WHITEPAPER

DESIGN PATTERNS IN PRODUCTION SYSTEMS

Wolfgang Laun, Thales Austria GmbH

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

A sceptic might argue that rules aren't really more than if statements, which doesn't leave much room for the application of various design patterns. But this is an oversimplification. First, a left-hand side is more than a Boolean expression: its matching capabilities with the implied repetitions cannot be expressed by a plain Boolean expression. Second, conditional elements like "forall" or "accumulate" result in iterations within the evaluation of the condition. Last, but not least, there is a number of ways for modifying the default semantics of rule evaluation.

Also, some applications require that their rules cooperate in a certain way, by processing facts in a specific order that must be derived from the fact data. This must be dealt with using more complex rule design patterns and typically in combination with temporary auxiliary facts.

Finally, rule groups may have to be applied in stages; the concept of rule flow results in design patterns where rule programming must be combined with logic at the session execution level.

All of this, taken together with the issue of how to design good classes for facts, is covered in Chapter 2, Rule Design Patterns.

A larger scale design issue evolves around the question: how do you construct an application around your rule-based system? This is discussed in Chapter 3, Application Design Patterns.

CHAPTER 2: RULE DESIGN PATTERNS

2.1 Preliminaries

2.1.1 Notes on Developing Classes for Facts

Object-Oriented Rule-Based Systems let you insert an object from any class into Working Memory. Parameterless methods returning a value can be used for selecting individual objects. But mostly you should use classes written in the JavaBean style, with getters and setters for fields containing a value of some simple type or a reference to another object. Note that the presence of a getter method alone is sufficient to make an attribute eligible for use in patterns; such "virtual" properties could result from the combination of two elementary fields or from the delegation to an object referenced in a field.

But what about the plethora of collections? Sets are a compact alternative for a series of Boolean properties, where set elements could be enum or possibly String or Integer objects. Consider the situation carefully before you decide on a List, Map or array. Although Drools provides the "from" clause to extract objects from a collection and make them available for condition pattern matching, such fields violate a basic design rule for data in a related area, viz., database design. Following the relational model and observing the principles of the first normal form should result in fact data sets that are easily combined with each other.

Another issue results from the question: Which technique should be used to link fact objects? Since facts are in memory, "hard" object references can be used where other data models must make do with (primary) key values as "symbolic" links. Using such a key property value for WMEs is, of course, still feasible, and it may have the advantage that this data is already in place, especially if fact data is backed by some database system or read in from file data (Using XML may result in either form, depending on the use of ID and IDREF values).

Lists or arrays are frequently used in object hierarchies for storing the references from a parent object to its children. Given the "from" clause, this may be sufficient for writing all required rules, even when the child objects are not facts themselves; it frequently turns out that a ("hard" or "symbolic") link from a child to its parent permits cleaner and more efficient conditions to achieve the same result.

Some attention should be given to the type hierarchy from which the fact classes are taken. Intermediary (abstract) classes and interfaces are eligible for being used as pattern types, which means that one rule may be sufficient for all its subclasses or subinterfaces.

Another aspect in the design of fact types is the lifetime of their objects. Persistent facts usually represent entities of the "real world", but immutable facts representing lookup-tables or parameter sets also fall into this category. Typically, they are created during system startup and remain in the Working Memory as long as the application runs, and they may have a backup in a database system. Other long-lived facts are those that are derived by rules, sometimes according to the principle of "truth maintenance". In contrast, there is also fact data that is short-lived, intended to be processed and discarded by rules. Frequently, such facts reflect events, commands or messages. When they are inserted, they cannot contain links to existing facts; they will have to be matched with static data according to key properties.

2.1.2 Basic Rule Formats

Even the simplest rules deserve some consideration, and all the more so when you expect a large number of facts. Both the compiler and the engine do some work behind the scenes to speed up the pattern-matching process, but there is still a "best practice" for writing rules. Moreover, changes between releases may affect certain patterns, and so it might be a good idea to develop benchmarks in parallel to the rules intended for production deployment.

The simplest condition of all is shown in Example 1.

Example 1: The Empty Condition

```
rule "fire once"
when
then
   // ...
end
```

An empty condition evaluates to true (true is the initial value for a conjunction over any number of Boolean terms). This means that initially one, and only one, activation of this rule is created, and that it will fire early on, according to its priority. This sort of kick-off rule is quite useful for running some initialization code that should not be coded as part of the embedding Java application. It's also helpful for running simple explorative tests.

Rules with a single pattern are frequently used for validation and classification. Field constraints should be used in preference to inline "eval" constraints. To restrict a field value to a range of values you might use the shorthand form where you do not need to repeat the left-hand side operand, e.g.,

```
Type( intField >= 1 && <= 10 )
```

A multiple choice could be written using the same syntactic form, e.g.,

```
Type( strField == "Huey" || == "Dewey" || == "Louie" )
```

but if you are testing for equality, the MultiValueRestriction is to be preferred, so that the preceding pattern is better written as

```
Type( strField in ( "Huey", "Dewey", "Louie" ) )
```

Note, however, that the equivalent test using regular expression matching is not as efficient, i.e., do not write

```
Type( strField matches "Huey|Dewey|Louie" ) )
```

Starting with version 5.2, Drools permits general boolean expressions as constraints. (Examples follow the more restrictive syntax required up to 5.1.1, which certainly is still valid.) This may open up surprising possibilities. The following pattern shows how to write a test for a string containing only upper case characters without using a regular expression:

```
Type( strField == (strField.toUpperCase()) )
```

Still using just a single pattern, slightly more complicated conditions are possible by using two or more fields, especially when different field values need to be combined. Luckily, Drools lets you access fields from the same fact without the need for a variable binding. Thus, a condition asserting that the famous triangle inequality is fulfilled for a triangle can simply be written as

```
Triangle( a < (b + c), b < (a + c), c < (a + b) )
```

The complementary condition is just as simple to write as the operator || can be used for combining constraints.

```
Triangle( a >= (b + c) || b >= (a + c) || c >= (a + b) )
```

Rising to the next level, we combine two patterns. Each of them may have its own selective restrictions, but now we can, in the second pattern, write constraints that use variables bound to fields of the fact matched by the first pattern, or even to the entire fact. One simple case is the equality of a field with the previously matched object:

```
$parent: Person()
$child: Person( father == $parent || mother == $parent )
```

As this can also be used as if it were a field, we can also write the patterns in reverse order, producing exactly the same activations:

```
$child: Person()
$parent: Person( this == ($child.getFather()) ||
                    this == ($child.getMother()) )
```

For the sake of completeness, here is also the combination of two facts of the same type where natural constraints do not prohibit any possible match. Given that you want to find all pairs of persons with equal hobbies, you must exclude the match of one Person fact with both patterns, which is easy:

```
$p1: Person( $h: hobby )
$p2: Person( this != $p1, hobby == $h )
```

But this will still produce every pair twice. To exclude the useless duplicates, use a unique comparable field instead of the object reference.

```
$p1: Person( $pnr: pnr, $h: hobby )
$p2: Person( pnr > $pnr, hobby == $h )
```

If there is no adequate property, consider adding one, if only for this purpose.

2.2 Fact Classification

Classifying objects or a small set of related facts according to a set of criteria is one of the popular applications of rules. The heritage of decision tables is apparent, and this has led to rule authoring tools that carry on this traditional style of decision making. While this section discusses techniques applicable to an implementation in DRL, readers interested in creating rules straight from decision tables are referred to Section 4.1.

Let's assume that the goal is to modify a fact of type Tint depending on two properties; one is an integer and the other one a string. Categories are defined by an interval of the numeric property in combination with a string value. Therefore, a rule to classify a fact is written according to Example 2.

Example 2: A Rule Classifying a Fact

```
rule "classify as low and red"
when
   $t: Tint( level >= 0 && < 10, colour == "red" )
then
   modify( $t ){
   setCategory( "Category.LOW_RED" )
   }
end
```

The thought of having to write dozens, if not hundreds, of similar rules with variations in the level bounds and the colour is appalling, to say nothing of the profusion of "magic" numbers and strings. Any approach using a table of the variable parameters to generate rules would save us the drudgery of writing the punishing repetitions with their hard-coded literals—but it still generates an abundance of rules, and this is bound to cost resources, at least during compilation. But there is a simple way of reducing the number of rules. It is based on the existence of parameter facts containing the constraining parameters and relating them to the required category setting. The corresponding rewrite of the preceding example is shown in Example 3.

Example 3: Generic Fact Classification

```
rule “classify by level and colour”
when
   Param( $ll: loLevel, $hl: hiLevel, $c: colour, $cat: category )
   $t: Tint( level >= $ll && < $hl, colour == $c, category == null )
then
   modify( $t ){
          setCategory( $cat )
   }
end
```

The constraint category == null is not redundant. It avoids the immediate reactivation of the rule due to the update of the matching Tint fact.

This approach also lets you implement rules that are hard to produce with any other approach. One is the discovery of the fact that fails to match any parameter sets, so that it cannot be classified. This is shown in Example 4.

Example 4: Discover Failing Fact Classification

```
rule "failure to classify by level and colour"
when
   $t: Tint( $l: level, $c: colour )
   not Param( loLevel <= $l, hiLevel > $l, colour == $c )
then
   System.err.println( "failure to categorize " + $t.toString() );
end
```

The other useful rule we can write is the detection of a fact matching more than one parameter set. This is demonstrated in Example 5.

Example 5: Detect Multiple Classification

```
rule "classify already classified Tint"
when
   Param( $ll: loLevel, $hl: hiLevel, $c: colour,
          $cat: category )
   $t: Tint( level >= $ll && < $hl, colour == $c,
          $setcat: category != null && != $cat )
then
   System.err.println( "duplicate category " + $t.toString() +
                ", also " + $cat );
end
```

Closely related to the classification by assigning a category to a fact is the task of producing a statistic report, i.e., counting facts according to some classification. Now we do not want to fire a rule for each event, we would rather have one firing for each category. Again, parameter facts are useful because we can base the firings on them and use "from collect" to tally the matches. Example 6 shows how to use the "accumulate" conditional element. A low salience postpones the evaluation of this rule until all classifications have gone through.

Example 6: Counting Facts by Category

```
rule “count by category”
salience -100
when
   $p: Param( $cat: category )
   Number( $count: intValue ) from accumulate (
          Tint( category == $cat ), count(1) )
then
   System.out.println( $cat + “: “ + $count );
end
```

As a concluding remark, note that it is quite easy to create the parameter facts based on an external data set, either an XML file or a spreadsheet. This leaves you with a highly generic code and the option of delegating the parameter sets to domain experts.

2.3 Accumulations

Arriving at a decision over the combined set or some subset of selected facts is a frequently required task. If these facts can be described by a pattern, the accumulate clause should be applied. Example 7 shows how to count a selection of objects according to a property. The selection consists of all facts of type Request and the counts should be done according to its property id.

Example 7: Counting Objects According to Property

```
rule countById
when
   $set: Set()
          from accumulate( Request( $id: id ),
          init( HashMap id2count = new HashMap() ),
          action( Integer i = (Integer)id2count.get( $id );
                 if( i == null ) i = 0;
                 i++;
                 id2count.put( $id, i ); )
          reverse( Integer i = (Integer)id2count.get( $id );
                 i--;
                 id2count.put( $id, i ); )
          result( id2count.entrySet() ) )
   java.util.Map.Entry( $id: key, $count: value > 1 ) from $set
then
   System.out.println( "Id " + $id + ": " + $count );
end
```

Counters are kept in a map, created in step init. The code doing the counting is in the accumulate steps action and reverse, written inline. The result is produced as the set of Map.Entry objects, readily available from the map. The last pattern on the LHS shows how the resulting entry set can be filtered: the rule fires once for each id value occurring more than once.

Occasionally it is useful to use a constraint to select a subset of facts over which the accumulation should be performed. This is shown in Example 8, where, given a fact of type String, the rule determines the fact of type Item where its field code contains the longest matching initial substring of that string. For example, given the String "556789" and code values "55", "556" and "5567", the last one should be found.

Example 8: Finding The Longest Match

```
rule matchLongest
when
   $string: String()
   $bestItem: Item() from accumulate(
          $item: Item( $code: code,
                 eval( $string.startsWith( $code ) ) ),
          init( Item bestItem = null;
                 int bestLength = 0; )
          action( if( $code.length() > bestLength ){
                 bestItem = $item;
```

```
                bestLength = $code.length();
                } )
          result( bestItem ) )
then
   System.out.println( "Best match for " + $string +
                " is " + $bestItem.getCode() );
   retract( $string );
end
```

Facts of class Item are selected by a call ascertaining that the investigated string starts with the item's code. The accumulate steps perform a maximum search, keeping track not only of the maximum length but also of the Item where the maximum occurs, and this object is the result (a more robust implementation has to handle the case that there is no match at all, because an accumulate function must never return null).

2.4 Handling Failure to Match

2.4.1 To Match or Not to Match

There are several application scenarios where a static fact base is repeatedly confronted with dynamically inserted facts, with the basic idea of associating the inserted fact with one or more static facts. Whether the criterion for the match is a simple primary key (e.g., an article number) or a complex combination of attributes doesn't matter: there is always the possibility that the inserted fact has no counterpart at all. But in any case, provisions need to be made for retracting the temporary fact after it has outlived its purpose, silently, if successful, and with a diagnostic upon failure. If you expect not more than a single match, you might try a solution as shown in Example 9.

Example 9: Match one fact and detect failure

```
rule "match article"
when
   $o: Order( $id: id )
   $a: Article( id == $id )
then
   // process Order
   retract( $o );
end
rule "no such article"
when
   $o: Order( $id: id )
   not Article( id == $id )
then
   // diagnostic
   retract( $o );
end
```

While this is perfectly correct, following this principle soon becomes a tangle if more constraints must be included in the matching operation. Inverting complex conditions is not insurmountable, but it is bound to create a maintenance problem. The simple alternative is shown in Example 10, showing a low-priority rule bound to catch all facts that have not been matched by the primary rule "match article" in Example 9.

Example 10: A low-priority catch-all rule

```
rule "unmatched order"
salience -100
when
   $o: Order()
then
   // diagnostic
   retract( $o );
end
```

The same technique is applicable when multiple matches are expected, e.g., when the inserted fact is not a definite order but just a search for candidates.

Example 11: Match multiple facts

```
rule "match article"
when
   $s: Search( $c: category, $p: price )
   $a: Article( category == $c, price <= $p )
then
   // ...
end
rule "remove Search fact"
salience -100
when
   $s: Search()
then
   retract( $s );
end
```

But while the second rule in Example 11 removes the obsolete Search fact, it provides no clue whether there were any matches at all. This could be solved by counting the matches, for which the Search fact would be a convenient place. Pursuing this idea results in the pair of rules shown in Example 12.

Example 12: Match multiple facts and count matches

```
rule "Match Point"
when
   $s: Search( $c: category, $p: price )
   $a: Article( category == $c, price <= $p )
then
   // ...
   $s.setCount( $s.getCount() + 1 );
end
rule "remove Search fact"
salience -100
```

```
when
   $s: Search()
then
   if( $s.getCount() == 0 ){
          // diagnostic
   }
   retract( $s );
end
```

Notice that the consequence of rule "match article" does not contain a modify statement to announce the update of the Search fact. This is one of the rare cases where a "dirty update" is appropriate. Telling the Engine about this change would cause an immediate reactivation of that rule. Trying to counter this with the rule attribute no-loop is not effective, because there may be another match with another article, and this clears the short memory of the loop-avoiding mechanism, so that the newly updated Search fact happily engages in yet another match with the previously visited Article. Not updating the Search fact prohibits writing individual rules for post-processing, but accessing the accumulated count on the RHS returns the correct value.

Finally, there is yet another useful pattern that may be employed for similar tasks. Example 13 illustrates the technique for collecting matches in a Collection type filed of the temporary fact, killing two birds with one stone: in the final rule we encounter a Search fact where the size of its field matches easily provides all the information we need, either for processing the matches or for emitting a diagnostic. Also, note that the third conditional element "eval" is a way to avoid an infinite loop of this rule due to its repeated update. Here, too, a "dirty update" combined with a low priority for the finalizing rule would constitute an alternative.

Example 13: Match multiple facts and collect matches

```
rule "collect matching article"
when
   $s: Search( $c: category, $p: price, $m: matches )
   $a: Article( category == $c, price <= $p )
   eval( ! $m.contains( $a ) )
then
   modify( $s ){
             getMatches().add( $a )
     }
end
rule "finalize Search fact"
salience -100
when
   $s: Search( $m: matches )
then
   if( $m.size() > 0 ){
          // process matches in $m
   } else {
          // diagnostic
   }
   retract( $s );
end
```

2.4.2 Learning the Reason for Failure

The design patterns presented in the previous subsection take care of processing matches and handling failures. Quite frequently, however, you are required not only to tell the user that there was no match but also to procure the reason for failure.

The well-known approach used in procedural programming is to have a series of if statements, each of them checking one of the constraining conditions. Given that the order of such tests doesn't matter, you can use the same logical expressions as conditions in rules, but you must make sure that all of these rules have higher precedence than the rule doing the actual processing.

For the simple matching task shown in Example 13, a supplemental set of rules precisely establishing the reason for failure is given in Example 14. One might argue that the conditional element "exists" in the second rule is superfluous, but in practice, users would very likely prefer to know whether their request is entirely hopeless or a little more cash might give them what they want.

Example 14: Determining the reason for failure

```
rule "no matching category"
when
   $s: Search( $c: category )
   not Article( category == $c )
then
   // diagnostic: no such category
   retract( $s );
end
rule "no article cheaper than"
when
   $s: Search( $c: category, $p: price )
   exists Article( category == $c )
   not Article( category == $c, price <= $p )
then
   // diagnostic: no $c article cheaper than $p
   retract( $s );
end
```

This is certainly not satisfactory. An alternative to the intricate condition in the second rule is to use salience, as shown in Example 15.

Example 15: Determining failures using salience

```
rule "no matching category"
salience 200
when
   $s: Search( $c: category )
   not Article( category == $c )
then
   // diagnostic: no such category
   retract( $s );
```

```
end
rule "no article cheaper than"
salience 100
when
   $s: Search( $c: category, $p: price )
   not Article( price <= $p )
then
   // diagnostic: no $c article cheaper than $p
   retract( $s );
end
```

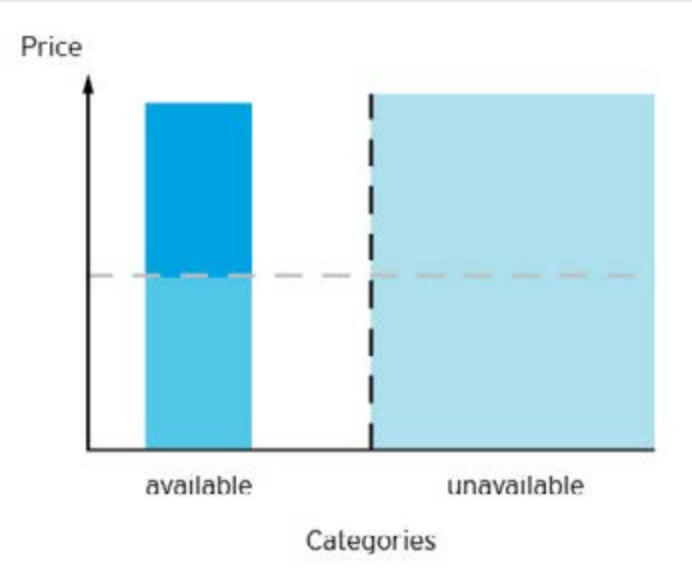

Figure 2.1 Price and Availability

As can be seen on Figure 2.1, the value space of the example has only two dimensions, each of which is subdivided into two sets of values. Out of the resulting four areas of the value space we want to distinguish three: the very bad one where there is no match for the category, a bad one with no articles that are cheap enough, and the one where all the matches are. You can imagine that each additional dimension is apt to complicate matters.

But we can do better than that by exploiting the idea we developed to collect all matches for accumulating all disqualifying conditions. Rather than emitting a diagnostic in each rule detecting a reason for failure, we just collect it in the Search object. Example 16 shows how the negative rules add entries to the additional field diagnostics of a Collection of Failure, an enum type. At first glance, this doesn't appear to be an improvement over the previous approach. But notice that we do not need salience in the rules determining failure reasons—we just collect them all, and let the final rule for processing failures determine how to process them. More importantly, however, is the consequence of delegating the decision of what to print as an error message: any additional property that needs to be tested requires only one additional rule besides the extension of the constraint in the rule determining a match.

Example 16: Collecting failures and matches

```
rule "no matching category"
when
   $s: Search( $c: category,
   $f: failures not contains Failure.CATEGORY )
   not Article( category == $c )
then
   modify( $s ){ getFailures().add( Failure.CATEGORY ) }
end
rule "no article cheaper than"
when
   $s: Search( $p: price,
   $f: failures not contains Failure.PRICE )
```

```
   not Article( price <= $p )
then
   modify( $s ){ getFailures().add( Failure.PRICE ) }
end
rule "collect matching article"
when
   $s: Search( $c: category, $p: price, $m: matches )
   $a: Article( category == $c, price <= $p )
   eval( ! $m.contains( $a ) )
then
   modify( $s ){ getMatches().add( $a ) }
end
rule "finalize Search fact - success"
salience -100
when
   $s: Search( $m: matches, eval( $m.size() > 0 ) )
then
   // process matches in $m
   retract( $s );
end
rule "finalize Search fact - failure"
salience -100
when
   $s: Search( $d: failures, eval( $d.size() > 0 ) )
then
   // process failures in $d
   retract( $s );
end
```

2.5 Data Validation

2.5.1 A Simple Validation Rule

Ensuring that some application operates on correct data is a standard procedure. Failure to implement a vetting process makes successive processing vulnerable or more expensive. For most applications, data validation can be defined through declarative data integrity rules, or it can be ensured by conventional procedural programming.

Validation rules span a wide range, covering simple representational issues such as the choice of the character set and encoding, restrictions concerning individual values or a combination of two or more values of a data item, up to rules that require complex evaluations of selected data items.

A simple example from the restrictions for a single value is implemented as a rule in Example 17.

Example 17: Range check of an integer value

```
rule "Check Item.code"
when
   $item: Item( $c: code < 1 || > 10 )
then
   System.out.println( "Invalid code " + $c + " in " + $item );
end
```

If the limits are part of the data model and not at the discretion of the rule author it is bad style to use literals in the rule. Class Item should contain static fields defining these limits. In either case, writing a large number of rules with similar constraints does not appear to be attractive. Using templates, presented in Chapter 4.2, is one way of creating many similar rules from a data set providing the variable parts.

2.5.2 Conditions as Facts

We could reduce the number of rules if it were possible to encode all information required to do the check into an object; then we can write a rule combining these "condition" objects with a data fact.

It is obvious that several condition classes are required, each one representing one specific conditional expression, a subclass of an abstract base class Condition. For the range constraint shown in Example 17, the class IntRangeCond is shown in Listing 1.

Listing 1: Classes Condition and IntRangeCond

```
import java.lang.reflect.Field;

public abstract class Condition {
   private Class<?> clazz;
   private String fieldName;
   protected Field field;
   private String error;

   public Condition( Class clazz, String fname, String error )
          throws Exception {
          this.clazz = clazz;
          this.fieldName = fname;
          this.error = error;
          this.field = clazz.getDeclaredField( fname );
          field.setAccessible( true );
   }

   public Class<?> getClazz(){
          return clazz;
   }

   public String getField(){
   return fieldName;
   }

   public String getError(){
   return error;
   }

   public abstract boolean isValid( Object object ) throws Exception;
}

public class IntRangeCond extends Condition {
   private int lower;
   private int upper;
```

```
    public IntRangeCond( Class clazz, String fname,
                         int lower, int upper, String error )
           throws Exception {
           super( clazz, fname, error );
           this.lower = lower;
           this.upper = upper;
      }

      public boolean isValid( Object object ) throws Exception {
           Object value = field.get( object );
           if( value == null ) return false;
           int intValue = (Integer)value;
           return lower <= intValue && intValue <= upper;
    }
}
```

The equivalent of the check in Example 17 is achieved by inserting an IntRangeCond object like this:

```
kSession.insert( new IntRangeCond( Item, “code”, 1, 10 “invalid code” ) );
```

and by writing the rule shown in Example 18.

Example 18: Integer range checks on Item

```
rule "Check all Item integer ranges"
when
   $item: Item()
   $cond: IntRangeCond( clazz == ( Item.class ) )
   eval( ! $cond.isValid( $item ) )
then
   System.out.println( $cond.getError() + " " + $item.toString() );
end
```

Attentive readers may have noticed that the rule does not make any specific references to Item attributes. Observing that a class is also available from the object, we can rewrite the rule to perform integer range checks on arbitrary objects. This is shown in Example 19.

Example 19: Integer range checks

```
rule "Check all integer ranges"
when
   $fact: Object()
   $cond: IntRangeCond( clazz == ( $fact.getClass() ) )
   eval( ! $cond.isValid( $fact ) )
then
   System.out.println( $cond.getError() + " " + $fact.toString() );
end
```

Another inspection shows that the rule doesn't make any use of the properties that are specific to IntRangeCond; only those of the abstract base class are referenced. Moving up once more, we arrive at the rule shown in Example 20 with the surprising realization that a single rule is sufficient for triggering all kinds of tests on all facts.

Example 20: All checks

```
rule "All checks"
when
   $fact: Object()
   $cond: Condition( clazz == ( $fact.getClass() ) )
   eval( ! $cond.isValid( $fact ) )
then
   System.out.println( $cond.getError() + " " + $fact.toString() );
end
```

Conversely, if firings require consequences differentiated by fact type, you may replicate the rule for various fact types, or use dedicated interface types for selecting specific fact type groups.

You may also have noticed that Condition objects being facts in Working Memory makes them eligible for being checked in the very same way.

2.5.3 Conditions as Annotations

If validation constraints are part of the data model, dealing with the rules enforcing them should not require the explicit creation and insertion of facts describing the underlying conditions. Ideally, their definition would be written as metadata along with the definition of the class. This is where Java's annotations can be put to work.

Following the same idea as in the previous subsection, we define an annotation class for holding the limits of a range check, as shown in Listing 2.

Listing 2: An annotation defining an integer range

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface AnnIntRange {
   Class<? extends ChkBase> handler() default ChkIntRange.class;
   int lower() default Integer.MIN_VALUE;
   int upper() default Integer.MAX_VALUE;
}
```

Example 21 shows an annotation in class Article.

Example 21: Annotation for a range check

```
public class Article {
   private String number;
   private String name;
   private String category;
   private int packCode;
```

```
    @AnnIntRange( lower = 0, upper = 2000 )
    private int stock;
    //...
}
```

Listing 3 presents class ChkIntRange, which is the one implementing the actual check, and ChkBase, its base class. An object of this class associates a Field (from java.lang.reflect object with an Annotation object. Using method getDeclaringClass in class Field it is possible to match with an object containing this field. Method check in class ChkIntRange is the workhorse comparing the value taken from the matched object to the upper and lower bounds defined in the annotation.

Listing 3: ChkIntRange: an integer range check

```
public abstract class ChkBase {
    private Field field;

    protected ChkBase( Field field ){
           this.field = field;
           field.setAccessible( true );
    }

      public Field getField(){
           return field;
    }

    public abstract boolean check( Object object ) throws Exception;

    public abstract String message( Object object )
           throws Exception;
}

public class ChkIntRange extends ChkBase {
    private AnnIntRange annotation;

    public ChkIntRange( Field field, AnnIntRange annotation ){
           super( field );
           this.annotation = annotation;
    }

    public AnnIntRange getAnnotation(){ return annotation; }

    public boolean check( Object object ) throws Exception {
           Object value = this.getField().get( object );
           if( value == null ) return true;
           int intValue = (Integer)value;
           int lower = annotation.lower();
           int upper = annotation.upper();
           return intValue < lower || upper < intValue;
    }
```

```
    public String message( Object object ) throws Exception {
           StringBuilder sb = new StringBuilder( "Range error in " );
           sb.append( object.toString() );
           sb.append( ": return value of " );
           sb.append( this.getField().getName() );
           sb.append( "() = " );
           sb.append( this.getField().get( object ) );
           sb.append( " is not in [" );
           sb.append( annotation.lower() );
           sb.append( "," );
           sb.append( annotation.upper() );
           sb.append( "]" );
           return sb.toString();
    }
}
```

Objects of various subclasses of ChkBase must be present as facts in Working Memory. Listing 4 contains the code for class ChkInserter. An object of this class is instantiated with an object of class StatefulKnowledgeSession as the destination of the objects acquired by subsequent method calls. Method analyse retrieves all suitable annotations from the fields of the given class. They contain a reference to some subclass of ChkBase, and an object of this type is created, holding a reference to the field and the annotation, and inserted into the session.

Listing 4: Class ChkInserter

```
public class ChkInserter {
    private StatefulKnowledgeSession kSession;

    public ChkInserter( StatefulKnowledgeSession kSession ){
           this.kSession = kSession;
    }

    public void analyse( Class<?> clazz ){
           Field[] fields = clazz.getDeclaredFields();
           for( Field field: fields ){
                  Annotation[] annotations =
                      field.getDeclaredAnnotations();
                  for( Annotation annotation: annotations ){
                      Class<? extends Annotation> annoClass =
                              annotation.annotationType();
                      Method getHandler = null;
                      try {
                              getHandler = annoClass.getMethod( "handler" );
                      } catch( Exception e ){
                              // ignore
                      }
                      if( getHandler == null ) continue;
                      Class<?> ghrt = getHandler.getReturnType();
                      if( ! Class.class.equals( ghrt ) ) continue;
                      try {
                              Class<ChkBase> handlerClass =
```

```
                                (Class<ChkBase>)getHandler.invoke(annotation);
                            Class<? extends Annotation> annoType =
                              annotation.annotationType();
                            Constructor<? extends ChkBase> constructor =
                              handlerClass.getConstructor( Field.class,
                                                           annoType );
                            ChkBase ca = constructor.newInstance( field,
                                                           annotation );
                            kSession.insert( ca );
                        } catch( Exception e ){
                            throw new IllegalStateException(
                                    "cannot create ChkAnn object", e );
                        }
                    }
                }
            }
        }
```

After an instance of ChkInserter has been invoked with method analyse for all relevant classes and after the insertion of some article facts, we can expect the rule presented in Example 22 to trigger range checks for all existing Article objects.

Example 22: Rule running validation checks

```
rule ChkIntRange
when
   $obj: Object()
   $fa: ChkBase( field.declaringClass == ( $obj.getClass() ) )
   eval( $fa.check( $obj ) )
then
   System.out.println( $fa.message( $obj ) );
end
```

It should not come as a surprise that a single rule is sufficient here, too. More selective applications are possible by using a specific fact type in the first pattern or a subclass of ChkBase in the second pattern.

2.5.4 Complex Validation Techniques

While the previous subsections illustrate the respective technique with a simple integer range test, other checks are equally simple to implement. For integers also sets of (isolated) values are appropriate for defining permissible values. For strings, range checks for the length, comparisons with a set of values and regular expressions are among the suitable ways of defining constraints.

Some of the aforementioned checks could be used repeatedly. If, for instance, the set of valid values consists of several intervals, range checks can be used simply by "bundling" the annotation AnnIntRange into another annotation containing an array, as shown in Listing 5.

Listing 5: Annotation for the repeated use of AnnIntRange

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface AnnIntRanges {
   Class<? extends ChkBase> handler() default ChkIntRanges.class;
   AnnIntRange[] value();
}
```

Its use is straightforward: a number of AnnIntRange annotations, enclosed in braces, is included as the value, as shown in Example 23.

Example 23: Using the AnnIntRanges annotation

```
public class
// USA SSN area code excludes 000, 666 and 900-999.
@AnnIntRanges( { @AnnIntRange(lower =   1, upper = 665 ),
                 @AnnIntRange(lower = 667, upper = 899 ) } )
private int areaCode;
```

2.6 Extending Rules

The rule option "extends" (see 4.3) enables you to continue a rule's condition in one or more other rules. All rules are activated independently, but it is self-evident that an activation of an extending rule can never happen without an activation of the extended rule.

A simple application of rule extension is the sharing of part of a condition among several rules. A special case of this is the creation of two rules that distinguish between a single constraint being true or false. Returning to the example from Subsection 2.4.2, we can write the rules shown in Example 24. Note how the same Article fact that was matched in the first rule is grabbed again in one of the follow-up rules, due to the constrained value of this.

Example 24: Extending a rule

```
rule "a search"
when
   $s: Search( $c: category, $p: price, $m: matches )
   $a: Article( category == $c )
then
end
rule "article too expensive"
extends "a search"
when
   Article( this == $a, price > $p )
then
   // ... too expensive
end
rule "article matches"
when
   Article( this == $a, price <= $p )
then
   // ... cheap enough
end
```

2.7 Reasoning with Interfaces

The full range of Java typing is available for use as facts and, therefore, in rule patterns. The absolutely constraint-free pattern Object() is possible, just like any of its subclasses, abstract or not. But it must not be overlooked that a pattern can also be written with an interface for its type. This permits the design of rule sets, operating on individual selections of facts, and where each rule addresses the properties exhibited by one interface. The full rule design pattern needs some additions for combining the results from individual matches, but this is more than compensated by the benefits brought forth with this approach.

The model used for illustrating this design pattern is a naive representation of data collected from appliances from some factory floor. The combinations of measured data such as temperature, pressure and operational data differ according to the kind of device. The challenge is to devise a compact set of rules simplifying the generation of reports on the various gadgets, according to their attributes. Listing 6 shows the condensed code for the classes and interfaces used in the example.

Listing 6: Types for the factory floor model

```
abstract class Station implements Id {
   private String id;
   protected Station( String id ){ ... }
   public String getId() { ... }
}
interface Id {
   public String getId();
}
interface Temperature extends Id {
   int getTemperature();
}
interface Pressure extends Id {
   double getPressure();
}
interface OpTime extends Id {
   double getOpTime();
}
interface OpCount extends Id {
   int getOpCount();
}
class Furnace extends Station
          implements Temperature, OpTime {
   private int temperature;
   private double opTime;
   // ...
}
class Compactor extends Station
          implements Pressure, OpCount {
   private double pressure;
   private int opCount;
   // ...
}
```

```
class Cutter extends Station
          implements OpCount {
   private int opCount;
   // ...
}
class Tank extends Station
          implements Pressure, Temperature, OpTime {
   private double pressure;
   private int temperature;
   private double opTime;
   // ...
}
```

Figure 2.2 shows the class hierarchy, whereas Figure 2.3 depicts the interface hierarchy.

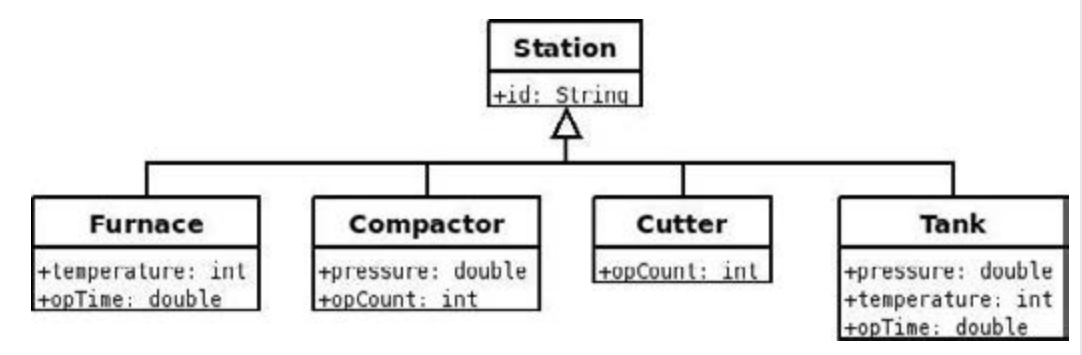

Figure 2.2: Subclasses of Station

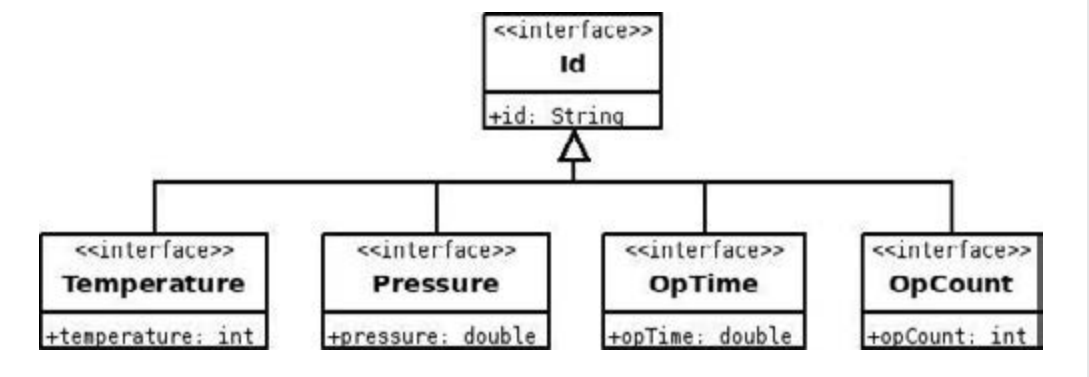

Figure 2.3: Subinterfaces of Id

We need two temporary fact types for the request and the reply, which is shown in Listing 7.

Listing 7: Report request and reply

```
class ReportRequest implements Id {
   private String id;
   // ...
}
public class ReportReply {
   private String id;
   private List<String> lines = new ArrayList<String>();
   // ...
}
```

The rules for processing the request and the reply use design patterns we have discussed in preceding sections.

Listing 8: Rules for report request and reply

```
rule "accept report request"
when
   $r: ReportRequest( $id: id )
   Station( id == $id )
then
   ReportReply reply = new ReportReply( $id );
   insert( reply );
   retract( $r );
end
rule "refuse report request"
when
   $r: ReportRequest( $id: id )
   not Station( id == $id )
then
System.err.println( "no such equipment: " + $id );
   retract( $r );
end
rule "send report reply"
salience -100
when
   $r: ReportReply( $id: id, $lines: lines )
then
   // process reply lines
   retract( $r );
end
```

In Listing 9 we have the rules for preparing all reports, and for any kind of device, no matter what its properties are. Each of the following rules match at most once, contributing a line for the report.

Listing 9: Rules for collecting report lines

```
rule "get temperature"
when
   $r: ReportReply( $id: id, $lines: lines )
   $t: Temperature( id == $id, $temp: temperature )
then
   $lines.add( "temperature: " + $temp );
end
rule "get pressure"
when
   $r: ReportReply( $id: id, $lines: lines )
   $t: Pressure( id == $id, $pres: pressure )
then
   $lines.add( "pressure: " + $pres );
end
rule "get operating time"
when
   $r: ReportReply( $id: id, $lines: lines )
   $t: OpTime( id == $id, $optm: opTime )
then
   $lines.add( "operating time: " + $optm );
end
rule "get operation count"
when
   $r: ReportReply( $id: id, $lines: lines )
   $t: OpCount( id == $id, $opct: opCount )
then
   $lines.add( "operation count: " + $opct );
end
```

Where is the benefit as compared to rules reporting on individual devices? We did save some statements for composing and printing report lines, but this is not the main advantage. For one thing, consider what happens when we introduce a new kind of device using existing measurement data types. It's self-evident that the Java class has to be added, but absolutely no change in the rule base is necessary. Extending the range of measurements requires a simple rule according to the pattern of the other ones, but the number of devices to which this new attribute is added does not matter. Admittedly, there is also the disadvantage of firing one rule for each line of the report, but which coin doesn't have two sides?

2.8 Combinatorial Joins

Some problems appear to require the creation of a large number of combinations of many facts from one or more domains even though selection criteria are expected to reduce this ultimately to a small number of solutions.

Ideally, one would construct the fact combinations based on the constraints, but this works only in relatively simple situations. Creating all combinations up front is usually impossible, since their number increases exponentially.

The famous "Zebra Puzzle" provides a nice example for this kind of problem. It is traditionally presented by the following text, according to Life International, December 17, 1962:

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in a house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water? Who owns the zebra?

Trying to compose individual houses with permitted attribute combinations quickly comes to a standstill because the neighbors in the green and ivory house cannot be placed unambiguously. Therefore we select the approach where we begin with all permutations of each of the five attributes, which requires 5 x 5! = 600 facts. Listing 10 shows the elementary classes needed for these facts.

Listing 10: Classes for the Zebra Puzzle

```
public enum Animal {
   SNAILS, ZEBRA, FOX, HORSE, DOG;
}

public enum Colour {
   RED, GREEN, YELLOW, IVORY, BLUE;
}

public enum Drink {
   TEA, COFFEE, MILK, ORANGE_JUICE, WATER;
}

public enum Nationality {
   ENGLISHMAN, NORWEGIAN, SPANIARD, UKRAINIAN, JAPANESE;
}

public enum Smoke {
```

```
    KOOLS, OLD_GOLD, CHESTERFIELDS, LUCKY_STRIKE, PARLIAMENTS;
}

public class Array<T extends Enum<T>> {

    private List<T> list;
    private Class<?> clazz;

    public Array( List<T> list ){
            this.list = list;
            this.clazz = list.get(0).getClass();
    }

    public int indexOf( T value ){
            for( int i = 0; i < list.size(); i++ ){
                    if( value == list.get( i ) ) return i;
            }
            throw new IllegalArgumentException( "no such value: " +
                                                    value );
    }

    public List<T> getList(){
            return list;
    }

    public T valueAt( int index ){
            if( 0 <= index && index < list.size() ){
                    return list.get( index );
            } else {
                    return null;
            }
    }

    public String toString(){
            return this.getClass().getSimpleName() + list.toString();
    }
}

public class Animals extends Array<Animal> {
    public Animals( List<Animal> list ){
            super( list );
    }
}
```

Classes Colours, Drinks, Nationalities and Smokes are declared in the same way as Animals; they let us refer more conveniently to the individual attribute sets.

Listing 24.6 presents a method for generating all permutations of an enum type and the code for using this, to insert all permutations as facts.

Listing 24.6: Creating the facts for the Zebra Puzzle

```
public class EnumUtil {
```

```
    private static <T extends Enum<T>> List<List<T>> perm(List<T> elems ){
           List<List<T>> res = new ArrayList<List<T>>();

           if( elems.size() == 1 ){
              res.add( new ArrayList<T>( elems ) );
              return res;
           }
           int lastIndex = elems.size() - 1;
           T x = elems.get( lastIndex );
           List<T> subList = elems.subList( 0, lastIndex );
           List<List<T>> perms = perm( subList );
           for( List<T> perm: perms ){
              for( int i = 0; i <= perm.size(); i++ ){
                  List<T> newPerm = new ArrayList<T>( perm );
                  newPerm.add( i, x );
                  res.add( newPerm );
              }
           }
           return res;
     }

     public static <T extends Enum<T>> List<List<T>> permute( Class<T> enumClass ){
              EnumSet<T> enumSet = EnumSet.allOf( enumClass );
              List<T> elems = new ArrayList<T>( enumSet );
              return perm( elems );
   }
}

public class Main {
   //...

   private void makeFacts(){
           for( List<Animal> list: EnumUtil.permute( Animal.class ) ){
                  kSession.insert( new Animals( list ) );
           }
           for( List<Colour> list: EnumUtil.permute( Colour.class ) ){
                  kSession.insert( new Colours( list ) );
           }
           for( List<Drink> list: EnumUtil.permute( Drink.class ) ){
                  kSession.insert( new Drinks( list ) );
           }
           for( List<Nationality> list: EnumUtil.permute( Nationality.class ) ){
           kSession.insert( new Nationalities( list ) );
           }
           for( List<Smoke> list: EnumUtil.permute( Smoke.class ) ){
                  kSession.insert( new Smokes( list ) );
           }
   }

   //...
}
```

Listing 24.7 contains the single rule that is now sufficient for combining attribute permutations according to the puzzle's constraints. Notice that the LHS begins with selecting Nationalities and Drinks; this is a good choice because each of these can be reduced individually, as the Norwegian lives in the first house and the milk drinker in the middle house.

Listing 24.7: The rule for the Zebra Puzzle

```
rule solve
when
   $n: Nationalities( $ln: list )
   eval( $ln.get(0) == Nationality.NORWEGIAN )
   $d: Drinks( $ld: list)
   eval( $ld.get(2) == Drink.MILK &&
          $n.indexOf( Nationality.UKRAINIAN ) ==
                 $d.indexOf( Drink.TEA ) )
   $c: Colours( $lc: list )
   eval( Math.abs( $n.indexOf(Nationality.NORWEGIAN) -
             $c.indexOf( Colour.BLUE ) ) == 1 &&
     $c.indexOf( Colour.GREEN ) ==
             1 + $c.indexOf( Colour.IVORY ) &&
     $c.indexOf( Colour.RED ) ==
             $n.indexOf( Nationality.ENGLISHMAN ) &&
     $c.indexOf( Colour.GREEN ) ==
             $d.indexOf( Drink.COFFEE ) )
     $s: Smokes( $ls: list)
   eval( $c.indexOf( Colour.YELLOW ) ==
             $s.indexOf( Smoke.KOOLS ) &&
     $n.indexOf( Nationality.JAPANESE ) ==
             $s.indexOf( Smoke.PARLIAMENTS ) &&
     $s.indexOf( Smoke.LUCKY_STRIKE ) ==
             $d.indexOf( Drink.ORANGE_JUICE ) )
     $a: Animals( $la: list )
   eval( $n.indexOf( Nationality.SPANIARD ) ==
             $a.indexOf( Animal.DOG ) &&
     $s.indexOf( Smoke.OLD_GOLD ) ==
             $a.indexOf( Animal.SNAILS ) &&
     Math.abs( $s.indexOf( Smoke.CHESTERFIELDS ) -
           $a.indexOf( Animal.FOX ) ) == 1 &&
     Math.abs( $s.indexOf( Smoke.KOOLS ) -
           $a.indexOf( Animal.HORSE ) ) == 1 )
then
   System.out.println( "Solution:" );
   System.out.println( $n );
   System.out.println( $c );
   System.out.println( $a );
   System.out.println( $d );
   System.out.println( $s );
end
```

Other constraints are written as boolean expressions based on method indexOf that obtains the index where the given enum value occurs in the referenced attribute list.

The reduction of fact combinations is quite impressive. Out of the 24,883,200,000 combinations, only 108 remain after looking at the drinks; after honouring the colours and the cigarettes we are down to 12 and 8, respectively. The single solution is printed like this:

```
Solution:
Nationalities[NORWEGIAN, UKRAINIAN, ENGLISHMAN, SPANIARD, JAPANESE]
Colours[YELLOW, BLUE, RED, IVORY, GREEN]
Animals[FOX, HORSE, SNAILS, DOG, ZEBRA]
Drinks[WATER, TEA, MILK, ORANGE_JUICE, COFFEE]
Smokes[KOOLS, CHESTERFIELDS, OLD_GOLD, LUCKY_STRIKE, PARLIAMENTS]
```

2.9 Active Facts

An active fact is not just a container in the JavaBeans style; its class also provides methods for doing some processing, usually involving data from the object and, possibly, additional arguments. There is one very simple reason to implement such computations as a class member: it helps to avoid exposing implementation details that should remain hidden.

Moreover, if you have several such classes and they are all subclasses of some (abstract) superclass, such a processing method could help you to reduce the number of rules dealing with facts from these classes.

As an example, look at class AndOp, which extends BinOp, shown in Listing 13. It uses Pin objects to represent the binary states of its input and output pins. The subclass does not add anything to the abstract base class; the only purpose of the new type is to provide a distinction between this class and sibling classes, such as OrOp or XorOp.

Listing 13: BinOp and AndOp as JavaBeans

```
public abstract class BinOp {
   private Pin inPin1;
   private Pin inPin2;
   private Pin outPin;

   public BinOp( Pin inPin1, Pin inPin2, Pin outPin ){
          this.inPin1 = inPin1;
          this.inPin2 = inPin2;
          this.outPin = outPin;
   }
   public Pin getInPin1() { return inPin1; }
   public Pin getInPin2() { return inPin2; }
   public Pin getOutPin() { return outPin; }
}

public class AndOp extends BinOp {
   public AndOp( Pin inPin1, Pin inPin2, Pin outPin ){
     super( inPin1, inPin2, outPin );
   }
}
```

Continuing this example, we can write a rule for computing the result of an AND gate, as shown in Listing 14. The inner logic of an AND gate is exposed in the constraint of the last Pin pattern and duplicated in the consequence, neither of which is "best practice".

Listing 14: Rule "AND gate"

```
rule "AND gate"
when
   $andOp: AndOp( $inPin1: inPin1, $inPin2: inPin2, $outPin: outPin )
           Pin( this == $inPin1 )
           Pin( this == $inPin2 )
           Pin( this == $outPin,
                  value != ( $inPin1.isValue() && $inPin2.isValue() ) )
then
   modify( $outPin ){
           setValue( $inPin1.isValue() && $inPin2.isValue() );
  }
end
```

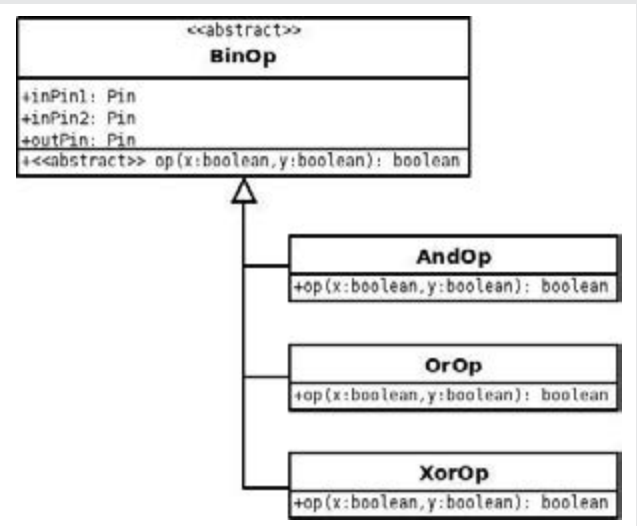

Figure 2.4: BinOp and Its Subclasses.

We can easily imagine that OR and XOR gates require almost identical rules, as only the logical operator differs in two places; however, all we need to know in the rule is that there is an operation to be performed for checking and updating the result pin. The implementation of this idea is shown in Listing 15. Calling the method is made easier by overloading method op with two accepting Pin objects as arguments. Subclasses merely need to provide an implementation of the method op with boolean arguments.

Figure 2.4 shows that fields are fully in the abstract base class BinOp so that its subclasses merely need to provide the implementation of the abstract method performing the operation.

Listing 15: BinOp and AndOp with "op" methods

```
public abstract class BinOp {
     private Pin inPin1;
     private Pin inPin2;
     private Pin outPin;

     public BinOp( Pin inPin1, Pin inPin2, Pin outPin ){
     this.inPin1 = inPin1;
     this.inPin2 = inPin2;
     this.outPin = outPin;
     }

     public Pin getInPin1() {
            return inPin1;
     }
     public Pin getInPin2() {
            return inPin2;
     }
     public Pin getOutPin() {
            return outPin;
     }

     protected abstract boolean op( boolean x, boolean y );

     public boolean op( Pin x, Pin y ){
            return op( x.isValue(), y.isValue() );
     }
}
public class AndOp extends BinOp {
   public AndOp( Pin inPin1, Pin inPin2, Pin outPin ){
          super( inPin1, inPin2, outPin );
    }

   protected boolean op( boolean x, boolean y ){
          return x && y;
    }
}
```

The rules implementing the activity of the AND, OR and XOR gates now collapse into a single rule. The duplication of the call of method op within the rule may be avoided by the addition of the simple method not() that inverts a pin's value. Also, perceiving that pins may be set by various agents, we should realize that changes to Pin objects are best propagated by the property change support, which relieves all callers to setValue or not from the obligation of informing the Engine that a fact has been updated. All of this combined is shown in Listing 16 and Listing 17.

Listing 16: Pin with property change support

```
public class Pin {
   private boolean value;

   private final PropertyChangeSupport changes =
          new PropertyChangeSupport( this );

   public Pin(){
          this.value = false;
   }

   public boolean isValue(){
          return value;
   }

   public void setValue( boolean value ){
          boolean oldValue = this.value;
          this.value = value;
          changes.firePropertyChange( "value", oldValue, value );
   }

   public void not(){
          setValue( ! value );
   }

   public void
   addPropertyChangeListener( PropertyChangeListener listener ){
          changes.addPropertyChangeListener( listener );
   }

   public void
   removePropertyChangeListener( PropertyChangeListener listener ){
          changes.removePropertyChangeListener( listener );
   }
}
```

Notice that Listing 17 contains a type declaration for Pin with the metadata definition for property change support – a convenient way of defining this feature for all inserted facts of this type.

Listing 17: Rule for a gate with two inputs

```
declare Pin
   @propertyChangeSupport(true)
end

rule "gate with two inputs"
when
   $binOp: BinOp( $inPin1: inPin1, $inPin2: inPin2, $outPin: outPin )
          Pin( this == $inPin1 )
          Pin( this == $inPin2 )
          Pin( this == $outPin,
                value != ( $binOp.op( $inPin1, $inPin2 ) ) )
then
   $outPin.not();
end
```

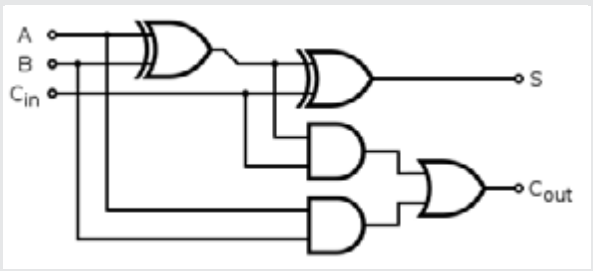

Figure 2.5: A Full Adder

These elementary classes may be used in a more elaborate environment. A full adder, as shown in Figure 2.5, is a combination of gates computing the sum and the carry of three binary inputs.

For a multiple-bit adder, 1-bit full adders are combined so that one of the inputs of a full adder is the carry output of the full adder of the previous adder. The resulting circuit computes the sum of two N-bit numbers. Listing 18 shows the Java classes implementing these circuits. The constructors support fact creation by accumulating the created component objects in lists that may be retrieved by the containing object and ultimately by the creator of the top-level object, simplifying the issue of getting all pins and gates cleanly into the Working Memory.

Listing 18: Full adder and Integer adder

```
public class FullAdder
{
   private Pin A, B, S, Cout;
   private List<Object> objects = new ArrayList<Object>();

   public FullAdder( Pin carry ){
     Pin rAND1, rXOR1, rAND2;
     objects.add( A = new Pin() );
     objects.add( B = new Pin() );
     objects.add( S = new Pin() );
     objects.add( Cout = new Pin() );

     objects.add( rXOR1 = new Pin() );
     objects.add( rAND1 = new Pin() );
     objects.add( rAND2 = new Pin() );
     objects.add( new XorOp( A, B, rXOR1 ) );
     objects.add( new XorOp( rXOR1, carry, S ) );
     objects.add( new AndOp( rXOR1, carry, rAND1 ) );
     objects.add( new AndOp( A, B, rAND2 ) );
```

```
      objects.add( new OrOp( rAND1, rAND2, Cout ) );
      objects.add( this );
    }

public Pin getA() { return A; }
public Pin getB() { return B; }
public Pin getS() { return S; }
public Pin getCout() { return Cout; }

public List<Object> getObjects() {
      List<Object> h = objects;
      objects = null;
      return h;
    }
}

public class IntAdder {
    private int width;
    private List<Pin> inA = new ArrayList<Pin>();
    private List<Pin> inB = new ArrayList<Pin>();
    private List<Pin> outS = new ArrayList<Pin>();
    private List<Object> objects = new ArrayList<Object>();

    public IntAdder( int width ){
           this.width = width;
           Pin carryOut = new Pin();
           objects.add( carryOut );
           for( int iBit = 0; iBit < width; iBit++ ){
                  FullAdder fullAdder = new FullAdder( carryOut );
                  inA.add( fullAdder.getA() );
                  inB.add( fullAdder.getB() );
                  outS.add( fullAdder.getS() );
                  carryOut = fullAdder.getCout();
                  objects.addAll( fullAdder.getObjects() );
           }
           objects.add( this );
    }

    public List<Object> getObjects() {
           List<Object> h = objects;
           objects = null;
           return h;
      }

    public void setA( int x ){ setRegister( inA, x ); }
    public void setB( int x ){ setRegister( inB, x ); }

    private void setRegister( List<Pin> pins, int x ){
           for( int iBit = 0; iBit < width; iBit++ ){
           Pin pin = pins.get( iBit );
              pin.setValue( x % 2 == 1 );
              x >>>= 1;
```

```
      }
    }

      public int getA(){ return getRegister( inA ); }
      public int getB(){ return getRegister( inB ); }
      public int getS(){ return getRegister( outS ); }

    private int getRegister( List<Pin> pins ){
      int res = 0;
      for( int iBit = width - 1; iBit >= 0; iBit-- ){
              Pin pin = pins.get( iBit );
              res = (res << 1) + (pin.isValue() ? 1 : 0 );
      }
      return res;
    }

    @Override
    public String toString(){
           return "A: " + getA() + ", B: " + getB() + ", S: " + getS();
    }
}
```

To illustrate how this works, we use a simple class Addition with integer fields a and b, and a rule setting the corresponding inputs of the IntAdder objects. The rule for this is shown in Listing 19.

Listing 19: Setting the registers of an IntAdder

```
rule "set A and B"
when
   $input: Addition( $a: a, $b: b )
   $adder: IntAdder()
then
   modify( $adder ){ setA( $a ), setB( $b ) }
   retract( $input );
end
```

A simple test for IntAdder is shown in Example 25, where an adder for 32-bit integers is created and an Addition object is inserted to trigger the rules.

Example 25: A test for IntAdder

```
IntAdder intAdder = new IntAdder( 32 );
for( Object object: intAdder.getObjects() ){
   insert( object );
}
insert( new Addition( 12345678, 87654321 ) );
```

2.10 Marker Facts

This section presents a technique that is frequently seen where rules have to operate on facts selected from a larger set. The underlying principle is simply stated by describing the cooperation of three rules.

1. The first rule fires, matching one or more facts. In the consequence of this rule, one or more "marker" facts containing a reference to the matched facts, or to facts obtained from them, are inserted.
2. A second rule is triggered by matching a marker fact. The fact to which the marker is bound is available via the reference in the marker. Besides doing the quintessential processing, the consequence of this rule may advance the marker to another fact. Thus, this rule may fire repeatedly.
3. A third rule is used for discovering conditions for terminating the propagation of markers, or to collect and retract them.

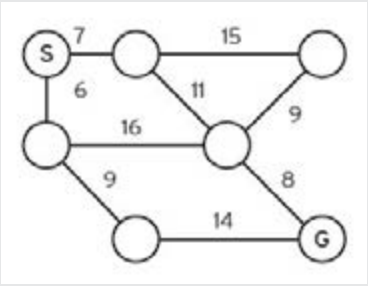

Figure 2.6: A Graph with Non-Negative Edge Costs.

2.10.1 Finding the Shortest Path

An application of this technical procedure is shown in a rule-based implementation of Dijkstra's algorithm [1] for finding the shortest path in a graph with non-negative edge costs, such as shown in Figure 2.6.

Coding follows the description of the algorithm given in the book "Algorithms and Data Structures" by Kurt Mehlhorn and Peter Sanders [2], and the interested reader may consult the textbook for a detailed description of the algorithm. Here it is sufficient to illustrate its workings by a very nice model. For this, you recreate the graph using thin threads along the edges and making knots at the nodes. Then, lift the starting knot until the entire network, down to the goal knot (and beyond), dangles freely. The sequence of taut threads from start to goal represents the shortest path.

The Java part of the implementation defines classes Node and Step. Both are simple JavaBean-style classes (see Listing 20).

Listing 20: Classes Node and Step

```
public class Node {
   private String       id;
   private boolean      start;
   private boolean      goal;
   private boolean      scanned;
   private boolean      initialized;
   private int          tentativeDist;
   private Node         parent;
   private List<Edge>   edges;

   // getters and setters
}

public class Step {
   private Node orig;
   private Node dest;
   private int cost;

   // getters and setters
}
```

Listing 21 shows the two main rules of Dijkstra's algorithm and two auxiliary rules.

Listing 21: The core rules of Dijkstra's algorithm

```
rule "relax"
agenda-group "relax"
when
   $node: Node( scanned == false, $td: tentativeDist < ( INFINITY ) )
          not Node( scanned == false, tentativeDist < $td )
then
   for( Edge e: $node.getEdges() ){
          insert( new Step( $node, e.getNodeB(), $td + e.getCost() ) );
   }
   modify( $node ){ setScanned( true ) }
   drools.setFocus( "accumulate" );
end

rule "accumulate cost"
agenda-group "accumulate"
when
   $step: Step( $na: orig, $nb: dest, $cost: cost )
   $node: Node( this == $nb, $td: tentativeDist > $cost )
then
   modify( $node ){
          setTentativeDist( $cost ),
                 setParent( $na )
   }
   retract( $step );
end

rule "delete unused Step facts"
salience -100
agenda-group "accumulate"
when
   $step: Step()
then
   retract( $step );
end

rule "leave accumulate"
agenda-group "accumulate"
when
   not Step()
then
   drools.setFocus( "relax" );
end
```

Rule "relax" locates the node with the smallest tentative distance, which is the best value determined for each node since the start of the algorithm. In the consequence, the marker nodes of class Step are inserted, one for each edge sprouting from the matched node. A Step object contains references to the connected nodes and the cost associated with this edge. After creating the markers, the set of "accumulate" rules is activated. Here, rule "accumulate cost" is

the workhorse, transferring, if it is less than the current optimum, the edge's cost from the Step object to the destination node; finally, the Step marker is retracted. Unused Step facts are simply discarded with a low salience rule, and another rule takes care of switching back to the "relax" agenda group.

Another marker fact is used while displaying the result: a Position fact contains a reference to a Node. Listing 22 contains the rules where it is used. It is inserted by the rule "show path length" and moved back towards the start node with each firing of rule "show path node". The last node retracts the position marker.

Listing 22: Displaying the shortest path

```
rule "show path length"
agenda-group "relax"
when
          Node( start == true, $si: id )
   $goal: Node( goal == true, $gi: id, $td: tentativeDist )
          not Node( scanned == false, tentativeDist < ( INFINITY ) )
then
   System.out.println( "shortest path from " + $si + " to " + $gi +
                       " costs " + $td );
   insert( new Position( $goal.getParent() ) );
end

rule "show path node"
agenda-group "relax"
when
   $node: Node( start == false )
   $pos: Position( node == $node )
then
   System.out.println( "via " + $node.getId() );
   modify( $pos ){ setNode( $node.getParent() ) }
end

rule "back at start"
agenda-group "relax"
when
   $node: Node( start == true )
   $pos: Position( node == $node )
then
   retract( $pos );
end
```

2.10.2 Investigating a Family Tree

Family trees are built according to an interesting graph structure. If edges are drawn between parent and child only, it should be free from cycles. As you might expect, connections between parent and child can be exploited in various ways, to detect other relationships. Given the availability of additional properties, a family tree is also suitable as a database for other tasks. This has led to the design of a class hierarchy suitable for the representation of royal families, where sovereigns and plain lords and ladies can be distinguished. The condensed Java code of the class hierarchy is presented in Listing 23.

Listing 23: Class hierarchy for a family tree

```
public abstract class Person {
   private String name;
   private String nick;
   private Person father;
   private Person mother;
   // ...
}

public class Lady extends Person {}

public class Lord extends Person {}

public abstract class Sovereign extends Person {
   private Date frm;
   private Date to;

   // ...
}

public class King extends Sovereign {}

public class Queen extends Sovereign {}
```

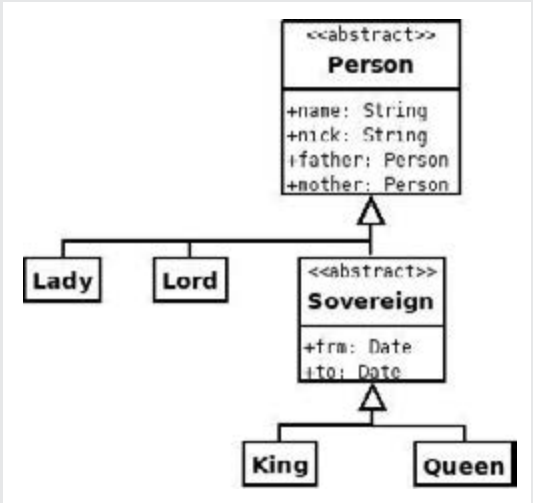

Figure 2.7: Person and Its Subclasses

A Person contains a field for the name and another one for his or her sobriquet (the courtly term for "byname" or "nickname"). Fields father and mother contain references to the person's parents, or null where we decide to cut the tree. For a sovereign, there are fields for registering the start and end year of his or her reign.

The UML diagrams shown in Figure 2.7 emphasize how all fields are kept in abstract classes, irrespective of a person's sex, this distinction being provided by the subclasses.

With a fact base like this, many evaluations are possible. A simple request processing scheme uses a class Request to apply some retrieval function to one of the persons in the tree. A rule aimed at one of the implemented function codes contains a pattern to locate the requested person. Simple functions can be dealt with by a single rule.

The simple functions are retrieval of a person's parents, children or siblings. The rules to achieve these three functions are shown in Listing 24.

Listing 24: Simple rules for a family tree

```
rule "locate children"
when
   $r: Request( function == Function.CHILDREN, $name: name )
   $p: Person( name == $name )
   $c: Person( father == $p || mother == $p )
then
   System.out.println( $c + " is a child of " + $name );
end
rule "locate parents"
when
   $r: Request( function == Function.PARENTS, $name: name )
   $c: Person( name == $name, $f: father, $m: mother )
then
   System.out.println( $f + " and " + $m +
                       " are the parents of " + $name );
end

rule "locate siblings"
when
   $r: Request( function == Function.SIBLINGS, $name: name )
   $p: Person( name == $name,
               $f: father != null, $m: mother != null )
   $s: Person( this != $p, father == $f, mother == $m )
then
   System.out.println( $s + " is a sibling of " + $p );
end
```

An interesting request results from the question: who is the successor of some sovereign? It's obvious that the links are of no avail. Therefore, we have to apply the condition that one sovereign reigns some later date to all Sovereign facts, and from them we have to pick the one that has no predecessor among them. The resulting rule is presented in Listing 25.

Listing 25: Finding the successor

```
rule "find successor"
when
   $r: Request( function == Function.SUCCESSOR, $name: name )
   $p: Sovereign( name == $name, $frm1: frm, $to1: to )
   $s: Sovereign( this != $p,
          $frm2: frm >= $to1 )
   not Sovereign( this != $s && this != $p,
          frm >= $frm1 && <= $frm2 )
then
   System.out.println( $s + " is the successor of " + $p );
end
```

A repeated application of this condition should result in finding the sequence of all sovereigns, starting with the first in our fact database. But there is one thing that needs to be added: the connection between repeated firings, which must maintain the "current" sovereign. A simple type called Marker with a single field for holding a Person reference is sufficient for this task.

The couple of rules solving this task is shown in Listing 26. The first rule finds the chronologically first sovereign and creates the Marker with a reference to the Sovereign object. The second rule repeatedly matches the Marker object, associates it (again) with the previously found sovereign, and locates the successor the way we have before. The third rule retracts the marker.

Listing 26: Finding all sovereigns

```
rule "find all sovereigns - 1"
when
   $r: Request( function == Function.SOVEREIGNS )
   $s: Sovereign( $frm: frm )
   not Sovereign( frm < $frm )
then
   insert( new Marker( $s ) );
   System.out.println( $s );
end

rule "find all sovereigns - 2"
when
   $m: Marker( $p: person )
   Sovereign( this == $p, $frm1: frm, $to1: to )
   $s: Sovereign( this != $p, $frm2: frm >= $to1 )
   not Sovereign( this != $s && this != $p, frm >= $frm1 && <= $frm2 )
then
   System.out.println( $s );
   modify( $m ){
          setPerson( $s )
}
end

rule "find all sovereigns - 3"
salience -100
when
   $m: Marker()
then
   retract( $m );
end
```

Applied to a fact database of the English Royals, the last three rules produce the output shown partly below.

```
 King William I the Conqueror (19 December 1066 - 3 September 1087)
 King William II Rufus (3 September 1087 - 26 July 1100)
 King Henry I (27 July 1100 - 24 November 1135)
...
 King Edward VIII (20 January 1936 - 11 December 1936)
 King George VI (11 December 1936 - 6 February 1952)
 Queen Elizabeth II (6 February 1952 - ?)
```

2.11 Fact Proxies

One description [3] for the intent of the design pattern called "Proxy" is to "provide a surrogate or placeholder for another object to control access to it." The Proxy pattern is one of the structural patterns, which typically provide an additional layer of access between the fundamental classes and the client code, which, in our case, is the DRL code.

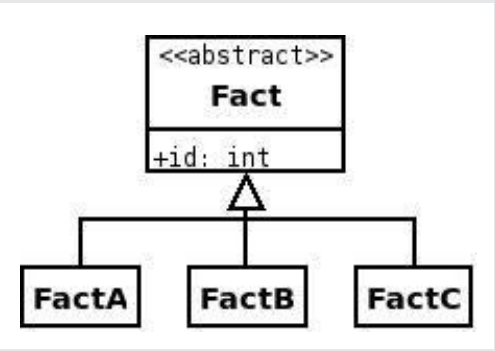

Figure 2.8: Person and Its Subclasses

How can adding proxy facts to the facts we have to deal with in the first place help? Why would we want to incur the overhead resulting from more fact types and the prospective complications in writing conditions?

2.11.1 Peephole Facts

A "peephole" fact confines the visibility of a large number of facts of the same type, thereby reducing potential matches. Although the matching prowess of the Inference Engine (IE) is respectable, exponentially growing Cartesian products are bound to have an impact. Peephole facts are one way of avoiding this.

Consider the simple case, as illustrated in Figure 2.8, where facts of three types with a common identification attribute have to be found, processed and retracted. The rule would be as simple as the one shown in Example 26.

Example 26: Combining three facts

```
rule "combine FactA, FactB and FactC"
when
   $a: FactA( $idA: id )
   $b: FactB( id == $idA )
   $c: FactC( id == $idA )
then
   // ...process $a, $b, $c
   retract( $a ); retract( $b ); retract( $c );
end
```

But as the fact counts for the three classes increase, a rapidly increasing amount of work and memory consumption goes on behind the scenes. Monitoring the Agenda shows that for n facts of each class n3 activations have to be created, resulting in the n expected firings.

We can now construct a peephole that significantly reduces the number of activations by creating a proxy fact that preselects a fact of some specific type and id. As we do not want to burden our Java application, we proceed to handle it all in DRL code, as shown in Listing 27. The type definition for the single additional type indicates that we use the fact class as an attribute to save us the bother of writing everything in triplicate. Rule "create Peephole" establishes another peephole fact for some fact class and id combination. The first pattern uses the type Fact, a common superclass or interface for the classes FactA, FactB and FactC. (We could avoid this by using the catch-all type Object and an additional constraint.) Rule "fill empty Peephole" does what this says; after it fires, the Peephole has grabbed another fact of the appropriate type. The last rule combines a triplet "seen" through three Peephole facts.

Listing 27: Reasoning through a Peephole fact

```
declare Peephole
   id: String
   clazz: Class
   fact: Object
end

rule "create Peephole"
when
   $e: Fact( $id: id )
   not Peephole( clazz == ($e.getClass()), id == $id )
then
   Peephole p = new Peephole();
   p.setId( $id );
   p.setClazz( $e.getClass() );
   insert( p );
end

rule "fill empty Peephole"
when
   $e: Fact( $id: id )
   $p: Peephole( clazz == ($e.getClass()), id == $id, fact == null )
then
   modify( $p ){ setFact( $e ) }
end
rule "combine FactA, FactB and FactC through Peepholes"
when
   $pa: Peephole( clazz == (FactA.class), $aid: id, $a: fact != null )
   $pb: Peephole( clazz == (FactB.class), id == $aid, $b: fact != null )
   $pc: Peephole( clazz == (FactC.class), id == $aid, $c: fact != null )
then
   // ...process $a, $b, $c
   retract( $a ); retract( $b ); retract( $c );
   modify( $pa ){ setFact( null ) }
   modify( $pb ){ setFact( null ) }
   modify( $pc ){ setFact( null ) }
end
```

But aren't the additional rules detrimental, causing us to pay too dearly? Such doubts are quickly resolved by a few test runs. Table 2.1 shows the results of running the rule set using Peephole proxies in comparison to those obtained with the plain rule shown in Example 2.26, always using 10 different id values and varying the number of facts of a type between 1 and 100.

But closer inspection of Table 2.1 does show a not quite agreeable rate of increase of the number of activation creations for the "Peephole" rules. Apparently, there is still a lot of wasted work being done in the network, due to the way the Fact objects are picked out, one by one.

TABLE 2.1: EXECUTION RESULTS PEEPHOLE VS. PLAIN RULES

Facts	Pain rule				Peephole rules			
	Time (s)	Created	Fired	Agenda max.	Time (s)	Created	Fired	Agenda max.
1	0.013	10	1	10	0.026	70	70	31
5	0.054	1250	50	1250	0.089	650	230	151
10	0.243	10000	100	10000	0.222	2050	430	301
50	6.780	1250000	500	1250000	0.814	40250	2030	1501
100	Out of memory				1.310	155500	4030	3001

2.11.2 Using a Collection as Proxy

The observation we made from the execution results in Table 2.1 suggests a procedure that collects facts of a kind as quickly and cheaply as possible and to use the resulting collections for combining them.

Again, we try to keep the technical details away from the application and do not change the way these facts are inserted. Listing 28 presents the DRL code, starting with the type definition of Sequence, the container for collecting facts of the same type and id. Its field facts of type java.util.List is where facts of the same class and id will be collected. Rule "create Sequence" is responsible for creating Sequence objects with a certain class and id combination. Rule "add to Sequence" snatches matching facts and adds them to the list; here the no-loop attribute is essential to avoid a loop by the rule being reactivated with the modified Sequence fact. The last rule combines three facts whenever all three lists are not empty, removes the head of the list, updates the Sequence fact, and processes the triplet as usual.

Listing 28: Reasoning over Sequence facts

```
declare Sequence
   id: String
   clazz: Class
   facts: ArrayList
end

rule "create Sequence"
when
   $e: Fact( $id: id )
   not Sequence( clazz == ($e.getClass()), id == $id )
then
   Sequence s = new Sequence();
   s.setId( $id );
   s.setClazz( $e.getClass() );
   s.setFacts( new ArrayList() );
   insert( s );
end

rule "add to Sequence"
```

```
no-loop true
when
   $e: Fact( $id: id )
   $s: Sequence( clazz == ($e.getClass()), id == $id )
then
   modify( $s ){
          getFacts().add( $e )
   }
end

rule "combine FactA, FactB and FactC from Sequence"
when
   $sa: Sequence( clazz == (FactA.class), $aid: id,
                  eval( $sa.getFacts().size() > 0 ) )
   $sb: Sequence( clazz == (FactB.class), id == $aid,
                  eval( $sb.getFacts().size() > 0 ) )
   $sc: Sequence( clazz == (FactC.class), id == $aid,
                  eval( $sc.getFacts().size() > 0 ) )
then
   FactA $a = (FactA)$sa.getFacts().remove( 0 );
   update( $sa );
   FactB $b = (FactB)$sb.getFacts().remove( 0 );
   update( $sb );
   FactC $c = (FactC)$sc.getFacts().remove( 0 );
   update( $sc );
   // ...process $a, $b, $c
   retract( $a ); retract( $b ); retract( $c );
end
```

Table 2.2 shows how nicely we were rewarded for our efforts.

TABLE 2.2: EXECUTION RESULTS PEEPHOLE VS. SEQUENCE RULES

Facts	Pain rule				Peephole rules			
	Time (s)	Created	Fired	Agenda max.	Time (s)	Created	Fired	Agenda max.
1	0.026	70	70	31	0.031	70	70	31
5	0.089	650	230	151	0.099	374	230	151
10	0.222	2050	430	301	0.211	727	430	301
50	0.814	40250	2030	1501	0.736	3696	2030	1501
100	1.310	155500	4030	3001	1.093	7297	4030	3001

2.12 Other Structural Patterns

Not all of the classic design patterns make sense in the context of rules—especially the behavioural patterns, where "behaviour" is represented by methods. Some structural patterns, however, might come in handy.

The Adapter pattern converts the interface of a class into what clients expect. Assume that you have a working set of rules expecting some data in a single fact object, but the facts you have contain that data in two or more related facts. You can still use the rules you have in combination with the class containing the assembled data. Assembly might very well be done through an additional rule, and this could advantageously use the fact insertion method insertLogical, as discussed in Subsection 4.4.

Another pattern that is a potentially useful pattern for rules is Decorator, which is used for attaching additional responsibilities to an object dynamically. Normally, adding fields and methods to a class is done by subclassing, but it's not always convenient or possible to do that.

Typically, an extension for an object of class Item contains a field of type Item for referencing the decorated object, and one or more fields for the additional properties. The resulting rule structure is straightforward, as outlined in Example 27.

Example 27: Matching a fact with an extension

```
rule "find comment author"
when
   $item: Item( ... )
   Comment( item == $item, $author: author,... )
   Author( this == $author )
then
   // ...
end
```

Since a single fact object can have multiple decorations of the same or different types, this approach is even more versatile than simple subclassing.

CHAPTER 3: APPLICATION DESIGN PATTERNS

3.1 Planning for an Application Using Drools

Drools can be used in a wide range of application scenarios, ranging from simple interactive programs to complex web-based services, or from small-scale PC "toy" programs to enterprise applications with database backup. Advances in the size of memory available to programs and the CPU processing speed have made Rete-based rule engines suitable for use in interactive applications where a high degree of responsiveness is essential.

From the survey of the most important Drools API functions in Chapter 4.5 it is evident that there is a rich arsenal of weaponry with which a plethora of problems can be tackled. Best service, however, is achieved by a selection of functions best suited to the task at hand. The following survey intends to make you aware of the criteria for selecting some Drools API function combination in favour of another. Subsequent sections discuss some typical application scenarios.

The data flow of facts is a key issue for deciding about the application design. Distinguishing between facts with external origin and internally created auxiliary facts is one aspect. Another one is the lifetime of facts as they should be present in a Working Memory, and here you should distinguish between facts that must continuously represent the state of some entity as opposed to facts that just need to be evaluated before they may be discarded.

Another distinction should be made between dynamic facts representing data to be analysed, a query or some similar transaction, and static facts that are the same for each run. The latter may range from a few parameters stored as fields in one or more facts up to an encyclopaedic collection of propositions.

The quantity structure of facts is bound to have an impact on the resource footprint of your application if it turns out to require keeping a large number of facts in Working Memory at the same time. But frequently enough, closer inspection may reveal that only a subset of facts is really required to be around concurrently, and that a relatively simple strategy for discarding processed facts and for pulling in fresh supply does the trick.

The remainder of this chapter is divided into two sections. Section 3.2 discusses short-term sessions, whereas Section 3.3 is dedicated to sessions that keep on running for an extended period of time.

3.2 Short-Term Sessions

3.2.1 Short Execution Cycle

Starting with a serialized Knowledge Base and creating a session from it is not expensive, and this circumstance promotes short-term sessions. For using a Stateful Knowledge Session, applications follow a program pattern consisting of the following steps:

1. Call the Knowledge Base method newStatefulKnowledgeSession to create the session.
2. Populate the Working Memory by inserting facts.
3. Call the session method fireAllRules.
4. After the safekeeping of all results, the application can terminate. Alternatively, another cycle for a new fact set may be prepared by calling the session method dispose.

Calling dispose is an important step (unless you terminate the application right away) that releases all resources allocated by the session. Simply erasing all references to the session object is not sufficient.

Using single letters as a shorthand for "session", "insert", "fire" and "dispose", we can describe this session pattern as (SIFD)*. Such sessions are very well suited for simple validation and analysis tasks. Even when the full power of a production rule system is not utilized, running a stateful session permits the insertion of derived facts, created and inserted in consequences.

3.2.2 Multi-Stage Execution

Execution patterns that are a little more complex emerge when rule groups are to be applied consecutively, in two or more stages. Rule grouping is achieved by the rule attribute agenda-group, as described in Section 4.6.

1. Call newStatefulKnowledgeSession to create the session.
2. Populate the Working Memory by inserting facts.
3. Call the session method fireAllRules.

4. Change the "focus", i.e., apply the rules of another agenda group, and repeat from the previous step. Alternatively, continue with the last step.
5. After the safekeeping of all results, the application can terminate. Alternatively, another overall cycle for a new fact set may be prepared by calling the session method dispose.

Notice that the inner cycle is not necessarily a loop but a succession of a focus change (C) followed by another activation of the IE. Using such a pattern, characterized by (SIF(CF)+D)*, is utilizing a technique known as "rule flow" in its simplest form.

Theoretically, organizing some rule flow is never necessary, because the indeterminism of rule firings can be countered by incorporating additional constraints or more elaborate sequencing mechanisms in the design of a set of rules. Also, for a well-designed rule set the results should be the same irrespective of variations in the firing order of the rules. But adding sequencing mechanisms tends to be cumbersome, and this is one good reason for using a rule flow. Also, letting rules do their work in stages has an interesting effect: it produces intermediate and final results in a less random order. If, for instance, the first rule group is dedicated to checking the initially inserted facts, the second one collects matches to produce intermediary results which the third one should consolidate, all error messages from the first stage precede all the results. This is less confusing to the user than a random mixture. Another goody of staging is that the WM is in an "unmixed" composition when the system advances from one stage to the next, and this facilitates testing and trouble-shooting.

3.2.3 Session Chaining

Occasionally designers are confronted with a situation where facts have to be processed by two or more sets of rules, each of which is known to be applicable to a distinct subset of the facts. If there are many rules and many facts, the overhead may become an issue, and you might consider to use "session chaining", a simple concept that is easy to implement.

The separate rule sets furnish individual Knowledge Bases for processing the distinct fact subsets. Assuming that the initial disposition of facts is also to be done by rules, we can create a session frontend from their Knowledge Base. Other sessions are established from the Knowledge Bases for the individual fact subsets. Their Stateful Knowledge Session objects need to be made available to the frontend. One good way of doing this is to enter them in a Map and store this in a global variable of the frontend; the Map key represents the fact category, to be established by the frontend rules.

Whenever a rule in the frontend has determined the category of a new fact, the consequence must retract the fact from the frontend's Working Memory and insert the fact into the Stateful Knowledge Session that can be retrieved from the global Map via the classification key. Unless the sessions are kept active all the time, fireAllRules should be called, too.

Similar chaining could be employed for passing facts through a sequence of engines, each of which would be driven by a distinct set of rules.

3.2.4 Master Facts, Variable Facts

Classic data processing has been using the terms "master data" and "variable data" from early days on. Fact data frequently exhibits traits that justify a corresponding distinction. Given that there is a large number of master facts against which a small number of variable facts has to be evaluated repeatedly, we are confronted with the problem of finding efficient ways of repeatedly establishing the same fact database. The previously described patterns, (SIFD)* and (SIF(CF)+D)*, are not quite as attractive any more because the initialization step must insert the same set of master facts for each session. What we would like to have is a pattern where we have separate insertion steps for master (I_m) and variable (I_v) facts and, more importantly, a

way of "cleaning out" the working memory after one batch of variable facts has been processed. This must be done in an additional step (R) where you retract all facts inserted after the insertion of the master facts. Then, the repetition is restricted to an inner loop, where only inserts, firing and retracts happen, i.e., $SI_m(I_vFR)+D$. Multiple stage processing proceeds likewise.

Listing 29 presents a simple mechanism for returning a Knowledge Session to some previous state, defined by the set of all facts in the Working Memory. It obtains the set of facts that should be retained in the "reset" operation and stores it as a set of FactHandle objects. Resetting is done by iterating over all currently existing handles and retracting all that are not contained in the stored set.

Listing 29: A fact marker

```
public class FactMarker {
   private StatefulKnowledgeSession kSession;
   private Set<FactHandle> factHandleSet;

   public FactMarker( StatefulKnowledgeSession kSession ){
     this.kSession = kSession;
     Collection<FactHandle> factHandles = kSession.getFactHandles();
     factHandleSet = new HashSet<FactHandle>( factHandles );
   }

   public void reset(){
          for( FactHandle factHandle: kSession.getFactHandles() ){
                if( ! factHandleSet.contains( factHandle ) ){
                    kSession.retract( factHandle );
                }
          }
   }
}
```

3.2.5 Session Pools

Whenever a session is provided as the backbone of a Web service, running several sessions in parallel may become advisable in order to keep response times as short as possible. Basically, the pool objects are threads, and these threads may use one of the design patterns presented in the previous subsections.

All the usual guidelines for thread pools apply here as well. Pool size should be limited according to the availability of system resources. Typically, the amount of fact data users may pass in for a single request will be limited, and rules should not create large quantities of derived facts.

3.3 Permanent Sessions

A permanent session is one that stays alive within the context of an application that keeps on running. The main reason for doing so is the existence of facts used to store the states of some external entities, with the intent of evaluating these states in combination with various event categories. Whether these "events" are actually implemented as events (as defined in Section 4.7) or remain plain facts is secondary; all session patterns described in this section can be used with and without Drools' event processing support.

For permanent sessions, two variants should be distinguished, depending on the way rules are fired: either with fireAllRules or with fireUntilHalt.

3.3.1 Repeated Activation

This pattern is characterized by $SI_m(I_eF(CF)^*)^+$. After creation and insertion of permanent master facts, the repeated execution cycle proceeds with the insertion of one or more events, the call to fireAllRules and, optionally, a rule flow through some agenda groups.

It must not be overlooked that the apparently simple step sequence I_eF is not without pitfalls. Its detailed organisation deserves some consideration regarding the choice between bulk insertion and inserting facts one by one.

Bulk insertions of several facts followed by a single call to fireAllRules are dangerous because activations are collected on the agenda during the insertion of facts and fired due to the call. Now consider that the execution of one of the consequences causes a change in the master set of facts that, in turn, causes one or more of the activations due to the insertions to be removed again. This may appear as rules failing to fire.

On the other hand, calling fireAllRules after each insertion is not unconditionally advisable either. If your rule set expects logically related sets of facts, and expects them to be complete, firing a partially completed set of activations may result in firing a rule that claims that something is amiss.

As a consequence, we can establish a design pattern that is transaction oriented.

3.3.2 Cyclic Transaction Processing

The Java logic for cyclic transaction processing of fact sets follows the structure presented in the previous subsection: $SI_m(I_eF(CF)^*)^+$. But in order to make it safe, calling fireAllRules must be done exactly whenever another set of facts has been inserted. A generic framework for running a rule-based application according to this mode must not only provide a method for inserting a single new fact, but also a method for indicating the end of a logically related set. As an alternative, such a set may also be presented by a method accepting a Collection of objects.

Straightforward execution of a set of insertions followed by a call to fireAllRules does not create any difficulties. But, since the term "transaction" has been used in this subsection's heading, we might ask whether it is possible to come close to the idea of a transaction as they are usually implemented in database systems, i.e., as operations that must succeed or fail as a unit, not leaving the data in some intermediate state. For this, a function known as rollback with the typical operations for starting, committing, and aborting a transaction must be implemented. Rollback, i.e., the undoing of all changes, is essential for the safe abortion of a transaction.

There is one feature of production rule systems that makes them fundamentally different from database systems: the interleaving of update operations on the fact database and the execution of consequences due to rule firing. The latter means that a transaction triggered by fireAllRules after the insertion of one or more facts may have caused irreversible side effects outside the scope of the fact database itself. We'll discuss a possible technique for this later on; for the time being we'll assume that there are no such side effects.

Implementing a rollback for Working Memory changes requires two things: registering the state of the system when a transaction starts or keeping track of all changes and returning the system to the initial state when the transaction is aborted. Registering the system state by saving the entire fact database (and more) is, in general, not advisable. Making a memento of all changes is well supported by the provision of listeners for Working Memory Events; the Working Memory Event objects contain references to objects and corresponding fact handles.

A memento of changes can be processed in reverse order. A retract is undone by an insert; an insert is nullified by a retract; an update is turned back by a retract and an insert. All of these reversals are enacted while the engine keeps on updating the Rete network, which may result in new activations being created. Therefore, after the end of the rollback, method clear of Agenda in package org.drools.runtime.rule must be called. The Agenda object can be retrieved by a call to method getAgenda of WorkingMemory. This will also take care of any activations that were still pending at the time the transaction was aborted.

Note that some optimization is possible by skipping pairs of insertions and retractions.

In order to let rollback also cancel changes on some resources that are not facts in Working Memory, these actions must be queued. Most services, such as writing lines to a Print Stream, require only a queue with simple entries, and therefore action queuing should not cause any problems. Any object containing a queue might also be designed as a proxy for the destination object, which would even make the mechanism transparent to the rule programmer.

3.3.3 Asynchronous Sessions: Fire Until Halt

An asynchronous session is one that executes fireUntilHalt in a thread of its own, while other threads are listening to data sources and inserting facts.

This kind of session is subject to a fundamental constraint due to the possibility of rules firing instantaneously after an insertion or any other Working Memory change. If groups of facts must be evaluated jointly, rules must be written to recognize complete groups, which must be indicated by appropriate marks in the data. But whenever facts arriving from more than one source are to be processed one by one—a situation typical for Event Processing, for instance—this type of session can be used without the need of additional synchronisation between the various threads servicing the sources.

We should note in passing that the JDK package java.nio lets you deal with an arbitrary number of data channels without the overhead of running threads for each connection. Ultimately, this would let you use a synchronous session even when there are many data source channels.

The remainder of this section is dedicated to the presentation of some basic patterns for subclasses of java.lang.Runnable as they might be used for establishing an asynchronous session. The central thread is dedicated to running the session. Other threads provide fact data, or monitor the session.

3.3.4 A Session Runner

A "session runner" is a thread dedicated to running a Stateful Knowledge Session with a single call to fireUntilHalt. Listing 30 shows a simple class extending Thread. The factory method for creation establishes a thread group and creates a new instance unless there already is an active one.

Listing 30: Class SessionRunner

```
public class SessionRunner extends Thread {
   private static ThreadGroup threadGroup;
   private static SessionRunner sessionRunner;

public static SessionRunner newSessionRunner(
          StatefulKnowledgeSession kSession ){
   if( sessionRunner != null ){
          throw new IllegalStateException( "already active" );
   }
```

```
    if( threadGroup == null ){
           threadGroup = new ThreadGroup( "Drools group" );
    }
    sessionRunner = new SessionRunner( threadGroup, kSession );
    return sessionRunner;
}

private StatefulKnowledgeSession kSession;

protected SessionRunner( ThreadGroup threadGroup,
          StatefulKnowledgeSession kSession ){
    super( threadGroup, "session runner" );
    this.kSession = kSession;
}

@Override
public void run() {
    kSession.fireUntilHalt();
    sessionRunner = null;
    }
}
```

3.3.5 A Fact Feeder

A "fact feeder" is a thread monitoring an internal or external fact source. Given access to the session that is being run in a parallel thread, the fact feeder inserts another fact, either as soon as it has been received or delayed according to some directive associated with the fact data. Here are two typical use cases where these alternatives crop up:

The IE is running in a long-lived application operating in real time and with fact data, usually describing events, arriving from environmental sources.

Rules are used for processing multiple streams of recorded events.

Listing 31 presents the abstract framework for a fact feeder. It uses objects of class Action containing data controlling its activity, and the abstract class Event as a placeholder for fact objects. Its method run iterates over the sequence of Action objects provided by an implementation, processing them one by one, by delaying for the requested duration and inserting the Event after setting its timestamp value.

Listing 31: Class EventFeeder and its auxiliary types

```
public abstract class Event {

    private static final Format dateFmt =
           new SimpleDateFormat( "yyy-MM-dd HH:mm:ss.SSS" );

    private Date timestamp;
    private long duration;

    public Event(){
    }
```

```
    public Event( long duration ){
           this.duration = duration;
    }

    public Date getTimestamp() {
           return timestamp;
    }

    public void setTimestamp(Date timestamp) {
           this.timestamp = timestamp;
    }

    public long getDuration() {
           return duration;
    }

    public void setDuration(long duration) {
           this.duration = duration;
    }

    @Override
    public String toString(){
           StringBuilder sb = new StringBuilder( "event at " );
           sb.append( dateFmt.format( timestamp.getTime() ) );
           if( duration > 0 ){
                  sb.append( " for " ).append( duration );
           }
           return sb.toString();
    }
}

public interface Action {
    public Event getEvent();
    public float getDelay();
    public void setDelay(float delay);
}

public abstract class EventFeeder implements Runnable {

    private StatefulKnowledgeSession kSession;

    protected EventFeeder( StatefulKnowledgeSession kSession ){
           this.kSession = kSession;
    }
    protected static void startFeeder( EventFeeder eventFeeder ){
           Thread thread = new Thread( eventFeeder );
           thread.start();
     }

    protected abstract Iterable<?> getActions();
```

```
    public void run() {
           Iterable<?> actions = getActions();
           for( Object obj: actions ){
           Action action = (Action)obj;
           Event event = action.getEvent();
           float delay = action.getDelay();
           int ms = (int)(delay*1000);
           if( ms > 0 ){
                  try {
                      Thread.sleep( ms );
                  } catch (InterruptedException e) {
                      e.printStackTrace();
                  }
           }
           long currTime =
                  kSession.getSessionClock().getCurrentTime();
           Date timeStamp = new Date( currTime );
           event.setTimestamp( timeStamp );
           kSession.insert( event );
           }
    }
}
```

Class EventFeeder can now be subclassed, with the option of overriding getActions in several ways. The implementation could access a file containing some recorded data (perhaps in XML), or it could receive event data from a socket connection, using "natural" delays between fact insertions. Moreover, any number of sources can be created for feeding a single session.

CHAPTER 4: NOTES ON RELATED TECHNIQUES

4.1 Decision Tables in Spreadsheets

This rule authoring technique uses a spreadsheet with a certain arrangement and disposition of cells to define sets of rules, each of them with a common structure. Variations are possible by using different values to be included in conditions and actions, or by omitting constraints and actions altogether.

4.2 Rule Templates

Technically related to decision tables, producing rules from rule templates uses parameterized text skeletons and an expander that inserts actual string parameters in place of the formal ones.

4.3 Extending a Rule Definition

Extending a rule by another rule results in two rules where an activation of the first rule is a precondition of the second rule. This means that patterns common to two or more rules can be written as a single, separate rule that is extended by two or more rules extending this rule and where only the remaining patterns and constraints need to be written.

4.4 Truth Maintenance

The term truth maintenance is used for a feature where a fact is inserted into WM with the connotation of depending on the continuing truth of a condition. As soon as this condition turn to false, the system automatically retracts the dependent fact.

Note that there may be noteworthy differences between the way truth maintenance works in Drools, as compared to other RBS. In Drools, the condition is always the LHS of a rule and the programmer creates a dependent fact by calling insertLogical.

In another RBS, a special CE must be used for bracketing the condition, which may be the initial part of some LHS, but then all facts inserted on the RHS of that rule become dependent of that condition.

4.5 Application Programming Interface

Drools API calls must be used to compile knowledge resources, e.g., files containing rules in the DRL language, to build a knowledge base from compiled DRL packages and to launch knowledge sessions from a knowledge base. With a session, fact insertion and running the session to fire rules are the most frequently used API calls.

Drools does not provide a "shell" that can be used for invoking DRL "programs". Although it is not difficult to provide a generic application capable of loading and executing one or more DRL files, fact insertion is difficult, as this has application-specific dependencies.

4.6 Rule Flow by Agenda Groups

The rule attribute agenda-group, followed by a string literal, puts a rule into a group with that name. The idea, then, is that rules from such a group will only fire whenever the group has been given the focus, which must be done with an API call. After all rules from the focussed group have fired, the focus automatically returns to the group that had it previously. Any number of groups may be in some semi-finished state of the agenda group stack.

4.7 Complex Event Processing

In Drools you can annotate any fact class with @role(event) and this adds two properties: a timestamp (type java.lang.Date) and a duration (type long, in milliseconds), which may coincide with properties already contained in the class. On request, Drools will add the timestamp when an event is inserted, and it may retract events automatically if they can't participate in rule firings anymore. Reasoning about event correlations is simplified by a number of temporal operators, e.g., before or during.

BIBLIOGRAPHY

1. Edsger Wybe Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1: 269-271, **http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf**
2. Kurt Mehlhorn, Peter Sanders, Algorithms and Data Structures, Springer-Verlag Berlin Heidelberg, 2008
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley, 1995

www.redhat.com
#10061027_0213