Partitionierung von Modellen mit Faktor-IPS

Gunnar Tacke
(Dokumentversion 16)

Überblick

Im Faktor-IPS Einführungstutorial haben wir das Modell für eine einfache Hausratversicherung entworfen. In der Praxis sind die fachlichen Modelle natürlich wesentlich komplexer, insbesondere wenn mehrere Sparten abgebildet werden. In diesem Artikel wollen wir daher auf die Partitionierung von Modellen eingehen. Unter Modellpartitionierung verstehen wir dabei die Zerlegung eines komplexen Gesamtmodells in kleinere Teilmodelle.

Grundmodell und spartenspezifische Modelle

Bei der Abbildung des Produktspektrums eines Versicherungsunternehmens bietet sich die Differenzierung nach fachlichen Sparten an. Prozesse und Produkte sind innerhalb der Sparten ähnlich, können aber zwischen den Sparten sehr unterschiedlich sein. Man findet diese Differenzierung auch in den Organisationsstrukturen der Versicherungsunternehmen wieder. Für die Modellierung von Verträgen und Produkten liegt es daher auch nahe, Teilmodelle für unterschiedliche Sparten zu bilden.

Das im Einführungstutorial von Faktor-IPS erstellte Modell dient der Abbildung von Verträgen und Produkten der Sparte Hausrat. Hausratverträge haben u.a. Eigenschaften wie Wirksamkeitsdatum, Zahlweise und Beitrag und können Deckungen wie Grunddeckung und Zusatzdeckungen enthalten.

Wenn wir uns nun überlegen, auch ein Modell für Verträge und Produkte der Sparte Kfz zu entwerfen, werden wir feststellen, das Kfz-Verträge auch Eigenschaften wie Wirksamkeitsdatum, Zahlweise und Beitrag besitzen und auch mehrere Deckungen enthalten können. Und auch bei der Betrachtung weiterer Sparten werden sich diese spartenübergreifenden Gemeinsamkeiten finden. Anstatt nun zum Beispiel die Zahlweise jeweils als eigenes Attribut der Klassen Hausratvertrag, Kfzvertrag, Unfallvertrag (...) zu modellieren, bietet es sich an, diese Gemeinsamkeiten einheitlich in einem spartenübergreifendem Teilmodell (Grundmodell) abzubilden, dass den spartenspezifischen Teilmodellen als Basis dient.

Die folgende Abbildung illustriert diese Trennung in spartenübergreifende und spartenspezifische Aspekte anhand des Beispiels aus dem Einführungstutorial. Um die Wiederverwendbarkeit des Grundmodells zu verdeutlichen, wurde das Beispiel hier um ein (fachlich ebenfalls stark vereinfachtes) Kfz-Modell erweitert.

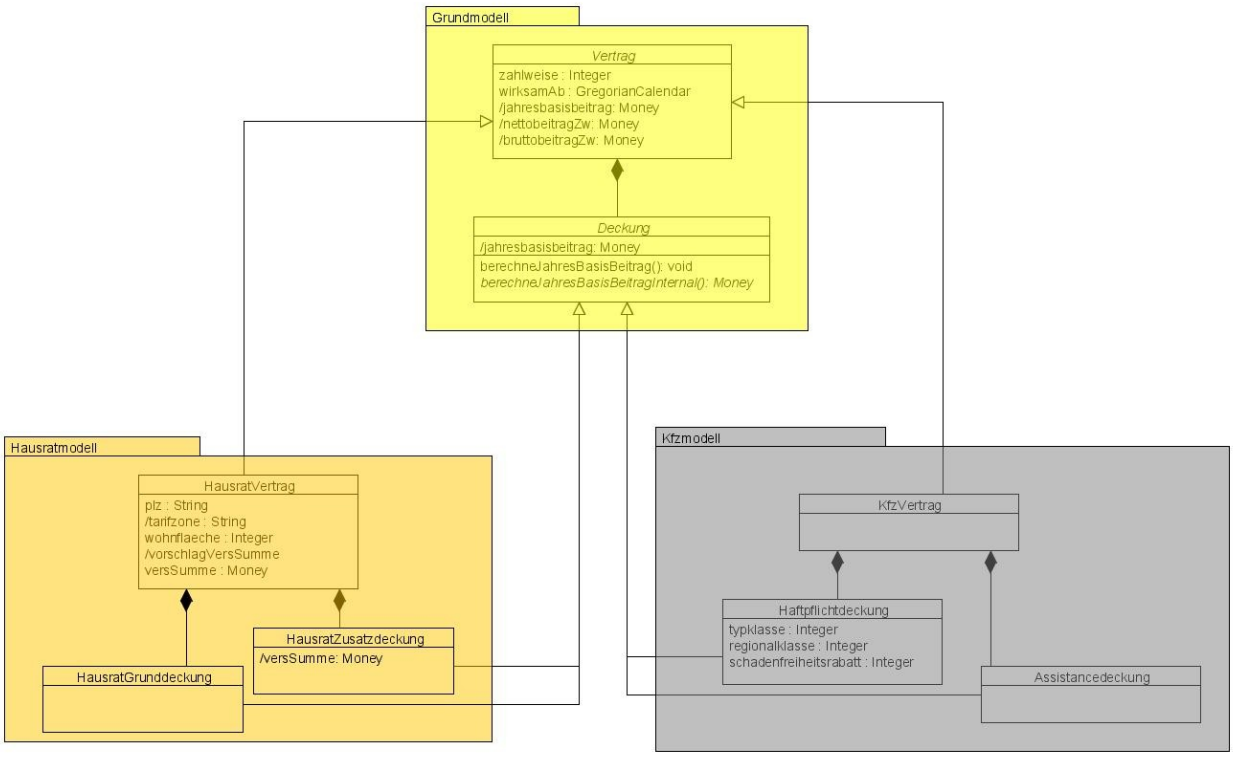

Abbildung 1: Spartenübergreifendes Modell und Spartenmodelle

Spartenübergreifende Attribute wie Zahlweise und Jahresbasisbeitrag sind dabei in entsprechenden Basisklassen Vertrag und Deckung verschoben worden. Ebenso ist die Beziehung zwischen Vertrag und Deckung Teil des spartenübergreifenden (Teil-)Modells. Spartenübergreifende Logik, wie zum Beispiel die Ermittlung des Jahresbeitrags für alle Deckungen eines Vertrages ist im Grundmodell abgebildet. Die Berechnung des Beitrags für die einzelne Deckung wird aber spartenspezifisch in den jeweiligen Subklassen der Spartenmodelle implementiert.

Umsetzung in Faktor-IPS

Im folgenden sehen wir, wie wir diese Trennung in Faktor-IPS abbilden. Auf die grundsätzlichen Konzepte und Bedienung von Faktor-IPS wird dabei nicht näher eingegangen; diese sind im Einführungstutorial zu Faktor-IPS erläutert.

Grundmodell anlegen

Für das Grundmodell legen wir ein eigenes Eclipse-Projekt org.faktorips.tutorial.de.Grundmodell an. Wir modellieren es gemäß dem folgendem Diagramm. Links sehen wir die Vertragsteilklassen (PolicyComponentTypes), auf der rechten Seite die Produktbausteinklassen (ProductComponent-Types).

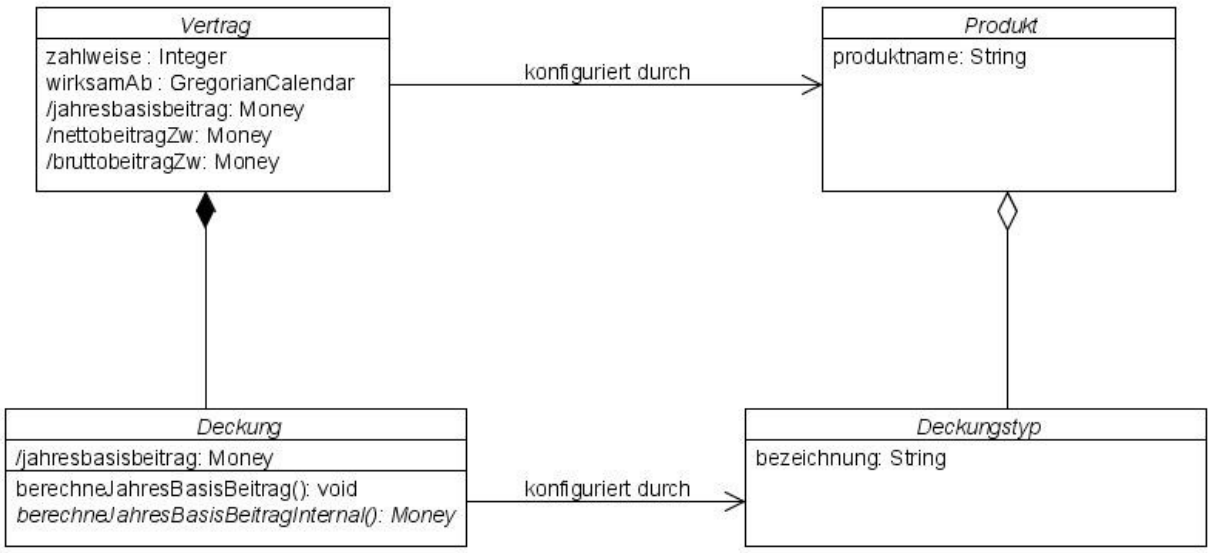

Abbildung 2: Grundmodell

Spartenmodelle anlegen

Für die Sparten Hausrat und Kfz legen wir jeweils eigene Projekte an, in denen wir die spartenspezifischen Eigenschaften modellieren. Exemplarisch dargestellt wird es im folgendem am Spartenmodell für Hausrat.

Wir legen das Eclipse-Project org.faktorips.tutorial.de.Hausratmodell an. Um im Hausratmodell auf das Grundmodell zugreifen zu können, müssen wir das Projekt mit dem Grundmodell in den IPS-Build-Path des Hausratmodells aufnehmen. Hierzu öffnen wir die Eigenschaften des Projekts und fügen unter dem Eintrag IPS-Build Path auf der Lasche Projekte das Projekt org.faktorips.tutorial.de.Grundmodell dem IPS-Build Path hinzu.

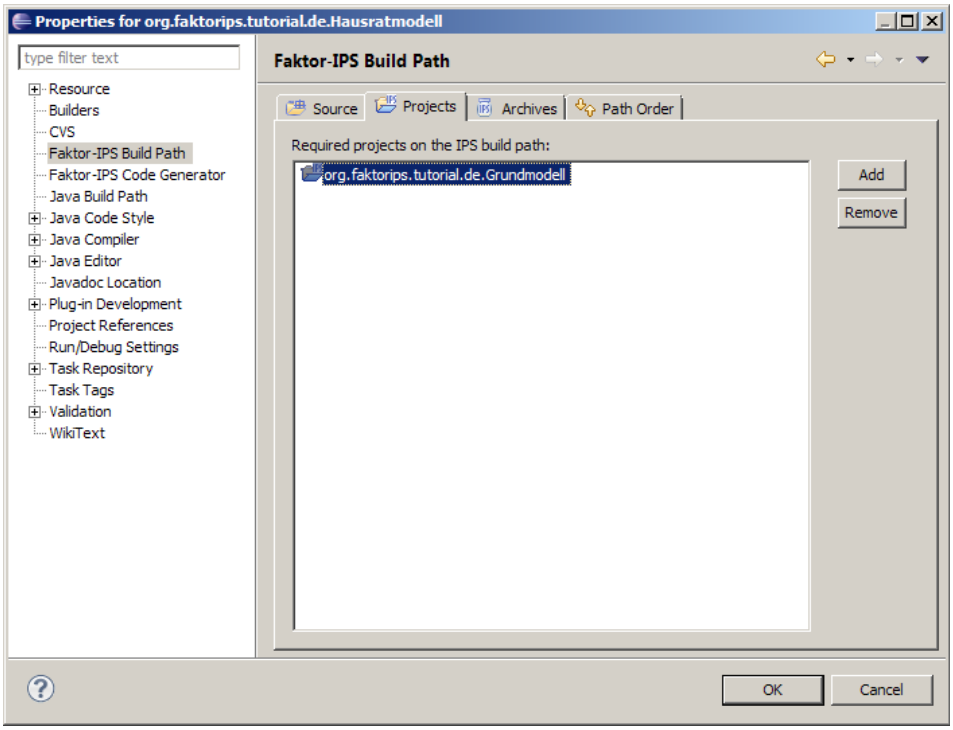

Abbildung 3: IPS-Build Path bearbeiten

Die gleiche Referenz müssen wir auch für den Java-Build-Path herstellen, damit im generierten Code des Hausratmodells auch auf den Code des Grundmodells zugegriffen werden kann. Wir nehmen also das Grundmodell in den Java Build Path des Hausratmodells auf[1].

Das Hausratmodell wird gemäß dem folgendem UML-Diagramm modelliert, wobei die Klassen auf der linken Seite als Vertragsteilklassen und die auf der rechten Seite als Produktbausteinklassen modelliert werden:

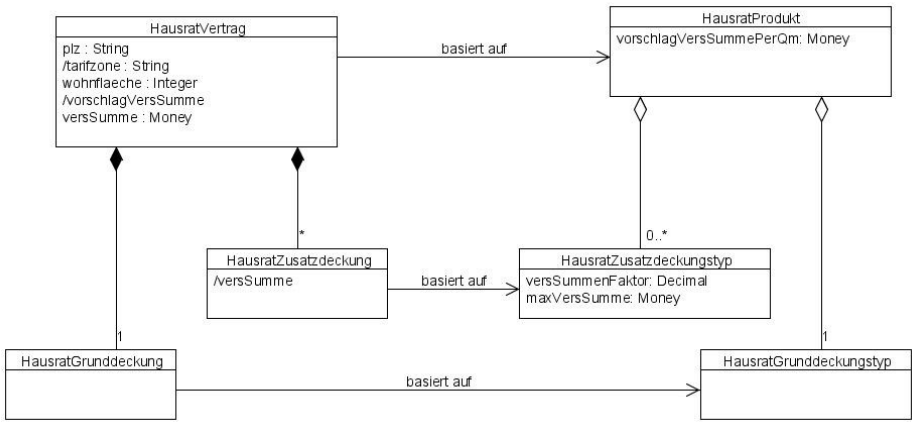

Abbildung 4: Hausratmodell

1 In den Projekten auf www.faktorzehn.org finden sich die Classpath-Beziehungen als PlugIn-Dependencies, um die Projekte als PlugIns verwenden zu können.

Vererbung vom Spartenübergreifenden Modell zum Hausratmodell

Nun soll der Hausratvertrag auch die Eigenschaften der Klasse Vertrag erben (z.B. das Attribut zahlweise), hierzu können wir in Faktor-IPS eine Vererbungsbeziehung im OO-Sinne herstellen. Hierzu öffnen wir die Vertragsteilklasse Hausratvertrag und tragen Vertrag als Superklasse ein:

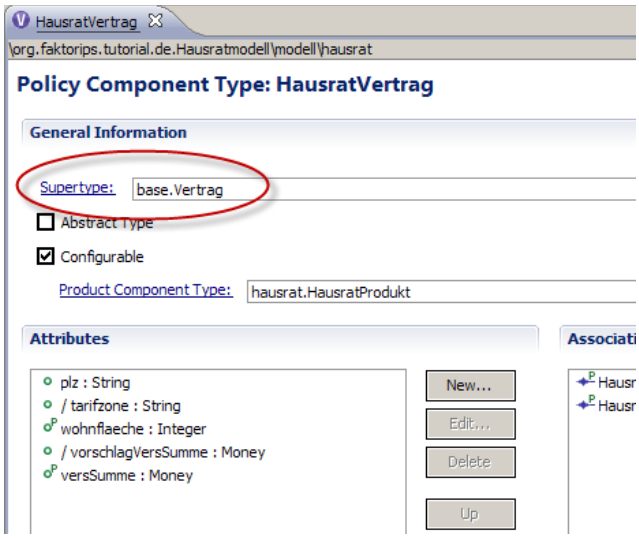

Abbildung 5: Vererbung Vertrag - Hausratvertrag

Weitere Vererbungsbeziehungen zwischen Klassen des Grund- und spartenspezifischen Modells werden gemäß des folgenden UML-Diagramms angelegt:

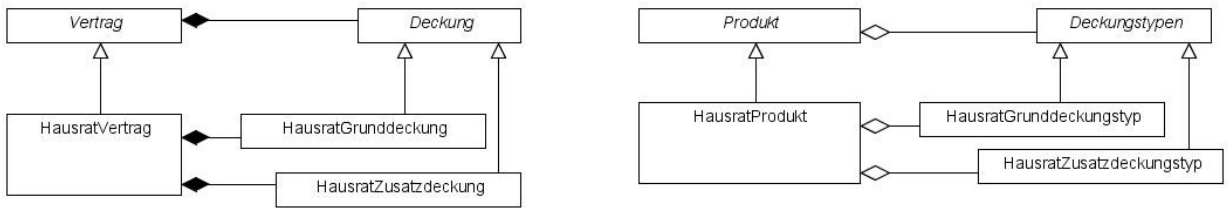

Abbildung 6: Vererbungsbeziehungen zwischen Grund- und Hausratmodell

Umgang mit Beziehungen: Derived Union

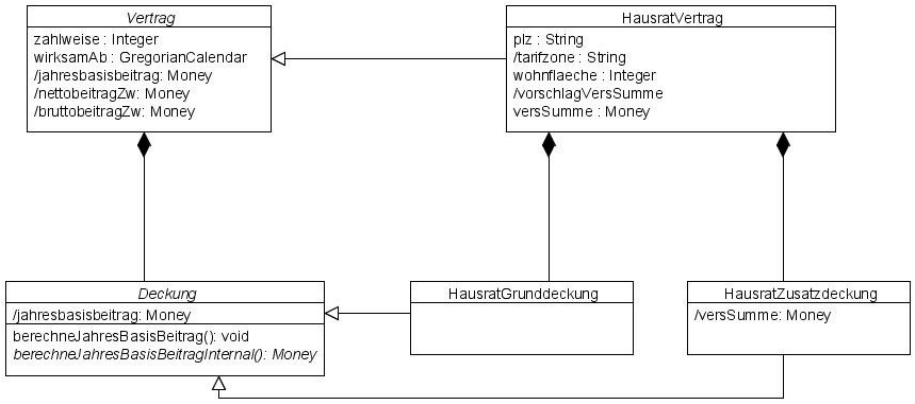

Abbildung 7: Kompositbeziehungen in Grundmodell und Hausratmodell

Betrachten wir noch einmal das UML Modell mit den Vererbungsbeziehungen zwischen Grundmodell und Hausratmodell (Abbildung 7). Es gibt eine Kompositbeziehung zwischen den spartenübergreifenden Klassen *Vertrag* und *Deckung* einerseits und zwischen dem *Hausratvertrag* und der *Hausratzusatzdeckung* bzw. -*grunddeckung* andererseits. Unsere Erwartungshaltung ist dabei die folgende:

- Zum Hausratvertrag können eine Hausratgrunddeckung und beliebig viele Hausrat-zusatzdeckungen hinzugefügt werden, aber zum Beispiel keine KFZ-Deckungen.
- Wenn man eine Hausratgrunddeckung oder Zusatzdeckung zum Hausratvertrag hinzufügt, erhält man diese auch zurück, wenn man sich vom Vertrag alle Deckungen (also die, deren Typen Deckung oder Subklassen von Deckung sind) zurückgeben lässt.

Diese erwartete Semantik ist aber nicht im Modell enthalten. Die Beziehungen zwischen *Vertrag* und *Deckung* bzw. *Hausratvertrag* und *Hausratgrunddeckung/-Zusatzdeckung* sind zunächst voneinander unabhängige Beziehungen. Ein Beispiel verdeutlicht dies:

In dem folgenden Codeausschnitt erzeugen wir eine Instanz von `HausratVertrag` und eine Instanz von `HausratZusatzdeckung` und fügen mit der Methode `addHausratZusatzdeckung(...)` die Zusatzdeckung dem Hausratvertrag hinzu. Fragen wir nun den Vertrag mit `getDeckungen()` nach seinen Deckungen, erhalten wir eine leere Liste zurück. Fragen wir den Hausratvertrag mit `getHausratZusatzdeckungen()` nach seinen Zusatzdeckungen, bekommen wir dagegen erwartungsgemäß eine Liste mit der Zusatzdeckung zurück.

```
HausratVertrag hausratVertrag = new HausratVertrag();
HausratZusatzdeckung neueZusatzdeckung = new HausratZusatzdeckung();
hausratVertrag.addHausratZusatzdeckung(neueZusatzdeckung);

hausratVertrag.getDeckungen(); /*liefert null*/
hausratVertrag.getHausratZusatzdeckungen(); /*liefert eine ArrayList mit neueZusatzdeckung als
Element*/
```

Wenn wir uns den Code anschauen, der aus dem Modell generiert wird, verstehen wir, warum. Die Komposit-Beziehungen zu *Deckung*, *HausratGrunddeckung* und *Hausratzusatzdeckungen* werden in Java jeweils als eigene Membervariablen abgebildet:

```
public abstract class Vertrag /* ... */ {

    private List<IDeckung> deckungen = new ArrayList<IDeckung>();

    public List<IDeckung> getDeckungen() {
        return Collections.unmodifiableList(deckungen);
    }
    // ...
}
```

```
public class HausratVertrag extends Vertrag /* ... */  {

    private HausratGrunddeckung hausratGrunddeckung = null;
    private List<IHausratZusatzdeckung> hausratZusatzdeckungen =
                                                        new ArrayList<IHausratZusatzdeckung>();

    public IHausratGrunddeckung getHausratGrunddeckung() {
        return hausratGrunddeckung;
    }

    public List<IHausratZusatzdeckung> getHausratZusatzdeckungen() {
        return Collections.unmodifiableList(hausratZusatzdeckungen);
    }
    // ...
}
```

Die von uns gewünschte Semantik können wir erreichen, indem wir die Deckungen nicht in einer Membervariablen des Vertrags verwalten, sondern die Deckungen eines Vertrags aus den anderen beiden Beziehungen ermitteln. Der folgende Codeausschnitt zeigt, wie dies gehen kann:

```
public List<IDeckung> getDeckungen() {
    List<IDeckung> result = new ArrayList<IDeckung>(getNumOfDeckungenInternal());
    // wenn es eine Grunddeckung gibt, fuege diese zu den Deckungen hinzu
    if (getHausratGrunddeckung() != null) {
        result.add(getHausratGrunddeckung());
    }
    // nun fuege alle Zusatzdeckungen hinzu
    result.addAll(getHausratZusatzdeckungen());
    return result;
}
```

Damit wird die Beziehung *Vertrag-Deckung* zu einer abgeleiteten Beziehung. Sie entspricht der Obermenge der beiden anderen Beziehungen entspricht. Solche Typen von Beziehungen werden in UML auch als „derived union“ bezeichnet. Das folgende Klassendiagramm zeigt das Modell unter Verwendung der abgeleiteten Vereinigung in UML Notation:

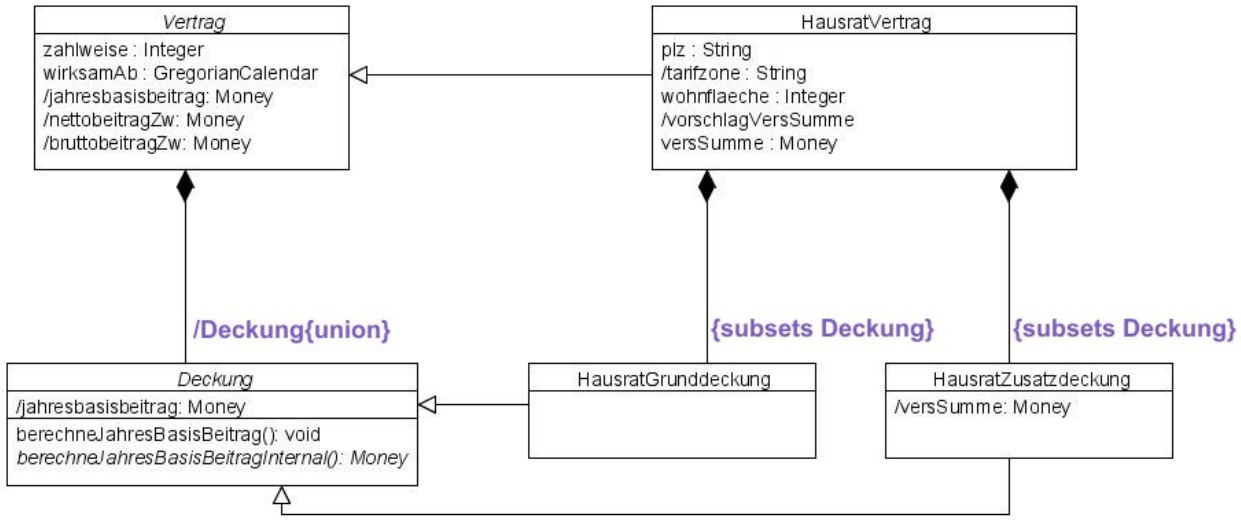

Abbildung 8: Abgeleitete Vereinigung Deckung

Verbleibendes Problem in unserer Implementierung ist nun noch, dass die Methode `getDeckungen()` so nur im Hausratmodell in der Subklasse `HausratVertrag` implementiert werden kann, da auf die hausratspezifischen Klassen `Hausratgrunddeckung` und `Hausratzusatzdeckung` zugegriffen wird. Wie kann nun einen Vertrag spartenunabhängig nach seinen Deckungen fragen? Dies lösen wir, indem wir in das Interface `IVertrag` die Methode `getDeckungen()` aufnehmen, die Klasse Vertrag als abstrakt markieren und die Methode erst in den spartenspezifischen Subklassen von Vertrag implementieren.

Einfacher geht es mit Faktor-IPS: Um die Beziehung *Vertrag-Deckung* als abgeleitete Vereinigung zu markieren, öffnen Sie den Editor für die Klasse *Vertrag* und dann den Dialog zur Bearbeitung der Beziehung. In dem Dialog haken Sie die Checkbox gemäß der folgenden Abbildung an.

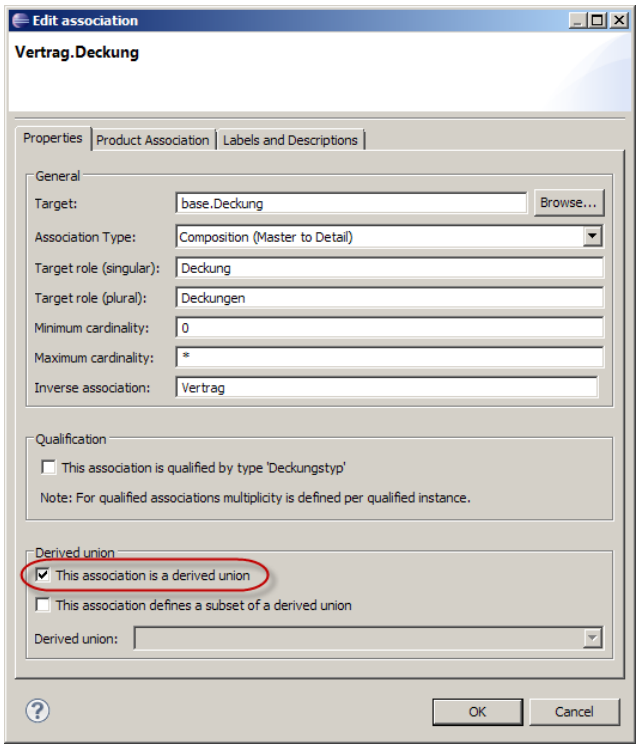

Abbildung 9: Definieren einer abgeleiteten Vereinigung

Wenn wir jetzt die Beziehung zwischen *Hausratvertrag* und *Hausratgrundeckung* bearbeiten, um diese als Teilmenge der derived union zu markieren, belegt der Assistent automatisch vor, dass die neue Beziehungen eine Teilmenge der abgeleiteten Vereinigungsbeziehung „Deckung“ ist.

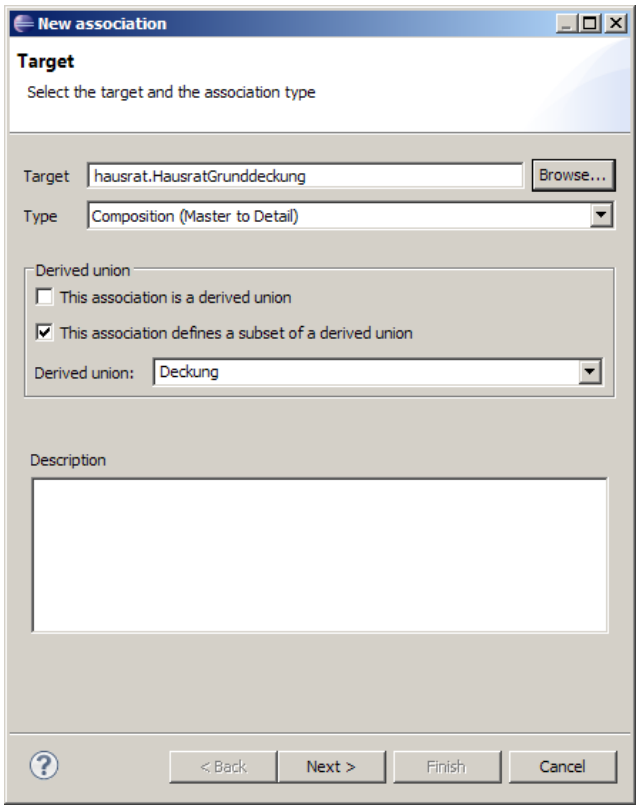

Abbildung 10: Beziehung als Teilmenge der abgeleiteten Vereinigung definieren

Der von Faktor-IPS generierte Sourcecode entspricht nun genau unserer ursprünglichen Implementierung. Wenn wir jetzt wieder unseren initialen Beispielcodeausschnitt anschauen und den `hausratVertrag` mit `getDeckungen()` nach seinen Deckungen fragen, enthält die zurückgegebene Liste auch die Instanz `neueZusatzdeckung`:

```
HausratVertrag hausratVertrag = new HausratVertrag();
HausratZusatzdeckung neueZusatzdeckung = new HausratZusatzdeckung();
hausratVertrag.addHausratZusatzdeckung(neueZusatzdeckung);

hausratVertrag.getDeckungen(); /*liefert eine ArrayList mit neueZusatzdeckung als Element*/

hausratVertrag.getHausratZusatzdeckungen(); /*liefert eine ArrayList mit neueZusatzdeckung als
Element*/
```

Verwendung der Derived Union Beziehung für spartenübergreifende Beitragsberechnung

Da wir nun in der spartenübergreifenden Klasse `Vertrag` auf alle Deckungen zugreifen können, können wir auch Teile der Beitragsberechnung spartenübergreifend implementieren. Der Jahresbasisbeitrag des Vertrags entspricht immer der Summe der Beiträge seiner Deckungen. Die Berechnung des Beitrags der Deckungen ist dann natürlich wieder sparten- und deckungsspezifisch. Um im spartenübergreifenden Modell die Beitragsberechnung von Deckungen aufrufen zu können, definieren wir in der Klasse `Deckung` die (abstrakte) Templatemethode

`berechneJahresbasisbeitragInternal()` wie folgt:

```
public void berechneJahresbasisbeitrag() {
       jahresbasisbeitrag = berechneJahresbasisbeitragInternal();
}

public abstract Money berechneJahresbasisbeitragInternal();
```

Nun können wir die Berechnung des Jahresbasisbeitrag im `Vertrag` wie folgt implementieren.

```
private void berechneJahresbasisbeitrag() {
    jahresbasisbeitrag = Money.euro(0, 0);
    List<IDeckung> deckungen = getDeckungen();
    for (int i = 0; i < deckungen.size(); i++) {
        IDeckung deckung = deckungen.get(i);
        deckung.berechneJahresbasisbeitrag();
        jahresbasisbeitrag = jahresbasisbeitrag.add(deckung.getJahresbasisbeitrag());
    }
}
```

Für die `HausratGrunddeckung` sieht die Implementierung der Methode `berechneJahresbasisbeitragInternal()` beispielsweise so aus:

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

Geeignete Projektstruktur unterstützt Aufgabenteilung im Entwicklungsprozess

Analog zum Hausratmodell können Teilmodelle für weitere Sparten angelegt werden. Diese spartenspezifischen Teilmodelle haben i.d.R. keine Abhängigkeit untereinander, so dass sie unabhängig voneinander entwickelt werden können. Damit sie auch im Konfigurationsmanagement sauber getrennt sind, legt man für jedes Teilmodell ein eigenes Projekt an. Durch die Aufteilung in unterschiedliche Projekte können einzelne Sparten so unabhängig voneinander entwickelt, versioniert und freigegeben werden.

Des weiteren hat es sich als hilfreich erwiesen, das Modell und die Produktdaten nicht in einem einzelnen Eclipse Projekt zu verwalten, sondern in zwei getrennt Projekten. Auf diese Weise kann eine Fachabteilung die Produktdaten unabhängig von der Anwendungsentwicklung definieren.

Insgesamt ergibt sich damit folgende beispielhafte Projektstruktur:

Projekt	***Inhalt***
Grundmodell	Spartenübergreifender Teil des Modells.
Grunddaten	Spartenübergreifende Produtkdaten wie z. B. Ratenzahlungszuschläge.
Hausratmodell	Hausratspezifischer Teil des Modells
Hausratprodukte	Produktdaten Hausrat.
KfzModell	KFZ-spezifischer Teil des Modells
KfzProdukte	Produktdaten KFZ.
...	Modelle / Produktdaten für weitere Sparten

Beispielprojekt zum Download

Die hier gezeigte Partitionierung des Hausratmodells in Grundmodell und Hausratmodell, steht als Beispielimplementierung zur Verfügung und kann auf www.faktorzehn.org heruntergeladen werden.

Des weiteren gibt es ein Tutorial, dass auf diesem Modell aufsetzt und zeigt, wie man mit dem Modell in einer operativen Anwendung arbeitet (Tutorial Angebotssystem).