redhat.

TECHNOLOGY DETAIL

RED HAT JBOSS BUSINESS RULES MANAGEMENT SYSTEM

Best Practices Guide

CONTENTS

1 OVERVIEW

This document contains information related to various best practices for development with Red Hat JBoss Business Rules Management System (BRMS). It will focus on three key areas of rules development; designing the domain or fact model, rule authoring, and various aspects related to rule execution.

Following the best practices outlined in this guide will result in systems with better design, easier testability, and superior maintainability. Developers will also be able to validate Red Hat JBoss BRMS requirements more easily. You do not need to read the content in sequence.

Please note: The code examples throughout this document utilize the 'mvel' dialect.

1.1 DROOLS AND RED HAT JBOSS BRMS

Drools is a family of related open source projects that are part of the JBoss.org community. The Drools community projects focus on fast paced innovation to give you the latest and greatest; with rapid releases every few months that include both new and sometimes experimental features. However, these community projects do not come with any support.

Red Hat JBoss BRMS is the enterprise product. It is based on a community release of Drools that is integrated, tested, and certified to work on a wide range of platforms and systems. It is designed for typical enterprise usage scenarios and includes patches, updates, SLA–based support, multi-year maintenance policies, and Red Hat Open Source Assurance. Red Hat JBoss BRMS offers the best of both worlds–rapid innovation, with long-term stability, supportability, and maintainability.

To learn more about using JBoss Community or Red Hat JBoss BRMS, please visit **http://www.redhat.com/brms.**

2 BUSINESS RULES DOMAIN MODEL

The business domain model (typically referred to as the fact model) is the foundation for expressing business requirements, and the quality of this model impacts how easy it is to translate business requirements into rules. As such, the business domain model greatly affects the way developers author rules. This section discusses various best practices for designing a good business domain model from a rules perspective.

Please note that in Red Hat JBoss BRMS, the business domain model is defined using Java Beans. These beans become the facts that you, the developer, will use when reasoning.*

* Note: Since Red Hat JBoss BRMS is based on Java, the same naming conventions that apply to Java also apply to JBoss BRMS.

2.1 DOMAIN MODEL STRUCTURE

When working with Red Hat JBoss BRMS, programming to interfaces is important. Developers can reason over different classes that implement the same interface just by using this interface in a rule's condition. By using an interface instead of the class itself, you can declare a rule's intent more specifically. As a result, rules are easier to understand. For example, imagine that `Student implements Person.` The rule pattern `Person ( )` will match all facts implementing the Person interface, including all `Students.`

If you are writing rules against read-only data, consider flattening your domain model. Rules work best with flat domain models because there is no need to traverse many object relationships when reasoning about something.

In a hierarchical domain model one `Customer` could have many `Addresses`. For instance, if you wanted to find a `Customer` that has an `Address` with no city, you would write your rule like this:

```
rule addressWithoutCity
    when
        $address : Address( city == null )
        Customer( address contains $address )
    then
        //...
end
```

Instead of having many `Addresses` for each `Customer`, a flat domain model allows all `Address` properties to be put into `Customer`. It is important to note, however, that this may result in more `Customer` instances if the customer has more than one address. This rule could be written as:

```
rule addressWithoutCity
    when
        Customer( addressCity == null )
    then
        //...
end
```

Obviously the disadvantage of flat domain models is the duplication of data and higher memory requirements. On the other hand, they better offer performance, because the rule engine has to do less work. The given example was simple, but imagine a bigger model where a `Doctor` can work at multiple `Clinics` for different time periods. Introducing a `DoctorSnapshot` fact would capture all information about a doctor at a clinic for a specific time period. Then it would be much easier to implement business logic for a specific point in time.

Even though flat domain models are very useful, there are some cases when they are not ideal; for example, when rules modify facts. Here it is better to have multiple small facts rather than a few large ones with many properties. When a fact is updated, all rules that use that particular fact in their conditions must be reevaluated. However, it goes without saying that properties that change together should stay together. Do not make the facts too small and needlessly complicate your model.

To summarize, use flat models for read-only data and use small facts (i.e. facts with fewer properties) when they will be updated often.

2.2 WHAT FACTS TO INSERT?

One of the first questions every developer working with Red Hat JBoss BRMS faces is this: What facts do I insert into the knowledge session? In general, you should only insert objects that you will be working with. There is no point in inserting an object into the knowledge session if there is no rule or query interested in it. Since these objects may create unnecessary matches that you will need to deal with later, it is better not to insert them at all. For example, if you are validating only one specific customer, there is no need to also insert any other customers into the knowledge session.

Consider a scenario where you need to validate a `Customer`. To accomplish this, you would first insert all facts related to this `Customer` into the knowledge session. Since we know that there is only one `Customer` in the knowledge session, matching data against this particular `Customer` is unnecessary.

So instead of writing the follow pattern matching script:

```
$address : Address( )

$customer : Customer( address == $address )
```

You can just write:

```
$address : Address( )

$customer : Customer( )
```

It is possible to avoid the `Customer` completely if it is not needed in the rule's consequence.

JBoss BRMS also supports nested accessors, such as

```
Order( customer.address.city != null ) ).
```

In this example, the pattern is matching on `Orders` that have a `customer` that has an `address` with the `city` set. The `address` object is accessed through a nested accessor. It does not need to be inserted into the knowledge session. However, this feature should be used with caution. (See section 2.4 – **Changing a fact**)

2.3 WHEN TO INSERT FACTS?

The next question developers must consider is when to insert facts into the session. Some initial facts need to be inserted from Java code before we call the `fireAllRules` method. Additional facts can be loaded by "setup" rules. These rules are executed first and their purpose is to initialize the knowledge session (that is to load, initialize, and insert facts). It is not necessary to load every fact at the outset; a better alternative is to load facts as they are needed.

The advantage of using "setup" rules as opposed to writing plain Java code is that everything is contained in one place. If you need to load additional data, you can just write a new "setup" rule without leaving our rule file.

For example, here is a setup rule that loads `Products` for a `Customer`:

```
global ProductService productService
rule loadProducts
  ruleflow-group "setup"
    when
        $customer : Customer( )
    then
        for (Product product : productService.
          loadProductsByCustomerId($customer.id)) {
            insert(product);
        }
end
```

Please note that the rule is in a "setup" ruleflow group. It is a good practice to separate the rules that load data from the other rules. This makes it easier to stop them from looping.

If possible, also avoid loading facts in a rule's condition (i.e., the when part or left hand side). This is an anti-pattern and you should only attempt this if you have a clear idea of what the results will be.

The following is an anti-pattern example that is shown for instructional purposes only. It should not be emulated in practice:

```
rule loadProducts_DontDoThis
    when
        Product( ) from productService.loadProducts()
        //...
    then
        //...
end
```

The impact is that the method call to `loadProducts` might be executed multiple times by the rule engine – without giving you control over it.

Another way to make data available to the rule engine is to use the **"`from`"** keyword. However, using "`from`" should be minimized (see section 3.10 – **Avoid using the from keyword**).

2.4 CHANGING A FACT

Sometimes a fact is no longer true and needs to be modified, for instance, when changing a property on an object. As a best practice, you should notify Red Hat JBoss BRMS every time you make a change to a fact from within a rule using the "`modify`" construct.

Here is an example (taken from a rule's consequence):

```
rule changeFact
when
  ...
  then
    modify($customer) {
      setFirstName("John")
    }
end
```

It is possible to also change a fact from outside the rule engine. In this case, you should use the `StatefulKnowledgeSession`'s `update` method.

You may be wondering if you should notify JBoss BRMS that a property has been changed even if there is currently no rule triggering on this property. As mentioned above, any time a fact is changed, the rule engine needs to be notified. By consistently notifying the rule engine, you can ensure that all changes to the rules – even if other developers make them, or you made them so long ago you forgot about them – are accounted for and prevent instances where code may stop working as expected.

2.5 NESTED ACCESSORS

Here is an example of a nested accessor:

```
Customer( address.city != null )
```

These should be avoided if the nested facts are also in the knowledge session. Otherwise, every time you change a child fact you would need to notify Red Hat JBoss BRMS that not only the child fact has been changed but also the parent fact as well.

Let's look at an example using two facts, `Customer` and `Address`, in greater detail. Each `Customer` will have one `Address` and each `Address` will have a `City` property.

```
declare Address
  city : String
end
declare Customer
  address : Address @key
end
```

Then, write an initialization rule that will insert one `Customer` fact and one `Address` fact into the session. The `city` property will be set to `null`. The `Customer` fact will reference this `Address` fact:

```
rule initialize
    when
    then
        Address address = new Address((String)null);
        insert(address);

        Customer customer = new Customer(address);
        insert(customer);
end
```

Now let's write a rule that sets the city to "`Berlin`":

```
rule setCity
    when
        $address : Address( city == null )
        //$customer : Customer( ) //<-- uncomment this later
    then
        modify($address) {
          city = "Berlin"
        }
        //update($customer); //<-- uncomment this later
end
```

Ignore the commented lines for the time being; they will be uncommented later. Now, add a rule that uses a nested accessor to trigger on a `Customer` that references an `Address` with no `city` set:

```
rule addressWithCity_nestedAccessor
    when
        Customer( address.city != null )
    then
end
```

Writing the same rule without using a nested accessor results in:

```
rule addressWithCity
    when
        $address : Address( city != null )
        Customer( address == $address )
    then
end
```

If we fire all rules, we'll see that the following rules fire: `initialize`, `setCity`, and `addressWithCity` (you can use the **KnowledgeRuntimeLogger** to see which rules have fired).

You will notice that the rule using the nested accessor – `addressWithCity_nestedAccessor` did not fire as you might expect. This is because after `city` has been set (through the `setCity` rule), JBoss BRMS was not notified that the `Customer` fact had been changed. The two lines above in the `setCity` rule are commented do just that. Uncomment them and fire the rules again; this time all rules will fire as expected.

2.6 FACT'S EQUALS AND HASHCODE METHODS

Every fact needs to implement the `equals` and `hashCode` methods. This is important in identifying if two facts are equal or not. Two facts are equal if and only if their equals methods return true for each other and if their hashCode methods return the same values. As a best practice, follow the rules laid out in **Effective Java** by Joshua Bloch. To make it easier, you can use the **EqualsBuilder** and **HashCodeBuilder** from the Apache Commons library. (Refer to its Javadoc on how to use them.) If you have a one-to-many relationship between two classes, be wary of cyclic calls; pick only one side of this relationship that will be responsible for propagating the `equals` and `hashCode` calls.

This practice applies also to "declared facts"—facts declared directly in a rules file. By default, declared facts don't override `equals` and `hashCode` methods. Use the `@key` annotation to mark a fact attribute as a key attribute. This attribute will then be used as a key identifier for the fact, and as such, the generated class will implement the `equals()` and `hashCode()` methods taking the attribute into account when comparing instances of this type (for an example see the declaration of the `Address` fact from the code snippet below).

Note that when a rule uses the equals operator '==', it uses the Java `equals()` method behind the scenes. If you want to compare two object references you need to place the comparison in an `eval` (for example `eval(customer == $customer)`).

2.7 CONSTRAINTS INDEXING

Red Hat JBoss BRMS uses indexing to improve performance. Indexing relies on correct `equals` and `hashCode` methods. When a fact's `hashCode` is changed*, JBoss BRMS must be notified.

Consider the following example (full rule file shown) – where `Customer` will be matched with `Address`. However to make it more interesting, let's modify the `Address`:

```
package brmsbestpractices;

dialect “mvel”;

declare Address
  city : String @key
end
declare Customer
  address : Address @key
end

rule initialize
    when

    then
        Address address = new Address(“London”);
        insert(address);
        insert(new Customer(address));
end
```

* As an example, consider a `Customer` fact that has `customerNumber` property that is part of the `hashCode` contract. If you change the `customerNumber` the customer's hash code will change as a result.

```
rule modifyAddress
salience 100
    when
        $address : Address( city != “Berlin” )
        //$customer : Customer( )  // <-- uncomment this later
    then
        modify($address) {
          city = “Berlin”
        }
        //update($customer);  // <-- uncomment this later
end
rule customerWithAddress
    when
        $address : Address( )
        Customer( address == $address )
    then
end
rule customerWithAddressWithEval
    when
        $address : Address( )
        Customer( eval(address == $address) )
    then
end
```

When you fire all rules, you see that the following rules will fire: `initialize, modifyAddress, customerWithAddressWithEval`.

As you can see, after the `Address` has been modified, only the `customerWithAddressWithEval` rule fires. This is is because `evals` are not indexed. If you uncomment the two lines above and fire all rules again, this is what you will see: `initialize, modifyAddress, customerWithAddress, customerWithAddressWithEval`.

Now all rules have fired as expected. After changing the `Address hashCode`, JBoss BRMS needed to be notified that the Customer has changed, because the `Address` field is part of the customer hash code contract (using the `@key` annotation). This does not seem very intuitive, as the `modifyAddress` rule should not care about what `Customer` facts reference the `Address`. This can be avoided by following these practices:

- If possible, only use immutable facts (see section 2.8 – **Favor immutability**).
- Do not modify the state of a fact that is part of its `hashCode` contract.
- If that is not possible, use primary keys when joining objects (i.e., `Customer(address.addressId == $address.addressId)`).
- Use object references. Remember that the "==" operator is translated to the `equals()` method, so if you want to compare object references, place the constraint inside an '`eval`' statement as shown in the `customerWithAddressWithEval` rule above.

2.8 FAVOR IMMUTABILITY

As in pure Java, immutable objects have a lot of value when used in rules. Instead of changing an existing object, it is better to create and insert a new special-purpose fact. In the following example you will apply a customer's discount to their order. Instead of modifying the `Order` fact, create a `CustomerPrice` fact:

```
rule applyCustomerDiscount
    when
        $customer : Customer( )
        $order : Order( )
        not CustomerPrice( )
    then
        insert( new CustomerPrice( $order.price *
          (1 - $customer.discount) ) );
end
```

Note that the rule above assumes that there will be only one `Customer` and one `Order` in the session.

In general, immutable objects help with looping prevention. They make it easier to explain why the rules reached a certain conclusion and they often help support future requirements because all knowledge is retained.

Sometimes, achieving true immutability requires a lot of effort, so the value it brings can start to diminish. To prevent this, consider adopting a technique where the properties of a fact that have been already set at some stage are not changed later on. In effect, this means that you can set properties that have not been set yet.

Consider the following example where the `valid` property of `Address` is not set when the `Address` is initialized, but is set later by the `validateAddressNoCity` rule.

```
rule validateAddressNoCity
    when
        $address : Address( city == null, valid != false )
    then
        modify($address) {
          setValid(false)
        }
end
```

2.9 TRUTH MAINTENANCE

At any point in time, the knowledge session should contain only facts that are true. If a fact is no longer true, it should be retracted from the session. This can be done manually, or more preferably, you can make use of the `insertLogical` construct. Logically inserted facts will be automatically retracted as soon as there are no rules and facts to back them up.

Consider the following example:

```
rule validateAddressMissingCity
    when
        $address : Address( city == null )
    then
        insertLogical( new ValidationReport("address.city.missing",
          $address) );
end
```

The `ValidationReport` is logically inserted. It will be automatically retracted if the rule's conditions are no longer true. For example if we modify the `Address` fact and set its `city` property, or if we retract the `Address` fact from the session.

3 BUSINESS RULES AUTHORING

This section will focus on best practices for writing rules.

3.1 USE CLEAR AND SIMPLE NAMES

When you look at the log of rules that have fired it should give you a high-level overview of what has happened during the execution. Give each rule a clear and simple name that expresses its intent. After applying the **"One Rule Once Concept"** principle, choosing a good name should be easy. For example if the requirement says, "validate that the customer address is entered," do not name the rule as "addressEntered." Consider calling it "noAddress" instead:

```
rule noAddress
    when
        Customer( address == null )
    then
        //..
end
```

You may be wondering if it is OK to name a rule after its outcome. Although not a common practice, there are cases where it would be appropriate.

3.2 HOW MANY RULES?

The number of rules a rule engine can handle is limited only by computer resources. In a nutshell, the algorithm behind the rule engine gives better results with more rules. This is because the conditions of rules are shared between rules and so will be evaluated only once.

However, this is not to say that the more rules you have, the better. In fact, too many rules make the engine harder to maintain. If you find yourself having to write thousands of rules that look more or less the same, you may need to change your approach (see section 3.5 – **Avoid duplicating rules**).

Group rules logically into files. If they change together, they should stay together. Don't be afraid to split rules into multiple files. The `KnowledgeBase` can be built from many files. As a general rule of thumb, try to have a maximum of 50 rules per one rule file.

The same best practices that apply to Java packages apply to `KnowledgePackages`. These packages are ways of organizing rules and other resources into namespaces. As such, the package name should usually correspond to some area of the business logic (validation, pricing, matching, etc).

3.3 ONE RULE ONE CONCEPT

The **single responsibility principle** in object–oriented programming states that every object should have a single responsibility. The same concept applies to rules; you should aim for smaller rules that are easy to understand and modify.

Consider this example, where you must validate that an order has both the customer and a list of products filled.

Instead of writing one rule that does this check, the solution is to write two: one for `Customer` and one for list of `Products`:

```
rule customerMissing
    when
        Order( customer == null )
    then
        insertLogical(new ValidationResult(
          “validation.customer.missing”));
end
rule productsMissing
    when
        Order( products == null || products.isEmpty() )
    then
        insertLogical(new ValidationResult(
          “validation.products.missing”));
end
```

3.4 USE 'INFERRED FACTS'

If the requirement is complex, try to apply the divide and conquer principle. Divide the requirement into smaller pieces by inserting "inferred facts," and then write rules that will draw conclusions from these inferred facts.

It is important to note that inferred facts encapsulate knowledge so that it can be reused later. They make it easier to refer to this knowledge by using the fact name.

In the following example a customer number is required, or at least the name and address are needed. If this information is not present, the request will be rejected or forwarded to a customer service representative, depending on the product contract.

To accomplish this, first write rules that insert an "inferred" `RequestIsValid` fact into the knowledge session. Then, a second set of rules will act as if there is no `RequestIsValid` fact:

```
rule hasCustomerNumber
  ruleflow-group "validation"
    when
        Customer( customerNumber != null )
    then
        insertLogical(new RequestIsValid());
end
rule hasCustomerNameAndAddress
  ruleflow-group "validation"
    when
        Customer( name != null )
        exists Address( )
    then
        insertLogical(new RequestIsValid());
end
rule autoReject
  ruleflow-group "post-validation"
    when
        Product( autoReject == true )
        not RequestIsValid()
    then
        insertLogical(new RequestDenied());
end
```

```
rule autoRejectFalse
  ruleflow-group “post-validation”
    when
        Product( autoReject == false )
        not RequestIsValid()
        $request : Request()
    then
        customerRelationsService.handleBadRequest($request);
end
```

Inferred facts are also useful when working with rule flows. Do not put complex logic in a “split” node. Instead, write a rule that will insert a fact (such as RequestIsValid). Then the split node will simply check if that fact exists. It is easier to keep track of logic within rules than within rule flow nodes.

Note that Red Hat JBoss BRMS supports queries inside rule conditions. This means that writing queries can achieve the same results as using inferred facts. This is especially useful when you don’t want to insert many inferred facts. For example:

```
query isValidRequest()
    Customer( customerNumber != null )
    or
    (
      Customer( name != null ) and
      exists Address( )
    )
end
```

This query can then be used instead of the `RequestIsValid` fact. Note that the two validation rules were merged into one query (because there can be only one i`sValidRequest` query definition).

3.5 AVOID DUPLICATING RULES

Duplication is considered an evil in any source code, and the same applies for rules. Here are some ways to fight duplication:

In a rule's conditions:

- Use inferred facts (see **Use Inferred Facts**).
- Using rule inheritance, one rule can extend all of its conditions to another rule (for examples visit **https://github.com/droolsjbpm/drools/blob/5.2.0.Final/drools-compiler/src/test/resources/org/drools/integrationtests/extend_rule_test.drl)**
- Consider writing your own custom operators (**http://blog.athico.com/2010/06/creating-pluggable-oprators.htmls**) and/or accumulate functions (**http://blog.athico.com/2009/06/how-to-implement-accumulate-functions.html**) although this helps with readability more than it solves the duplication problem

In a rule's consequence:

- Use functions, utility classes, or listeners (see section 5.4 – **Make use of listeners**)
- Rule templates or decision tables: If you find yourself writing many rules with minor differences, use rule templates or decision tables. (Rule templates are a more powerful version of decision tables, allowing you more customization of both a rule's condition and consequence)

3.6 AVOID RULE ORDERING*

* Note: this section does not talk about processing a set of facts in a certain order.

Rule ordering is considered bad practice in declarative programming; rules should fire opportunistically whenever they are applicable. However, sometimes rule ordering cannot be avoided, especially when many things must be done in a certain sequence. For example, imagine that you have some scoring rules that assign a score to each potential match, and you want to pick the match with the highest score only after all scoring rules have fired. In this case, you need to define a order between rules, and Red Hat JBoss BRMS offers various constructs to help:

- `ruleflow-group`: This is the preferred option for ordering rules. You can group multiple rules into groups and then use a rule flow (i.e. a process) to define the execution order between these groups. Every non-trivial knowledge base should have a corresponding rule flow process definition.
- Use inferred facts (also called "control facts") to impose order: Imagine a situation where you receive an order for some products. First you need to validate it, then find the customer that ordered the products (let's say that `customerNumber` on the order is not always set) and finally you must calculate the price. We could create three control facts – `ValidationPhase`, `MatchingPhase`, `PricingPhase` – and insert them into the session as each phase becomes active. Each rule will ensure that its phase is active (for example `exists ValidationPhase()` in a rule condition). Some "control rules" will be needed to manage the execution of phases by insertion/retraction of the control facts. These control rules will most likely need to use "`salience`" so that they fire only after all rules within a phase. The advantage of this approach is that it allows for more flexibility in choosing the next phase than the ruleflow-group approach. The obvious disadvantage is that visibility into the flow is lost.

- `salience`: This is probably the easiest option, but it should be used with great care because it introduces hidden dependencies between rules. Always use `salience` in combination with ruleflow-group. Using `salience` to define order within a group of rules is more acceptable, and you should not need more than a dozen `salience` values. To follow best practices, be sure to document why the salience is needed using comments in the rule.

3.7 MINIMIZE DEPENDENCIES BETWEEN RULES

It is inevitable that some rules will depend on other rules. These dependencies include the ordering of rules (see section 3.6 – **Avoid rule ordering**) and dependencies on fact insertion/modification.

We need to differentiate between bad and acceptable dependencies. Ideally, if we remove a rule, the system should fail gracefully. To do this, the conditions of each rule should be defensive. The following example illustrates bad dependencies:

```
rule initializeOrder
    when
        $order : Order( items == null )
    then
        modify($order) {
          // set items
        }
end
rule calculatePrice_DontDoThis
salience -100   // bad dependency on initializeOrder rule
    when
        $order : Order( price == null )
    then
        modify($order) {
          setPrice( calculatePrice($order.items) );
        }
end
```

The `calculatePrice_DontDoThis` rule depends on the `initializeOrder` rule via its use of `salience`. It is depending on the $order.items property to be set. This is not good because if we change the `initializeOrder` rule (for example we move it to a ruleflow group that is executed later) the `calculatePrice_DontDoThis` rule might break (it will potentially calculate an incorrect price). It would be better if we remove this dependency:

```
rule calaulatePrice
    when
        $order : Order( items != null, price == null )
    then
        modify($order) {
          setPrice( calculatePrice($order.items) );
        }
end
```

As you can see, the `calculatePrice` rule is more independent now. It will calculate price only if the items are set on the order.

3.8 NO CONDITIONAL LOGIC AND RULE CONSEQUENCES

Because the `then` part of a rule can contain any valid Java code, it is sometimes tempting to add some conditional logic. Use caution here: It does not make sense to use "`if`" or "`switch`" statements in a rule's consequence to implement some business logic. That would defeat the purpose of a rule engine. Instead, split this rule into multiple rules (See the see section 3.3 – **One rule one concept** and section 3.4 – **Use 'inferred facts'**).

Here is an example that uses an "`if`" statement in the "`then`" part. Do not emulate this in your code:

```
rule assignDiscount_DontDoThis
    when
        $customer : Customer( age > 50 )
    then
        if ($customer.isVIP) {
          modify($customer) {setDiscount(BigDecimal.valueOf("0.15"))}
        } else {
          modify($customer) {setDiscount(BigDecimal.valueOf("0.1"))}
        }
end
```

Instead, refactor this rule into two rules:

```
rule assignDiscountVIPCustomer
    when
        $customer : Customer( age > 50, isVIP == true )
    then
        modify($customer) {
          setDiscount(BigDecimal.valueOf(“0.15”))
        }
end
rule assignDiscountNonVIPCustomer
    when
        $customer : Customer( age > 50, isVIP == false )
    then
        modify($customer) {
          setDiscount(BigDecimal.valueOf(“0.1”))
        }
end
```

That being said, it is acceptable to use conditional logic when implementing some non-business logic. In this case, it is best to extract this logic and put it into a function or some utility class.

You may be wondering if it is OK to use looping structures like “`for`” and “`while`.” These maybe needed in some cases, but it would always be best to avoid them.

In general, try to minimize the amount of Java code within a rule’s consequence.

3.9 IS IT A GLOBAL OR A FACT?

When you want to make some data available to the rule engine, you can insert it into the session or make it accessible through a global (for example, write a rule that pulls some data from a service and inserts it as facts into the knowledge session).

As a best practice, do not use globals inside rule conditions; use them only in rule consequences. The following example uses an `orderService` to create a new order:

```
global OrderService orderService;

rule addOrder

    when

        //...

    then

        orderService.createOrder(new Order( .. ));

end
```

It is also acceptable to use globals to get data out of the knowledge session. For example, create a `ResultHolder` class that will contain data that you want to pass out of the session:

```
global ResultHolder resultHolder;

rule extractResult

    when

        // …

    then

        resultHolder.result = ..

end
```

You can then simply set the global, fire all rules, and the result should be set:

```
ResultHolder holder = new ResultHolder();

session.setGlobal("resultHolder", holder);

session.fireAllRules();

//at this point the holder.result should be set
```

3.10 AVOID USING THE FROM KEYWORD

Red Hat JBoss BRMS allows you to reason over facts that are not in the knowledge session by using the "`from`" keyword. This should be avoided if possible. In some instances, it is used when inserting all data into the knowledge session, but it is not practical. Since these "facts" are not physically inserted in the knowledge session, the rule engine cannot apply the usual optimizations. Instead, it has to process all facts sequentially each time the rule is evaluated. Instead of using "`from`," consider inserting facts directly into the session.

3.11 AVOID OVERUSE OF THE EVAL KEYWORD

Use eval in a rule's conditions only when necessary. The rule engine has limited options to optimize them (for example, they cannot be indexed). The same applies to method calls inside rule conditions. Think about splitting `evals` into elemental operations on properties.

Watch out for `eval` statements that are not part of a pattern's conditional element that don't need to be; these are bad for readability and performance. For example:

```
rule customerIsChild_DontDoThis
    when
        $customer : Customer( )
        eval( $customer.age < 10 )
    then
        //..
end
```

Instead, use this:

```
rule customerIsChild
    when
        $customer : Customer( age < 10 )
    then
        //..
end
```

3.12 DOCUMENTATION

Aim for writing rules that are easy to understand and do not need much documentation. After all, rules are written declaratively, and well-written rules will explain their meaning upon first reading. However, there are still cases when a rule is more complex and a brief explanation helps to clarify it. To accomplish this, it is usually enough to add a few notes to a complex rule's condition.

Another useful item to document is the intended amount of facts within the knowledge session. Specifically, document how many occurrences of a given fact you are expecting, one or many. This simplifies rules by eliminating unnecessary joins. For instance, if you know that there is only one `Customer` instance in a session, you don't need to match an `Order` with a `Customer` because all `Orders` belong to this single `Customer`.

4 BUSINESS RULES DEPLOYMENT

Red Hat JBoss BRMS supports many deployment options. At the beginning of a project, especially where quick turnaround is important, you can build the `KnowledgeBase` from source files. This is also very useful for debugging because you can change a rule, restart the application, and verify the results.

If you want to release your application later, but don't want to give away rule source files, or if you have lots of integration tests and you don't want to re-compile the `KnowledgeBase` for each one, you can consider pre-compiling the KnowledgeBase. With JBoss BRMS, you can pre-compile individual KnowledgePackages or the whole KnowledgeBase. The `KnowledgePackages` can then be used to create the KnowledgeBase. In addition, this option might be useful if you have few `KnowledgePackages`, but you are combining them and creating lot of `KnowledgeBases`.

JBoss BRMS allows users to pre-compile the KnowledgeBase in a variety of ways. You can use Business Rules Manager (Guvnor), do it as part of your build with an Ant task, or you can do it yourself programmatically.

`KnowledgeBase` is an expensive object to create, so pre-compiling the `KnowledgeBase` and/or `KnowledgePackages` is one way to reduce this cost. Reusing the `KnowledgeBase` is also encouraged because it is thread-safe and these objects can be shared without issues.

4.1 KNOWLEDGE BASE PARTITIONING

A knowledge base usually will contain assets such as rules, processes, and domain models that are related to one subject, business entity, or unit of work. Understanding how to partition these assets in knowledge bases can have a huge impact on the overall solution. BRMS tools are better at optimizing sets of rules than they are at optimizing individual rules. The larger the rule set, the better the results will be when compared to the same set of rules split among multiple rule sets. On the other hand, increasing the rule set by including non-related rules has the opposite effect, as the engine will be unable to optimize unrelated rules. The application must still pay for the overhead of the additional logic. The best practice is to partition knowledge bases by deploying related rules into a single knowledge base. Avoid monolithic knowledge bases, as well as too fine-grained knowledge bases.

5 BUSINESS RULES EXECUTION

Here we'll look at various best practices relating to runtime execution.

5.1 KNOWLEDGE SESSIONS ARE CHEAP

`KnowledgeSessions` are very lightweight. They are intended for use by a single thread only (note that there are exceptions when working with entry-points). Do not be afraid to create `KnowledgeSessions`. It is better to split a problem into chunks and execute them in different sessions. For example, let's say we need to validate several orders. In this case, it is better if we validate each order independently. Note that this assumes there are not a lot of rules between orders. If there are, it might be better to validate all orders within one single session.

When considering this second alternative, one single session, it is important to realize that the greater the number of objects of the same type in the session, the more matches Red Hat JBoss BRMS has to try. Consider the following rule that looks for `Order` with the highest price. If there are 10 `Orders` in the session, JBoss BRMS needs to try all possible combinations, which is in this case 100 combinations (10 x 10 = 100).

```
ule orderWithMaxPrice
    when
        $maxOrder : Order( )
        not Order( price > $maxOrder.price )
    then
        //order with maximum price is $maxOrder
end
```

If possible, it is better use multiple smaller sessions than one session with lots of facts. Note that this rule will fire once for each order with the highest price.

Also note that although this may not be the most efficient way for finding the maximum, if the number of facts is low, this method will suffice. If the number of facts is higher, consider more advanced techniques such as those described here: **http://community.jboss.org/wiki/RulesFindMax**.

5.2 LOOPING PREVENTION

One of the difficulties that every rule writer faces at some stage is stopping a rule from executing over and over again. Red Hat JBoss BRMS provides the following options to prevent looping:

- Write the rule in a way that stops looping; for example, if a rule calculates some value, then add a check in the condition to ensure that this value is not yet set.
- If multiple rules trigger each other, consider moving one of the rules into a different ruleflow group so that these rules won't affect each other. For example, if you need to initialize a fact with data coming from many sources, this is best achieved in a "setup" ruleflow group that won't affect subsequent ruleflow groups that need to work with that particular fact.
- If the above is not possible, then try to **use inferred facts** (that is, split the rule into multiple rules and introduce new facts). For an example, see section 2.8 – **Favor immutability**.

- Use `no-loop`: This rule attribute has a very limited use because it only prevents a rule from re-activating itself. However, other rules that are modifying a fact that our rule depends on can re-activate it. As you can see, use of this attribute also creates a hidden dependency on other rules.
- Use `lock-on-active`: This takes a ruleflow/agenda group name as a parameter. It prevents rule activating when the ruleflow group is active. This is sufficient for most scenarios, but only use as a last resort when the practices described above have failed.

No method or strategy can guarantee that rules will not loop. However, there are options that can at least detect this behavior. As the simplest solution, create a `LoopDetectionEventListener` that has a single counter that counts the number of all rules fired and throws an exception once the count reaches a certain threshold*.

* For more sophisticated loop detection see **http://blog.athico.com/2009/08/loop-detection-ideas.html**

5.3 IS THE ORDER OF CONDITIONS IMPORTANT?

You may have wondered if putting the `Customer` fact before the `Address` fact in a rule's condition is important. In general, you do not have to worry about the order of a rule's conditions. It is the rule engine's job to execute your rules as efficiently as possible. It is your job to write rules that are easy to maintain.

If you look under the covers of Red Hat JBoss BRMS you will see that the order of conditions is important and it affects the performance of the rule engine. You may try to optimize for performance, but the results may vary across each JBoss BRMS release. This is because the rule engine might change its approach, or it might implement various optimization techniques that could make your optimizations ineffective. However, if you still want to optimize your rules, here are some quick tips:

- Put the common constraints first: If you have two identical rules with just one condition, such as Customer(`firstName="Alex", lastName="Brown"`), they will share the core of the Rete network (see **http://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/ch03.html#d0e1092**). However, if you reverse the order of the constraints in one of the rules, `Customer( lastName="Brown", firstName="Alex"`), no nodes will be shared (meaning the rule engine has to do more work).
- Put the common conditional elements first: If two rules differ only slightly in their conditions, put the distinguishing conditions to the end. These two rules will share the nodes in the Rete network (up to the different conditions). For example these two rules will share the same node in the Rete network up to the conditional element on `Customer`.

```
rule customerOrder_Adam
    when
        $order : Order( )
        Customer( name == “Adam” )
    then
end
```

```
rule customerOrder_John
    when
        $order : Order( )
        Customer( name == “John” )
    then
end
```

- If a fact changes often, put the condition that uses this fact at the end. For instance consider a rule with the following conditions:

```
$order : Order( )
$item : Product( ) from $order.items
$customer : Customer( )
```

  If the `$customer` fact is modified, the earlier conditional elements do not need to be reevaluated. This can save a considerable amount of computing resources if the `$order.items` collection contains many elements.

- Place conditions that are going to filter the result set most at the beginning: This is the same concept that is used when writing an SQL query. For example, imagine that a customer can have many orders and each order can have many items. There are many customers, orders, and items in the knowledge session. Let’s write a rule where a given `customerId` will find all of the customer’s ordered items:

```
rule customersProduct_Bad
    when
        $order : Order( )
        $product : Product( this memberOf $order.items )
        $customer : Customer( customerId = 1,this == $order.customer)
    then
end
```

  The problem with this rule is that each order will be matched with its list of items first. This includes the orders of all customers, so the rule engine has to do a lot of unnecessary matching. Let’s put the customer condition first since it will filter the result set the most:

```
rule customersProduct_Good
    when
        $customer : Customer( customerId = 1 )
        $order : Order( customer == $customer )
        $product : Product( this memberOf $order.items )
    then
end
```

- If you need to use a standalone `eval` conditional element, consider putting it at the end of the conditions list. Ordering your conditions in this way will minimize the number of times the `eval` conditional element will be evaluated.

Please note that some pieces of advice given here might conflict with others. For example, if a fact is modified often but it also filters the result set the most, it is not clear whether to make it the first or the last condition. As with any general performance advice, effectiveness will vary from system to system. Always test on your target system first to make sure you achieve desired result.

5.4 MAKE USE OF LISTENERS

Red Hat JBoss BRMS allows you to set up listeners for many events, including `WorkingMemoryEventListener`, `AgendaEventListener`, `KnowledgeBaseEventListener`, and `KnowledgeAgentEventListener`. It also allows you to set up a listener for query changes, such as ViewChangedEventListener. When implementing logic that is duplicated in many rules, always think about using listeners. They can be easily added or removed, and they will help you keep the rules simple and focused. Listeners are especially well suited to some use cases, like logging or auditing activities.

5.5 ALWAYS TEST YOUR RULES

Write a set of tests for each rule file and then test each rule to determine if it works and whether rules work together as expected. Testing is straightforward:

1. Prepare the inputs: Insert data into the session and set up mock objects for globals as needed.
2. Fire all rules.
3. Verify the outcomes: What are the contents of the session after all of the rules have fired? The `AgendaEventListener` can be used to track fired rules so you can verify expected results.

EXAMPLE ON RULE UNIT TESTING USING JUNIT 4

Unit testing rules gives you great confidence that the rule under test behaves as expected. It helps to prevent bugs early on in development and is invaluable during Red Hat JBoss BRMS version upgrades. With unit testing, you will know immediately if rules are behaving the same after the upgrade.

Here is a test for the `validateAddressMissingCity` rule from the **Truth maintenance** section:

```
rule validateAddressMissingCity
    when
        $address : Address( city == null )
    then
        insertLogical( new ValidationReport("address.city.missing",
          $address) );
end
```

First, create a test class with the following properties:

```
public class ValidationTest {
    private static KnowledgeBase knowledgeBase;
    private StatefulKnowledgeSession session;
    private TrackingAgendaEventListener trackingListener;
    private KnowledgeRuntimeLogger logger;
```

The `knowledgeBase` will be used to create knowledge sessions. The `trackingListener` will keep track of rules that have fired so you can verify expectations, and the logger can be used if things don't go as planned and you need to debug.

Notice that the `knowledgeBase` is a `static` property. It is an expensive object and so it should be shared between test methods (consider sharing across test cases as well).

Next we'll initialize the `knowledgeBase`:

```
@BeforeClass
public static void setUpClass() throws Exception {
    knowledgeBase = loadKnowledgeBase(Arrays.asList(
      "validation.drl"));
}
```

The `loadKnowledgeBase` method is not shown, but it simply takes a list of resources and creates the `KnowledgeBase`.

Next comes the test `setUp` method that will be executed before each test method. Go ahead and initialize the remaining properties:

```
@Before
public void setUp() {
    trackingListener = new TrackingAgendaEventListener();
    session = knowledgeBase.newStatefulKnowledgeSession();
    session.addEventListener(trackingListener);
    // logger = KnowledgeRuntimeLoggerFactory.newFileLogger(session,
    // “audit”);
}
```

Notice that the logger is commented out for now. You can un-comment it when you need to do some debugging. In that case, an `audit.log` file will be created that can be opened using the JBoss Developer Studio Drools Audit log plug-in.

After each test finishes, do some cleanup:

```
@After
public void tearDown() {
    if (logger != null) {
      logger.close();
    }
    if (session != null) {
      session.dispose();
    }
}
```

Now look at the `TrackingAgendaEventListener` in more detail. It will simply capture all rules that have fired:

```
public class TrackingAgendaEventListener extends
  DefaultAgendaEventListener {
  private List<String> rulesFired = new ArrayList<String>();
```

```
    @Override
    public void beforeActivationFired(BeforeActivationFiredEvent e){
      rulesFired.add(e.getActivation().getRule().getName());
    }

    public boolean isRuleFired(String ruleName) {
      return rulesFired.contains(ruleName);
    }
  }
```

The `isRuleFired` method will be used to verify whether a rule has fired. Other useful methods can be added, including "`reset`," which clears the `rulesFired` collection or `getRuleFiredCount`, which returns the number of times a given rule has fired.

Now you can write the test methods. The first test verifies that, if there is no `city`, a new `ValidationReport` will be inserted into the session:

```
  @Test
  public void testMissingCity() throws Exception {
    // insert Address with no city (null)
    Address address = new Address(null);
    session.insert(address);

  // execute using agenda filter
    session.fireAllRules(new RuleNameEqualsAgendaFilter(
      "validateAddressMissingCity"));

  // verify that the rule has fired
    Assert.assertTrue(trackingListener.isRuleFired(
      "validateAddressMissingCity"));
```

```
    // verify that there is report in the session
    Collection<Object> objects = session.getObjects(
      new ClassObjectFilter(ValidationReport.class));
    Assert.assertEquals(1, objects.size());
    ValidationReport report = (ValidationReport)
      objects.iterator().next();

    //verify that the report has all properties set
    Assert.assertEquals("address.city.missing", report.getKey());
    Assert.assertSame(address, report.getObject());
  }
```

Notice that `AgendaFilter` is being used to filter out unwanted activations. In this case, only the `validateAddressMissingCity` rule is allowed to fire*.

The second test verifies that there will be no report if the `Address` has the `city` property set:

```
  @Test
  public void testHasCity() throws Exception {
    // insert Address with city set
    Address address = new Address("Sydney");
    session.insert(address);

    // execute using agenda filter
    session.fireAllRules(new RuleNameEqualsAgendaFilter(
      "validateAddressMissingCity"));
```

* Note that `AgendaFilters` don't work well with rules that are part of some ruleflow group. In this case we have to test all rules with the ruleflow together, or if we really want to test each rule in isolation we could for example pre-process the source file first and remove the ruleflow group (just for the test).

```
// verify that the rule did not fire
Assert.assertFalse(trackingListener.isRuleFired(
  “validateAddressMissingCity”));

// verify that there is no report
Collection<Object> objects = session.getObjects(
  new ClassObjectFilter(ValidationReport.class));
Assert.assertTrue(objects.isEmpty());
```

For more tips and tricks, see
http://blog.athico.com/2011/10/cookbook-how-to-test-rules-using-xunit.html.

ABOUT RED HAT

Red Hat is the world’s leading provider of open source solutions, using a community-powered approach to provide reliable and high-performing cloud, virtualization, storage, Linux, and middleware technologies. Red Hat also offers award-winning support, training, and consulting services. Red Hat is an S&P company with more than 70 offices spanning the globe, empowering its customers’ businesses.

NORTH AMERICA
1-888-REDHAT1

EUROPE, MIDDLE EAST AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
latammktg@redhat.com

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#10794747_v2_0413