

Faktor-IPS Tutorial

Table of Contents

Teil 1: Modellierung und Produktkonfiguration	3
Einleitung	3
Hello Faktor-IPS	4
Arbeiten mit Modell und Sourcecode	10
Erweiterung des Hausratmodells	23
Aufnahme von Produktaspekten ins Modell	26
Definition der Hausratprodukte	36
Zugriff auf Produktinformationen zur Laufzeit	44
Part 1: Modeling and Product Configuration	48
Introduction	48
Hello Faktor-IPS	49
Working with the Model and Source Code	54
Extending the Home Contents Model	66
Adding Product Aspects to the Model	68
Defining the Products	78
Runtime Access to Product Information	85
Teil 2: Verwendung von Tabellen und Formeln	89
Überblick	89
Verwendung von Tabellen	89
Tarifzonentabelle	89
Beitragstabelle	94
Implementieren der Beitragsberechnung	97
Beitragsberechnung für die Hausrat-Deckungen	100
Verwendung von Formeln	102
Beitragsberechnung für die Zusatzdeckungen	106
Part 2: Using Tables and Formulas	113
Overview	113
Using Tables	113
Rating District Table	113
Rate Table	117
Implementing Premium Computation	120
Premium Computation for Coverages	123
Using Formulas	124
Computing the Premiums for Extra Coverages	128
Teil 3: Testen mit Faktor-IPS	134
Überblick	134

Konzeptionelle Grundlagen	134
Testen mit Faktor-IPS am Beispiel Hausratversicherung	136
Testfalltyp für Beitragsberechnung Hausrat	136
Testfall anlegen	142
Sonderfall: Testen von abgeleiteten Attributen	148
Testfälle durch Kopieren erzeugen	151
JUnit-Adapter für Faktor-IPS Testfälle	155
Konfiguration des Maven Surefire Plugin für JUnit 5 Tests	156
Zusammenfassung	157
Part 3: Testing with Faktor-IPS	159
Overview	159
Conceptual Foundation	159
Testing with Faktor-IPS using Home Insurance as an Example	160
A Test Case Type for the Premium Computation	161
Creating a Test Case	166
Special case: Testing Derived Attributes	172
Creating Test Cases by Copying them	175
A JUnit-Adapter for Faktor-IPS Test Cases	178
Konfiguration of the Maven Surefire Plugin for JUnit 5 tests	180
Summary	180
FAQ	182
Teil 1: Modellierung und Produktkonfiguration	182
Teil 2: Verwendung von Tabellen und Formeln	185
Teil 3: Testen mit Faktor-IPS	185
FAQ	187
Part 1: Modeling and Product Configuration	187
Part 2: Using Tables and Formulas	190
Part 3: Testing with Faktor-IPS	190

Teil 1: Modellierung und Produktkonfiguration

Einleitung

Dieses Tutorial führt in die Konzepte von und die Arbeitsweise mit Faktor-IPS ein. Faktor-IPS ist ein OpenSource-Werkzeug zur modellgetriebenen Entwicklung [1] versicherungsfachlicher Softwaresysteme mit Fokus auf der einheitlichen Abbildung des Produktwissens. Insbesondere können mit Faktor-IPS nicht nur die Modelle der Systeme bearbeitet, sondern auch die Produktinformationen selbst verwaltet werden.

Neben reinen Produktdaten können einzelne Produktdaspekte auch über eine Excel ähnliche Formelsprache definiert werden. Darüber hinaus können Tabellen verwaltet und fachliche Testfälle definiert und ausgeführt werden.

Als durchgängiges Beispiel verwenden wir dazu eine stark vereinfachte Hausratversicherung. Die grundlegenden Konstruktions- und Modellierungsprinzipien lassen sich auch anhand dieses sehr fachlichen Modells darstellen. Insbesondere behandeln wir in dem Tutorial die Möglichkeiten zur Produktkonfiguration.

Das Tutorial ist in zwei Teile gegliedert:

1. Der erste Teil führt in die Arbeit mit dem Modellierungswerkzeug und dem generierten Sourcecode ein und zeigt wie konkrete Produkte konfiguriert werden.
2. Der zweite Teil beschreibt die Verwendung von Tabellen, die Implementierung der Beitragsberechnung und zeigt, wie mit Hilfe von Formeln das Hausratmodell flexibel gestaltet werden kann.

Das Tutorial ist für Softwarearchitekten und Entwickler mit fundierten Kenntnissen über objektorientierte Modellierung mit der UML geschrieben. Erfahrungen mit der Entwicklung von Java-Anwendungen in Eclipse sind hilfreich, aber nicht unbedingt erforderlich.

Wenn Sie die einzelnen Schritte des Tutorials nicht selber durchführen wollen, können Sie das Endergebnis auch von doc.faktorzehn.org herunterladen und installieren.

Überblick über Teil 1 des Tutorials

Der erste Teil des Tutorials ist wie folgt gegliedert:

- **Hello Faktor-IPS**

In diesem Kapitel wird ein erstes Faktor-IPS Projekt angelegt und eine erste Klasse definiert.

- **Arbeiten mit Modell und Sourcecode**

Anhand eines Modells einer Hausratversicherung wird der Umgang mit dem Modellierungswerkzeug und dem generierten Sourcecode erläutert.

- **Erweitern des Hausratmodells**

Das Hausratsmodell wird vervollständigt indem der Klasse HausratVertrag weitere Attribute hinzugefügt werden.

- **Aufnahme von Produktaspekten ins Modell**

In diesem Kapitel wird erläutert, wie der Produktaspekt im Modell abgebildet wird.

- **Definition der Hausratprodukte**

Auf Basis des Modells werden nun zwei Hausratprodukte erfasst. Hierzu wird die Produktdefinitionsperspektive [2] verwendet, die speziell für die Fachabteilung konzipiert ist.

- **Zugriff auf Produktinformationen zur Laufzeit**

In dem Kapitel wird erläutert wie man zur Laufzeit, also in einer Anwendung oder einem Testfall auf Produktinformationen zugreift.

1 Model driven software development (MDSD). Eine sehr gute Beschreibung der zugrundeliegenden Konzepte findet sich in Stahl, Völter: Modellgetriebene Softwareentwicklung.

2 Die Produktdefinitionsperspektive ist eine spezielle Eclipse-Ansicht für Produktentwickler, welche nur Produkt-Projekte anzeigt und Modell-Tools aus der Toolbar ausblendet. Die Anzeige der Bausteine kann im Modell angepasst werden.

Hello Faktor-IPS

Im ersten Schritt dieses Tutorials legen wir ein Faktor-IPS Projekt an, definieren eine Modellklasse und generieren Java-Sourcecode zu dieser Modellklasse.

Falls Sie Faktor-IPS noch nicht installiert haben, tun Sie das jetzt. Die Software und die Installationsanleitung finden Sie auf <https://www.faktorzehn.org/de/download/>. In diesem Tutorial verwenden wir Eclipse 4.30 (2023-12) und Faktor-IPS 25.1.1.release. Eclipse wird auf Englisch verwendet. Faktor-IPS hingegen auf Deutsch (also mit installiertem Language Pack). Achten Sie darauf, auch die Faktor-IPS Add-ons (m2e und Groovy) zu installieren.

Starten Sie Eclipse. Am besten verwenden Sie für dieses Tutorial einen eigenen Workspace. Wenn Faktor-IPS korrekt installiert ist, sollten Sie bei geöffneter *Java-Perspektive* [3] in der Toolbar folgende Symbole sehen [4]:

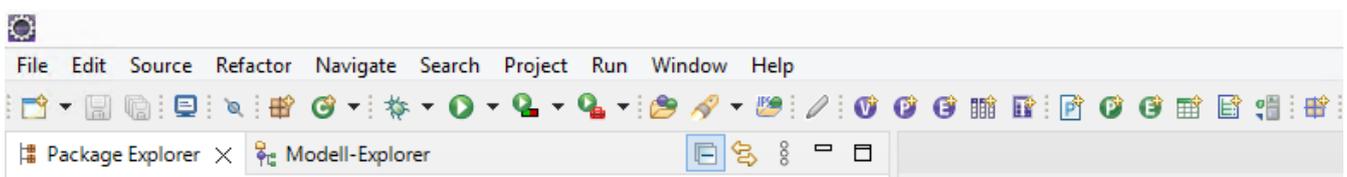


Figure 1. Faktor-IPS Icons

3 In der Menüleiste *Window* ► *Perspective* ► *Open Perspective* ► *Java* auswählen

4 In den Tutorials von Faktor-IPS wird von einer Installation mit Faktor-IPS German Language-Pack ausgegangen. Wenn Sie das Language-Pack installiert haben, aber trotzdem ohne die Übersetzung arbeiten wollen, können sie einfach Eclipse mit einer anderen Locale starten, z.B. mit `eclipse -vmargs -Duser.language=en`

Falls der *Modell-Explorer* nicht sichtbar ist, liegt das daran, dass Sie diesen Workspace bereits vor der Installation von Faktor-IPS verwendet haben. Rufen Sie in diesem Fall im Menü *Window* ► *Perspective* ► *Reset Perspective* auf.

Faktor-IPS-Projekte sind normale Java-Projekte oder Maven-Projekte mit einer zusätzlichen Faktor-IPS-Nature. Als erstes legen Sie also ein neues Maven-Projekt mit dem Namen `Hausratmodell` an.

Hierzu kann der Maven-Archetype für Faktor-IPS genutzt werden, der ein Maven-Projekt mit Faktor-IPS Nature erstellt.

Öffnen Sie dazu die Kommandozeile und navigieren Sie in den Ordner in dem Sie das Projekt anlegen möchten. Führen Sie dort den folgenden Befehl aus:

```
mvn archetype:generate -DarchetypeGroupId=org.faktorips
-DarchetypeArtifactId=faktorips-maven-archetype -DarchetypeVersion=25.1.1.release
-DgroupId=org.faktorips.tutorial -DartifactId=Hausratmodell -Dversion=1.0
-Dpackage=org.faktorips.tutorial.model -DjavaVersion=17 -DIPS-Language=de -DIPS
-IpsModelProject=true -DIPS-IpsProductDefinitionProject=false -DIPS-SourceFolder=model
-DIPS-RuntimeIdPrefix=hausrat. -DIPS-ConfigureIPSBUILD=true
```

Dieser Befehl hat mit Hilfe des Faktor-IPS Archetype das Maven-Projekt `org.faktorips.tutorial.model` erstellt. Dem Projekt wurde die Faktor-IPS Nature und damit die Laufzeitbibliotheken von Faktor-IPS hinzugefügt.

In diesem Projekt werden wir die Modellklassen anlegen. Das Sourceverzeichnis in dem die Modelldefinitionen abgelegt werden haben wir `org.faktorips.tutorial.model` genannt (durch den Parameter `-DIPS-SourceFolder=model`). Unterhalb dieses Verzeichnisses kann die Modellbeschreibung wie in Java durch Packages (Pakete) strukturiert werden. Faktor-IPS verwendet wie Java qualifizierte Namen zur Identifikation der Klassen des Modells.

Das Basispackage für die generierten Java-Klassen wurde `org.faktorips.tutorial.model` genannt (durch den Parameter `-Dpackage=org.faktorips.tutorial.model`). Als Runtime-ID-prefix haben wir `hausrat.` gewählt. Die Bedeutung des Runtime-ID-Prefixes wird im Kapitel „Definition der Produkte“ erläutert.

Die genauen Funktionen der verwendeten Parameter können in der Dokumentation des Archetype nachgelesen werden: [Dokumentation - Faktor-IPS Maven-Archetype](#).

Anschließend muss das neu erstellte Projekt in den Eclipse Workspace importiert werden. Rufen Sie dazu *File* ► *Import* ► *Maven* ► *Existing Maven Projects* auf. Im Dialog wählen Sie dann den Projektordner als Root Directory aus und importieren dann das Projekt mit einem Klick auf *Finish*.

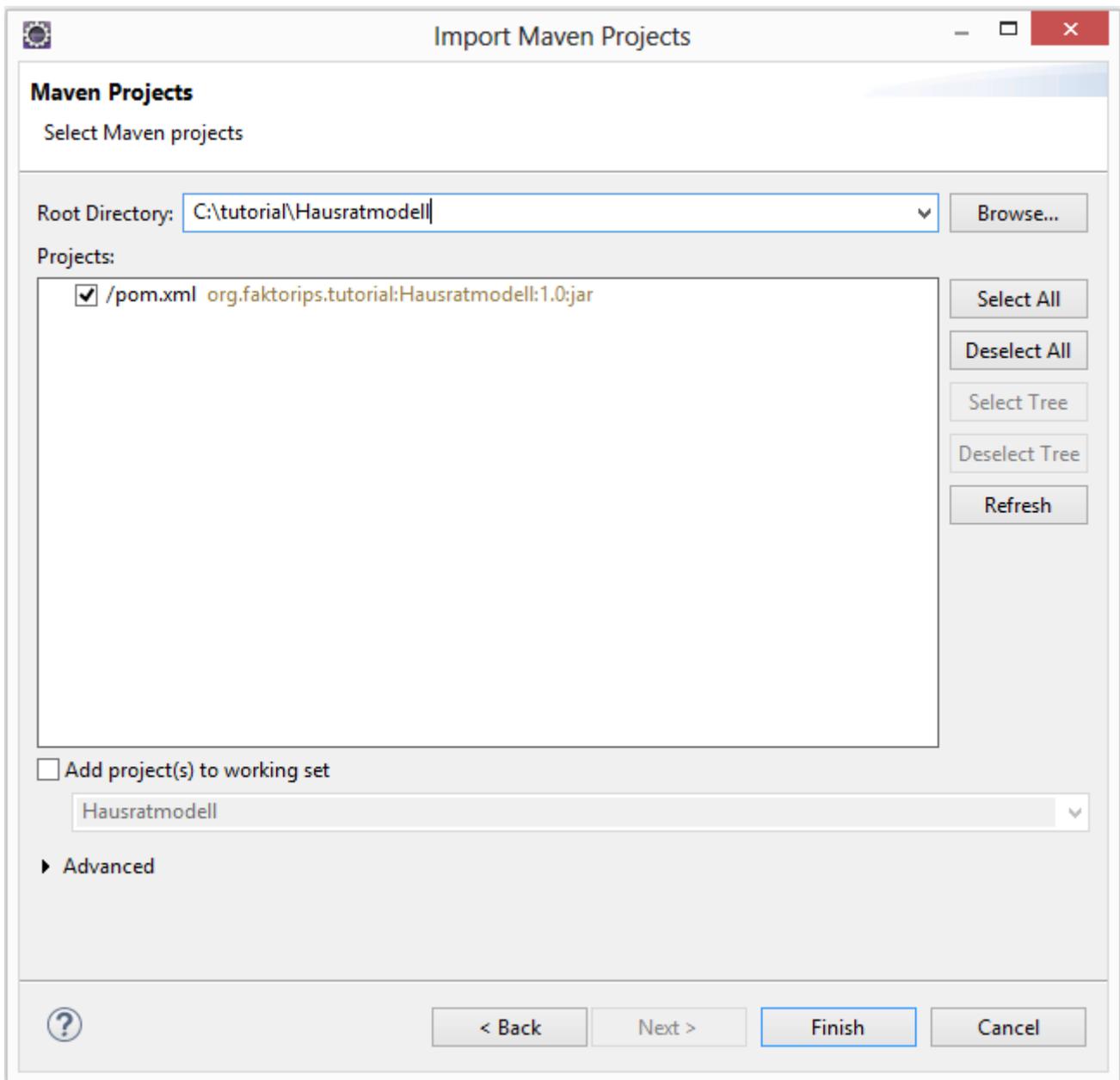


Figure 2. Maven Projekt in Eclipse importieren

Bevor wir die erste Klasse `HausratVertrag` definieren, stellen Sie noch ein, dass der Workspace automatisch gebaut wird (im Menü: *Project* ► *Build automatically*).

Wechseln Sie zunächst in den *Modell-Explorer* von Faktor-IPS direkt neben dem *Package-Explorer*.

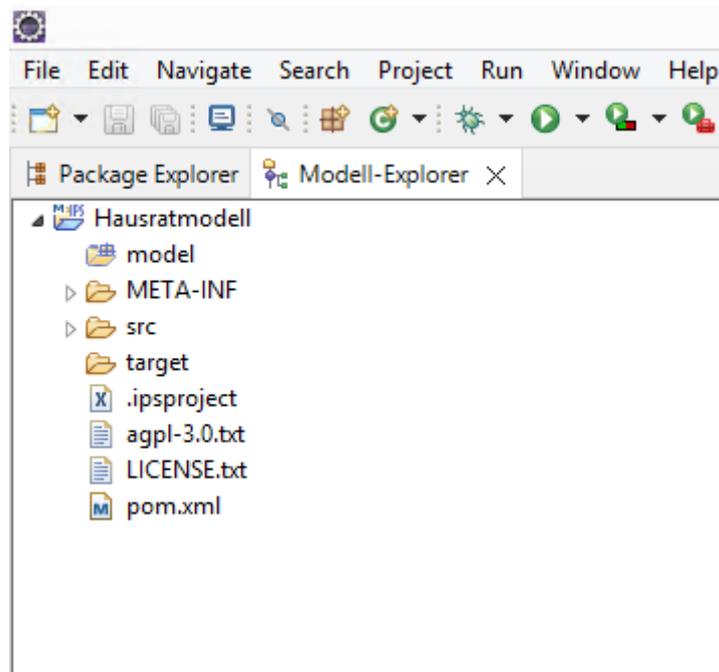


Figure 3. Ansicht der Projekte im Modell-Explorer

Im *Modell-Explorer* wird die Modelldefinition ohne die Java-Details dargestellt. In der Datei `.ipsproject` sind die Eigenschaften des Faktor-IPS Projektes gespeichert. Hierzu gehören zum Beispiel die gerade im *Add IpsNature*-Dialog eingegebenen Informationen, Einstellungen für die Codegenerierung, die erlaubten Datentypen etc. Der Inhalt ist in XML abgelegt und ausführlich in der Datei dokumentiert.

Die Klassen werden wir in einem Package mit dem Namen `hausrat` ablegen. Zum Anlegen des IPS Packages Rechtsklick auf das Sourceverzeichnis `model`, *neu* ► *IPS Package*. Danach den Namen des neuen Packages (`hausrat`) eingeben und anschließend auf Button Finish drücken. Alternativ können Sie IPS Packages über den Button , in der Toolbar, anlegen.

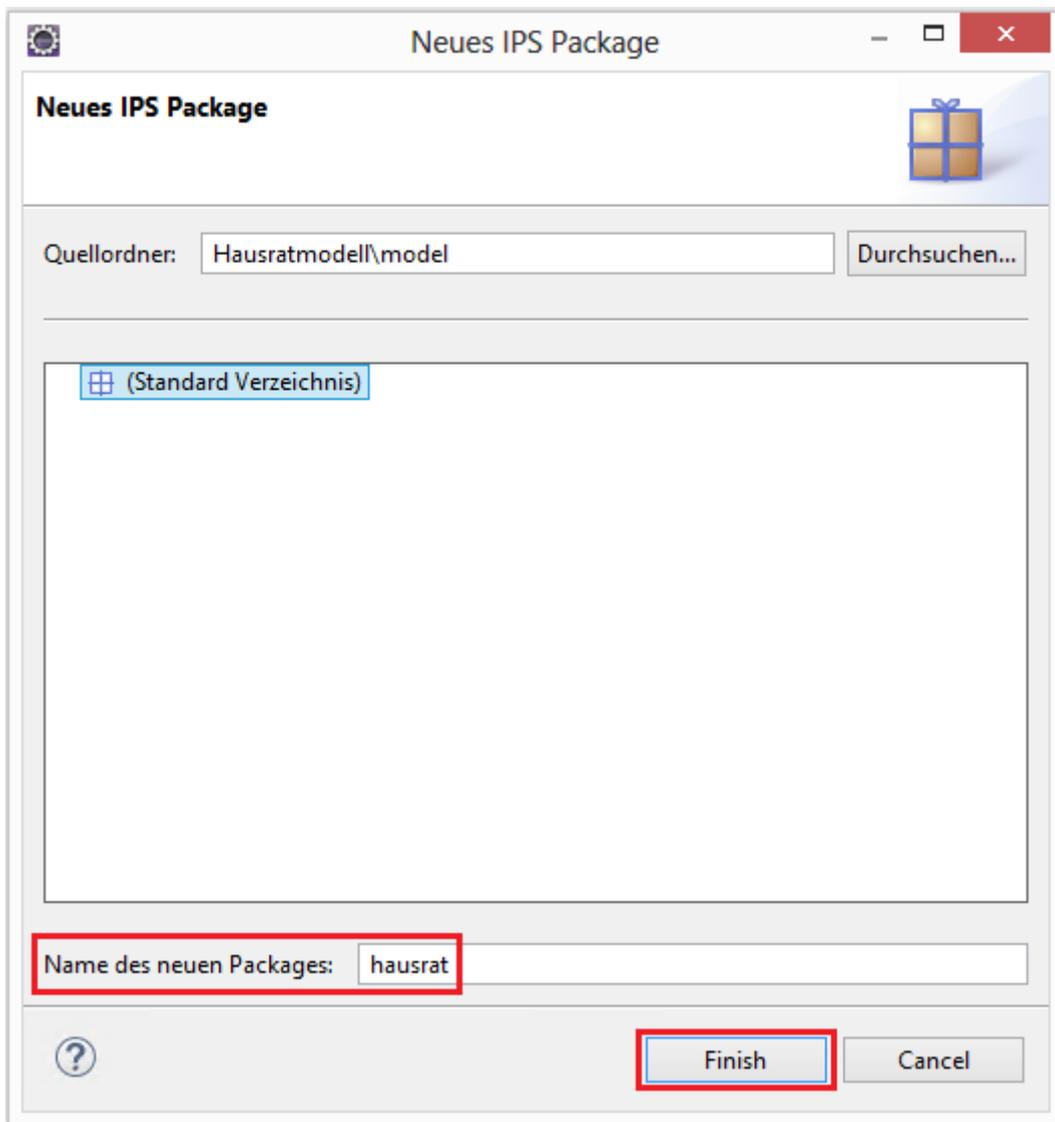


Figure 4. Anlegen eines IPS Packages

Als nächstes wollen wir eine Klasse anlegen, die unseren Hausratvertrag repräsentiert. Markieren Sie dazu das neu angelegte Package im Package Explorer und drücken auf den Button  in der Toolbar.

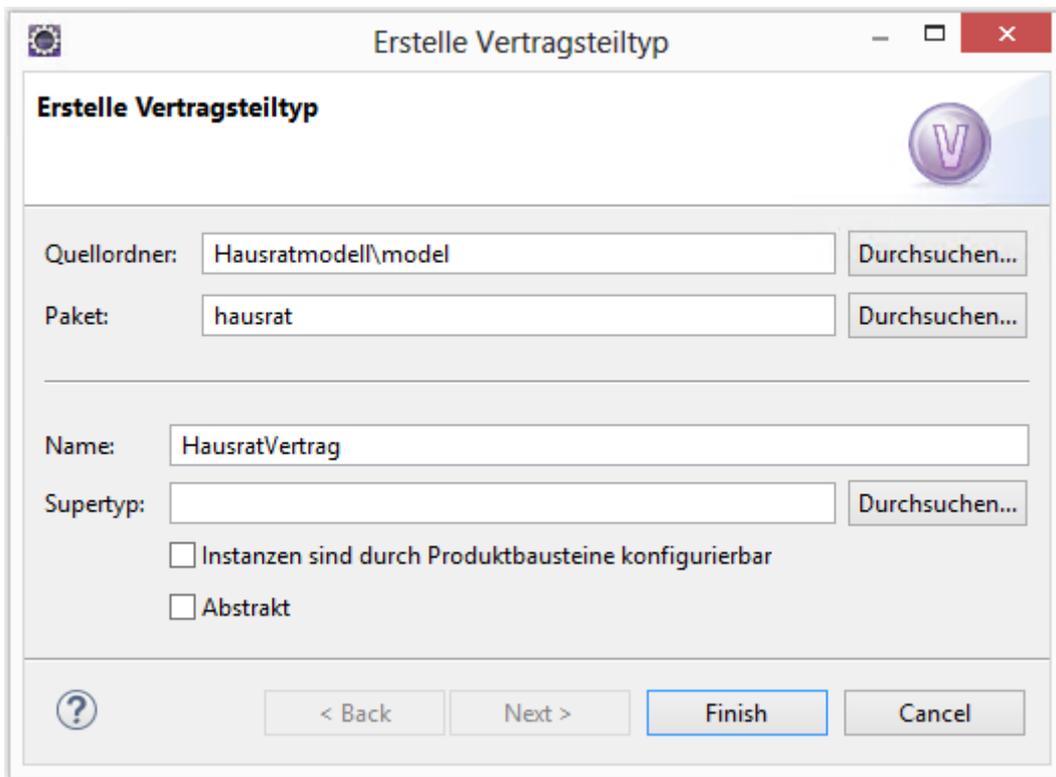


Figure 5. Anlegen einer neuen Vertragsklasse

In dem Dialog sind Sourceverzeichnis und Package bereits entsprechend vorbelegt und Sie geben noch den Namen der Klasse an, also `HausratVertrag` und klicken auf *Finish*. Faktor-IPS hat jetzt die neue Klasse angelegt und den Editor zur Bearbeitung geöffnet. Wechseln Sie zurück in den Package-Explorer. Sie sehen, dass die Klasse `HausratVertrag` in einer eigenen Datei mit dem Namen `HausratVertrag.ipspolicympttype` gespeichert ist.

Weiterhin hat der Codegenerator von Faktor-IPS bereits zwei Java-Sourcefiles erzeugt `org.faktorips.tutorial.model.hausrat.HausratVertrag` und `org.faktorips.tutorial.model.hausrat.HausratVertragBuilder`.

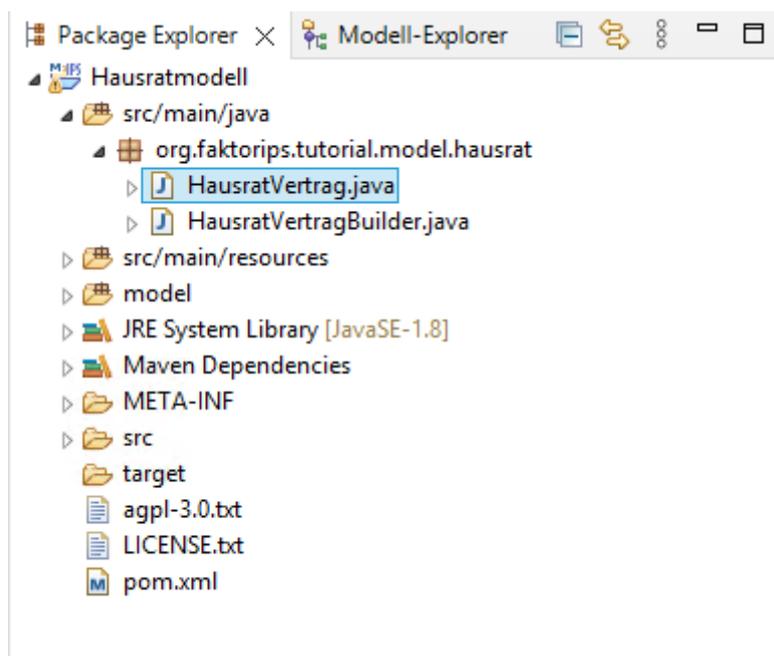


Figure 6. Generierte Klassen

Ein kurzer Blick in den Sourcecode von `HausratVertrag` zeigt, dass hier schon einige Methoden generiert worden sind. Diese Methoden dienen unter anderem zur Konvertierung der Objekte in XML und zur Unterstützung von Prüfungen.

Die Klasse `HausratVertragBuilder` erzeugt nach dem [Erbauer-Entwurfsmuster](#) Vertragsstrukturen.

Arbeiten mit Modell und Sourcecode

Im zweitem Schritt des Tutorials erweitern wir unser Modell und arbeiten mit dem generierten Sourcecode.

Als erstes erweitern wir die Klasse `HausratVertrag` um ein Attribut `zahlweise`. Wenn der Editor mit der Vertragsklasse nicht mehr geöffnet ist, öffnen Sie diesen nun durch Doppelklick im Modell-Explorer. In dem Editor klicken Sie auf den Button *Neu...* im Abschnitt *Attribute*. Es öffnet sich der folgende Dialog:

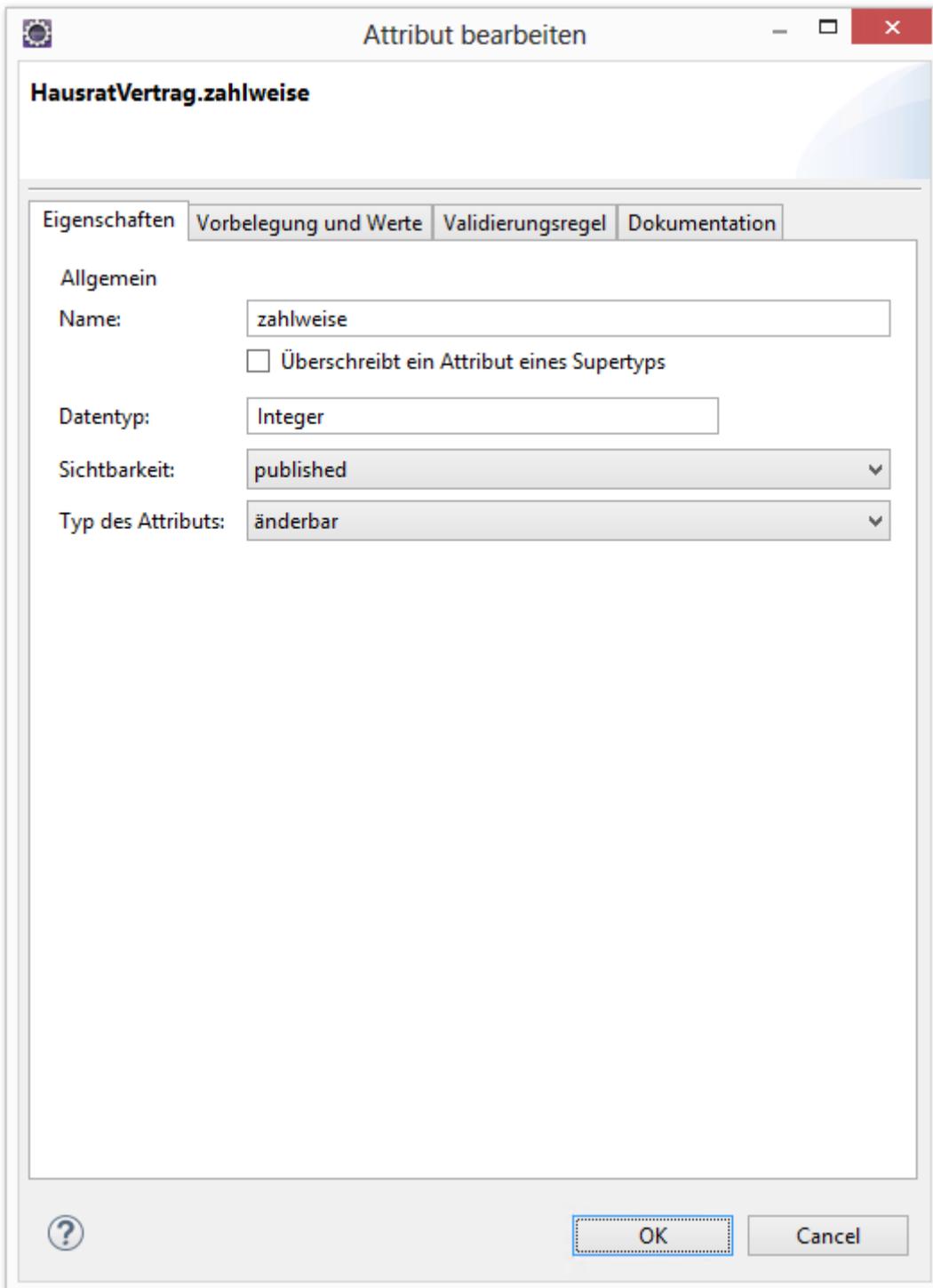


Figure 7. Dialog zum Anlegen eines neuen Attributs

Die Felder haben folgende Bedeutung:

Table 1. Attribut Felder

Feld	Bedeutung
Name	Der Name des Attributs
Checkbox	Indikator, ob dieses Attribut bereits in einer Superklasse definiert worden ist und in dieser Klasse lediglich Eigenschaften wie z.B. der Default Value überschrieben werden [5]

Feld	Bedeutung
Datatype/Datentyp	Datentyp des Attributs
Modifier/Sichtbarkeit	Analog zum Modifier in Java. Der zusätzliche Modifier <code>published</code> bedeutet, dass die Eigenschaften ins <code>published</code> Interface aufgenommen wird. [6]
Attribute type/Typ des Attributs	<p>Der Typ des Attributs.</p> <p>* änderbar Änderbare Eigenschaften, also solche mit Getter- und Setter- Methoden</p> <p>* konstant Konstante, nicht änderbare Eigenschaft</p> <p>* abgeleitet (cached, Berechnung durch expliziten Methodenaufruf) Abgeleitete Eigenschaften im UML Sinne. Die Eigenschaft wird durch einen expliziten Methodenaufruf berechnet und das Ergebnis ist danach über die Getter-Methode abfragbar. Zum Beispiel kann die Eigenschaft <code>bruttobeitrag</code> durch eine Methode <code>berechneBeitrag</code> berechnet und danach über <code>getBruttobeitrag()</code> abgerufen werden.</p> <p>* abgeleitet (Berechnung bei jedem Aufruf der Gettermethode) Abgeleitete Eigenschaften im UML Sinne. Die Eigenschaft wird bei jedem Aufruf der Getter-Methode berechnet. Zum Beispiel kann das Alter einer versicherten Person bei jedem Aufruf von <code>getAlter()</code> aus dem Geburtstag ermittelt werden.</p>

5 Entspricht der `@override` Annotation ab Java 5.

6 **Hinweis:** Die Aktivierung/Deaktivierung der Generierung der `Published-Interfaces` erfolgt über das Kontextmenü > Eigenschaften > Faktor-IPS Code Generator des entsprechenden Projekts

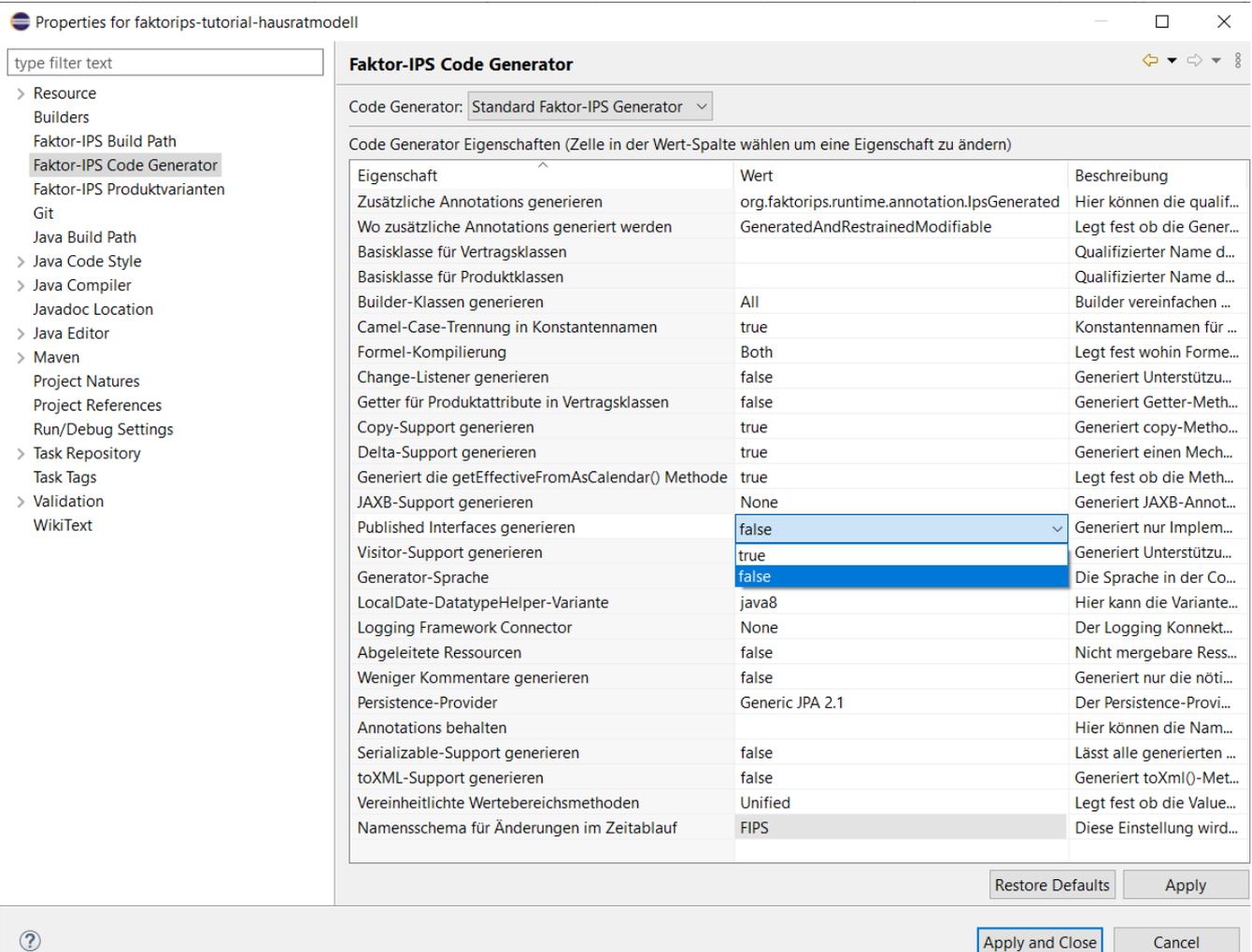


Figure 8. Aktivierung/Deaktivierung der Generierung der Published-Interfaces

Geben Sie als Namen **zahlweise** und als Datentyp *Integer* ein. Wenn Sie auf den Browse Button neben dem Feld klicken, öffnet sich eine Liste mit den verfügbaren Datentypen. Alternativ dazu können Sie wie in Eclipse üblich auch mit STRG-Space eine Vervollständigung durchführen. Wenn Sie zum Beispiel „D“ eingeben und STRG-Space drücken, sehen Sie alle Datentypen, die mit „D“ beginnen. Die anderen Felder lassen Sie wie vorgegeben und drücken jetzt *OK*, danach speichern Sie die geänderte Vertragsklasse.

Der Codegenerator hat nun bereits die Java-Sourcefiles aktualisiert. Die Klasse `␣HausratVertrag␣` enthält nun Zugriffsmethoden für das Attribut und speichert den Zustand in einer privaten Membervariable (Falls in Ihrer .ipsproject-Datei in der Einstellung "additionalAnnotations" `@IpsGenerated` inkludiert ist, wird diese vor generierten Faktorips-Methoden zusätzlich gesetzt).

```
/**
 * Diese Konstante enthaelt den Namen der Eigenschaft zahlweise.
 *
 * @generated
 */
public static final String PROPERTY_ZAHLWEISE = "zahlweise";
/**
 * Die maximal erlaubten Werte fuer die Eigenschaft zahlweise.
 *
 * @generated
```

```

    */
    @IpsAllowedValues("zahlweise")
    public static final ValueSet<Integer> MAX_ALLOWED_VALUES_FOR_ZAHLWEISE = new
UnrestrictedValueSet<>(true);
    /**
    * Der Vorgabewert des Attributs zahlweise.
    *
    * @generated
    */
    @IpsDefaultValue("zahlweise")
    public static final Integer DEFAULT_VALUE_FOR_ZAHLWEISE = null;
    /**
    * Membervariable fuer zahlweise.
    *
    * @generated
    */
    private Integer zahlweise = DEFAULT_VALUE_FOR_ZAHLWEISE;

    /**
    * Erzeugt eine neue Instanz von HausratVertrag.
    *
    * @generated
    */
    public HausratVertrag() {
        super();
    }

    /**
    * Gibt den erlaubten Wertebereich fuer das Attribut zahlweise zurueck.
    *
    * @generated
    */
    @IpsAllowedValues("zahlweise")
    public ValueSet<Integer> getAllowedValuesForZahlweise() {
        return MAX_ALLOWED_VALUES_FOR_ZAHLWEISE;
    }

    /**
    * Gibt den Wert des Attributs zahlweise zurueck.
    *
    * @generated
    */
    @IpsAttribute(name = "zahlweise", kind = AttributeKind.CHANGEABLE, valueSetKind =
ValueSetKind.AllValues)
    public Integer getZahlweise() {
        return zahlweise;
    }

    /**
    * Setzt den Wert des Attributs zahlweise.
    *

```

```

    * @generated
    */
    @IpsAttributeSetter("zahlweise")
    public void setZahlweise(Integer newValue) {
        this.zahlweise = newValue;
    }

```

Das JavaDoc für einige Methoden sind mit einem `@generated` markiert. Dies bedeutet, dass die Methode zu 100% generiert wird. Bei einer erneuten Generierung wird dieser Code genau so wieder erzeugt, unabhängig davon, ob er in der Datei gelöscht oder modifiziert worden ist. Änderungen seitens des Entwicklers werden also überschrieben. Möchten Sie die Methode modifizieren, so fügen Sie hinter die Annotation `@generated` ein `NOT` hinzu. Probieren wir das einmal aus. Fügen Sie jeweils die Zeile `System.out.println` (siehe folgende Codestelle) in die Getter- und Setter-Methode ein, und ergänzen bei der Methode `setZahlweise()` mit `NOT` hinter der Annotation:

```

/**
 * Gibt den Wert des Attributs zahlweise zurueck.
 *
 * @generated
 */
@IpsAttribute(name = "zahlweise", kind = AttributeKind.CHANGEABLE, valueSetKind =
ValueSetKind.AllValues)
public Integer getZahlweise() {
    System.out.println("getZahlweise"); ①
    return zahlweise;
}

```

```

/**
 * Setzt den Wert des Attributs zahlweise.
 *
 * @generated NOT
 */
@IpsAttributeSetter("zahlweise")
public void setZahlweise(Integer newValue) {
    System.out.println("setZahlweise"); ①
    this.zahlweise = newValue;
}

```

① Handgeschriebener Code

Generieren Sie jetzt den Sourcecode für die Klasse `HausratVertrag` neu. Dies können Sie wie in Eclipse üblich auf zwei Arten erreichen:

- Entweder bauen Sie mit *Project* ► *Clean* das gesamt Projekt neu, oder
- Sie speichern die Modellbeschreibung der Klasse `HausratVertrag` erneut.

Nach dem Generieren ist das `System.out.println(...)`-Statement aus der Getter-Methode entfernt worden, in der Setter-Methode ist es erhalten geblieben.

Methoden und Attribute, die neu hinzugefügt werden, bleiben nach der Generierung erhalten. Auf diese Weise kann der Sourcecode beliebig erweitert werden.

Nun erweitern wir noch die Modelldefinition der Zahlweise um die erlaubten Werte. Öffnen Sie dazu den Dialog zum Bearbeiten des Attributes und wechseln auf die zweite Tabseite. Bisher sind alle Werte des Datentyps als zulässige Werte für das Attribute erlaubt. Wir schränken dies nun auf 1, 2, 4, 12 für jährlich, halbjährlich, quartalsweise und monatlich ein. Ändern Sie hierzu den Typ auf „Aufzählung“ und geben Sie in die Tabelle die Werte 1, 2, 4 und 12 ein [7].

7 Faktor-IPS unterstützt auch die Definition von Enums. Auf dieses Feature wird an dieser Stelle aber verzichtet. Darüber hinaus können über einen Extension Point beliebige Java Klassen als Datentyp registriert werden. Diese Java-Klassen sollten dabei entsprechend des Musters *ValueObject* realisiert sein.

Setzen Sie nun einmal den Default Value auf 0. Faktor-IPS markiert den Default Value mit einer Warnung, da der Wert nicht in der im Modell erlaubten Wertemenge enthalten ist.

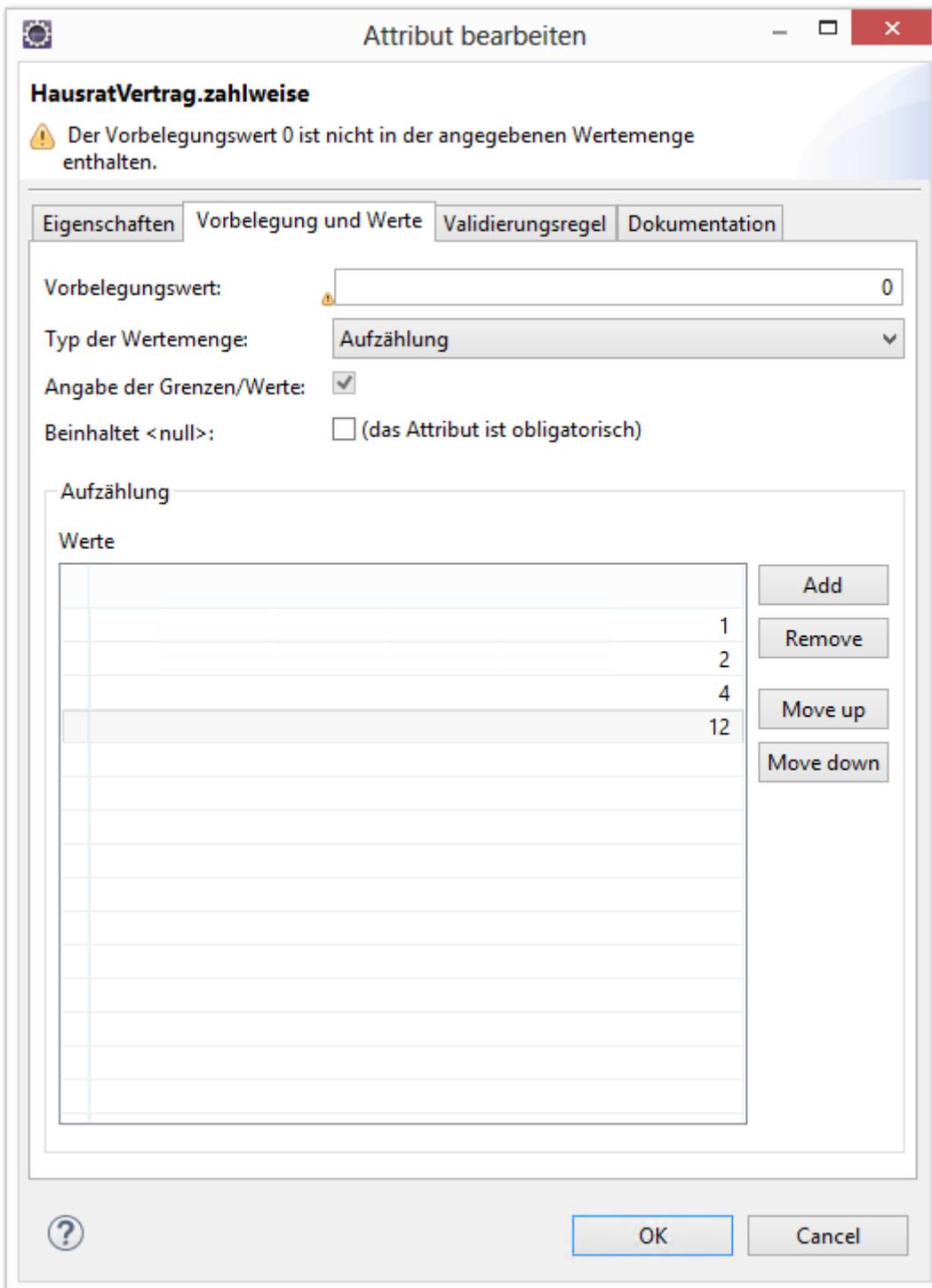


Figure 9. Wertebereich für das Attribut *zahlweise*

Es handelt sich also um einen möglichen Fehler im Modell. Lassen wir das aber für einen Augenblick so stehen. Das gibt uns die Gelegenheit die Fehlerbehandlung von Faktor-IPS zu erläutern. Schließen Sie dazu den Dialog und speichern die Vertragsklasse. Im *Problems-View* von Eclipse wird nun die gleiche Warnung wie im Dialog angezeigt. Faktor-IPS lässt Fehler und Inkonsistenzen im Modell zu und informiert den Benutzer darüber im Eclipse-Stil, also in den Editoren und als so genannte Problemmarker, die im *Problems-View* und in den Explorern sichtbar sind.

Description	Resource	Path	Location	Type
0 errors, 1 warning, 0 others				
Warnings (1 item)				
Der Vorbelegungswert 0 ist nicht in der angege...	HausratVertra...	/Hausratmodell/m...	Unknown	Faktor-IPS Pr...

Figure 10. Anzeige von Fehlern im Problems-View

Löschen Sie die „0“ als Vorbelegungswert und speichern Sie die Vertragsklasse. Die Warnung wird damit wieder aus dem Problems-View entfernt.

Faktor-IPS generiert eine Warnung und keinen Fehler, da es durchaus sinnvoll sein kann, wenn der Defaultwert nicht im Wertebereich ist. Insbesondere gilt dies für den Defaultwert null. Wird zum Beispiel ein neuer Vertrag angelegt, so kann es gewolltes Verhalten sein, dass die Zahlweise nicht vorbelegt ist sondern noch null ist, um die Eingabe der Zahlweise durch den Benutzer zu erzwingen. Erst wenn der Vertrag vollständig erfasst ist, muss auch die Bedingung erfüllt sein, dass die Eigenschaft Zahlweise einen Wert aus dem Wertebereich enthält.

Das Kapitel schließen wir mit der Definition einer Klasse `HausratGrunddeckung` und der Kompositionsbeziehung zwischen `HausratVertrag` und `HausratGrunddeckung` gemäß dem folgenden Diagramm:

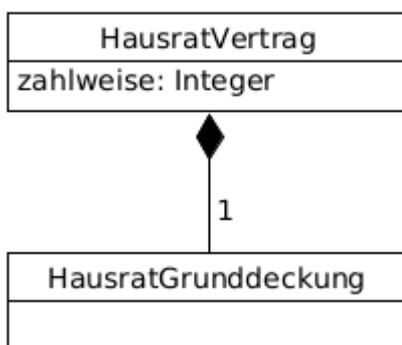


Figure 11. Modell der Vertragsseite

Legen Sie zunächst die Klasse `HausratGrunddeckung` analog zur Klasse `HausratVertrag` an. Wechseln Sie anschließend zu der Klasse `HausratVertrag`. Starten Sie den Assistenten zur Anlage einer neuen Beziehung. Klicken Sie auf den Button *Neu...* im Abschnitt *Beziehungen*. [8]

8 In Faktor-IPS wird in Übereinstimmung mit der UML-Begriff Association verwendet. In dem Tutorial verwenden wir den im Sprachgebrauch üblicheren Begriff Beziehung.

Figure 12. Anlegen einer neuen Beziehung

Als *Target* wählen Sie bitte die gerade angelegte Klasse „HausratGrunddeckung“ aus. Hier steht Ihnen wieder die Vervollständigung mit STRG-Space zur Verfügung. Bereich *Überschreiben / Abgeleitete Vereinigung* ignorieren Sie zunächst. Das Konzept wird im Tutorial zur Modellpartitionierung erläutert. Danach Button Next drücken.

Auf der nächsten Seite geben Sie als *Minimale Kardinalität* 1 und als *Maximale Kardinalität* 1 ein, als *Rollennamen* `HausratGrunddeckung` bzw. `HausratGrunddeckungen`. Die Mehrzahl wird verwandt, damit der Codegenerator verständlichen Sourcecode generieren kann.

Figure 13. Rollennamen und Kardinalitäten einer Beziehung

Auf der nächsten Seite können Sie auswählen, ob es auch eine Rückwärtsbeziehung (*Inverse Beziehung*) von `□HausratGrunddeckung□` zu `□HausratVertrag□` geben soll. Beziehungen in Faktor-IPS sind immer gerichtet, so ist es auch möglich die Navigation nur in eine Richtung zuzulassen. Hier wählen Sie bitte *Neue inverse Beziehung* und gehen zur nächsten Seite.

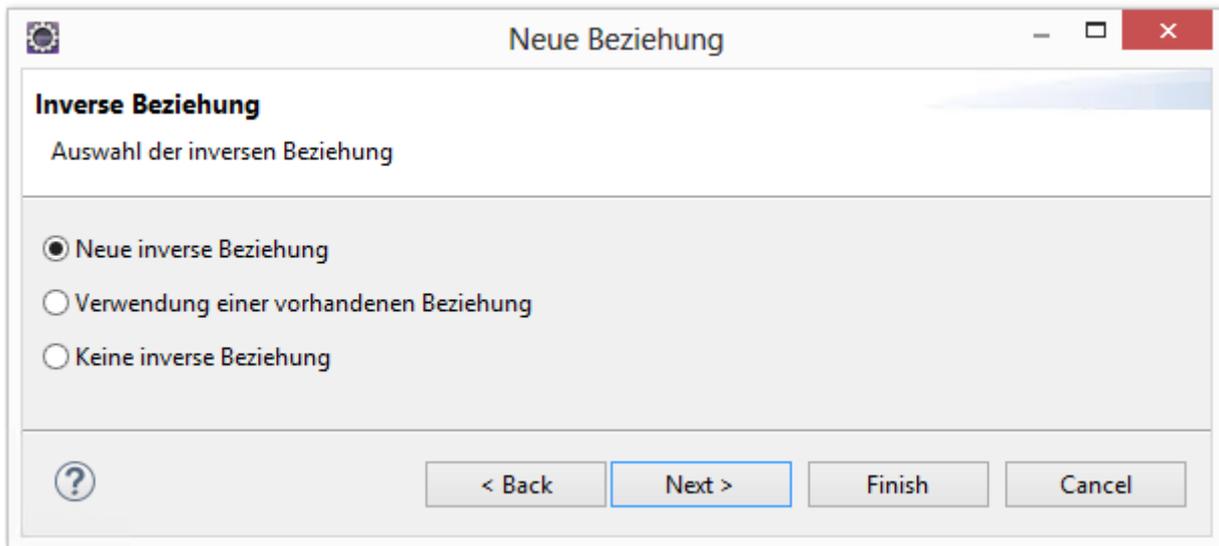


Figure 14. Neue inverse Beziehungen anlegen

Im nächsten Fenster geben Sie die Rollenbezeichnung der inversen Beziehung ein.

Figure 15. Eigenschaften der inversen Beziehung festlegen

Mit *Finish* legen Sie die beiden Beziehungen (vorwärts und rückwärts) an und speichern danach noch die Klasse `HausratVertrag`. Wenn Sie sich jetzt die Klasse `HausratGrunddeckung` ansehen, ist dort die Rückwärtsbeziehung eingetragen.

Zum Abschluss werfen wir noch einen kurzen Blick in den generierten Sourcecode. In die Klasse `HausratVertrag` wurden Methoden generiert, die Grunddeckung dem `HausratVertrag` hinzuzufügen. In der Klasse `HausratGrunddeckung` gibt es eine Methode, um zum `HausratVertrag` zu navigieren. Wenn sowohl die Vorwärts- als auch die Rückwärtsbeziehung im Modell definiert ist, werden hierbei beide Richtungen berücksichtigt. Das heißt nach dem Aufruf von `setHausratGrunddeckung(HausratGrunddeckung d)` auf einer Instanz `v` von `HausratVertrag` liefert `d.getHausratVertrag()` wieder `v` zurück. Dies zeigt ein kurzer Blick in die Implementierung der

Methode `setHausratGrunddeckung()` in der Klasse `HausratVertrag`:

```
@IpsAssociationAdder(association = "HausratGrunddeckung")
public void setHausratGrunddeckung(HausratGrunddeckung newObject) {
    if (hausratGrunddeckung != null) {
        hausratGrunddeckung.setHausratVertragInternal(null);
    }
    if (newObject != null) {
        newObject.setHausratVertragInternal(this);
    }
    hausratGrunddeckung = newObject;
}
```

Der Vertrag wird in der Deckung als der Vertrag gesetzt, zu dem die Deckung gehört (zweites `if`-Statement der Methode).

Erweiterung des Hausratmodells

In diesem Abschnitt werden wir unser Hausratmodell vervollständigen. Die folgende Abbildung zeigt das Modell.

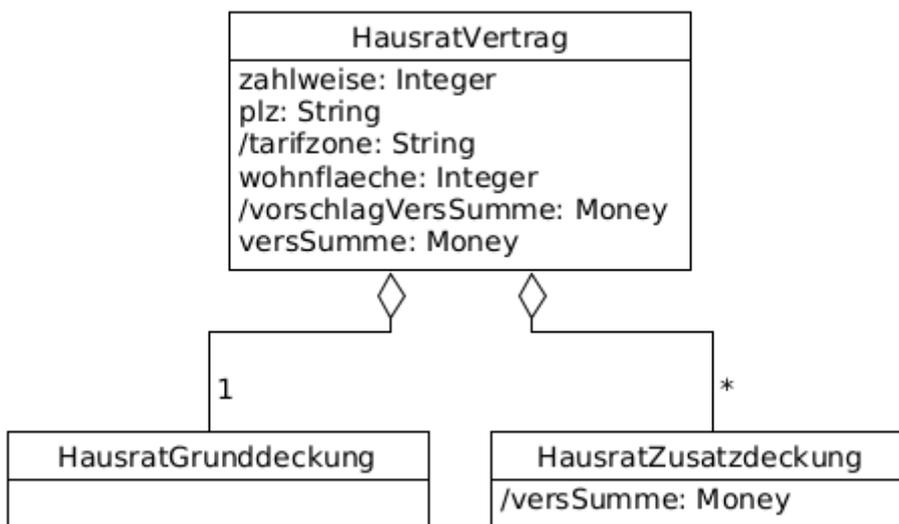


Figure 16. Hausratmodell mit Grunddeckung und Zusatzdeckung

Jeder Hausratvertrag hat genau eine Grunddeckung und kann beliebig viele Zusatzdeckungen haben. Die Grunddeckung deckt immer die im Vertrag definierte Versicherungssumme. Darüber hinaus kann ein Hausratvertrag optional Zusatzdeckungen enthalten, typischerweise sind dies Deckungen gegen Risiken wie zum Beispiel Fahrraddiebstahl oder Überspannungschäden. Die Zusatzdeckung werden wir im zweiten Teil des Tutorials näher betrachten.

Wir öffnen die Klasse `HausratVertrag` und definieren die Attribute der Klasse (analog `zahlweise`):

Table 2. Attribut Felder der Klasse `HausratVertrag`

Name : Datentyp	Beschreibung, Bemerkung
plz: <i>String</i>	Postleitzahl des versicherten Hausrats
/tarifzone : <i>String</i>	Die Tarifzone (I, II, III, IV, V oder VI) ergibt sich aus der Postleitzahl und ist maßgeblich für den zu zahlenden Beitrag. → Achten Sie also bei der Eingabe darauf <i>Typ des Attributs auf abgeleitet (Berechnung bei jedem Aufruf der Gettermethode)</i> zu setzen!
wohnflaeche : <i>Integer</i>	Die Wohnfläche des versicherten Hausrats in Quadratmetern. Der erlaubte Wertebereich ist min=0 und unbeschränkt. Den Wertebereich definieren Sie auf der zweiten Seite des Dialogs „Vorbelegung und Werte“. In der Auswahlbox „Typ der Wertemenge“ wählen Sie Bereich aus. Für „Minimum“ geben Sie eine 0 ein, Felder „Maximum“ und „Schrittweite“ lassen Sie leer.
/vorschlagVersSumme : <i>Money</i>	Vorschlag für die Versicherungssumme. Wird auf Basis der Wohnfläche bestimmt. → Achten Sie bei der Eingabe darauf den <i>Typ des Attributs auf abgeleitet (Berechnung bei jedem Aufruf der Gettermethode)</i> zu setzen!
versSumme : <i>Money</i>	Die Versicherungssumme. Der erlaubte Wertebereich ist min=0 EUR und max lassen Sie leer.

Der Editor, der die Klasse `□HausratVertrag□` anzeigt, sieht wie folgt aus:

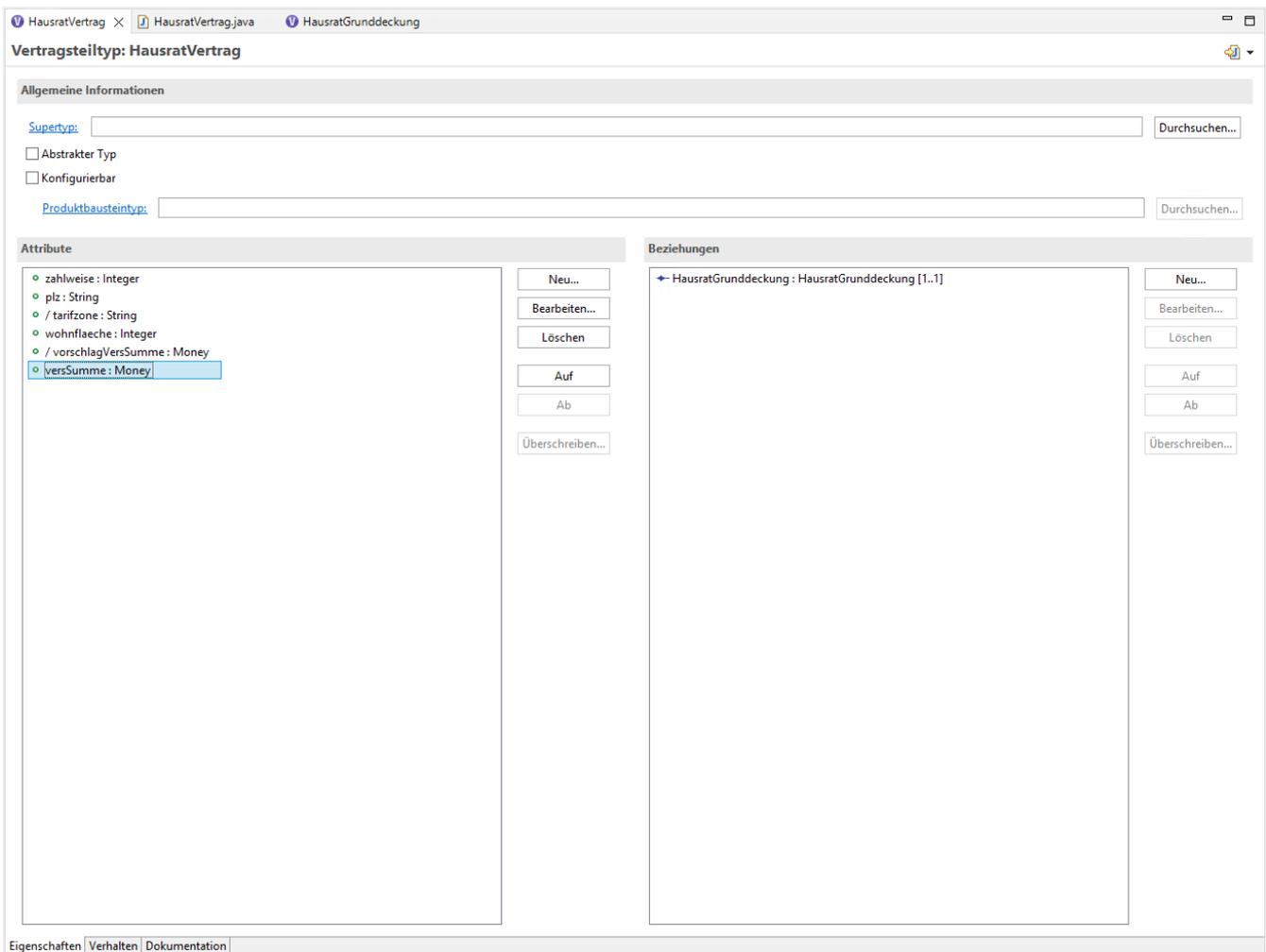


Figure 17. Klasse `HausratVertrag`

Die abgeleiteten Attribute werden UML-konform mit einem vorangestellten Schrägstrich angezeigt.

Öffnen Sie nun die Klasse `HausratVertrag` im Java-Editor und implementieren die Gettermethoden für die beiden abgeleiteten Attribute `tarifzone` und `vorschlagVersSumme` wie folgt:

```
/**
 * Gibt den Wert des Attributs tarifzone zurueck.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "tarifzone", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public String getTarifzone() {
    // begin-user-code
    // TODO: Wird spaeter anhand einer Tarifzontabelle ermittelt
    return "I";
    // end-user-code
}

/**
 * Gibt den Wert des Attributs vorschlagVersSumme zurueck.
 *
```

```

* @restrainedmodifiable
*/
@IpsAttribute(name = "vorschlagversSumme", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getVorschlagversSumme() {
    // begin-user-code
    // TODO: Der Multiplikator wird spaeter aus den Produktdaten ermittelt
    return Money.euro(650).multiply(wohnflaeche);
    // end-user-code
}

```

`@restrainedmodifiable` wird bei bestimmten Methoden (z.B. in generierten Testklassen oder Regeln) vom Generator anstelle von `@generated` erzeugt und weist darauf hin, dass der Entwickler eigenen Code hinzufügen kann. Der Abschnitt, in dem der eigene Code stehen darf, wird durch Kommentare gekennzeichnet. `@restrainedmodifiable` kann nur verwendet werden, wenn die Annotation vom Generator erzeugt wurde. Ein Ersetzen von `@generated` und Einfügen der entsprechenden Kommentarzeilen funktioniert nicht und wird vom Generator überschrieben.

Aufnahme von Produktaspekten ins Modell

Nun beschäftigen wir uns damit, wie Produktaspekte im Modell abgebildet werden. Bevor wir dies mit Faktor-IPS tun, diskutieren wir das Design auf Modellebene.

Schauen wir uns die bisher definierten Attribute unserer Klasse `HausratVertrag` an und überlegen, was in einem Produkt konfigurierbar sein soll:

Table 3. Eigenschaften der Klasse `HausratVertrag`

Eigenschaften von HausratVertrag	Konfigurationsmöglichkeiten
<code>zahlweise</code>	Die im Vertrag erlaubten Zahlweisen. Der Vorgabewert für die Zahlweise bei Erzeugung eines neuen Vertrags.
<code>wohnflaeche</code>	Bereich (min, max), in dem die Wohnfläche liegen muss.
<code>vorschlagVersSumme</code>	Vorgeschlagener Wert für einen Quadratmeter Wohnfläche. Der Vorschlag für die komplette Versicherungssumme ergibt sich dann durch Multiplikation mit der Wohnfläche [9].
<code>versSumme</code>	Bereich, in dem die Versicherungssumme liegen muss.

9 Alternativ könnten wir auch die Formel zur Berechnung des Vorschlags konfigurierbar machen. Zunächst beschränken wir uns aber auf den Vorschlag für den Quadratmeterwert.

Wir wollen zwei Hausratprodukte erstellen. `HR-Optimal` soll einen umfangreichen Versicherungsschutz gewähren während `HR-Kompakt` einen Basisschutz zu einem günstigen Beitrag

bietet. Die folgende Tabelle zeigt die Eigenschaften der beiden Produkte bzgl. der oben aufgeführten Konfigurationsmöglichkeiten:

Konfigurationsmöglichkeit	HR-Kompakt	HR-Optimal
Vorgabewert Zahlweise	jährlich	jährlich
Erlaubte Zahlweisen	halbjährlich, jährlich	monatlich, vierteljährlich, halbjährlich, jährlich
Erlaubte Wohnfläche	0-1000 qm	0-2000 qm
Vorschlag Versicherungssumme pro qm Wohnfläche	600 Euro	900 Euro
Versicherungssumme	10 Tsd - 2 Mio Euro	10 Tsd - 5 Mio Euro

Wir bilden dies im Modell ab, indem wir eine Klasse `HausratProdukt` einführen. Das Produkt enthält die Eigenschaften und Konfigurationsmöglichkeiten, die bei allen Hausratverträgen, die auf dem gleichen Produkt basieren, identisch sind. Die beiden Produkte `HR-Optimal` und `HR-Kompakt` sind Instanzen der Klasse „HausratProdukt“. Das folgende UML Diagramm zeigt das Modell:

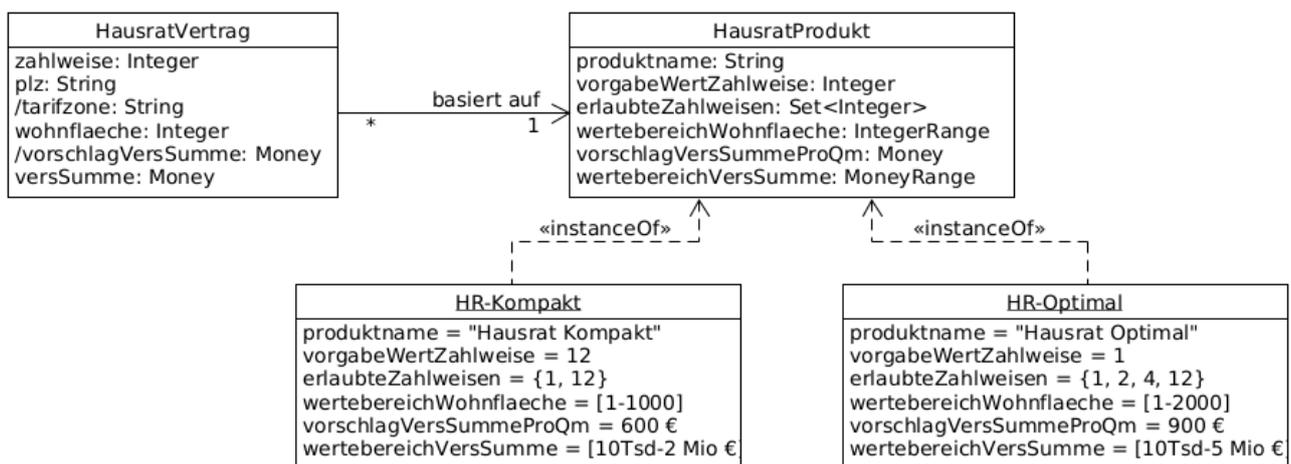


Figure 18. Hausratmodell mit Produktklassen

Erweitern wir unser Modell in Faktor-IPS um die Produktklassen. Als erstes definieren wir die Klasse `HausratProdukt`. Zum Erzeugen klicken Sie in der Toolbar auf den Button . In dem Wizard geben Sie den Namen der neuen Klasse an (`HausratProdukt`) und geben Sie im Feld *Policy Component Type (Vertragsteiltyp)* an, welche Klasse konfiguriert wird, in diesem Fall also `HausratVertrag`, danach den Button *Finish* drücken.

Es öffnet sich der Editor zur Bearbeitung von Produktklassen.

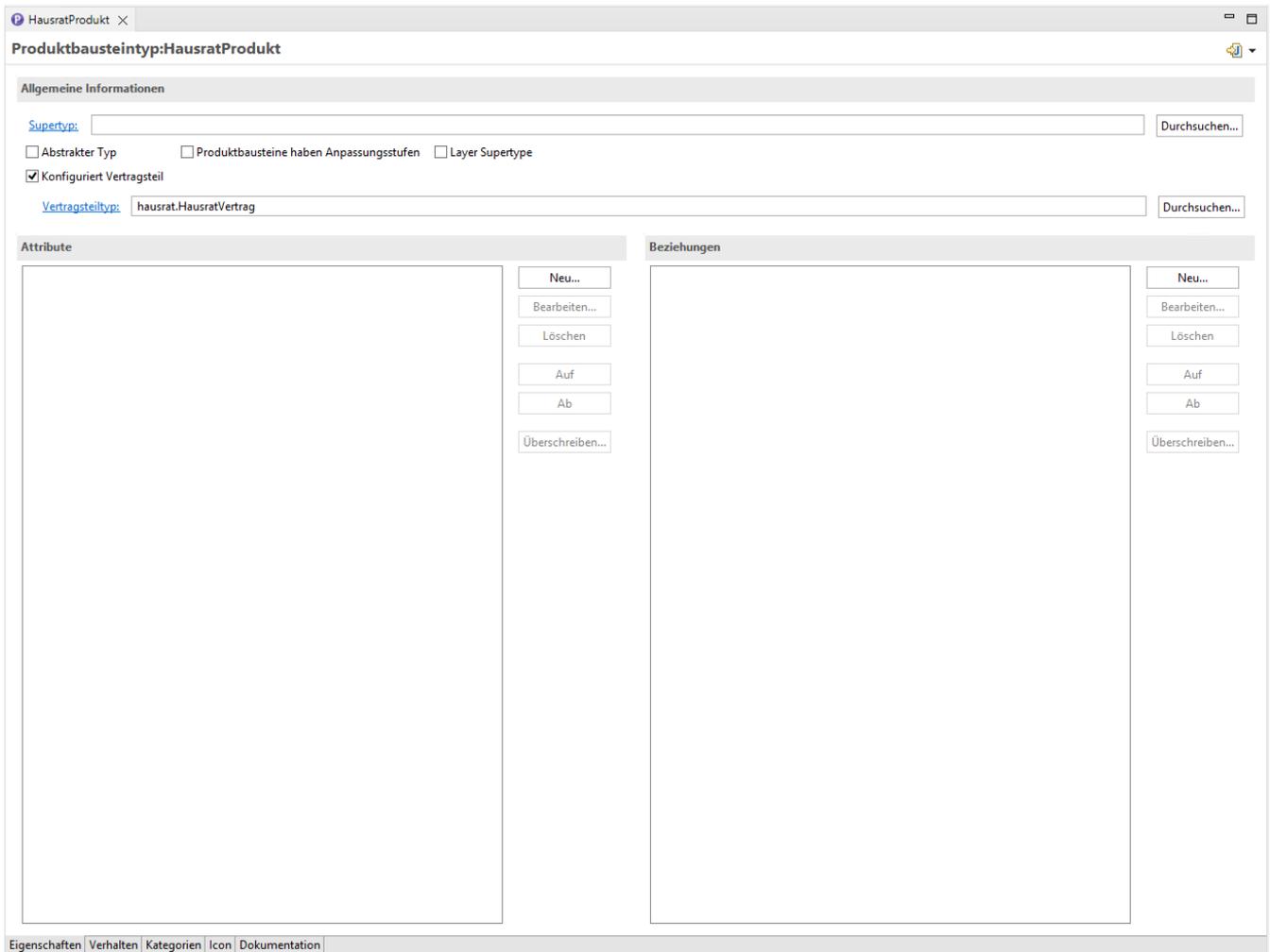


Figure 19. Editor für Produktklassen

Im Abschnitt *Allgemeine Informationen* sehen wir die gerade im Wizard eingegebene Information, dass die Klasse `HausratProdukt` die Klasse `HausratVertrag` konfiguriert. Ansonsten ist der Aufbau der ersten Seite des Editors analog zum Editor für Vertragsklassen [10].

10 Sie können in den Preferences einstellen, ob Sie alle Informationen zu einer Klasse auf einer Seite oder auf zwei Seiten dargestellt bekommen möchten.

Gleichzeitig hat Faktor-IPS nun die Implementierungsklasse `HausratProdukt` generiert.

Folgende Aspekte sollen in der Klasse `HausratProdukt` konfigurierbar sein:

- der Name des Produktes
- die erlaubten Zahlweisen und der Vorbelegungswert für die Zahlweise.

Beginnen wir mit dem Produktnamen. Hierzu legen Sie ein neues Attribut `produktname` vom Datentyp *String* an. Dies geschieht analog zum Anlegen eines Attributes für Vertragsklassen. Die folgende Abbildung zeigt den entsprechenden Dialog:

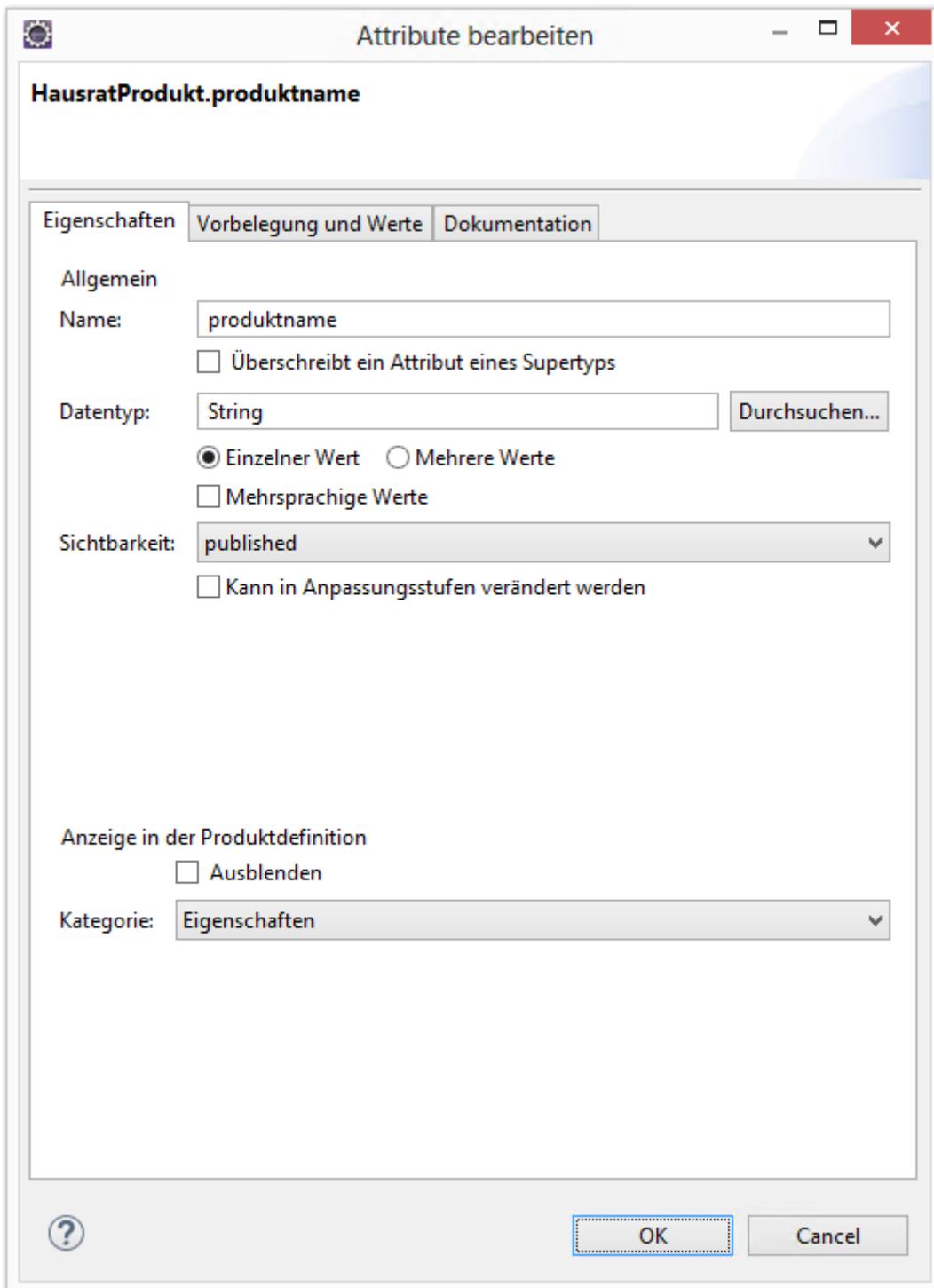


Figure 20. Dialog zum Editieren von Produktattributen

Nun konfigurieren wir die in einem Hausratvertrag erlaubten Zahlungsweisen und den Vorgabewert für die Zahlweise im Produkt. Hierzu öffnen wir zunächst den Editor für die Klasse `HausratVertrag`. In den Abschnitt *Allgemeine Informationen* hat der Wizard eingetragen, dass die Klasse `HausratVertrag` durch die Klasse `HausratProdukt` konfiguriert wird.

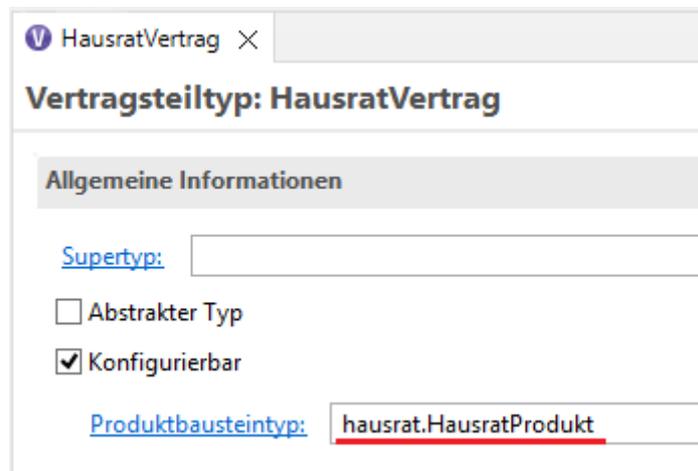


Figure 21. General Information Abschnitt im Editor für die Klasse Vertrag

Entsprechend ist der Vertragsteiltyp als konfigurierbar eingestellt. Öffnen Sie das Attribut `zahlweise` im Bereich *Attribute*. Im Dialogfenster unter dem Bereich *Konfiguration* können Sie dieses Attribut editieren.

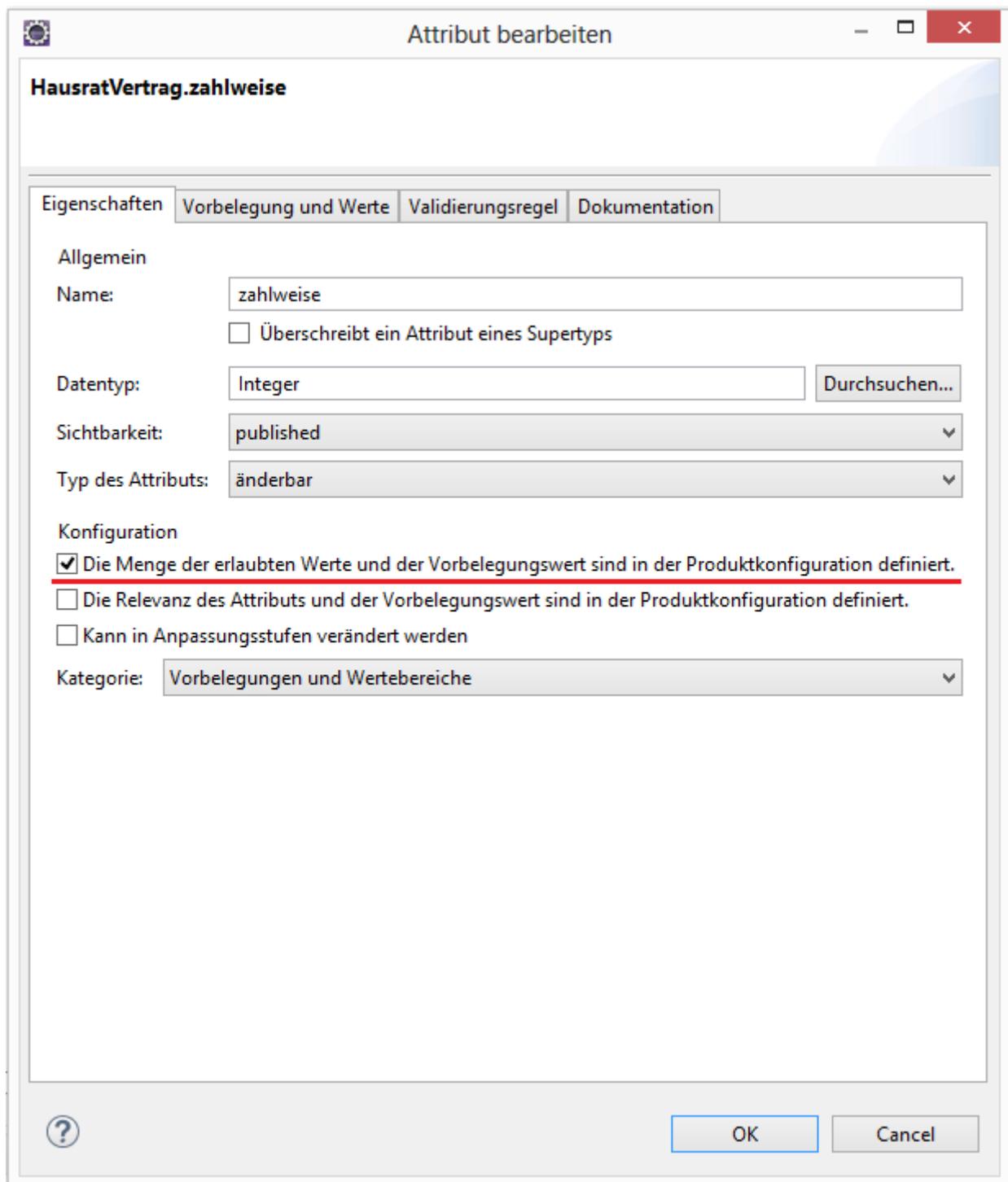


Figure 22. Dialog für ein Vertragsklassenattribut mit Konfigurationsmöglichkeit

Um die erlaubten Zahlweisen und den Vorgabewert für die Zahlweisen im Produkt definieren zu können, müssen Sie die entsprechende Checkbox anhaken. Button *OK* drücken und anschließend Änderungen speichern.

Werfen wir nun einen Blick in den Sourcecode. In der Klasse `HausratProdukt` gibt es jeweils eine Methode, um den Produktnamen, den Vorgabewert für die Zahlweise und die erlaubten Werte für die Zahlweise abzufragen.

```
/**
 * Gibt den Wert der Eigenschaft produktname zurueck.
 *
```

```

* @generated
*/
@IpsAttribute(name = "produktname", kind = AttributeKind.CONSTANT, valueSetKind =
ValueSetKind.AllValues)
@IpsGenerated
public String getProduktname() {
    return produktname;
}

/**
 * Gibt den Defaultwert fuer die Eigenschaft zahlweise zurueck.
 *
 * @generated
 */
@IpsDefaultValue("zahlweise")
@IpsGenerated
public Integer getDefaultValueZahlweise() {
    return defaultValueZahlweise;
}

/**
 * Gibt den erlaubten Wertebereich fuer das Attribut zahlweise zurueck.
 *
 * @generated
 */
@IpsAllowedValues("zahlweise")
@IpsGenerated
public OrderedValueSet<Integer> getAllowedValuesForZahlweise(IValidationContext
context) {
    return allowedValuesForZahlweise;
}

```

In der Klasse `HausratVertrag` gibt es Methoden, um auf das `HausratProdukt` zuzugreifen.

```

/**
 * Gibt HausratProdukt zurueck, welches HausratVertrag konfiguriert.
 *
 * @generated
 */
@IpsGenerated
public HausratProdukt getHausratProdukt() {
    return (HausratProdukt) getProductComponent();
}

/**
 * Setzt neuen HausratProdukt.
 *
 * @param hausratProdukt
 *         Der neue HausratProdukt.
 * @param initPropertiesWithConfiguredDefaults

```

```

*      <code>true</code> falls die Eigenschaften mit den Defaultwerten aus
*      HausratProdukt belegt werden sollen.
*
* @generated
*/
@IpsGenerated
public void setHausratProdukt(HausratProdukt hausratProdukt, boolean
initPropertiesWithConfiguratedDefaults) {
    setProductComponent(hausratProdukt);
    if (initPropertiesWithConfiguratedDefaults) {
        initialize();
    }
}
}

```

Markieren Sie abschließend noch die Attribute `wohnflaeche` und `versSumme` analog zur `zahlweise` als konfigurierbar.

Nun überarbeiten wir die Berechnung des Vorschlags der Versicherungssumme. Im Kapitel „Erweiterung des Hausratmodells“ hatten wir die Methode `getVorschlagVersSumme()` der Klasse `HausratVertrag` bisher wie folgt implementiert:

```

/**
 * Gibt den Wert des Attributs vorschlagversSumme zurueck.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "vorschlagversSumme", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getVorschlagversSumme() {
    // begin-user-code
    // TODO: Der Multiplikator wird spaeter aus den Produktdaten ermittelt.
    return Money.euro(650).multiply(wohnflaeche);
    // end-user-code
}

```

Nun wollen wir die Höhe des Multiplikators im Hausratprodukt konfigurieren können. Hierzu legen Sie zunächst an der Klasse `HausratProdukt` ein neues Attribut `vorschlagVersSummeProQm` vom Datentyp `Money` an. Dies ist der Vorschlagswert für einen Quadratmeter Wohnfläche. Nach dem Speichern der Klasse `HausratProdukt` hat Faktor -IPS an der Klasse `HausratProdukt` die entsprechende Gettermethode `getVorschlagVersSummeProQm()` generiert. Diese nutzen wir nun in der Berechnung des Vorschlags für die Versicherungssumme. Passen Sie den Sourcecode in der Klasse `HausratVertrag` dazu wie folgt an:

```

/**
 * Gibt den Wert des Attributs vorschlagversSumme zurueck.
 *
 * @restrainedmodifiable

```

```

*/
@IpsAttribute(name = "vorschlagversSumme", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getVorschlagversSumme() {
    // begin-user-code
    HausratProdukt prod = getHausratProdukt();
    if (prod == null) {
        return Money.NULL;
    }
    return prod.getVorschlagVersSummeProQm().multiply(wohnflaeche);
    // end-user-code
}

```

Definieren wir nun noch die Produktseite des Modells für die Grunddeckung. Dazu markieren wir die Klasse `HausratGrunddeckung` als Konfigurierbar. Die neu anzulegende Produktbausteinklasse nennen wir `HausratGrunddeckungstyp`. Wir definieren zunächst nur ein Attribut `bezeichnung` mit Datentyp `String` an dieser Klasse.

The screenshot shows a configuration window for 'HausratGrunddeckung'. Under the 'Allgemeine Informationen' section, the 'Konfigurierbar' checkbox is checked and highlighted with a red border. The 'Produktbaustein' field is filled with the value 'hausrat.HausratGrunddeckungstyp'.

Figure 23. Konfiguration der HausratGrunddeckung

Zum Abschluss dieses Kapitels beschäftigen wir uns noch mit den Beziehungen zwischen den Klassen der Produktseite. Wir wollen hierüber abbilden, welche (Hausrat-)Deckungstypen in welchen (Hausrat-)Produkten enthalten sind. Das folgende UML Diagramm zeigt das Modell:

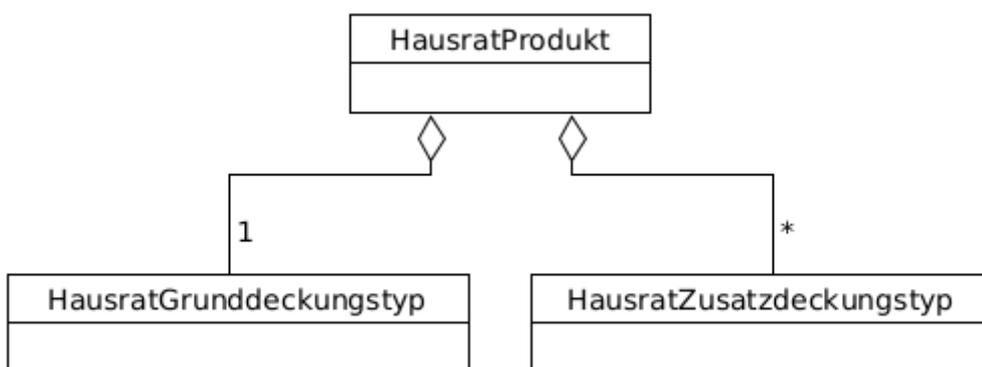


Figure 24. Modell der Produktkonfigurationsklassen

Das Hausratprodukt verwendet genau einen Grunddeckungstyp und beliebig viele Zusatzdeckungstypen. Andersherum kann ein Grunddeckungstyp bzw. ein Zusatzdeckungstyp in beliebig vielen Hausratprodukten verwendet werden. Die primäre Navigation ist immer vom Hausratprodukt zum Grund- bzw. Zusatzdeckungstyp, nicht umgekehrt, da ein Deckungstyp immer unabhängig von den Produkten sein sollte, die ihn verwenden.

Definieren wir die Beziehung zwischen `HausratProdukt` und `HausratGrunddeckungstyp` also nun in Faktor-IPS. Öffnen Sie hierzu zunächst den Editor für die Klasse `HausratProdukt` und legen im Bereich *Beziehungen* durch klicken auf *New* eine neue Beziehung an. Es öffnet sich der folgende Dialog, in den Sie die Daten wie hier abgebildet eintragen. Achten Sie darauf die maximale Kardinalität auf eins zu setzen. Den Zusatzdeckungstypen legen wir im Teil 2 des Tutorials an.

The dialog box 'Beziehung bearbeiten' is shown for the relationship between 'HausratGrunddeckungstyp' and 'HausratGrunddeckungstyp'. The 'Eigenschaften' tab is selected. The 'Ziel' field is set to 'hausrat.HausratGrunddeckungstyp'. The relationship type is 'Aggregation'. The 'Rolle (singular):' is 'HausratGrunddeckungstyp' and the 'Rolle (plural):' is 'HausratGrunddeckungstypen'. Both 'Min Kardinalität' and 'Max Kardinalität' are set to 1. There are checkboxes for 'Änderungen im Zeitablauf' (unchecked), 'Anzeige in der Produktdefinition' (unchecked), and 'Überschreiben / Abgeleitete Vereinigung' (all unchecked).

Figure 25. Dialog für Beziehungen zwischen Produktklassen

Definition der Hausratprodukte

In diesem Kapitel definieren wir nun die Produkte `HR-Optimal` und `HR-Kompakt` in Faktor-IPS. Hierzu wird die speziell für den Fachbereich entwickelte *Produktdefinitionsperspektive* verwendet.

Als erstes richten Sie ein neues Projekt mit dem Namen `Hausratprodukte` und dem Sourceverzeichnis `produkt Daten` ein. Dazu wieder mit Hilfe des Archetype über die Kommandozeile ein neues Maven-Projekt mit der Faktor-IPS Nature erzeugen.

```
mvn archetype:generate -DarchetypeGroupId=org.faktorips
-DarchetypeArtifactId=faktorips-maven-archetype -DarchetypeVersion=25.1.1.release
-DgroupId=org.faktorips.tutorial -DartifactId=Hausratprodukte -Dversion=1.0
-Dpackage=org.faktorips.tutorial.produkt Daten -DjavaVersion=17 -DIPS-Language=de -DIPS
-IpsModelProject=false -DIPS-IpsProductDefinitionProject=true -DIPS
-SourceFolder=produkt Daten -DIPS-RuntimeIdPrefix=hausrat. -DIPS-ConfigureIPSBUILD=true
```

Als Typ wählen Sie diesmal *Produktdefinitions-Project* ("`-DIPS-IsProductDefinitionProject=true`"), als Packagename `org.faktorips.tutorial.produkt Daten` und als Runtime-ID Prefix `hausrat.`. Achten Sie darauf, dass der Prefix mit einem Punkt (.) endet. Faktor-IPS erzeugt für jeden neuen Produktbaustein eine Id, mit der der Baustein zur Laufzeit identifiziert wird. Standardmäßig setzt sich diese Runtime-ID aus dem Prefix gefolgt von dem (unqualifizierten) Namen zusammen [11]. Der qualifizierte Name eines Bausteines wird nicht zur Identifikation zur Laufzeit verwendet, da die Packagestruktur zur Organisation der Produktdaten zur Entwicklungszeit dient. Auf diese Weise können die Produktdaten umstrukturiert (refactored) werden, ohne dass dies Auswirkungen auf die nutzenden operativen Systeme hat.

11 Für die Implementierung eigener Verfahren zur Vergabe der Runtime-ID wird ein entsprechender Extension Point bereitgestellt werden.

Das Projekt muss anschließend über *File* ► *Import* ► *Maven* ► *Existing Maven Projects* in den Eclipse Workspace importiert werden.

Faktor-IPS generiert Java Sourcefiles und kopiert XML-Dateien, die zu 100% generiert werden, in das Verzeichnis "`src/main/resources`". Der Inhalt des Verzeichnisses kann also jederzeit gelöscht und neu erzeugt werden. Da in diesem Ordner im Verlauf dieses Tutorials auch Source-Code für Formeln generiert wird, muss Maven angewiesen werden, diesen auch im "`src/main/resources`"-Ordner zu bauen. Dazu muss in der `pom.xml`-Datei unter `<plugins>` der folgende Code ergänzt werden:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>3.6.0</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
```

```

        <goal>add-source</goal>
    </goals>
    <configuration>
        <sources>
            <source>${project.basedir}/src/main/resources</source>
        </sources>
    </configuration>
</execution>
</executions>
</plugin>

```

Die Verwaltung der Produktdaten in einem eigenen Projekt erfolgt vor dem Hintergrund, dass die Verantwortung für die Produktdaten bei anderen Personen liegt und sie auch einen anderen Releasezyklus haben können. So könnte die Fachabteilung zum Beispiel ein neues Produkt `HR-Flexible` erstellen und freigeben, ohne dass das Modell geändert wird. Damit in dem neuen Projekt auf die Klassen des Hausratmodells zugegriffen werden kann, muss in Faktor-IPS im Produktdatenprojekt eine Referenz auf das Hausratmodell-Projekt definiert werden. Das funktioniert in Faktor-IPS analog zur Definition des Build Path in Java. Damit auch die Javaklassen in dem Projekt verfügbar sind, muss die `pom.xml`-Datei des Projekts `Hausratprodukte` um eine Abhängigkeit zum Projekt `Hausratmodell` ergänzt werden. Dazu fügen Sie einfach unter `<dependencies>` in der `pom.xml`-Datei den folgenden Eintrag hinzu:

```

<dependency>
    <groupId>org.faktorips.tutorial</groupId>
    <artifactId>Hausratmodell</artifactId>
    <version>1.0</version>
</dependency>

```

Anschließend müssen noch die JUnit 5 Bibliotheken zu dem Projekt hinzugefügt werden, da wir später eine Testklasse schreiben werden. Dazu ergänzen Sie in der `pom.xml`-Datei die folgenden Abhängigkeiten:

```

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.11.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.11.3</version>
    <scope>test</scope>
</dependency>

```

Falls wie im Archetype vorgeschlagen Java 17 genutzt wurde, muss die folgende Abhängigkeit ergänzt werden:

```

<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
  <version>4.0.0</version>
</dependency>

```

Damit die Änderungen in der `pom.xml`-Datei wirksam werden, führen Sie einen Rechtsklick auf das Projekt `Hausratprodukte` aus und wählen *Maven ▶ Update Project ▶ Ok*.

Öffnen Sie nun zunächst die Produktdefinitionsperspektive über *Window ▶ Perspective ▶ Open Perspective ▶ Other ▶ Produktdefintion* auswählen [12]. Falls Sie noch Editoren geöffnet haben, schließen Sie diese jetzt, um die Sichtweise der Fachabteilung auf das System zu haben. Damit Sie im *Problems-View* ausschließlich die Marker von Faktor-IPS sehen (und nicht auch Java-Marker u.a.) müssen Sie im *Problems-View* den Faktor-IPS Filter ein- und alle anderen Filter (standardmäßig mindestens der Defaultfilter) ausschalten.

12 Für die Verwendung von Faktor-IPS durch die Fachabteilung gibt es auch eine eigene Installation (in Eclipse Terminologie: ein eigenes Produkt), bei der ausschließlich die Produktdefinitionsperspektive verfügbar ist.

Zunächst legen wir zwei IPS-Packages an, eines für die Produkte und eines für die Deckungen. Dies geschieht wie in der Java Perspektive entweder über das Kontextmenü oder die Toolbar (als Quellordner `Hausratprodukte` wählen).

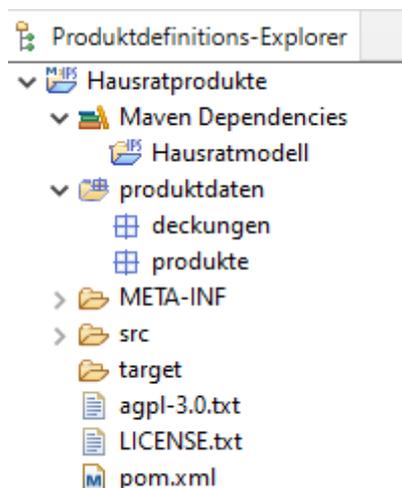


Figure 26. Anlegen zweier IPS Packages

Bemerkung: Sie können in dem Projekt beliebig viele Verzeichnisse anlegen. Zum Beispiel ein doc-Verzeichnis zur Verwaltung von Produktdokumenten.

Als erstes legen wir jetzt das Produkt `HR-Optimal` an. Markieren Sie dazu das gerade angelegte Package `produkte` und klicken dann in der Toolbar  an. Es öffnet sich der Wizard zum Erzeugen eines neuen Produktbausteins. Der Wizard bietet Ihnen nun die im Modell verfügbaren Produktklassen zur Auswahl, zu denen Sie Produktbausteine erstellen können. Wählen Sie `HausratProdukt`.

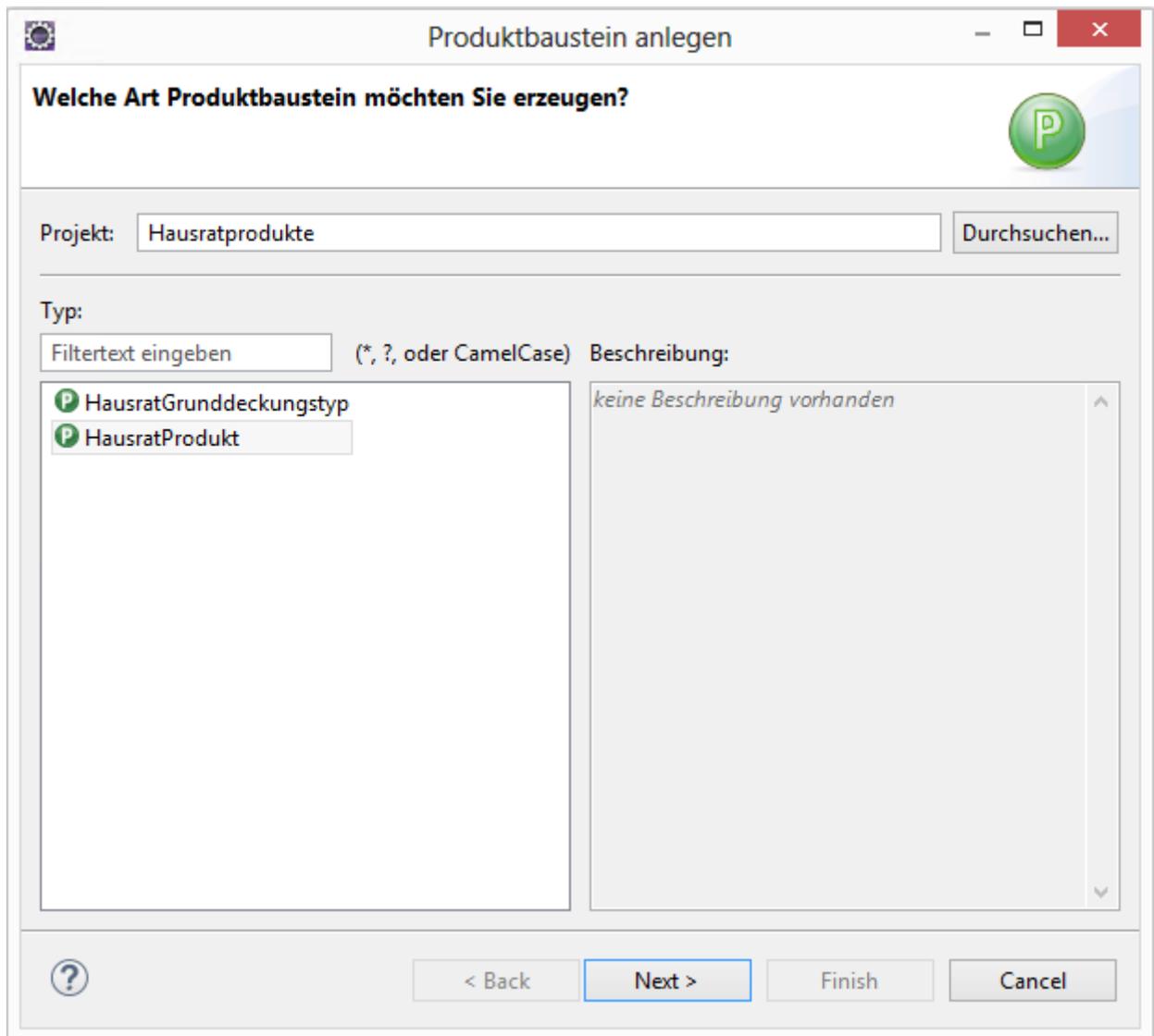


Figure 27. Auswahl der Produktbaustein-Klasse

Falls Sie keine Produktklasse finden, fehlt noch die Referenz auf das Hausratmodell-Projekt im *Faktor-IPS Build Path* (s.o.). Gehen Sie mit *Next* zur nächsten Seite des Wizards.

Hier geben Sie den Namen des Bausteins ein. Wenn Sie nun *Finish* drücken wird der Produktbaustein im Dateisystem angelegt.

Produktbaustein anlegen

HausratProdukt anlegen

Der Vollständige Name lautet "HR-Optimal 2019-07"

Typ auswählen:

Filtertext eingeben (*, ?, oder CamelCase) Beschreibung:

HausratProdukt keine Beschreibung vorhanden

Name: HR-Optimal

Gültig ab: 10.07.2019 Generation-ID: 2019-07

Laufzeit ID: hausrat.HR-Optimal 2019-07

Figure 28. Anlegen eines neuen Produkts

Mit Doppelklick auf den Produktbaustein im Produktdefinitions-Explorer öffnen Sie den Editor für den Baustein.

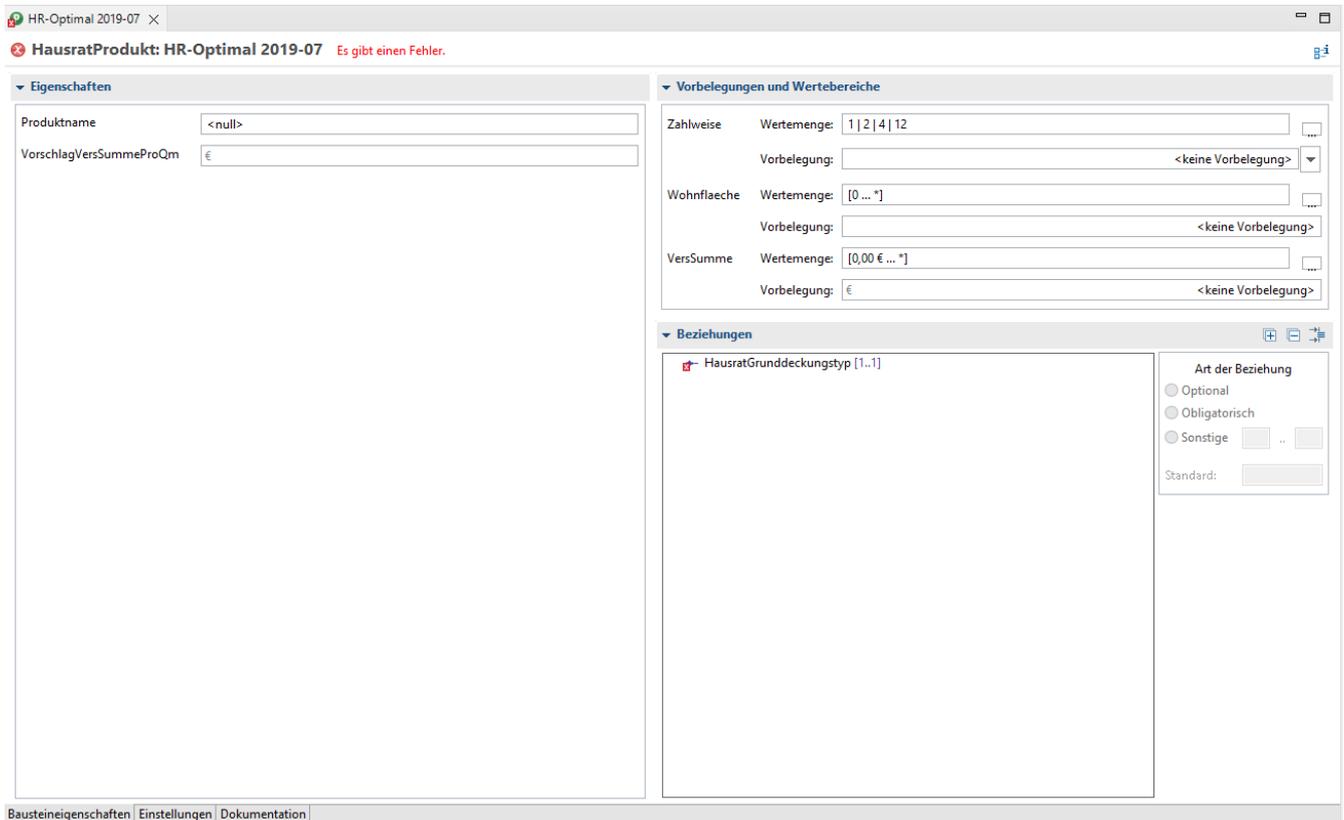


Figure 29. Produktbaustein-Editor für HR-Optimal

Die erste Seite des Editors zeigt:

- *Eigenschaften*
Enthält die Eigenschaften der Anpassungsstufe. Hier ist jedes in der Produktklasse definierte Attribut aufgelistet.
- *Vorbelegungen und Wertebereiche*
Enthält die Vorbelegungswerte und Wertebereiche für die Vertragseigenschaften.
- *Beziehungen*
Enthält die verwendeten anderen Produktbausteine.

Geben Sie Daten für das Produkt HR-Optimal entsprechend der folgenden Tabelle ein:

Konfigurationsmöglichkeit	HR-Optimal
Produktname	Hausrat Optimal
Vorschlag Versicherungssumme pro qm Wohnfläche	900 EUR
Vorgabewert Zahlweise	1 (jährlich)
Erlaubte Zahlweisen	1, 2, 4, 12
Vorgabewert Wohnflaeche	<keine Vorbelegung>
Erlaubte Wohnflaeche	0-2000
Vorgabewert Versicherungssumme	<keine Vorbelegung>
Versicherungssumme	10000 EUR - 5000000 EUR

Beim Anlegen des Produktbausteins haben Sie folgende Fehlermeldung bekommen:

Problems				
1 error, 0 warnings, 0 others				
Description	Resource	Path	Location	Type
Errors (1 item)				
Es wurden nur 0 Beziehungen des Typs HausratGrunddeckungstyp gefunden, es sind jedoch mindestens 1 Beziehungen notwendig.	HR-Optimal 2019-07.ipproduct	/Hausratprodukte/produktaten/produkte	Unknown	Faktor-IPS Problem

Figure 30. Fehlermeldung beim Anlegen des Produktbausteins

Um diese zu beheben, legen wir nun den Grunddeckungstyp für das Produkt an. Markieren Sie hierzu den Ordner Deckungen und legen einen neuen Produktbaustein mit Namen `HRD-Grunddeckung-Optimal` an basierend auf der Klasse `HausratGrunddeckungstyp`.

Nun müssen wir noch den Deckungstyp dem Produkt `HR-Optimal` zuordnen. Dies kann man bequem per Drag&Drop aus dem Produktdefinitions-Explorer erledigen. Öffnen Sie das Produkt `HR-Optimal`. Ziehen Sie die `HRD-Grunddeckung-Optimal` aus dem Explorer auf den Knoten HausratGrunddeckungstyp im Bereich *Beziehungen*.

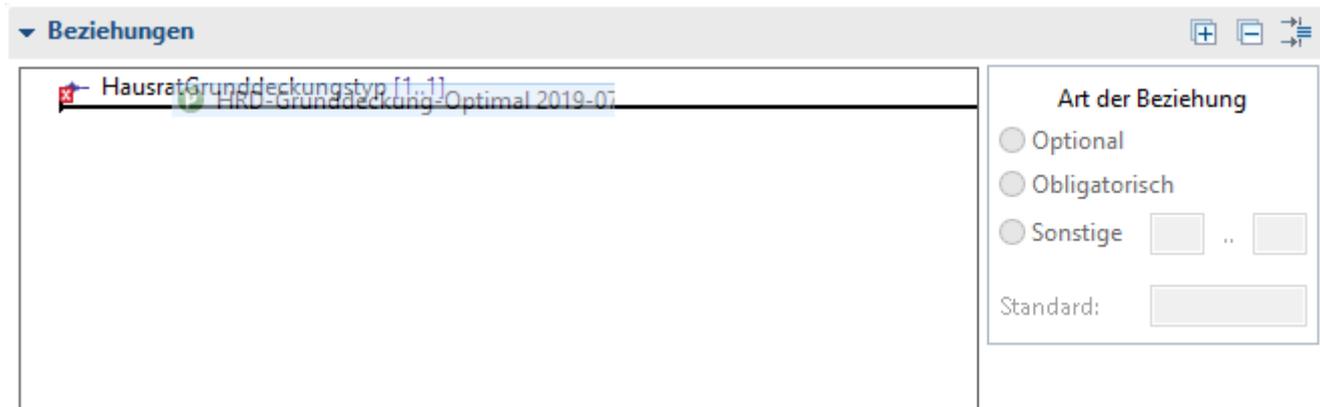


Figure 31. Zuordnung von Deckungstypen zu einem Produkt

Unter „Art der Beziehung“ muss *Obligatorisch* gewählt werden, da es sich um eine 1..1(1) Relation handelt. (Grund dafür ist die definierte Beziehung zwischen `HausratProdukt` und `HausratGrunddeckungstyp`. Siehe Abbildung 25)

Im nächsten Schritt legen wir das Produkt `HR-Kompakt` inklusive der Grunddeckung `HRD-GrunddeckungKompakt` an. Dies können Sie analog zum Produkt `HR-Optimal` machen. Alternativ können Sie einen Kopierassistenten verwenden, mit dem Sie einen Produktbaustein inklusive aller verwendeter Bausteine kopieren können. Wenn Sie dies ausprobieren möchten, markieren Sie das Produkt `HR-Optimal` im Produktdefinitions-Explorer und wählen im Kontextmenü *New ► Copy Product ...*

Im geöffneten Dialog geben Sie als Search Pattern (Suchen nach) `Optimal` und als *Replace Pattern* (Ersetzen durch) `Kompakt` ein, klicken Next und dann Finish. Faktor-IPS legt die Bausteine `HR-Kompakt` und `HRD-Grunddeckung` neu an. Öffnen Sie `HR-Kompakt` und geben Sie Daten entsprechend der nachfolgenden Tabelle ein:

Konfigurationsmöglichkeit	HR-Kompakt
Produktname	Hausrat Kompakt

Konfigurationsmöglichkeit	HR-Kompakt
Vorschlag Versicherungssumme pro qm Wohnfläche	600 EUR
Vorgabewert Zahlweise	jährlich
Erlaubte Zahlweisen	halbjährlich, jährlich
Vorgabewert Wohnflaeche	<keine Vorbelegung>
Erlaubte Wohnflaeche	0-1000 qm
Vorgabewert Versicherungssumme	<keine Vorbelegung>
Versicherungssumme	10000 EUR - 2000000 EUR

Damit ist die Definition der beiden Produkte zunächst abgeschlossen. Im *Produktdefinitions-Explorer* sollten Sie folgendes sehen.

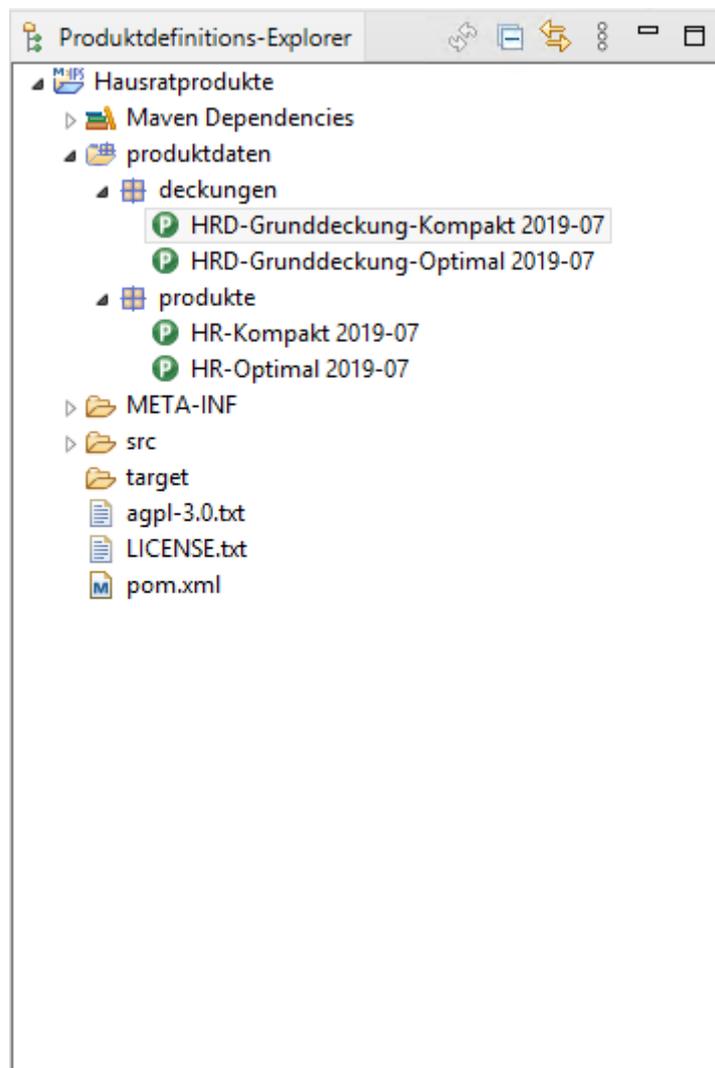


Figure 32. Darstellung der Produkte im Produktdefinitions-Explorer

Neben dem *Produktdefinitions-Explorer* stehen Ihnen zwei weitere Werkzeuge zur Analyse der Produktdefinition zur Verfügung. Die Struktur eines Produktes können Sie sich mit *Show Structure* im Kontextmenü anzeigen lassen. Die unterschiedliche Verwendung eines Bausteins mit *Search References*. Darüber hinaus können Sie die Reihenfolge der Pakete über den Menüpunkt *Edit Sort Order* frei festlegen.

Rechts neben dem Editor zur Eingabe der Produktdaten befindet sich der *Model Description View*. Dieser zeigt passend zum in Bearbeitung befindlichen Produktbaustein die Dokumentation der zugehörigen Produktklasse. Wenn Sie die einmal ausprobieren wollen, dokumentieren Sie z.B. das Attribut `produktname` im Modell, schließen Sie den Bausteineditor und öffnen ihn erneut.

Zugriff auf Produktinformationen zur Laufzeit

Nachdem wir die Produktdaten erfasst haben, beschäftigen wir uns nun damit, wie man zur Laufzeit (in einer Anwendung/einem Testfall) auf diese zugreift. Hierzu werden wir einen *JUnit-Test* schreiben, den wir im Laufe des Tutorials weiter ausbauen.

Zum Zugriff auf Produktdaten stellt Faktor-IPS das Interface `IRuntimeRepository` bereit. Die Implementierung `ClassLoaderRuntimeRepository` erlaubt den Zugriff auf die mit Faktor-IPS erfassten Produktdaten und lädt die Daten über einen Classloader. Damit dies möglich ist, macht Faktor-IPS zwei Dinge:

1. Die Dateien, die die Produktinformationen enthalten, werden in den Java Sourcefolder mit dem Namen `src/main/resources` kopiert. Damit sind diese Dateien im Build Path des Projektes enthalten und können über den Classloader geladen werden.
2. Welche Daten sich im `ClassLoaderRuntimeRepository` befinden, ist in einem Inhaltsverzeichnis vermerkt. Dieses Inhaltsverzeichnis (Englisch: table of contents, toc) wird von Faktor-IPS ebenfalls in eine Datei generiert, die als Toc-File bezeichnet wird. Die Datei heißt standardmäßig `faktorips-repository-toc.xml`. Die Namen lassen sich in der „ipsproject“ Datei im Abschnitt `IpsObjectPath` konfigurieren.

Ein `ClassLoaderRuntimeRepository` wird über die statische `create(...)`-Methode der Klasse erzeugt. Als Parameter wird der Pfad zum Toc-File übergeben. Das Toc-File wird direkt beim Erzeugen des Repositorys über `ClassLoader.getResourceAsStream()` gelesen. Alle weiteren Daten werden erst (wiederum über den Classloader) geladen, wenn auf sie zugegriffen wird.

Das Laden der Daten über den Classloader hat im Gegensatz zum Laden aus dem Filesystem den großen Vorteil, dass es völlig plattformunabhängig ist. So kann der Programmcode z.B. ohne Änderungen auf z/OS laufen.

Einen Produktbaustein kann man über die Methode `getProductComponent(...)` erhalten. Als Parameter übergibt man die Runtime-ID des Bausteins. Da das Interface `IRuntimeRepository` unabhängig vom konkreten Modell (in unserem Fall also dem `Hausratmodell`) ist, muss man das Ergebnis noch auf die konkrete Produktklasse casten.

Probieren wir dies doch einmal in einem JUnit Testfall aus. Legen Sie hierzu zunächst im Projekt Hausratprodukte einen neuen Java Sourcefolder „test“ an. Am einfachsten geht dies, indem Sie im `Package-Explorer` das Projekt markieren und im Kontextmenü `Build Path` ► `New Source Folder...` aufrufen.

Danach markieren Sie den neuen Sourcefolder und legen einen JUnit Testfall an, indem Sie in der Toolbar auf den Pfeil neben dem Icon



klicken und dann *JUnit Test Case* auswählen. In dem Dialog geben Sie als Namen für die Testfallklasse „TutorialTest“ ein und haken an, dass auch die `setUp()` Methode generiert werden soll. Die Warnung, dass die Verwendung des Defaultpackages nicht empfohlen wird, ignorieren wir in dem Tutorial. Für unseren Test verwenden wir die JUnit 5 *JUnit Jupiter*. Beim Beenden des Wizards werden wir gefragt, ob wir die JUnit Library zum Build Path des Projektes hinzufügen wollen. Klicken Sie in diesem Dialog auf *OK*. Der nächste Kasten enthält den Sourcecode der Testfallklasse.

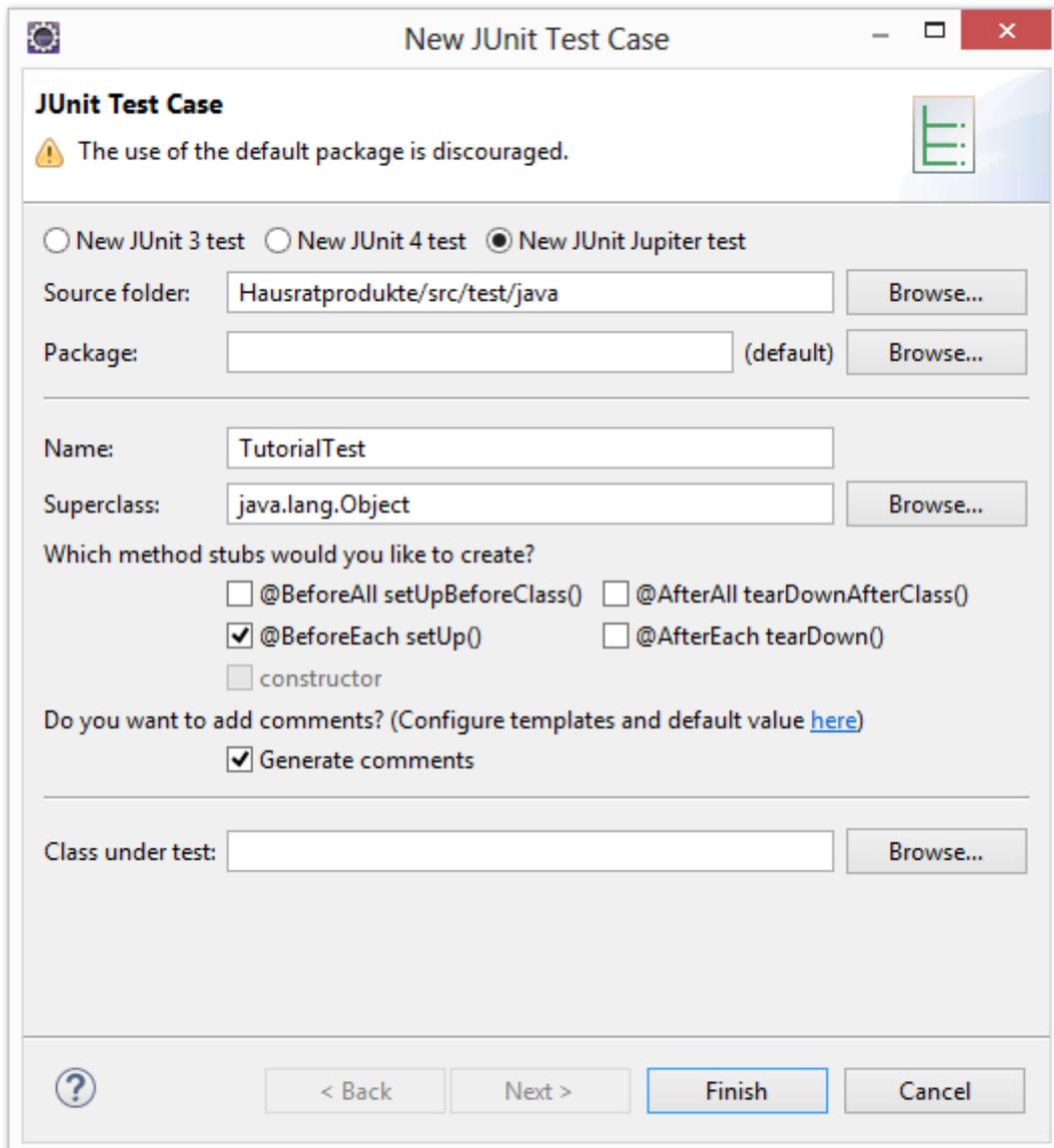


Figure 33. Wizard zur Erstellung eines JUnit Tests

Anstatt die Korrektheit der Produktdaten mit `assert*`-Statements zu testen, geben wir sie hier mit `println` auf der Konsole aus. Vergessen Sie nicht, das Jahr anzupassen, wenn Sie einen anderen Gültigkeitsbereich als 2019-07 gewählt haben.

```
public class TutorialTest {  
  
    private IRuntimeRepository repository;  
    private HausratProdukt kompaktProdukt;
```

```

@BeforeEach
public void setUp() throws Exception {
    // Repository erzeugen
    repository = ClassloaderRuntimeRepository
        .create("org/faktorips/tutorial/produktdaten/internal/faktorips-
repository-toc.xml");

    // Referenz auf das Kompaktprodukt aus dem Repository holen
    IProductComponent pc = repository.getProductComponent("hausrat.HR-Kompakt
2019-07");

    // Auf die eigenen Modellklassen casten
    kompaktProdukt = (HausratProdukt) pc;
}

@Test
public void test() {
    System.out.println("Produktname: " + kompaktProdukt.getProduktname());
    System.out.println("Vorschlag Vs pro 1qm: " +
kompaktProdukt.getVorschlagVersSummeProQm());
    System.out.println("Default Zahlweise: " +
kompaktProdukt.getDefaultValueZahlweise());
    System.out.println("Erlaubte Zahlweisen: " +
kompaktProdukt.getAllowedValuesForZahlweise());
    System.out.println("Default Vs: " +
kompaktProdukt.getDefaultValueVersSumme());
    System.out.println("Bereich Vs: " +
kompaktProdukt.getAllowedValuesForVersSumme());
    System.out.println("Default Wohnflaeche: " +
kompaktProdukt.getDefaultValueWohnflaeche());
    System.out.println("Bereich Wohnflaeche: " +
kompaktProdukt.getAllowedValuesForWohnflaeche());
}
}

```

Wenn Sie den Test nun ausführen, sollte er folgendes ausgeben:

```

Produktname: Hausrat Kompakt
Vorschlag Vs pro 1qm: 600.00 EUR
Default Zahlweise: 1
Erlaubte Zahlweisen: [1, 2]
Default Vs: MoneyNull
Bereich Vs: 10000.00 EUR-2000000.00 EUR
Default Wohnflaeche: null
Bereich Wohnflaeche: 0-1000

```

Damit haben wir einen Einblick in die Modellierung mit Faktor -IPS bekommen und haben erste, einfache Produkte angelegt und zur Laufzeit auf Produktdaten zugegriffen.

In Teil 2 werden wir in die Verwendung von Tabellen & Formeln einführen. Diese werden wir

nutzen, um das Hausratmodell so zu erweitern, dass den Produkten flexibel Zusatzdeckungen durch Anwender aus der Fachabteilung hinzu konfiguriert werden können, ohne dass das Modell erweitert werden muss.

Part 1: Modeling and Product Configuration

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 doc.faktorzehn.org and install it on your machine.

Part 1 Overview

The first part of the tutorial is organized as follows:

- **Hello Faktor-IPS**

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

- **Extending the Home Contents Model** The home contents model is completed by adding additional attributes to the HomeContract class.

- **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.

- 1 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.

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 <https://www.faktorzehn.org/de/download/>. In this tutorial we will use Eclipse 4.30 (2023-12) and Faktor-IPS 25.1.1.release in English. Make sure to also install the Faktor-IPS addons (m2e and Groovy).

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 [3]:

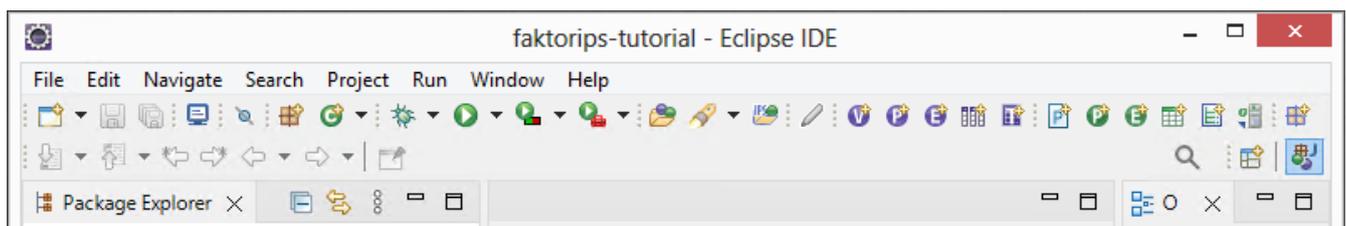


Figure 34. Faktor-IPS Icons

- 3 Click in the menu options *Window* ► *Perspective* ► *Open Perspective* ► *Java*

If the *Model Explorer* is not visible, that’s because you used this workspace before you installed Faktor-IPS. In this case, select *Window* ► *Perspective* ► *Reset Perspective*.

Faktor-IPS projects are regular Java projects or Maven projects with an additional Faktor-IPS Nature. First, create a new Maven project named "**HomeModel**". For this the Maven archetype for Faktor-IPS can be used, which creates a Maven project with the Faktor-IPS Nature.

Open the command line and navigate to the directory in which you want to create the project. From there, run the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.faktorips
-DarchetypeArtifactId=faktorips-maven-archetype -DarchetypeVersion=25.1.1.release
-DgroupId=org.faktorips.tutorial -DartifactId=HomeModel -Dversion=1.0
-Dpackage=org.faktorips.tutorial.model -DjavaVersion=17 -DIPS-Language=en -DIPS
-IpsModelProject=true -DIPS-IpsProductDefinitionProject=false -DIPS-SourceFolder=model
-DIPS-RuntimeIdPrefix=home. -DIPS-ConfigureIPSBUILD=true
```

This command created the Maven project "HomeModel". The Faktor-IPS Nature and with it the runtime libraries of Faktor-IPS have been added to the project.

In this project we will create the model classes. We have named the source directory in which the model definitions are stored as "model" (through the parameter `-DIPS-SourceFolder=model`). Within this directory, the model can be structured using packages, as in Java. Like Java, Faktor-IPS uses qualified names to identify the classes of the model.

The base package for the generated Java classes was named "org.faktorips.tutorial.model" (through the parameter `-Dpackage=org.faktorips.tutorial.model`). The runtime-ID-prefix is "home.". The meaning of the runtime-ID-prefix is explained in the chapter "Defining the Products".

The exact functions of the used parameters can be found in the [documentation of the archetype](#).

The newly created project must be imported into the Eclipse workspace. Click *File* ► *Import* ► *Maven* ► *Existing Maven Projects*. In the dialog, add the project folder as the root directory and import the project by clicking on *Finish*.

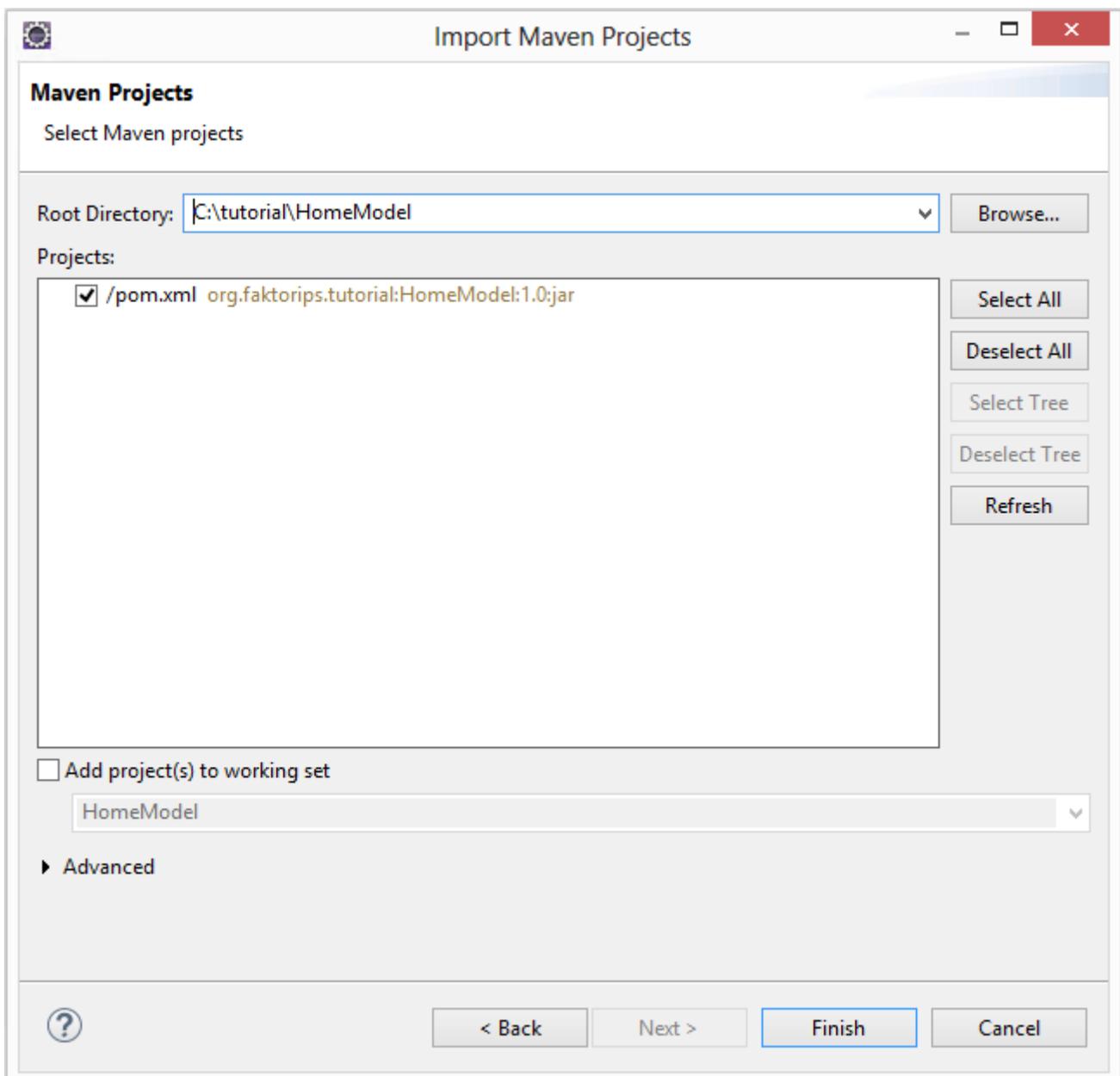


Figure 35. Import the Maven project into Eclipse

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*.

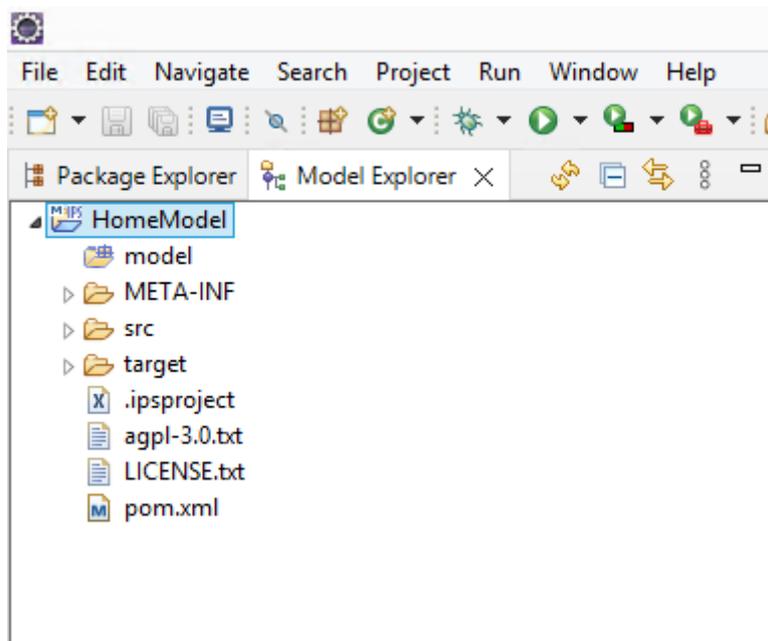


Figure 36. The Project View of the Model Explorer

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 in the AddIpsNature dialog box/archetype parameters, settings relating to the generation 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"* (*new ► IPS Package*). Alternatively, you can create IPS packages using the button  in the toolbar.

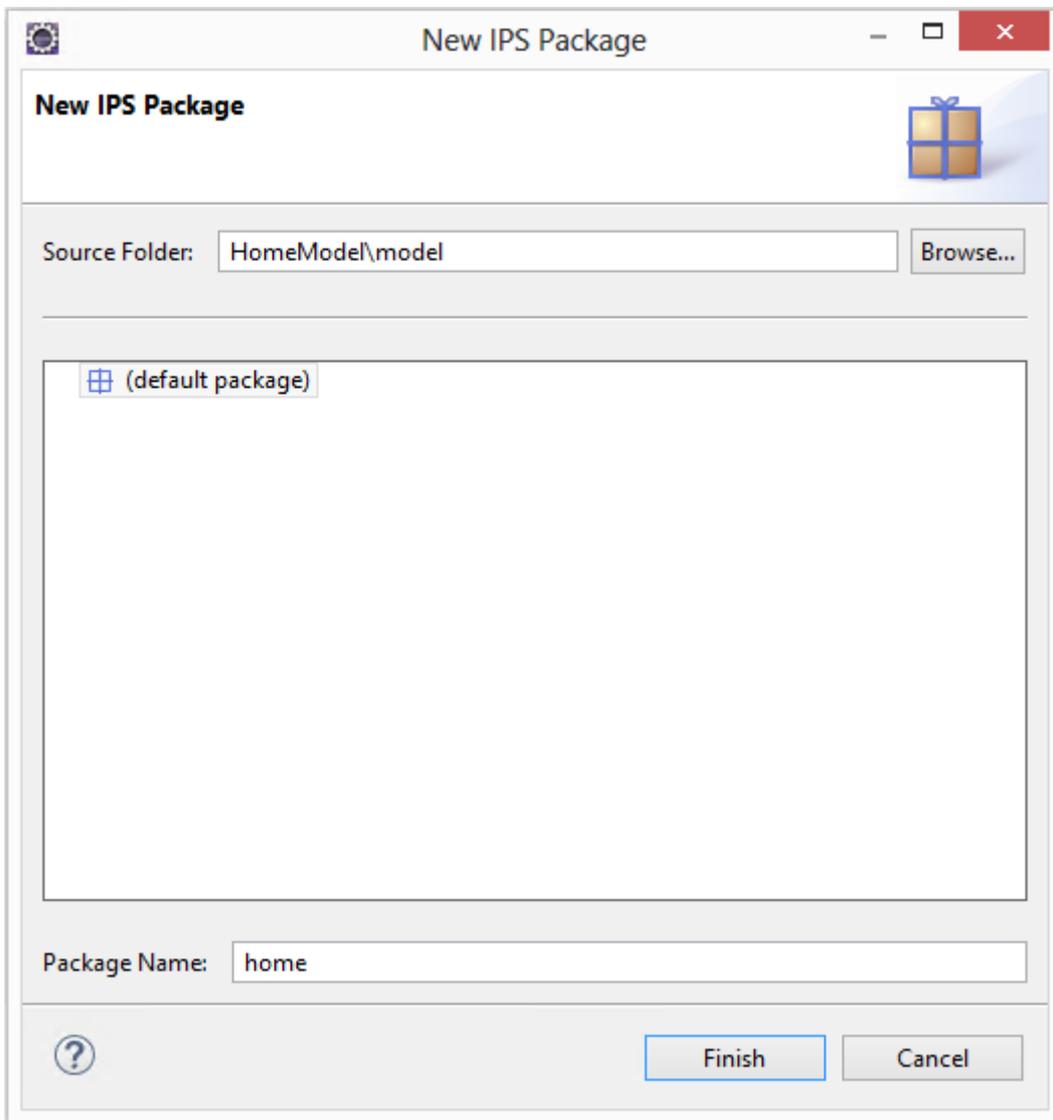


Figure 37. Creating a IPS package

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 .

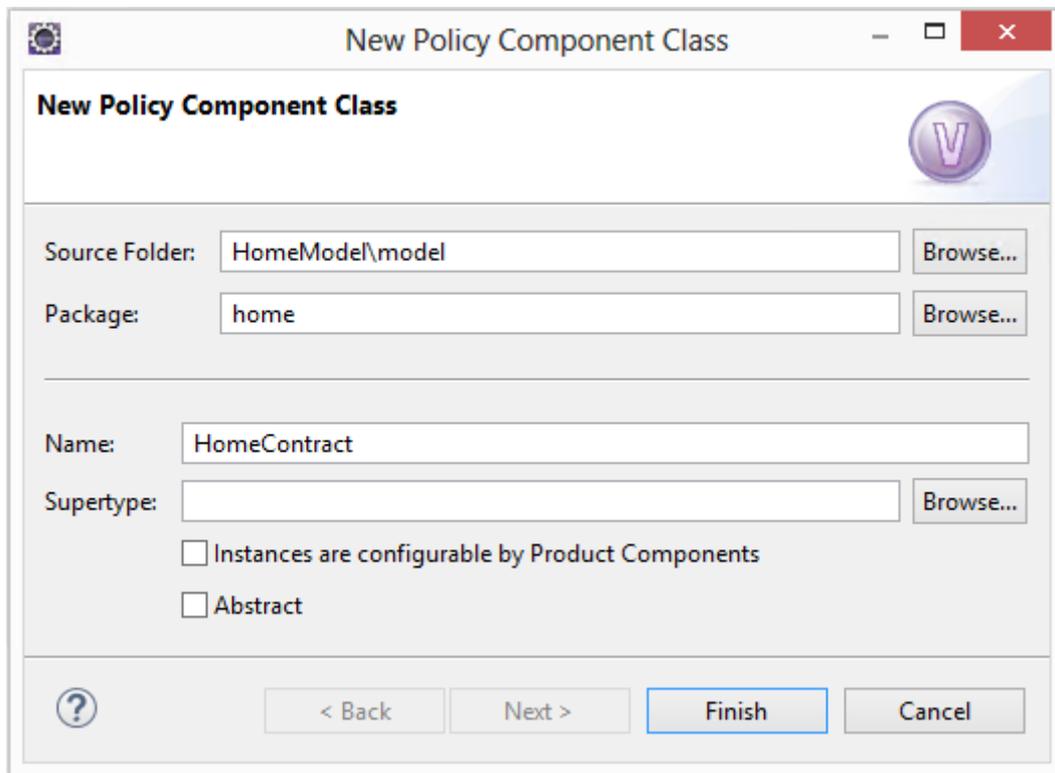


Figure 38. Creating a new Contract Class

In the dialog box, your source directory and package are already prefilled with the appropriate names, so you just have to add the class name `"HomeContract"` 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.ipspolicyempttype"`.

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

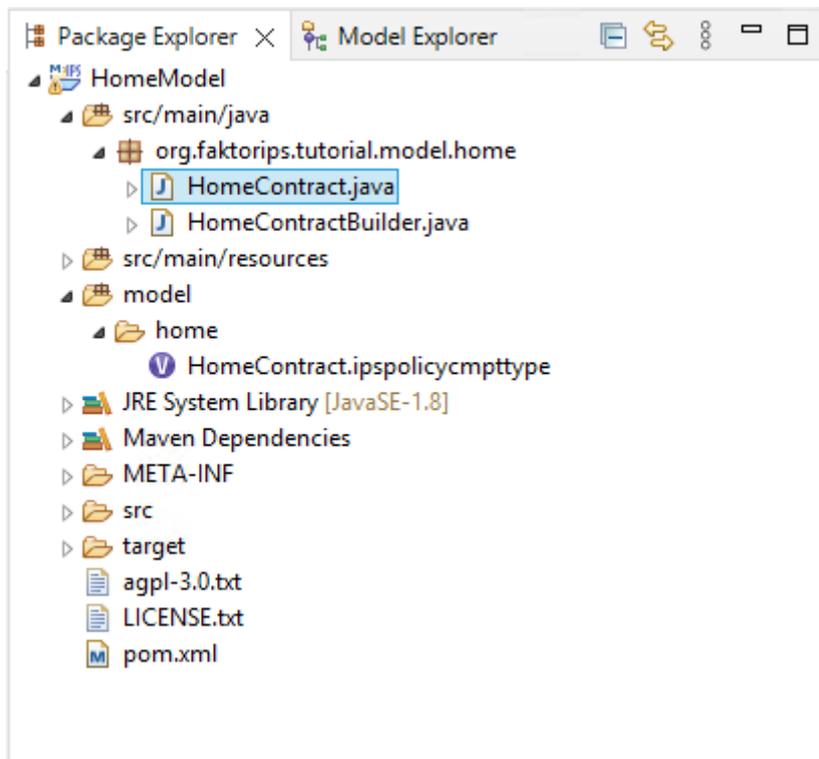


Figure 39. Generated Classes

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 class `HomeContractBuilder` creates contract structures using the [builder design pattern](#).

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:

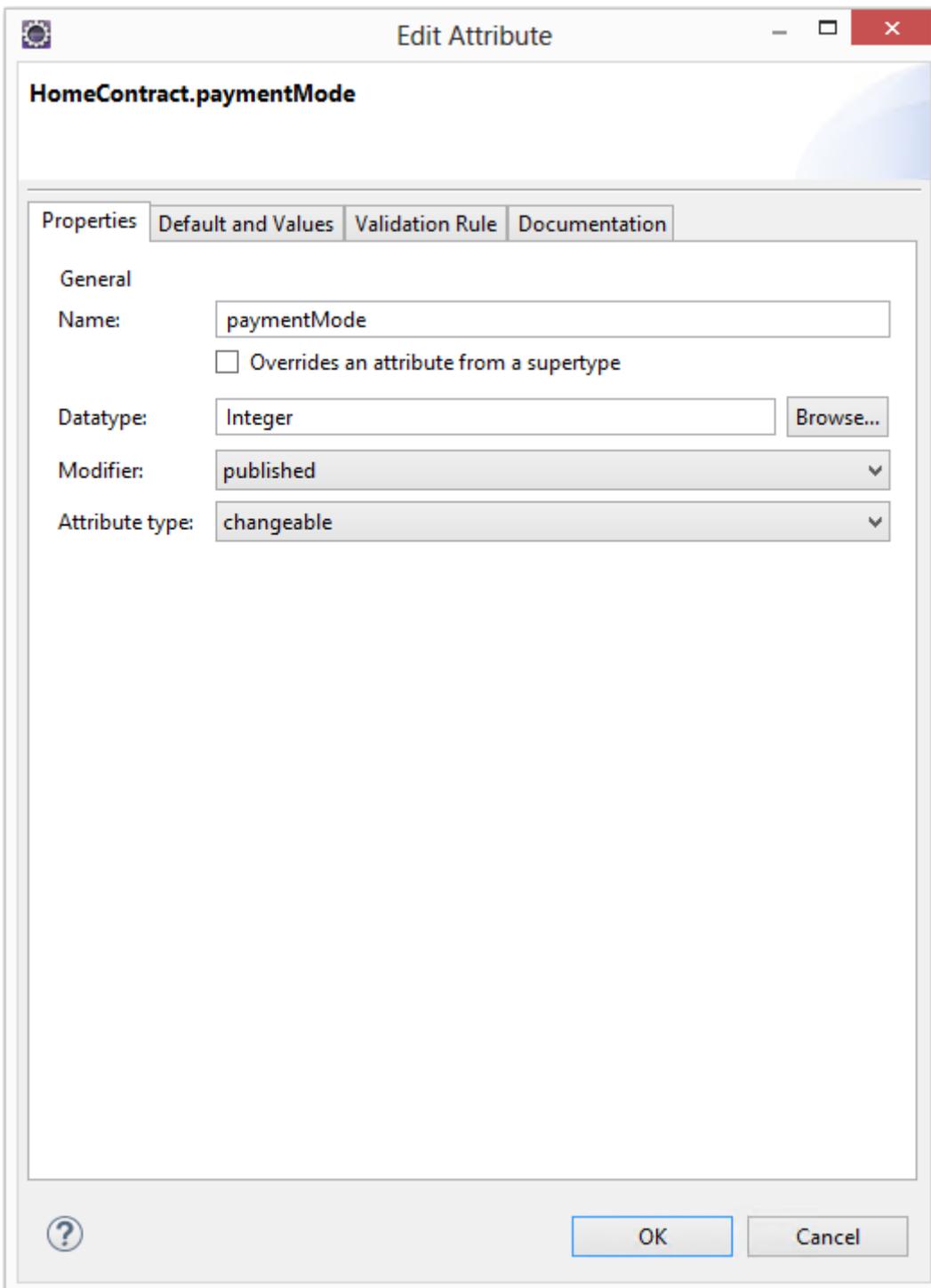


Figure 40. Dialog for Creating a New Attribute

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 just overrides certain properties (e.g. the default value) [4].
Datatype	The datatype of the attribute.

Value	Meaning
Modifier	Similar to a Java modifier. The additional published modifier means that the property is included in the published interface [5].
Attribute type	<p>The type of the attribute.</p> <p>* changeable Applies to changeable properties with getter and setter methods.</p> <p>* constant Applies to constant, immutable properties.</p> <p>* 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 be queried by a getter method. For example, the property <code>grossPremium</code> can be calculated by a method named <code>computePremium()</code> and subsequently retrieved by the <code>getGrossPremium()</code> method.</p> <p>* 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 <code>getAge()</code> by means of that person's date of birth.</p>

4 Corresponds to the `@Override` annotation of Java 5 and newer.

5 **Note:** The activation/deactivation of the generation of the published interfaces takes place via the context menu ► properties ► Faktor-IPS Code Generator of the corresponding project.

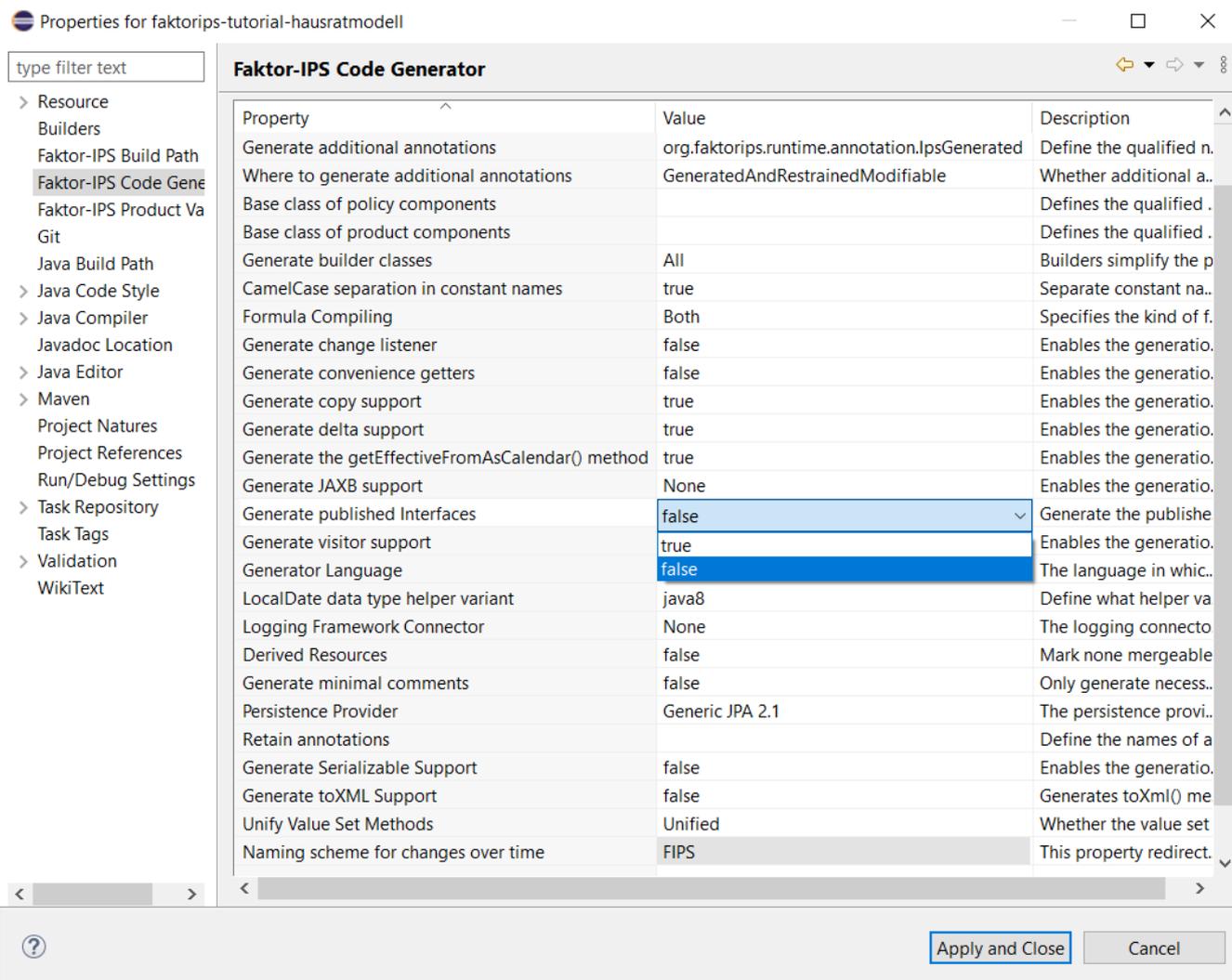


Figure 41. Activation/Deactivation of the generation of the published interfaces

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 class `HomeContract` now includes accessor methods for the attribute and saves the state in a private member variable.

```

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

/**
 * Creates a new HomeContract.
 *
 * @generated
 */

```

```

@IpsGenerated
public HomeContract() {
    super();
}

/**
 * Returns the paymentMode.
 *
 * @generated
 */
@IpsAttribute(name = "paymentMode", kind = AttributeKind.CHANGEABLE, valueSetKind =
ValueSetKind.AllValues)
@IpsGenerated
public Integer getPaymentMode() {
    return paymentMode;
}

/**
 * Sets the value of attribute paymentMode.
 *
 * @generated
 */
@IpsAttributeSetter("paymentMode")
@IpsGenerated
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:

```

/**
 * Returns the paymentMode.
 *
 * @generated
 */
@IpsAttribute(name = "paymentMode", kind = AttributeKind.CHANGEABLE, valueSetKind =
ValueSetKind.AllValues)
@IpsGenerated
public Integer getPaymentMode() {
    System.out.println("getPaymentMode");
    return paymentMode;
}

```

```

/**
 * Sets the value of attribute paymentMode.
 *
 * @generated NOT
 */
@IpsAttributeSetter("paymentMode")
@IpsGenerated
public void setPaymentMode(Integer newValue) {
    System.out.println("setPaymentMode");
    this.paymentMode = newValue;
}

```

Now re-generate the source code of the `HomeContract` class. 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 [6].

6 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*.

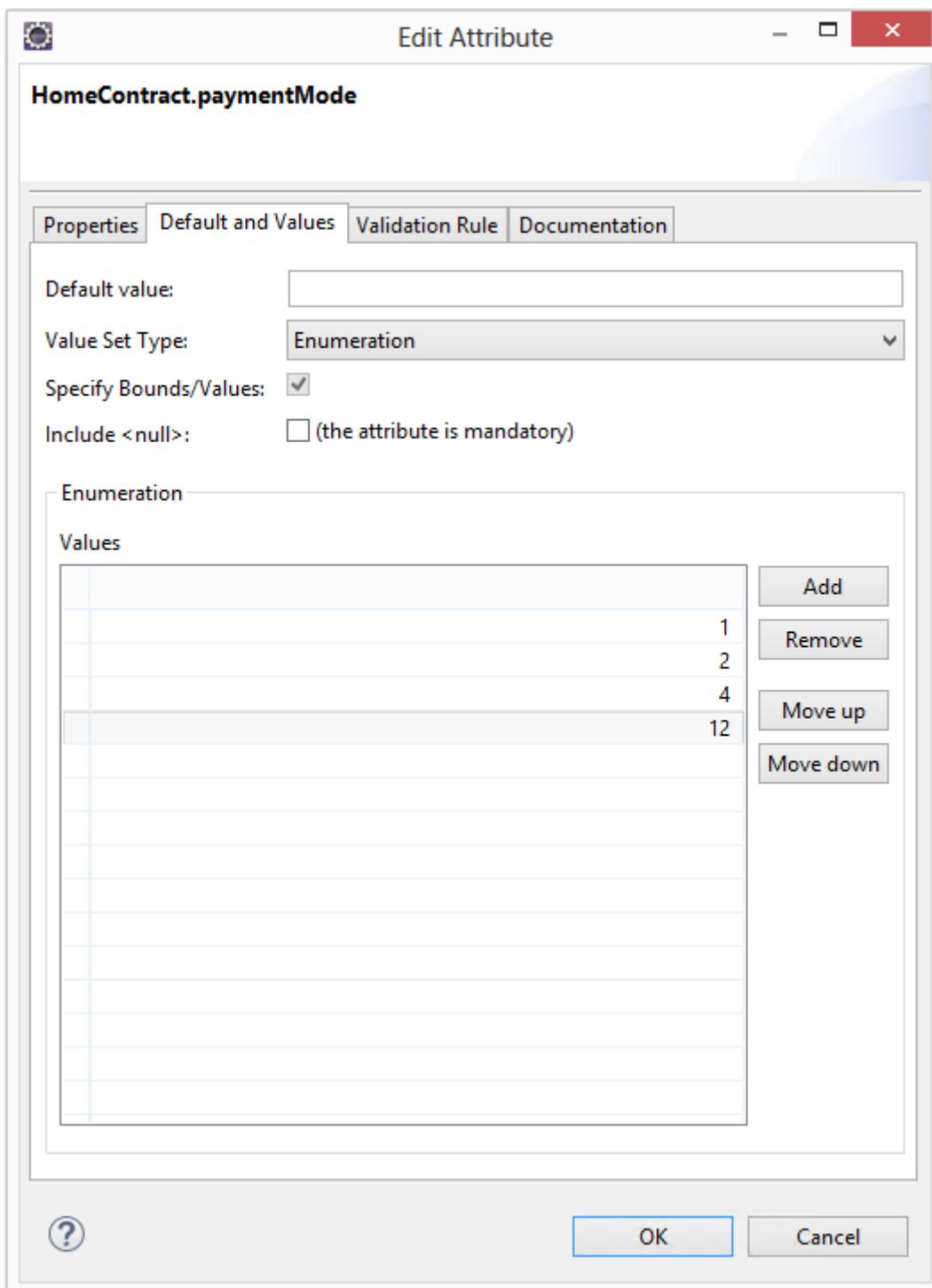


Figure 42. Define Legal Values for an Attribute

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 so called problem markers that appear in the *Problem View* and in the Explorers.

Description	Resource	Path	Location	Type
0 errors, 1 warning, 0 others				
Warnings (1 item)				
⚠ The default value 0 is no member of the specified valueSet!	HomeContract.ipspolicycmpttype	/HomeModel/model/home	Unknown	Faktor-IPS Problem

Figure 43. Error Warnings Inside the Problems View

Now delete the default value 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 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:

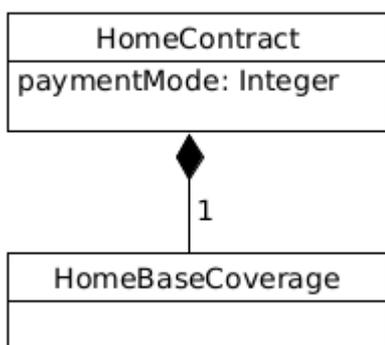


Figure 44. Contract model

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[7].

⁷ 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.

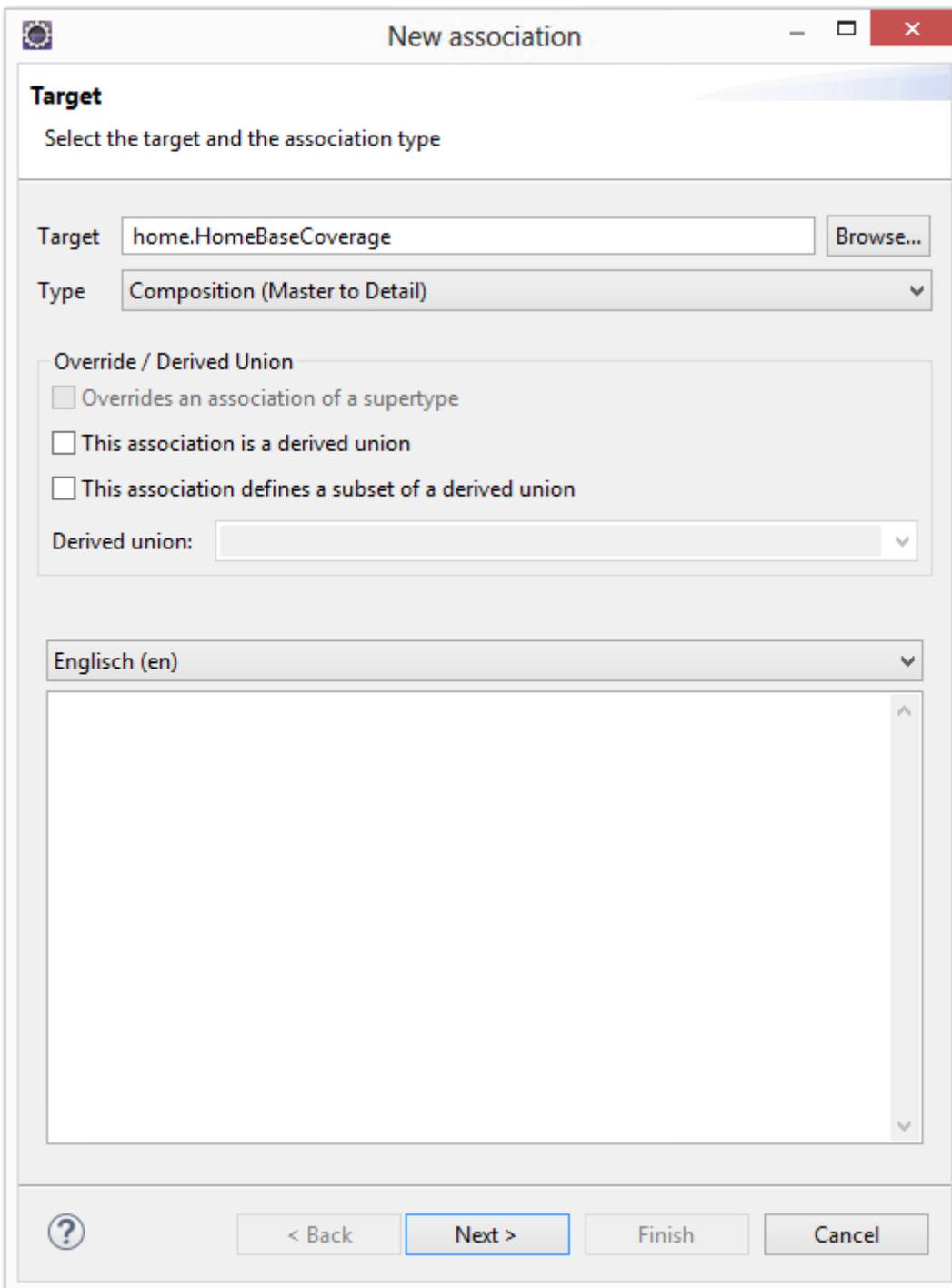


Figure 45. Creating a new Relationship

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 singular role `HomeBaseCoverage` as suggested and leave the plural role empty. The plural is used so that the code generator can create comprehensible source code for associations that allow multiple targets.

New association

Association properties
Define association properties

Properties

Target role (singular)

Target role (plural)

Minimum cardinality

Maximum cardinality

Note: This association is not constrained by product structure.

Qualification

This association is qualified

Note: Qualification is only applicable, if the target type is configurable by a product

? < Back Next > Finish Cancel

Figure 46. 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.

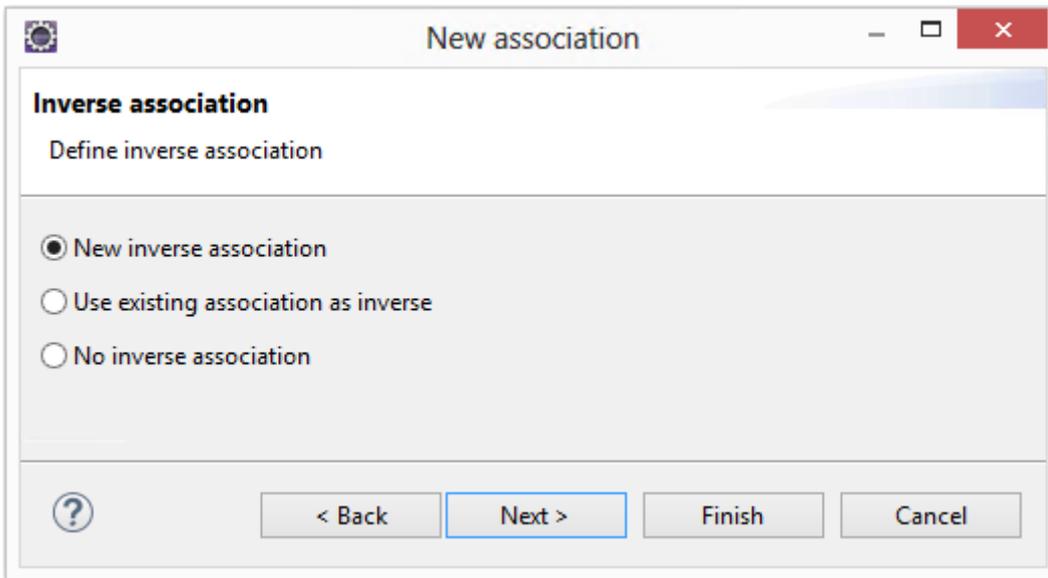


Figure 47. Create new backward relationship

In the next window, enter the role description of the inverse relationship.

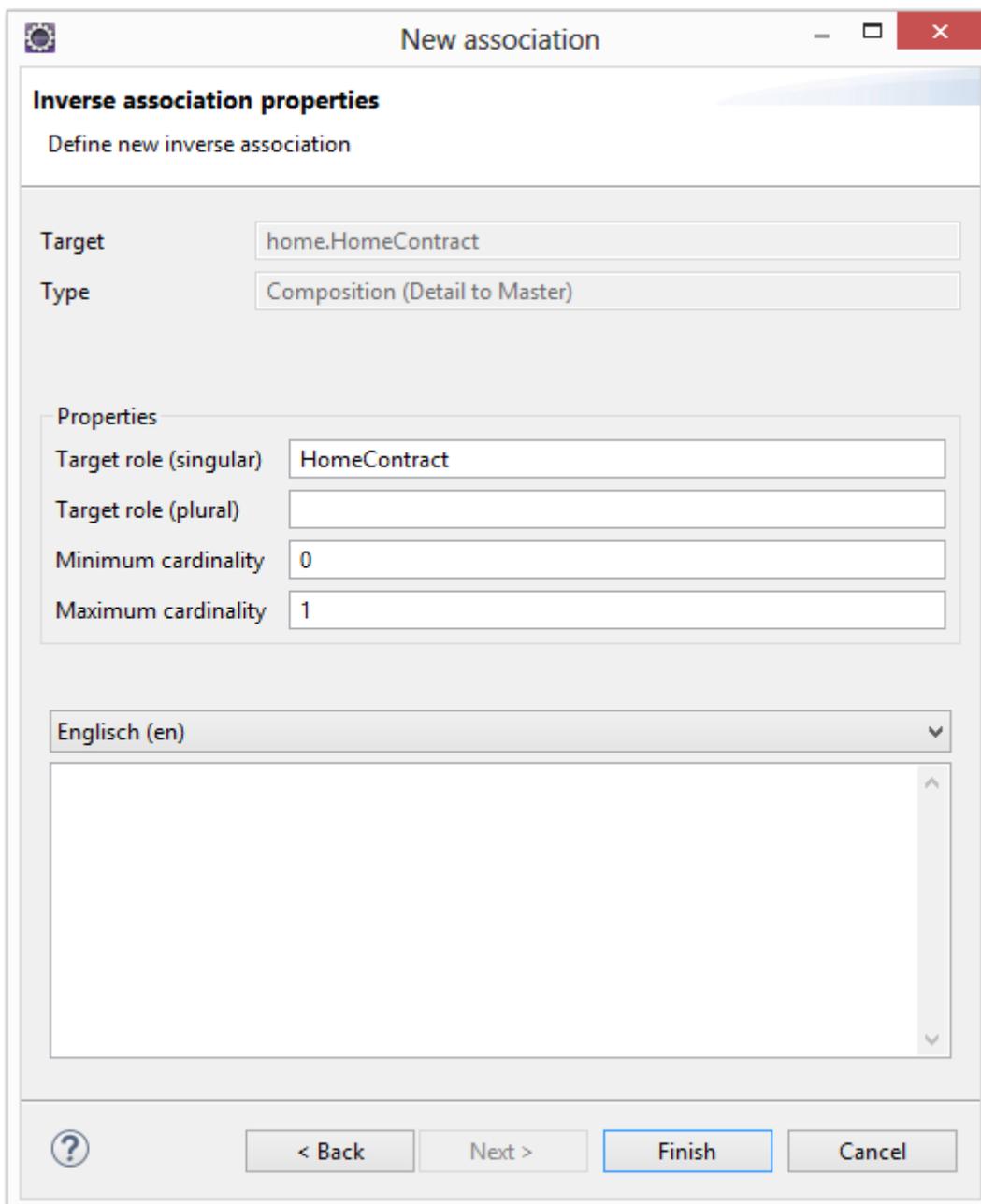


Figure 48. Properties of the backward relationship

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 class `HomeContract` methods have been created to add basic coverage to the `HomeContract`. Inside the class `HomeBaseCoverage` exists a method to navigate to `HomeContract`. If the model defines both a forward and a backward relationship, both directions are taken into account. This means that if `setHomeBaseCoverage(HomeBaseCoverage cov)` is called on a contract instance of `HomeContract`, `cov.getHomeContract()` will return that contract again. This will be clear if you take a look at the implementation of the `setHomeBaseCoverage()` method within the `HomeContract` class:

```
@IpsAssociationAdder(association = "HomeBaseCoverage")
public void setHomeBaseCoverage(IHomeBaseCoverage newObject) {
    if (homeBaseCoverage != null) {
```

```

        homeBaseCoverage.setHomeContractInternal(null);
    }
    if (newObject != null) {
        homeBaseCoverage.setHomeContractInternal(this);
    }
    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. The following figure shows the model as is.

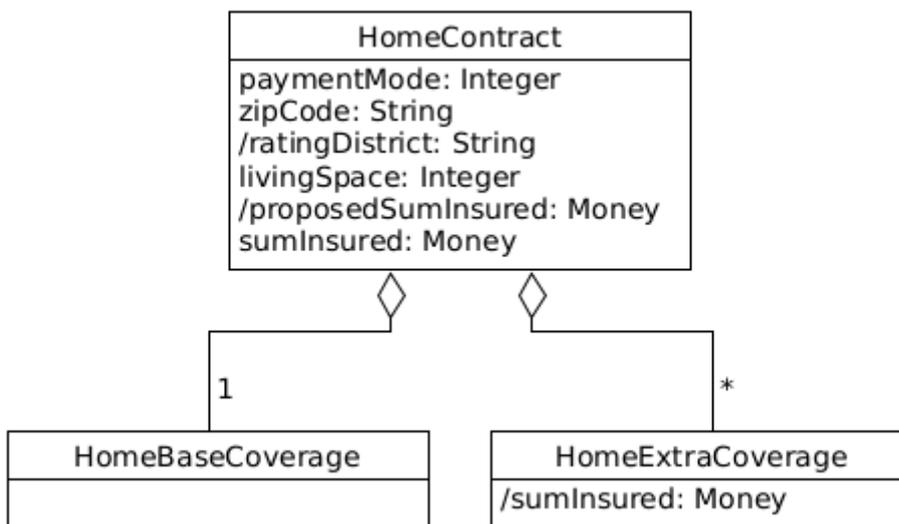


Figure 49. Home Contents Model with Base Coverage and Extra Coverage

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
<code>zipcode: String</code>	The zipcode of the insured home contents
<code>/ratingDistrict: String</code>	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)!

Name: Datatype	Description, Comments
<code>livingSpace</code> : <i>Integer</i>	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, leave the maximum field empty.
<code>/proposedSumInsured</code> : <i>Money</i>	A suggested value for the sum insured. It is determined based on the living space. → Make sure to set the <i>AttributeType</i> to derived (computation on each call of the getter method)!
<code>sumInsured</code> : <i>Money</i>	The sum insured. Allowable values range from min=0 EUR and max=unlimited (empty field).

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

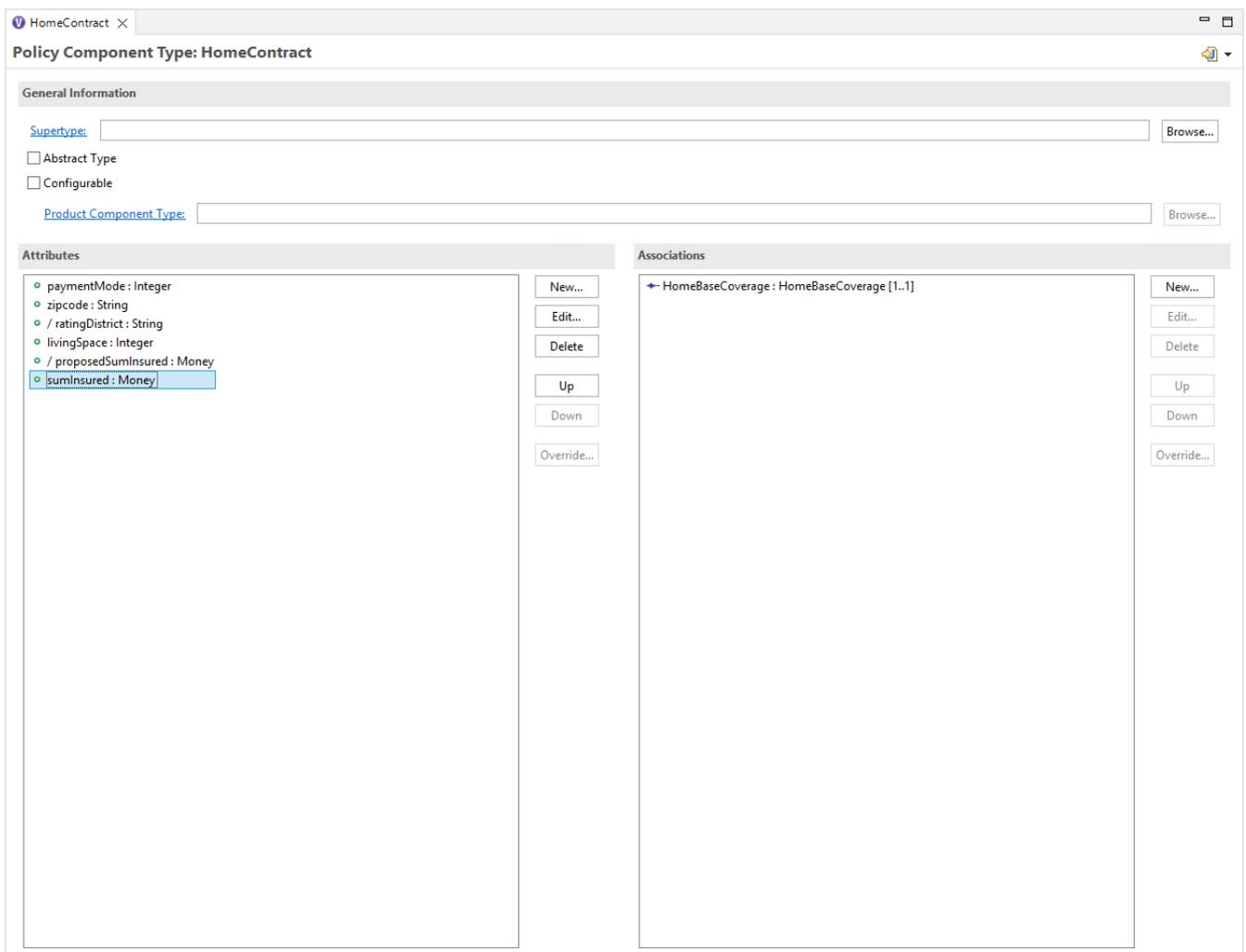


Figure 50. The `HomeContract` Class

The derived attributes are prefixed with 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.

```
/**
 * Returns the ratingDistrict.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "ratingDistrict", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public String getRatingDistrict() {
    // begin-user-code
    // TODO: later we'll implement this with a table lookup
    return "I";
    // end-user-code
}

/**
 * Returns the proposedSumInsured.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "proposedSumInsured", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getProposedSumInsured() {
    // begin-user-code
    // TODO: later we'll implement this with a product data lookup
    return Money.euro(650).multiply(livingSpace);
    // end-user-code
}
```

`@restrainedmodifiable` is generated by the generator instead of `@generated` for certain methods (e.g. in generated test classes or rules) and indicates that the developer can add their own code. The section in which the own code may appear is indicated by comments. `@restrainedmodifiable` can only be used if the annotation was created by the generator. A replacement of `@generated` and insertion of the appropriate comment lines does not work and is overwritten by the generator.

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.

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.

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 paymentMode	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	10 Tsd - 2 Mio Euro	10 Tsd - 5 Mio 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:

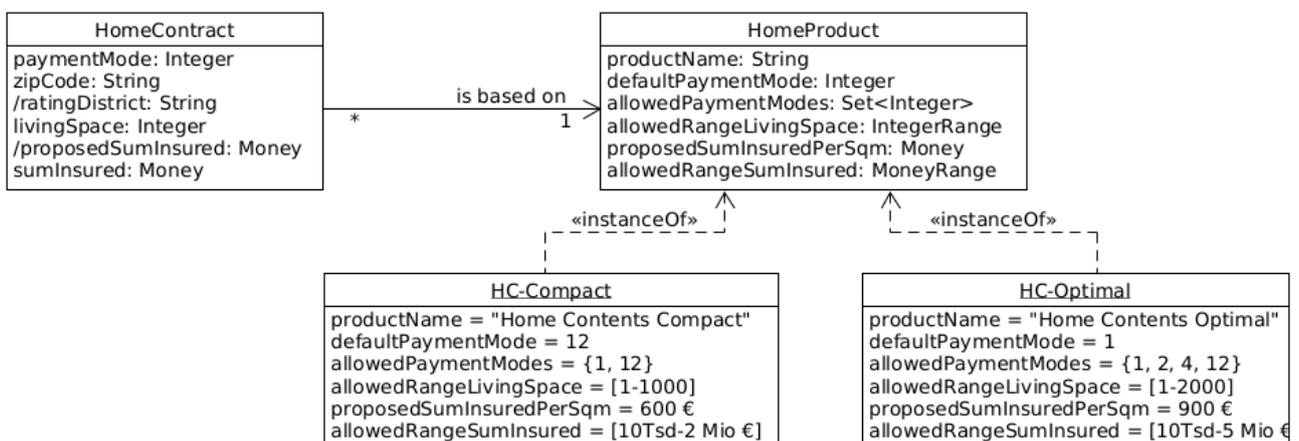


Figure 51. Home Contents Model with Product Classes. Changes Over Time are Not Considered.

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 Type* field, enter "home.HomeContract". When you click *Finish*, the editor for product classes will open for you.

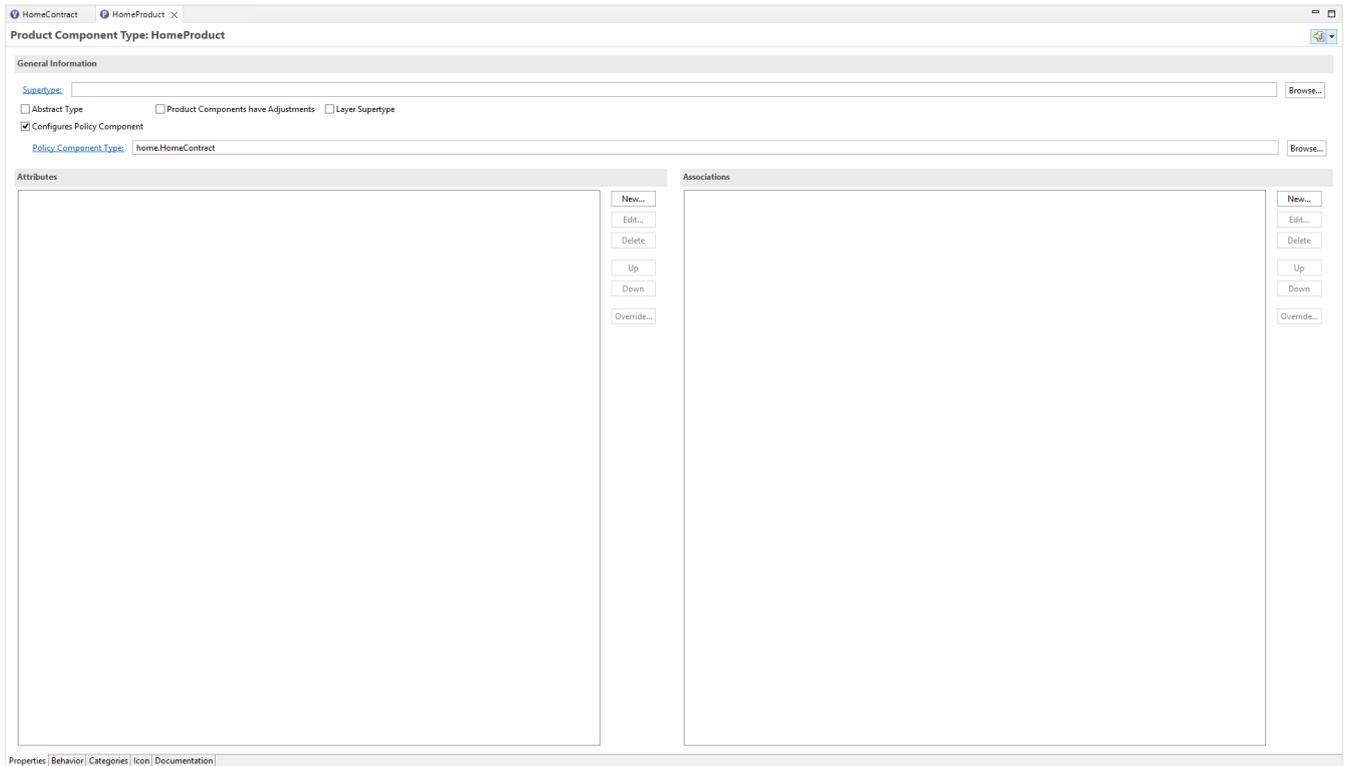


Figure 52. Editor for Product Classes

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 [9].

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

At the same time, Faktor-IPS has generated the implementation class "HomeProduct".

The following aspects should be configurable in the `HomeProduct` class

- the product name
- 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 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.

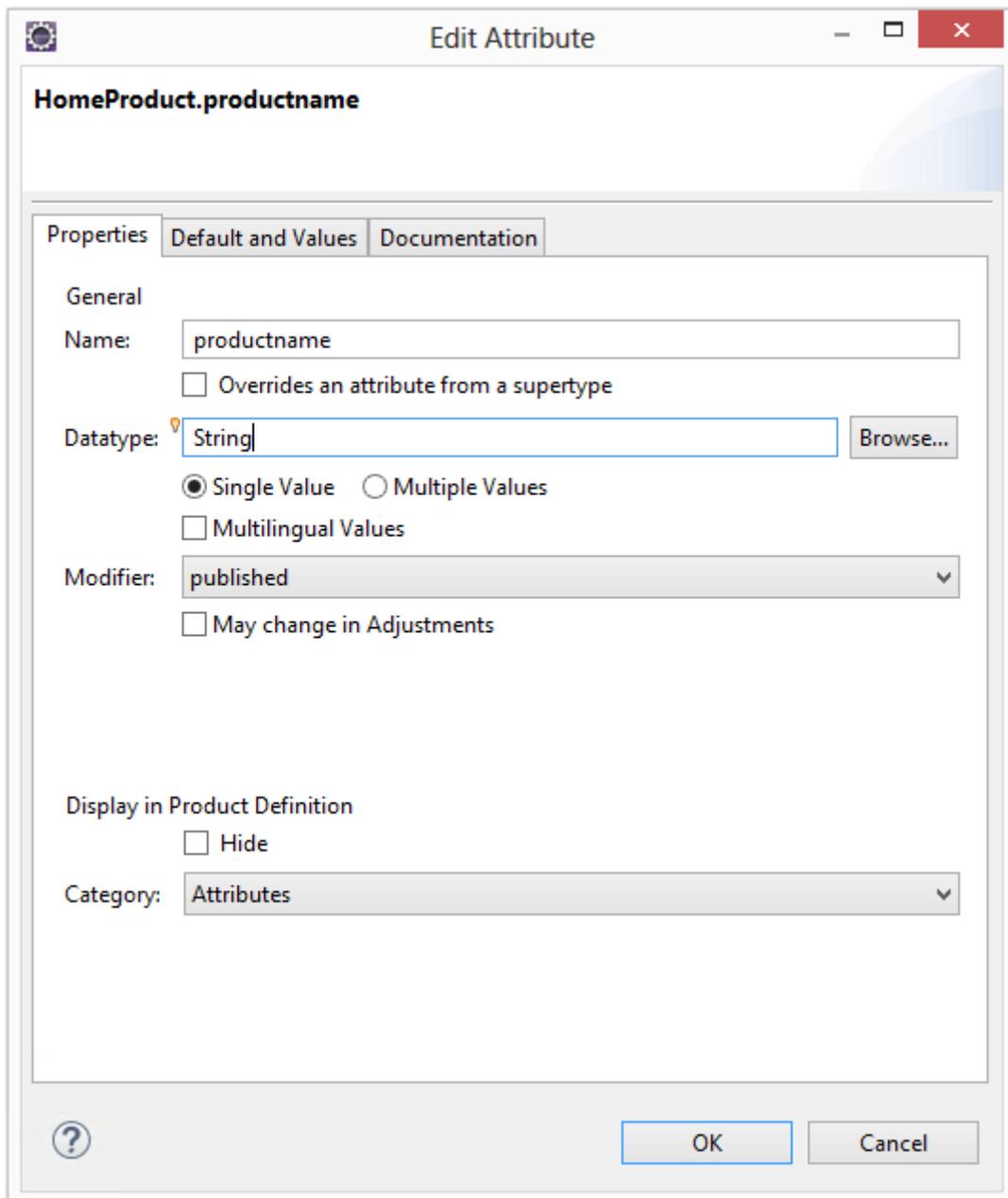


Figure 53. Dialog for Editing Product Attributes

Now, we will define that the allowable modes of payment for a HomeContract and the default payment mode 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.

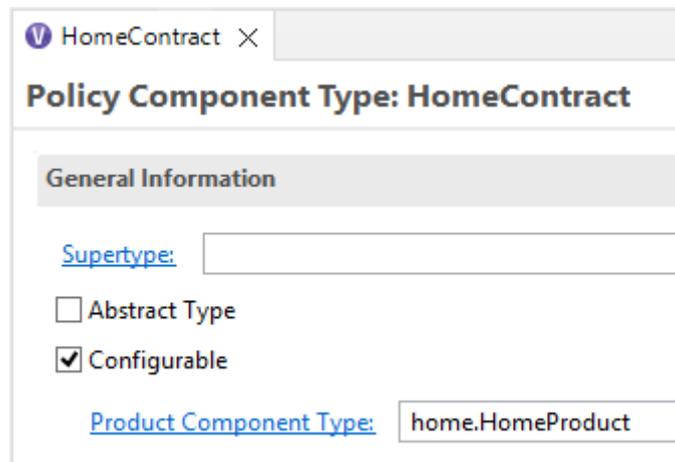


Figure 54. The Editors General Information Section for the HomeContract Class

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

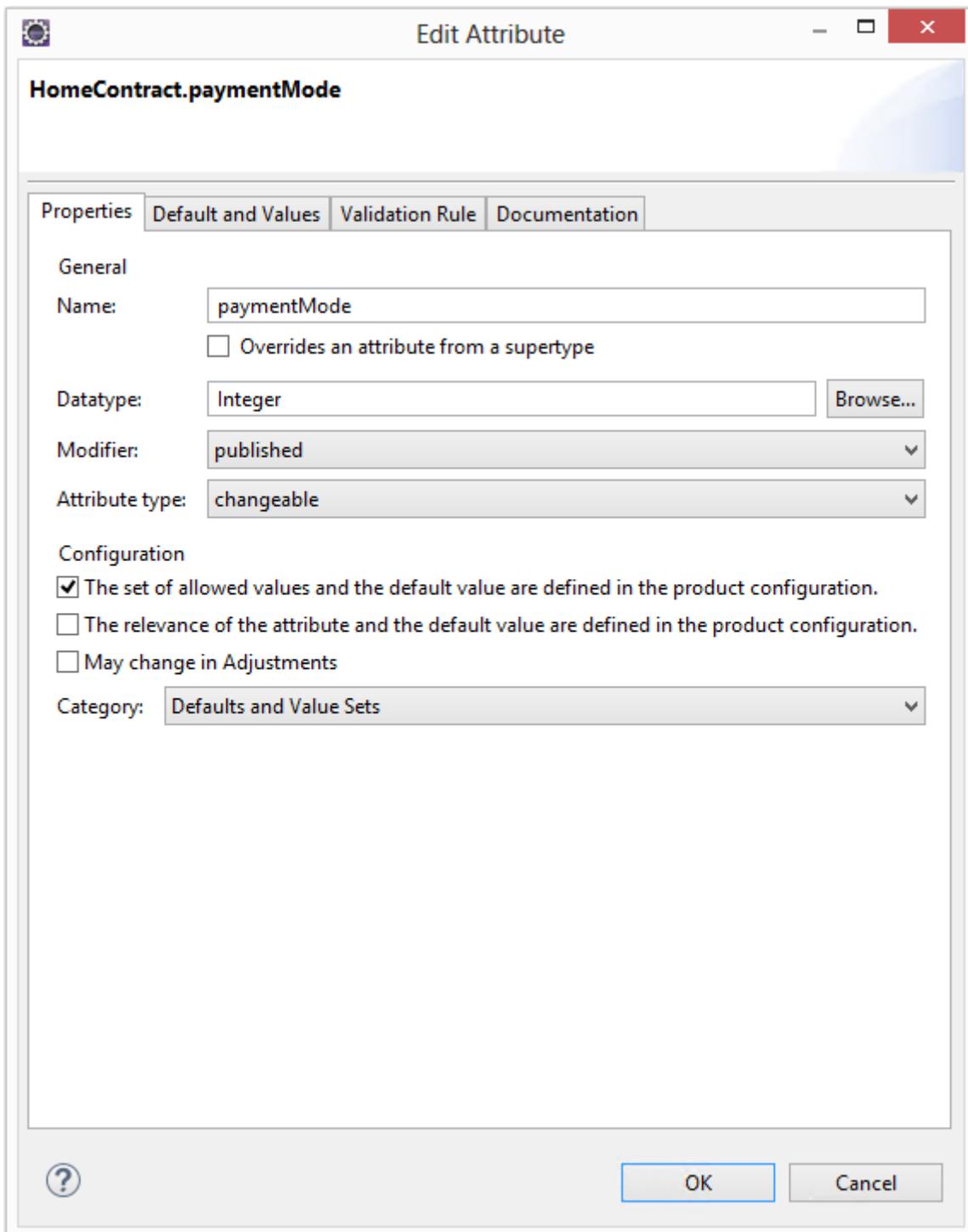


Figure 55. Dialog for a Configurable Contract Class Attribute

Note that the checkbox for "May change in Adjustments" is not checked.

Let us now have a look at the source code. The class `HomeProduct` now contains methods to retrieve the product name, the default payment mode, and the allowable values for payment mode, respectively.

```
/**
 * Returns the value of productname.
 *
 * @generated
 */
```

```

@IpsAttribute(name = "productname", kind = AttributeKind.CONSTANT, valueSetKind =
ValueSetKind.AllValues)
@IpsGenerated
public String getProductname() {
    return productname;
}

/**
 * Returns the default value for paymentMode.
 *
 * @generated
 */
@IpsDefaultValue("paymentMode")
@IpsGenerated
public Integer getDefaultValuePaymentMode() {
    return defaultValuePaymentMode;
}

/**
 * Returns the set of allowed values for the property paymentMode.
 *
 * @generated
 */
@IpsAllowedValues("paymentMode")
@IpsGenerated
public OrderedValueSet<Integer> getAllowedValuesForPaymentMode(IValidationContext
context) {
    return allowedValuesForPaymentMode;
}

```

In the class `HomeContract` exist methods to access the `HomeProduct`.

```

/**
 * Returns the HomeProduct that configures this object.
 *
 * @generated
 */
@IpsGenerated
public HomeProduct getHomeProduct() {
    return (HomeProduct) getProductComponent();
}

/**
 * Sets the new HomeProduct that configures this object.
 *
 * @param homeProduct          The new HomeProduct.
 * @param initPropertiesWithConfiguredDefaults <code>true</code> if the
 *                                     properties should be
 *                                     initialized with the defaults
 *                                     defined in the HomeProduct.
 *
 *
 *
 *

```

```

*
* @generated
*/
@IpsGenerated
public void setHomeProduct(HomeProduct homeProduct, boolean
initPropertiesWithConfiguratedDefaults) {
    setProductComponent(homeProduct);
    if (initPropertiesWithConfiguratedDefaults) {
        initialize();
    }
}
}

```

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 [Extending the Home Contens Model](#), we implemented the `getProposedSumInsured()` method of the `HomeContract` class like this:

```

/**
 * Returns the proposedSumInsured.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "proposedSumInsured", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getProposedSumInsured() {
    // begin-user-code
    // TODO: later we'll implement this with a product data lookup
    return Money.euro(650).multiply(livingSpace);
    // end-user-code
}

```

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 `HomeProduct` 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:

```

/**
 * Returns the proposedSumInsured.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "proposedSumInsured", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)

```

```

@IpsGenerated
public Money getProposedSumInsured() {
    // begin-user-code
    HomeProduct prod= getHomeProduct();
    if(prod==null) {
        return Money.NULL;
    }
    return prod.getProposedSumInsuredPerSqM().multiply(livingSpace);
    // end-user-code
}

```

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 `HomeBaseCoverageType`. For this class, you define an attribute named "name" of type `String`. In the course of this tutorial, we will further extend this class.

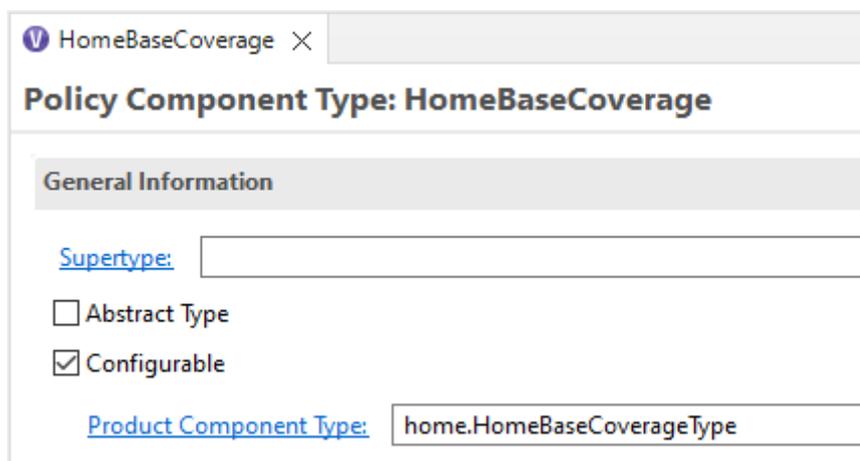


Figure 56. `HomeBaseCoverage` is configured by `HomeBaseCoverageType`

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:

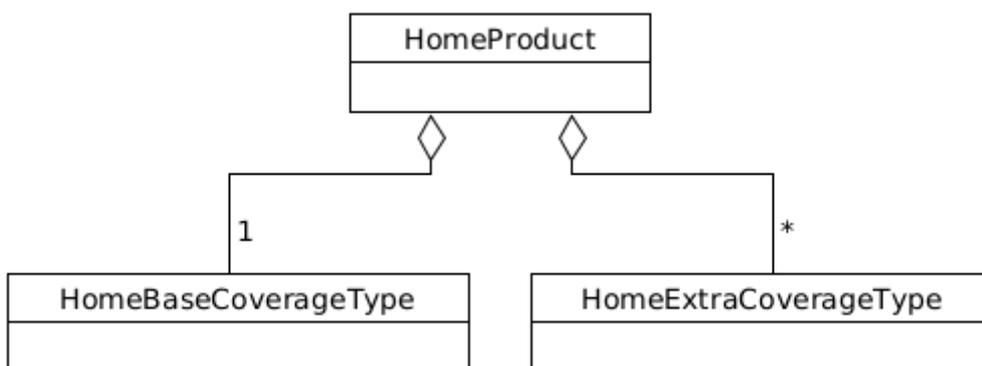


Figure 57. Model of the Product Configuration Classes

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 `HomeBaseCoverageType` 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.

The image shows a software dialog box titled "Edit Association" with a subtitle "HomeProduct.HomeBaseCoverageType". It has three tabs: "Properties", "Policy Association", and "Documentation". The "Properties" tab is active and contains the following fields and options:

- General**
- Target:** A text box containing "home.HomeBaseCoverageType" and a "Browse..." button.
- Type:** A dropdown menu set to "Aggregation".
- Change over time:** A checkbox labeled "May change in Adjustments" which is unchecked.
- Target Role Singular:** A text box containing "HomeBaseCoverageType".
- Target Role Plural:** An empty text box.
- Min Cardinality:** A text box containing "1".
- Max Cardinality:** A text box containing "1".
- Display in Product Definition:** A checkbox labeled "Hide" which is unchecked.
- Override / Derived Union:** Three checkboxes, all unchecked:
 - Overrides an association of a supertype
 - This association is a derived union
 - This association defines a subset of a derived union
- Derived union:** An empty text box.

At the bottom of the dialog, there is a help icon (question mark in a circle) on the left, and "OK" and "Cancel" buttons on the right.

Figure 58. 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, use the Faktor-IPS archetype on the command line.

```
mvn archetype:generate -DarchetypeGroupId=org.faktorips
-DarchetypeArtifactId=faktorips-maven-archetype -DarchetypeVersion=25.1.1.release
-DgroupId=org.faktorips.tutorial -DartifactId=HomeProducts -Dversion=1.0
-Dpackage=org.faktorips.tutorial.productdata -DjavaVersion=17 -DIPS-Language=en -DIPS
-IpsModelProject=false -DIPS-IpsProductDefinitionProject=true -DIPS
-SourceFolder=productdata -DIPS-RuntimeIdPrefix=home. -DIPS-ConfigureIPSBUILD=true
```

This time, the chosen type is *product definition project* (`-DIPS-IsProductDefinitionProject=true`), the package name `org.faktorips.tutorial.productdata` and the runtime-ID prefix `home.`. 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 [10]. 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.

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

The newly created project must be imported into the Eclipse workspace. Click *File* ► *Import* ► *Maven* ► *Existing Maven Projects*. In the dialog add the project folder as the root directory and import the project by clicking on *Finish*.

Faktor-IPS generates Java source files and copies XML files, which are 100% generated, into the directory `src/main/resources`. The contents of the directory can therefore be deleted and recreated at any time. Since the source code for formulas is also generated in this folder in the course of this tutorial, Maven must be instructed to also build the `src/main/resources` folder. To do this, the following code must be added to the `pom.xml` file under `<plugins>`:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>3.6.0</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```

        <configuration>
            <sources>
                <source>${project.basedir}/src/main/resources</source>
            </sources>
        </configuration>
    </execution>
</executions>
</plugin>

```

We will manage the product data in a separate project to account for the fact that the responsibility for the 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. In order to make the `HomeModel` Java classes available to the project, you have to add a dependency to the project `HomeModel` to the `pom.xml` of the project `HomeProducts`. Simply add the following entry under `<dependencies>` in the `pom.xml` file:

```

<dependency>
    <groupId>org.faktorips.tutorial</groupId>
    <artifactId>HomeModel</artifactId>
    <version>1.0</version>
</dependency>

```

Then the JUnit 5 libraries have to be added to the project, as we will write a test class later. To do this, add the following dependencies to the `pom.xml` file:

```

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.11.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.11.3</version>
    <scope>test</scope>
</dependency>

```

If, as suggested in the archetype, Java 17 was used, the following dependency must be added:

```

<dependency>
    <groupId>jakarta.xml.bind</groupId>

```

```
<artifactId>jakarta.xml.bind-api</artifactId>
<version>4.0.0</version>
</dependency>
```

In order for the changes in the `pom.xml` file to take effect, right-click on the `HomeProduct` project and select `Maven ► Update Project ► Ok`.

As a first step, open the product definition perspective by clicking `Window ► Perspective ► Open Perspective ► Other ► Product Definition [11]`. 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.).

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

Initially, we will create two IPS packages; one for the products (`products`) and one for the coverages (`coverages`). As in the Java perspective, this is done either via the dropdown menu or the toolbar (select "HomeProducts" as source folder).

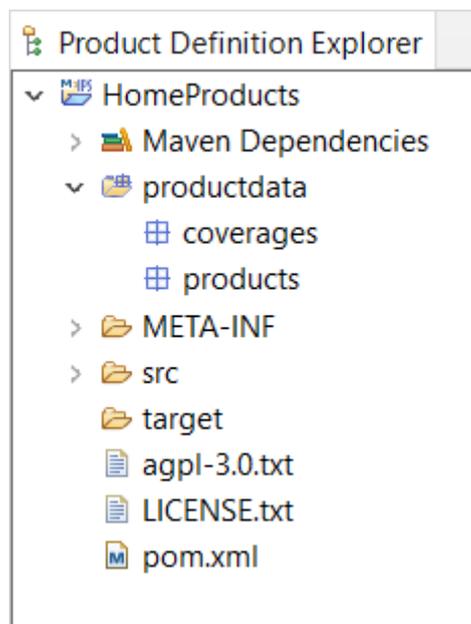


Figure 59. Creating IPS Packages

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

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

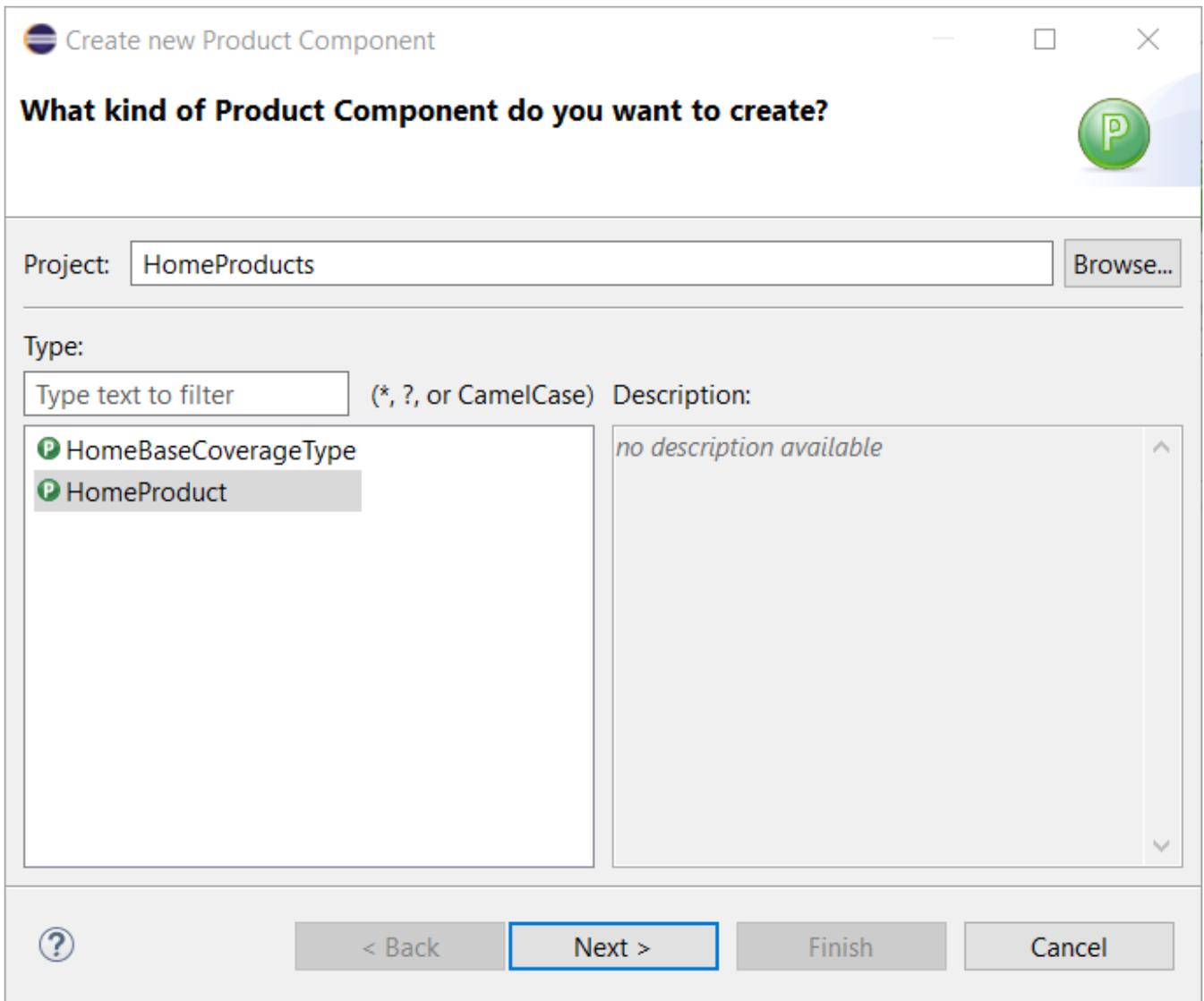


Figure 60. Selecting the product component type

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.

Here you enter the name of the product component. When you click *Finish*, the product component will be created in the file system.

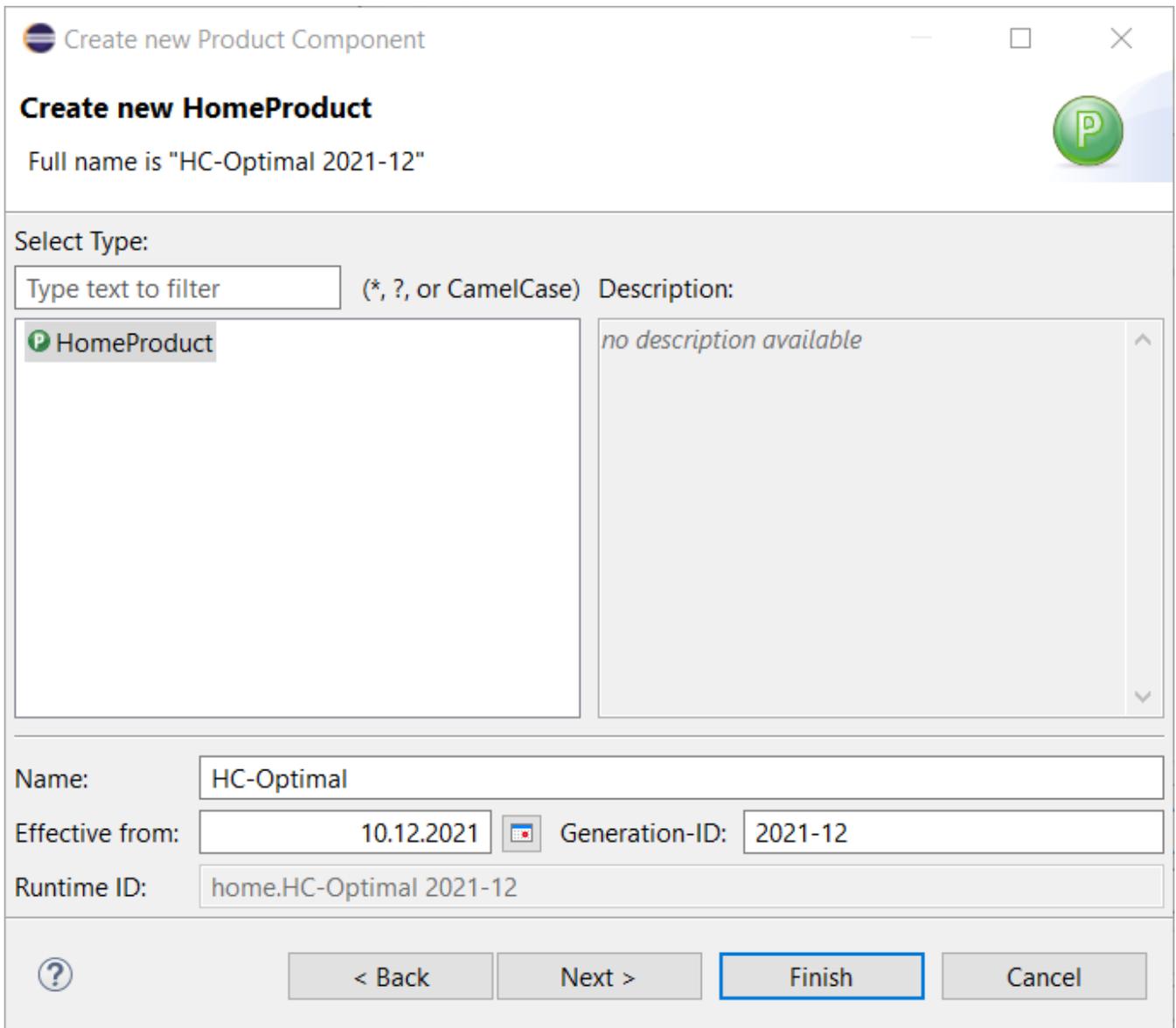


Figure 61. Creating a New Product

By double clicking on the product component in the product structure explorer the editor for the component will open.

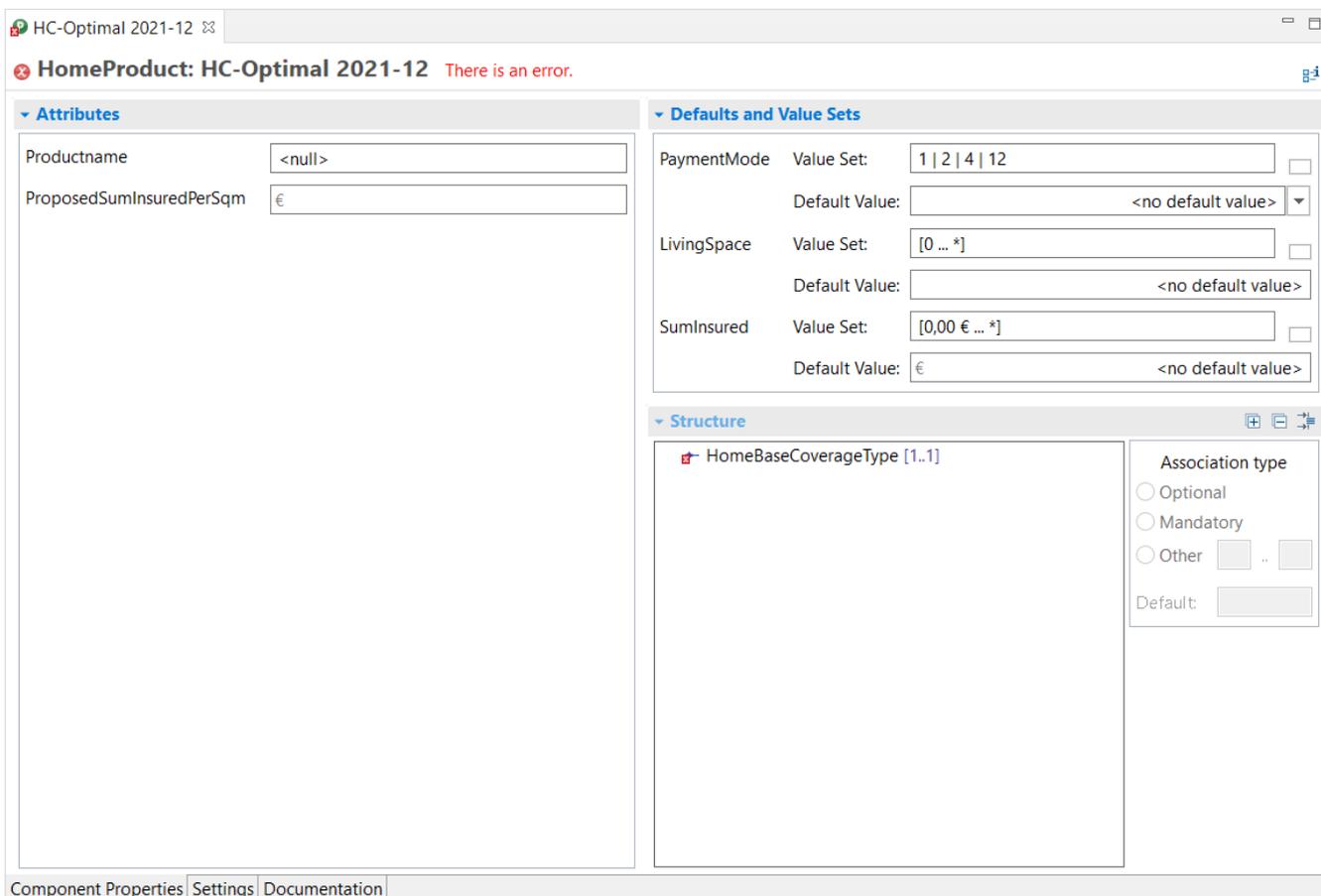


Figure 62. The Product Component Editor for HC-Optimal

The first editor page shows:

- *Attributes*
Contains the properties of the product. Any attributes defined in the product class will be listed here.
- *Defaults and Value Sets*
Contains defaults and value ranges for the different policy attributes.
- *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	900 EUR
Default payment mode	1 (annual)
Allowed payment modes	1, 2, 4, 12
Default living space	<no default value>
Allowed living space	0-2000
Default sum insured	<no default value>
Sum insured	10000 EUR - 5000000 EUR

When creating the product component, you received the following error message:

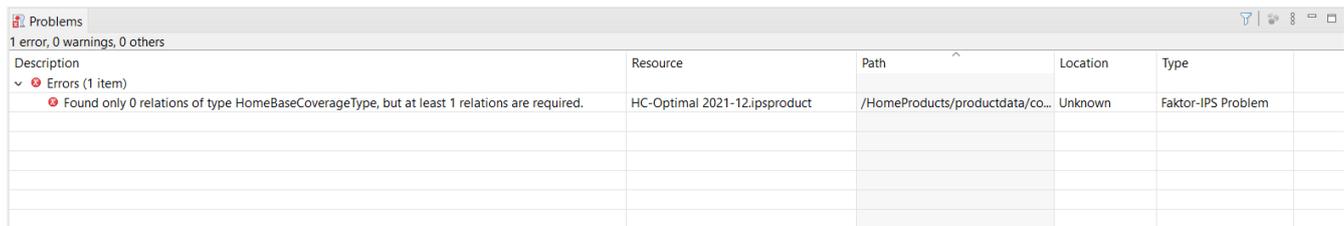


Figure 63. Error message when creating the product component

To fix this, we will now 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.

Under *Association type* we must select **Mandatory**, because it is a 1..1(1) relation. (The reason is the defined association between **HomeProduct** and **HomeBaseCoverageType**.)

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** and **BaseCoverage-Compact**.

You can now open the new product **HC-Compact** and enter its data:

Configuration property	HC-Compact
Product name	Home Contents Compact
Proposed sum insured per sqm of living space	600 EUR
Default payment mode	1 (annual)
Allowed payment modes	1, 2
Default living space	<no default value>
Allowed living space	0-1000
Default sum insured	<no default value>
Sum insured	10000 EUR - 2000000 EUR

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

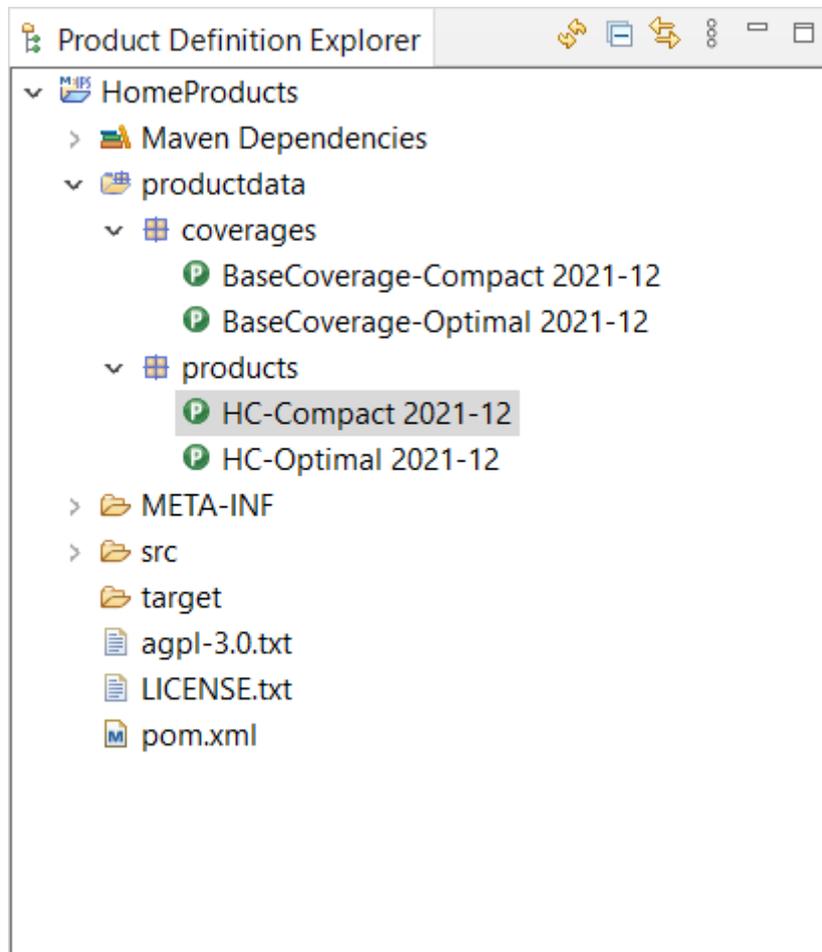


Figure 64. The Products are displayed in the Product Definition Explorer

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 `src/main/resources`. 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`.

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.

To get a product component, invoke the `getProductComponent(...)` method that gets the `RuntimeId` of the component passed 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. Create the source folder `src/test/java` 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.

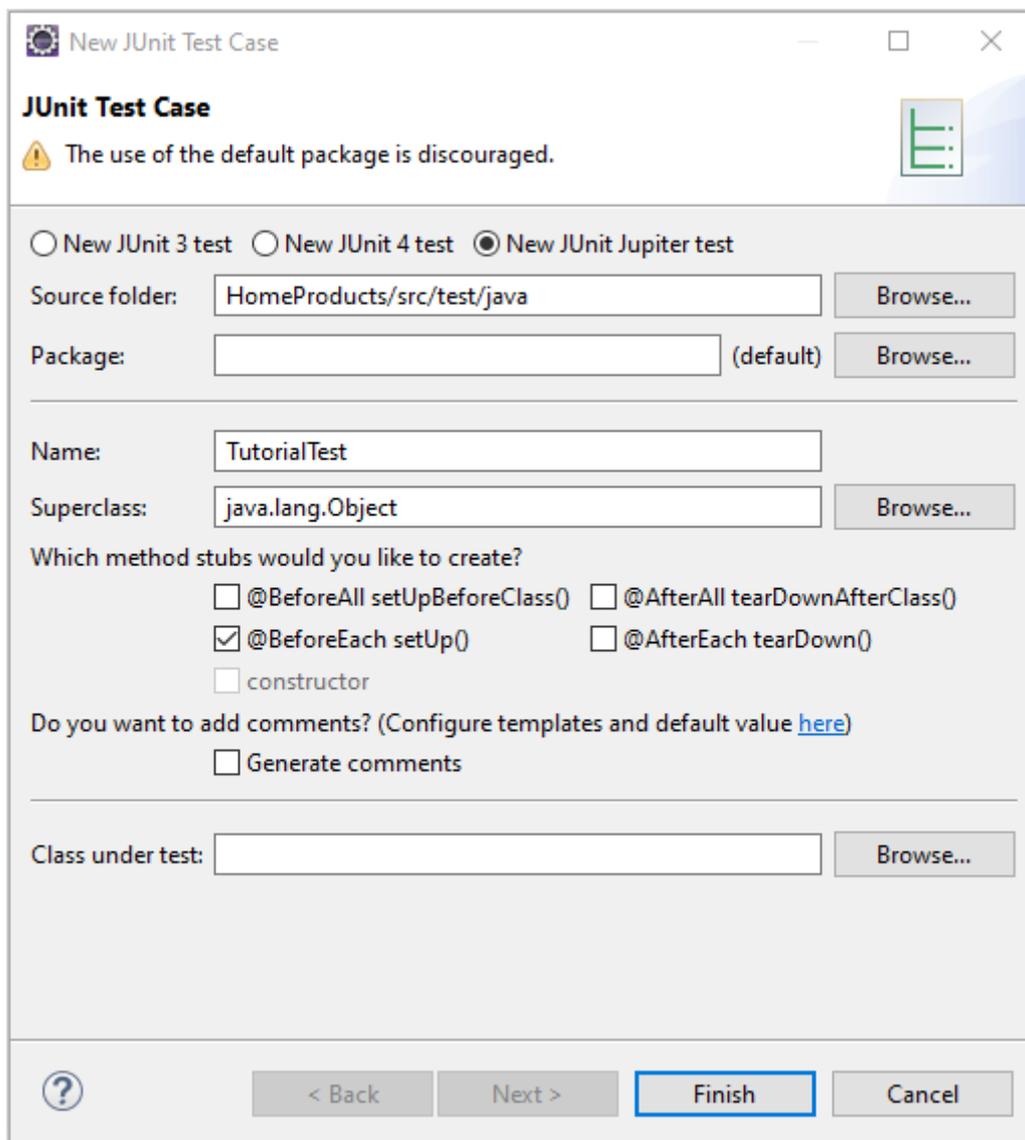


Figure 65. Wizard for creating a JUnit test

Instead of carrying out checks with assert statements, we will just print the results on the console with `println`. Do not forget to adjust the year in your class name if it is not 2019-07

```
public class TutorialTest {

    private IRuntimeRepository repository;
    private HomeProduct compactProduct;

    @BeforeEach
    public void setUp() throws Exception {
        // Repository erzeugen
        repository = ClassloaderRuntimeRepository
            .create("org/faktorips/tutorial/productdata/internal/faktorips-
repository-toc.xml");

        // Referenz auf das compactProduct aus dem Repository holen
        IProductComponent pc = repository.getProductComponent("home.HC-Compact 2021-
12");
    }
}
```

```

        // Auf die eigenen Modellklassen casten
        compactProduct = (HomeProduct) pc;
    }

    @Test
    public void test() {
        System.out.println("Product name: " + compactProduct.getProductname());
        System.out.println("Proposed sum insured per sqm: " +
compactProduct.getProposedSumInsuredPerSqm());
        System.out.println("Default modes of payment: " +
compactProduct.getDefaultValuePaymentMode());
        System.out.println("Allowed modes of payment: " +
compactProduct.getAllowedValuesForPaymentMode());
        System.out.println("Default sum insured: " +
compactProduct.getDefaultValueSumInsured());
        System.out.println("Range sum insured: " +
compactProduct.getAllowedValuesForSumInsured());
        System.out.println("Default living space: " +
compactProduct.getDefaultValueLivingSpace());
        System.out.println("Range living space: " +
compactProduct.getAllowedValuesForLivingSpace());
    }
}

```

If you run the test now, it should print the following:

```

Product name: Home Contents Compact
Proposed sum insured per sqm: 600.00 EUR
Default modes of payment: 1
Allowed modes of payment: [1, 2]
Default sum insured: MoneyNull
Range sum insured: 10000.00 EUR-2000000.00 EUR
Default living space: null
Range living space: 0-1000

```

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.

Teil 2: Verwendung von Tabellen und Formeln

Überblick

Im ersten Teil wurde die Modellierung mit Faktor-IPS und die Konfiguration von Produkten anhand einer einfachen Hausratversicherung erläutert. Im zweiten Teil werden wir die Verwendung von Tabellen und Formeln erklären. Dazu erweitern wir das in Teil 1 erstellte Hausratmodell.

Die Kapitel gliedern sich wie folgt:

- **Verwendung von Tabellen**

In dem Kapitel wird das Modell zunächst um eine Tabelle zur Ermittlung der Tarifzone erweitert und der Zugriff auf den Tabelleninhalt realisiert. In einem zweiten Schritt werden produktspezifische Beitragstabellen definiert und der Zusammenhang zu den Produkten modelliert.

- **Implementierung der Beitragsberechnung**

In diesem Abschnitt wird die Beitragsberechnung implementiert und mit Hilfe eines JUnit-Tests getestet. Innerhalb der Beitragsberechnung wird auf die produktspezifischen Beitragstabellen zugegriffen.

- **Verwendung von Formeln**

In diesem Kapitel wird das Hausratmodell um Zusatzdeckungen erweitert. Die Modellierung erlaubt es der Fachabteilung flexibel neue Zusatzdeckungen z.B. gegen Fahrraddiebstahl oder Überspannungsschäden zu definieren, ohne dass jedes Mal das Modell erweitert werden muss. Erreicht wird dies durch den Einsatz von durch die Fachabteilung definierbaren Formeln.

Verwendung von Tabellen

In diesem Kapitel erweitern wir das Modell um Tabellen zur Abbildung der Tarifzonen und Beitragssätze und programmieren die Ermittlung der für einen Vertrag gültigen Tarifzone.

Tarifzontentabelle

Aufgrund der in Deutschland regional unterschiedlichen Schadenswahrscheinlichkeit durch Einbruchdiebstahl unterscheiden Versicherungsunternehmen in der Hausratversicherung zwischen unterschiedlichen Tarifzonen. Hierzu verwenden Versicherer eine Tabelle, mit der einem Postleitzahlenbereich eine Tarifzone zugeordnet ist. Im Tutorial nutzen wir folgende Tabelle die später noch benötigt wird:

Plz-von	Plz-bis	Tarifzone
17235	17237	II
30159	45549	III
59174	59199	IV

Plz-von	Plz-bis	Tarifzone
47051	47279	V
63065	63075	VI
...

Tarifzonentabelle

Für alle Postleitzahlen, die in keinen der Bereiche fallen, gilt die Tarifzone I.

Faktor-IPS unterscheidet zwischen der Definition der Tabellenstruktur und dem Tabelleninhalt. Die Tabellenstruktur wird als Teil des Modells angelegt. Der Tabelleninhalt kann abhängig vom Inhalt und der Verantwortung für die Pflege der Daten sowohl als Teil des Modells oder als Teil der Produktdefinition verwaltet werden. Zu einer Tabellenstruktur kann es dabei mehrere Tabelleninhalte geben. Dies entspricht dem Konzept von Tabellenpartitionen in relationalen Datenbankmanagementsystemen.

Legen wir zunächst die Tabellenstruktur für die Ermittlung der Tarifzonen an. Hierzu wechseln Sie zunächst zurück in die Java-Perspektive. Im Projekt Hausratmodell unter dem Ordner `model` markieren Sie den Ordner `hausrat` und klicken dann in der Toolbar auf . Die Tabellenstruktur nennen Sie `Tarifzonentabelle` und klicken Finish.

Als Tabellentyp wählen Sie den Typ Single Content, da es für diese Struktur nur einen Inhalt geben soll. Nun legen wir zunächst die Spalten der Tabelle an. Alle drei Spalten (plzVon, plzBis, tarifzone) sind vom Datentyp String.

Interessant wird es jetzt bei der Definition des Postleitzahlenbereiches: Die von uns angelegte Tabellenstruktur dient uns letztendlich dazu, die Funktion (im mathematischen Sinne) $\text{tarifzone} \rightarrow \text{plz}$ abzubilden. Allein mit der Spaltendefinition und einem möglichen UniqueKey ist diese Semantik allerdings nicht abbildbar. In Faktor-IPS gibt es aus diesem Grund die Möglichkeit zu modellieren, dass die Spalten (oder eine Spalte) einen Bereich darstellen. Legen Sie jetzt einen neuen Bereich an (Bereiche ► Neu). Da die Tabelle Von- und Bis-Spalten enthält, wählen Sie als Typ Two Column Range. Als `Parameter Name` geben Sie jetzt `plz` ein und ordnen noch die beiden Spalten plzVon und plzBis zu.

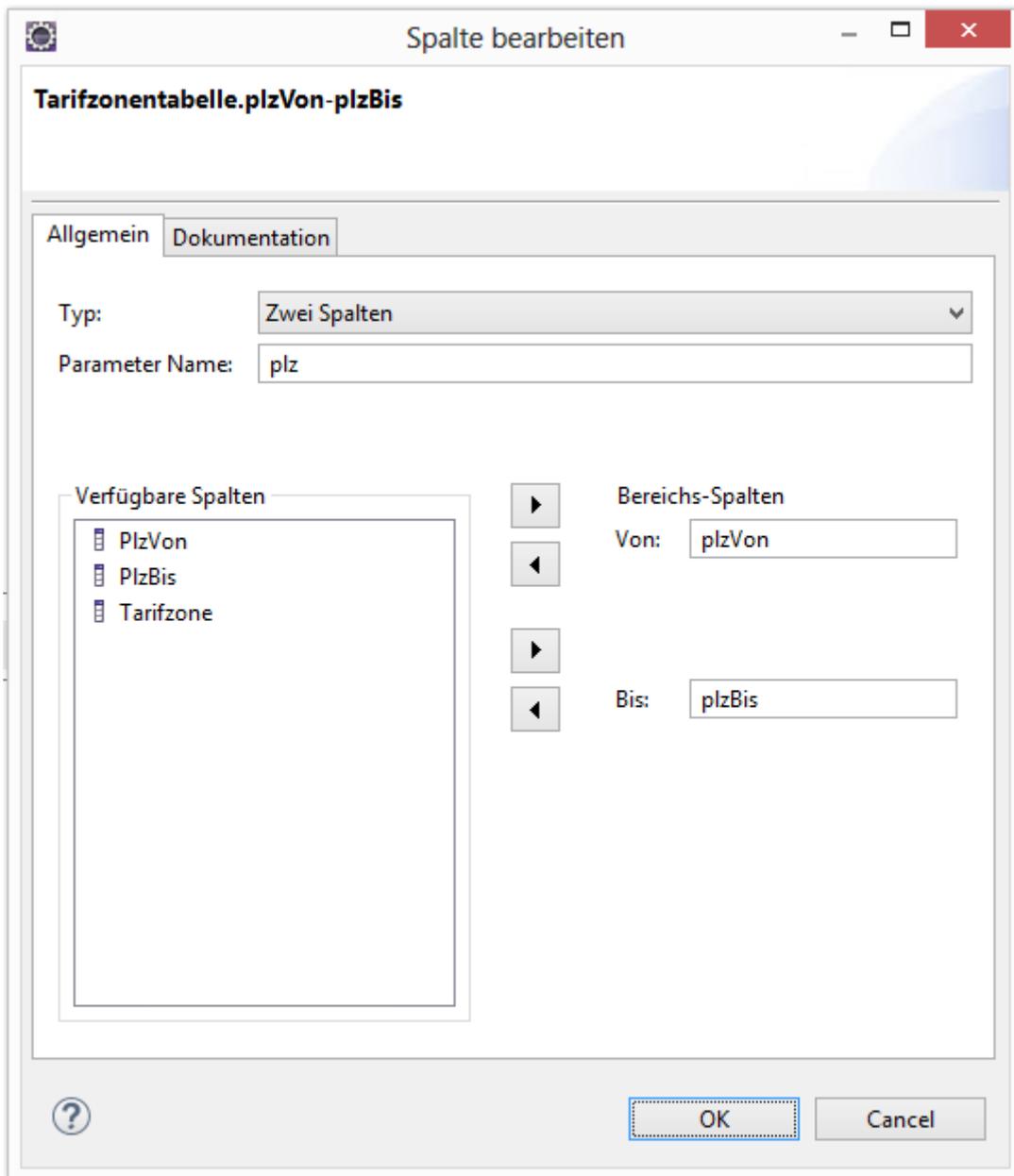


Figure 66. Bereich der Tabellenstruktur anlegen

Jetzt legen Sie noch einen neuen UniqueKey an (Indices ► Neu). Dem UniqueKey ordnen Sie jetzt **nicht** die einzelnen Spalten plzVon und plzBis zu, sondern den Bereich und speichern dann die Strukturbeschreibung.

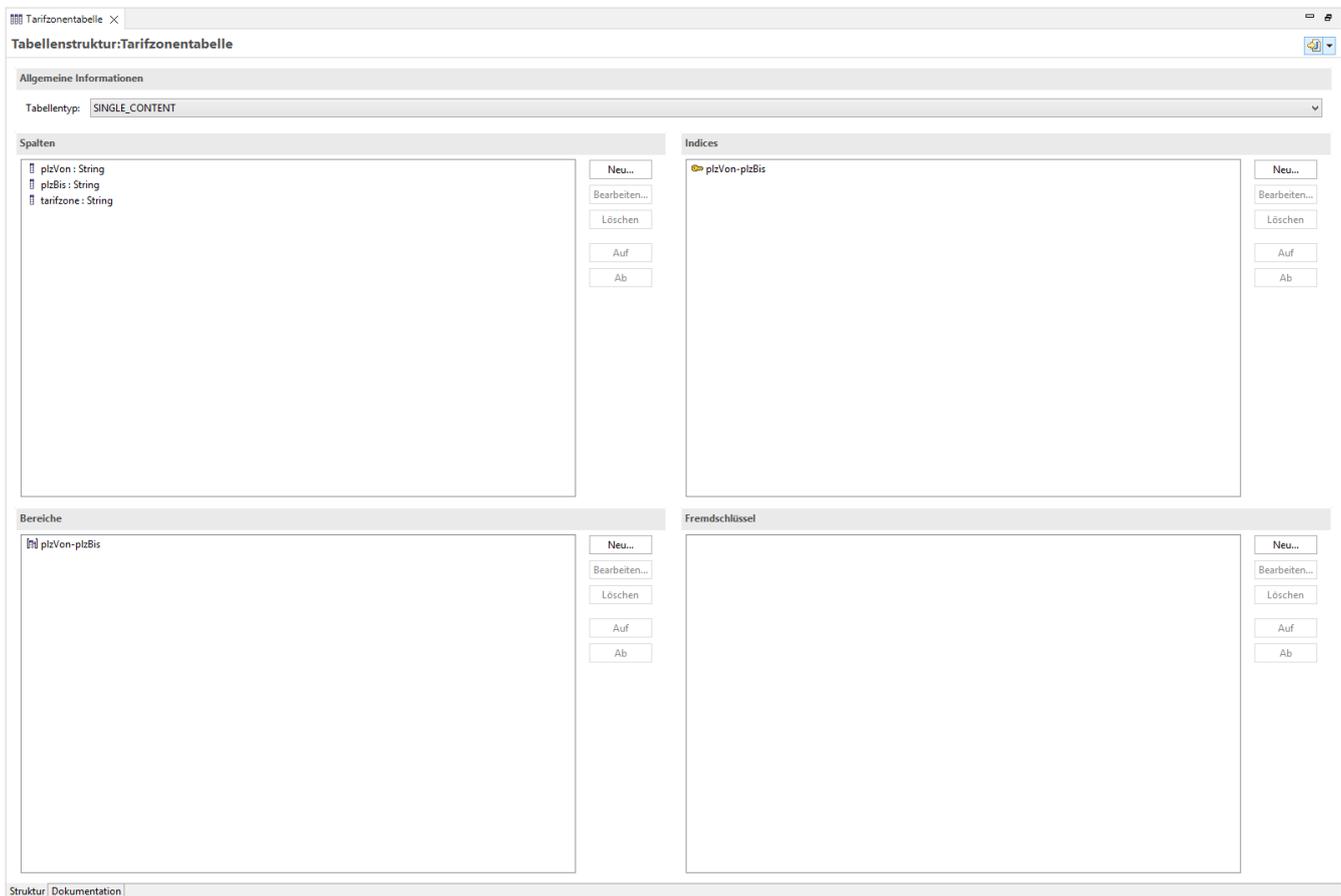


Figure 67. Tabellenstruktur Tarifzontabelle

Faktor-IPS hat nun für die Tabellenstruktur zwei neue Klassen im Source folder `src` unter dem Package `org.faktorips.tutorial.model.hausrat` generiert.

Die Klasse `TarifzontabelleRow` repräsentiert eine Zeile der Tabelle und enthält für jede Spalte eine Membervariable mit entsprechenden Zugriffsmethoden. Die Klasse `Tarifzontabelle` repräsentiert den Tabelleninhalt. Neben Methoden, um den Tabelleninhalt aus XML zu initialisieren, wurde aus dem UniqueKey eine Methode zum Suchen einer Zeile generiert:

```
public TarifzontabelleRow findRow(String plz) {
    // Details der Implementierung sind hier ausgelassen
}
```

Nutzen wir jetzt diese Klasse, um die Ermittlung der Tarifzone für den Hausratvertrag zu implementieren. Die Tarifzone ist eine abgeleitete Eigenschaft des Hausratvertrags und in der Klasse `HausratVertrag` gibt es somit die Methode `getTarifzone()`. Diese hatten wir bereits wie folgt implementiert:

```
public String getTarifzone() {
    // begin-user-code
    // TODO wird spaeter anhand einer Tarifzontabelle ermittelt
    return "I";
    // end-user-code
}
```

Nun ermitteln wir die Tarifzonen anhand der Postleitzahl aus der gerade angelegten Tabelle wie folgt:

```
public String getTarifzone() {
    // begin-user-code
    if (plz==null) {
        return null;
    }
    IRuntimeRepository repository = getHausratProdukt().getRepository();
    Tarifzontabelle tabelle = Tarifzontabelle.getInstance(repository);
    TarifzontabelleRow row = tabelle.findRow(plz);
    if (row==null) {
        return "I";
    }
    return row.getTarifzone();
    // end-user-code
}
```

Es bedarf an dieser Stelle noch einer Erläuterung, wie man an die Instanz der Tabelle herankommt. Da es zur Tarifzontabelle nur einen Inhalt gibt, hat die Klasse `Tarifzontabelle` eine `getInstance()` Methode, die diesen Inhalt zurück liefert. Als Parameter bekommt diese Methode das `RuntimeRepository`, welches zur Laufzeit Zugriff auf die Produktdaten inklusive der Tabelleninhalte gibt. An dieses kommen wir leicht über das Produkt, auf dem der Vertrag basiert [1].

1 Das Übergeben des `RuntimeRepositories` in die Methode `getInstance()` hat den Vorteil, dass das konkrete Repository in Testfällen leicht ausgetauscht werden kann.

Jetzt legen wir noch den Tabelleninhalt an. Die Zuordnung der Postleitzahlen zu den Tarifzonen soll von der Fachabteilung gepflegt werden. Zur Strukturierung fügen Sie noch ein neues IPS Package `Tabellen` in dem Projekt `Hausratprodukte` ein. Danach markieren Sie das neue Package und klicken in der Toolbar auf . Wählen Sie in dem Dialog die Tarifzontabelle als Struktur aus. Als Namen für den Tabelleninhalt übernehmen Sie den Namen `Tarifzontabelle` und klicken `Finish`. In dem Editor können Sie nun die oben beispielhaft aufgeführten Zeilen erfassen. Die Projektstruktur im Produktdefinitions-Explorer sollte danach wie folgt aussehen:

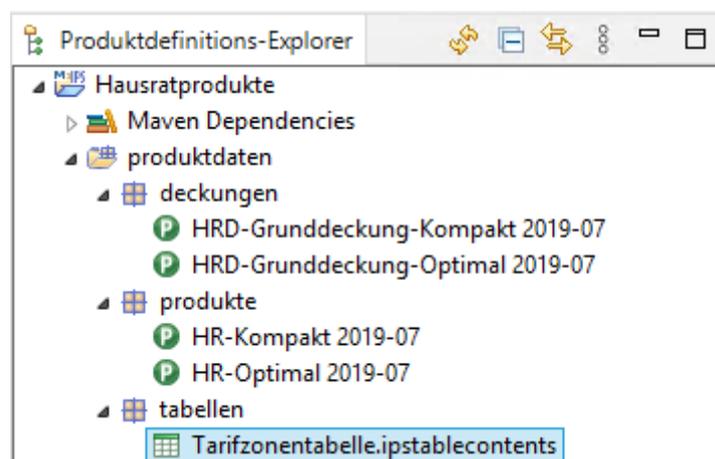


Figure 68. Projektstruktur der Produktdefinition

Zum Schluss testen wir noch die Ermittlung der Tarifzone. Hierzu erweitern wir den im ersten Teil des Tutorials angelegten JUnit Test „TutorialTest“ um die folgende Testmethode [2].

2 Im Tutorial *Softwaretests mit Faktor-IPS* wird u.a. beschrieben, wie man diesen Test mit Faktor-IPS Hilfsmitteln komfortabel erzeugen und durchführen kann.

```
@Test
public void testGetTarifzone() {
    // Erzeugen eines Hausratvertrags mit der Factorymethode des Produktes
    HausratVertrag vertrag = kompaktProdukt.createHausratVertrag();
    vertrag.setPlz("45525");
    assertEquals("III", vertrag.getTarifzone());
}
```

Beitragstabelle

Der Beitragssatz für die Grunddeckung der Hausratversicherung soll anhand der Tarifzone aus einer Tariftabelle ermittelt werden. Dabei sollen für die beiden Produkte unterschiedliche Beitragssätze gemäß den folgenden Tabellen gelten.

Tarifzone	Beitragssatz
I	0.80
II	1.00
III	1.44
IV	1.70
V	2.00
VI	2.20

Table 4. Beitragstabelle HR-Optimal

Tarifzone	Beitragssatz
I	0.60
II	0.80
III	1.21
IV	1.50
V	1.80
VI	2.00

Beitragstabelle HR-Kompakt

Die Daten für die unterschiedlichen Produkte werden häufig in einer Tabelle zusammengefasst, in der es dann noch eine Spalte „ProduktID“ gibt. In Faktor-IPS kann man aber auch zu einer Tabellenstruktur mehrere Inhalte anlegen und den Zusammenhang zu den Produktbausteinen explizit modellieren!

Legen Sie hierfür eine Tabellenstruktur *TariftabelleHausrat* mit den beiden Spalten Tarifzone (String) und Beitragssatz (Decimal) an. Definieren Sie einen UniqueKey auf die Spalte Tarifzone. Als Tabellentyp wählen Sie diesmal Multiple Contents aus, da wir für jedes Produkt einen eigenen Tabelleninhalt anlegen wollen.

Erzeugen Sie nun für die Produkte *HR-Optimal* und *HR-Kompakt* (bzw. genauer für deren Grunddeckungstypen) jeweils einen Tabelleninhalt mit dem Namen „Tariftabelle Optimal 2019-07“ und „Tariftabelle Kompakt 2019-07“ [3].

3 Die Endung „2019-07“ sollten Sie dabei entsprechend des von Ihnen verwendeten Wirksamkeitsdatums anpassen.

Das folgende Diagramm zeigt den Zusammenhang zwischen der Klasse *Grunddeckungstyp* und der Tabellenstruktur *TariftabelleHausrat* und die entsprechenden Objektinstanzen.

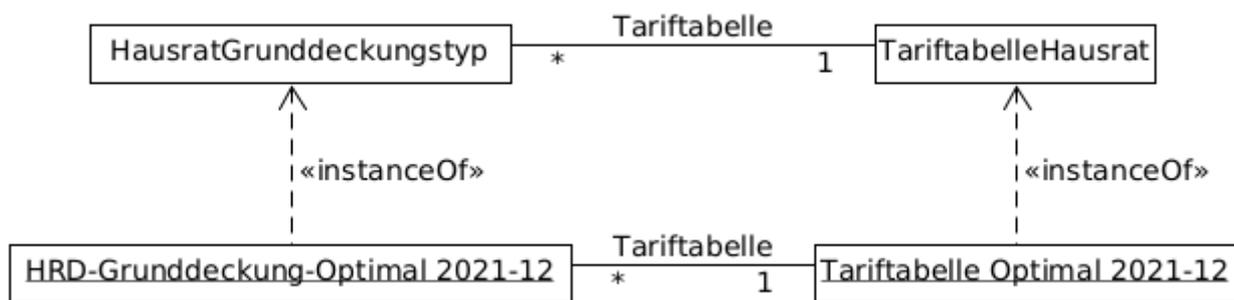


Figure 69. Zusammenhang Produktbausteine & Tabellen

Um den Zusammenhang zwischen Tabellen und Produkten in Faktor-IPS zu definieren, öffnen Sie die Klasse *HausratGrunddeckungstyp*. Auf der zweiten Seite „Verhalten“ im Editor [4] im Abschnitt *Table Usages (Verwendete Tabellen)* können Sie eine neue Tabellenverwendung definieren. Hierzu klicken Sie auf den *New Button* im Abschnitt „Verwendete Tabellen“. Geben Sie „tariftabelle“ als Rollennamen ein, wählen Sie „Tabelleninhalt erforderlich“ aus und fügen Sie „Tabellenstruktur“ *TariftabelleHausrat* hinzu, laut nachfolgender Abbildung:

4 Vorausgesetzt, Sie haben in den Preferences eingestellt, dass die Editoren zwei Abschnitte pro Seite verwenden sollen.

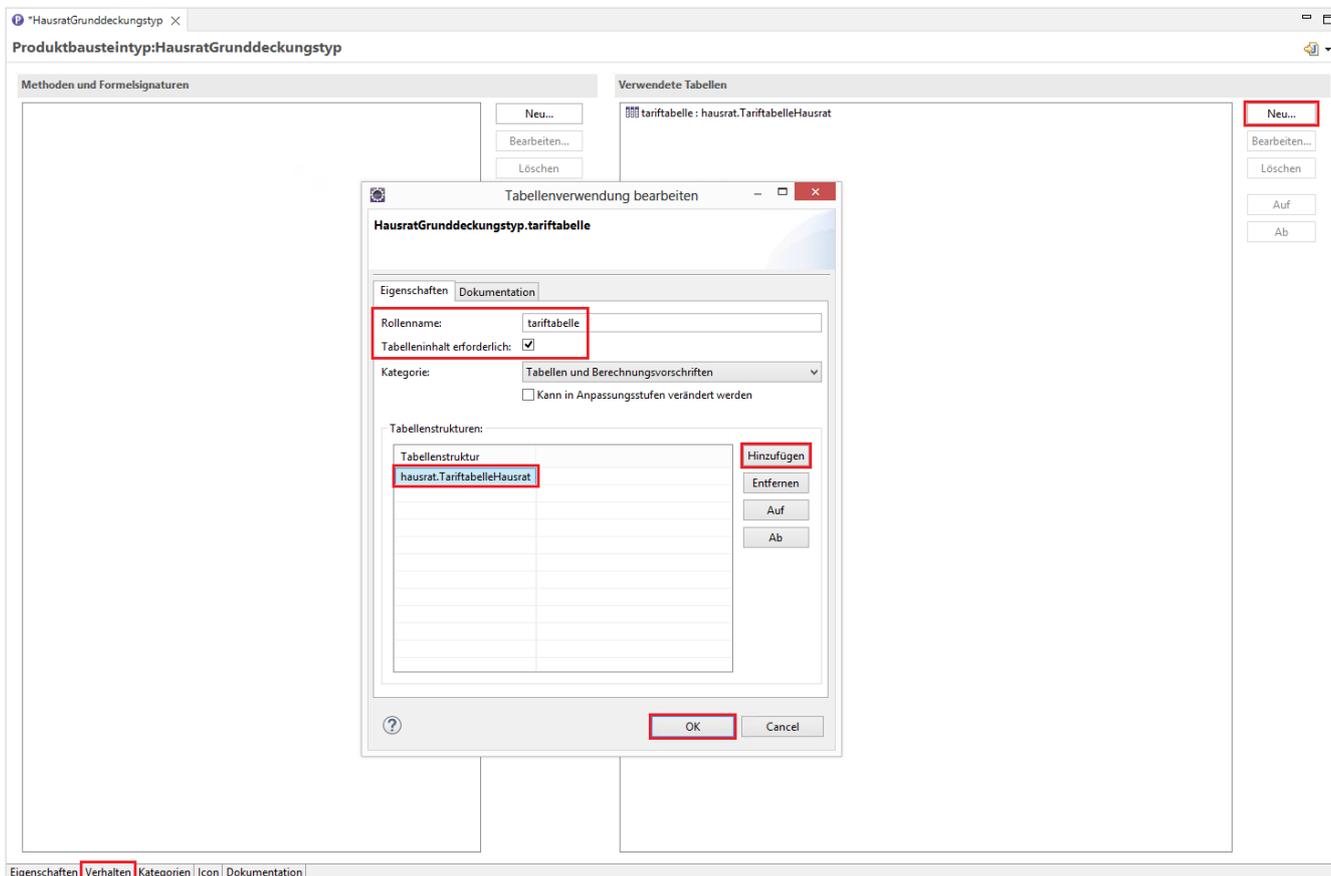


Figure 70. Modellierung der Verwendung von Tabellen

An dieser Stelle können Sie auch mehrere Tabellenstrukturen zuordnen, da im Laufe der Zeit z.B. neue Tarifierungsmerkmale hinzukommen und so unter der Rolle Tariftabelle unterschiedliche Tabellenstrukturen möglich sein können. Haken Sie noch die Checkbox *Table content required* an, da für jeden Grunddeckungstypen eine Tariftabelle angegeben werden muss, dann schließen Sie den Dialog und speichern.

Nun können wir die Tabelleninhalte den Grunddeckungstypen zuordnen. Öffnen Sie zunächst *HRD-Grunddeckung-Kompakt 2019-07*. Den Dialog, der Sie darauf hinweist, dass die Tariftabelle noch nicht zugeordnet ist, bestätigen Sie mit *Fix*. In dem Abschnitt *Tabellen und Berechnungsvorschriften* können Sie nun die Tariftabelle für HR-Kompakt zuordnen und dann speichern.

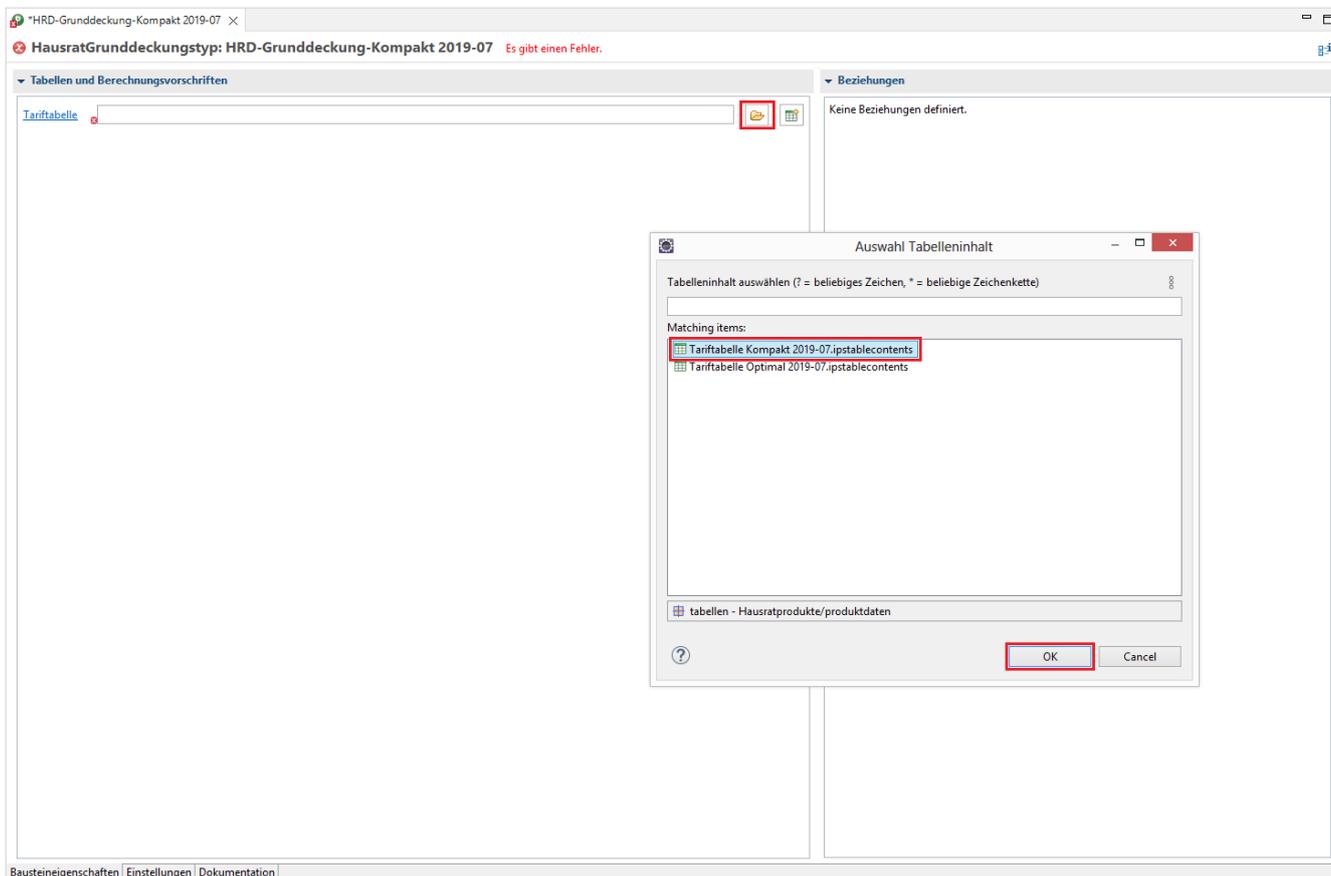


Figure 71. Zuordnung der Tabelleninhalte

Analog verfahren Sie für HR-Optimal.

Zum Schluss dieses Kapitels werfen wir noch einen Blick auf den generierten Sourcecode. In der Klasse *HausratGrunddeckungstyp* gibt es eine Methode, um den zugeordneten Tabelleninhalt zu erhalten:

```
public TariftabelleHausrat getTariftabelle() {
    if (tariftabelleName == null) {
        return null;
    }
    return (TariftabelleHausrat) getRepository().getTable(tariftabelleName);
}
```

Da auch die Findermethoden an der Tabelle generiert sind, lässt sich so mit sehr wenigen Zeilen Sourcecode ein effizienter Zugriff auf eine Tabelle realisieren.

Implementieren der Beitragsberechnung

In diesem Kapitel werden wir nun die Beitragsberechnung für unsere Hausratprodukte implementieren.

Wir erweitern unser Modell um die Methoden gemäß der folgenden Abbildung.

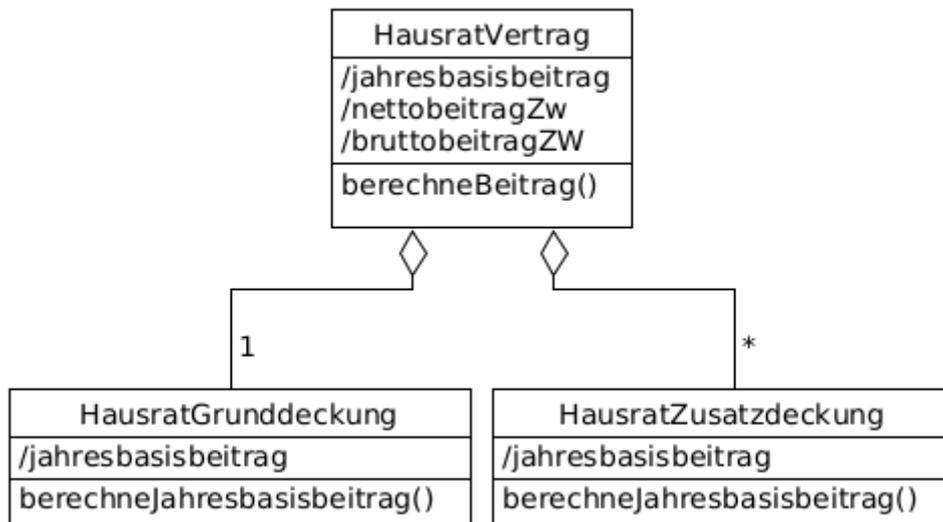


Figure 72. Berücksichtigung der Beitragsberechnung im Modell

Jede Deckung hat einen Jahresbasisbeitrag (/jahresbasisbeitrag). Der Jahresbasisbeitrag des Vertrags ist die Summe über die Jahresbasisbeiträge seiner Deckungen.

Der Nettobeitrag je Zahlungsperiode (/nettobeitragZw) ist der vom Beitragszahler zu zahlende Beitrag ohne Versicherungssteuer. Er ergibt sich aus dem Jahresbasisbeitrag dividiert durch Anzahl der Zahlungsperioden und multipliziert mit 1+Höhe des Ratenzahlungszuschlages.

Der Bruttobeitrag je Zahlungsweise (/bruttobeitragZW) ist der vom Beitragszahler zu zahlende Bruttobeitrag pro Zahlung. Er ergibt sich aus der Multiplikation des NettobeitragZw mit 1+Versicherungssteuersatz. In dem Tutorial verwenden wir Ratenzahlungszuschlag in Höhe von 3% und Versicherungssteuer von 19%.

Legen Sie nun die neuen Attribute in den Klassen *HausratVertrag* und *HausratGrunddeckung* an. Um die *HausratZusatzdeckungen* kümmern wir uns im nächsten Kapitel. Alle Attribute sind abgeleitet (cached, Berechnung durch expliziten Methodenaufruf) und vom Datentyp Money. Da es sich um ein gecachetes, abgeleitetes Attribut handelt, generiert Faktor-IPS je eine Membervariable und eine Gettermethode.

Die Berechnung aller Beitragsattribute erfolgt durch die Methode `berechneBeitrag()` der Klasse *HausratVertrag*. Die Methode berechnet dabei alle Beitragsattribute des Vertrags und auch den Jahresbasisbeitrag der Deckungen. Hierzu verwendet sie natürlich die Methode `berechneJahresbasisbeitrag()` der Deckungen.

Als nächstes legen wir diese Methoden an. Öffnen Sie die Klasse „HausratVertrag“, im Editor wechseln Sie auf die zweite Seite „Verhalten“ (analog wie in Abbildung 5). Im Bereich „Methoden“ drücken Sie auf Button „Neu“, geben den Namen für die Methode ein und drücken auf Button „OK“. Die folgende Abbildung zeigt den Dialog zum Bearbeiten einer Methodensignatur.

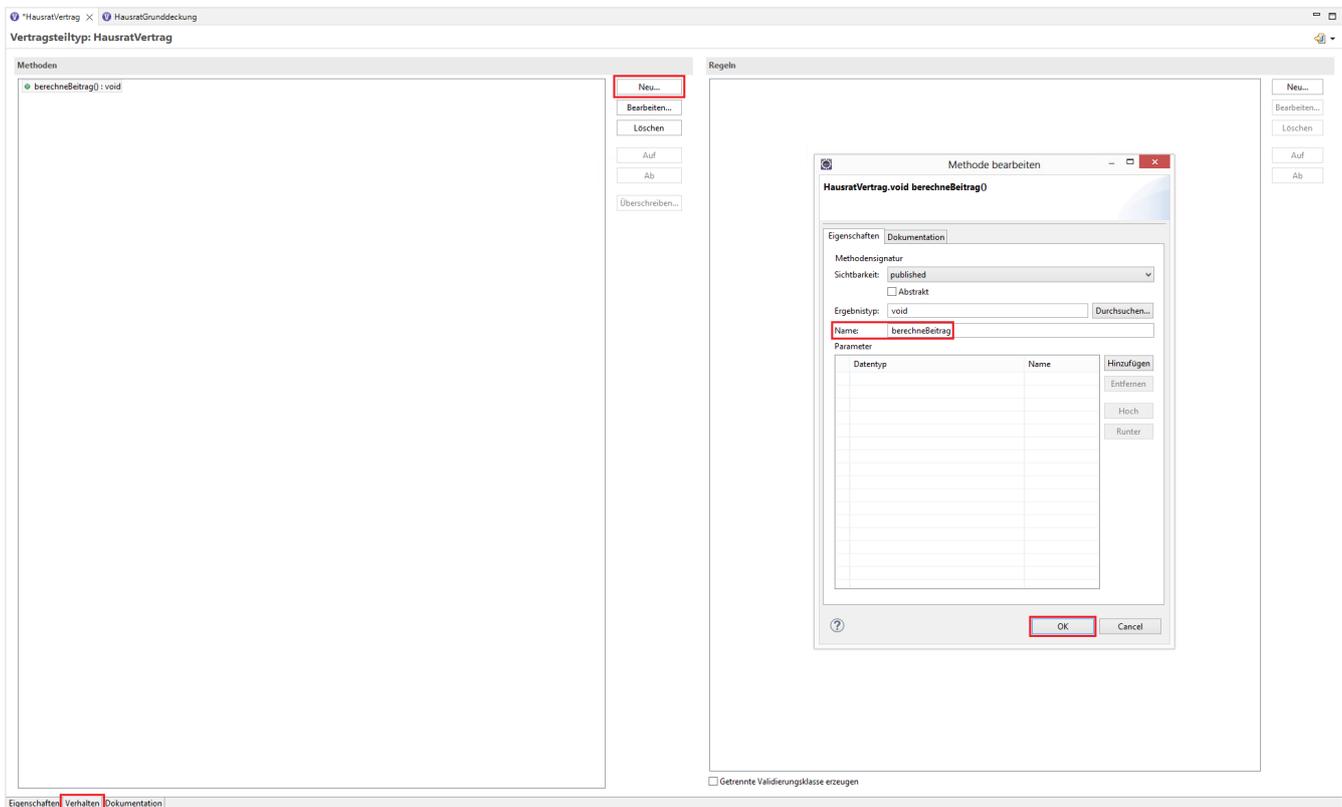


Figure 73. Dialog zum Bearbeiten einer Methodensignatur

Im Vergleich zur Modellierung von Beziehungen und Attributen bietet die Codegenerierung für Methoden weniger Vorteile. Methoden können daher natürlich auch direkt im Sourcecode definiert werden.

Der folgende Sourcecodeausschnitt zeigt die Implementierung der spartenübergreifenden Beitragsberechnung in der Klasse `HausratVertrag`. Aus Übersichtlichkeitsgründen implementieren wir die Berechnung von Jahresbasisbeitrag und NettobeitragZw in zwei eigenen privaten Methoden, die wir direkt in den Sourcecode schreiben, ohne sie ins Modell aufzunehmen.

```
/**
 * @generated NOT
 */
@IpsGenerated
public void berechneBeitrag() {
    berechneJahresbasisbeitrag();
    berechneNettobeitragZw();
    Decimal versSteuerFaktor = Decimal.valueOf(119, 2);
    // 1+Versicherungssteuersatz=1.19 (119 Prozent)
    bruttobeitragZw = nettobeitragZw.multiply(versSteuerFaktor, RoundingMode.HALF_UP);
}

private void berechneJahresbasisbeitrag() {
    jahresbasisbeitrag = Money.euro(0, 0);
    HausratGrunddeckung hausratGrunddeckung = getHausratGrunddeckung();

    hausratGrunddeckung.berechneJahresbasisbeitrag();
}
```

```

    jahresbasisbeitrag =
jahresbasisbeitrag.add(hausratGrunddeckung.getJahresbasisbeitrag());
    /*
     * TODO: wenn das Modell um Zusatzdeckungen erweitert wird, muss deren
     * Beitrag an dieser Stelle natürlich auch hinzuaddiert werden
     */
}

private void berechneNettobeitragZw() {
    if (zahlweise == null) {
        nettobeitragZw = Money.NULL;
        return;
    }
    if (zahlweise.intValue() == 1) {
        nettobeitragZw = jahresbasisbeitrag;
    } else {
        Decimal rzFaktor = Decimal.valueOf(103, 2);
        // 1+ratenzahlungszuschlag=1.03 (103 Prozent)
        nettobeitragZw = jahresbasisbeitrag.multiply(rzFaktor, RoundingMode.HALF_UP);
    }
    nettobeitragZw = nettobeitragZw.divide(zahlweise.intValue(),
RoundingMode.HALF_UP);
}

```

Nach dem Anpassen wird `berechneJahresbasisbeitrag();` als Fehler angezeigt. Dies wird im nächsten Schritt behandelt.

Beitragsberechnung für die Hausrat-Deckungen

Für die Hausratversicherung muss nun die Beitragsberechnung auf Deckungsebene implementiert werden.

Der Jahresbasisbeitrag für die Grunddeckung wird wie folgt berechnet:

- Ermittlung des Beitragssatzes pro 1000 Euro Versicherungssumme aus der Tariftabelle.
- Division der Versicherungssumme durch 1000 Euro und Multiplikation mit dem Beitragssatz.

Da sich diese Berechnungsvorschrift nicht ändert, implementieren wir sie direkt in der Javaklasse *HausratGrunddeckung*. Bei den Zusatzversicherungen werden wir dann der Fachabteilung erlauben den Jahresbasisbeitrag über Berechnungsformeln festzulegen.

Zunächst definieren wir die Methode `berechneJahresbasisbeitrag()` im Editor der *HausratGrunddeckung* (mit Access Modifier `published`).

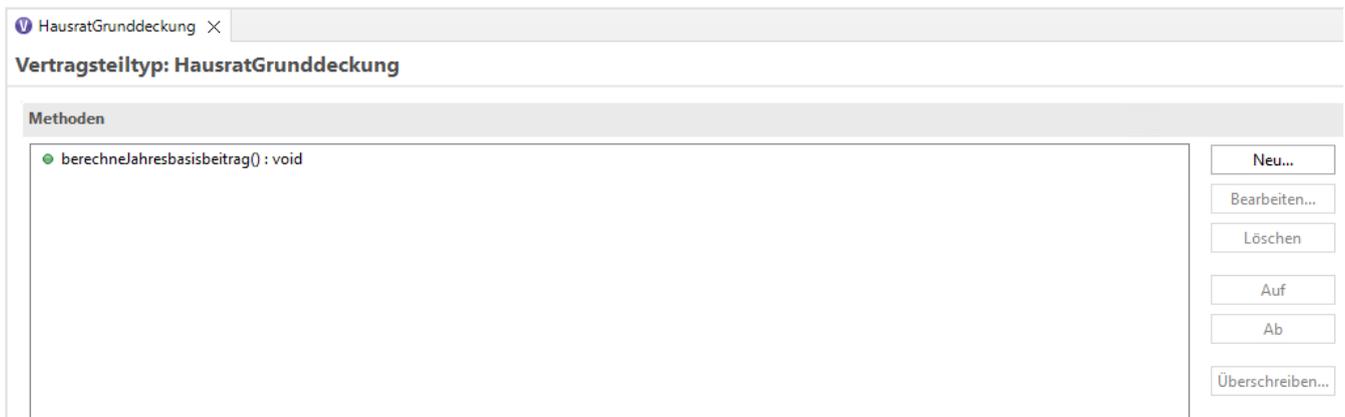


Figure 74. Anlegen der Methode *berechneJahresbasisbeitrag*

Öffnen Sie nun die Javaklasse im Editor und implementieren Sie die Methode wie folgt:

```
/**
 * @generated NOT
 */
@IpsGenerated
public void berechneJahresbasisbeitrag() {
    TariftabelleHausrat tabelle = getTariftabelle();
    TariftabelleHausratRow row = null;
    if (tabelle != null) {
        row = tabelle.findRow(getHausratVertrag().getTarifzone());
    }
    if (row == null) {
        jahresbasisbeitrag = Money.NULL;
        return;
    }
    Money vs = getHausratVertrag().getVersSumme();
    Decimal beitragsatz = row.getBeitragsatz();
    jahresbasisbeitrag = vs.divide(1000, RoundingMode.HALF_UP).multiply(beitragsatz,
    RoundingMode.HALF_UP);
}
```

Wir testen die Beitragsberechnung indem wir wieder unseren JUnit-Test erweitern.

```
@Test
public void testBerechneBeitrag() {
    // Erzeugen eines hausratvertrags mit der Factorymethode des Produktes
    HausratVertrag vertrag = kompaktProdukt.createHausratVertrag();
    // Vertragsattribute setzen
    vertrag.setPlz("45525"); // => tarifzone 3
    vertrag.setVersSumme(Money.euro(60000));
    vertrag.setZahlweise(new Integer(2)); // halbjährlich
    /*
     * Grunddeckungstyp holen, der dem Produkt in der Anpassungsstufe
     * zugeordnet ist.
     */
    HausratGrunddeckungstyp deckungstyp = kompaktProdukt
```

```

        .getHausratGrunddeckungstyp();
// Grunddeckung erzeugen und zum Vertrag hinzufügen
HausratGrunddeckung deckung = vertrag
        .newHausratGrunddeckung(deckungstyp);
// Beitrag berechnen und Ergebniss prüfen
vertrag.berechneBeitrag();

// tarifzone 3 => beitragsatz = 1.21 jahresbasisbeitrag
// = versicherungssumme / 1000 * beitragsatz = 60000 / 1000 * 1,21
// = 72,60
assertEquals(Money.euro(72, 60), deckung.getJahresbasisbeitrag());

// Jahresbasisbeitrag vertrag = Jahresbasisbeitrag deckung
assertEquals(Money.euro(72, 60), vertrag.getJahresbasisbeitrag());

// NettobeitragZw = 72,60 / 2 * 1,03 (wg. Ratenzahlungszuschlag von 3%)
// = 37,389
// => 37,39
assertEquals(Money.euro(37, 39), vertrag.getNettobeitragZw());

// BruttobeitragZw = 37,39 * Versicherungssteuersatz = 37,39 * 1,19
// = 44,49
assertEquals(Money.euro(44, 49), vertrag.getBruttobeitragZw());
}

```

Verwendung von Formeln

Unser bisheriges Hausratmodell bietet der Fachabteilung wenig Flexibilität. Ein Produkt kann genau eine Grunddeckung haben, der Beitrag wird über die Tariftabelle festgelegt. Jetzt wollen wir es der Fachabteilung erlauben, flexibel Zusatzdeckungen zu definieren, ohne dass das Modell oder der Programmcode (durch die Anwendungsentwicklung) geändert werden muss. Für die Berechnung des Beitrags werden wir hierzu die Formelsprache von Faktor-IPS verwenden.

Zusatzdeckungen gegen Fahrraddiebstahl und Überspannungsschäden dienen uns als Beispiel:

	<i>Fahrraddiebstahl</i>	<i>Überspannung</i>
Versicherungssumme der Zusatzdeckung	1% der im Vertrag vereinbarten Summe, maximal 3000 Euro.	5% der im Vertrag vereinbarten Summe. Keine Deckelung.
Jahresbasisbeitrag	10% der Versicherungssumme der Fahrraddiebstahldeckung	10 Euro + 3% der Versicherungssumme der Überspannungsdeckung

Die Zusatzdeckungen haben eine eigene Versicherungssumme, die von der im Vertrag vereinbarten Versicherungssumme abhängt. Der Jahresbasisbeitrag hängt wiederum von der Versicherungssumme der Deckung ab. Um solche Zusatzdeckungen abbilden zu können, erweitern wir das Modell nun wie im folgenden Klassendiagramm abgebildet:

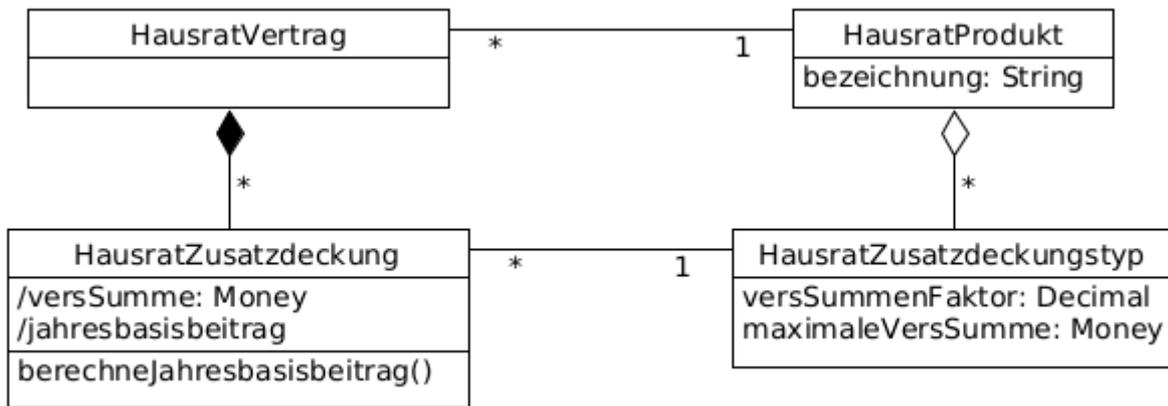


Figure 75. Modellausschnitt für Zusatzdeckungen

Ein *HausratVertrag* kann beliebig viele solcher Zusatzdeckungen enthalten. Die Konfigurationsklasse zur *HausratZusatzdeckung* nennen wir *HausratZusatzdeckungstyp*. Diese hat die Eigenschaften „versSummenFaktor“ und „maximaleVersSumme“. Die Versicherungssumme der Zusatzdeckung ergibt sich durch Multiplikation der Versicherungssumme des Vertrags mit dem Faktor und wird durch die maximale Versicherungssumme gedeckelt.

Unsere beiden Beispieldeckungen sind Instanzen der Klasse *HausratZusatzdeckungstyp*. Dies zeigt das folgende Diagramm.

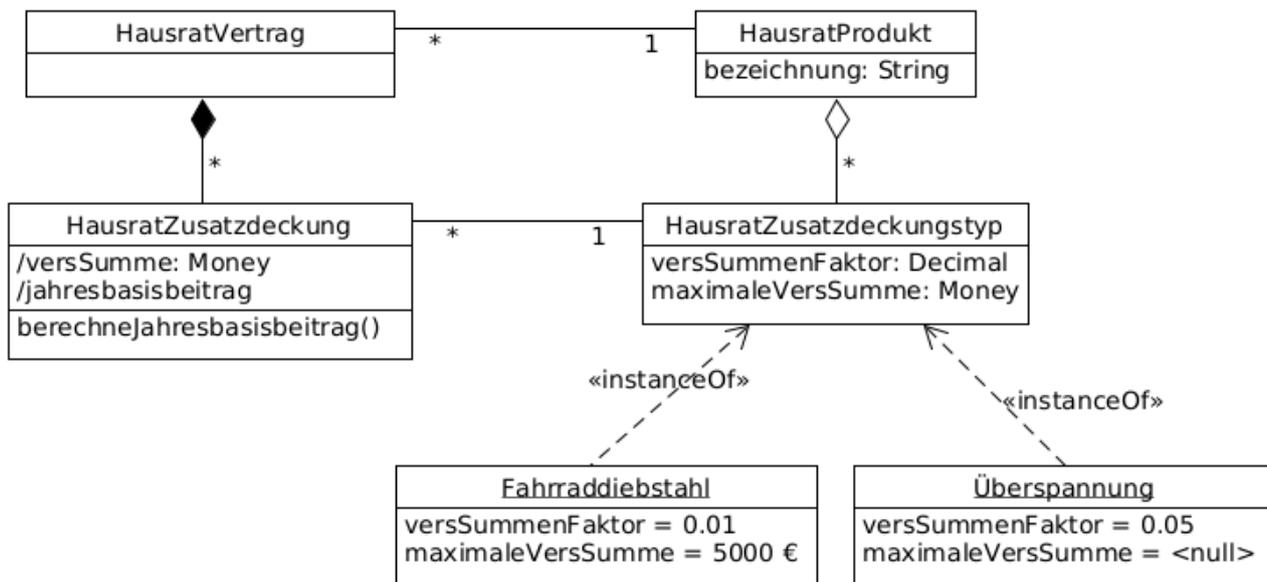


Figure 76. Modellausschnitt für Zusatzdeckungen mit Instanzen

Bevor wir zur Beitragsberechnung kommen, legen wir zunächst die beiden neuen Klassen *HausratZusatzdeckung* und *HausratZusatzdeckungstyp* an. Mit dem Assistenten zum Anlegen einer neuen Vertragsklasse kann man auch direkt die zugehörige Konfigurationsklasse mit erzeugen. Starten Sie nun den Assistenten und geben auf der ersten Seite die Daten wie im folgenden Bild zu sehen ein

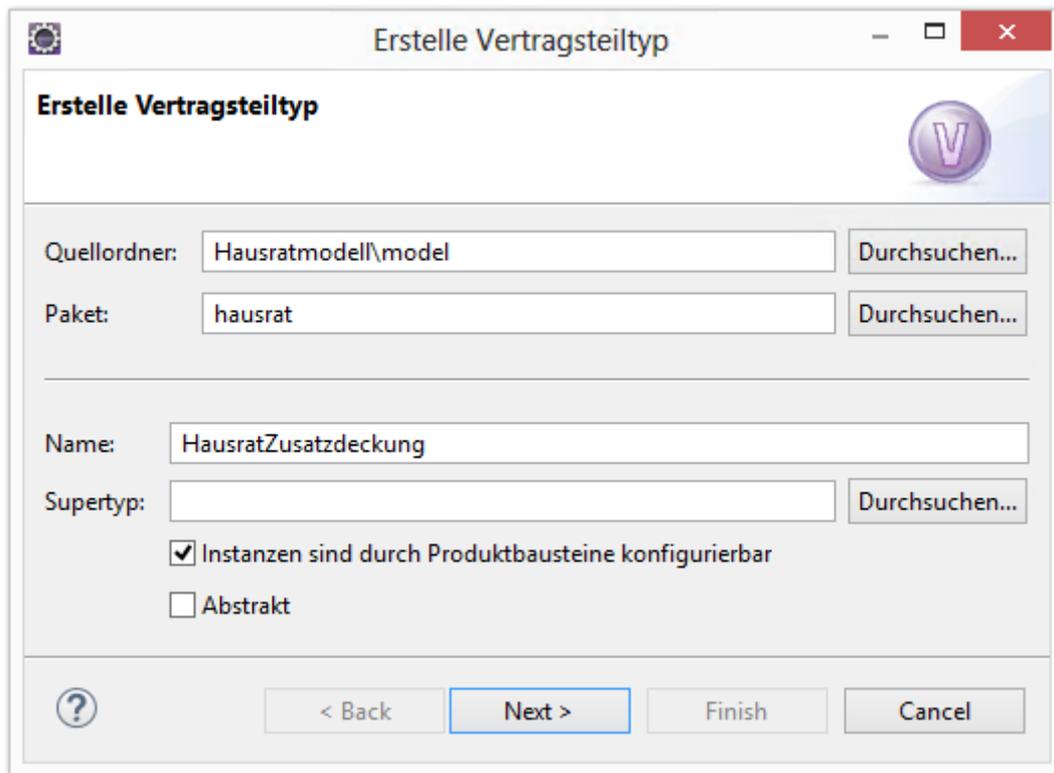


Figure 77. Assistent zum Anlegen der Vertragsklasse *HausratZusatzdeckung*

Auf der zweiten Seite geben Sie noch den Klassennamen *HausratZusatzdeckungstyp* ein und klicken *Finish*. Faktor-IPS legt nun beide Klassen an und richtet auch die Referenzen aufeinander ein.

Nun müssen wir noch die Beziehung zwischen *HausratVertrag* und *HausratZusatzdeckung* und entsprechend zwischen *HausratProdukt* und *HausratZusatzdeckungstyp* anlegen. Hierzu verwenden Sie den Assistenten zum Anlegen einer neuen Beziehung im Vertragsklasseneditor. Dieser erlaubt es Ihnen gleichzeitig auch die Beziehung auf der Produktseite mit anzulegen.

Nachdem die Beziehungen definiert sind, legen Sie an der Klasse *HausratZusatzdeckung* das abgeleitete (Getter) Attribut *versSumme* (Money) an und an der Klasse *HausratZusatzdeckungstyp* die Attribute *versSummenFactor* (Decimal), *maximaleVersSumme* (Money), und *bezeichnung* (String). Nachdem dies geschehen ist, können wir auch direkt die Ermittlung der Versicherungssumme implementieren. Hierzu implementieren wir in der Klasse *HausratZusatzdeckung* die Methode *getVersSumme()* wie folgt:

```
/**
 * Gibt den Wert des Attributs versSumme zurueck.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "versSumme", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getVersSumme() {
    // begin-user-code
    HausratZusatzdeckungstyp typ = getHausratZusatzdeckungstyp();
    if (typ == null) {
```

```

    return Money.NULL;
}
Decimal faktor = typ.getVersSummenFaktor();
Money vsVertrag = getHausratVertrag().getVersSumme();
Money vs = vsVertrag.multiply(faktor, RoundingMode.HALF_UP);
if (vs.isNull()) {
    return vs;
}
Money maxVs = typ.getMaximaleVersSumme();
if (vs.greaterThan(maxVs)) {
    return maxVs;
}
return vs;
// end-user-code
}

```

Nun legen wir die beiden Deckungstypen für die Versicherung von Fahrraddiebstählen bzw. Überspannungsschäden an. Wechseln Sie hierzu wieder in die Produktdefinitionsperspektive und markieren im Produktdefinitionsexplorer im Projekt Hausratprodukte das Paket „deckungen“. Nun legen Sie zwei Produktbausteine mit den Namen HRD-Fahrraddiebstahl 2019-07 und HRD-Überspannung 2019-07 basierend auf der Klasse HausratZusatzdeckungstyp an und geben im Editor deren Eigenschaften ein.

	HRD-Fahrraddiebstahl 2019-07	HRD-Überspannung 2019-07
Bezeichnung	Fahrraddiebstahl	Überspannungsschutz
VersSummenFaktor	0.01	0.05
Maximale VersSumme	3000 EUR	<null>

Nun müssen die neuen Deckungen noch den Produkten zugeordnet werden. Dies erfolgt analog zur Zuordnung der Grunddeckungen. Bei Verträgen, die auf Basis des Produktes HR-Optimal (das im Ordner „produkte“ liegt) abgeschlossen werden, sollen die Deckungen immer enthalten sein (Art der Beziehung ist Obligatorisch), beim Produkt HR-Kompakt sollen Sie optional dazu gewählt werden können (Art der Beziehung Optional). Sie können dies über die Art der Beziehung im Bausteineditor einstellen.



Figure 78. Ausschnitt aus dem Bausteineditor zum Bearbeiten der Beziehungen

Beitragsberechnung für die Zusatzdeckungen

Die Berechnung des Jahresbasisbeitrags soll durch die Fachabteilung durch eine Formel festgelegt werden. Der Beitrag für eine Zusatzdeckung wird i.d.R. von der Versicherungssumme und evtl. weiteren risikorelevanten Merkmalen abhängen [5]. In der Formel muss man also auf diese Eigenschaften zugreifen können. Hierzu sind prinzipiell zwei Wege denkbar:

- die Formelsprache erlaubt eine beliebige Navigation durch den Objektgraphen
- die in der Formel verwendbaren Parameter werden explizit festgelegt.

In Faktor-IPS wird die zweite Alternative verwendet. Hierfür gibt es zwei Gründe:

1. Die Syntax für die Navigation durch den Objektgraphen kann schnell komplex werden. Wie sieht zum Beispiel eine für die Fachabteilung leicht verständliche Syntax zur Ermittlung der Deckung mit der höchsten Versicherungssumme aus?
2. Aktualität von abgeleiteten Attributen

5 In der Hausratversicherung neben der Tarifzone zum Beispiel die Art des Hauses (Ein-/Mehrfamilienhaus) oder die Bauweise.

Der zweite Aspekt lässt sich am besten an einem Beispiel erläutern. Für die Berechnung des Beitrags einer Zusatzdeckung wird die Versicherungssumme benötigt. Diese ist selbst ein abgeleitetes Attribut. Wenn es sich dabei um ein gecachtes Attribut handelt, muss vor dem Aufruf der Formel zur Beitragsberechnung sichergestellt werden, dass die Versicherungssumme berechnet wurde. Erlaubt man nun eine beliebige Navigation durch den Objektgraphen muss man für alle erreichbaren Objekte sicherstellen, dass die abgeleiteten Attribute berechnete Werte enthalten. Da dies leicht zu Fehlern führt und auch bzgl. der Performance ungünstig ist, werden in Faktor-IPS die in einer Formel verwendbaren Parameter explizit festgelegt.

Als Formelparameter können sowohl einfache Parameter wie z. B. die Versicherungssumme definiert werden als auch ganze Objekte wie z.B. der Vertrag. Letzteres hat den Vorteil, dass die Parameterliste nicht erweitert werden muss, wenn die Fachabteilung auf bisher nicht genutzte Merkmale zugreift. Für die Berechnung des Jahresbasisbeitrags der Hausratzusatzdeckung verwenden wir die Zusatzdeckung selbst und den Hausratvertrag (zu dem die Deckung gehört) als Parameter.

Um in Zusatzdeckungen die Formel für die Beitragsberechnung hinterlegen zu können, muss zunächst in der Klasse *HausratZusatzdeckungstyp* die Formelsignatur mit den Parametern definiert werden. Öffnen Sie nun den Editor für die Klasse *HausratZusatzdeckungstyp*. Klicken Sie auf der zweiten Seite im Abschnitt *Methods and Formula Signatures* (Methoden und Formelsignaturen) auf den *New* Button, um eine Formelsignatur anzulegen und tragen Sie die Daten wie im folgenden Screenshot zu sehen ein.

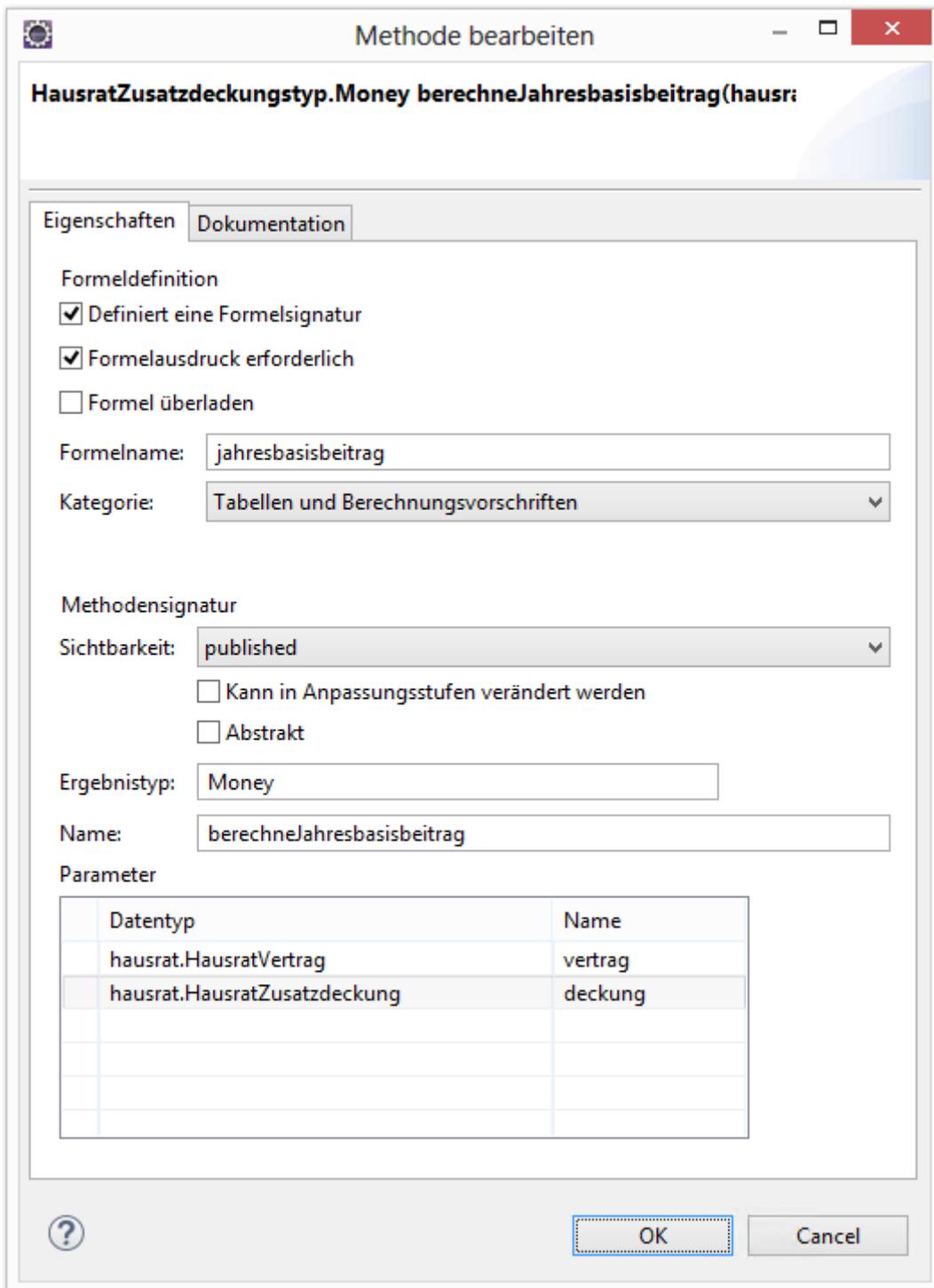


Figure 79. Dialog zur Definition einer Formelsignatur

Schließen Sie den Dialog und speichern. In der Klasse `HausratZusatzdeckungstyp` befindet sich nun die Methode `berechneJahresbasisbeitrag(...)` zur Berechnung des Basisbeitrags.

Öffnen wir nun die `Fahrraddiebstahldeckung`, um die Formel für die Beitragsberechnung festzulegen. Beim Öffnen erscheint zunächst ein Dialog, in dem angezeigt wird, dass es im Modell eine neue Formel gibt, die es bisher nicht in der Produktdefinition gab.

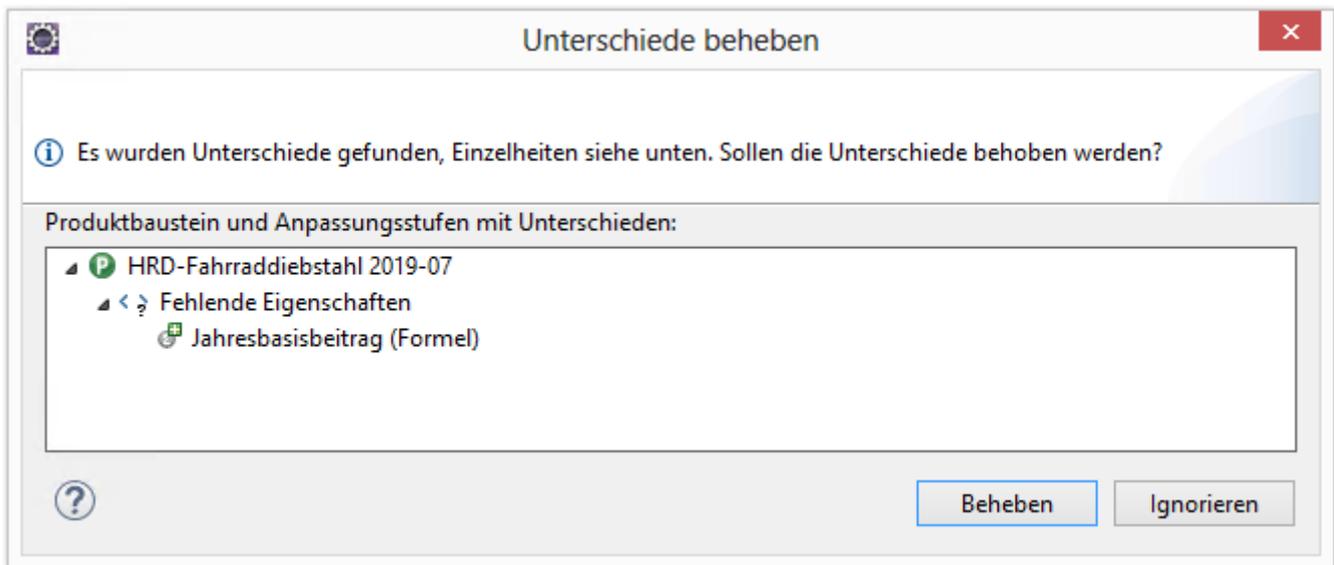


Figure 80. Dialog Unterschiede beheben

Bestätigen Sie mit Beheben, dass diese hinzugefügt werden soll. In dem Abschnitt *Tables & Formulas* (Tabellen und Berechnungsvorschriften) wird nun die noch leere Formel für den Beitragssatz angezeigt. Klicken Sie auf den Button neben dem Formelfeld, um die Formel zu editieren. Es öffnet sich der folgende Dialog, in dem Sie die Formel bearbeiten können und in dem auch die verfügbaren Parameter angezeigt werden:

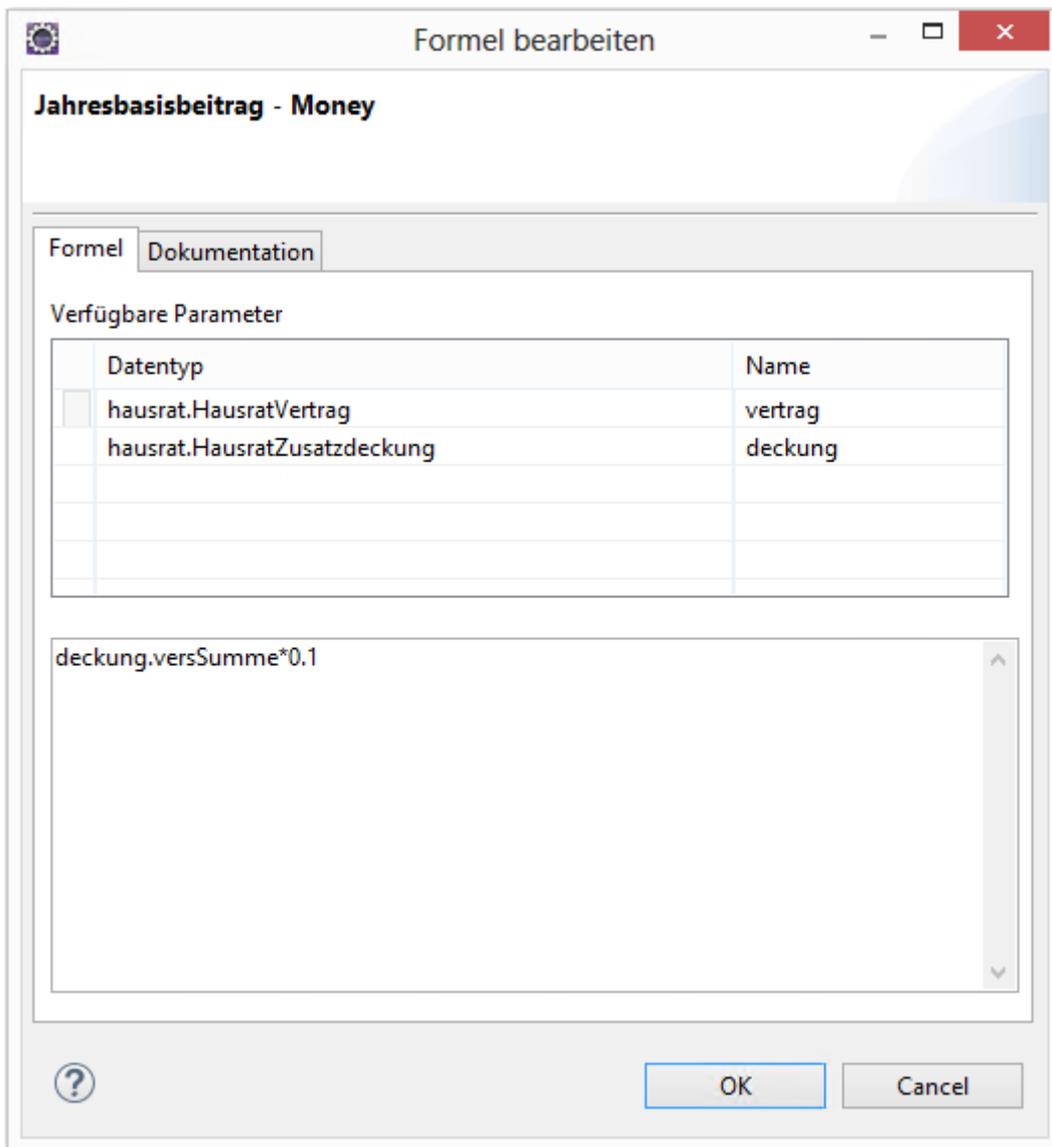


Figure 81. Anlegen einer Formel

Der Beitrag für die Fahrraddiebstahlversicherung soll 10% der Versicherungssumme der Zusatzdeckung betragen. Drücken Sie Strg-Space in der mittleren Eingabebox und Sie sehen die Parameter und Funktionen, die Sie zur Verfügung haben. Wählen Sie den Parameter „deckung“ aus und geben danach noch einen Punkt ein. Sie bekommen die Eigenschaften der Deckung zur Auswahl angeboten. Wählen Sie die „versSumme“. Nun multiplizieren Sie die Versicherungssumme noch mit 0.1.

Schließen Sie den Dialog und speichern den Baustein. Analog definieren Sie nun, dass der Beitrag für die Überspannungsdeckung 10Euro + 3% der Versicherungssumme beträgt (Formel: 10EUR + deckung.versSumme * 0.03).

Faktor-IPS hat nun die Subklassen für die beiden Produktbausteine mit der in Java Sourcecode übersetzten Formel generiert. Sie finden beide Klassen im Java-Sourcefolder „src/main/resources“ im Package `org.faktorips.tutorial.produktdaten.internal.deckungen` [6]. Der folgende Sourcecode enthält die generierte Methode `berechneJahresbasisbeitrag(...)` für die Fahrraddiebstahldeckung.

6 Da der Name von Produktbausteinen auch Blanks und Bindestriche enthalten kann, diese aber nicht in Java- Klassennamen erlaubt sind, wurden diese durch Unterstriche ersetzt. Konfigurieren können Sie die Ersetzung in der „ipsproject“ Datei im Abschnitt `ProductCmptNamingStrategy`.

```

public Money berechneJahresbasisbeitrag(final HausratVertrag vertrag, final
HausratZusatzdeckung deckung) throws FormulaExecutionException {
    try {
        return deckung.getVersSumme().multiply(Decimal.valueOf("0.1"),
RoundingMode.HALF_UP);
    } catch (Exception e) {
        StringBuffer parameterValues = new StringBuffer();
        parameterValues.append("vertrag=");
        parameterValues.append(vertrag == null ? "null" : vertrag.toString());
        parameterValues.append(", ");
        parameterValues.append("deckung=");
        parameterValues.append(deckung == null ? "null" : deckung.toString());
        throw new FormulaExecutionException(toString(), "deckung.versSumme*0.1",
parameterValues.toString(), e);
    }
}

```

Hinweis: Faktor-IPS bietet verschiedene Codegenerator-Optionen für die Formel-Kompilierung an. So kann über die Property `formulaCompiling` in der „ipsproject“ Datei (oder über das Kontextmenü > Eigenschaften > Faktor-IPS Code Generator des entsprechenden Projekts) festgelegt werden, ob Formeln im XML generiert und zur Laufzeit abgerufen werden können (`value="XML"`), für Produktbausteine die Formeln enthalten Subklassen der Produktklasse generiert werden, die die entsprechenden Methoden überschreiben (`value="Subclass"`), oder beide Varianten generiert werden (`value="Both"`).

Setzen Sie die Property `formulaCompiling` für das Projekt `faktorips-tutorial-hausratprodukte` von `Both` auf `Subclass`.

Tritt beim Ausführen der in Java übersetzten Formel ein Fehler auf, wird eine `RuntimeException` geworfen, die den Formeltext sowie die String-Repräsentation der übergebenen Parameter enthält.

Nun müssen wir noch dafür sorgen, dass die Formel im Rahmen der Beitragsberechnung auch aufgerufen wird, indem wir in der Klasse `HausratZusatzdeckung` die Methode `berechneJahresbasisbeitrag()` implementieren. Dies geht nun einfach, indem wir an die Berechnungsmethode im `Zusatzdeckungstyp` delegieren und als Parameter die Zusatzdeckung (`this`) und den Vertrag, zu dem die Deckung gehört, übergeben. Legen Sie in der Modellklasse `HausratZusatzdeckung` die Methode `berechneJahresbasisbeitrag` mit dem Rückgabewert `Money`, der Sichtbarkeit `published` und ohne Parameter an und speichern.

Öffnen Sie nun die Java-Klasse `HausratZusatzdeckung` und implementieren Sie die Methode wie folgt.

```

/**
 * @generated NOT
 */
@IpsGenerated
public Money berechneJahresbasisbeitrag() {
    return
getHausratZusatzdeckungstyp().berechneJahresbasisbeitrag(getHausratVertrag(), this);
}

```

```
}
```

Damit der Beitrag der Zusatzdeckungen auch dem Gesamtbeitrag des Hausratvertrags zugeschlagen wird, erweitern wir noch die Beitragsberechnung in der Klasse Hausratvertrag:

```
private void berechneJahresbasisbeitrag() {
    jahresbasisbeitrag = Money.euro(0, 0);
    HausratGrunddeckung hausratGrunddeckung = getHausratGrunddeckung();

    hausratGrunddeckung.berechneJahresbasisbeitrag();
    jahresbasisbeitrag =
jahresbasisbeitrag.add(hausratGrunddeckung.getJahresbasisbeitrag());

    /*
     * Über Zusatzdeckungen iterieren und jeweils den Beitrag dem Gesamtbeitrag
     * hinzuaddieren:
     */
    List<? extends HausratZusatzdeckung> zusatzdeckungen =
getHausratZusatzdeckungen();
    for (int i = 0; i < zusatzdeckungen.size(); i++) {
        HausratZusatzdeckung hausratZusatzdeckung = zusatzdeckungen.get(i);
        jahresbasisbeitrag =
jahresbasisbeitrag.add(hausratZusatzdeckung.berechneJahresbasisbeitrag());
    }
}
```

Zum Abschluss des Kapitels testen wir die neue Funktionalität wieder durch die Erweiterung des JUnit-Tests wie folgt.

```
@Test
public void testBerechneJahresbasisbeitragFahrraddiebstahl() {
    // Erzeugen eines hausratvertrags mit der Factorymethode des Produktes
    HausratVertrag vertrag = kompaktProdukt.createHausratVertrag();

    // Vertragsattribute setzen
    vertrag.setVersSumme(Money.euro(60_000));

    /*
     * Zusatzdeckungstyp Fahrraddiebstahl holen. Der Einfachheit halber nehmen wir
     * hier an, der erste Zusatzdeckungstyp ist Fahrraddiebstahl
     */
    HausratZusatzdeckungstyp deckungstyp =
kompaktProdukt.getHausratZusatzdeckungstyp(0);

    // Zusatzdeckung erzeugen
    HausratZusatzdeckung deckung = vertrag.newHausratZusatzdeckung(deckungstyp);

    // Jahresbasisbeitrag berechnen und testen
}
```

```
deckung.berechneJahresbasisbeitrag();

/*
 * Versicherungssumme der Deckung = 1% von 60.000, max 5.000 => 600 Beitrag =
 * 10% von 600 = 60
 */
assertEquals(Money.euro(60, 0), deckung.berechneJahresbasisbeitrag());
}
```

In diesem zweiten Teil des Tutorials haben wir gesehen, wie Tabellen in Faktor-IPS genutzt werden, haben die Beitragsberechnung implementiert und getestet und am Beispiel der Zusatzdeckungen gesehen, wie man ein Modell so gestaltet, dass es flexibel erweitert werden kann.

Ein weiteres Tutorial zeigt, wie man mit den hier erstellten Modellklassen in einer operativen Anwendung arbeitet (Tutorial Hausrat Angebotssystem).

Im Tutorial zur Modellpartitionierung wird gezeigt, wie man komplexe Modelle in sinnvolle Teile zerlegt und damit umgeht. Insbesondere die Trennung in einzelne Sparten und die Trennung von spartenspezifischen und spartenübergreifenden Aspekten wird dort beispielhaft gezeigt.

Wie man beim Testen von Faktor-IPS unterstützt wird, zeigt das Tutorial Softwaretests mit Faktor-IPS.

Part 2: Using Tables and Formulas

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 which we need later on:

zipcodeFrom	zipcodeTo	rating district
17235	17237	II
30159	45549	III
59174	59199	IV
47051	47279	V
63065	63075	VI
...

Rating District Table

For all zipcodes that do not fall into one of these 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.

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

Select the table type `SINGLE_CONTENT`, because we want this structure to have only one content. Now we will 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` \rightarrow `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.

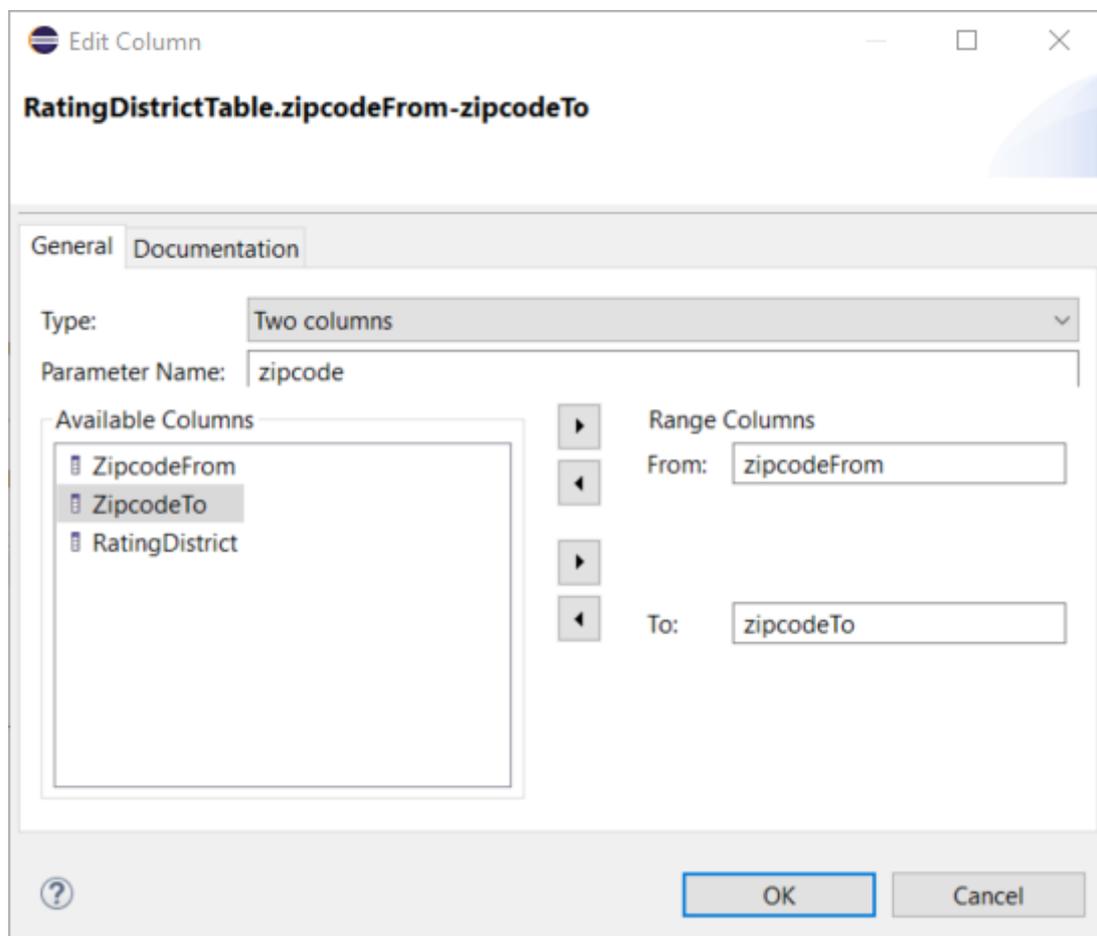


Figure 82. Creating a Range in a Table Structure

Now you have to create a new unique key index. 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.

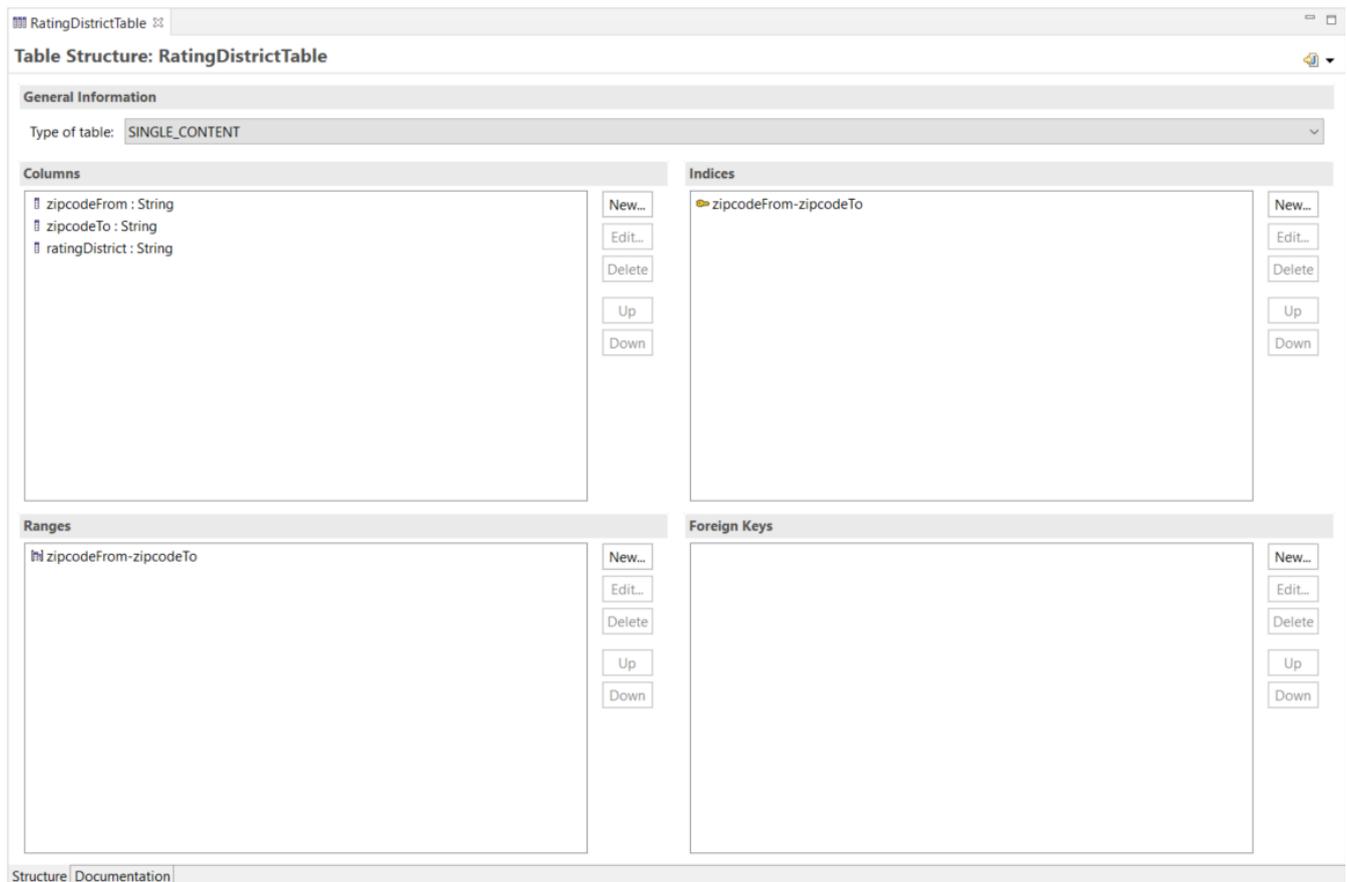


Figure 83. Tabellen Structure Rating District Table

Faktor-IPS has now created two more classes for the table structure in the source folder `src/main/java` in the package `org.faktorips.tutorial.model.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 `RatingDistrictTable` 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() {
    // begin-user-code
    // TODO: later we'll implement this with a table lookup
    return "I";
}
```

```
// end-user-code  
}
```

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

```
public String getRatingDistrict() {  
    // begin-user-code  
    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();  
    // end-user-code  
}
```

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 [1].

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

Next, we will create the table content. The business users will be responsible for maintaining the mapping of zipcodes to rating districts. In order to enhance the overall structure, please add a new package named `tables` underneath the `productdata` package of the `HomeProducts` project. Then select the new package and click the toolbar button . When the dialog box opens, choose the `RatingDistrictTable` structure. Name the table content *RatingDistrictTable* 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:

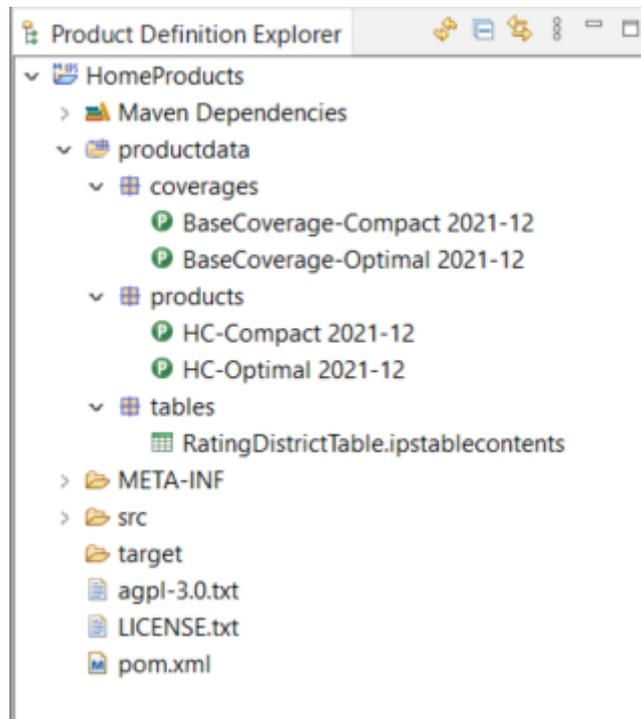


Figure 84. The Product Definition Project Structure

Finally, we will test if the rating district is determined correctly. In order to do this, we will extend the JUnit test `TutorialTest` of the first part of this tutorial by adding the following test method [2].

2 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.

```
@Test
public void testGetRatingDistrict() {
    // Create a new HomeContract with the products factory method
    HomeContract contract = compactProduct.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 premiums 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

rating district	premium rate
VI	2.20

Table 5. 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

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 index 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.

For both HC-Optimal and HC-Compact (or, more precisely, for their base coverage types), create two table contents named "RateTable Optimal 2021-12" and "RateTable Compact 2021-12", respectively [3].

3 Replace the "2021-12" suffix with the respective effective date that you are using.

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

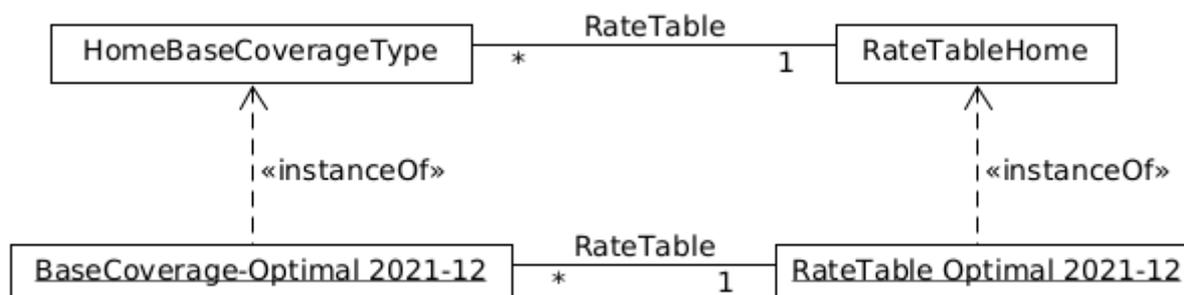


Figure 85. Relationship between Product Components and Tables

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[4], 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 Provided that you have set your Preferences such that your Editors use 2 sections per page.

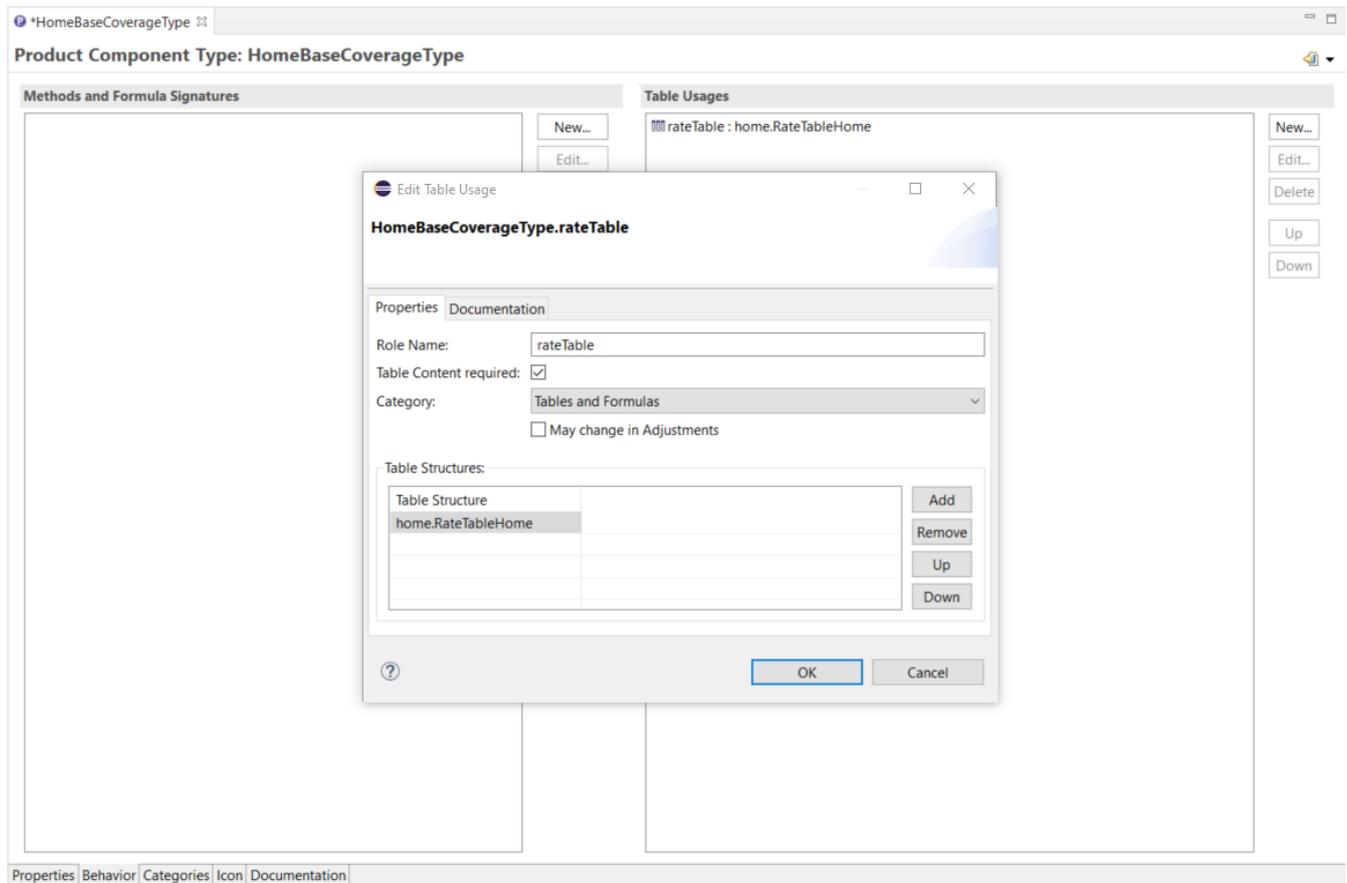


Figure 86. Modellierung der Verwendung von Tabellen

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 2021-12](#). When a dialog box pops up to tell you that the rate table has not yet been mapped, confirm it with *Fix*. In the *Tables and Formulas* section, you can now map the rate table for Optimal and save your work.

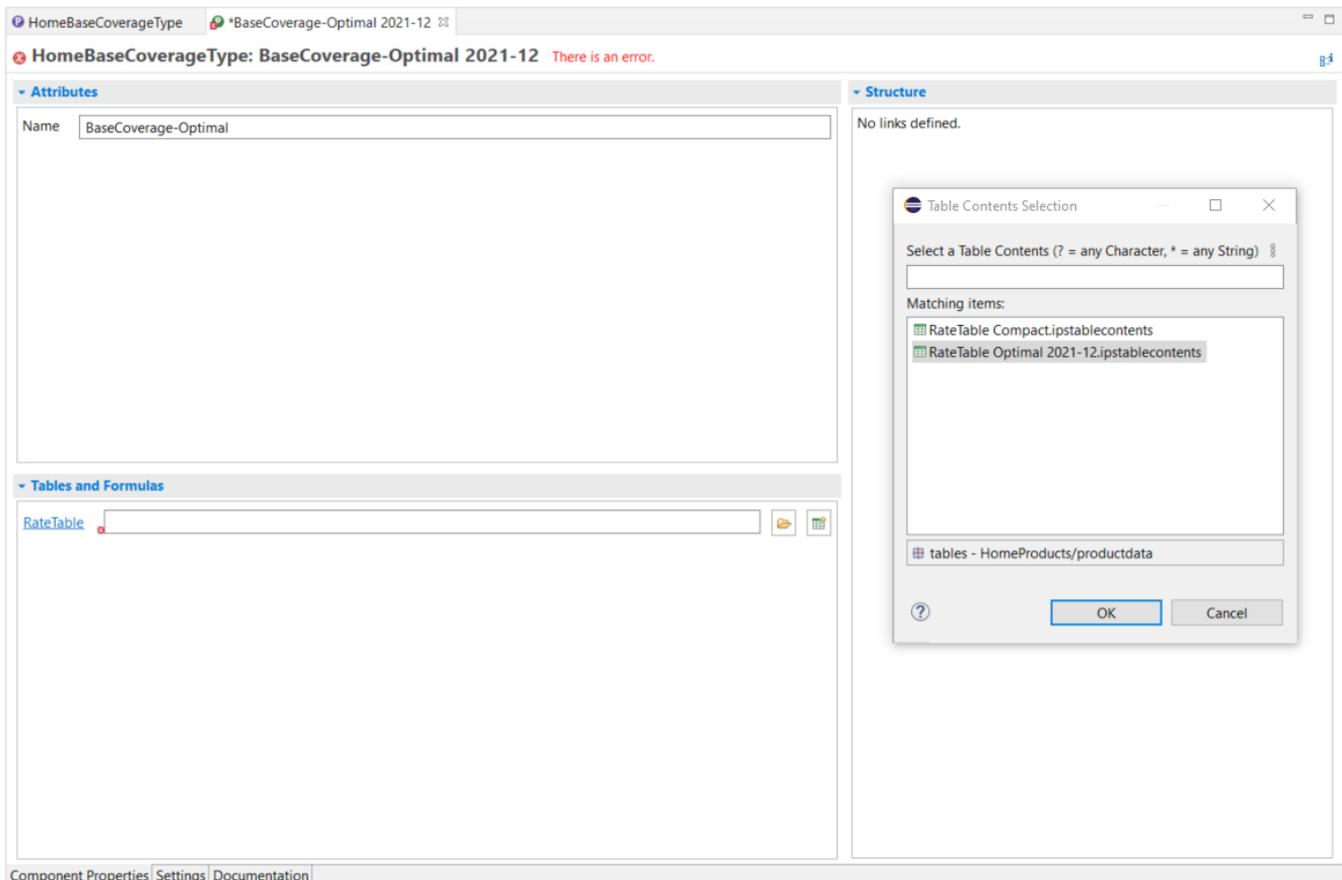


Figure 87. Assigning Table Contents

The same process applies to BaseCoverage-Compact.

At the end of this chapter, we will take a look at the generated source code. In the `HomeBaseCoverageType` 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.

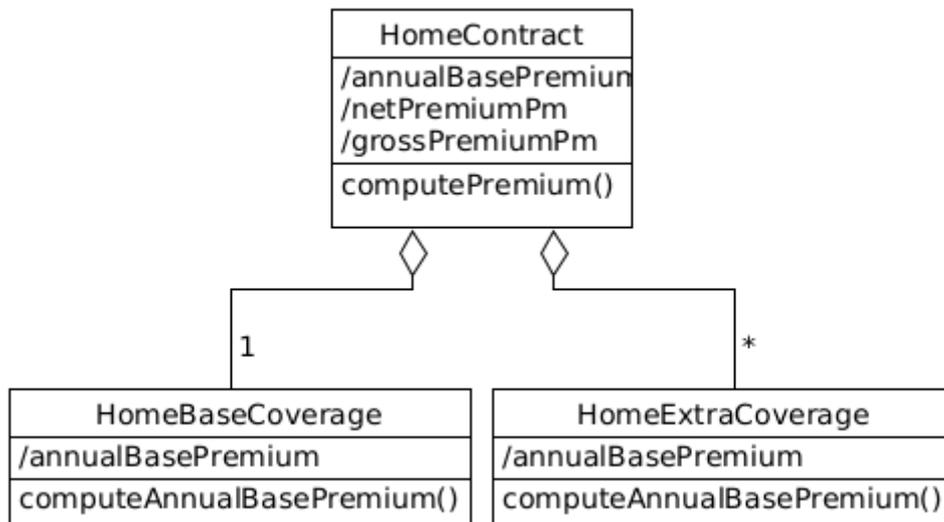


Figure 88. The Premium Computation is Captured in the Model

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` and `HomeBaseCoverage` classes. The dialog box for editing a method signature is shown in the next figure.

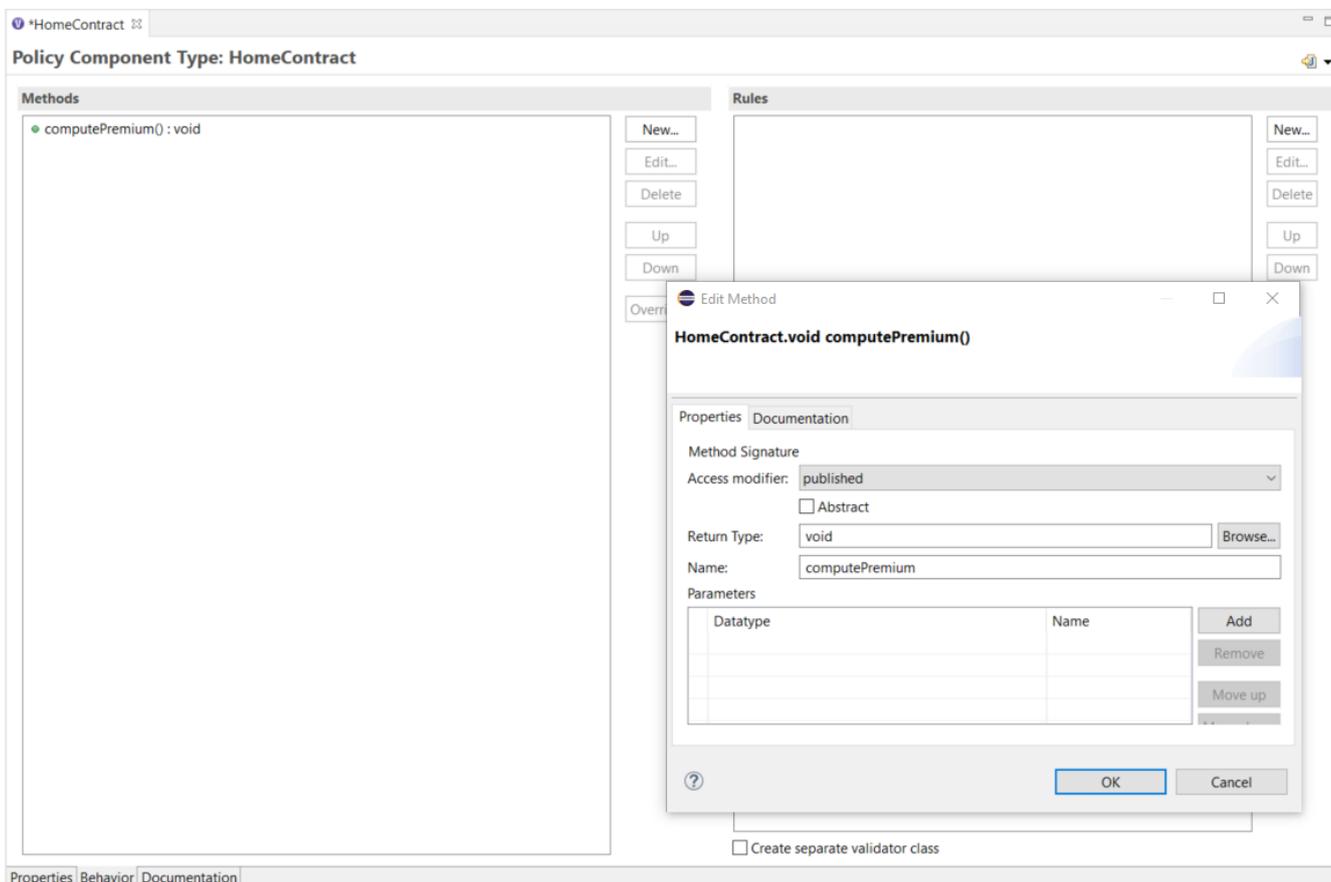


Figure 89. Edit Dialog for Method Signatures

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.

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.

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

private void computeAnnualBasePremium() {
    annualBasePremium = Money.euro(0, 0);
    HomeBaseCoverage baseCoverage = getHomeBaseCoverage();
    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.intValue() == 1) {
        netPremiumPm = annualBasePremium;
    } else {
        Decimal factor = Decimal.valueOf(103, 2); // 1 + 0.03 surcharge for non-annual
payment
        netPremiumPm = annualBasePremium.multiply(factor, RoundingMode.HALF_UP);
    }
    netPremiumPm = netPremiumPm.divide(paymentMode.intValue(), RoundingMode.HALF_UP);
}
}

```

After customizing the code `computeAnnualBasePremium()`; will be marked as an error. This will be solved in the next step.

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.

We already defined the method `computeAnnualBasePremium()` in the `HomeBaseCoverage` class at the beginning of this chapter. Open the Java class `HomeBaseCoverage` in the editor and implement the method as follows:

```

/**
 * @generated NOT
 */
@IpsGenerated
public void computeAnnualBasePremium() {
    RateTableHome table = getRateTable();
    RateTableHomeRow row = null;
}

```

```

if(table!=null) {
    row=table.findRow(getHomeContract().getRatingDistrict());
}
if(row==null) {
    annualBasePremium=Money.NULL;
    return;
}
Money si = getHomeContract().getSumInsured();
Decimal premiumRate = row.getPremiumRate();
annualBasePremium = si.divide(1000, RoundingMode.HALF_UP).multiply(premiumRate,
RoundingMode.HALF_UP);
}

```

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

```

@Test
public void testComputePremium() {
    // Create a new HomeContract with the products factory method
    HomeContract contract = compactProduct.createHomeContract();
    // Set contract attributes
    contract.setZipcode("45525");
    contract.setSumInsured(Money.euro(60_000));
    contract.setPaymentMode(2);
    // Get the base coverage type that is assigned to the product
    HomeBaseCoverageType coverageType = compactProduct.getHomeBaseCoverageType();
    // Create the base coverage and add it to the contract
    HomeBaseCoverage coverage = contract.newHomeBaseCoverage(coverageType);
    // Compute the premium and check the results
    contract.computePremium();
    // rating district III => premiumRate = 1,21 annualBasePremium = sumInsured /
    // 1000 * premiumRate = 60000 / 1000 * 1,21 = 72,60
    assertEquals(Money.euro(72, 60), coverage.getAnnualBasePremium());
    // contract.annualBasePremium = baseCoverage.annualBasePremium
    assertEquals(Money.euro(72, 60), contract.getAnnualBasePremium());
    // netPremiumPm = 72,60 / 2 * 1,03 (semi-annual, 3% surcharge) = 37,389 => 37,39
    assertEquals(Money.euro(37, 39), contract.getNetPremiumPm());
    // grossPremiumPm = 37,39 * taxMultiplier = 37,39 * 1,19 = 44,49
    assertEquals(Money.euro(44, 49), 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:

	<i>Bicycle Theft</i>	<i>Overvoltage Damage</i>
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	10 Euro + 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:

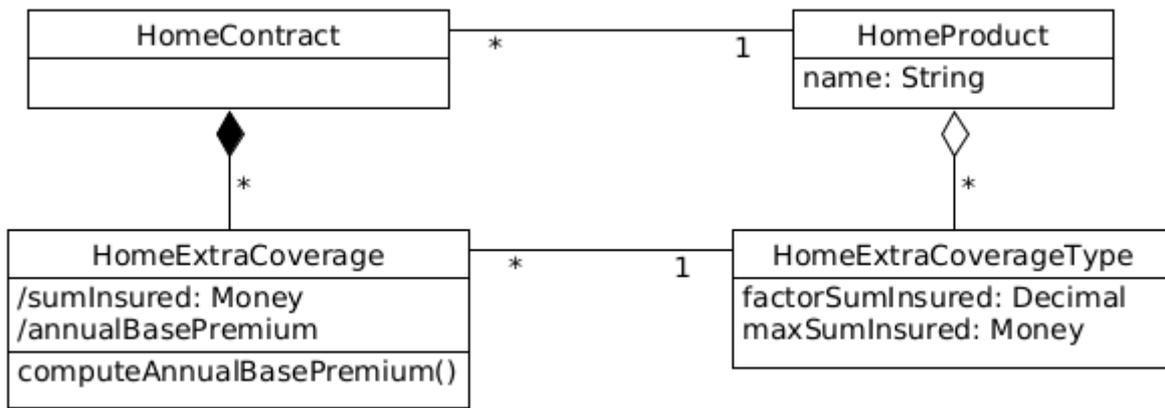


Figure 90. A Section of the Model Showing Extra Coverages

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.

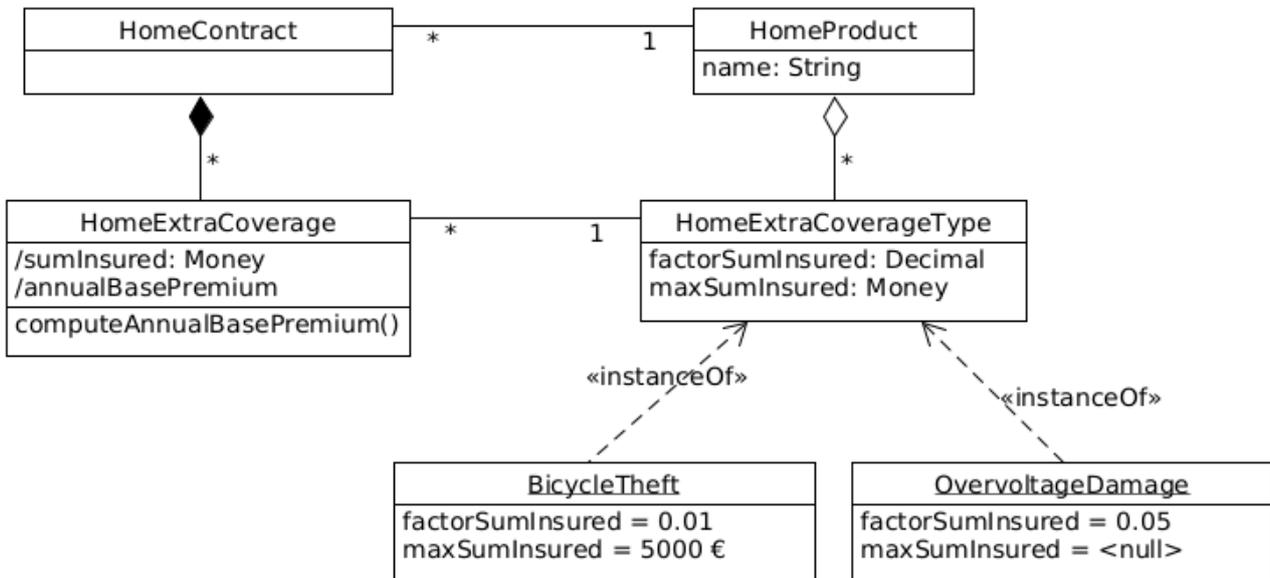


Figure 91. A Section of the Model Showing Extra Coverages with Instances

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.

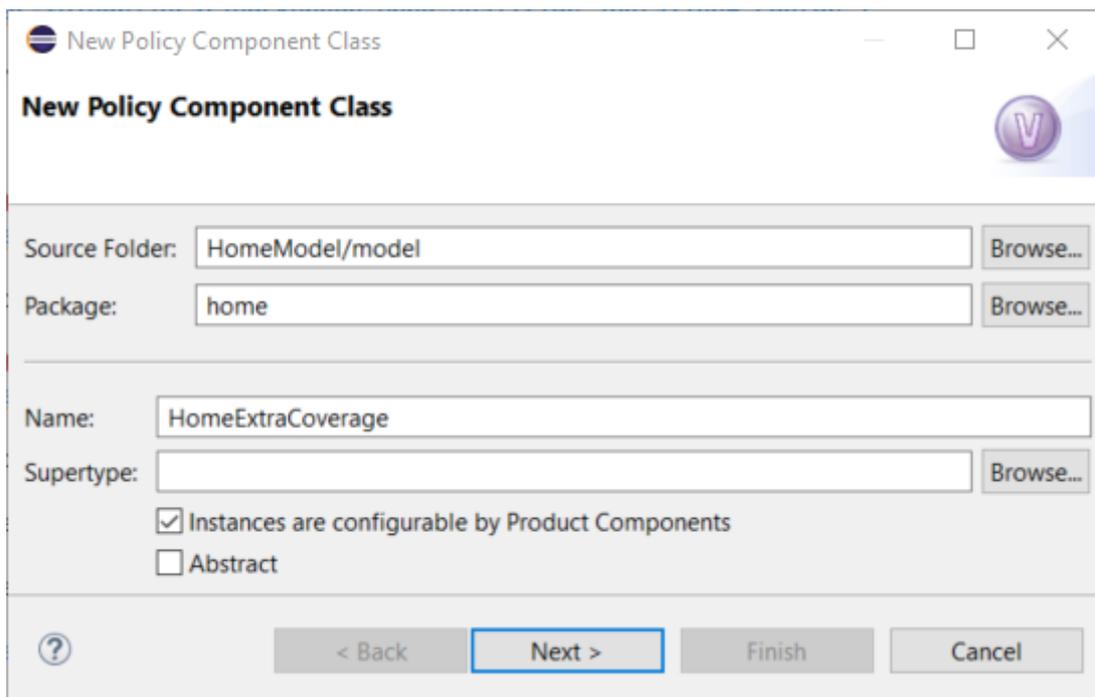


Figure 92. The Wizard for Creating the HomeExtraCoverage Contract Class

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 `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:

```
/**
 * Returns the sumInsured.
 *
 * @restrainedmodifiable
 */
@IpsAttribute(name = "sumInsured", kind = AttributeKind.DERIVED_ON_THE_FLY,
valueSetKind = ValueSetKind.AllValues)
@IpsGenerated
public Money getSumInsured() {
    // begin-user-code
    HomeExtraCoverageType type = getHomeExtraCoverageType();
    if(type == null) {
        return Money.NULL;
    }
    Decimal factor = type.getFactorSumInsured();
    Money sumInsuredContract = getHomeContract().getSumInsured();
    Money sumInsured = sumInsuredContract.multiply(factor, RoundingMode.HALF_UP);
    if(sumInsured.isNull()) {
        return sumInsured;
    }
    Money maxSumInsured = type.getMaxSumInsured();
    if(sumInsured.greaterThan(maxSumInsured)) {
        return maxSumInsured;
    }
    return sumInsured;
    // end-user-code
}
```

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 2021-12* and *OvervoltageDamage 2021-12*, respectively, and define their properties in the editor.

	Bicycle Theft 2019-07	OvervoltageDamage 2019-07
Name	Bicycle Theft	Overvoltage Damage
SumInsuredFactor	0.01	0.05
MaxSumInsured	3000 EUR	<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 93. 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 [5]. 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, could 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

⁵ 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 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.

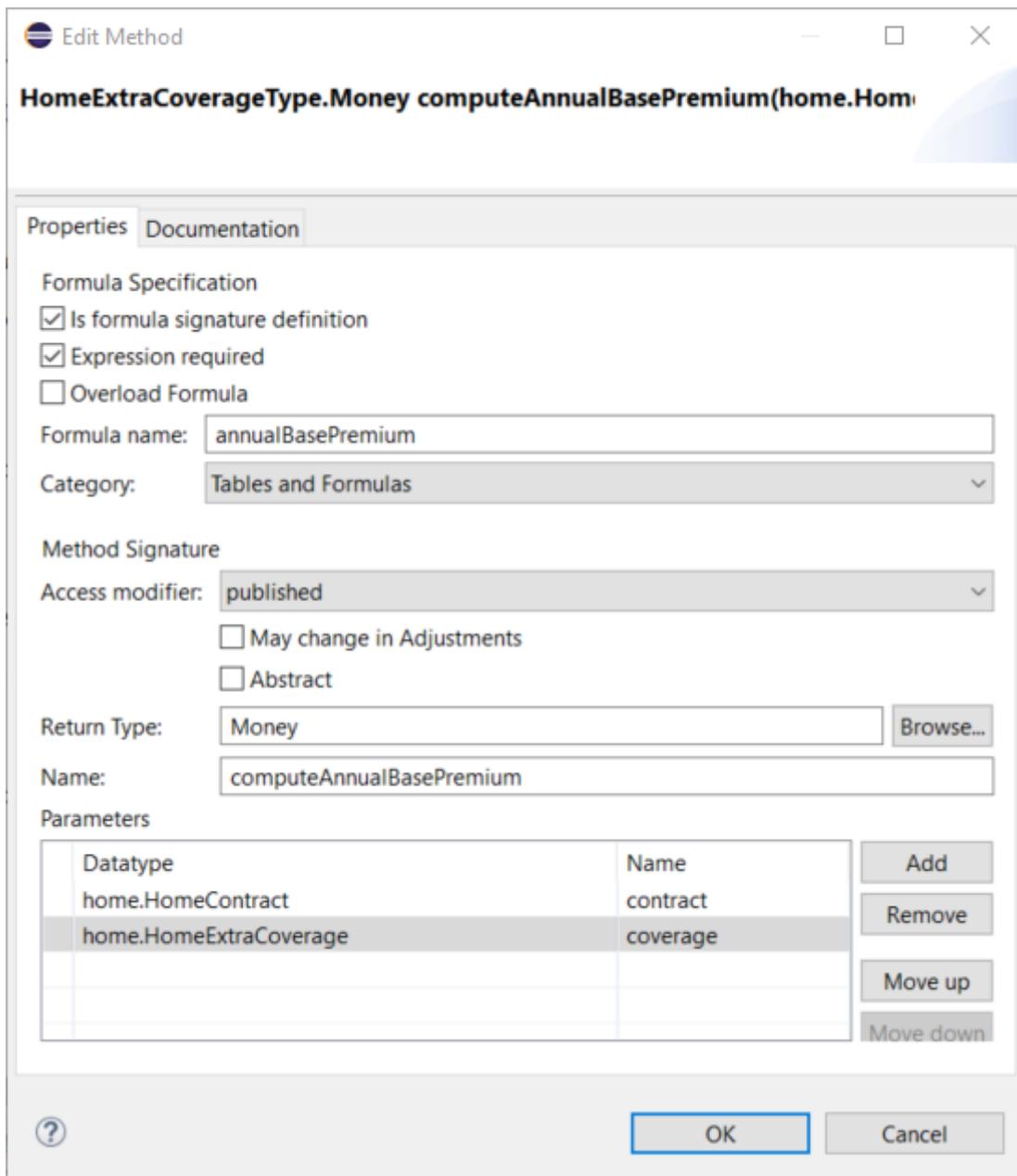


Figure 94. Dialog box for Defining a Formula Signature

Close the dialog box and save everything. The `HomeExtraCoveragesType` now now includes a `computeAnnualBasePremium(...)` method to compute the base premium.

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.

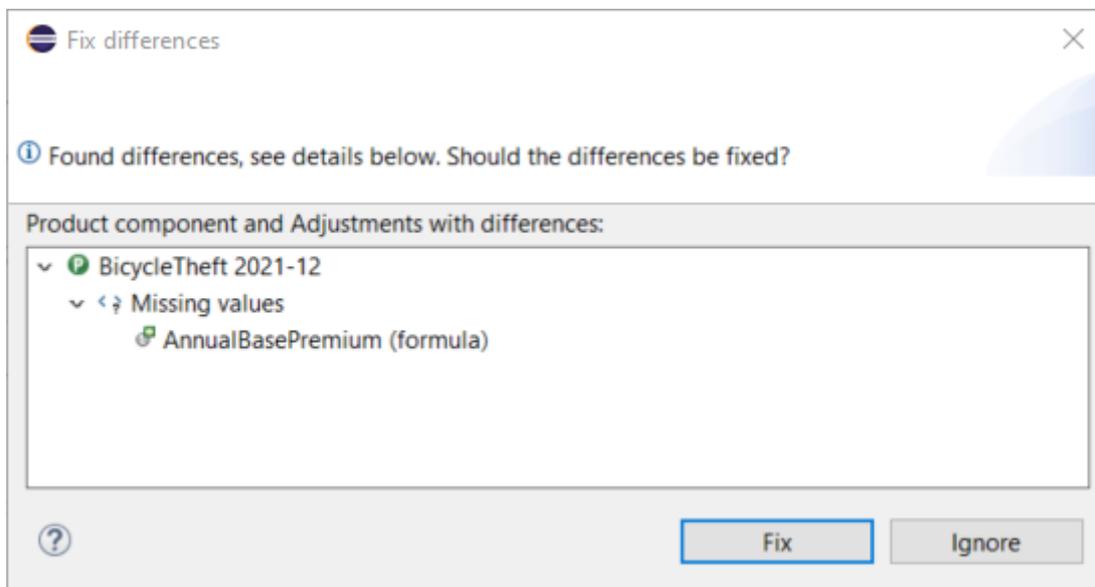


Figure 95. Fix Differences Dialog

Click *Fix* to confirm that the formula should be added. In the *Tables and Formulas* section, the premium rate formula will be displayed, as yet 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:

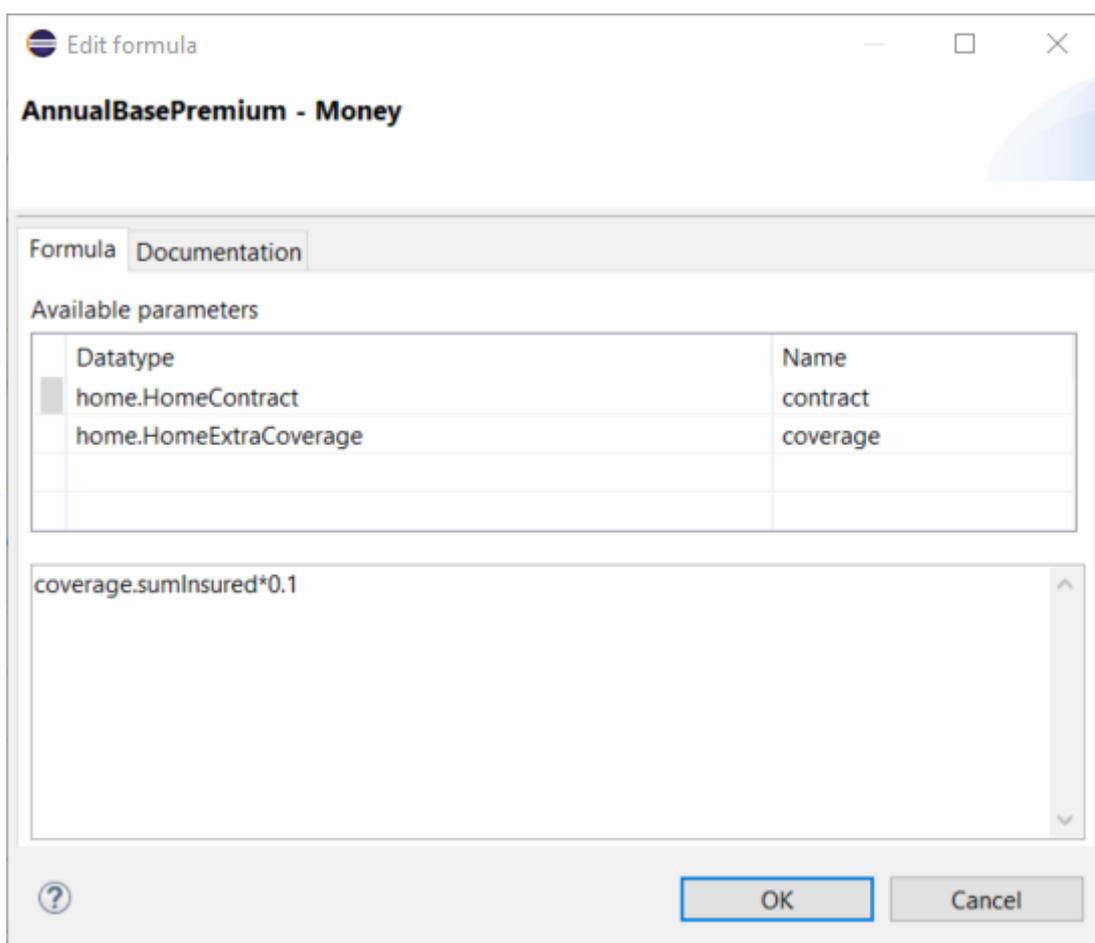


Figure 96. Creating a Formula

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.

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 package `org.faktorips.tutorial.productdata.internal.coverage` in the Java source folder `src/main/resources` [6]. The following code shows the generated `computeAnnualBasePremium(...)` method for the bicycle theft coverage.

6 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 HomeContract contract, final
HomeExtraCoverage coverage)
    throws FormulaExecutionException {
    try {
        return coverage.getSumInsured().multiply(Decimal.valueOf("0.1"),
RoundingMode.HALF_UP);
    } catch (Exception e) {
        StringBuilder parameterValues = new StringBuilder();
        parameterValues.append("contract=");
        parameterValues.append(contract == null ? "null" : contract.toString());
        parameterValues.append(", ");
        parameterValues.append("coverage=");
        parameterValues.append(coverage == null ? "null" : coverage.toString());
        throw new FormulaExecutionException(toString(), "coverage.sumInsured*0.1",
parameterValues.toString(), e);
    }
}
```

Note: Faktor-IPS provides different code generator options for formula compilation. You can use the `formulaCompiling` property in the ".ipsproject" file (or via the context menu > Properties > Faktor-IPS Code Generator of the respective project) to specify whether formulas are generated in XML and retrieved at runtime (`value="XML"`), whether subclasses of the product class are generated for product components that contain formulas (these subclasses override the corresponding methods) (`value="Subclass"`), or whether both options are generated (`value="Both"`).

Change the `formulaCompiling` property in the `faktorips-tutorial-hausratprodukte` project from `Both` to `Subclass`.

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:

```
/**
 * @generated NOT
 */
@IpsGenerated
public Money computeAnnualBasePremium() {
    return getHomeExtraCoverageType().computeAnnualBasePremium(getHomeContract(),
this);
}
```

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, 0);
    HomeBaseCoverage baseCoverage = getHomeBaseCoverage();
    baseCoverage.computeAnnualBasePremium();
    annualBasePremium = annualBasePremium.add(baseCoverage.getAnnualBasePremium());
    // Iterate over extra coverages and sum their base premiums
    for (HomeExtraCoverage coverage : getHomeExtraCoverages()) {
        annualBasePremium =
annualBasePremium.add(coverage.computeAnnualBasePremium());
    }
}
```

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

```
@Test
public void testComputePremiumBicycleTheft() {
    // Create a new HomeContract with the products factory method
    HomeContract contract = compactProduct.createHomeContract();
    // Set contract attributes
    contract.setSumInsured(Money.euro(60_000));
    // Get ExtraCoveragetype BicycleTheft
    // (To make it easy, we assume that this is the first one)
    HomeExtraCoverageType coverageType =
compactProduct.getHomeExtraCoverageType(0);
    // Create extra coverage
    HomeExtraCoverage coverage = contract.newHomeExtraCoverage(coverageType);
    // compute AnnualBasePremium and check it
    coverage.computeAnnualBasePremium();
}
```

```
// Coverage.SumInsured = 1% from 60,0000, max 5.000 => 600
// Premium = 10% from 600 = 60
assertEquals(Money.euro(60, 0), coverage.computeAnnualBasePremium());
}
```

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".

Teil 3: Testen mit Faktor-IPS

Überblick

Das Testen von Software ist in jedem Entwicklungsprojekt mit erheblichen Kosten verbunden. Besonders aufwändig ist die manuelle Durchführung von Regressionstests. Automatisierte Regressionstests gelten daher seit langem als Best Practice.

Im Java-Umfeld wird dazu seit vielen Jahren sehr erfolgreich JUnit verwendet. Die Tests werden dabei vom Entwickler in Java geschrieben. Die Ausführung der Testfälle kann direkt in der Java-Entwicklungsumgebung erfolgen. Ebenso kann man JUnit in gängige Buildwerkzeuge wie Gradle und Maven integrieren.

Auch in Faktor-IPS Projekten können wir JUnit verwenden, da Faktor-IPS testbaren Java Sourcecode generiert. Darüber hinaus bietet Faktor-IPS eine eigene, weitergehende Testunterstützung. Diese umfasst sowohl die Definition als auch die Ausführung von Testfällen.

Dieses Tutorial erläutert zunächst die zugrunde liegenden Konzepte. Danach wird die Funktionsweise anhand eines Beispiels ausführlich demonstriert. Zum Abschluss wird gezeigt, wie sich die Ausführung von Faktor-IPS Tests in Buildwerkzeuge integrieren lässt.

Konzeptionelle Grundlagen

In JUnit werden Testdaten i.d.R. unmittelbar im Java-Sourcecode erzeugt. Es gibt keine Trennung zwischen der Testlogik und den Testdaten. Das führt dazu, dass die gleiche Testlogik für unterschiedliche Testdaten nicht wiederverwendbar ist. Die Definition von Testfällen verlangt Java-Kenntnisse und ist dadurch auf Java-Entwickler beschränkt.

Für fachliche Funktionen wie die Beitragsberechnung ist die Trennung von Testdaten und Testlogik von erheblichem Vorteil, da es i.d.R. sehr viele Testfälle gibt, die sich nur bzgl. der Testdaten unterscheiden. Die folgende Tabelle verdeutlicht dies anhand von drei Testfällen für die Beitragsberechnung der Hausratversicherung.

Parameter	Testfall 1	Testfall 2	Testfall 3
Produkt	<i>HR-Kompakt 2019-07</i>	<i>HR-Optimal 2019-07</i>	<i>HR-Optimal 2019-07</i>
Zusatzdeckungen	<i>HRD-Fahrraddiebstahl 2019-07</i> <i>HRD-Ueberspannung 2019-07</i>	<i>HRD-Fahrraddiebstahl 2019-07</i> <i>HRD-Ueberspannung 2019-07</i>	<i>HRD-Ueberspannung 2019-07</i>
Zahlweise	jährlich	jährlich	jährlich
Postleitzahl	81673	81673	81673
Versicherungssumme	60.000 EUR	60.000 EUR	100.000EUR
Erwartete Ergebnisse			

Parameter	Testfall 1	Testfall 2	Testfall 3
NettobeitragZw	196,00 EUR	208,00 EUR	123,60 EUR

Der Ablauf dieser Testfälle ist der gleiche:

1. Es wird ein Hausratvertrag auf Basis des angegebenen Produktes mit den angegebenen Zusatzdeckungen erzeugt und die Attribute Zahlweise, Postleitzahl und Versicherungssumme mit den Werten aus dem Testfall belegt.
2. Es wird die Beitragsberechnung ausgeführt. Hierzu wird die entsprechende Methode am Hausratvertrag aufgerufen.
3. Der NettobeitragZw des Hausratvertrags wird mit dem im Testfall hinterlegten erwarteten Wert verglichen.

In Faktor-IPS wird im Gegensatz zu JUnit die Testlogik von den Testdaten getrennt, indem man zwischen Testfalltypen und Testfällen unterscheidet. Ein Testfalltyp definiert den Ablauf und die Struktur der Testdaten, ein Testfall ist eine Ausprägung eines Testfalltyps mit konkreten Testdaten. Die Testdaten beschreiben alle notwendigen Eingangswerte für den Test und die erwarteten Ergebnisse.

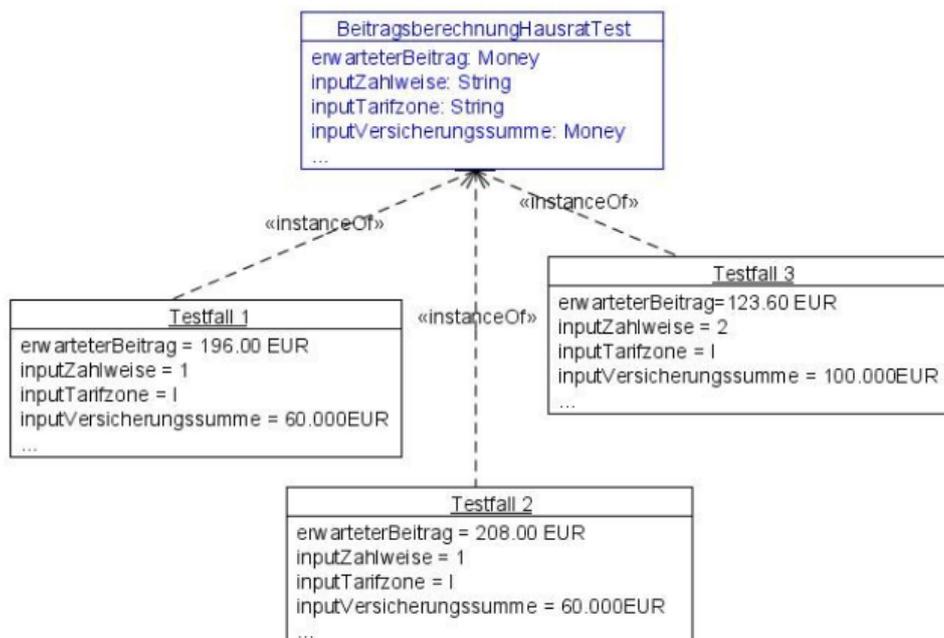


Figure 97. Testfälle als Instanzen vom Testfalltyp BeitragsberechnungHausratTests

Im objektorientierten Sinn entspricht der Testfalltyp einer Klasse und der Testfall einer Instanz eines Testfalltypen. Diese Aufteilung unterstützt auf einfache Weise eine Rollenverteilung bei der Testentwicklung. Testfalltypen werden vom Softwareentwickler bereitgestellt. Sie erstellen die Struktur und programmieren die Testlogik. Anwender der Fachabteilungen können nun auf Basis der Testfalltypen konkrete Testfälle mit den Testdaten erfassen.

Testen mit Faktor-IPS am Beispiel Hausratversicherung

Als Ausgangspunkt verwenden wir die im Einführungstutorial erstellten Projekte zur Hausratversicherung [1]. Wir wollen zunächst die Beitragsberechnung von Hausratverträgen testen. Die Berechnung ist so implementiert, dass abhängig von der gewählten Versicherungssumme und des gewählten Produkts ein Grundbeitrag errechnet wird und dieser abhängig von der Tarifzone (die wiederum von der PLZ des Hausrats abhängt) mit einem Tarifzonen-Faktor multipliziert wird. Werden Zusatzdeckungen (z.B. Fahrraddiebstahl), eingeschlossen, wird ein (im Produkt konfigurierter) Anteil des Grundbeitrags aufgeschlagen (z.B. Fahrraddeckung: + 10% des Beitrags der Grunddeckung). Abhängig von der Zahlungsweise kommt ggf. ein Ratenzahlungszuschlag hinzu.

Wir erstellen im nächsten Kapitel zunächst einen Testfalltypen für die Beitragsberechnung, der die Grundlage unserer Testfälle darstellt.

1 Die fertigen Projekte können auch direkt zum Eclipse-Import heruntergeladen werden: doc.faktorzehn.org

Testfalltyp für Beitragsberechnung Hausrat

Für den Testfalltypen erweitern wir die Package-Struktur unterhalb des Sourcefolders "modell" im Projekt *faktorips-tutorial-hausratmodell* um das IPS-Package "test".

Mit dem Kontextmenü Neu ► *Testfalltyp* im Model Explorer legen wir einen neuen Testfalltypen an (alternativ können Testfalltypen auch über das Icon  in der Toolbar angelegt werden):

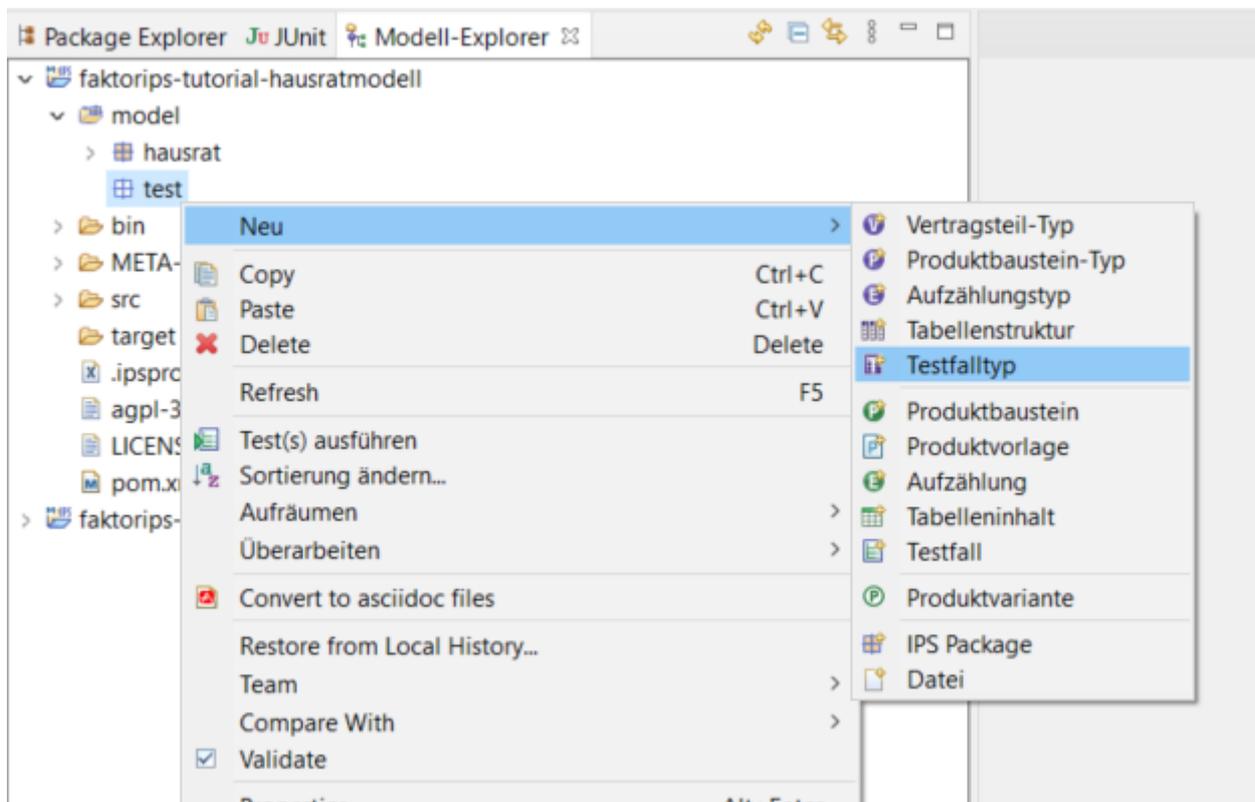


Figure 98. Neuen Testfall-Typen anlegen

Nachdem wir im folgenden Dialog den Namen des Testfalls *BeitragsberechnungHausratTest* definiert haben, öffnet sich der Editor für Testfalltypen (s. Abbildung 2). Auf der linken Seite sehen wir die (zunächst leere) Struktur des Testfalls, auf der rechten Seite jeweils die Details zu den Elementen der Struktur. Wir beginnen, die Struktur unseres Testfalls anzulegen. Testfalltypen werden in einer Baumstruktur erfasst. Zunächst werden wir also das Wurzelement definieren. Dazu rufen wir mit *Neu...* den Wizard zum Anlegen von Testparametern auf. Zunächst müssen wir die Art des Testparameters bestimmen. Es gibt drei Arten von Testparametern (zunächst unabhängig davon, ob diese als Eingabeparameter oder erwartete Ergebnisse benutzt werden):

- Vertragsteilklass
- Wert
- Regel

Für unseren Testfalltyp, mit dem wir die Beitragsberechnung von Hausratverträgen testen wollen, wählen wir als Wurzelement die Art Vertragsteiltyp:

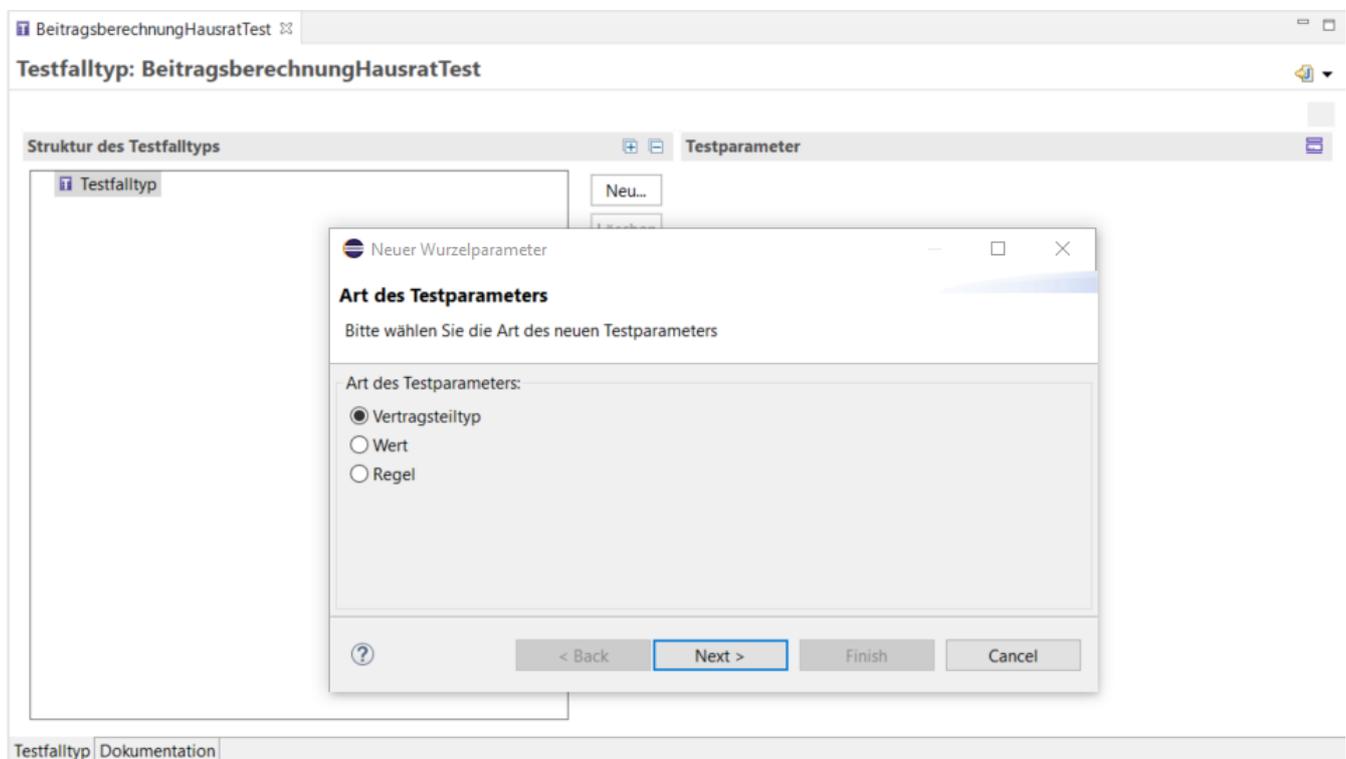
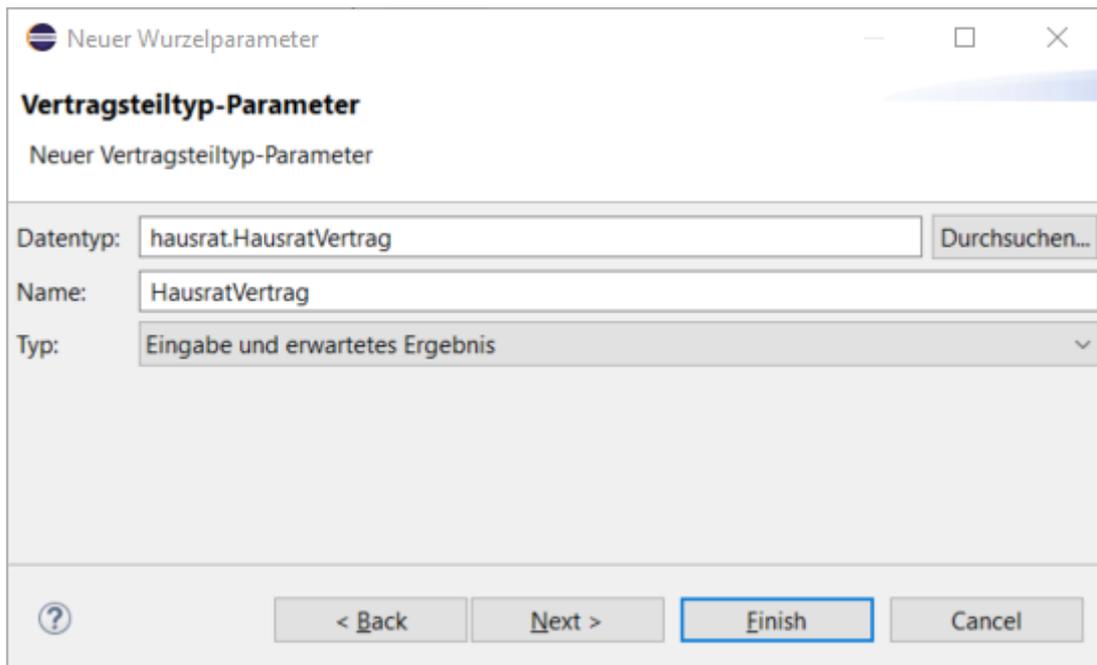


Figure 99. Testfalleditor: neues Root-Element anlegen

Nun wählen wir als Datentyp unsere Vertragsteilklass *HausratVertrag*. Der Name wird mit dem Namen des Datentypen vorbelegt, dies belassen wir für unser Beispiel auch so. Im Feld *Typ* können wir spezifizieren, welche Funktion der Parameter im Testfall hat:

- Eingabe
Attribute der Vertragsteilklass dienen ausschließlich als Eingabedaten für Testfälle
- Erwartetes Ergebnis
Attribute der Vertragsteilklass dienen ausschließlich als erwartete Werte der Testfälle
- Eingabe und erwartetes Ergebnis
Attribute der Vertragsteilklass können entweder Eingabeparameter oder erwarteter Wert sein

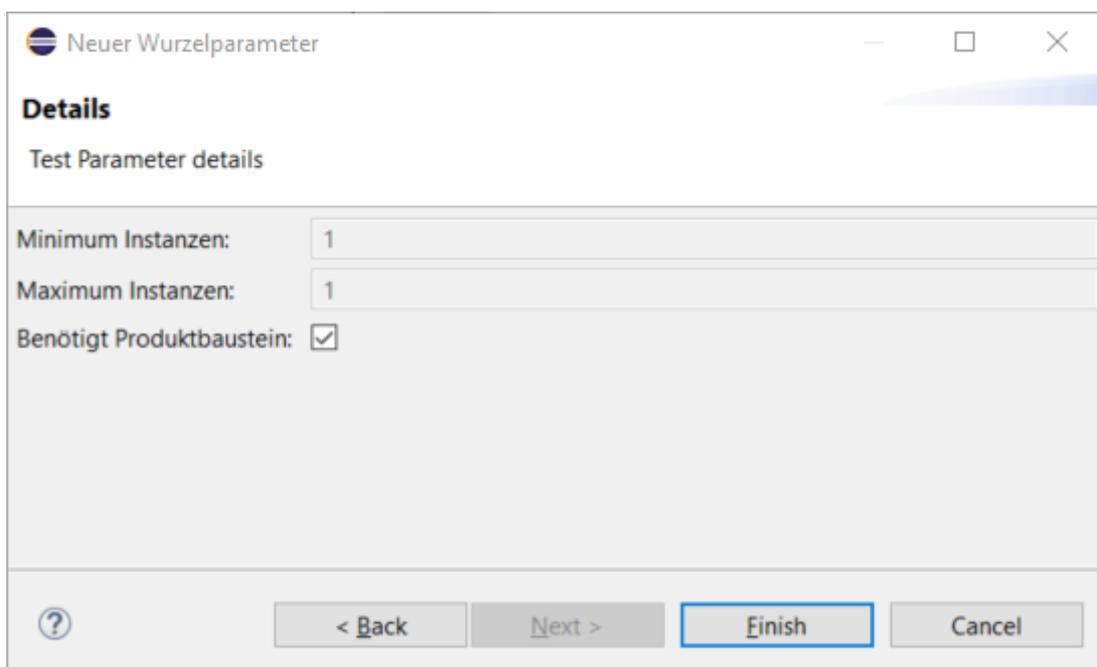
Für unser Beispiel wählen wir *Eingabe und erwartetes Ergebnis* als Parametertyp, da wir an den zu definierenden Hausratverträgen sowohl Eingabeparameter als auch das erwartete Ergebnis, in unserem Fall der berechnete Beitrag, definieren wollen:



The screenshot shows a dialog box titled 'Neuer Wurzelparameter' with the subtitle 'Vertragsteiltyp-Parameter'. Below the subtitle is the text 'Neuer Vertragsteiltyp-Parameter'. The dialog contains three input fields: 'Datentyp:' with the value 'hausrat.HausratVertrag' and a 'Durchsuchen...' button; 'Name:' with the value 'HausratVertrag'; and 'Typ:' with a dropdown menu showing 'Eingabe und erwartetes Ergebnis'. At the bottom, there are four buttons: a help icon (?), '< Back', 'Next >', and 'Finish' (highlighted with a blue border), and 'Cancel'.

Figure 100. Datentyp des Testparameters definieren

Auf der nächsten Dialogseite des Wizards kann die Kardinalität der Parameter eingeschränkt werden (bei Wurzelparametern allerdings nicht weiter, es gibt immer genau einen). Zusätzlich kann spezifiziert werden, ob der Parameter im Test durch einen Produktbaustein konfiguriert werden muss. Wir setzen die Checkbox *Benötigt Produktbaustein* und bewirken damit, dass bei der Definition der Testfälle für jeden Hausratvertrag das zu verwendende Hausratprodukt spezifiziert werden muss:



The screenshot shows the 'Details' step of the 'Neuer Wurzelparameter' dialog box. The subtitle is 'Test Parameter details'. There are three input fields: 'Minimum Instanzen:' with the value '1', 'Maximum Instanzen:' with the value '1', and 'Benötigt Produktbaustein:' with a checked checkbox. At the bottom, there are four buttons: a help icon (?), '< Back', 'Next >', and 'Finish' (highlighted with a blue border), and 'Cancel'.

Figure 101. Kardinalität und Produktabhängigkeit festlegen

Die Kardinalität und das Kennzeichen, ob eine Konfiguration durch einen Produktbaustein

notwendig ist, kann natürlich auch noch nachträglich direkt im Testfalltyp-Editor geändert werden.

Nun gilt es zu spezifizieren, welche Attribute zu testen sind, und welche als Eingabeparameter dienen. Zunächst werden wir das Attribut `nettobeitragZw` als erwartetes Ergebnis der Testfälle anlegen. Hierzu wählen wir in der Strukturansicht (links) die Beziehung zur Vertragsteilklassse `HausratVertrag` aus und legen auf der rechten Seite mit *Neu...* ein neues Testattribut an, welches auf einer Vertragsklasse basiert:

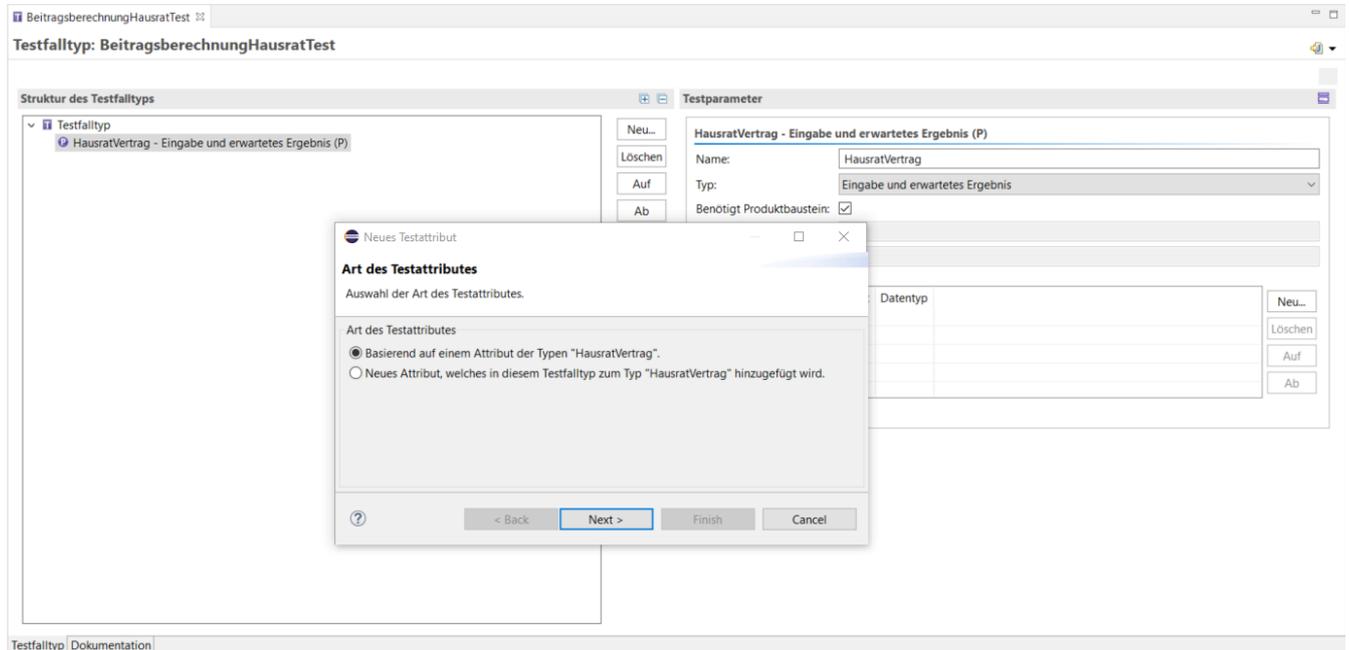


Figure 102. Testattribut anlegen

Im folgenden Dialog können die Attribute der Vertragsteilklassse ausgewählt und in den Testfalltyp übernommen werden.

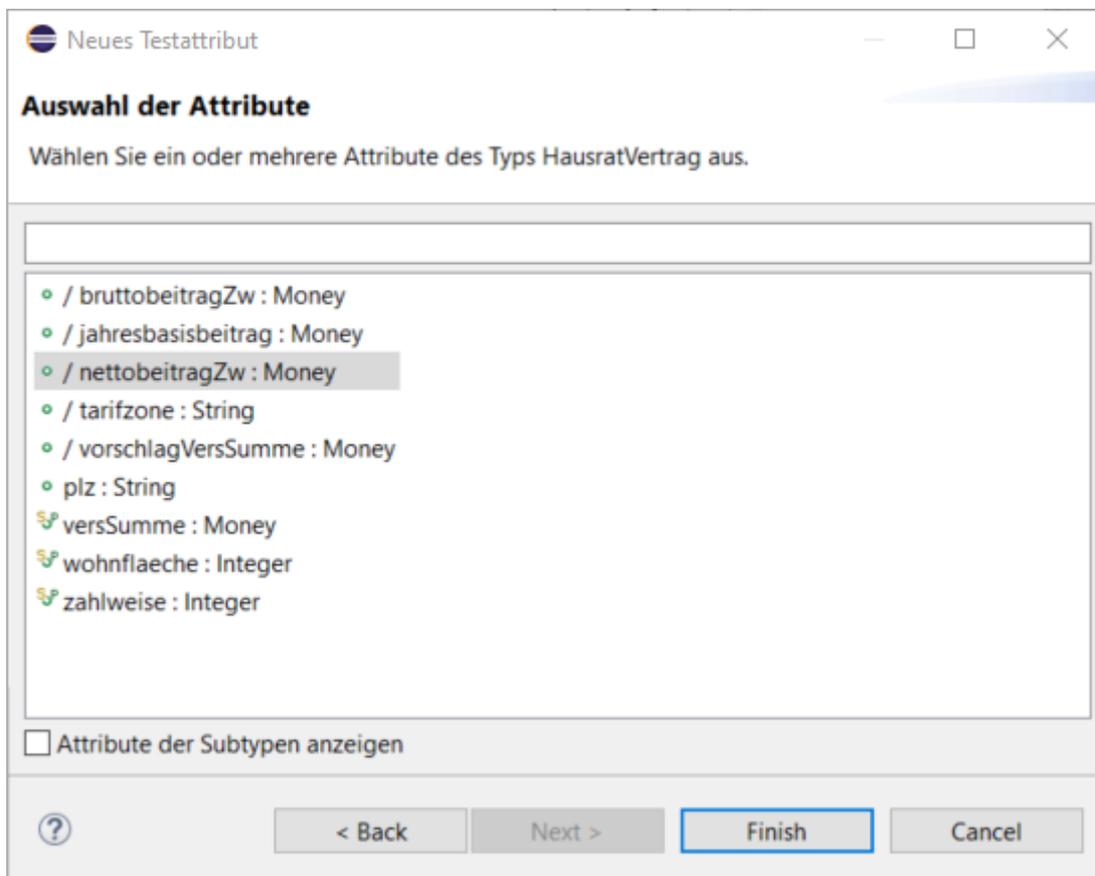


Figure 103. Testattribut auswählen

In unserem Fall erfassen wir die Parameter *nettobeitragZw*, *versSumme*, *plz* und *zahlweise*. Danach werden auf der Detail-Seite des Editors die Attribute und ihr Typ angezeigt. Das abgeleitete Attribut */nettobeitragZw* wird automatisch mit dem Typ *Erwartetes Ergebnis* vorbelegt, die weiteren Attribute dienen uns als Eingabeparameter.

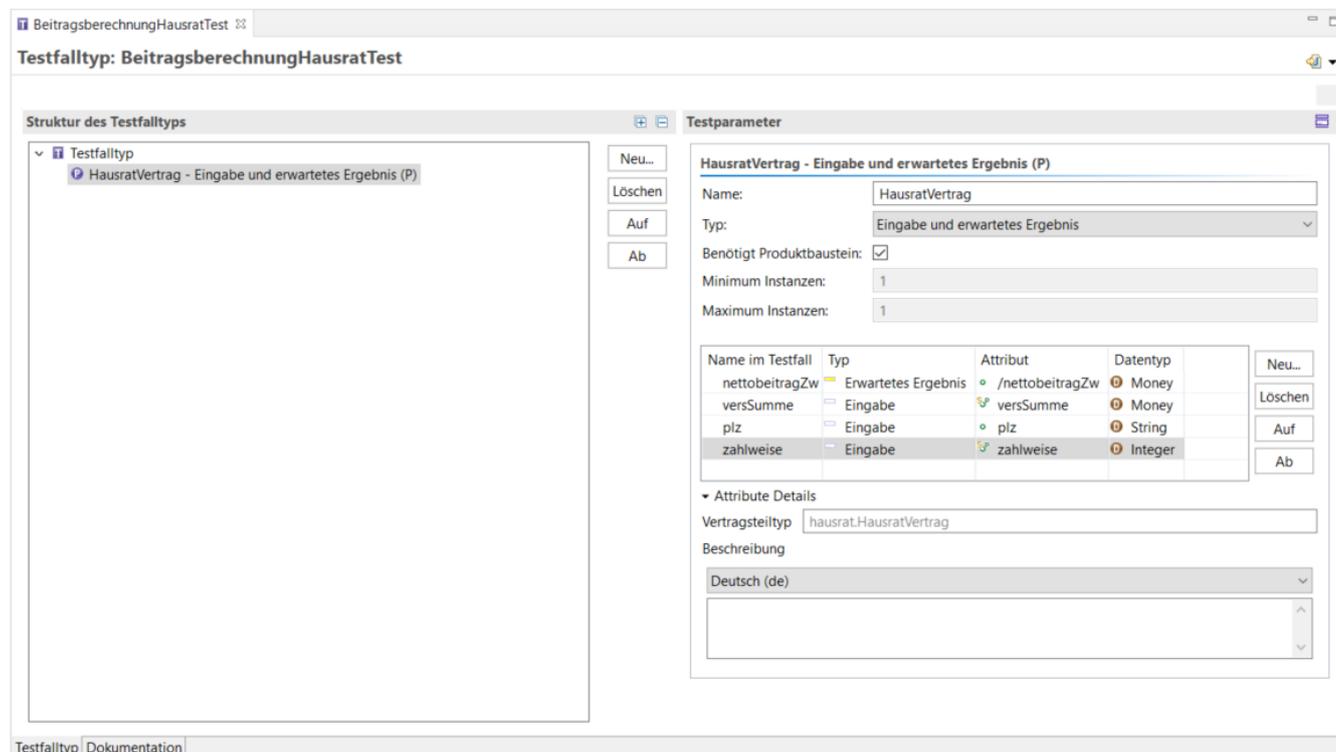


Figure 104. Testtype-Editor mit Testattributen des Hausratvertrags

Jetzt erweitern wir die Struktur unseres Testfalltyps um alle weiteren notwendigen Elemente. Wir nehmen die Vertragsteilklassen *HausratGrunddeckung* und *HausratZusatzdeckung* in den Testfalltyp auf. Dazu markieren wir in der Struktur das Element *HausratVertrag* und erzeugen jeweils mit *Neu...* die Beziehungen *HausratGrunddeckung* und *HausratZusatzdeckungen*. Für *HausratGrunddeckung* wählen wir als Typ *Eingabe*, als Kardinalität 1..1 und setzen die Checkbox *Benötigt Produktbaustein*. Für die *HausratZusatzdeckungen* wählen wir als Type *Eingabe*, als Kardinalität 0..* und setzen ebenfalls die Checkbox *Benötigt Produktbaustein*. Zur Erinnerung: Die Auswahl der Checkbox bedeutet, dass die jeweilige Vertragsteilkategorie im Testfall durch einen Produktbaustein konfiguriert werden muss. Wir werden dies später sehen, wenn wir einen Testfall anlegen.

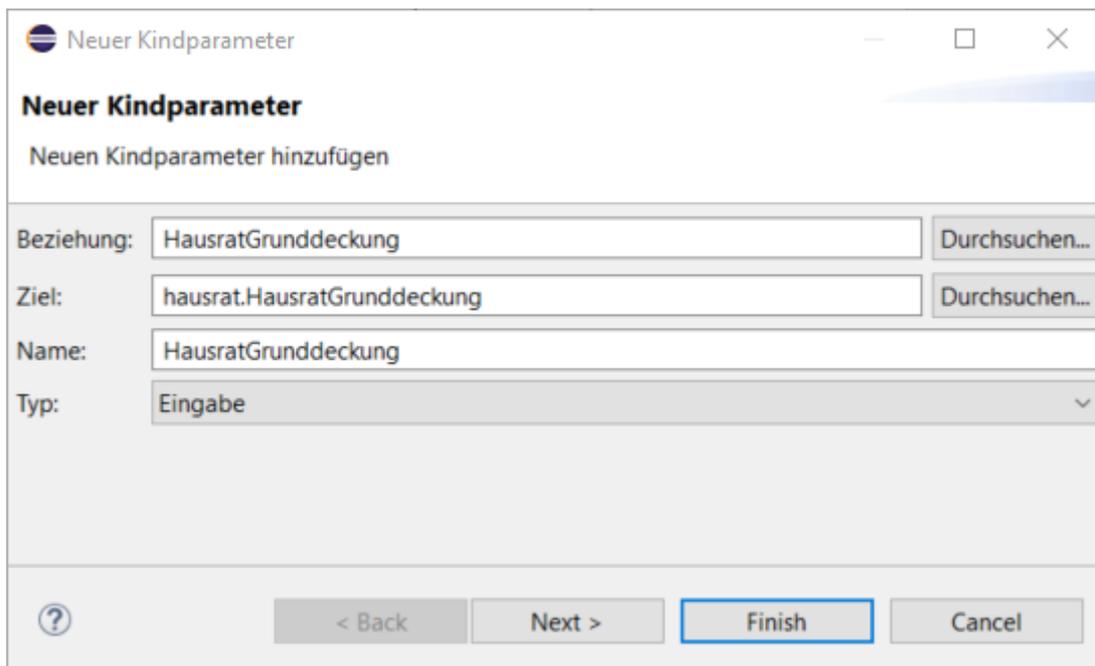


Figure 105. *HausratGrunddeckung* als Kind-Eingabeparameter von *Hausratvertrag* definieren.

Danach sollte der Testfalltyp-Editor so aussehen:

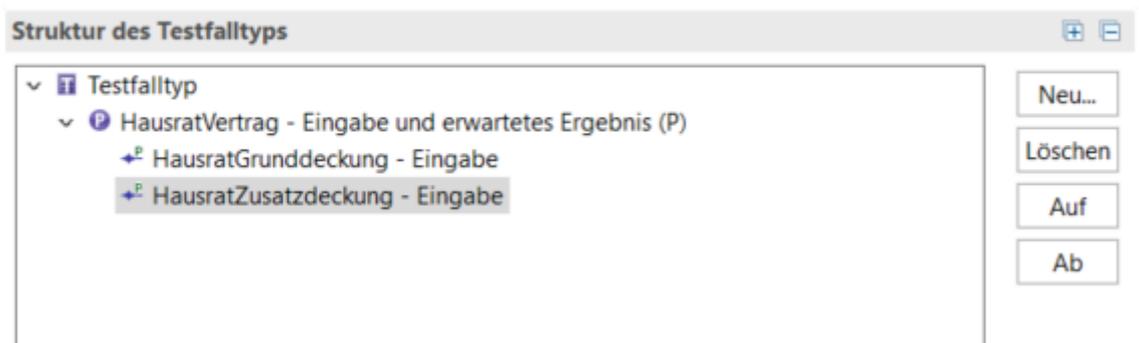


Figure 106. Beziehung von *HausratVertrag* zu *HausratZusatzdeckung*

Damit haben wir definiert, dass mit unserem Testfalltyp eine Instanz eines Hausratvertrags, mit einer Grunddeckung und beliebig vielen Zusatzdeckungen getestet werden kann. Jede Vertragsteilkategorie muss sich auf einen konkreten Produktbaustein beziehen.

Wenn wir den Testfalltypen speichern, wird im Source-Verzeichnis des Projekts *faktorips-tutorial-hausratmodell* im Package `org.faktorips.tutorial.hausrat.model.test` eine zugehörige Java-Klasse

generiert, die den Namen des Testfalltypen trägt, und in der wir die Testlogik implementieren. Wechseln wir auf den Package-Explorer und schauen uns zunächst die Struktur der generierten Klasse genauer an:

```
public class BeitragsberechnungHausratTest extends IpsTestCase2 {
    //...
    private HausratVertrag inputHausratVertrag;

    private HausratVertrag erwartetHausratVertrag;

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

    public void executeAsserts(IpsTestResult result) {
        // begin-user-code
        // TODO : Hier muessen die durchzufuehrenden Pruefungen implementiert werden.
        throw new RuntimeException(
            "Keine Pruefungen vorhanden. Diese muessen in der Java-Klasse, die den
            Testfalltyp repraesentiert, implementiert werden.");
        // end-user-code
    }
}
```

- Membervariablen `inputHausratVertrag` und `erwartetHausratVertrag` vom Typ `HausratVertrag`:
Da wir den Root-Parameter `HausratVertrag` als "Erwartetes Ergebnis und Eingabeparameter" festgelegt haben, sind zwei entsprechende Instanzvariablen angelegt worden. `inputHausratVertrag` enthält die Eingabewerte des Testfalls, gemäß der Definition der Eingabewerte im Testfalltyp. `erwartetHauratVertrag` enthält die im Testfall erwarteten Ergebnisse (gemäß der Definition im Testfalltyp). Zur Testfalllaufzeit haben wir so zwei Instanzen von `HausratVertrag`, die wir miteinander vergleichen können.
- leere Methode `executeBusinessLogic()`:
In dieser Methode wird die zu testende Geschäftslogik aufgerufen. Zum Beispiel durch Aufruf einer Methode auf den Eingabeobjekten (hier `inputHausratVertrag`). Die Methode wird ausgeführt, bevor die Methode `executeAsserts(...)` aufgerufen wird.
- Testmethode `executeAsserts()`:
Hier wird der Vergleich von Ist- und Sollwerten implementiert. Zunächst ist hier eine Default-Implementierung generiert, die den Test fehlschlagen lässt, um eine Implementierung durch den Entwickler zu erzwingen.

Bevor wir die Klasse `BeitragsberechnungHausratTest` abschließend implementieren, legen wir im nächsten Kapitel einen Testfall an, der auf unserem Testfalltyp basiert.

Testfall anlegen

Im Projekt `faktorips-tutorial-hausratprodukte` legen wir unterhalb von `produkt Daten` ein neues IPS Package `test` an, in dem wir unsere Testfälle ablegen. Dazu wechseln wir wieder in den Model

Explorer, markieren das Verzeichnis `produkt` und wählen über das Kontextmenü `Neu ▶ IPS Package` an. Wir legen mit `Neu ▶ Testfall` im Kontextmenü einen neuen Testfall an. Nun wählen wir im Wizard zur Testfallanlage den zuvor angelegten Testfalltyp `test.BetragsberechnungHausratTest` aus und geben dem Testfall einen Namen.

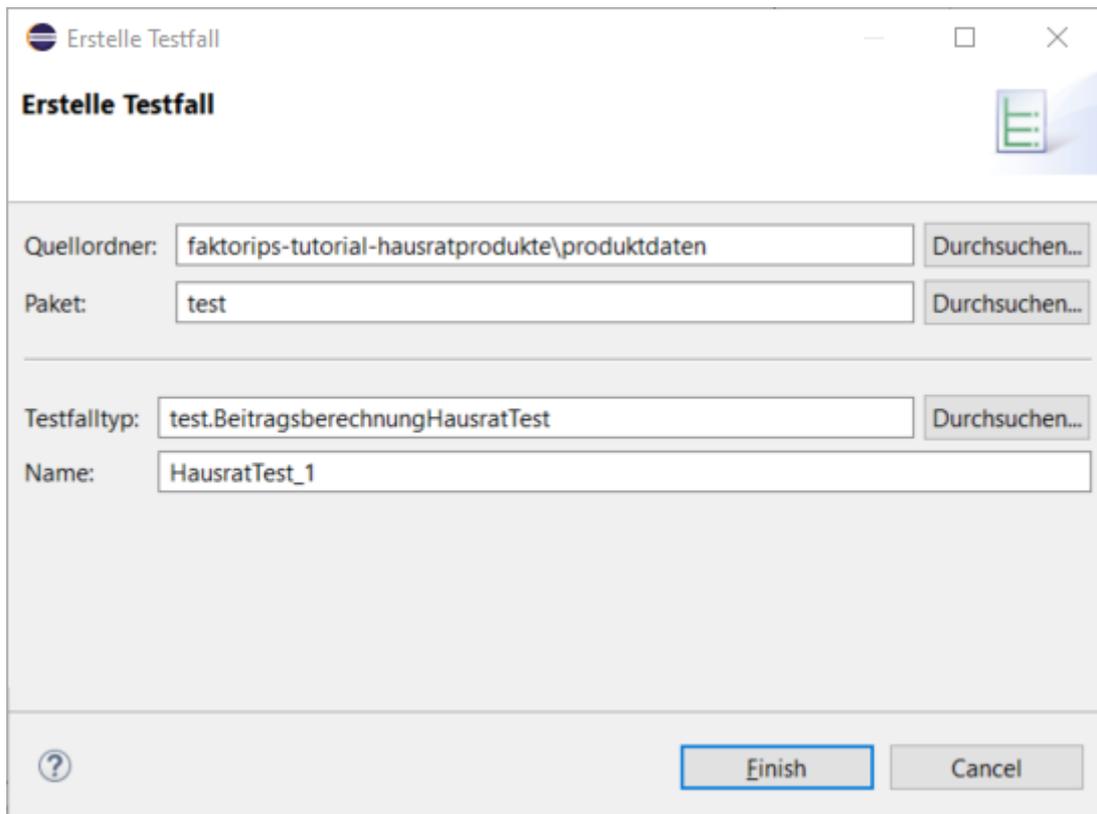


Figure 107. Neuen Testfall anlegen

Nun öffnet sich der Testfalleditor: Auf der linken Seite befindet sich die Struktur des Testfalls. Sie entspricht der Struktur, die wir im Testfalltypen definiert haben. Auf der rechten Seite sind die Testdaten. Für jedes, im Testfalltypen definierte Attribut, ist hier ein Eingabefeld zu sehen, wobei Eingabewerte weiß und erwartete Werte gelb hinterlegt sind.



Figure 108. Testfalleditor mit Hinweis auf fehlenden Produktbaustein

Auf der linken Seite müssen wir die Vertragsteile jetzt noch durch Zuordnung von Produktbausteinen ausprägen. Mit dem Button `Produktbaustein zuordnen` wird der Produktbaustein der aktuellen Vertragsteilkategorie ausgewählt.

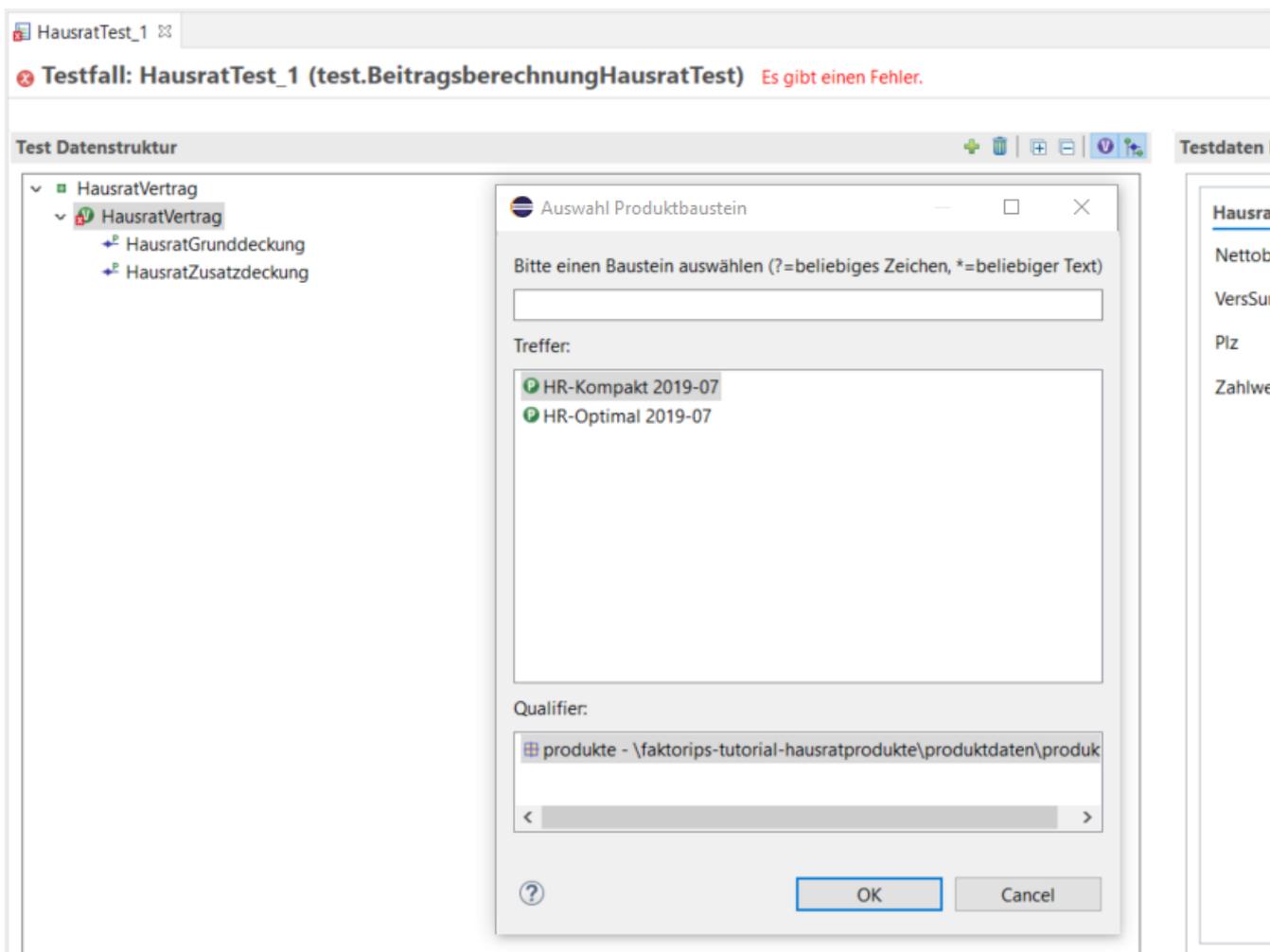


Figure 109. Auswahl eines Produktbausteins mit "Produktbaustein zuordnen"

Wir wählen für unser Beispiel den Produktbaustein HR-Kompakt 2019-07 für HausratVertrag. Danach fügen wir mit *Hinzufügen* und anschließend *Produktbaustein zuordnen* weitere Objekte hinzu: HRD-Grunddeckung-Kompakt 2019-07 für HausratGrunddeckung, und prägen zwei mal HausratZusatzdeckung, jeweils mit HRD-Fahrraddiebstahl 2019-07 und HRD-Überspannung 2019-07 aus. Danach vervollständigen wir den Testfall gemäß folgender Tabelle:

Table 6. Testfall Beitragsberechnung Hausrat

Parameter	Wert
Produkt	_HR-Kompakt 2019-07
Grunddeckungstyp	HRD-Grunddeckung-Kompakt 2019-07
Zusatzdeckungstypen	HRD-Fahrraddiebstahl 2019-07 HRD-Überspannung 2019-07
Zahlweise	1 (jährlich)
Postleitzahl	81673 (Tarifzone I)
Versicherungssumme	60.000 EUR
Nettobeitrag gemäß Zahlweise	196,00 EUR

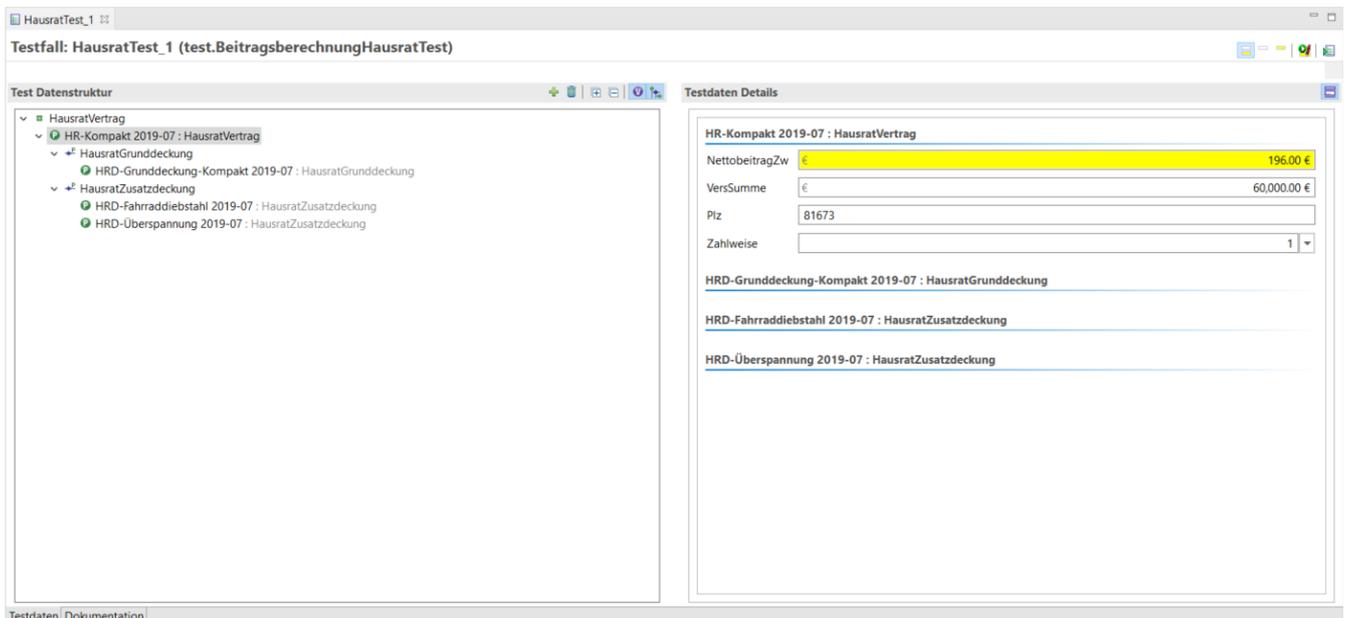


Figure 110. Erfasster Testfall

Nun starten wir die Ausführung des Testfalls mit dem Icon *Test ausführen* (🏃) in der oberen rechten Ecke des Testfall-Editors. (Der Testfall kann auf zwei Arten gestartet werden. Wir starten mit *Test ausführen*. Die Variante *Ermitteln der erwarteten Werte* sehen wir später im Kapitel "Testfälle durch Kopieren erzeugen").



Figure 111. Testfallausführung im Testrunner schlägt fehl

Der Editor verrät uns durch einen roten Balken im Titel und durch einen roten Balken im Faktor-IPS Testrunner, dass der Test fehlgeschlagen ist. In den Fehler Details sehen wir, warum. Wir werden daran erinnert, dass wir im Testfalltypen die Prüfungen noch nicht implementiert haben (an der Stelle ist eine Runtime-Exception generiert, deren Nachricht hier angezeigt wird). Holen wir dies also nach:

```
public class BeitragsberechnungHausratTest extends IpsTestCase2 {
    //...

    /**
     * Fuehrt die zu testende Geschaeftslogik aus.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeBusinessLogic() {
        // begin-user-code
    }
}
```

```

        inputHausratVertrag.berechneBeitrag();
        // end-user-code
    }

    /**
     * Fuehrt die Pruefungen (Asserts) aus, d.h. vergleicht die erwarteten Werte mit
den
     * tatsaechlichen Ergebnissen.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeAsserts(IpsTestResult result) {
        // begin-user-code
        assertEquals(erwartetHausratVertrag.getNettobeitragZw(),
            inputHausratVertrag.getNettobeitragZw(),
            result);
        // end-user-code
    }

    //...
}

```

In der Methode `executeBusinessLogic()` rufen wir die zu testende Geschäftslogik auf: Auf der `input`-Instanz rufen wir die Methode `berechneBeitrag()` auf. Faktor-IPS sorgt dafür, dass die Instanz zur Laufzeit die Eingabewerte aus dem jeweiligen Testfall enthält. In der Methode `executeAsserts(...)` implementieren wir die Prüfungen. In unserem Fall wollen wir überprüfen, ob der erwartete Nettobeitrag mit dem berechneten Beitrag übereinstimmt. Dazu benutzen wir die `assert*`-Methoden der Klasse `IpsTestCaseBase`.

Nun führen wir unseren Testfall erneut aus. Der grüne Balken im Testfalleditor und im Testfallrunner zeigt uns, dass das erwartete Ergebnis und das berechnete Ergebnis übereinstimmen.

Machen wir die Gegenprobe und ändern das erwartete Ergebnis:

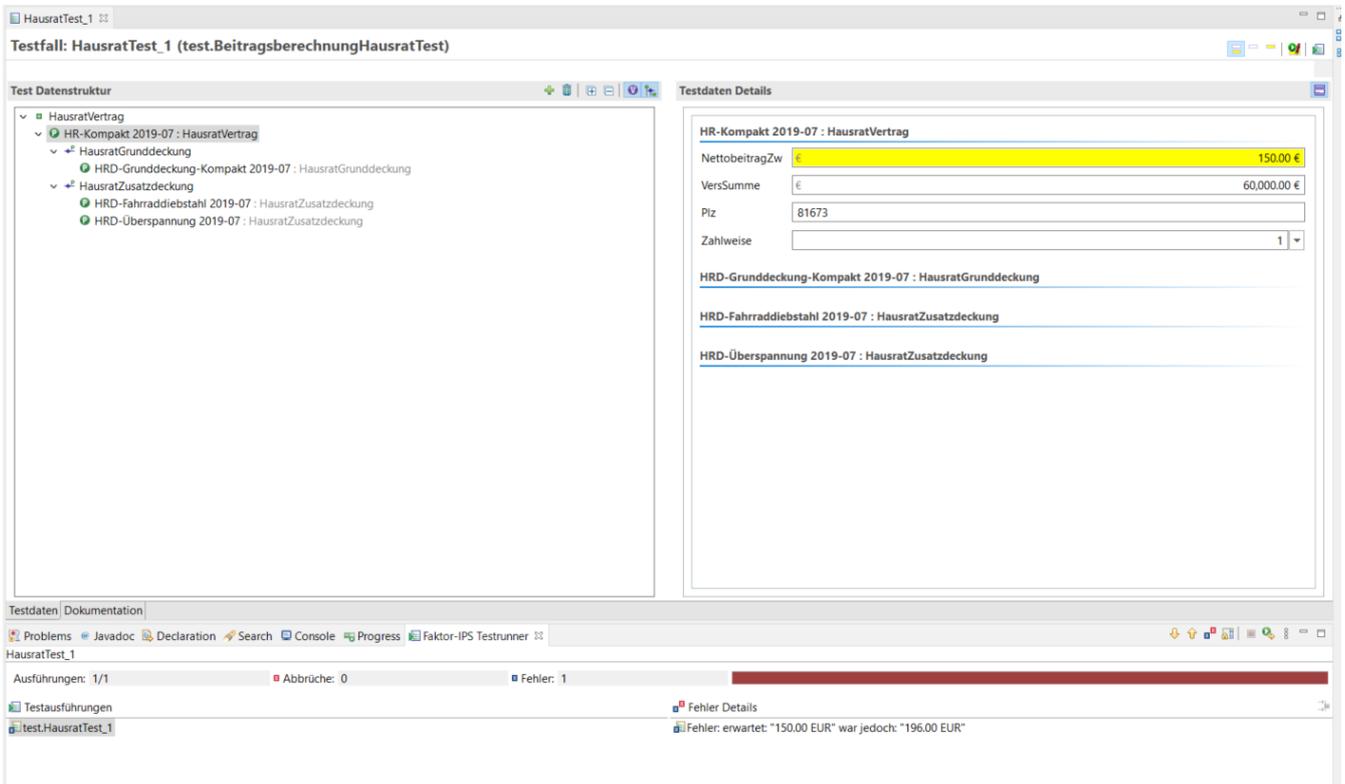


Figure 112. Testfall schlägt fehl

Der Testfall schlägt fehl. In den Fehler Details sehen wir, dass der berechnete Wert 196 EUR nicht dem erwarteten Wert entspricht. Wir sehen der Fehlermeldung aber noch nicht an, auf welches Testattribut sie sich bezieht. In unserem Fall ist dies zwar einfach, da wir nur ein erwartetes Attribut vergleichen, aber es können ja durchaus mehrere Attribute in einem Testfall verglichen werden (z.B. die einzelnen Beiträge je Deckung und der Gesamtbeitrag). Hierzu können wir den `assert*`-Statements eine Referenz auf das Attribut aus der Definition des Testfalltyps mitgeben. Als ersten weiteren Parameter übergeben wir einen String, der das Testobjekt identifiziert, als weiteren Parameter den Namen des fehlerhaften Attributes. Wenn es mehrere Instanzen eines Objekts im Test gibt, muss der Index der Instanz, beginnend bei 0, getrennt von einer Raute angefügt werden, z.B. `Zusatzdeckung#0` für die erste Instanz des Objekts `Zusatzdeckung`. Für das Attribut `nettobeitragZw` in unserem Beispiel sieht die Implementierung dann wie folgt aus:

```
/**
 * Führt die Prüfungen (Asserts) aus, d.h. vergleicht die erwarteten Werte mit den
 * tatsächlichen Ergebnissen.
 *
 * @restrainedmodifiable
 */
@Override
@IpsGenerated
public void executeAsserts(IpsTestResult result) {
    // begin-user-code
    assertEquals(erwartetHausratVertrag.getNettobeitragZw(),
        inputHausratVertrag.getNettobeitragZw(),
        result,
        "HausratVertrag",
        HausratVertrag.PROPERTY_NETTOBEITRAGZW);
    // end-user-code
}
```

}

Führen wir nun den Testfall erneut aus (weiterhin mit "falschem" erwartetem Ergebnis), wird auch das entsprechende Attribut auf der Oberfläche rot hinterlegt und die Meldung in den Fehler Details sagt uns nun auch, auf welches Attribut sie sich bezieht:

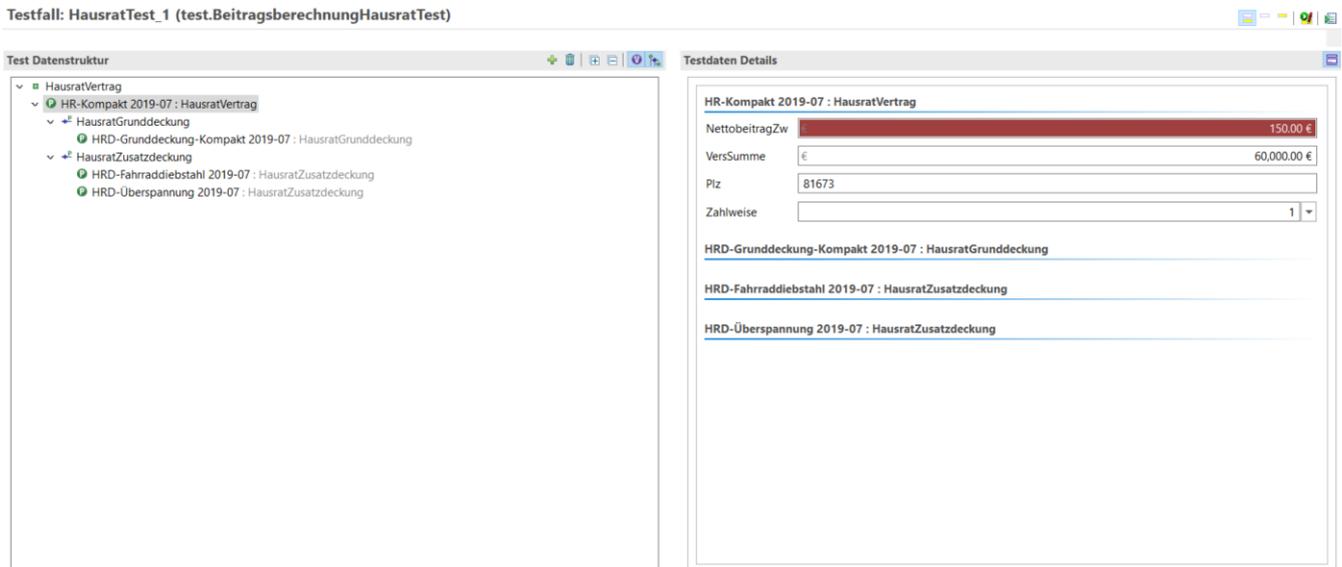


Figure 113. Markierung des fehlerhaften Attributs im Testeditor

Damit haben wir einen Testfalltypen komplett implementiert, einen Testfall angelegt und ausgeführt.

Sonderfall: Testen von abgeleiteten Attributen

Bisher haben wir die Vergleichslogik im Testfalltyp aufgrund des Vergleichs von Membervariablen der Testobjekte implementiert. In der Input-Instanz haben wir das Attribut berechnen lassen, in der Erwartet-Instanz haben wir das Attribut mit dem Testeditor eingetragen und die Attribute beider Instanzen verglichen. Es handelte sich dabei um abgeleitete Attribute für die eine Membervariable generiert wird und die erst durch den expliziten Aufruf einer Methode (hier `inputHausratVertrag.berechneBeitrag()`) berechnet werden. Diese Art abgeleiteter Attribute ("cached") können genauso wie änderbare Attribute getestet werden.

In Faktor-IPS haben wir aber auch die Möglichkeit, abgeleitete Attribute zu definieren, die bei jedem Aufruf der Getter-Methode ("on the fly") berechnet werden. Für diese Attribute wird keine Membervariable generiert. Sie stellen daher einen Sonderfall für die Testimplementierung dar, da wir in unserer Erwartet-Instanz keine Membervariable haben, die wir für einen Vergleich nutzen können.

Die Tarifzone des Hausratvertrags ist ein solches Attribut. Wir erstellen nun wie bereits beschrieben einen neuen Testfalltypen (ErmittlungTarifzoneTest), mit dem wir die Ermittlung der Tarifzone anhand der Postleitzahl überprüfen können.

Als Eingabeparameter definieren wir die Attribute `plz` der Vertragsteilklassse `HausratVertrag`. Für `HausratVertrag` setzen wir die Checkbox `Benötigt Produktbaustein`. Anstatt die Tarifzone als erwartetes Ergebnis basierend auf der Vertragsteilklassse `Vertrag` anzugeben, legen wir ein Testfall-

Attribut an, das nicht auf einer Vertragsklasse basiert:

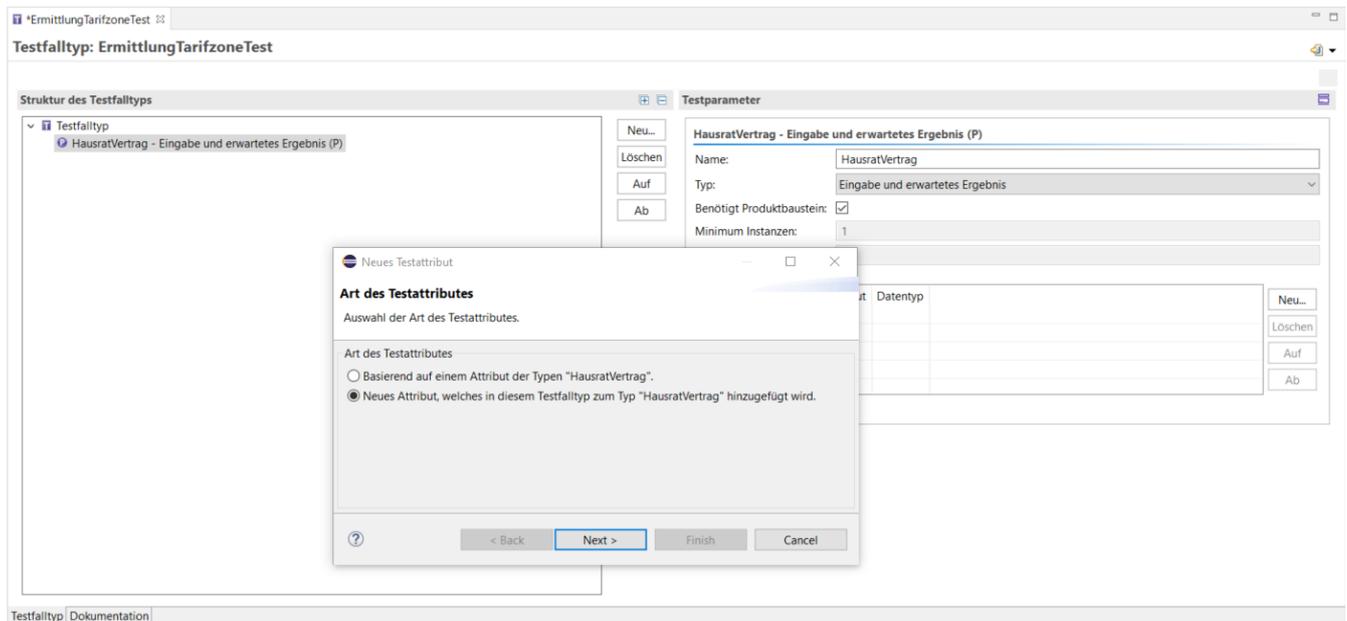


Figure 114. Testfall-Attribute anlegen, dass nicht auf einer Vertragsteilklassse basiert

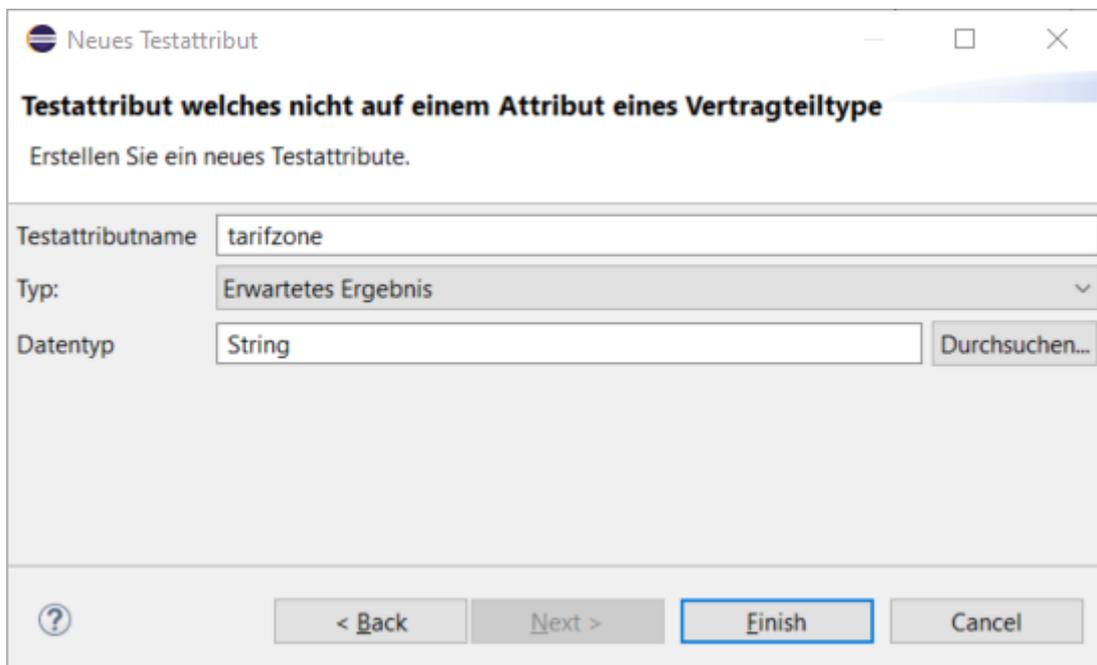


Figure 115. Attribut tarifzone definieren

In der generierten Testklasse wird nun eine Konstante generiert, mit der man den Inhalt des Attributs im Testfall lesen kann. Dieses wiederum kann man dann im assert-Statement mit dem berechneten Wert aus der Input-Instanz vergleichen:

```
public class ErmittlungTarifzoneTest extends IpsTestCase2 {  
  
    /**  
     * @generated  
     */  
    public static final String TESTATTR_HAUSRAT_VERTRAG_TARIFZONE = "tarifzone";  
}
```

```

//...

/**
 * Fuehrt die zu testende Geschaeftslogik aus.
 *
 * @restrainedmodifiable
 */
@Override
@IpsGenerated
public void executeBusinessLogic() {
    // begin-user-code
    // nichts zu tun (die zu testende Logik wird durch getter-Aufruf ausgefuehrt)
    // end-user-code
}

/**
 * Fuehrt die Pruefungen (Asserts) aus, d.h. vergleicht die erwarteten Werte mit
den
 * tatsaechlichen Ergebnissen.
 *
 * @restrainedmodifiable
 */
@Override
@IpsGenerated
public void executeAsserts(IpsTestResult result) {
    // begin-user-code
    String erwartetTarifzone = (String) getExtensionAttributeValue(
        erwartetHausratVertrag, TESTATTR_HAUSRAT_VERTRAG_TARIFZONE);

    String inputTarifzone = inputHausratVertrag.getTarifzone();
    assertEquals(erwartetTarifzone, inputTarifzone, result, "HausratVertrag",
TESTATTR_HAUSRAT_VERTRAG_TARIFZONE);
    // end-user-code
}

//...
}

```

Mit `inputHausratVertrag.getTarifzone()` wird auf die berechnete Tarifzone zugegriffen, mit `getExtensionAttributeValue(...)` auf die im Testfalltypen definierte, erwartete Tarifzone.

Zur Überprüfung legen wir einen Testfall (`ErmittleTarifzoneTest_1`) basierend auf dem neuen Testfalltyp an und befüllen die Testdaten für *Plz*. Dem HausratVertrag ordnen wir noch ein Hausratprodukt (z.B. HR-Kompakt 2019-07) zu. Gemäß der Tarifzonentabelle erwarten wir für die Postleitzahl 63066 die Tarifzone VI:

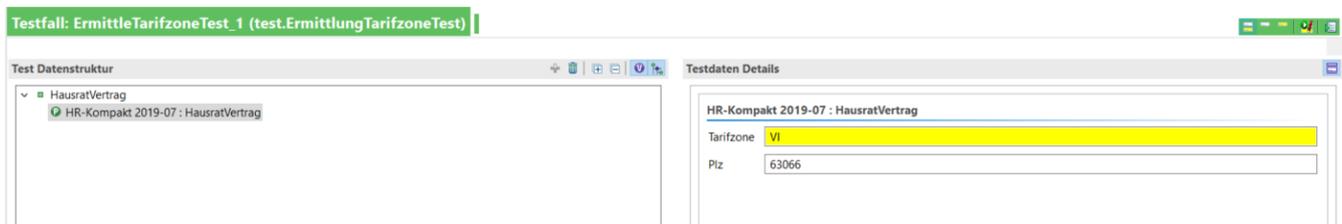


Figure 116. Testfall zur Ermittlung der Tarifzone

Nach der Ausführung des Tests bestätigt uns der grüne Balken die Korrektheit des Testfalls.

Testfälle durch Kopieren erzeugen

In dem folgenden Beispiel zeigen wir, wie wir durch einfaches Kopieren neue Tests erzeugen und anpassen können. Wir wollen einen Testfall erstellen, der statt auf dem Produkt *HR-Kompakt 2019-07* auf dem Produkt *HR-Optimal 2019-07* basiert.

Dazu markieren wir im Model Explorer den zu kopierenden Testfall *HausratTest_1* und rufen im Kontextmenü *Kopiere Testfall...* auf.

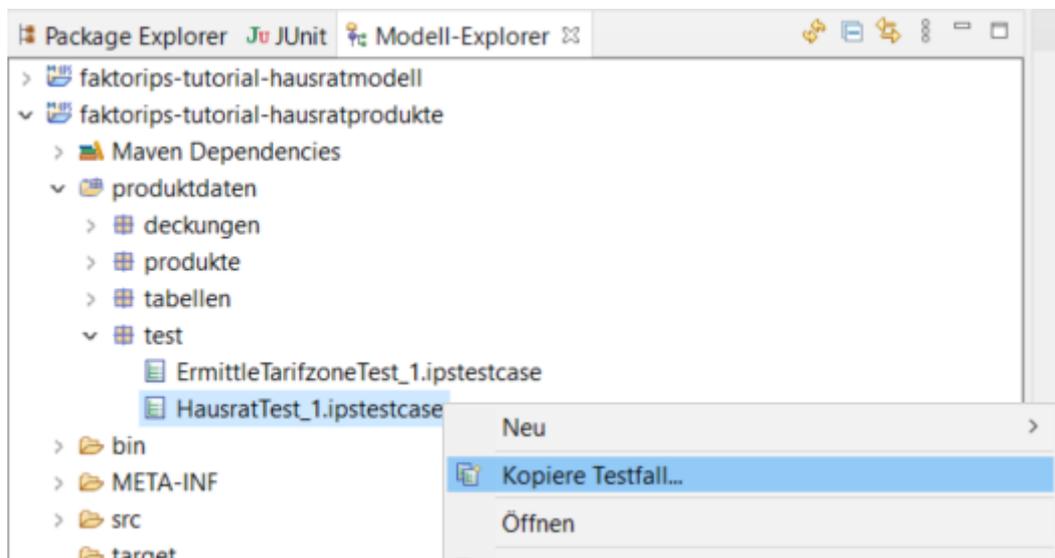


Figure 117. Test kopieren

Der Wizard führt uns durch die Anlage des zu erzeugenden Testfalls. Wir vergeben einen Namen, und entscheiden mit dem Radio-Button *Durch neue Produktbausteine*, dass wir die Produktbausteine aus dem Quelltestfall durch andere Produktbausteine (die von *HR-Optimal*) ersetzen wollen. Eingabe- und erwartete Werte wollen wir auch in den Zieltestfall übernehmen, lassen daher die beiden Checkboxes für *Übernahme der Testwerte* aktiv und bestätigen mit *Next >*.

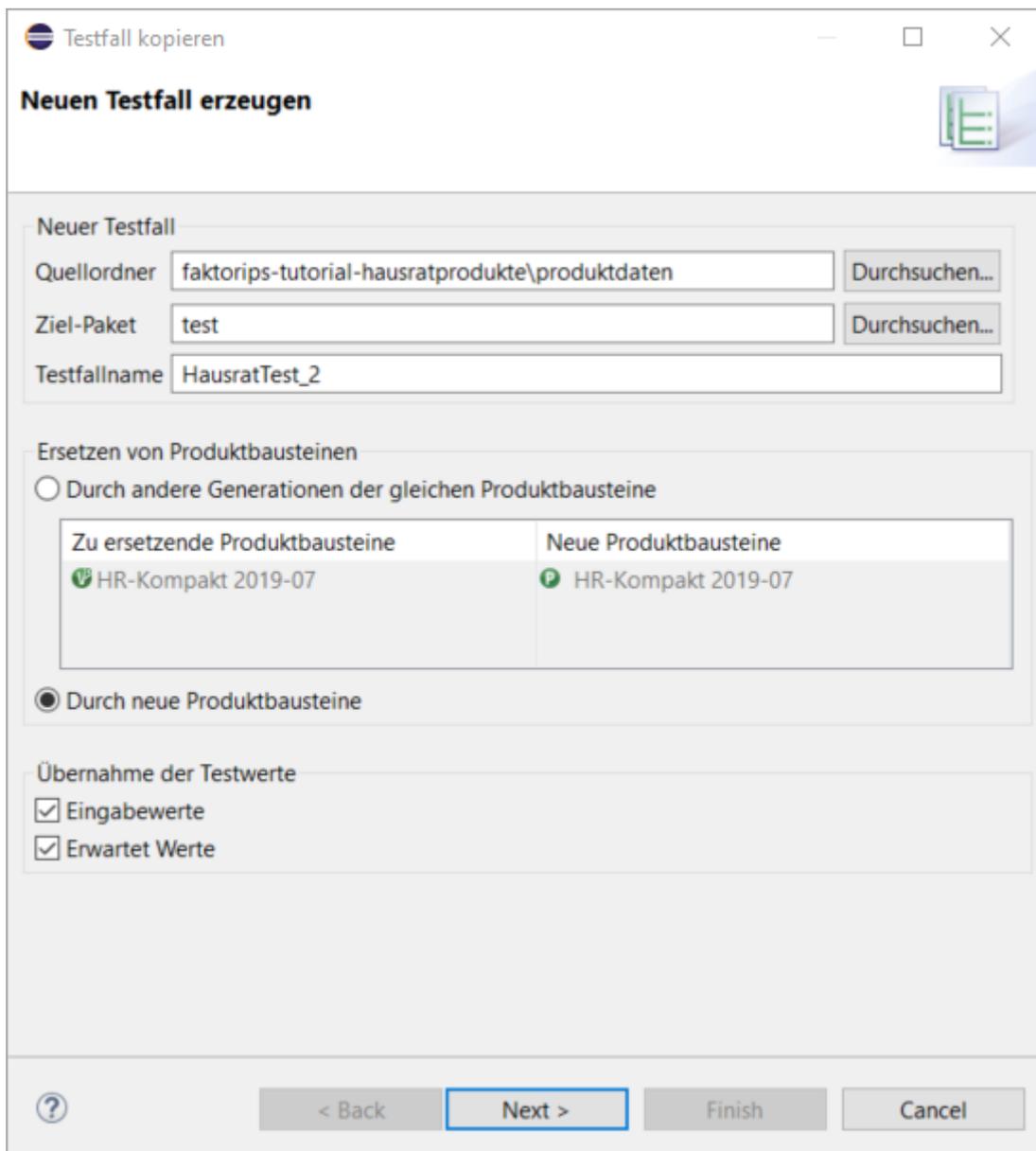


Figure 118. Kopiere Testfall

Wir haben nun die Möglichkeit, Produktbausteine abzuwählen oder durch andere zu ersetzen. Hierzu markieren wir den zu ersetzenden Baustein in der Strukturansicht auf der linken Seite. In der Liste auf der rechten Seite werden nun alle, für diese Beziehung passenden Produktbausteine angezeigt und wir wählen den neuen Produktbaustein aus. Wir ersetzen *HR-Kompakt 2019-07* durch *HR-Optimal 2019-07* und dementsprechend *HRD-Grunddeckung-Kompakt 2019-07* durch *HRD-Grunddeckung-Optimal 2019-07*.

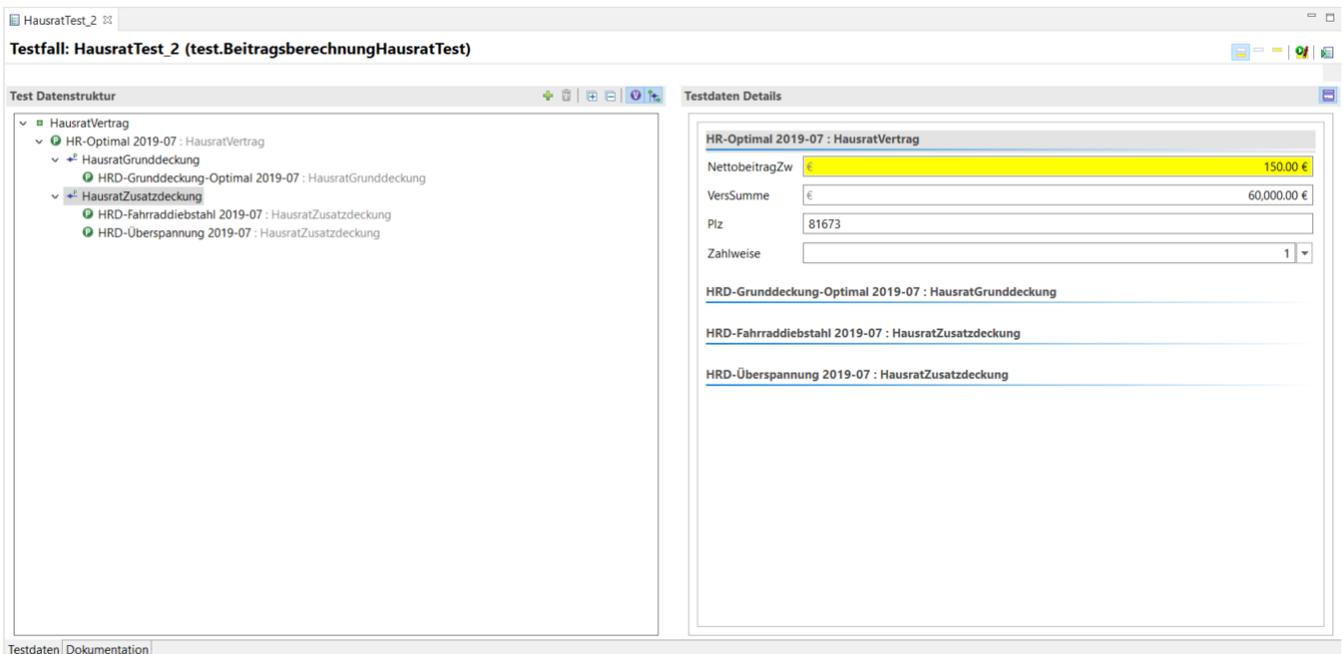


Figure 120. Kopierter Testfall

Wir lassen den Testfall laufen, da sich das Produkt HR-Optimal unter anderem durch die Beiträge vom Produkt HR-Kompakt unterscheidet, bekommen wir eine Abweichung des erwarteten Ergebnisses. Wir können aber mit einem einfachen Hilfsmittel das berechnete Ergebnis als erwartetes Ergebnis übernehmen. Hierzu benutzen wir das Icon *Ermitteln der erwarteten Werte*:

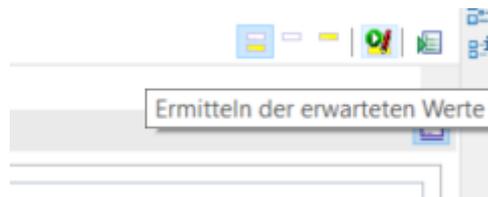


Figure 121. Ermitteln der erwarteten Werte

Der Testfall wird nun ausgeführt und die berechneten Werte werden in unseren Testfall übernommen. Somit haben wir auf einfache Weise das erwartete Ergebnis ermittelt und wieder einen korrekten Testfall.

HR-Optimal 2019-07 : HausratVertrag	
NettobeitragZw	208.00 €
VersSumme	€ 60,000.00 €
Plz	81673
Zahlweise	1 ▾
HRD-Grunddeckung-Optimal 2019-07 : HausratGrunddeckung	
HRD-Fahrraddiebstahl 2019-07 : HausratZusatzdeckung	
HRD-Überspannung 2019-07 : HausratZusatzdeckung	

Figure 122. Ermittelte Werte wurden in Testfall übernommen

JUnit-Adapter für Faktor-IPS Testfälle

Die Faktor-IPS Runtime enthält Adapter, um aus den Faktor-IPS Testfällen JUnit-Testfälle bzw. Testsuiten zu machen. So lassen sich die Faktor-IPS Testfälle auch mit Gradle oder Maven ausführen, da diese über eine entsprechende JUnit-Integration verfügen. Damit ist auch eine automatisierte Ausführung der Faktor-IPS Testfälle im Rahmen einer kontinuierlichen Integration problemlos möglich.

Mit folgendem Code können wir uns einen Adapter erstellen, der aus unseren Faktor-IPS-Testfällen eine JUnit 3/JUnit 4-Testsuite macht:

```
import org.faktorips.runtime.ClassloaderRuntimeRepository;
import org.faktorips.runtime.IRuntimeRepository;
import org.faktorips.runtime.test.IpsTestSuiteJUnitAdapter;

import junit.framework.Test;

public class HausratJUnit3Test extends IpsTestSuiteJUnitAdapter {

    public static Test suite() {
        IRuntimeRepository repositoryHausrat = ClassloaderRuntimeRepository.create(
            "org/faktorips/tutorial/produktdaten/internal/faktorips-repository-
            toc.xml");

        return createJUnitTest(repositoryHausrat.getIpsTest(""));
    }
}
```

Und mit folgendem Code erstellen wir einen Adapter, der unsere Faktor-IPS-Testfälle in JUnit 5 Dynamic Tests umwandelt:

```

import org.faktorips.runtime.ClassloaderRuntimeRepository;
import org.faktorips.runtime.IRuntimeRepository;
import org.faktorips.runtime.test.IpsTestSuiteJUnit5Adapter;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

public class HausratJUnit5Test extends IpsTestSuiteJUnit5Adapter {

    @TestFactory
    public Stream<DynamicTest> getTests() {
        IRuntimeRepository repositoryHausrat = ClassloaderRuntimeRepository.create(
            "org/faktorips/tutorial/produktdaten/internal/faktorips-repository-
toc.xml");

        return createTests(repositoryHausrat.getIpsTest(""));
    }
}

```

Wir erzeugen ein `RuntimeRepository` mit den Produktdaten, lassen uns durch den Aufruf der Methode `getIpsTest("")` eine Testsuite mit allen Testfällen aus dem Repository geben. Für JUnit 3 und JUnit 4 erzeugt die Methode `createJUnitTest(...)` der Klasse `IpsTestSuiteJUnitAdapter` daraus eine JUnit-Testsuite. Für JUnit 5 gibt die Methode `createTests(...)` der Klasse `IpsTestSuiteJUnit5Adapter` einen Stream von `DynamicTests` zurück. Führen wir die Testklassen mit dem jeweiligen JUnit-Testrunner aus, sehen wir, wie unsere Faktor-IPS-Testfälle von JUnit ausgeführt und interpretiert werden.

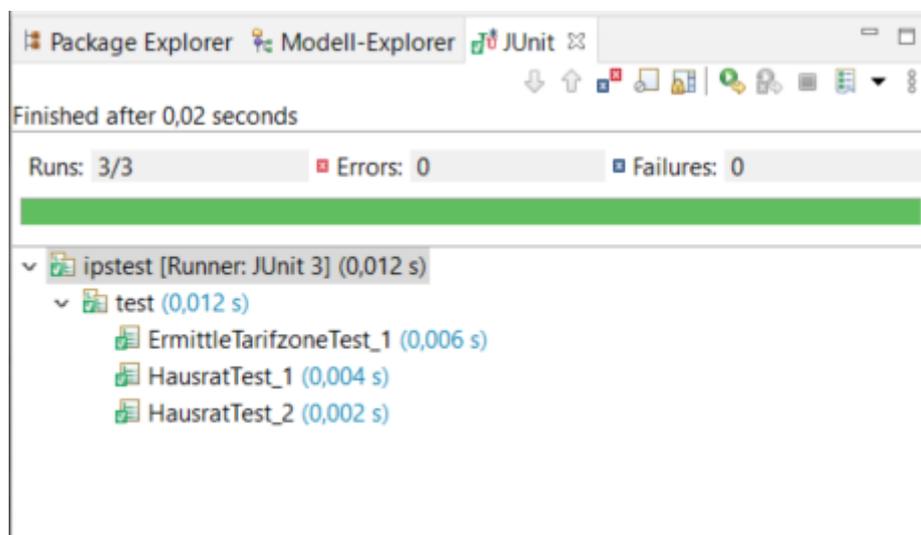


Figure 123. Testadapter mit JUnit-GUI ausführen

Konfiguration des Maven Surefire Plugin für JUnit 5 Tests

In den Testreports des Maven Surefire Plugins werden die aus IPS-Testfällen erzeugten Dynamic Tests nicht mit ihren Anzeigenamen (den IPS-Testfall-Namen) aufgeführt sondern mit dem Namen der mit `@TestFactory` annotierten Methode. Für das obige Codebeispiel würde beispielsweise ein Testreport generiert werden, bei dem drei Tests mit dem Namen "getTests" aufgelistet sind.

Dadurch ist es nicht möglich aus den Testreports herauszulesen, welche IPS-Testfälle fehlgeschlagen sind.

Dieses Problem kann gelöst werden, indem das Plugin in der `pom.xml` wie folgt konfiguriert wird. Dadurch werden die korrekten Anzeigenamen der Dynamic Tests in den Reports verwendet.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M6</version>
  <configuration>
    <statelessTestsetReporter
implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5Xml30Stateles
sReporter">
      <usePhrasedTestCaseMethodName>true</usePhrasedTestCaseMethodName>
    </statelessTestsetReporter>
    <argLine>-Dfile.encoding="UTF-8"</argLine>
  </configuration>
</plugin>
```

Zusammenfassung

In einem typischen Faktor-IPS Projekt werden ca. 60-80% des Fachmodells generiert. Dieser generierte Anteil des Sourcecodes braucht nicht mehr getestet zu werden, da dies einmalig im Rahmen der Faktor-IPS Entwicklung erfolgt. Die Tests für den verbleibenden, individuell entwickelten Anteil können mit JUnit oder mit den Faktor-IPS Testfunktionen definiert werden. Die folgende Grafik zeigt Kriterien auf, wann welches Verfahren besser geeignet ist.

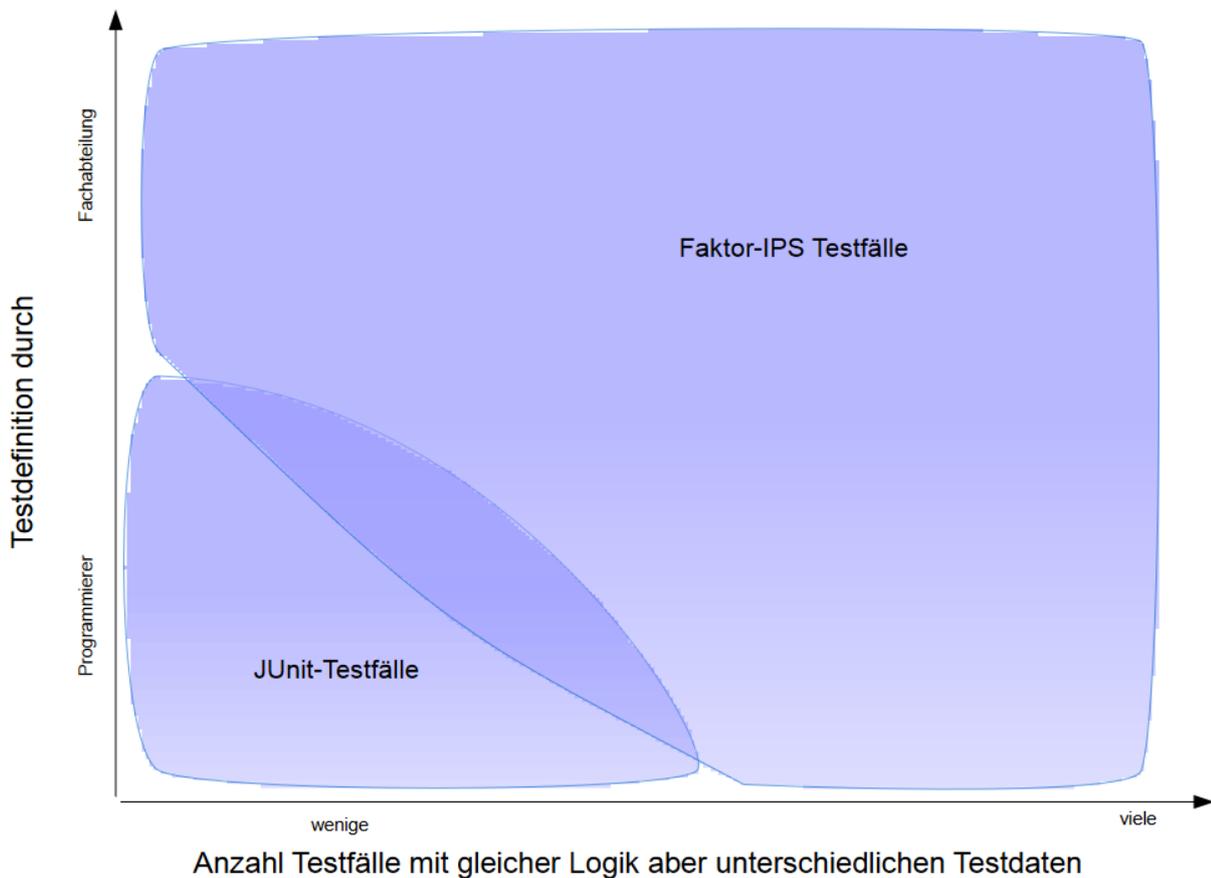


Figure 124. Entscheidungskriterien Testwerkzeug

Die Testunterstützung in Faktor-IPS basiert auf der Unterscheidung zwischen Testfalltypen und Testfällen. Testfalltypen werden durch den Anwendungsentwickler definiert. Dabei wird wie bei der Erstellung des Fachmodells ein modellgetriebener Ansatz verfolgt. Die Struktur der Testdaten wird modelliert und der Sourcecode zum Einlesen der Testdaten aus konkreten Testfällen generiert. Der Entwickler muss lediglich noch den Aufruf der zu testenden Funktionen und den Vergleich der tatsächlichen mit den erwarteten Ergebnissen implementieren.

Auf Basis eines Testfalltypen können Testfälle mit konkreten Testdaten angelegt werden. Hierzu steht ein entsprechender Testfalleditor zur Verfügung. Mit dem Testrunner können Testfälle ausgeführt und etwaige Differenzen angezeigt werden. Dabei kann man einen einzelnen Testfall, mehrere Testfälle oder alle Testfälle eines Projektes ausführen.

Testfalleditor und Testrunner sind in die Produktdefinitionsansicht von Faktor-IPS integriert und können problemlos von Anwendern der Fachabteilung verwendet werden. Damit steht der Fachabteilung eine einheitliche Oberfläche zur Produkt- und Testdefinition zur Verfügung, mit der sie neue Produkte unabhängig von den operativen Systemen testen kann.

Part 3: Testing with Faktor-IPS

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 many 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 Gradle 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	<i>HC-Compact 2021-12</i>	<i>HC-Optimal 2021-12</i>	<i>HC-Optimal 2021-12</i>
ExtraCoverages	<i>Bicycle Theft 2021-12 Overvoltage Damage 2021-12</i>	<i>Bicycle Theft 2021-12 Overvoltage Damage 2021-12</i>	<i>Overvoltage Damage 2021-12</i>
PaymentMode	annual	annual	annual
ZipCode	81673	81673	81673
SumInsured	60.000 EUR	60.000 EUR	100.000EUR
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.

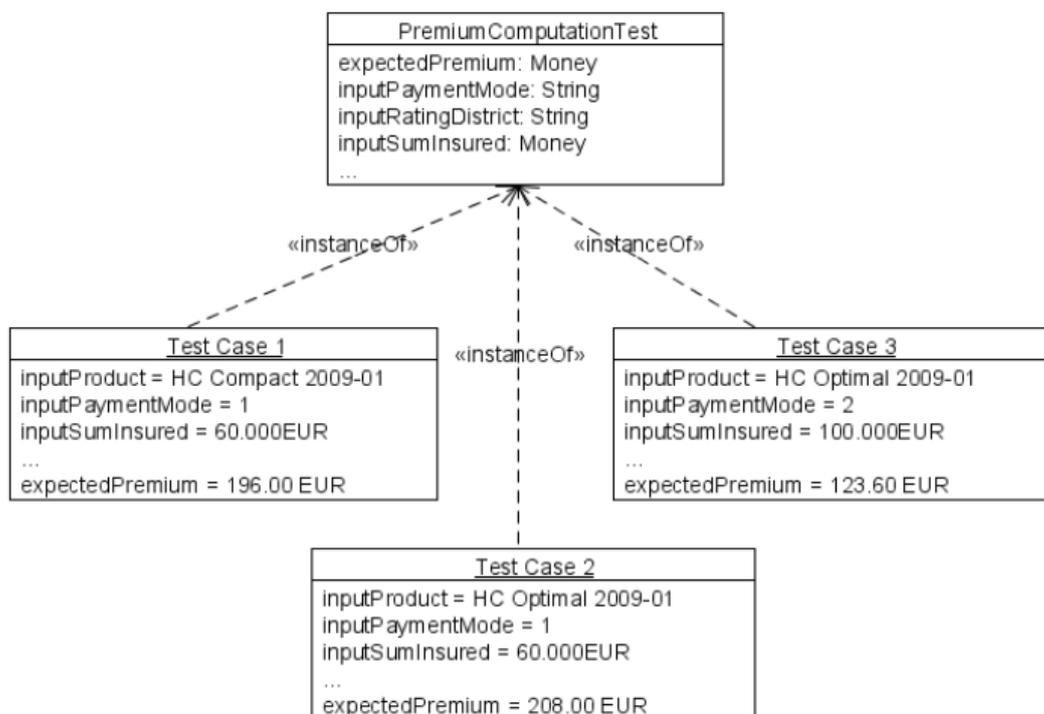


Figure 125. Test cases as instances of the test case type *PremiumComputationTest*

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 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.

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

A Test Case Type for the Premium Computation

First we extend the package structure below the source folder named "model" within the project "HomeModel" by adding an IPS package named "test". 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):

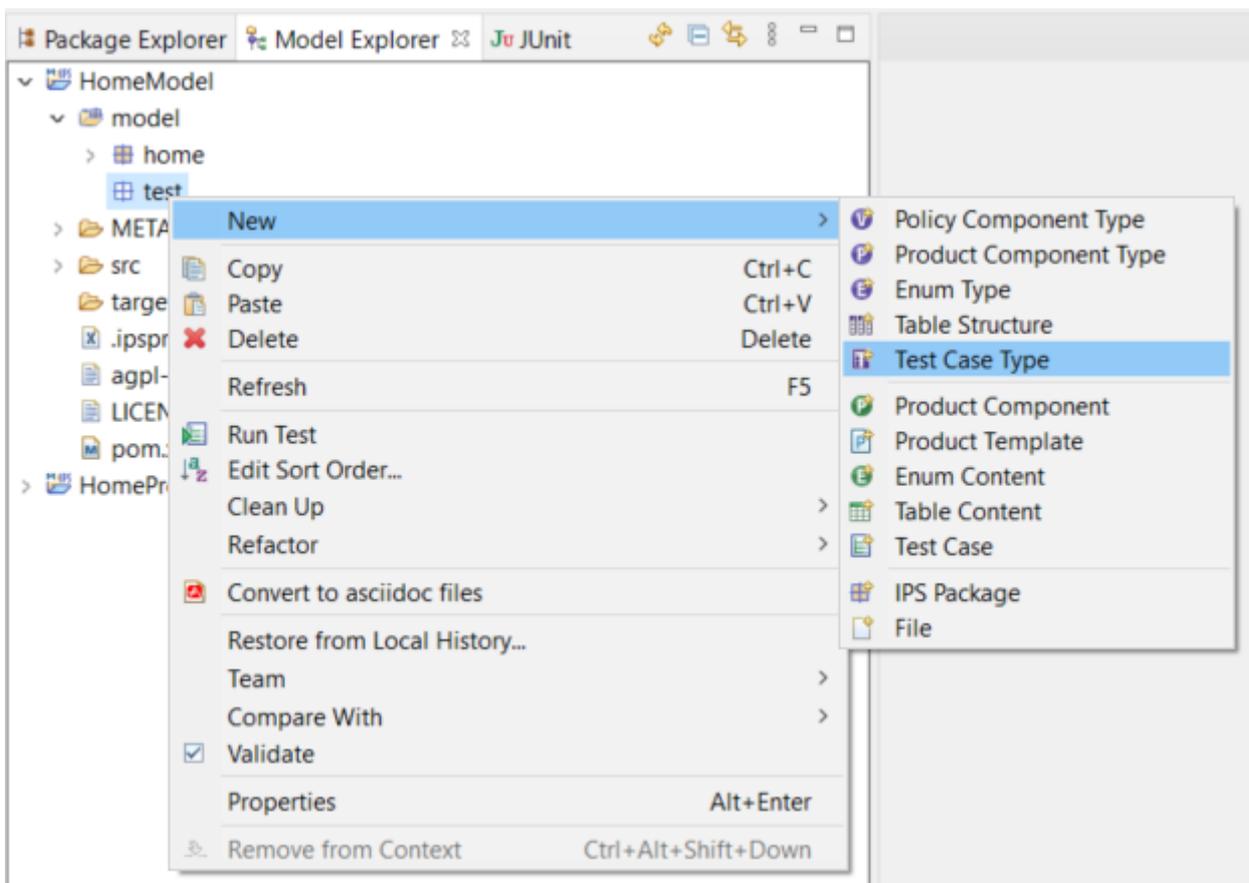


Figure 126. Creating a new test case type

Once we have defined the test case type name *PremiumComputationTest* in the ensuing dialog, the Test Case Type Editor will open. 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
 - A validation rule to be tested

We choose Policy Component Type and click *Next*.

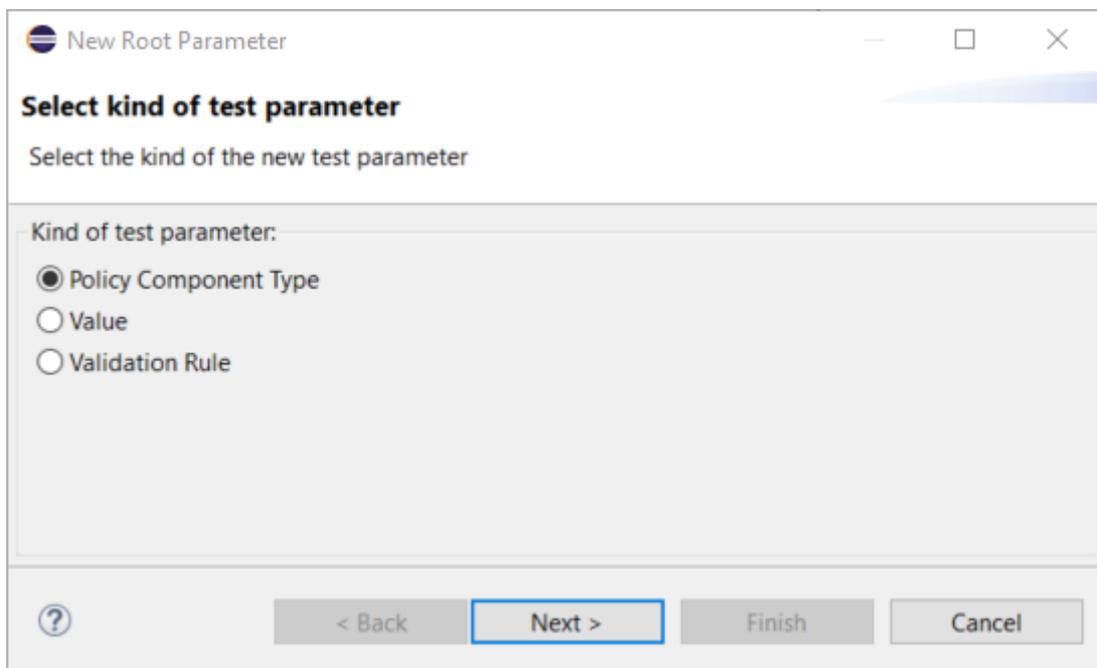


Figure 127. Creating a new root element in the Test Case Editor

On the second page, we select *HomeContract* as data type. By default the parameter has 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:

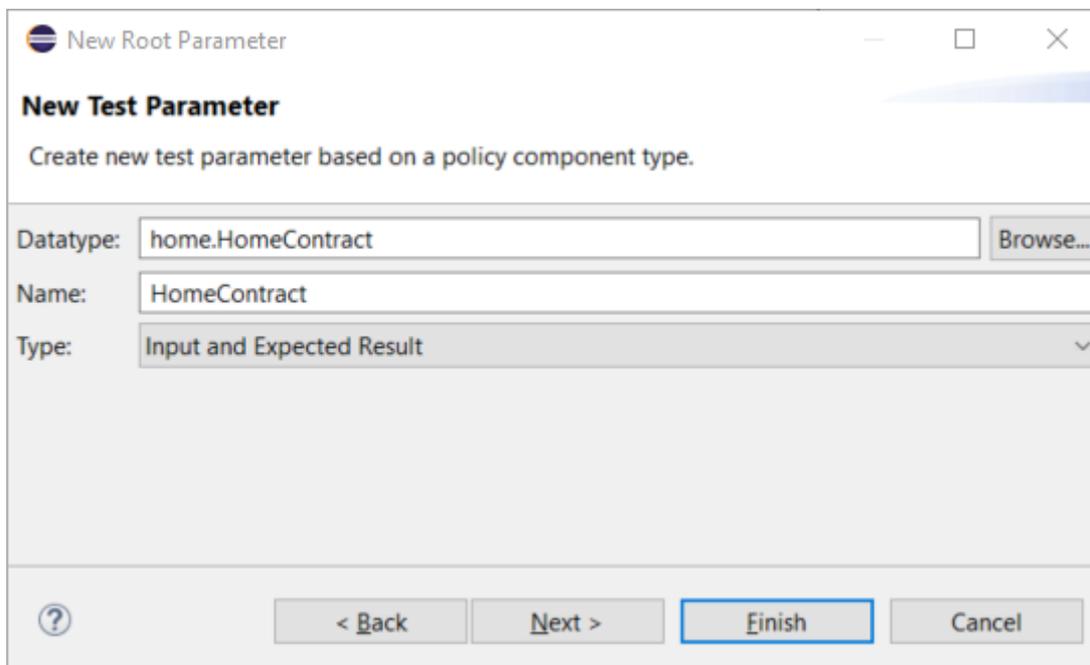


Figure 128. Defining the data type of the test parameter

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:

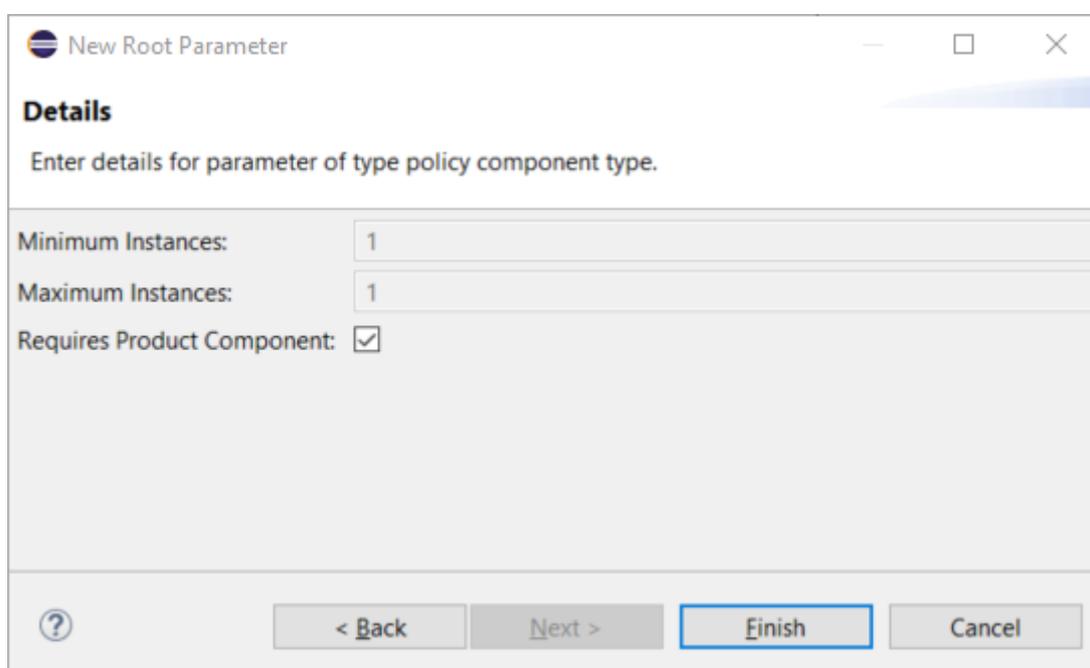


Figure 129. Setting cardinality and product dependency

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.

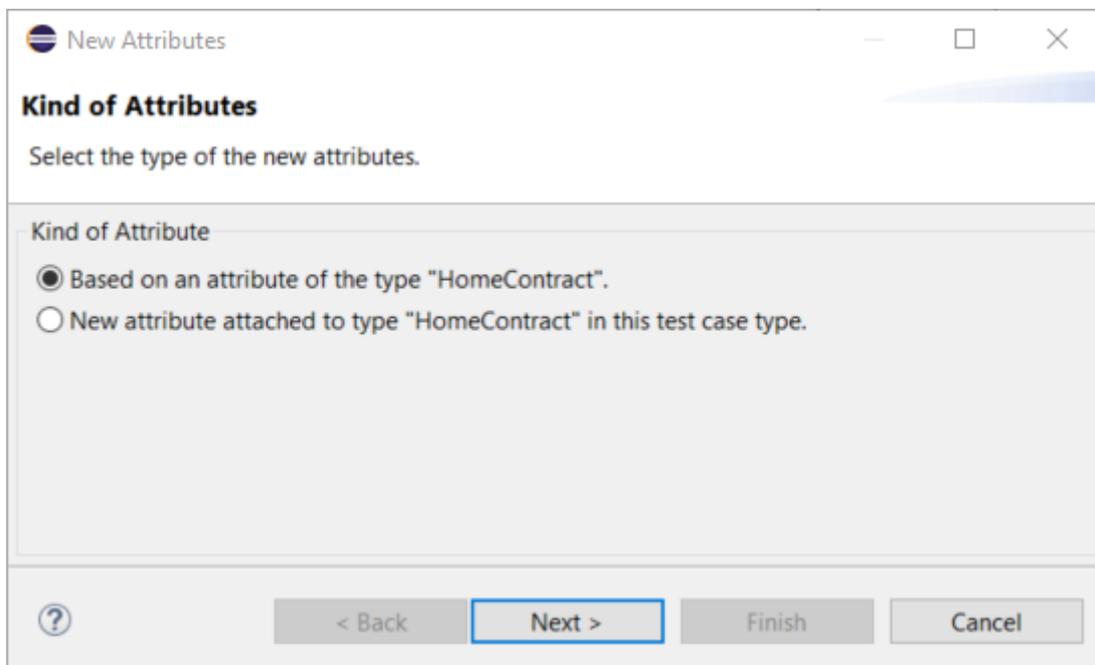


Figure 130. Creating a test attribute

In the ensuing dialog, you can select attributes of the type *HomeContract* and add them to the test case type. In this case, we capture the parameters *netPremiumPm*, *sumInsured*, *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.

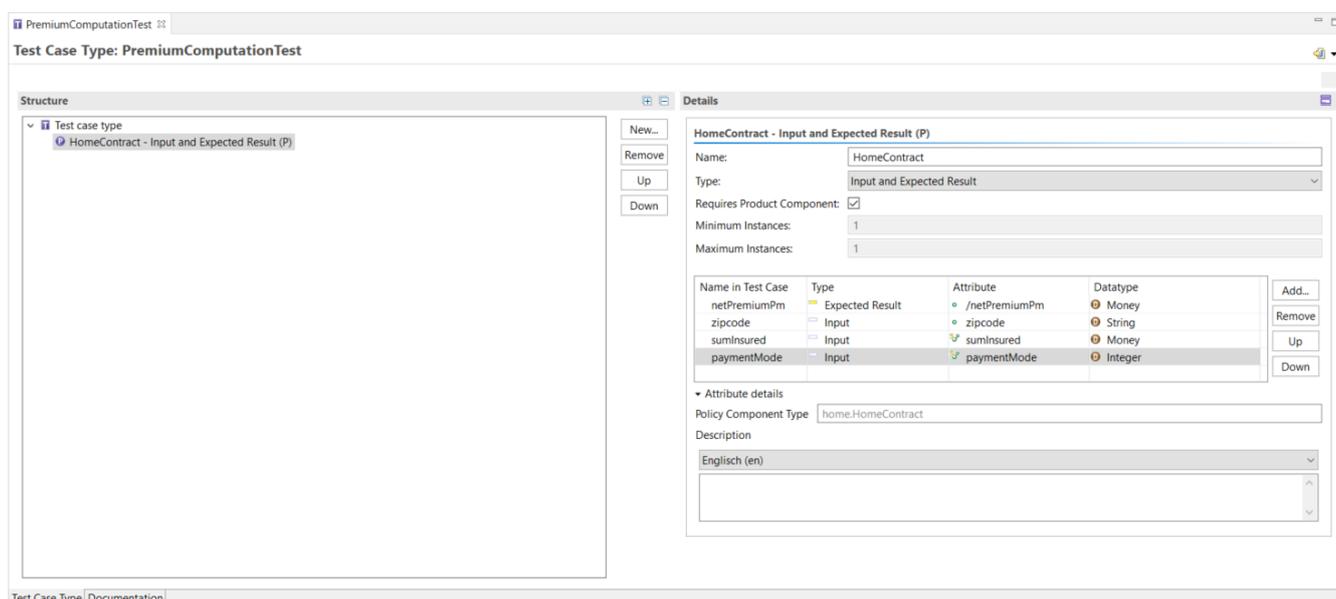


Figure 131. The Test Case Type Editor displaying the Home Contract's attributes being used in the test

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*, 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.

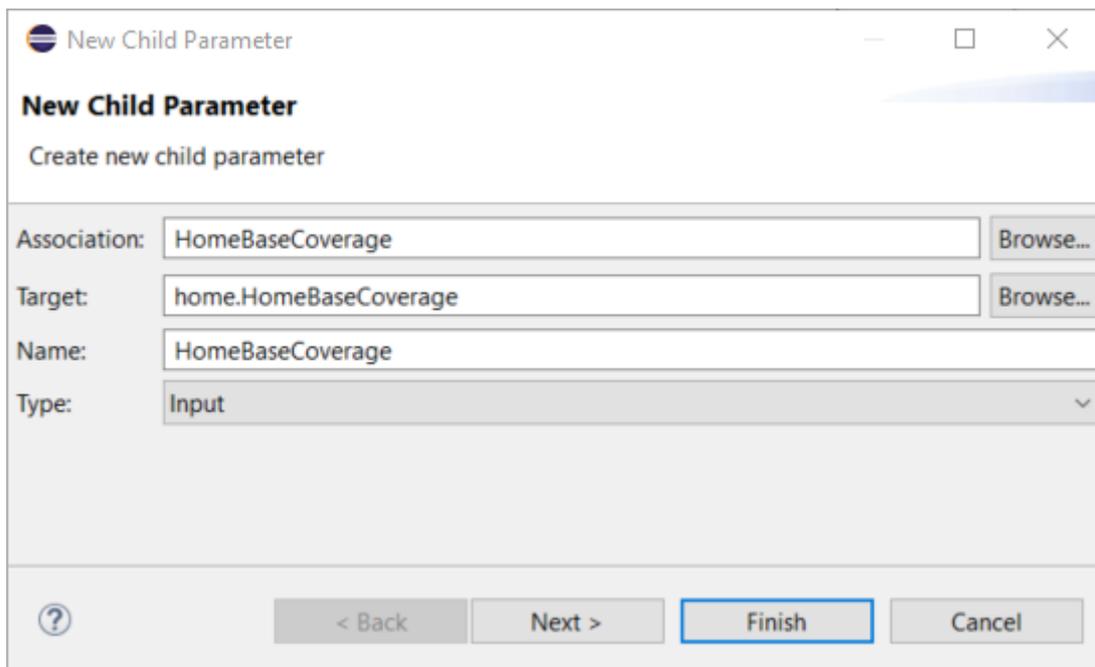


Figure 132. Defining HomeBaseCoverage as a child parameter of HomeContract.

The Test Case Type Editor should now look as follows:

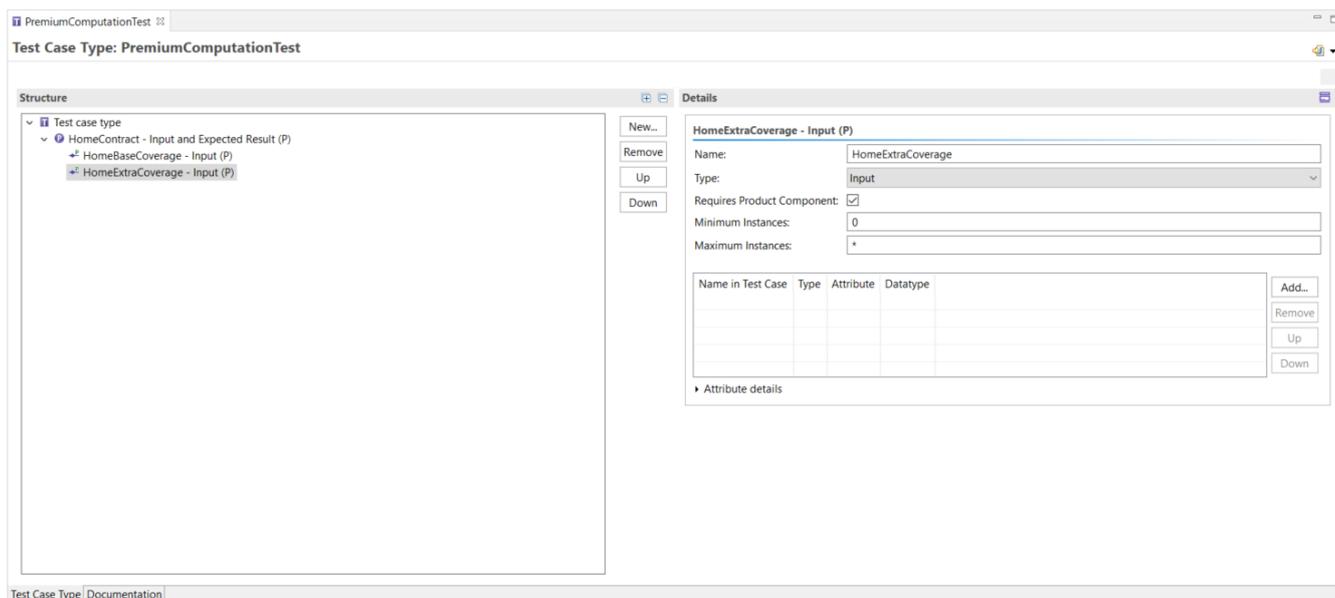


Figure 133. Test Case Type Editor showing the PremiumComputationTest

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.model.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:

```
public class PremiumComputationTest extends IpsTestCase2 {
```

```

//...
private HomeContract inputHomeContract;

private HomeContract expectedHomeContract;

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

public void executeAsserts(IpsTestResult result) {
    // begin-user-code
    // TODO : Inserts the asserts to execute.
    throw new RuntimeException("No asserts implemented in the Java class that
represents the test case type.");
    // end-user-code
}
}

```

- 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 "HomeProducts" we add a new IPS Package named `tests` under the `productdata` directory. This package will be used to hold our test cases. To do this, we return to the Model Explorer, select the `productdata` 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.

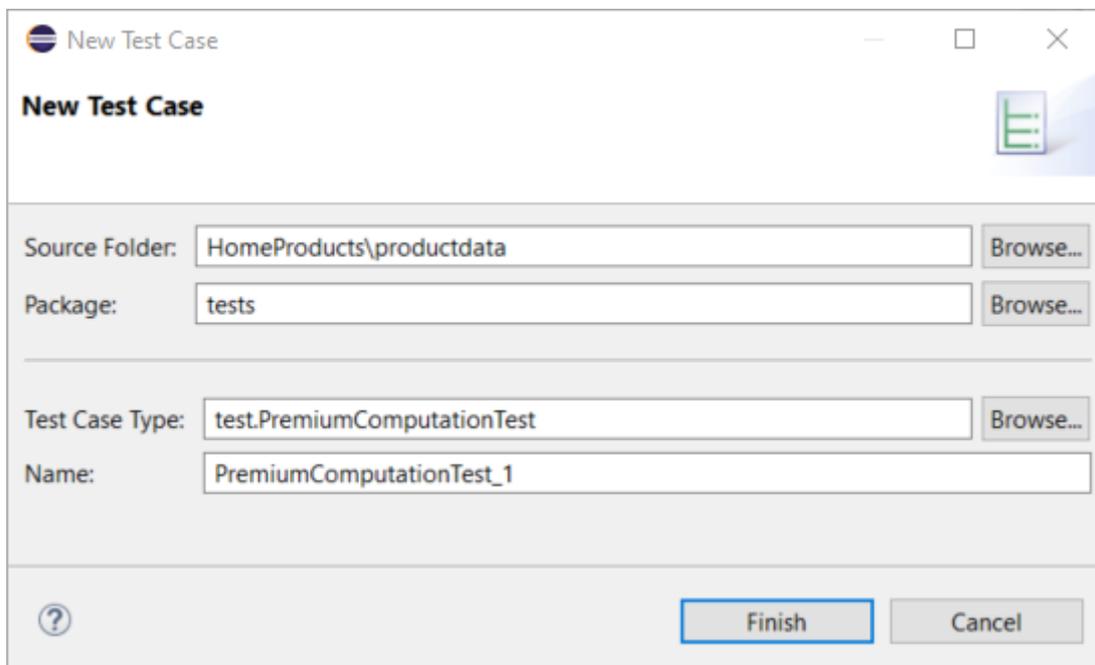


Figure 134. Creating a new test case

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.

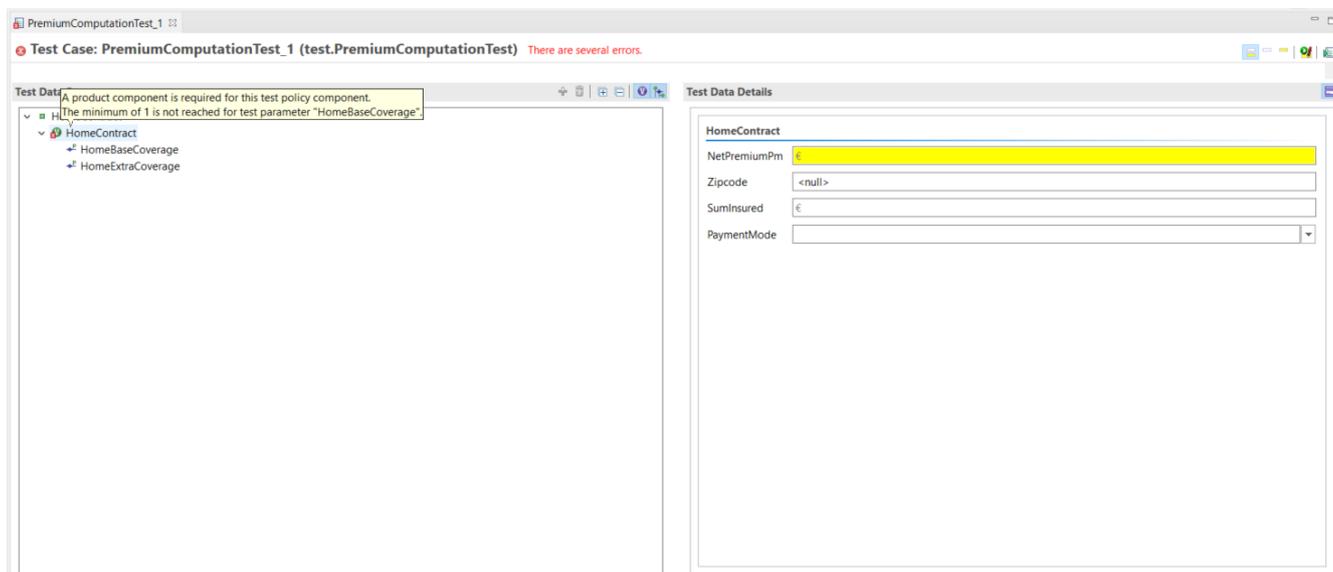


Figure 135. The Test Case Editor hinting at a lacking product component

First we have to assign a product to the *HomeContract*. This is done by selecting the *HomeContract*, opening the context menu and selecting *Assign Product Component*. For our example, we assign the product component HC-Compact 2021-12 to the *HomeContract*.

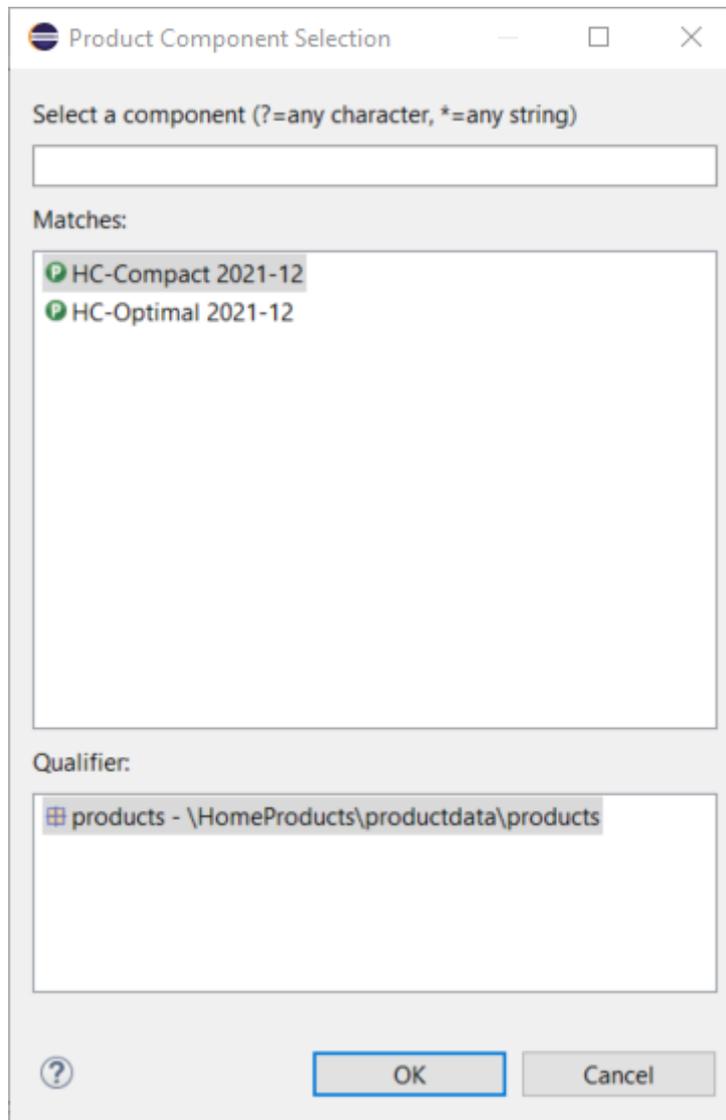


Figure 136. Selecting a product component using "Product Component"

We then use the *Add* option to add further objects, including a *BaseCoverage-Compact 2021-12* for *HomeBaseCoverage* and two instances of *HomeExtraCoverage*, one with *BicycleTheft 2021-12* and one with *OvervoltageDamage 2021-12*. Finally we complete the test case according to the following table:

Table 7. Test data for the premium computation

Parameter	Value
Product	<i>HC-Compact 2021-12</i>
Base Coverage Type	<i>BaseCoverage-Compact 2021-12</i>
Extra Coverage Types	<i>Bicycle Theft 2021-12</i> <i>Overvoltage Damage 2021-12</i>
PaymentMode	1 (annually)
ZipCode	81673 (RatingDistrict I)
SumInsured	60.000 EUR
Net Premium according to Payment Mode	196,00 EUR

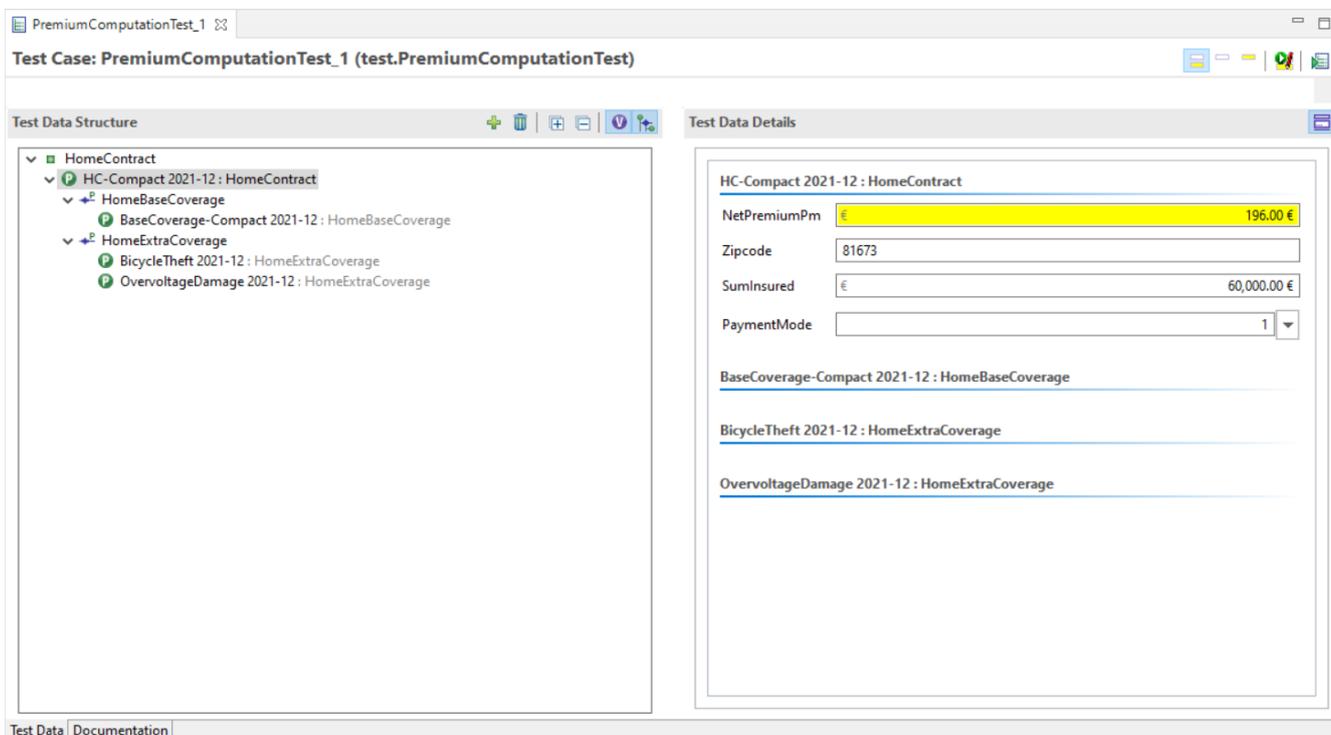


Figure 137. Test Case Editor after all the data has been entered

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").

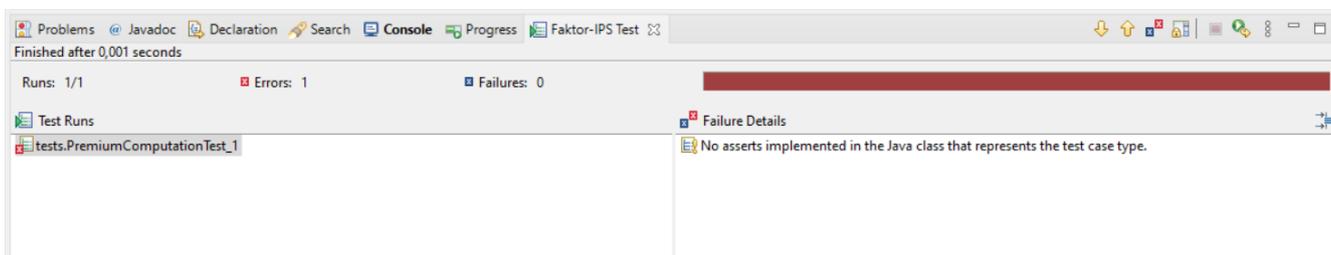


Figure 138. The test case execution with the Testrunner fails

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 show

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

    /**
     * Executes the business logic under test.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeBusinessLogic() {
```

```

        // begin-user-code
        inputHomeContract.computePremium();
        // end-user-code
    }

    /**
     * Executes the asserts that compare actual with expected result.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeAsserts(IpsTestResult result) {
        // begin-user-code
        assertEquals(expectedHomeContract.getNetPremiumPm(),
            inputHomeContract.getNetPremiumPm(),
            result);
        // end-user-code
    }
}

```

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.

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.

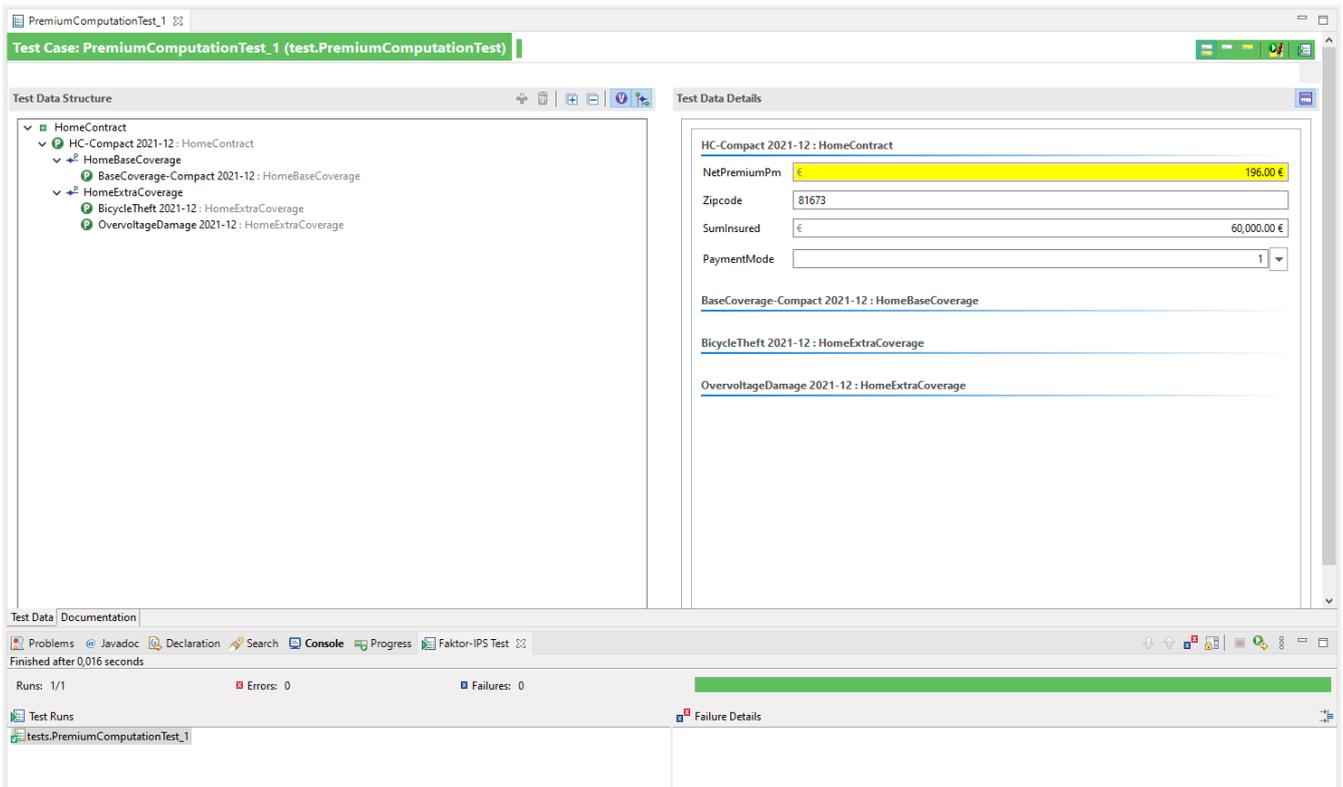


Figure 139. A successful test

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

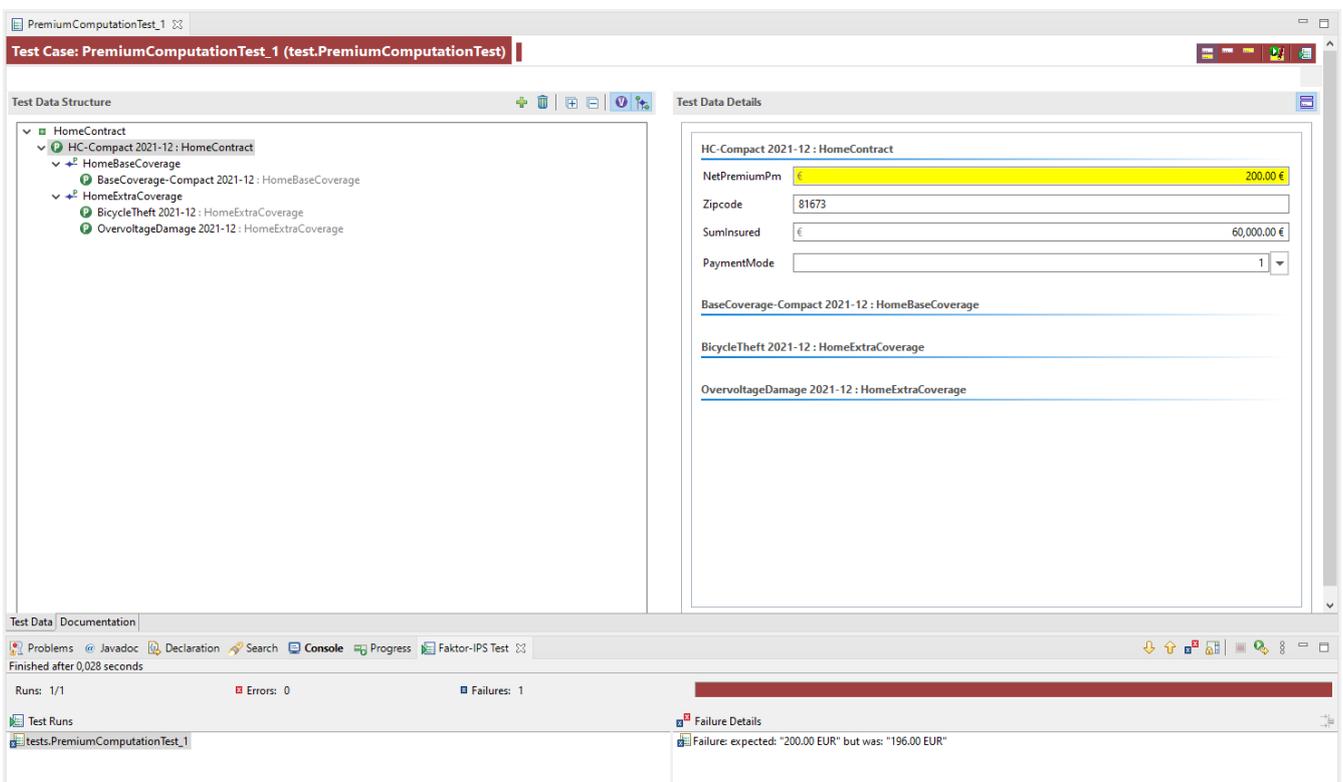


Figure 140. The test case fails

Unfortunately, the error message doesn't yet tell us which attribute it is referring to. In this case, it's not a problem as we have only one expected value, but there can 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 asserts that compare actual with expected result.
 *
 * @restrainedmodifiable
 */
@Override
@IpsGenerated
public void executeAsserts(IpsTestResult result) {
    // begin-user-code
    assertEquals(expectedHomeContract.getNetPremiumPm(),
        inputHomeContract.getNetPremiumPm(),
        result,
        "HomeContract#0",
        HomeContract.PROPERTY_NETPREMIUMPM);
    // end-user-code
}
```

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:

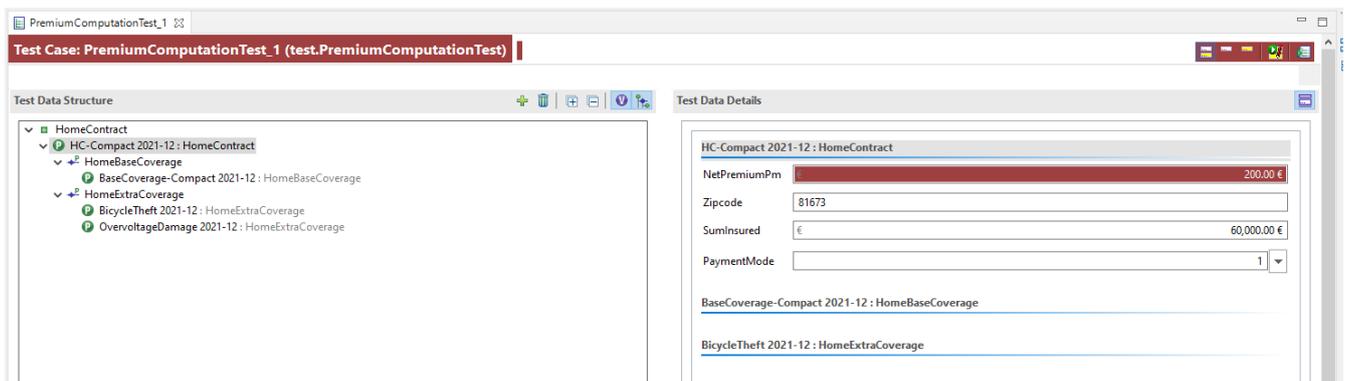


Figure 141. The attribute that caused the error is highlighted in the Test Editor

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 *ratingDistrict* 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* of the type *HomeContract* as input parameters and we enable the *Requires Product Component* checkbox for *HomeContract*. Now we attach a new attribute called *exptectedDistrict* to the type *HomeContract* by clicking the *Add* button.

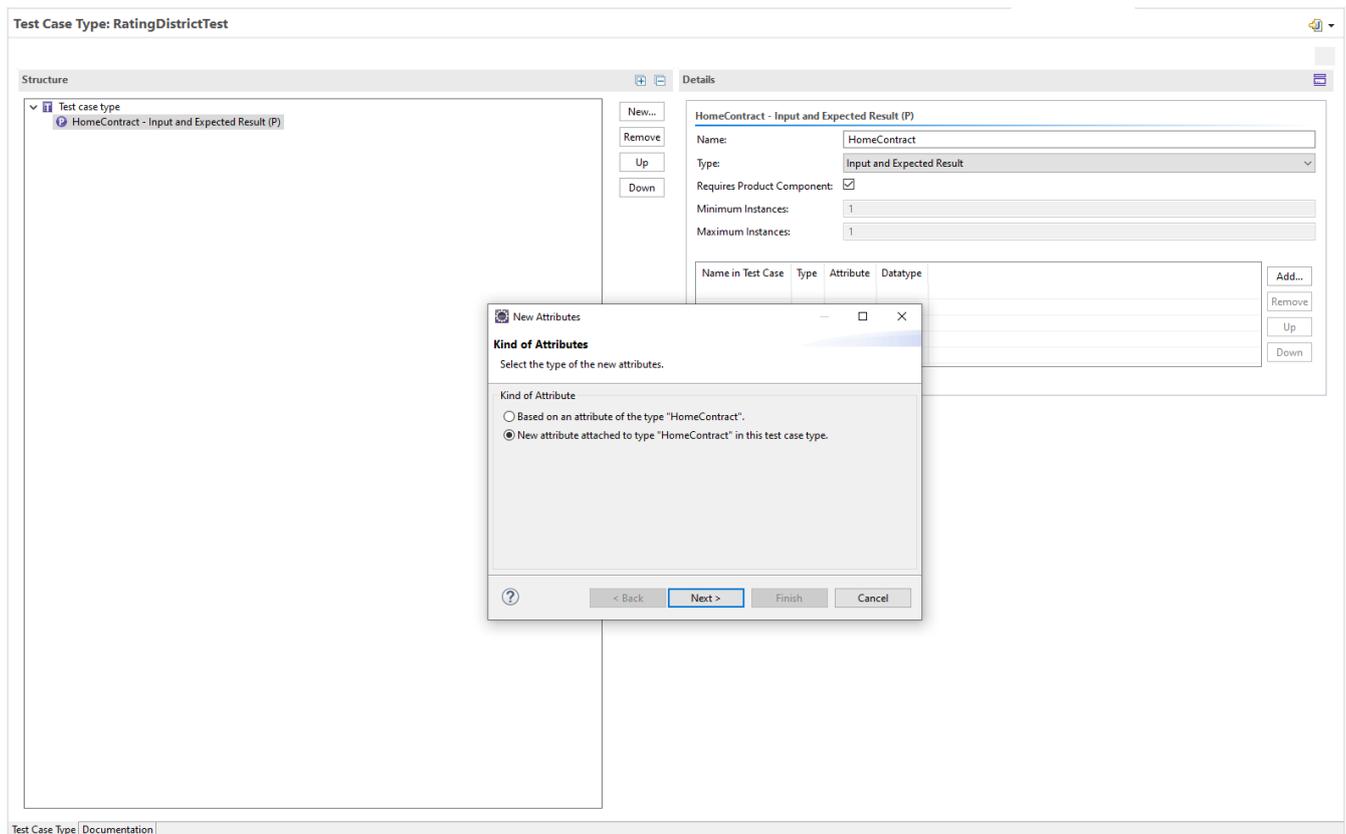


Figure 142. Attach the new attribute "exptectedDistrict" to type *HomeContract*. Step 1

Figure 143. 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 {

    /**
     * @generated
     */
    public static final String TESTATTR_HOME_CONTRACT_EXPECTED_DISTRICT =
"expectedDistrict";

    //...

    /**
     * Executes the business logic under test.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeBusinessLogic() {
        // begin-user-code
```

```

        // nothing to do (the logic to be tested is run by calling the getter)
        // end-user-code
    }

    /**
     * Executes the asserts that compare actual with expected result.
     *
     * @restrainedmodifiable
     */
    @Override
    @IpsGenerated
    public void executeAsserts(IpsTestResult result) {
        // begin-user-code
        String expectedDistrict = (String)getExtensionAttributeValue(
            expectedHomeContract, TESTATTR_HOME_CONTRACT_EXPECTED_DISTRICT);
        String computedDistrict = inputHomeContract.getRatingDistrict();
        assertEquals(expectedDistrict, computedDistrict, result,
            "HomeContract#0", TESTATTR_HOME_CONTRACT_EXPECTED_DISTRICT);
        // end-user-code
    }

    //...
}

```

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 *ZipCode*. In addition, we assign a home product (e.g., *HC-Compact 2021-12*) to the contract. According to the rating district table we expect the district VI for zip code 63066:

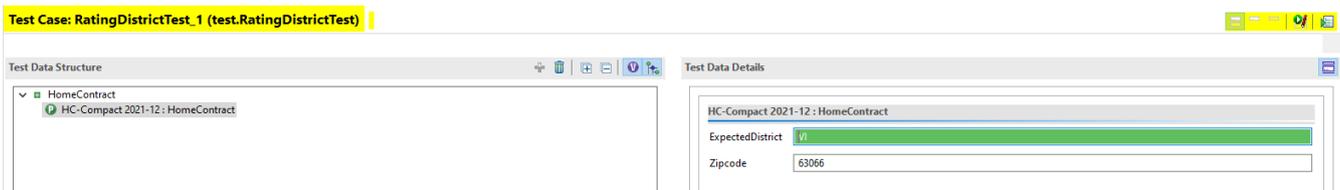


Figure 144. Testcase to verify the tarifzone. Step 3

After running the test, a green bar confirms that our test case is correct.

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 2021-12* instead of *HC-Compact 2021-12*.

To do this, we select the test case we want to copy - *PremiumComputationTest1* - in the Model Explorer and call *Copy Test Case...* from the context menu.

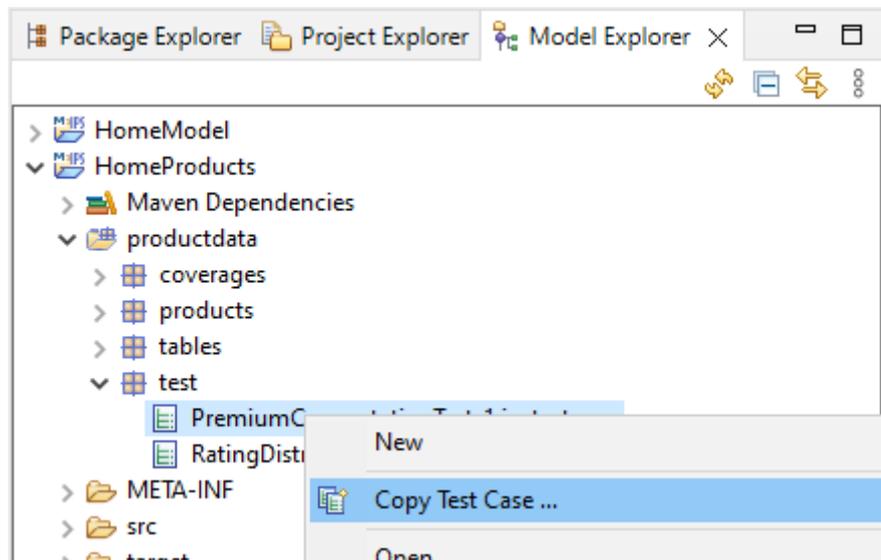


Figure 145. Copy test case context menu

The wizard shown in the next figure guides us through the creation of the test case.

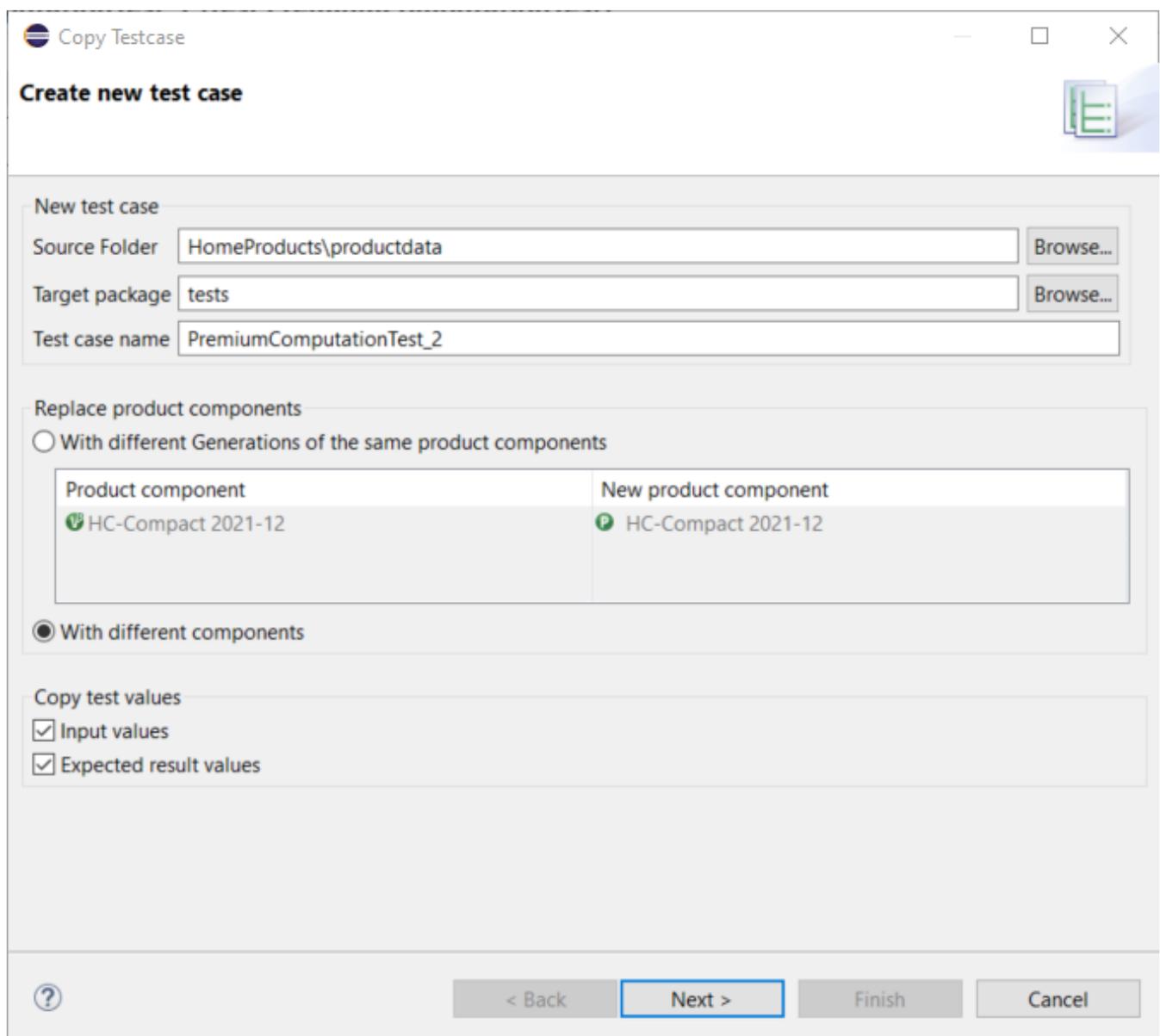


Figure 146. Copy test case wizard, Page 1

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 2021-12 by HC-Optimal 2021-12 and BaseCoverage-Compact 2021-12 by BaseCoverage-Optimal 2021-21, respectively.

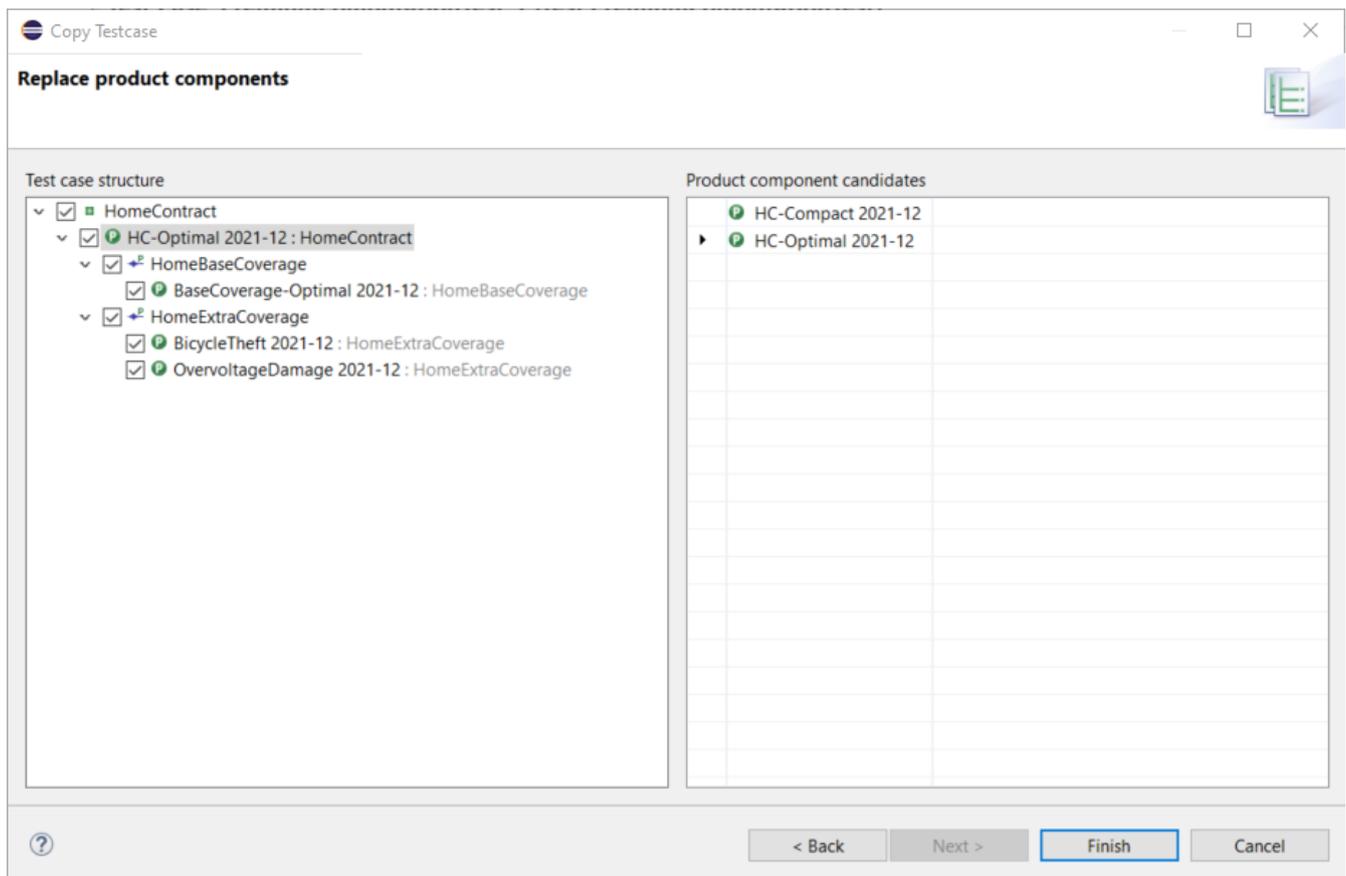


Figure 147. Replacing product components in a test copy

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

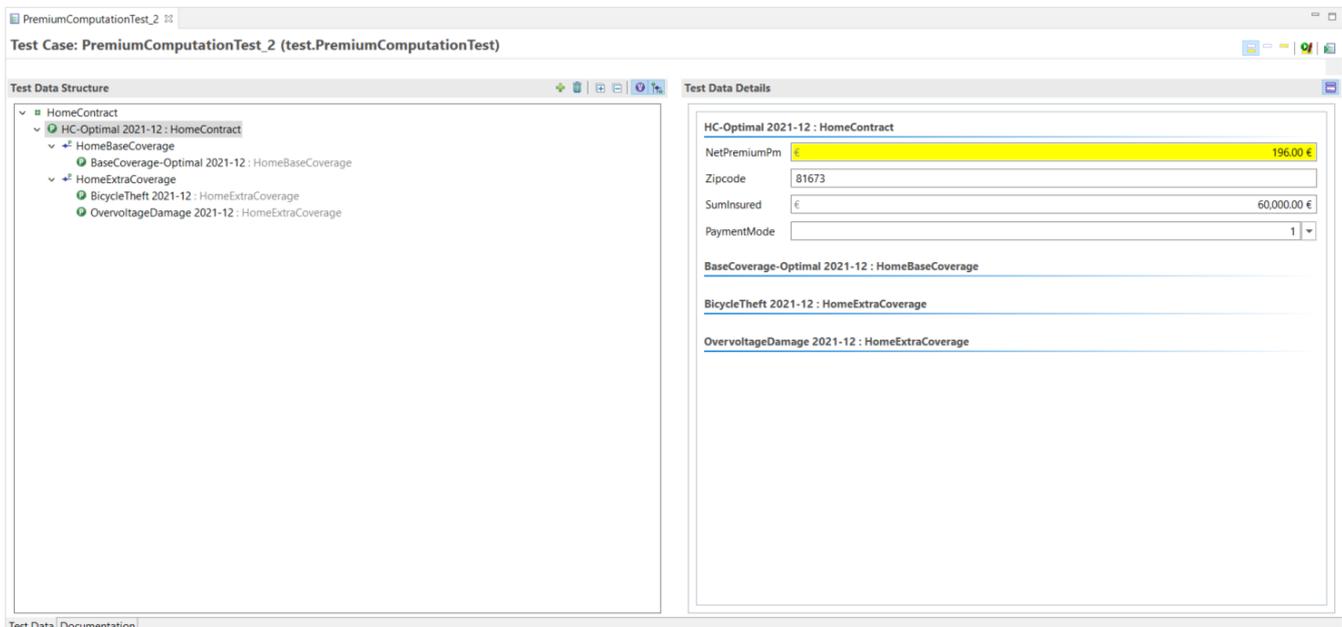


Figure 148. A copied test case

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 *Compute expected values* icon: 🏠

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.

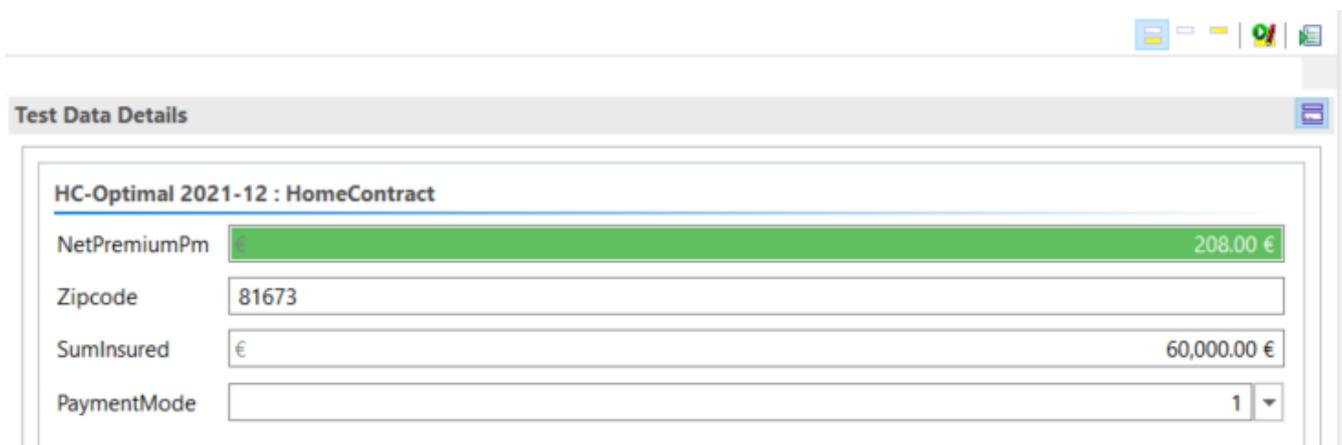


Figure 149. The computed values have been applied to the test case

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 Gradle 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 3/JUnit 4 test suite:

```

import org.faktorips.runtime.ClassloaderRuntimeRepository;
import org.faktorips.runtime.IRuntimeRepository;
import org.faktorips.runtime.test.IpsTestSuiteJUnitAdapter;

import junit.framework.Test;

public class HomeContentsJUnit3Test extends IpsTestSuiteJUnitAdapter {

    public static Test suite() {
        IRuntimeRepository repositoryHomeContents =
ClassloaderRuntimeRepository.create(
            "org/faktorips/tutorial/productdata/internal/faktorips-repository-
toc.xml");

        return createJUnitTest(repositoryHomeContents.getIpsTest(""));
    }
}

```

With the following code we create an adapter, that converts our Faktor-IPS test cases into JUnit 5 Dynamic Tests:

```

import org.faktorips.runtime.ClassloaderRuntimeRepository;
import org.faktorips.runtime.IRuntimeRepository;
import org.faktorips.runtime.test.IpsTestSuiteJUnit5Adapter;

import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.TestFactory;

public class HomeContentsJUnit5Test extends IpsTestSuiteJUnit5Adapter {

    @TestFactory
    public Stream<DynamicTest> getTests() {
        IRuntimeRepository repositoryHomeContents =
ClassloaderRuntimeRepository.create(
            "org/faktorips/tutorial/productdata/internal/faktorips-repository-
toc.xml");

        return createTests(repositoryHomeContents.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. For JUnit 3/JUnit 4 the `createJUnitTest(...)` method of class `IpsTestSuiteJUnitAdapter` takes the test suite and makes it a JUnit test suite. For JUnit 5 the method `createTest(...)` of class `IpsTestSuiteJUnit5Adapter` returns a `Stream` of `DynamicTests`. If we execute the test class with the respective JUnit-Testrunner, we can see how our Faktor-IPS test cases are executed and interpreted by JUnit.

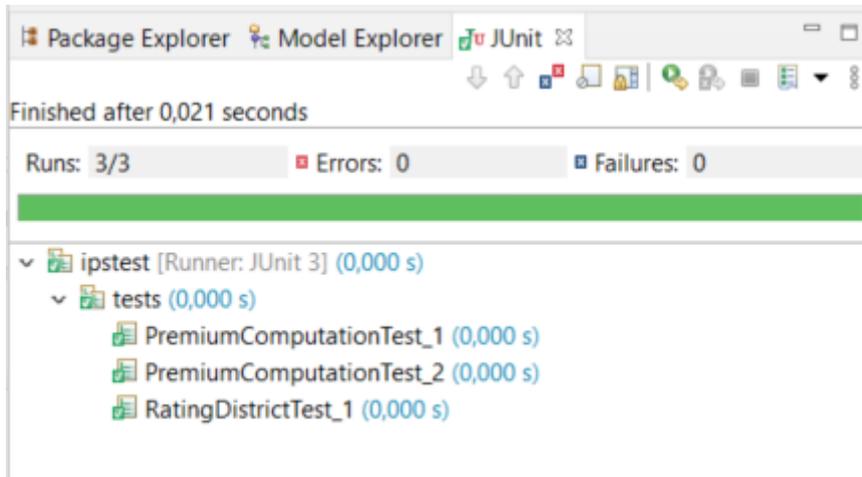


Figure 150. Running a test adapter in the JUnit GUI

Configuration of the Maven Surefire Plugin for JUnit 5 tests

In the test reports of the Maven Surefire Plugin, the dynamic tests generated from the IPS test cases are not listed with their display names (the IPS test case names) but with the name of the method that is annotated with `@TestFactory`. For example, the code sample above would generate a test report listing three tests named "getTests". That makes it impossible to find out from the test report which IPS test cases have failed.

This problem can be solved by configuring the plugin in the `pom.xml` as follows. The correct display names of the dynamic tests will then be used in the test reports.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M6</version>
  <configuration>
    <statelessTestsetReporter
implementation="org.apache.maven.plugin.surefire.extensions.junit5.JUnit5Xml30Stateles
sReporter">
      <usePhrasedTestCaseMethodName>true</usePhrasedTestCaseMethodName>
    </statelessTestsetReporter>
    <argLine>-Dfile.encoding="UTF-8"</argLine>
  </configuration>
</plugin>
```

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.

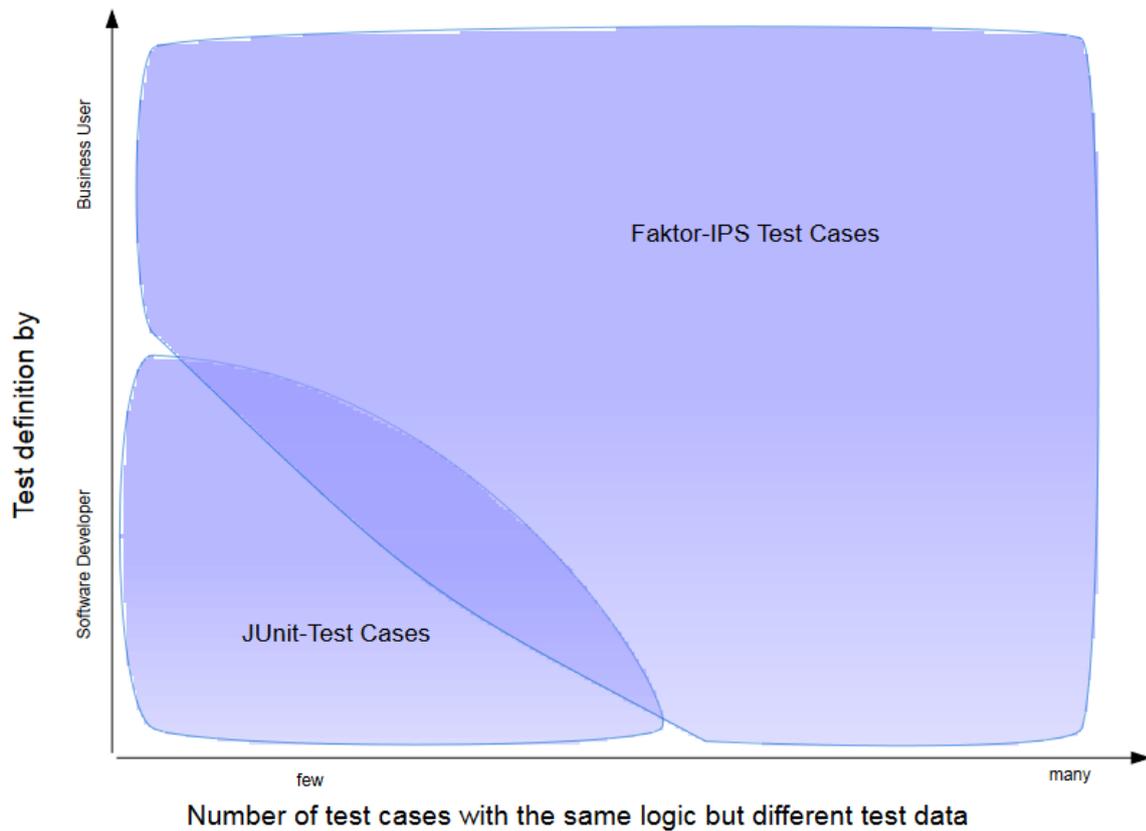


Figure 151. Decision criteria for test tool

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 it is the case with the business object model, a model-driven approach is 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 test runner 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 test runner 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.

FAQ

Dieses FAQ wurde von einem Nicht-Entwickler erstellt um bei möglichen auftretenden Fragen zu unterstützen. Es beinhaltet sowohl Erklärungen von technischen Begriffen als auch Lösungsansätze bei Problemstellungen.

Generell gilt im Tutorial:

Modellobjekte haben lila Icons und werden immer in "Hausratmodell" angelegt.

Produktobjekte haben grüne Icons und werden immer in "Hausratprodukte" angelegt.

Java-Testordner (z.B. für JUnit-Tests) sind grundsätzlich unter "src/test/java" anzulegen.

Wenn nach dem Copy-Paste einer Codestelle Fehlermeldungen angezeigt werden kann man mit der Tastenkombination "Strg-Shift-O" fehlende Dependencies automatisch importieren.

Bei den automatisch generierten Codestellen steht "@since 1.0" dabei. Im Tutorial ist das nicht zu sehen - warum?

Die Musterlösung des Tutorials wurde ursprünglich ohne Archetype erstellt. Im Tutorial wird nun aber zum Erstellen der Projekte der Archetype genutzt. Dieser setzt die Version des Projekts auf 1.0 und definiert einen Version Provider in der ".ipsproject"-Datei. Dadurch wird an generierten Codestellen die Version mitgeneriert.

Dies kann ausgeschaltet werden indem in der Datei ".ipsproject" der folgende Eintrag gelöscht wird:

```
<Version versionProvider="org.faktorips.maven.mavenVersionProvider"/>
```

Teil 1: Modellierung und Produktkonfiguration

Hello Faktor-IPS

- *Faktor-IPS mit anderer Locale (Sprache) starten*

Im Eclipse Installationsordner die Datei "eclipse.ini" suchen. Dort für eine Änderung auf Englisch

```
-nl  
en
```

eintragen. Für Deutsch "de" nutzen.

- *Was ist ein Maven Archetype?*

Ein Archetype ist ein Maven-Projekt-Template, welches die wichtigsten Projekteinstellungen bereits enthält. Daraus lässt sich nach Eingabe einiger Parameter ein funktionierendes Projekt mit allen benötigten Dateien generieren.

- *Probleme beim Archetype mit dem Befehl "mvn"*

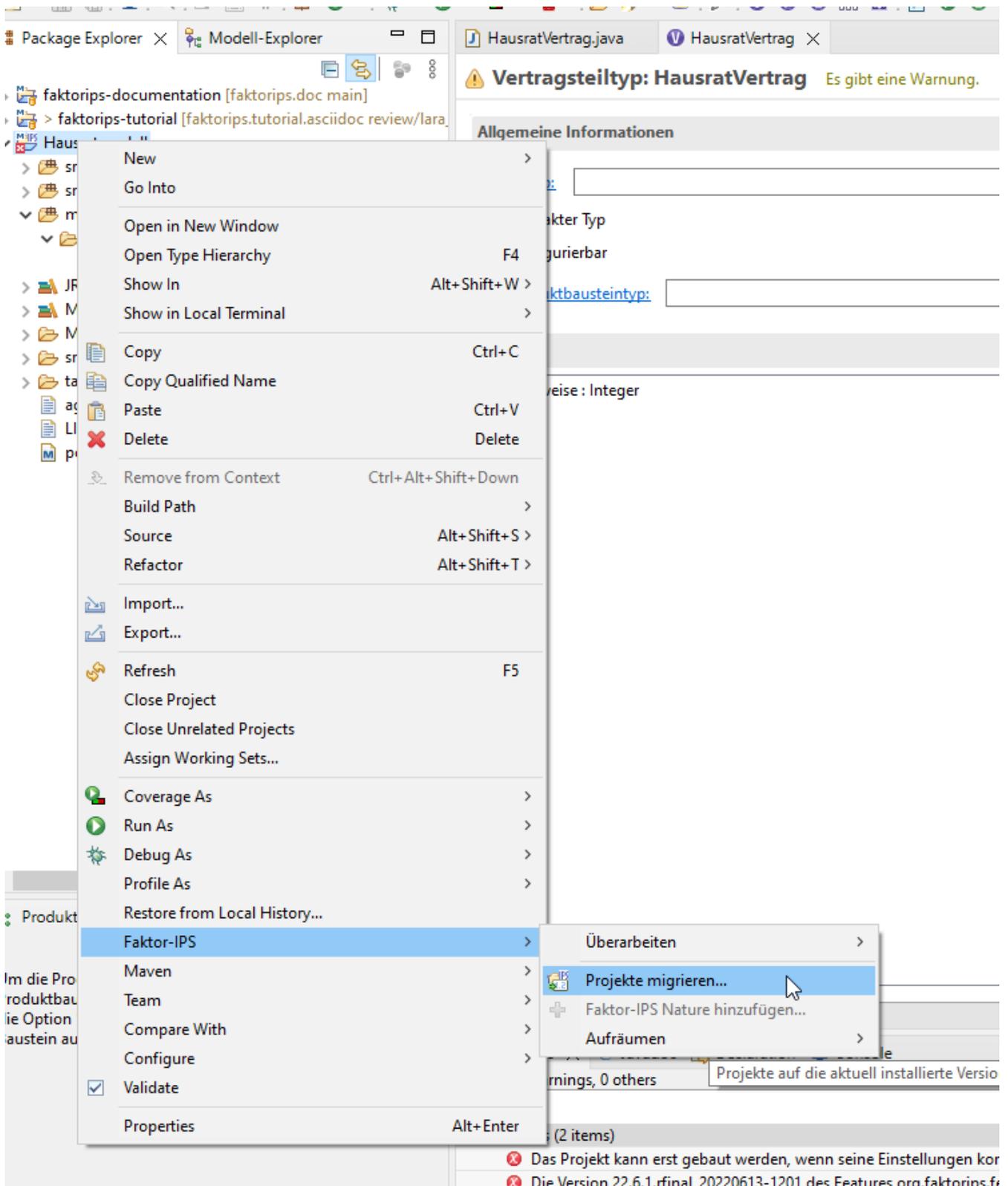
Dies deutet darauf hin, dass Maven nicht installiert ist und eine PATH-Variable, die den mvn-Befehl übersetzt, fehlt. Dies ist normalerweise in Eclipse integriert, funktioniert aber aufgrund eines Bugs nicht ordnungsgemäß. Um den Fehler zu lösen muss Maven manuell installiert

werden:

1. <https://maven.apache.org/download.cgi> (korrekte Binary downloaden)
 2. In Windows den entpackten Ordner (z.B. apache-maven-3.8.6) nach C:\Program Files\ verschieben
 3. Settings - System - About - Advanced System Settings - Environmental Variables
 4. Systemvariablen (zweite Box) - Variable "Path" suchen und "Bearbeiten..." - Neu - "C:\Program Files\apache-maven-3.8.6\bin" einfügen und speichern
 5. Die PATH Variable für den Befehl "mvn" ist nun gesetzt und wird nun nach einem Neustart der Kommandozeile dort erkannt. Es kann nun mit dem Tutorial fortgefahren werden.
- *Fehlermeldung: No plugin found for prefix 'archetype' in the current project and in the plugin groups (org.jenkins-ci.tools, org.glassfish.maven.plugin, org.apache.maven.plugins, org.codehaus.mojo) available from the repositories local (C:\Users\USER\.m2\repository), nexus <https://nexus.faktorzehn.de/content/groups/private> → (Help 1)*
Die Java Version ist zu alt (idR 1.7). Diese kann mit folgendem Befehl über die Kommandozeile gesetzt werden: "export JAVA_HOME=<Pfad zur aktuellen Java Version (bspw Java 11)>".
Beispiel: "export JAVA_HOME=C:/Users/USER/eclipse/jdk-11.0.15+10"

Arbeiten mit Modell und Sourcecode

- *Wo finde ich die Einstellung "Published Interfaces generieren"?*
Zu finden unter Project → Properties (und nicht unter Window → Preferences).
- *Fehlermeldung: Die Version "22.6.1.rfinal_20220613-1201 des Features org.faktorips.feature ist zu der von diesem Projekt verlangten Version 21.6.0 nicht kompatibel. Versuchen Sie das Projekt zu migrieren."*
Das Projekt wurde mit einem zu alten Archetype erstellt. Man kann Projekte in Faktor-IPS einfach migrieren. Bevorzugtes Benennungsschema (in dieser Version!): "Unified".



Zugriff auf Produktinformationen zur Laufzeit

- *Was ist ein Classloader?*

Der Classloader lädt die einzelnen Programmdateien (Klassen und Ressourcen), aus denen ein Java-Programm besteht. Faktor-IPS nutzt ihn auch, um die neben den Klassen liegenden Produktdaten zu finden.

- *Ergebnis auf eine Klasse casten - was bedeutet das?*

Klassen können voneinander abgeleitet sein. So kann HausratProdukt eine Spezialisierung von Produkt sein (fachliche Spezialisierung), welches wiederum eine Spezialisierung von

ProductComponent ist (technische Spezialisierung). Da das Repository nur weiß, dass alle Produkte Instanzen dieser Basisklasse sind müssen wir dem Programm mitteilen, dass wir hier davon ausgehen zu wissen, welche Spezialisierung in diesem Fall genutzt wird und "casten" darauf.

Statt

```
ProductComponent p = repository.getProductComponent("Hausrat Kompakt");
```

schreiben wir

```
HausratProdukt hp = (HausratProdukt) repository.getProductComponent("Hausrat Kompakt");
```

und können dadurch auch auf die Methoden zugreifen, die HausratProdukt definiert.

Teil 2: Verwendung von Tabellen und Formeln

Verwendung von Tabellen

- Erzeugen Sie nun für die Produkte HR-Optimal und HR-Kompakt (bzw. genauer für deren Grunddeckungstypen) jeweils einen Tabelleninhalt mit dem Namen „Tariftabelle Optimal 2019-07“ und „Tariftabelle Kompakt 2019-07“.

Im Modell Explorer auf HR-Kompakt Rechtsklick → Neu... → Tabelleninhalt → TariftabelleHausrat → Name befüllen → Finish

Verwendung von Formeln

- Die Beziehung zwischen HausratZusatzdeckungs und HausratVertrag sollte in HausratVertrag mit dem Typ "Elternteil zu Kind" angelegt werden. Ansonsten kann die HausratZusatzdeckung mehrere HausratVerträge haben (und daher funktioniert die Logik nicht mehr).

Teil 3: Testen mit Faktor-IPS

Testfall anlegen

Erster Testfall schlägt fehl:

```
Can't create instance for toc entry  
TocEntry(TestCase:produktdaten/test/HausratTest_1.ipstestcase)  
java.lang.reflect.InvocationTargetException  
Keine Pruefungen vorhanden. Diese muessen in der Java-Klasse, die den Testfalltyp  
repraesentiert, implementiert werden.
```

Dies bedeutet, dass keine Asserts in BeitragsberechnungHausratTest.java implementiert wurden:

```
public void executeAsserts(IpsTestResult result) {  
    // begin-user-code  
    // TODO : Hier muessen die durchzufuehrenden Pruefungen implementiert werden.
```

```
    throw new RuntimeException("Keine Pruefungen vorhanden. Diese muessen in der Java-  
Klasse, die den Testfalltyp repraesentiert, implementiert werden.");  
    // end-user-code  
}
```

Hier nun die Zeile auskommentieren. Es kommt zu einem weiteren Fehler "Not implemented yet!". Es wird in der Methode `executeBusinessLogic()` die Funktion `inputHausratVertrag.berechneBeitrag()` aufgerufen. Wir springen zu dieser indem wir `berechneBeitrag()` anklicken und F3 drücken. Hier sehen wir, dass die Funktion noch nicht implementiert wurde.

```
public void berechneBeitrag() {  
    // TODO implement model method.  
    throw new RuntimeException("Not implemented yet!");  
}
```

FAQ

This FAQ was created by a non-developer to provide support for questions that may arise. It contains explanations of technical terms as well as solutions to problems.

In general, the following applies throughout the tutorial:

The model objects have purple icons and should always be created in "HomeModel".

The product objects have green icons and should always be created in "HomeProduct".

Java test folders (e.g. JUnit tests) should always be created in "src/test/java".

If error messages are displayed after copy-pasting a code, missing dependencies can be imported automatically with the key combination "Ctrl-Shift-O".

@since 1.0 is added to the automatically generated codes. You can't see that in the tutorial - why?

The sample solution of this tutorial was originally created without an archetype. In the tutorial the archetype is used to create the projects. This sets the version of the project to 1.0 and defines a version provider in the ".ipproject" file. As a result, the version is also generated in the generated code.

This can be switched off by deleting the following entry in the ".ipproject" file:

```
<Version versionProvider="org.faktorips.maven.mavenVersionProvider"/>
```

Part 1: Modeling and Product Configuration

Hello Faktor-IPS

- *Start Faktor-IPS with a different locale (language)*

Search for the file "eclipse.ini" in the eclipse installation folder. To change the language to english insert

```
-nl  
en
```

For german "de".

- *What is a Maven Archetype?*

An archetype is a Maven project template that already contains the most important project settings. After entering a few parameters, a working project with all the necessary files will be generated.

- *Archetype issues with "mvn" command*

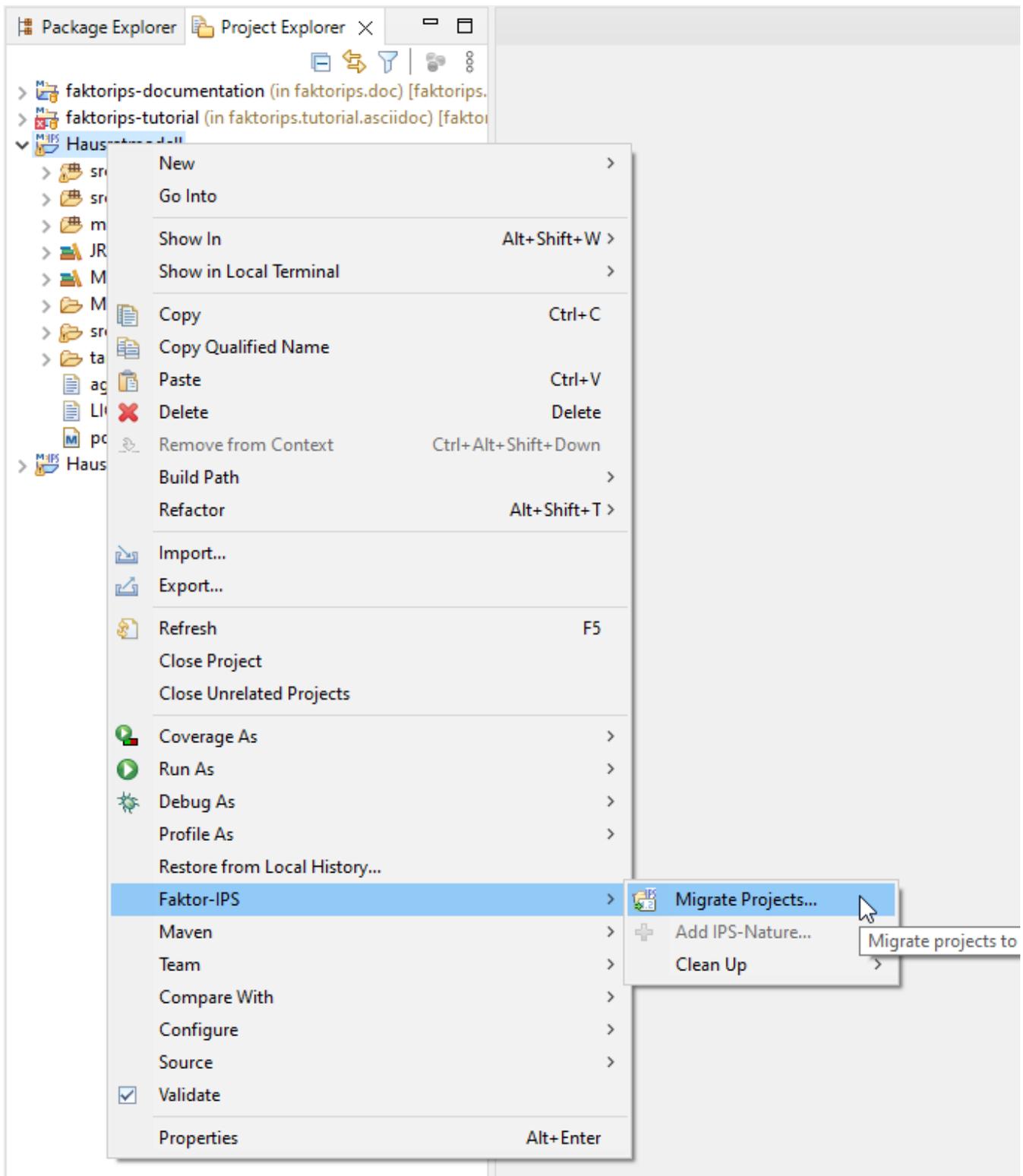
This indicates that Maven is not installed and the PATH variable that translates the mvn-command is missing. This is usually built into Eclipse but doesn't work properly at the moment due to a bug. To solve the error, Maven has to be installed manually:

1. <https://maven.apache.org/download.cgi> (download correct binary)
2. In Windows move the unpacked folder (apache-maven-3.8.6) to C:\Program Files\

3. Settings - System - About - Advanced System Settings - Environmental Variables
 4. System Variables (second box) - Find variable "Path" and "Edit..." - New - Paste "C:\Program Files\apache-maven-3.8.6\bin" and save
 5. The PATH variable for the "mvn" command is now set and will be recognized after restarting the command line. It is possible to continue with the tutorial now.
- *Error message: No plugin found for prefix 'archetype' in the current project and in the plugin groups (org.jenkins-ci.tools, org.glassfish.maven.plugin, org.apache.maven.plugins, org.codehaus.mojo) available from the repositories local (C:\Users\USER\.m2\repository), nexus https://nexus.faktorzehn.de/content/groups/private → (Help 1)*
The Java version is too old (usually 1.7). This can be set with the following command via the command line: "export JAVA_HOME=<path to a newer java version (eg Java 11)>". Example: "export JAVA_HOME=C:/Users/USER/eclipse/jdk-11.0.15+10"

Working with the Model and Source-code

- *Error: The version 22.6.1.rfinal_20220613-1201 of the feature org.faktorips.feature is not compatible with the version 21.6.0 required by this project. Try to migrate the project.*
Migrate Faktor-IPS. Preferred Naming Scheme (in this version!): "Unified"



Runtime access to Product Information

- *What is a classloader?*
The classloader loads the individual program files (classes and resources) that are needed to run the Java code. Faktor-IPS also uses it to find the product data next to the classes.
- *Cast result to a class - what does that mean?*
Classes can be derived from each other. HomeProduct can be a specialization of Product (functional specialization), which in turn is a specialization of ProductComponent (technical specialization). Since the repository only knows that all products are instances of this base class, we have to tell the program that we assume we know which specialization is used in this case

and "cast" to it.

Instead of

```
ProductComponent p = repository.getProductComponent("Home Contents Compact");
```

we write

```
HomeProduct hp = (HomeProduct) repository.getProductComponent("Home Contents Compact");
```

and can thus also access the methods that HomeProduct defines.

Part 2: Using Tables and Formulas

Using Tables and Formulas

- Now create two table contents for both the products HC-Optimal and HC-Compact (or, more precisely, for their basic coverage types) with the names "RateTable Optimal 2021-12" and "RateTable Compact 2021-12".

In the Model Explorer, right-click on HC-Optimal/(HC-Compact) → New... → Table content → RateTableHome → Fill in name → Finish.

Using Formulas

- The association between HomeContract and HomeExtraCoverage should be created in HomeContract with the type "Parent to Child". Otherwise the HomeExtraCoverage can have several HomeContracts (and therefore the logic no longer works).

Part 3: Testing with Faktor-IPS

Creating a test case

First test case fails:

```
Can't create instance for toc entry  
TocEntry(TestCase:productdata/test/PremiumComputationTest1.ipstestcase)  
java.lang.reflect.InvocationTargetException  
No asserts implemented in the Java class that represents the test case type.
```

This means that no asserts were implemented in PremiumComputationTest.java:

```
public void executeAsserts(IpsTestResult result) {  
    // begin-user-code  
    // TODO : Inserts the asserts to execute.  
    throw new RuntimeException("No asserts implemented in the Java class that  
represents the test case type.");  
    // end-user-code  
}
```

Comment out the line. Now another error occurs: "Not implemented yet!".

In the method `executeBusinessLogic()` the function `HomeContract.computePremium()` is called. Skip to the function by clicking `computePremium()` and pressing F3 afterwards.

You'll notice that the function is not implemented yet.

```
public void computePremium() {  
    // TODO implement model method.  
    throw new RuntimeException("Not implemented yet!");  
}
```