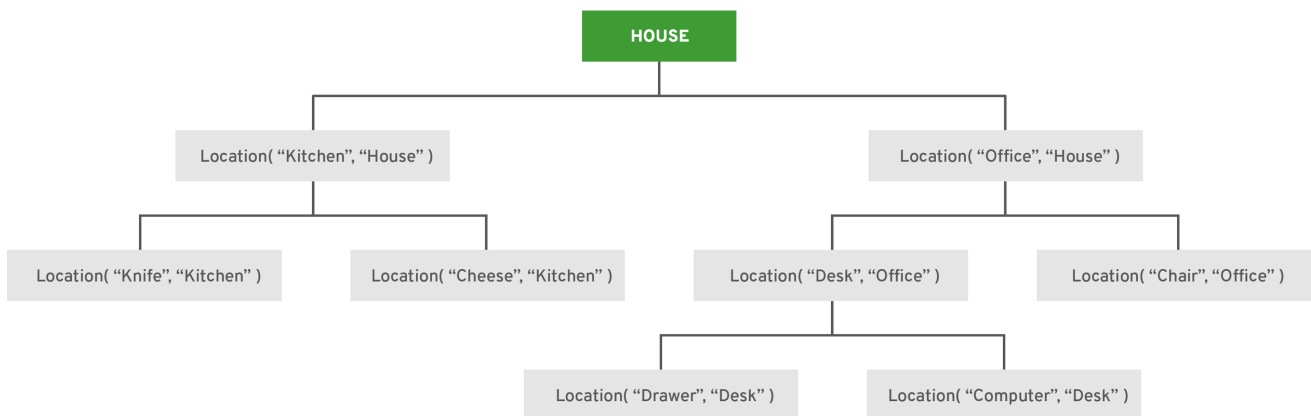
```

```
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

ルールの例では、家の構造の中で全アイテムおよび部屋の場所を判断するのに、後向き連鎖と再帰を使用します。

以下の図は、House of Doom の構造と、その構造内のアイテムと部屋を示しています。

図7.28 House of Doom の構造



RHDM_2_0319

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.backwardchaining.HouseOfDoomMain** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
Key is in the Office
---
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
```

```

Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk

```

この例のルールはすべて実行されて、家の全アイテムの場所を検出し、家の中の全アイテムの場所を検出して、出力でそれぞれの場所を出力します。

再帰クエリーおよび関連のルール

再帰クエリーは、要素間の関係におけるデータ構造階層を使用して繰り返し検索を行います。

House of Doom の例では、**BC-Example.drl** ファイルに、この例のルールの大半が使用する **isContainedIn** クエリーが含まれており、家のデータ構造を再帰的に評価して、デシジョンエンジンに挿入するデータがないかを確認します。

BC-Example.drl の再帰クエリー

```

query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end

```

"go" のルールは、システムに挿入する文字列をすべて出力し、アイテムをどのように導入し、"go1" ルールが **isContainedIn** クエリーを呼び出すかを判断します。

ルール "go" および "go1"

```

rule "go" salience 10
  when
    $s : String()
  then
    System.out.println( $s );
  end

rule "go1"
  when
    String( this == "go1" )
    isContainedIn("Office", "House"; )
  then
    System.out.println( "Office is in the House" );
  end

```

この例は、"**go1**" 文字列をデシジョンエンジンに挿入して、"**go1**" ルールを有効にし、**House** の場所にある **Office** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

IDE コンソールでのルール "**go1**" の出力

```
go1
Office is in the House
```

推移閉包ルール

推移閉包は、階層構造で複数レベル、上層にある親要素に含まれる要素間の関係です。

"**go2**" ルールは、**Drawer** と **House** の推移閉包の関係を特定します。**Drawer** は、**House** の中の、**Office** の中の、**Desk** の中にあります。

```
rule "go2"
when
    String( this == "go2" )
    isContainedIn("Drawer", "House");
then
    System.out.println( "Drawer is in the House" );
end
```

この例は、"**go2**" 文字列をデシジョンエンジンに挿入して、"**go2**" ルールを有効化し、最終的に **House** の場所に含まれる **Drawer** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

IDE コンソールのルール "**go2**" の出力

```
go2
Drawer is in the House
```

デシジョンエンジンは、以下のロジックをもとにこの結果を判断します。

1. クエリーは再帰的に、家の中の複数レベルを検索して、**Drawer** と **House** の間の推移閉包を検出します。
2. **Drawer** は **House** に直接含まれないので、**Location(x, y;)** を使用する代わりに、このクエリーは **(z, y;)** の値を使用します。
3. **z** の引数は現在バインドされておらず、値が指定されていないので、引数に含まれるものはすべて返されます。
4. **y** の引数は現在、**House** にバインドされているので、**z** は **Office** と **Kitchen** を返します。

- クエリーは、**Office** からの情報を収集して、**Drawer** が **Office** に含まれているかを再帰的にチェックします。これらのパラメーターに対して、クエリーの行 `isContainedIn(x, z;)` が呼び出されます。
- Office** に直接含まれる **Drawer** が存在しないので、一致するものではありません。
- z** のバインドがない場合は、このクエリーでは **Office** 内のデータが返され、`z == Desk` と判断されます。

```
isContainedIn(x==drawer, z==desk)
```

- isContainedIn** クエリーは再帰的に 3 回検索し、3 回目に、このクエリーにより **Desk** の中に **Drawer** があることが検出されます。

```
Location(x==drawer, y==desk)
```

- 最初の場所で上記が一致した後に、このクエリーにより再帰的に構造を上方向に検索し、**Drawer** が **Desk** の中に、**Desk** が **Office** の中に、**Office** が **House** の中にあることを判断します。このように、**Drawer** は **House** の中にあるので、このルールは満たされます。

リアクティブクエリールール

リアクティブクエリーでは、データ構造の階層を検索して、要素間に関係があるかを確認し、構造内の要素が変更されると動的に更新されます。

"go3" ルールは、リアクティブクエリーとして機能し、推移閉包により、新しいアイテム **Key** が **Office** に含まれるかどうかを検出します (**Office** の中の **Key** の中の **Drawer** など)。

ルール "go3"

```
rule "go3"
when
    String( this == "go3" )
    isContainedIn("Key", "Office");
then
    System.out.println( "Key is in the Office" );
end
```

この例は、"**go3**" 文字列をデシジョンエンジンに挿入して、"**go3**" ルールを有効化します。最初、**Key** が家の構造に存在するので、このルールは満たされないため、出力は生成されません。

文字列の挿入とルールの実行

```
ksession.insert( "go3" );
ksession.fireAllRules();
```

IDE コンソールのルール "go3" の出力 (条件を満たさない)

```
go3
```

この例では、**Office** の中にある **Drawer** の場所に、新しいアイテム **Key** を挿入します。この変更で、"**go3**" ルールの推移閉包が満たされ、それに合わせて出力が生成されます。

新規アイテムの場所の挿入とルールの実行


```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

IDE コンソールのルール "go3" の出力 (条件を満たす)

オフィス内の鍵

またこの変更で、クエリーにより、後続く再帰検索に含まれるよう、この構造に別のレベルが追加されます。

ルールにバインドなしの引数が含まれたクエリー

バインドなしの引数が1つ以上あるクエリーでは、クエリーの定義済み (バインドされている) 引数に含まれる未定義 (バインドされていない) アイテムすべてを返します。クエリー内の引数でバインドされているものがない場合には、クエリーはクエリーの範囲内のアイテムをすべて返します。

"go4" ルールは、バインドされている引数を使用して、**Office** 内の特定のアイテムを検索するのではなく、バインドされていない引数 **thing** を使用して、バインドされている引数 **Office** 内の全アイテムを検索します。

ルール "go4"

```
rule "go4"
when
  String( this == "go4" )
  isContainedIn(thing, "Office");
then
  System.out.println( thing + "is in the Office" );
end
```

この例では "go4" 文字列をデシジョンエンジンに挿入して、"go4" ルールをアクティベートし、**Office** の全アイテムを返します。

文字列の挿入とルールの実行

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

IDE コンソールのルール "go4" の出力

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

"go5" ルールは、バインドされていない引数 **thing** と **location** を使用して、**House** の全データ構造の中に含まれる全アイテムとその場所を検索します。

ルール "go5"

```
rule "go5"
when
```

```
String( this == "go5" )
isContainedIn(thing, location; )
then
  System.out.println(thing + " is in " + location );
end
```

この例は **"go5"** 文字列をデシジョンエンジンに挿入して、**"go5"** ルールをアクティベートし、**House** データ構造に含まれる全アイテムとその場所を返します。

文字列の挿入とルールの実行

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

IDE コンソールのルール "go5" の出力

```
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

第8章 DRL 使用時のパフォーマンスチューニングに関する考慮点

以下の主要な概念または推奨のプラクティスを使用すると、DRL ルールとデシジョンエンジンのパフォーマンス最適化に役立ちます。本セクションでこの概念についてまとめており、随時、他のドキュメントを相互参照して詳細を説明します。本セクションは、Red Hat Process Automation Manager の新しいリリースで、必要に応じて拡張または変更します。

パターンの制約のプロパティおよび値は左から右方向に定義する

DRL パターンの制約では、ファクトプロパティ名は演算子の左側に、値 (定数または変数) は右側に配置されるようにします。プロパティ名は常に、インデックスの値ではなく、キーでなければなりません。たとえば、**Person("John" == firstName)** ではなく **Person(firstName == "John")** のように指定します。制約のプロパティと値を右から左に定義すると、デシジョンエンジンのパフォーマンスが低下する可能性があります。

DRL パターンおよび制約の詳細については、「[DRL のルール条件 \(WHEN\)](#)」を参照してください。

パターン制約では他の演算子よりも等価演算子タイプをできるだけ使用する

デシジョンエンジンは、ビジネスルールロジックの定義に使用可能な DRL 演算子タイプを多数サポートしていますが、等価演算子 **==** の評価を最も効率的に実行します。実用的な場合には、他の演算子ではなく、この演算子を使用してください。たとえば、パターン (**Person(firstName == "John")**) は **Person(firstName != "OtherName")** よりも効率的に評価されます。等価演算子だけを使用すると実用的ではない場合があるので、DRL 演算子を使用するときはビジネスロジックの要件とオプションをすべて検討してください。

最も制限の厳しいルールの条件を先にリストする

ルールに複数の条件がある場合には最も制限の厳しい条件から順にリストしてください。こうすることで、最も制限の厳しい条件が満たされていない場合は、デシジョンエンジンがすべての条件セットを評価せずに済むようになります。

たとえば、以下の条件は、航空券とホテルをあわせて予約した旅行者には割引を適用する旅行予約ルールの一部です。このシナリオでは、ホテルを予約するお客様がこの割引を受けるために航空券をあわせて予約することはほぼないので、ホテルの条件はほぼ満たされず、このルールは実行されません。そのため、必要のない航空券の条件を頻繁に評価しないので、1つ目の条件の順番のほうがより効率的です。これは、1つ目の条件では、ホテルの条件が満たされていない場合に不必要かつ頻繁に、航空券の条件をデシジョンエンジンが評価せずに済むからです。

優先条件の順番: ホテルと航空券

```
when
    $h:hotel() // Rarely booked
    $f:flight()
```

効率の悪い条件の順番: 航空券とホテル

```
when
    $f:flight()
    $h:hotel() // Rarely booked
```

DRL パターンおよび制約の詳細については、「[DRL のルール条件 \(WHEN\)](#)」を参照してください。

from 句を過剰に使用する、サイズの大きいオブジェクトコレクションで反復を回避する

以下の例のように、サイズの大きいオブジェクトコレクションで反復を行う DRL ルールでは **from** 要素の使用を回避してください。

from 句を使用した条件の例

```
when
  $c: Company()
  $e : Employee ( salary > 100000.00) from $c.employees
```

このような場合には、デシジョンエンジンはルールの条件が評価されるたびにサイズの大きいグラフを反復するので、ルール評価の妨げになります。

他の手段として以下の例のように、サイズの大きいグラフが含まれるオブジェクトを追加してデシジョンエンジンが頻繁に反復作業を行うのではなく、コレクションを KIE セッションに直接追加して、条件内でコレクションを結合します。

from 句なしの条件の例

```
when
  $c: Company();
  Employee (salary > 100000.00, company == $c)
```

この例では、デシジョンエンジンは1回だけリストを反復して、ルールの評価を効率化します。

from 要素または他の DRL 条件要素に関する情報は、「[DRL でサポートされるルール条件要素 \(キーワード\)](#)」を参照してください。

ロギングのデバッグには、**System.out.println** ステートメントの代わりに、デシジョンエンジンのイベントリスナーをルール内で使用します。

ルールのデバッグやコンソール出力に、**System.out.println** ステートメントをルールアクションで使用できますが、多数のルールに対してこれを行うと、ルール評価の妨げになります。別の効率的な方法として、できるだけ内蔵のデシジョンエンジンイベントリスナーを使用してください。このイベントリスナーが貴社の要件に満たない場合には、Logback、Apache Commons Logging または Apache Log4j など、デシジョンエンジンがサポートするシステムロギングユーティリティを使用してください。

サポート対象のデシジョンエンジンのイベントリスナーおよびロギングユーティリティに関する詳細は、「[Red Hat Process Automation Manager のデシジョンエンジン](#)」を参照してください。

第9章 次のステップ

- 『[テストシナリオを使用したデシジョンサービスのテストs](#)』
- 『[Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ](#)』

付録A バージョン情報

本ドキュメントの最終更新日: 2020 年 3 月 18 日 (水)