### CHAPTER 6

## INSURANCE PRODUCTS

In this chapter we describe the specification of financial and insurance products for a web portal. Unlike the other cases presented in this book, we focus here on modeling only static structures: The platform already provided the implementation for the behavior. The allocation of the specification work is interesting in this case. The modeling language is used by insurance experts, and thus by non-programmers, to specify various insurance products. The same people then use a generator to produce the required implementation as Java code for a J2EE web portal.

6.1 INTRODUCTION AND OBJECTIVES

The starting point for this case was a business decision: A company acting as an information provider and broker was building a portal for handling various financial products and insurance products. The portal aimed to compare information about insurance products from different providers and share the data with insurance companies and other financial service providers. The nature of the service was to target business users rather than consumers. To codify the product information (like health insurance), various insurance details such as insurance coverage, indemnities, payments, covered risks, damages, bonuses, and tariffs needed to be formally specified for each product and made available for analysis and comparison. The platform then provided the common parts, such as those dealing with money and

Domain-Specific Modeling: Enabling Full Code Generation, Steven Kelly and Juha-Pekka Tolvanen Copyright # 2008 John Wiley & Sons, Inc.

120

### DEVELOPMENT PROCESS

payments. The specification of insurance products was not simple since insurance products have a lot of detail and differences among similar kinds of products from different insurance companies were of great interest. In addition, each product is usually related to other insurance products from the same provider and thus combined insurance packages needed to be specified too.

The specification work was seen as considerable since the portal would include hundreds of products, and they all would need to be specified formally so they could be compared and analyzed in the future. The domain of insurance products also evolves, requiring modifications to already defined products and also new products that are defined based on market situation, legislation, and so on. So in terms of software engineering, the specifications would need to be maintained over time.

To implement the portal, the first option considered was to ask programmers towrite the product specifications in Java. This step would most likely be supported by insurance experts who would first codify relevant parts of the insurance products into requirement documents. These would then be used by programmers to write the implementation. This step was taken first and applied to implement a few product specifications. The second option, used in production, was creating a Domain-Specific Modeling (DSM) solution that would allow domain experts, that is, insurance people, to specify the products completely and then generate the needed implementation. This choice led to significantly faster development, allowed specifications to be checked early, leading to fewer errors, and reduced the amount of resources needed. Next we describe the development process and the DSM solution in more detail.

6.2 DEVELOPMENT PROCESS

The creation of a DSM solution started because the company was seeking an efficient way to capture financial and insurance products and implement its portal-based service. Manual programming was simply seen as too costly, time-consuming, and leading to errors that would later cause problems when insurance information would be analyzed and compared. These typical problems of manual programming had already become very visible when the company had implemented its first specifications of existing insurance products.

For the implementation of the DSM language, generator, and related tooling, the company invited bids from two external companies. The key requirements for the selection were the time needed to implement the DSM solution and its development costs. The company decided to use a commercial metamodeling tool and an external consultant to implement DSM. The actual implementation work took 11 man-days. The project was started in early August and the DSM solution was delivered to the company at the end of September. In October a pilot study was conducted with a few modelers and the DSM solution was introduced to the organization with a 2-day training course.

Before defining the DSM solution began, a target platform had already been selected: a repository product following the Meta Object Facility (MOF) as a storage format and interface with other tools. As mentioned, a few product specifications were already implemented as Java code. This code was used to test the other components used, such as the libraries in the framework and the repository. During DSM creation, the manually written Java code specifying the few products was used as a reference for testing the modeling language and the code generator. The main input for the language specification came from the “domain model” that was stated as a common semantic model for all insurance products. Based on the domain model, knowledge of all insurance products was explicitly specified and made available for further analysis, comparison, and modification. The domain model was used to unify the way insurance products are specified, and it offered comparability of the individual products. In terms of language creation, the domain model was a metamodel: All products were instantiated from the domain model. Looking at this another way, no insurance product could have data about an item not mentioned in the domain model. The domain model was largely defined before creation of the DSM solution began, but during the process it needed to be updated and complemented because the language specification required details that were lacking in the original domain model.

The domain model was originally formulated as an extension of MOF. It basically consisted of a number of (insurance) domain specific types that were all derived from MofClass or MofAttribute. In addition, it provided a set of rules determining the possible ways in which these domain-specific types could participate in relation to each other (like MofAssociations). These rules ensured that a certain insurancespecific product structure was followed. Tagged values were added to certain model elements to distinguish the role that a model element can play under certain insurancespecific points of view. For example, tagging an attribute with an “A”-tag meant that the attribute is needed for an application, a “T”-tag meant that the attribute is needed for calculating the premium, and so on.

The company wanted to follow standards and therefore the domain-specific language was also expected to be defined directly based on MOF types. This requirement was satisfied by first implementing MOF as a metamodel into a modeling tool and then specializing the defined MOF concepts to make the domain-specific modeling concepts. In the metamodel, the modeling concepts were, therefore, subtypes of MOF types. Rather than implementing MOF completely, just those parts relevant for the language definition were implemented. Figure 6.1 represents the major concepts of MOF defined as a metamodel using the metamodeling language of MetaEdit+, the tool applied (see Appendix A). Each rectangle represents an object type, and connections with a diamond show aggregation relationships, for example, MOFClass may have multiple attributes. Then, these concepts were used in the modeling language directly since many of the modeling concepts (object) types were created by inheriting from MOFClass.

In addition to the MOF elements described above, Association, Generalization, and Dependency relationships were implemented from MOF types. Similarly, an AssociationEnd role type was implemented and used as a supertype for the domainspecific role types different objects can take when connected to other modeling objects. We describe their use in more detail later when presenting the metamodel for the modeling language.

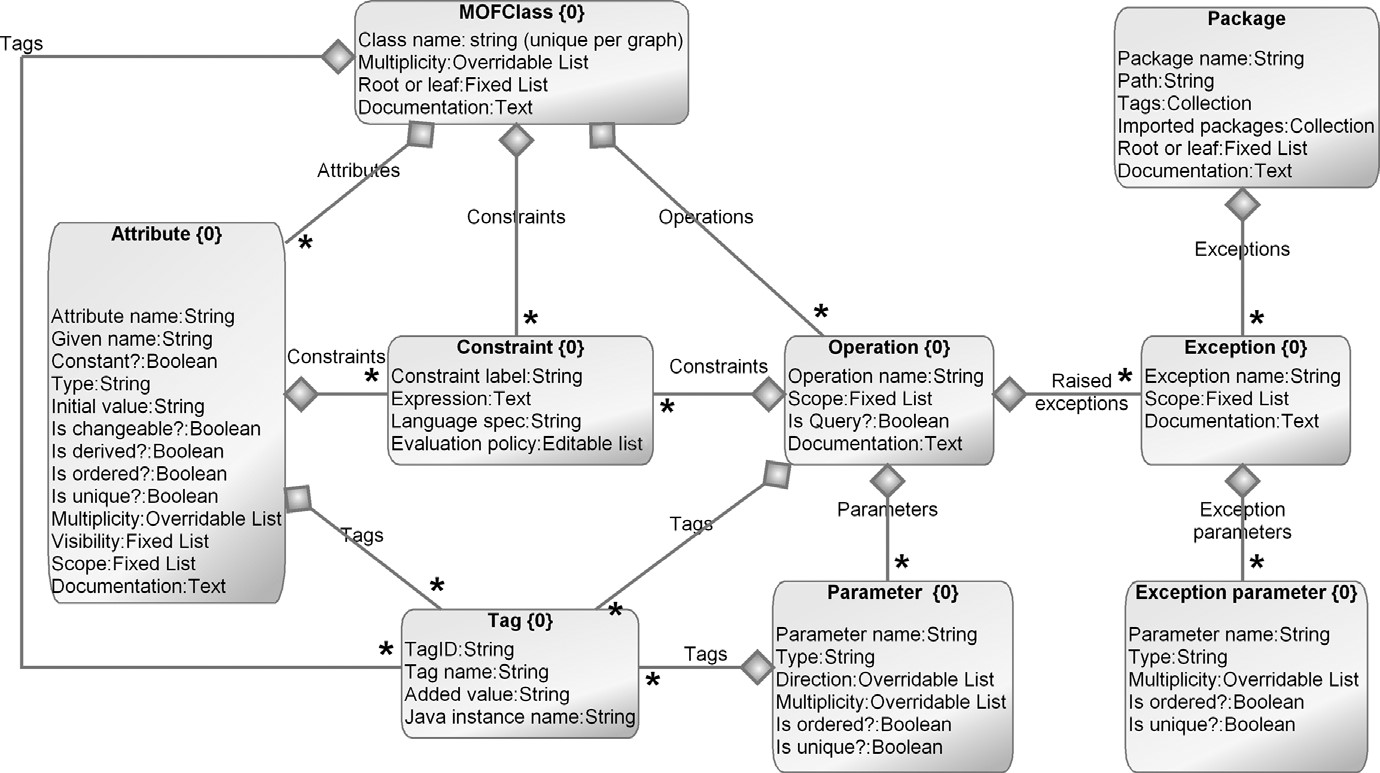


FIGURE 6.1 Metamodel of the relevant parts of MOF

The requirement to apply MOF, however, was later noticed to lead to an unnecessarily complex modeling language. For example, associations and their role names were provided in the language, since they came with MOF, but were not used at all. The requirement to follow the standard for metamodeling also caused additional delay in the DSM creation: In the middle of the DSM implementation, the company realized that the version of MOF had changed, and they needed to recheck the definitions made in the domain model. Following the newer MOF releases was seen as relevant since the target environment used for storing the information about the financial products was expected to change along with new MOF versions. However, for language creation, it was not relevant since the domain-specific language could have been implemented without any relation to MOF. However, to satisfy the company policy for following standards, some terms were changed to the existing implementation of MOF (Fig. 6.1). These changes were mostly cosmetic, and the DSM tool updated the language definitions automatically after the supertypes (i.e., MOF implementation) were changed.

6.3 LANGUAGE FOR MODELING INSURANCES

The starting point for the language definition was an existing domain model: a common semantic model for all insurance products. This domain model was made by the insurance experts of the company, and the task for the developer of the DSM solution was to make the domain model formal and usable as a modeling language. In that respect, the modeling language creation was easy since the domain was already well bounded and had established semantics. Many of the modeling concepts could be derived directly from the domain model, as could some constraints. The missing constraints were added and refined in collaboration with domain experts. This process was rather simple since most questions were easily answered by domain experts. A contributing factor was that the questions were related to the already known domain model. The questions dealt with missing relationships between objects, cardinality constraints, having binary relationships instead of n-ary relationships, and organizing specifications larger than one diagram.

6.3.1 Modeling Concepts

The analyses of the domain model showed that most differences among the insurance object types were related to the relationships they may have with each other rather than to their individual property types. In fact, almost all the object types had the same property types, those inherited from MOF. Figure 6.2 illustrates part of the original domain model.

The large number of rules related to relationships between domain concepts led to the definition of multiple different object types. Each domain-specific object type then had related roles and constraints. Another alternative considered was having just one object type with a classifying stereotype along with a constraint definition for connecting objects. This approach, however, was impractical since some objects

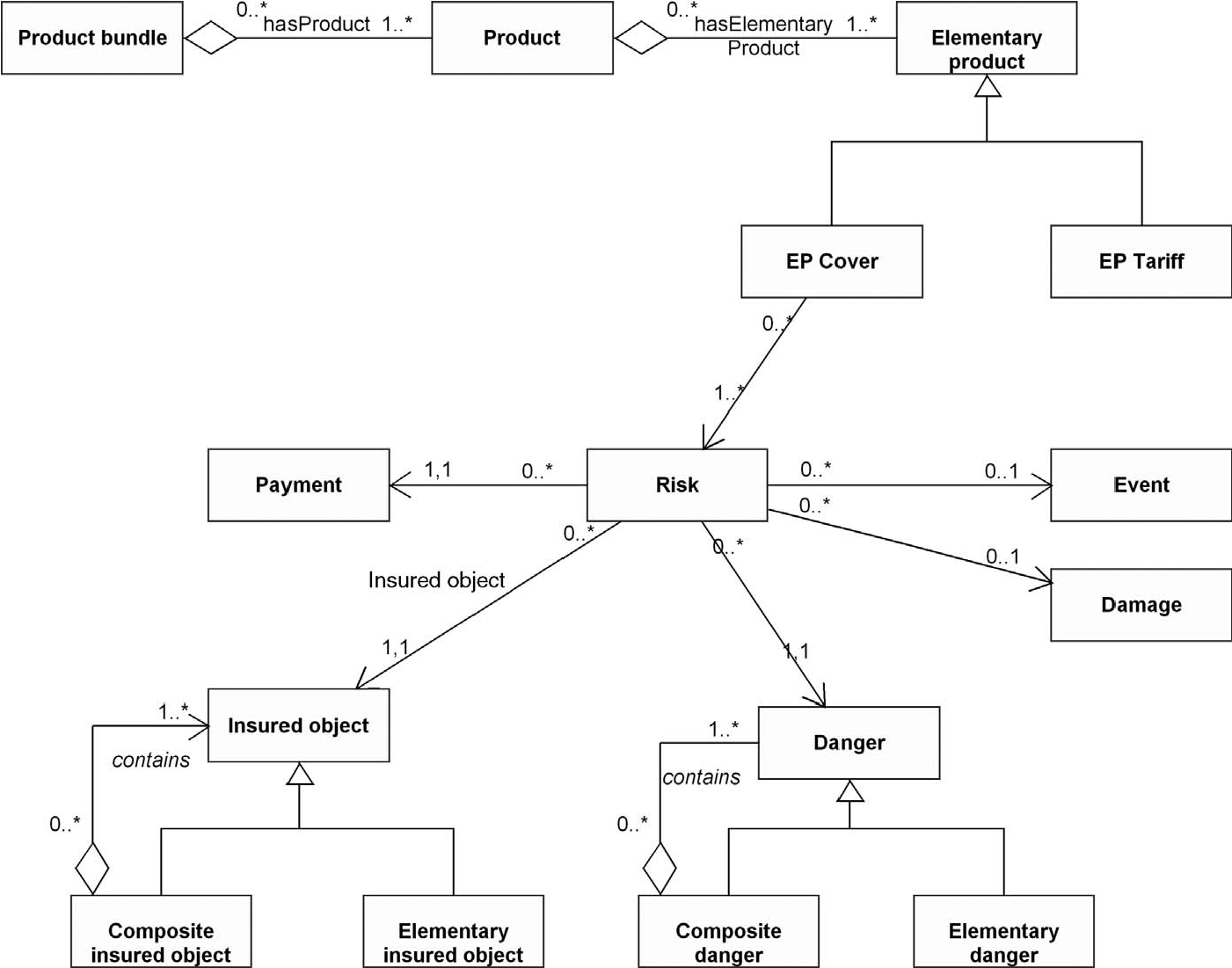


FIGURE 6.2 Part of the domain model

had additional properties, and it would make the constraints too complex to define and maintain. The original domain model, having different object types for each domain concept, also favored creating individual object types in the language.

Language definition started by categorizing the domain concepts into groups and implementing the root concepts first. In the case of insurance products, these root concepts are an insurance product and a bundle of insurance products. Since both of these had common properties, a new supertype, called Domain Class, was added to the metamodel. This supertype had the common properties like Given name and Type. Figure 6.3 shows the main modeling concepts in relation to indemnity insurances. For the modeling of life insurance products, the language needed more modeling concepts. At this point, it is worth mentioning that the complete metamodel was defined as one diagram, and for the sake of its presentation here, we have divided the metamodel into separate figures.

While the Product bundle and Product refer to whole insurance products, the rest of the domain concepts are used to describe their characteristics. When specifying the metamodel, all the candidate language concepts already had an existing definition. For example, an event was defined as a process of the real world, such as achieving a given age, thunderbolt, theft, suicide, natural death, or buyback, and a damage was defined as an insurance-technical classification of cases of damage as effects of events (e.g., destruction of house contents, loss of cash, disablement, damage to vehicles). The names for the modeling concepts were chosen directly from the domain model to make the language easier to learn and use.

Some domain concepts were further characterized with specific property types, such as followings:

. An insured person has a property type Role to specify the related policy outline, such as connected life insurance or add-on widow insurance.

. Calculation basis has a property type Computation purpose, which could have values like a balance and a calculation.

. Surplus may be based on different types, such as a bonus, an immediate risk bonus, and an immediate premium deduction.

. Payments could be further characterized by their type.

. Tariff has two property types: Type for specifying the tariff type used, such as new contracts, compensation, and dynamic, and a property type Variant for specifying if different tariffs are used over the time.

All these properties had some predefined values that could be directly used as a basis for selection. Therefore, the property types were defined as different kinds of lists, such as an overridable list for either selecting one of the predefined values or typing one-off values, or an editable list where the modeler can add values to the list for future selection. The language was defined mostly to use editable lists and it was decided that their use would be analyzed later once the language had been used for defining a larger number of products.

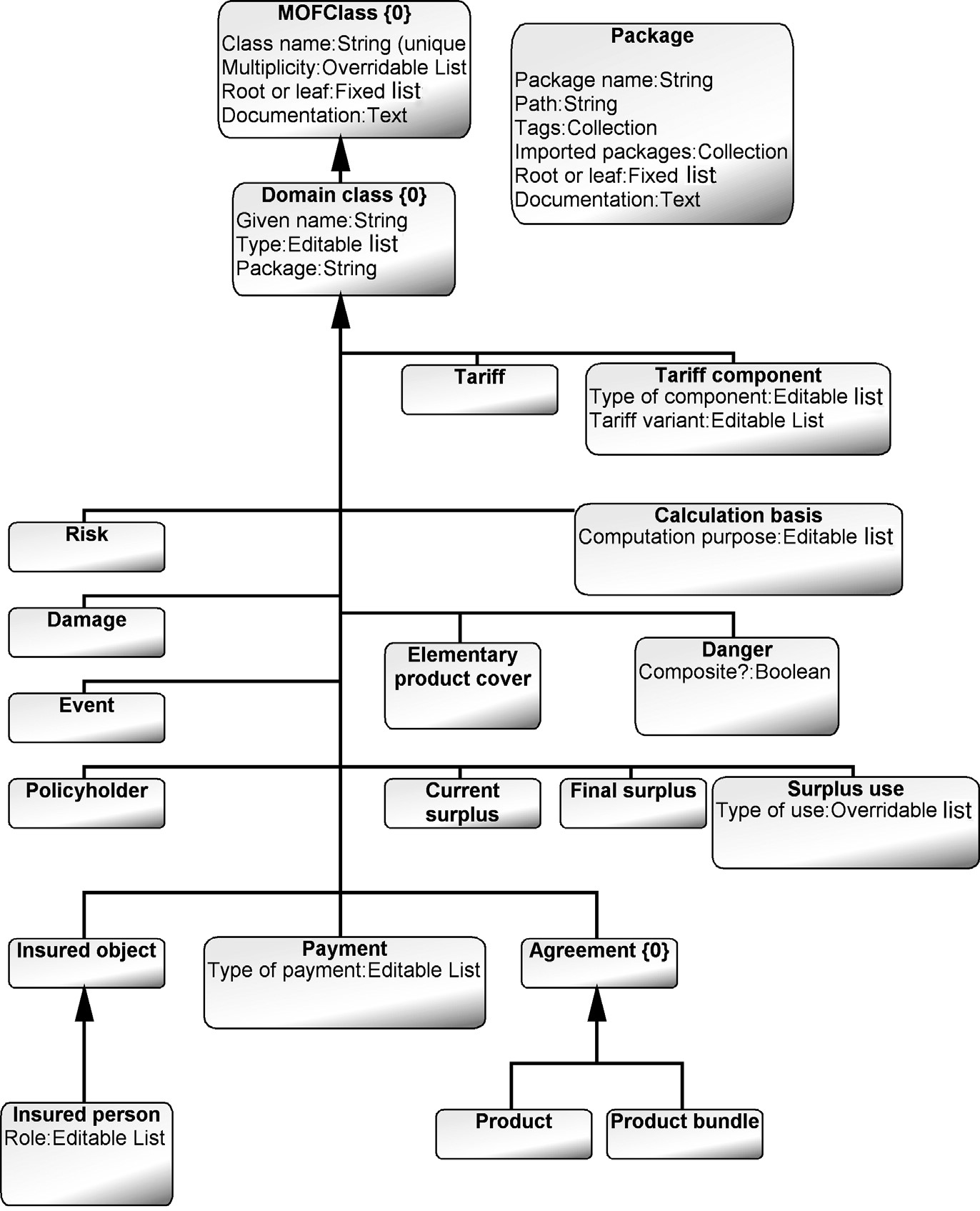


FIGURE 6.3 Insurance-specific objects inheriting from MOFClass

Each modeling concept needed a reference property type for specifying the package to which it belongs. This property was added to the common supertype Domain class (see Fig. 6.3). Its data type was made an object so that it referred directly to a Package model element instead of having just a mapping with a string value of the package name. This package reference was needed only for cases where a product specification represented in a single diagram had elements from multiple packages. Otherwise, all elements in a diagram belonged to the same package: the model hierarchy specified the package and product they belonged to.

6.3.2 Modeling Rules

In addition to object types and property types, various relationships and their related constraints were identified and defined. Depending on their nature, the constraints were either implemented directly into the metamodel or checked using generators. Most of the relationships deal with insurance-specific rules, such as that insured objects can be related to risks and that surpluses can be connected to tariff components. Figure 6.4 illustrates some of these relationships, such as that a Product bundle can consist of Products, which can further consist of either Elementary product cover or Tariff elements. This allows us to describe insurance products from a tariffcentric or a product cover point of view.

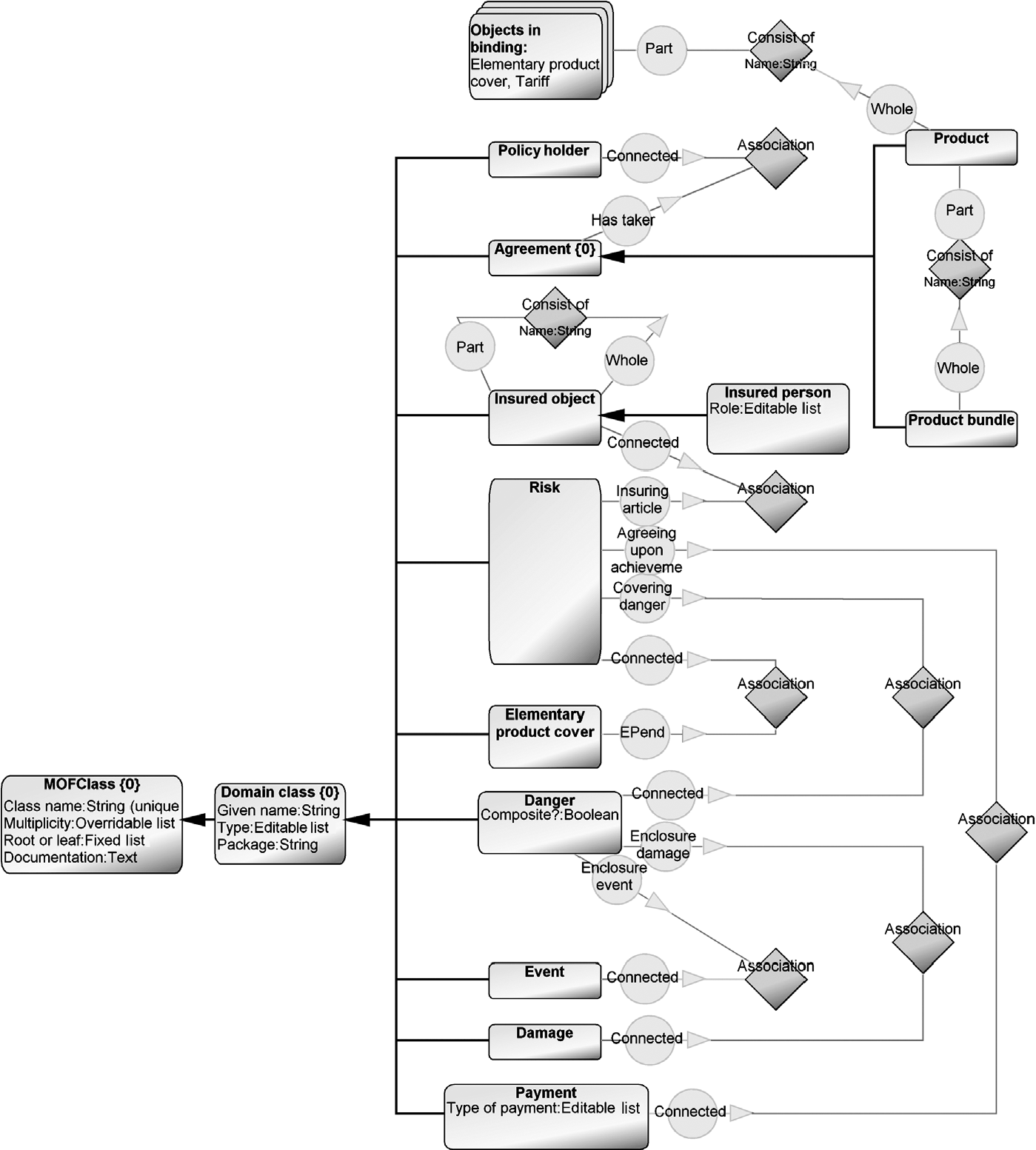


FIGURE 6.4 Relationships between insurance object types, Part 1

Rather than using different relationship types for connecting specific object types, the relationship types from MOF, namely, Association and Aggregation, were used. When the metamodel was used in the modeling tool, the right relationship type was chosen by the tool. If multiple relationship types were possible, the modeling tool asked the user to choose among the possible ones.

The legal connections were specified in the metamodel by using specific role types. Most of the role types were again inherited from the MOF AssociationEnd, and therefore had property types like name, multiplicity, and navigability. The latter two had predefined values for speeding up the modeling work. For multiplicity, the most typical values (0,1; 1,1; 0,M; 1,M) were added to a predefined list for quick selection. For navigability, the selection list was defined as having a mandatory value with a default value of having no navigation. The other possible values that a modeler could choose were restricted to “Is navigable” and “Is navigable and references.”

All the relationships were defined as binary, although n-ary relationships would require less modeling work: with an a-ary relationship just one aggregation

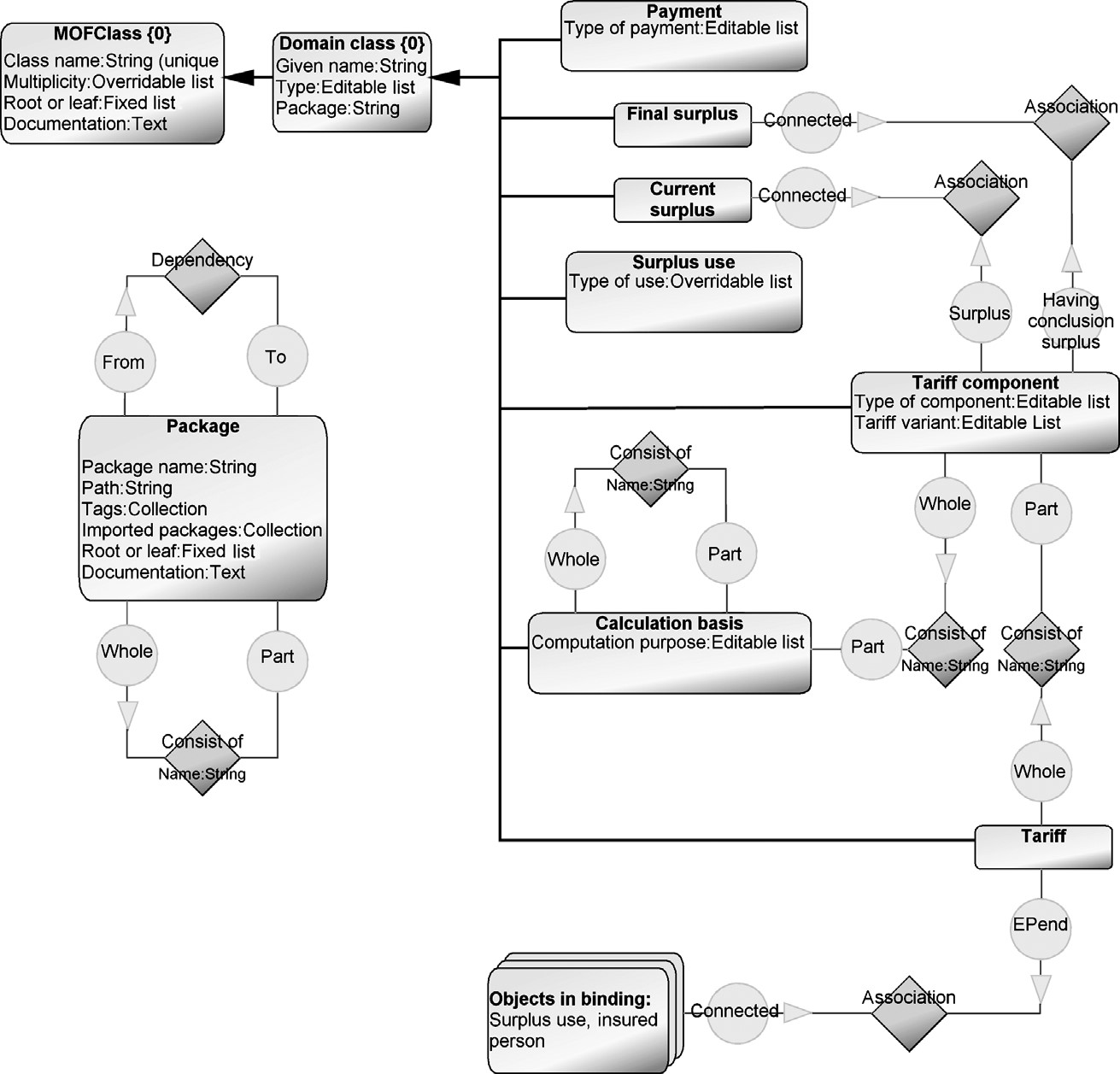


FIGURE 6.5 Relationships between insurance object types, Part 2

relationship between the Product bundle and Products would be needed, rather than drawing each aggregation as a separate relationship and filling the same aggregation information multiple times. Since the users of the language were not experienced in modeling, the choice to make model creation simple was emphasized. Figure 6.5 represents the remaining object types and their connections.

Along with the bindings between the object types, role types, and relationship types, constraints related to these domain concepts were defined. These constraints added to the metamodel and include the following:

. A Danger could be connected to only one Event.

. A Danger could be connected to only one Damage.

. A Product can have only one Insurant.

. A Product bundle can have only one policy holder.

. A Risk can be connected to only one Danger.

. A Risk can be connected to only one Payment.

. A Risk can be connected to only one Insured object.

. A Tariff component can be connected to only one Surplus.

Implementation of the metamodel also revealed some mismatches in the original definition of the metamodel. These were partly because MOF was used. For example, the inheritance relationship originating from MOF was changed in the metamodel so that inheritance was only possible between similar types of domain elements: risks could inherit properties from another risk element but not from a product or an insured person. Implementation of the modeling language also revealed missing information from the domain model, such as which properties need to have values, which are legal data types, and which parameters (inherited from MOF) must be defined as returns. All the additional constraints were added to the metamodel and the original domain model was not updated. The metamodel of the modeling language became more detailed and precise. It could also be tested as a language once instantiated, unlike the documentation of the domain model. Testing of the language definition was done by modeling the reference applications and generating their implementation code.

Model Hierarchy To model large or complex insurance products, the modeling language was extended to support model hierarchies. This was done by using a package concept: each package could be described in detail with a submodel. The submodel was based on the same modeling language, so all the modeling concepts available in the higher level model could be used in the submodel. In the metamodel, an optional decomposition link was defined from each Package object type to the product modeling language (see Fig. 6.6). In addition, package structures could be specified in one diagram using a Consist of relationship, and the language also had support for specifying dependencies among packages. These latter two are described in the metamodel represented in Fig. 6.5.

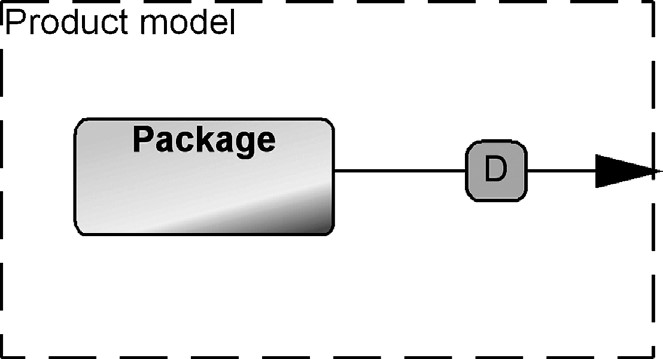


FIGURE 6.6 A part of the metamodel specifying model hierarchies by allowing a decomposition from an object type Package to the Product modeling language

Model Checking Not all domain rules could be included in the metamodel since their checking would not make sense at modeling time. For example, each insurance product needs to include at least one elementary product cover, but this kind of rule can’t be checked at modeling time, since immediately after adding a product object to the model the design would be invalid. Various generators were implemented for checking such rules. These included, for example, that every Product cover needs to refer to a Risk object and that each Product bundle must refer to at least one Product. Although these model checking topics were detected while creating the metamodel, they were implemented last when it was possible to test them by using the language to specify some insurance products. The available models then acted as test material for the model checking generators. In addition to model checking, additional generators were made for producing documentation and generating an overview in HTML format.

6.3.3 Modeling Notation

Since the modelers were not software developers, language visualization (e.g., the visual appearance of the notation), ease of use, and user friendliness were emphasized. To gain better acceptance for the introduced language, the notational symbols were asked to be defined by the users of the language, the insurance experts. Most notational elements were taken from existing signs related to events, risks, and payments, such as traffic signs and currency. Figure 6.7 shows the notation for some modeling concepts when using the language to create specifications.

A specific question on the notation dealt with showing detailed properties of model elements. Since insurance elements usually had 5–10 properties, showing all would make large symbols that took most of the modeling space. Instead, the functionality of the modeling tools was used to hide the details and make them available via browsers and dialogs related to the model element. The symbols were used to show just the most important information, such as name, given name, and composite information. Since the modeling concepts were new in the beginning, each symbol showed its type name too. Later, once the language had been used, this extra cue for identifying the different types was removed.

Since it was not always desirable to specify inheritance with an explicit relationship, each domain element was modified by adding an inheritance property type. Its value was then the possible supertype. This property type was defined as an object, instead of using name matching, so that the details of the supertype could be

### MODELING INSURANCE PRODUCTS

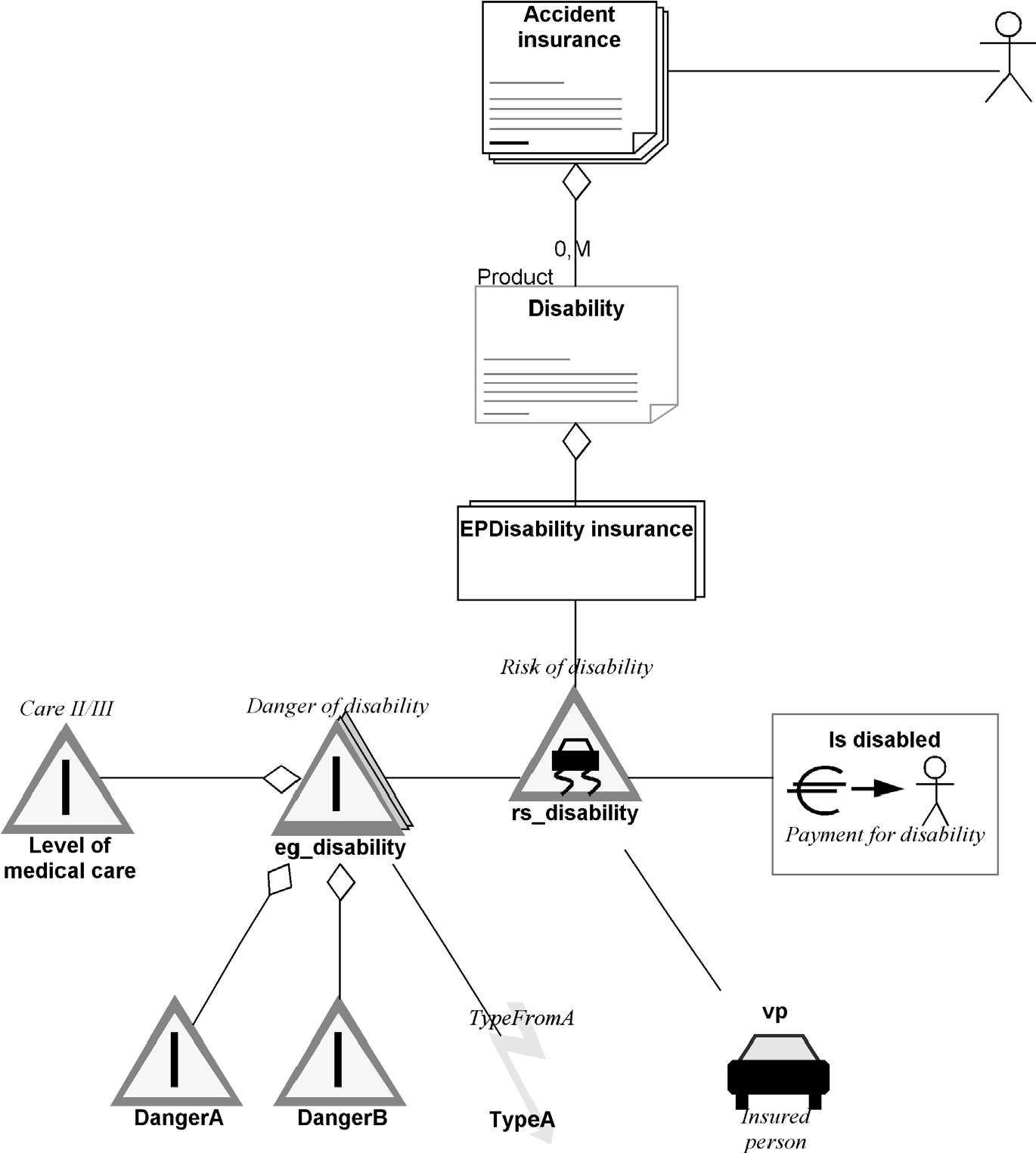


FIGURE 6.7 Sample insurance product (partial model)

viewed during model creation. This language structure also allowed creating new supertypes where such did not yet exist.

6.4 MODELING INSURANCE PRODUCTS

6.4.1 Example Models

The defined modeling language was used to specify the characteristics of products, and therefore,thelanguagewascalledaproductmodelinglanguage.Thefirstmodelscreated were related to testing the language during its definition. The language was tested in a realistic situationbyspecifyingtheinsuranceapplications alreadyimplementedbyhand in Java. Figure 6.7 illustrates the specification of one small insurance product.

6.4.2 Use Scenarios

The modeling language was used by insurance experts. These nonprogrammers drew models similartoFig. 6.7,specifying insurance products, andthenexecuted a generator toproducetherequiredcodeforaJ2EEwebsite.Thelanguageclearlyraisedthelevelof abstraction since insurance experts could apply the already known insurance concepts. Actually, in the beginning the modelers did not even realize that they also generated the Java code. Later we will discuss the code generation in more detail.

During the introduction of the DSM solution a 2-day course was given to train the insurance experts in using the modeling language. For the training, the sample reference application was used. After using the DSM solution, the number of created specifications increased quickly, and after 6 months several hundred products had been specified covering well over 500 risks. To copewith similarities among products, a template specification was created for each insurance type to be used when specifying similar products from different insurance companies. Product templates were used as incomplete specifications and models made from them could be changed as needed while creating the final specifications.

Evaluation of the Language Later, analysis of the models revealed that similar patterns occurred in models depending on the insurance type, for example, indemnity products were modeled differently from life insurances. To minimize the modeling need, it would have been possible to create metamodels for each insurance type. Although the effort needed to create and maintain multiple similar languages would not have been large, the company decided to use just one language to specify all the insurance products. It was thus accepted that modelers needed to draw some structures almost identically for all insurance products of the same category. The tool helped here as it allowed the same structures to be copied from a template library either by value or by reference.

Since the metamodel also included MOF concepts, more for “standard compliance” than real need, it was unnecessarily large. For example, none of the product specifications used a generalization relationship, a concept taken from MOF. The modeling tool allowed removing the unused concepts or just hiding them from the modelers. Particularly useful was that this was possible even when the language was already used to specify dozens of insurance products.

6.5 GENERATOR FOR JAVA

The target platform included a quotation engine that used the information from the models in the form of Java code. Later, the Java code was expected to be replaced by XML. With a DSM solution, this change was not seen as problematic since it would only require changing the generator. Just one person, the generator developer, would have to work to make a new generator allowing generation of the desired XML from the same specification models. The users of the modeling language would not even notice the change in the generated output!

The main starting point for the generator definition was the expected output: the code that was written manually before considering a DSM solution. Although the manually written code was taken as a requirement and used as a test case for the generator, its structure was not implemented in the generator: the generator produced the same functionality but in a different structure from the manually written code. The reasons for not following the manually written code were twofold: the manually written code was unnecessarily long and inconsistent. The latter was especially relevant since similar kinds of features were sometimes written differently even inside the same insurance product. This was partly expected since the company had only specified a few insurance products, and both Java as a programming language and the way to implement them were new. The code was also unnecessarily complex since many classes where implemented in the code by first declaring them and then later defining them. For these reasons, the reference applications were unified to follow one set of best practices instead of using different styles and personal preferences. Let’s look next at the structure of the code generator and then samples of the generated Java code.

6.5.1 Generator Structure

Figure 6.8 illustrates the structure of the final generator by describing the generator modules and call structures between them. The numbers on the lines connecting the generator modules indicate the execution order of the generator modules: the first, 1, produces the main class based on the package, and the last, 12, the associations between model elements.

Implementation of the generator, however, did not follow the execution order. Instead, the implementation started from the main insurance product concept that produced the main product definition. This was followed by implementing each individual domain concept based on the metamodel structure, that is, first, all elements related to the product concept were handled and then following the connections they have until all domain concepts were handled. Finally, generators for producing the common infrastructure code, such as package, importing, and data-type definitions, were implemented. This order did not allow testing of the generated code in the target environment since not all infrastructure code was available. This approach was partly a result of using an external consultant to make the generator, without access to the target environment for testing purposes. Therefore, generator testing during development was done solely by comparing the generated code to the manually written reference applications.

The generator was structured so that the handling of domain concepts was checked at the generation time: when MOF classes and their related attributes were generated, the characteristics of the domain concept were used to change the code generation. In this way, for example, the risk related classes were created differently from the payment related classes although the generator modules were the same. This approach was therefore opposite to that of the previous CPL (Chapter 5) or other examples described here that have a generator module for each domain concept. In the insurance case the decision to use the same generator module for multiple domain concepts was

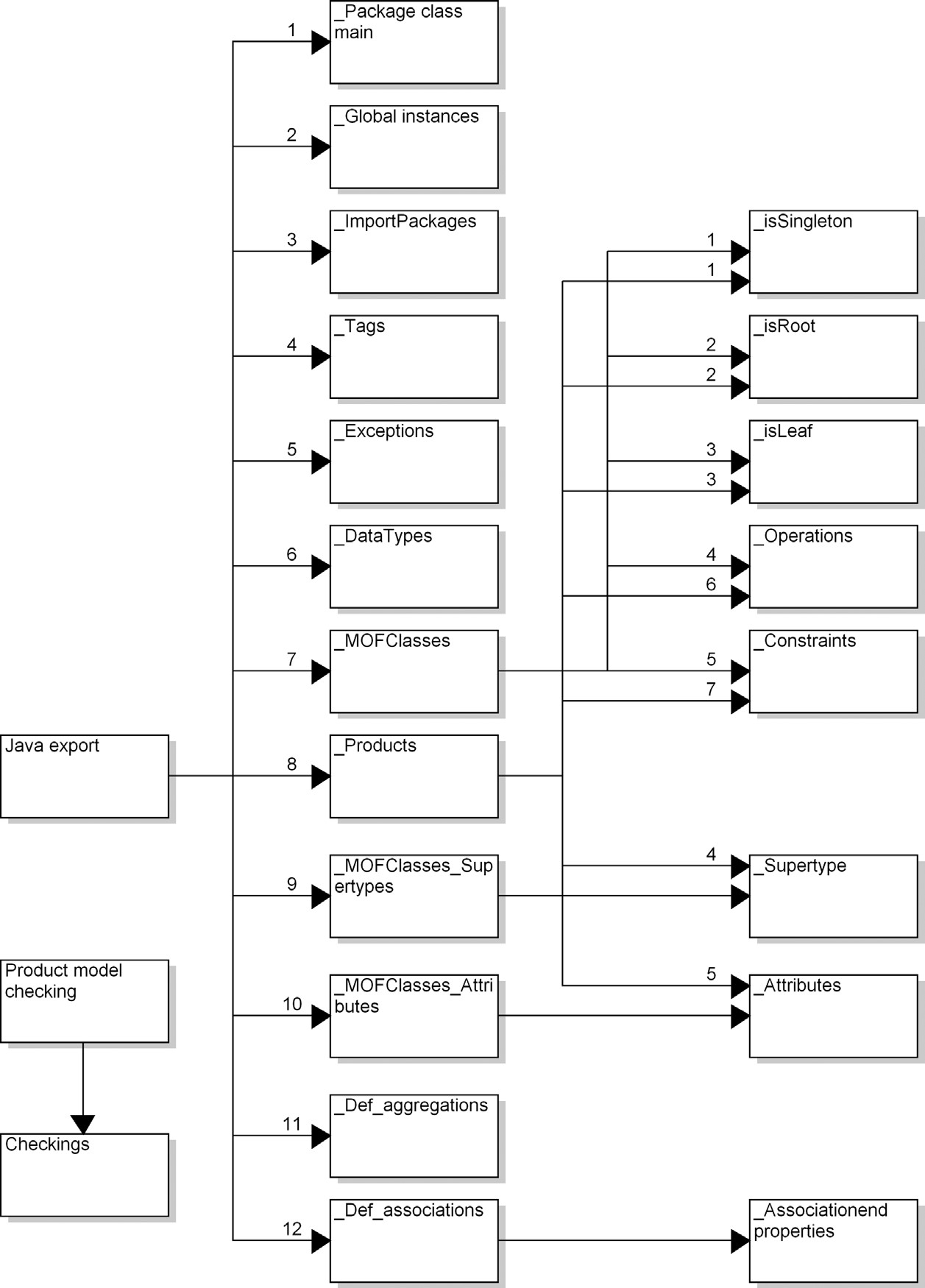


FIGURE 6.8 Structure of the generator

possible since the domain concepts were very similar and their main difference was among the legal connections they could have (see Figs. 6.4 and 6.5).

As shown in the generator structure, a product had its own generator module. The product concept was different from others since it created the product-specific Java class and acted as the main composite concept referring to other insurance-specific concepts. However, since the product concept was inherited from MOFClass and Domain class (see Fig. 6.3), similar to other insurance-specific modeling objects, parts of the generators were the same. Therefore, those parts producing the common MOFClass related code were put in their own generator modules for reuse. These handled if the concept was a singleton, root, or leaf, and also produced code related to constraints and operations. Listing 6.1 shows the generator for producing the singleton code. In the generator definition the “multiplicity” string in line 3 refers to the property in a model element. Its value in the model element is then used to define if isSingleton is set true or false, “.setIsSingleton(true);” or “.setIsSingleton(false);”

Listing 6.1 A generator for producing singleton code.

1. Report '\_isSingleton'
2. /\*Singleton is true if multiplicity value is 0,1 or 1,1\*/
3. if :Multiplicity = '1,1' or '0,1'
4. then '.setIsSingleton(true);'
5. else
6. '.setIsSingleton(false);'
7. endif
8. endreport

The names for the code, class name, attributes, operations, and so on, were taken directly from the names used in models. Another alternative considered was allowing the generator to produce unique names. This would work well for modelers since the insurance modelers were not expected to view the code they generated. For implementing the generator, however, it was considered better to use more descriptive names that could be taken from the model. Mapping names closely to the model also made the generators easier to test and later maintain. However, using just the name from the model was not enough since the same name could be used for several things, like the name of a cover and a related risk. For this reason, each name was extended with its type name. If no property names were used, for example, often AssociationEnd had no role names, the tool generated a unique name.

6.5.2 Generator in Action

The generator produces a Java class for each product, one file per specification. If the product specification was more complex, having multiple separate diagrams, all the diagrams in the hierarchy were used to create the class file. The class implements a product creation method that adds all insurance-specific details from the model to the product specification. In other words, it instantiates them as Java objects on the platform based on a predefined metamodel implementation.

From the product specification described in Fig. 6.7, the generator produces 962 lines of Java code. In production use, the average size of generated code was 2000 lines per insurance product and the largest was over 4000 lines. Next let’s examine the generator by looking at the generated code. We will only inspect some parts here: the main product creation code, code produced from a modeling object that is a domain-specific concept part of the insurance product, and code created based on the connections the domain concepts treated as objects in the model have. The sample code is taken from a generated insurance specification, 2188 lines of code in total.

Listing 6.2 shows part of the generated code. Numbers at the beginning of the lines refer to the lines of the generated file. This code is produced by the generator “\_Package class main” (see Fig. 6.8). The string “Basis” in line 15 refers to the product name taken from the Product object and its property Given name (see the metamodel in Fig. 6.3). Similarly, the string “BBBasis” in lines 17 and 18 is taken from the name property of the Package object that refers to the insurance specification. All the rest is produced by the generator since they are common for the implementation.

Listing 6.2 Produced code for a product called BASIS.

1. public class Basis extends ProductRepository
2. {
3. private static BBBasisProductRepository instance;
4. private BBBasisProductRepository (String name)
5. {
6. super(name);
7. MofPackage productpackage = createProduct();
8. this.addMofPackage(productpackage);
9. }

This code is followed by the definition of the variables for each domain-specific type used in the specified product. The generator named “\_Global instances” (Fig. 6.8) created the necessary global variables. To follow naming conventions, the variable names had to be written starting with a small letter. For such a convention, the generator could either translate each name for a required naming convention or just add an arbitrary lower case letter in front of the name. Note that it was not possible to ask the user to give a name correctly for code generation or place in the metamodel a rule that the name start with a lower case letter since the same value is used elsewhere starting with a capital letter. Having two properties for each domain object type (one for the type name and the other for a variable name) was not seen as a viable option.

After having produced the code for variables, imported packages, tags, exceptions, and data types, code for each element of the insurance is generated. Listing 6.3 shows the generated code for one danger. The code generator simply iterates through all the domain concepts as modeling objects in the models.

Listing 6.3 Generated code for a danger “extended notification period”.

122 // \*\*\*\*\*\* Elementary Danger Extended Notification Period \*\*\*\*\*\*\*\* 123 elementaryDanger = new ElementaryDanger ("extended\_notification\_period");

1. elementaryDanger.setGivenName("Extended Notification Period");
2. elementaryDanger.setIsSingleton(false); 126 elementaryDanger.setIsRoot(false);
3. elementaryDanger.setIsLeaf(false);
4. elementaryDanger.setDomaintype ("extended\_notification\_period");
5. selectionViewFalse\_.addTaggedModelElement(elementaryDanger);

130

131 productpackage\_.addMofClass(elementaryDanger);

This code maps to one object of the modeling languages referring to a domain concept Danger (see the metamodel in Fig. 6.3). Line 122 shows the comment using the type name of the model element and its given name. This particular danger is related to the extended notification period and this name is taken from the model and used in different conventions. Lines 125–127 are produced based on the MOF compliance for inputting the specification into the repository. The value “false” is taken from the property value of the respective model element. For example, the code for the singleton in line 125 is produced by the generator shown in Listing 6.1.

Once the domain concepts and their attributes and operations are produced, the generator starts to navigate through any relationships the modeling objects have. Inheritance relationships (or property values for each domain concept) are already used if they existed, but the main relationships usually dealt with the aggregations and associations each object may have. To avoid reporting the same relationships multiple times, the generator started the navigation from relationships rather than from the objects. Listing 6.4 shows the code for an association between a particular risk and danger. This type of relationship was already defined in the metamodel during language design; see Fig. 6.4.

Listing 6.4 Code for specifying association for a risk and a danger.

1561 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1562 // Associations

1563 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1564

1565 // \*\*\*\*\* RiskRIS\_dailyallowance\_X\_CompositeDangerCGF\_basis \*\*\*\*\* 1566 risk = (Risk) productpackage\_.lookupElementExtended

("RIS\_dailyallowance");

1. AssociationEnd end19\_12014 = new AssociationEnd("end19\_12014", iRisk);
2. end19\_12014.setMultiplicity(new MultiplicityType(1,1, false,true));
3. end19\_12014.setAggregation(AggregationType.SHARED);

1570

1. compositeDanger = (CompositeDanger) productpackage\_. lookupElementExtended("CGF\_basis");
2. AssociationEnd end19\_7992 = new AssociationEnd("end19\_7992", iCompositeDanger);
3. end19\_7992.setMultiplicity(new MultiplicityType(1,1, false,true));
4. end19\_7992.setAggregation(AggregationType.SHARED);

1575

1. mofAssociation = MofAssociation.createAssociation

("Risk19\_9291\_X\_CompositeDanger19\_10072", end19\_12014, end19\_7992);

1. productpackage\_.addMofAssociation(mofAssociation);

1578

1579 reference = new Reference("Risk19\_9291\_X\_CompositeDanger19\_10072", end19\_12014, end19\_7992); 1580 risk.addReference(reference);

Lines 1565–1569 specify the association a particular risk has and lines 1571–1574 specify the other end of the association, connecting to a danger. The string value consisting of numbers is a unique name for that association end. The tool needs to give names for associations since role names are neither mandatory nor unique in a model. Unique names are used in line 1576 to create an association by using the defined roles. The name for the association is again derived from the names of the objects types and their unique identifiers. Finally, if an object in an association had a reference, it was added to the object having the reference (line 1580).

The generator always produced the code in a standard manner: as was originally written in the reference applications. The only difference to manually written code was that the generated code used different naming conventions for some variables (generated unique names).

6.6 FRAMEWORK SUPPORT

Creation of the DSM solution was restricted by the already selected components of the portal: the repository to store the product specifications and its quotation engine to analyze and compare the insurance products. Since they could not be changed, the DSM solution itself needed to adapt to the existing target platform. This meant making the generator produce code as required by the target environment. No additional framework codewas written and the input format itselfformed the interface to the generation.

6.7 MAIN RESULTS

With the DSM solution, the company fundamentally changed its development process: the domain experts not only specified the products but also could test their work immediately by running the produced code in the portal. This was a big difference from the traditional method of first creating requirement documents, which programmers would then use to write the test cases, implement the code, and then finally test the results from a technical point of view and for compliance with the requirements. The CTO of the company saw the change as significant: “Traditional programming has largely disappeared and we can build systems up to five times faster with fewer errors.” These results were similar to the gains discussed in Chapter 2. In particular, the automatic generation made the development process easy and safe: modelers did not need to consider if the correct version of the supporting code was available during code generation.

The insurance experts could start using the modeling language and related tool relatively quickly—after 2 days of training. The main difficulties lay in learning to reuse existing product specifications or their parts and learning to use the MOFspecific parts visible in the modeling language. Later, analysis of the specifications showed that some of the MOF-specific concepts were not needed at all and thus could be removed: they just led to an unnecessarily complex language.

Particularly impressive to the modelers was their capability to generate working specifications that they could immediately see in the portal. In a similar vein, the sheer amount of code generated was considerable, 2000–4000 lines of code per product. This was almost completely a consequence of the input format used by the underlying

### SUMMARY

repository of the web portal. If the input format could have been changed and supported by additional framework code, the code generators would have been significantly simpler: now the generator produces a lot of MOF-related Java code needed just for the repository. Since the modeling language raised the level of abstraction, the company’s plan to move from Java to an XML generator was seen as a minor issue: just one software engineer would be needed to modify the generator to produce a different output format. When planning the change of the target code to XML, it was also soon realized that the main generator structure (Fig. 6.8) would be largely the same—excluding only the MOF specific parts.

6.8 SUMMARY

The case of insurance product specification has shown how a DSM solution can be defined to raise the level of abstraction beyond the technical programming domain to the business domain. With the created DSM solution, nontechnical domain experts can create specifications using terminology they already know and generate the implementation code completely. The DSM solution also allowed the company to start defining the insurance products even when the underlying platform was not yet in use.

Since the company had already defined a domain model and had sample implementations, the DSM solution was largely defined and implemented by an external consultant. The modeling language was co-designed together with the main users, but its formalization as a metamodel and implementation into a modeling language were done by the external consultant. The code generator was developed solely by the consultant. This was possible since the sample applications provided reference implementations and test material for generator development.