

Softwaretests mit Faktor-IPS

Gunnar Tacke, Jan Ortmann
(Dokumentversion 105)

Überblick

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

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

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

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

Konzeptionelle Grundlagen

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

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

<i>Parameter</i>	<i>Testfall 1</i>	<i>Testfall 2</i>	<i>Testfall 3</i>
Produkt	<i>HR-Kompakt 2012-03</i>	<i>HR-Optimal 2012-03</i>	<i>HR-Optimal 2012-03</i>
Zusatzdeckungen	<i>HRD-Fahrraddiebstahl 2012-03</i> <i>HRD-Ueberspannung 2012-03</i>	<i>HRD-Fahrraddiebstahl 2012-03</i> <i>HRD-Ueberspannung 2012-03</i>	<i>HRD-Ueberspannung 2012-03</i>
Zahlweise	jährlich	jährlich	halbjährlich
Postleitzahl	81673	81673	81673
Versicherungs- summe	60.000 EUR	60.000 EUR	100.000 EUR
Erwartete Ergebnisse			
NettobeitragZw	121,00 EUR	133,00 EUR	84,98 EUR

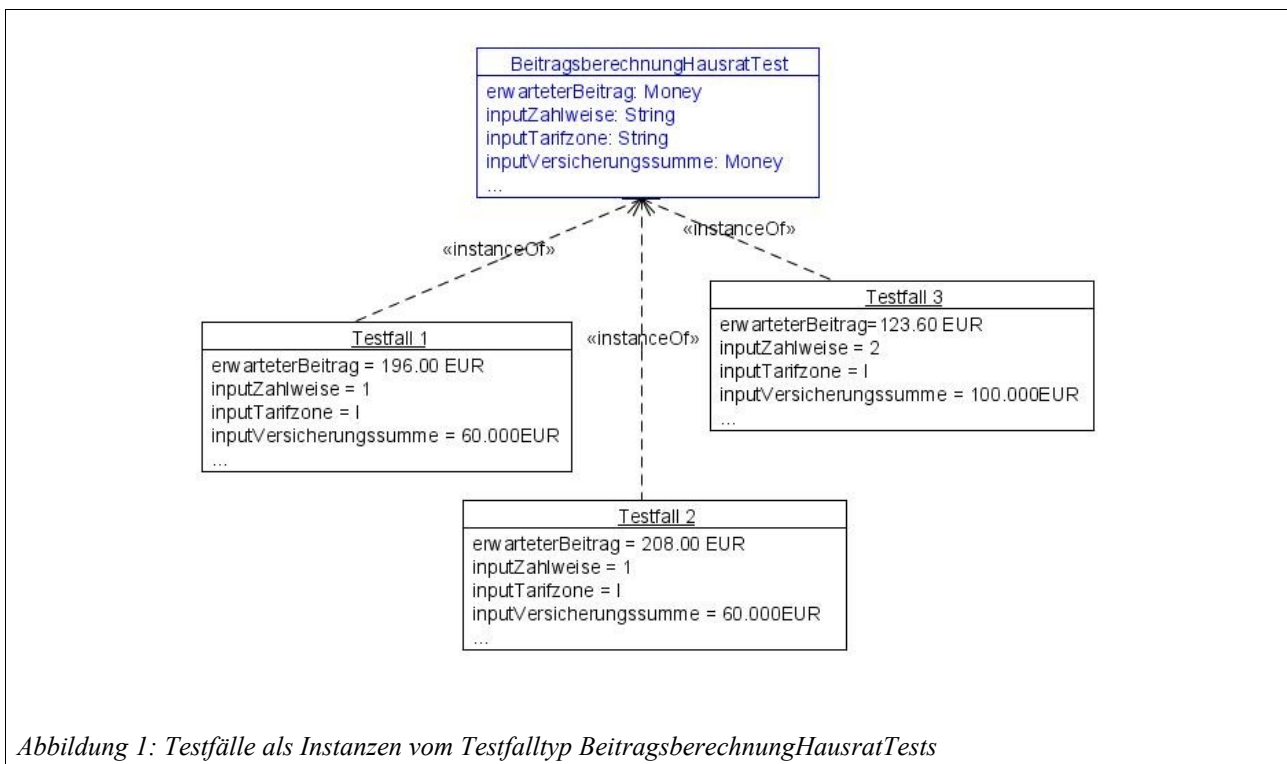
Der Ablauf dieser Testfälle ist der gleiche:

1. Es wird ein Hausratvertrag auf Basis des angegebenen Produktes mit den angegebenen

Zusatzdeckungen erzeugt und die Attribute Zahlweise, Postleitzahl und Versicherungssumme mit den Werten aus dem Testfall belegt.

2. Es wird die Beitragsberechnung ausgeführt. Hierzu wird die entsprechende Methode am Hausratvertrag aufgerufen.
3. Der NettobeitragZw des Hausratvertrags wird mit dem im Testfall hinterlegten erwarteten Wert verglichen.

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



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

Testen mit Faktor-IPS am Beispiel Hausratversicherung

Als Ausgangspunkt verwenden wir die im Einführungstutorial erstellten Projekte zur Hausratversicherung¹.

Wir wollen zunächst die Beitragsberechnung von Hausratverträgen testen. Die Berechnung ist so


¹ Die fertigen Projekte können auch zum Eclipse-Import von der Faktor-IPS Website geladen werden: http://faktorzehn.org/_media/fips:tutorial-projekte.zip

implementiert, dass abhängig von der gewählten Versicherungssumme und des gewählten Produkts ein Grundbeitrag errechnet wird und dieser abhängig von der Tarifzone (die wiederum von der PLZ des Hausrats abhängt) mit einem Tarifzonen-Faktor multipliziert wird. Werden Zusatzdeckungen (z.B. Fahrraddiebstahl), eingeschlossen, wird ein (im Produkt konfigurierter) Anteil des Grundbeitrags aufgeschlagen (z.B. Fahrraddeckung: + 10% des Beitrags der Grunddeckung). Abhängig von der Zahlungsweise kommt ggf. ein Ratenzahlungszuschlag hinzu.

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

Testfalltyp für Beitragsberechnung Hausrat

Für den Testfalltypen erweitern wir die Package-Struktur unterhalb des Sourcefolders „modell“ im Projekt org.faktorips.tutorial.de.Hausratmodell um das IPS-Package „test“².

Mit dem Kontextmenü New ► Test Case Type im Model Explorer legen wir einen neuen Testfalltypen an (alternativ können Testfalltypen auch über das Icon  in der Toolbar angelegt werden):

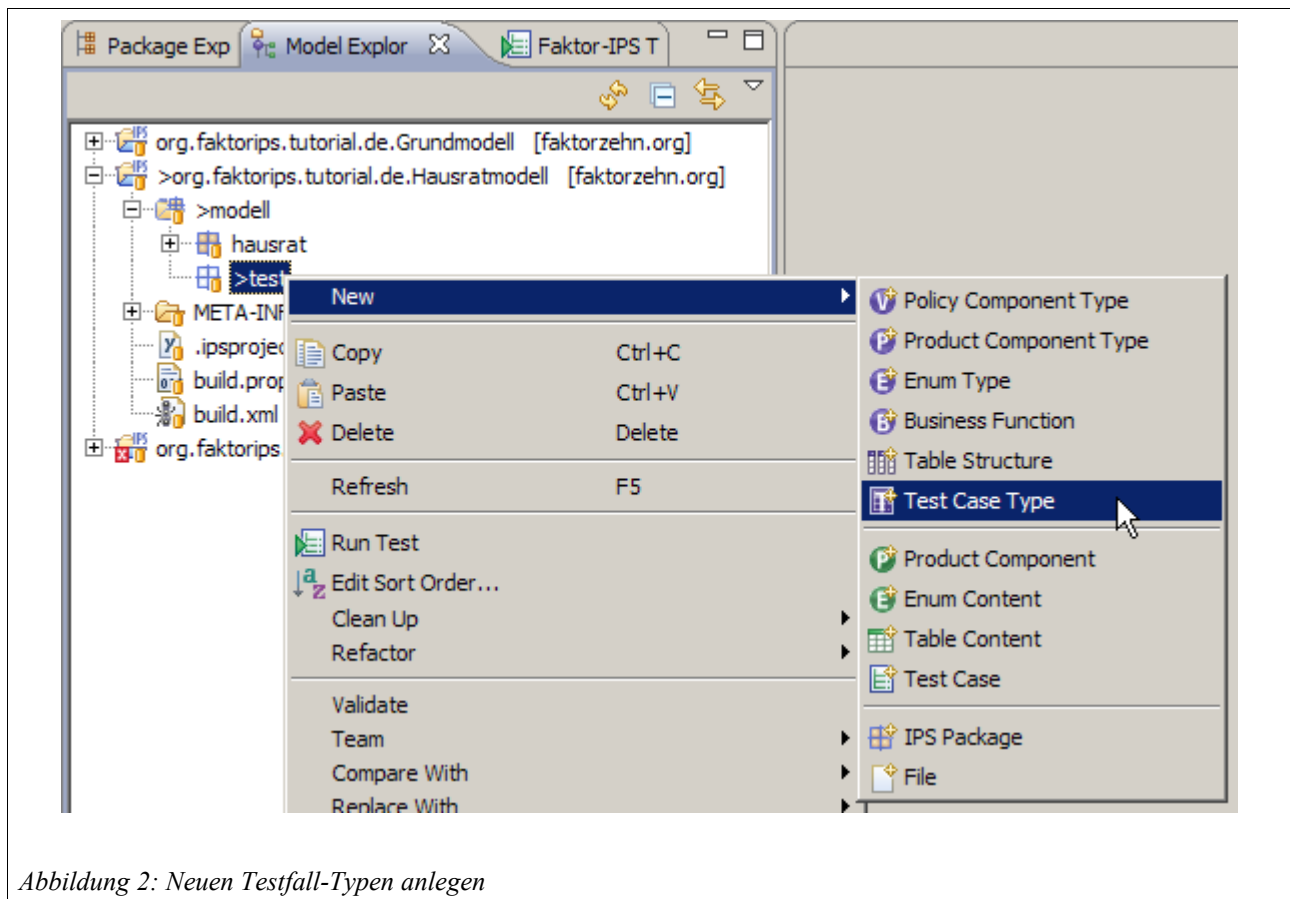


Abbildung 2: Neuen Testfall-Typen anlegen

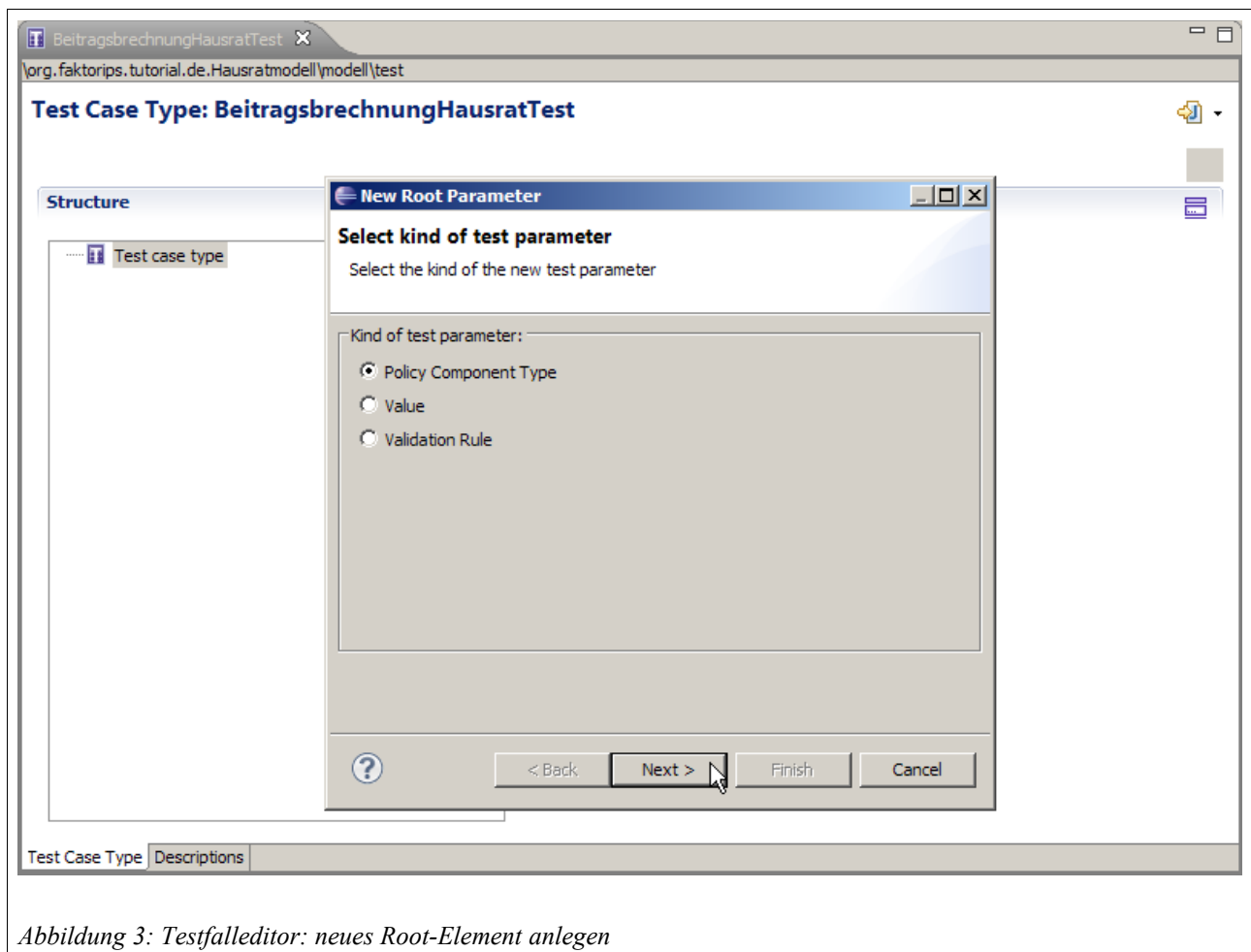
Nachdem wir im folgenden Dialog den Namen des Testfalls *BeitragsberechnungHausratTest* definiert haben, öffnet sich der Editor für Testfalltypen (s. Abbildung 2). Auf der linken Seite sehen wir die (zunächst leere) Struktur des Testfalls, auf der rechten Seite jeweils die Details zu den Elementen der Struktur. Wir beginnen, die Struktur unseres Testfalls anzulegen. Testfalltypen

² In der Praxis bietet es sich an, mit einem eigenen Sourcefolder für Tests zu arbeiten, der Einfachheit halber wird in diesem Beispiel alles im Sourcefolder *modell* abgelegt.

werden in einer Baumstruktur erfasst. Zunächst werden wir also das Wurzelement definieren. Dazu rufen wir mit New... den Wizard zum Anlegen von Testparametern auf. Zunächst müssen wir die Art des Testparameters bestimmen. Es gibt drei Arten von Testparametern (zunächst unabhängig davon, ob diese als Eingabeparameter oder erwartete Ergebnisse benutzt werden):

- Policy Component Type (Vertragsteilkasse)
Eine Vertragsteilkasse als Testparameter
- Value (Wert)
Ein Wert
- Validation Rule (Regel)
Eine zu testende Regel

Für unseren Testfalltyp, mit dem wir die Beitragsberechnung von Hausratverträgen testen wollen, wählen wir als Wurzelement den Typ Vertragsteilkasse (Policy Component Class):

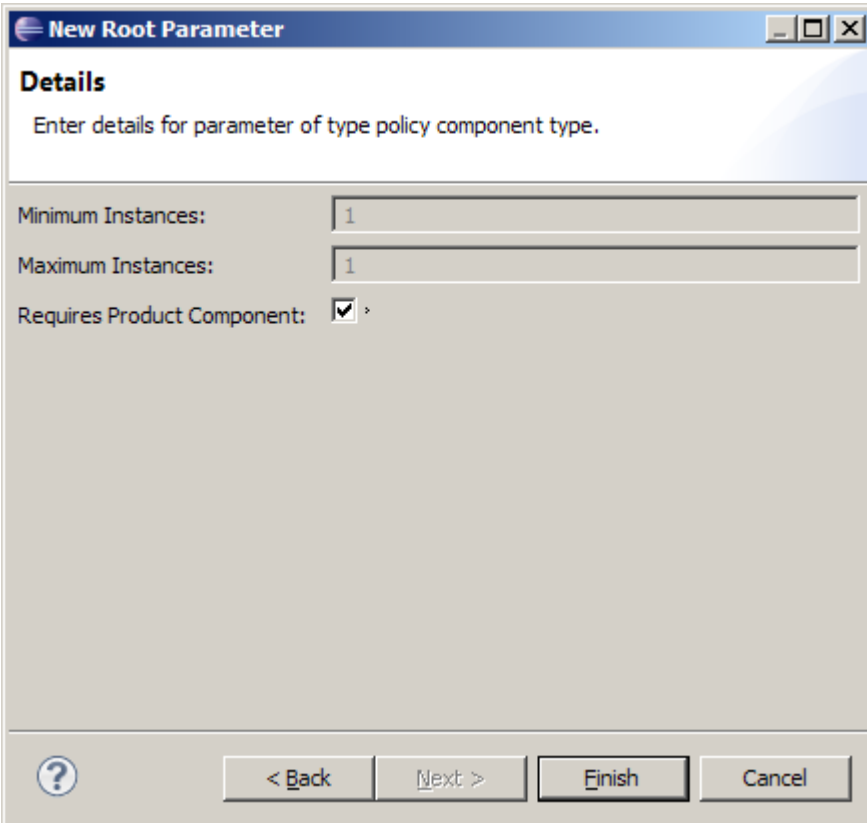


Nun wählen wir als Datatype unsere Vertragsteilkasse *HausratVertrag*. Der Name wird mit dem Namen des Datentypen vorbelegt, dies belassen wir für unser Beispiel auch so. Im Feld Type können wir spezifizieren, welche Funktion der Parameter im Testfall hat:

- Input
Attribute der Vertragsteilkasse dienen ausschließlich als Eingabedaten für Testfälle

- Expected Result
Attribute der Vertragsteilkategorie dienen ausschließlich als erwartete Werte der Testfälle
- Input and Expected Result
Attribute der Vertragsteilkategorie können entweder Eingabeparameter oder erwarteter Wert sein

Für unser Beispiel wählen wir Input and Expected Result als Parametertyp, da wir an den zu definierenden Hausratverträgen sowohl Eingabeparameter als auch das erwartete Ergebnis, in unserem Fall der berechnete Beitrag, definieren wollen:



The screenshot shows a dialog box titled "New Root Parameter". The main area is labeled "Details" and contains the instruction "Enter details for parameter of type policy component type." Below this, there are three input fields: "Minimum Instances" with the value "1", "Maximum Instances" with the value "1", and "Requires Product Component" with a checked checkbox. At the bottom of the dialog, there is a question mark icon, a "< Back" button, a "Next >" button, a "Finish" button, and a "Cancel" button.

Abbildung 4: Kardinalität und Produktabhängigkeit festlegen

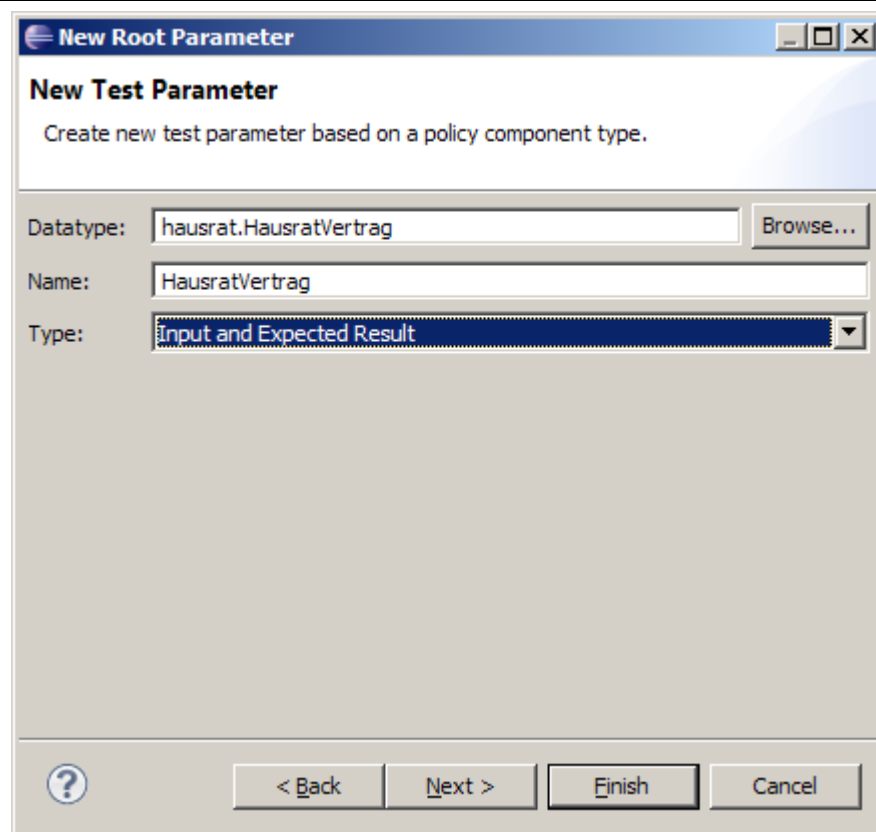


Abbildung 5: Datentyp des Testparameters definieren

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

Die Kardinalität und das Kennzeichen, ob eine Konfiguration durch einen Produktbaustein notwendig ist, kann natürlich auch noch nachträglich direkt im Testfalltyp-Editor geändert werden.

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

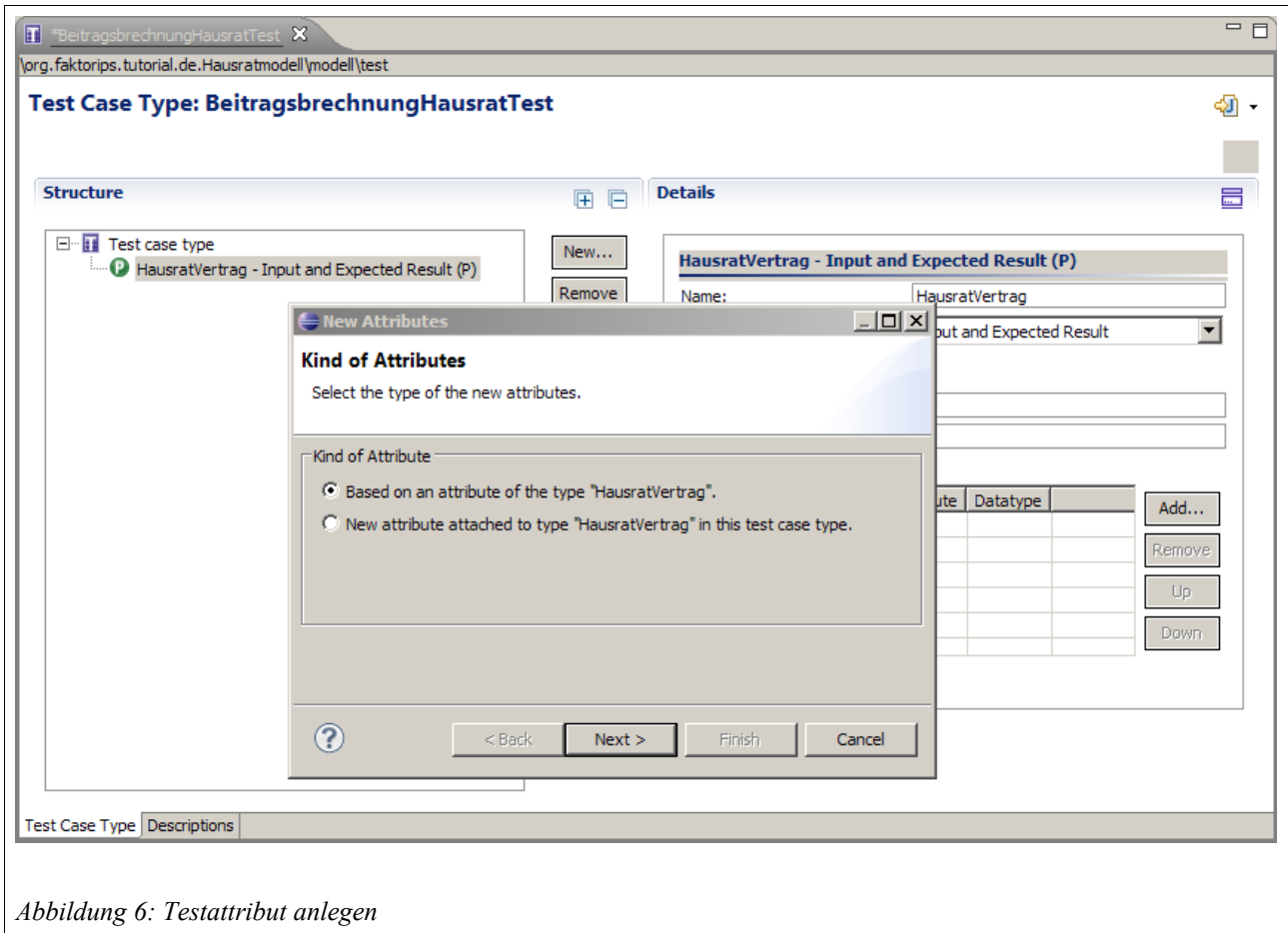


Abbildung 6: Testattribut anlegen

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

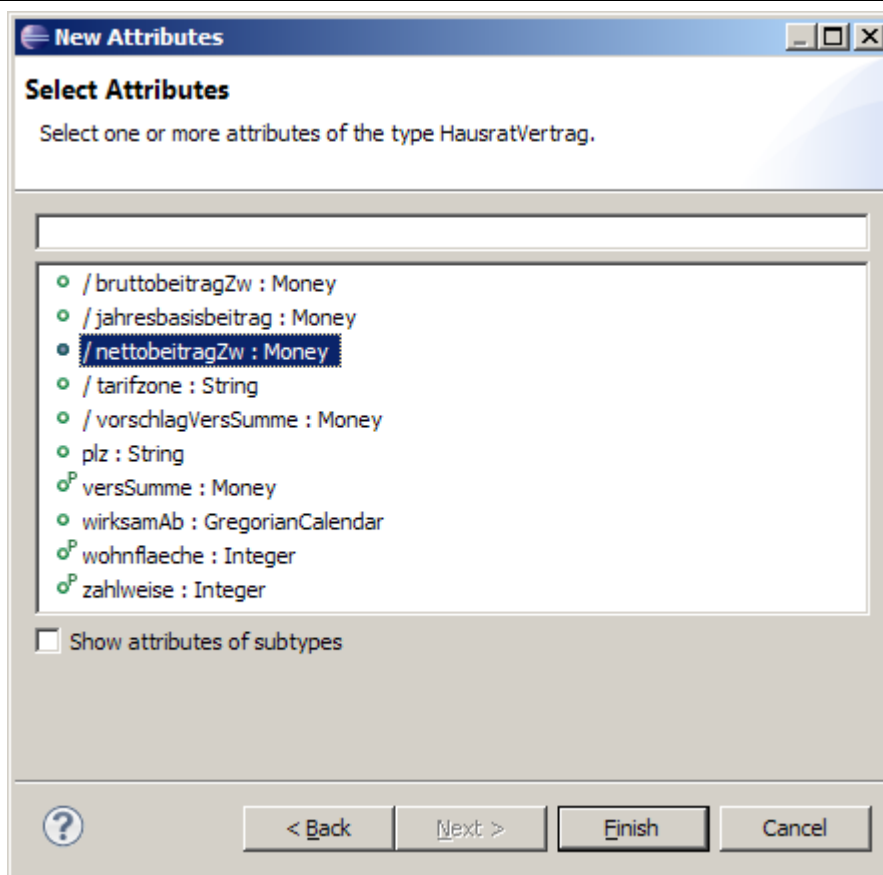


Abbildung 7: Testattribut auswählen

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

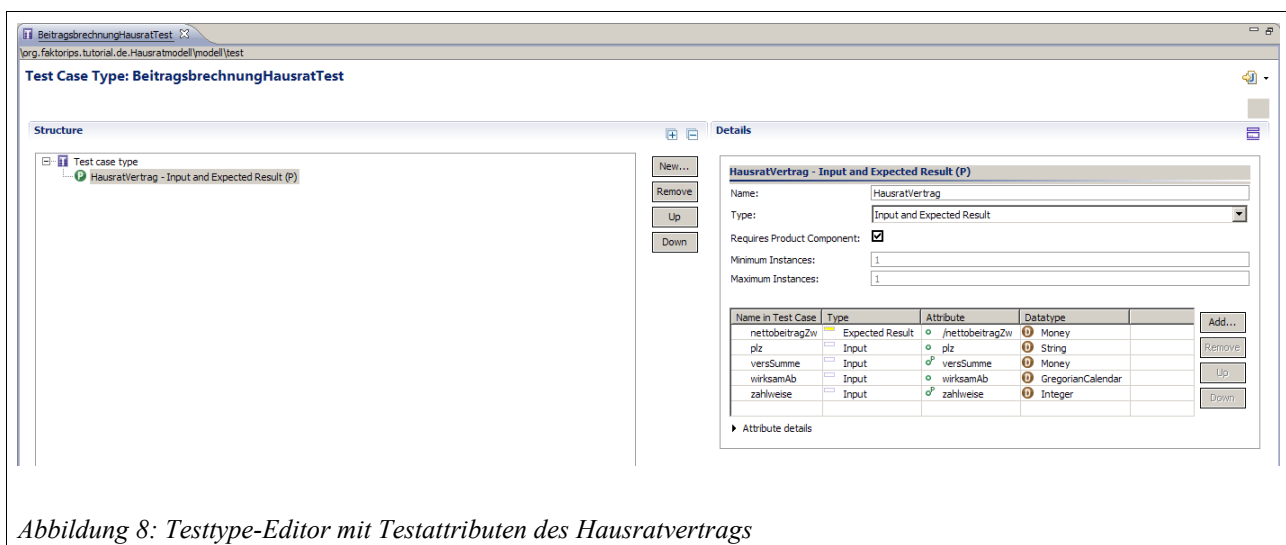


Abbildung 8: Testtype-Editor mit Testattributen des Hausratvertrags

Jetzt erweitern wir die Struktur unseres Testfalltyps um alle weiteren notwendigen Elemente . Wir

nehmen die Vertragsteilklassen *HausratGrunddeckung* und *HausratZusatzdeckung* in den Testfalltyp auf. Dazu markieren wir in der Struktur das Element *HausratVertrag* und erzeugen jeweils mit New... die Beziehungen *HausratGrunddeckung* und *HausratZusatzdeckungen*. Für *HausratGrunddeckung* (s. Abbildung 9) wählen wir als Type Eingabe, als Kardinalität 1..1 und setzen die Checkbox Requires Product Component. Für die *HausratZusatzdeckungen* wählen wir als Type Input, als Kardinalität 0..* und setzen ebenfalls die Checkbox Requires Product Component. Zur Erinnerung: Die Auswahl der Checkbox bedeutet, dass die jeweilige Vertragsteilkategorie im Testfall durch einen Produktbaustein konfiguriert werden muss. Wir werden dies später sehen, wenn wir einen Testfall anlegen.

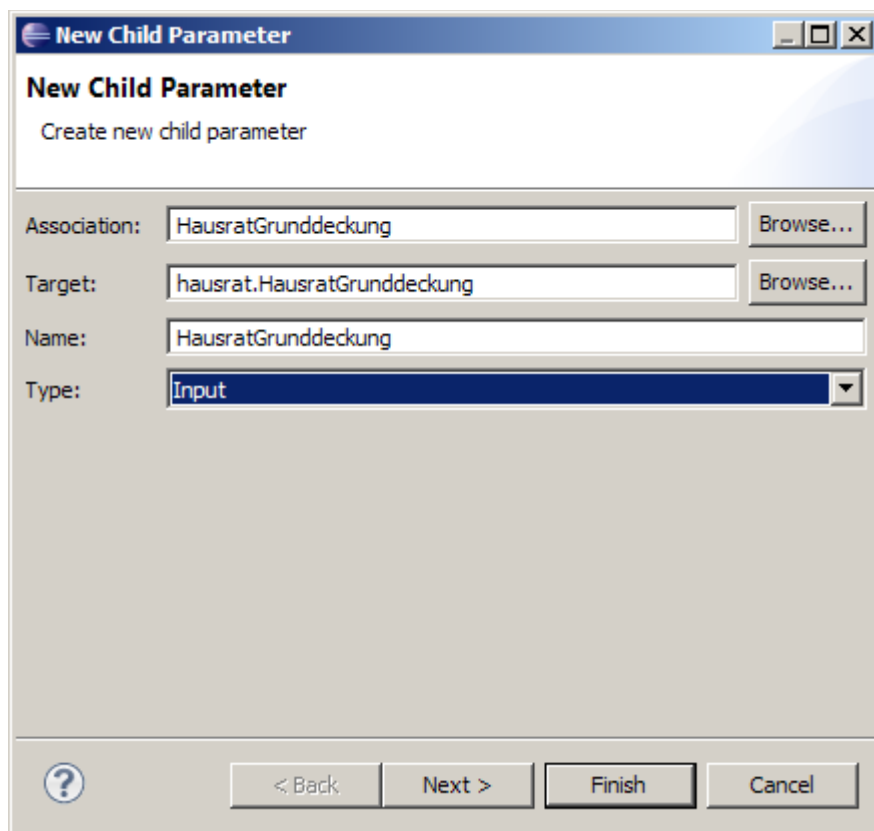


Abbildung 9: *HausratGrunddeckung* als Kind-Eingabeparameter von *Hausratvertrag* definieren.

Danach sollte der Testfalltyp-Editor so aussehen:

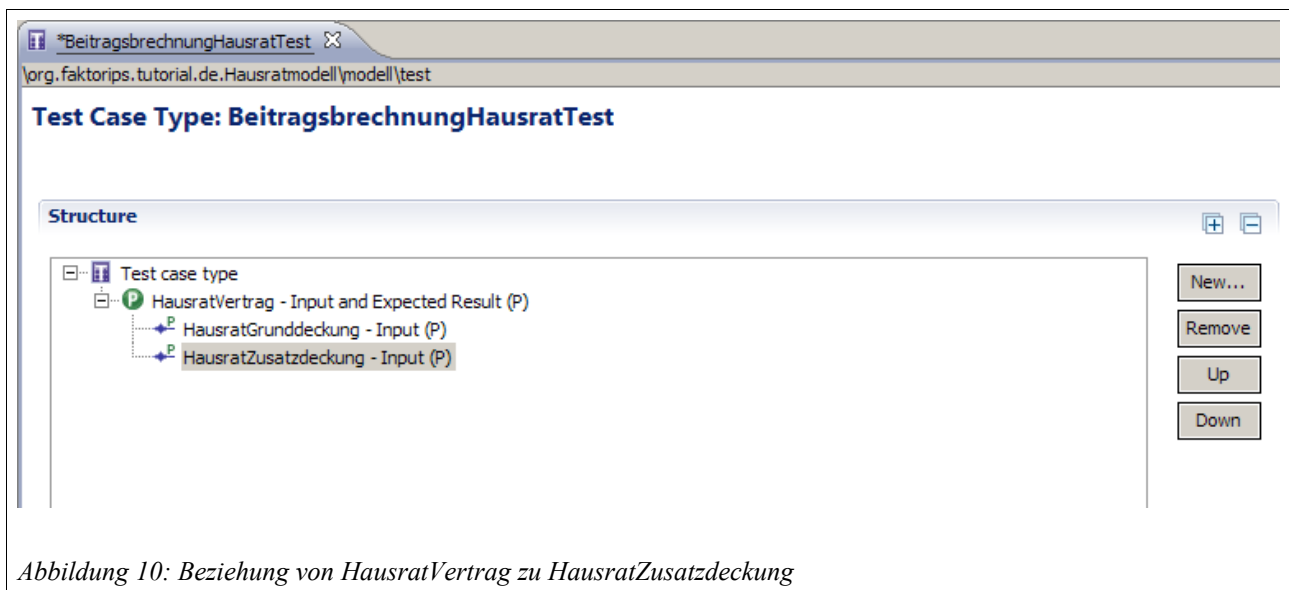


Abbildung 10: Beziehung von HausratVertrag zu HausratZusatzdeckung

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

Wenn wir den Testfalltypen speichern, wird im Source-Verzeichnis des Projekts HausratModell im Package `org.faktorips.tutorial.hausrat.internal.test` eine zugehörige Java-Klasse generiert, die den Namen des Testfalltypen trägt, und in der wir die Testlogik implementieren. Wechseln wir auf den Package-Explorer und schauen uns zunächst die Struktur der generierten Klasse genauer an:

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

    private HausratVertrag erwartetHausratVertrag;

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

    public void executeAsserts(IpsTestResult result) {
        throw new RuntimeException(
            "Keine Pruefungen vorhanden. Diese muessen in der Java-Klasse, die den Testfalltyp
            repraesentiert, implementiert werden.");
    }
}
```

- **Membervariablen** `inputHausratVertrag` und `erwartetHausratVertrag` vom Typ `HausratVertrag`:

Dies entspricht dem als erwarteter- und Eingabeparameter definierten Root-Parameter. Da wir den Root-Parameter *HausratVertrag* als Erwartetes Ergebnis und Eingabeparameter festgelegt haben, sind zwei entsprechende Instanzvariablen angelegt worden. `inputHausratVertrag` enthält die Eingabewerte des Testfalls, gemäß der Definition der Eingabewerte im Testfalltyp. `erwartetHausratVertrag` enthält die im Testfall erwarteten Ergebnisse (gemäß der Definition im Testfalltyp). Zur Testfalllaufzeit haben wir so zwei

Instanzen von `HausratVertrag`, die wir miteinander vergleichen können.

- leere Methode `executeBusinessLogic()`:
In dieser Methode wird die zu testende Geschäftslogik aufgerufen. Zum Beispiel durch Aufruf einer Methode auf den Eingabeobjekten (hier `inputHausratVertrag`). Die Methode wird ausgeführt, bevor die Methode `executeAsserts(...)` aufgerufen wird.
- Testmethode `executeAsserts()`:
Hier wird der Vergleich von Ist- und Sollwerten implementiert. Zunächst ist hier eine Default-Implementierung generiert, die den Test fehlschlagen lässt, um eine Implementierung durch den Entwickler zu erzwingen.

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

Testfall anlegen

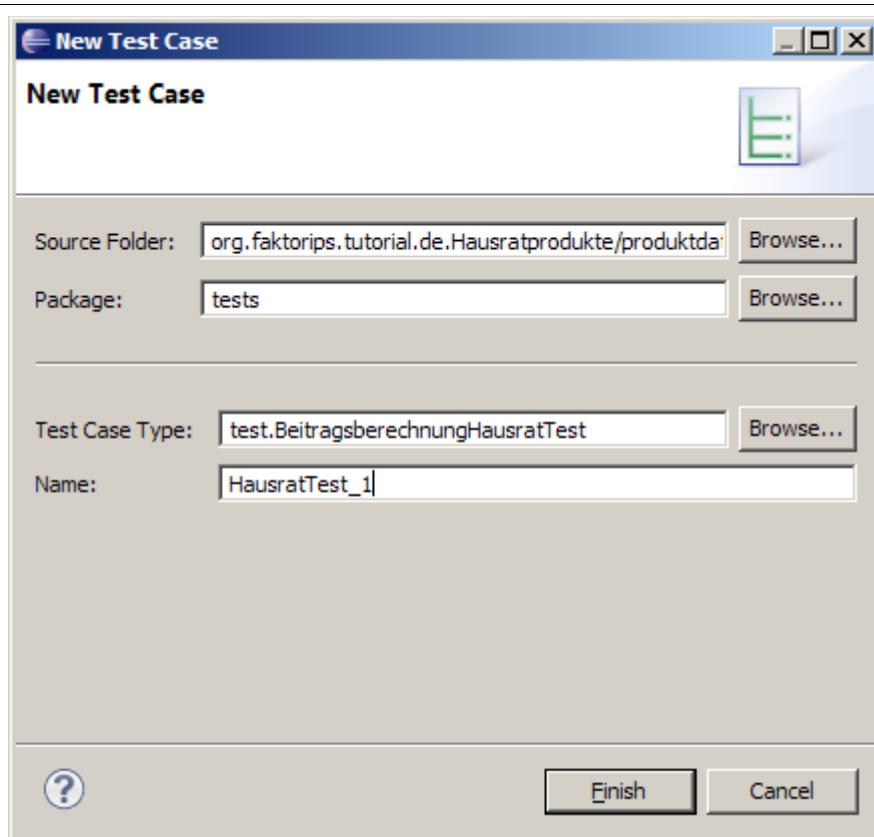


Abbildung 11: Neuen Testfall anlegen

Im Projekt `org.faktorips.tutorial.de.Hausratprodukte` legen wir unterhalb von `produktdata` ein neues IPS Package `tests` an, in dem wir unsere Testfälle ablegen. Dazu wechseln wir wieder in den Model Explorer markieren das Verzeichnis `produktdata` und wählen über das Kontextmenü `New ► IPS Package` an. Wir legen mit `New ► Test Case` im Kontextmenü einen neuen Testfall an. Nun wählen wir im Wizard zur Testfallanlage den zuvor angelegten Testfalltyp `test.Bei...beitragsberechnungHausratTest` aus und geben dem Testfall einen Namen. Nun öffnet sich der Testfalleditor: Auf der linken Seite befindet sich die Struktur des Testfalls. Sie entspricht der

Struktur, die wir im Testfalltypen definiert haben. Auf der rechten Seite sind die Testdaten. Für jedes, im Testfalltypen definierte Attribut, ist hier ein Eingabefeld zu sehen, wobei Eingabewerte weiß und erwartete Werte gelb hinterlegt sind.

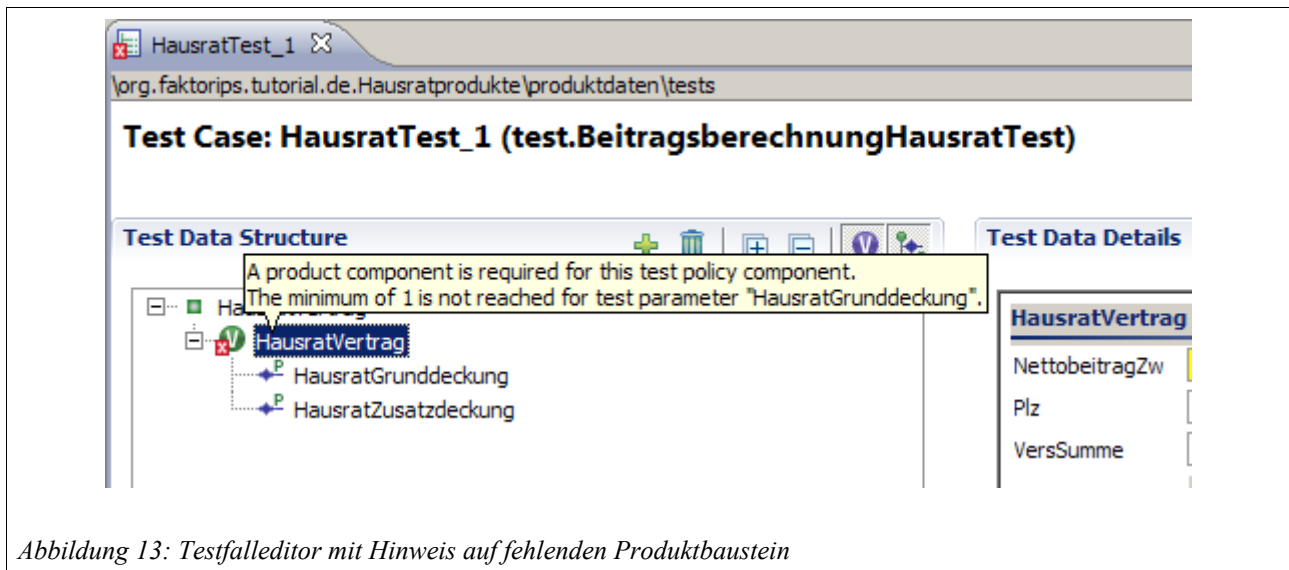


Abbildung 13: Testfalleditor mit Hinweis auf fehlenden Produktbaustein

Auf der linken Seite müssen wir die Vertragsteile jetzt noch durch Zuordnung von Produktbausteinen ausdrängen. Mit dem Button Assign Product Component wird der Produktbaustein der aktuellen Vertragsteilklassse ausgewählt.

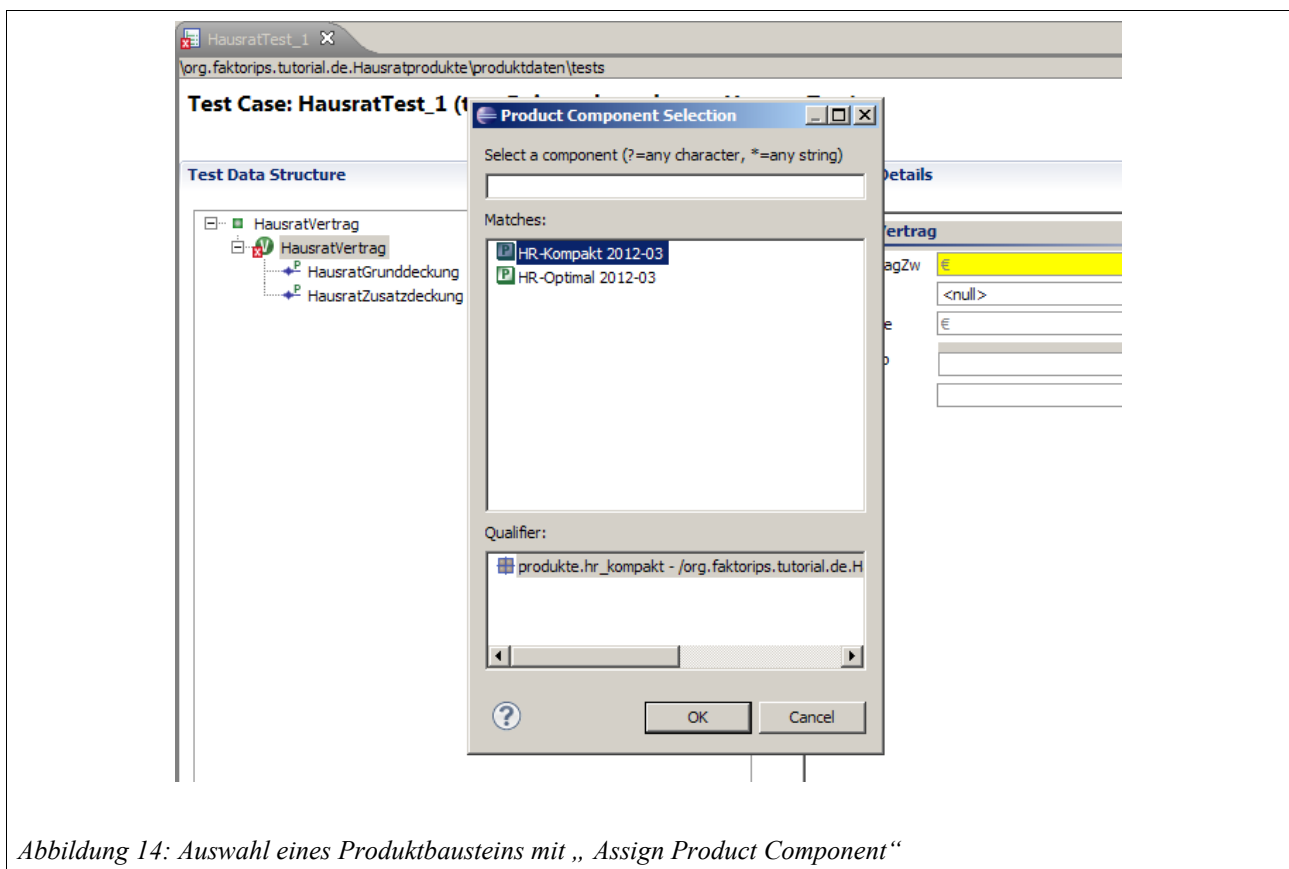


Abbildung 14: Auswahl eines Produktbausteins mit „Assign Product Component“

Wir wählen für unser Beispiel den Produktbaustein *HR-Kompakt 2012-03* für HausratVertrag.

Danach fügen wir mit Add weitere Objekte hinzu: *HRD-Grunddeckung-Kompakt 2012-03* für HausratGrunddeckung, und prägen zwei mal HausratZusatzdeckung, jeweils mit *HRD-Fahrraddiebstahl 2012-03* und *HRD-Ueberspannung 2012-03* aus. Danach vervollständigen wir den Testfall gemäß folgender Tabelle:

<i>Parameter</i>	<i>Wert</i>
Produkt	HR-Kompakt 2012-03
Grunddeckungstyp	HRD-Grunddeckung-Kompakt 2012-03
Zusatzdeckungstypen	HRD-Fahrraddiebstahl 2012-03 HRD-Ueberspannung 2012-03
Versicherungssumme	60.000 EUR
Wirksam ab	2012-03-01
PLZ	81673 (Tarifzone I)
Zahlweise	1 (jährlich)
<i>Nettobeitrag gemäß Zahlweise</i>	<i>121,00 EUR</i>

Tabelle 1: Testfall Beitragsberechnung Hausrat

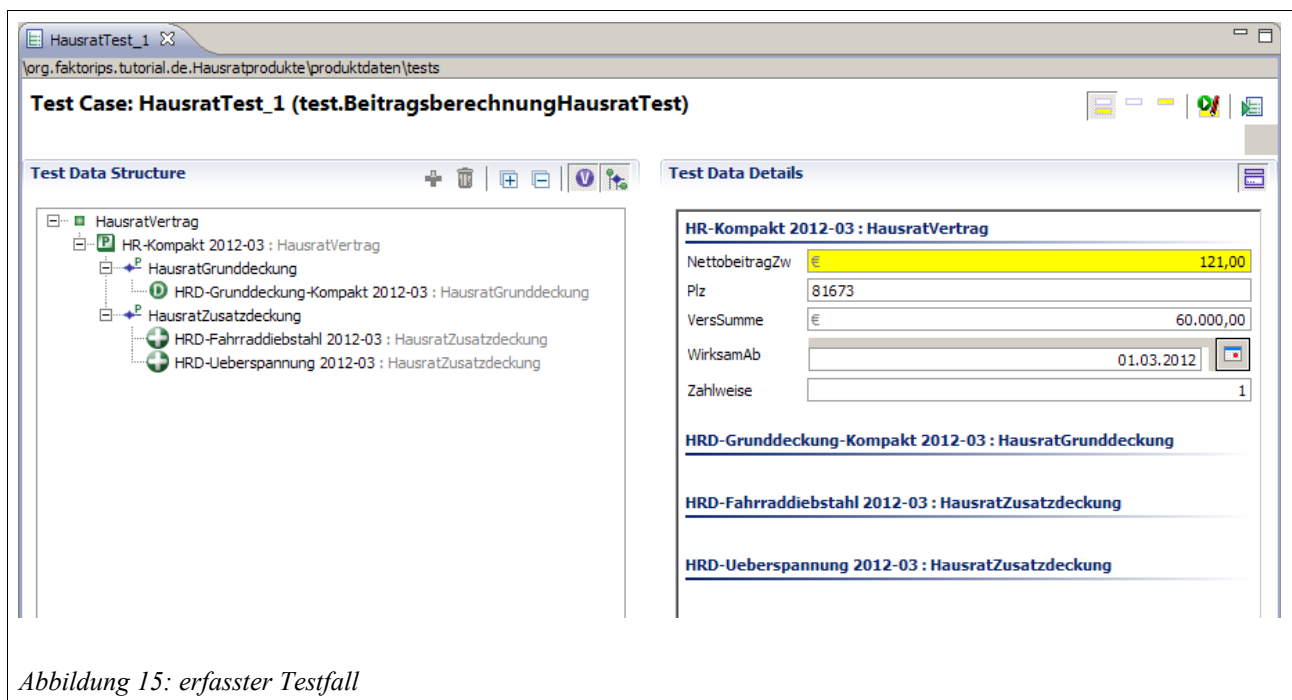


Abbildung 15: erfasster Testfall

Nun starten wir die Ausführung des Testfalls mit dem Icon Run Test (🏃) in der oberen rechten Ecke des Testfall-Editors. (Der Testfall kann auf zwei Arten gestartet werden. Wir starten mit Run test. Die Variante Compute Expected Values sehen wir später im Kapitel Testfälle durch Kopieren erzeugen).

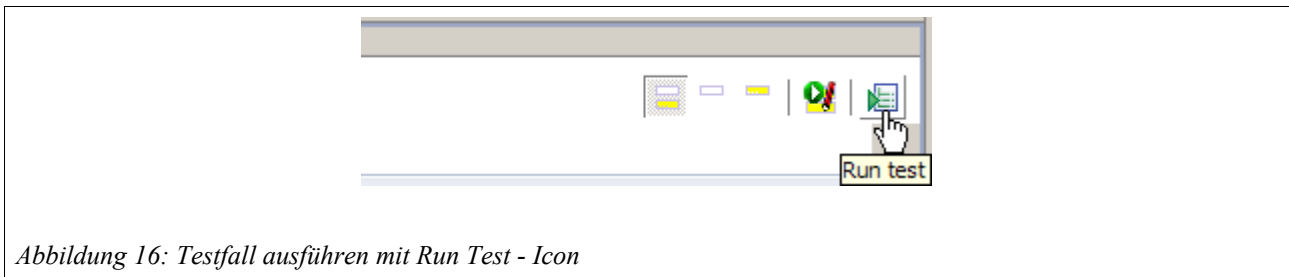


Abbildung 16: Testfall ausführen mit Run Test - Icon

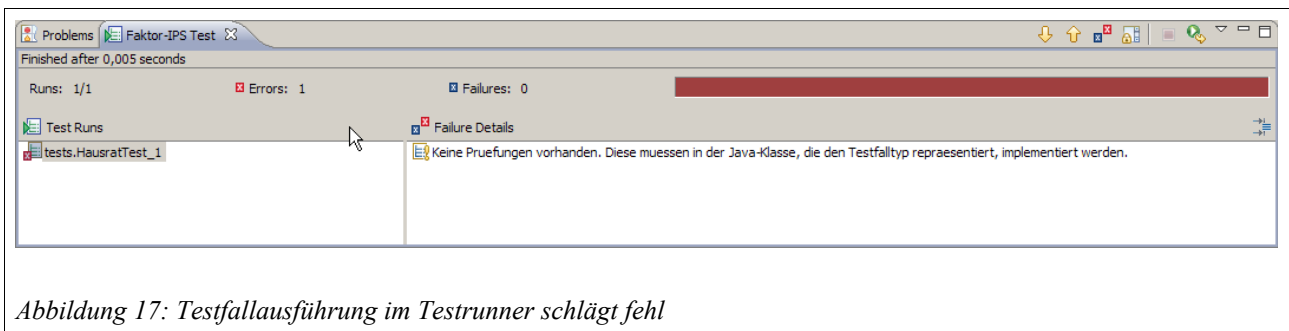


Abbildung 17: Testfallausführung im Testrunner schlägt fehl

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

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

    /**
     * Fuehrt die zu testende Geschaeftslogik aus.
     *
     * @generated NOT
     */
    public void executeBusinessLogic() {
        inputHausratVertrag.berechneBeitrag();
    }

    /**
     * Fuehrt die Pruefungen (Asserts) aus, d.h. vergleicht die erwarteten
     * Werten mit den tatsaechlichen Ergebnissen.
     *
     * @generated NOT
     */
    public void executeAsserts(IpsTestResult result) {
        assertEquals( erwartetHausratVertrag.getNettobeitragZw(),
            inputHausratVertrag.getNettobeitragZw(),
            result);
    }
}
```

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

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

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

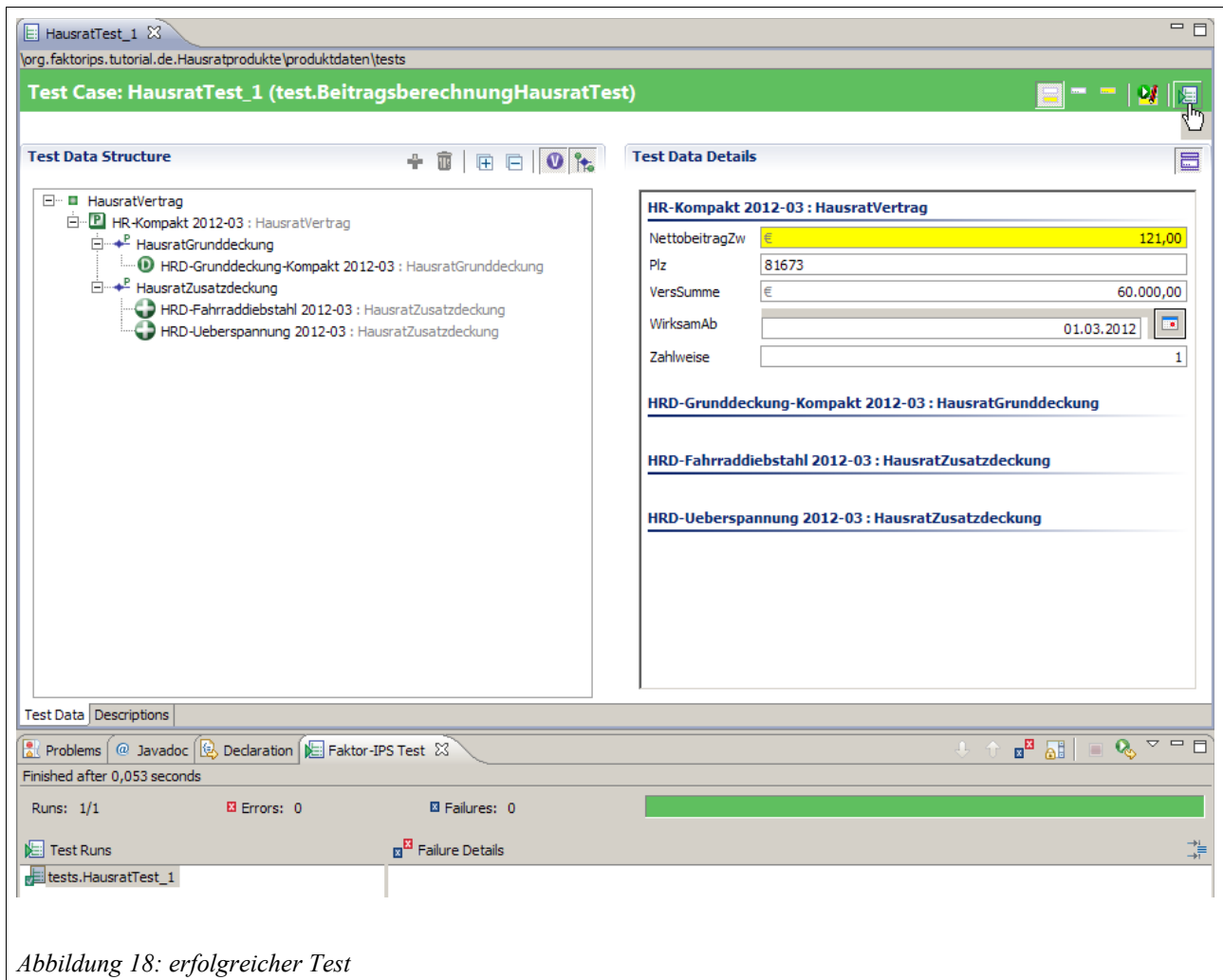


Abbildung 18: erfolgreicher Test

Machen wir die Gegenprobe und ändern das erwartete Ergebnis:

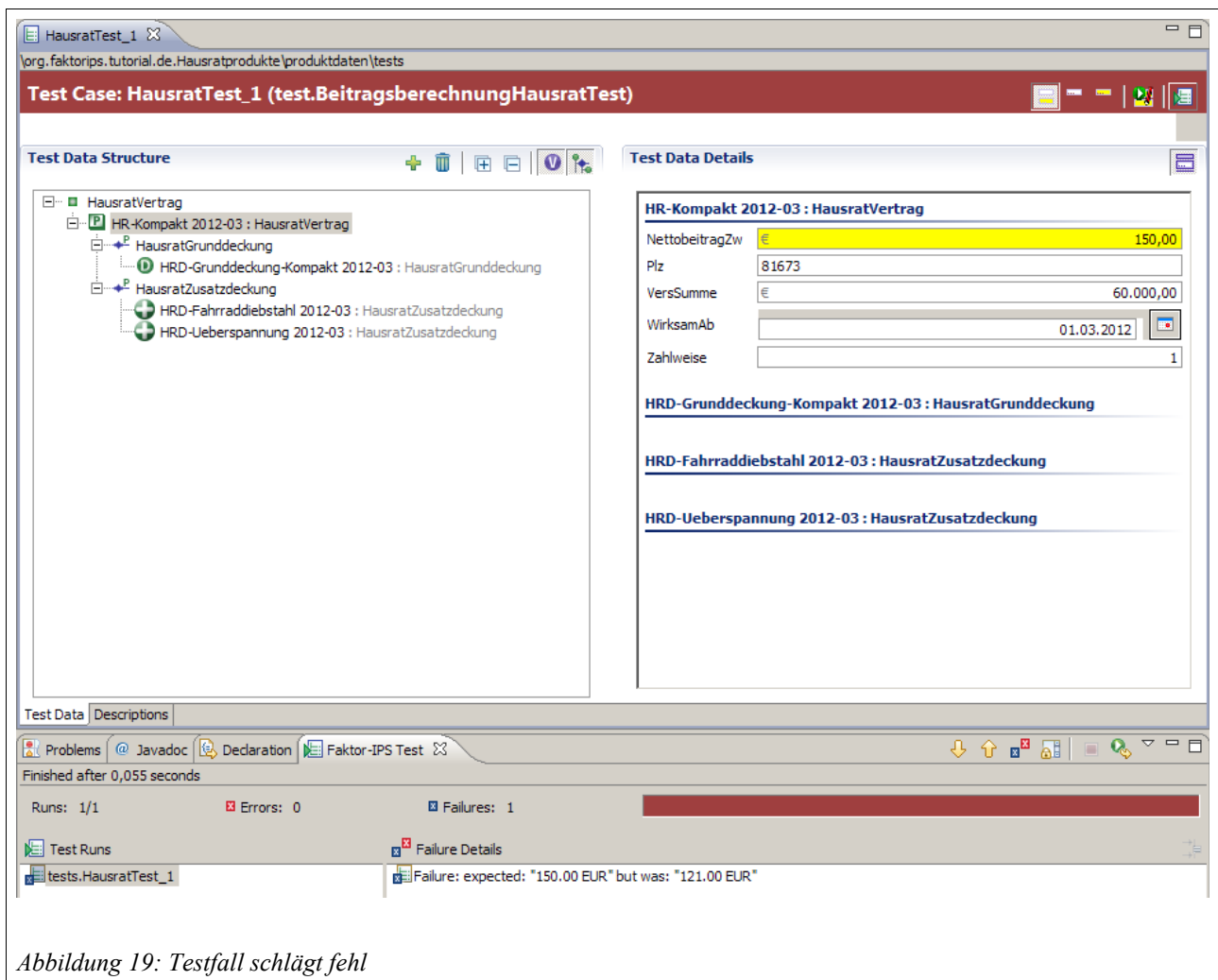


Abbildung 19: Testfall schlägt fehl

Der Testfall schlägt fehl. In den Failure Details sehen wir, dass der berechnete Wert 121 EUR nicht dem erwarteten Wert entspricht. Wir sehen der Fehlermeldung aber noch nicht an, auf welches Testattribut sie sich bezieht. In unserem Fall ist dies zwar einfach, da wir nur ein erwartetes Attribut vergleichen, aber es können ja durchaus mehrere Attribute in einem Testfall verglichen werden (z.B. die einzelnen Beiträge je Deckung und der Gesamtbeitrag). Hierzu können wir den `assert*`-Statements eine Referenz auf das Attribut aus der Definition des Testfalltyps mitgeben. Als ersten weiteren Parameter übergeben wir einen String, der das Testobjekt identifiziert, als weiteren Parameter den Namen des fehlerhaften Attributes. Wenn es mehrere Instanzen eines Objekts im Test gibt, muss der Index der Instanz, beginnend bei 0, getrennt von einer Raute angefügt werden, z.B. `Zusatzdeckung#0` für die erste Instanz des Objekts `Zusatzdeckung`. Für das Attribut `nettobeitragZw` in unserem Beispiel sieht die Implementierung dann wie folgt aus:


```

/**
 * Fuehrt die Pruefungen (Asserts) aus, d.h. vergleicht die erwarteten
 * Werten mit den tatsaechlichen Ergebnissen.
 *
 * @generated NOT
 */
public void executeAsserts(IpsTestResult result) {
    assertEquals( erwartetHausratVertrag.getNettobeitragZw(),
                 inputHausratVertrag.getNettobeitragZw(),
                 result,
                 "HausratVertrag#0",
                 "nettobeitragZw");
}

```

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

Abbildung 20: Markierung des fehlerhaften Attributs im Testeditor

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

Sonderfall: Testen von abgeleiteten Attributen

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

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

wir in unserer *Erwartet*-Instanz keine Membervariable haben, die wir für einen Vergleich nutzen können.

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

Als Eingabeparameter definieren wir die Attribute *plz* und *wirksamAb* der Vertragsteilklass *HausratVertrag*. Für *HausratVertrag* setzen wir die Checkbox *Requires Product Component*³. Anstatt die Tarifzone als erwartetes Ergebnis basierend auf der Vertragsteilklass *Vertrag* anzugeben, legen wir ein Testfall-Attribut an, das nicht auf einer Vertragsklasse basiert:

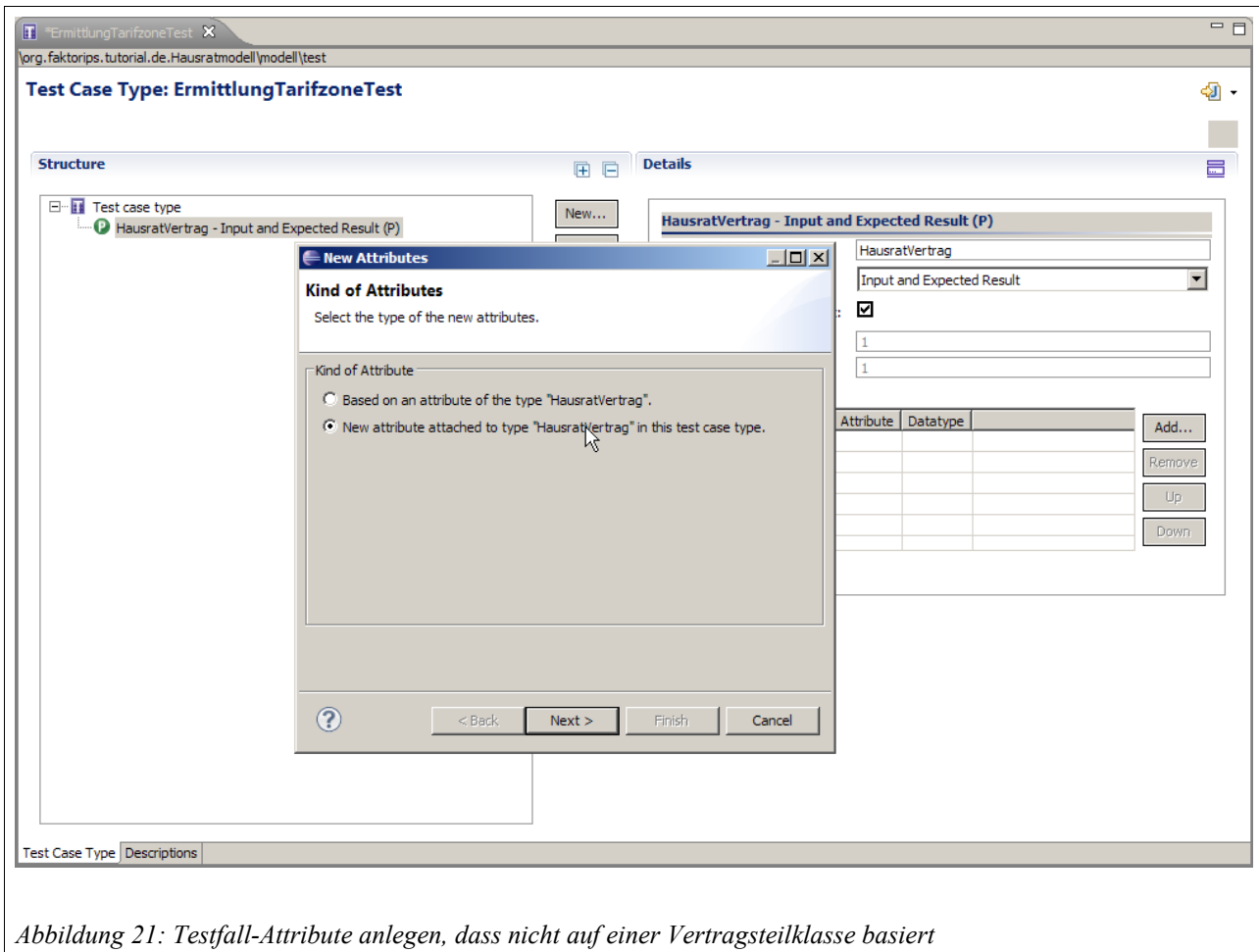


Abbildung 21: Testfall-Attribute anlegen, dass nicht auf einer Vertragsteilklass basiert

3 Im Hausratbeispiel ist die Tarifzone zwar unabhängig vom ausgewählten Produkt. In der Implementierung im Basistutorial wird der zugeordnete Produkbaustein jedoch dazu verwendet, um eine Referenz auf das RuntimeRepository zu bekommen, aus dem die Tarifzontabelle geladen wird. Damit unsere Implementierung im Testfalltypen funktioniert, machen wir den Hausratvertrag konfigurierbar, obwohl es für die Tarifzonenermittlung egal ist, welches Produkt im Testfall zugeordnet wird. In der Praxis kann man sich, z.B. in Hinblick auf zeitliche Änderungen, produkt- bzw. generationsabhängige Tarifzontabellen vorstellen.

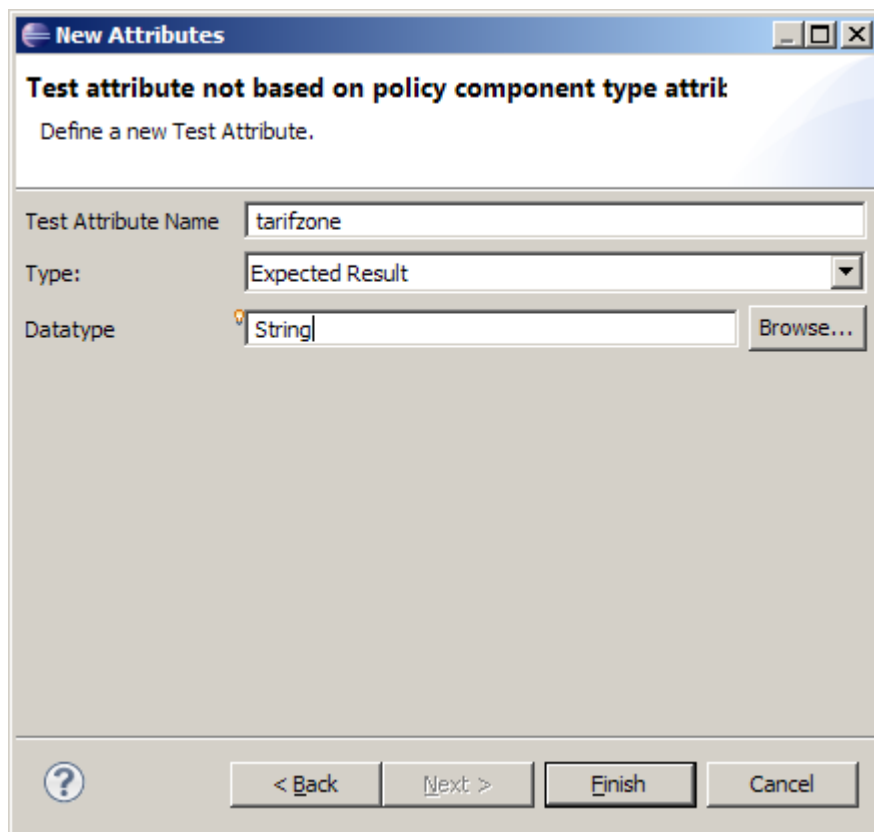


Abbildung 22: Attribut tarifzone definieren

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

```
public class ErmittlungTarifzoneTest extends IpsTestCase2 {
public final static String TESTATTR_HAUSRATVERTRAG_TARIFZONE = "tarifzone";
//...
public void executeBusinessLogic() {
// nichts zu tun (die zu testende Logik wird durch getter-Aufruf ausgeführt)
}

/**
 * Fuehrt die Pruefungen (Asserts) aus, d.h. vergleicht die erwarteten
 * Werten mit den tatsaechlichen Ergebnissen.
 *
 * @generated NOT
 */
public void executeAsserts(IpsTestResult result) {

    String erwartetTarifzone = (String) getExtensionAttributeValue(
        erwartetHausratVertrag, TESTATTR_HAUSRATVERTRAG_TARIFZONE);

    String inputTarifzone = inputHausratVertrag.getTarifzone();

    assertEquals(erwartetTarifzone, inputTarifzone, result,
        "HausratVertrag#0", TESTATTR_HAUSRATVERTRAG_TARIFZONE);
}
//...
}
```

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

Zur Überprüfung legen wir einen Testfall (*ErmittleTarifzoneTest_1*) basierend auf dem neuen Testfalltyp an und befüllen die Testdaten für `WirksamAb` und `Plz`. Dem `HausratVertrag` ordnen wir noch ein Hausratprodukt (z.B. *HR-Kompakt 2012-03*) zu. Gemäß der Tarifzontabelle erwarten wir für die Postleitzahl 63066 die Tarifzone VI:

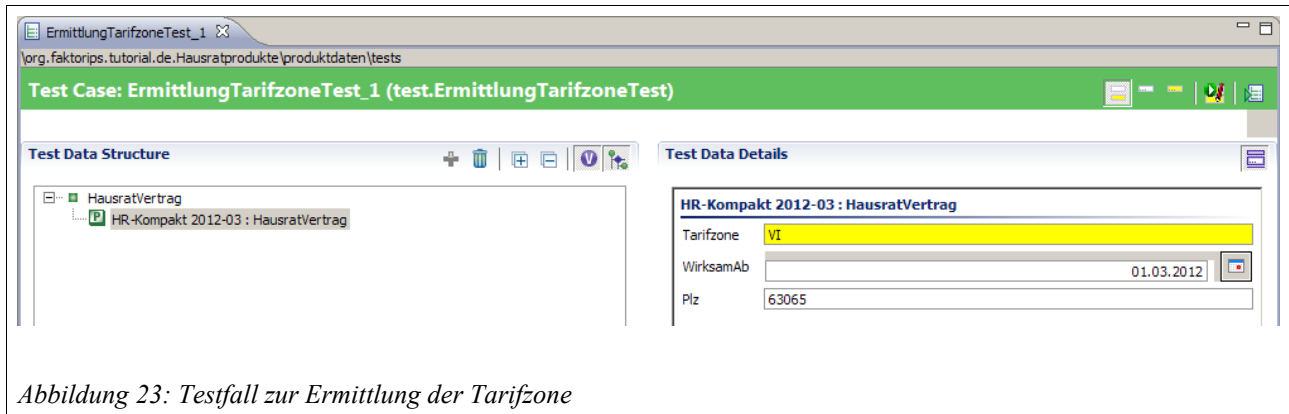


Abbildung 23: Testfall zur Ermittlung der Tarifzone

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

Testfälle durch Kopieren erzeugen

In dem folgenden Beispiel zeigen wir, wie wir durch einfaches Kopieren neue Tests erzeugen und anpassen können. Wir wollen einen Testfall erstellen, der statt auf dem Produkt *HR-Kompakt 2012-03* auf dem Produkt *HR-Optimal 2012-03*

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

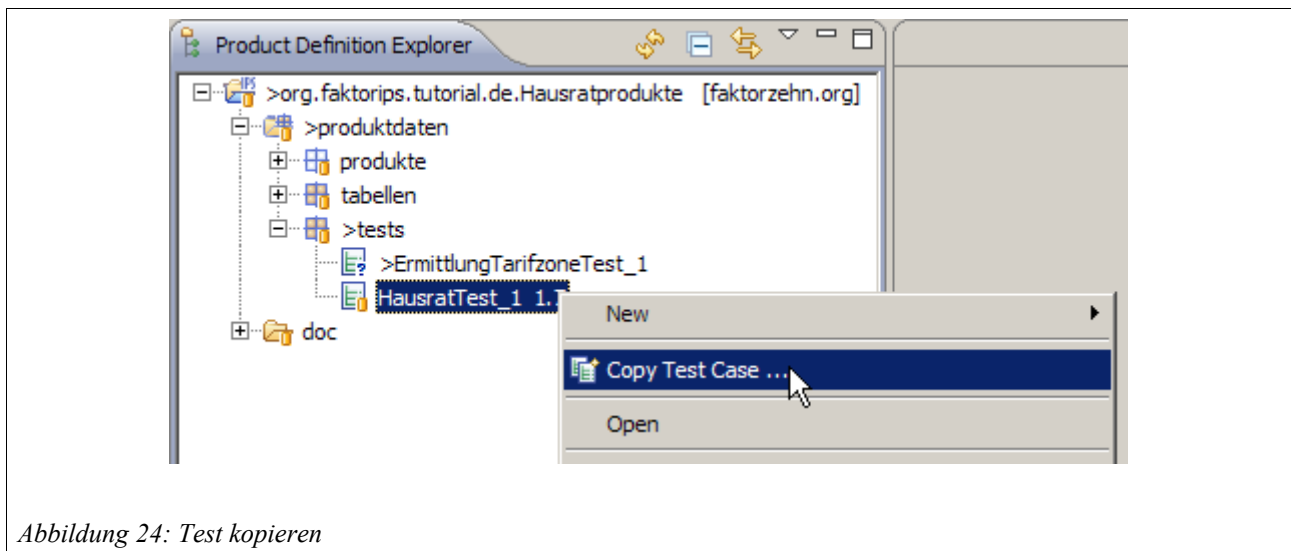


Abbildung 24: Test kopieren

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

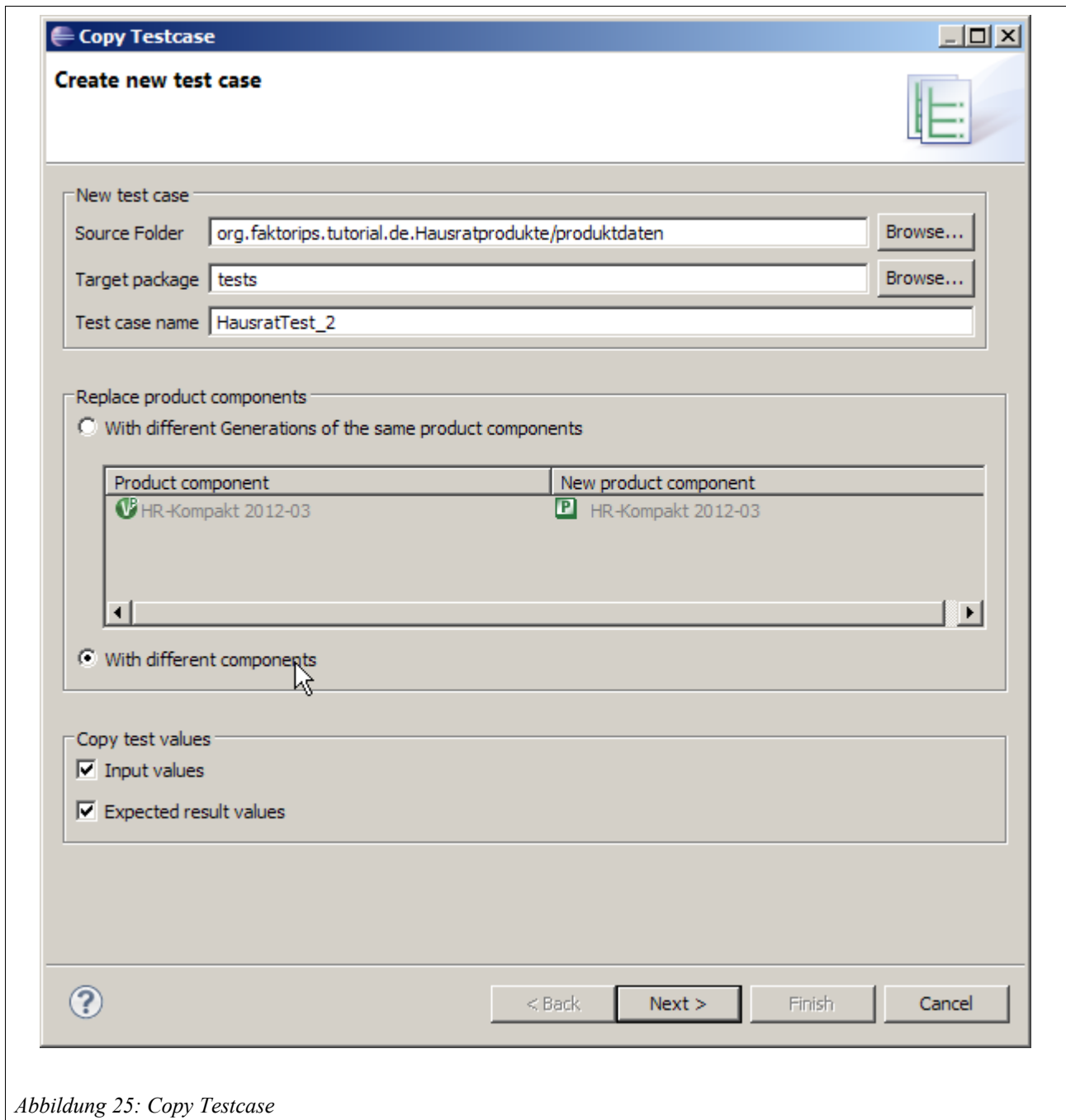


Abbildung 25: Copy Testcase

Wir haben nun die Möglichkeit, Produktbausteine abzuwählen oder durch andere zu ersetzen. Hierzu markieren wir den zu ersetzenden Baustein in der Strukturansicht auf der linken Seite. In der Liste auf der rechten Seite werden nun alle, für diese Beziehung passenden Produktbausteine angezeigt und wir wählen den neuen Produktbaustein aus. Wir ersetzen *HR-Kompakt 2012-03* durch *HR-Optimal 2012-03* und dementsprechend *HRD-Grunddeckung-Kompakt 2012-03* durch *HRD-Grunddeckung-Optimal 2012-03*.

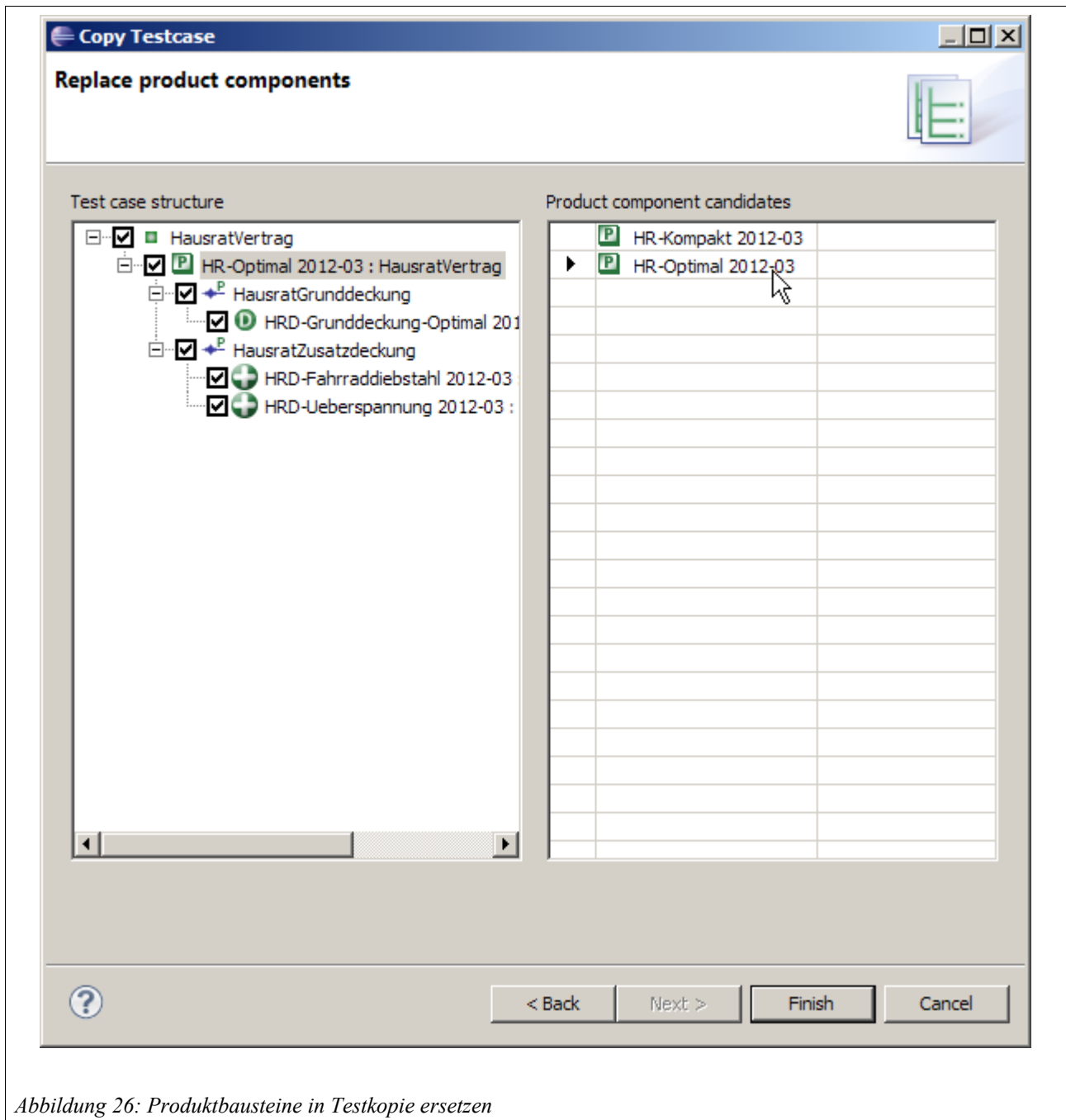


Abbildung 26: Produktbausteine in Testkopie ersetzen

Mit Finish verlassen wir den Wizard, der neue Testfall wird angelegt und im Testfalleditor geöffnet.

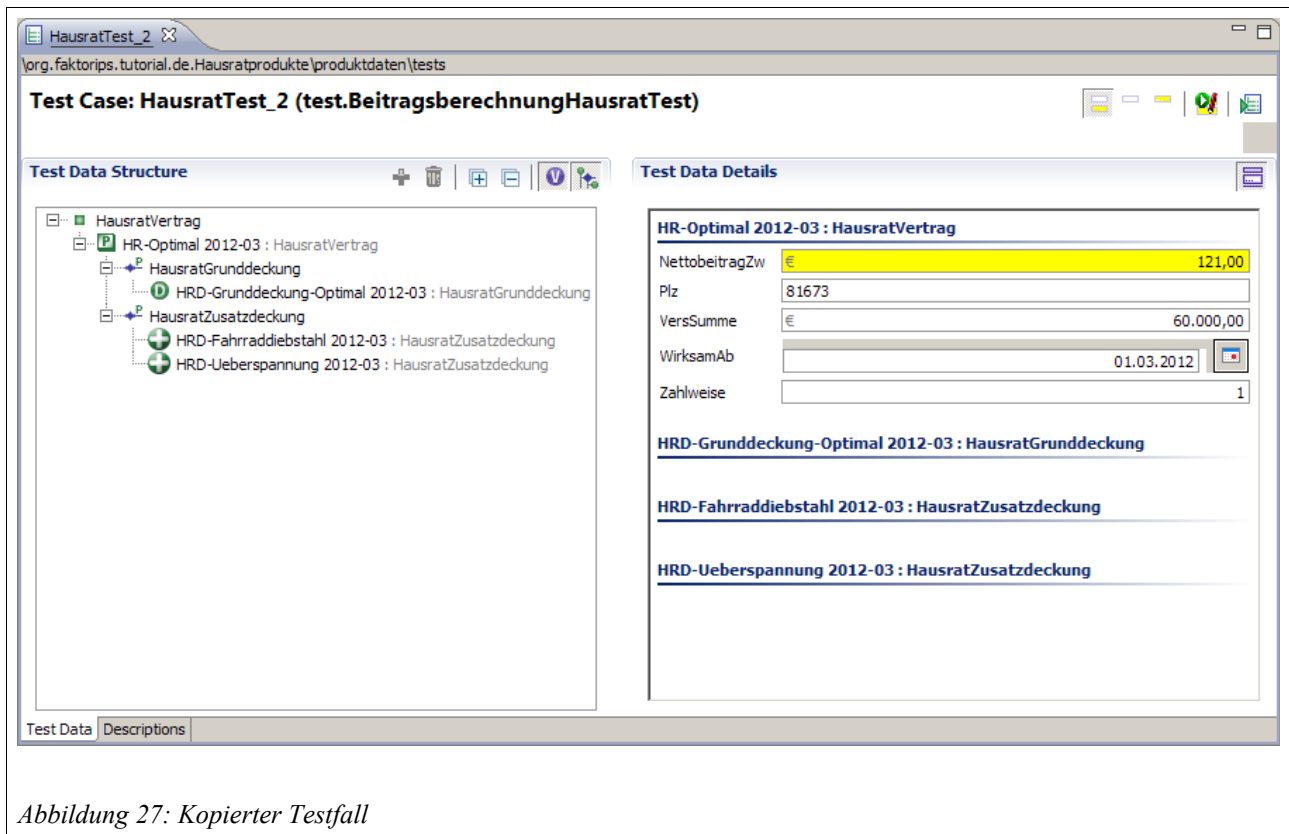


Abbildung 27: Kopierter Testfall

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

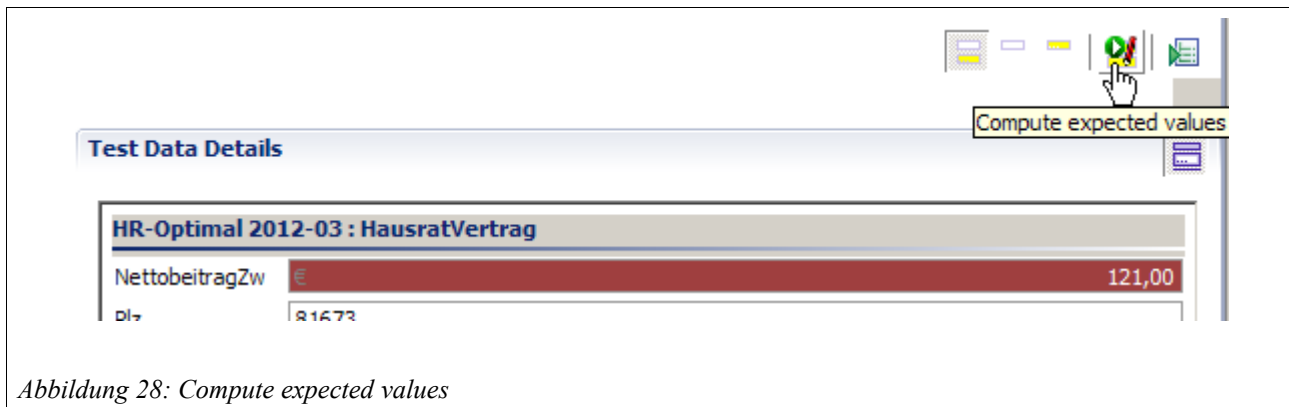


Abbildung 28: Compute expected values

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

HR-Optimal 2012-03 : HausratVertrag	
NettobeitragZw	133,00
Plz	81673
VersSumme	€ 60.000,00
WirksamAb	01.03.2012 
Zahlweise	1

HRD-Grunddeckung-Optimal 2012-03 : HausratGrunddeckung

HRD-Fahrraddiebstahl 2012-03 : HausratZusatzdeckung

HRD-Ueberspannung 2012-03 : HausratZusatzdeckung

Abbildung 29: Ermittelte Werte wurden in Testfall übernommen

JUnit-Adapter für Faktor-IPS Testfälle

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

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

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

import javax.xml.parsers.ParserConfigurationException;

import junit.framework.Test;

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

public class HausratJUnitTest extends IpsTestSuiteJUnitAdapter {

    public static Test suite() throws ParserConfigurationException {
        IRuntimeRepository repositoryHausrat = new ClassloaderRuntimeRepository(
            HausratTestAdapter.class.getClassLoader(),
            "org.faktorips.tutorial.produktdaten.internal");

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

Wir erzeugen ein RuntimeRepository mit den Produktdaten, lassen uns durch den Aufruf der Methode `getIpsTest()` eine Testsuite mit allen Testfällen aus dem Repository geben. Die Methode `createJUnitTest(...)` der Klasse `IpsTestSuiteJUnitAdapter` erzeugt daraus eine JUnit-Testsuite. Führen wir die Testklasse `HausratJUnitTest` mit dem JUnit-Testrunner aus, sehen wir, wie unsere Faktor-IPS-Testfälle von JUnit ausgeführt und interpretiert werden.

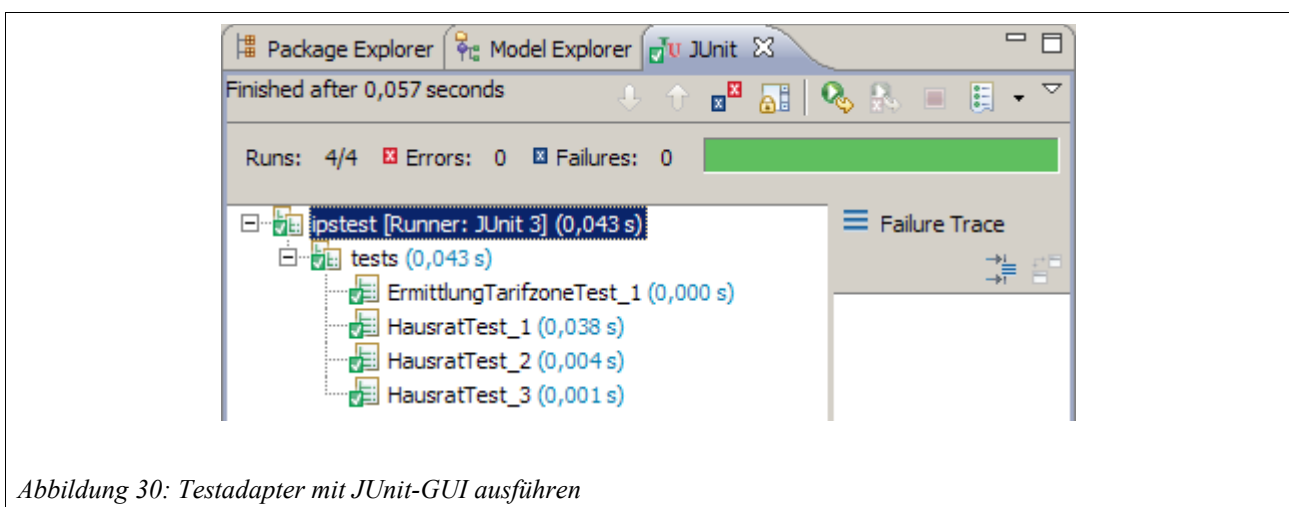
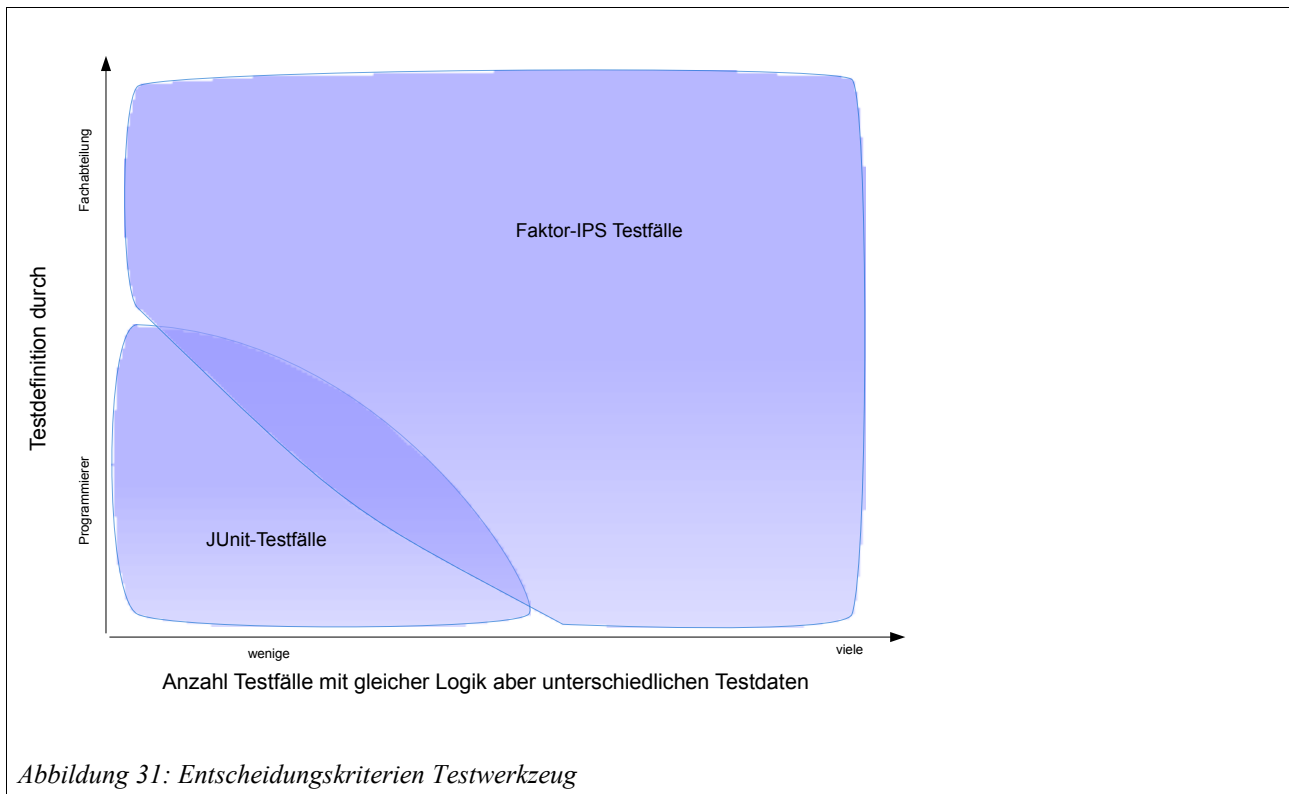


Abbildung 30: Testadapter mit JUnit-GUI ausführen

Zusammenfassung

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



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

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

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