

# Chapter 10

## Rule Authoring Infrastructure in JRules

### *Target audience*

- *Developer, rule author, business analyst (may skip 3.3)*

### *In this chapter you will learn*

- *The structure of rule projects in JRules*
- *The different components of a rule project*
- *Rule project relationships and their importance in modularizing rule development*
- *The Business Object Model (BOM), which is used for rule authoring, and how to build it from the application (or executable) object model*
- *Best practices for organizing rules, and the artifacts they depend on, in rule projects*
- *Best practices for the design of a stable and flexible BOM*

### *Key points*

- *Getting the rule project structure right is an important first step in rule authoring.*
- *Rule project dependencies can be used to modularize rule development and to maximize the reuse of rule artifacts.*
- *The BOM to XOM mapping is a powerful mechanism for obtaining a vocabulary that embodies business needs from an application model geared towards IT needs.*
- *The BOM update and refactoring capabilities of Rule Studio enable us to selectively propagate some changes from the XOM to the BOM, and to shield the BOM – and rules – from the others.*

## 10.1 Introduction

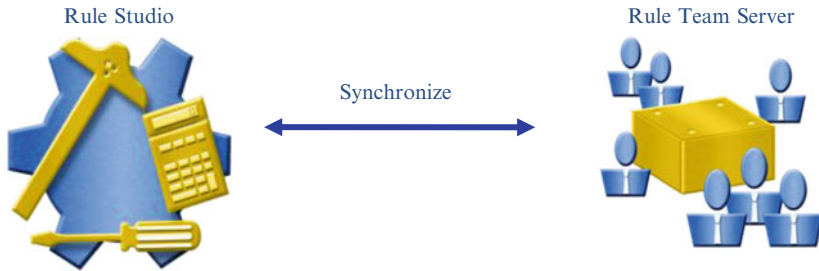
In Chap. 9, we explored the design space for rule authoring, in a technology-independent way, and proposed patterns and best practices for authoring. In this chapter and next, we present the JRules rule authoring artifacts, languages, and

tools. This chapter focuses on the *rule authoring infrastructure*, i.e., the rule project structure, and the set-up of the rule authoring vocabulary; the next chapter focuses on the rule *authoring* per se. This chapter is by no means a user-manual into the JRules rule authoring infrastructure tools. Instead, we focus on some of the design dimensions that were discussed in Chap. 9 that relate to the business object model and to rule organization, but within the context of the JRules product. We start by presenting the concept of a *rule project* in JRules (Sect. 10.2), and the Eclipse-based project dependency relationships, which provide a powerful modularization mechanism. The *Business Object Model* is presented in Sect. 10.3. In particular, we stress the layered structure of the BOM that enables us to (a) separate the business view of the data from the implementation view – the BOM to XOM mapping, (b) separate the terminology from the semantics – the notion of *vocabulary*, and (c) shield rules from (most) refactoring in the implementation model, while propagating changes in the terminology. Best practices for project organization and BOM design are presented in Sect. 10.4. We conclude in Sect. 10.5. Material for further reading is presented in section on “Further Reading.”

## 10.2 Rule Projects

Referring back to Sect. 9.4.1, a *rule project* contains a set of rule artifacts (if-then rules, decision tables, decision trees, etc.) grouped in rule packages, and the elements needed to define them, chief among them, the business object model (BOM). In JRules, projects are first created in Rule Studio (RS), which is an Eclipse-based rule authoring environment with the power – and extensibility – of the Eclipse platform (see Chap. 8). Typically, this is a job for developers who know JRules, as opposed to your typical business analysts or *policy managers*. Indeed, as we will show later, the definition and the customization of the business object model (BOM) requires a good knowledge of the Java language and a good knowledge of the ILOG rules technical language – or IRL – and we will explain why. In addition to setting the BOM, a *rule architect* would typically design the higher levels of the package hierarchy, create some rule templates to be used by rule authors for rule authoring, write some impact analysis or deployment queries, and design the execution behavior of the ruleset (ruleset parameters, ruleflow, etc.). We call this the *rule entry infrastructure*. Once the rule entry infrastructure is set, the developer hands the project over to a rule author for authoring the rules. Depending on a number of factors – discussed in Sect. 10.2.4 – rule authors may work with the Rule Studio environment, or within *Rule Team Server* (RTS), the web-based rule authoring environment. To make the project available to Rule Team Server (or RTS), the developer instantiates a remote connection to RTS from Rule Studio, and uploads the project to RTS. Later on, changes can be made to the project in either environment, and so the two versions will need to be synchronized. Figure 10.1 illustrates this.

We first start by discussing the structure of rule projects by going over the different artifacts that they can contain and their relationships. In particular, we



**Fig. 10.1** Rule projects are first born in Rule Studio before they can be shipped to Rule Team Server

would go over the *business object model* (BOM) without delving into the details since the BOM will be discussed more thoroughly in Sect. 10.3 of this chapter. Next (Sect. 10.2.2), we talk about dependencies between rules projects. Section 10.2.3 presents best practices about organizing rules in projects. Synchronization between Rule Studio and Rule Team Server will be discussed in Sect. 10.2.4.

### 10.2.1 The Structure of Rule Projects in Rule Studio

Roughly speaking, a rule project contains rules grouped in packages plus a bunch of other things needed to define them. Rule projects are a special case of Eclipse projects and consist of the following components:

- *Rule artifacts*, grouped in packages
- The *Business Object Model*
- Rule queries
- Rule templates

#### 10.2.1.1 Rule Artifacts

JRules supports all the rule artifacts discussed in Sect. 9.2.2, and more:

- *Business Action Language (BAL) rules*, which are rules written in a natural language-like format; the Business Action Language (BAL) will be discussed in Sect. 11.2.3
- *Technical rules*, which are rules written in the native execution format;<sup>1</sup> the technical rule language – IRL, for ILOG rule language – will be discussed in Sect. 11.2.2
- *Decision tables*, discussed in Sect. 11.2.4

<sup>1</sup>Almost. More on this later.

- *Decision trees*, discussed in Sect. 11.2.5
- *Scorecards*, discussed in Sect. 11.2.6
- *IRL Functions*, which can be thought of as IRL macros used internally by the rule engine; the IRL language is described in Sect. 11.2.1
- *Ruleflows*, which are procedural constructs used to orchestrate the evaluation and execution of rules
- *Ruleset variables*, which represent variables that can be referenced from within rules and ruleflows
- *Ruleset parameters*, which are used to pass data to the rule engine (and back) from outside applications

Ruleflows, ruleset variables, and ruleset parameters will be discussed in Sect. 11.3.

### 10.2.1.2 Business Object Model

With the exception of ruleset parameters, which are defined as *rule project properties*, all of the rule artifacts are organized in a hierarchy of packages under the “rules” folder. The *Business Object Model* (BOM) represents the business view of the data, and it consists of a set of *Business Object Model entries* (BOM entries). Figure 10.2 shows a single BOM entry, called “claim processing BOM.” Each BOM entry is associated with an *eXecution Object Model* (XOM), which is the native format for the actual objects manipulated by the rule engine. In Sect. 9.2.1, we described the relationship between the business view of the data and the physical view as analogical to that between a relational view, and the physical model view. With JRules, this analogy is a fairly accurate one; we will talk about

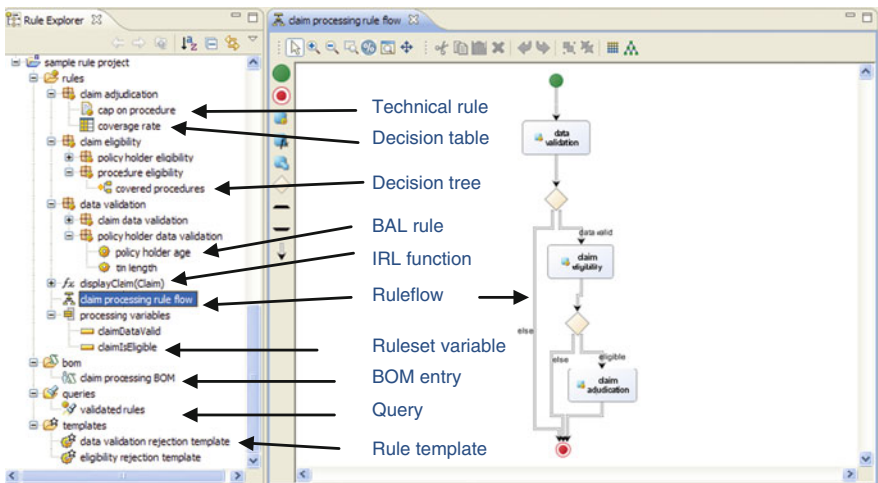


Fig. 10.2 A screenshot of a sample Rule Studio project

the BOM in Sect. 10.3 of this chapter. For the purposes of this section, suffice it to say that *typically* each BOM entry maps to – and references – an *eXecution Object Model* (XOM). Execution object models *typically* consist of the set of Java classes contained in JAR file, or the set of Java classes contained in an Eclipse Java project within the same workspace as the rule project. In the latter case, the rule project has a *project dependency* or *reference* to a Java project.

In JRules, the BOM is the *vocabulary* or *domain of discourse* for the rule artifacts mentioned above: BAL rules, technical rules, decision tables, decisions trees, ruleset variables, IRL functions, etc. Different artifacts use different *views* of the BOM. For example, technical rules and IRL functions use the “raw format” of the BOM, whereas BAL rules and decision tables, for example, use the natural language-like veneer on top the raw format; this will be discussed in Sect. 10.3.

### 10.2.1.3 Rule Queries

Rule projects also contain *rule queries*. One can write queries on the rules of a project<sup>2</sup> based on:

- *Their metadata*. Rule properties include things such as the rule name, its author, its effective date (date at which the rule comes into effect), its expiration date (date at which a rule expires), its development status (e.g., one of “new,” “defined,” “validated,” “rejected,” “deployable,” and “retired”), its jurisdiction (e.g., a particular state or county), etc.<sup>3</sup>
- *The (business) object model elements they reference/modify*. For example, all the rules that reference the “age” attribute of the policy holder or that modify the “payment” field of the claim.
- *Their semantics*. For example, rules that might be triggered by a particular boolean condition, or that trigger another rule.

Queries have many uses in JRules, including:

- *Reporting*. Producing various reports on the contents of a project (e.g., rules authored by X).
- *Impact analysis*. For example to assess the impact of modifying or replacing a particular BOM element (a class, an attribute, an operation) by finding those rules that reference it.
- *Logical analysis*. To compute logical relationships between rules, for analysis and validation (semantic queries).

<sup>2</sup>If a rule project *depends on* another rule project, the domain of the query will be extended to include the rules included in that project. And so on (recursively). More on project dependencies in Sect. 10.2.2.

<sup>3</sup>JRules comes with a predefined set of rule properties, but developers can *extend* the *rule model* and add organization or application-specific metadata.

- *Ruleset deployment.* As mentioned in Chap. 8, JRules makes a clear distinction between the development-time organization of rules – rule projects – and the run-time organization of rules – rulesets. JRules provides a default mapping between the two that packages the contents of a project into a ruleset. With queries, we can filter which rules actually get deployed into the ruleset: example criteria include development status (e.g., only validated rules are deployed), jurisdiction, effective/expiration date, etc. We talk about *rule set extractors* in Chap. 12.

Rule queries are written in the *Business Query Language* (BQL), which is very similar to the *Business Action Language* (BAL).<sup>4</sup>

### 10.2.1.4 Rule Templates

Rule *templates* are fill-in-the-blank rules that rule authors can use to author rules, instead of starting with an empty rule. Figure 10.3 shows the example of a template for rules that reject claims because of invalid data. In this example, rule authors need only write the condition part, which corresponds to a data validation constraint violation; the action part is pretty much completed where only the message argument (the string “Data is invalid: . . .”) needs to be edited.

Rule templates are not only convenient, but they can be used as safeguards to make sure that business rule authors do not mess up the business logic. For example, our processing logic may rely on the fact that data validation rules set the value of a particular data member of class **Claim** to signal a data validation constraint violation. We have to make sure that all the rules that detect validation errors properly set that data member. Using a rule template that builds in that action



Fig. 10.3 A rule template

<sup>4</sup>On some level, a rule is like a query: the condition part ‘queries’ the working memory for tuples of objects satisfying some conditions, and the action part applies some actions to the result.

would do the trick. In the template of Fig. 10.3, we put into the rule action part two actions that set the appropriate data member to the appropriate data value – what the action “reject the claim” actually does behind the scenes<sup>5</sup> – and logs the rule firing. In this case, it so happens that those two actions are *frozen*, i.e., they cannot be removed from rules generated from the template; the rule template editor enables us to freeze or unfreeze selected parts of the rule template, down to the function argument level. In the remainder of Sect. 10.2, and the subsequent sections, we will revisit various rule project constituents that we discussed here.

While rule projects can only be *created* in Rule Studio, they can be modified in either Rule Studio, or in Rule Team Server, the web-based rule authoring environment (see Chap. 8). To edit a rule project in Rule Team Server (RTS), we need to first publish it from Rule Studio to Rule Team Server (see Sect. 10.2.4). After logging in to Rule Team Server, we can edit project elements there. We will revisit this in Sect. 10.2.3, but for the time being, we can think of the two projects (the RS version and the RTS version) as equivalent.<sup>6</sup>

## 10.2.2 Rule Project Dependencies

We mentioned in the previous section that rule projects *typically* contain the definition of a *business object model* (BOM), and that the BOM is *typically* derived from Java class definitions contained in JAR file, or in a Java project within the same workspace. JRules enables us also to define dependencies *between* rule projects. If a rule project  $RP_A$  *references* a rule project  $RP_B$ , then:

- The BOM defined in  $RP_B$  is *visible* within  $RP_A$ , meaning that we can write rules that refer to the BOM of  $RP_B$ .
- The ruleset parameters defined in  $RP_B$  are accessible within  $RP_A$ , meaning that we can refer to them in the rule artifacts of  $RP_A$ .
- The ruleset variables defined in  $RP_B$  are accessible with  $RP_A$  by using an import statement.
- The rule artifacts defined in  $RP_B$  are accessible within  $RP_A$ . In particular (a) templates defined within  $RP_B$  can be used within  $RP_A$ , (b) rule queries defined in  $RP_B$  can be used to define ruleset extractors in  $RP_A$ , (c) rule packages defined within  $RP_B$  can be assigned to rule tasks within  $RP_A$ , (d) rule flows defined within  $RP_B$  can be assigned as *flow tasks* rule tasks within  $RP_A$ , and (e) rules in  $RP_B$  can override rules that are in project  $RP_A$ .
- Queries defined in  $RP_B$  can be run on the artifacts of  $RP_B$  alone, or be extended to artifacts in  $RP_A$ .

Within Rule Studio, project dependencies are limited to the rule projects within the same workspace. They are editable through the project’s *properties* under

<sup>5</sup>More about this in Sect. 10.3.

<sup>6</sup>Not true, but an acceptable first approximation.

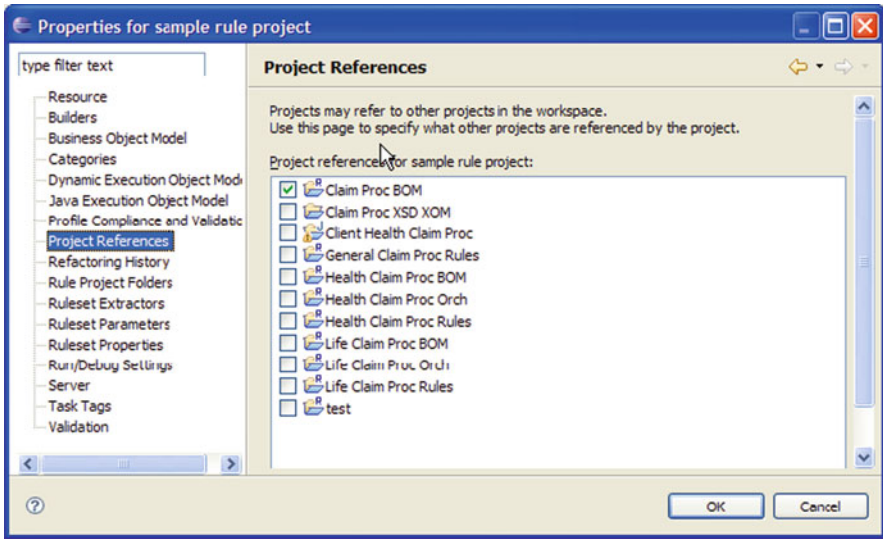


Fig. 10.4 Adding project references to a rule project

“Project references,” like any Eclipse project. Figure 10.4 shows a screenshot of the corresponding property editor. In this case, we have 11 projects in this workspace, and the project “sample rule project” (our  $RP_B$  above) references project “Claim Proc BOM” (our  $RP_A$  above). Within Rule Team Server, project dependencies are limited to projects within the same repository, and only users with role “Configurator” and above can edit a project’s dependencies.<sup>7</sup>

One of the subtle implications of rule project dependencies is the notion of *BOM path*, similar to the concept of *class path* with Java code. Indeed, because the BOM of a project  $RP_A$  is made visible to project  $RP_B$  if  $RP_B$  references  $RP_A$ , potential conflicts could arise if the same entity – a BOM class – is defined in both the BOM of  $RP_A$  and the BOM of  $RP_B$ . Hence, the notion of a *BOM path*. For each project, JRules defines a default BOM precedence, which an author can edit. The default precedence rules of BOM entries are:

- Within a single project with no external references, BOMs appear by order of definition, with the most recent BOM entry showing last.
- Within a given project, *dynamic BOM entries*, i.e., BOM entries generated from XSDs, appear *before* BOM entries that are built from Java XOMs.
- If project  $RP_B$  references project  $RP_A$ , then the BOM entries of  $RP_A$  appear *before* the locally defined BOM entries.

<sup>7</sup>This means that a regular rule author cannot edit projects’ dependencies, and that is a good thing, because of the subtle implications on rule writing (BOM visibility, variable visibility, etc.).



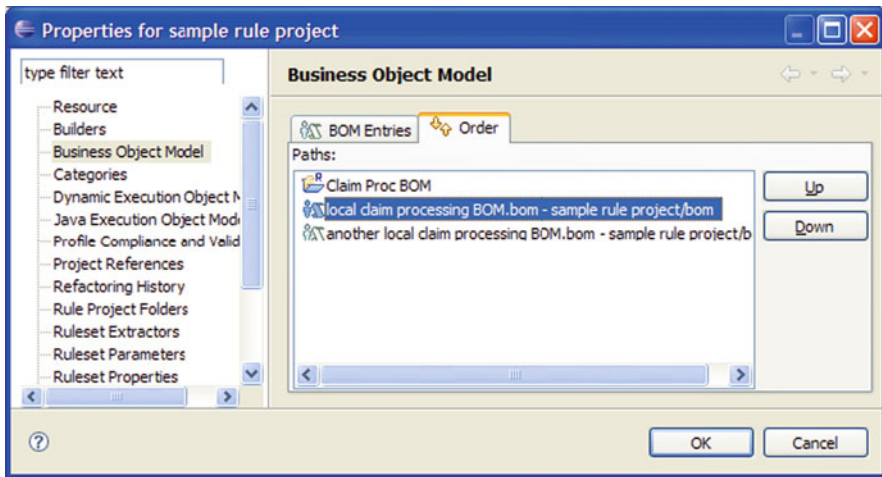


Fig. 10.5 BOM path

These rules apply recursively, to a dependency chain of any length. Users can override these defaults by moving BOM entries up and down the BOM path (see Fig. 10.5).

With the BOM path of Fig. 10.5, if some class `com.mywebinsurance.model.Claim` is present in both the BOM of the rule project “Claim Proc BOM,” and in the locally defined “local claim processing BOM,” it is the definition found in the project “Claim Proc BOM” that will be taken.<sup>8</sup>

Rule project dependencies enable us to modularize our rule development, and to get an effective division of labor. Section 10.4.1 will present best practices about organizing rules in different related projects.

### 10.2.3 Synchronizing Projects Between Rule Studio and Rule Team Server

As mentioned above, projects are first born in Rule Studio. However, Rule Studio is not an appropriate tool for your typical business rule author (*policy manager*), for three reasons:

- Its GUI metaphors are more geared towards developers, and relate little to business metaphors.
- Its computational requirements often exceed the capabilities of the desktop (or laptop) of the typical business user.

<sup>8</sup>This is counter-intuitive to inheritance, where locally defined structures and behavior prime over inherited ones.

- It is a dangerous tool to put in the hands of business users, who may inadvertently break the structure of a rule project and the underlying business logic.

Hence Rule Team Server (RTS) differs from Rule Studio (RS) in two major ways:

- Some rule project elements are *not* editable in RTS, for two main reasons: (a) safety, to maintain the integrity of the project, and (b) because the information needed to edit the element is not available in RTS. Examples of (a) include *rule-flows*, *ruleset parameters*, the *business object model* (BOM), and the BOM path. Examples of (b) include the BOM, again, as well as customizable features that require Java code.
- A (far) more granular access control to the elements of a rule project. Whereas RS controls access to projects and project elements through access control mechanisms of the underlying source code management (SCM) tool (ClearCase, Subversion, MKS, etc.), RTS combines a coarse-grained role-based access, which controls which roles have access to which functionalities, with a fine-grained permission management, to manage read/write access to the rule artifacts of a project.

There are a number of other minor functional differences, some of which were explained in Chap. 8.

Implementation-wise, RS and RTS use different representations of projects and project elements:

- RS uses a file-based representation of project elements. Typically, we have one file per artifact, be it a BAL rule, a technical rule, a decision table, a decision tree, a scorecard, a rule flow, a variable set, a function, etc. Aggregate artifacts are represented as a directory that includes a file that represents the aggregate metadata, and the actual contents of the aggregate, as files or subdirectories included therein. Figure 10.6 shows part of the file hierarchy for our sample project. Hence access to these artifacts is managed through the underlying source code control system.
- RTS uses database persistence. Roughly speaking, RTS uses different tables to represent different types of artifacts<sup>9</sup> and the relationships between them. Concurrent access to these artifacts is thus managed through the database locking mechanism.

JRules provides functionality for exporting a project from RS to RTS. This functionality makes a remote connection from RS to a running instance of RTS and either creates a fresh new project in RTS to receive the exported RS, or synchronizes the current state of the RS project with the most current state of the corresponding RTS project.

Figure 10.7 shows a screenshot of the project synchronization wizard. In this particular case, we are synchronizing our “sample rule project” between RS and the

---

<sup>9</sup>We don't have a table for *each* rule project element types as similar element types are represented by the same table, but have one attribute distinguish between them.

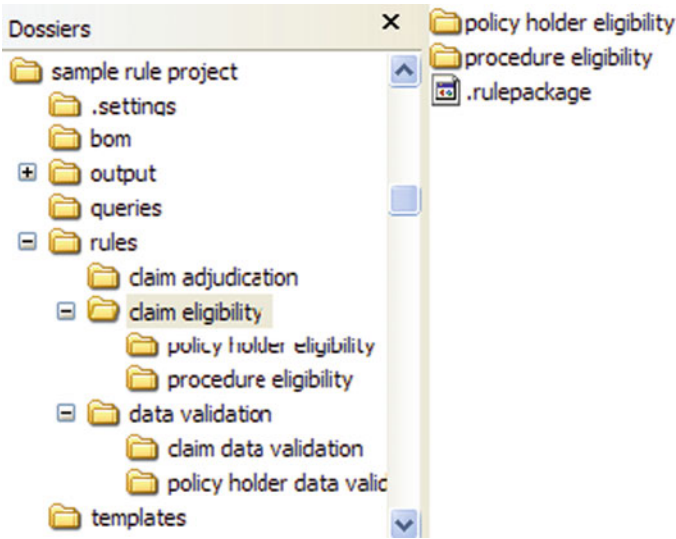


Fig. 10.6 RS represents rule projects using the file system. The right-hand side shows the contents of “claim eligibility” directory, which represents the “claim eligibility” rule package

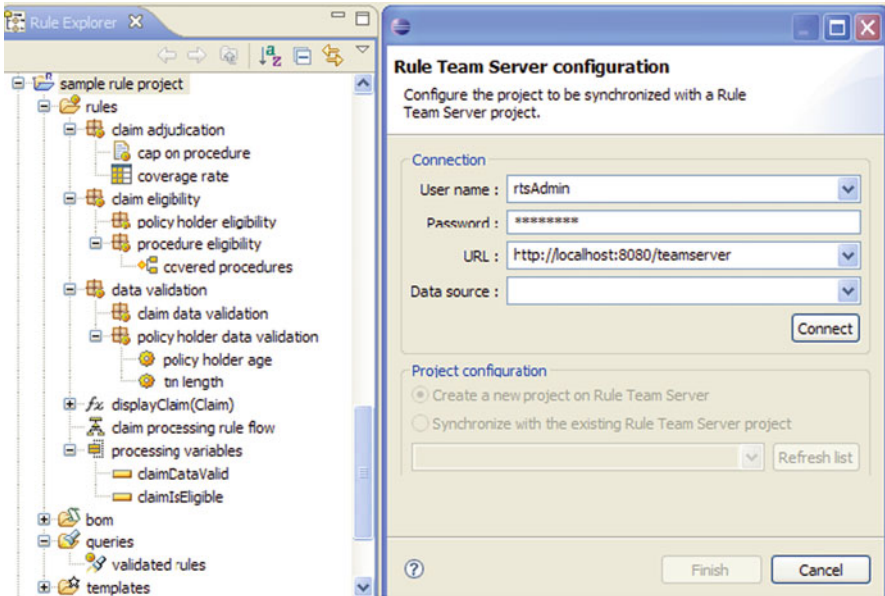
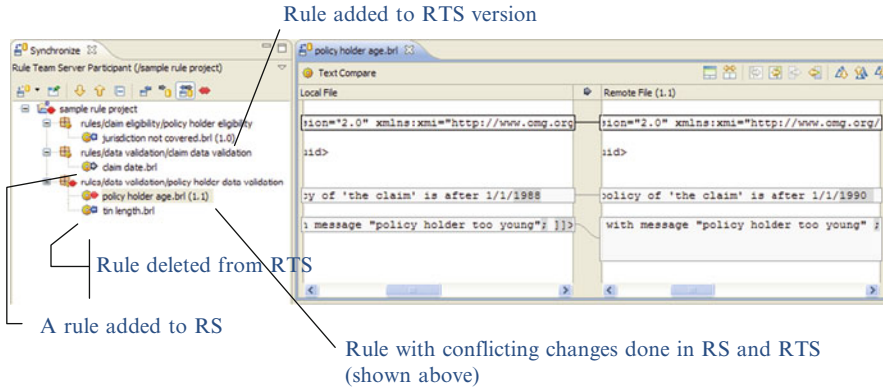


Fig. 10.7 Synchronizing the RS “sample rule project” with the instance of RTS running at <http://localhost:8080/teamserver>



**Fig. 10.8** A case where synchronization yields several changes in both RS and RTS versions

instance of RTS. To *create* a new project on RTS, we need to connect to RTS as an administrator. But to simply synchronize an RS project with an *existing* RTS project, regular user (policy manager) privileges suffice.

Having exported an RS project to RTS, the two projects will typically continue to evolve independently, and will go out of sync. When we synchronize the two versions, the synchronization functionality will perform a three-way comparison between (a) the current version of the RS project, (b) the current version of the RTS project, and (c) the initial version of the project in RTS. Indeed, the initial version of the project in RTS corresponds to the one point in time where the two versions were synchronized. Figure 10.8 shows a screenshot of the synchronization view.

Figure 10.8 illustrates cases where:

- A rule was added to the RTS version: case of rule “jurisdiction not covered,” in package “claim eligibility/policy holder eligibility.”
- A rule was added to the RS version: case of rule “claim date,” added to package “data validation/claim data validation.”
- A rule was deleted from RTS version: case of rule “tin length” in package “data validation/policy holder data validation.”
- A rule was changed concurrently in both RS and RTS, and the changes are conflicting: the case of rule “policy holder age” in package “data validation/policy holder data validation.” In this case, RS puts side to side the two versions, and does a text compare, highlighting the parts that were changed.

Users of the Eclipse environment will recognize the familiar look and semantics of source code management (SCM) plug-ins, when a developer tries to check in their local copy of a project in the corresponding SCM repository. Roughly speaking, in those cases where the change is one-sided, the user has the option of either accepting the change, or rejecting it. In the case of conflict, the user can selectively combine changes coming from either side, if she/he wishes to. The

example of Fig. 10.8 shows three conflicting changes within the rule “policy holder age.”<sup>10</sup>

While JRules offers functionality to synchronize projects between RS and RTS, an organization needs to put in place a set of processes, both manual, and automated, to prevent development chaos; having to resolve conflicting changes between two versions of rules should not be a way of rule project life. The first question that we ask is: notwithstanding the initial creation of a rule project, do we really need to have projects edited in both environments. The answer lies in which environment is being used for creating and maintaining rules, which depends on who is responsible for maintaining the rules. We have encountered three typical scenarios.

*Business users are responsible for creating and maintaining the rules, and they use RTS.* This is the textbook scenario of usage of the tool set. In this case, developers create the project(s) in RS, export them to RTS, and let the business users edit the projects there. At first glance, there should be no reason in this scenario for the version that resides in RS to change. Thus, when developers synchronize the RS projects with the RTS versions,<sup>11</sup> they should not encounter any conflicts. However, even in this case, there are going to be cases where the RS projects need to be updated. First, recall that the BOM and the ruleflow are not editable in RTS. Thus, if either needs to be changed, we can only implement the change in RS. Second, there are cases where rule testing identifies a problem with a ruleset that cannot be identified through tracing. In such cases, we need debugging of the kind that is available only in RS, and a developer may have to correct the problem (i.e., edit the rule).

So what we do in those cases where we do have to change a project in RS? The safest – and coarsest – solution consists of freezing the project in RTS by making it not editable there,<sup>12</sup> then synchronizing RS with RTS to bring the most recent version from RTS to RS, then making the desired changes in RS, then synchronizing the projects again to export the change to RTS, then releasing the project in RTS. This solution will work in all cases. However, it makes an RTS project unavailable for editing for the time it takes the make the needed changes within RS. This may be justifiable if we are making a change to the BOM, for two reasons: (a) to take advantage of RS’s refactoring functionality and (b) to prevent the business users from using the old BOM. Idem for the ruleflow, as it provides the execution context for the rules, and rule authors need it to be current.

---

<sup>10</sup>Actually, in this case only the one about the threshold birthdate for the policy holder is substantive. The others are due to small – non essential – differences in serialization format between the two environments.

<sup>11</sup>One reason we may want to do that is if deployment to the Rule Execution Server (RES) of the development, testing, QA, or production environment is performed from RS as opposed to from RTS, which is the recommended practice.

<sup>12</sup>We can do that by removing a user group from the groups of users who have the right to access the project. By doing so, the project no longer shows up in their project selection in RTS.

If the change that we need to make in RS concerns a single or a handful of rules, then we can make our “freezing” in RTS more selective, as opposed to freezing an entire project. An administrator can log into RTS and lock those rules that are being debugged for the duration of the debugging, to release them later, possibly with changes.

Whatever the case, to prevent conflicts between changes made in RS and RTS, we serialize those changes in time so that, at any given point in time, a project is being edited only in one environment. This is a combination of (a) JRules basic functionality (synchronization, RTS permission management, RTS locking), and (b) a human process that uses this functionality. Adherence to the human process is, naturally, crucial and is part of rule governance, to be discussed in Chaps. 15 and 16.

*IT developers are responsible for creating and maintaining the rules, and they use RS.* While one of the goals of the business rules approach is to have business take over the creation and maintenance of rules, the *complete take over* from rule discovery all the way to rule coding does not *always* happen, on the initial development of the business application, or in subsequent rule maintenance mode. Many customers that we encountered prefer to leave rule authoring and unit testing to IT.<sup>13</sup> In this case, developers use RS to author and maintain rules. However, rules are available for business to *view* within RTS, in a read-only mode. This scenario is easy to handle: developers simply synchronize the RS project with RTS whenever they have a new stable version (or “release”) of the RS project to share with business.

*Business is responsible for creating and maintaining the rules, and they use RS.* We also encountered this scenario in many situations where business units have in their midst what we call “technical business analysts” who are, typically, ex-developers who are quite comfortable with Rule Studio’s interaction metaphors. This scenario is no different from the previous one and presents no challenges.

#### 10.2.4 Managing Multiple Users

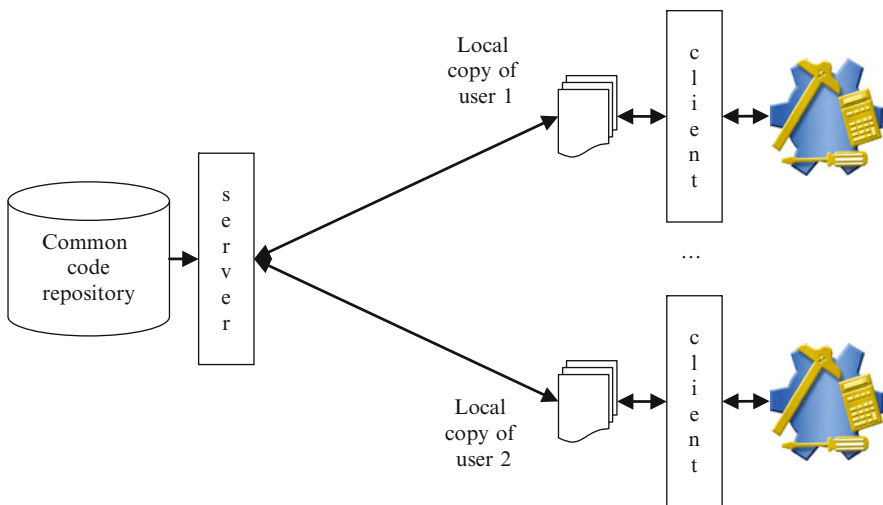
Whether we are using RS or RTS to author and maintain rules, we need to manage multiple users creating and modifying rules within the same environment. As mentioned in the previous section, RS uses a file representation of rule project elements, and manages the different versions of rule project elements through the connectivity of RS to a source code management system. By contrast, RTS uses a database to store rule project elements, and manages concurrent access through the database. In the remainder of this section, we will first summarize the concept of operations of source code management software, and see how those apply to rule projects. We will then discuss the many access control features of RTS.

---

<sup>13</sup>We have also encountered situations where IT developers are transferred, administratively, from the IT department over to business units where they report to a manager within the business unit.

In Rule Studio, coordination between multiple users is managed by the underlying source code management software. There a number of source code management applications, both commercial, such as SourceSafe, ClearCase, MKS, or Perforce, as well as open source ones, such as CVS, Subversion, and others. There are two general approaches to handling multiple users, *pessimistic locking*, of the kind done by databases, where only a single user has write access to a particular resources, and *optimistic locking*, which is a euphemism for no-locking at all. This strategy is *optimistic* because it makes the optimistic hypothesis that users will work on different parts, and thus, there will be no need to lock entire projects, say, if one just wants to change a rule. If the “optimistic hypothesis” turns out to be wrong, then we deal with it with conflict resolution, as illustrated for the case of synchronization between RS and RTS.

The tools mentioned above differ in functionality, but most use optimistic locking. This model is illustrated in Fig. 10.9. Different users work on their own local copies of rule projects, which they synchronize regularly with the state of a common repository. To have access to a given rule project, a user needs to be *registered* within the SCM and be granted access to the repository containing that project. The first time they access the project, they typically *check-out* a particular version of the project from the repository, to get a local copy on their machines or private workspace. They can then work off-line from the SCM repository making as many changes as they wish. If they want to make their work available to others – or simply to back it up – they *check-in* their work. This creates a new version of the project in the repository. If other users have checked out the same version as the current user and have already checked-in their changes, then the tool performs the kind of comparisons we showed in the previous section. If a user judges that there are irreconcilable differences between the version of the project that they want



**Fig. 10.9** The concept of operations of source code management software plug-ins to RS

to check in, and the latest version in the repository, they can then start a new development *branch*.

Note that the model illustrated in Fig. 10.9 applies to the case where the client component of the SCM is well-integrated with the Rule Studio, i.e., as an Eclipse plug-in. If that is not the case, then the client module and the local copy in Fig. 10.9 trade places: we can use a separate command-line SCM client interface to check-in and check-out projects, and use RS on the local copy, totally unaware of the SCM. This model is workable but is less user-friendly and more brittle.<sup>14</sup> However, if there are no Eclipse plug-ins – or if the existing ones are buggy – we have no alternative.

A common complaint about traditional SCMs is the lack of granularity of their access control mechanism. Some tools grant read or write access in an all-or-nothing fashion: to the entire repository, or none. Others, such as subversion, will support access control to the path level: enabling a particular group of users to access only some subdirectories of the repository in a read or write fashion. Either way, the permissions being file-based, this means two restrictions:

- We cannot grant a user or user group access to certain kinds of artifacts and not others. For example, if a user has read/write access to a rule package, they can modify everything in that package, including rules, ruleflows, functions, variable sets, and so forth.
- For a given artifact, the access is all or nothing: either the user can modify both the contents and the metadata, or they cannot modify either.

If we have business users working with RS, this can be a problem.

RTS's access control mechanism is much more fine-grained than can be afforded with RS and SCM software. First of all, RTS supports role-based access, where users belong to roles, and the roles that a user has determine what *functionality* the user has access to. The tool comes with four default roles, which can be customized to the needs of the organization:

- **rtsUser**. This role corresponds to the typical rule authors. They can browse projects, create rule folders (packages) and various rule artifacts, query projects, analyze them, produce rule reports, export/import rules and decision tables to Microsoft Office documents,<sup>15</sup> and (re)deploy *existing* rule apps with the most current version of the project.
- **rtsConfigurator**. In addition to **rtsUser** functionalities, this role also has access to project and environment configuration functionalities, including: (a) managing project baselines, (b) editing project dependencies, (c) generating

---

<sup>14</sup>SCM files deal in files: they do not know about rule projects, rule packages, and the like, and thus, performing a (text) file-based reconciliation of two rule project versions is typically tedious and error prone.

<sup>15</sup>JRules includes Office plug-ins that enables to edit rules in Word documents, and decision tables in Excel spreadsheets. See Chap. 8.



ruleset archives, (d) editing ruleset extractors, (e) editing RES server configurations, and (f) editing/managing ruleapps.

- **rtsAdministrator**. In addition to **rtsConfigurator** functionalities, it includes functionalities for (a) enabling and configuration of project-level security, (b) running diagnostics on the current RTS, and (c) configuring the current installation of RTS (schema, persistence locale, message file, etc.).
- **rtsInstaller**. It has access to the installation functionalities for RTS.

Any RTS user has to be a member of one of these four groups, plus, as the case may be, other site or domain-specific groups. For example, MyWebInsurance may decide to create three groups, **policyUnderwriting**, **claimProcessing**, and **tester**, where rule authors working on policy underwriting will be members of both **rtsUser** and **policyUnderwriting**, rule authors working on claim processing will be members of both **rtsUser** and **claimProcessing**, and testers will be members of both **rtsUser** and **tester**. An RTS user who creates a rule artifact can assign the artifact to one of the groups of which she or he is a member. We discuss below the use of groups.

By default, RTS does not enforce project level security: all users of all groups can access all the projects of the repository. However, an administrator (**rtsAdministrator** group) can enforce security for a specific project, and specify which groups have access to the project. Figure 10.10 shows a screenshot of the set-up form.

Having defined the groups that can access the project, we can specify what kind of access. This is done through *permissions*. A *permission* is specified through four parameters:

- *The specific action*. Create, view (read), update, delete.
- *The value*. Can be yes or no for create, and yes, group, or no for the view, update and delete.

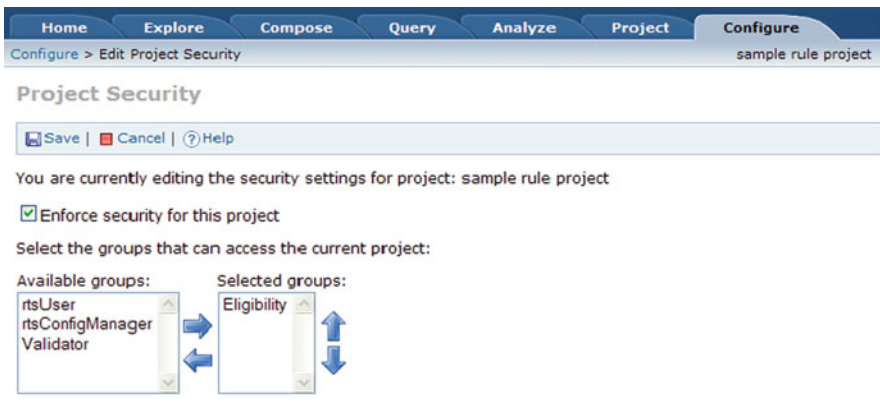


Fig. 10.10 Enforcing security for a particular project

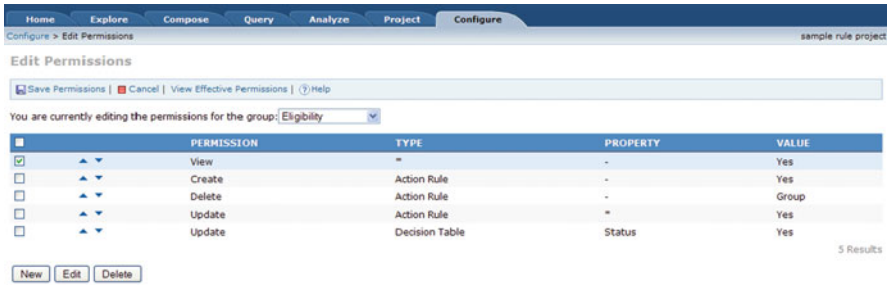


Fig. 10.11 Examples of five permissions

- *The type*. Refers to the type of project artifact for which we want to specify the permission. Types include: Action Rule, Technical Rule, Decision Table, Decision Tree, Template, etc., or the wild card (“\*”), which means all types.
- *Property*. Refers to a property of the selected type, and the wildcard (“\*”) to mean all properties of the selected type.<sup>16</sup>

The last two parameters describe the *scope* of the permission. For example, to allow the viewing of all the artifacts of a project, we define the permission: <View, Yes, \*, - >. To allow the update of the “status” attribute of a decision table, we define the permission <Update, Yes, Decision Table, Status>. To allow the deletion of action rules created by members of the same group, we write <delete, Group, Action Rule, - >. Figure 10.11 shows five permissions defined for the “Eligibility” group on our “sample rule project.” Effectively, members of this group can view all the artifacts of the project (first permission), create action rules, delete action rules created by other members of the group, update all the aspects of an action rule (content and metadata), and update the “status” attribute of decision tables – but nothing else.

How about creating a decision table, or changing the “effective date” of a decision tree? Once we enforce security for a given project, all the actions become forbidden, unless explicitly allowed. Hence, with these permissions, it is not possible to create a decision table or to update the “effective date” of a decision tree. Further, more specific permissions override less specific ones. For example, the result of the two permissions: <Update, No, Technical Rule, \*> and <Update, Yes, Technical Rule, Status> means that I cannot update any aspect of a Technical Rule, except for its status property.<sup>17</sup> The tool enables us to view the *effective permissions* based on the ones that were explicitly defined, where it shows all of the defaults and takes into account the overrides.

<sup>16</sup>When the selection of a property does not make sense or when “all” is implied by default, the property parameter takes the value “-.”

<sup>17</sup>In this case, the permission <Update, No, Technical Rule, \*> may not even be needed, as what is not explicitly allowed is not permitted by default . . . unless we have a more generic permission such as <Update, Yes, \*, - > that we want to override for Technical Rules. And so forth.

In terms of managing *concurrent access* to project artifacts, RTS automatically *write-locks* a project element whenever a user starts editing that element. The write-lock is released when the element is saved. An RTS user can also explicitly write-lock an element, and hold the lock even after the current session terminates. The write-lock can be released by the same user or by an administrator.

## 10.3 The Business Object Model

The Business Object Model embodies the business view of the application data. It represents the domain of discourse for rule authoring. A lot of the *artistry* in setting up rule authoring deals with the confection of the BOM. We start with the basics of the BOM in Sect. 10.3.1. Section 10.3.2 deals with the BOM to XOM mapping in more depth. In particular, we show how to recreate a *differentiated* business object model from a generic and stripped down *execution object model* (see discussion in Sect. 9.2.1). Because execution object models will likely evolve during the development phase and lifetime of an application.

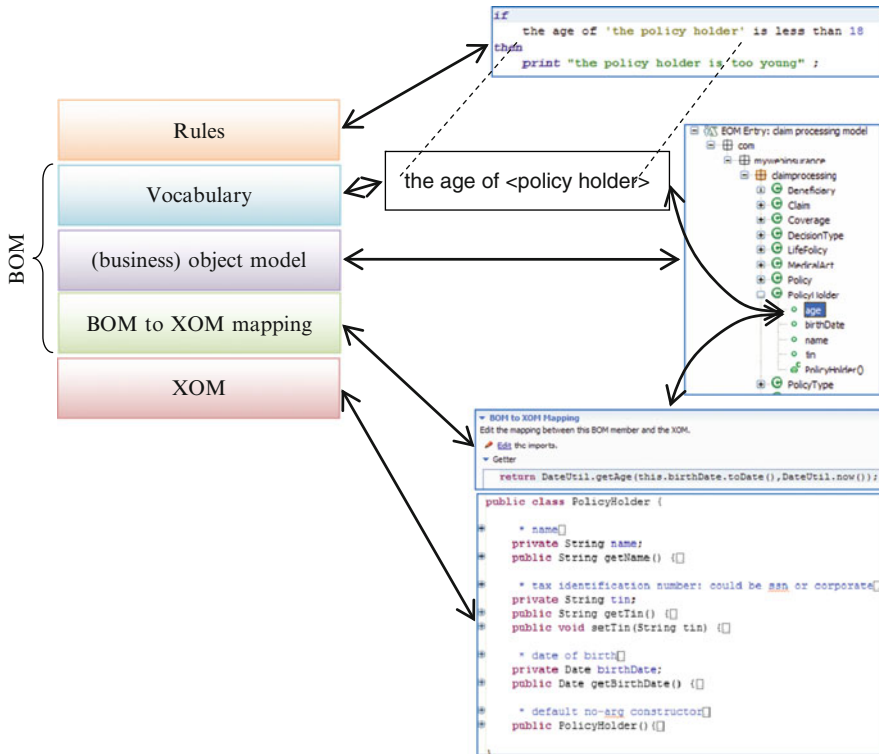
### 10.3.1 The Basics of the BOM

The Business Object Model embodies the domain of discourse for business rules. It represents the link or *bridge* between the implementation of the application data – what we call *eXecutable Object Model*, or XOM – and the business rules. Roughly speaking, the BOM consists of a three layers:

- The *vocabulary*, which is the collection of natural language-like expressions that we use to write rules, such as “the date of <the claim>” or “the age of <the policy holder>”
- The *business object model* itself, which is an object model defined in terms of packages, classes, attributes, methods, and associations
- The *BOM to XOM mapping*, which describes how the BOM maps to the actual XOM

Figure 10.12 illustrates the three layers and the relationships between them, and to the rules, on one end, and to the XOM, on the other. *Typically*, the starting point for building a BOM is a XOM, and the *typical* XOM is a set of Java classes packaged as a Java project, a class directory, or a JAR file. JRules has a utility that builds a *default* BOM based on the XOM, i.e., default values for the three layers mentioned above. We will explain what those defaults are when we explain the relationships between the various layers.

Let us start from the middle: the business object model looks like a regular object model, with nested packages (com.mywebinsurance.claimprocessing), a bunch of classes which, in turn, have attributes and methods. The **PolicyHolder** BOM class



**Fig. 10.12** The structure of the BOM and how it links natural language-like rules to (Java) application objects

appears to have *four* public data members, “age,” “birthDate,” “name,” and “tin,” for “tax identification number” which, for the case of individuals, consists of their social security number; we will worry about the constructor later.

In Java, to write a condition about the age of a **PolicyHolder**, one would write something like:

```
if (myPolicyHolder.age < 18) {...}
```

In the example of Fig. 10.12, we are using the JRules *Business Action Language* (BAL, see Sect. 10.4.2) which uses natural language-like “paraphrases” of the various elements of the BOM. With the BAL, the default reference to an attribute ATT of some class CLS is through a phrase “the {ATT} of {this}” where {ATT} refers to the name of the attribute, and {this} will be replaced by an object variable name of the type CLS. The collection of such phrases is called the *vocabulary* of the BOM. The vocabulary file is like a property file where the key refers to the BOM model element, and the value refers to the phrase template. The following are excerpts from the vocabulary file for our claim processing BOM:

```

...
com.mywebinsurance.claimprocessing.PolicyHolder.age#phrase.navigation = {age} of {this}
...
com.mywebinsurance.claimprocessing.PolicyHolder.tin#phrase.action = set the tin of {this} to {tin}
com.mywebinsurance.claimprocessing.PolicyHolder.tin#phrase.navigation = {tin} of {this}
...

```

Note that we have a single entry for the “age” attribute, corresponding to *navigation*, i.e., to read/get the value of the attribute. Because the “tin” attribute is read/write, we have both a “navigation” phrase (getter) and an “action” phrase (setter). Generally speaking, the vocabulary file will have one or two entries for every attribute or every BOM class, which correspond to reading or writing the value of the attribute. It will also have one entry for the class itself to define the term. The good news is that JRules generates the vocabulary automatically, and does a pretty good job at it, provided that the BOM – and the XOM – use recommended naming conventions; more on this in Sect. 10.3.2.

Let us now turn to the bottom half of Fig. 10.12, i.e., the relationship between the BOM and the underlying XOM. If we look more closely at the **PolicyHolder** BOM class, and compare it to the Java class, we notice three main differences:

- In the BOM **PolicyHolder** class, the attributes “name” “birthDate” and “tin” are public whereas in the Java **PolicyHolder** class, they are private.
- The Java **PolicyHolder** class has getter/setter functions for those private attributes whereas the BOM **PolicyHolder** class has no such functions.
- The BOM **PolicyHolder** class has an “age” attribute which does not appear in the Java **PolicyHolder** class.

The first two differences are related: in the process of building a BOM class for a Java class, JRules ignores the data and function members that are *not* public. However, it assumes that the Java class uses the Java Beans naming convention, and thus, if the Java class has functions that follow either of (or both) the two patterns:

```

TypeT getSomeName();
void setSomeName(TypeT value);

```

the corresponding BOM class will have a read (or write, or read/write) attribute called `someName`.

The “age” attribute illustrates the power and flexibility of the BOM to XOM mapping: we can define an attribute in the BOM class that is not physically stored in the Java class, but that is *computed on the fly* based on some actual physical attribute. Hence, if business rule authors like to think in terms of age, we can provide them with an “age” attribute in the BOM **PolicyHolder** class, as long as

we provide a way of computing it from the actual/physical data stored in the Java **PolicyHolder** class. That mapping is illustrated in Fig. 10.12 by the expression:

```
return DateUtil.getAge(this.birthdate.toDate(),DateUtil.now());
```

which is entered as the “BOM to XOM mapping” for the “getter” of the “age” attribute, where DateUtil is a custom Java utility class that manipulates dates. More powerful mappings will be discussed in Sect. 10.3.3.

The structure described so far corresponds to a single *BOM entry*. Each BOM entry consists of three distinct files corresponding to the three layers shown in Fig. 10.12. In the example of Fig. 10.12, we have a BOM entry called “claim processing model” consisting of three files:

- “claimprocessingmodel\_en.voc”. It is the vocabulary file. Notice the “\_en” suffix in the file name, which represents the locale. Indeed, we can have different vocabularies associated with a BOM based on the locale.
- “claimprocessingmodel.bom”. It represents the model itself represented in a java interface-like textual format. The following shows excerpts from that file. In addition to the Java signatures of the various attributes, methods, and constructors, the file may contain other BOM-specific properties such as domains and categories, to be explained in Sect. 10.3.5.

```
package com.mywebinsurance.claimprocessing;
...
public class PolicyHolder {
    public java.util.Date birthDate;
    public string name;
    public string tin;
    public PolicyHolder();
}
...
```

- “claimprocessingmodel.b2x”. It groups in a single file all the *custom* BOM to XOM mappings for the current model entry. We will come back to this in Sect. 10.3.3.

A typical project BOM would consist of several *BOM entries*.

We can build a BOM entry in Rule Studio (RS) in one of two ways:

- *From scratch*, by manually adding packages, classes, data, and function members using various RS (Eclipse) wizards. In this case, we have to do everything manually (1) specify the names of the various model elements (packages, classes, attributes, methods, and constructors), (2) their Java types, where applicable, (3) generate their verbalizations, and (4) specify the BOM to XOM mappings.
- *From an existing XOM*, which can be either a Java project within the same workspace, or an external Java jar file or class directory, or an *XML schema* – referred

to as a *dynamic XOM*. In either case, Rule Studio analyzes the XOM, and then creates BOM elements from that XOM. Thanks to the BOM entry creation wizard, Rule Studio will perform 90% of the job with a few selections and clicks, using default verbalizers and default BOM to XOM mappings. A user can later edit the BOM to override some defaults or add virtual elements to the BOM, such as the “age” attribute mentioned above (more on that in Sect. 10.3.3).

Which method is preferred? Clearly, if you already have the target XOM, then you should build the BOM (entry) from the existing XOM. However, there are situations where one would want to start authoring rules before the underlying implementation code has been completed, and there one would build the BOM from scratch. These issues will be discussed in Sect. 10.4.2.

### 10.3.2 Verbalization

Verbalization is the process of assigning a term or phrase to a BOM model element, e.g., as in assigning the phrase “the {age} of {this}” to the attribute “age” of the BOM class **PolicyHolder**. Rule Studio has a default “verbalizer” that can verbalize the elements of an entire BOM entry, in one batch, or single BOM elements (classes, attributes, and methods), one by one. Developers can override the default verbalization for a given model element. Default verbalization follows simple rules that we explain below.

First, we look at the verbalization of identifiers. The rules are illustrated in Table 10.1.

In particular, if we adopt the Java nomenclature for spelling multi-word identifiers – capitalizing the first letter of every word, with the possible exception of the first – the verbalizer will actually separate out the individual words.

Consider now the verbalization of attributes. First, the name of the attribute is verbalized to generate what JRules calls a *subject*, which is then used to generate a *navigation phrase* (a getter expression) and an *action phrase* (a setter action). Table 10.2 illustrates the default verbalizations for non-boolean attributes.

Notice that the name of the attribute in the navigation and action phrases appears in a template form (i.e., between curly brackets) between it is editable. For example, a BOM developer may choose to use the term “date of birth” instead of “birth date,” and adjust the plural to “dates of birth” (as opposed to the default “date of births”). The figure across shows the wizard for editing terms. This wizard knows a bit

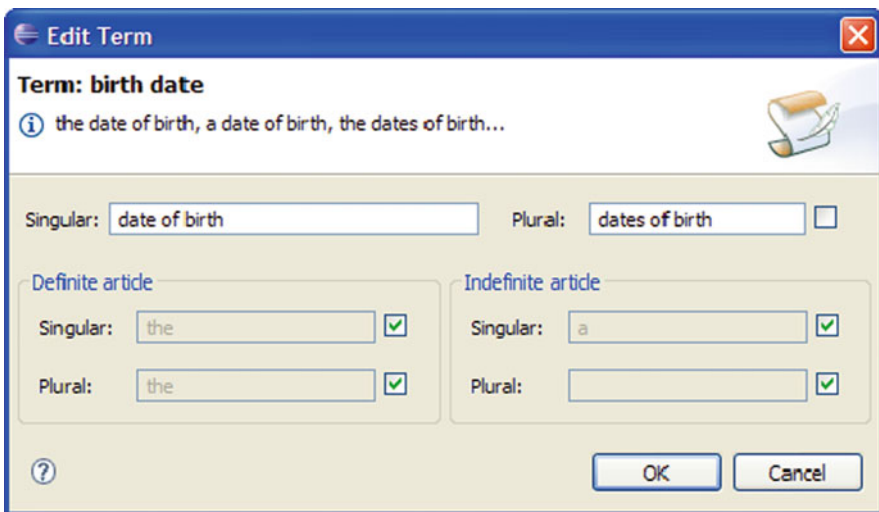
**Table 10.1** Verbalization of identifiers

Identifier	Its default verbalization
lowercasename	lowercasename
UPPERCASENAME	UPPERCASENAME
UpperFirstLetter	upper first letter
upperFirstLetter	upper first letter
lowerSECOND	lower SECOND

**Table 10.2** Verbalization of non-boolean attributes

Attribute	Verbalization (subject)	Navigation/action phrases	Examples
Name	name	{name} of {this}	the name of <b>"my policy holder"</b>
		set the name of {this} to {name}	set the name of <b>"my policy holder"</b> to <b>"John"</b> ;
birthDate	birth date	{birth date} of {this}	the birth date of <b>"my policy holder"</b>
		set the birth date of {this} to {birth date}	... set the birth date of <b>"my policy holder"</b> to <b>13/4/1991</b> ;

about the English language so that the indefinite singular form for “age” is “an age” and not “a age,” and the plural of “bankruptcy” is “bankruptcies” and not “bankruptcys.”



The verbalization of boolean attributes is different and is illustrated in Table 10.3.

The last example shows an instance where the default verbalization does not work: both the navigation and action phrases need to be edited to get rid of the extra “is” (underlined in the table).

Finally, JRules enables us also to verbalize methods, as those may be used within conditions – those that return values – or actions of rules – those that return void. The parameters of such functions then become data prompts for the rule author. Assume that our class **PolicyHolder** has a method with signature:

```
void addAccident(Date d, Responsibility resp);
```



**Table 10.3** Verbalization of boolean attributes

Attribute	Verbalization (subject)	Navigation/action phrases	Examples
approved	approved	{this} is approved make it {approved} that {this} is approved	"my claim" is approved make it true that "my claim" is approved;
isRejected	rejected	{this} is rejected make it {rejected} that {this} is rejected	my claim" is rejected make it true that "my claim" is rejected;
hasBeenPaid	has been paid	{this} <u>is</u> has been paid make it {has been paid} that {this} <u>is</u> has been paid	"my claim" <u>is</u> has been paid make it true that "my claim" <u>is</u> has been paid;

where **Responsibility** is an enumerated type with the values AT\_FAULT and NO\_FAULT. The default verbalization for this method is the ugly:

```
{this}.addAccident({0},{1})
```

Notice here {0} and {1} that stand for the first and second positional parameters of the function (date and responsibility of the accident, respectively), and that become data prompts for the rule authors using this action. In a rule, this action would appear as follows:

```
'my policy holder'.addAccident(21/8/2009, "AT_FAULT");
```

Not exactly business user friendly. We can change the verbalization template to the following:

```
add a {1} accident to {this} on {0}
```

where {1} stands for the *second* parameter of the function,<sup>18</sup> i.e., the responsibility, {this} stands for the policy holder, and {0} stands for the *first* positional parameter, i.e., the date. This action will then appear in a rule as follows:

```
add a "AT_FAULT" accident to 'my policy holder' on 21/8/2009;
```

<sup>18</sup>We can all thank Java, C++, C, or the assembly language for this numbering convention, depending on how far back you want to go ☺.

This is part of the artistry that goes into setting up the verbalizations for the BOM elements, and that can make rule authoring – and reading – more intuitive and more business friendly. The person responsible for configuring the BOM (business analyst or rule developer) needs to be familiar with the “vocabulary” and phrasing used by business to mimic it as closely as possible in the BOM.

### 10.3.3 BOM to XOM Mapping

We introduced the BOM to XOM mapping in Sect. 10.3.1 and explained how it acts as a bridge between the business view of the data (BOM) and the actual implementation of the business data (XOM). Recall also that Rule Studio enables us to create a *BOM entry* from a XOM such as a Java project, a Java jar file, or an XML schema. The BOM creation utility uses a *default* BOM to XOM mapping, which we can override or customize. We will first talk about the default mapping, and then talk about custom BOM to XOM mappings.

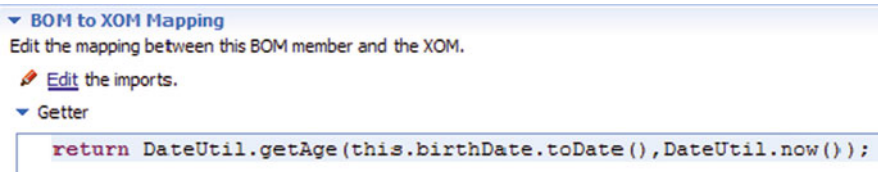
Table 10.4 shows the main default mappings for a Java XOM. Anything that is not *public* does not appear in the BOM. When we build a BOM entry from a specific XOM, Rule Studio does not store these default mapping in the B2X file (BOM to XOM), which starts out empty. The B2X file is *only* used to store custom mappings.

We now explore some typical uses for the BOM to XOM mapping. Our first example of Fig. 10.12 showed one case of custom BOM to XOM mapping. In that example, we added an “age” attribute to the **PolicyHolder** BOM class, that did not exist in the Java **PolicyHolder** class, but that was *computed* from the “birthDate” attribute. Generally speaking, the Rule Studio BOM editors enables us to manually add data and function members to a BOM class that have no equivalent in the corresponding XOM, provided that we supply the BOM to XOM mapping for that

**Table 10.4** Default Java XOM to BOM mappings

Java construct	BOM construct
Package	Package
Public class	Class
Public interface	Interface
public Type attName;	public Type attName;
public Type getAttName()	public readonly Type attName;
with no corresponding setter	
public void setAttName(Type arg)	public writeonly Type attName;
with no corresponding getter	
A getter/setter pair get/setAttName(Type a)	public Type attName;
A non-getter/setter public function	A similar public function
Public constructor	constructor
An “extends” relationship between Java classes or interfaces	An “extends” relationship between corresponding Java classes or interfaces
An “implements” relationship between a class and an interface	An “implements” relationship between the corresponding BOM class/interface

data or function member. We call those *virtual* data or function members. For a *virtual data member*, the BOM to XOM editor enables us to specify a “getter” and/or a “setter” expression, depending on whether the data member is readonly, writeonly, or read/write. Below, we reproduce parts of the BOM to XOM editor for the “age” attribute, shown in Fig. 10.12. This fragment is part of the BOM class data member editor, for the attribute “age.”



In this case, because the “age” attribute is computed, it is read only, and we only specify the getter.

### 10.3.3.1 Virtual Functions


We can also specify *virtual* functions, i.e., functions that exist only in the BOM. If such a function returns a non-void value, it will appear in the condition part; if it returns a void, it will appear in the action part. Assume now that a claim is only eligible if it has been filed less than 180 days after the expense was incurred, for an ongoing policy, and less than 90 days after the expiration of the policy, for an expired policy. We could add a boolean function to the BOM class **Claim** with the following signature:

```
boolean filedMoreThanNDaysAfterDate(int nDays, Date aDate);
```

We can then specify how to compute such a function in the BOM to XOM mapping, as shown below. The variables `nDays` and `aDate` refer to the arguments of the function, with types `int` and `java.util.Date`, respectively. The pseudo-variable `this` refers to a **Claim**, and the dot reference `this.date`, to the date of the claim.<sup>19</sup>

<sup>19</sup>So which object model do we refer to in the BOM to XOM mapping? Logically, this should be the XOM, as we are showing how BOM elements map to XOM elements. In practice, we can refer to the BOM, and to any Java object model referenced in the rule project, included but not limited to the XOM. Thus to access the attribute “date” of **Claim**, I can use either “myClaim.date” or “myClaim.getDate().”

▼ **BOM to XOM Mapping**  
 Edit the mapping between this BOM member and the XOM.

 [Edit the imports.](#)

▼ **Body**

```
Calendar calendar = Calendar.getInstance();
calendar.setTime(aDate);
calendar.add(Calendar.DAY_OF_YEAR, nDays);
return this.date.toDate().after(calendar.time);
```

This function can then be verbalized as follows:

```
{this} was filed more than {0} days after {1}
```

And used in a rule:

**Business Rule: claim date - expiration policy**

► General Information    ► Category Filter

► Documentation

Code

```
if
  'the claim' was filed more than 90 days after the end date of the policy of 'the claim'
then
  reject 'the claim' ;
  log that this rule has fired on 'the claim' with message "Filed more than 90 days after end of policy" ;
```

In this rule, `{this}` was substituted by the variable **“the claim,”** the first argument `{0}` was set to **90**, and the second argument `{1}`, which is a date, was replaced by the end date of **“the claim.”** Incidentally, the second rule action (log that this rule has fired on ...) is itself a virtual function of the BOM class **Claim** with the signature:


```
void logRuleFiringWithMessage(String message)
```

The verbalization:

```
log that this rule has fired on {this} with message {0}
```

and the BOM to XOM mapping:

▼ **BOM to XOM Mapping**  
 Edit the mapping between this BOM member and the XOM.

 [Edit the imports.](#)

▼ **Body**

```
Object[] args= {this};
String claimString = (String)context.invokeFunction("printClaim",args);
String fullMessage = "Rule [" + ?instance.ruleName + "] fired on " + claimString + " with message: " + message;
this.messageList.add(fullMessage);
```

This mapping is a bit more complex, and illustrates some advanced features of the BOM to XOM mapping and of the IRL language. First, rule `{this}` refers to a claim. In the first line, we are initializing an array of **Object**'s with `{this}`

(the claim). In the second line, we are building the string that represents a claim by making what looks like a reflective call to some one-argument function called “printClaim” (sort of a custom `toString()` method), and that is exactly what it is: the method `invokeFunction(String functionName, Object[] args)` is an **IlrContext** (the class that represents rule engines) method that invokes an *IRL function* called `functionName` with the arguments `args`. We described IRL functions briefly in Sect. 10.2.1 as macro-like functions defined within rule projects. Such functions can be used within the action parts of IRL rules (covered in Sect. 11.2.1), function task bodies (see Sect. 11.3.2), within the initial actions and final actions of ruleflow tasks (Sect. 11.3.2), within *other* IRL functions, or within BOM to XOM mappings. In all the places *but* BOM to XOM mappings, these functions would be invoked normally, as in “`printClaim(my_claim); .`” However, within the BOM to XOM mapping, they need to be invoked reflectively.<sup>20</sup> This code fragment shows also the use of two predefined IRL variables: `context`, which refers to the engine currently executing this piece of code, and `?instance`, which refers to the rule instance currently executing.<sup>21</sup> The latter enables us to access the rule name, the tuple of objects for which the rule fired, the priority of the rule instance on the agenda, etc., which makes it possible to write generic – and detailed – rule logging capabilities, as the rule above illustrates.

### 10.3.3.2 Virtual Classes

We now turn our attention to *virtual classes*. A *virtual class* is a class that exists only in the BOM with no direct equivalent in the XOM. As with virtual attributes and methods, we only need to specify what XOM (Java) class this BOM class maps to. First, let us explore a scenario where you would want to create a virtual class, and then we will show how to define such a virtual class.

We have shown in Sect. 9.2.1 the requirements that we place on BOM and on XOM. In particular, we identified *specificity* as a desirable property of the BOM and *genericity* as a desirable property of the XOM. Assume that our business people created the model shown in Fig. 10.13 on paper, before handing it over to IT to implement. This model distinguishes between an AUTO policy and a Health policy and makes a distinction between a medical claim and a car repair claim.

Upon closer inspection of the attributes of the various classes (not shown in this figure), an object designer, or a data architect, might find this model unnecessarily differentiated, and might decide to implement the model shown in Fig. 10.14, instead. In this model, both kinds of policies are represented by the same class

<sup>20</sup>The BOM to XOM mapping is used by the engine during run-time. Further, IRL functions can refer to BOM virtual functions. If we allow IRL functions to be called normally within the BOM to XOM mapping, we could end up with an arbitrarily long – and potentially circular – translation sequence from BOM to XOM.

<sup>21</sup>The Java type for `?instance` is `ilog.rules.engine.IlrRuleInstance`.

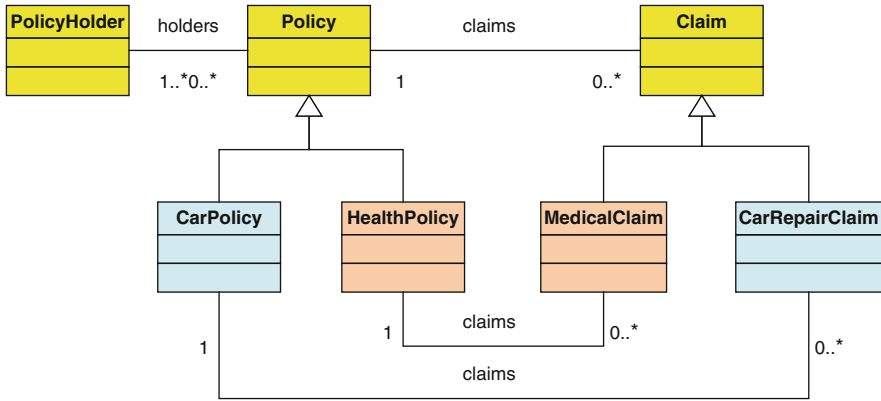


Fig. 10.13 The business view of the application data (“BOM on paper”)

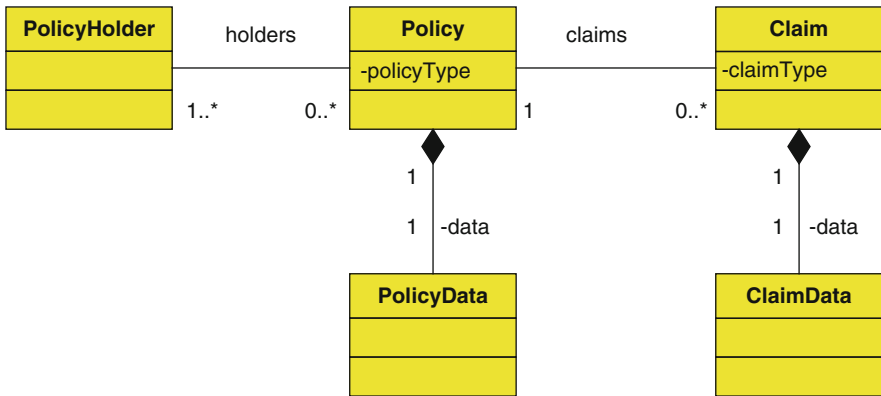
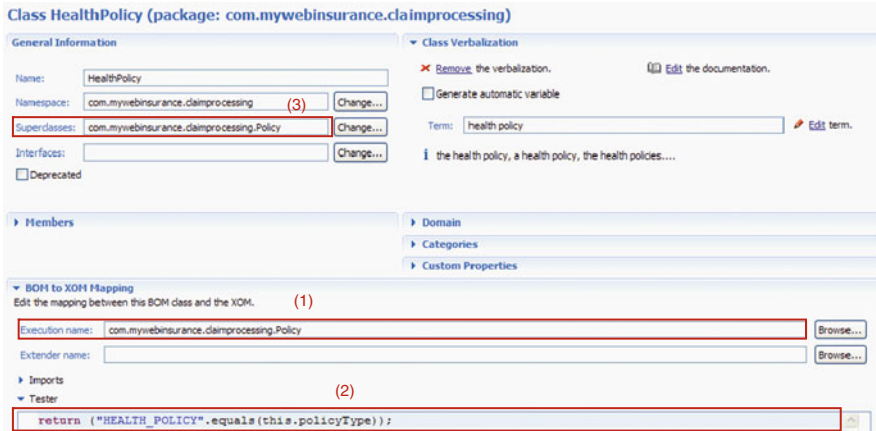


Fig. 10.14 The actual implemented XOM

**Policy**, which now has an attribute called `policyType`. Common policy attributes are represented in the class **Policy** itself, whereas policy type-specific attributes are represented in a **PolicyData** object. Idem for the **Claim** class.

If we create a BOM from this XOM, rule authors will not have the concept of a “health policy” or of a “car policy,” but they can talk about a “policy” whose `policyType` equals “HEALTH\_POLICY” or “CAR\_POLICY.” Virtual classes allow us to (re)create the BOM classes **HealthPolicy** and **CarPolicy** even though there is a single underlying XOM class, **Policy**. Figure 10.15 shows the BOM editor for classes. We are defining the class **HealthPolicy** as a class from the package `com.mywebinsurance`. `claimprocessing` and specifying its execution name (see (1) marker on figure) as `com.mywebinsurance.claimprocessing`.



**Fig. 10.15** BOM editor for BOM classes. To define a virtual BOM class, we need to (1) specify its execution name (i.e., corresponding XOM class), (2) specify what conditions instances of the XOM class need to satisfy (tester), and (3) specify its BOM superclasses

**Policy.** However, a **HealthPolicy** is not *any* **Policy**: it is a policy whose “tester” expression (see (2) marker) returns true:

```
return ("HEALTH_POLICY".equals(this.policyType));
```

where the variable `this` refers to a **Policy**.

Having defined the class **HealthPolicy**, we now wish to use it within rules, and test its data members, such as `startDate`, `endDate`, like any regular **Policy**, i.e., we wish to *inherit* the data and function members of the ... BOM class **Policy**. To do that, we need to specify the *BOM class* **Policy** as a *superclass* of **HealthPolicy** (see marker (3)). Figure 10.16 illustrates the required steps to define the virtual class **HealthPolicy**.

Note that, unlike with Java classes, BOM classes do support multiple inheritance: I can specify two or more BOM superclasses for any given BOM class. We will come back to this feature when we talk about best practices.

### 10.3.3.3 Dynamic XOM

Finally, we talk about *dynamic XOMs*, and more specifically, the XSD-based XOM. In short, JRules enables us to write and execute rules about XML data. This means two things:

- *At rule definition time.* The BOM is defined from an *XML schema* as opposed to a set of Java classes.
- *At rule execution time.* The rule engine can manipulate a generic and efficient run-time representation of XML data through the same object-based API that is used to access Java objects. This object-based API abstracts away the way objects are created, and their attributes read and set.

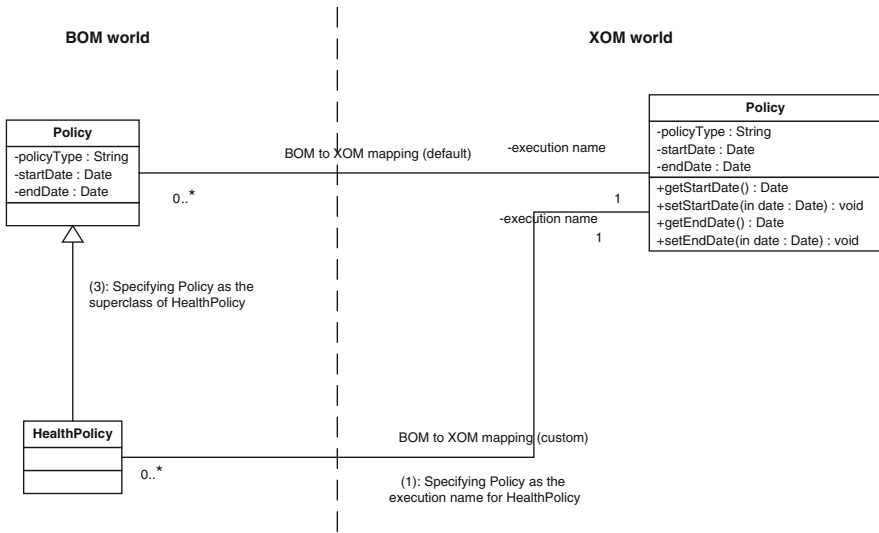


Fig. 10.16 Steps to specialize a BOM class with a virtual BOM class

Table 10.5 The basics of the XSD to BOM mapping

XSD	BOM element
Complex type	BOM class
XSD element or attribute	Read/write BOM attribute
An XSD element with maxOccurs > 1	A java.util.Vector attribute with a collection domain of the element type (i.e., a multi-valued attribute)
Built-in XSD simple types	Corresponding java types
Extension and restriction	BOM class inheritance
Restricted simple types	Corresponding Java type with a literals domain

Table 10.5 shows the basics of the XSD-based BOM to XOM mapping. Roughly speaking, XSD’s complex types map to classes, where the type’s <element>s and attributes map to read/write BOM class data members, and the built-in XSD types map to the corresponding Java types. If an <element> has a maxOccurs higher than one, than the element is mapped to a java.util.Vector, with a BOM annotation that specifies the element type.

Table 10.6 shows excerpts from an XSD, and the corresponding excerpts from the BOM classes. For the sake of presentation, in the BOM column, the package names were omitted from the class names, with an ellipsis shown instead (“...”). The reader will notice that all BOM classes in this case inherit from the default **IlrXmlObject**, which is the actual implementation class for XML data (more on this below). The XSD types **string**, **float**, and **int** map to the Java types **java.lang.String**, **float**, and **int**, respectively. The XSD **date** type maps to the ILOG type **ilog.rules.xml.types.IlrDate**, which knows how to convert back and forth to a **java.util.Date**. Notice also how the Policy XSD



**Table 10.6** The basics of the XSD to BOM mapping

Excerpts of an XSD schema	Excerpts of the corresponding BOM
<pre> &lt;xs:complexType   name="PolicyHolder"&gt;   &lt;xs:sequence&gt;     &lt;xs:element name="name"       type="xs:string"/&gt;     &lt;xs:element name="tin"       type="xs:string"/&gt;     ...   &lt;/xs:sequence&gt; &lt;/xs:complexType&gt;  &lt;xs:complexType name="Coverage"&gt;   &lt;xs:sequence&gt;     &lt;xs:element name="procedure"       type="Procedure"/&gt;     &lt;xs:element name="deductible"       type="xs:float"/&gt;     &lt;xs:element name="yearlyCap"       type="xs:float"/&gt;     &lt;xs:element name="totalToDate"       type="xs:float"/&gt;   &lt;/xs:sequence&gt; &lt;/xs:complexType&gt;  &lt;xs:complexType name="Policy"&gt;   &lt;xs:sequence&gt;     &lt;xs:element name="number"       type="xs:int"/&gt;     &lt;xs:element name="startDate"       type="xs:date"/&gt;     &lt;xs:element name="endDate"       type="xs:date"/&gt;     &lt;xs:element name="policyHolder"       type="PolicyHolder"/&gt;     &lt;xs:element name="insured"       type="PolicyHolder"       minOccurs="0"       maxOccurs="unbounded"/&gt;     &lt;xs:element name="coverage"       type="Coverage"       minOccurs="1"       maxOccurs="unbounded"/&gt;   &lt;/xs:sequence&gt; &lt;/xs:complexType&gt; </pre>	<pre> public class PolicyHolder extends   ilog.rules.xml.IlrXmlObject   // some custom properties   property ... {   ... }  public class Coverage extends   ilog.rules.xml.IlrXmlObject   // some custom properties   property ... {   ... }  public class Policy extends   ilog.rules.xml.IlrXmlObject   // some custom properties   property ... {public int number   property // custom properties    public ...IlrDate startDate   property // custom properties    public ...IlrDate endDate   property // custom properties    public ...PolicyHolder policyHolder   property // custom properties    public ...Vector insuredList   domain 0,* class ...PolicyHolder   property // custom properties    public ...Vector coverageList   domain 1,* class ...Coverage   property // custom properties    ... // plus some other stuff } </pre>

elements `insured` and `coverages` mapped to `java.util.Vector` instance variables, with a *domain* that is of the appropriate element type (`PolicyHolder` and `Coverage`, respectively); we will talk about *domains* in Sect. 10.3.5 of this chapter.<sup>22</sup>

<sup>22</sup>This feature of JRules has been around since 2002, i.e. prior to JDK 1.5 (which came out in the fall of 2004) which introduced support for genericity, and the handling of collections has remained unchanged for backward compatibility reasons.

Notice that the classes **PolicyHolder**, **Coverage**, and **Policy** exist only in the *BOM*; during run-time, the underlying XML data – that conforms to the XML schema – will be represented by instances of the **IlrXmlObject** class, regardless of the BOM class. Indeed, unlike JAX-RPC, JAXB, or JAX-WS frameworks, JRules does *not* generate Java classes for the corresponding XML schema complex types; we can think of the XSD-mapped classes as *virtual classes* with an execution class **IlrXmlObject**.

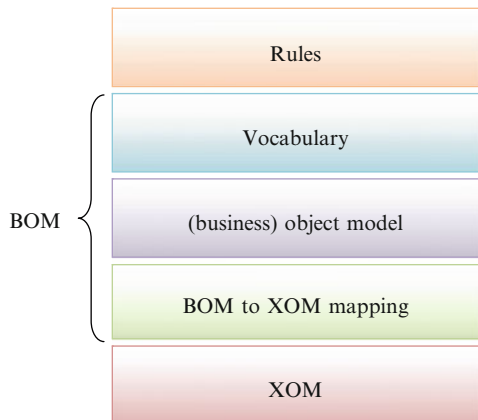
For now, suffice it to say that if your application manipulates business data that comes in an exotic, self-describing, and evolving data format, you too could develop support for your exotic format, at both rule definition time (BOM to XOM mapping) and at rule execution time.

### 10.3.4 Refactoring

To use a common *cliché*, the only constant in today’s business applications is change. The BOM has a *pure* layered architecture where each layer depends only on the layer just below it (see Fig. 10.17). Eclipse’s refactoring functionality has been extended by the JRules plug-in to propagate changes in a given layer to the layer just above it. In this section, we review the different kinds of changes, and the available functionalities to propagate them through the layered structure of Fig. 10.17.

#### 10.3.4.1 Changes to the XOM

A stable XOM is an elusive goal in many rule projects, especially ones where both the business layer (i.e., business entities) and the business rules are within scope of the development or modernization effort. We might as well live with it, considering



**Fig. 10.17** The BOM’s layered architecture enhances the rules’ resilience to change

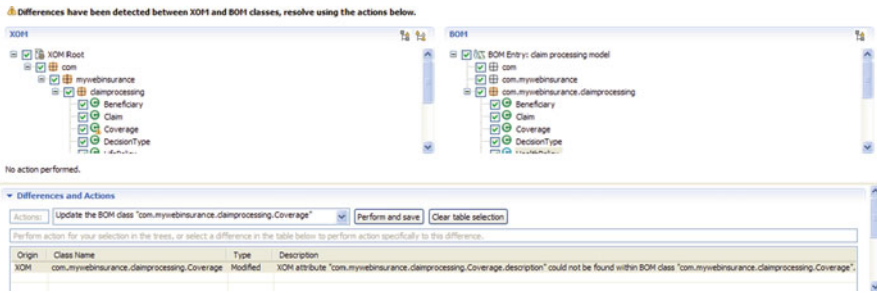


Fig. 10.18 An example of the “Update BOM” wizard, for BOMs created from a XOM

that Rule Studio enables us to cope with the most common situations. If we build a BOM from a XOM (see Sects. 10.3.1 and 10.3.3), whenever we change the XOM, we can ask the tool to update the BOM accordingly. This is done by selecting the corresponding BOM entry, and selecting the action “BOM Update” in the contextual menu. This will compare the current state of the XOM to the current state of the BOM, and (a) identify the differences, and (b) propose actions to bridge those differences. Figure 10.18 shows the interface of Rule Studio’s BOM – XOM synchronization wizard. The top part shows, side by side, the XOM and the BOM. The class Coverage (left-hand side) has a warning sign, which indicates that it is not consistent with the corresponding BOM version. The list of differences is shown in the lower pane – here just one indicating that the attribute “description” of the XOM class Coverage could not be found in the corresponding BOM class. For each difference identified, the tool proposes one or more (generally two) actions that can be performed to bridge the difference. In this case, the tool proposes to update the BOM class – and thus, add the attribute “description” to it.

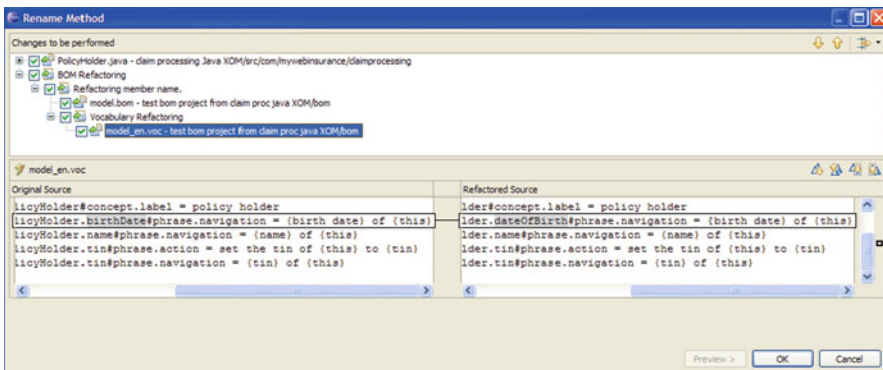
Let us now consider the typical changes to the XOM, and how they could be handled:

1. *Additions.* If we add XOM elements, when we re-synchronize the BOM with the XOM, the tool offers to propagate those additions to the BOM. This works for both the addition of XOM classes and the addition of function or data members to existing XOM classes. This will have no effect on existing BOM or rules.
2. *Removals.* If we remove a XOM element, be it a class or a member of a class, the BOM update wizard will note that the corresponding BOM element has been “orphaned” and will offer to either delete it or to “deprecate it.” We could also do nothing. If we do nothing, Rule Studio will complain that the now-orphaned BOM element has no XOM corresponding element, in which case, we should use the BOM to XOM mapping to map it to *other* existing XOM elements. Deprecating it means that we set the property “deprecated” of the BOM element to true, which will flag rules that use it with “deprecated” warnings.<sup>23</sup> This will also stop making the element available in the pull-down lists or code completion

<sup>23</sup>You may need to “clean” the project for the warnings to show up.

feature of the rule editor. We still have to use the BOM to XOM mapping to map the orphaned BOM element to something. If we delete the BOM element . . . we better make sure that the element is not used in any rule, first.<sup>24</sup>

3. *Renaming*. If we rename a XOM element, depending on where it is performed, the changed can be propagated automatically throughout the BOM, or will have to be done manually:
  - *Renaming done through Eclipse’s refactoring menu*. This is possible if the XOM is a Java project included in the same workspace as the project containing the BOM. In this case, Eclipse’s refactoring functionality will propagate the renaming to (a) the BOM, by renaming the corresponding BOM element, and (b) the vocabulary, by changing the *corresponding key in the vocabulary file*, but leaving the *value*, i.e., the actual verbalization, unchanged. Figure 10.19 shows an example of rename refactoring, which is propagated all the way to the key part of the vocabulary file.
  - *Renaming done manually*. This would be the case for any XOM that is *not* a Java project within the same workspace, such as an XSD XOM, or a Java XOM supplied as a jar file, or a bunch of .class files. In this case, the “BOM Update” facility will note an addition to the XOM, corresponding to the new name, and a removal from the XOM, corresponding to the old name. We can handle it either by renaming the BOM element manually, which will propagate it to the vocabulary (see below, BOM changes), or by using the BOM to XOM mapping to map the BOM element with the old name to the XOM element with the new name.



**Fig. 10.19** Renaming a Java (XOM) method through Eclipse’s refactor will propagate the change to the .bom and vocabulary file, without affecting the rules that use the member

<sup>24</sup>Rule queries and Eclipse search functionality enable us to ascertain that. We could also first deprecate the element then see if any rules generate “deprecated” warnings, and if none do, we could then *safely* delete it.

4. *More complex refactorings.* In this case, Eclipse’s refactor menu will not do the trick, as the tool gets confused. Instead, we should use the “BOM Update” functionality to propagate some of the changes, and fix the rest manually.<sup>25</sup>
5. *Non-semantics preserving changes.* If we change the type signature of a method (e.g., number or types of parameters), then we use “BOM Update,” and it should be treated as an addition and a removal. In some cases, it may be appropriate to map the old BOM method to the new XOM method using the BOM to XOM mapping. For example, if the new XOM method has an additional parameter, perhaps that parameter has a reasonable default value, and we can keep the old BOM method, but as a virtual method. This would need to be handled on a case-by-case basis.

The important thing to note from this analysis is that in any of the above change scenarios, the rules are shielded by the changes to the XOM, as the change is absorbed in the various intervening layers between the XOM and the rules.

#### 10.3.4.2 Changes to the BOM

Most changes to the BOM originate from the XOM and were discussed above. The changes that do originate from the BOM correspond to the addition, modification, or removal of virtual BOM elements:

1. *Additions of virtual BOM elements.* This is, for the most part, non-problematic, except in those cases where the new BOM element has the same verbalization as an existing BOM element.
2. *Modification of a virtual BOM element.* Renamings have no effect as the vocabulary absorbs the change. As mentioned earlier and illustrated in Fig. 10.19, the vocabulary file assigns BOM element phrases that will appear in rules, in a key = value. Renaming the BOM element will only modify the key part, as illustrated in Fig. 10.19, leaving the verbalization – and the rules – unchanged. More substantial changes can break existing rules. For example, if we modify the signature of a virtual method, its verbalization will need to change to account for the additional/fewer parameters, which will be propagated to the rules that use it (see below). This, in turn, will break those rules.
3. *Removal.* See the discussion above regarding removal, and removal versus deprecation.

---

<sup>25</sup>For example, if we move a data member and its accessors up the hierarchy of classes, the right thing to do would be to move the corresponding public data member up the BOM hierarchy. However, the tool cannot do that on its own: the “BOM Update” will enable us to add the member to the superclass, but will not remove it from the original class, and will not complain about it since it does have a XOM equivalent. However, if the data member is used in a rule, the rule editor will complain about an “ambiguous sentence,” meaning that two data members have the same verbalization.

### 10.3.4.3 Changes to the Vocabulary

If we make a change to the verbalization of a BOM element, *and save* the BOM, Rule Studio will prompt the user to confirm the verbalization modification as it may affect existing rules. If the user accepts to proceed, a refactoring menu is presented to the user, showing the various rules that use that BOM element/its verbalization with the before and after text. The user has then the options of (1) rejecting the change (save), or (2) accepting it and propagating it to all concerned rules or a subset thereof.

Notice that the propagation of verbalization changes to rules *will only work if the rule is syntactically correct to start with*. If the rule is wrong, the result can be unpredictable: in the best case, the change will not be propagated and the rule will remain wrong. In the worst case, you lose parts of the rule text.

## 10.3.5 Enhancing the Rule Authoring Experience

JRules offers a set of bells and whistles that make life easier for rule authors. We will present two important ones, *categories* and *domains*.

### 10.3.5.1 Categories

Any self-respecting BOM will have dozens of classes and hundreds if not thousands of members. While editing rules, the number of drop-downs that are provided to rule authors is likely to be overwhelming. However, any given rule will typically address only one facet of the data. In our claim processing example, a rule about the *eligibility* of a given *procedure* will be concerned with coverages attached to the policy, and not about personal or credit data about the policy holder. JRules offers a way to *filter* those BOM elements that show up in rule editors based on the *categories* assigned to the BOM elements and the *categories* assigned to rules. Figure 10.20 illustrates the relationship between BOM elements, categories, and rules.

Categories are defined at the project level. By default, a rule project starts with a single category “Any,” and all BOM elements and rules are assigned the category “Any.” Thus, by default, rules will pull in all the BOM elements. Assume that we add the categories “Claim eligibility” and “Claim adjudication” to the project. We can then assign the category “Claim eligibility” to those BOM elements that we think are relevant to assessing the eligibility of a claim, and “Claim adjudication” to those BOM elements that we think are relevant to adjudicating the claim. Naturally, some BOM elements will be relevant to both areas, and can have both categories. In this case, the class Claim is relevant to both and will have both categories. Figure 10.21 shows the Rule Studio wizard for assigning categories to a BOM class.



Fig. 10.20 The relationship between BOM elements, categories, and rules

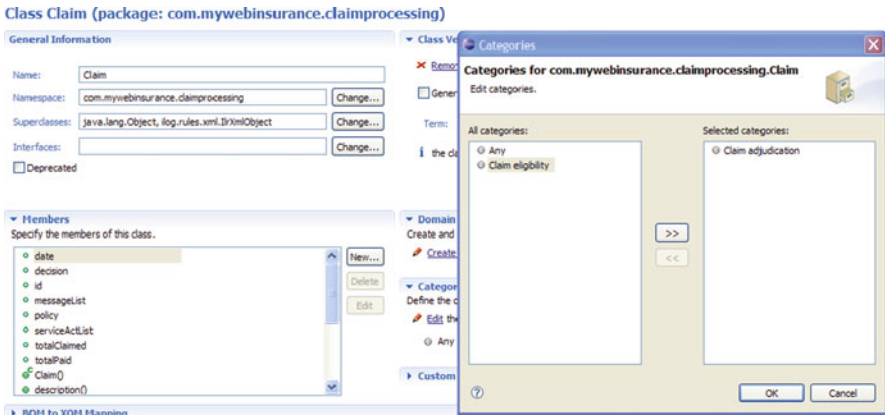


Fig. 10.21 Assigning categories to a BOM class

Table 10.7 Semantics of category filters

BOM element	BE_1	BE_2	BE_3	BE_4	BE_5
Categories	Eligibility	Adjudication	Eligibility,	Any	
Rule	Category filter		Adjudication		
Rule_1	Eligibility	Visible	Not	Visible	Visible
Rule_2	Adjudication	Not	Visible	Visible	Visible
Rule_3	Eligibility,	Visible	Visible	Visible	Visible
	Adjudication				
Rule_4	Any	Visible	Visible	Visible	Visible
					Not

When we define a rule, we can also edit its *category filter*, which uses a similar wizard to that of Fig. 10.21 to assign one or more categories to the rule. In so doing, we determine the subset of BOM elements that are selectable – and thus usable – in the rule. The default category “Any” plays the role of a wildcard: a BOM element with category “Any” is available to all rules, regardless of their category filters, and a rule with category filter “Any” will have access to the entire BOM.<sup>26</sup> Table 10.7 illustrates the semantics of the category filter.

<sup>26</sup>It is technically possible to assign no category to a BOM element, which makes it unavailable for rule authoring, altogether.

Notice that when we assign a category to a class, it is not “inherited” by members of the class. This may sound counter-intuitive but in the above example, while the Claim class itself is relevant to both claim eligibility and adjudication, some of its members will be relevant to only eligibility while others will be relevant to only adjudication.

### 10.3.5.2 Domains

The BOM uses Java types, regardless of its origin, be it a Java XOM or an XSD XOM. Pre-Java 5, if we wanted to represent the state component of a US address, say, we had two choices: (1) use the Java String type for the java attribute, but then control what values can be assigned through the input forms, or (2) use what is called the (pre-JDK 5) Java enumeration pattern with a class **State** as illustrated below.

```
public class State {
    private String stateCode;
    private String stateName;

    // getters
    ...

    private State(String code, String name){
        stateCode = code;
        stateName = name;
    }

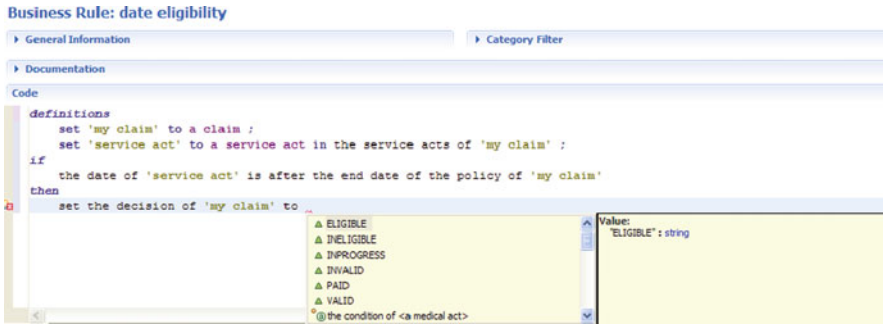
    public final static State AL = new State("AL", "ALABAMA");
    ...
    // 50 states later
    public final static State WY = new State("WY", "WYMONING");
}
```

JRules enables us to restrict the set of values that a BOM attribute, a BOM function parameter, or a BOM function return value can take using *domains*. With domains, when a rule author is prompted to enter a value for that attribute/parameter/return value, they will get a dropdown list of the domain values, as illustrated in Fig. 10.22. Here, the “decision” attribute of a Claim, a String, has been restricted to the values shown using a *literal domain*, i.e., a domain whose values are explicitly enumerated.

Generally speaking, JRules enables us to define five kinds of domains:

1. *Literal domains*. In this case, the values are enumerated. This works for scalar types and for String. This will also work for actual Java 5 (and beyond) enumerations: if a BOM attribute or function parameter or return value is a Java enum, then it will have a literal domain consisting of the elements of the enumeration.





**Fig. 10.22** The “decision” attribute of Claim has type String, but with a domain {“ELIGIBLE”, “INELIGIBLE”, “INPROGRESS”, “INVALID”, “PAID”, “VALID”}

2. *Bounded domains.* For numerical types, where we can specify a range of values.
3. *Static references.* This corresponds to our State example above. If our Java class uses the enumeration pattern illustrated below, Rule Studio will *automatically* create a domain that includes *all* of the public static final data members for each attribute, parameter or return value. We can later edit that domain to remove values. For example, with the State class above, any BOM attribute, parameter, or function return value will have a domain consisting of all the enumerated states. That domain can later be edited to remove or put back states.
4. *Collection domains.* If a BOM data member or a function parameter or a function return value is a Java collection (Vector, ArrayList, List, etc.), we can define a *collection domain* on that attribute/parameter/return value by specifying the *type of the elements of the collection*. For example, the class **Policy** has an attribute called `coverageList`, with the java type `java.util.Vector`, we can specify a collection domain on `coverageList` by stating that its *elementType* is **com.mywebinsurance.claimprocessing.Coverage**. Naturally, if your Java 5 (and beyond) class used the generic type variety for the vector, i.e., **Vector<com.mywebinsurance.claimprocessing.Coverage>**, then Rule Studio will add the collection domain automatically, with the appropriate element type. With pre-JDK 5 collections, we have to add them explicitly.
5. *Other.* We can specify custom domains in cases that do not fit the above patterns, using an esoteric notation.

Domains are useful for three reasons: (a) convenience to rule authors, (b) maintaining data value integrity, and (c) support for powerful rule constructs with the *Business Action Language*, discussed in Sect. 10.4.2.

Notice that Rule Studio supports the dynamic computation of domains. Consider a domain that enumerates the possible medical procedures. That list will likely be updated as new medical procedures are developed every day. We would want that domain to be updated automatically whenever new procedures are added so that rule authors will *automatically* get the most up-to-date list of procedures to

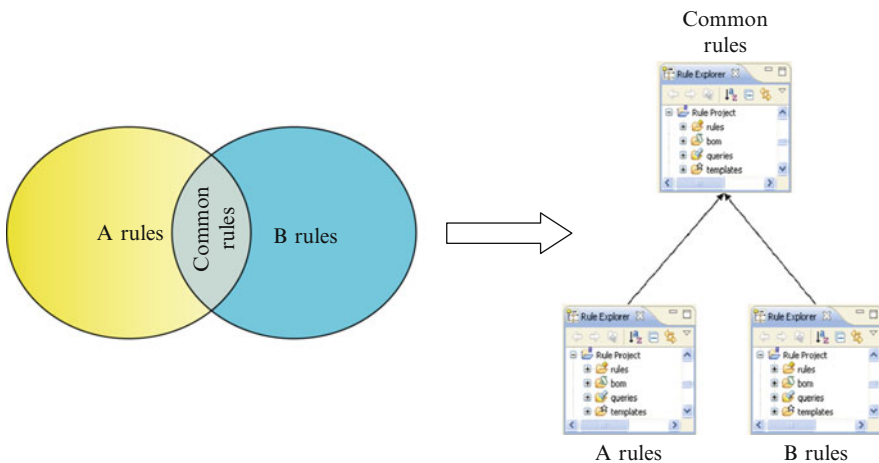
write their rules. It is possible to set-up *dynamic domains* in both Rule Studio and Rule Team Server, which get initialized at the beginning of each session with the tool.

## 10.4 Best Practices

In this section, we present best practices related to the organization of rule projects, and to the design of the BOM.

### 10.4.1 Best Practices for Organizing Rule Projects

We just saw in Sect. 10.2.4 how JRules deals with multiple users accessing and updating the same rule projects, in both the RS and RTS environments. While both RS and RTS support multiple users concurrently accessing the same rule project, a rule project does represent an easily manageable modularization boundary, in both RS and RTS. As such, it can be used as a unit for work for an effective division of labor. However, when we are trying to divide up work between the members of a team, we need to be concerned about *both* (a) enabling team members to work separately on things that are within their exclusive jurisdiction, with no interference from others, *and* (b) enabling them to share the things that are common to their work. This is where *project dependencies*, discussed in Sect. 10.2.2, come in handy. Figure 10.23 illustrates the idea. The common rules are defined in a separate



**Fig. 10.23** Using project dependencies to better modularize rule projects that share some rules

project, and are thus (a) made accessible to both projects, and (b) maintained separately from them.

Project dependencies also come in handy for building and maintaining BOMs. Because different decisions/rulesets may use the same BOM, we should define the BOM in a separate project and have projects for rules that depend on that BOM refer to that project. Going one step further, we could also use project dependencies and the notion of a BOM path (see Sect. 10.2.2) to build the BOM incrementally. For example, in the case of MyWebInsurance, we use rules for new policy underwriting, rules for policy renewal, and rules for claim processing. All three decision areas refer to a **Policy** and the **PolicyHolder**'s basic data. Policy underwriting and policy renewal would *also* refer to the **PolicyHolder**'s **DrivingRecord** and **CreditProfile**. On the other hand, policy renewal and claim processing refer to **Claim**'s, past (for renewal) and present (for claim processing). We could thus a first rule project with no rules in it but just the basic common BOM, i.e., containing **Policy** and **PolicyHolder**. Other BOM-only rule projects are then created to add process-specific BOMs. And so forth. Figure 10.24 illustrates this idea.

The example of Fig. 10.24 is just another variation of Fig. 10.2: when have two or more rule projects that overlap (BOM-wise or rule-wise, or ruleflow-wise, etc.), we separate the common parts from the exclusive parts and put each in a project where the projects with the exclusive parts refer to the project with the common parts. Figure 10.25 shows the full pattern.

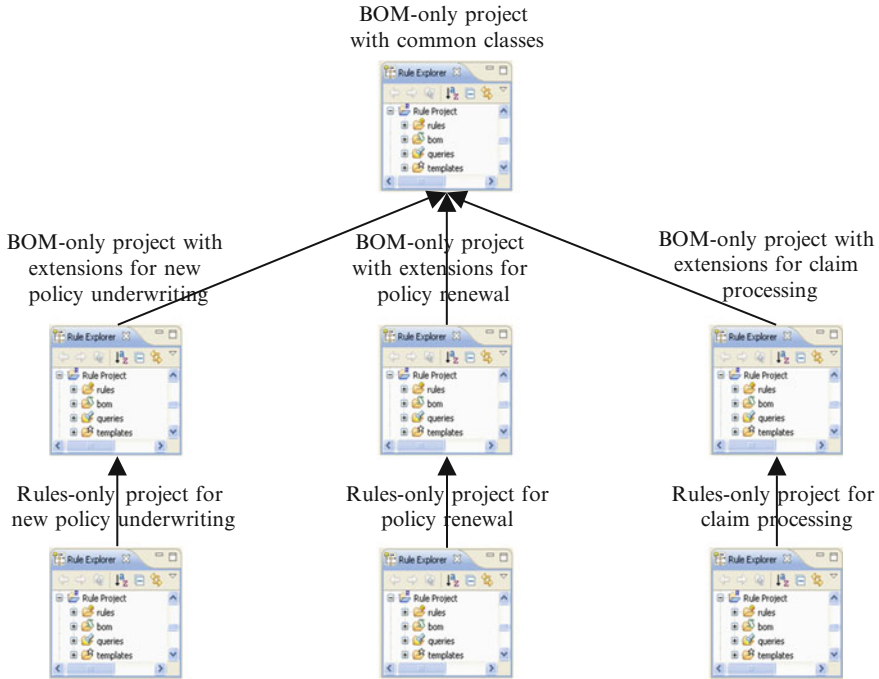
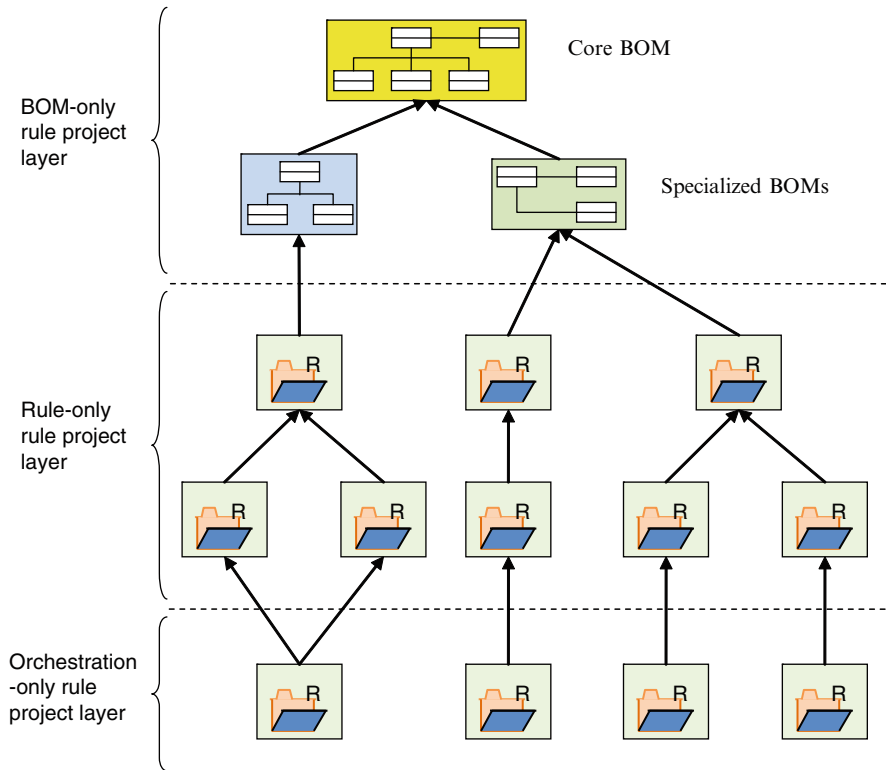


Fig. 10.24 Using project dependencies to build specialized BOM by leveraging commonalities



**Fig. 10.25** A recommended three-layer rule project structure

We will talk about rule execution orchestration in more detail in Sect. 11.3. For the time being, suffice it to say that an orchestration-only rule project is a rule project that defines a *ruleflow*, which is a process flow for rule execution where each task of the process flow *typically* runs the rules contained within a rule package. Thus, an orchestration-only project would define a rule flow that sequences the execution of rules (rule packages) defined in the rules-only layer. This enables us to reuse the same set of rules for different processes. For example, the same policy data validation rules could be used for both new policy underwriting and for policy renewal. Thus, such rules would be defined in one rule project, which could be referenced by two orchestration-only rules that pull those rules in for both processes. We will revisit this topic briefly in Sect. 11.3.

## 10.4.2 Best Practices for the Design of the BOM

The clear separation that JRules draws between the actual implementation of application data (the XOM) and the business view of it (the BOM) is a very powerful feature. It provides “rule architects” with lots of degrees of freedom,

and, alas, too much creativity. As with many features of the product, they should be used wisely, and we should show restraint in ringing all of the bells and blowing all of the whistles. In this section, we present some best practices.

### 10.4.2.1 Best Practice 1: Build Your BOM from Interfaces

If you are doing anything remotely OO or Java, there are a bazillion reasons to program to interfaces, as opposed to classes, and most authors hammer that message, and most frameworks rely on such a separation. We will give you a few more reasons to separate interfaces from implementations, by showing you the benefits of building your BOM based on interfaces, as opposed to based on the classes that implement them.

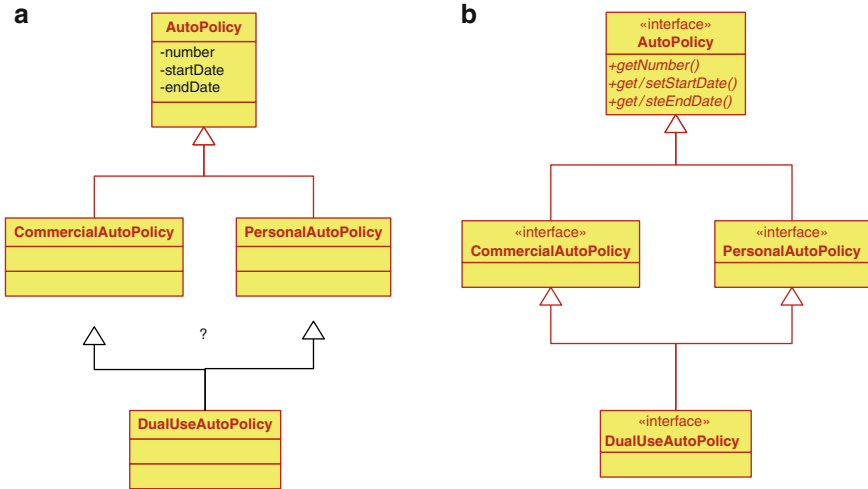
Recall that when you build a BOM from XOM, the BOM builder ignores all of the XOM elements that are *not* public: any model element that is private, protected, or package is *ignored*. Second, implementation classes will typically have *business functions*, but will also include many utility methods that provide services to the business functions, or that implement non-business infrastructural services (saving, loading, serializing, logging, etc.). Such methods will only clutter the BOM, and we know that they will not be needed to write rules. Thus, *business interfaces will contain all of the necessary and sufficient elements you will need in the BOM.*<sup>27</sup>

Then there is the issue of nomenclature. Business interfaces (typically/should) use implementation neutral terminology. A policy is called Policy and not PolicyBean, or PolicyImpl or PolicyTransferObject, or PolicyDAOImpl, or PolicyDAOBeanObserver, or PolicyDAOImplFacade. As we saw above in Sect. 10.3.2, the Rule Studio BOM builder does a pretty good job of verbalizing your classes. If you use the names used in your implementation classes, they are likely to be polluted by initials of the authors, prefixes or suffixes of the various frameworks that you are using, markers of the coolest design pattern you just read about, etc.

But the strongest argument of all is the applicability of your rules. Assume that you build your BOM from the classes shown in Fig. 10.26. Here we assume that **PersonalAutoPolicy** deals with vehicles used *exclusively* for personal activities whereas **CommercialAutoPolicy** deals with vehicles used exclusively for business activity. The BOM classes will mirror this structure. If you write a rule about a **AutoPolicy**, and you hand your rule engine an **PersonalAutoPolicy**, the rule will be evaluated on that policy and will fire if applicable. Idem for **CommercialAutoPolicy**. You would also typically have rules specific to **PersonalAutoPolicy** and others specific to **CommercialAutoPolicy**. Assume now that MyWebInsurance decides to create a novel auto insurance product that combines the features of a personal and business auto policy: we will call it **DualUseAutoPolicy**. This kind of insurance will share *some*

---

<sup>27</sup>Naturally, provided they are properly designed.



With classes, a DualUseAutoPolicy can inherit rules about either CommercialAutoPolicy or PersonalAutoPolicy, not both

With interfaces, we can

**Fig. 10.26** Building the BOM from interfaces yields more robust and more reusable rules

characteristics with **PersonalAutoPolicy** and others with **CommercialAutoPolicy**. We would also want to *reuse* the corresponding rules about **PersonalAutoPolicy** and about **CommercialAutoPolicy**. With Java classes, this would not be possible: the **DualUseAutoPolicy** class can inherit from either **PersonalAutoPolicy** or **CommercialAutoPolicy**, but not both. And thus, we would be able to reuse either the relevant rules about **PersonalAutoPolicy** or the ones about **CommercialAutoPolicy**, but not both. With Java interfaces, we can define a java interface **DualUseAutoPolicy** that extends both **PersonalAutoPolicy** and **CommercialAutoPolicy**, the BOM classes (interfaces in this case) will mirror that structure, and rules specific to either **PersonalAutoPolicy** or **CommercialAutoPolicy**, will apply to **DualUseAutoPolicy**.<sup>28</sup>

### 10.4.2.2 Best Practice 2: Too Much of a Good Thing ...

The layered architecture of the rule authoring stack of JRules provides a very clean abstraction mechanism. This enables us to limit the impact of the changes we make to the different layers (see Fig. 10.12). This is important because when we start identifying and coding rules, we frequently identify new data requirements, typically new *business* attributes or actions that rule authors need to write rules

<sup>28</sup>OK, maybe you do not want *all* of the rules that are specific to either **PersonalAutoPolicy** or **CommercialAutoPolicy** to apply to **DualUseAutoPolicy**'s and that is OK, because the tool allows you to pick and choose (see Sect. 10.5 about rule orchestration).

(e.g., new attributes). In general, if the *application* object model has been thought out thoroughly, *most* of the data and functionality will be present in the XOM in a “raw” form: then, it is just a matter of *computing* the required attributes from the existing ones (e.g., computing age from birth date), or implementing virtual functions that provide a convenient shorthand for some XOM functionality. Either way, the new data and function requirements will be defined in the BOM to XOM mapping. Naturally, there will be cases when the required data or functionality is not present in any shape of form. In that case, we need to make changes to the XOM. But as the project progresses through the various iterations of ABRD (see Chaps. 3 to 5), and as the application goes into maintenance mode, the XOM should become fairly stable.

At what point does the BOM to XOM mapping become too much of a good thing? As mentioned in Chap. 8, the BOM and the XOM need to satisfy different sets of requirements: the BOM needs to be close to the business, at the expense of some redundancy, and the XOM needs to be “canonical,”<sup>29</sup> at the expense of some readability.

First, by *design*, given a well-designed XOM, all of the virtual BOM elements will be more or less redundant with other elements. The convenience of having a BOM element to express *every* nuance and *relationship* between concepts comes at a price:

1. The possible confusion between close BOM concepts
2. The conceptual overhead of learning a rich vocabulary

For example, assume that some rules need to reason about service acts that cost more than some value. We could either add a virtual method to the class **Claim** that does just that:

```
Collection<ServiceAct> getServiceActsCostingMoreThan(float cost);
```

with the verbalization:

```
the service acts of {this} that cost more than {0};
```

and code the “cost filter” in the BOM to XOM mapping by iterating over the service acts of the claim, and returning those that cost more than the argument. Or, assuming that the service acts of a **Claim** are verbalized as “the service acts of {this},” we could code the “cost filter” directly in the rule language (see BAL in Sect. 10.4.2) as in:

```
set 'costly service acts' to all service acts in the service acts of
  'my claim' where the cost of each service act is at least 500;
```

---

<sup>29</sup>That is, it contains a minimum number of “orthogonal” concepts that can accommodate the most data or functionality needs.

Editing the BOM to accommodate new rules should be an *exceptional occurrence*, especially in rule maintenance mode; we should not have to edit the BOM – and add one virtual function – for *every* condition any business user or policy manager can think of. Alas, we have seen many customer projects where the BOM grows linearly with the rule set . . .

There is another reason why one should *not* put too much in the BOM to XOM mapping. Unlike Java code, which source-code management software handles quite well (class/file-level versioning, class member granularity for merging conflicting versions, documenting changes), the BOM to XOM mappings are all lumped into a single file, with coarse-grained versioning, and little or awkward visibility to developers. This makes it into the least manageable part of your code. Of course, there are legitimate uses for virtual functions and the BOM to XOM mapping, and once an application has gone into production, we certainly do not want to deploy a new version of the application (the XOM) each time some rule needs a new computed attribute or a new convenience method. Between application/Java code releases, we should use all the tricks of the book. However, with each planned code release, we should take the time to revisit the virtual BOM elements and their BOM to XOM mappings and assess whether they should be pushed back to the XOM. *If* the virtual BOM element embodies *significant* and *generally useful* business logic, then that element and its BOM to XOM mapping should be pushed back to the XOM at the next opportunity. Computing an age from a birth date does not represent *significant* business logic – it is rather trivial. Computing the compound yearly interest rate of a loan based on the daily interest rate – or vice-versa – is *significant and generally useful*, i.e., useful to *other* parts of the business application besides the rule service.

### 10.4.2.3 Best Practice 3: Do Not Be Too Creative

Consistency is a highly desirable property in software: you choose an architecture, a design, a pattern, a coding style, a nomenclature, a file structure pattern, what have you, and you apply it uniformly. Consistency is desirable because it makes software understandable, maintainable, scalable, etc., and its components reusable, portable, and all around adorable. Consistency comes at a cost: whichever pattern you choose (architectural, design, coding, structuring, naming, etc.), it will *not* be *optimal* in every situation, or for every component or part of your software. If you choose your patterns carefully, they will be optimal or near optimal *most* of the time. For the remaining cases, live with the awkwardness or sub-optimality: it is a small price to pay for the resulting consistency!

This general principle applies also to the BOM and the vocabulary. Do not be too creative. Take the example of verbalization. Rule Studio generates decent to good verbalizations, 95% of the time.<sup>30</sup> It also knows a bit about grammar. Do not tweak

---

<sup>30</sup>Less if you use lousy naming patterns – or none at all – in your XOM/Java code.



verbalizations to death so that your rules will read like English: they will not and they do not need to. We are not writing poetry: the rules need to be understandable and precise, not necessarily perfectly constructed English sentences or pleasant to the ear. Consistency makes learning the BOM and the vocabulary much easier, and the resulting rules less error-prone.

## 10.5 Discussion

In this chapter, we discussed rule projects and the Business Object Model. Together, they provide the basics of the rule authoring infrastructure. They also play a crucial role in the quality and the manageability of the rules. A poor BOM design can lead to rules that are barely better than programming code. A poor BOM design can also lead to rules that are brittle, i.e., that are not properly shielded from non-essential changes taking place on the XOM side. A poor rule project design can lead to an inefficient and chaotic division of labor between rule developers. It can also lead to poor rule reuse and sharing and to nightmarish rule maintenance. It is critical to get those designed right before rule authoring can start. Naturally, the BOM will most likely continue to evolve during rule authoring and maintenance, but we have to get the basic *architecture* of the BOM right before we start. The design guidelines and best practices provided in this chapter should give you a head start.

The design tasks, skills, and decisions described in this chapter fall within the purview of the *rule architect*. A typical rule *writer* does not have the skill, and should not be given the responsibility, of designing rule project structure and the various BOMs. Chapter 11 will address rule writer-specific tasks and skills within the context of JRules.

## 10.6 Further Reading

As this chapter is JRules specific, additional sources of information can be found in the product documentation and on IBM's support site for *Websphere Ilog JRules*. More information about the rule engine execution algorithms can be found in Chap. 6 and its references. The web site <http://www.agilebrdevelopment.com>, which is dedicated to this book, contains complementary information.