Software Tests with Faktor-IPS

Gunnar Tacke, Jan Ortmann

(Dokumentversion 203)

Overview

In each software development project, software testing entails considerable expenses. Running regression tests manually is a particularly costly approach, so automated regression tests have been around as a best practice for a long time.

In the Java community, JUnit has been successfully used for several years to run regression tests. With JUnit, the tests are written in Java by the developer. Test cases can be executed directly inside the Java development environment. Similarly, JUnit can be integrated in common build tools like Ant and Maven.

We can also use JUnit in Faktor-IPS projects because Faktor-IPS generates testable Java source code. In addition, Faktor-IPS offers its own, extended test support which includes both the definition and execution of test cases.

This tutorial will first explain the underlying concepts and then use an example to demonstrate in detail how this test environment works. Finally we explain how the execution of Faktor-IPS tests can be integrated in build tools.

Conceptual Foundation

In JUnit, usually the Java source code also contains the test data. There is no separation between the test logic and test data. As a result, you can't reuse the same test logic for different test data. Because some Java knowledge is required to define test cases, this task can only be done by Java developers.

For business functions such as the calculation of insurance premiums, the separation of test data and test logic is a tremendous advantage because there are usually many test cases that vary only with respect to the test data. The following table illustrates this by showing three test cases on the premium computation described in the introductory tutorial.

Parameter	***Test Case 1***	***Test Case 2***	***Test Case 3***
Product	*HC-Compact 2009-01*	*HC-Optimal 2009-01*	*HC Optimal 2009-01*
ExtraCoverages	*Bicycle Theft 2009-01* *Overvoltage Damage 2009-01*	*Bicycle Theft 2009-01* *Overvoltage Damage 2009-01*	*Overvoltage Damage 2009-01*
PaymentMode	annual	annual	bi-annual
ZipCode	81673	81673	81673
SumInsured	60,000 EUR	60,000 EUR	100,000 EUR
Expected Results			
NetPremiumPm	196.00 EUR	208.00 EUR	123.60 EUR

All these test cases are processed in the same way:

1. A home contract is created based on the details of the product and the specified extra coverages. The attributes *PaymentMode*, *ZipCode* and *SumInsured* are populated with the

values defined in the test case.

2. Next, the premium is computed by calling the appropriate method on the home contract.
3. The home contract's NetPremiumPm is matched against the expected value that is defined in the test case.

Unlike JUnit, Faktor-IPS separates the test logic from the test data by distinguishing between test case types and test cases themselves. A **test case type** defines the control flow and the structure of the test data, whereas a **test case** instantiates a test case type with specific test data. The test data describe all input values that are required for the test, as well as the expected results.

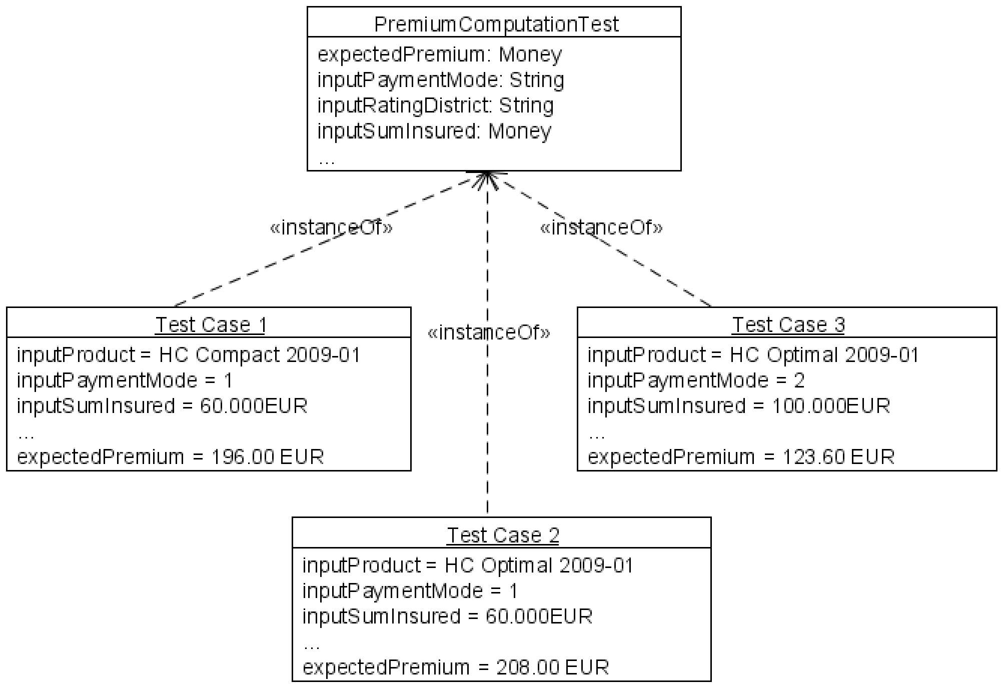

PremiumComputationTest
expectedPremium: Money
inputPaymentMode: String
inputRatingDistrict: String
inputSumInsured: Money
...
«instanceOf»
«instanceOf»
«instanceOf»
Test Case 1
inputProduct = HC Compact 2009-01
inputPaymentMode = 1
inputSumInsured = 60.000EUR
...
expectedPremium = 196.00 EUR
Test Case 3
inputProduct = HC Optimal 2009-01
inputPaymentMode = 2
inputSumInsured = 100.000EUR
...
expectedPremium = 123.60 EUR
Test Case 2
inputProduct = HC Optimal 2009-01
inputPaymentMode = 1
inputSumInsured = 60.000EUR
...
expectedPremium = 208.00 EUR

Figure 1: Test cases as instances of the test case type PremiumComputationTest

Using OO terminology the test case type corresponds to a class and the test case itself is an instance of a test case type. This division is a simple means to foster the separation of roles during test development. The test case types are created by software developers who define the test data structure and write the test logic. Business users can then use these test case types to capture specific test cases with the respective test data.

Testing with Faktor-IPS Using Home Insurance as an Example

As a starting point we will use the home insurance projects we created in the introductory tutorial[1].

First we want to test the premium computation for home contracts. This computation is implemented in such a way that it determines a basic premium according to the sum insured and the selected product. Depending on the rating district (which, in turn, is dependent on the zip code of

1 You can also download the projects ready for import in Eclipse from the Faktor-IPS website.

the respective home contents), this basic premium will then be multiplied by a rating district factor. If extra coverages, such as bicycle theft, are included, a specific percentage of the basic premium will be added, for example, + 10% for bicycle theft. This percentage is configured in the product. Depending on the payment mode, an installment charge may be added as well.

First we will create a test case type for premium computation. This type will constitute the basis of our test cases.

A Test Case Type for the Premium Computation

First we extend the package structure below the source folder named „*model*“ within the project *“org.faktorips.tutorial.en.HomeModel”* by adding an IPS package named „*test*“[2].

Using the context menu option New ► Test Case Type, we create a new test case type in the Model Explorer (alternatively, this can also be done by clicking in the toolbar):

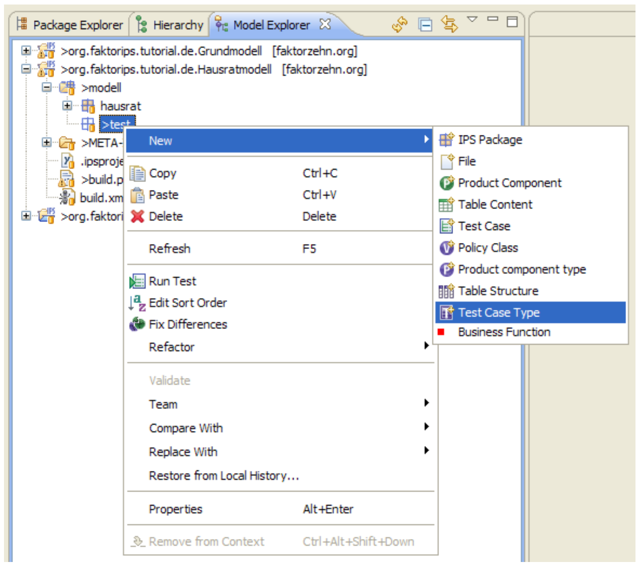


Figure 2: Creating a new test case type

Once we have defined the test case type name *PremiumComputationTest* in the ensuing dialog, the Test Case Type Editor will open (see Figure 2). On the left, you can see the (still empty) structure of the test, and on the right the details of each structural element. Now we can start building the structure of our test case type. Test case types are represented in a tree structure, so we first have to define the root element. To do this, we click New... to open the wizard for creating test parameters. Our first task is to select which kind of test parameter we want to create. There are three possible kinds of parameters (we don't yet take into consideration if they are to be used as input or expected results):

- Policy component type
- Value

  A single value
- Validation Rule

2 In practice it would obviously be better to create a separate source folder for tests. For simplicity, in our example, we will store just everything in the *modell* source folder.

A validation rule to be tested

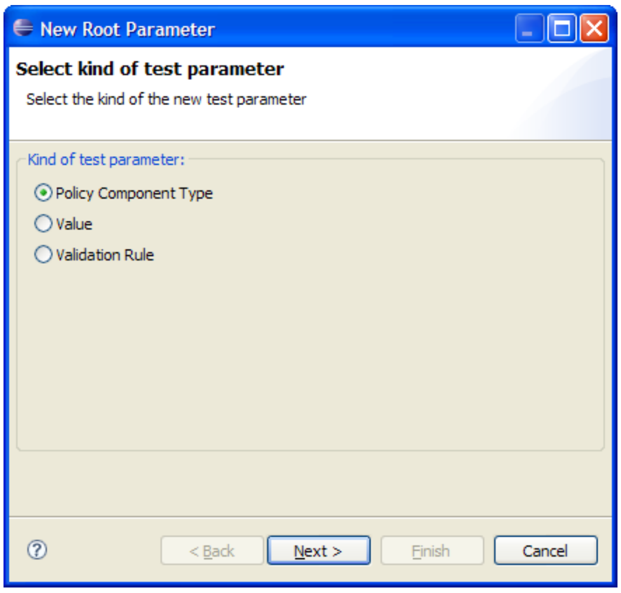


Figure 3: Creating a new root element in the Test Case Editor

We choose Policy Component Type and click Next. On the second page, we select *HomeContract* as data type. By default the parameterhas the same name as the data type, which is just right for our example. In the Type field we can specify the parameter's purpose in the test case:

- Input

  Attributes of the policy component type are only used as input data for test cases

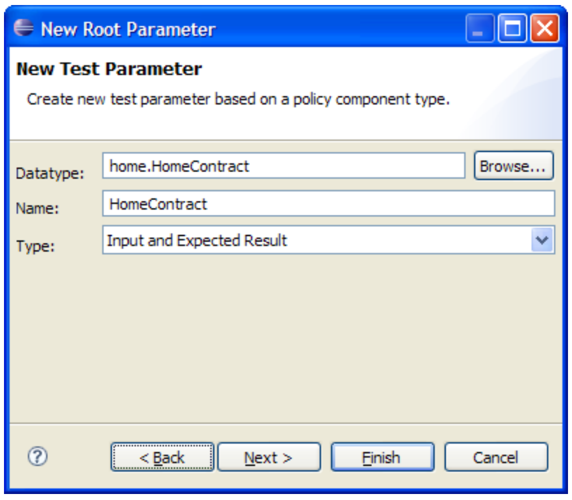
- Expected result

  Attributes of the policy component type are only used as expected result of the test cases

- Input and expected Result

  Attributes of the policy component type can either be used as input parameters or as expected results.

For our example we will choose the parameter type Input and expected Result because we want to define both the input parameters and the expected result (in our case the computed premium) on the home contracts we will create:


New Root Parameter

New Test Parameter

Create new test parameter based on a policy component type.

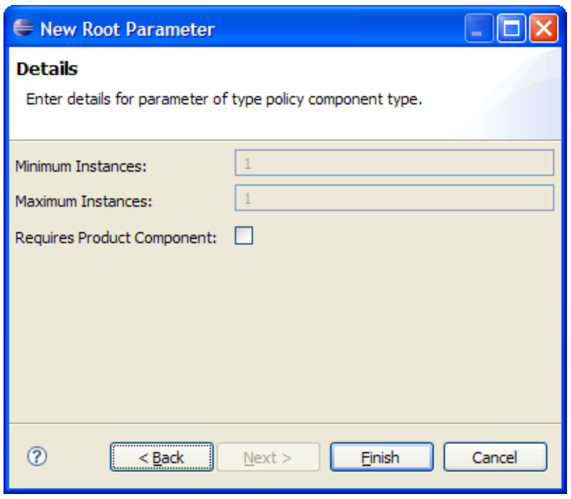
Datatype: home.HomeContract

Name: HomeContract

Type: Input and Expected Result

Abbildung 4: Defining the data type of the test parameter

On the next page of the wizard, you can restrict the parameter cardinality (except for the root parameters, for there can only be one at a time). In addition, you can specify if the policy component must be configured by a product component when defining a test case. By selecting the Requires Product Component checkbox we ensure that for each home contract, the respective home product has to be specified when setting up a test case:


New Root Parameter

Details

Enter details for parameter of type policy component type.

Minimum Instances: 1

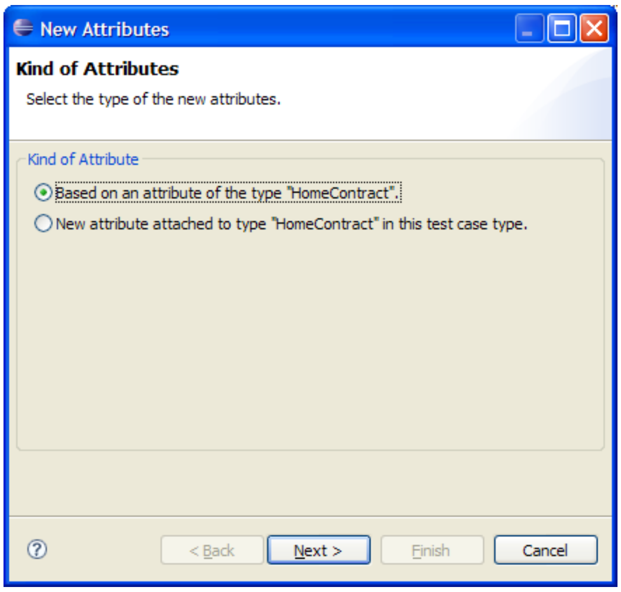
Maximum Instances: 1

Requires Product Component: ☐

Figure 5: Setting cardinality and product dependency

The cardinality setting and the tag that says if a product component will be required or not, can of course be changed later on directly in the editor.

The next step is to specify which attributes have to be tested and which ones are to be used as input parameters. To do this, we choose the structure view (left), select *HomeContract* and click Add... on the right-hand pane.



Kind of Attributes

Select the type of the new attributes.

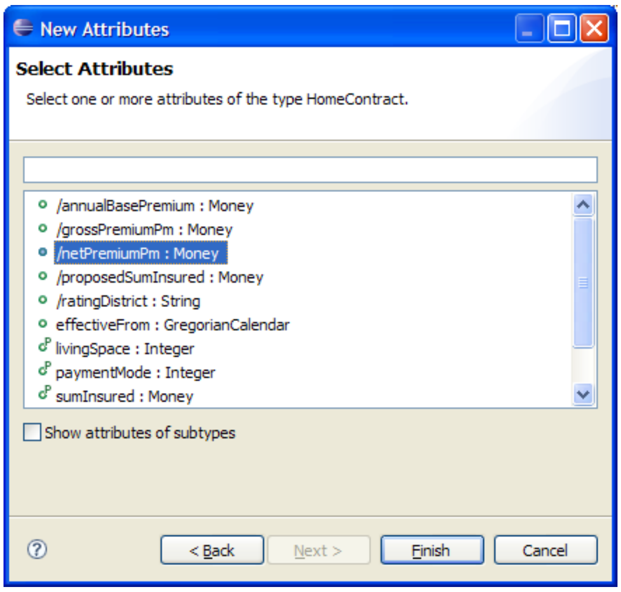
Kind of Attribute

(●) Based on an attribute of the type "HomeContract".

( ) New attribute attached to type "HomeContract" in this test case type.

Figure 6: Creating a test attribute

In the ensuing dialog, you can select attributes of the type *HomeContract* and add them to the test case type.



Select Attributes

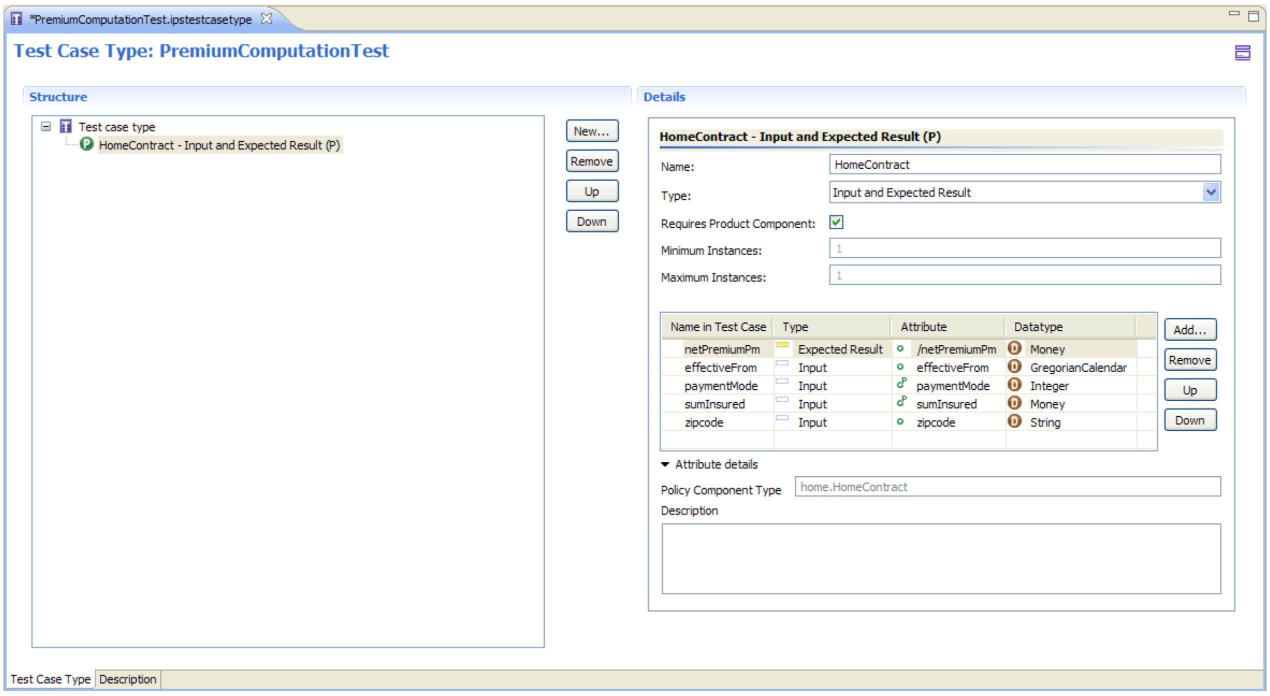
Select one or more attributes of the type HomeContract.

- /annualBasePremium : Money
- /grossPremiumPm : Money
- /netPremiumPm : Money
- /proposedSumInsured : Money
- /ratingDistrict : String
- effectiveFrom : GregorianCalendar
- livingSpace : Integer
- paymentMode : Integer
- sumInsured : Money

[ ] Show attributes of subtypes

Figure 7: Selecting a test attribute

In this case, we capture the parameters *netPremiumPm*, *sumInsured*, *effectiveFrom*, *zipCode*, and *paymentMode*. Then, the editor's Details page will display the attributes and their types, the derived attribute *netPremiumPm* will automatically be predefined with the Expected Result type, while the remaining attributes will serve as our input parameters.


*PremiumComputationTest.ipstestcasetype

Test Case Type: PremiumComputationTest

Structure

- Test case type
  - HomeContract - Input and Expected Result (P)

Details

HomeContract - Input and Expected Result (P)

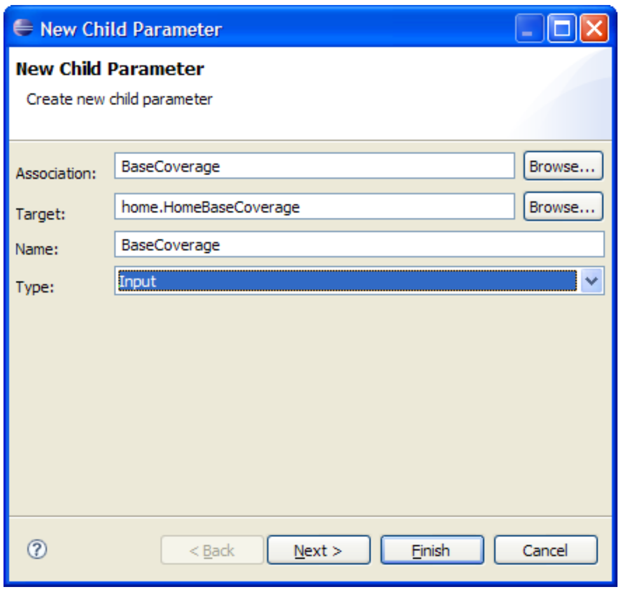
Name: HomeContract
Type: Input and Expected Result
Requires Product Component: ✓
Minimum Instances: 1
Maximum Instances: 1

Name in Test Case	Type	Attribute	Datatype
netPremiumPm	Expected Result	/netPremiumPm	Money
effectiveFrom	Input	effectiveFrom	GregorianCalendar
paymentMode	Input	paymentMode	Integer
sumInsured	Input	sumInsured	Money
zipcode	Input	zipcode	String

Attribute details
Policy Component Type: home.HomeContract
Description

Figure 8: The Test Case Type Editor displaying the Home Contract's attributes being used in the test

Next, we expand the structure of our test case type by adding all other necessary elements to it. In our case, we add the types *HomeBaseCoverage* and *HomeExtraCoverage*. To do this, we select the HomeContract element in the structure and create the relationships *HomeBaseCoverage* and *HomeExtraCoverages* using the New... button. We select the Input type for our *HomeBaseCoverage* (see Figure 9), set a cardinality of 1..1 and select the Requires Product Component checkbox. For the *HomeExtraCoverages* we choose the Input type and a cardinality of 0..*, while selecting the Requires Product Component checkbox as before. Remember that selecting the checkbox means that the respective policy component has to be configured by a product component in the test case. We'll see this later on, when we are going to create a test case.


New Child Parameter

New Child Parameter

Create new child parameter

Association: BaseCoverage

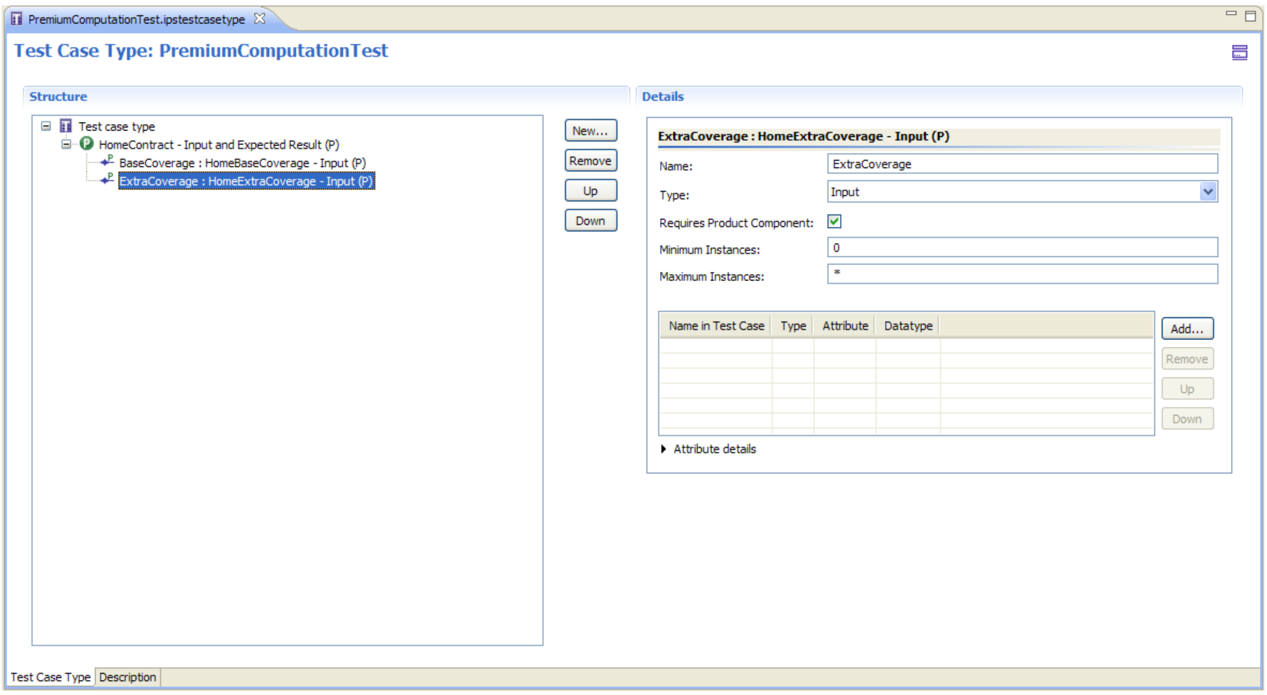
Target: home.HomeBaseCoverage

Name: BaseCoverage

Type: Input

Figure 9: Defining HomeBaseCoverage as a child parameter of HomeContract.

The Test Case Type Editor should now look as follows:


PremiumComputationTest.ipstestcasetype

Test Case Type: PremiumComputationTest

Structure

- Test case type
  - HomeContract - Input and Expected Result (P)
    - BaseCoverage : HomeBaseCoverage - Input (P)
    - ExtraCoverage : HomeExtraCoverage - Input (P)

Details

ExtraCoverage : HomeExtraCoverage - Input (P)

Name: ExtraCoverage

Type: Input

Requires Product Component: ✓

Minimum Instances: 0

Maximum Instances: *

Name in Test Case	Type	Attribute	Datatype

Attribute details

Figure 10: Test Case Type Editor showing the PremiumComputationTest

This way, we have defined that our test case type is appropriate for testing an instance of a home contract, including a base coverage and any number of extra coverages. Each Policy component must relate to a specific product component.

When storing the test case type, a corresponding Java class will be created within the source directory of the package `org.faktorips.tutorial.internal.test`. This Java class will have the same name as the test case type and it will be used to implement the test logic. Let's now go to the Package Explorer and have a closer look at the structure of this generated class:

```java
public class PremiumComputationTest extends IpsTestCase2 {

    private HomeContract inputHomeContract;

    private HomeContract expectedHomeContract

    public void executeBusinessLogic() {
        //...
    }

    public void executeAsserts(IpsTestResult result) {
        //...
        throw new RuntimeException(
            "No asserts implemented in the Java class that represents the test case type.");
    }
}
```

- Member variables `inputHomeContract` and `expectedHomeContract` of type `HomeContract`:

  These correspond to the root parameter of type *HomeContract*. As we have declared the root parameter as input and expected result, two appropriate instance variables have been created: `inputHomeContract` stores the test case's input values according to the definition that is provided by the test case type. `expectedHomeContract` contains the expected results of the test case (again according to the test case type definition). Hence, at test case run time, we get two `HomeContract` instances that we can match against each other.

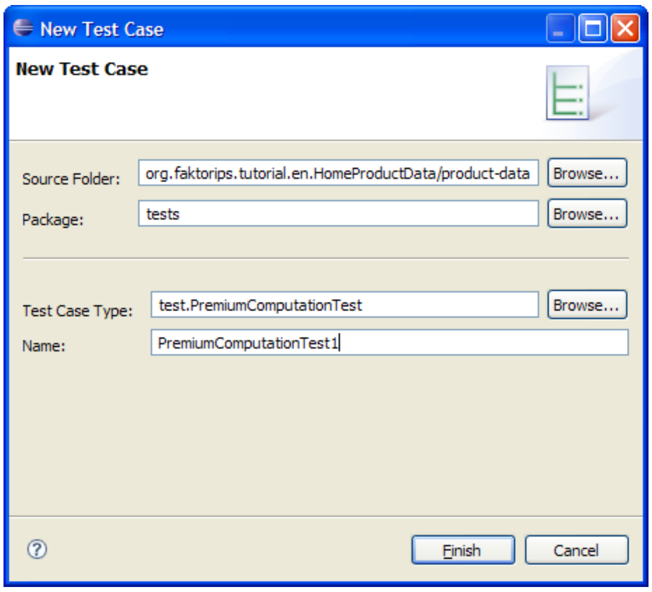
- Empty method `executeBusinessLogic()`:
  The business logic we want to test will be called inside this method, for example, by calling a method on the input objects (in this case `inputHomeContract`). This method is executed before the `executeAsserts(...)` method gets called.

- Test method `executeAsserts()`:
  This method implements the comparison of actual and expected values. Initially it contains a generated default implementation that will make the test fail, so the developer is forced to substitute his/her own implementation.

Before providing a final implementation of our `PremiumComputationTest` class, we create a test case based on our test case type.

Creating a Test Case

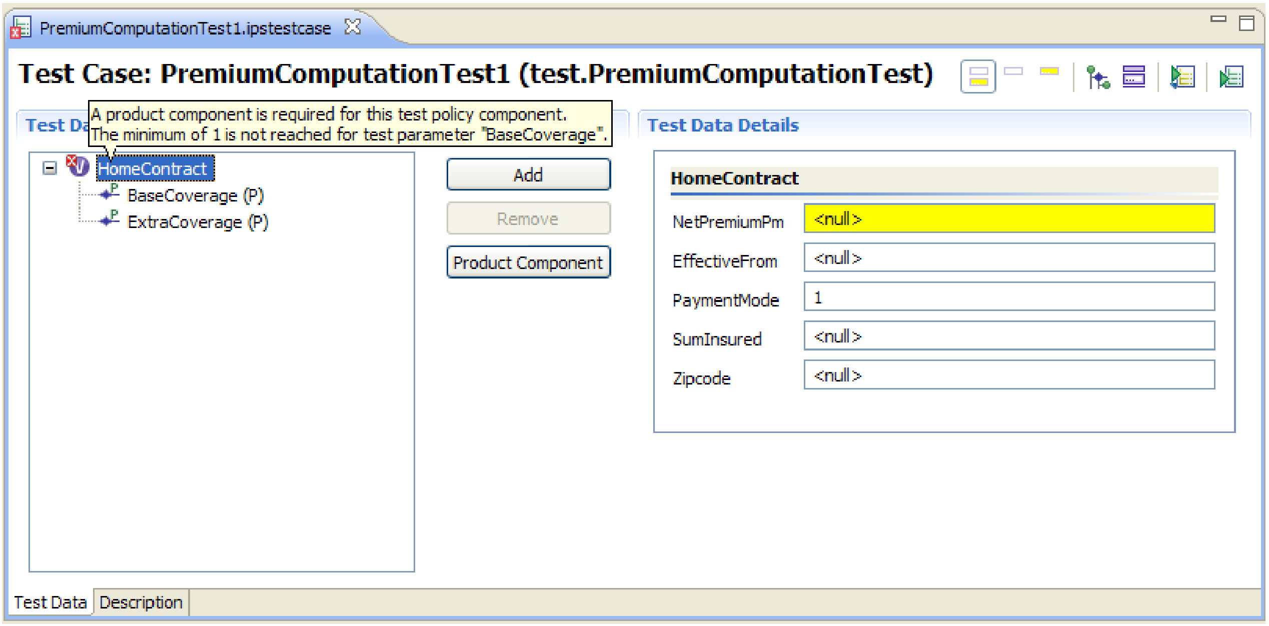
In the project "*org.faktorips.tutorial.en.HomeProductData*" we add a new IPS Package named *tests* under the *product-data* directory. This package will be used to hold our test cases. To do this, we return to the Model Explorer, select the *product-data* directory and choose New ► IPS Package from the context menu. Using New ► Testcase (once again in the context menu), we create a new test case. In the wizard we select the previously created test case type named *test.PremiumComputationTest* and give the the test case a name.



Source Folder: org.faktorips.tutorial.en.HomeProductData/product-data
Package: tests
Test Case Type: test.PremiumComputationTest
Name: PremiumComputationTest1

Figure 11: Creating a new test case

After clicking Finish the Test Case Editor will open, displaying the structure of the test case on the left. This structure corresponds to the one we have defined in the test case type. On the right, you can see the test data. For each attribute defined in the test case type, an input field is provided, with the input values on a white background and the expected values highlighted in yellow.



Test Case: PremiumComputationTest1 (test.PremiumComputationTest)

A product component is required for this test policy component.
The minimum of 1 is not reached for test parameter "BaseCoverage".

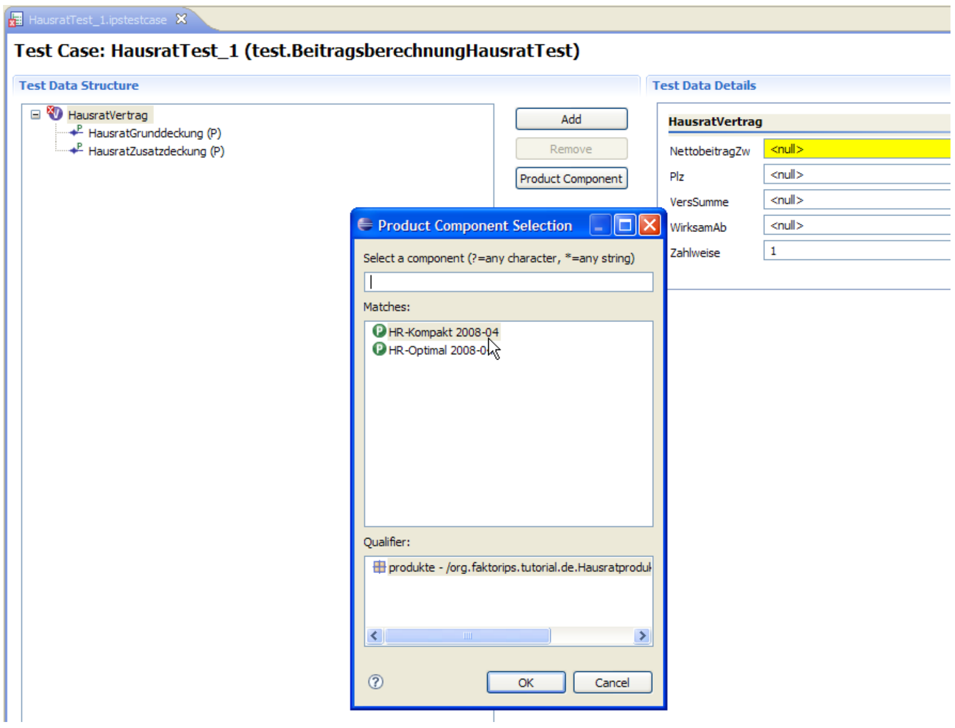
HomeContract

NetPremiumPm	<null>
EffectiveFrom	<null>
PaymentMode	1
SumInsured	<null>
Zipcode	<null>

Figure 13: The Test Case Editor hinting at a lacking product component

First we have to assign a product to the *HomeContract*. This is done by selecting the *HomeContract* in the test structure and clicking on the Product Component button.

For our example, we assign the product component *HC-Compact 2009-01* to the *HomeContract*. We then use the Add option to add further objects, including a *BaseCoverage-Compact 2009-01* for



Test Case: HausratTest_1 (test.BeitragsberechnungHausratTest)

Test Data Structure

HausratVertrag
HausratGrunddeckung (P)
HausratZusatzdeckung (P)

Test Data Details

HausratVertrag

NettobeitragZw	<null>
Plz	<null>
VersSumme	<null>
WirksamAb	<null>
Zahlweise	1

Product Component Selection

Select a component (?=any character, *=any string)

Matches:
HR-Kompakt 2008-04
HR-Optimal 2008-0

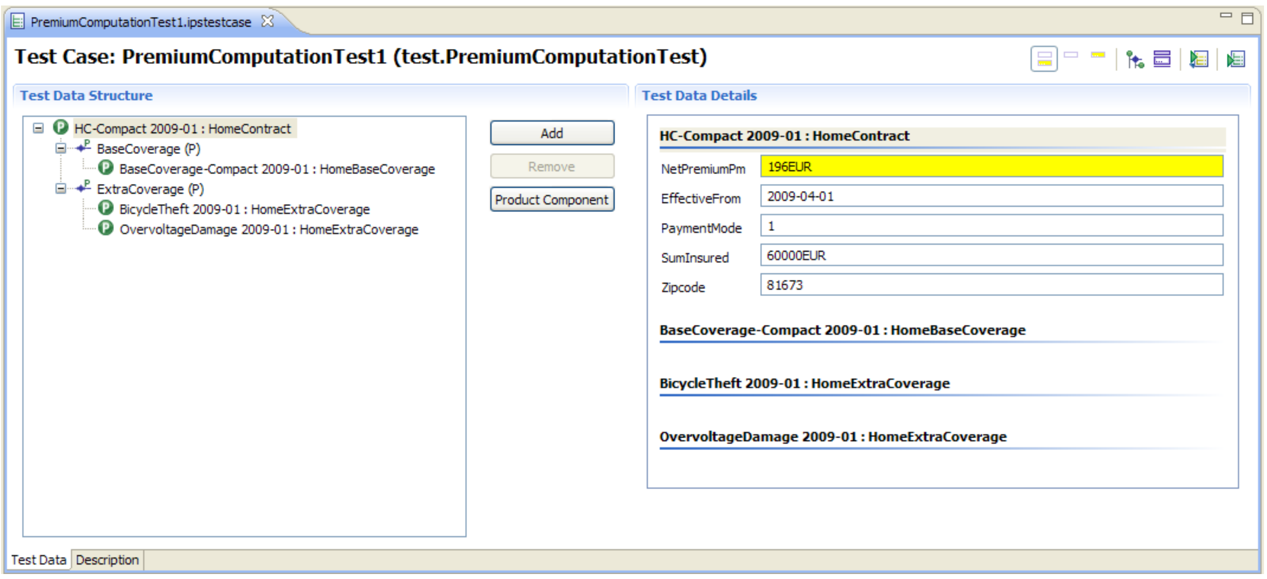
Qualifier:
produkte - /org.faktorips.tutorial.de.Hausratprodul

Figure 14: Selecting a product componenet using,,Product Component"

HomeBaseCoverage and two instances of HomeExtraCoverage, one with *BicycleTheft 2009-01* and one with *OvervoltageDamage 2009-01*. Finally we complete the test case according to the following table:

Parameter	***Value***
Product	HC-Compact 2009-01
Base Coverage Type	BaseCoverage-Compact 2009-01
Extra Coverage Types	BicycleTheft 2009-01 OvervoltageDamage 2009-01
Sum Insured	60,000 EUR
Effective From	2009-04-01
ZIP Code	81673 (RatingDistrict I)
Payment Mode	1 (annually)
Net Premium according to Payment Mode	*196,00 EUR*

Table 1: Test data for the premium computation


PremiumComputationTest1.ipstestcase

Test Case: PremiumComputationTest1 (test.PremiumComputationTest)

Test Data Structure

- HC-Compact 2009-01 : HomeContract
  - BaseCoverage (P)
    - BaseCoverage-Compact 2009-01 : HomeBaseCoverage
  - ExtraCoverage (P)
    - BicycleTheft 2009-01 : HomeExtraCoverage
    - OvervoltageDamage 2009-01 : HomeExtraCoverage

Test Data Details

HC-Compact 2009-01 : HomeContract

NetPremiumPm	196EUR
EffectiveFrom	2009-04-01
PaymentMode	1
SumInsured	60000EUR
Zipcode	81673

BaseCoverage-Compact 2009-01 : HomeBaseCoverage

BicycleTheft 2009-01 : HomeExtraCoverage

OvervoltageDamage 2009-01 : HomeExtraCoverage

Figure 15: Test Case Editor after all the data has been entered

We now start to execute our test case by clicking the icon Run Test( ) in the upper right corner of the test case editor. (There are two ways to start a test case, we simply use Run test. The other variant, Run test and store differences, will be shown later in the chapter "Creating Test Cases by Copying them").

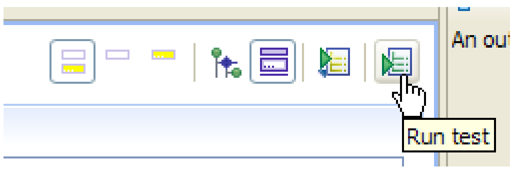
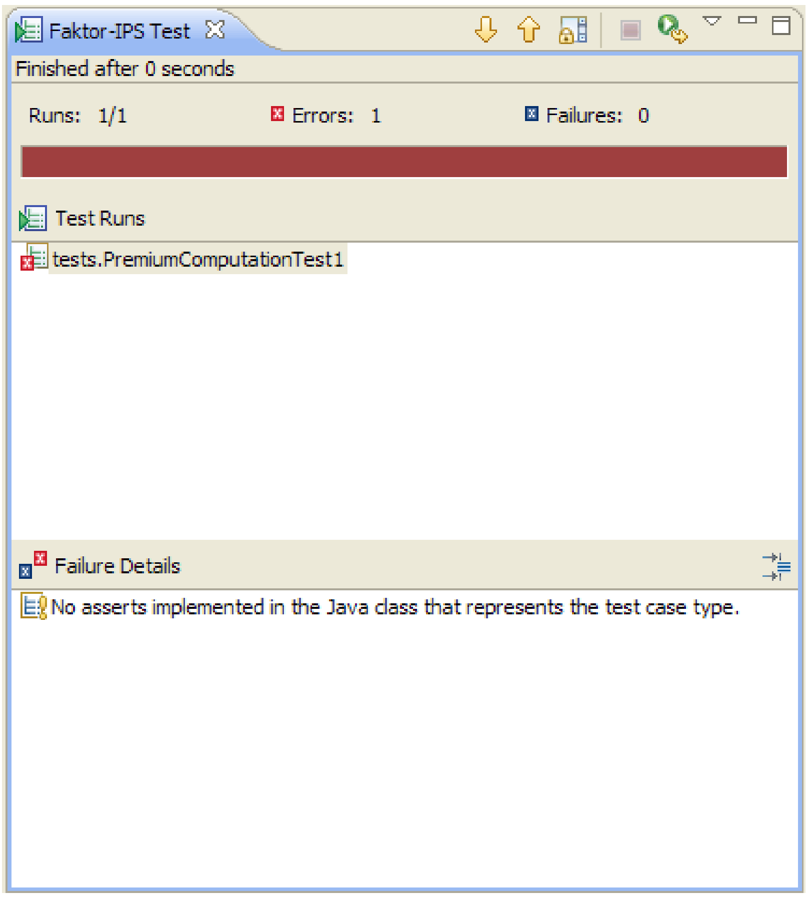


Figure 16: Running a test case using the Run Test - Icon



Finished after 0 seconds

Runs: 1/1 Errors: 1 Failures: 0

Test Runs

tests.PremiumComputationTest1

Failure Details

No asserts implemented in the Java class that represents the test case type.

Figure 17: The test case execution with the Testrunner fails

Faktor-IPS tells us that the test has failed by displaying red bars in the editor's title section and inside the Testrunner. We look up the Failure Details to see the reason for this failure and we realize that we still have to implement the assert-statements in the test case type. (In place of the asserts, a runtime exception is generated, whose message shows up here). So now we're going to fix this:

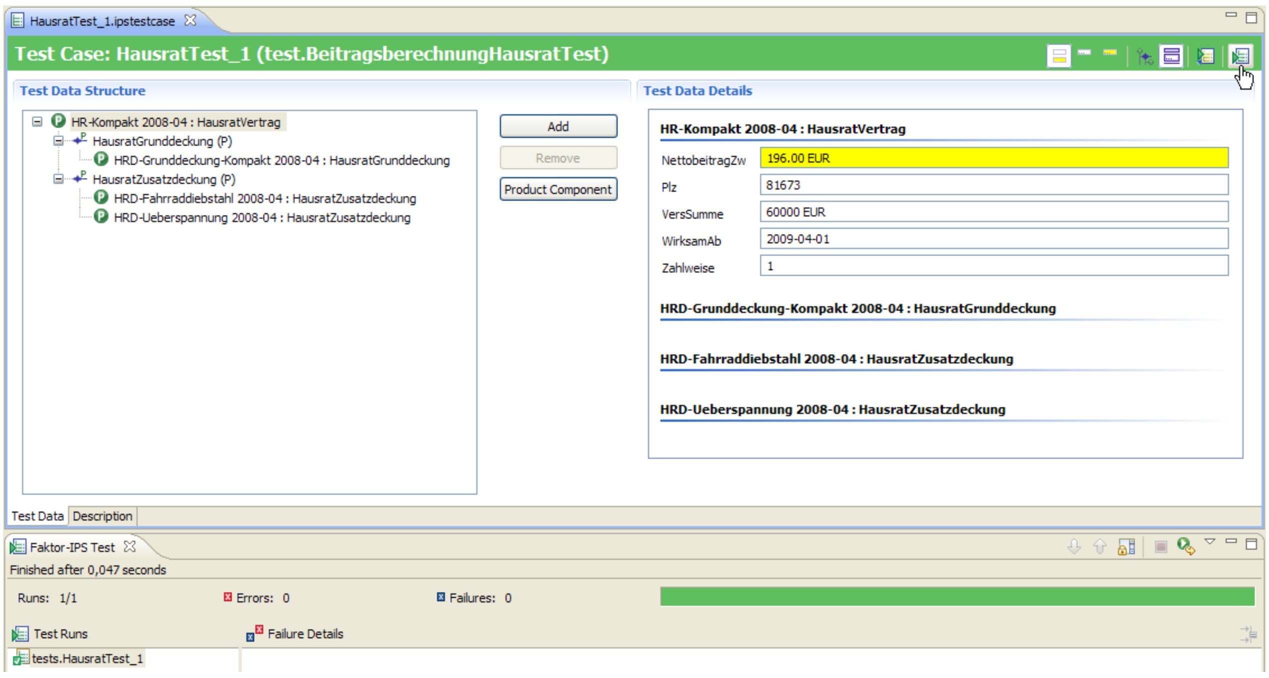
```java
public class PremiumComputationTest extends IpsTestCase2 {
//...

/**
 * Executes the business logic to be tested.
 *
 * @generated NOT
 */
public void executeBusinessLogic() {
    inputHomeContract.computePremium();
}

/**
 * Executes the checks (asserts), i.e., matches the expected
 * values against the actual values.
 *
 * @generated NOT
 */
public void executeAsserts(IpsTestResult result) {
    assertEquals(expectsHomeContract.getNetPremiumPm(),
                 inputHomeContract.getNetPremiumPm(),
                 result);
}
```

Inside the `executeBusinessLogic()` method we call the business logic to be tested:
To do this, we first call the `computePremium()` method on the input instance. Faktor-IPS ensures that the instance is provided with the current test case's input value at runtime. We implement the checks in the `executeAsserts(...)` method. In our case we want to check if the expected net premium is equal to the computed net premium. To do this, we use the `assert*` methods of the `IpsTestCase2` class. Remember that `@generated` must be followed by `NOT` so that the code you added manually is not overridden!

Now we will run our test case again. The green bars inside both the Test Case Editor and the TestRunner show us that the expected result and the computed result are the same.


HausratTest_1.ipstestcase
Test Case: HausratTest_1 (test.BeitragsberechnungHausratTest)
Test Data Structure
HR-Kompakt 2008-04 : HausratVertrag
HausratGrunddeckung (P)
HRD-Grunddeckung-Kompakt 2008-04 : HausratGrunddeckung
HausratZusatzdeckung (P)
HRD-Fahrraddiebstahl 2008-04 : HausratZusatzdeckung
HRD-Ueberspannung 2008-04 : HausratZusatzdeckung
Add
Remove
Product Component
Test Data Details
HR-Kompakt 2008-04 : HausratVertrag

NettobeitragZw	196.00 EUR
Plz	81673
VersSumme	60000 EUR
WirksamAb	2009-04-01
Zahlweise	1

HRD-Grunddeckung-Kompakt 2008-04 : HausratGrunddeckung
HRD-Fahrraddiebstahl 2008-04 : HausratZusatzdeckung
HRD-Ueberspannung 2008-04 : HausratZusatzdeckung
Test Data
Description
Faktor-IPS Test

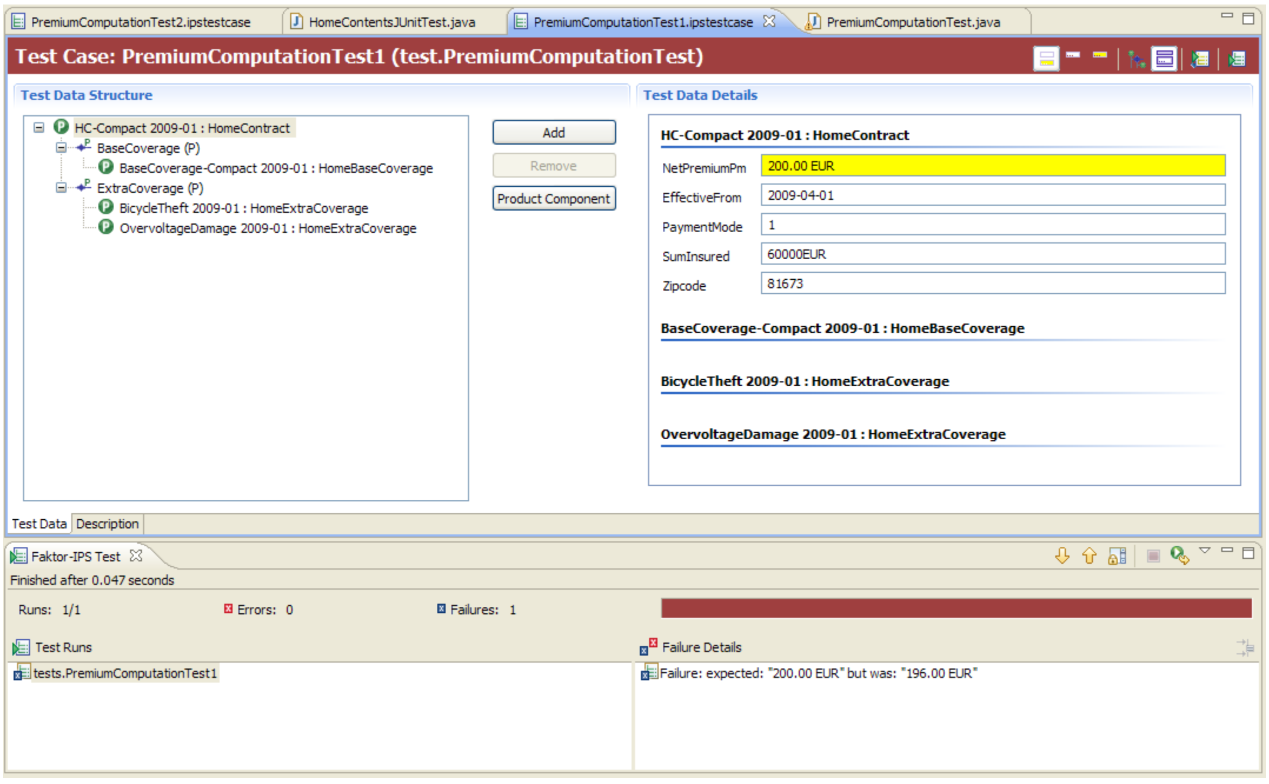
Finished after 0,047 seconds

Runs: 1/1 Errors: 0 Failures: 0

Test Runs
Failure Details
tests.HausratTest_1

Figure 18: A successful test

Let's make a cross check by changing the expected result to 200EUR. The test case fails. In the Failure Details you can see that the computed value of 196 EUR is not the same as the expected value.


PremiumComputationTest2.ipstestcase
HomeContentsJUnitTest.java
PremiumComputationTest1.ipstestcase
PremiumComputationTest.java
Test Data Structure
Test Data Details
Add
Remove
Product Component
Test Data
Description
Faktor-IPS Test
Test Runs
Failure Details

Test Case: PremiumComputationTest1 (test.PremiumComputationTest)

HC-Compact 2009-01 : HomeContract
BaseCoverage (P)
BaseCoverage-Compact 2009-01 : HomeBaseCoverage
ExtraCoverage (P)
BicycleTheft 2009-01 : HomeExtraCoverage
OvervoltageDamage 2009-01 : HomeExtraCoverage

HC-Compact 2009-01 : HomeContract

NetPremiumPm	200.00 EUR
EffectiveFrom	2009-04-01
PaymentMode	1
SumInsured	60000EUR
Zipcode	81673

BaseCoverage-Compact 2009-01 : HomeBaseCoverage

BicycleTheft 2009-01 : HomeExtraCoverage

OvervoltageDamage 2009-01 : HomeExtraCoverage

Finished after 0.047 seconds

Runs: 1/1 Errors: 0 Failures: 1

tests.PremiumComputationTest1

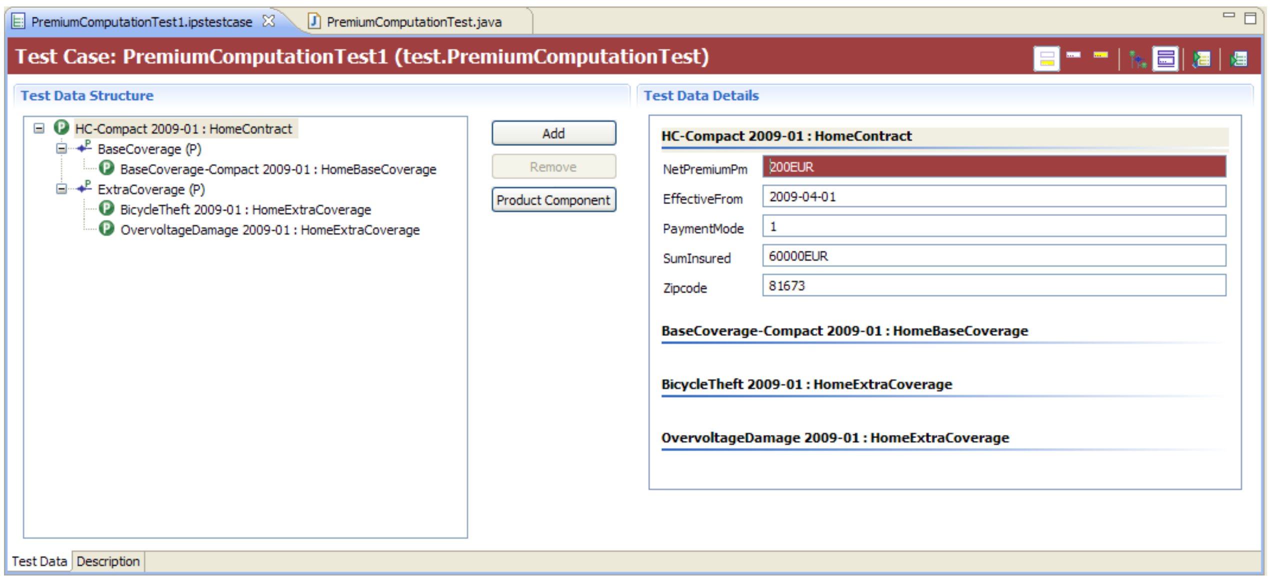
Failure: expected: "200.00 EUR" but was: "196.00 EUR"

Figure 19: The test case fails

Unfortunately, the error message doesn't yet tell us which attribute it is referring to. In this case, its not a problem as we have got only one expected value, but there can be be cases where you want to compare multiple attributes in one test case – for example, the individual premiums per coverage and the total premium. To get more detailed information about the failure, you can pass the `assert*` statements a reference to the attribute. So we pass two additional parameters; the first one is a string that identifies the test object and the second one is the name of the incorrect attribute. If there are multiple instances of an object in the test, they have to be suffixed by a hash (#) sign followed by an index of the instance, where the index numbers start by 0, as in `ExtraCoverage#0` for the first instance of the *ExtraCoverage* object. The implementation for the `netPremiumPm` attribute in our example will then look as follows:

```
/**
 * Executes the checks (asserts), i.e., matches the expected
 * values against the actual values.
 *
 * @generated NOT
 */
public void executeAsserts(IpsTestResult result) {
    assertEquals(expectedHomeContract.getNetPremiumPm(),
                 inputHomeContract.getNetPremiumPm(),
                 result,
                 "HomeContract#0",
                 IHomeContract.PROPERTY_NETPREMIUMPM);
}
```

If we execute the test case again, (still with the „wrong“ expected result), the respective attribute will also be highlighted in red on the user interface and a message in the Failure Details will tell us exactly which attribute it is referring to:


PremiumComputationTest1.ipstestcase
PremiumComputationTest.java
Add
Remove
Product Component
Test Data
Description

Test Case: PremiumComputationTest1 (test.PremiumComputationTest)

Test Data Structure

- HC-Compact 2009-01 : HomeContract
  - BaseCoverage (P)
    - BaseCoverage-Compact 2009-01 : HomeBaseCoverage
  - ExtraCoverage (P)
    - BicycleTheft 2009-01 : HomeExtraCoverage
    - OvervoltageDamage 2009-01 : HomeExtraCoverage

Test Data Details

HC-Compact 2009-01 : HomeContract

NetPremiumPm	200EUR
EffectiveFrom	2009-04-01
PaymentMode	1
SumInsured	60000EUR
Zipcode	81673

BaseCoverage-Compact 2009-01 : HomeBaseCoverage

BicycleTheft 2009-01 : HomeExtraCoverage

OvervoltageDamage 2009-01 : HomeExtraCoverage

Figure 20: The attribute that caused the error is highlighted in the Test Editor

So now we have completely implemented a test case type and created and run a test case.

Special case: Testing Derived Attributes

So far, we have implemented the comparison logic in the test case type based on the comparison of member variables of the test objects. In the *Input* instance, we had the attribute computed, while in the *Expected* instance, we entered it into the Test Editor. Then we compared the attributes of both instances. The *netPremiumPm* attribute we used so far, is a derived attribute for which a member variable is generated. The value of this variable is computed by explicitly calling a method (in this case `inputHomeContract.computePremium()`). This type of derived attributes can be tested just like mutable attributes.

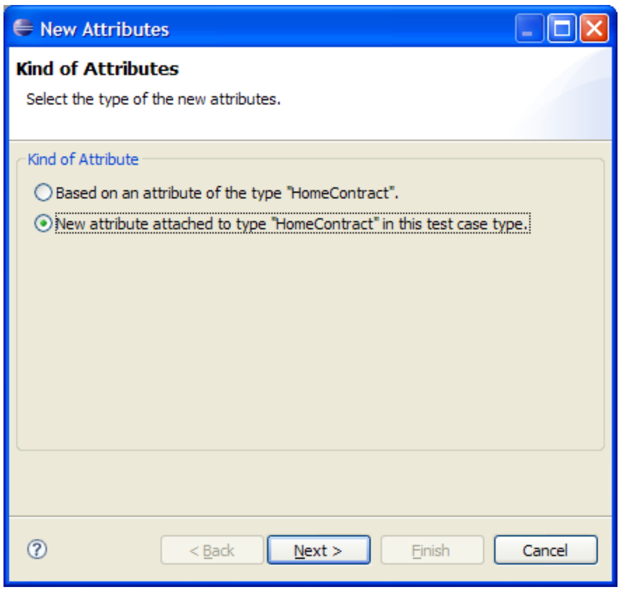
In Faktor-IPS we may also define derived attributes that are computed „on the fly“ each time the getter method gets called. No member variable is generated for these attributes. Thus, they constitute a special case for test implementations because in our *Expected* instance, we don't have any member variable that can possibly be used in a comparison operation.

The home contract's r*atingDistrict* is such an attribute. We will now create a new test case type (named *RatingDistrictTest*) in order to verify the rating district computation based on the zip code.

We define the attributes *zipCode* and *effectiveFrom* of the type *HomeContract* as input parameters and we enable the Requires Product Component[3] checkbox for *HomeContract*. Now we attach a new

3 In the home contents example the RatingDistrict is independent of the selected product, whereas our implementation in the basic tutorial uses the related product component to get a reference to the RuntimeRepository from which the RatingDistrict_Table is loaded. While the RatingDistrict discovery doesn't care about which product is assigned in the test case, we make the home contract configurable so that our implementation in the test case type can work. In practice, it is conceivable to have product-dependent or generation-dependent RatingDistrict tables, for example, to account for changes over time.

attribute called *exptectedDistrict* to the type *HomeContract* by clicking the Add button.



New Attributes

Kind of Attributes

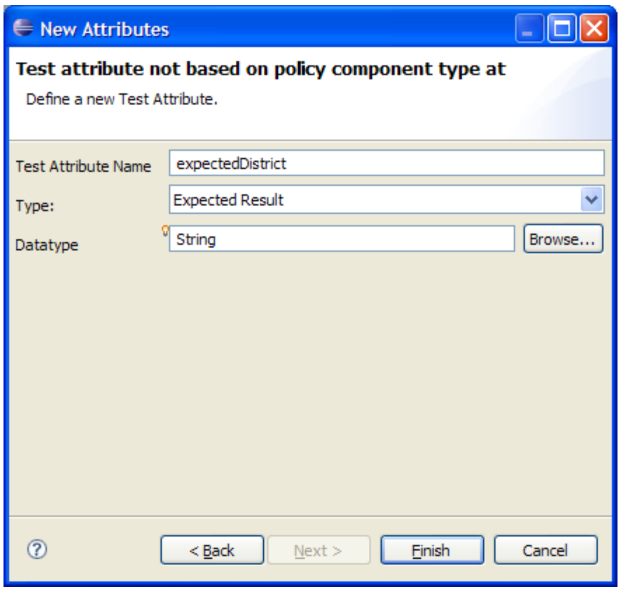
Select the type of the new attributes.

Kind of Attribute

( ) Based on an attribute of the type "HomeContract".

[✓ New attribute attached to type "HomeContract" in this test case type.]

Figure 21: Attach the new attribute "expectedDistrict" to type HomeContract. Step 1


New Attributes

Test attribute not based on policy component type at

Define a new Test Attribute.

Test Attribute Name: expectedDistrict

Type: Expected Result

Datatype: String

Figure 22: Attach the new attribute "expectedDistrict" to type HomeContract. Step 2

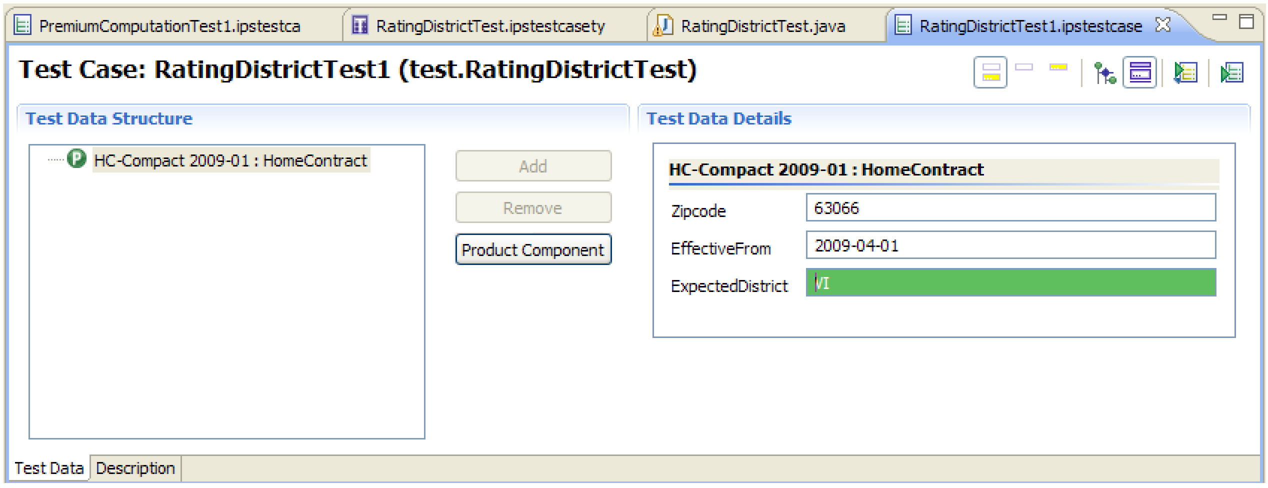
Now save the test case type and open the Java class `RatingDistrictTest` that is generated for it. In the Java source code you can access the value of "attached" attributes via `getExtensionAttributeValue(String attrName)`. Instead of hard coding the attribute name you should use the constant that is generated for it. The following code section shows the details.

```
public class RatingDistrictTest extends IpsTestCase2 {

public final static String
    TESTATTR_HOMECONTRACT_EXPECTEDDISTRICT = "expectedDistrict";
    //...
    public void executeBusinessLogic() {
        // nothing to do (the logic to be tested is run by calling the getter)
    }

    /**
     * Executes the checks (asserts), i.e., matches the expected
     * values against the actual values.
     *
     * @generated NOT
     */
    public void executeAsserts(IpsTestResult result) {
        String expectedDistrict = (String) getExtensionAttributeValue(
            expectedHomeContract, TESTATTR_HOMECONTRACT_EXPECTEDDISTRICT);
        String computedDistrict = inputHomeContract.getRatingDistrict();
        assertEquals(expectsRatingDistrict, computedDistrict, result,
              "HomeContract#0", TESTATTR_HOMECONTRACT_EXPECTEDDISTRICT);
      }
//...
}
```

In order to verify this, we create a test case (*RatingDistrictTest_1*) based on the new test case type and populate the test data for *EffectiveFrom* and *ZipCode*. In addition, we assign a home product (e.g., *HC-Compact 2009-01*) to the contract. According to the rating district table we expect the district VI for zip code 63066. After running the test, a green bar confirms that our test case is correct.



Test Case: RatingDistrictTest1 (test.RatingDistrictTest)

HC-Compact 2009-01 : HomeContract

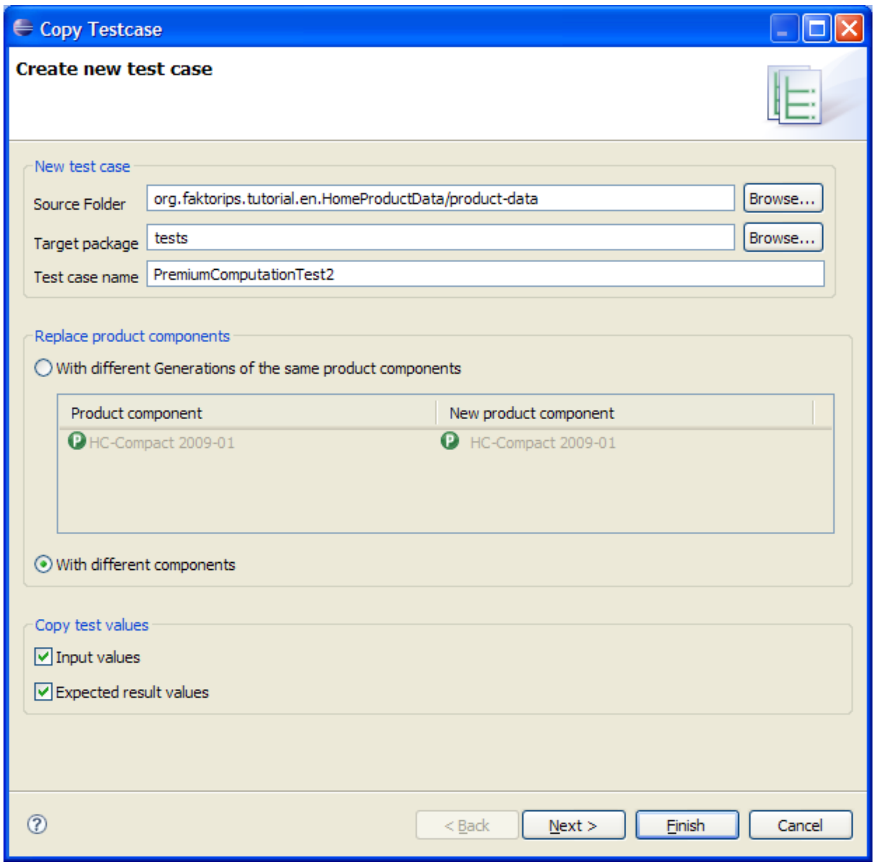
HC-Compact 2009-01 : HomeContract	
Zipcode	63066
EffectiveFrom	2009-04-01
ExpectedDistrict	VI

Figure 23: A test for the rating district evaluation

Creating Test Cases by Copying them

In the following example we will show you how to create and customize new tests by simply copying them. We want to create a premium computation test for the product *HC-Optimal 2009-01* instead of *HC-Compact 2009-01*.

To do this, we select the test case we want to copy - *PremiumComputationTest1* - in the Model Explorer and call New ► Copy Test Case ... from the context menu. The wizard shown in the next figure guides us through the creation of the test case.



Copy Testcase

Create new test case

New test case

Source Folder: org.faktorips.tutorial.en.HomeProductData/product-data

Target package: tests

Test case name: PremiumComputationTest2

Replace product components

With different Generations of the same product components

Product component	New product component
HC-Compact 2009-01	HC-Compact 2009-01

[✓ With different components]

Copy test values

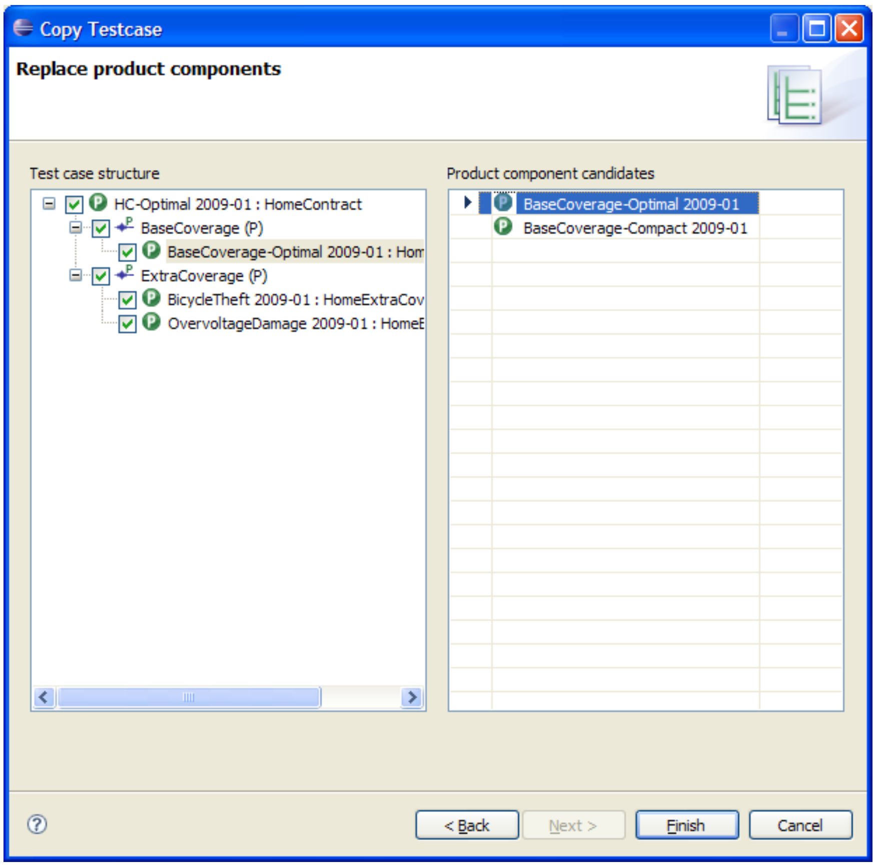
[✓] Input values

[✓] Expected result values

Figure 24: Copy test case wizard, Page 1

We give the test case a name and, using the radio button With different components, we decide that we want to replace the product components of the source test case by other product components. As we want our target test case to inherit the existing input values and the expected results, we leave both Copy test values checkboxes enabled and click Next > to confirm our settings.

We are now able to cancel or replace product components. In order to replace a product component, we select it in the left pane of the the structure view. As a result, the list in the right-hand pane now displays all product components that are suitable for this relationship. We then select the new product component and replace *HC-Compact 2009-01* by *HC-Optimal 2009-01* and *BaseCoverage-Compact 2009-01* by *BaseCoverage-Optimal 2009-01*, respectively.



Copy Testcase

Replace product components

Test case structure

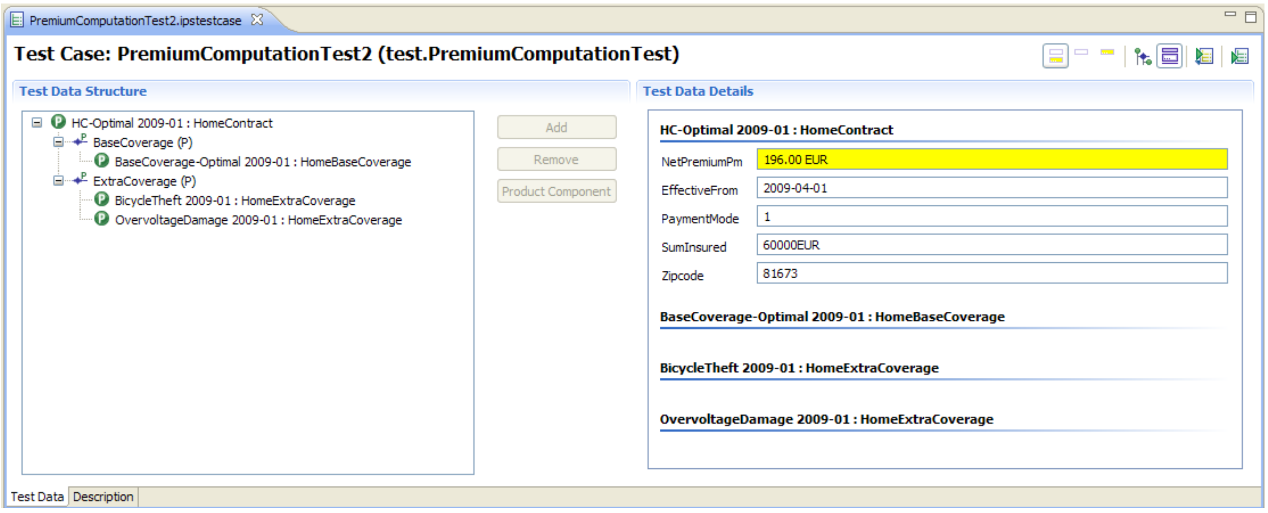
- HC-Optimal 2009-01 : HomeContract
  - BaseCoverage (P)
    - BaseCoverage-Optimal 2009-01 : Hom
  - ExtraCoverage (P)
    - BicycleTheft 2009-01 : HomeExtraCov
    - OvervoltageDamage 2009-01 : HomeE

Product component candidates

- BaseCoverage-Optimal 2009-01
- BaseCoverage-Compact 2009-01

Figure 25: Replacing product components in a test copy

Once we click Finish to exit the wizard, the new test case will be created and opened in the Test Case Editor.



PremiumComputationTest2.ipstestcase

Test Case: PremiumComputationTest2 (test.PremiumComputationTest)

Test Data Structure

- HC-Optimal 2009-01 : HomeContract
  - BaseCoverage (P)
    - BaseCoverage-Optimal 2009-01 : HomeBaseCoverage
  - ExtraCoverage (P)
    - BicycleTheft 2009-01 : HomeExtraCoverage
    - OvervoltageDamage 2009-01 : HomeExtraCoverage

Test Data Details

HC-Optimal 2009-01 : HomeContract

NetPremiumPm	196.00 EUR
EffectiveFrom	2009-04-01
PaymentMode	1
SumInsured	60000EUR
Zipcode	81673

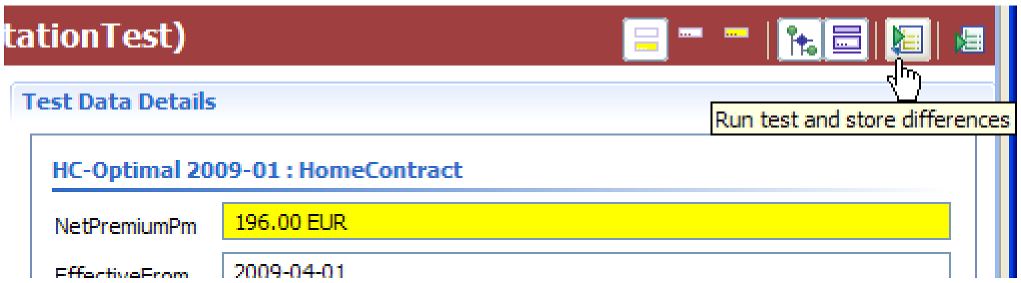
BaseCoverage-Optimal 2009-01 : HomeBaseCoverage

BicycleTheft 2009-01 : HomeExtraCoverage

OvervoltageDamage 2009-01 : HomeExtraCoverage

Figure 26: A copied test case

If we execute the test case, we get a deviation in our expected result because the HC-Optimal product has, among others, different premiums than HC-Compact. However, there is a simple means to accept the computed result as the expected result. We just have to click the Run test and store differences icon:

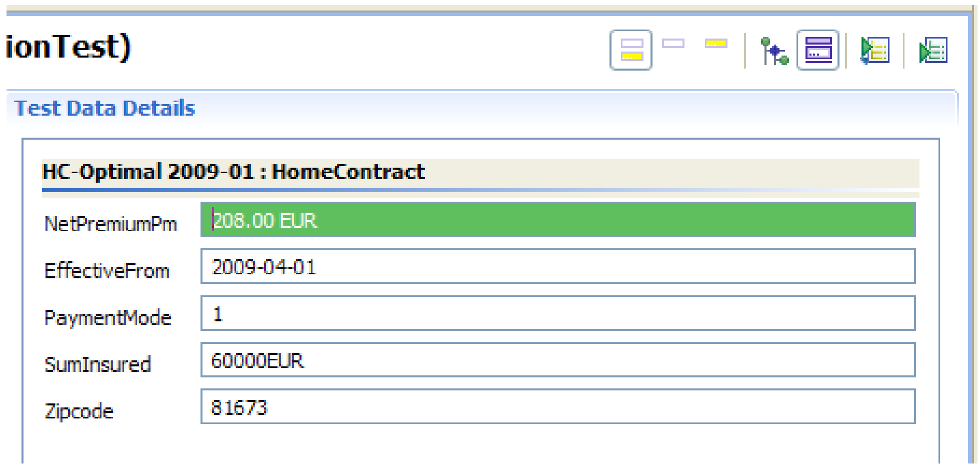


HC-Optimal 2009-01 : HomeContract

NetPremiumPm	196.00 EUR
EffectiveFrom	2009-04-01

Figure 27: Run test and store differences

The test case will now be run and the computed results will be imported in our test case. So this is a straightforward approach to determine the expected result and to construct a correct test case.



HC-Optimal 2009-01 : HomeContract

NetPremiumPm	208.00 EUR
EffectiveFrom	2009-04-01
PaymentMode	1
SumInsured	60000EUR
Zipcode	81673

Figure 28: The computed values have been applied to the test case

A JUnit-Adapter for Faktor-IPS Test Cases

The Faktor-IPS Runtime includes an adapter to convert Faktor-IPS test cases into JUnit test cases or test suites. This way, Faktor-IPS test cases can also be executed with Ant or Maven, because these tools provide a suitable JUnit integration. At the same time, this enables effortless automatic execution of Faktor-IPS test cases within a continuous integration environment.

We can use the following code to create an adapter that converts our Faktor-IPS test cases into a JUnit test suite:

```
package org.faktorips.tutorial.test;

public class HomeContentsJUnitTest extends IpsTestSuiteJUnitAdapter {

    public static Test suite() throws ParserConfigurationException{
        IRuntimeRepository repositoryHome = new ClassloaderRuntimeRepository(
            HomeContentsJUnitTest.class.getClassLoader(),
            "org.faktorips.tutorial.productdata.internal");
            return createJUnitTest(repositoryHome.getIpsTest(""));
      }
}
```

We create a `RuntimeRepository` populated with the product data and call the `getIpsTest()` method to provide us a test suite with all the test cases stored in the repository. The `createJUnitTest(...)` method of class `IpsTestSuiteJUnitAdapter` then takes this test suite and makes it a JUnit test suite. If we execute the `HomeContentsJUnitTest` test class with the JUnit-Testrunner, we can see how our Faktor-IPS test cases are executed and interpreted by JUnit.

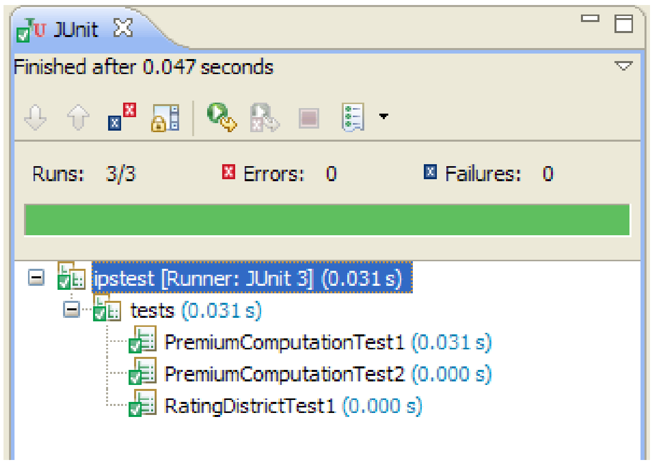

JUnit
Finished after 0.047 seconds
Runs: 3/3
Errors: 0
Failures: 0
ipstest [Runner: JUnit 3] (0.031 s)
tests (0.031 s)
PremiumComputationTest1 (0.031 s)
PremiumComputationTest2 (0.000 s)
RatingDistrictTest1 (0.000 s)

Figure 29: Running a test adapter in the JUnit GUI

Summary

In a typical Faktor-IPS project, approx. 60-80% of the business model are generated. This generated part of the source code needs no further testing because it was already tested once and for all in the course of the Faktor-IPS development. Tests for the remaining, custom made part of the source code can be defined with JUnit or using the Faktor-IPS test functions. The following diagram shows criteria that can help you to decide which approach is better suited under the given circumstances.

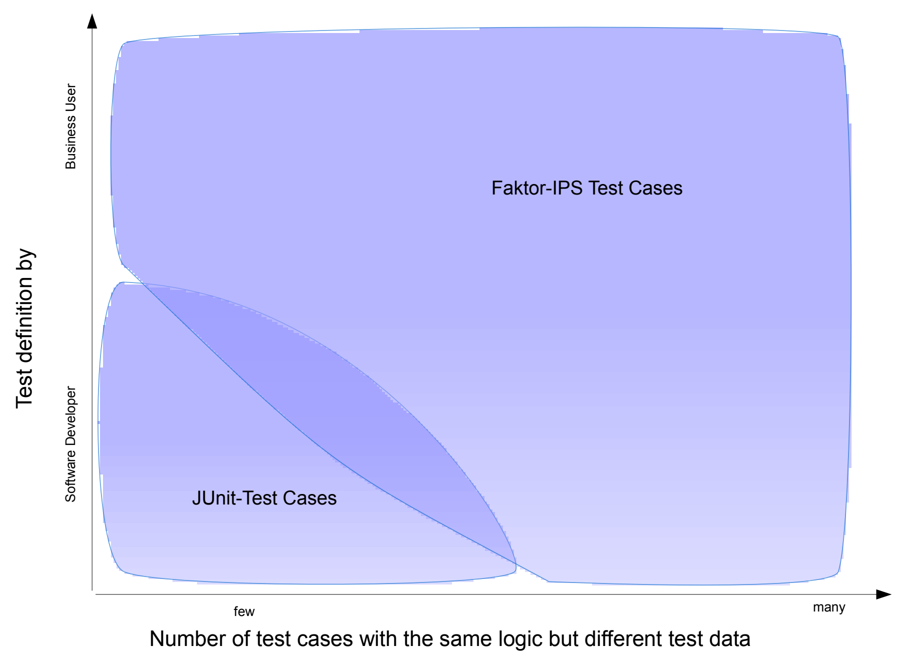


Figure 30: Decision criteria for test tool

The test support in Faktor-IPS is based on the separation between test case types and the test cases themselves. Test case types are defined by the application developer. As is the case with the business object model, a model-driven approach ist used. This approach consists in modeling the structure of the test data first and then generating the source code to read the test data of specific test cases. The only thing that remains for the developer to do, is to call the functions he wants to have tested and to match the actual results against the expected results.

Test cases with concrete test data can be created on the basis of a test case type. To do this, a specific Test Case Editor is provided. You can execute test cases in the Testrunner and display any differences that might show up. What's more, you can run one test case, multiple test cases, or all test cases in a project.

Both the Test Case Editor and Testrunner are integrated into the product definition perspective of Faktor-IPS and can easily be used by business users. This way, business users can take advantage of a unified user interface for defining products and tests. It can be used to define and test new products independently of the operational systems.