



# **Tutorial zur Version 2.0**

**(Dokumentversion 1422)**

---

## Inhaltsverzeichnis

Einleitung.....	3
„Hello Faktor-IPS“ .....	5
Arbeiten mit Modell und Sourcecode.....	8
Definition eines einfachen Hausratmodells.....	16
Aufnahme von Produktaspekten ins Modell.....	24
Definition der Produkte.....	35
Zugriff auf Produktinformationen zur Laufzeit.....	40
Verwendung von Tabellen.....	42
Implementieren der Beitragsberechnung.....	47
Flexibilität für die Fachabteilung am Beispiel von Zusatzdeckungen.....	52
Schlussbemerkungen.....	60
Anhang: Tariftabellen.....	62

## Einleitung

Faktor-IPS ist ein OpenSource-Werkzeug zur modellgetriebenen Entwicklung versicherungsfachlicher Softwaresysteme<sup>1</sup> 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 Produktaspekte auch über eine Excel-ähnliche Formelsprache definiert werden. Darüber hinaus können Tabellen verwaltet und fachliche Testfälle definiert und ausgeführt werden.

Dieses Tutorial führt in die Konzepte von und die Arbeitsweise mit Faktor-IPS ein. Als durchgängiges Beispiel verwenden wir dazu eine stark vereinfachte Hausratversicherung. Die grundlegenden Konstruktions- und Modellierungsprinzipien lassen sich auch anhand dieses sehr einfachen fachlichen Modells darstellen. Insbesondere behandeln wir in dem Tutorial wie Möglichkeiten zur Produktkonfiguration inklusive der Änderungen im Zeitablauf in einem fachlichen Modell abgebildet werden und wie ein Gesamtmodell in spartenübergreifende und spartenspezifische Teilmodelle partitioniert werden kann.

Das Tutorial ist wie folgt gegliedert:

- Hello Faktor-IPS  
In dem Kapitel wird ein erstes Faktor-IPS Projekt angelegt und eine erste Klasse definiert.
- Arbeiten mit Modell und Sourcecode  
Anhand eines minimalen spartenübergreifenden Modells wird der Umgang mit dem Modellierungswerkzeug und dem generierten Sourcecode erläutert.
- Definition eines einfachen Hausratmodells  
Aufsetzend auf den spartenübergreifenden Klassen wird das Modell einer einfachen Hausratversicherung erfasst.
- Aufnahme von Produktaspekten ins Modell  
In dem Kapitel wird das Modell der Hausratversicherung um die Möglichkeiten zur Produktkonfiguration erweitert.
- Definition der Hausratprodukte  
Auf Basis des Modells werden nun zwei Hausratprodukte erfasst. Hierzu wird die Produktdefinitionsperspektive 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.
- Verwendung einer Tabelle zur Ermittlung der Tarifzone  
In dem Kapitel wird das Modell um eine Tabelle zur Ermittlung der Tarifzone erweitert und der Zugriff auf den Tabelleninhalt realisiert.
- Implementieren der Beitragsberechnung  
Es wird die Beitragsberechnung implementiert und mit Hilfe eines JUnit-Tests getestet.
- Erweitern des Hausratmodells um Zusatzdeckungen  
In diesem Kapitel wird das Hausratmodell um Zusatzdeckungen erweitert. Die Modellierung

---

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

erlaubt es der Fachabteilung flexibel neue Zusatzdeckungen z. B. gegen Fahrraddiebstahl oder Überspannungsschäden zu definieren, ohne dass jedesmal das Modell erweitert werden muss.

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 [www.faktorips.org](http://www.faktorips.org) herunterladen und installieren.

## „Hello Faktor-IPS“

Im ersten Schritt dieses Tutorial 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 Installationsleitung finden Sie auf [www.faktorips.org](http://www.faktorips.org). In diesem Tutorial verwenden wir Eclipse 3.3 und Faktor-IPS in Englisch. Starten Sie nun Eclipse. Am besten verwenden Sie für dieses Tutorial einen eigenen Workspace. Wenn Faktor-IPS korrekt installiert ist, sollten Sie bei geöffneter Java-Perspektive in der Toolbar folgende Symbole sehen:



Abbildung 1: Faktor-IPS Icons

Faktor-IPS-Projekte sind normale Java-Projekte mit einer zusätzlich Faktor-IPS-Nature. Als erstes legen Sie also ein neues Java-Projekt mit dem Namen „Grundmodell“ an. Hierzu klicken Sie im Menü auf File→New→Java Project. In dem Dialog brauchen Sie lediglich den Namen des Projektes angeben und klicken dann auf *Finish*.

In diesem Projekt werden wir die spartenübergreifenden Modellklassen anlegen. Die Faktor-IPS-Nature fügen Sie dem Projekt hinzu, indem Sie es im Java Package-Explorer markieren und im Kontextmenü *Add IPS-Nature* wählen.

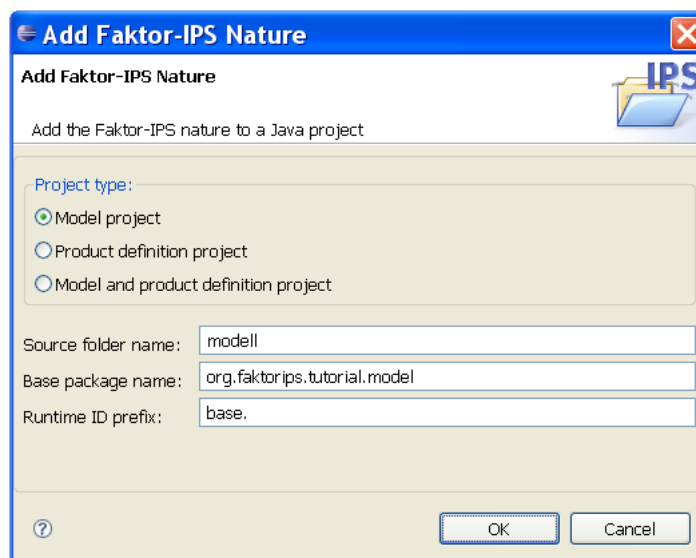


Abbildung 2: Hinzufügen der IPS-Nature

Als Sourceverzeichnis geben Sie „modell“ ein, als Basis-Package für die generierten Java-Klassen „org.faktorips.tutorial.modell“ und drücken *Ok*. Dem Projekt werden die Laufzeitbibliotheken von Faktor-IPS hinzugefügt und das angegebene Sourceverzeichnis („modell“) angelegt. In dem Sourceverzeichnis wird die Modelldefinition abgelegt. 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. Die Bedeutung des RuntimeID Prefixes wird im Kapitel „Definition der Produkte“ erläutert.

Darüber hinaus wurde dem Projekt ein neues Java Sourceverzeichnis mit dem Namen „derived“ hinzugefügt. In dieses Verzeichnis generiert Java Sourcefiles und kopiert XML-Dateien, die zu 100% generiert werden. Der Inhalt des Verzeichnisses kann also jederzeit gelöscht und neu erzeugt werden. Im Gegensatz hierzu enthält das ursprüngliche Java Sourceverzeichnis Dateien, die vom Entwickler bearbeitet werden können und die beim Generieren gemerged werden.

Bevor wir die erste Klasse Vertrag<sup>2</sup> definieren, stellen Sie in den Preferences (*Window→Preferences*) noch ein, dass der Workspace automatisch gebaut wird (*General→Workspace: Build automatically*) und den Compliance Level des Java Compilers auf 1.4 (*Java→Compiler: Compiler Compliance Level*)<sup>3</sup>.

Wechseln Sie zunächst in den Modell-Explorer von Faktor-IPS direkt neben dem Package-Explorer. 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 Menu *Window→ResetPerspektive* auf.

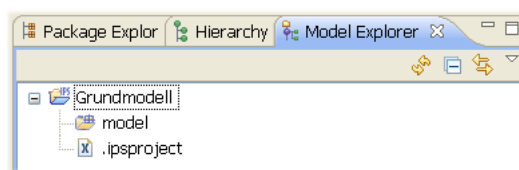



Abbildung 3: Ansicht der Projekte im Modell-Explorer

Im Modell-Explorer wird die Modelldefinition ohne die Java-Details dargestellt. Die spartenübergreifenden Klassen werden wir in einem Package mit dem Namen „base“ ablegen. Zum Anlegen des Packages markieren Sie zunächst das Sourceverzeichnis „modell“ und erzeugen über das Kontextmenü ein neues Package mit dem „base“. Markieren Sie nun das neu angelegte Package im Explorer und drücken auf den Button  in der Toolbar.

---

2 Genau genommen definieren wir natürlich die Klasse „base.Vertrag“.

3 Der Codegenerator von Faktor-IPS generiert Sourcecode für das JDK 1.4. Um Typsicherheit auch bei Collections zu haben, werden an den Schnittstellen Arrays anstatt der generischen Klassen des Collection-Frameworks verwendet. Der Sourcecode ist aber natürlich auch unter 5.0 lauffähig ist.

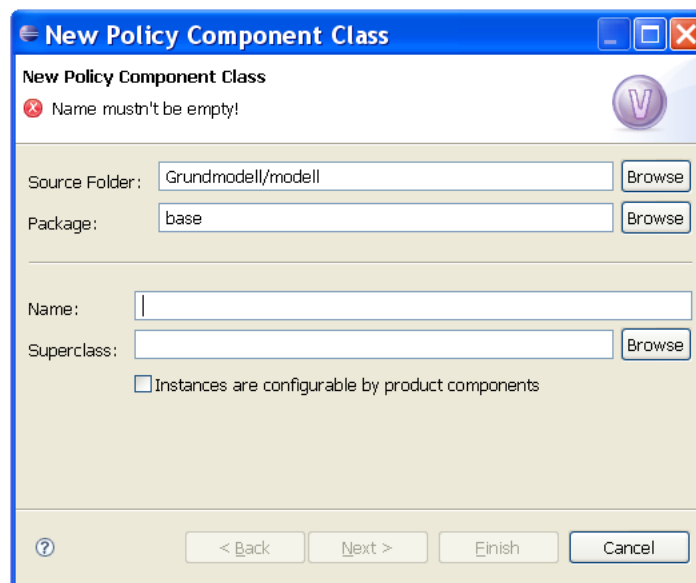


Abbildung 4: Anlegen einer neuen Vertragsklasse

In dem Dialog sind Sourceverzeichnis und Package bereits entsprechend vorbelegt und Sie geben noch den Namen der Klasse an, also Vertrag 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 Vertrag in einer eigenen Datei mit dem Namen „Vertrag.ipspolycympttype“ gespeichert ist.

Weiterhin hat der Codegenerator von Faktor-IPS bereits zwei Java-Sourcefiles erzeugt  
`org.faktorips.tutorial.modell.base.IVertrag` und  
`org.faktorips.tutorial.modell.internal.base.Vertrag`.

Die erste Datei enthält das sogenannte published Interface der Modellklasse Vertrag. Es enthält alle Eigenschaften, die für Clients des Modells sichtbar und nutzbar sind. Da wir bisher noch keine Eigenschaften der Klasse Vertrag definiert haben, enthält dieses Interface erst einmal keine Methoden.

Die Klasse Vertrag ist die modellinterne Implementierung des published Interface. Ein kurzer Blick in den Sourcecode 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.

Durch die Trennung von published Interface und Implementierung können die Modellklassen auf unterschiedliche Packages aufgeteilt werden, ohne dass dies zur Folge hat, dass modellinterne Methoden auch für Clients sichtbar sind<sup>4</sup>.

<sup>4</sup> In Java müssen Methoden public sein, die von einer Klasse eines anderen Package aufgerufen werden. Es kann nicht unterschieden werden, ob es sich um die gleiche oder eine andere Schicht der Softwarearchitektur handelt.

## Arbeiten mit Modell und Sourcecode

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

Als erstes erweitern wir die Klasse Vertrag 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 *New* im Abschnitt *Attributes*. Es öffnet sich der folgende Dialog:

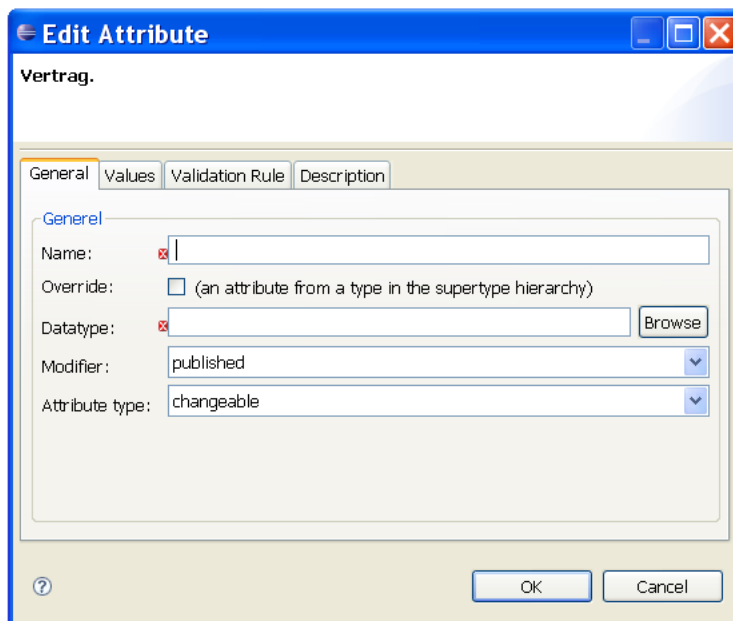


Abbildung 5: Dialog zum Anlegen eines neuen Attributs

Die Felder haben die folgende Bedeutung:

<i><b>Feld</b></i>	<i><b>Bedeutung</b></i>
Name	Der Name des Attributs.
Override	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 <sup>5</sup> .
Datatype	Datentyp des Attributs.
Modifier	Analog zum Modifier in Java. Der zusätzliche Modifier published bedeutet, dass die Eigenschaft ins published Interface aufgenommen wird.

<sup>5</sup> Entspricht der `@override` Annotation in Java 5.



<i>Feld</i>	<i>Bedeutung</i>
Attribute type	<p>Der Typ des Attributs.</p> <ul style="list-style-type: none"> <li>• <b>changeable</b> Änderbare Eigenschaften, also solche mit Getter- und Setter-Methoden.</li> <li>• <b>constant</b> Konstante, nicht änderbare Eigenschaft.</li> <li>• <b>derived (cached, computation by explicit method call)</b> Abgeleitete Eigenschaften im UML Sinne. Die Eigenschaft wird durch eine expliziten Methodenaufruf berechnet und das Ergebnis ist danach über die Getter-Methode abfragbar. Zum Beispiel kann die Eigenschaft bruttobeitrag durch eine Methode berechneBeitrag berechnet und danach über getBruttobeitrag() abgerufen werden.</li> <li>• <b>derived (computation on each call of the getter method)</b> 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 getAlter() aus dem Geburtstag ermittelt werden.</li> </ul>

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. Das published Interface `IVertrag` enthält nun Zugriffsmethoden für das Attribut. Die Implementierung `Vertrag` implementiert diese Methoden und speichert den Zustand in einer privaten Membervariable.

```

/**
 * Membervariable fuer zahlweise.
 *
 * @generated
 */
private String zahlweise = null;

/**
 * {@inheritDoc}
 *
 * @generated
 */
public Integer getZahlweise() {
    return zahlweise;
}

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

```

Das JavaDoc für die Membervariable und für die Getter-Methode ist 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 eine Zeile in die Getter- und Setter-Methode ein, und ergänzen bei der Methode `setZahlweise()` `NOT` hinter der Annotation, also etwa

```

/**
 * {@inheritDoc}
 *
 * @generated
 */
public Integer getZahlweise() {
    System.out.println("getZahlweise");
    return zahlweise;
}

```

```

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

```

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

- Entweder bauen Sie mit Project→Clean das gesamte Projekt neu, oder
- Sie speichern die Modellbeschreibung der Klasse Vertrag erneut.

Nach dem Generieren ist das „System.out.println()“ 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 „Enumeration“ und geben Sie in die Tabelle die Werte 1, 2, 4 und 12 ein<sup>6</sup>.

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. 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 sogenannte Problemmarker, die im Problem-View und in den Explorern sichtbar sind.

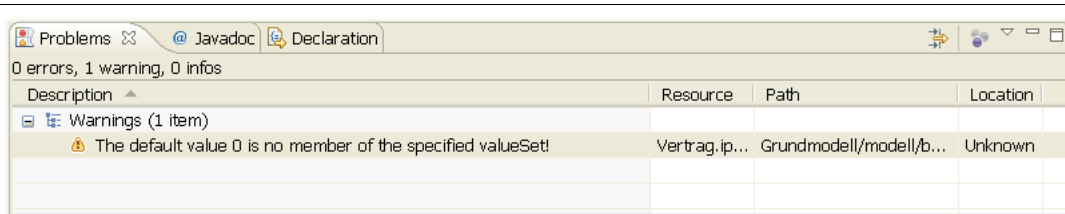


Abbildung 6: Anzeige von Fehlern in der Problems-View

Setzen Sie als Defaultwert wieder <null> ein und speichern die Vertragsklasse. Die Warnung wird damit wieder aus dem Problem-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 vorgelegt 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 Deckung und der Kompositionsbeziehung zwischen Vertrag und Deckung gemäß dem folgenden Diagramm:

<sup>6</sup> Sie können auch eine Klasse Zahlungsweise als Datentyp verwenden, wenn Sie diese in der „ipsproject“-Datei registrieren (s. folgendes Kapitel).

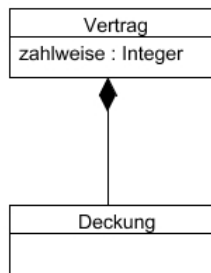


Abbildung 8: Spartenübergreifendes Modell

Legen Sie zunächst die Klasse Deckung analog zur Klasse Vertrag an. Danach wechseln Sie zurück in den Editor, der die Vertragsklasse anzeigt. Starten Sie den Assistenten zur Anlage einer neuen Beziehung durch Klicken auf den *New* Button rechts neben dem Abschnitt, der mit *Associations* überschrieben ist.

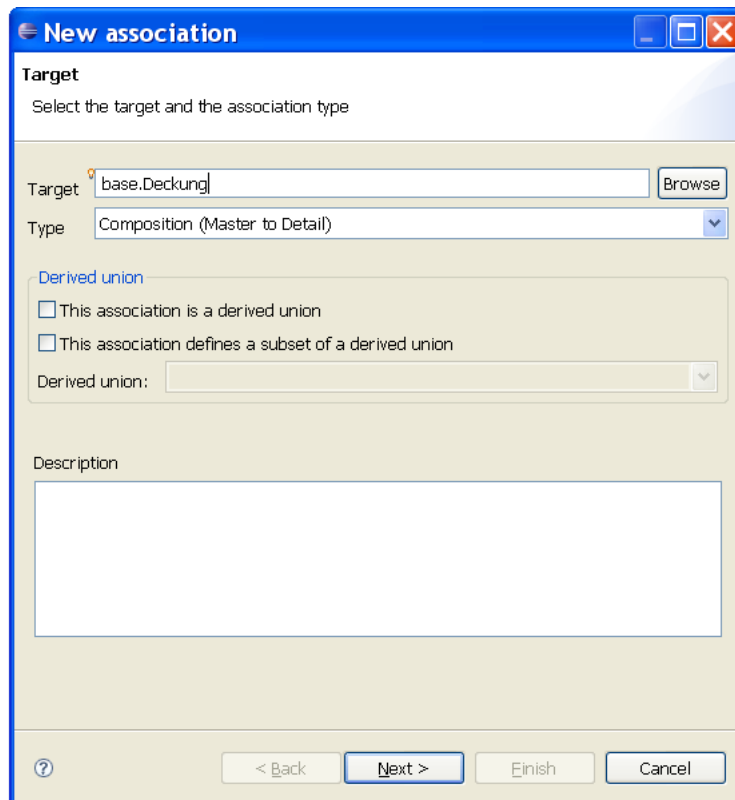


Abbildung 9: Anlegen einer neuen Beziehung

Als Target wählen Sie bitte die gerade angelegte Klasse Deckung aus. Hier steht Ihnen wieder die Vervollständigung mit *STRG-Space* zur Verfügung. Die mit *Derived Union* überschriebene Box ignorieren Sie zunächst. Das Konzept wird später im Tutorial erläutert.

Auf der nächsten Seite geben Sie als minimale Kardinalität 0 und als maximale Kardinalität \* ein,

als Rollennamen Deckung bzw. Deckungen. Die Mehrzahl ist erforderlich, damit der Codegenerator verständlichen Sourcecode generieren kann.

Abbildung 10: Rollennamen und Kardinaliten einer Beziehung

Auf der nächsten Seite können Sie auswählen, ob es auch eine Rückwärtsbeziehung von Deckung zu Vertrag 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 *New inverse association* und gehen zur nächsten Seite. Hier brauchen Sie nur noch die Rollenbezeichnung einzugeben. Mit *Finish* legen Sie die beiden Beziehungen (vorwärts und rückwärts) an und speichern danach noch die Klasse Vertrag. Wenn Sie sich jetzt die Klasse Deckung ansehen, ist dort die Rückwärtsbeziehung eingetragen.

Zum Abschluss werfen wir noch einen kurzen Blick in den generierten Sourcecode. In das published Interface `IVertrag` wurden Methoden generiert, um Deckungen zu einem Vertrag hinzuzufügen, aus einem Vertrag zu entfernen, etc. In dem Interface `IDeckung` gibt es eine Methode, um zum Vertrag zu navigieren, zu dem die Deckung gehört. Kompositbeziehungen werden also immer über die Containerklasse hergestellt (und aufgelöst). Wenn sowohl die Vorwärts- als auch die Rückwärtsbeziehung im Modell definiert ist, werden hierbei beide Richtung berücksichtigt. Das heißt nach dem Aufrufe von `addDeckung(IDeckung d)` auf einer Vertragsinstanz `v` liefert `d.getVertrag()` wieder `v` zurück. Dies zeigt ein kurzer Blick in die Implementierung der Methode `addDeckung`<sup>7</sup> in der Klasse `Vertrag`.

<sup>7</sup> Die generierten Javadocs zeigen wir von nun an nicht mehr im Tutorial, da die Methoden im Text erläutert werden.

```
public void addDeckung(IDeckung objectToAdd) {
    if (objectToAdd == null) {
        throw new NullPointerException("Can't add null to relation Deckung of " + this);
    }
    if (deckungen.contains(objectToAdd)) {
        return;
    }
    deckungen.add(objectToAdd);
    ((DependantObject) objectToAdd).setParentModelObjectInternal(this);
}
```

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

## Definition eines einfachen Hausratmodells

In diesem Abschnitt werden wir ein einfaches Hausratmodell definieren, welches uns durch den Rest des Tutorials begleiten wird. Die folgende Abbildung zeigt das Modell.

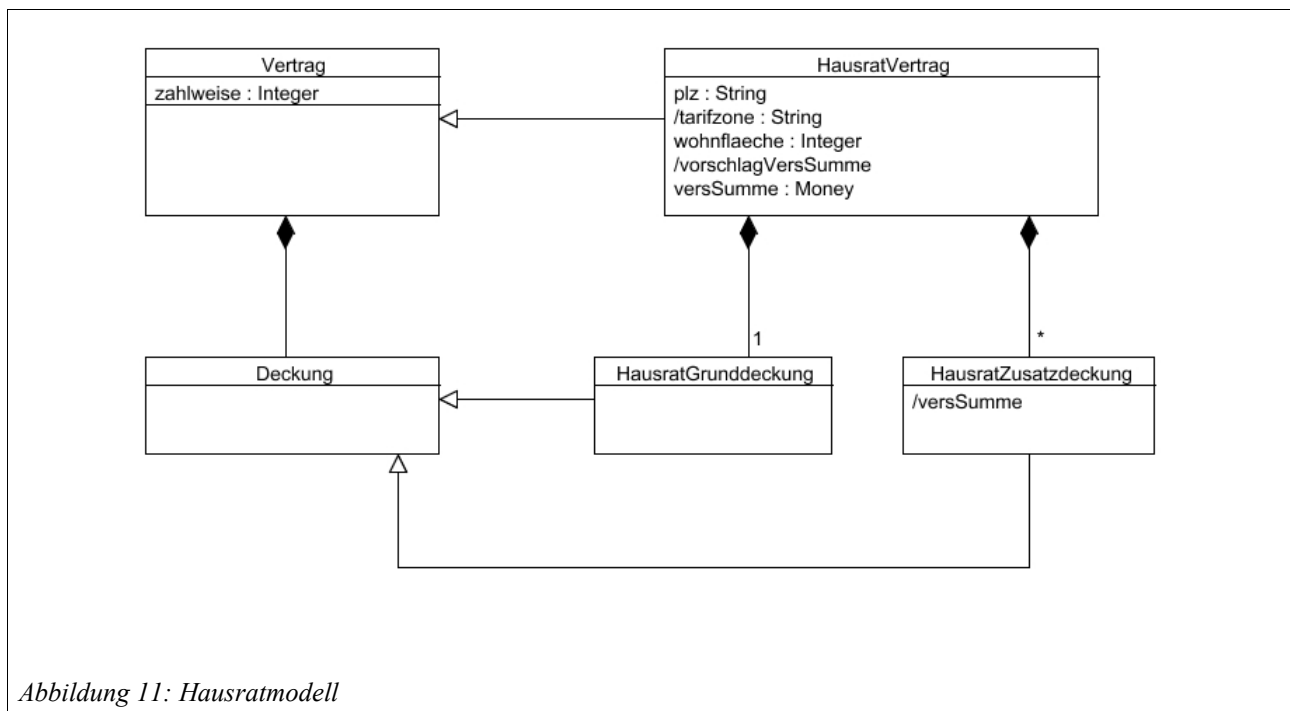


Abbildung 11: Hausratmodell

Der Hausratvertrag ist natürlich ein spezifischer Vertrag. Die genaue Bedeutung der einzelnen Attribute erläutern wir weiter unten. Jeder Hausratvertrag muss genau eine Grunddeckung enthalten und kann beliebig viele Zusatzdeckungen enthalten. Hausratgrunddeckung und Hausratzusatzdeckung sind Ableitungen von Deckung. 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 Überspannungsschäden. Die Zusatzdeckung stellen wir bis zum Schluss des Tutorial zurück und konzentrieren uns bis dahin auf den Hausratvertrag und die Hausratgrunddeckung.

Die hausratspezifischen Klassen wollen wir in einem zweiten Projekt mit dem Namen Hausratmodell verwalten. Jede Sparte in einem eigenen Projekt zu verwalten bietet den Vorteil, dass unterschiedliche Entwicklerteams an den Projekten arbeiten können und die Sparten voneinander unabhängige Releasezyklen haben können. Legen Sie also zunächst dieses neue Projekt an (erst ein Java-Projekt anlegen, dann die IPS-Nature hinzufügen, s. Kapitel 2).

Da die hausratspezifischen Klassen von den spartenübergreifenden Klassen Vertrag und Deckung ableiten, müssen die spartenübergreifenden Klassen natürlich in dem neuen Projekt bekannt sein. Hierzu gibt es in Faktor-IPS ein Konzept analog zum Classpath in Java. In jedem Faktor-IPS-Projekt gibt es eine Datei „ipsproject“, in dem die Eigenschaften des Projektes gespeichert werden. Öffnen Sie die Datei per Doppelklick. Das XML-Format ist im Detail direkt in der Datei dokumentiert. Suchen Sie bitte das XML-Tag `IpsObjectPath`. Der `IpsObjectPath` beschreibt, wo Faktor-IPS für das Projekt nach Faktor-IPS-Objekten wie z. B. den Klassendefinitionen sucht.

Fügen Sie den im folgenden Ausschnitt fett markierten Eintrag zum `IpsObjectPath` Tag hinzu und speichern die Datei.

```
<IpsObjectPath ...>
  <Entry .../>
  <Entry type="project" referencedIpsProject="Grundmodell"/>
</IpsObjectPath ...>
```

Die Projekteigenschaften werden mit dem Speichern der Datei aktiv und die spartenübergreifenden Klassen können damit aus dem Hausratmodell referenziert werden. Damit auch die Javaklassen im Hausratmodell-Projekt verfügbar sind, müssen Sie natürlich auch den Java Classpath entsprechend anpassen. Hierzu rufen Sie im Package-Explorer das Kontextmenü des Projektes auf und wählen dort *Build Path* → *Configure Build Path*. Auf der Tabseite *Projects* fügen Sie das Grundmodell zu dem Hausratmodell hinzu.

Sie haben sich vielleicht schon gefragt, wo definiert wird, in welchen Java-Sourcefolder und in welche Java-Packages Faktor-IPS den generierten Sourcecode ablegt. Dies geschieht ebenfalls im „ipsproject“ File im `IpsObjectPath`.

Von besonderer Bedeutung in der Datei ist noch das `<Datatypes>` Tag. Hier wird festgelegt welche Datentypen in dem Projekt verwendet werden können. Mit Faktor-IPS werden vordefinierte Standard-Datentypen mitgeliefert<sup>8</sup>. Beim Anlegen eines neuen Projektes werden zunächst alle diese Datentypen in das Projekt aufgenommen. In einem Projekt können auch alle Datentypen verwendet werden, die in Projekten definiert sind, von denen dieses abhängt. Sie können also im Projekt „Hausratmodell“ alle Datentypen innerhalb des `<UsedPredefinedDatatypes>` Tags löschen.

Nach diesen Vorarbeiten können wir nun das oben beschriebene Modell vollständig eingeben. Als erstes legen wir analog zum Grundmodell ein (Faktor-IPS-)Package „hausrat“ im Sourceverzeichnis an. Dies geht einfach über das Kontextmenüs des Modellexplorers.

Nun erzeugen Sie die Klasse „HausratVertrag“ und geben in dem Dialog die Superklasse des spartenübergreifenden Modells, also „base.Vertrag“ an (an den qualifizierten Namen denken). Definieren Sie nun die Attribute der Klasse:

<i><b>Name : Datentyp</b></i>	<i><b>Beschreibung, Bemerkung</b></i>
plz : String	Postleitzahl des versicherten Hausrats
/tarifzone : String	Die Tarifzone (I, II, III, IV oder V) ergibt sich aus der Postleitzahl und ist maßgeblich für den zu zahlenden Beitrag. => Achten Sie also bei der Eingabe darauf den AttributeType auf derived (computation on each method call) zu setzen!
wohnflaeche : Integer	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. Für einen unbeschränkten Wertebereich wird das Feld max auf <null> gesetzt.

<sup>8</sup> Eigene Java Klassen, die einen Wert repräsentieren, können Sie ebenfalls in der „ipsproject“-Datei bekannt machen. Voraussetzung ist, dass die Klasse im Java-Classpath verfügbar ist und ein Wert(-objekt) in einen String konvertiert werden kann und umgekehrt.



## Definition eines einfachen Hausratmodells

<i>Name : Datentyp</i>	<i>Beschreibung, Bemerkung</i>
/vorschlagVersSumme : Money	Vorschlag für die Versicherungssumme. Wird auf Basis der Wohnfläche bestimmt. => Achten Sie bei der Eingabe darauf den AttributeType auf derived (computation on each method call) zu setzen!
versSumme : Money	Die Versicherungssumme. Der erlaubte Wertebereich ist min=0 EUR und max=<null>.

Der Editor, der die Klasse HausratVertrag anzeigt, müsste nun wie folgt aussehen:

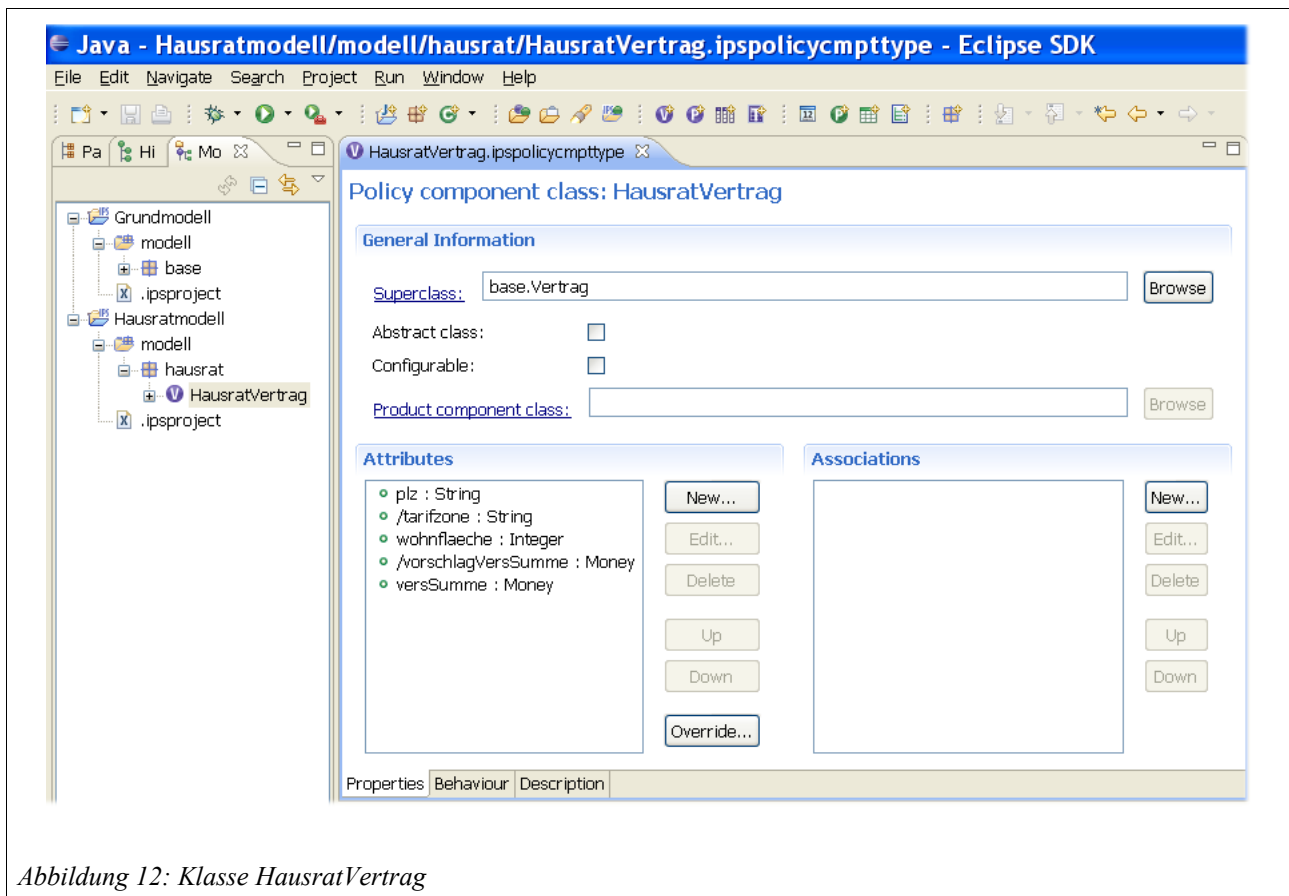


Abbildung 12: Klasse HausratVertrag

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.

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public String getTarifzone() {
    return "I"; // TODO wird spaeter anhand einer Tarifzonentabelle ermittelt
}

/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getVorschlagVersSumme() {
    // TODO: der multitiplikator wird spaeter aus den produktdaten ermittelt.
    return Money.euro(650).multiply(wohnflaeche);
}
```

Denken Sie daran, hinter `@generated` ein `NOT` zu schreiben, damit der manuell eingefügte Code nicht überschrieben wird!

Legen Sie nun die Klasse „HausratGrunddeckung“ an. Bevor wir nun die Beziehungen zwischen den Hausratklassen anlegen, analysieren wir noch einmal das oben abgebildete UML Modell. Es gibt eine Kompositbeziehung zwischen den spartenübergreifenden Klassen Vertrag und Deckung einerseits und zwischen dem Hausratvertrag und der Hausratzusatzdeckung bzw. -grunddeckung andererseits. Unsere Erwartungshaltung ist dabei die folgende:

- Zum Hausratvertrag kann eine Hausratgrunddeckung und beliebig viele Hausratzusatzdeckungen hinzugefügt werden, aber zum Beispiel keine Haftpflichtdeckung.
- Wenn man eine Hausratgrunddeckung oder Zusatzdeckung zum Hausratvertrag hinzufügt, erhält man diese auch zurück, wenn man sich vom Vertrag alle Deckungen zurückgeben lässt.

Diese erwartete Semantik ist aber nicht explizit in dem Modell enthalten. Die Beziehungen zwischen Vertrag und Deckung bzw. Hausratvertrag und Hausratgrunddeckung/-Zusatzdeckung sind voneinander unabhängige Beziehungen.

In UML kann die von uns gewollte Semantik ins Modell abgebildet werden, indem wir die Beziehung VertragDeckung als abgeleitete Vereinigung („derived union“) definieren und in die Beziehungen zwischen Hausratvertrag und seinen Deckungsklassen jeweils auf diese abgeleitete Beziehung referenzieren. Auf Instanzebene werden damit die Beziehungen in den Hausratklassen verwaltet. Zusätzlich kann in der spartenübergreifende Vertragsklasse (base.Vertrag) auf alle Deckungen zugegriffen werden und damit spartenübergreifende Funktionalität entwickelt werden. So kann zum Beispiel der Beitrag eines Vertrages mit einer Schleife über alle Deckungen ermittelt werden. Das folgende Klassendiagramm zeigt das Modell unter Verwendung der abgeleiteten Vereinigung in UML Notation.

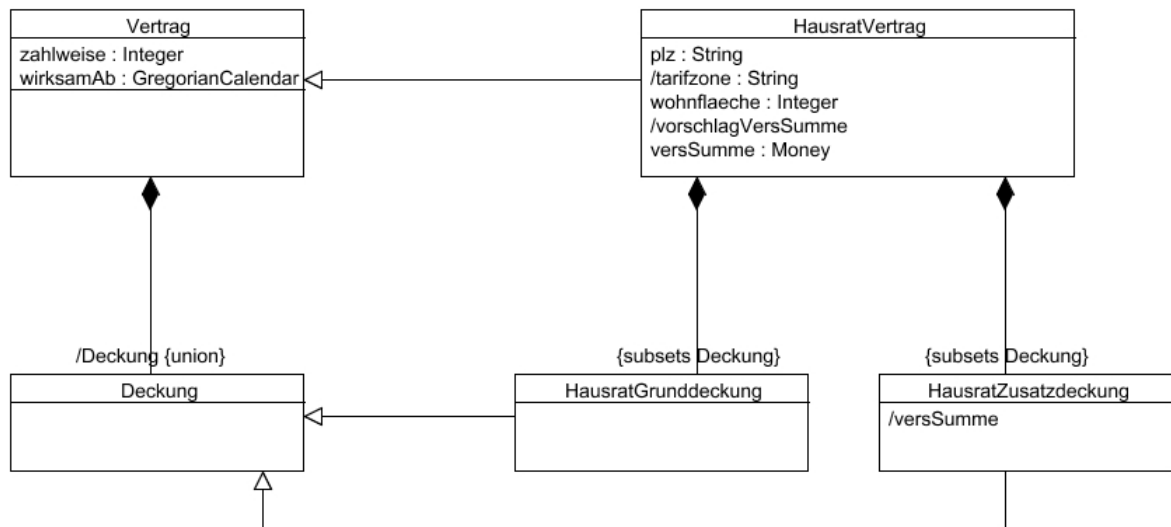


Abbildung 13: Hausratmodell mit Verwendung einer abgeleiteten Vereinigung

Faktor-IPS unterstützt explizit die Modellierung von abgeleiteten Vereinigungen. Als erstes markieren wir nun die Vertrag-Deckung-Beziehung als abgeleitete Vereinigung. Hierzu öffnen Sie zunächst den Editor für die Klasse Vertrag und dann den Dialog zur Bearbeitung der Beziehung. In dem Dialog haken Sie die entsprechende Checkbox an wie in der folgenden Abbildung zu sehen ist:

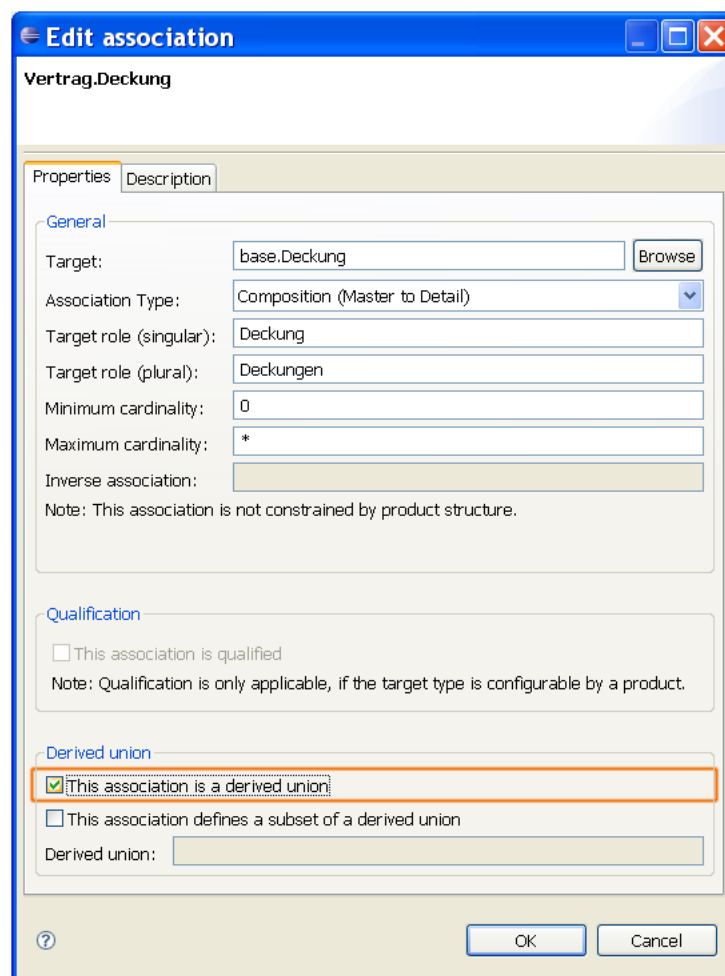


Abbildung 14: Markieren einer Beziehung als abgeleitete Vereinigung

Markieren Sie nun noch die Klasse Vertrag als abstrakt. Dies ist erforderlich, da nun die für die Beziehung generierten Methoden wie zum Beispiel `getDeckungen()` nur im published Interface `IVertrag` definiert sind, aber nicht mehr in der Klasse Vertrag implementiert werden, sondern erst in den spartenspezifischen Klassen (Codebeispiel s.u.). Speichern Sie die Klasse. Vielleicht ist Ihnen aufgefallen, dass mit dem Speichern der Klasse Vertrag auch die Klasse HausratVertrag als fehlerhaft markiert wurde. Das liegt daran, dass Faktor-IPS die Abhängigkeiten zwischen den Modellklassen verwaltet und bei Änderungen an einer Klasse auch alle abhängigen Klassen neu prüft. Da das Modell im jetzigen Zustand einen Fehler enthält, ist aktuell auch der generierte Sourcecode nicht fehlerfrei.

Nun können Sie die Beziehung zwischen Hausratvertrag und Hausratgrunddeckung mit dem Ihnen schon bekannten Assistenten anlegen. Sobald Sie die Klasse HausratGrunddeckung als Target ausgewählt haben, belegt Faktor-IPS automatisch vor, dass die neue Beziehungen eine Teilmenge der abgeleitete Vereinigungsbeziehung ist. Denken Sie bitte daran die maximale Kardinalität der Beziehung auf 1 zu setzen.

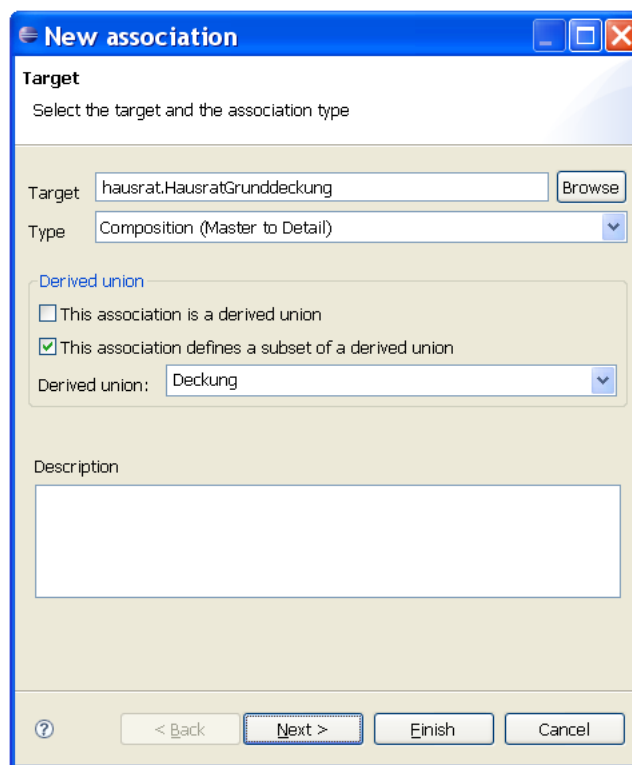


Abbildung 15: Anlegen der Beziehung zwischen Hausratvertrag und Grunddeckung

Das gesamte Modell sollte im Modellexplorer nun wie folgt aussehen:

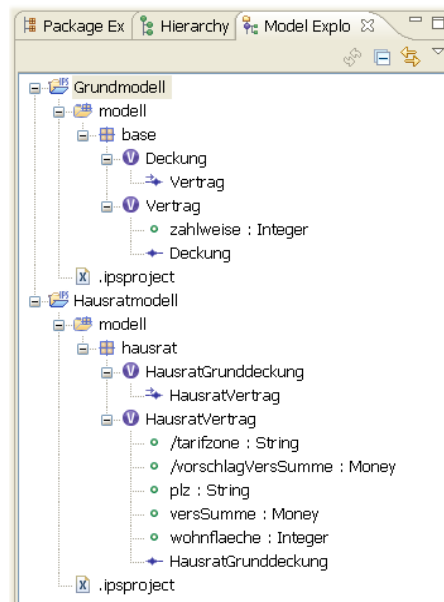


Abbildung 16: Zwischenstand des Modells im Modellexplorer

Werfen wir zum Abschluss des Kapitels noch einen Blick in den generierten Sourcecode<sup>9</sup>. Im published Interface `IVertrag` gibt es eine Methode `getDeckungen()`. Diese wird (in der jetzt abstrakten) Klasse `Vertrag` nicht implementiert. Die generierte Implementierung dieser Methode in der Klasse `HausratVertrag` liefert die Grunddeckung zurück, wenn der Hausratvertrag eine Grunddeckung enthält. Der folgende Kasten enthält den generierten Sourcecode dieser Methode.

```
public IDeckung[] getDeckungen() {
    IDeckung[] result = new IDeckung[getAnzahlDeckungen()];
    int counter = 0;
    if (getHausratGrunddeckung() != null) {
        result[counter++] = getHausratGrunddeckung();
    }
    return result;
}
```

Wenn wir zum Schluß des Tutorials auch noch die Zusatzdeckung in das Modell aufgenommen haben, wird die Methode sowohl die Grunddeckung als auch die Zusatzdeckungen zurückgeben.

```
public IDeckung[] getDeckungen() {
    IDeckung[] result = new IDeckung[getAnzahlDeckungen()];
    int counter = 0;
    if (getHausratGrunddeckung() != null) {
        result[counter++] = getHausratGrunddeckung();
    }
    IDeckung[] elements = getHausratZusatzdeckungen();
    for (int i = 0; i < elements.length; i++) {
        result[counter++] = elements[i];
    }
    return result;
}
```

---

<sup>9</sup> Sollte der Java Sourcecode Compilefehler enthalten, haben Sie wahrscheinlich vergessen, das Projekt Grundmodell in den Classpath vom Projekt Hausratmodell aufzunehmen.

## Aufnahme von Produktaspekten ins Modell

Im vierten Schritt des Tutorials beschäftigen wir uns nun – endlich – 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 Eigenschaften unserer Klasse HausratVertrag inkl. der von Vertrag geerbten Eigenschaften an und überlegen, was für diese Eigenschaften in einem Produkt konfigurierbar sein soll:

<i><b>Eigenschaft von Hausratvertrag</b></i>	<i><b>Konfigurationsmöglichkeiten</b></i>
zahlweise	Die im Vertrag erlaubten Zahlweisen. Der Vorgabewert für die Zahlweise bei Erzeugung eines neuen Vertrags.
wohnflaeche	Bereich (min, max), in dem die Wohnfläche liegen muss.
vorschlagVersSumme	Definition eines Vorschlagwertes für einen Quadratmeter Wohnfläche. Der Vorschlag für die Versicherungssumme ergibt sich dann durch Multiplikation mit der Wohnfläche <sup>10</sup> .
versSumme	Bereich, in dem die Versicherungssumme liegen muss.

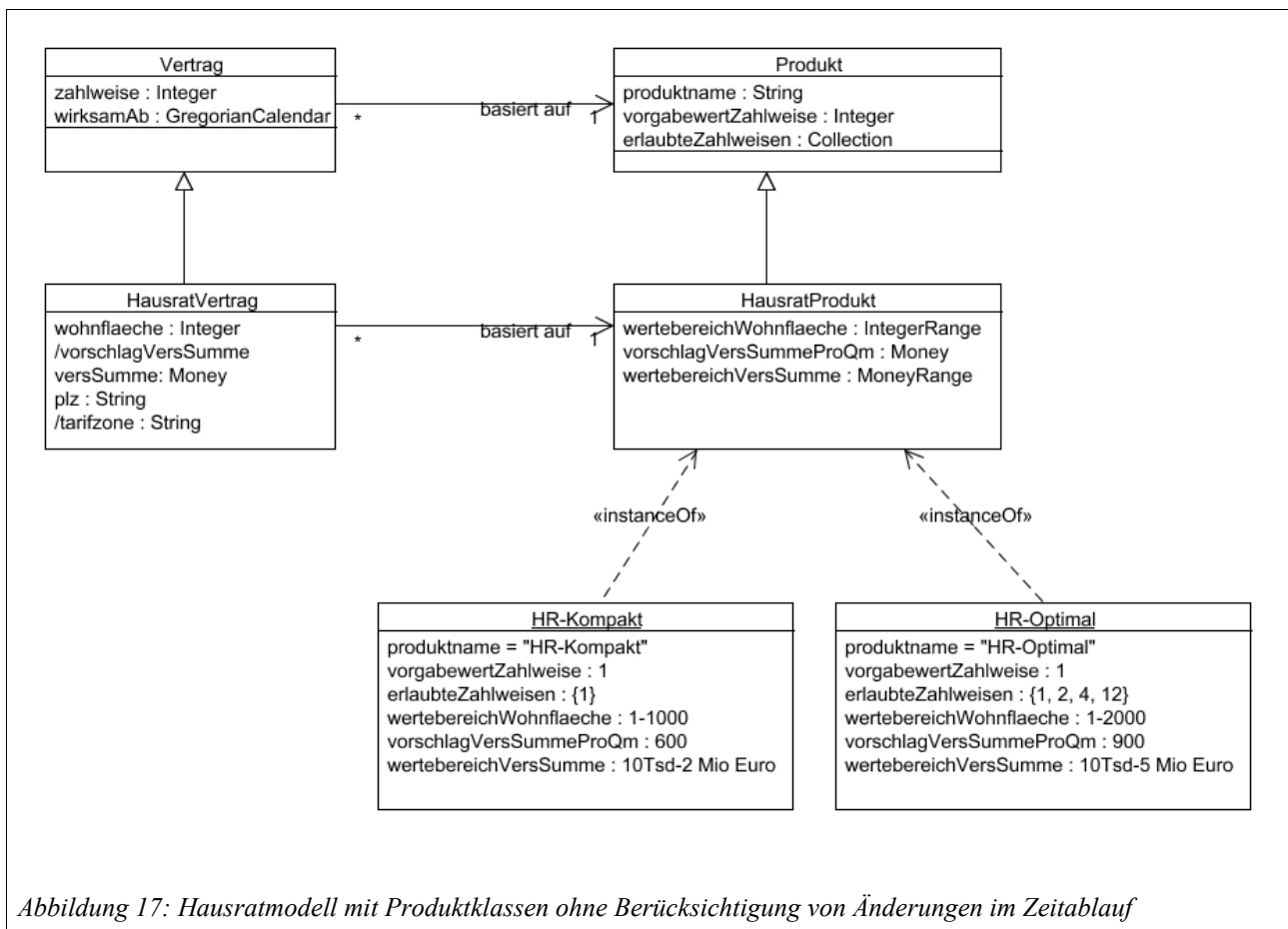
In diesem Tutorial wollen wir zwei Hausratprodukte erstellen. HR-Optimal soll einen umfangreichen Versicherungsschutz gewähren während HR-Kompakt einen Basisschutz zu einem günstigen Beitrag bietet. Die folgende Tabelle zeigt die Eigenschaften der beiden Produkte bzgl. der oben aufgeführten Konfigurationsmöglichkeiten:

<i><b>Konfigurationsmöglichkeit</b></i>	<i><b>HR-Kompakt</b></i>	<i><b>HR-Optimal</b></i>
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	10Tsd – 2Mio Euro	10Tsd – 5Mio Euro

Wir bilden dies im Modell ab, indem wir eine Klasse Produkt einführen. Ein Vertrag basiert auf dem Produkt, auf einem Produkt können beliebig viele Verträge basieren. Analog führen wir eine Klasse „HausratProdukt“ ein. Das Produkt enthält die Eigenschaften und Konfigurationsmöglichkeiten, die bei allen Verträgen, 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:

---

<sup>10</sup> Alternativ könnten wir auch eine Formel zur Berechnung des Vorschlags als Konfigurationsmöglichkeit verwenden. Zunächst beschränken wir uns aber auf den Faktor.



Nun ändern sich Produkte aber im Laufe der Zeit. Bei Versicherungsprodukten unterscheidet man dabei zwischen zwei Arten der Änderung<sup>11</sup>:

### Version

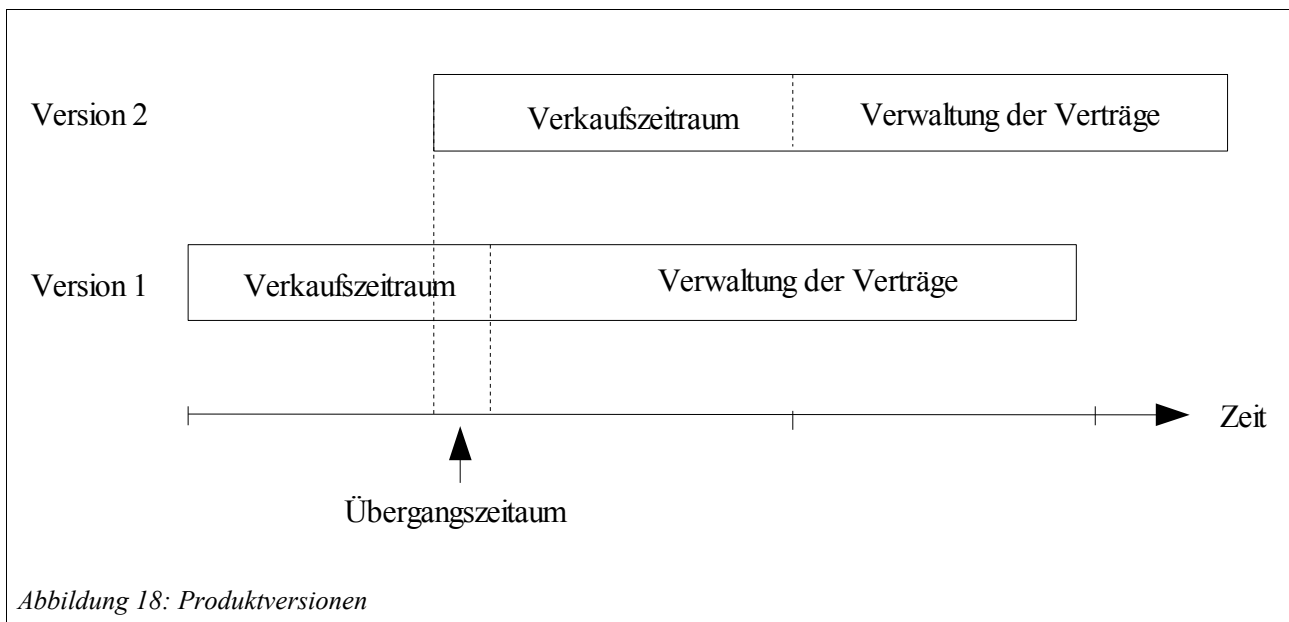
Versionen werden aufgelegt, um für das Neugeschäft geänderte Bedingungen anbieten zu können. Verträge können während des Verkaufszeitraums einer Version zu den in der Version definierten Bedingung abgeschlossen werden.

Bestehende Verträge bleiben von der Einführung einer neuen Version unberührt, es sei denn, es findet ein expliziter Produkt(versions)wechsel statt.

I.d.R. gibt es zu einem Zeitpunkt eine für das Neugeschäft gültige Version. Das ist die, in deren Verkaufszeitraum der Zeitpunkt fällt. Es kann allerdings zu einem Zeitpunkt auch mehrere verkaufbare Versionen eines Produktes geben. In der Praxis kommt dies vor, wenn in der Übergangszeit von einer alten auf eine neue Version beide verkauft werden. Dies zeigt die folgende Abbildung:

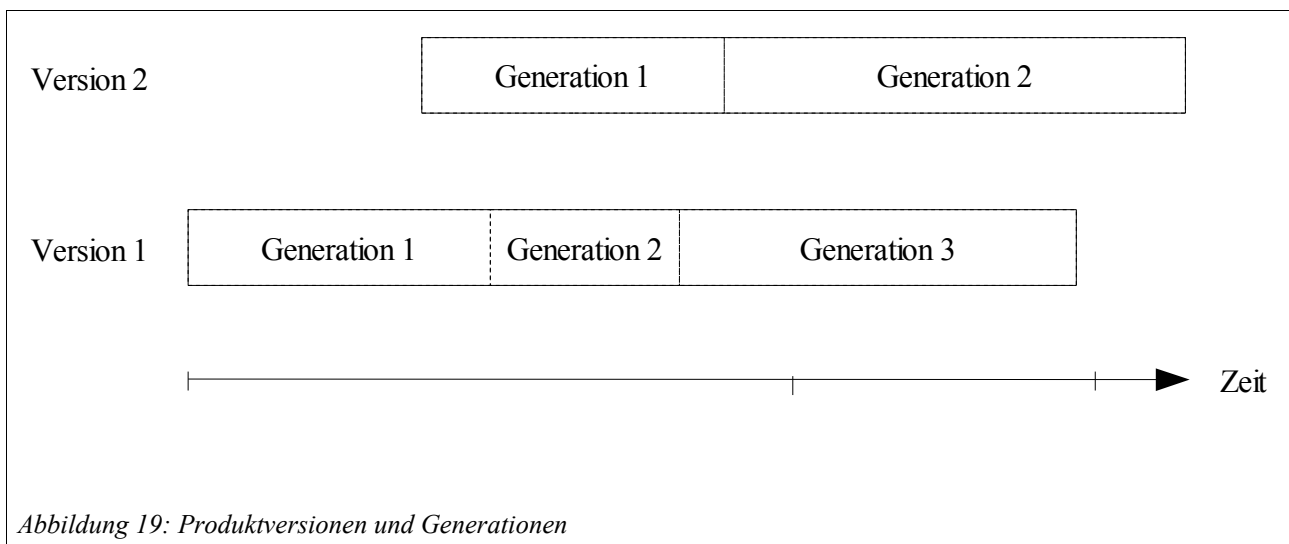
<sup>11</sup> Leider existiert keine einheitliche Namensgebung für die beiden Arten der Produktänderung. Im Tutorial verwenden wir die Begriffe wie Sie vom GDV in der Anwendungarchitektur der Versicherungswirtschaft (VAA) definiert sind. In Faktor-IPS kann die verwendete Namensgebung sowohl für die Benutzeroberfläche (Window→Preferences: Faktor-IPS: Naming scheme for changes over time) als auch für den generierten Sourcecode (in der „ipsproject“ Datei im Abschnitt GeneratedSourcecode) konfiguriert werden.



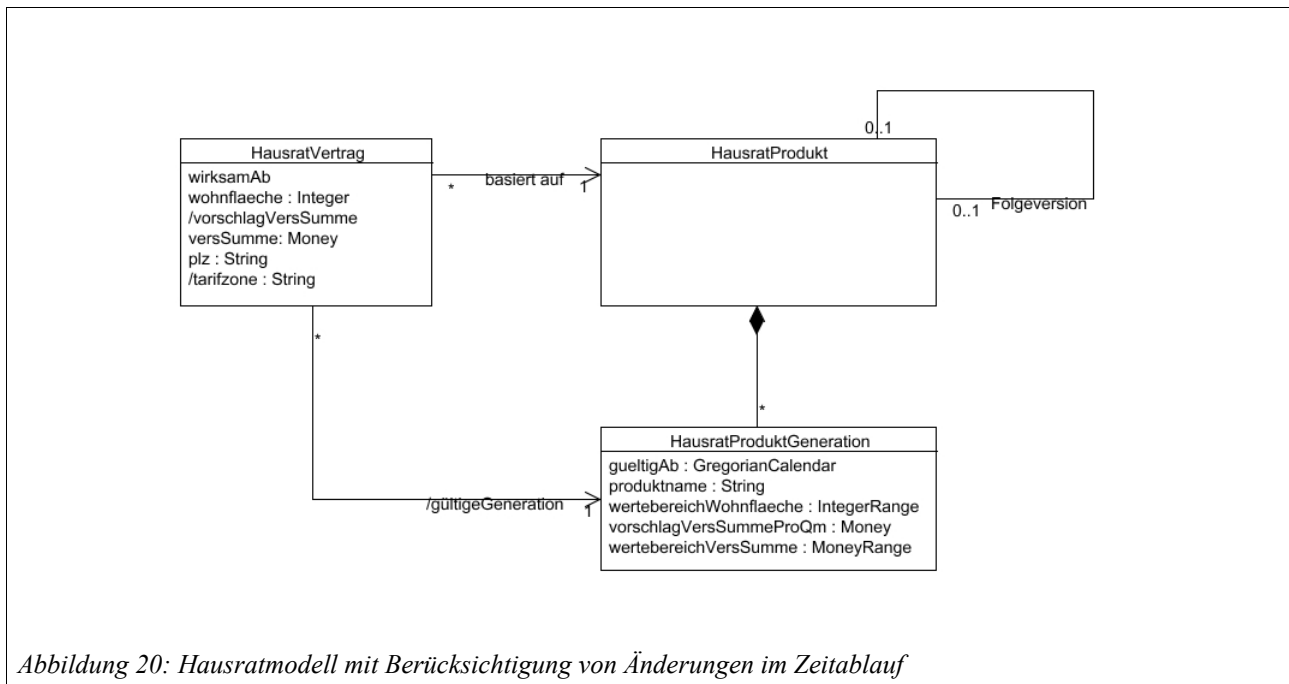


## Generation

Eine Generation ist eine Änderung, die für alle auf Basis einer Version abgeschlossenen Verträge gelten soll. Innerhalb des Gültigkeitszeitraums einer Generation gibt es keine Änderungen. Zu einem gegebenen Zeitpunkt gibt es genau eine gültige Generation einer Produktversion. Den Zusammenhang zwischen Versionen und Generationen zeigt die folgende Abbildung.




Wie berücksichtigen wir die Änderungen im Zeitablauf im Modell? Wir gehen davon aus, dass sich alle Eigenschaften im Zeitablauf ändern können. Die oben modellierten Eigenschaften sind also keine Eigenschaften des Hausratproduktes sondern der Produktgeneration. Die folgende Abbildung zeigt das Modell im Überblick, wobei wir aus Übersichtlichkeitsgründen hier nur die hausratspezifischen Klassen zeigen.



Das Konzept Produktversion bilden wir über eine Beziehung zwischen Produkten ab, da die Unterscheidung zwischen einem neuen Produkt und einer neuen Version sehr unscharf ist. Ob zum Beispiel die Umstellung des Kfz-Tarifs auf einen Scoringtarif eine neue Version oder ein neues Produkt darstellt ist eine eher akademische Betrachtung<sup>12</sup>.

Wenn der Vertrag auf Produkteigenschaften zugreifen muss, wird offensichtlich ein Datum benötigt, um die gültige Produktgeneration zu ermitteln. In einem Bestandssystem wird hierzu das Wirksamkeitsdatum, ab dem der Vertragsstand gültig ist, verwendet, in einem Angebotssystem, kann auch der Versicherungsbeginn verwendet werden. Wir verwenden in dem Tutorial ein Attribut *wirksamAb* an der Klasse Vertrag. Genau genommen repräsentiert die Klasse Vertrag damit eher den Stand eines Vertrags, der von diesem Datum an wirksam ist (bis zum nächsten Vertragsstand)<sup>13</sup>.

Genug Theorie, erweitern wir unser Modell in Faktor-IPS um die Produktklassen. Als erstes definieren wir die spartenübergreifende Klasse „base.Produkt“. Zum Erzeugen klicken Sie in der Toolbar auf den Button . In dem Wizard tragen Sie den Namen der neuen Klasse an („Produkt“) und geben im Feld *Policy component class* an, welche Klasse konfiguriert wird, in diesem Fall also Vertrag (bzw. genauer base.Vertrag) und drücken *Finish*. Es öffnet sich der Editor zur Bearbeitung von Produktklassen.

12 Die Beziehung „Nachfolgeversion“ ist nicht auf Instanzen der selben Klasse beschränkt. So kann auch ein strukturell sehr unterschiedliches Produkt (wie z. B. ein neuer Scoringtarif) Nachfolger eines bestehenden Produktes sein.

13 Da der Vertrag selbst lediglich die identifizierende Klammer über die Vertragsstände darstellt und kaum benötigt wird, nennen wir den Vertragsstand verkürzend Vertrag. Sie können dies natürlich auch anders handhaben.

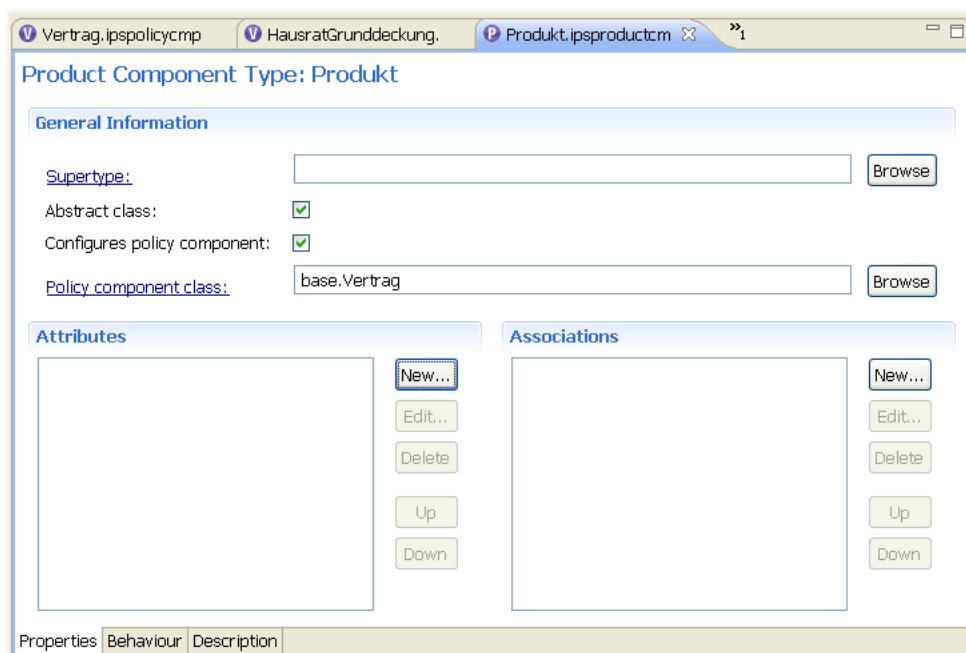


Abbildung 21: Editor für Produktklassen

Im Abschnitt *General information* sehen wir die gerade im Wizard eingegebene Information, dass die Klasse Produkt die Klasse Vertrag konfiguriert. Ansonsten ist der Aufbau der ersten Seite des Editors analog zum Editor für Vertragsklassen<sup>14</sup>.

Gleichzeitig hat Faktor-IPS nun die published Interfaces `IProdukt` und `IProduktGen` sowie die zugehörigen Implementierungsklassen `Produkt` und `ProduktGen` generiert. Im published Interface des Produktes gibt es eine Methode, um zu einem Stichtag die gültige Generation zu ermitteln.

Im spartenübergreifenden Teil des Modell sollen wie bereits beschrieben die folgenden Aspekte in der Klasse Produkt konfigurierbar sein:

- Der Name des Produktes sowie
- 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. Wie für Attribute von Vertragsklassen können die erlaubten Werte über Bereiche oder Aufzählungen eingeschränkt werden. Für Produktnamen machen wir von dieser Möglichkeit aber keinen Gebrauch. Die folgende Abbildung zeigt den entsprechenden Dialog.

<sup>14</sup> Sie können in den Preferences einstellen, ob Sie alle Informationen zu einer Klasse auf einer Seite oder auf zwei Seiten dargestellt bekommen möchten.

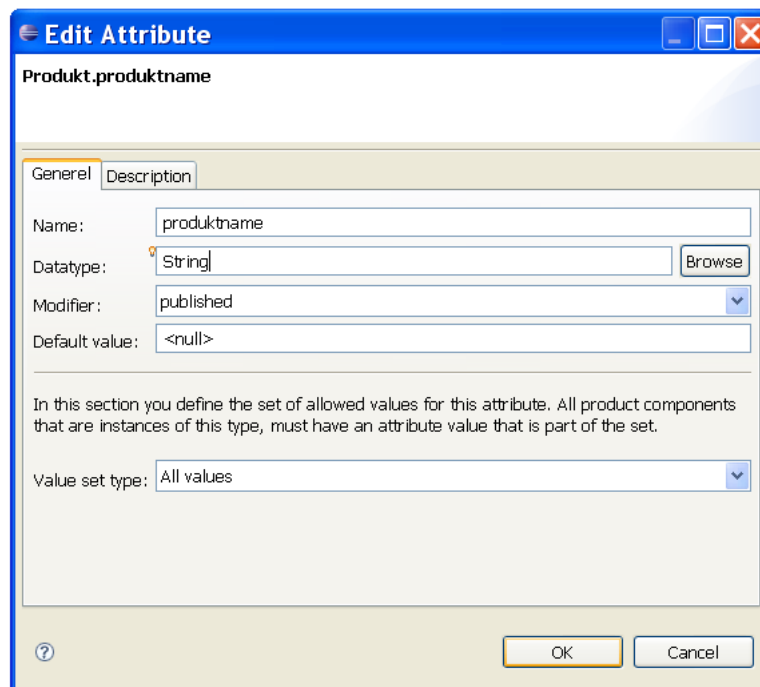


Abbildung 22: Dialog zum Editieren von Produktattributen

Nun definieren wir, dass die in einem Vertrag erlaubten Zahlungsweisen und der Vorgabewert für die Zahlweise im Produkt konfiguriert werden können. Hierzu öffnen wir zunächst den Editor für die Klasse Vertrag. In den Abschnitt *General information* hat der Wizard eingetragen, dass die Klasse Vertrag durch die Klasse Produkt konfiguriert wird.

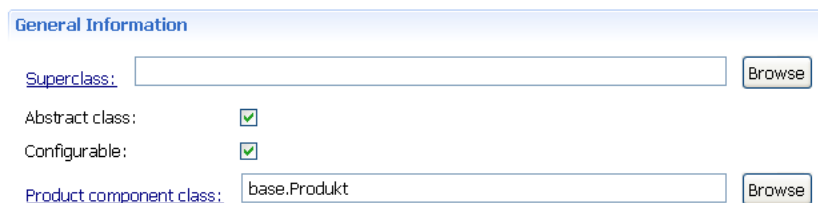


Abbildung 23: General Information Abschnitt im Editor für die Klasse Vertrag

Nun öffnen Sie den Dialog zum Editieren des Attributes „zahlweise“. Da Verträge nun als konfigurierbar definiert sind, gibt es im Dialog nun einen Bereich *Configuration*. In diesem kann festgelegt werden, ob und wenn ja, wie ein einzelnes Attribut konfigurierbar ist. Die Möglichkeiten unterscheiden sich in Abhängigkeit vom Typ des Attributes.

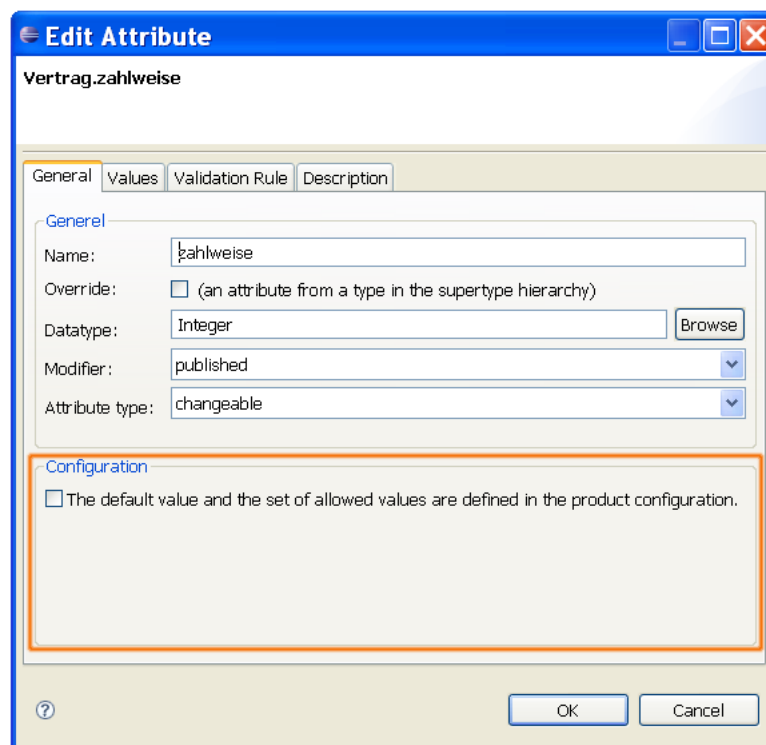


Abbildung 24: Dialog für ein Vertragsklassenattribut mit Konfigurationsmöglichkeit

Um die erlaubten Zahlweisen und den Vorgabewert für die Zahlweisen im Produkt definierten zu können, müssen Sie die entsprechende Checkbox anhängen. Nun schließen Sie den Dialog wieder.

Werfen wir nun einen Blick in den Sourcecode. Im published Interface der Produktgeneration gibt es jetzt jeweils eine Methode, um den Produktnamen, den Vorgabewert für die Zahlweise und die erlaubten Werte für die Zahlweise abzufragen.

```
public String getProduktname();
public Integer getVorgabewertZahlweise();
public EnumValueSet getAllowedValuesForZahlweise(String businessFunction);
```

Im published Interface des Vertrags gibt es jetzt Methoden, um auf das Produkt und die Produktgeneration auf der der Vertrag basiert, zuzugreifen.

```
public IProdukt getProdukt();
public void setProdukt(
    IProdukt produkt,
    boolean initPropertiesWithConfiguratedDefaults);
public IProduktGen getProduktGen();
```

Nun legen Sie noch das Vertragsattribut „wirksamAb“ mit Datentyp GregorianCalendar an. Dieses

Attribut ist nicht produktseitig konfigurierbar. Woher weiß nun die Vertragsklasse, dass Sie mit dem Attribut „wirksamAb“ die gültige Generation ermitteln muss? Hierzu hat Faktor-IPS in die Klasse Vertrag bereits folgende Methode generiert:

```
/**
 * {@inheritDoc}
 *
 * @generated
 */
public Calendar getEffectiveFromAsCalendar() {
    return null; // TODO Implementieren des Zugriffs auf das Wirksamkeitsdatum.
    // Damit diese Methode bei erneutem Generieren nicht neu ueberschrieben
    // wird, muss ein NOT hinter die Annotation @generated geschrieben werden!
}
```

Diese Methode wird von der Methode `getProduktGen()` aufgerufen, um den Stichtag zur Ermittlung der Produktgeneration zu erhalten<sup>15</sup>. Wir geben hier einfach das `wirksamAb` zurück (und schreiben ein NOT hinter die `@generated` Annotation, also

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

Die Implementierung der published Interfaces `IProdukt` und `IProduktGen` sorgen dafür, dass die Produktdaten zur Laufzeit zur Verfügung stehen. Die Details sprengen allerdings den Rahmen dieses Tutorials.

Die Implementierung der Methode `getEffectiveFromAsCalendar()` ist nur in der Klasse `Vertrag` zu implementieren. In abhängigen Klassen wie z.B. den Deckungen wird standardmäßig die `getEffectiveFromAsCalendar()` Methode der jeweiligen Vaterklasse aufgerufen.

### Erweiterung des hausratspezifischen Modells um Produktaspekte

Nun definieren wir die Konfigurationsmöglichkeiten des Hausratvertrags. Dazu legen Sie als erstes analog zur Klasse `Produkt` die Klasse `HausratProdukt` an, die den `HausratVertrag` konfiguriert. Dann markieren Sie die Attribute „wohnflaeche“ und „versSumme“ analog zur „zahlweise“ als konfigurierbar.

Nun überarbeiten wir die Berechnung des Vorschlags der Versicherungssumme. In Kapitel 4 hatten wir die Methode `getVorschlagVersSumme()` der Klasse `HausratVertrag` bisher wie folgende implementiert:

---

<sup>15</sup> Es handelt sich also um eine `TemplateMethod` (im Sinne der GoF-Patterns), die in der abstrakten Basisklasse `PolicyComponent` definiert ist.

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getVorschlagVersSumme() {
    // TODO: der multiplikator wird spaeter aus den produktdaten ermittelt.
    return Money.euro(650).multiply(wohnflaeche);
}
```

Nun wollen wir die Höhe des Multiplikators im Hausratprodukt konfigurieren können. Hierzu legen Sie zunächst an der Klasse HausratProdukt ein neues Attribute „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 HausratProduktGen 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:

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public Money getVorschlagVersSumme() {
    IHausratProduktGen gen = getHausratProduktGen();
    if (gen==null) {
        return Money.NULL;
    }
    return gen.getVorschlagVersSummeProQm().multiply(wohnflaeche);
}
```

Definieren wir nun noch die „Produktseite“ des Modells für die Deckungen. Beginnen wir mit der spartenübergreifenden Deckung. Die entsprechende Konfigurationsklasse nennen wir „Deckungstyp“. Der spartenübergreifende Deckungstyp bekommt eine Eigenschaft Bezeichnung. Analog wird für die Klasse HausratGrunddeckung ein HausratGrunddeckungstyp angelegt, allerdings ohne eine Eigenschaft Bezeichnung, diese wird von der Superklasse geerbt. Die Konfigurationsmöglichkeiten der Grunddeckung verschieben wir noch bis wir die Beitragsberechnung implementieren. Legen Sie die beiden Klassen und das Attribut nun an.

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:

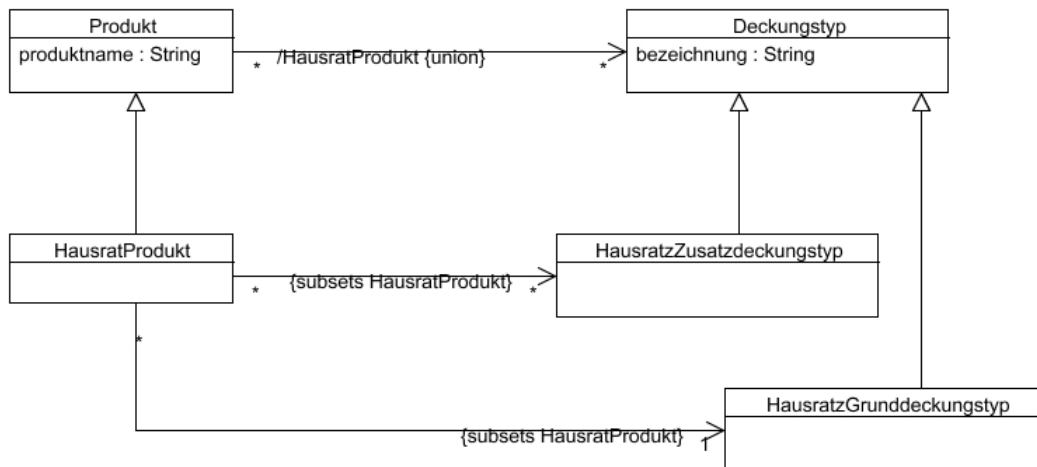


Abbildung 25: Modell der Produktkonfigurationsklassen

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. Wie auf der Vertragsseite des Modells ist die spartenübergreifende Beziehung eine abgeleitete Vereinigung, die von den spartenspezifischen Beziehungen realisiert wird.

Definieren wir diese Beziehungen also nun in Faktor-IPS. Öffnen Sie hierzu zunächst den Editor für die Klasse Produkt und legen im Bereich *Associations* durch klicken auf *New* eine neue Beziehung an. Es öffnet sich der folgende Dialog, in den Sie die Daten wie hier abgebildet eintragen.

**Edit Association**

Produkt.Deckungstyp

Properties Description

General

Target: base.Deckungstyp [Browse]

Type: Aggregation

Target role singular: Deckungstyp

Target role plural: Deckungstypen

Min Cardinality: 0

Max Cardinality: \*

Derived union

☒ This association is a derived union

☐ This association defines a subset of a derived union

Derived union:

OK Cancel

Abbildung 26: Dialog für Beziehungen zwischen Produktklassen



Schließen Sie den Dialog und speichern die Klasse. Da wir nur vom Produkt zum Deckungstyp navigieren ist die Anlage einer Rückwärtbeziehung nicht erforderlich<sup>16</sup>. Aus diesem Grund gibt es für die Anlage von Beziehungen auf Produktseite keinen Assistenten, sondern ausschließlich einen Dialog.

Wenn Sie einen Blick in das published Interface `IProduktGen` werfen, sehen Sie die Methode `getDeckungstypen()` zur Ermittlung der verwendeten Deckungstypen. Die (generierte) Implementierung dieser Methode erfolgt wie bei der Beziehung zwischen Vertrag und Deckung erst in den spartenspezifischen Klassen.

Analog definieren Sie jetzt noch die Beziehungen zwischen Hausratprodukt und HausratGrunddeckungstyp. Achten Sie darauf die maximale Kardinalität auf eins zu setzen und die Referenz auf die abgeleitete Vereinigungsbeziehung zu setzen. Den Zusatzdeckungstypen legen wir zum Schluß dieses Tutorials an.

---

<sup>16</sup> Natürlich können Sie auch programmatisch suchen, in welchen Produkten ein Deckungstyp enthalten ist. Die Rückwärtsbeziehung ist nur nicht im Modell unmittelbar navigierbar.

## Definition der Produkte

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

Als erstes richten Sie bitte noch ein neues Projekt mit dem Namen „Hausratprodukte“ und dem Sourceverzeichnis „produktdata“ ein (erst ein neues Java-Projekt erzeugen, dann „Add IpsNature“ im Package-Explorer aufrufen). Als Typ wählen Sie diesmal *Product Definition Project*, als Packagename „org.faktorips.tutorial.produktdata“ 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 RuntimeId aus dem Prefix gefolgt von dem (unqualifizierten) Namen zusammen<sup>17</sup>. Zur Laufzeit wird nicht der qualifizierte Name eines Bausteines zur Identifikation 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.

Die Verwaltung der Produktdaten in einem eigenen Projekt erfolgt wieder 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-Flexibel“ erstellen und freigeben, ohne dass das Modell geändert wird. Damit in dem neuen Projekt das Hausratmodell bekannt ist, fügen Sie in der „*ipsproject*“ Datei die folgende Zeile zum `IpsObjectPath` hinzu.

```
<Entry type="project" referencedIpsProject="Hausratmodell"/>
```

Für das Grundmodell brauchen Sie keinen Eintrag hinzufügen, da bereits das Hausratmodell auf das Grundmodell referenziert. Damit auch die Javaklassen in dem Projekt verfügbar sind, müssen Sie noch den Java-Classpath um die beiden Java-Projekte Grundmodell und Hausratmodell erweitern.

Öffnen Sie nun zunächst die Produktdefinitionsansicht über „Window→Open Perspective→Other“, in der Liste dann *Produktdefinition* auswählen<sup>18</sup>. Falls Sie noch Editoren geöffnet haben, schließen Sie diese jetzt, damit Sie die Sichtweise der Fachabteilung auf das System haben. Damit Sie im Problemsview ausschließlich die Marker von Faktor-IPS sehen (und nicht auch Java-Marker u.a.) müssen Sie im Problemsview den Faktor-IPS Filter ein- und alle anderen Filter (standardmäßig mindestens der Defaultfilter) ausschalten.

---

<sup>17</sup> In einer folgenden Version von Faktor-IPS wird für die Implementierung eigener Verfahren zur Vergabe der RuntimeId ein entsprechender Extension Point bereitgestellt werden.

<sup>18</sup> 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 Produktdefinitionsansicht verfügbar ist.

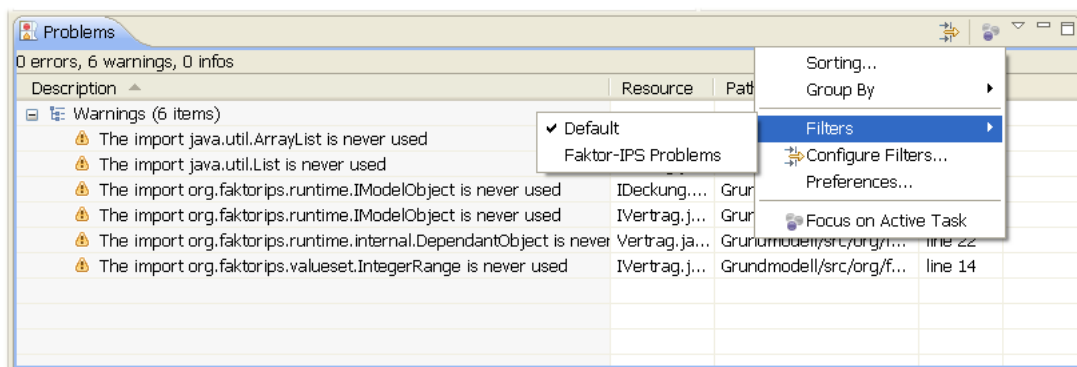




Abbildung 27: Ausschalten der Problemfilter bis auf den Faktor-IPS Filter

Zunächst legen wir zwei Packages an, einen für die Produkte und einen für die Deckungen. Dies geschieht wie in der Java Perspektive entweder über das Kontextmenü oder die Toolbar.

Sie können in dem Projekt im übrigen auch beliebige andere Verzeichnisse anlegen, zum Beispiel ein doc-Verzeichnis, um Dokumente zu den Produkten zu verwalten.

Unsere Produkte sollen ab dem nächsten Quartalsbeginn verfügbar sein. Bevor wir nun also mit dem Anlegen der Produkte beginnen, setzen wir noch das Wirksamkeitsdatum, mit dem wir arbeiten wollen. Klicken Sie hierzu in der Toolbar auf , geben den nächsten Quartalsbeginn ein und drücken auf Ok. Alle Änderungen werden von nun an mit diesem Wirksamkeitsdatum durchgeführt.

Als erstes legen wir jetzt das Produkt HR-Optimal an. Markieren Sie dazu das gerade angelegte Package „Produkte“ und klicken dann in der Toolbar auf . Es öffnet sich der Wizard zum Erzeugen eines neuen Produktbausteins.

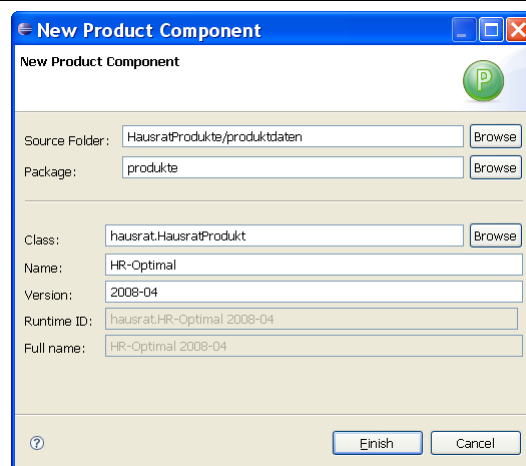


Abbildung 28: Anlegen eines neuen Produkts

Die ersten beiden Felder sind bereits vorbelegt und den Fokus hat das Feld zum Auswählen der

Produktklasse, auf dem der neue Baustein basieren soll. Wählen Sie „hausrat.HausratProdukt“ aus. Sie können hier wie immer in Eclipse und die Vervollständigung mit *Strg-Space* aufrufen. Falls sie keine Produktklasse finden, fehlt die Referenz auf das Hausratmodellprojekt in der „ipsproject“-Datei (s.o.). Danach geben Sie den Namen des Bausteines ein. Die Versionsnummer des Bausteins wird anhand des Wirksamkeitsdatum vorbelegt. Das Format der Versionsnummer kann in der „ipsproject“-Datei definiert werden. Standardmäßig ist es „JJJJ-MM“ mit einem zusätzlichen optionalen Postfix, also z. B. „2006-12b“. Ebenfalls vorbelegt wird die Runtime ID anhand des im Projekt definierten Prefixes und dem unqualifizierten Namen.

Standardmäßig ist diese Vorbelegung nicht änderbar. Dies kann aber in den Einstellungen (Window→Preferences...→Can modify runtime id) geändert werden.

Wenn Sie nun **Finish** drücken wird der Produktbaustein im Dateisystem angelegt und der Editor geöffnet.

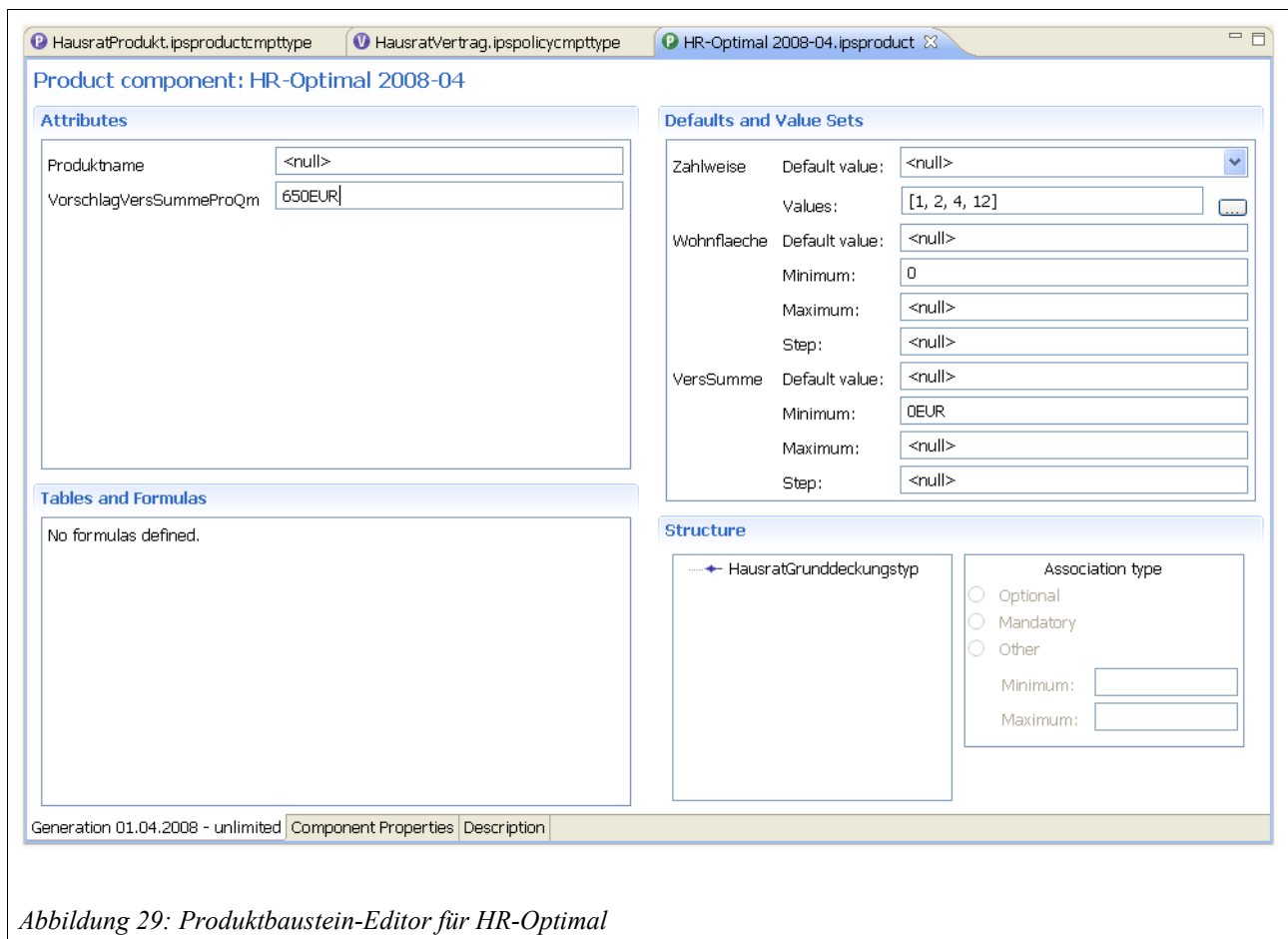


Abbildung 29: Produktbaustein-Editor für HR-Optimal

Die erste Seite des Editors zeigt die in Bearbeitung befindliche Generation des Produktbausteins. Nach der Anlage hat ein Baustein immer eine Generation, die ab dem eingestellten Wirksamkeitsdatum gültig ist. Die Seite ist in vier Bereich eingeteilt:

**Attributes**

Enthält die Eigenschaften der Generation. Hier ist jedes in der Produktklasse definierte Attribute aufgelistet.

**Tables and Formulas**

Enthält die Berechnungsvorschriften und Referenzen auf Tabellen. Hierzu mehr bei der Implementierung der Beitragsberechnung.

**Defaults and Value Sets**

Enthält die Vorbelegungswerte und Wertebereiche für die Vertragseigenschaften.

**Structure**

Enthält die verwendeten anderen Produktbausteine.

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

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

Nun legen wir 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 Explorer erledigen. Öffnen Sie das Produkt „HR-Optimal“. Ziehen Sie die „HRD-Grunddeckung-Optimal“ aus dem Produktdefinitions-Explorer auf den Knoten „HausratGrunddeckungstyp“ im Bereich *Structure*. Markieren Sie nun die neue Beziehung und wählen als Art der Beziehung obligatorisch (Englisch: mandatory).

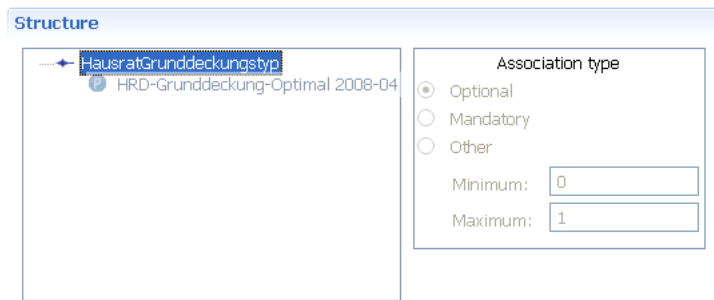


Abbildung 30: Zuordnung von Deckungstypen zu einem Produkt

Nun legen wir noch das Produkt HR-Kompakt inklusive der Grunddeckung HRD-Kompakt an. Dies können sie analog zum Produkt HR-Optimal machen. Alternativ können 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 ...*. Auf der ersten Seite geben Sie als *Search Pattern* Optimal und als *Replace Pattern* Kompakt ein, klicken *Next* und dann *Finish*. Faktor-IPS legt das HR-Kompakt 2008-04 und HRD-Grunddeckung 2008-04 neu an. In der Praxis wird der Kopierassistent i.d.R. zum Anlegen neuer Versionen verwendet, der Aufruf erfolgt im Kontextmenü über *New→Create New Version ...*

Öffnen Sie nun das neue Produkt und geben seine Daten ein.

<b>Konfigurationsmöglichkeit</b>	<b>HR-Kompakt</b>
Produktname	Hausrat Kompakt
Vorschlag Versicherungssumme pro qm Wohnfläche	600EUR
Vorgabewert Zahlweise	jährlich
Erlaubte Zahlweisen	halbjährlich, jährlich
Vorgabewert Wohnflaeche	<null>
Erlaubte Wohnflaeche	0-1000 qm
Vorgabewert Versicherungssumme	<null>
Versicherungssumme	10000EUR – 2000000EUR

Damit ist die Definition der beiden Produkte zunächst abgeschlossen. Im Produktdefinitions-Explorer sollten Sie wie nun folgt angezeigt werden.

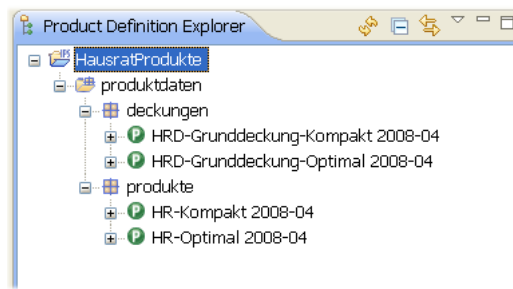


Abbildung 31: Darstellung der Produkte im Produktdefinitions-Explorer

Neben dem Produktexplorer stehen Ihnen zwei weitere Werkzeuge zur Analyse der Produktdefinition zur Verfügung. Die Struktur eines Produktes können Sie sich über „Show Structure“ im Kontextmenü anzeigen lassen. Die unterschiedlichen Verwendung eines Bausteins über „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“, 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 „derived“ kopiert. Damit sind diese Dateien im Buildpath 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“<sup>19</sup>.

Der folgende Screenshot zeigt den Inhalt des Sourcefolders „derived“ im Package-Explorer.

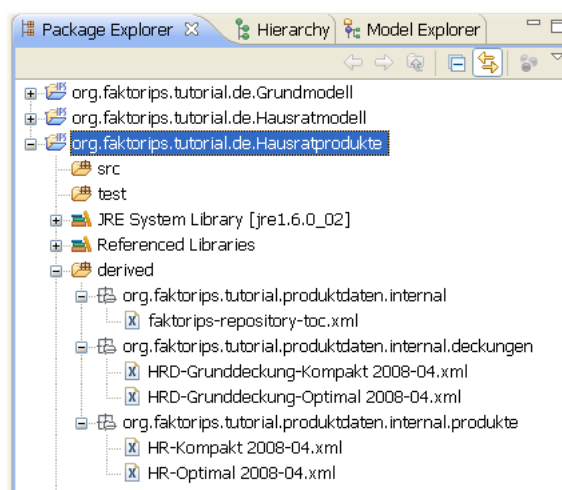


Abbildung 32: Inhalt des Sourcefolders „derived“

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

<sup>19</sup> Die Namen lassen sich in der „ipsproject“ Datei im Abschnitt `IpsObjectPath` konfigurieren.



Parameter übergibt man die RuntimeId des Bausteins. Da das Interface `IRuntimeRepository` unabhängig vom konkreten Modell (in unserem Falls also dem Hausratmodell) ist, muss man das Ergebnis auf 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 *Buildpath*→*New Source Folder...* aufrufen.

Danach markieren Sie den neuen Sourcefolder und legen einen JUnit Testfall an, indem Sie in der Toolbar auf  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 geraten wird, ignorieren wir in dem Tutorial. Im unteren Teil des Dialogs gibt es einen Link, mit dem Sie die benötigte JUnit Library zum Java Buildpath des Projektes hinzufügen können. Der nächste Kasten enthält den Sourcecode der Testfallklasse.

```
public class TutorialTest extends TestCase {

    private IRuntimeRepository repository;
    private IHausratProdukt kompaktProdukt;
    private IHausratProduktGen kompaktGen;

    public void setUp() {
        // 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 2008-04");

        // Juengste Productgeneration holen (wir wissen es gibt nur eine).
        IProductComponentGeneration pcGen = pc.getLatestProductComponentGeneration();

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

    public void testProduktDatenLesen() {
        System.out.println("Produktname: " + kompaktGen.getProduktname());
        System.out.println("Vorschlag Vs pro lqm: " + kompaktGen.getVorschlagVersSummeProQm());
        System.out.println("Default Zahlweise : " + kompaktGen.getDefaultValueZahlweise());
        System.out.println("Erlaubte Zahlweisen: "
            + kompaktGen.getAllowedValuesForZahlweise(null));
        System.out.println("Default Vs: " + kompaktGen.getDefaultValueVersSumme());
        System.out.println("Bereich Vs: " + kompaktGen.getRangeForVersSumme(null));
        System.out.println("Default Wohnflaeche: " + kompaktGen.getDefaultValueWohnflaeche());
        System.out.println("Bereich Wohnflaeche: " + kompaktGen.getRangeForWohnflaeche(null));
    }
}
```

Anstatt die Korrektheit der mit Assert-Statements zu testen, geben wir sie hier mit `println` auf der Konsole aus. Wenn Sie den Test nun ausführen, sollte er folgendes ausgeben:

```
Produktname: Hausrat Kompakt
Vorschlag Vs pro lqm: 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
```

## 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.


### Tarifzonentabelle

Aufgrund der in Deutschland regional unterschiedlichen Schadenswahrscheinlichkeit durch Einbruchdiebstahl unterscheiden Versicherungsunternehmen in der Hausratversicherung üblicherweise zwischen unterschiedlichen Tarifzonen. Dazu gibt es in der Regel eine Tabelle, mit der einem Postleitzahlenbereich eine Tarifzone zugeordnet ist, also zum Beispiel

<i>Plz-Von</i>	<i>Plz-bis</i>	<i>Tarifzone</i>
17235	17237	II
45525	45549	III
59174	59199	IV
47051	47279	V
63065	63075	VI
...	...	...

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<sup>20</sup>.

Legen wir zunächst die Tabellenstruktur für die Ermittlung der Tarifzonen an. Hierzu wechseln Sie zunächst zurück in die Java-Perspektive. Im Projekt Hausratmodell 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 voreingestellten 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. Die Bedienung dürfte selbsterklärend sein.

Interessanter 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 darstellt. Legen Sie jetzt einen neuen Bereich an. Da die Tabelle Von- und Bis-Spalten enthält, wählen Sie als Typ „Two Column Range“. Als Parameternamen in der Zugriffsfunktion geben Sie jetzt „plz“ ein und ordnen noch die beiden Spalten plzVon und plzBis zu.

Jetzt legen Sie noch einen neuen UniqueKey an. Dem UniqueKey ordnen Sie jetzt nicht die einzelnen Spalten plzVon und plzBis zu, sondern den Bereich und speichern danach die

<sup>20</sup> Dies entspricht dem Konzept von Tabellenpartitionen in relationalen Datenbankmanagementsystemen.

## Strukturbeschreibung.

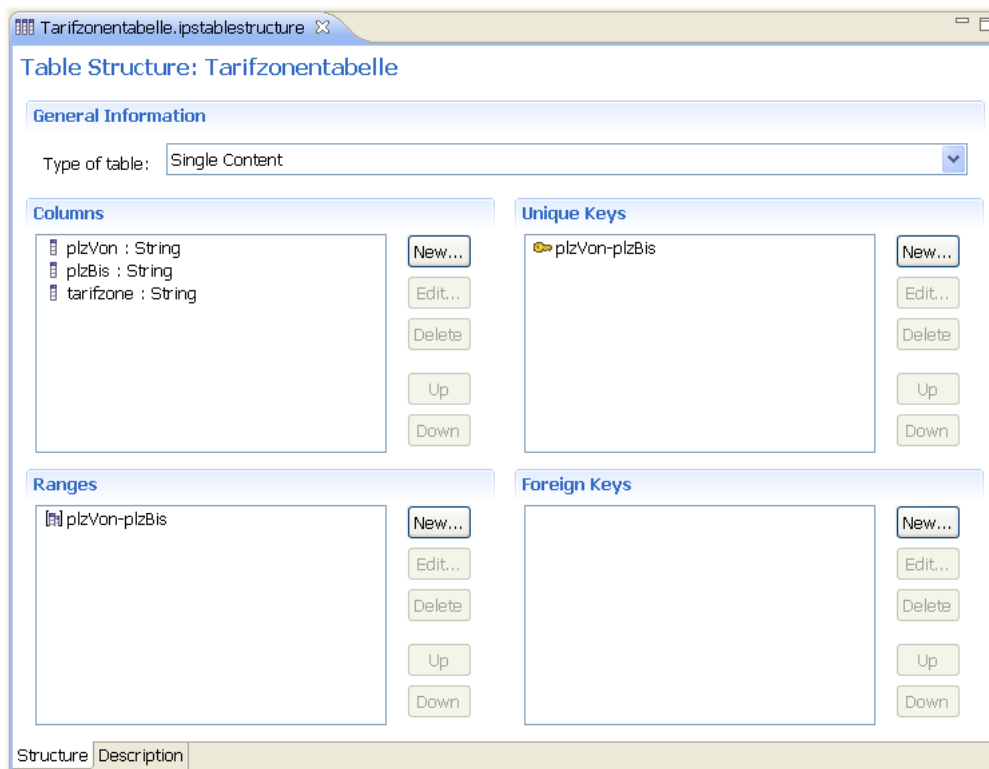


Abbildung 33: Anlegen einer Tabellenstruktur

Faktor-IPS hat nun für die Tabellenstruktur zwei neue Klassen im Package `org.faktorips.tutorial.modell.internal.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 im Kapitel 3 wie folgt implementiert:

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

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

```
public String getTarifzone() {  
    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();  
}
```

Erläuterung Bedarf an dieser Stelle noch, 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 zurückliefert. 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<sup>21</sup>.

Jetzt legen wir nun noch den Tabelleninhalt an. Die Zuordnung der Postleitzahlen zu den Tarifzonen soll von der Fachabteilung gepflegt werden. Zur Strukturierung fügen Sie noch ein neues Package „tabellen“ unterhalb dem Package „hausrat“ 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:

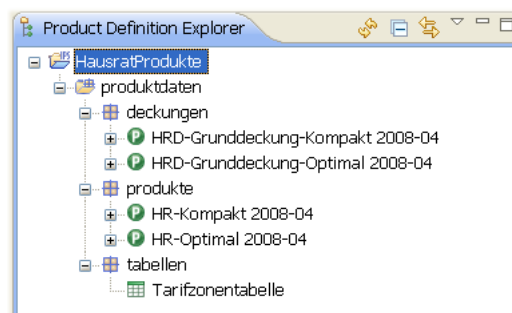


Abbildung 34: Projektstruktur der Produktdefinition

Zum Schluss testen wir noch die Ermittlung der Tarifzone. Hierzu erweitern wir unseren JUnit Test „TutorialTest“ um die folgenden Testmethode:

<sup>21</sup> Das Übergeben des `RuntimeRepository`s in die Methode `getInstance()` hat den Vorteil, dass das konkrete Repository in Testfällen leicht ausgetauscht werden kann.

```
public void testGetTarifzone() {
    // Erzeugen eines Hausratvertrags mit der Factorymethode des Produktes
    IHausratVertrag vertrag = kompaktProdukt.createHausratVertrag();

    // Wirksamkeitsdatum des Vertrages setzen, damit die Produktgeneration gefunden wird!
    // Dies muss nach dem Gueltigkeitsbeginn der Generation liegen!
    vertrag.setWirksamAb(new GregorianCalendar(2010, 0, 1));

    vertrag.setPlz("45525");
    assertEquals("III", vertrag.getTarifzone());
}
```

### Beitragstabelle

Der Beitragssatz für die Grunddeckung der Hausratversicherung soll anhand der Tarifzone aus einer Tariftabelle ermittelt. Legen Sie hierfür eine Tabellenstruktur „TariftabelleHausrat“ mit den beiden Spalten Tarifzone (String) und Beitragssatz (Decimal) an. Definieren Sie einen UnquieKey auf die Spalte Tarifzone. Als Tabellentyp wählen Sie diesmal „Multiple Contents“ aus, da wir für jedes Produkt einen eigenen Tabelleninhalt anlegen wollen. In der Praxis verwendet man es üblicherweise dazu die Beitragssätze pro Version in unterschiedlichen Tabelleninhalte abzulegen. Dieses Vorgehen hat den Vorteil, dass die Tabellen in der Regel eine überschaubare Größe haben und unabhängig voneinander bearbeitet und versioniert werden können. Wird eine neue Produktversion eingeführt, wird einfach ein neuer Tabelleninhalt hinzugefügt. Dies kann dann auch einfach durch den Import von Excel-Dateien erfolgen. Vorteilhaft ist die Trennung der Tabelleninhalte auch zur Laufzeit. Da auf die Daten älterer Tarifversionen seltener zugegriffen wird, kann hierfür eine andere Caching-Strategie verwendet werden kann.

Erzeugen Sie nun für die Produkte HR-Optimal und HR-Kompakt (bzw. genauer für deren Grunddeckungstypen) jeweils einen Tabelleninhalt mit dem Namen „Tariftabelle Optimal 2008-04“ und „Tariftabelle Kompakt 2008-04“ an<sup>22</sup>. Ein Vorschlag für die Tabelleninhalte finden Sie im Anhang. Das folgende UML-Diagramm zeigt diesen fachlichen Zusammenhang:

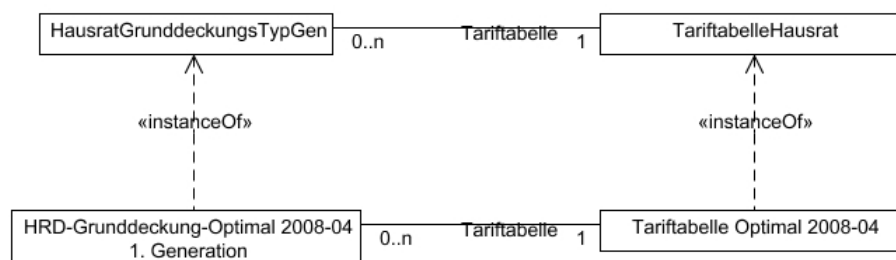


Abbildung 35: Zusammenhang Produktbausteine & Tabellen

Eine Generation eines HausratGrunddeckungstyp verwendet eine TariftabelleHausrat in der Rolle Tariftabelle. Die erste Generation des Grunddeckungstyps „HRD-Grunddeckung-Optimal 2008-04“ verwendet den Tabelleninhalt „Tariftabelle Optimal 2008-04“.

Der Zusammenhang zwischen Tabellen und Produkten kann in Faktor-IPS explizit definiert

<sup>22</sup> Die Endung „2008-04“ sollten Sie dabei entsprechend des von ihnen verwendeten Wirksamkeitsdatums anpassen.

werden. Hierzu wechseln Sie in den Editor für die Klasse `HausratGrunddeckungstyp`. Auf der zweiten Seite im Editor<sup>23</sup> im Abschnitt *Table Usages* sind die verwendeten Tabellen aufgelistete. Klicken Sie auf den *New* Button neben dem Abschnitt, um eine neue Tabellenverwendung zu definieren.

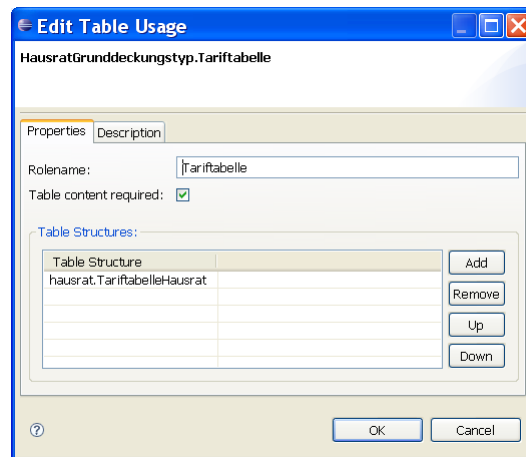


Abbildung 36: Modellierung der Verwendung von Tabellen

In dem Dialog geben Sie als Rollenname „Tariftabelle“ ein und ordnen die Tabellenstruktur „TariftabelleHausrat“ zu. 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 2008-01“. Den Dialog, der Sie darauf hinweist, dass die Tariftabelle noch nicht zugeordnet ist, bestätigen Sie mit *Fix*. In dem Abschnitt *Calculation Formulas and Tables* können Sie nun die Tariftabelle für HR-Kompakt zuordnen und dann speichern. Analog verfahren Sie für HR-Optimal.

Zum Schluss des Kapitels werfen wir noch einen Blick auf den generierten Sourcecode. In der Klasse `HausratGrunddeckungstypGen` 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.

<sup>23</sup> Vorausgesetzt, sie haben in den Preferences eingestellt, dass die Editoren 2 Abschnitte pro Seite verwenden sollen.

## Implementieren der Beitragsberechnung

In diesem Kapitel werden wir nun die Beitragsberechnung für unsere Hausratprodukte implementieren. Insbesondere werden wir dabei den spartenübergreifenden Teil in den spartenübergreifenden Klassen implementieren und die Funktionalität dann in der Beitragsberechnung für Hausrat verwenden.

### Spartenübergreifende Berechnung

Beginnen wir mit dem spartenübergreifenden Modell gemäß der folgenden Abbildung.

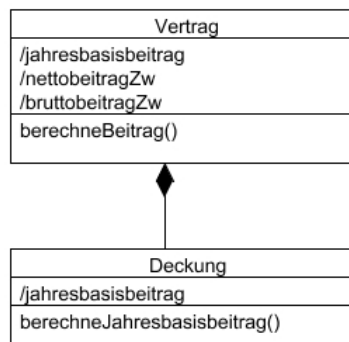


Abbildung 37: Berücksichtigung der Beitragsberechnung im spartenübergreifenden Modell

Jede Deckung hat einen Jahresbasisbeitrag. Der Jahresbasisbeitrag ist der pro Jahr zu zahlende Nettobeitrag (also ohne Versicherungssteuer) ohne Berücksichtigung eines Zuschlags für eine Ratenzahlung. Der Jahresbasisbeitrag des Vertrags ist die Summe über die Jahresbasisbeiträge seiner Deckungen. Der NettobeitragZw des Vertrags ist der vom Beitragszahler pro Zahlungsvorgang zu zahlende Nettobeitrag. Er ergibt sich aus dem Jahresbasisbeitrag durch Zuschlag eines Ratenzahlungszuschlages bei nicht jährlicher Zahlung und Division durch die Anzahl der Zahlungen, also zum Beispiel zwölf bei monatlicher Zahlung. Der BruttobeitragZw ist der vom Beitragszahler zu zahlende Bruttobeitrag pro Zahlung, also inkl. der Versicherungssteuer. Er ergibt sich aus dem NettobeitragZw durch Multiplikation mit  $1 + \text{Versicherungssteuersatz}$ . Vereinfachend implementieren wir in dem Tutorial, dass der Ratenzahlungszuschlag immer 3% und die Versicherungssteuer immer 19% beträgt.

Legen Sie nun die neuen Attribute an den spartenübergreifenden Klassen Vertrag und Deckung an. Alle Attribute sind abgeleitet (cached) und vom Datentyp Money. Da es sich um gecachte abgeleitete Attribute handelt, generiert Faktor-IPS eine Membervariable und eine Gettermethode. Die Berechnung aller Beitragsattribute erfolgt durch die Methode `berechneBeitrag()` der Klasse Vertrag. Die Methode berechnet dabei alle Beitragsattribute des Vertrags und auch den Jahresbasisbeitrag der Deckungen. Hierzu verwendet sie die natürlich die Methode `berechneJahresbasisbeitrag()` der Deckung.

Legen Sie nun diese beiden Methoden an. Dies geschieht auf der zweiten Seite im Editor für die Klasse Vertrag. Die folgende Abbildung zeigt den Dialog zum Bearbeiten einer Methodensignatur.

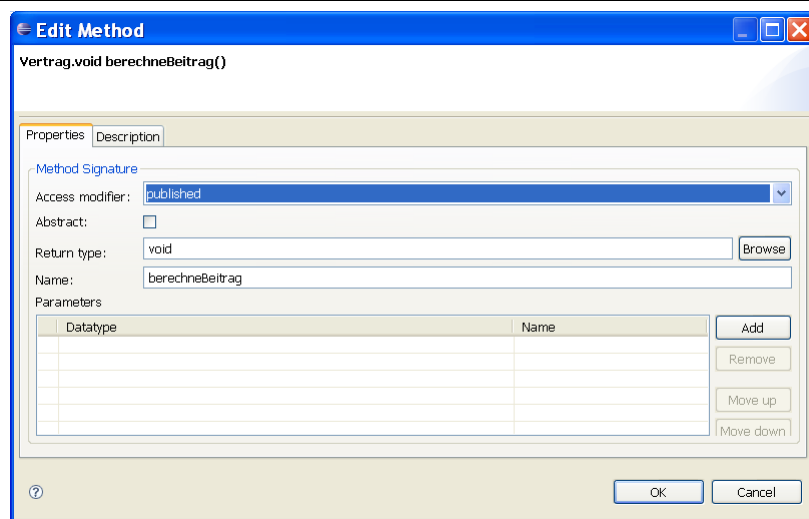


Abbildung 38: Dialog zum Bearbeiten einer Methodensignatur

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. Wir gehen i.d.R. einen Mittelweg indem wir Methoden des published Interface zur besseren Dokumentation in Faktor-IPS erfassen und modellinterne Methoden ausschließlich im Sourcecode definieren.

Der folgende Sourcecodeausschnitt zeigt die Implementierung der spartenübergreifenden Beitragsberechnung in der Klasse Vertrag. 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.

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

private void berechneJahresbasisbeitrag() {
    jahresbasisbeitrag = Money.euro(0, 0);
    IDeckung[] deckungen = getDeckungen();
    for (int i=0; i<deckungen.length; i++) {
        deckungen[i].berechneJahresbasisbeitrag();
        jahresbasisbeitrag =
            jahresbasisbeitrag.add(deckungen[i].getJahresbasisbeitrag());
    }
}
```



```
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,
                                                    BigDecimal.ROUND_HALF_UP);
    }
    nettobeitragZw = nettobeitragZw.divide(zahlweise.intValue(),
                                           BigDecimal.ROUND_HALF_UP);
}
```

Die Berechnung des Jahresbasisbeitrags der Deckung kann natürlich nicht spartenübergreifend realisiert werden, sondern muss in den Subklassen von Deckung implementiert werden. Da wir in diesen aber keinen Zugriff auf die private Membervariable `jahresbasisbeitrag` haben, implementieren wir die Zuweisung in der Methode `berechneJahresbasisbeitrag()` der Klasse Deckung und delegieren die eigentliche Berechnung an eine abstrakte Methode `berechneJahresbasisbeitragInternal()`. Der folgende Kasten zeigt den entsprechenden Sourcecode der Klasse Deckung.

```
/**
 * {@inheritDoc}
 *
 * @generated NOT
 */
public void berechneJahresbasisbeitrag() {
    jahresbasisbeitrag = berechneJahresbasisbeitragInternal();
}

/**
 * Methode muss in Subklassen implementiert werden und den Jahresbasisbeitrag
 * der Deckung zurueckliefern.
 */
public abstract Money berechneJahresbasisbeitragInternal();
```

### Beitragsberechnung für Hausrat

Für die Hausratversicherung muss nun die Beitragsberechnung auf Deckungsebene implementiert werden. Die Summation über die einzelnen Deckungen und die Berechnung von Versicherungssteuer und Bruttobeitrag ist durch die Ableitung von den spartenübergreifenden Klasse bereits umgesetzt.

Fachlich berechnet sich der Jahresbasisbeitrag für die Grunddeckung wie folgt:

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

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.

Wenn Sie nun die Javaklasse im Editor öffnen werden Sie feststellen, dass die Klasse aktuell als fehlerhaft markiert ist. Das liegt daran, dass die gerade eingeführte Methode `berechneJahresbasisbeitragInternal()` noch nicht implementiert ist. Die Methodensignature erzeugen Sie am schnellsten über die Eclipsefunktion *Source*→*Override/Implement Methods...*

Der folgende Abschnitt zeigt den vollständigen Sourcecode der Methode:

```
public Money berechneJahresbasisbeitragInternal() {
    HausratGrunddeckungstypGen gen =
        (HausratGrunddeckungstypGen) getHausratGrunddeckungstypGen();
    if (gen==null) {
        return Money.NULL;
    }
    TariftabelleHausrat tab = gen.getTariftabelle();
    TariftabelleHausratRow row=tab.findRow(getHausratVertrag().getTarifzone());
    if (row==null) {
        return Money.NULL;
    }
    Money vs = getHausratVertrag().getVersSumme();
    Decimal beitragsatz = row.getBeitragsatz();
    return vs.divide(1000, BigDecimal.ROUND_HALF_UP).multiply(beitragsatz,
        BigDecimal.ROUND_HALF_UP);
}
```

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

```
public void testBerechneBeitrag() {
    // Erzeugen eines hausratvertrags mit der Factorymethode des Produktes
    IHausratVertrag vertrag = kompaktProdukt.createHausratVertrag();

    // Wirksamkeitsdatum des Vertrages setzen, damit die Produktgeneration gefunden wird!
    // Dies muss nach dem Gueltigkeitsbeginn der Generation liegen!
    vertrag.setWirksamAb(new GregorianCalendar(2010, 0, 1));

    // Vertragsattribute setzen
    vertrag.setPlz("45525"); // => tarifzone 3
    vertrag.setVersSumme(Money.euro(60000));
    vertrag.setZahlweise(new Integer(2)); // halbjaehrlich

    // Grunddeckungstyp holen, der dem Produkt in der Generation zugeordnet ist.
    IHausratGrunddeckungstyp deckungstyp = kompaktGen.getHausratGrunddeckungstyp();

    // Grunddeckung erzeugen und zum Vertrag hinzufuegen
    IHausratGrunddeckung deckung = vertrag.newHausratGrunddeckung(deckungstyp);

    // Beitrag berechnen und Ergebniss pruefen
    vertrag.berechneBeitrag();

    // tarifzone 3 => beitragsatz = 1.21
    // jahresbasisbeitrag = versicherungsumme / 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());
}
```

## Flexibilität für die Fachabteilung am Beispiel von Zusatzdeckungen

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><b>Fahrraddiebstahl</b></i>	<i><b>Überspannung</b></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	10Euro + 3% der Versicherungssumme der Überspannungsdeckung

Solche Zusatzdeckungen haben also 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:

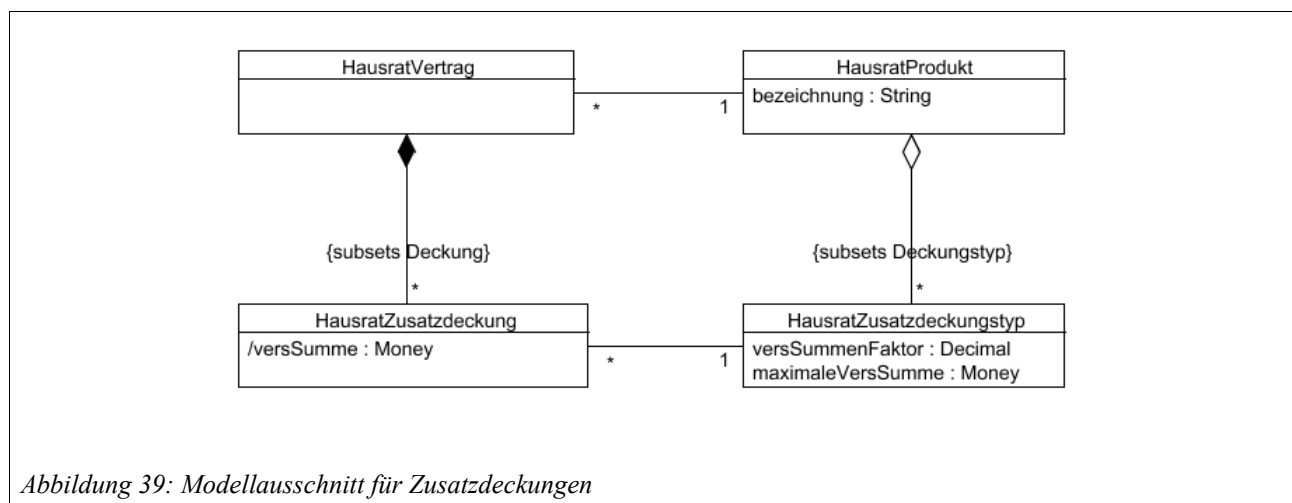


Abbildung 39: Modellausschnitt für Zusatzdeckungen

Ein HausratVertrag kann beliebig viele solcher Zusatzdeckungen enthalten. Die Konfigurationsklasse zur HausratZusatzdeckung nennen wir HausratZusatzdeckungstyp. Diese hat die zwei 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. Die Klasse HausratZusatzdeckung leitet natürlich von der spartenübergreifenden Klasse Deckung ab und HausratZusatzdeckungstyp von Deckungstyp, die spartenübergreifenden Klassen sind der Übersichtlichkeit halber nicht im Diagramm abgebildet.

Unsere beiden Beispieldeckungen sind Instanzen der Klasse HausratZusatzdeckungstyp. Dies zeigt

das folgende Diagramm.

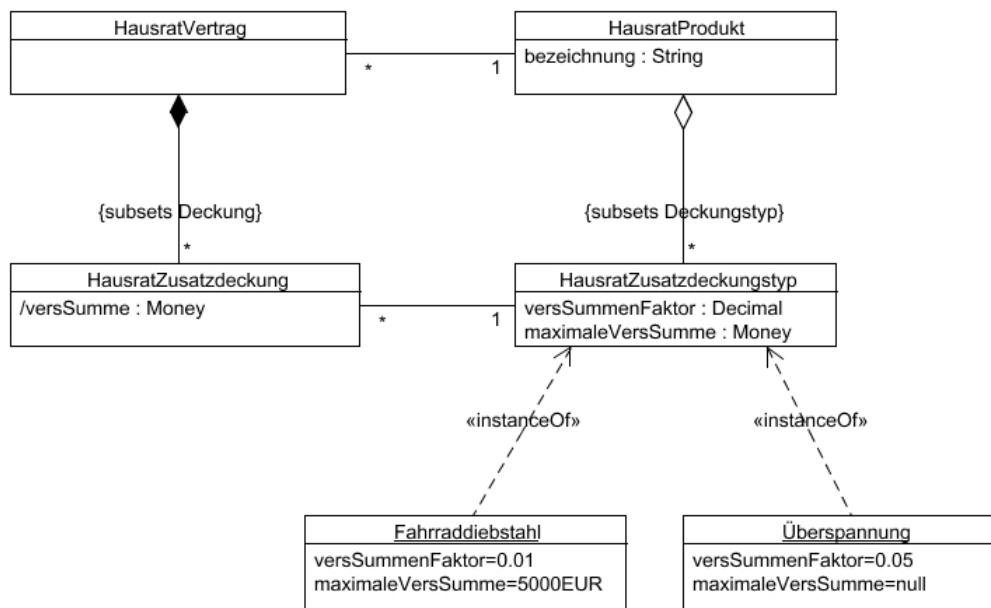


Abbildung 40: Modellausschnitt für Zusatzdeckungen mit Instanzen

Genau genommen sind die Produktbausteine „Fahrraddiebstahl“ und „Überspannung“ natürlich Instanzen der Generationsklassen ProduktGen und HausratZusatzdeckungstypGen, dieses Detail ist hier allerdings der Einfachheit halber weggelassen.

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.

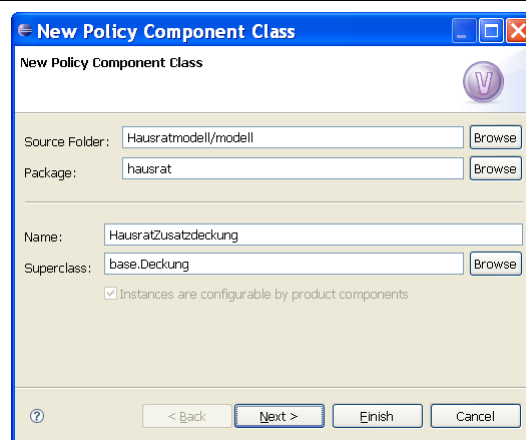


Abbildung 41: Assistenz zum Anlegen der Vertragsklasse HausratZusatzdeckung

Auf der zweiten Seite geben sie noch den Klassennamen HausratZusatzdeckungstyp ein und klicken *Finish*. Faktor-IPS legt nun beide Klasse 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.

Nach dem die Beziehungen definiert sind, legen Sie an der Klasse HausratZusatzdeckung das abgeleitete Attribut versSumme an und an der Klasse HausratZusatzdeckungstyp die beiden Attribute versSummenFaktor und maximaleVersSumme. Nachdem dies geschehen ist, können wir auch direkt die Ermittlung der Versicherungssumme implementieren. Hierzu codieren wir in der Klasse HausratZusatzdeckung die Methode getVersSumme() wie folgt:

```
public Money getVersSumme() {
    IHausratZusatzdeckungstypGen gen = getHausratZusatzdeckungstypGen();
    if (gen==null) {
        return Money.NULL;
    }
    Decimal faktor = gen.getVersSummenFaktor();
    Money vsVertrag = getHausratVertrag().getVersSumme();
    Money vs = vsVertrag.multiply(faktor, BigDecimal.ROUND_HALF_UP);
    if (vs.isNull()) {
        return vs;
    }
    Money maxVs = gen.getMaximaleVersSumme();
    if (vs.greaterThan(maxVs)) {
        return maxVs;
    }
    return vs;
}
```

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 2008-04“ und „HRD-Überspannung 2008-04“ basierend auf der Klasse HausratZusatzdeckungstyp an und geben im Editor deren Eigenschaften ein.

	HRD-Fahrraddiebstahl 2008-04	HRD-Überspannung 2008-04
Bezeichnung	Fahrraddiebstahl	Überspannungsschutz
VersSummenFaktor	0.01	0.05
MaximaleVersSumme	3000EUR	<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“ abgeschlossen werden, sollen die Deckungen immer enthalten sein, beim Produkt „HR-Kompakt“

sollen Sie optional dazu gewählt werden können. Sie können dies über die Art der Beziehung im Bausteineditor einstellen.

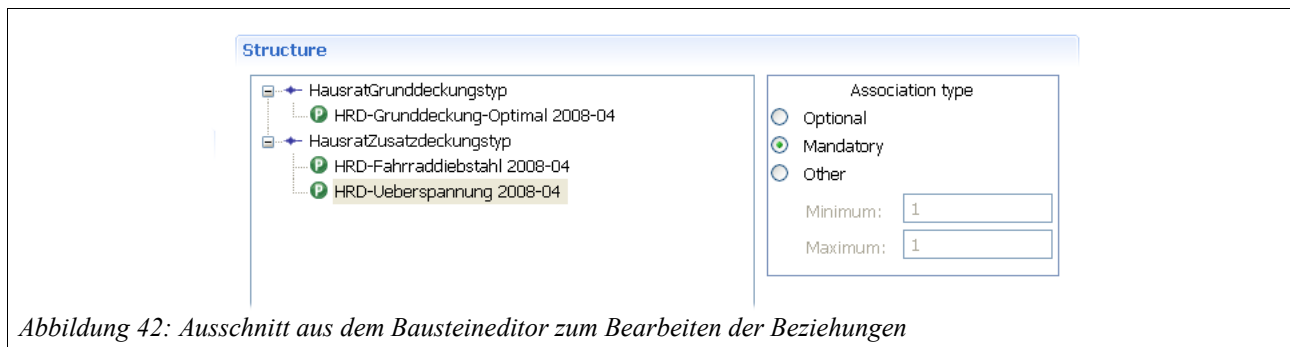


Abbildung 42: Ausschnitt aus dem Bausteineditor zum Bearbeiten der Beziehungen

### 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<sup>24</sup>. 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. Der Syntax für die Navigation durch den Objektgraphen kann schnell komplex werden. Wie sieht zum Beispiel ein für die Fachabteilung leicht verständlicher Syntax zur Ermittlung der Deckung mit der höchsten Versicherungssumme aus?
2. Aktualität von abgeleiteten Attributen

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 beliebigen Navigation durch den Objektgraphen muss man für alle erreichbaren Objekte sicherstellen, dass die abgeleiteten Attribute korrekte 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 Zusatzdeckungstyp die Formelsignatur mit den Parametern definiert werden.

<sup>24</sup> In der Hausratversicherung neben der Tarifzone zum Beispiel die Art des Hauses (Ein-/Mehrfamilienhaus) oder die Bauweise.

Öffnen Sie nun den Editor für die Klasse `HausratZusatzdeckungstyp`. Klicken Sie auf der zweiten Seite<sup>25</sup> im Abschnitt *Methods and Formula Signatures* auf den *New* Button, um eine Formelsignatur anzulegen und tragen Sie die Daten wie im folgenden Screenshot zu sehen ein.

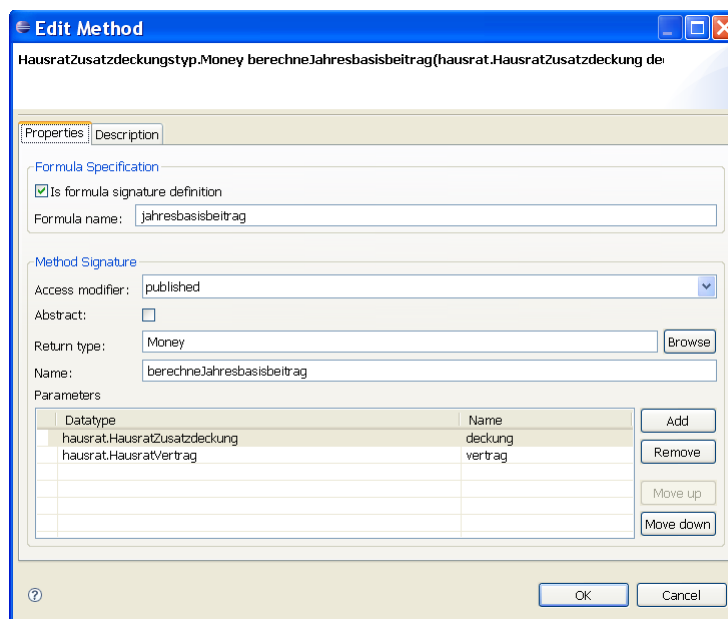


Abbildung 43: Dialog zur Definition einer Formelsignature

Schließen Sie den Dialog und speichern. Die Java-Klasse `HausratZusatzdeckungstypGen` ist jetzt abstrakt und hat eine abstrakte Methode `berechneJahresbasisbeitrag(...)` zur Berechnung des Basisbeitrags. Da unterschiedliche Produktbausteine der gleichen Modellklasse (hier Fahrraddiebstahl und die Überspannungsdeckung) unterschiedliche Formeln haben, kann der Sourcecode hierfür nicht in die Basisklasse generiert werden. Faktor-IPS generiert für jeden Produktbaustein, der eine Formel enthält, eine eigene Subklasse<sup>26</sup>. Die abstrakte Methode wird dort implementiert, indem die Formel durch einen Formelcompiler in Java Sourcecode übersetzt wird.

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 gibt. Bestätigen Sie mit *Fix*, dass diese hinzugefügt werden soll. In dem Abschnitt *Tables & Formulas* 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:

<sup>25</sup> Vorausgesetzt, sie haben in den Preferences eingestellt, dass die Editoren 2 Abschnitte pro Seite verwenden sollen.

<sup>26</sup> Genau genommen wird für jede Generation eine Klasse generiert, da jede Generation eine andere Formel haben kann.



Datatype	Name
hausrat.HausratZusatzdeckung	deckung
hausrat.HausratVertrag	vertrag

deckung.versSumme \* 0.1

Parameter	Value
deckung.versSumme	<null>

Result:

Abbildung 44: Anlegen einer Formel

Der Beitrag für die Fahrraddiebstahlversicherung soll 10% der Versicherungssumme der Zusatzdeckung betragen. Löschen Sie „<null>“ im Formeltext und drücken *Ctrl-Space*. 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.

Sie können die Formel in dem Dialog auch testen, indem Sie in dem Bereich *Formula Test* einen Wert für die Versicherungssumme eingeben und *Calculate* drücken.

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 die beiden Klassen Java-Sourcefolder „derived“ im Package `org.faktorips.tutorial.modell.internal.deckungen`<sup>27</sup>. Der folgende Sourcecode enthält die generierte Methode `berechneJahresbasisbeitrag(...)` für die Fahrraddiebstahldeckung.

<sup>27</sup> 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 IHausratZusatzdeckung deckung,  
    final IHausratVertrag vertrag)  
    throws FormulaExecutionException {  
  
    try {  
        return deckung.getVersSumme().multiply(Decimal.valueOf("0.01"), BigDecimal.ROUND_HALF_UP);  
    } catch (Exception e) {  
        StringBuffer parameterValues = new StringBuffer();  
        parameterValues.append("deckung=");  
        parameterValues.append(deckung == null ? "null" : deckung.toString());  
        parameterValues.append(", ");  
        parameterValues.append("vertrag=");  
        parameterValues.append(trag == null ? "null" : vertrag.toString());  
        throw new FormulaExecutionException(toString(), "deckung.versSumme * 0.01",  
            parameterValues.toString(), e);  
    }  
}
```

Tritt beim Ausführen der in Java übersetzten Formel eine Fehler auf, wird eine `RuntimeException` geworfen, die den Formeltext sowie die Stringrepräsentation 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 `berechneJahresbasisbeitragInternal()` 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. Öffnen Sie nun die Klasse im Java-Editor und implementieren Sie die Methode wie folgt.

```
public Money berechneJahresbasisbeitragInternal() {  
    return getHausratZusatzdeckungstypGen().berechneJahresbasisbeitrag(this, getHausratVertrag());  
}
```

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

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

    // Wirksamkeitsdatum des Vertrages setzen, damit die Produktgeneration gefunden wird!
    // Dies muss nach dem Gueltigkeitsbeginn der Generation liegen!
    vertrag.setWirksamAb(new GregorianCalendar(2010, 0, 1));

    // Vertragsattribute setzen
    vertrag.setVersSumme(Money.euro(60000));

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

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

    // Jahresbasisbeitrag berechnen und testen
    deckung.berechneJahresbasisbeitrag();

    // Vericherungssumme der Deckung = 1% von 60.0000, max 5.000 => 600
    // Beitrag = 10% von 600 = 60
    assertEquals(Money.euro(60, 0), deckung.getJahresbasisbeitrag());
}
```

## Schlussbemerkungen

In diesem Tutorial haben wir eine einfache Hausrattarifierung abgebildet. Auch wenn das Modell sehr einfach ist, sollten die folgenden Aspekte deutlich geworden sein:

- Faktor-IPS ist ein Werkzeug zur modellgetriebenen Entwicklung versicherungsfachlicher Systeme. Es unterstützt die Abbildung von Produktinformationen durch die Bereitstellung von Design Patterns für die typischen in diesem Zusammenhang existierenden Anforderungen wie zum Beispiel die Abbildung von Produktänderungen im Zeitablauf und den Zugriff auf Tabellen.
- Die Modellierung und Codegenerierung ist so in Eclipse integriert, dass für Softwareentwickler kein großer Einarbeitungsaufwand anfällt. Für die Entwicklung werden neben Faktor-IPS die Standardwerkzeuge von Eclipse wie Compiler und Debugger genutzt. Der generierte Sourcecode ist leicht verständlich, da die Struktur sich aus dem Modell ergibt.
- Für die Fachabteilung existiert eine leicht zu bedienende Benutzeroberfläche für die Produktdefinition. Neben der reinen Produktstruktur und den Produktdaten kann die Fachabteilung einzelne Produktaspekte auch durch Formeln in Excel-Syntax beschreiben. Programmierung ist seitens der Fachabteilung nicht erforderlich.
- Durch die Aufteilung in unterschiedliche Projekte können einzelne Sparten unabhängig voneinander entwickelt, versioniert und released werden. Dies geschieht wiederum mit den Standardwerkzeugen von Eclipse.

Auf die folgenden in diesem Einführungstutorial nicht behandelten Aspekte wollen wir an dieser Stelle noch einen kurzen Ausblick geben:

### Plausibilisierung

Ein umfangreicher Teil bei der Entwicklung von Geschäftsobjekten ist die Plausibilisierung der Objekte. Wir unterteilen die Plausibilisierung in die beiden Aspekte Auskunft und Prüfung. Die Unterscheidung wird am besten an einem Beispiel deutlich:

- Auskunft: Welche Versicherungssummen können in dem Vertrag gewählt werden?
- Prüfung: Enthält der Vertrag eine gültige Versicherungssumme?

Faktor-IPS unterstützt explizit die Modellierung von Prüfungen und Auskunftsinhalten.

### Testunterstützung

Faktor-IPS erlaubt die Definition und Ausführung von fachlichen Tests durch die Fachabteilung. Hierzu ist diese Funktionalität in die Produktdefinitionsperspektive integriert. Durch einen JUnit-Adapter lassen sich diese Tests darüber hinaus in automatisierte Testprozesse z. B. mit Ant und Cruisecontrol<sup>28</sup> integrieren.

Darüber hinaus wird das Testen des Modells unabhängig von konkreten Produktdaten unterstützt. Dies erlaubt das Testen von Grenzwerten und Kombination, die in den aktuellen Produkten nicht vorkommen. Wenn wir zum Beispiel testen wollen, ob bei unserem Beispiel der Vorschlagswert für die Versicherungssumme richtig berechnet wird, wenn die Fachabteilung den Vorschlagswert pro Quadratmeter mit Nachkommastellen eingibt, dann kann dies mit den aktuellen Produkten HR-Optimal und HR-Kompakt nicht getestet werden, da der Wert auf 600 bzw. 900 Euro festgelegt

---

<sup>28</sup> Cruisecontrol ist ein Werkzeug für die kontinuierliche Integration, einer Best Practice für die Organisation des Buildprozesses. Details unter <http://cruisecontrol.sourceforge.net/>

ist. Um dies zu umgehen können in JUnit-Tests beliebige Produkte und Tabellen im Hauptspeicher erzeugt werden ohne auf die tatsächlichen Produktdaten zugreifen zu müssen.

### Teamunterstützung

Da Faktor-IPS alle Daten Eclipse-konform in Dateien im Eclipse-Workspace ablegt, können Sie wie alle anderen Dateien versioniert werden. Damit stehen alle Teamfunktionen wie Änderungshistorie, Vergleichswerkzeuge etc. zur Verfügung. Für die Fachabteilung gibt es ein spezielles Vergleichswerkzeug für Produktbausteine und Tabelleninhalte.

### Integration in operative Systeme

In diesem Tutorial haben wir die Geschäftsobjektklassen für eine einfache Hausratversicherung implementiert. Die Klassen sind alles einfache Javaklassen<sup>29</sup>, die unabhängig von einer konkreten Infrastruktur wie z. B. einem EJB Container sind.

Die Frage ist natürlich nun, wie diese Klassen in operativen Systemen verwendet werden. Hierzu gibt es grob zwei Varianten:

- **Verwendung als Geschäftsobjekte eines operativen Systems**  
Sie können die Klassen direkt als die Geschäftsobjektklassen eines operativen Systems verwenden. In der Regel sind Anwendungen nach dem Model-View-Controller Design Pattern organisiert (oder sollten es zu mindestens sein). Die Klassen würden in diesem Fall die Rolle des Modells bzw. eines Teils hiervon einnehmen. Der technische Zugriff auf die Modellklassen erfolgt gemäß der Architektur des operativen Systems. Im einfachsten Fall erfolgt dies durch lokale Methodenaufrufe. In einer klassischen J2EE-Architektur sollte der Aufruf gemäß der Patterns BusinessDelegate, DataTransferObject und RemoteFassade erfolgen. Die Persistierung der Geschäftsobjekte erfolgt mit der von ihnen gewählten Persistenztechnologie/-framework.
- **Produktkomponente / Produktserver**  
Unter einer Produktkomponente oder einem Produktserver verstehen wir eine Komponente, die für andere Komponenten oder Anwendungen produktabhängige Services zur Verfügung stellt. Zu diesen Services gehören zum Beispiel Tarifberechnungen, produktbezogene Prüfungen und Produktauskunft. Die Clients der Produktkomponente greifen auf diese natürlich über eine Schnittstelle zu. Im einfachsten Fall greifen die Clients über lokale Methodenaufrufe auf die von Faktor-IPS generierten published Interfaces zu. Für eine klassische J2EE-Architektur würde man wiederum BusinessDelegates, DataTransferObjects und RemoteFassaden verwenden.

Für diese Themen sind weitere Tutorials bzw. Artikel in Vorbereitung. Sie können weitergehende Fragen aber natürlich auch an die Mailingliste schicken oder FaktorZehn kontaktieren.

---

<sup>29</sup> Auch häufig als Plain old Java objects oder kurz POJOs bezeichnet.

## Anhang: Tariftabellen

### HR-Optimal

<i><b>Tarifzone</b></i>	<i><b>Beitragssatz</b></i>
I	0.80
II	1.00
III	1.44
IV	1.70
V	2.00
VI	2.20

### HR-Kompakt

<i><b>Tarifzone</b></i>	<i><b>Beitragssatz</b></i>
I	0.60
II	0.80
III	1.21
IV	1.50
V	1.80
VI	2.00