Faktor-IPS Tutorial

Part 2: Using Tables and Formulas

Jan Ortmann, Gunnar Tacke
(Dokumentversion 1582)

Overview

In the first part of our tutorial we explained modeling and product configuration with Faktor-IPS using a simplified example of a home contents insurance. In the second part we will demonstrate the use of tables and formulas. To do this, we will expand on the home contents model we created in part 1.

The chapters are organized as follows:

- Using Tables
  In this chapter, we will add a rating district table to our model and implement access to the table contents. In a second step, we will define product-specific rate tables and model the relationships between rate tables and products.
- Implementing Premium Computation
  Next, the computation of insurance premiums will be implemented and subsequently tested with a JUnit test. The premium computation will access the product-specific rate tables.
- Using Formulas
  In this chapter we will add extra coverages to our home contents model. This way, the business users will be able to flexibly add extra coverages, such as insurances against risks like bicycle theft or overvoltage damage, without having to extend the model each time they do this. We will achieve this by giving the business user the capability to define and use formulas.

Using Tables

In this chapter we will create tables that capture rating districts and premium rates and add them to our model. After that we will write code to determine which rating district will be applied to a particular contract.

Rating District Table

As the risk of damage through burglary varies from region to region, insurers usually apply different rating districts to their home contents insurance products. This is generally done using a table that maps zipcode areas to their respective rating districts. In Germany, such a table could look like this:

zipcodeFrom	*zipcodeTo*	*rating district*
17235	17237	II
45525	45549	III
59174	59199	IV
47051	47279	V
63065	63075	VI
...	...	...

For all zipcodes that do not fall into one of this areas, rating district I will be applied.

Faktor-IPS distinguishes between the definition of table structure and table contents. The structure of a table is created as part of the model, whereas the table contents can be managed either as part of the model or as part of the product definition, depending on what it contains and who is responsible for maintaining the data. There can be many table contents relating to one table structure[1].

Let us first create the table structure of the rating district table. To do this, you have to switch to the Java-Perspective. In the home model project, select the “home” folder and click the toolbar button  . Name the table structure “RatingDistrictTable” and click Finish.

You can leave the default table type Single Content as is, because we want this structure to have only one content. Now we will first create the table columns. All three columns (zipcodeFrom, zipcodeTo, ratingDistrict) are of type String.

The task of defining the zipcode area is where it gets interesting. The table structure we have created so far ultimately serves to establish a function (in the mathematical sense) of ratingDistrict→zipcode. This, however, can not be done just with the column definition and a potential Unique Key. Therefore, Faktor-IPS provides a way to model columns (or one column) representing a range. You can now go ahead and create a new range. As the table contains the columns zipcodeFrom and zipcodeTo, you can choose the type Two Column Range. Enter „zipcode“ as parameter name for the accessor method and map both the zipcodeFrom column and the zipcodeTo column.

1 This corresponds to the concept of table partitions in relational database management systems.

Now you have to create a new Unique Key. Make sure NOT to map the separate columns zipcodeFrom and zipcodeTo to this key; instead map the range to it. You can then save the structure description.

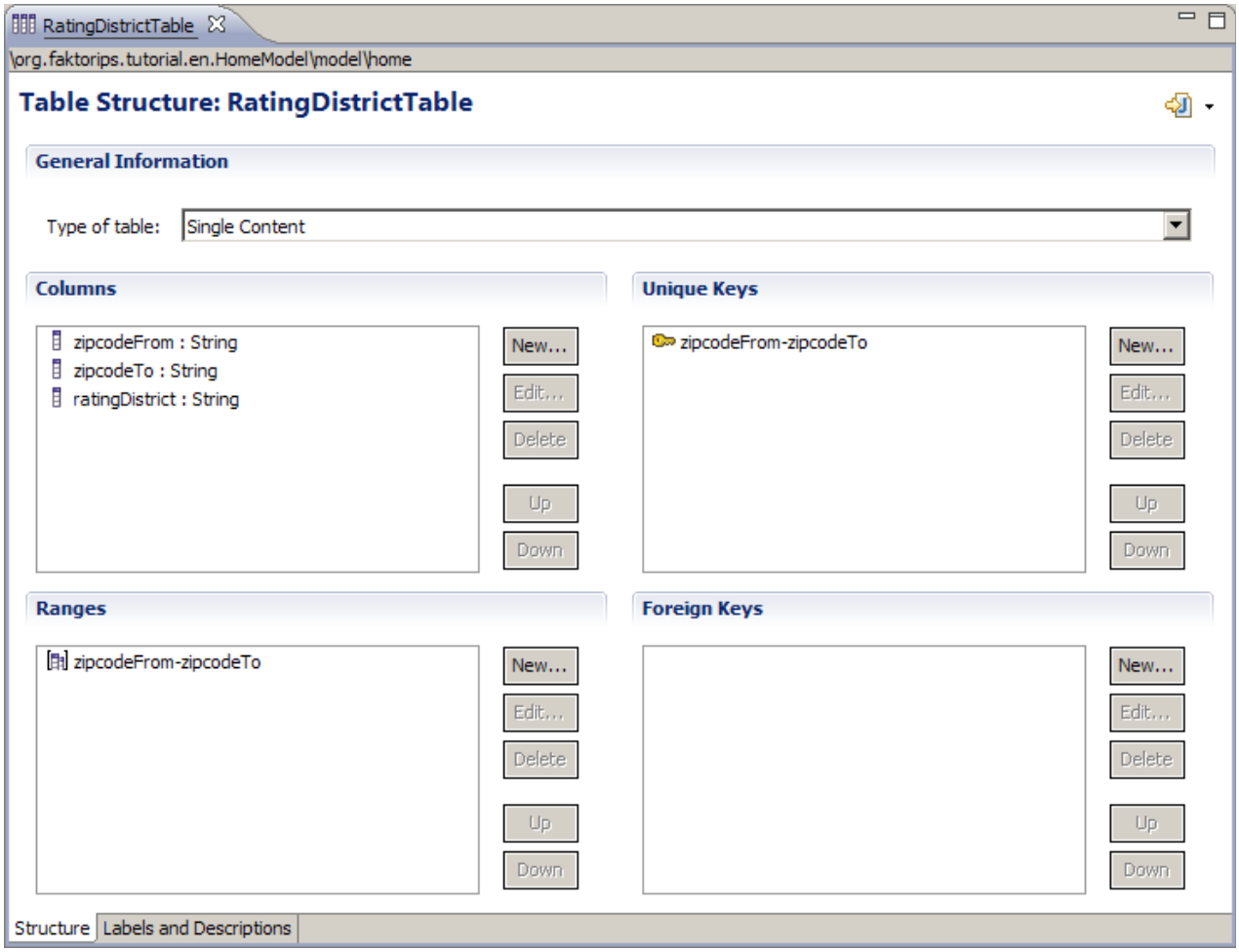

Figure 1: Creating a Table Structure

Faktor-IPS has now generated two more classes for the table structure in the package `org.faktorips.tutorial.model.internal.home`.

The `RatingDisctrictTableRow` class represents one row of the table and it contains one member variable per column together with the necessary accessor methods. The `RatingDisctrictTable` class represents the table contents. In addition to methods for initializing the table contents from XML, a method for finding a particular row has been generated using the Unique Key:

```
public RatingDistrictTableRow findRow(String zipcode) {
   // implementation details are omitted
}
```

Let us now use this class to implement a way to determine the rating district of a home contract. The rating district is a derived property of the home contract, so there is a `getRatingDistrict()` method in the `HomeContract` class. This method has already been implemented as follows:

```
public String getRatingDistrict() {
    return "I"; // TODO later we will implement this with a table lookup
}
```

Next, we will determine the rating district based on the zipcode by accessing our new table:

```
public String getRatingDistrict() {
    if (zipcode==null) {
        return null;
    }
    IRuntimeRepository repository = getHomeProduct().getRepository();
    RatingDistrictTable table = RatingDistrictTable.getInstance(repository);
    RatingDistrictTableRow row = table.findRow(zipcode);
    if (row==null) {
        return "I";
    }
    return row.getRatingDistrict();
}
```

At this point, you will probably want to know how to get an instance of our table. Because the rating district table only has one content, it provides a `getInstance()` method that returns this content. The parameter to this method is the RuntimeRepository that provides runtime access to the product data, including the table contents. In order to get it, we use the product that the contract is based on[2].

Next, we will create the table content. The business users will be responsible to maintain the mapping of zipcodes to rating districts. In order to enhance the overall structure, please add a new package named "tables" underneath the "home" package of the HomeProducts project. Then select the new package and click the toolbar button  . When the dialog box opens, choose the *RatingDistrictTable* structure. Accept the name "RatingDistrictTable" to name the table contents and click Finish. In the editor you can now enter the example rows showed above. After that, the project structure should look as follows in the Project Definition Explorer:

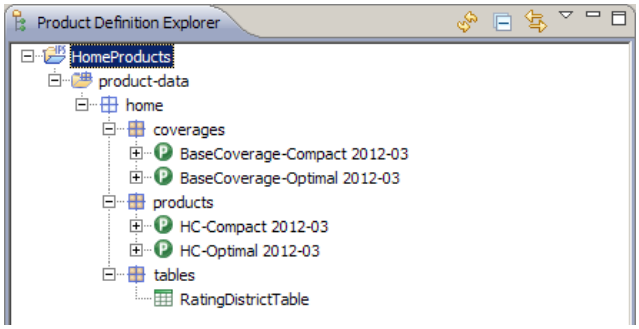

Figure 2: The Product Definition Project Structure

Finally, we will test if the rating district is determined correctly. In order to to this, we will extend the JUnit test `TutorialTest` of the first part of this tutorial by adding the following test method[3]:

2 Passing RuntimeRepositories to the `getInstance()` method offers the advantage that the Repository can easily be replaced in test cases.

3 The tutorial "Software tests with Faktor-IPS" describes, among other things, how this test can be generated and carried out comfortably with the help of Faktor-IPS tools.

```
public void testGetRatingDistrict() {
    // Create a new HomeContract with the product's factory method
    IHomeContract contract = homeProduct.createHomeContract();

    contract.setZipcode("45525");
    assertEquals("III", contract.getRatingDistrict());
}
```

Rate Table

We want to use a rate table to determine the insurance premium for the base coverage type of our home contents insurance. In the process, we will apply different premiumss for our two products. These rates will be based upon the following tables:

rating district	*premium rate*
I	0.80
II	1.00
III	1.44
IV	1.70
V	2.00
VI	2.20

Table 1: Rate Table for HC-Optimal

rating district	*premium rate*
I	0.60
II	0.80
III	1.21
IV	1.50
V	1.80
VI	2.00

Table 2: Rate Table for HC-Compact

The data for different products are often grouped in a single table that includes an additional “ProduktID” column. In Faktor-IPS, however, you can also create multiple contents for one table structure and define the relationships between tables and products!

To do this, create a table structure named “RateTableHome” with a String column named “ratingDistrict” and a Decimal column named “premiumRate”. Define a Unique Key on the

"ratingDistrict" column and choose Multiple Contents as the table type, because this time we want to create different table contents for each product.

In practice, this is generally used to store the rates for each product version in different table contents. This approach has the advantage that tables do not get too big and can be edited and versioned separately. If you want to introduce a new product version, you just add new table contents. This can then be done by simply importing Excel files, for example. The separation of table contents also has benefits at runtime. As the data of older tariff versions are not accessed too often, they can be handled by a different caching strategy.

For both *HC-Optimal* and *HC-Compact* (or, more precisely, for their base coverage types), create two table contents named "RateTable Optimal 2012-03" and"RateTable Compact 2012-03", respectively[4].

The following diagram shows the relationship between the *BaseCoverageType* class and the table structure *RateTableHome*, as well as the related object instances.

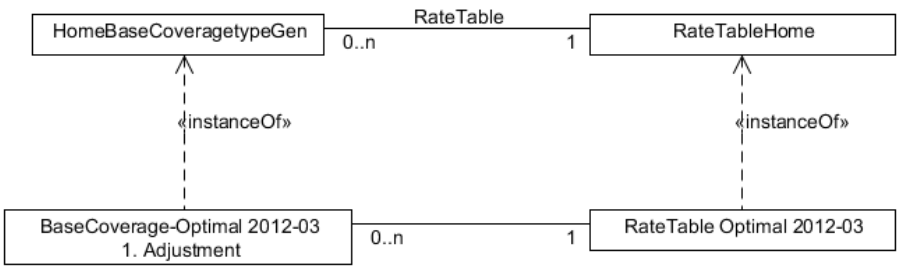

Figure 3: Relationship between Product Components and Tables

Each generation of a base coverage type uses a rate table. The first generation of the *BaseCoverage-Optimal 2012-03* uses the table contents *RateTable Optimal 2012-03*.

In order to define the relationship between tables and products in Faktor-IPS, go to the editor of the *HomeBaseCoverageType* class. On the second page of the Editor[5], the Table Usages section lists each table structure currently in use. To define a new table usage, just click on the New button near this section.

4 Set the "2012-03" suffix to the respective effective date that you are using.

5 Provided that you have set your Preferences such that your Editors use 2 sections per page.

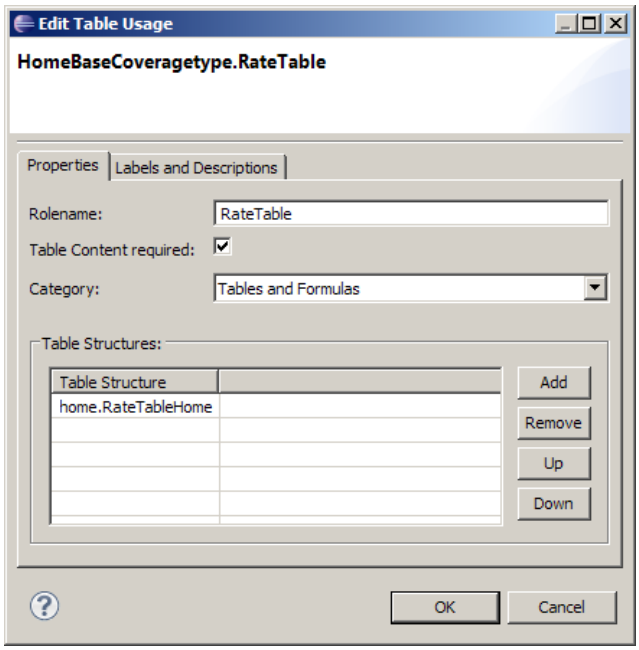

Figure 4: Defining a new Table Usage

In the dialog box, enter the role name"RateTable" and assign it the *RateTableHome* table structure. At this point, you could also assign multiple table structures to allow for different table structures under the rate table role because, for example, new rate characteristics might emerge over time. Finally, enable the Table content required checkbox, because for each base coverage type a rate table has to be specified. After that, close the dialog box and save your settings.

Now we can map the table contents to the base coverage types. First, open *BaseCoverage Optimal 2012-03*. When a dialog box pops up to tell you that the rate table has not yet been mapped, confirm it with Fix. In the Calculation Formulas and Tables section, you can now map the rate table for *HC-Optimal* and save your work. The same process applies to *HC-Compact.*

At the end of this chapter, we will take a look at the generated source code. In the `HomeBaseCoverageTypeGen` class, you can find a method to get the assigned table contents:

```
public RateTableHome getRateTable() {
    if (rateTableName == null) {
        return null;
    }
    return (RateTableHome) getRepository().getTable(rateTableName);
}
```

As the respective finder methods are also generated on the table, you can implement efficient table access with just a few lines of code.

Implementing Premium Computation

In this chapter we will implement the premium computation for our home contents products. We will extend our model to include the attributes and methods shown in the following figure.

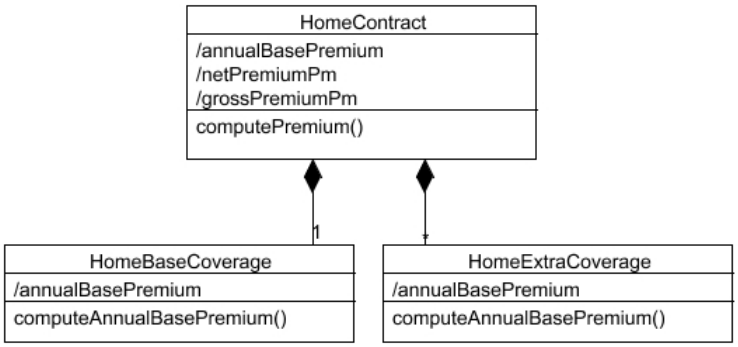

Figure 5: The Premium Computation is Captured in the Model

Each coverage is subject to an annual base premium. An annual base premium is the net premium payable per annum, insurance tax and surcharges not included. The contract's annual base premium is the total value of all coverages' annual base premiums. The contract's netPremiumPm is the net premium due per payment. It is the result of the annual base premium plus an installment surcharge (if the premium is not paid annually), divided by the number of payments, e.g., 12 in the case of monthly payment. The grossPremiumPm is the gross premium due per payment, i.e. including insurance tax. It amounts to the netPremiumPm times 1+insurance tax rate. For the sake of simplicity, we will assume in this tutorial that the installment surcharge is always 3% and the insurance tax is always 19%.

Your next task will be to create the new attributes in the classes *HomeContract* and *HomeBaseCoverage*. We will leave the *HomeExtraCoverages* for the next chapter. All attributes are derived (cached) and of the type Money. Because they are cached, derived attributes, Faktor-IPS generates one member variable and one getter method for each attribute. All premium attributes are computed by the `computePremium()` method of the HomeContract class. This method is able to compute all premium attributes of the contract, as well as the annual base premium of the coverages. To do this, of course, it uses the `computeAnnualBasePremium()` method of the coverages.

You can now create these methods on the second editor page for the *HomeContract* class. The dialog box for editing a method signature is shown in the next figure.

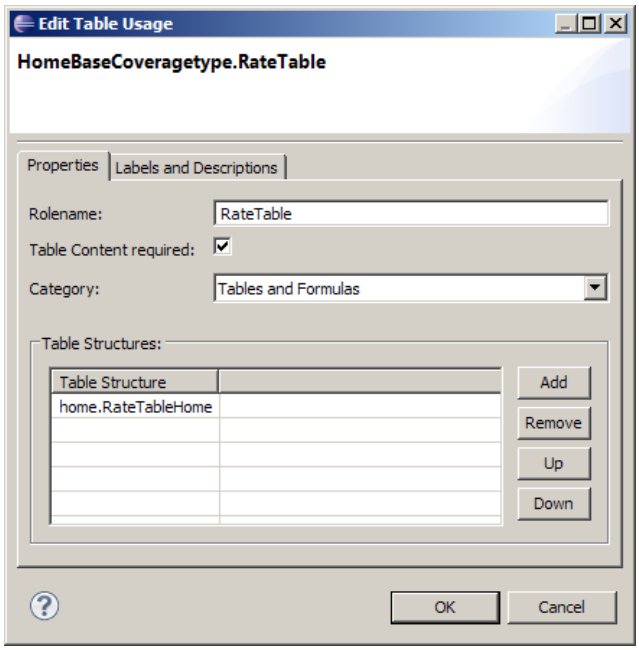

Figure 6: Edit Dialog for Method Signatures

Generating code offers more advantages for relationships and attributes than for methods. Hence, methods can, of course, also be defined directly in the source code. In order to strike a balance, we capture methods of the published interface in Faktor-IPS for documentation purposes, while defining methods that are internal to the model only in the source code.

The following code snippet shows the premium computation implementation in the `HomeContract` class. For the sake of clarity, we will implement the computation of the annual base premium and the netPremiumPm in two separate, private methods that will be written directly in the source code, but not added to the model.

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public void computePremium() {
    computeAnnualBasePremium();
    computeNetPremiumPm();
    Decimal taxMultiplier = Decimal.valueOf(119, 2); // 1 + 19% tax rate
    grossPremiumPm = netPremiumPm.multiply(taxMultiplier, BigDecimal.ROUND_HALF_UP);
}

private void computeAnnualBasePremium() {
    annualBasePremium = Money.euro(0);
    IHomeBaseCoverage baseCoverage = getBaseCoverage();
    baseCoverage.computeAnnualBasePremium();
    annualBasePremium = annualBasePremium.add(baseCoverage.getAnnualBasePremium());
    /*
     * TODO: When extra coverages are added to the model, their premium
     * of course has to be added here as well.
     */
}
```

```
private void computeNetPremiumPm() {
    if (paymentMode==null) {
        netPremiumPm = Money.NULL;
        return;
    }
    if (paymentMode==1) {
        netPremiumPm = annualBasePremium;
    } else {
        Decimal factor = Decimal.valueOf(103, 2); // 1 + 0.03 surcharge for none annual payment
        netPremiumPm = annualBasePremium.multiply(factor, BigDecimal.ROUND_HALF_UP);
    }
    netPremiumPm = netPremiumPm.divide(paymentMode, BigDecimal.ROUND_HALF_UP);
}
```

Premium Computation for Coverages

For our home contents insurance, the annual base premium computation has to be implemented at the coverages level.

Technically, the annual base rate for the base coverage is computed like this:

- From the rates table, determine the rate per 1000 Euro sum insured
- Divide the sum insured by 1000 Euro and multiply with the premium rate.

As this formula is not subject to change, we will implement it directly in the `HomeBaseCoverage` Java class. For extra coverages, we will allow the business users to define the annual base premium using computation formulas.

First, we will define the method `computeAnnualBasePremium()` in the `HomeBaseCoverage` class (with Access Modifier published).

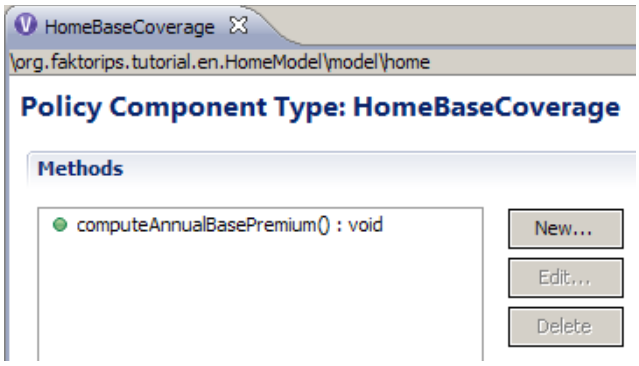

Figure 7: Creating the method computeAnnualBasePremium

Next, open the Java class in the Editor and implement the method as follows:

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public void computeAnnualBasePremium() {
    RateTableHome rateTable = getRateTable();
    if (rateTable==null) {
        annualBasePremium = Money.NULL;
        return;
    }
    RateTableHomeRow row = rateTable.findRow(getHomeContract().getRatingDistrict());
    if (row==null) {
        annualBasePremium = Money.NULL;
        return;
    }
    Money sumInsured = getHomeContract().getSumInsured();
    Money sumInsuredDiv1000
             = sumInsured.divide(Decimal.valueOf(1000, 0), BigDecimal.ROUND_HALF_UP);
    Decimal premiumRate = row.getPremiumRate();
    annualBasePremium = sumInsuredDiv1000.multiply(premiumRate, BigDecimal.ROUND_HALF_UP);
}
```

We will test the premium computation by extending our JUnit Test once more.

```
public void testComputePremium() {
    // Create a new HomeContract with the product's factory method
    IHomeContract contract = homeProduct.createHomeContract();

    // Set the contract's effective date, so that we find the adjustment
    // This must be a date after the adjustments valid from date.
    contract.setEffectiveFrom(new GregorianCalendar(2012, 3, 1));

    // Set the contract's attributes
    contract.setZipcode("45525"); // => rating district III
    contract.setSumInsured(Money.euro(60000));
    contract.setPaymentMode(new Integer(2)); // semi-annual

    // Get the base coverage type that is assigned to the product
    IHomeBaseCoveragetype coveragetype = homeProductAdj.getBaseCoveragetype();

    // Create the base coverage and add it to the contract
    IHomeBaseCoverage coverage = contract.newBaseCoverage(coveragetype);

    // Compute the premium and check the results
    contract.computePremium();

    // rating district III => premiumRate = 1.44
    // annualBasePremium = sumInsured / 1000 * premiumRate  = 60000 / 1000 * 1.44 = 86.40
    assertEquals(Money.euro(86, 40), coverage.getAnnualBasePremium());

    // contract.annualBasePremium  = baseCoverage.annualBasePremium
    assertEquals(Money.euro(86, 40), contract.getAnnualBasePremium());

    // netPremiumPm = 86,40 / 2 * 1,03 = 44,496 => 44,50 (semi-annual, 3% surcharge)
    assertEquals(Money.euro(44, 50), contract.getNetPremiumPm());

    // grossPremiumPm = 44,50 * taxMultiplier = 44,50 * 1,19 = 52,955 => 52,96 (19% tax rate)
    assertEquals(Money.euro(52, 96), contract.getGrossPremiumPm());
}
```

Using Formulas

So far, our home contents model does not offer much flexibility to the business user. A product can have precisely one base coverage and the rate is determined from the rate table. Next, we will allow the business user to flexibly define extra coverages without having to modify the model or the code. We will now use the formula language of Faktor-IPS to compute the insurance premiums.

We will use extra coverages against bicycle theft and overvoltage damage as examples:

	Bicycle Theft	***Overvoltage Damage***
Extra coverage sum insured	1% of the contract's sumInsured, maximum 3000 Euro	5% of the contract's sumInsured. No maximum value.
Annual base premium	10% of sum insured in bicycle theft coverage	10Euro + 3% of sum insured in overvoltage coverage

Extra coverages of this type have their own sum insured that is dependent on the sum insured agreed upon in the contract. The annual base premium, on the other hand, is dependent on the sum insured in the coverage. In order to be able to represent these sorts of extra coverages, we extend our model as shown in the following class diagram:

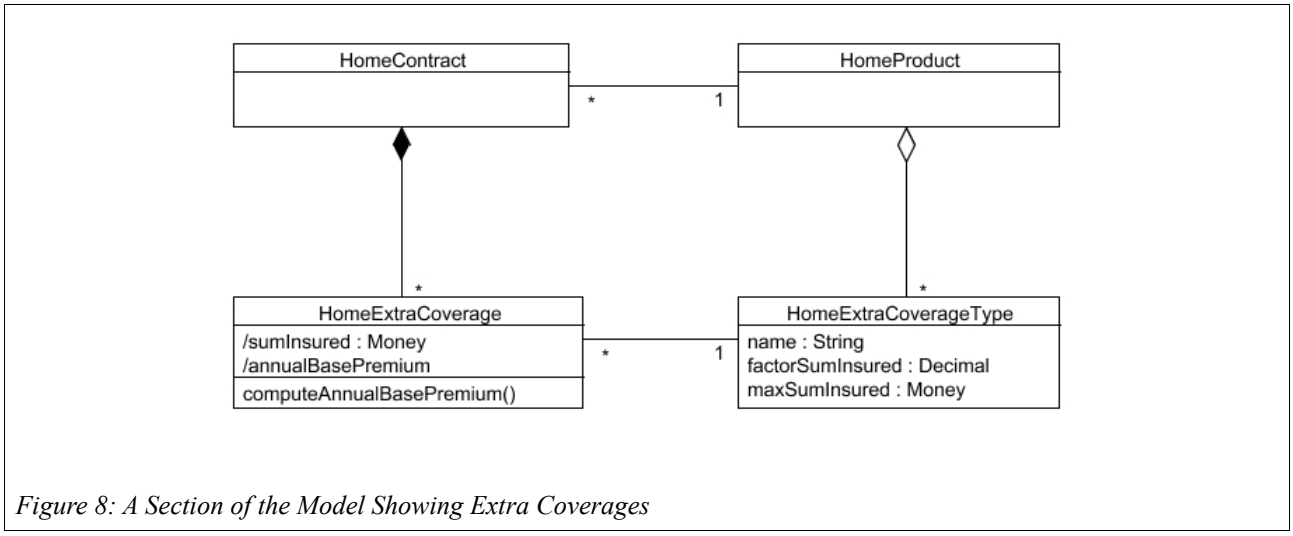

Figure 8: A Section of the Model Showing Extra Coverages

One HomeContract can contain any number of extra coverages. The configuration class pertaining to *HomeExtraCoverage* will be named *HomeExtraCoverageType*. It includes the properties “factorSumInsured” and “maxSumInsured”. The sum insured in the extra coverage is computed by the sum insured in the contract times the factor, and it can not exceed the maximum sum insured.

Both example coverages are instances of the *HomeExtraCoverageType* class, as shown in the following diagram.

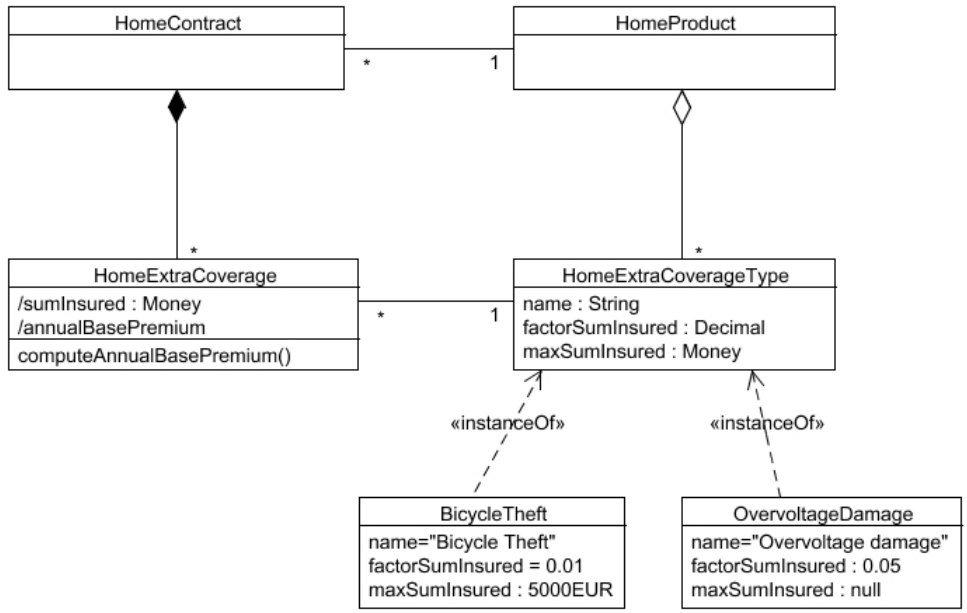

Figure 9: A Section of the Model Showing Extra Coverages with Instances

Strictly speaking, *BicycleTheft* and *OvervoltageDamage* are of course instances of the generation class `HomeExtraCoveragesTypeAdj`, but this detail has been omitted for the sake of simplicity.

Before we look at the premium computation, we first have to create the new classes *HomeExtraCoverage* and *HomeExtraCoverageType*. The contract class creation wizard enables you to create both classes in the same process. Start the wizard and enter the data shown in the following picture on the first wizard page.

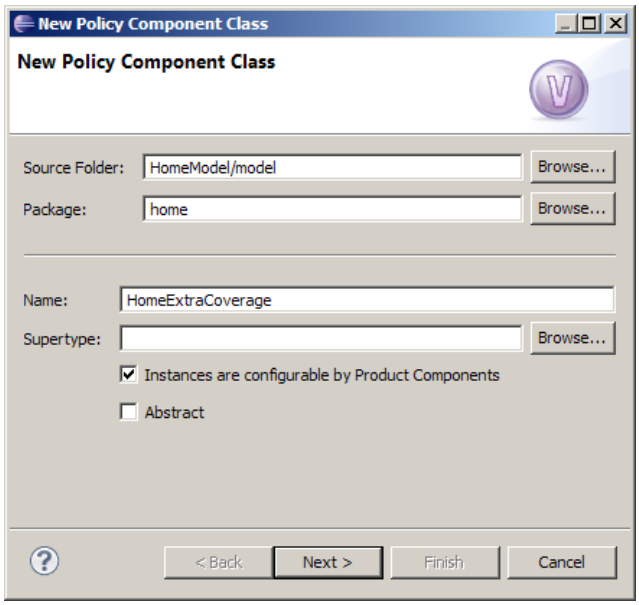

Figure 10: The Wizard for Creating the HomeExtraCoverage Contract Class

On the second page, enter the class name HomeExtraCoverageType and click Finish. Faktor-IPS will now create both classes as well as their references to one another.

Next, we have to define the relationships between *HomeContract* and *HomeExtraCoverage*, and between *HomeProduct* and *HomeExtraCoverageType*, respectively. To do this, you can use the wizard for creating new relationships in the contract class editor. The wizard enables you to create the relationship on the product side at the same time.

Once the relationships have been defined, add the derived sumInsured attribute (of type Money) to the *HomeExtraCoverage* class and the attributes name (String), factorSumInsured (Decimal) and maxSumInsured (Money) to the *HomeExtraCoverageType* class. When you have completed the attributes, you can go on and implement the computation of the sum insured in the `HomeExtraCoverage` class, as follows:

```
public Money getSumInsured() {
    IHomeExtraCoveragetypeAdj adj = getHomeExtraCoveragetypeAdj();
    if (gen==null) {
        return Money.NULL;
    }
    Decimal factor = adj.getFactorSumInsured();
    Money contractSumInsured = getHomeContract().getSumInsured();
    Money sumInsured = contractSumInsured.multiply(factor, BigDecimal.ROUND_HALF_UP);
    if (sumInsured.isNull()) {
        return Money.NULL;
    }
    Money maxSumInsured = adj.getMaxSumInsured();
    if (sumInsured.greaterThan(maxSumInsured)) {
        return maxSumInsured;
    }
    return sumInsured;
}
```

We will then create the coverage types for bicycle theft and overvoltage damage. Go back to the Product Definition Perspective and, in the Product Definition Explorer, select the "coverages" package of the HomeProducts project. Based on the *HomeExtraCoverageType* class, you will now create two product components named *BicycleTheft 2012-03* and *OvervoltageDamage 2012-03*, respectively, and define their properties in the editor.

	BicycleTheft 2012-03	***OvervoltageDamage 2012-03***
Name	Bicycle Theft	Overvoltage Damage
SumInsuredFactor	0.01	0.05
MaxSumInsured	3000EUR	<null>

The next step is to assign the coverages to the products, just as we have done before with the base coverages. Contracts based on the *HC-Optimal* product should always include both coverages, whereas with the *HC-Compact* product, they are optional. You can set this by means of the association type in the component editor.

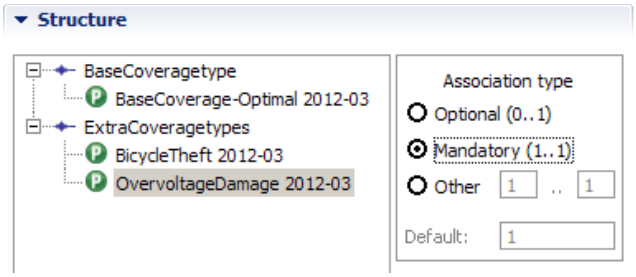

Figure 11: The Component Editor's Section for Editing Relationships

Computing the Premiums for Extra Coverages

The computation of the annual base premium should be defined with a formula by the business users. The premium due for an extra coverage will usually depend on the sum insured and any other risk-related characteristics[6]. Hence, the formula needs to access these properties. There are essentially two possible ways to do this:

- The formula language allows random navigation of the object graph.
- The parameters used in the formula are defined explicitly.

Faktor-IPS uses the second alternative, essentially for two reasons:

1. The syntax for navigating the object graph can quickly get too complex. How, for example, would you write a syntax determining the coverage with the highest sum insured in a way that is still easily comprehensible for business users?
2. With the second approach, the derived attributes are always up to date.

6 In a home contents insurance, these could be aspects such as whether it is a house for one family or for multiple families, or which construction method has been applied.

The second aspect is best explained by means of an example. In order to compute the premium for an extra coverage, you need to know the sum insured, which is a derived attribute. If it is also a cached attribute, you have to ensure that the sum insured has been computed before calling the premium computation formula. If you want to enable any navigation through the object graph, you have to ensure that all available objects use correct values for their derived attributes. As this is error-prone and negatively affects performance, Faktor-IPS requires you to explicitly define all parameters that can be used in a formula.

Formula parameters can be of a simple type, such as the sum insured, but they can also be complex objects like, for example, the contract itself. The upside of using objects as parameters is that the parameter list does not have to be extended each time the business users need to access attributes that have not been used before. To compute the annual base premium of the extra coverage, we will use the extra coverage itself and its underlying home contract as parameters.

Before we can define the premium computation formula in the extra coverages, we have to define the formula signature with its parameters in the *HomeExtraCoverageType* class. To do this, open the editor for the *HomeExtraCoverageType* class. On the second page[7] click the New button in the Methods and Formula Signatures section to create a formula signature and enter the data according to the following screenshot.

7 Provided you have set your Preferences so that the Editors show 2 sections per page.

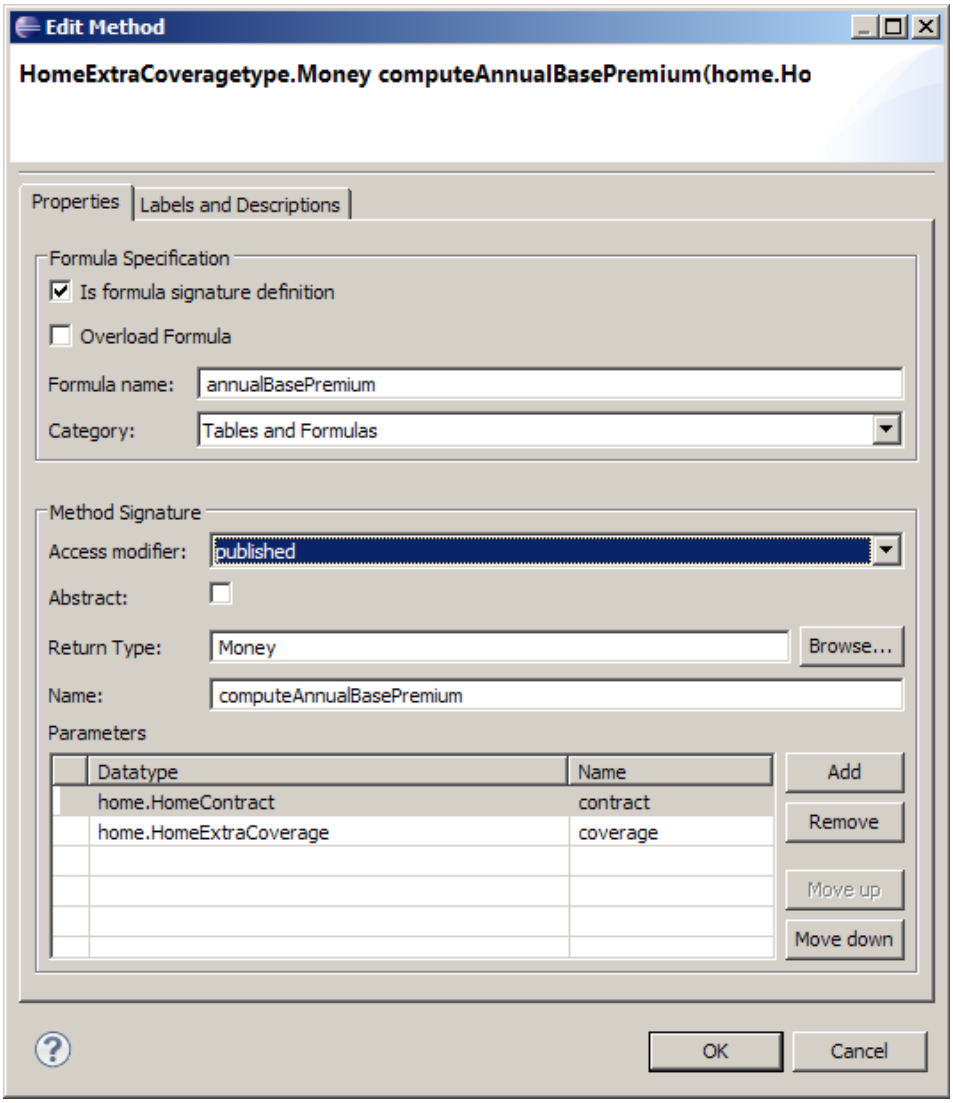

Figure 12: Dialog box for Defining a Formula Signature

Close the dialog box and save everything. The `HomeExtraCoveragesTypeGen` class is now an abstract Java class that includes an abstract `computeAnnualBasePremium(...)` method to compute the base premium. As different product components of the same model class (bicycle theft and overvoltage damage, in this case) can have different formulas, their source code can not be generated in the base class. For each product component that contains a formula, Faktor-IPS will create a separate subclass[8]. In this subclass, the abstract method is implemented by compiling the formula in Java code using a formula compiler.

Let us now open the bicycle theft coverage in order to define the premium computation formula. When you do this, you will first see a dialog box telling you that the model contains a new formula that has not yet been captured in the product definition. Click Fix to confirm that the formula should be added. In the Tables & Formulas section, the premium rate formula will be displayed, as yet

8 Strictly speaking, a class is generated for each product generation, because each generation can have a different formula.

empty. Click the button next to the formula box to edit the formula. The following dialog box will open, enabling you to edit the formula and to view the available parameters:

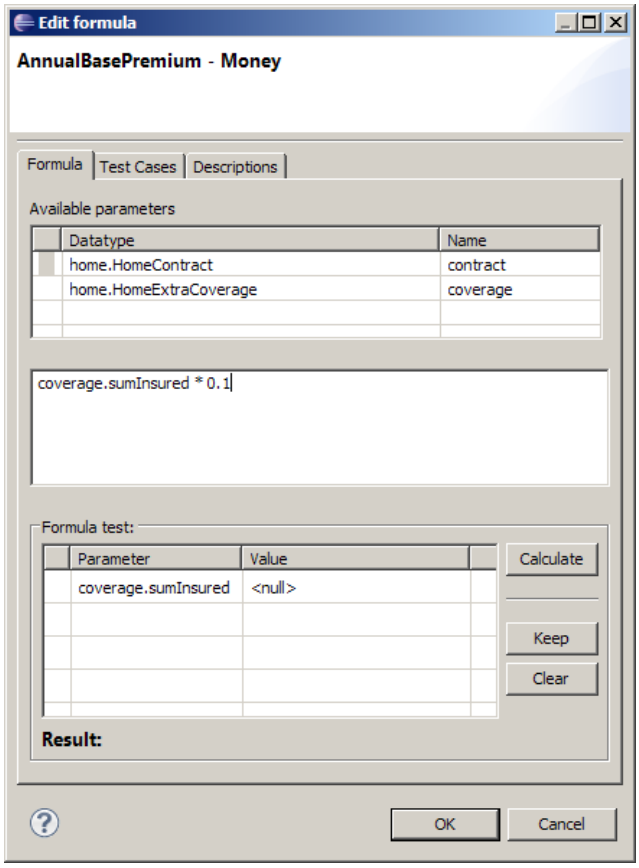

Figure 13: Creating a Formula

The bicycle theft insurance premium shall amount to 10% of its sum insured. Press Ctrl-Space in the middle input box and you will now see all the parameters and functions that are available to you. Choose the "coverage" parameter and enter a dot (.) at the end. Next, you will see a choice of properties pertaining to the coverage. Choose "sumInsured" and multiply the sum insured by 0.1.

The dialog box also allows you to test the formula by entering a value (e.g., "100EUR") for the sum insured into the Formula Test section and clicking the Calculate button.

Close the dialog box and save. Similarly, you can then define that the premium for overvoltage coverage is 10Euro + 3% of the sum insured (the formula is: 10EUR + coverage.sumInsured * 0.03).

Faktor-IPS has now generated the subclasses of both product components with the formula translated in Java code. You can find the two classes in the Java source folder"derived" of package `org.faktorips.tutorial.productdata.internal.home.coverages`[9]. The following code shows the generated `computeAnnualBasePremium(...)` method for the bicycle theft coverage.

9 Names of product components can contain special characters like spaces or hyphens. As these are not allowed in Java class names, they have been replaced by underscores. This replacement operation can be configured in the ProductCmptNamingStrategy section of the ".ipsproject" file.

```
public Money computeAnnualBasePremium(final IHomeExtraCoverage coverage,
        final IHomeContract contract) throws FormulaExecutionException {
    try {
        return coverage.getSumInsured().multiply(Decimal.valueOf("0.1"),
                BigDecimal.ROUND_HALF_UP);
    } catch (Exception e) {
        StringBuffer parameterValues = new StringBuffer();
        parameterValues.append("coverage=");
        parameterValues.append(coverage == null ? "null" : coverage
                .toString());
        parameterValues.append(", ");
        parameterValues.append("contract=");
        parameterValues.append(contract == null ? "null" : contract
                .toString());
        throw new FormulaExecutionException(toString(),
                "coverage.sumInsured * 0.1", parameterValues.toString(), e);
    }
}
```

If an error is encountered while running the compiled formula in Java, Faktor-IPS will throw a RuntimeException containing the formula text and the String representation of the parameters passed in.

The only remaining task is to ensure that the formula is called when calculating the premium. To enable this, we will implement the `computeAnnualBasePremium()` method in the `HomeExtraCoverage` class. We can achieve this simply by delegating to the computation method in the extra coverage type, passing in as parameters the extra coverage (this) and the contract to which it belongs. Define the method computeAnnualBasePremium in the model class *HomeExtraCoverage with the return value Money, visibility published and without parameters and* save everything.

Now open the `HomeExtraCoverage` Java class and implement your method as follows:

```
public void computeAnnualBasePremium() {
    annualBasePremium =
        getHomeExtraCoveragetypeAdj().computeAnnualBasePremium(this, getHomeContract());
}
```

In order to have the extra coverages premium added to the total premium of the home contract, we will extend the premium computation in the `HomeContract` class accordingly:

```
private void computeAnnualBasePremium() {
    annualBasePremium = Money.euro(0);
    IHomeBaseCoverage baseCoverage = getBaseCoverage();
    baseCoverage.computeAnnualBasePremium();
    annualBasePremium = annualBasePremium.add(baseCoverage.getAnnualBasePremium());
    /*
     * Iterate through extra coverages and add the respective premiums to
     * the total premium:
     */
    List<IHomeExtraCoverage> extraCoverages = getExtraCoverages();
    for (Iterator<IHomeExtraCoverage> it = extraCoverages.iterator(); it.hasNext();) {
        IHomeExtraCoverage extraCoverage = it.next();
        extraCoverage.computeAnnualBasePremium();
        annualBasePremium = extraCoverage.getAnnualBasePremium();
    }
}
```

Finally, at the end of this chapter, w will test our new functionality by extending our JUnit test once more:

```
public void testComputeAnnualBasePremiumBicycleTheft() {
    // Create a new HomeContract with the product's factory method
    IHomeContract contract = homeProduct.createHomeContract();

    // Set the contract's effective date, so that we find the product generation
    // This must be a date after the genration's valid from date.
    contract.setEffectiveFrom(new GregorianCalendar(2012, 3, 1));

    // Set the contract's attributes
    contract.setZipcode("45525"); // => rating district III
    contract.setSumInsured(Money.euro(60000));

    // Get ExtraCoveragetype BicycleTheft
    // (To make it easy, we assume that this is the first one)
    IHomeExtraCoveragetype coveragetype = homeProductAdj.getExtraCoveragetype(0);

    // Create extra coverage
    IHomeExtraCoverage coverage = contract.newExtraCoverage(coveragetype);

    // compute AnnualBasePremium and check it
    coverage.computeAnnualBasePremium();

    // Coverage.SumInsured = 1% from 60,0000, max 3.000 => 600
    // Premium = 10% from 600 = 60
    coverage.computeAnnualBasePremium();
    assertEquals(Money.euro(60, 0), coverage.getAnnualBasePremium());
}
```

In this second part of the tutorial we have taken a look at how tables are used in Faktor-IPS, how to implement premium computation and, using extra coverages as an example, we examined how to design a model in a way which allows it to be extended flexibly.

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

The tutorial on model partitioning shows how to divide complex models into meaningful parts and how to handle these. Specifically, by way of various examples, the tutorial illustrates the separation into different lines of business (lob) and how to separate lob-specific from cross-lob aspects.

The support available in testing Faktor-IPS is shown in the tutorial “Software tests with Faktor-IPS”.