Faktor-IPS Tutorial

Part 1: Modeling and Product Configuration

Jan Ortmann, Gunnar Tacke
(Document version 1677)

Introduction

Faktor-IPS is an open source tool for model-driven development[1] of software systems targeted at the insurance business. It is geared towards the uniform representation of product knowledge. Faktor-IPS allows you not only to edit your business object models, but also to manage product information. As well as straight product data, individual product aspects can be defined by way of an Excel-like formula language. In addition, tables can be created and business test cases can be defined and executed.

The present tutorial provides an introduction to the concepts and usage of Faktor-IPS. Throughout the tutorial we will use an extremely simplified home contents insurance as an example. This simplistic business object model suffices to show the basic construction and modeling principles.

The tutorial is comprised of two parts:

1. Part 1 is an introduction to working with the modeling tool and the generated source code. We also explain how to configure specific products based on the model.
2. Part 2 covers the usage of tables and the implementation of insurance premium computation. Furthermore, you will see how to implement a flexible design of a home contents model[2] using formulas.

This tutorial is written for software architects and developers with a good working knowledge of object-oriented modeling with UML. Some familiarity with the development of Java applications with Eclipse would be useful, but is not indispensable.

If you don't want to follow along with each step of the tutorial, you can download the end result from www.faktorips.org and install it on your machine.

A further tutorial demonstrates how to work with the model classes created here in a practical application (Tutorial “Home Contents Offer System”).

In addition, a tutorial on the partitioning of large models is available. Specifically, it illustrates by way of example, the separation into different lines of business (lob) and how to separate lob-specific from cross-lob aspects.

Part I Overview

The first part of the tutorial is organized as follows:

- Hello Faktor-IPS

1 Model Driven Software Development (MDSD). An excellent description of the underlying concepts can be found in Stahl, Völter: Modellgetriebene Softwareentwicklung.

2 For simplicity and conciseness, we will refer to “home contents” simply by the term “home” throughout this tutorial.

In this chapter we will create our first Faktor-IPS project and define our first class.

- Working with the Model and Source Code
  This chapter uses a model of a home contents insurance in order to explain how to work with the modeling tool and the generated source code.
- Adding product aspects to the model
  In this chapter we will add product configuration aspects to the home contents model.
- Defining the Products
  Based on the model, we will define two home contents products. For this purpose, we will use the product definition view specifically designed for end users
- Runtime Access to Product Information
  This chapter explains how to access product information in a running application or test case.

“Hello Faktor-IPS”

First we will create a Faktor-IPS project, define a model class, and generate the Java source code of this class.

If you have not yet installed Faktor-IPS, you must do it now. The software and installation instructions are available at www.faktorips.org. In this tutorial we will use Eclipse 3.7 and Faktor-IPS 3.6.0 in English. Start Eclipse now. Ideally, you should use a separate workspace for this tutorial. If Faktor-IPS has been installed correctly, you will see the following toolbar icons when you open the Java Perspective:

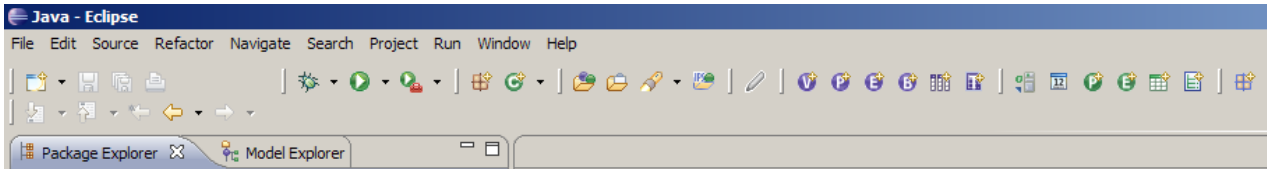

Figure 1: Faktor-IPS Icons

Faktor-IPS projects are regular Java projects with an additional Faktor-IPS Nature. First, create a new Java project named “HomeModel” by clicking the menu options File►New►Java Project. In the dialog simply enter the project name and click Finish.

In this project we will create the model classes. To add the Faktor-IPS Nature to the project, select it in the Java Package Explorer and choose Faktor-IPS ► Add IPS-Nature... in the dropdown menu.

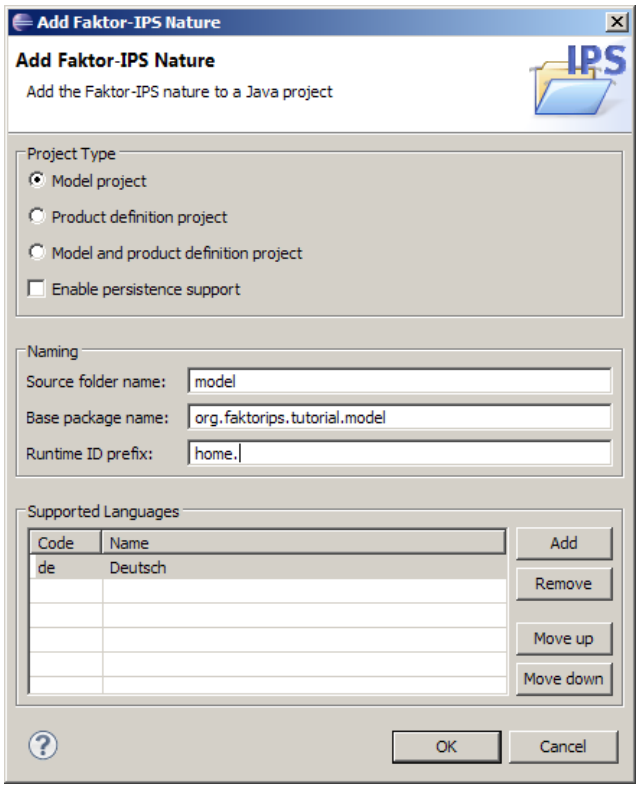

Figure 2: Adding the IPS-Nature

Enter “model” as the source folder name and “org.faktorips.tutorial.model” as the base package for generated Java classes, “home.” as runtime ID prefix and click OK. The Faktor-IPS runtime libraries will be added to your project and the designated source directory (“model”) will be created. The model definition will be stored in the source directory. Inside this directory, the model description can be organized in packages, just like in Java. As in Java, Faktor-IPS uses qualified names to identify the classes of the model. The meaning of the RuntimeID prefix will be explained in the chapter “Defining the Products“.

Furthermore, a new Java source code directory named “derived” has been added to the project. In this directory, Java will generate source files and copy XML files that are 100% generated. The contents of this directory can be deleted and regenerated anytime. By contrast, the original Java source directory contains files that can be edited by developers and merged when generated.

Before defining your first class named HomeContract, you have to configure your Project settings to automatically build your workspace (Project►Build automatically).

First, go to the Faktor-IPS Model Explorer right next to the Package Explorer.

If the Model Explorer is not visible, this might be because you have used your workspace prior to the installation of Faktor-IPS. In this case you must choose the menu option Window►Reset Perspective.

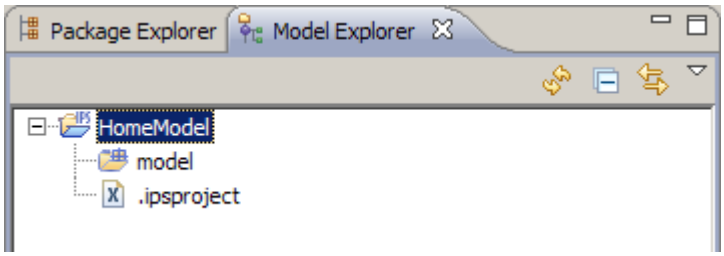

Figure 3: The Project View of the Model Explorer

The Model Explorer will display your model definition without the Java details.

The properties of the Faktor-IPS project are stored in the file “.ipsproject”. This includes, for example, the information entered just now in the AddIpsNature dialog box, settings in relation to the generating of code, permitted data types etc. The content is stored in XML and documented in detail in the file.

We will store our classes inside a package named “home”. To create IPS packages, right-click on the “model” source directory and use the dropdown menu to define a new IPS package named “home”.

Our next step is to create a class that will represent our home contract. Select your new package in the Package Explorer and press the toolbar button .

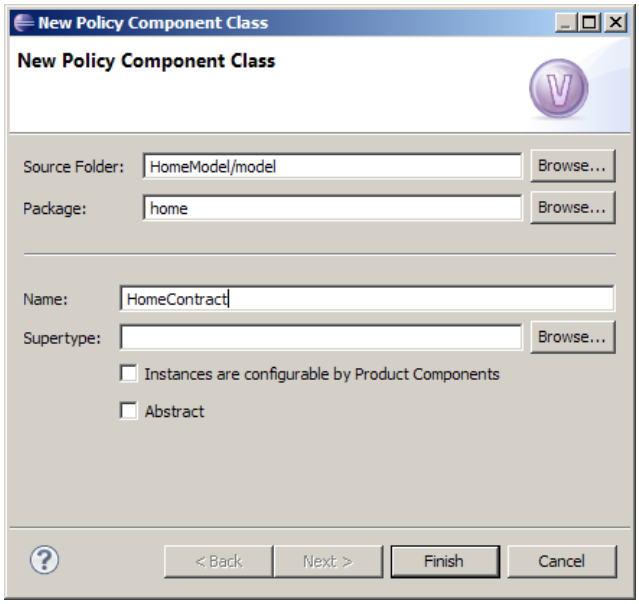

Figure 4: Creating a new Contract Class

In the dialog box, your source directory and package are already tagged with the appropriate names, so you just have to add the class name and click Finish. Faktor-IPS has now created the new class and opened the editor. When you switch back to the Package Explorer, you can see that the

HomeContract class resides in a separate file named "HomeContract.ipspolicycmpttype".

Furthermore, the Faktor-IPS source code generator will have created two Java source files : `org.faktorips.tutorial.model.home.IHomeContract` and `org.faktorips.tutorial.model.internal.home.HomeContract`.

The first file contains the so called published interface of the *HomeContract* model class. It defines all properties that will be visible and usable for clients of this model. Up to now, we have not defined any properties of the *HomeContract* class, so this interface does not exhibit any methods yet.

The *HomeContract* class is the implementation of the published interface internal to the model. A quick glance at the source code reveals that some methods have already been generated here, including methods for converting objects to XML and for validation.

The separation of published interface and implementation allows you to organize the model classes into different packages in such a way that the methods internal to the model are not disclosed to clients[3].

3 In Java, methods have to be public if they are called by classes that belong to other packages. It is not possible to differentiate if the same tier or another tier of the software architecture is concerned.

Working with the Model and Source Code

In this second step of our tutorial we will expand our model and work with the generated source code.

First, we will add an attribute named “paymentMode” to our *HomeContract* class. If the editor showing the contract class has been closed in the meantime, you can open it by double clicking on the class inside the Model Explorer. Within the editor, click on the New button in the Attributes section to open the following dialog box:

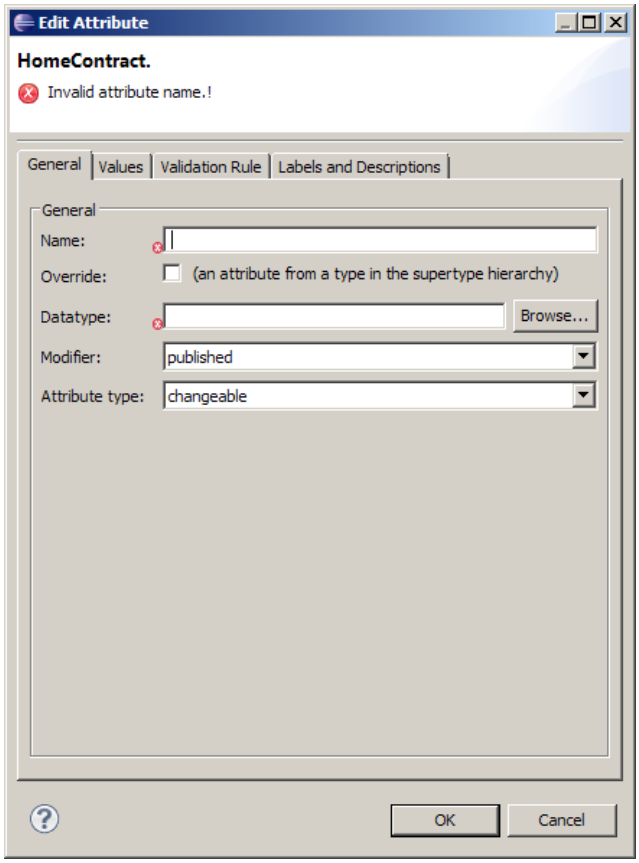

Figure 5: Dialog for Creating a New Attribute

The values have the following meanings:

Value	***Meaning***
Name	The name of the attribute.
Override	Indicates that the attribute has already been defined in a superclass, so that this class is just an override of certain properties (e.g. the Default Value)[4].
Datatype	The datatype of the attribute.

4 This corresponds to the `@override` annotation in Java 5.

Value	*Meaning*
Modifier	Similar to a Java modifier. The additional published modifier means that the property is included in the published interface.
Attribute type	The type of the attribute. • changeable Applies to changeable properties with getter and setter methods. • constant Applies to constant, immutable properties. • derived (cached, computation by explicit method call) Applies to a UML-style derived property. This property is calculated by an explicit method call and the result can then be queried by a getter method. For example, the property `grossPremium` can be calculated by a method named `computePremium` and subsequently retrieved by the `getGrossPremium()` method. • derived (computation on each call of the getter method) Applies to a UML-style derived property. This property is calculated each time the getter method is called. For example, the age of an insured person can be determined with each call of `getAge()` by means of this person's date of birth.

Enter “paymentMode” as the name and “Integer” as the datatype of the attribute. When you click the Browse button next to the text box, a list of available datatypes will open for you. Alternatively, you can use Ctrl-Space to perform an Eclipse-like auto completion. For example, if you enter “D” and press Ctrl-Space, you will see all datatypes beginning with “D”. You can leave the other text boxes at their default values, click OK, and save the contract class.

The code generator has already updated the Java source files. The published interface `IHomeContract` will now include accessor methods for the attribute and the `HomeContract` implementation will implement these methods and save the state in a private member variable.

```
/**
 * Member variable for paymentMode.
 *
 * @generated
 */
private Integer paymentMode = null;

/**
 * {@inheritDoc}
 *
 * @generated
 */
public Integer getPaymentMode() {
    return paymentMode;
}

/**
 * {@inheritDoc}
 *
 * @generated
 */
public void setPaymentMode(Integer newValue) {
    this.paymentMode = newValue;
}
```

The JavaDoc for this member variable and its getter method will be tagged as `@generated`, meaning that the method is 100% auto generated. With each new Adjustment, this code will be created in exactly the same way, even if it has been deleted or modified within the file in the meantime. This means that modifications made by the developer will be overridden. If you want to modify the method, you have to add the word `NOT` to the `@generated` annotation.

Let us try this out. Add one line to both the getter method and the setter method and add `NOT` behind the annotation of the `setPaymentMode()` method, like this:

```
/**
 * {@inheritDoc}
 *
 * @generated
 */
public Integer getPaymentMode() {
    System.out.println("getPaymentMode");
    return paymentMode;
}

/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public void setPaymentMode(Integer newValue) {
    System.out.println("setPaymentMode");
    this.paymentMode = newValue;
}
```

Now re-generate the source code of the *HomeContract* class. As is customary in Eclipse, you can do this in two ways:

- You build the entire project using Project▶Clean, or
- You save the model description of the *HomeContract* class again.

When the Adjustment has been completed, the `System.out.println(...)` has been removed from

the getter method while it is still present in the setter method.

Methods and attributes that have been added are maintained throughout the Adjustment process, so you can extend your source code as you wish.

Now we will extend the model definition of the mode of payment by adding the allowed values. To do this, you must open the edit dialog for attributes and go to the second tab page. Up to now, all values of the indicated data type have been accepted as legal attribute values. We will now limit this to the values 1, 2, 4, 12 meaning monthly, quarterly, bi-annually, and annually, respectively. Change the type to “Enumeration” and enter the values 1, 2, 4, and 12 into the table[5].

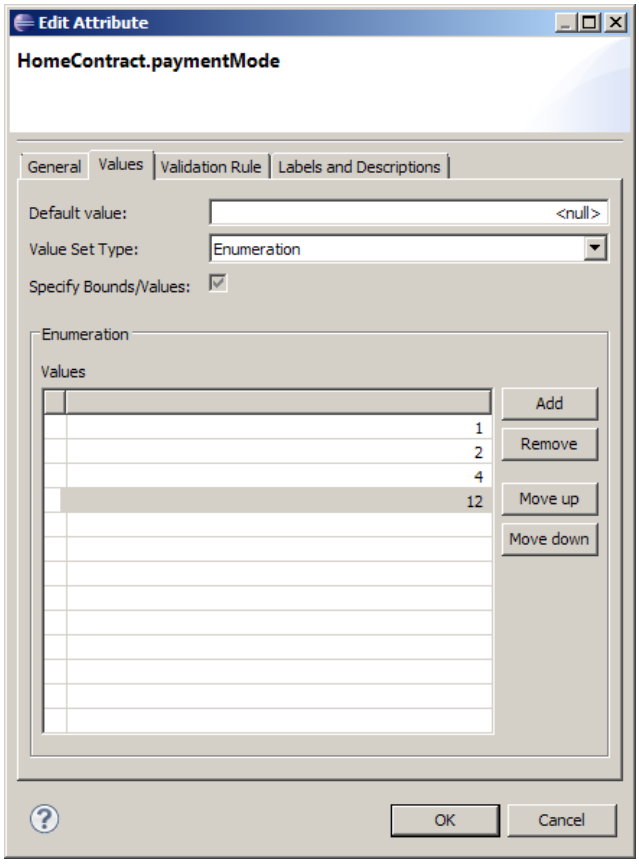

Figure 6: Determining Legal Values for an Attribute

Now set the Default Value to 0. Faktor-IPS will mark the Default Value with a warning, because that value is not included in the set of legal values. Consequently, this could indicate an error in the model. We will leave it this way for the time being, because it will give us an opportunity to examine the Faktor-IPS error handling. Close the dialog box and save the contract class. The same warning message as in the dialog now appears within the Eclipse Problems View. Faktor-IPS permits errors and inconsistencies in the model, just informing the user that a problem has been encountered. As in Eclipse, this information is conveyed in the editors and by means of so called problem markers that appear in the Problem View and in the Explorers.

5 In addition, Faktor-IPS supports the definition of Enums, though we will not use this feature here. You can also use an extension point to register any Java classes as data types. These Java classes should be implemented as ValueObject.

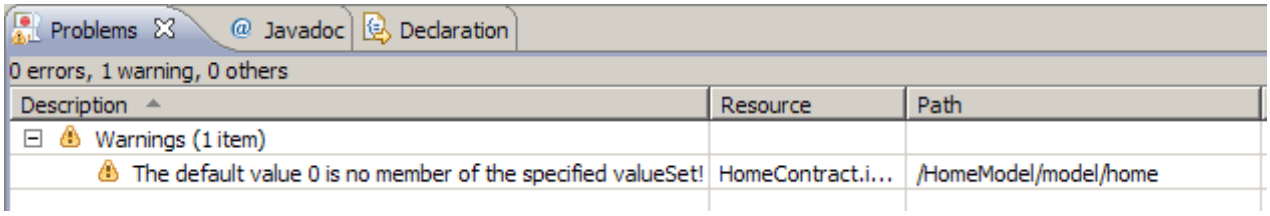

Figure 7: Error Warnings Inside the Problems View

Now set the default value to <null> again and save the contract class. This way, the warning will be cleared from the Problems View.

Faktor-IPS will generate a warning instead of an error, because in some cases it can make sense to provide a default value that is not included in the range of legal values. This is especially true for a default value of `null`*. For example, if a new contract is created, it might be desirable not to preset a mode of payment and just leave this field at a null default in order to force the user to enter a mode of payment. Only when the contract is eventually completed, the condition saying that the paymentMode property must contain a value from the legal range of values must be met.*

At the end of this chapter we will now define a class named *HomeBaseCoverage* and determine the composition relationship between *HomeContract* and *HomeBaseCoverage* according to the following diagram:

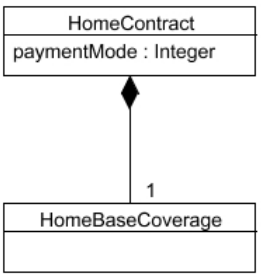

Figure 8: Contract model

First, you have to create the *HomeBaseCoverage* class in line with the *HomeContract* class. Then you go back to the *HomeContract* class and open it in the editor. To start the wizard for creating new relationships, you have to click the New button at the right-hand side next to the Associations section.[6]

6 In accordance with UML, Faktor-IPS uses the term “association”. In the text, however, we prefer the term “relationship” that is more common in general language usage.

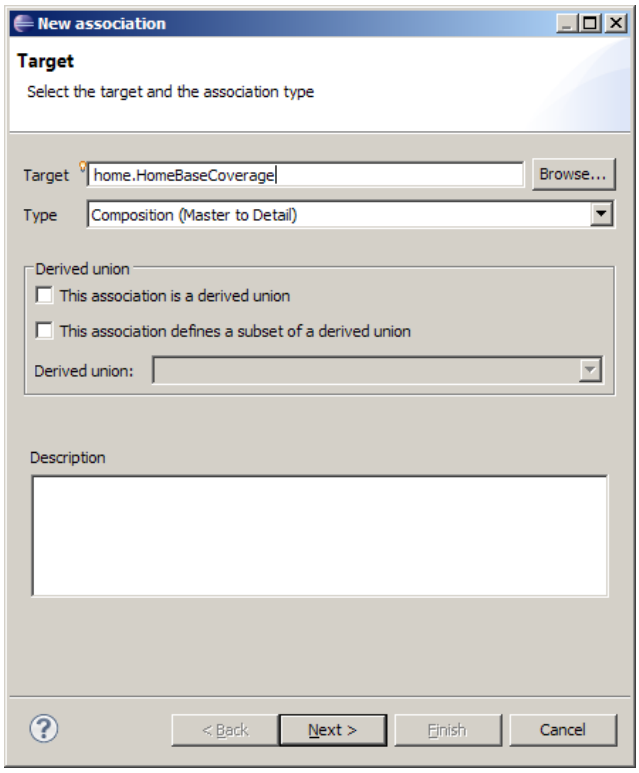

Figure 9: Creating a new Relationship

Your target will be the recently created *HomeBaseCoverage* class. Again, you can use the auto completion functionality with Ctrl-Space. At this point, please ignore the text box named Derived Union. This concept will be dealt with in the tutorial on model partitioning.

On the following page, enter 1 as both minimum and maximum cardinality and name the roles HomeBaseCoverage and HomeBaseCoverages, respectively. The plural is used so that the code generator can create comprehensible source code.

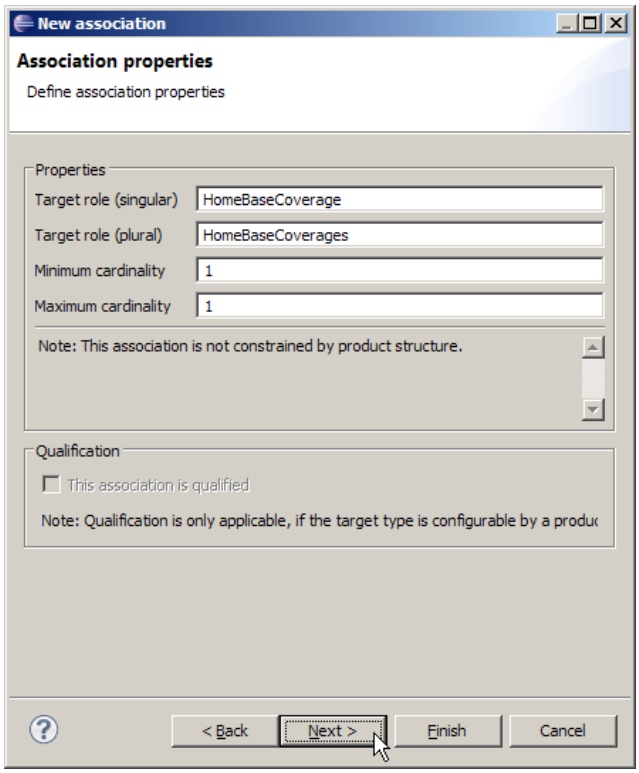

Figure 10: Role Names and Cardinalities in a Relationship

On the next page you can choose whether there is to be a backward relationship between *HomeBaseCoverage* and *HomeContract*. Relationships in Faktor-IPS are always directed, so it is possible to allow navigation in only one direction. Choose New inverse association and go to the next page. Just enter the role identifier, click Finish to establish both relationships (forward and backward), and save the *HomeContract* class. When you look at the *HomeBaseCoverage* class you will see that the backward relationship has been added.

Finally, we will take a quick look at the generated source code. Inside the published interface `IHomeContract`, methods have been created to add basic coverage to the *HomeContract*, etc. The interface `IHomeBaseCoverage` includes a method to navigate to the *HomeContract*, where the *HomeBaseCoverage* resides. Hence, composition relationships are always created (and resolved) through the container class. If the model defines both a forward and a backward relationship, both directions are taken into account. This means that if `setHomeBaseCoverage(IHomeBaseCoverage cov)` is called on a `contract` instance of *HomeContract*, `cov.getHomeContract()` will return `contract` again. This will be clear if you take a look at the implementation of the `setHomeBaseCoverage`[7] method within the `HomeContract` class.

7 From now on, generated Javadocs will no longer be shown in this tutorial as the methods are explained in the text.

```
public void setHomeBaseCoverage(IHomeBaseCoverage newObject) {
    if (homeBaseCoverage != null) {
        ((DependentObject) homeBaseCoverage).setParentModelObjectInternal(null);
    }
    if (newObject != null) {
        ((DependentObject) newObject).setParentModelObjectInternal(this);
    }
    homeBaseCoverage = (HomeBaseCoverage) newObject;
}
```

Inside the coverage, the contract will be set up as the contract to which the coverage belongs (second if-statement of the method).

Extending the Home Contents Model

In this section we will expand our home contents model. Figure 11 shows the model as is.

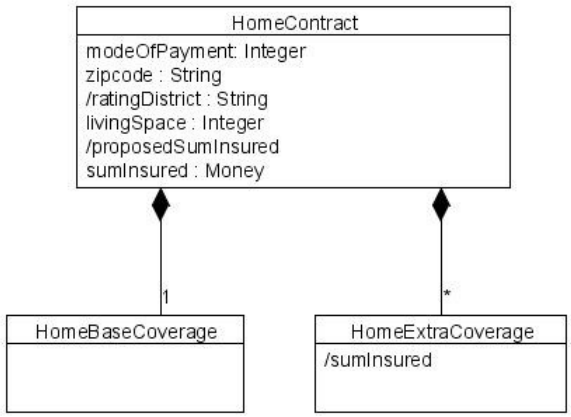

Figure 11: Home Contents Model with Base Coverage and Extra Coverage

Each *HomeContract* must include precisely one base coverage and any number of extra coverages. The base coverage always covers the sum insured according to the contract. In addition, a *HomeContract* can include optional extra coverages, typically covering risks like bicycle theft or overvoltage damage. We will cover these in the second part of our tutorial and focus on the *HomeContract* and *HomeBaseCoverage* for the time being.

Let us open the *HomeContract* class and define its attributes:

Name : Datatype	***Description, Comments***
zipcode : String	The zipcode of the insured home contents
/ratingDistrict : String	The rating district (I, II, III, IV, or V) depends on the zipcode and determines the insurance rate. => Make sure to set the AttributeType to derived (computation on each call of the getter method) !
livingSpace : Integer	The living space of the insured home contents in square meters. Allowable values range from min=0 to unlimited. The value range is defined on the second dialog page. If you want the value range to be unlimited, set the max field to <null>.
/proposedSumInsured : Money	A suggested value for the sum insured. It is determined based on the living space. => Make sure to set the AttributeType to derived (computation on each call of the getter method)!
sumInsured : Money	The sum insured. Allowable values range from min=0 EUR and max=<null>.

The editor showing the *HomeContract* class should now look as follows:

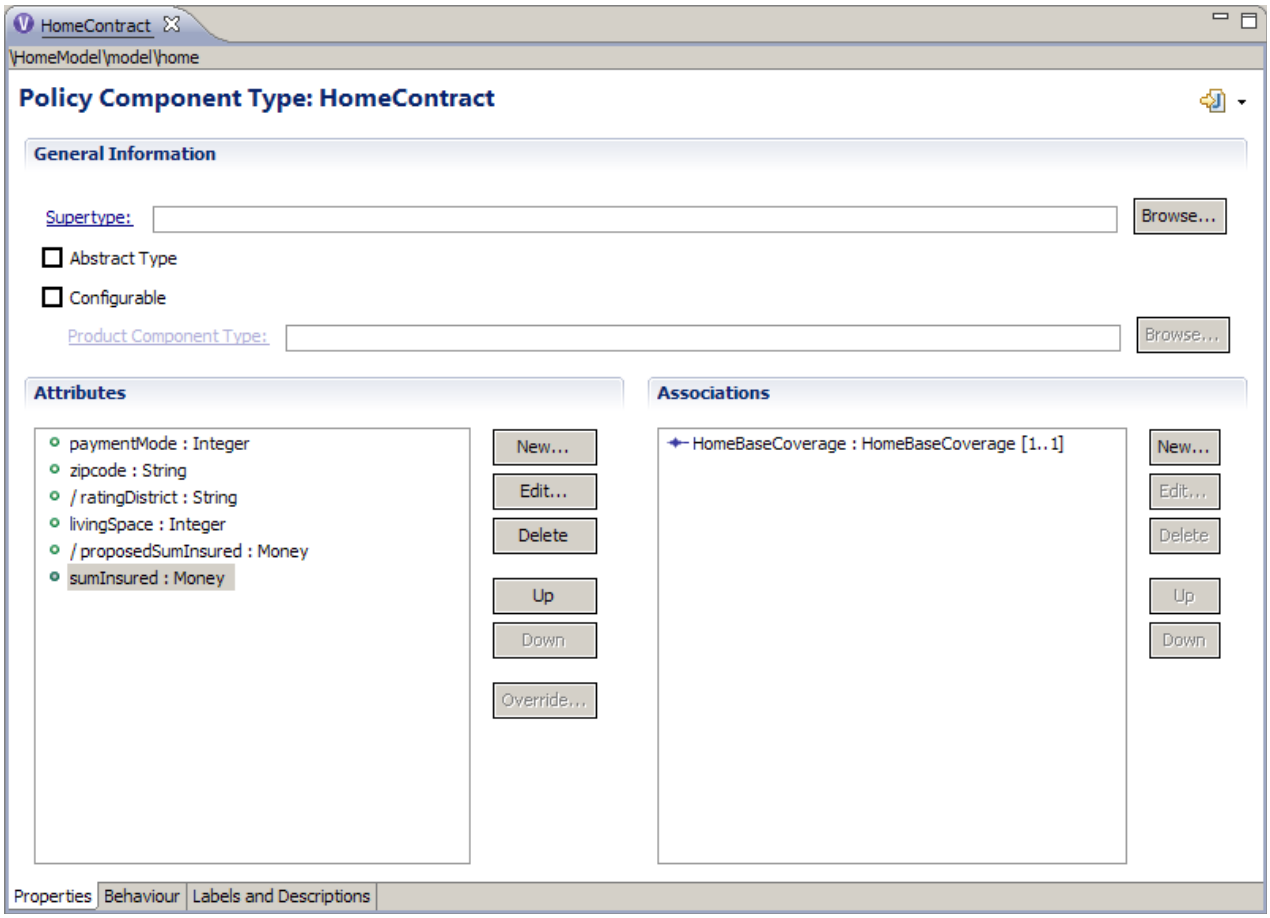

Figure 12: The HomeContract Class

The derived attributes are prefixed by a slash according to the UML notation.

Next, open the `HomeContract` class in the Java Editor and implement the getter methods for the derived attributes "ratingDistrict" and "proposedSumInsured" as follows. Remember to add the word `NOT` after `@generated`, so the code inserted manually will not be overwritten!

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public String getRatingDistrict() {
    return "I"; // TODO later we'll implement this with a table lookup
}
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getProposedSumInsured() {
    // TODO: later we'll implement this with a product data lookup
    return Money.euro(650).multiply(livingSpace);
}
```

Adding Product Aspects to the Model

Now we will finally start to model the product aspects. Before we do this with Faktor-IPS, we will discuss the design at the model level.

Let us have a look at the properties defined for our *HomeContract* class so far and consider which aspects of these properties should be configurable in an insurance product:

Properties of HomeContract	***Configuration options***
paymentMode	The payment modes permitted in the contract. The default value for the payment mode upon creation of a new contract.
livingSpace	The range (min, max) of the living space.
proposedSumInsured	Definition of a default value per square meter of living space. The proposed sum insured will then be computed by multiplying this value by the living space[8].
sumInsured	The value range of the sum insured.

We will create two home contents products. *HC-Optimal* will offer a comprehensive insurance coverage, while *HC-Compact* provides a basic insurance at low cost. The following table shows the properties of both products with respect to the above configuration possibilities:

Configuration Option	***HC-Compact***	***HC-Optimal***
Default paymentMode	annually	annually
Allowed paymentModes	bi-annually, annually	monthly, quarterly, bi-annually, annually
Allowed range of living space	0-1000 sqm	0-2000 sqm
Proposed sum insured per square meter of living space	600 Euro	900 Euro
Sum insured	10Tsd – 2Mio Euro	10Tsd – 5Mio Euro

We represent this in the model by introducing a class named *HomeProduct*. This product contains all properties and configuration possibilities that have to be identical for home contracts based on the same product. Both *HC-Optimal* and *HC-Compact* are instances of the *HomeProduct* class. The model is shown in the following UML diagram:

8 Alternatively, we could implement this configuration using a formula to compute a proposed sum insured. But we will focus on the factor for a start.

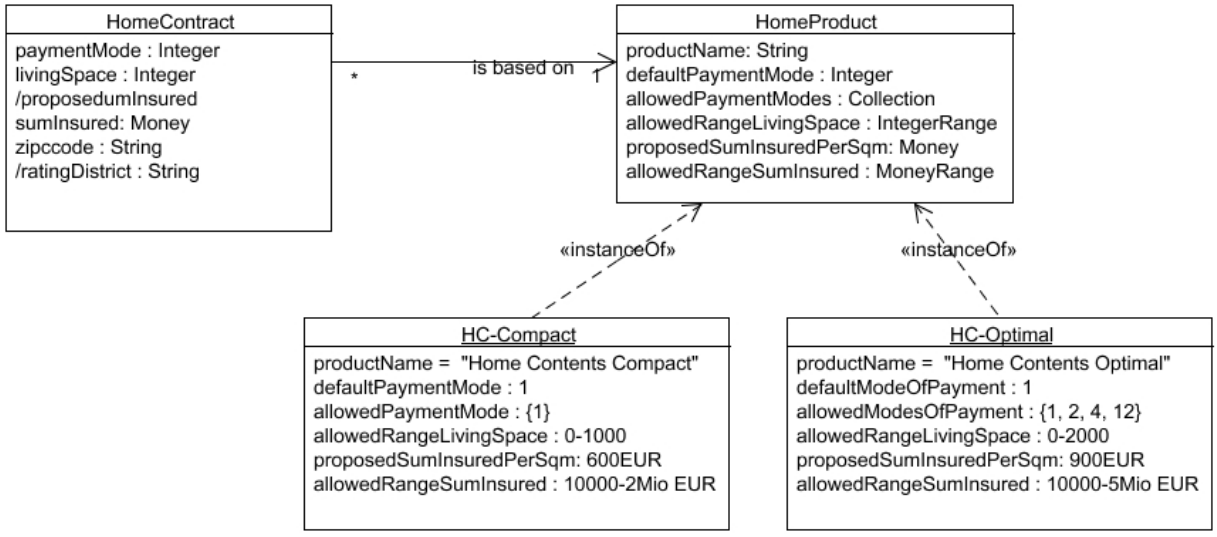

Figure 13: Home Contents Model with Product Classes. Changes Over Time are Not Considered.

But in the real world, products change over time. With insurance products, two common types of change can be distinguished[9]:

Generations

Generations are released so as to be able to offer different conditions for new contracts. Contracts can be signed according to the current generation's terms and conditions during that generation's sales period.

Existing contracts will not be affected by the introduction of a new generation, unless an explicit product (generation) change takes place.

At any one point in time, there is usually one valid generation for new contracts; this is the generation whose sales period is running. It is possible, however, that multiple generations of the same product are on sale at the same time. In practice, this occurs during transition from an old generation to a new one. During this time, both generations are sold, as shown in the following figure:

9 Unfortunately there is no uniform naming convention for these two types of product change. This tutorial uses the terms Adjustment and Adjustment. In Faktor-IPS, this naming convention can be configured both for the GUI (Window►Preferences: Faktor-IPS: Naming scheme for changes over time) and for the generated source code (see ".ipsproject" file in section GeneratedSourcecode).

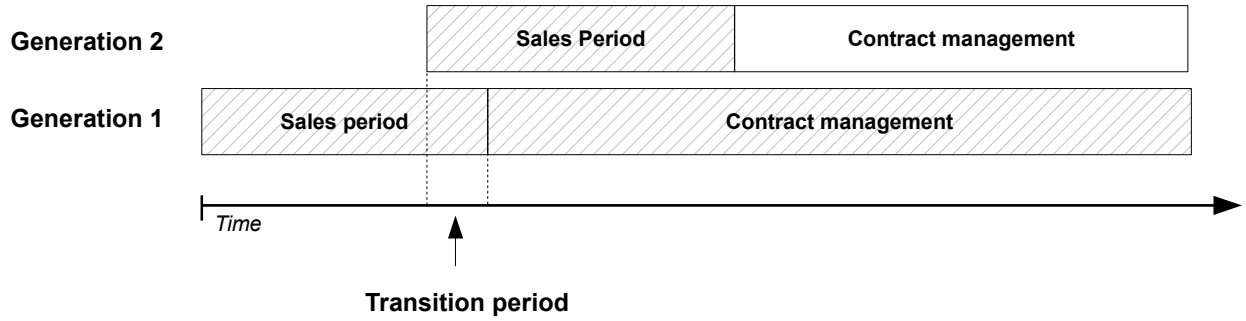

Figure 14: Product Generations

Adjustment

An adjustment is a change that should be valid for all contracts that are based on a particular generation. During the validity period of an adjustment, no changes can be made. At each given time, there is no more than one valid adjustment of a product generation. The following figure shows the relationship between generations and adjustments.

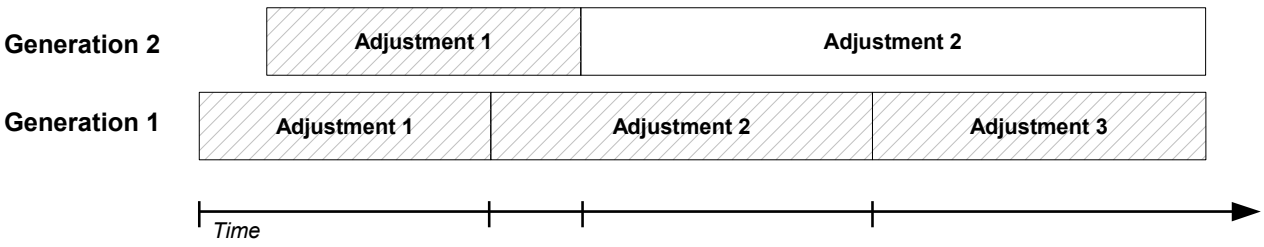

Figure 15: Product Generations and their Adjustments

How can the model account for the changes over time? We assume that any property can change over time, so the properties modeled above belong to the product adjustment rather than to the home product. The following figure shows an overview of the model.

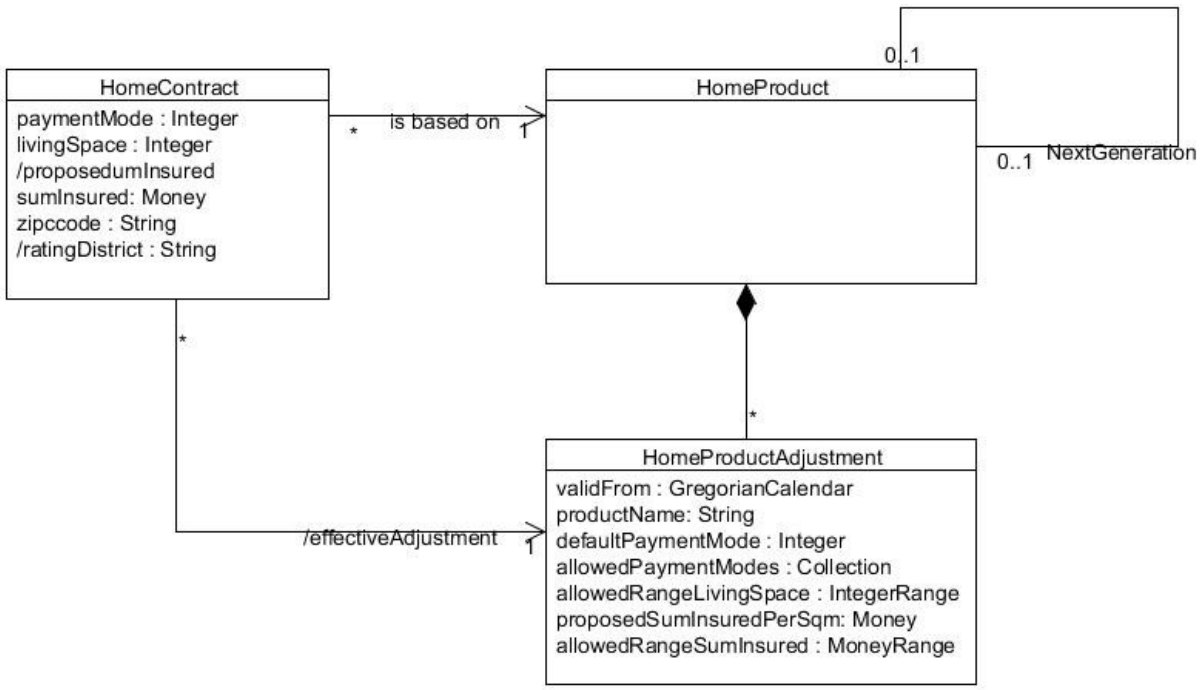

Figure 16: Our Home Contents Model now accounts for Changes Over Time

We will map the concept of a product generation using a relationship between products, because the distinction between a new product and a new generation is very fuzzy. Hence, the question of whether switching a car insurance to a scoring tariff constitutes a new generation or a new product is rather an academic one[10].

If the contract needs to access product properties, a date is required to determine the current product Adjustment. In a policy management system, this is the effective date at which the contract comes into force, but in a sales and service system, it can just as well be the inception date of the insurance. In this tutorial, we will use an attribute named effectiveFrom at the *HomeContract* class. Strictly speaking, the *HomeContract* class thus actually represents a contract Generation that shall be taking effect as of this date (until the next contract Generation)[11].

10 The “nextGeneration” relationship is not limited to instances of the same class, so it is possible to declare a structurally very different product (e.g. a new scoring tariff) the successor to an existing product.

11 The contract itself will hardly ever be used; it just serves as a means of grouping and identifying the contract states. For this reason, we will use the short form “contract” to refer to the contract state. You can, of course, handle this your own way.

After all this theory, let us now add product classes to our model in Faktor-IPS. First, we will define the *home.HomeProduct* class. To do this, click the toolbar button . When the wizard opens, enter the name of the new class ("HomeProduct") and, in the Policy component class field, enter *home.HomeContract*. When you click Finish, the editor for product classes will open for you.

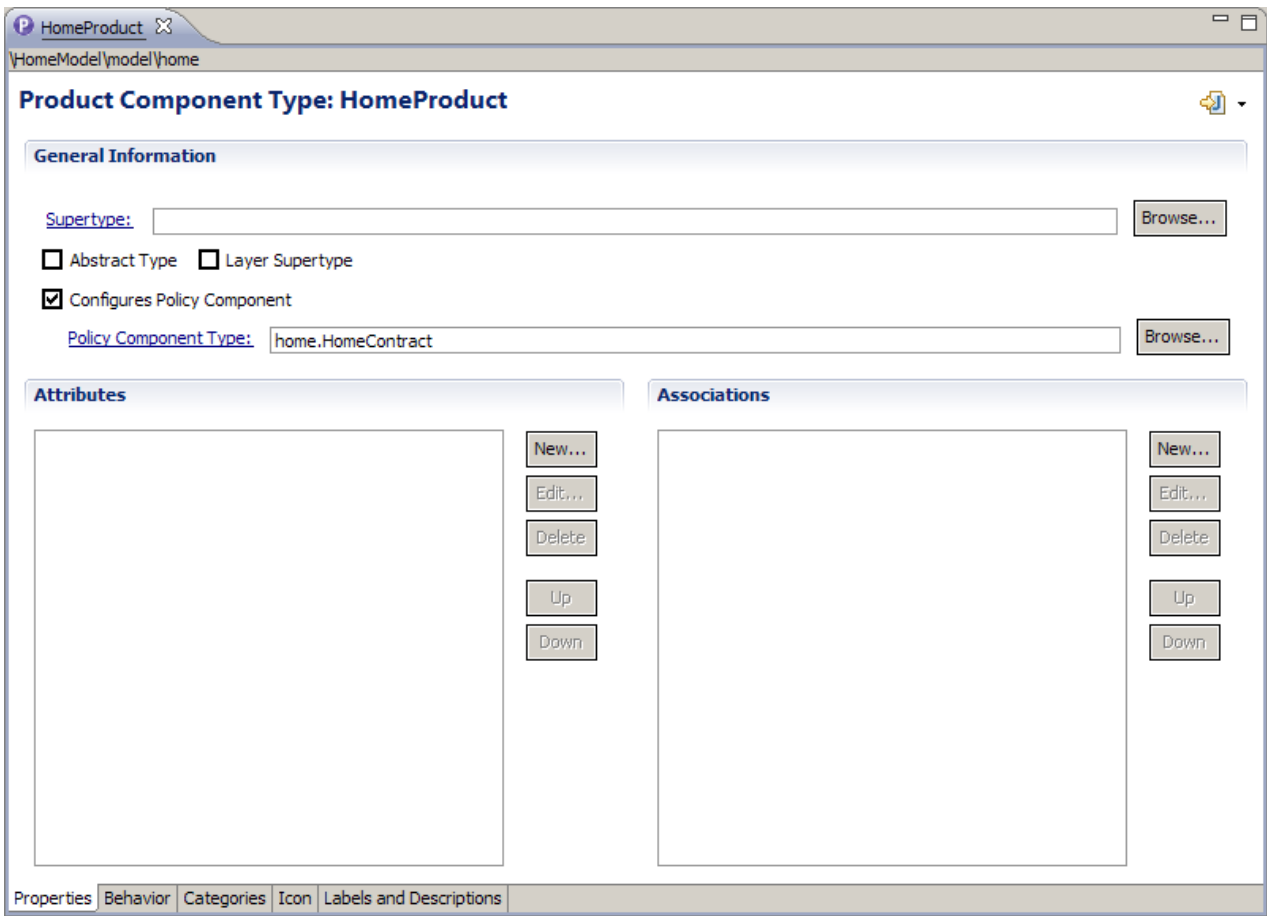

Figure 17: Editor for Product Classes

In the General Information section we can see that the *HomeProduct* class configures the *HomeContract* class. This corresponds to the information we provided before. Otherwise the first editor page is structured similarly to the contract class editor[12].

At the same time, Faktor-IPS has generated the published interfaces `IHomeProduct` and `IHomeProductAdj` as well as the respective implementation classes `HomeProduct` and `HomeProductAdj`. The published product interface contains a method to determine the valid adjustment for each given date.

The following aspects should be configurable in the *HomeProduct* class:

- the product name and
- the legal paymentModes as well as the default paymentMode.

Let us start with the product name. Create a new String attribute named "productname", just like

12 Within Preferences you can choose if you want to get all information about a given class on one page or on two pages.

you would create a contract class attribute. As with contract class attributes, the legal values can be limited using ranges or enumerations, but we will not use this option for our product names. The dialog box is shown in the following figure.

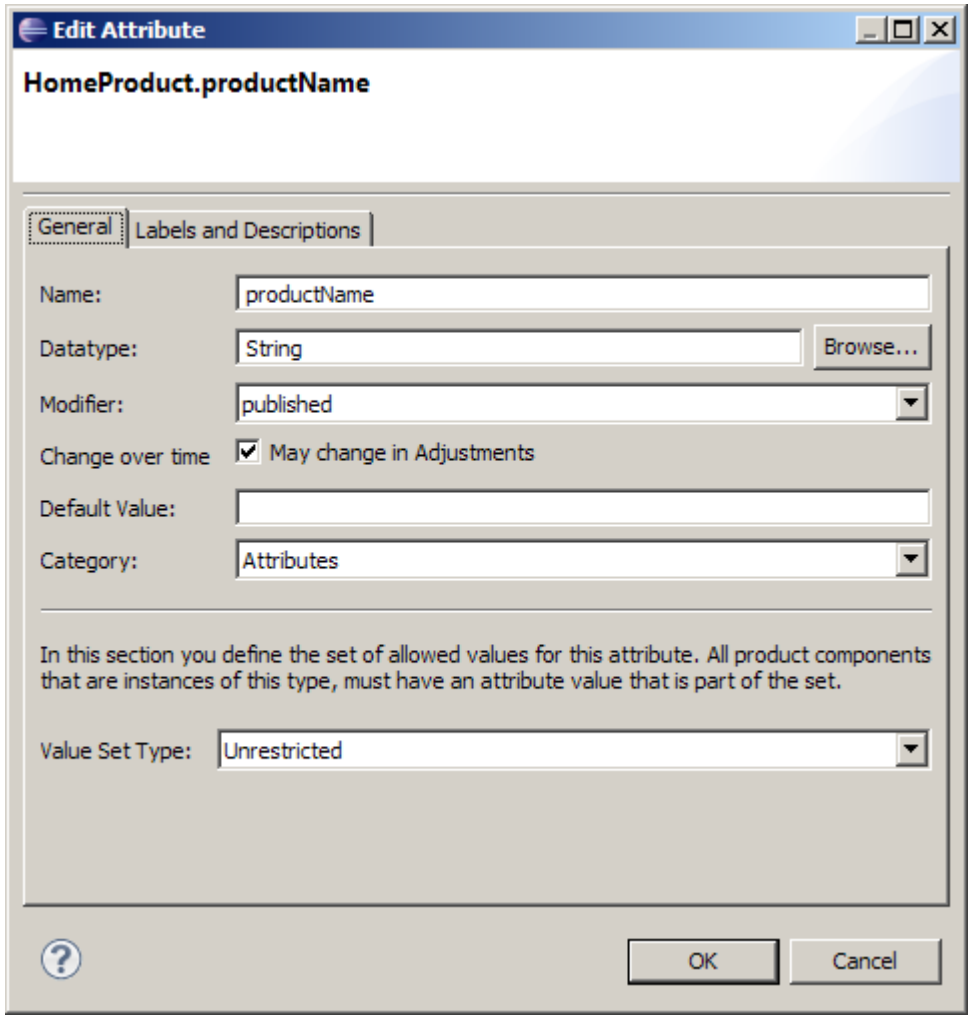

Figure 18: Dialog for Editing Product Attributes

Now, we will define that the allowable modes of payment for a *HomeContract* and the default paymentMode can be configured within the product. To do this, we will first open the editor for the HomeContract class. In the General information section, the wizard has stated that the *HomeContract* class is configurable by the *HomeProduct* class.

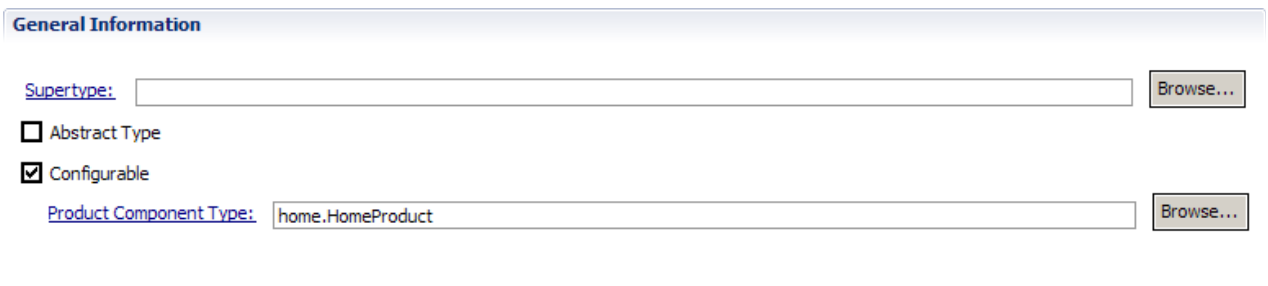

Figure 19: The Editor General Information Section for the HomeContract Class

Next, open the dialog box for editing the "paymentMode" attribute. Since contracts are now declared as configurable, the dialog box includes a new Configuration section. In this section you can determine whether and how each attribute can be configured. Depending on the attribute type, there are various ways to do so.

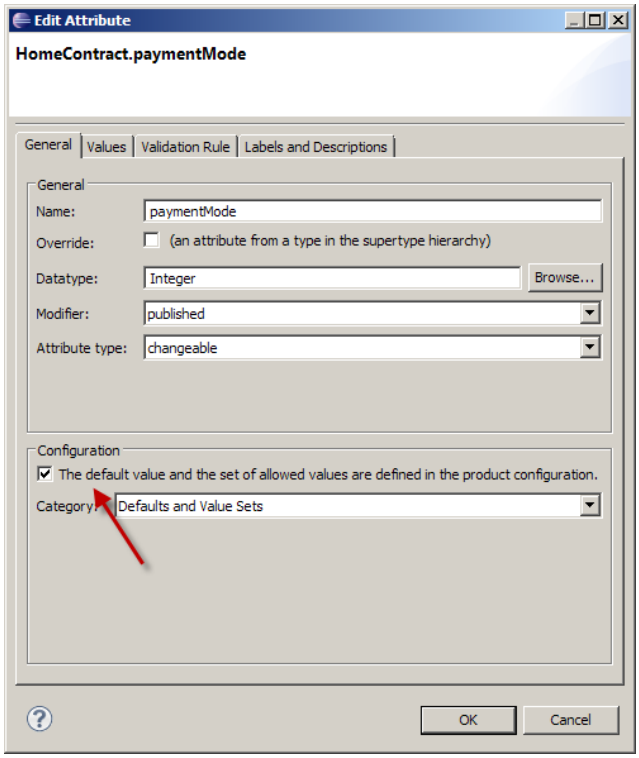

Figure 20: Dialog for a Configurable Contract Class Attribute

To be able to define the valid paymentModes and the default paymentModes within the product, you have to mark the appropriate checkbox. Now close the dialog box and save your settings.

Let us now have a look at the source code. The product adjustment published interface now contains methods to retrieve the product name, the default paymentMode, and the allowable values for paymentMode, respectively.

```
public String getProductName();

public Integer getDefaultPaymentMode();

public OrderedValueSet<Integer> getAllowedValuesForPaymentMode(IValidationContext context);
```

In the HomeContract published interface `IHomeContract`, methods to access the product and the underlying product Adjustment of the contract have been created.

```
public IHomeProduct getHomeProduct();

public void setHomeProduct(
        IHomeProduct homeProduct,
        boolean initPropertiesWithConfiguredDefaults);

public IHomeProductAdj getHomeProductAdj();
```

Finally, we will create the contract attribute “effectiveFrom” of type GregorianCalendar. This attribute is not configurable on the product side. How, then, can the contract class know that it has to use the “effectiveFrom” attribute to determine which adjustment is valid. To this end, Faktor-IPS has already generated the following method inside the HomeContract class:

```
/**
 * {@inheritDoc}
 *
 * @generated
 */
@Override
public Calendar getEffectiveFromAsCalendar() {
        return null; // TODO Implement access to effective from.
        // To avoid that the generator overwrites the implementation,
        // you have to add NOT after @annotation in the Javadoc!
}
```

This method will be invoked by `getHomeProductAdj()` to get the effective date needed to determine the product adjustment[13]. In this case we will just return effectiveFrom (and add the word `NOT` after the `@generated` annotation):

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Calendar getEffectiveFromAsCalendar() {
  return effectiveFrom;
}
```

The published interfaces `IHomeProduct` and `IHomeProductAdj` are implemented in such a way that the product data will be available at runtime. However, the details of this would go beyond the scope of this tutorial.

The `getEffectiveFromAsCalendar()` method must only be implemented in the HomeContract class. In dependent classes like e.g. Coverages, the `getEffectiveFromAsCalendar()` method in the respective superclass is called by default.

13 So this is a TemplateMethod (in the GoF Pattern's sense), that is defined in the abstract PolicyComponent base class.

Our last step is to mark the attributes “livingSpace” and “sumInsured” as configurable, as we did it before with “paymentMode”.

Next, we will review our computation of the proposed sum insured. In chapter 4, we implemented the `getProposedSumInsured()` method of the `HomeContract` class like this:

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getProposedSumInsured() {
    // TODO: later we'll implement this with a product data lookup
    return Money.euro(650).multiply(livingSpace);
}
```

As a next step, we want to be able to configure the multiplier for the home product. To do this, we first add a new attribute named “*proposedSumInsuredPerSqm*” of the type *Money* to the *HomeProduct* class. This is the suggested value per square meter of living space. After saving the *HomeProduct* class, Faktor-IPS has generated the appropriate getter method `getProposedSumInsuredPerSqm()` to the `HomeProductAdj` class. We will now take advantage of this getter to compute our proposal for the sum insured. Customize the source code in the `HomeContract` class as follows:

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getProposedSumInsured() {
    IHomeProductAdj gen = getHomeProductAdj();
    if (gen==null) {
        return Money.NULL;
    }
    return gen.getProposedSumInsuredPerSqm().multiply(livingSpace);
}
```

Let us now define the “product side” of our model for the base coverage. To do this we mark the class *HomeBaseCoverage* as “configurable”. Our new class for this purpose will be named *HomeBaseCoveragesType (see Figure 21)*. For this class, you define an attribute named “name” of type String. In the course of this tutorial, we will further extend this class.

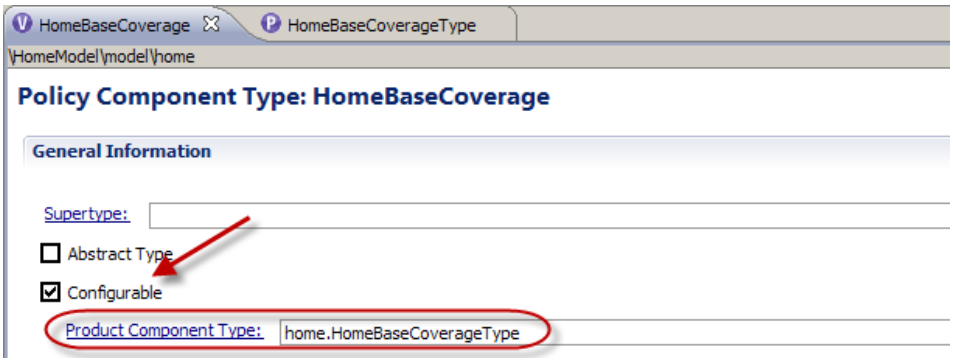

Figure 21: HausratBaseCoverage is configured by HomeBaseCoverageType

At the end of this chapter, we will consider the relationships between the classes on the product side. By means of these relationships, we want to capture which (home) coverage types are included in which (home) products. The model is depicted in the following UML diagram:

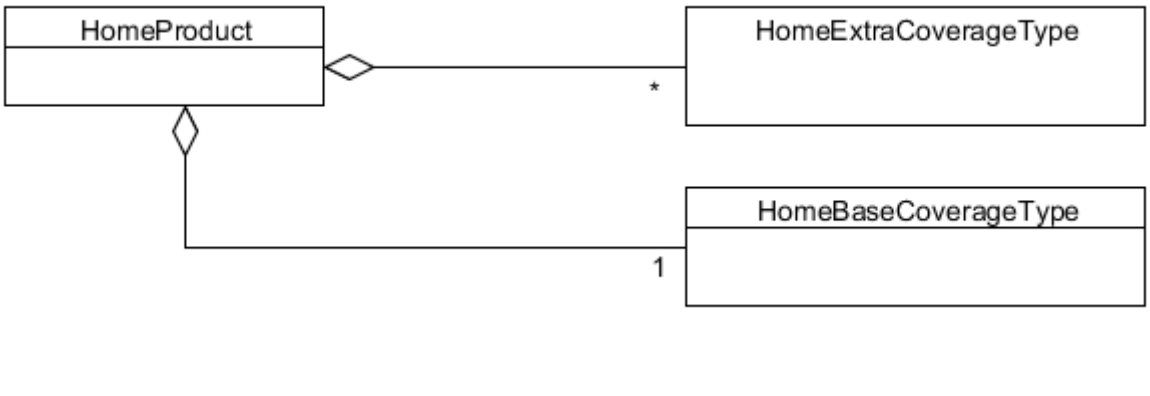

Figure 22: Model of the Product Configuration Classes

The home product uses precisely one base coverage type and any number of extra coverage types. Conversely, one base coverage type or one extra coverage type can apply to any number of home products. The primary navigation always goes from home product to coverage type (base or extra), but not in the other direction, because a coverage type should never depend on the products that use it.

Finally, let us define the relationship between *HomeProduct* and *HomeBaseCoveragesType* in Faktor-IPS. To do this, open the editor for the *HomeProduct* class and click New in the Associations section to create a new relationship. The following dialog box will open for you; please insert the same values as shown in the figure. Note that you must set the maximum cardinality to one. The extra coverage type will be created in part 2 of the tutorial.

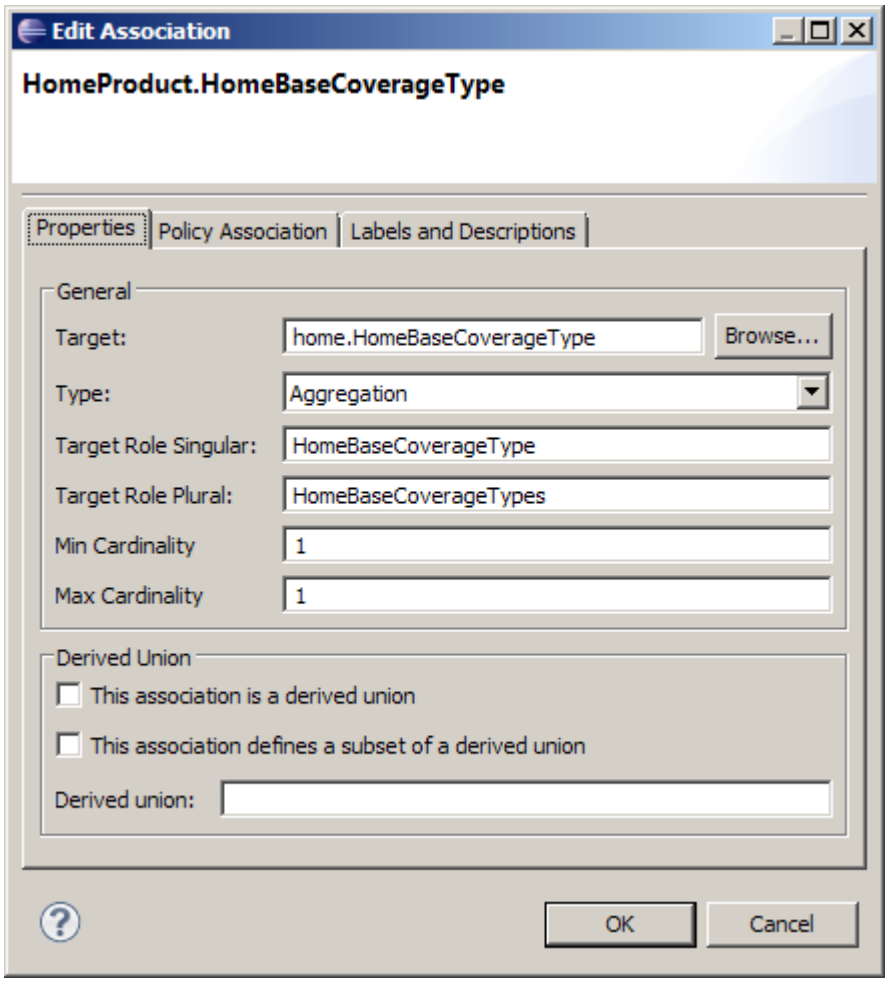

Figure 23: Dialog box for the Relationships between Product Classes

Defining the Products

In this chapter we will define our products *HC-Optimal* and *HC-Compact* in Faktor-IPS. For this purpose, we will use the product definition perspective specifically designed for the end users.

First, you have to create a new project named “HomeProducts” with a source directory “productdata”. To do this, create a new Java project and then call Add IpsNature in the Package Explorer. This time, choose the type Product Definition Project, package name “org.faktorips.tutorial.productdata”, and use “home.” as a Runtime-ID Prefix. Make sure to enter the full stop (.) at the end of the prefix. For each of the new product components, Faktor-IPS will define an Id with which to identify the product component at runtime. Per default, this RuntimeId is composed of the prefix, followed by the (unqualified) name of the product component[14]. The qualified name is not used for identification purposes at runtime, because the package structure only serves to organize the product data at development time. This way the product data can always be refactored without affecting the operational systems that might be involved.

We will manage the product data in a separate project to account for the fact that the responsibility for these product data might well be held by a different team and might have different release cycles. The product definition team could, for example, design and release a new product named *HC-Flexible* without making changes to the model. In order for the classes of the HomeModel to be available to the new project, a reference to the HomeModel project needs to be defined in the product definition project in Faktor-IPS. In Faktor-IPS this is achieved in a way which corresponds to the definition of the build path in Java. The dialog box which allows the definition of the Faktor-IPS build path is structured in the same way as the corresponding dialog for the Java build path and is accessed via the product properties. The following image shows the dialog box for the Faktor-IPS build path of the product definition project with reference to the HomeModel project.

The build path is stored in the “.ipsproject” file in the XML-element IpsObjectPath. Once the reference has been applied the element contains the following new entry:

<Entry type="project" referencedIpsProject="HomeModel"/>

As well as source directories and project references Faktor-IPS (again, same as Java) also supports archives/libraries in the build path. Faktor-IPS archives can be created, in the same way as JARs, through an export wizard.

14 In an upcoming Generation of Faktor-IPS there will be an extension point so you can implement your own way of assigning a RuntimeId.

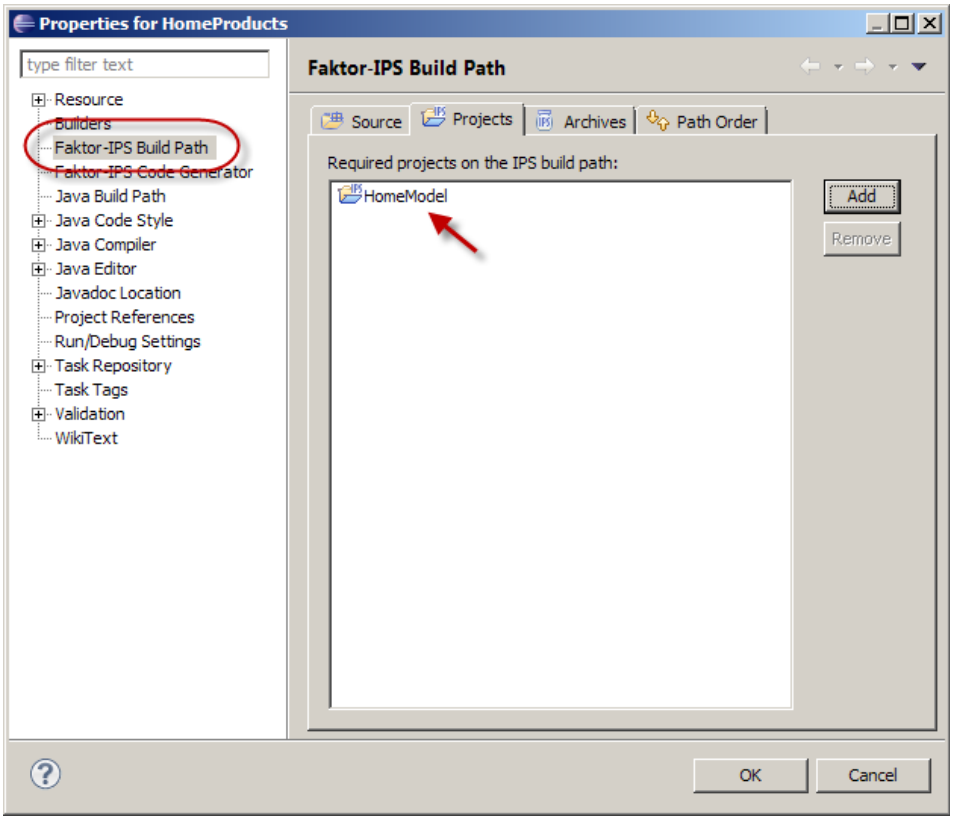

Figure 24: Dialog box for the Faktor-IPS build path

Next, you have to add the HomeModel Java project to your Java Classpath in order to make the Java classes available to the project.

As a first step, open the product definition perspective by clicking "Window►Open Perspective►Other" and choosing Product Definition from the list of available options[15]. If you have any open editors, close them now in order to have the end users' view of the system. In the Problems View, you have to deactivate all filters except for the Faktor-IPS filter (usually, there should be at least the default filter) so that only Faktor-IPS markers remain visible in that view (but no Java markers, etc.).

Initially, we will create two IPS packages; one for the products and one for the coverages. As in the Java perspective, this is done either via the dropdown menu or the toolbar.

You can also create any additional directories such as a doc directory for managing documents pertaining to the products.

We want our products to be available from the beginning of the next quarter, so before we begin to create the products, we should set the effective date we want to work with. Click the toolbar button , enter the start date of the quarter, and click OK. Any modifications from now on will be performed with this effective date.

First, we will create the HC-Optimal product. Select the above created package named "products"

15 There is a special installation option (in Eclipes terms: a special product) for business users wishing to use Faktor-IPS. This installation provides only the product definition perspective.

and click the toolbar button . A wizard for creating new product components will open. The wizard lists those product classes available in the model, for which you can create product components. Select *HomeProduct*.

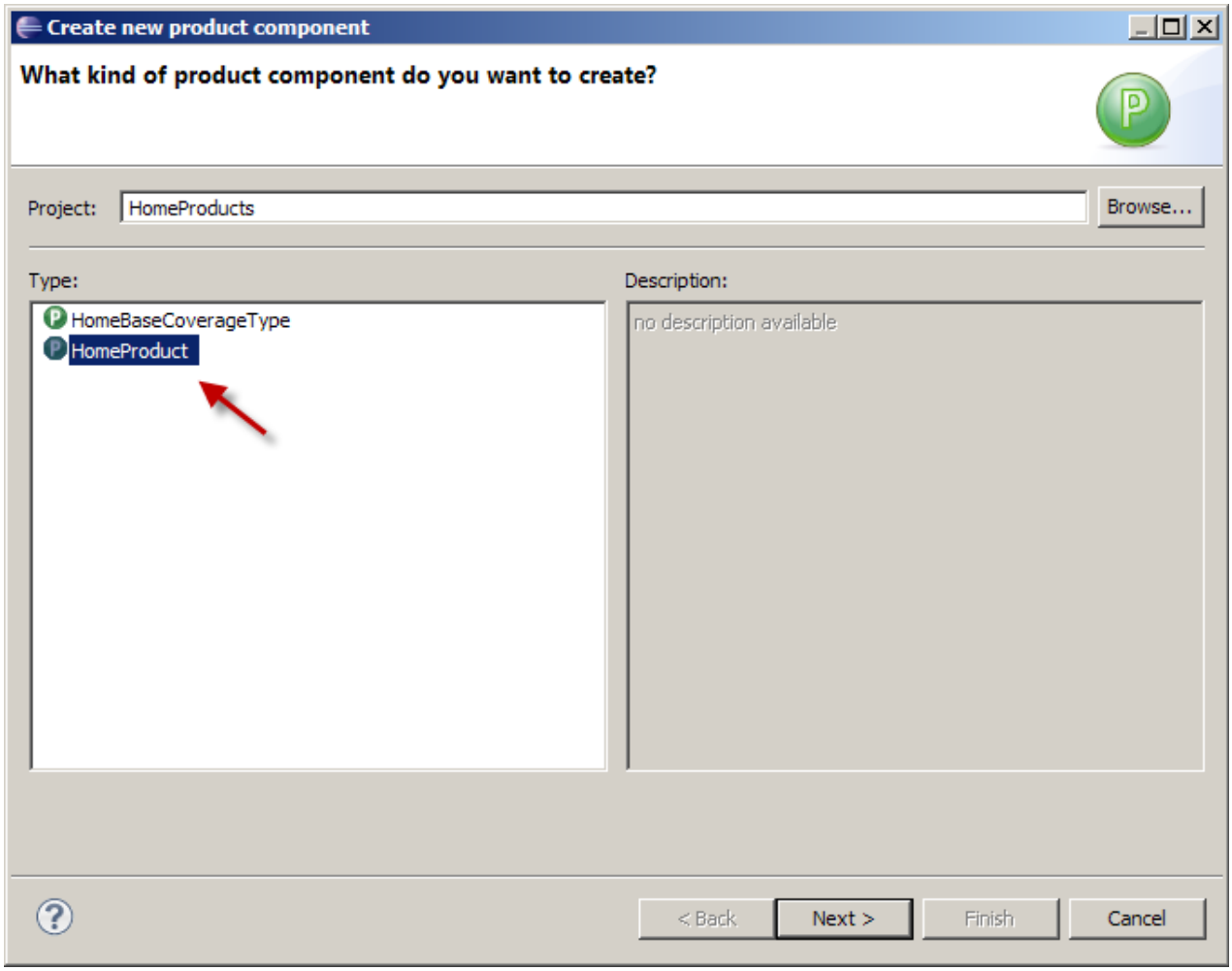

Figure 25: Selecting the product component type

In case you cannot see a product class, the reference to the HomeModel project in the Faktor-IPS build path is missing (see above). Select Next> to go to the next page of the wizard.

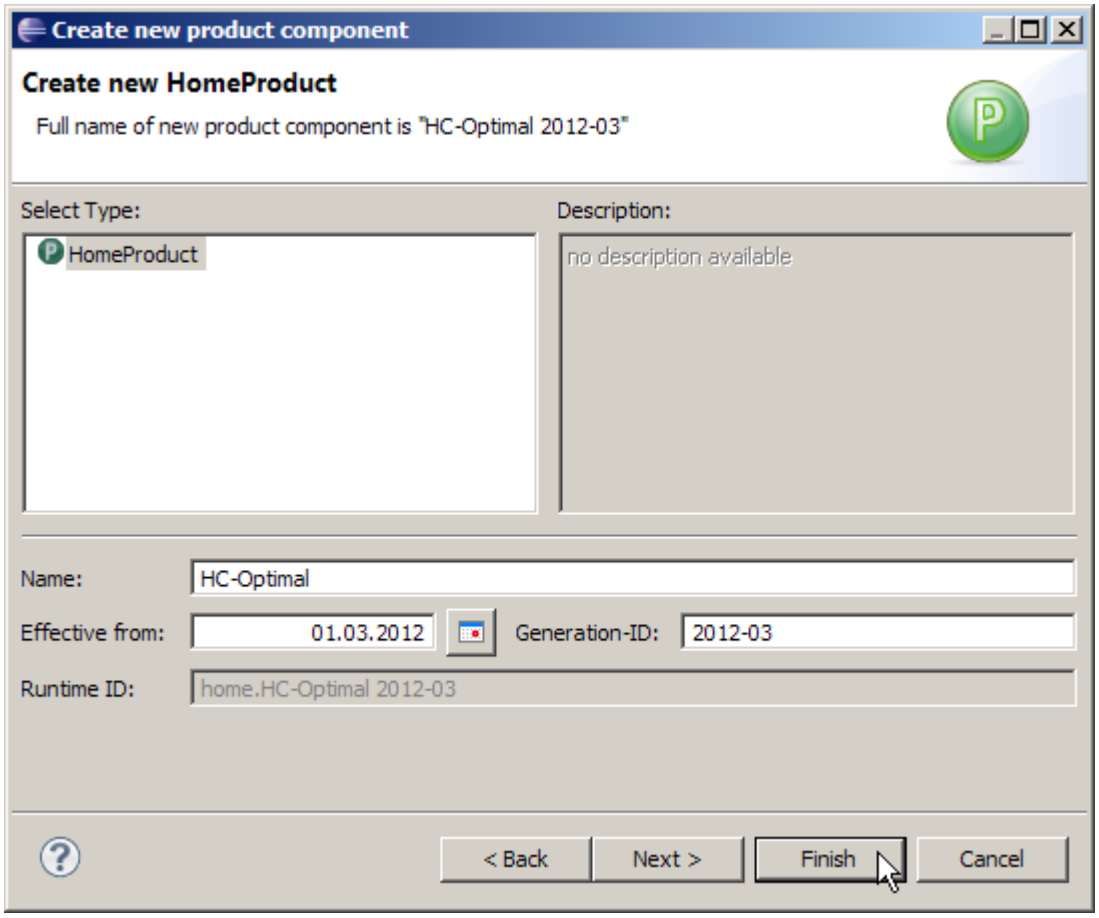

Figure 26: Creating a New Product

Here you enter the name of the product component. Its Generation number is filled in automatically according to the effective date. The Generation number format can be defined inside the ".ipsproject" file; per default it is "YYYY-MM" plus an additional, optional postfix, e.g. "2012-03b". A default Runtime ID will also be assigned; it consists of the prefix that has been defined within the project and the unqualified name.

This default can usually not be changed, unless you change the setting of Window►Preferences...►Can modify runtime id).

When you click Finish, the product component will be created in the file system. By double clicking on the product component in the product structure explorer the editor for the component will open.

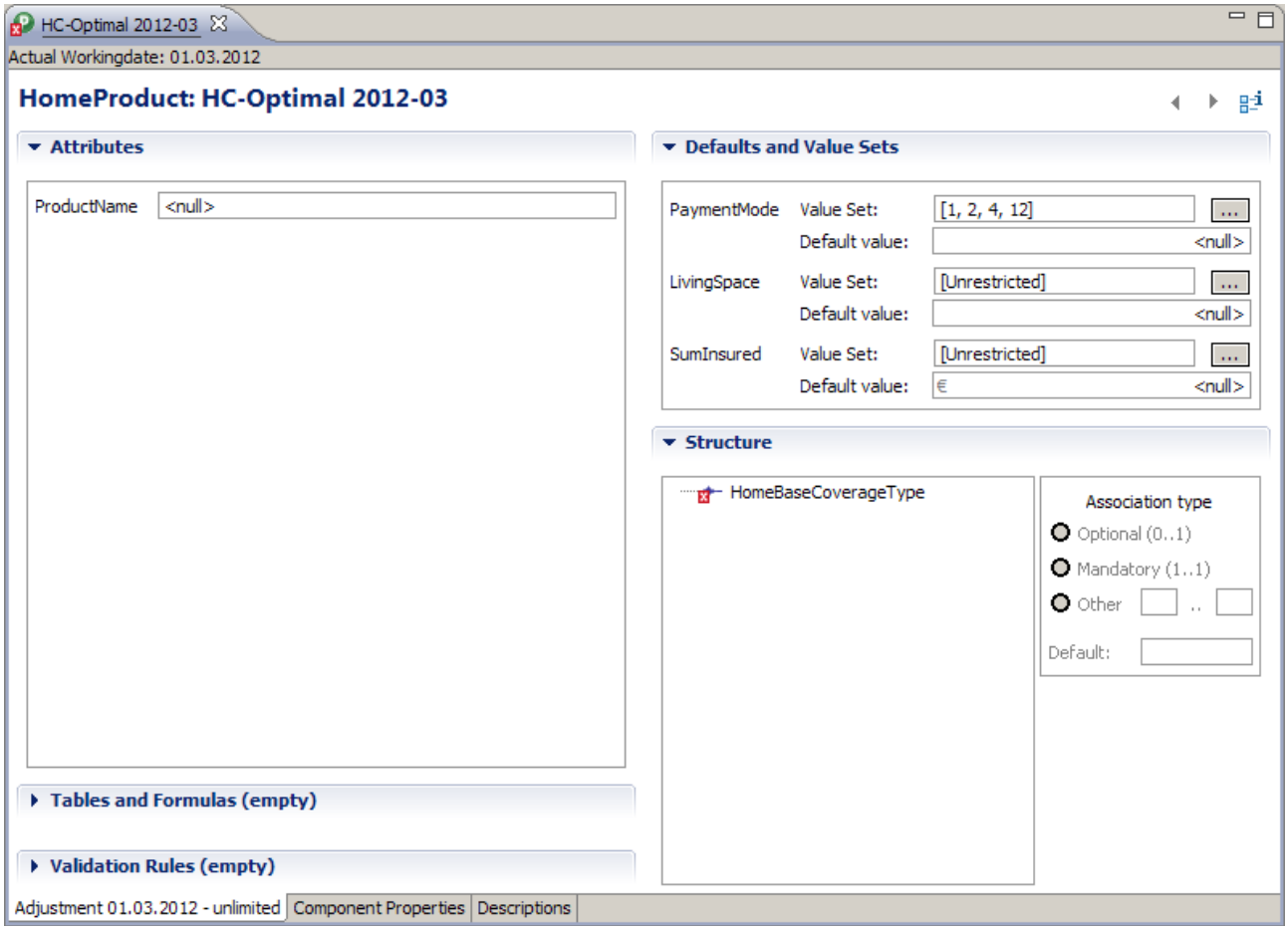

Figure 27: The Product Component Editor for HC-Optimal

The first editor page shows the product component Adjustment that is currently under construction. After its first creation, a component always has one Adjustment that is valid from the effective date that has been set up. By default, the page is divided into five sections[16]:

Attributes
Contains the properties of the Adjustment. Any attributes defined in the product class will be listed here.

Tables and Formulas
Contains computation formulas and table references. We will expand on this subject when we start implementing the premium computation in part 2.

Validation Rules

Contains those validation rules which have been marked as configurable in the model.

Defaults and Value Sets
Contains defaults and value ranges for the different contract properties.

16 Faktor-IPS 3.6 introduced the possibility of configuring the page set-up and thereby grouping the product component properties according to functional criteria.

Structure
Contains any other product components that are used.

Now enter the data for the HC-Optimal product according to the following table:

Configuration property	***HC-Optimal***
Product name	Home Contents Optimal
Proposed sum insured per sqm of living space	900EUR
Default payment mode	1 (annual)
Allowed payment modes	1, 2, 4, 12
Default living space	<null>
Allowed living space	0-2000
Default sum insured	<null>
Sum insured	10000EUR – 5000000EUR

Next, we will create the base coverage type for the product. Select the "coverages" package and define a new product component named *BaseCoverage-Optimal* based on the *HomeBaseCoveragetype* class.

Now we must assign the coverage type to the *HC-Optimal* product. You can do this by simply dragging and dropping the coverage type from the Model Explorer. Open the *HC-Optimal* product. Drag the *BaseCoverage-Optimal* from the Product Definition Explorer to the node *HomeBaseCoveragetype* in the Structure section.

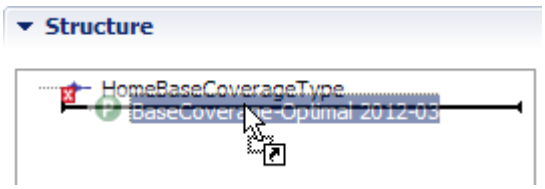

Figure 28: Coverage Types are Assigned to a Product

In the next step, we will create the *HC-Compact* product including the *BaseCoverage-Compact*. The process is the same as for the *HC-Optimal* product. Alternatively, you can use a copy wizard that enables you to copy a product component along with any other components it may use. If you want to try this, select the *HC-Optimal product* within the Product Definition Explorer and choose New►Copy Product ... from the dropdown menu. On the first page, enter "Optimal" as the Search Pattern and "Compact" as the Replace Pattern, click Next and then click Finish. Faktor-IPS will now create the *HC-Compact 2012-03* and *BaseCoverage-Compact 2012-03*. In practice, the copy wizard is typically used to create new Generations; to invoke it, you have to choose New►Create New Generation ... from the dropdown menu.

You can now open the new product and enter its data.

Configuration property	*HC-Compact*
Product name	Home Contents Compact
Proposed sum insured per square meter living space	600EUR
Default payment mode	annual
Allowed payment modes	bi-annual, annual
Default living space	<null>
Allowed living space	0-1000 sqm
Default sum insured	<null>
Sum insured	10000EUR – 2000000EUR

For the time being, the definition of both products is now complete. They should appear as follows inside the Product Definition Explorer:

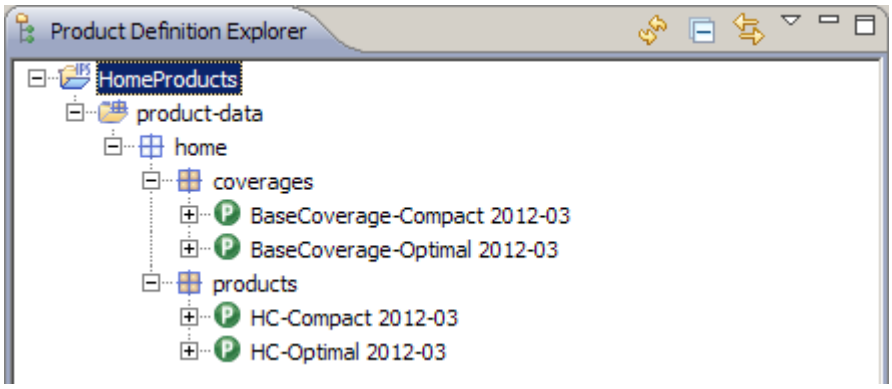

Figure 29: The Products are displayed in the Product Definition Explorer

In addition to the Product Explorer, two more tools are available to analyze the product definition. You can use the dropdown menu option Show Structure to view a product's structure and Search References to see the different usages of a component. Furthermore, you can define any package order by clicking Edit Sort Order.

On the right hand side of the product component editor, you can find a Model Description View that will show the documentation of the product class that relates to the product component you are currently editing. If you want to try this, you can document an attribute, e.g. “productName”, in the model, close the component editor and open it again.

Runtime Access to Product Information

Now that we have captured the product data, we will examine how to access them at runtime (inside an application or test case). We will write a JUnit-Test and extend it further in the second part of this tutorial.

For product data access, Faktor-IPS supplies the `IRuntimeRepository` interface. The implementation `ClassloaderRuntimeRepository` provides access to the product data captured with Faktor-IPS and loads the data using a classloader. Faktor-IPS does two things to enable this:

1. Any files containing product information are copied into the Java source folder named "derived". Consequently, those files are included in the build path of the project and can be loaded with the classloader.
2. A table of contents details which data are contained in the ClassloaderRuntimeRepository. Faktor-IPS generates this table of contents (toc) to a file that is referred to as a toc file and has a standard name of "faktorips-repository-toc.xml"[17].

The following screenshot shows the contents of the source folder "derived" within the Package Explorer.

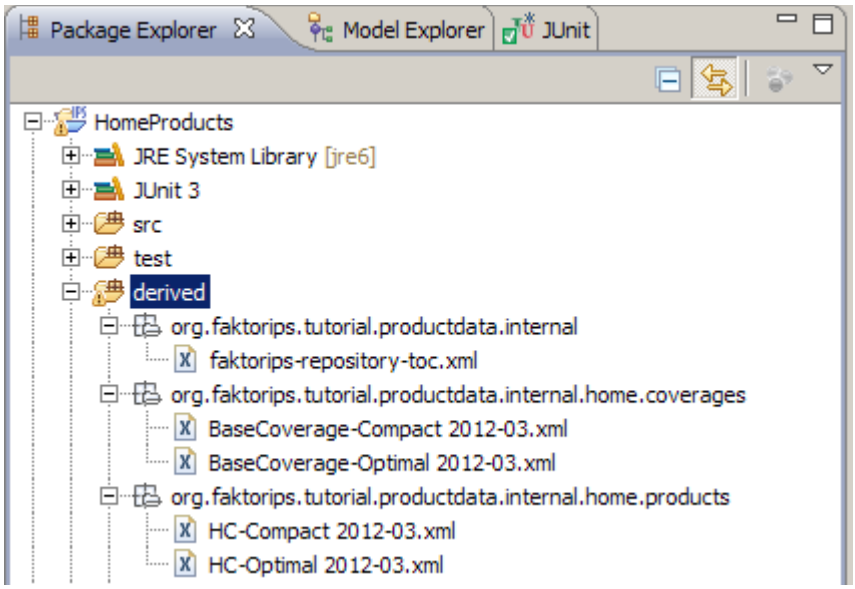

Figure 30: Contents of the Source Folder "derived"

A ClassloaderRuntimeRepository is created by the static `create(...)` method of the class. The path to the toc file is passed as a parameter. The `Classloader.getResourceAsStream()` method reads the toc file directly upon creation of the repository. Any additional data are loaded (also via the classloader), when they are accessed.

There is one big advantage to loading data with a classloader rather than from the file system: It is completely platform independent. This way, the code can run unchanged, for example, under z/OS.

17 The names can be configured in the ".ipsproject" file in section IpsObjectPath.

To get a product component, invoke the `getProductComponent(...)` method that is passed the RuntimeId of the component as a parameter. As the `IRuntimeRepository` interface does not depend on a specific model (like the HomeModel in our case), you have to cast the result to the respective product class.

Let us try this on a JUnit test case. First, you have to create a new Java source folder named "test" for our HomeProducts project. The easiest way to do this is to select the project in the Package Explorer and to choose Buildpath▶New Source Folder... from the dropdown menu. Then, mark the new source folder and create a JUnit test case by clicking the toolbar button  and choosing JUnit Test Case. In the dialog box, enter "TutorialTest" as the name of the test case class and mark the checkbox stating that the `setUp()` method should be generated as well. For the purposes of our tutorial, we will ignore the warning that advises against the usage of the default packages. In the bottom part of the dialog box there is a link which allows you to add the required JUnit Library to the Java build path of the project. The following box shows the source code of the test case class.

```
public class TutorialTest extends TestCase {

   private IRuntimeRepository repository;
   private IHomeProductAdj homeProductAdj;

   public void setUp() {
      // Create repository
      repository = ClassloaderRuntimeRepository.create(
       "org/faktorips/tutorial/productdata/internal/faktorips-repository-toc.xml");

      // Get reference to the comfort product from the repository
      IProductComponent pc = repository.getProductComponent("home.HC-Optimal 2012-03");

      // Get the latest adjustment (we know there is only one)
      IProductComponentGeneration pcAdj = pc.getLatestProductComponentGeneration();

      // Cast to our own model class
      homeProductAdj = (IHomeProductAdj) pcAdj;
   }

   public void testReadProductData() {
      System.out.println("Product name: "
               + homeProductAdj.getProductName());
      System.out.println("Proposed sum insured per sqm "
               + homeProductAdj.getProposedSumInsuredPerSqm());
      System.out.println("Default mode of payment: "
               + homeProductAdj.getDefaultValuePaymentMode());
      System.out.println("Allowed modes of payment: "
               + homeProductAdj.getAllowedValuesForPaymentMode(null));
      System.out.println("Default sum insured: "
               + homeProductAdj.getDefaultValueSumInsured());
      System.out.println("Range sum insured: "
               + homeProductAdj.getRangeForSumInsured(null));
      System.out.println("Default living space: "
               + homeProductAdj.getDefaultValueLivingSpace());
      System.out.println("Range living space: "
               + homeProductAdj.getRangeForLivingSpace(null));
   }
}
```

Instead of carrying out checks with assert statements, we will just print the results on the console with print1n. If you run the test now, it should print the following:

```
Product name: Home Contents Optimal
Proposed sum insured per sqm: 900.00 EUR
Default mode of payment: 1
Allowed modes of payment: [1, 2, 4, 12]
Default sum insured: MoneyNull
Range sum insured: 0.00 EUR-5000000.00 EUR
Default living space: null
Range living space: 0-2000
```

We have now gained some insight into modeling with Faktor-IPS. In addition, we have created some simple products and accessed product data at runtime.

The second part of this tutorial will give an introduction to the usage of tables and formulas. These will then be used to extend the model so that business users can flexibly add extra coverages without having to extend the model.