The Rogue BOM

When is a BOM a time bomb? Pitfalls and anti-patterns frequently seen while creating a JRules Business Object Model (BOM) and how to avoid them

Summary

The Business Object Model (BOM) is a fundamental component that provides the business vocabulary used by business policy managers to define rules built using WebSphere ILOG JRules, which is IBM's Business Rule Management System (BRMS). This article discusses the frequently observed anti-patterns and missteps in the creation of a BOM and describes techniques for avoiding them.

Prerequisites

This article is written for the intermediate JRules developer. It is assumed that the reader has a basic understanding of the ILOG JRules product from a developer's perspective. Please review the Resources section for relevant links to acquire the prerequisite knowledge for completing this article. The product versions used in this article are:

- WebSphere ILOG Rule Studio V7.1.x

Introduction

Essentially, the Business Object Model (BOM) is the representation of the domain model in scope for efficient rule processing. It is this representation of the model that rules are written against. BOM elements are the classes, attributes and methods grouped into packages. Natural-language verbalization is attached to the BOM elements to make rule authoring user friendly. Verbalizations on all the BOM elements form the *vocabulary* that business policy managers use to define rules using JRules.

The BOM is not the model used during runtime – that would be the Execution Object Model (XOM). Loosely speaking, the XOM is related to the BOM as a database table is related to a database view or an implementation model is related to the logical model. The XOM is defined either as Java classes or XML schema definitions. At runtime, it is the methods on instances of the XOM that are executed as rule conditions and actions. The BOM-to-XOM mapping (B2X) defined in the BOM is used to translate the BOM into the corresponding XOM element.

Since the BOM is the foundation on which rules are written, creating a good BOM is a crucial part of the rule development process. On the face of it, creating a BOM seems like a straight-forward process, especially since a XOM can be directly used in a Rule Studio wizard to generate the BOM. However, experience has shown that there are several pitfalls in creating a BOM that can come in the way of rule projects delivering on their promise of easy maintenance leading to expensive refactoring or even

failure down the line. This article examines these missteps and pitfalls and suggests ways to avoid them.

Pitfall: neglecting "business" in the BOM

As should be evident from its name, the essence of a BOM is that it is the business user's view of their domain. However, this basic tenet is sometimes overlooked leading to detrimental effects. This usually happens when the decision service uses an enterprise model or industry standard information model as the XOM. These models tend to be complex and large in their breadth and depth as they need to cater to all the data requirements across all the enterprise applications. Enterprise models deal with classes that do not necessarily correspond with the concepts used by policy managers and subject matter experts (SMEs). The class names and attribute names tend to be IT-centric and are usually not business friendly. The least cost path of then directly using this XOM in a Rule Studio wizard to generate the BOM leads to a BOM that is unsuitable for writing rules. Some of this can be mitigated through appropriate BOM verbalizations. However, more troublesome is when rule authors need to understand and navigate through a complex class hierarchy, one that is very different from their conceptual model. This discrepancy leads to *cognitive dissonance,* which is a feeling of discomfort and confusion that results from simultaneously holding two models at variance.

As an example, consider an insurance company that has decided to use ACORD[1] as its standard. This in itself is a fine enterprise level architectural decision as it promotes interoperability among applications within the company and potentially even across partners in the industry. Data will be passed between applications using ACORD data types. The insurance company uses a JRules rule application to provide online insurance policy quotes for new customers for which the ACORD format is used in the service contract design. Therefore, this decision service receives data in ACORD format.

Figure 1 and Figure 2 show a slightly modified view of a portion of the ACORD model. This input to the decision service is an ACORD object. As depicted in Figure 1, navigation to the policy quote request itself requires us to jump across two links on this class diagram starting from the root ACORD request.

[1] See resources for a link to additional information on ACORD.

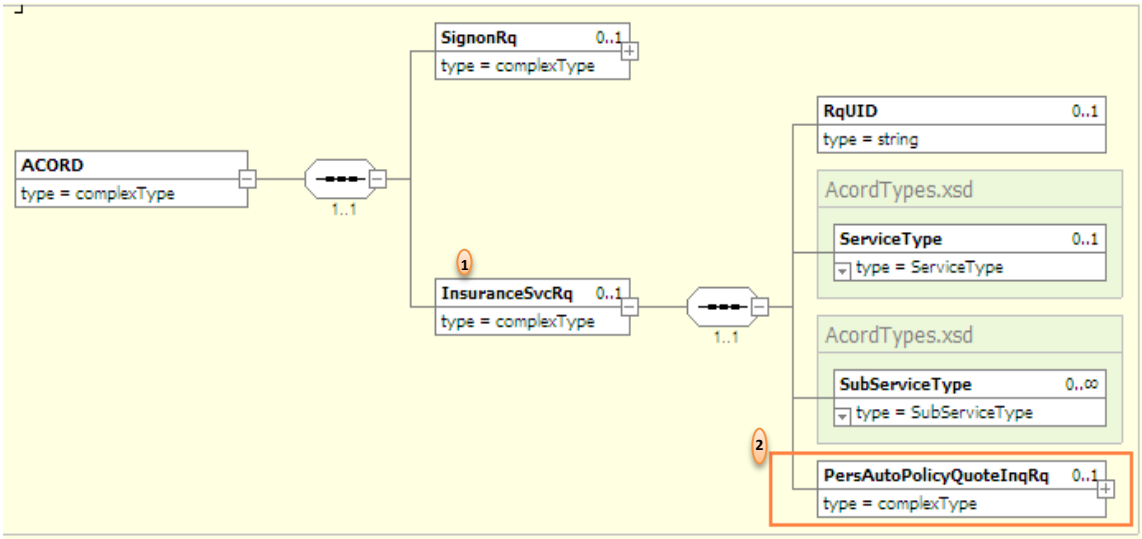

Figure 1: Navigating to the Policy Quote request

From the policy quote, we need to jump through four more links to navigate to the birth date of the driver in a rule that references the age of the driver. This is shown in Figure 2.

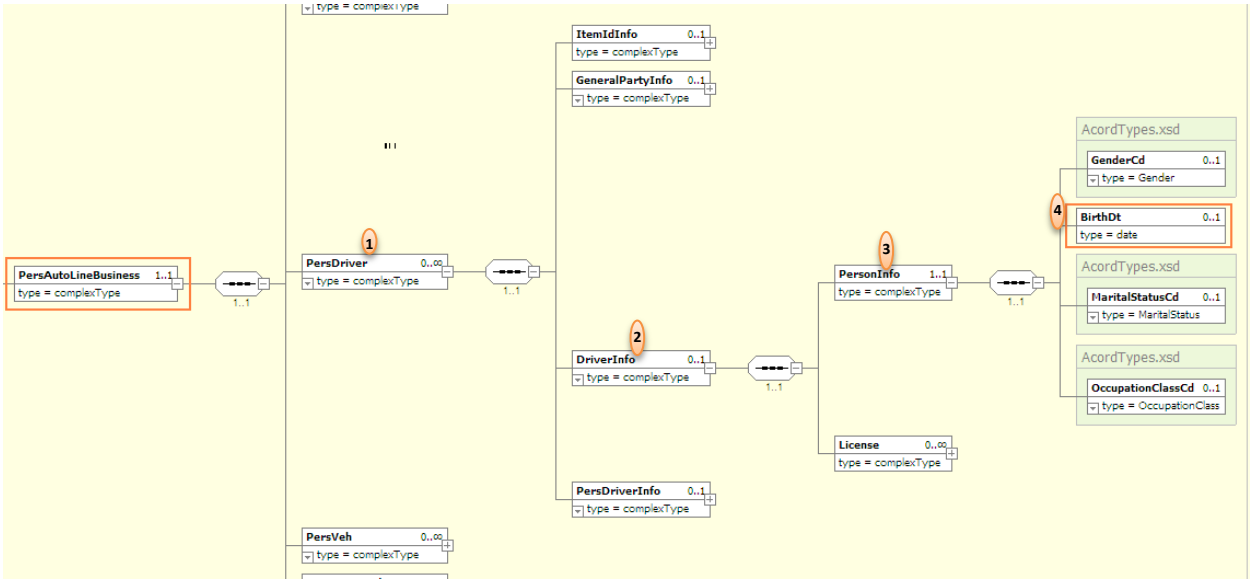

Figure 2: Navigating to the birth date of the driver

In other words, a simple conceptual attribute, such as 'age of the driver' requires the rule author to navigate through six classes!

Consider a simple business policy:

> *Refuse policy if any drivers are under 21 years of age in CA and NV.*

A business rule implementation of this using the above BOM is listed below.

```
definitions
    set 'State' to the controlling state prov cd of the pers policy of the
pers auto policy quote inq rq of the insurance svc rq of 'request Acord';
    set 'persAutoLineBusiness' to a pers auto line business from the pers
auto line business of the pers auto policy quote inq rq of the insurance svc
rq of 'request Acord';
    set 'persDriver' to a pers driver in the pers drivers of
persAutoLineBusiness;
    set 'Driver Info' to a driver info from the driver info of persDriver;
    set 'Personal Info' to a person info from the person info of 'Driver
Info';
    set 'CurrentAge' to the calculated age using the birth dt of 'Personal
Info';
if
    all of the following conditions are true :
            - State is one of { CA, NV }
            - CurrentAge is less than 21
then
…
```

Clearly, the above rule is unnecessarily complex, hard to read and hard to maintain. Moreover, as it does not match with the policy manager's conceptual model, it feels obscure and burdensome to maintain for the business user. The conceptual model in the policy manager's mind is radically simpler than the enterprise model. For example, they may conceive the policy quote request (or a portion of it) as a simple model depicted in Figure 3.

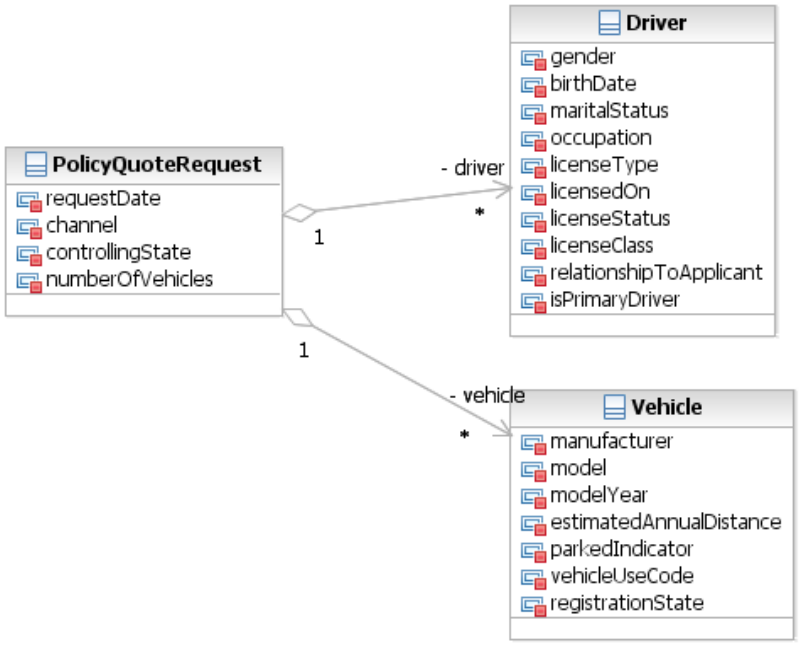

Figure 3: Conceptual policy model

The BOM should mimic this conceptual model. Now with the BOM aligned with the conceptual model, the same rule can be expressed in a much simpler and straightforward manner, as illustrated in this listing below, thereby eliminating the cognitive dissonance.

```
definitions
    set 'driver' to a driver in the drivers of 'policy request';
    set 'currentAge' to the calculated age using the birth date of 'driver';
if
    all of the following conditions are true :
            - the controlling state of 'policy request' is one of { CA, NV }
            - currentAge is less than 21
then
…
```

While this example highlights the problems inherent in directly using the ACORD structure as the BOM, this applies to most other industry standards as well such as MISMO model for mortgage loans and Primavera model for insurance.

As is evident, the BOM should present a very specific viewpoint of the domain that is most suitable for the policy manager to conceptualize and author business rules. Anything that upsets this viewpoint, be it new concepts, new terminology or discrepancy in relationships, introduces cognitive dissonance. Another situation where this may occur is when the modeler decides to model the “real world” instead of just the specific business viewpoint. With this intent, the modeler zestfully introduces many new concepts and associations that at best are superfluous to the business viewpoint and at worst actually

discordant with it. This “real world” BOM may start to look like an enterprise model, with all its negative consequences.

Traditional object oriented analysis (OOA) makes the distinction between analysis models and design models. The chasm between the BOM and the decision service model runs even deeper than this distinction in the aforementioned examples since the BOM should represent the *rule analysis* model while the external model should represent the object oriented design model.

Techniques for avoiding this pitfall

As illustrated in the above example, the BOM should match the conceptual model of a policy manager. However, input data into the decision service is the enterprise information model. So how do we reconcile these two seemingly conflicting requirements? There are two ways:

1) Use a *Transformer* in the decision service
2) Build a BOM adapter layer using B2X

Both these approaches essentially utilize a data adapter layer to build a façade.

Transformer

In this approach, we build a XOM that is aligned with the simple conceptual model, essentially ignoring the enterprise model until later. Therefore, the BOM derived from this XOM is also simple and easy to use for rule authoring. However, the service contract is based on the enterprise model. During rule invocation, a Transformer is used in the decision service to convert the request sent by a rule client from the complex enterprise model into the simple XOM. The Transformer may just be some simple Java classes or may employ XSLT. Alternatively, if using service oriented architecture (SOA), it may be a transformation service provided by service oriented integration (SOI) middleware or an ESB. These transformed XOM objects are then sent to the rule engines for rule execution using the Rule Session API. The rule response also is transformed from the XOM objects to the enterprise objects before returning to the rule client. This is depicted in Figure 4.

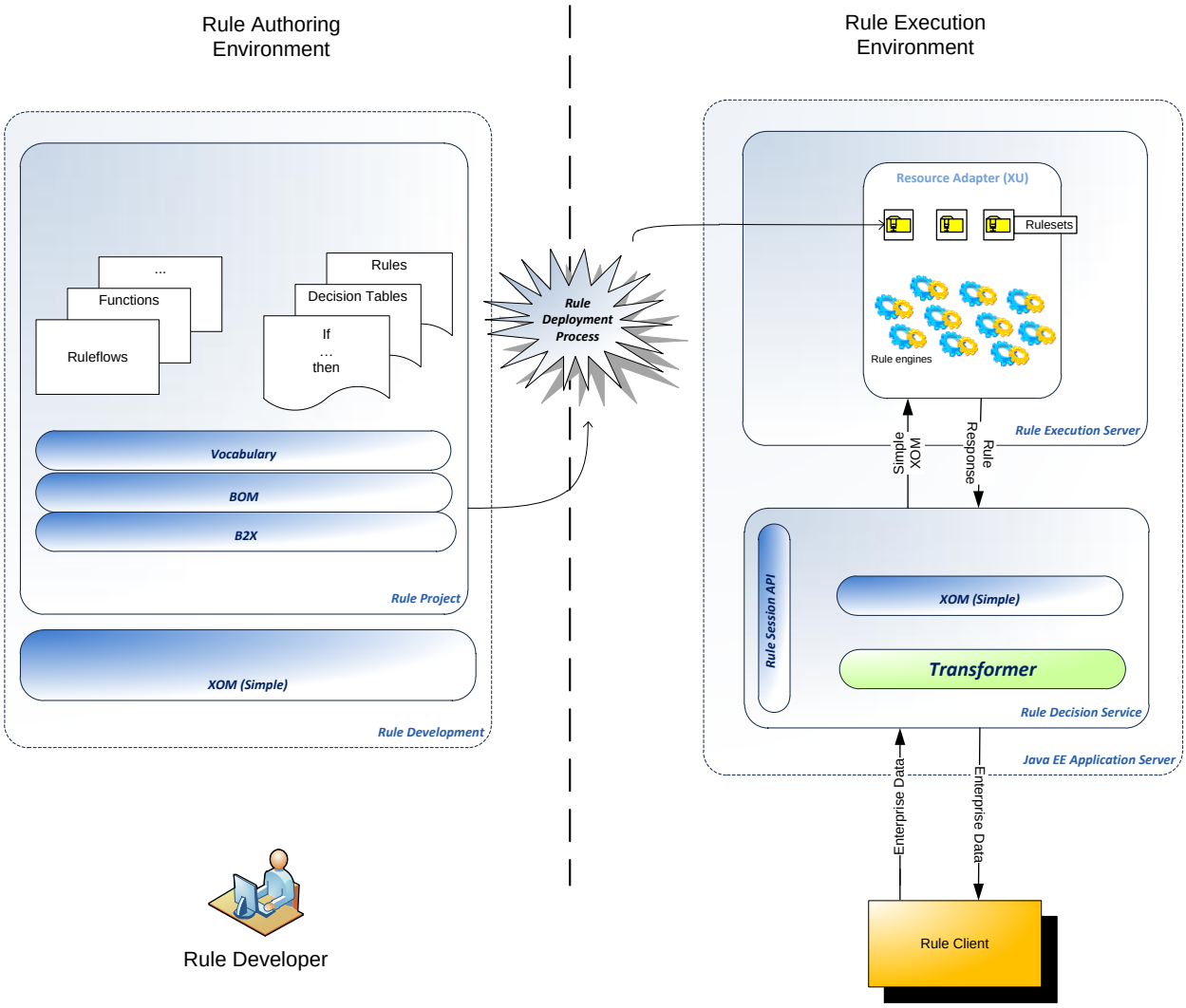

Figure 4: Using a transformer to convert the enterprise model into a simple XOM

This approach leads to a loosely coupled system and has the advantage that the rule developer can practically ignore the complexities of the enterprise model. Only the decision service developer who is responsible for providing the integration layer needs to be even aware of the enterprise model. Additionally, unit testing of rules is easier using the simple model. The disadvantage is that a new component, namely the Transformer, has to be built as a data adapter. Moreover, as the ruleset evolves, if a rule makes use of an existing enterprise model element that has thus far not been included in the BOM, then a new BOM element will need to be added. This will require modifications to the Transformer and redeployment of the Rule Decision Service.

BOM Adapter Layer

This approach uses the complex enterprise model as the XOM. However, the BOM is not directly derived from this XOM, but is created as an empty BOM entry in the Rule Studio. Then, the rule developer builds all the BOM classes as virtual classes by cherry-picking from the enterprise model and applying verbalizations and BOM to XOM mappings thereby providing a view of the enterprise model that

matches the conceptual model. The B2X mapping is used to flatten the enterprise model vertically (class hierarchies) and horizontally (multiple navigations) which eliminates the need for complex navigations when authoring rules. The rule execution environment and rule authoring environment with the B2X adapter layer is illustrated in Figure 5.

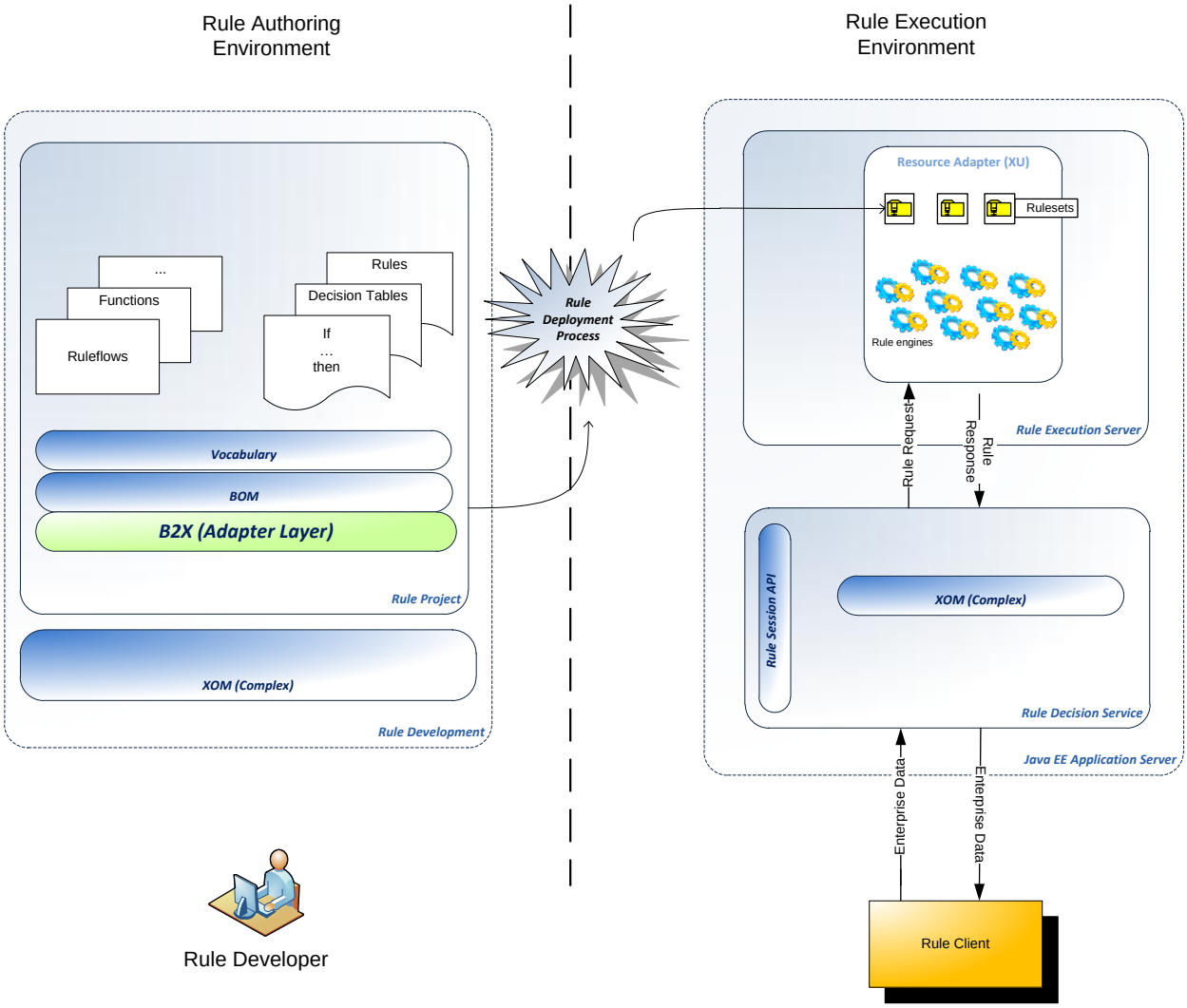

Figure 5: Using B2X to map to a simple BOM

Implementation of the BOM adapter layer for a portion of the BOM outlined in Figure 3 is described below. As illustrated in Figure 6, a virtual *PolicyQuoteRequest* class is defined which has the *ACORD* class as its superclass as well as its execution name (in its BOM to XOM mapping).

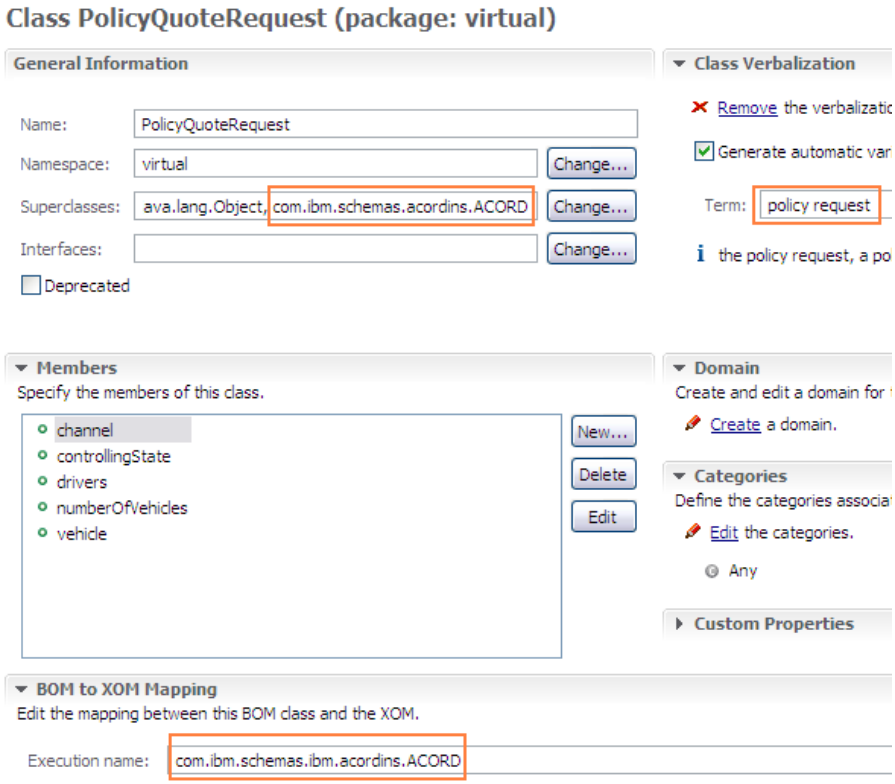

Figure 6: The virtual PolicyQuoteRequest BOM class

Attributes are defined on this virtual class to match the conceptual model. For example, as illustrated in Figure 7, the *controllingState* attribute is defined with a suitable verbalization and an appropriate data type (*StateType*, in this case, which is an enumerated list defined in the enterprise model).

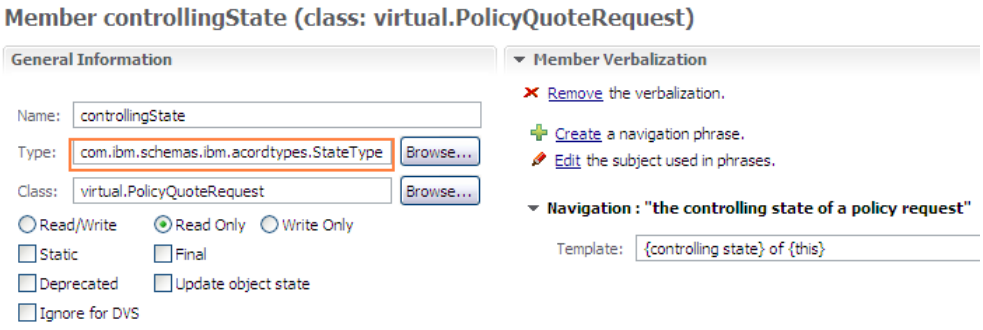

Figure 7: Defining the controllingState member on the virtual class

Since this is a synthetic attribute, the BOM to XOM mapping is used to actually map to the execution model. The BOM to XOM mapping for the *controllingState*, listed below, navigates through a series of objects of the enterprise model to retrieve the controlling state code.

```
return (com.ibm.schemas.ibm.acordtypes.StateType)
this.InsuranceSvcRq.PersAutoPolicyQuoteInqRq.PersPolicy.ControllingStateProvCd;
```

Only these virtual classes are verbalized and therefore rules are only offered the business view of the domain in the BOM. The ugly complexities are hidden in the BOM to XOM mappings.

The main advantage to this approach is that the entire enterprise model is available for potential use in the future. Adding a new BOM element that references an existing enterprise model element can be handled through suitable B2X edits in the BOM editor and is deployed as part of the ruleset, with no changes to the Rule Decision Server. Therefore, it can be hot-deployed.

The biggest drawback is that it requires a lot more development effort to build out the B2X. Also, unit testing is much harder, especially using the Decision Validation Services (DVS) module, since the data is a lot more nested and complex. Moreover, it is not straight-forward to test the rules since building a test case requires knowledge of how the enterprise data is mapped to the BOM elements. Another drawback is that the performance is slightly compromised since the B2X code is not as efficient as Java code.

Considering these drawbacks, unless hot deployment of new BOM elements is critical, it is better to use the Transformer approach.

Using either approach, since the BOM is detached from the enterprise model, a side benefit is that the rule project is insulated from changes to the external model. This is tremendously valuable when the enterprise model is in a state of flux.

Both these approaches prevent cognitive dissonance by minimizing new concepts in the BOM. In addition, the structure and cardinality of relationships that the policy manager uses in *their* view of the domain is carefully preserved.

However, there are cases where we may need to introduce a new element to the BOM to facilitate rule processing; for example, we may use a 'processing status' to keep track of whether the claim has already been denied, using which we avoid further processing. In such cases, it is very important to educate the business users on the purpose and usage of these. The policy managers should be fully cognizant of comfortable with each and every verbalized element in the BOM.

Pitfall: the "abstract" BOM

Philosophically, abstraction is the thought process where ideas are distanced from objects[2]. This thought process, applied to modeling, leads to higher order concepts, taxonomies and ontologies. These are very useful in certain areas, but can be detrimental when applied to a BOM. As the adage goes: the road to hell is paved with good intentions.

This is precisely because it introduces new concepts which lead to *cognitive dissonance*, as discussed in the previous section. This is best illustrated through an example. Consider a claim processing rule application that is responsible for automating the decision of whether to deny or permit a claim. There are several checks that would need to be made on the claim. One such simple rule that a policy manager may verbalize is:

[2] From http://en.wikipedia.org/wiki/Abstraction.

'If the effective date of the policy is after the date of service on any of the service lines, then deny the claim'.

Using a BOM where a claim and the associated policy are input parameters, this can be implemented using the simple rule below.

```
definitions
    set 'the service line' to a service line in the service lines of 'the claim';
if
    the effective date of the policy is after the date of service of the service line
then
    deny the claim
```

This utilizes a simple BOM that matches the business viewpoint of the model, a portion of which is shown in Figure 8. It has concrete classes such as *Claim, ServiceLine* and *InsurancePolicy.*

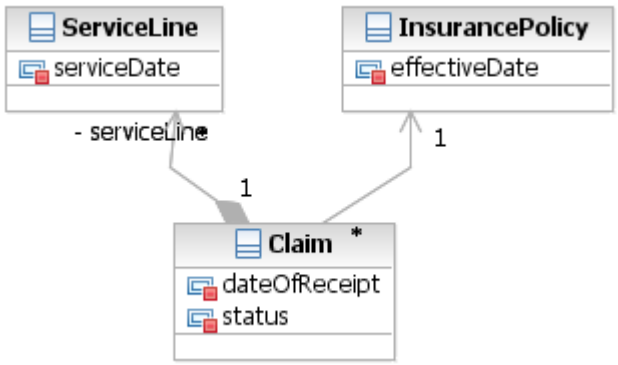

Figure 8: Simple claim model

However, with the goal of flexibility in mind, one may model the claim as a 'form' with several 'form fields' in it. Additionally, one may identify a policy as a just a relationship between the insurance company and the payer and note that these are all entities that have several attributes associated with them. These can then be used to create an abstract model, or an "adaptive object model", as shown in Figure 9, containing concepts that may be understandable, but not commonly used by a policy manager.

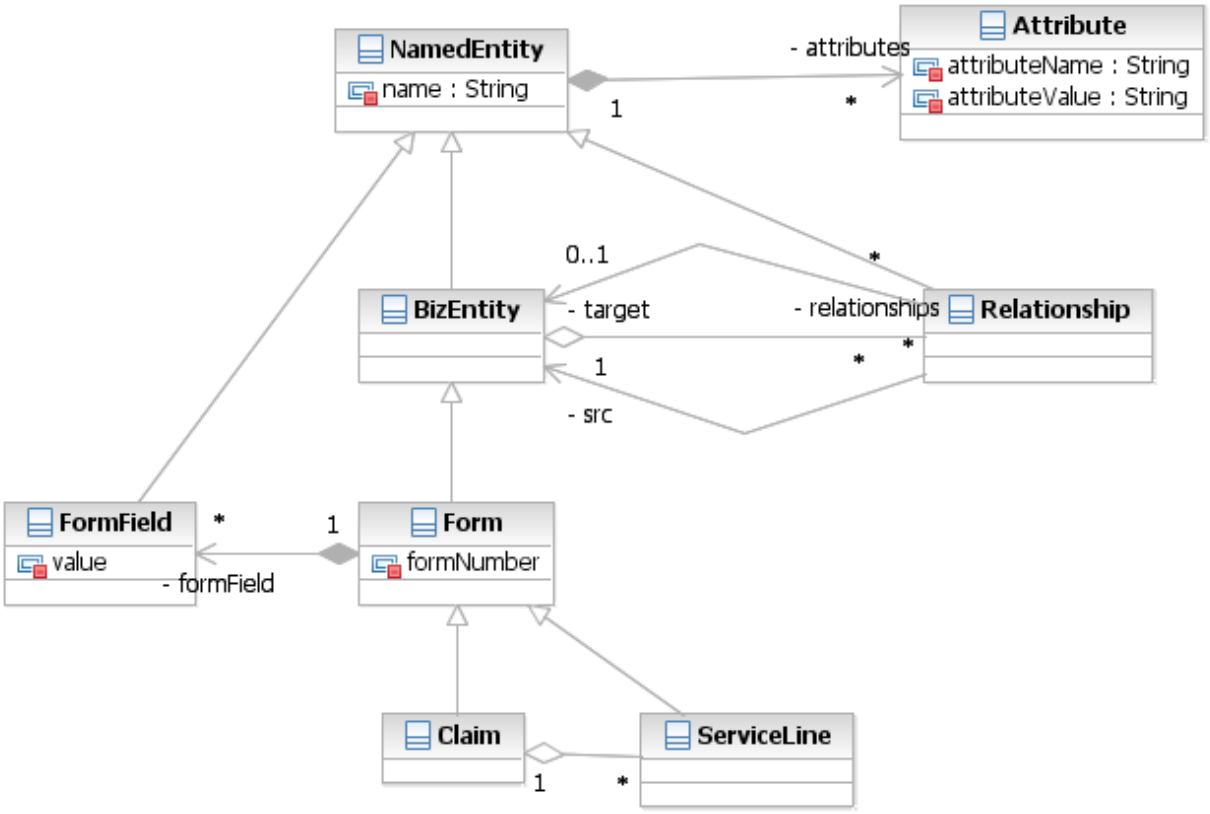

Figure 9: An abstracted claim model

Now this model is highly flexible, because new forms and form fields can be added without making any changes whatsoever to the XOM. New relationships and attributes may be dynamically added too. But at what price? The sample rule implemented with this model as the BOM is listed below.

```
definitions

    set 'the service line' to a service line in the service lines of 'the
claim';
    set 'dos field' to a form field in the form fields of 'the service line'
      where the name of this form field contains "Date of Service";
    set 'the event date' to the value of 'dos field';
    set 'payer' to a relationship in the relationships of 'the claim'
      where the name of this relationship is "Payer";
    set 'the policy effective date attribute' to an attribute in the
attributes of 'payer'
      where the attribute name is "dateOfService";
    set 'the policy effective date' to the attribute value of 'the policy
effective date attribute';
if
    'the policy effective date' converted to date is after 'the event date'
converted to date
then
```

```
    deny 'the claim'
```

From a business user's standpoint, this rule is simply horrific! As can easily be imagined, the policy manager does not find this easy to read, easy to author or easy to maintain. There is a large gap between how the policy manager conceptualizes it and how it is verbalized. This frequently leads to a technical developer having to handle rule maintenance and impact analysis, thereby defeating one of the key potential benefits of a Business Rule Management System. Moreover, using this XOM one cannot answer even simple static structural questions about the model, such as what the attributes of a *Claim* are. This makes it hard for the invoking application to correctly pass the data, and errors are detected only at runtime, not at compile time. Even if flexibility requirements demand such a flexible XOM, the policy manager should be shielded from this through techniques described in the earlier section titled "Techniques for avoiding this pitfall".

Pitfall: the "loose" BOM

With the intent of keeping the model flexible, sometimes a modeler may choose not to use strong typing of the data. For example, consider an input *Account* class as shown in the figure below where all of the attributes are defined to be of type "String". Since the account type is defined as just a string, there is no restriction on what account types may be passed in. So as new account types are added, there is absolutely no change required to this model. Additionally, the '*establishDate*' attribute can accept dates in different formats.

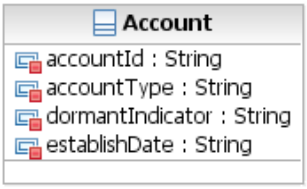

Figure 10: Loose data types on Account

However, when writing business rule with this as the BOM, several problems arise:

1) During rule authoring, no hints are provided about the possible list of values of any of these attributes. The rule author is required to know this list – for example that account type value may only be "Consumer" and "Business".
2) Mistakes made in the string constant, such as incorrect letter case or invalid use of spaces in the string, are not detected during rule authoring. They are only detected during run time when a rule does not fire as expected. Even unit tests may not catch this mistake since the unit test data may use the same string constant used in the rules.
3) The rule client which has to populate the rule request data does not have any restrictions or hints on the values for any of these attributes. For example the model does not tell the rule client if the dormant indicator should be entered as "NOT DORMANT" or "NOT_DORMANT" or "False".

4) Rule conditions have to convert from string to the required data type for usage in methods. For example, in the listing below, the establish date needs to be converted into a date object before comparing with the current date.
5) The biggest chunk of time spent by JRules execution is in XOM method execution. Using Strings, and thus equals operators, through the Rete Networks ops will eventually prove costlier than using '=='.

```
if
    the type of the account is "Consumer" and
    the dormant indicator of the account is "NOT DORMANT" and
    the establish date of the account converted to date is after today
then
…
```

Using a strongly typed BOM as in Figure 11, some of the flexibility is sacrificed, but the aforementioned problems are overcome and the rule authoring and testing process is greatly facilitated. In this model, a Java enumeration called *AccountType* restricts the values that may be passed in from the rule client for this attribute. Since this is a Java enum, there is no possibility for entering illegal data in the rule client. All of the other attributes are strongly typed too.

Figure 11: Account class with strongly typed attributes

With this strong typing, the JRules Intellirule Editor prompts the rule author with the appropriate set of values, as shown in Figure 12; so there is no need for the author to guess the exact string representations and the possibility for this error is eliminated.

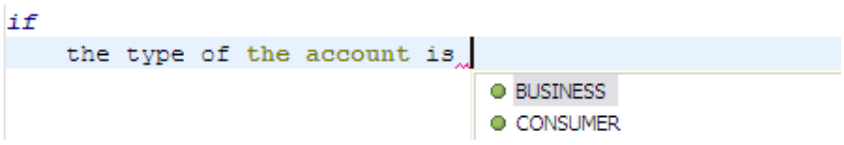

Figure 12: Intellirule prompts for auto completion

Also, no intermediate functions are necessary for rules to use the attributes – as seen in the *establish date* usage in the rule definition below.

```
if
   the type of the account is CONSUMER and
   it is not true that the account is dormant and
   the establish date of the account is after today
```

```
then
…
```

As already noted, the approach of strong data typing comes at a price – that of flexibility. This is particularly true in the case of enumerations. For instance, consider a scenario in the future where an account type is added - this requires a change to the XOM thereby necessitating a change to the rule client, redeployment of the decision service and possibly changes even to the decision service WSDL. So what about those situations where the model needs to be dynamic and redeployment is not an option? Well, there is a solution: the XOM is not strongly typed for those dynamic attributes and restrictions are implemented in the BOM layer. For instance, the account type is defined not as a Java enum, but as a string in the XOM. In the BOM, there are two approaches to restrict this string value to a list of values.

1) Using literal domains
2) Using a virtual enumeration type

Literal domains

A simple way to restrict a string attribute in the BOM is by using a list of literal values as a literal domain. For example, as shown in Figure 13, the account *type* is defined as a string with a domain type consisting of literals *CONSUMER* and *BUSINESS*.

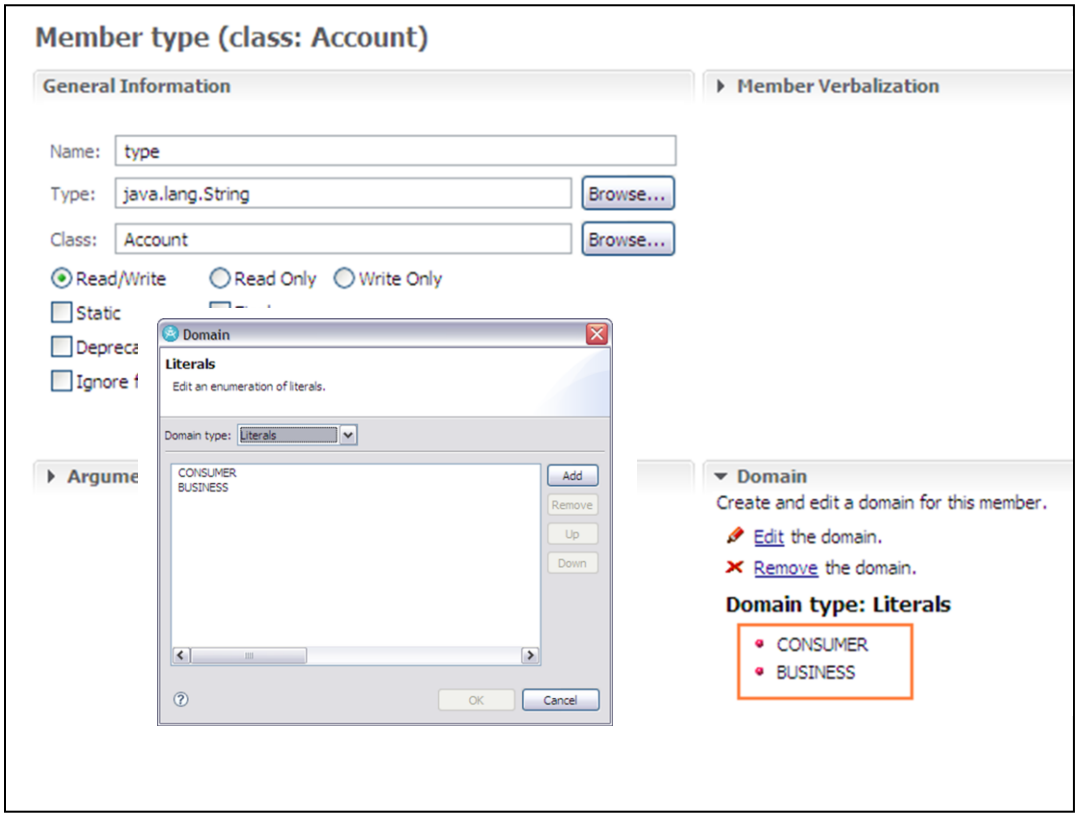

Figure 13: Literal domain

While this approach is simple, the drawback is that these literals may need to be redefined in multiple

places. For instance, if the account type is used as an argument in other methods, these literal domains will have to be redefined for each usage of account type as an argument. In such cases, it is better to define a virtual enumeration type.

Virtual enumeration type

In this approach, a virtual enumeration type is defined as a BOM class. For instance, *AccountType* is defined as a BOM class (as illustrated in Figure 14). The execution class in its B2X is specified as `java.lang.String` and so is its superclass.

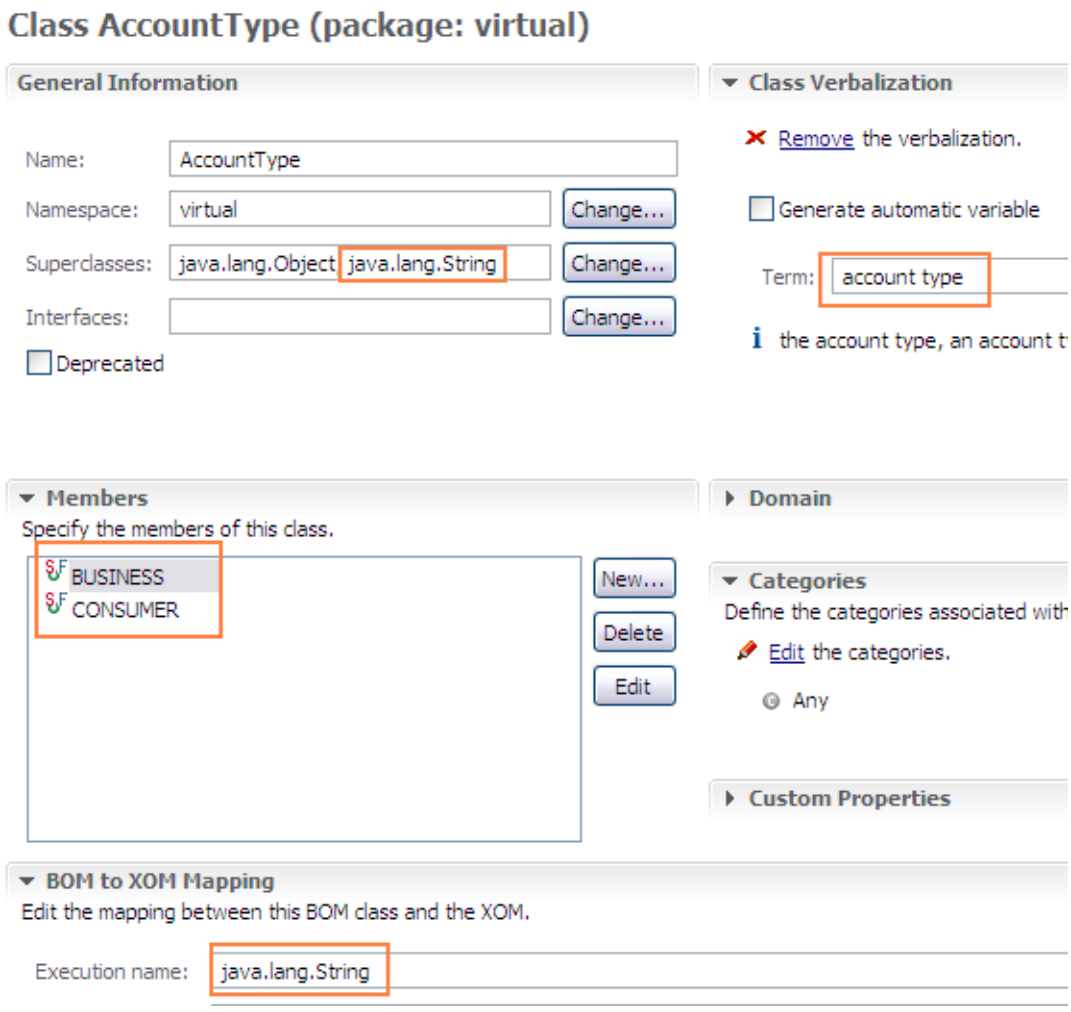

Figure 14: Virtual AccountType BOM class

Now each of the possible values is defined as members of this virtual enumeration class, with the type of that virtual class. For example, *BUSINESS* is defined as a read-only static final member with a data type of *AccountType* and a B2X mapping of "BUSINESS", as depicted in Figure 15.

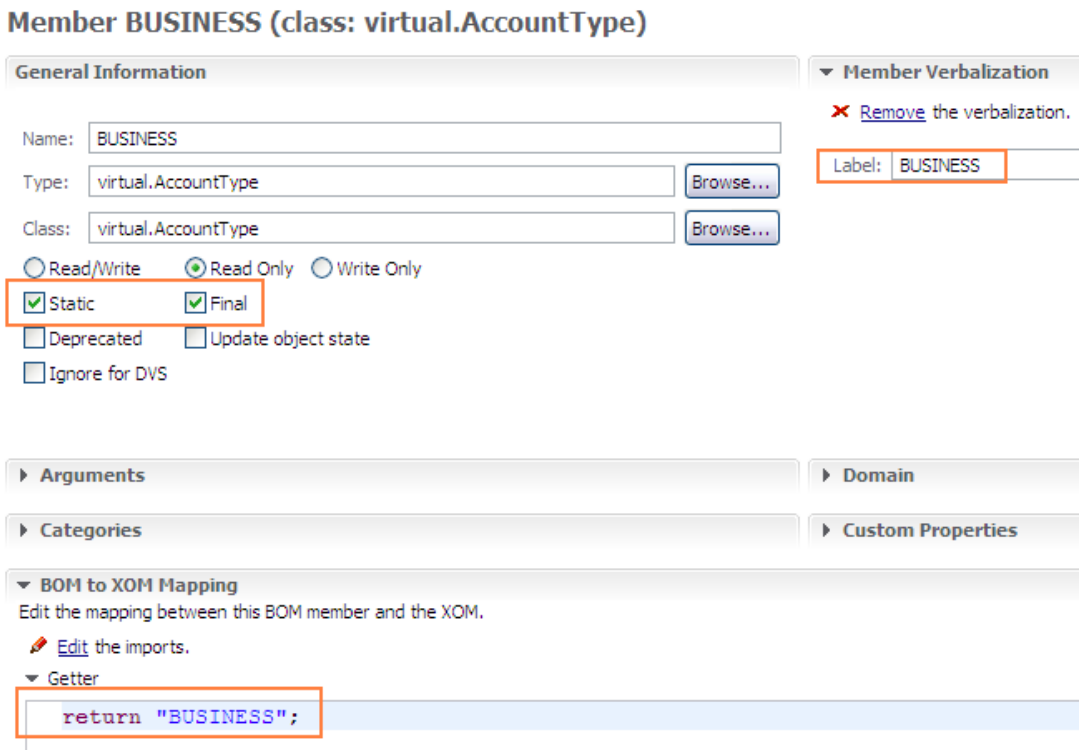

Figure 15: Member definition of virtual AccountType

This virtual enumeration type is used as the data type for the attribute in the Account BOM class as shown in Figure 16.

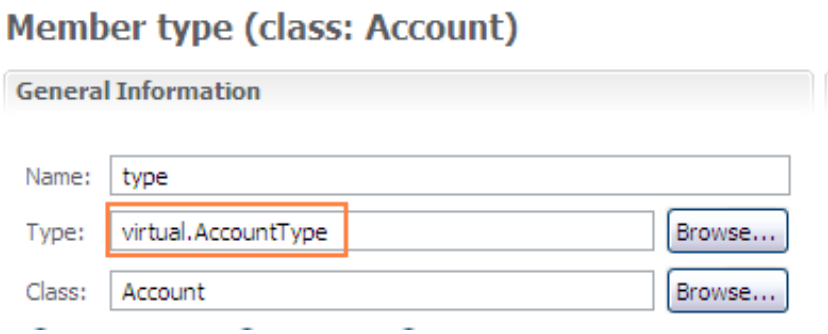

Figure 16: Usage of virtual enumeration class in Account

The main advantage of using this approach is that this virtual enumeration class can be reused to restrict other attributes and methods that utilize *accountType*. However, this approach requires more development effort.

When dealing with a large list of values, this approach can get very tedious. In those cases, it would be preferable to use dynamic domains, which is beyond the scope of this article but a tutorial can be found in the resources section.

Pitfall: Ignoring the rule author

The BOM should be well verbalized as it forms the vocabulary that is used by rule authors when defining rules. Superfluous or poor verbalization leads to rules that are hard to read and maintain and becomes an impediment to the rule author.

Poor verbalization

Frequently, blind usage of the Rule Studio wizard to generate the BOM from a XOM leads to a BOM with less than ideal verbalizations, especially when it comes to methods with multiple arguments. For example, consider a Java XOM method on an *Util* class that adds a notification given an account, the notification message, an error id, the name of the rule producing this notification and the current timestamp:

```
    public static void addNotification(Account account, String message,
String id, String rulename, Date timestamp) {
    …
```

This creates a default verbalization of:

```
util.addNotification({0}, {1}, {2}, {3}, {4})
```

A rule action using this method looks like this:

```
then
  util.addNotification ( the account , "Account not yet established must be
dormant", "W2101", the name of this rule , today ) ;
```

There are some issues with this:

1) The rule author is prompted to enter a string for the second, third and fourth arguments, but there is no contextual information of what these strings are. The author has to "know" that the *message* is to be entered first and then the *id, etc*.
2) The verbalization is java-like and not business friendly.
3) For each occurrence, the rule author has to include the rule name and date, even though it is the current values.

To alleviate these problems, it is better to not verbalize the *Util.addNotification* method, but instead create a new virtual static method on the same *Util* class that only takes the necessary arguments (i.e. without rule name and timestamp). We use a business friendly verbalization that provides contextual information:

```
add a notification using {0, account}, {1, message}, {2, id}
```

Now when the rule author uses this method, there is an indicator of what the argument represents, as shown in Figure 17.

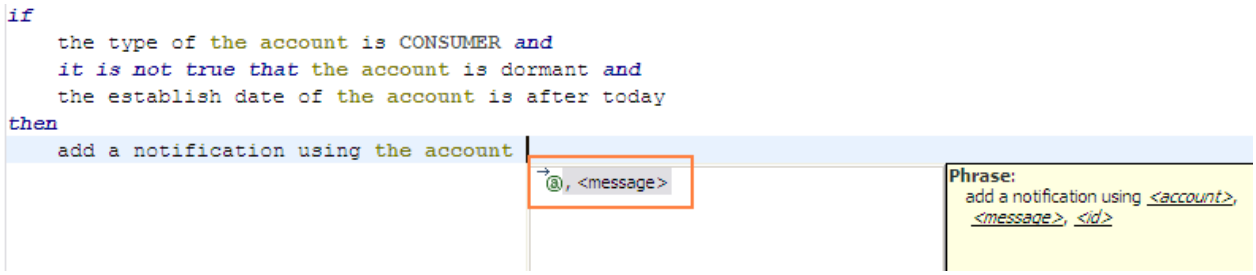

Figure 17: Context specific argument information

Since this is a virtual method, it needs a B2X mapping. Notice that the rule name and timestamp are not passed in as method arguments - it is in this mapping that the rule name and current timestamp are added, as listed below.

```
String name =
ilog.rules.brl.IlrNameUtil.getBusinessIdentifier(instance.ruleName);
java.util.Date now = new java.util.Date();
Util.addNotification(account, message, id, name, now);
```

Superfluous verbalization

In a rush to generate the BOM from the XOM, the Rule Studio wizard for BOM creation may be used indiscriminately without paying attention to which of the attributes and methods should actually be verbalized. This often results in many superfluous verbalizations, especially among setter methods of attributes, thereby needlessly cluttering the drop downs offered during rule authoring. This impedes the ease of use from a rule author perspective. To avoid this, carefully select only those attributes and methods that do need verbalization.

Another scenario where BOM elements may be redundantly added is when different rule projects refer to some common XOM elements. For instance, consider a rule project that has to determine the credit rating of a mortgage customer and another that determines the price to charge for the mortgage. Typically, the common business entities such as *Account* and *Customer* would be in a *CommonXOM* Java project, while credit specific classes such as *CreditBureau* would be in *CreditXOM* while classes such as *PricingSheet* would be in *PricingXOM*. Now, the *PricingRules* rule project depends on both the common XOM and the pricing XOM – so it could have a BOM with two entries mapping to them; and likewise with the *CreditRatingRules* project, as shown in Figure 18.

Figure 18: Common BOM elements duplicated in Pricing BOM and Credit BOM

However, that would be a mistake because the common classes are now duplicated in the BOM entries or *PricingRules* as well as *CreditRatingRules*. Any change to the common BOM would need to be propagated to both these entries, thus leading to maintenance issues.

The technique to avoid this issue is to create a separate BOM rule project that only contains the common BOM and is reused by the two rule projects, as shown in Figure 19. Thus, *PricingRules* references the common BOM that is defined as a separate rule project and the BOM entry in *PricingRules* only contains the BOM classes that are specific to pricing.

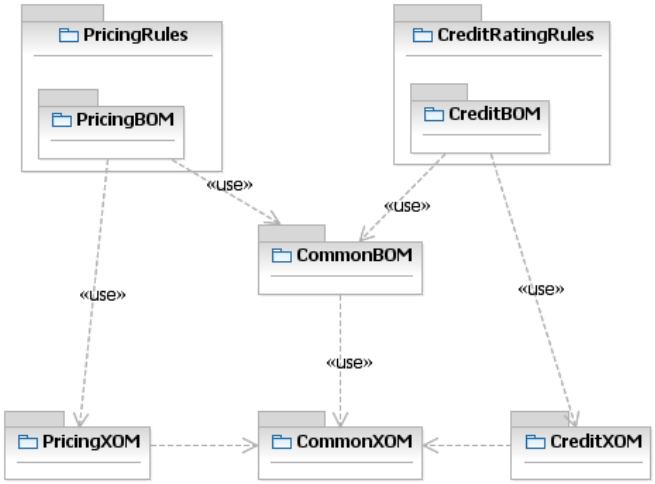

Figure 19: Common BOM is exteranlized and shared

In addition, if some classes and members of the common BOM apply only to the Pricing or Credit rules, then that can be handled by creating two categories called *pricing* and *credit* and assigning appropriate categories to those BOM elements and filtering them during rule authoring through rule category filters.

Conclusion

A poorly designed BOM can greatly impede the productivity of rule authors and maintainability of rule projects. The most important consideration is to prevent cognitive dissonance by absolutely minimizing new concepts and/or terminology in the BOM and by preserving the business viewpoint.

There may be a big chasm between the BOM and the XOM, as the BOM is a product of rule analysis, while the XOM is the design model that is produced through object oriented analysis. The BOM is the business view of the domain from the viewpoint of a policy manager. The XOM is designed with flexibility and performance in mind. Techniques for bridging this chasm have been described in this article.

If introducing new concepts or terminology, it is very important to educate the business users on the purpose and usage of these. The policy managers should be fully cognizant of comfortable with each and every element in the BOM.

The Rule Studio wizard for importing the XOM to create a BOM is an extremely useful tool, but it should not be used blindly. Not only is this true to maintain the necessary separation in viewpoints of the BOM and XOM, but also to judiciously control the verbalization of the BOM elements. Additionally, when BOM elements are to be reused across different projects, enable reuse by creating a separate rule project just to house the common BOM.

Resources

Learn

- WebSphere ILOG JRules Information Center
  The information center contains information describing the IBM WebSphere ILOG JRules BRMS product line and features.

- WebSphere ILOG JRules Home Page
  IBM WebSphere ILOG JRules provides functionality to build and deploy rule-based applications for Java, mainframe and SOA-based environments.
- ACORD
  ACORD (Association for Cooperative Operations Research and Development) is a global, nonprofit standards development organization serving the insurance industry and related financial services industries.
- http://en.wikipedia.org/wiki/Abstraction
- The Adaptive Object-Model Architectural Style
- Dynamic Domain
  Method of integrating business rules with back-end systems in order to populate drop-down menus in the rule editor, which allows users to select valid values when authoring rules

- Agile Business Rule Development: Process, Architecture, and JRules Examples

  Jérôme Boyer (Author), Hafedh Mili (Author)

- Proven Practices for Enhancing Performance – IBM Redbooks

  Pierre-Andre Paumelle

  This IBM® Redpaper™ provides information on the performance aspects of the WebSphere ILOG BRMS. 7.0. A question and answer (Q&A) format is used in order to cover as many dimensions as possible. Furthermore, performance is considered from various perspectives, with an emphasis on production environments and execution.

Get products and technologies

- WebSphere ILOG JRules V7.1
  Get the trial download.