



Tutorial zur Version 1.0

(Dokumentversion 1276)

Inhaltsverzeichnis

Einleitung.....	3
„Hello FaktorIPS“.....	5
Arbeiten mit Modell und Sourcecode.....	9
Definition eines einfachen Hausratmodells.....	16
Aufnahme von Produktaspekten ins Modell.....	21
Definition der Produkte.....	30
Verwendung von Tabellen.....	35
Implementieren der Beitragsberechnung.....	40
Spartenübergreifende Berechnung.....	40
Beitragsberechnung für Hausrat.....	42
Schreiben eines JUnit-Tests für die Beitragsberechnung.....	47
Schlussbemerkungen.....	49
Anhang: Tariftabellen.....	51

Einleitung

FaktorIPS ist ein OpenSource-Werkzeug zur modellgetriebenen Entwicklung versicherungsfachlicher Softwaresysteme¹ mit Fokus auf der einheitlichen Abbildung des Produktwissens. Insbesondere können mit FaktorIPS 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 FaktorIPS 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 FaktorIPS
In dem Kapitel wird ein erstes FaktorIPS 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.
- Definitions 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.
- 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 unter Verwendung der Excel-basierten Formelsprache von FaktorIPS implementiert.
- Schreiben eines JUnit-Tests für die Beitragsberechnung
Zum Schluss wird die Beitragsberechnung mit einem JUnit-Testfall getestet.

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

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 herunterladen und installieren.

„Hello FaktorIPS“

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

Falls sie FaktorIPS noch nicht installiert haben, tun sie das jetzt. Die Software und die Installationsleitung finden Sie auf www.faktorips.org. In diesem Tutorial verwenden wir Eclipse 3.2 und FaktorIPS in Englisch. Starten sie nun Eclipse. Am besten verwenden sie für dieses Tutorial einen eigenen Workspace. Wenn FaktorIPS korrekt installiert ist, sollten sie bei geöffneter Java-Perspektive in der Toolbar folgende Symbole sehen:



Abbildung 1: FaktorIPS Icons

FaktorIPS-Projekte sind normale Java-Projekte mit einer zusätzlich FaktorIPS-Nature. Als erstes legen sie also ein neues Java-Projekt mit dem Namen „Grundmodell“ an. In diesem Projekt werden wir die spartenübergreifenden Klassen modellieren. Achten sie darauf, dass sie für den Sourcecode und die Klassendateien (.class) getrennte Verzeichnisse verwenden.

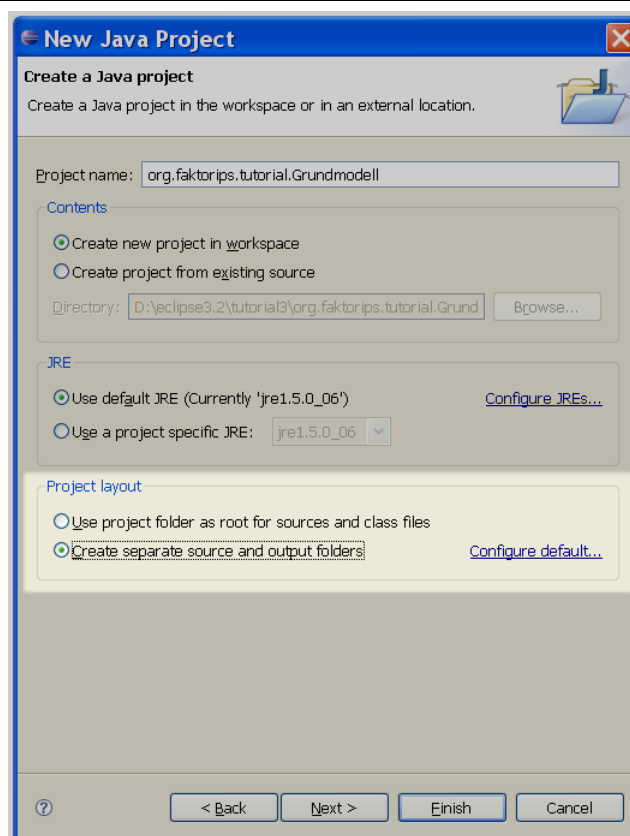


Abbildung 2: Getrennte Verzeichnisse für Sourcecode und Klassendateien

Die FaktorIPS-Nature fügen Sie dem Projekt hinzu, indem sie es im Java Package-Explorer markieren und im Kontextmenü *Add IPS-Nature* wählen.

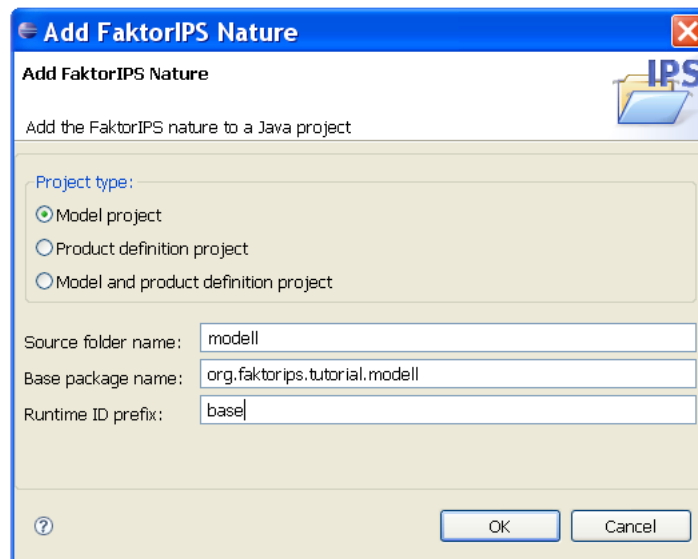


Abbildung 3: Hinzufügen der IPS-Nature

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 FaktorIPS 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 strukturiert werden. FaktorIPS 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 Vertrag² 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)³.

Wechseln sie zunächst in den Modell-Explorer von FaktorIPS (direkt neben dem Package-Explorer⁴).

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

3 Der Codegenerator von FaktorIPS 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.

4 Sollte der Modell-Explorer nicht angezeigt werden, öffnen sie ihn über Window → Show View → Other... → FaktorIPS Produktdefinitionen → Modell-Explorer

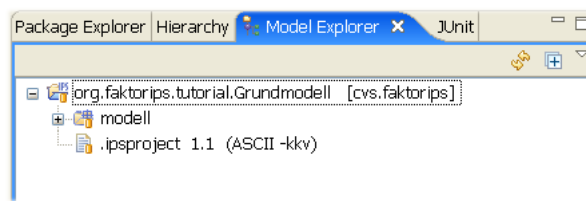



Abbildung 4: Ansicht der Projekte im Modell-Explorer

Im Modell-Explorer wird die Modelldefinition ohne die Java-Details dargestellt. Die spartenübergreifenden Sparten 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.

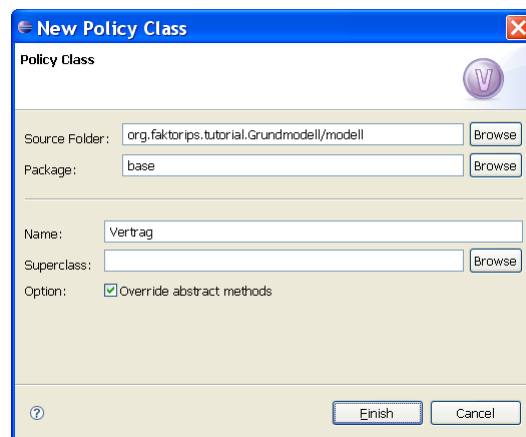


Abbildung 5: Anlegen einer neuen Vertragsklasse

In dem Dialog sind Sourceverzeichnis und Package bereits entsprechend vorgelegt und sie geben noch den Namen der Klasse an, also Vertrag und klicken auf *Finish*. FaktorIPS hat jetzt eine neue Datei mit dem Namen „Vertrag.ipspct“ angelegt und den Editor zur Bearbeitung dieser Klasse geöffnet.

Weiterhin hat der Codegenerator 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 sind⁵.

⁵ 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 Package-Explorer auf die Datei „Vertrag.ipspect“. In dem Editor klicken sie auf den Button *New* im Abschnitt *Attributes*. Es öffnet sich der folgende Dialog:

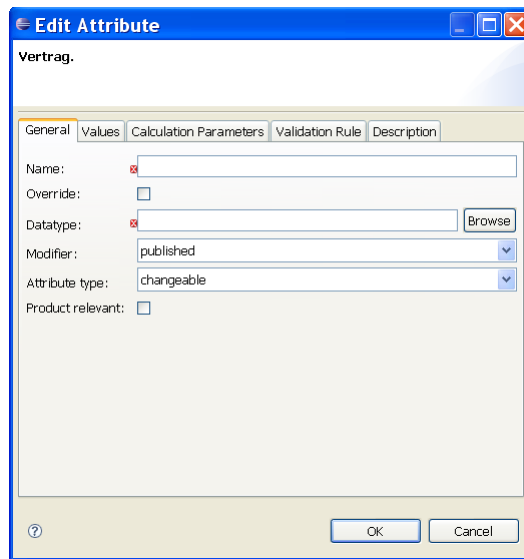


Abbildung 6: Dialog zum Anlegen eines neuen Attributs

Die Felder haben die folgende Bedeutung:

<i>Feld</i>	<i>Bedeutung</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 ⁶ .
Datatype	Datentyp des Attributs.
Modifier	Analog zum Modifier in Java. Der zusätzliche Modifier published bedeutet, dass die Eigenschaft ins published Interface aufgenommen wird.

⁶ 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"> • changeable Änderbare Eigenschaften, also solche mit Getter- und Setter-Methoden. • constant Konstante, nicht änderbare Eigenschaft. • Derived (cached, computation by explicit method call) 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 die getBruttobeitrag() abgerufen werden. • Derived (computation on each call of the getter method) Abgeleitete Eigenschaften im UML Sinne. Die Eigenschaft wird bei jedem aufruf der Getter-Methode berechnet. Zum Beispiel kann das Alter einer versicherten Person jedem Aufruf von getAlter() aus dem Geburtstag ermittelt werden.
Product relevant	Indikator, ob es für dieses Attribute produktseitig Konfigurationsmöglichkeiten gibt. Hierzu später mehr.

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 gesamt 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 ein⁷.

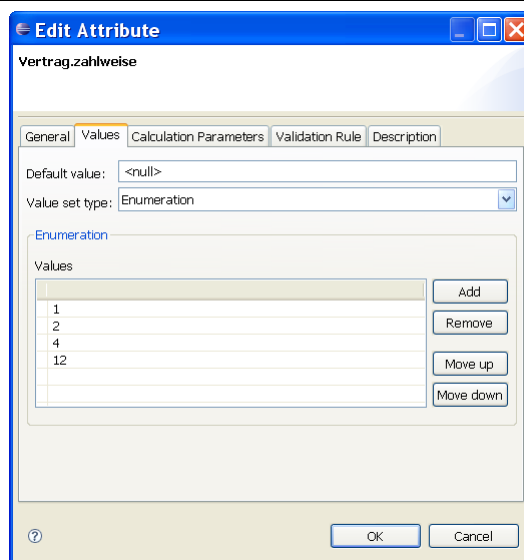


Abbildung 7: Festlegen zulässiger Werte für ein Attribut

Setzen sie nun einmal den Default Value auf 0. FaktorIPS markiert den Default Value mit einer

⁷ Sie können auch eine Klasse Zahlungsweise als Datentyp verwenden, wenn sie diesen entsprechend registrieren. Details folgen.

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 FaktorIPS zu erläutern. Schließen sie dazu den Dialog und speichern die Vertragsklasse. Im Problems-View von Eclipse wird nun die Warnung als Problem angezeigt. FaktorIPS 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.

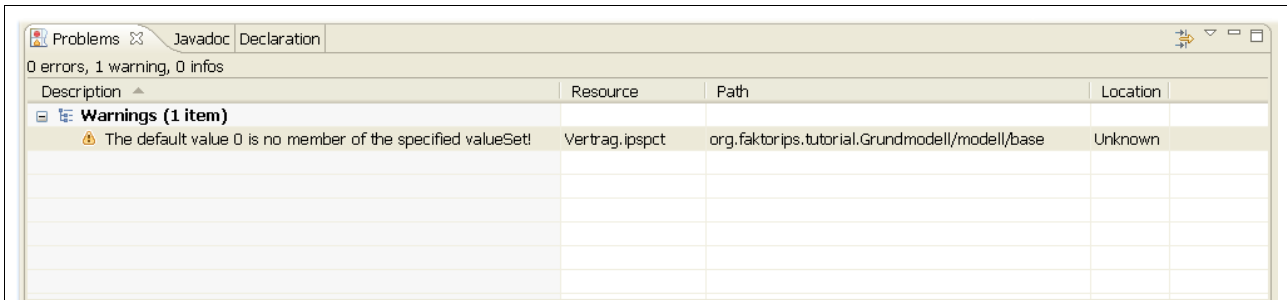


Abbildung 8: Anzeige von Fehlern in der Problems-View

Setzen sie als Defaultwert wieder `<null>` ein und speichern die Vertragsklasse. Die Warnung wird damit wieder aus dem Problem-View entfernt.

FaktorIPS 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:

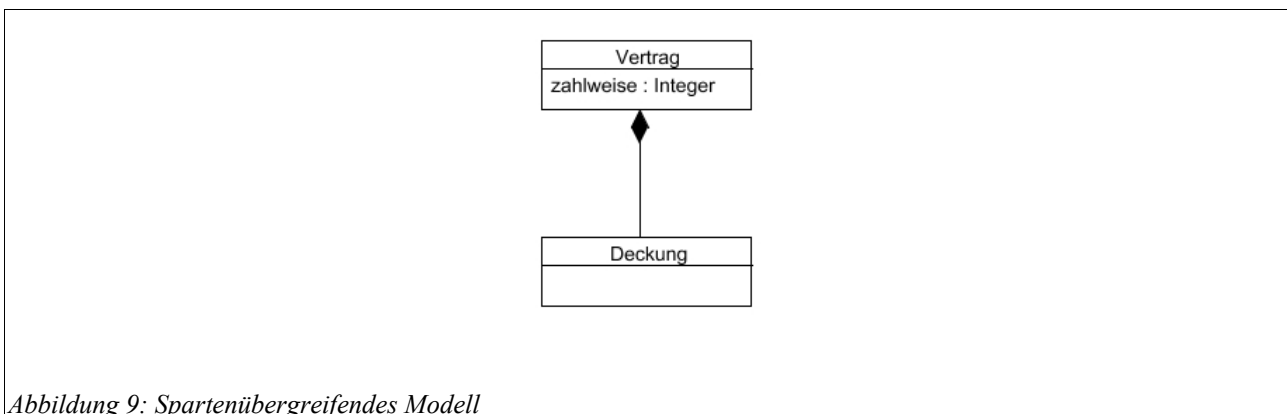


Abbildung 9: Spartenübergreifendes Modell

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 *Relations* überschrieben ist.

The 'New relation' dialog box is shown with the 'Target' tab selected. The title bar reads 'New relation'. Below the title bar, the text 'Select the target and the relation type' is displayed. The 'Target:' field contains 'base.Deckung' and has a 'Browse' button to its right. The 'Type:' dropdown menu is set to 'Composition'. The 'Read-only container:' checkbox is unchecked. There is a large empty text area for 'Description:'. At the bottom, there are four buttons: a help icon (?), '< Back', 'Next >', 'Finish', and 'Cancel'.

Abbildung 10: 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. Das Feld *Read-only container* lassen sie bitte leer. 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.

The 'New relation' dialog box is shown with the 'Relation properties' tab selected. The title bar reads 'New relation'. Below the title bar, the text 'Define relation properties' is displayed. The 'Policy side:' section contains four fields: 'Minimum cardinality:' with '0', 'Maximum cardinality:' with '*', 'Target role (singular):' with 'Deckung', and 'Target role (plural):' with 'Deckungen'. The 'Product side:' section contains five fields: 'Product relevant:' with an unchecked checkbox, 'Minimum cardinality:' with '0', 'Maximum cardinality:' with '*', 'Target role (singular):' with an empty field, and 'Target role (plural):' with an empty field. At the bottom, there are four buttons: a help icon (?), '< Back', 'Next >', 'Finish', and 'Cancel'.

Abbildung 11: 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 FaktorIPS sind immer gerichtet, so ist es auch möglich die Navigation nur in eine Richtung zuzulassen. Hier wählen sie bitte *New reverse relation* und gehen zur nächsten Seite. Hier brauchen sie nur noch die Rollenbezeichnung im Plural 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 `v.addDeckung(IDeckung d)` liefert `d.getVertrag()` den Vertrag `v` zurück. Dies zeigt ein kurzer Blick in die Implementierung der Methode `addDeckung`⁸ in der Klasse `Vertrag`.

```
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).

⁸ Die generierten Javadocs zeigen wir von nun an nicht mehr im Tutorial, da die Methoden im Text erläutert werden.

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.

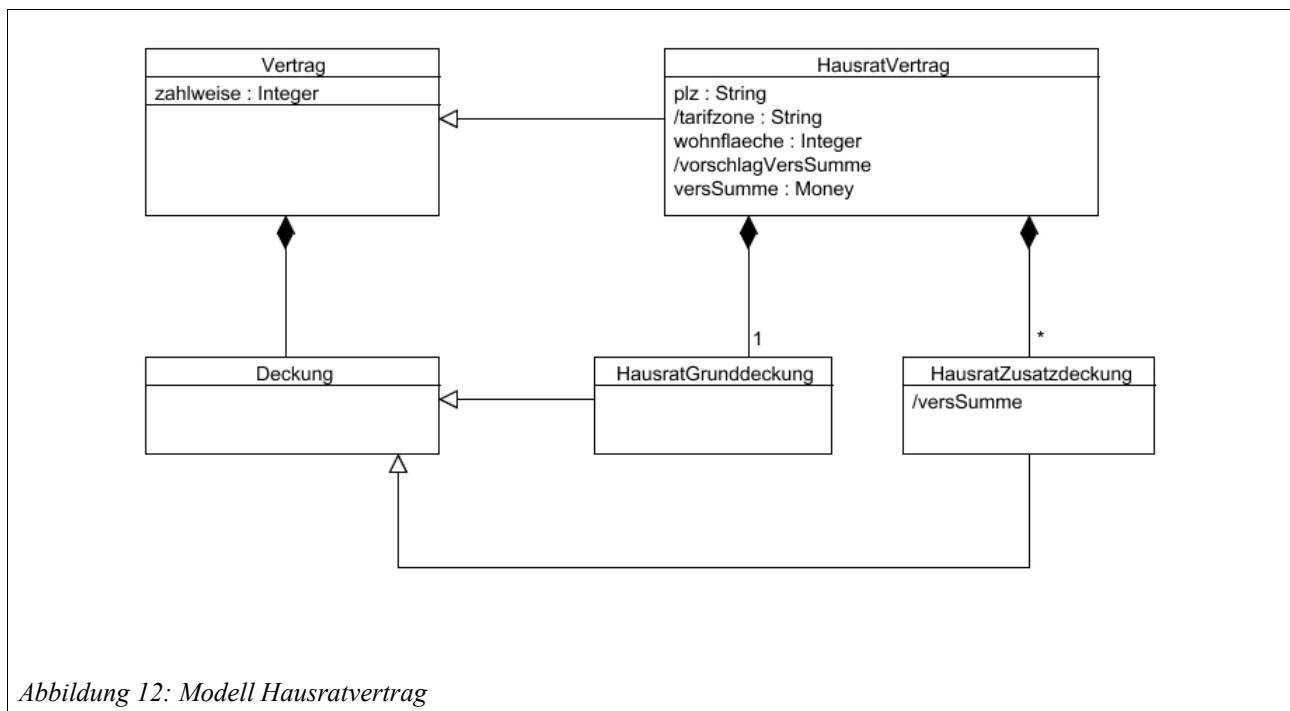


Abbildung 12: Modell Hausratvertrag

Der Hausratvertrag ist natürlich ein spezifischer Vertrag. Die genaue Bedeutung der einzelnen Attribute erläutern wir weiter unten, bei der Anlage der Attribute. 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. Die Zusatzdeckungen haben eine eigene Versicherungssumme, die aus der Versicherungssumme des Vertrags errechnet wird. Die Versicherungssumme der Zusatzdeckung ist deswegen ein abgeleitetes Attribut⁹. Wie die Versicherungssumme einer Zusatzdeckung berechnet wird ist Teil der Produktkonfiguration und wird im nächsten Kapitel beschrieben.

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).

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 FaktorIPS ein Konzept analog zum Classpath in Java. In jedem FaktorIPS-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.

⁹ Abgeleitete Attribute werden in UML durch einen vorangestellten Querstrich (/) gekennzeichnet.

Suchen sie bitte das XML-Tag `IpsObjectPath`. Der `IpsObjectPath` beschreibt, wo FaktorIPS für das Projekt nach FaktorIPS-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 FaktorIPS 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 FaktorIPS werden vordefinierte Standard-Datentypen mitgeliefert¹⁰. 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 (FaktorIPS-)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>Name : Datentyp</i>	<i>Beschreibung, Bemerkung</i>
plz : String	Postleitzahl des versicherten Hausrats
/tarifzone : String	Die Tarifzone 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 max=unbeschränkt.

¹⁰ 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>
	Hierzu das Feld max leer lassen.
/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=unbeschränkt.

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

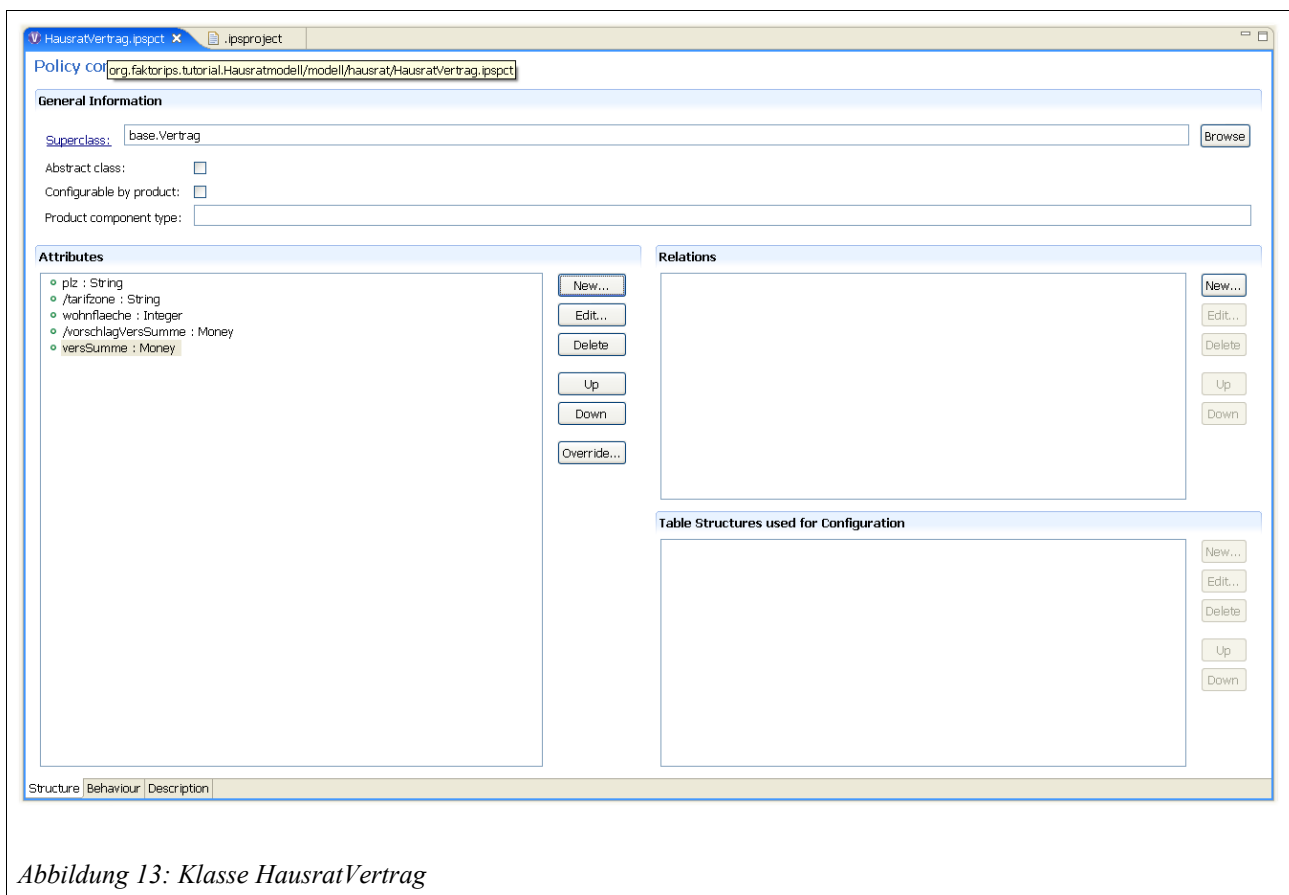


Abbildung 13: Klasse HausratVertrag

Legen sie nun die beiden Klassen „HausratGrunddeckung“ und „HausratZusatzdeckung“ an. An der Zusatzdeckung definieren sie noch das abgeleitete Attribut „versSumme“.

Bevor wir nun die Beziehungen zwischen den Hausratklassen anlegen, analysieren wir noch einmal das oben abgebildete UML Modell. Es gibt eine Kompositbeziehung zwischen den sparten-übergreifenden 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 Hausrat-

zusatzdeckungen 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 FaktorIPS können wir die von uns gewollte Semantik ins Modell aufnehmen, indem wir die Beziehung VertragDeckung als Read-only-Container definieren und in die Beziehungen zwischen Hausratvertrag und seinen Deckungsklassen jeweils auf diese Containerbeziehung referenzieren¹¹. 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.

Markieren wir als erstes die Vertrag-Deckung-Beziehung als Containerbeziehung. Hierzu öffnen sie zunächst den Editor für die allgemeine Vertragsklasse 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:

¹¹ In UML Terminologie handelt es sich um eine abgeleitete Beziehung deren genaue Semantik über eine Constraint beschrieben werden müsste.

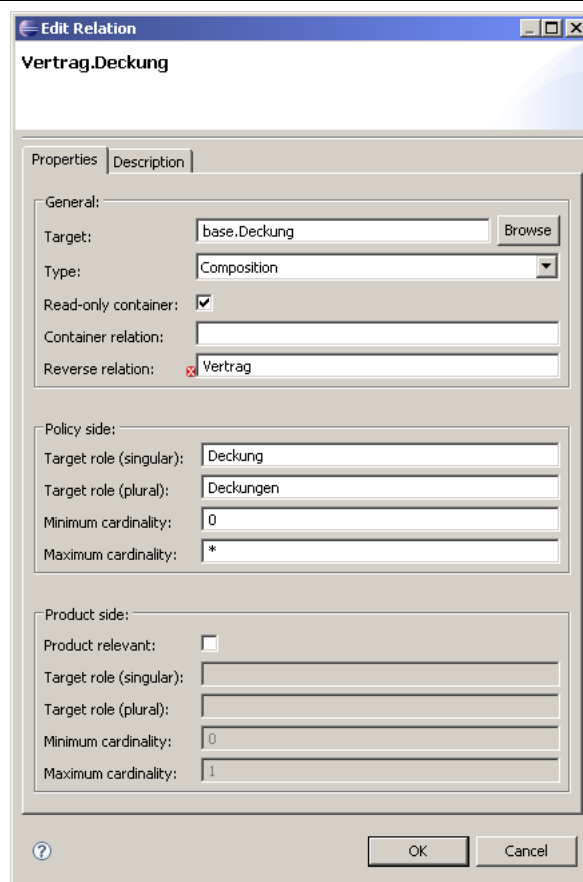


Abbildung 14: Markieren einer Beziehung als Containerbeziehung

Mit dem Anhängen der Checkbox wird das Feld *Reverse relation* als fehlerhaft markiert, da auch die Rückwärtsbeziehung konsistent als Containerbeziehung markiert werden muss. Schließen sie den Dialog. 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 (Codebeispiel folgt), sondern erst in den spartenspezifischen Klassen. 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 FaktorIPS die Abhängigkeiten zwischen den Modellklassen verwaltet und bei Änderungen an einer Klasse auch alle abhängigen Klassen neu prüft.

Öffnen sie nun den Editor für die Klasse *Deckung* und markieren diese analog als abstrakt und die Rückwärtsbeziehung als Containerbeziehung und speichern die Klasse.

Nun können sie die Beziehungen zwischen *Hausratvertrag*, *Hausratgrunddeckung* und *Hausratzusatzdeckungen* mit dem ihnen schon bekannten Assistenten anlegen. Dieser bietet ihnen nun eine neue zweite Seite an, auf der sie beim Anlegen die Containerbeziehung auswählen. Bei der Beziehung *HausratVertrag*-*HausratGrunddeckung* setzen sie die Maximum Cardinality auf 1.

Das gesamte Modell sollte bei im Modellexplorer nun wie folgt aussehen:

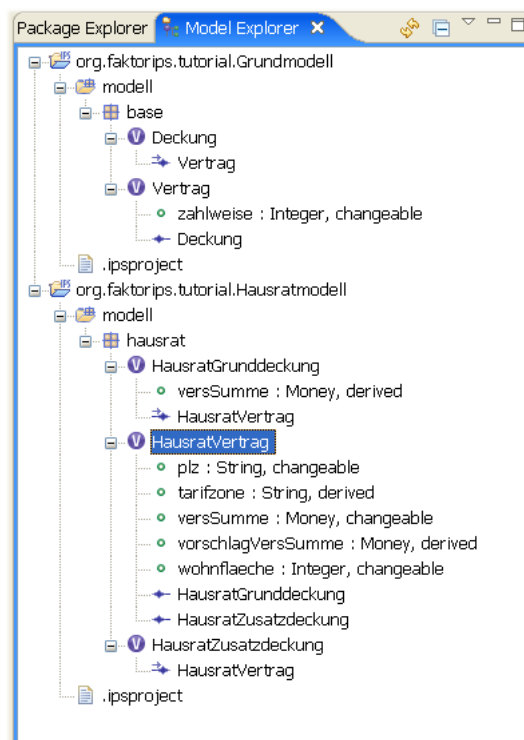


Abbildung 15: Zwischenstand des Modells im Modellexplorer

Werfen wir zum Abschluss des Kapitels noch einen Blick in den generierten Sourcecode¹². 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 (wenn vorhanden) und alle Zusatzdeckungen zurück.

```
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;
}
```

¹² 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 FaktorIPS machen, 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:

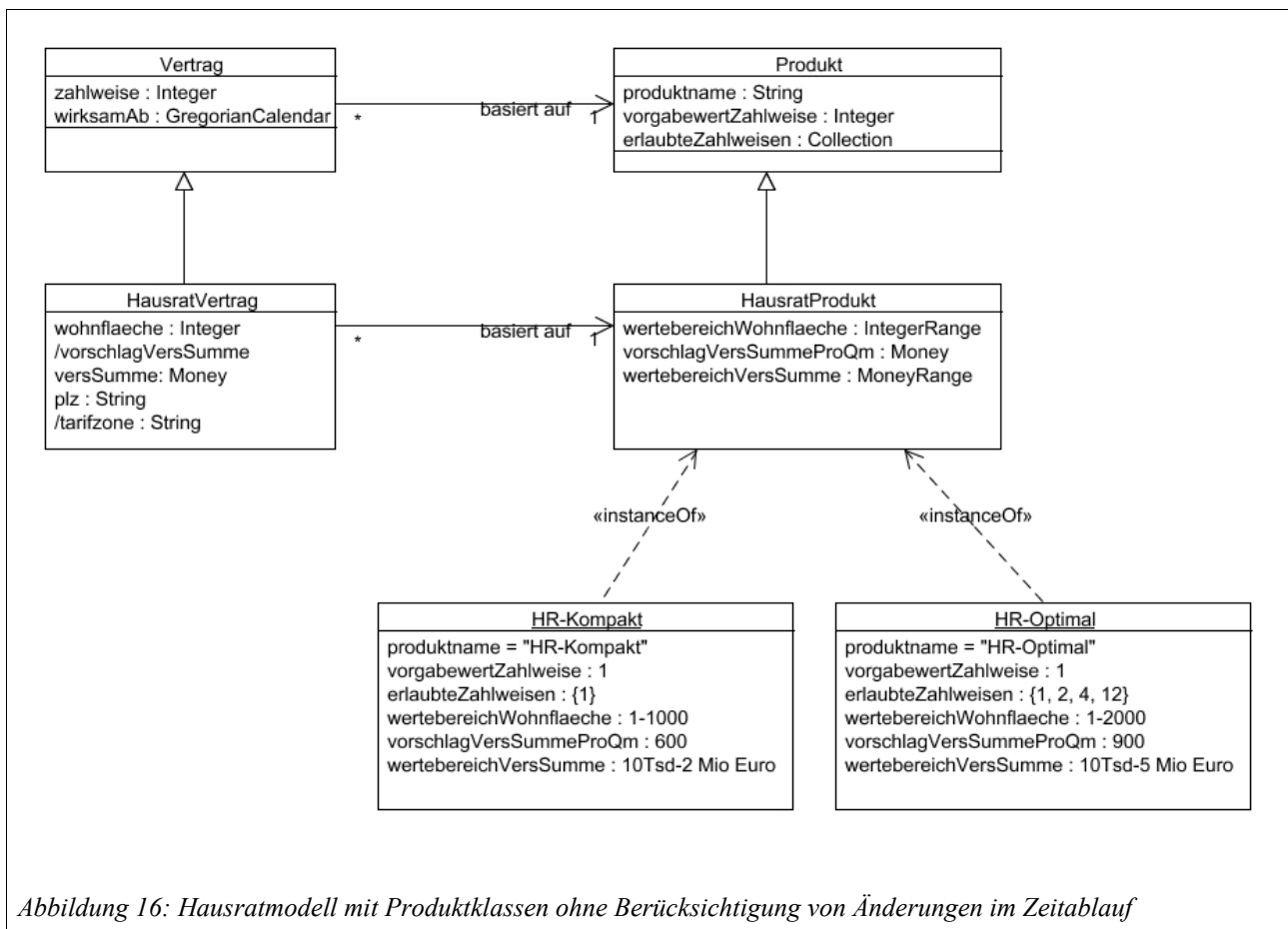
<i>Eigenschaft von Hausratvertrag</i>	<i>Konfigurationsmöglichkeiten</i>
zahlweise	Die im Vertrag erlaubten Zahlweisen. Der Vorgabewert für die Zahlweise bei Erzeugung eines neuen Vertrags.
wohnflaeche	Bereich (min, max), in dem die Wohnfläche liegen muss.
vorschlagVersSumme	Definition eines Vorschlagwertes für einen Quadratmeter Wohnfläche. Der Vorschlag für die Versicherungssumme ergibt sich dann durch Multiplikation mit der Wohnfläche ¹³ .
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>Konfigurationsmöglichkeit</i>	<i>HR-Kompakt</i>	<i>HR-Optimal</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:

¹³ 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 Änderung¹⁴:

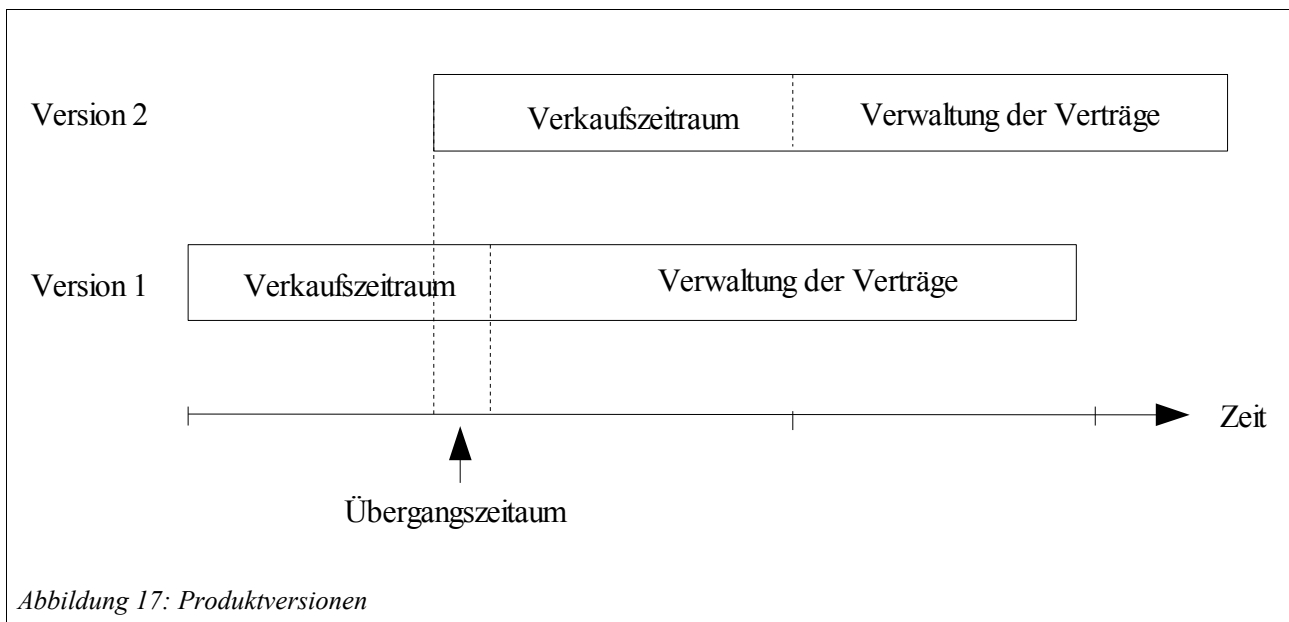
Version

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.

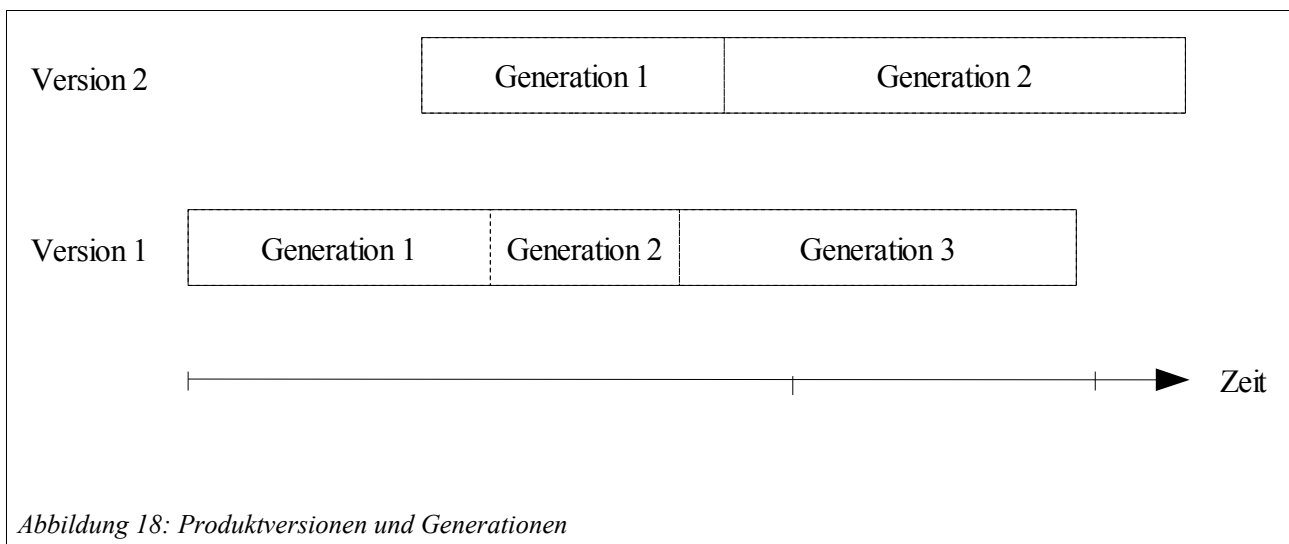
I.d.R. gibt es zu einem Zeitpunkt eine für das Neugeschäft gültige Version. Das ist die, in deren Verkaufszeitraum der Zeitpunkt fällt. Es kann allerdings zu einem Zeitpunkt auch mehrere verkaufbare Versionen eines Produktes geben. In der Praxis kommt dies vor, wenn in der Übergangszeit von einer alten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

¹⁴ 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 FaktorIPS kann die verwendete Namensgebung sowohl für die Benutzeroberfläche (Window→Preferences: FaktorIPS) als auch für den generierten Sourcecode (in der „ipsproject“ Datei im Abschnitt GeneratedSourcecode) konfiguriert werden.

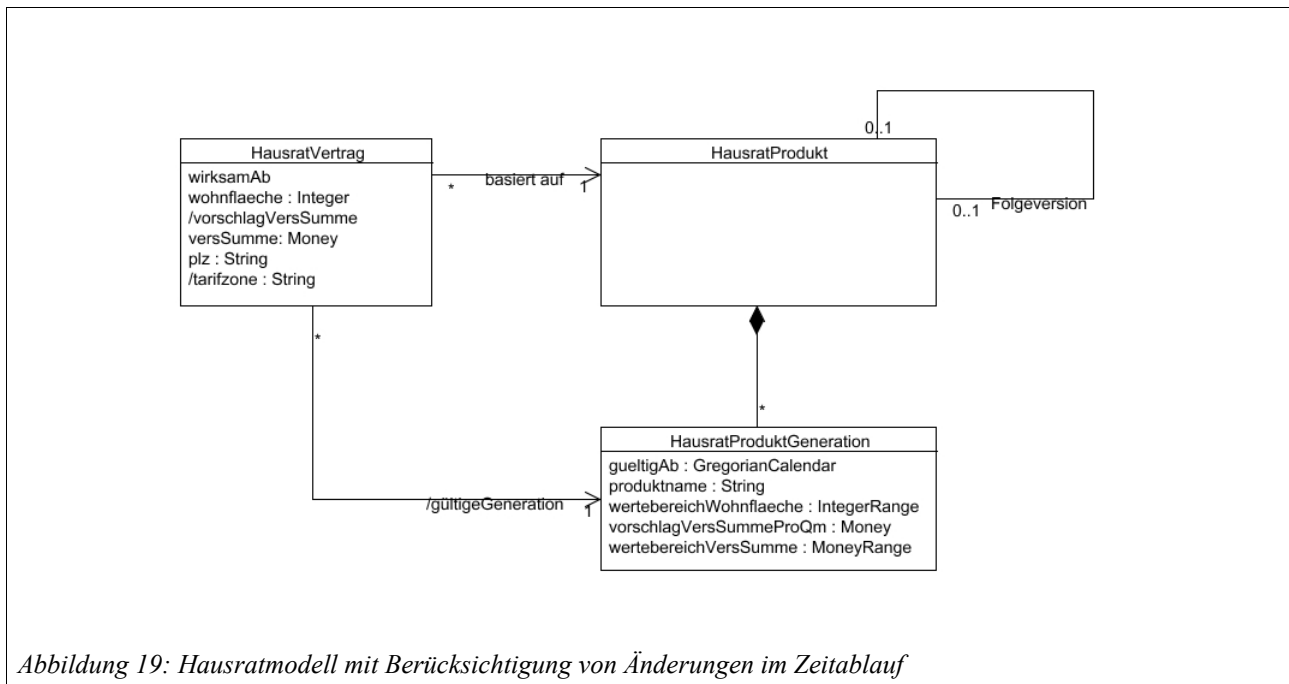


Generation

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.



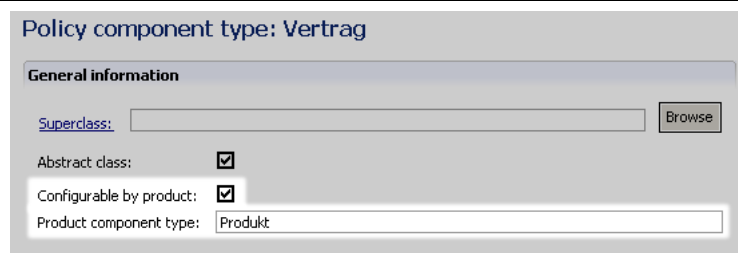
Das Konzept Produktversion bilden wir über eine Beziehung zwischen Produkten ab, da die Unterscheidung zwischen einem neuen Produkt und einer neuen Version sehr unscharf ist. Ob zum Beispiel die Umstellung des Kfz-Tarifs auf einen Scoringtarif eine neue Version oder ein neues Produkt darstellt ist eine eher akademische Betrachtung¹⁵.

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)¹⁶.

Genug Theorie, erweitern wir unser Hausratmodell in FaktorIPS um die Produktklassen. Als erstes definieren, wir dass der spartenübergreifende Vertrag (`base.Vertrag`) auf einem Produkt basiert. Hierzu öffnen sie den Editor für die Vertragsklasse, haken „Configurable by product“ an und geben im Feld *Product component type* „Produkt“ ein.

¹⁵ Die Nachfolgeversions-Beziehung ist in FaktorIPS 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.

¹⁶ 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.



Policy component type: Vertrag

General information

Superclass: Browse

Abstract class: ☒

Configurable by product: ☒

Product component type:

Abbildung 20: Setzen einer Vertragsklasse als Produkt-konfigurierbar

Nun definieren wir noch, dass das Produkt einen Namen hat. Hierzu legen sie ein neues konstantes, produkt-relevantes Attribut „produktname“ vom Datentyp String an.

Nun definieren wir, dass die erlaubten Zahlungsweisen und der Vorgabewert im Produkt konfiguriert werden können. Hierzu öffnen sie zunächst den Dialog zum Editieren des Attributs „zahlweise“, haken die Checkbox *Product relevant* an und schließen den Dialog wieder.

Nach dem Speichern der Änderungen hat FaktorIPS nun zusätzlich zu den Typen `IVertrag` und `Vertrag` 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 published Interface der Produktgeneration gibt es 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 Generation zuzugreifen, auf der der Vertrag basiert.

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

Nun legen sie noch das Attribut „wirksamAb“ mit Datentyp `GregorianCalendar` an. Dieses Attribut ist nicht produktrelevant. Woher weiß nun die Vertragsklasse, dass sie mit dem Attribut „wirksamAb“ die gültige Generation ermitteln muss? Hierzu hat FaktorIPS 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 erhalten¹⁷. 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.

Nun definieren wir die Konfigurationsmöglichkeiten des Hausratvertrags. Definieren sie die Klasse als erstes als *Configurable by product*, die Konfigurationsklasse nennen wir `HausratProdukt`. Markieren sie die Attribute „wohnflaeche“ und „versSumme“ als produkt-relevant.

Um den Vorschlag für die Versicherungssumme berechnen zu können, definieren wir ein neues konstantes (Attribute Type=constant), produkt-relevantes Attribut „vorschlagVersSummeProQm“ vom Datentyp `Money`. Dies ist der Vorschlagswert für einen Quadratmeter Wohnfläche. Diese Produkteigenschaft nutzen wir nun um den Vorschlag für die Versicherungssumme des Vertrags zu berechnen. Hierzu öffnen sie die Javaklasse `HausratVertrag`. `FaktorIPS` hat für die abgeleitete Eigenschaft „vorschlagVersSumme“ eine Methode `getVorschlagVersSumme()` generiert. In dieser implementieren wir nun die – zugegeben triviale - Ermittlung des Vorschlags. Dazu multiplizieren wir die Wohnfläche des Vertrags mit dem Vorschlagswert für einen Quadratmeter wie folgt:

¹⁷ Es handelt sich also um eine `TemplateMethod` (im Sinne der GoF-Patterns), die in der abstrakten Basisklasse `PolicyComponent` definiert ist.

```
/**
 * {@inheritDoc}
 *
 * @modifiable
 */
public Money getVorschlagVersSumme() {
    IHausratProduktGen gen = getHausratProduktGen();
    if (gen==null) {
        return Money.NULL;
    }
    return gen.getVorschlagVersSummeProQm().multiply(wohnflaeche);
}
```

Die Annotation `@modifiable` bedeutet, dass der Inhalt dieser Methode vollständig vom Entwickler zu implementieren ist.

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`. Öffnen sie bitte den Editor für die Deckung und fügen dieser den Deckungstyp und das konstante, produkt-relevante Attribut „`bezeichnung`“ hinzu. Analog definieren sie für die Klasse `HausratGrunddeckung` einen `HausratGrunddeckungstyp`, allerdings ohne eine Eigenschaft `Bezeichnung`, diese wird von der Superklasse geerbt. Die Konfigurationsmöglichkeiten der Grunddeckung verschieben wir noch bis wir die Beitragsberechnung implementieren.

Die Konfigurationsklasse der `HausratZusatzdeckung` nennen wir `HausratZusatzdeckungstyp`. Bei der Zusatzdeckung müssen wir noch festlegen, wie die Versicherungssumme ermittelt wird. Hierzu definieren wir die zwei Konfigurationseigenschaften „`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 folgende Abbildung zeigt das Klassenmodell:

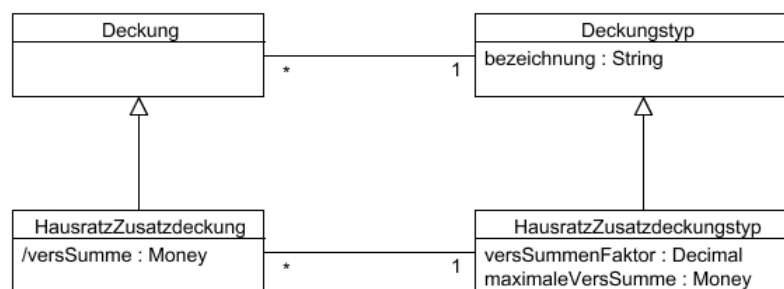


Abbildung 21: `HausratZusatzdeckung` und `HausratZusatzdeckungstyp`

Öffnen sie also noch den Editor für die `HausratZusatzdeckung` und definieren den `HausratZusatzdeckungstyp` mit den beiden konstanten, produktrelevanten Attributen `versSummenFaktor`

und maximaleVersSumme. Nun implementieren wir auch direkt die Ermittlung der Versicherungssumme. 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 (maxVs.greaterThan(vs)) {
        return maxVs;
    }
    return vs;
}
```

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:

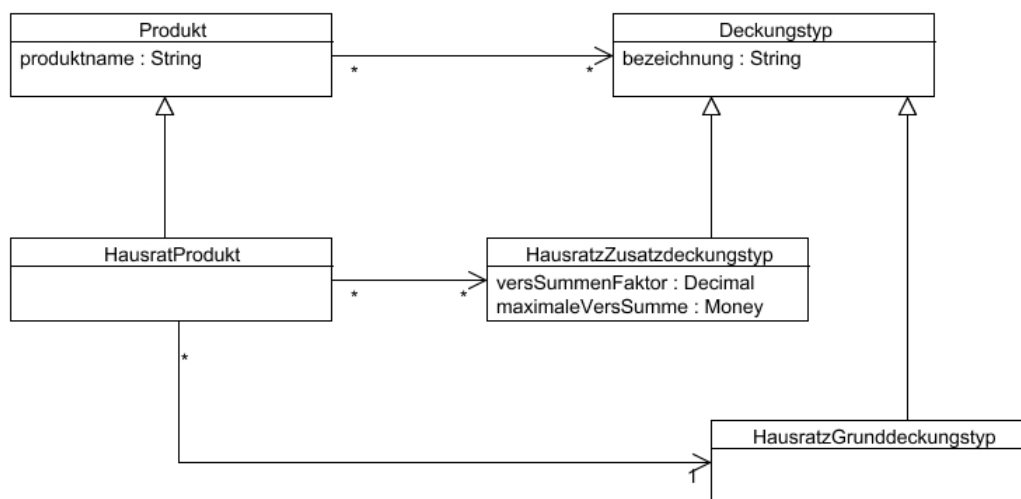


Abbildung 22: Modell der Produktkonfigurationsklassen

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

Definieren wir diese Beziehungen also nun in FaktorIPS. Öffnen sie hierzu zunächst den Editor für die Klasse Vertrag und hier den Dialog zum Bearbeiten der Beziehung zur Deckung. In dem Abschnitt *Product side* haken sie bitte die Checkbox *Product relevant* an und geben als Rollenname „Deckungstyp“ bzw. „Deckungstypen“ und für die Kardinalitäten 0 und * ein. Schließen sie den Dialog und speichern den Vertrag. Da wir nur vom Produkt zum Deckungstyp navigieren bleibt die Rückwärtsbeziehung unverändert¹⁸. 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 Klassen¹⁹.

Analog definieren sie jetzt noch die Beziehungen zwischen Hausratprodukt und den Hausratzusatzdeckungstypen. Achten sie bei dem Grundzusatzdeckungstyp darauf die maximale Kardinalität auf eins zu setzen.

¹⁸ Natürlich können sie auch programmatisch suchen, in welchen Produkten ein Deckungstyp enthalten ist. Die Rückwärtsbeziehung ist nur nicht im Modell unmittelbar navigierbar.

¹⁹ Bei der Beziehung handelt es sich also auch um eine abgeleitete Beziehung im Sinne der UML.

Definition der Produkte

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

Als erstes richten sie bitte noch ein neues Projekt mit dem Namen „Hausratprodukte“ und dem Sourceverzeichnis „produktdaten“ 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 produktdaten“ und als *Runtime-ID Prefix* „hausrat.“. Achten sie darauf, dass der Prefix mit einem Punkt (.) endet. FaktorIPS 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 zusammen²⁰. 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 andern 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 Produktdefinitionsperspektive über „Window→Open Perspective→Other“, in der Liste dann *Produktdefintion* auswählen²¹. 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 FaktorIPS sehen (und nicht auch Java-Marker u.a.) müssen sie im Problemsview den FaktorIPS ein- und alle anderen Filter (standardmäßig mindestens der Defaultfilter) ausschalten.

20 In der folgenden Version von FaktorIPS wird für die Implementierung eigener Verfahren zur Vergabe der RuntimeId ein entsprechender Extension Point bereitgestellt werden.

21 Für die Verwendung von FaktorIPS durch die Fachabteilung gibt es auch eine eigene Installation (In Eclipse Terminologie: ein eigenes Produkt), bei der ausschließliche die Produktdefinitionsperspektive verfügbar ist.

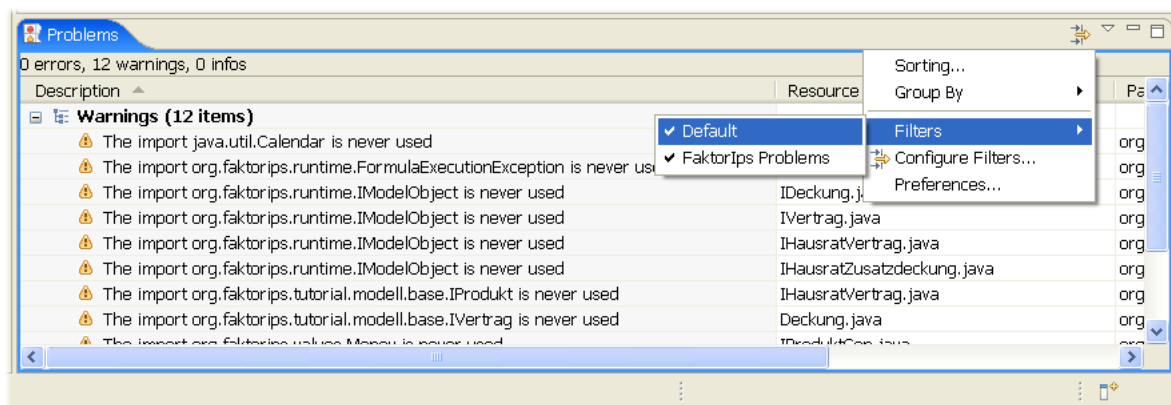




Abbildung 23: Ausschalten der Problemfilter bis auf den FaktorIPS-Filter

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

Sie können unter dem IpsProjekt-Knoten 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.

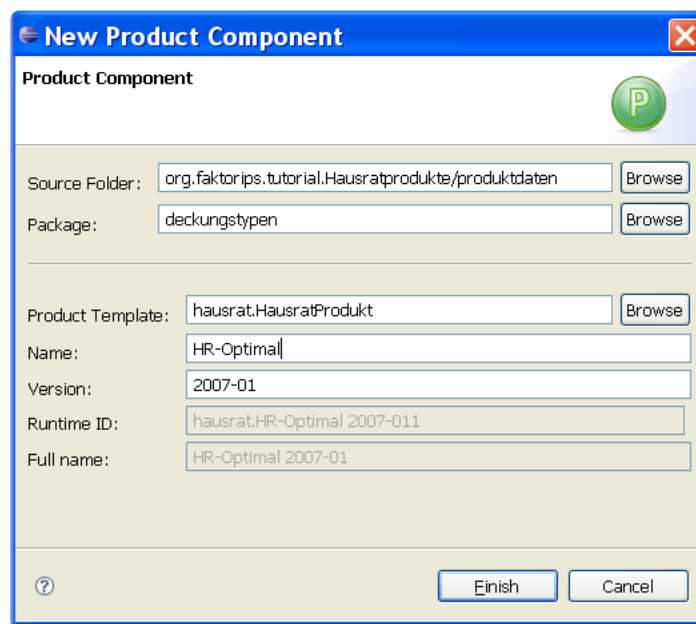


Abbildung 24: Anlegen eines neuen Produkts

Die ersten beiden Felder sind bereits vorbelegt und den Fokus hat das Feld zum Auswählen der Produktklasse, auf dem der neue Baustein basieren soll. In der Perspektive für die Fachabteilung sprechen wir von Vorlage. Wählen sie „hausrat.HausratProdukt“ aus. Sie können hier wie immer in Eclipse und FaktorIPS die Vervollständigung mit *Strg-Space* aufrufen. 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...→FaktorIPS→Can modify runtime id) geändert werden.

Drücken sie nun Finish. FaktorIPS legt den Produktbaustein im Dateisystem an und öffnet den Editor.

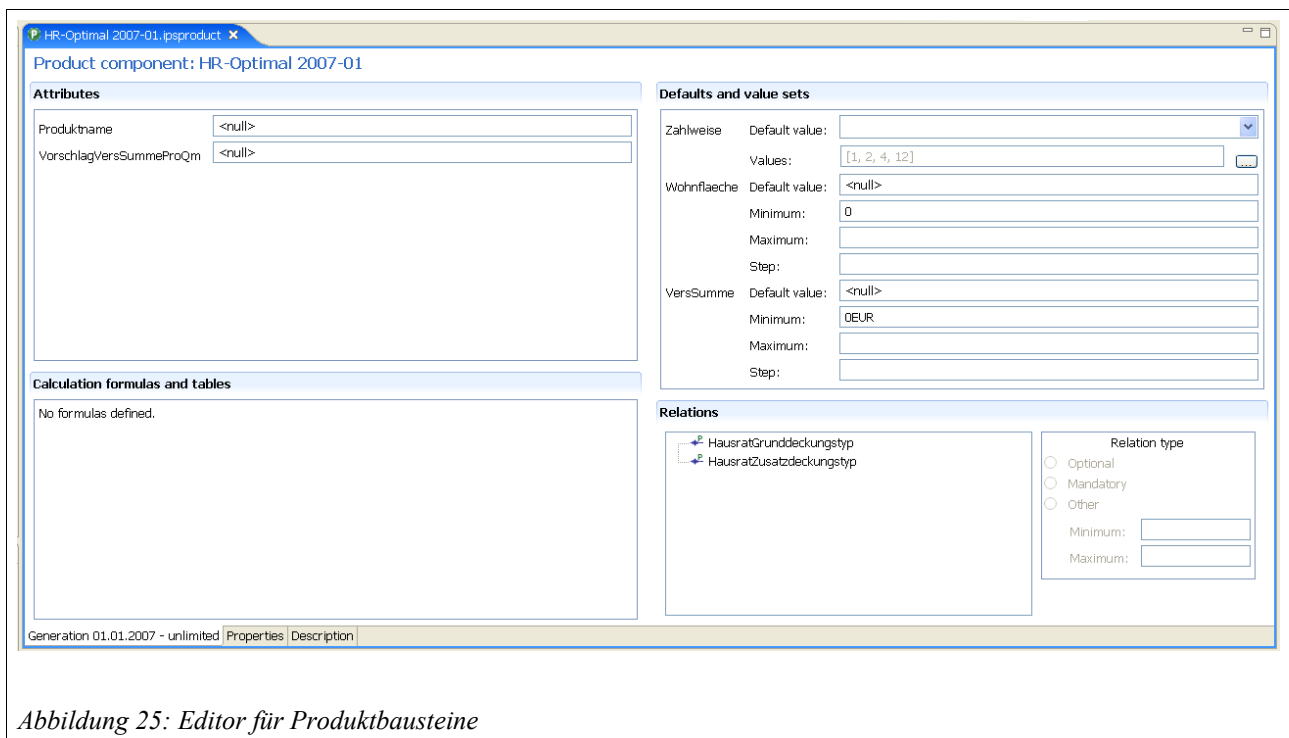


Abbildung 25: Editor für Produktbausteine

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 Bereiche eingeteilt:

Attributes

Enthält die Eigenschaften der Generation.

Calculation Formulas and Tables

Enthält die Berechnungsvorschriften. Hierzu mehr bei der Implementierung der Beitragsberechnung.

Defaults and Value Sets

Enthält die Vorbelegungswerte und Wertebereiche für die Vertragseigenschaften.

Relations

Enthält die verwendeten anderen Produktbausteine.

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

<i>Konfigurationsmöglichkeit</i>	<i>HR-Optimal</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

<i>Konfigurationsmöglichkeit</i>	<i>HR-Optimal</i>
Vorgabewert Versicherungssumme	<null>
Versicherungssumme	10000EUR – 5000000EUR

Nun legen wir den Grunddeckungstyp für das Produkt an. Markieren sie hierzu den Ordner „Deckungstypen“ und legen einen neuen Produktbaustein mit Namen „HRD-Grunddeckung-Optimal“ an basierend auf der Vorlage „HausratGrunddeckungstyp“. Der Einfachheit halber legen wir direkt auch noch den Baustein „HRDGrunddeckung-Kompakt“ an.

Nun wollen wir zwei Zusatzdeckungen für die Versicherung von Fahrraddiebstählen bzw. Überspannungsschäden definieren. Bei der Fahrraddiebstahlsdeckung soll die Versicherungssumme auf 1% der im Vertrag vereinbarten Versicherung und maximal 3000 Euro begrenzt sein. Bei der Überspannungsdeckung sollen es 5% ohne Deckelung sein.

Legen sie also zwei Produktbausteine „HRD-Fahrraddiebstahl“ und „HRD-Überspannung“ basierend auf der Vorlage „HausratZusatzdeckungstyp“ im Ordner „Deckungstypen“ an und tragen im Editor die entsprechenden Werte ein (5% werden als 0.05 eingegeben).

Nun müssen wir die Deckungstypen noch dem Produkt HR-Optimal zuordnen. Dies kann man bequem per Drag&Drop aus dem Modellexplorer erledigen. Öffnen sie das Produkt „HR-Optimal“. Ziehen sie die „HRD-Grunddeckung-Optimal“ aus dem Produktdefinitions-Explorer auf den Knoten „HausratGrunddeckungstyp“ im Bereich „Relations“. Markieren sie nun die neue Beziehung und wählen als Art der Beziehung obligatorisch (Englisch: mandatory).

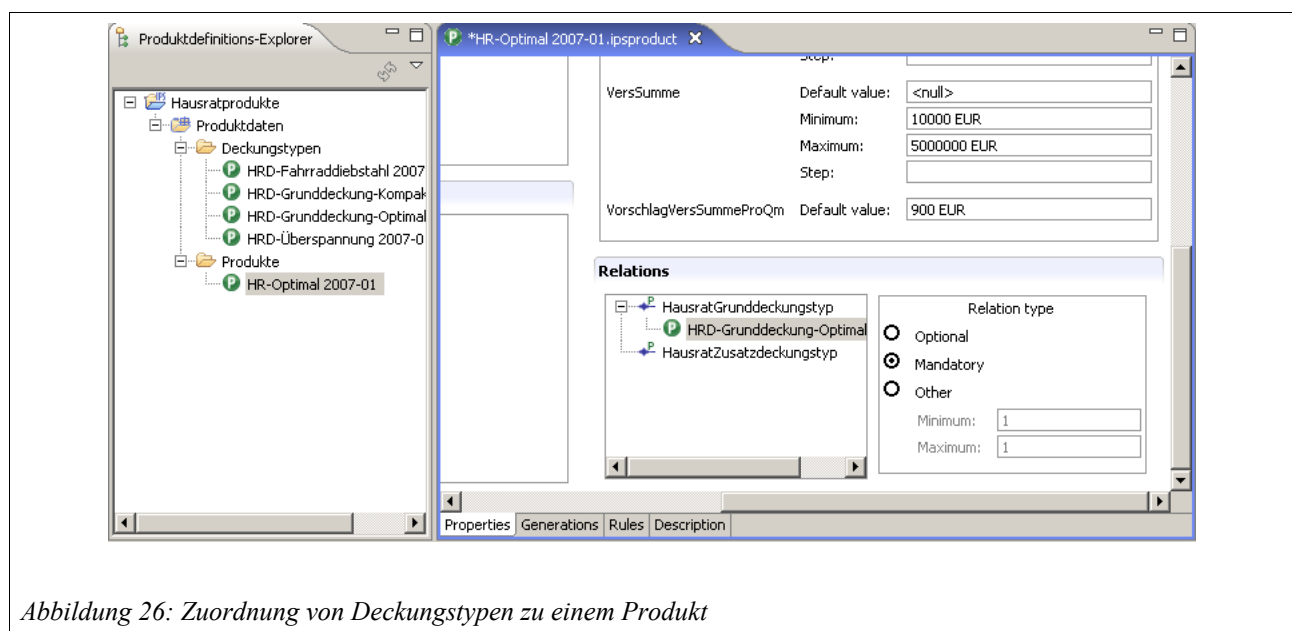


Abbildung 26: Zuordnung von Deckungstypen zu einem Produkt

Analog ziehen sie die Fahrraddiebstahl- und Überspannungsdeckung auf den Knoten Zusatzdeckungstypen. Im Produkt HR-Optimal sollen auch diese immer enthalten sein, die Art der Beziehungen ist also auch jeweils obligatorisch. Speichern sie das Produkt.

Nun legen wir noch das Produkt HR-Kompakt an. Dies können wir per Copy&Paste im Produktdefinitionsexplorer durchführen. Markieren sie also zunächst das Produkt „HR-Optimal“, kopieren es in die Zwischenablage (über das Kontextmenü oder *CTRL-C*) und fügen es wieder ein

(Kontextmenü oder *CTRL-V*). In den Dialog zur Vermeidung des Namenkonfliktes ersetzen sie „Optimal“ durch „Kompakt“. Öffnen sie nun das Produkt und geben seine Daten ein.

Konfigurationsmöglichkeit	HR-Kompakt
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

Die Beziehung zum Grunddeckungstyp von Optimal entfernen sie und ersetzen ihn durch den für Kompakt. Die Zusatzdeckungstypen sollen in dem Kompaktprodukt optional hinzugewählt werden können. Ändern sie also die Art der Beziehung entsprechend.

Damit ist die Definition der beiden Produkte zunächst abgeschlossen. Im Produktexplorer sollten sie wie nun folgt angezeigt werden.

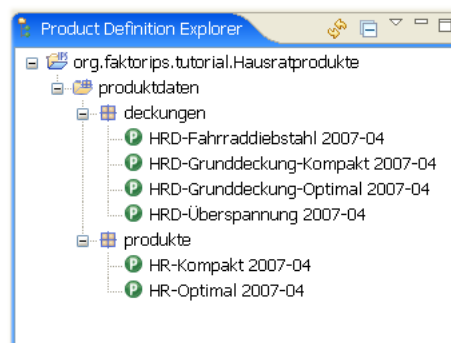


Abbildung 27: Darstellung der Produkte im Produktexplorer

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“.

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.

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.

FaktorIPS 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. Im Unterschied zu relationalen Datenbanken kann es dabei zu einer Tabellenstruktur mehrere Tabelleninhalte geben. Dies werden wir nutzen, um für die einzelnen Produktversionen unterschiedliche Tabelleninhalte anzulegen. 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.

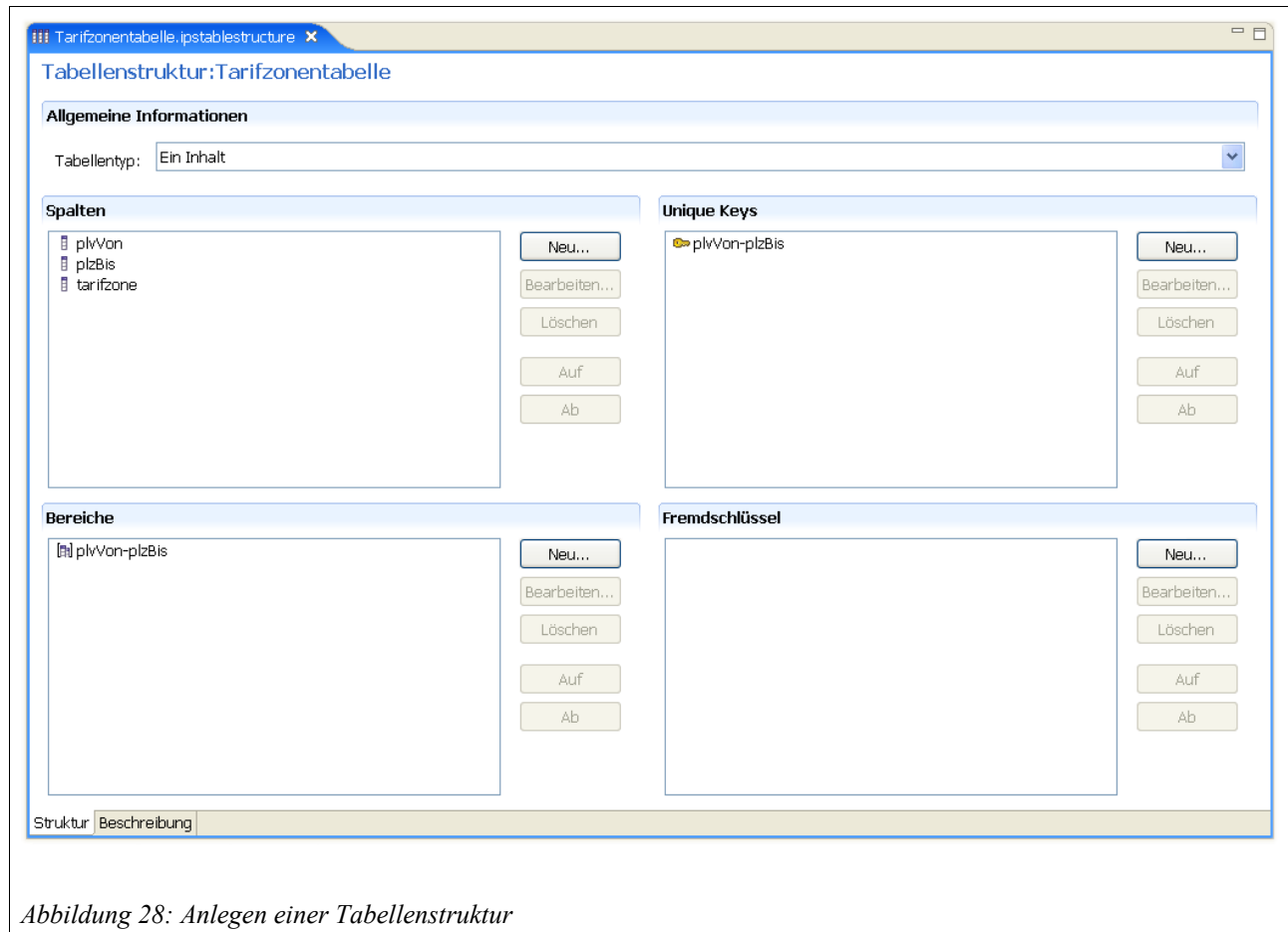
Legen wir zunächst die Tabellenstruktur für die Ermittlung der Tarifzonen an. Hierzu wechseln sie zunächst zurück in die Java-Perspektive. Im Projekt Hausratmodell markieren sie den Ordner „hausrat“ und klicken dann in der Toolbar auf . Die Tabellenstruktur nennen sie „Tarifzontentabelle“ und klicken 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 `tarifzone→plz` abzubilden. Allein mit der Spaltendefinition und einem möglichen UniqueKey ist diese Semantik allerdings nicht abbildbar. In FaktorIPS 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 Strukturbeschreibung.



FaktorIPS 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 implementieren wir

jetzt unter der Verwendung der 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 alle Produktdaten inklusive der Tabelleninhalte gibt. An dieses kommen wir leicht über das Produkt, auf dem der Vertrag basiert²².

Zum Schluss müssen wir nun noch den Tabelleninhalt anlegen. 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. Die Projektstruktur im Produktdefinitionsexplorer sollte danach wie folgt aussehen:

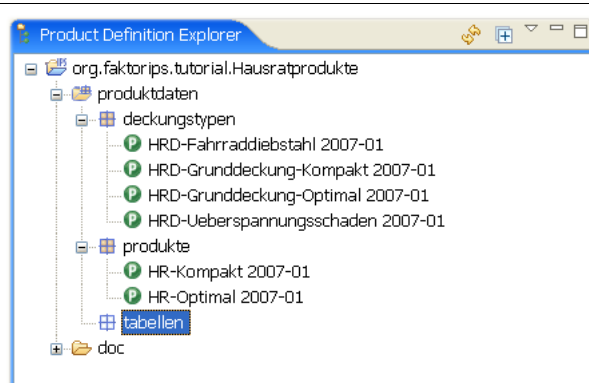



Abbildung 29: Projektstruktur der Produktdefinition

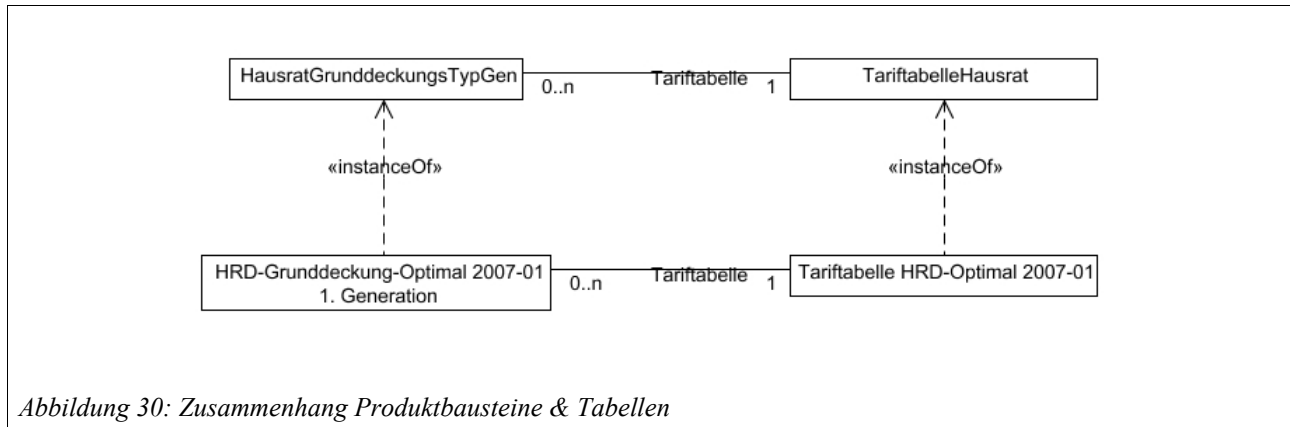
Danach markieren sie das neue Package und klicken in der Toolbar auf . Wählen sie in dem Dialog die Tarifzontabelle als Struktur aus. Als Namen für den Tabelleninhalt übernehmen sie den Namen `Tarifzontabelle` und klicken Finish. In dem Editor können sie nun die oben beispielhaft aufgeführten Zeilen erfassen.

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 UnqieKey auf die

²² Das Übergeben des `RuntimeRepository`s in die Methode `getInstance()` hat den Vorteil, dass das konkrete Repository in Testfällen leicht ausgetauscht werden kann.

Spalte Tarifzone. Als Tabellentyp wählen sie diesmal „Multiple Contents“ aus, da wir für jedes Produkt einen eigenen Tabelleninhalt anlegen wollen.

Erzeugen sie nun für die Produkte HR-Optimal und HR-Kompakt (bzw. genauer für deren Grunddeckungstypen) jeweils einen Tabelleninhalt mit dem Namen „Tariftabelle HRD-Optimal 2007_01“ und „Tariftabelle HRDKompakt 2007_01“ an²³. Das folgende UML-Diagramm zeigt diesen fachlichen Zusammenhang:



Eine Generation eines HausratGrunddeckungstyp verwendet eine TariftabelleHausrat in der Rolle Tariftabelle. Die erste Generation des Grunddeckungstyps „HRD-Grunddeckung-Optimal 200701“ verwendet den Tabelleninhalt „Tariftabelle HRD-Optimal 2007-01“.

Der Zusammenhang zwischen Tabellen und Produkten kann in FaktorIPS explizit definiert werden. Hierzu wechseln sie in den Editor für die Klasse HausratGrunddeckung. Im Abschnitt *Table Structures used for Configuration* sind die verwendeten Tabellen aufgelistete. Klicken sie auf den *New Button* neben dem Abschnitt, um eine neue Tabellenverwendung zu definieren.

²³ Die Endung „2007-01“ sollten sie dabei entsprechend des von ihnen verwendeten Wirksamkeitsdatums anpassen. Ein Vorschlag für die Tabelleninhalte finden sie im Anhang.

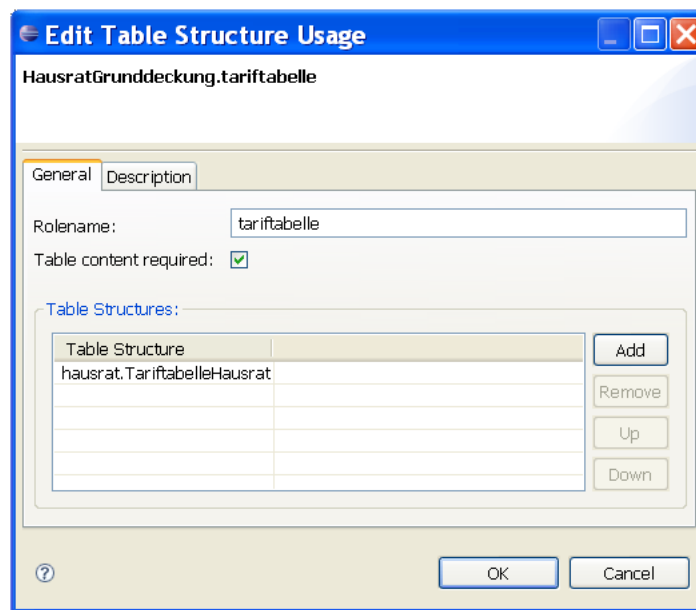


Abbildung 31: Modellierung der Verwendung von Tabellen

In dem Dialog geben sie als Rollename „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, schließen den Dialog und speichern.

Nun können wir die Tabelleninhalte den Grunddeckungstypen zuordnen. Öffnen sie zunächst „HRD-Grunddeckung-Kompakt 2007-01“. 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);
}
```

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.

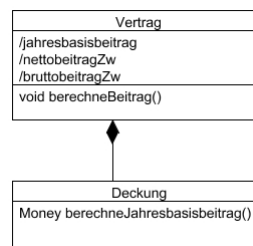


Abbildung 32: Berücksichtigung der Beitragsberechnung im spartenübergreifenden Modell

Die Deckung hat eine abstrakte Methode, um ihren Jahresbasisbeitrag zu berechnen. 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 Rate zu zahlende Nettobeitrag. Er ergibt sich aus dem Jahresbeitrag durch Zuschlag eines Ratenzahlungszuschlages bei nicht jährlicher Zahlung und Division durch die Anzahl der Raten, also zum Beispiel zwölf bei monatlicher Zahlung. Der BruttobeitragZw ist der vom Beitragszahler zu zahlende Bruttobeitrag pro Rate, 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 an. Alle Attribute sind nicht-produktrelevant, abgeleitet (cached) und vom Datentyp Money. Da es sich um gecachte abgeleitete Attribute handelt, hat FaktorIPS eine Membervariable und eine Gettermethode für sie generiert. Die Berechnung erfolgt durch die Methode `Vertrag.berechneBeitrag()`. Die Methode berechnet dabei alle Beitragsattribute des Vertrags und auch den Jahresbasisbeitrag der Deckungen. Letzteres natürlich unter Verwendung der entsprechenden Methode an der Deckung. Legen sie nun die beiden Methoden mit FaktorIPS an und speichern die beiden Klassen. Dies geschieht auf der zweiten Seite im jeweiligen Editor für Vertrag bzw. Deckung.

Die folgende Abbildung zeigt den Dialog zum Bearbeiten einer Methodensignatur.

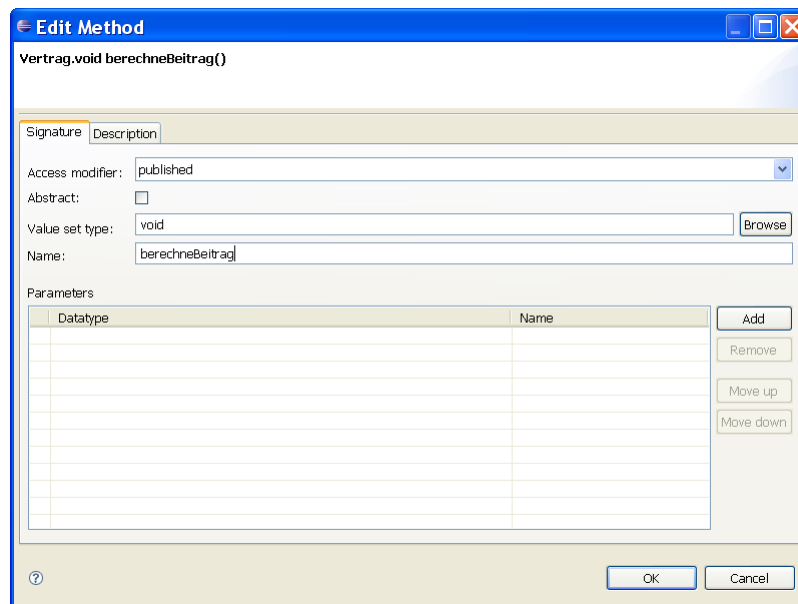


Abbildung 33: Dialog zum Bearbeiten einer Methodensignatur

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 FaktorIPS erfassen und modellinterne Methoden ausschließlich im Sourcecode definieren.

Die Berechnung des Jahresbasisbeitrags der Deckung kann natürlich nicht spartenübergreifend implementiert werden. Die abstrakte Methode ist an der Basisklasse Deckung definiert, um die spartenübergreifende Beitragsberechnung in der Klasse Vertrag realisieren zu können.

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}
 *
 * @modifiable
 */
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(1033, 2);
        // 1+ratenzahlungszuschlag=1.03 (103 Prozent)
        nettobeitragZw = jahresbasisbeitrag.multiply(rzFaktor,
                                                        BigDecimal.ROUND_HALF_UP);
    }
    nettobeitragZw = nettobeitragZw.divide(zahlweise.intValue(),
                                              BigDecimal.ROUND_HALF_UP);
}

```

Beitragsberechnung für Hausrat

Nun fehlt uns noch die Berechnung des Jahresbasisbeitrags für die Hausratdeckungen. Wir wollen sowohl bei der Grunddeckung als auch bei der Zusatzdeckung der Fachabteilung erlauben den Basisbeitrag der Deckung durch eine Formel zu definieren.

Beitragsberechnung für die Zusatzdeckungen

Beginnen wir mit den Zusatzdeckungen. Der Beitrag für eine Zusatzdeckung wird i.d.R. von der Versicherungssumme und evtl. weiteren risikorelevanten Merkmalen abhängen²⁴. Um uns nicht auf einzelne Merkmale festlegen zu müssen, stellen wir der Fachabteilung als Parameter sowohl den Hausratvertrag mit seinen Eigenschaften als auch die Zusatzdeckung zur Verfügung. Hierzu definieren wir ein neues produkt-relevantes, berechnetes Attribut „jahresbasisbeitrag“ vom Datentyp Money. Auf der Tabseite *Calculation Parameters* im Dialog zum Bearbeiten der Attribute definieren wir nun die Parameter für die Formel. Fügen sie den HausratVertrag und die HausratZusatzdeckung entsprechend der folgenden Abbildung hinzu.

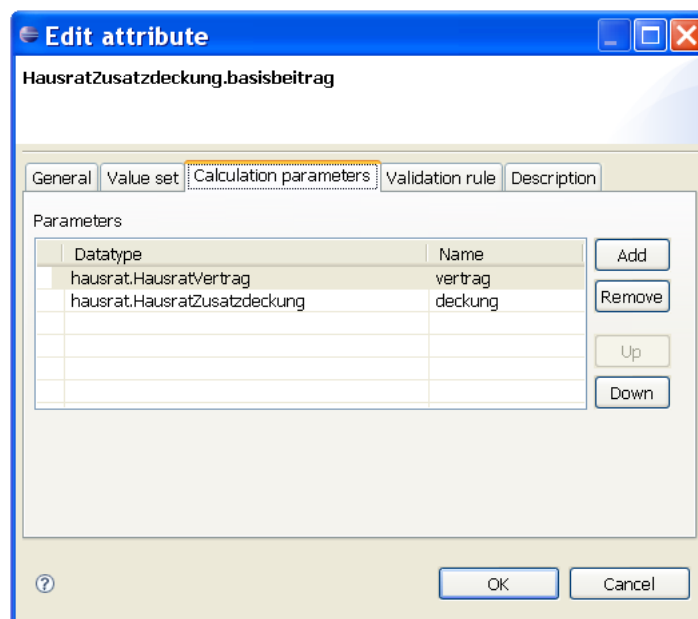


Abbildung 34: Anlegen von Parametern für eine berechnetes Attribut

Schließen sie den Dialog und speichern. Die Klasse `HausratZusatzdeckungstypGen` ist jetzt abstrakt und hat eine abstrakte Methode `computeBasisbeitrag(...)` 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. FaktorIPS generiert für jeden Produktbaustein, der eine Formel enthält, eine eigene Unterklasse²⁵. 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 Berechnungsvorschrift gibt, die es bisher nicht in der Produktdefinition gibt. Bestätigen sie mit *Fix*, dass dies behoben werden soll. In dem Abschnitt *Calculation Formulas* wird nun die noch leere Formel für den Beitragssatz angezeigt. Klicken sie auf den Button neben dem

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

²⁵ Genau genommen wird für jede Generation eine Klasse generiert, da jede Generation eine andere Formel haben kann.

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:

Datatype	Name
hausrat.HausratVertrag	vertrag
hausrat.HausratZusatzdeckung	deckung

Parameter	Value
deckung.versSumme	10000EUR

Result: 100.00 EUR

Abbildung 35: 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 dann noch einen Punkt ein. Sie bekommen die Eigenschaften der Deckung zur Auswahl angeboten. Wählen sie die „versSumme“. Nun multiplizieren sie die Versicherungssumme mit 0.1, also „deckung.versSumme * 0.1“. Sie können die Formel 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 3% der Versicherungssumme beträgt. FaktorIPS hat nun die Subklassen für die beiden Produktbausteine mit der in Java Sourcecode übersetzten Formel generiert. Sie finden die beiden Klassen im Package `org.faktorips.tutorial.modell.internal.deckungen`²⁶. Über die Methoden `computeBasisbeitrag(...)` kann die Formel aufgerufen werden.

²⁶ 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`.

Nun müssen wir noch dafür sorgen, dass die Formel im Rahmen der Beitragsberechnung auch aufgerufen wird, indem wir in der Klasse `HausratZusatzdeckung` die Methode `berechneJahresbasisbeitrag()` implementieren. Hierzu müssen wir zunächst die Methode im Modell anlegen, indem wir im Editor für die Klasse auf der 2. Seite auf *Override...* anklicken und die Methode auswählen. Der folgende Sourcecode-Ausschnitt zeigt die Implementierung der Methode.

```
public Money berechneJahresbasisbeitrag() {
    // Generation holen und auf Implementierung casten, da die compute()
    // Methode nicht zum published Interface gehoert.
    HausratZusatzdeckungstypGen gen =
        (HausratZusatzdeckungstypGen) getHausratZusatzdeckungstypGen();
    if (gen==null) {
        // wenn die Generation nicht ermittelt werden kann, kann der Beitrag
        // nicht berechnet werden
        return Money.NULL;
    }
    // Vertrag holen und auf Implementierung casten, da die compute() Methode
    // die Implementierung in der Signature verwendet, damit in der Berechnung
    // auch auf Eigenschaften zugegriffen werden kann, die nicht zum
    // published Interface gehoeren.
    HausratVertrag vertrag = (HausratVertrag) getHausratVertrag();
    // Berechnungsmethode aufrufen und Ergebnis zurueckgeben
    return gen.computeJahresbasisbeitrag(vertrag, this);
}
```

Beitragsberechnung für die Grunddeckung

Zum Schluss müssen wir noch die Beitragsberechnung für die Grunddeckung realisieren. Fachlich berechnet sich der Beitrag 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.

Öffnen sie den zunächst den Editor für die Klasse `HausratGrunddeckungstyp`. Wie für die Zusatzdeckung müssen wir noch ein neues produkt-relevantes, berechnetes Attribut „jahresbasisbeitrag“ vom Datentyp `Money` einführen. Als Parameter für die Berechnung definieren wir analog den `HausratVertrag` und die `HausratGrunddeckung`. Zudem müssen wir wieder die abstrakte Methode `berechneJahresbeitrag()` überschreiben (auf der 2. Seite im Editor *Override...* anklicken). Die Implementierung erfolgt analog zur Methode in der Klasse `HausratZusatzdeckung`.

Danach öffnen sie die Grunddeckung für HR-Optimal und bestätigen, dass die Formel für den Jahresbasisbeitrag neu angelegt werden soll. In dem Abschnitt *Calculation Formulas and Tables* haben wir im vorherigen Kapitel bereits die zu verwendende Tariftabelle zugeordnet. Auf diese müssen wir nun in der Formel zugreifen. Hierzu öffnen sie zunächst den Dialog zur Bearbeitung der Formel. Wenn sie jetzt *Strg-Space* drücken, sehen sie auch die Funktion `Tariftabelle.beitragssatz(String tarifzone)` mit der sie auf die Tariftabelle zugreifen können. Als Parameter für den Zugriff wird die Tarifzone des Vertrags benötigt. Das Ergebnis ist der Beitragsatz für 1000 EUR und muss noch mit der Versicherungssumme des Vertrags multipliziert

werden und dann durch 1000 geteilt werden. Insgesamt ergibt sich also für die Beitragsberechnung die folgenden Formel :

$$\text{Tariftabelle.beitragssatz(vertrag.tarifzone)} * \text{vertrag.versSumme} / 1000$$

Natürlich können sie auch diese Formel wieder direkt testen. Für 10.000EUR Versicherungssumme und Tarifzone III sollten 14,40Euro herauskommen, wenn sie die Daten aus dem Anhang verwenden.


Schließen sie nun den Dialog und speichern den Produktbaustein. Dann definieren sie analog die Formel für die Grunddeckung von HR-Kompakt.

Zur Berechnung des Beitrags für die Grunddeckung haben wir in diesem Tutorial eine Formel verwendet, um den Zugriff auf Tabellen aus Formeln heraus zu demonstrieren. Alternativ kann die Beitragsberechnung natürlich auch direkt im Java Sourcecode implementiert werden und zwar in der Methode `berechneJahresbeitrag()`. In der Praxis verwendet man Formeln, um der Fachabteilung Flexibilität bei der Definition neuer Produkte zu geben. Ist diese Flexibilität nicht erforderlich oder gewünscht, da Konfigurationsmöglichkeiten wie Bausteineigenschaften und Tabelleninhalte ausreichen, verzichtet man auf sie.

Schreiben eines JUnit-Tests für die Beitragsberechnung

Nachdem wir nun die Hausratversicherung modelliert, die Produkte erfasst und die Beitragsberechnung implementiert haben, wollen wir zum Abschluss des Tutorials die Beitragsberechnung auch wirklich ausführen. Hierzu schreiben wir einen einfachen JUnit-Test²⁷, in welchem wir einen Hausratvertrag programmatisch erzeugen und die Berechnung ausführen. An dem Beispiel sollte deutlich werden, wie die Modellobjekte von Clients (UI, Batch, Tests, Webservices) genutzt werden.

Legen sie 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. Eclipse nimmt nun zunächst die JUnit Library in den Classpath des Projektes auf (Rückfrage mit „Yes“ bestätigen). In dem Dialog geben sie als Namen für die Testfallklasse „BeitragsberechnungTest“ 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.

Der folgende Sourcecode zeigt den Testfallklasse. Erläuterung bedarf noch die `setUp()` Methode. Zur Laufzeit befinden sich die Produktdaten, also Produktbausteine und Tabellen im sogenannten `RuntimeRepository`. Welche Daten sich im `RuntimeRepository` befinden ist in einem Inhaltsverzeichnis vermerkt. Diese Datei heißt standardmäßig „faktorips-repository-toc.xml“ und wird von FaktorIPS generiert. Im Projekt Hausratprodukte befindet es sich im Java Package `org.faktorips.tutorial.produktdaten.internal`.

In der `create(...)` Methode des `ClassLoaderRuntimeRepository` muss man den Pfad zu dieser Datei übergeben. Beim Erzeugen wird des Repositories wird das Inhaltsverzeichnis 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 plattformunabhängig ist. So kann zum Beispiel die Produktlogik ohne auch direkt unter CICS verwendet werden, ohne das Anpassungen erforderlich sind.

²⁷ <http://www.junit.org>

```
public class BerechnungsTest extends TestCase {

    private IRuntimeRepository repository;

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

    public void test() {
        // Ermitteln des Hausratproduktes HR-Kompakt 2007-01 aus dem Repository
        // "hausrat.HP-Kompakt 2007-01" ist die RuntimeId des Produktes
        IHausratProdukt produkt = (IHausratProdukt)
            repository.getProductComponent("hausrat.HR-Kompakt 2007-01");

        // Erzeugen eines Vertrags auf Basis des Hausratproduktes ueber die
        // Factorymethode am Produkt.
        IHausratVertrag vertrag = produkt.createHausratVertrag();

        // Vertragsattribute setzen
        GregorianCalendar wirksamAb = new GregorianCalendar(2007, 4, 1);
        vertrag.setWirksamAb(wirksamAb);
        vertrag.setPlz("17236");
        vertrag.setVersSumme(Money.euro(60000));
        vertrag.setWohnflaeche(new Integer(100));
        vertrag.setZahlweise(new Integer(2)); // halbjaehrlich

        // Generation des Produktes ermitteln, die zum WirksamAb gueltig ist.
        IHausratProduktGen produktGen = produkt.getHausratProduktGen(wirksamAb);

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

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

        // Vertrag berechnen
        vertrag.berechneBeitrag();

        // Ergebniss pruefen (3 Prozent Ratenzahlungszuschlag=72Cent!)
        assertEquals(Money.euro(48, 0), deckung.getJahresbasisbeitrag());
        assertEquals(Money.euro(48, 0), vertrag.getJahresbasisbeitrag());
        assertEquals(Money.euro(24, 72), vertrag.getNettobeitragZw());
        assertEquals(Money.euro(29, 42), vertrag.getBruttobeitragZw());
    }
}
```

Schlussbemerkungen

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

- FaktorIPS 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.
- Die Modellierung und Codegenerierung ist so in Eclipse integriert, dass für Softwareentwickler kein großer Einarbeitungsaufwand anfällt. Für die Entwicklung werden neben FaktorIPS 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?

FaktorIPS unterstützt explizit die Modellierung von Prüfungen und Auskunftsinhalten. Einen Teil der zu Auskunftszwecken generierten Methoden haben wir im Zusammenhang mit den Produktklassen kennengelernt.

Testunterstützung

FaktorIPS 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 Cruisecontrol²⁸ integrieren.

Darüber hinaus unterstützt FaktorIPS das Testen des Modells unabhängig von konkreten Produktdaten. 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

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

Produkten HR-Optimal und HR-Kompakt nicht getestet werden, da der Wert auf 600 bzw. 900 Euro festgelegt 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 FaktorIPS seine Daten Eclipse-konform in Dateien im Eclipse-Workspace ablegt, können sie wie alle anderen Dateien versioniert werden. Es gibt keinen Unterschied in der Handhabung von FaktorIPS Dateien und anderen Dateien. 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 Javaklassen²⁹, 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 zumindestens 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 RemoteFacade 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 FaktorIPS 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.

²⁹ Auch häufig als Plain old Java objects oder kurz POJOs bezeichnet.

Anhang: Tariftabellen

HR-Optimal

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

HR-Kompakt

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