BPM Voices: Organize your rule projects for flexible rule validation

Pierre Berlandier (pberland@us.ibm.com)
Senior Technical Staff Member
IBM

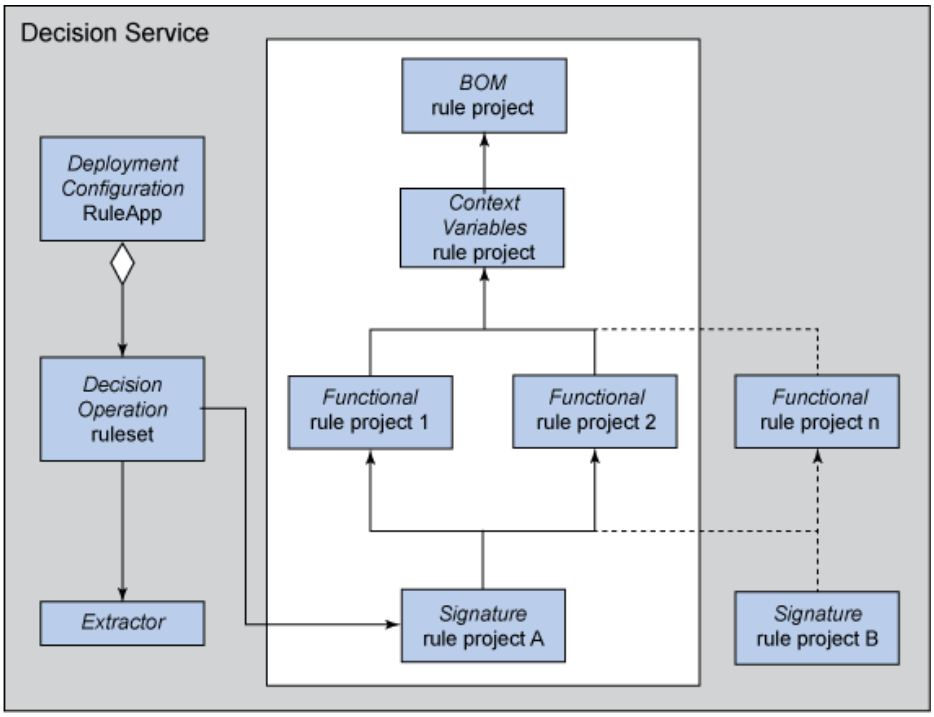
28 August 2013

A well-designed rule project organization is essential to allow distributed rule authoring and facilitate rules sharing. In this column, I'll show that project organization is also key to supporting complex rule testing activities using IBM® Operational Decision Manager Decision Validation Services (DVS). Besides presenting how to address testing on abstract and complex Java™ structures, the column also broadens the perspective on DVS use, showing for example how to alleviate problems such as testing BOM code and reducing the impact of BOM change on test scenarios.

Introduction

A recent IBM RedPaper® on decision governance for the IBM Operational Decision Manager (ODM) platform (see Resources) recommends decomposing and organizing rule projects according to their primary role in building a decision. In particular, it recommends separating the projects that contain the rules artifacts from the projects that define the signature of the decision. The typical organization of projects contributing to the construction of a decision service is illustrated in the diagram in Figure 1.

Figure 1. Organizing rule projects to support decision services

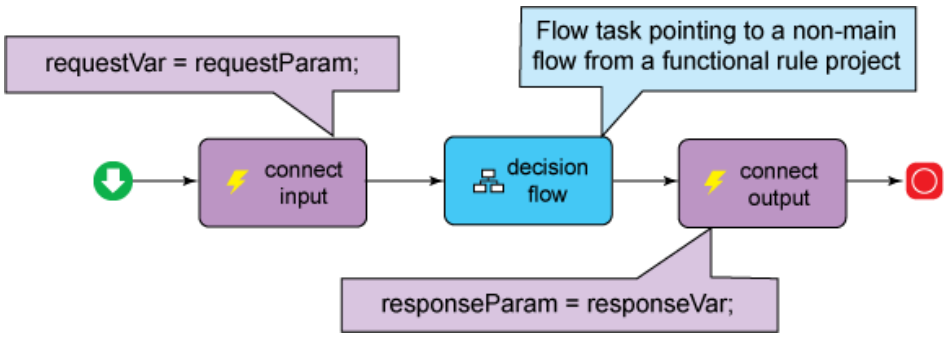


Typically, you would first define the main business object model (BOM) in a project by itself. Another project, referencing the BOM, would define a set of ruleset variables that provide the context of data that is used to write the business rules. Functional rule projects, which reference the context variables project, contain the packages, rules, and rule flows that define the core of the decision operation implementation.

Finally, the signature project defines the ruleset parameters and how they connect to the ruleset variables defined in the context rule project, as well as the main rule flow for the ruleset. Figure 2 shows a typical example of the flow from a signature project, where it is assumed that:

- `requestVar` and `responseVar` are variables defined by the context variables project
- `requestParam` and `responseParam` are the ruleset parameters defined by the signature project

Figure 2. Simple example of a signature project flow



The flexibility provided by this project organization makes it possible to create many different rulesets from the same base set of rules, and to introduce some mediation services between the

rules defined in the functional rule projects and the executable ruleset definition, which enables you to use:

- A variety of ruleset signatures
- Different rule flows
- Additional BOM classes

In this column, I'll describe various benefits of this project organization model for the purpose of validating rule artifacts using the Decision Validation Services (DVS) component of IBM Operational Decision Manager.

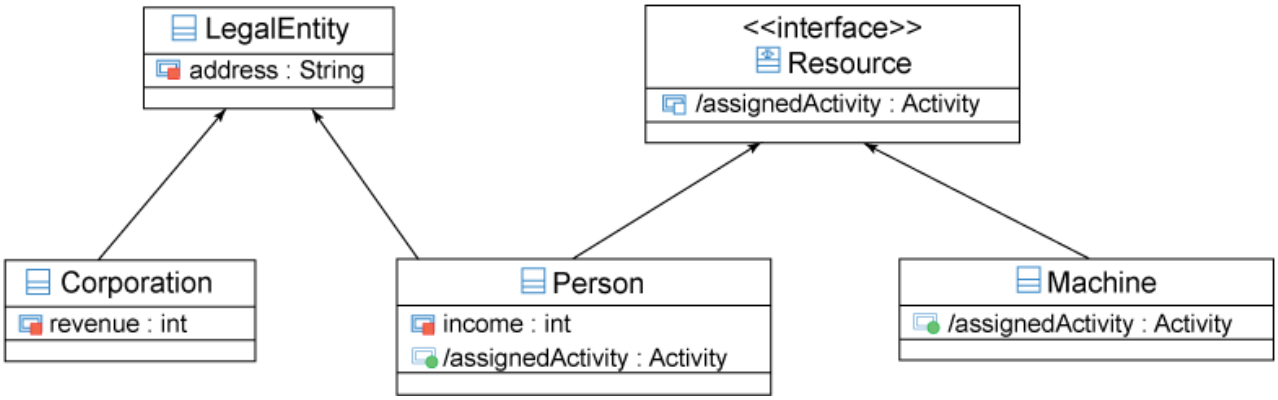
Handling abstract classes and interfaces

When generating a Microsoft® Excel® scenario file template, DVS uses introspection on the input and output ruleset parameters classes from the BOM to generate the required columns in the template Excel spreadsheet.

There is nothing preventing a rule project designer from using a Java™ interface, an abstract class, or simply a non-final Java class to build a BOM entry and use the corresponding BOM entities as types for the ruleset parameters. Actually, it is often the case that, as the adoption of the BRM technology expands in the enterprise, the BOMs from different lines of business get consolidated and the resulting enterprise ontology includes such abstract entities.

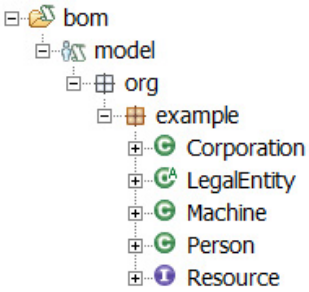
Let's take the example of a simple (and artificial) eXecution Object Model (XOM), as shown in Figure 3.

Figure 3. XOM with interface and abstract class



Here, `LegalEntity` is an abstract class and `Resource` is an interface. Once imported in a BOM entry, they will be associated respectively with a BOM abstract class and interface, as shown in Figure 4.

Figure 4. BOM in Operational Decision Manager with interface and abstract class



Let's assume that we've created a rule project in which one of the ruleset parameters is of type `LegalEntity`. An obvious problem for testing this rule project with DVS is figuring out an Excel scenario file template that allows the definition of test scenarios using either a `Corporation` or a `Person` instance.

One solution is described in the IBM Support TechNote BOM class inheritance and Excel scenario files. It entails the creation of new constructor for the `LegalEntity` BOM class, which, depending on the value of a selector parameter, will return either a `Corporation` or a `Person` instance. This constructor is then marked as the DVS constructor for the BOM class and will thus be used for building all DVS templates involving a `LegalEntity`.

The advantage of this solution is that it is quick to implement from any project configuration. However, one inconvenience is its intrusiveness: the BOM needs to be extended with an artifact (the DVS constructor) that is visible to all, but that is used only in the context of testing activities. Also, as the BOM evolves and subclasses of the `LegalEntity` are added or removed, the BOM to XOM (B2X) code for the DVS constructor will need to be updated accordingly.

Instead, if you adopt the project organization model presented in Figure 1, you would have a signature project that has a `LegalEntity` ruleset parameter. Then, in order to define and execute scenarios on `Corporation` for example, you would just have to clone this signature project, and change the type of the ruleset parameter from `LegalEntity` to `Corporation`.

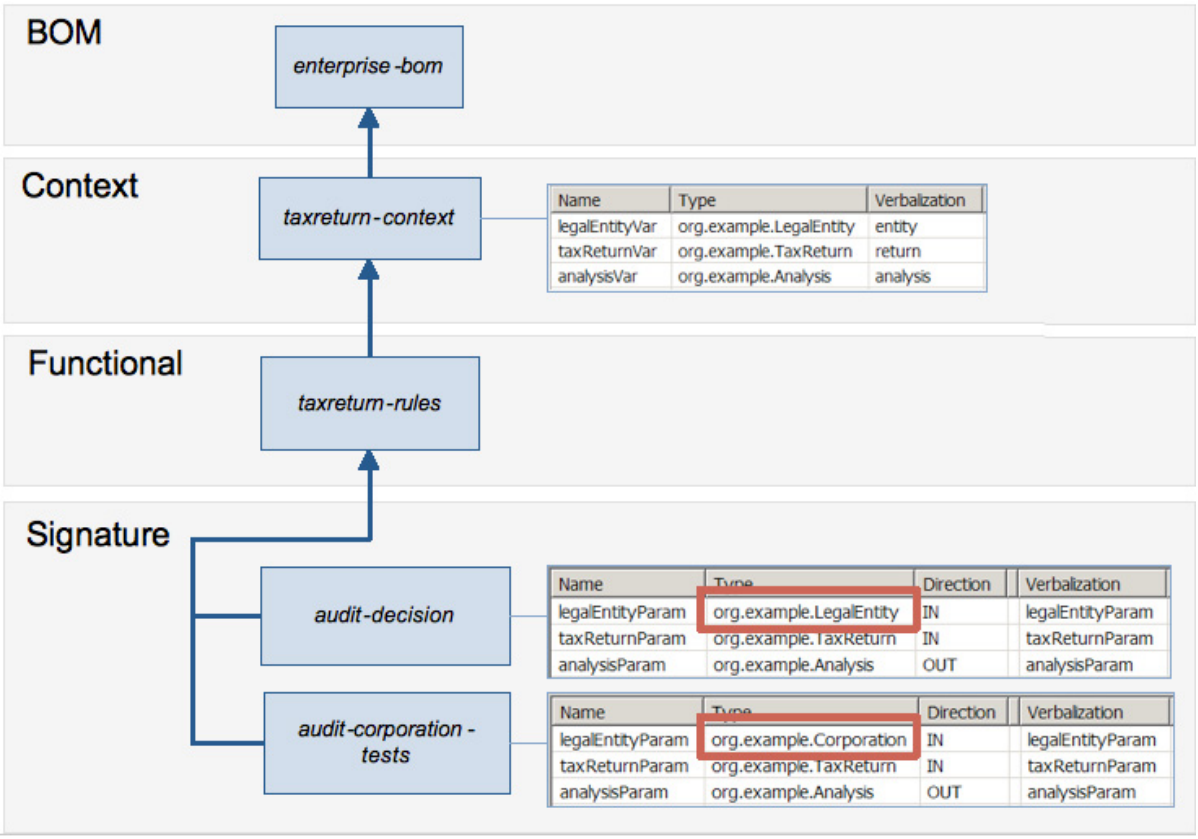
Figure 5 presents such an example, in which:

- `audit-decision` is the signature project that will yield the production ruleset.
- `audit-corporation-tests` is a signature project that is used only for the purpose of performing tests on `Corporation` entities.

The `audit-corporation-tests` project is a simple copy of the `audit-decision` project. The only difference between the two is the type of `LegalEntity` parameter, highlighted in red in Figure 5. When generating a DVS test suite template for `audit-corporation-tests`, the `Corporation` attributes will be directly available on the Scenario worksheet of the spreadsheet.

Note that the same process could be applied for a decision operation that is using an interface, such as `Resource`.

Figure 5. Testing concrete classes using alternate signature projects



Handling complex Java structures

The easiest and most common way to leverage DVS is to define test scenarios as Excel spreadsheets. The spreadsheets provide a business-user-friendly and generic way to serialize a decision's input and output, which works well for object hierarchies, including single or multi-valued attributes.

However, attributes that are defined as complex, nested structures (such as, maps or lists) directly defined from Java types can be problematic. In that case, a special DVS constructor will be required, and structure definitions cannot be reused across test scenarios because they are not associated with an object class and thus cannot be part of a separate tab in the Excel spreadsheet. Let's take for example, a structure that represents the yearly income records of a person by state and then by occupation. The structure could be defined as a data member of a class `YearlyIncomeMap` using standard Java language types as shown below.

```
public class YearlyIncomeMap {
  private HashMap<String, HashMap<String, Integer>> incomes;
  …
}
```

Initializing this map for DVS in an Excel spreadsheet would require a virtual DVS constructor for the `Person` BOM class with, for example, three list arguments defined as shown in Table 1.

Table 1. Virtual DVS constructor signature for Person

Name	Type	Domain
states	java.util.ArrayList	0,* class string
occupations	java.util.ArrayList	0,* class string
incomes	java.util.ArrayList	0,* class java.lang.Integer

Then, in the test scenarios Excel spreadsheet, the definition of the map as an input parameter would use the three columns, as shown in Figure 6.

Figure 6. DVS test scenario data example

		person	income map		
Scenario ID	description		states	occupations	incomes
Scenario 1		Joe	california	fisherman	23000
			california	lifeguard	12000
			texas	roofer	30600

Note that the definition of the income records will need to be repeated for each individual scenario, making the test scenario definition process tedious and error-prone.

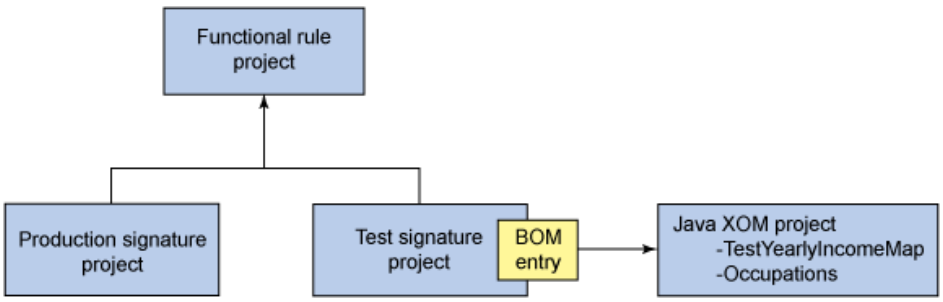
Instead, with the help of a signature project dedicated to testing, you can use an input that is based on a set of custom classes. These classes can be defined in a separate Java project, which is used as the XOM for a BOM entry associated only with the test signature project. You can define, for example, the following two classes:

```
public class TestYearlyIncomeMap extends HashMap<String, Occupations> {
  public TestYearlyIncomeMap(List<String> s, List<Occupations> o){…}
  public YearlyIncomeMap map() {…}
}

public class Occupations extends HashMap<String, Integer> {
  public Occupations (List<String> o, List<Integer> i){…}
}
```

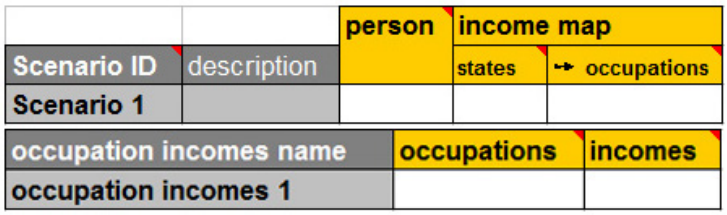
The test signature project will then use the proxy `TestYearlyIncomeMap` class as an input parameter, instead of `YearlyIncomeMap`, as shown in Figure 7, and will internally map the objects using the `map` method, when connecting the context variables.

Figure 7. Using proxy BOM classes for test projects



As a result, the test scenario Excel spreadsheet will support multiple tabs, allowing the reuse of some test data definitions, as shown in Figure 8. Also, the definition of the original XOM and BOM are not impacted, and there is no need to define virtual constructors in the original BOM.

Figure 8. DVS test scenario data with multiple tabs



Note that a reasonable argument can be made here that an XOM should, whenever possible, strive to define meaningful business entities instead of generic structures. However, there are some situations where XOM definition is not under the control of the rule project development process and where having the opportunity to control the structure of the input and output parameters can be helpful.

Validating ILOG Rule Language (IRL) functions and BOM-to-XOM mapping

IBM ODM allows the creation of new attributes, methods and constructors on the BOM classes. The semantics of these new members is defined by the B2X code that is written using the IRL language. The B2X code defines the link between a new BOM member and the members of the underlying XOM. These BOM class members are often called *virtual* members.

In general, it's a good practice to limit the number of virtual members to a minimum (shifting members to the Java XOM layer as much as possible), and also to keep the B2X code short. In some cases, however, (for example, when using an XSD-based XOM, or when being forced to work with a predefined set of XOM classes that cannot be altered or subclassed), reliance on virtual members and IRL functions is necessary.

Since there is no specific facility for testing the BOM layer and IRL functions in isolation from the rules, thorough testing of these entities is often overlooked. However, implementation issues on IRL functions and virtual members can be costly to debug because their code is not directly visible from the rules that use them. It is therefore a good idea to try to test the IRL code.

A dedicated signature project can be used to facilitate the testing of the BOM and the IRL functions. This project should establish a dependency on the BOM, as well as other projects where IRL functions to be tested are implemented.

Let's take the example where the `Person` BOM class is derived from an XML complex type, and is characterized by a date of birth and a list of dependents. In order to author the desired rules, you would define the following virtual methods in the BOM:

- `getAgeOnDate(Date)`: returns a person's age, as of the execution time, or at a specific date.

- `getOldestDependent()`: returns a person's oldest person in his list of dependents.

To test these methods, you can create a dedicated signature project, with the following ruleset parameters:

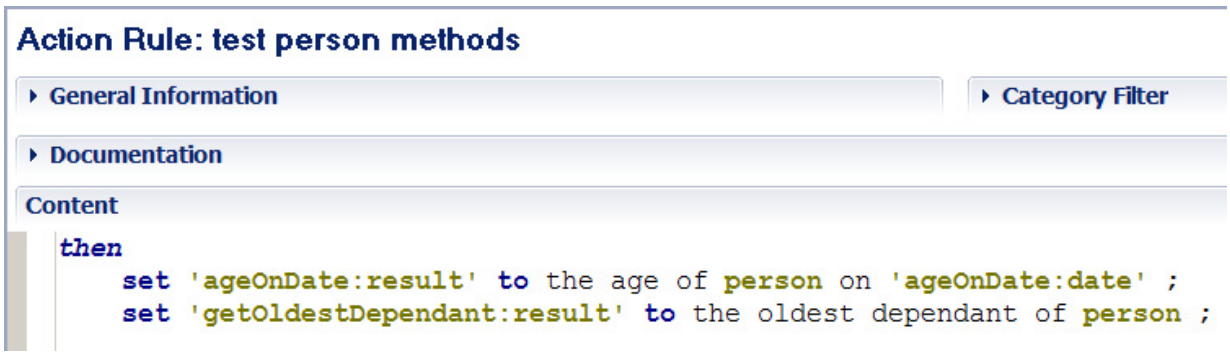
Table 2. Rule project signature for testing BOM members

Name	Type	Direction	Verbalization
person	org.example.Person	IN	person
ageOnDate_date	java.util.Date	IN	ageOnDate:date
ageOnDate_result	int	OUT	ageOnDate:result
getOldestDependent_result	org.example.Person	OUT	agetOldestDependent:result

The inputs are the objects on which the virtual methods are defined (here, the `Person`), along with any parameters required by the methods (here, the target date for the `ageOnDate` method). The outputs are the results from the methods invocations that you want to test.

Within the test project, the methods that need to be tested can be simply invoked through a rule, such as the one shown in Figure 9.

Figure 9. Rule exercising BOM methods to be tested



Using this configuration, you can define test scenarios for the methods using a DVS Excel spreadsheet. Figure 10 shows an example of the different tabs for test scenarios.

Figure 10. DVS test scenario data to test BOM methods

		person					ageOnDate:date
Scenario ID	description	address	→ dependants	dob	income	name	
Scenario 1			Joe	10/20/1968		Rob	
			Jim				
Scenario 2				12/23/1981		Sally	12/24/2013

person name	address	→ dependants	dob	income	name
Joe			8/25/2001		Joe
Jim			8/25/2000		Jim

Scenario ID	ageOnDate:result equals	the name of getOldestDependant:result equals
Scenario 1		Jim
Scenario 2	32	

Creating some insulation from BOM changes

The development of a business rules based decision follows the Agile Business Rule Development process (see Resources). During the first iterations of the project development (the `prototyping` cycle and, to a lesser extent, the `building` cycle of agile business rule development), the BOM is likely to evolve rapidly and significantly.

This state of flux usually makes it difficult to maintain the test suites that are based on the BOM structure: new columns or new tabs may need to be updated, added or deleted from one version to the next, and values need to be reshuffled accordingly.

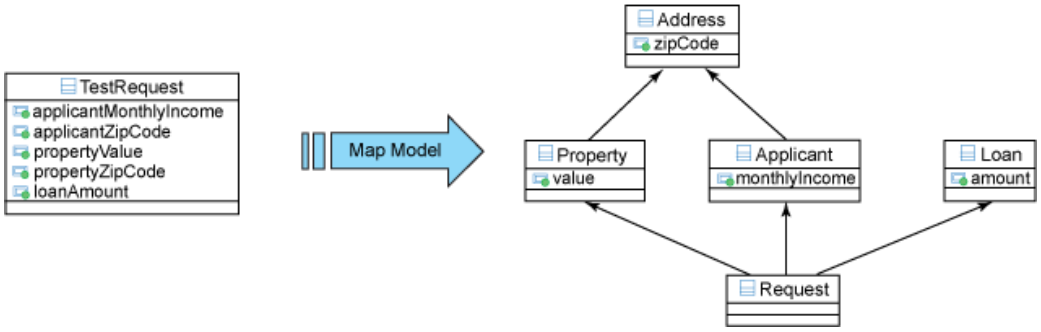
On the other hand, during this same initial prototyping period, when the BOM structure is iteratively elaborated, the number of terms actually used by the rules that are prototyped is actually quite limited and early test scenarios can often be expressed using a flat set of attributes.

For example, say we are working on a mortgage lending application. From a rule perspective, we may have isolated a list of terms that are needed to write some initial rules (such as the applicant monthly income, the property value, the property zip code, the loan amount, and so on). Meanwhile, from a BOM design perspective, we have categorized the terms and identified entities such as `Applicant`, `Address`, `Property`, `Loan`, and so forth, to which the terms belong.

In order to (temporarily) decouple the set of terms used by the rules from the object model that is being constructed, you can use a pattern similar to the one we used for handling complex Java structure (Figure 6)), in which you create a signature project and proxy classes for input parameters.

Here, the goal of the proxy class is to facilitate the input of values for terms used by the rules. The simplest input structure is a flat class, with one attribute per term, as shown in Figure 11. This proxy class also has a mapping function that transforms a proxy instance from a flat set of attributes into a part hierarchy, as intended by the main BOM.

Figure 11. Map model



The test scenarios can then be written easily in a single Excel spreadsheet tab.

- Whenever the BOM design changes, the mapping function needs to be updated accordingly.
- If no attribute is added or removed, the test suite spreadsheets do not need to be updated.
- If attributes are added or removed, corresponding columns simply need to be added or removed from the test suites.

Naturally, as the complexity of the rules in the project grows and you reach the core of the rule building exercise, the flat input structure will likely become inappropriate. However, this should also be the time when the BOM design becomes more stable and structure changes become more rare, and thus less likely to impact the test suites definitions.

Conclusion

In this short column, you've seen the benefits you can realize in rule project validation by adopting a principled decomposition of rule projects into BOM, context variables, and functional and signature projects. Using this type of decomposition enables you to tackle complex situations linked to the complex nature of the XOM (abstract classes, complex Java structures), but also addresses peripheral concerns related to rule projects development, such as testing of IRL code or maintaining test suites through BOM design churn.

Resources

- Governing Operational Decisions in an Enterprise Scalable Way, IBM Redbooks
- BOM class inheritance and Excel scenario files, IBM Support TechNote
- Agile Business Rules Development process
- IBM Software Services for WebSphere: Find out how IBM expertise in cutting-edge and proven technologies can help you achieve your business and IT goals.
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal.

About the author

Pierre Berlandier

Pierre Berlandier has worked for ILOG services for the past 15 years, during which time he has helped ILOG's clients leverage the successive incarnations of the rules programming paradigm from expert systems to real-time intelligent agents, and now business rules applications.