Tutorial Angebotssystem

Anwendungsentwicklung mit Faktor-IPS

Gunnar Tacke
(Dokumentversion 1100)

Motivation

Das einführende Tutorial zu Faktor-IPS [1] erläutert anhand des Beispiels einer vereinfachten Hausratversichung das Arbeiten mit Faktor-IPS. Es wird die Klassenmodellierung, Codegenerierung, die Implementierung von fachlichen Methoden und die Konfiguration von Produkten vorgestellt.

Dieses weiterführende Tutorial erläutert nun, wie man auf Basis der Modellklassen eine Anwendung entwickeln kann und wie man in der Anwendung die Produktinformationen verwendet.

Ziel ist es dabei, die Anwendung so zu schreiben, dass die Anwendung auf Produktänderungen der Fachabteilung reagiert, ohne dass der Programmcode angepasst werden muss. Teil 1 des Tutorials zeigt, wie man auf Produktänderungen in der Anwendung reagiert.

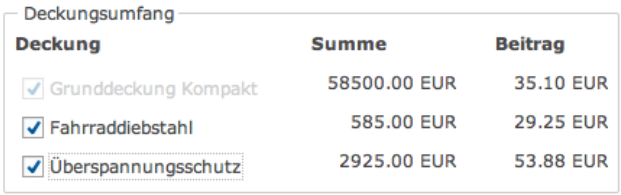

Abbildung 1: Die Anwendung reagiert auf Produktdaten-Änderungen: Zusatzdeckungen werden dynamisch angezeigt

Im zweiten Teil wird die Anwendung so erweitert, dass auch auf Modelländerungen dynamisch reagiert werden kann und so z.B. weitere Modellattribute ohne Änderungen an der Anwendung hinzugefügt werden können.

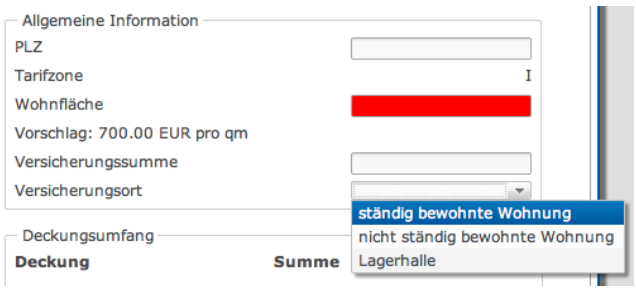

Abbildung 2: Die Anwendung stellt das im Modell hinzugefügte Attribut Versicherungsort dar

Im Kapitel Inbetriebnahme der Beispielanwendung ist beschrieben, wie man die bereitgestellte Anwendung lokal starten kann.

Teil 1: Die Beispielanwendung

Als Beispielanwendung bauen wir einen Ausschnitt eines Angebotssystems für Hausratversicherungen.

Das System soll es dem Anwender ermöglichen, eine neues Angebot zu erstellen und eine Beitragsberechnung durchzuführen. Dazu gibt der Anwender als erstes den gewünschten Versicherungsbeginn an. Abhängig davon füllt das System eine Auswahlliste mit den zu diesem Termin gemäß der Produktkonfiguration angebotenen Produkten (Abbildung 3).

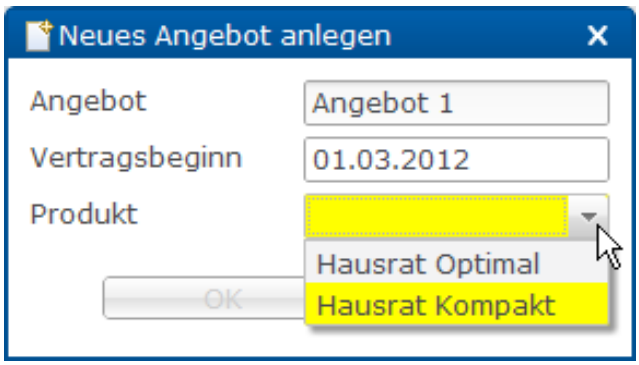

Abbildung 3: Produktauswahl

Nachdem der Anwender das gewünschte Produkt gewählt hat, zeigt das System ein Eingabeformular mit den für dieses Produkt relevanten Feldern. Am Ende des Eingabeformulars werden die errechneten Beitragszahlungen ausgegeben und fortlaufend aktualisiert, während der Anwender seine Daten eingibt oder ändert. Siehe hierzu Abbildung 4: Eingabefelder sind blau markiert, berechnete Felder gelb.

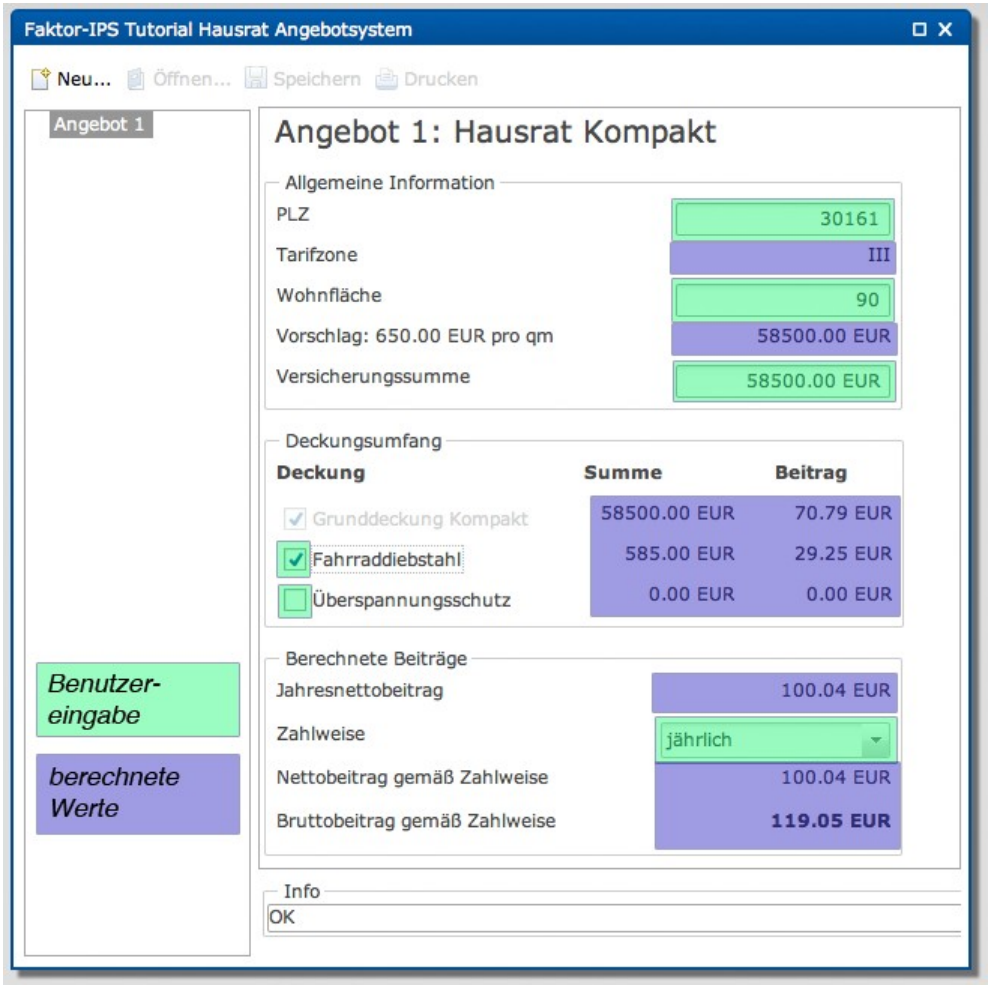

Abbildung 4: Eingabeformular mit Ausgabe der berechneten Beiträge

In dem Formular werden die folgenden Produktinformationen verwendet:

- Vorschlagswert für einen Quadratmeter Versicherungssumme
  Für das Produkt HR-Kompakt 600Euro, für HR-Optimal 900Euro.
- Die Liste der erlaubten Deckungen
  In beiden Produkten sind die Deckungen Fahrraddiebstahl und Überspannungsschutz erlaubt. In HR-Kompakt sind sie allerdings optional, während sie in HR-Optimal nicht optional sind. In letzterem Fall werden die Deckungen automatisch im Angebot berücksichtigt und die Checkboxen sind disabled, da die Deckungen nicht abgewählt werden können.
- Die Liste der erlaubten Zahlweisen
  In HR-Kompakt sind nur halbjährlich und jährlich erlaubt, in HR-Optimal auch noch monatlich und vierteljährlich.

- Der Defaultwert für die Zahlweise
  In beiden Produkten jährlich.

Szenario: Anlegen eines neuen Produktes

Die Beispielanwendung wurde so implementiert, dass sie ohne Codeänderung mit neuen Produkten oder Produktänderungen umgehen kann. Um dies zu zeigen, wollen wir mit Faktor-IPS ein neues Produkt anlegen, es mit einer weiteren Zusatzdeckung versehen und darauf hin unsere Beispielanwendung neu starten.

Da wir lediglich Produktdaten und keinen Code ändern müssen, können wir in der Faktor-IPS Produktdefinitionsperspektive arbeiten, die speziell für die Arbeit der Fachabteilungen mit Produktdaten ausgelegt ist.

Als neues Produkt wollen wir das Produkt „Hausrat Plus“ implementieren, das im Vergleich zu den bereits vorhandenen Produkten eine optionale Zusatzdeckung für Elementarschäden (z. B. Schäden durch Überschwemmung oder Erdbeben, die in Hausratprodukten in der Regel ausgeschlossen sind) beinhaltet. Für unser Szenario nehmen wir an, dass die Versicherungssumme der des Vertrags entspricht und das der Beitragssatz 0,23 € je 1000 € Versicherungssumme beträgt.

Neues Produkt durch Kopie erstellen

Hierzu werden wir das Produkt HR-Optimal mit Faktor-IPS kopieren. Dazu starten wir im Kontextmenü des Produktdefinitions-Explorers auf dem Produkt HR-Optimal mit Copy Product... den Copy-Wizard von Faktor-IPS (Abbildung 5).

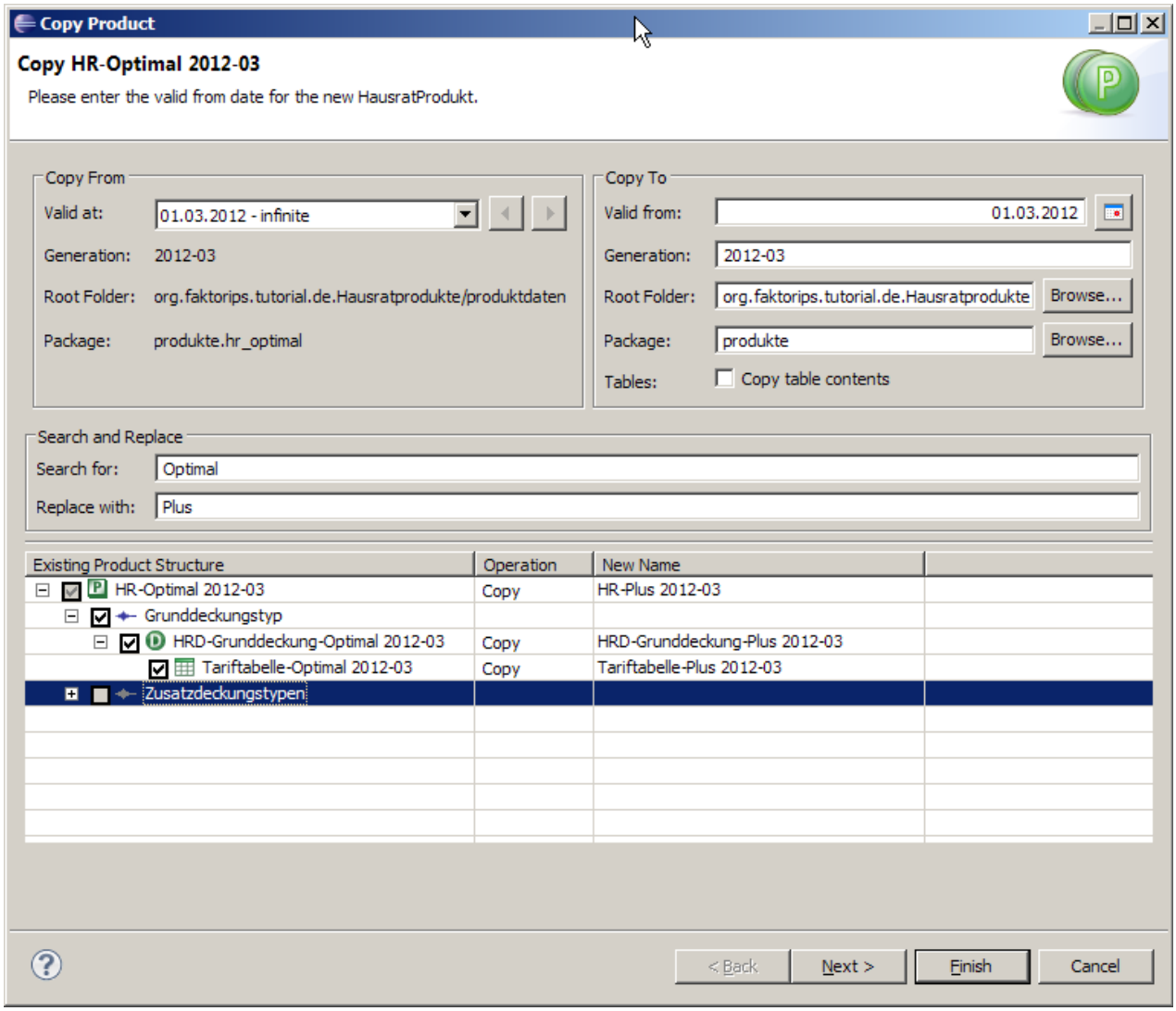

Abbildung 5: Produkt Hausrat Plus durch Kopie erzeugen

Die Grunddeckung wird mitkopiert, die beiden Zusatzdeckungen nicht; diese werden wir wiederverwenden. Jetzt ändern wir noch den Produktnamen in Hausrat Plus und speichern unsere Änderungen.

Zusatzdeckung anlegen

Nun legen wir die Zusatzdeckung Elementarschäden an, die wir in einem zweiten Schritt unserem neuen Produkt zuordnen werden.

Im Faktor-IPS Model Explorer wird im Projekt org.faktorips.tutorial.de.Hausratprodukte im package `hr_zusatzdeckungen` eine neue Deckung angelegt (Abbildung 6).

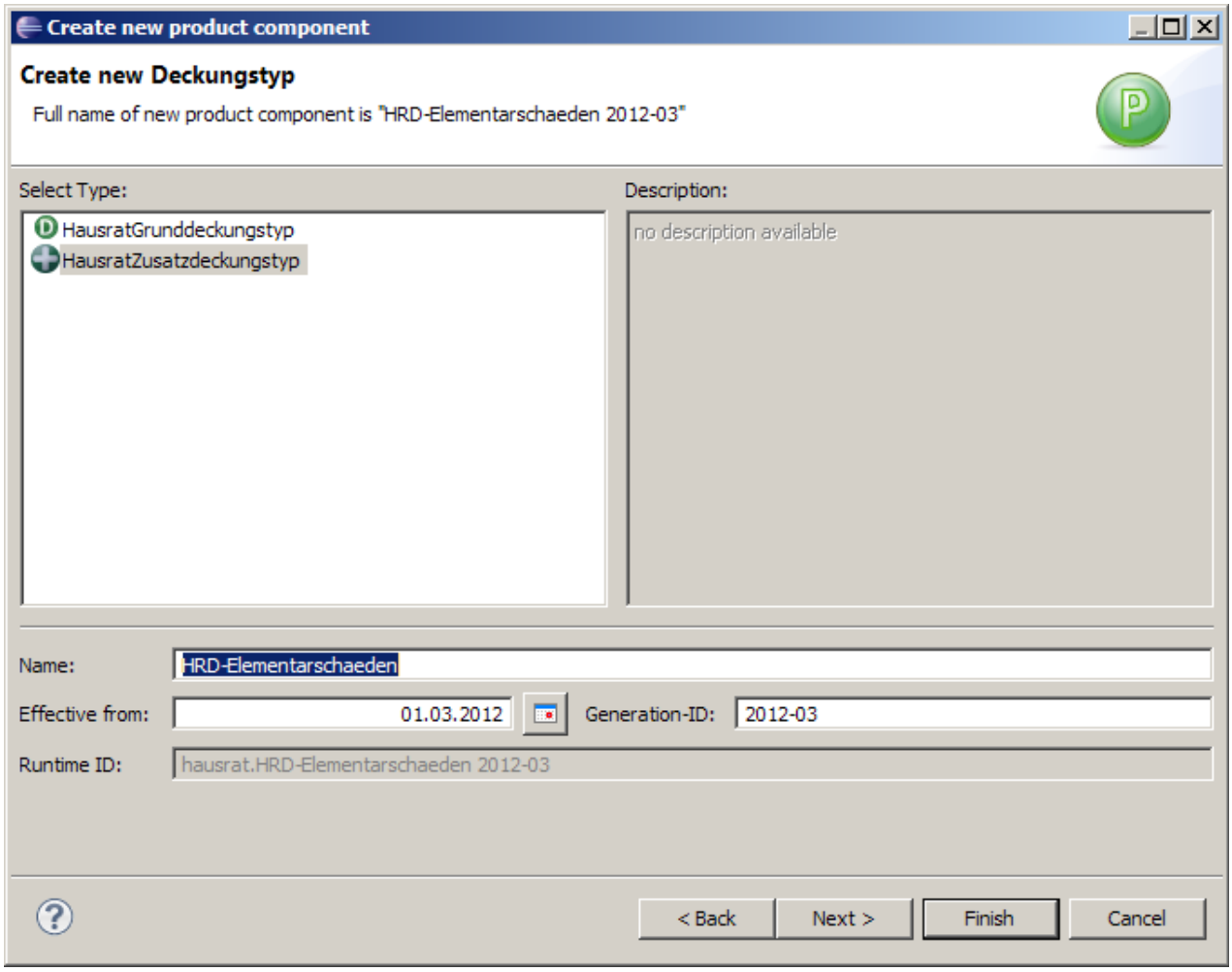

Abbildung 6: Anlegen einer der Zusatzdeckung HRD-Elementarschaeden

Die Deckung bekommt den VersSummenFaktor 1 (d.h. Versicherungssumme der Zusatzdeckung = Versicherungssumme des Vertrags) und die Berechnung des Jahresbasisbeitrag wird durch die Formel „*deckung.versSumme / 1000 * 0.23*“ definiert (Abbildung 7).

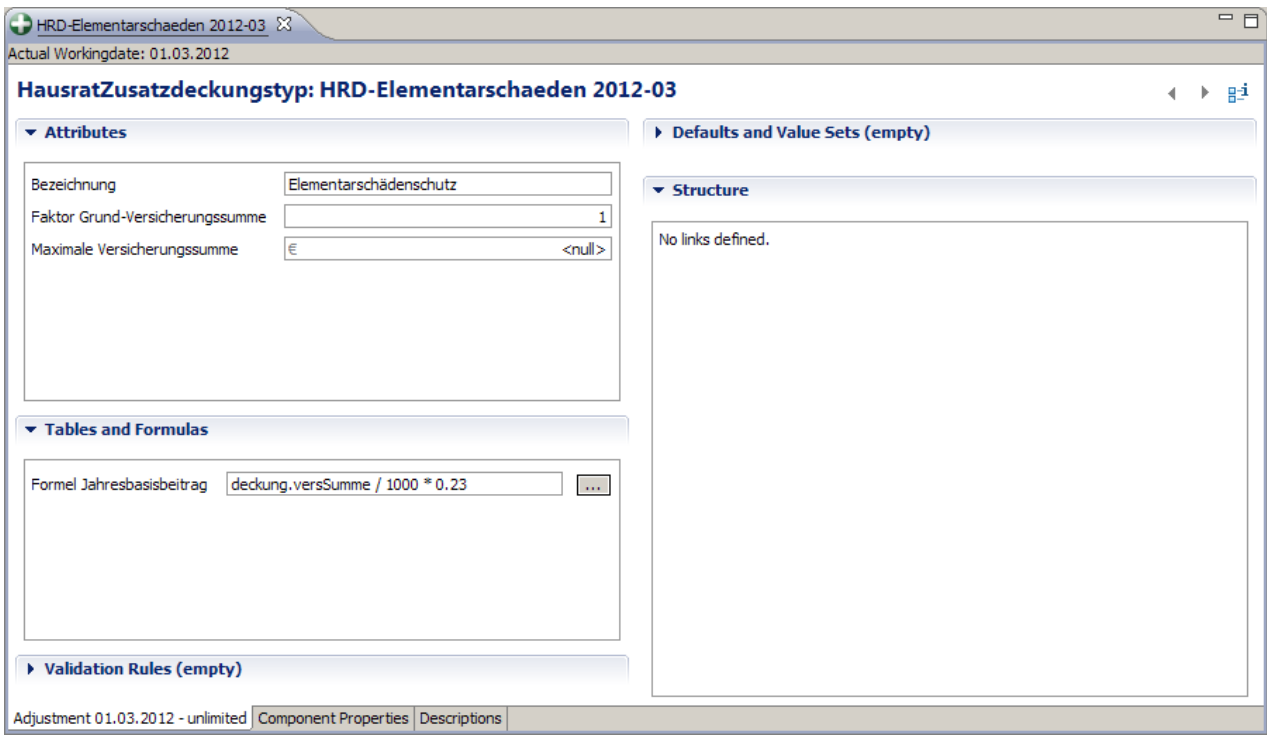

Abbildung 7: Definition der Zusatzdeckung HRD-Elementarschaeden

Jetzt müssen wir unserem neuen Produkt noch die Zusatzdeckungen zuordnen. Dies können wir per Drag-And-Drop in der Produktdefinitions-Perspektive erledigen(Abbildung 8).

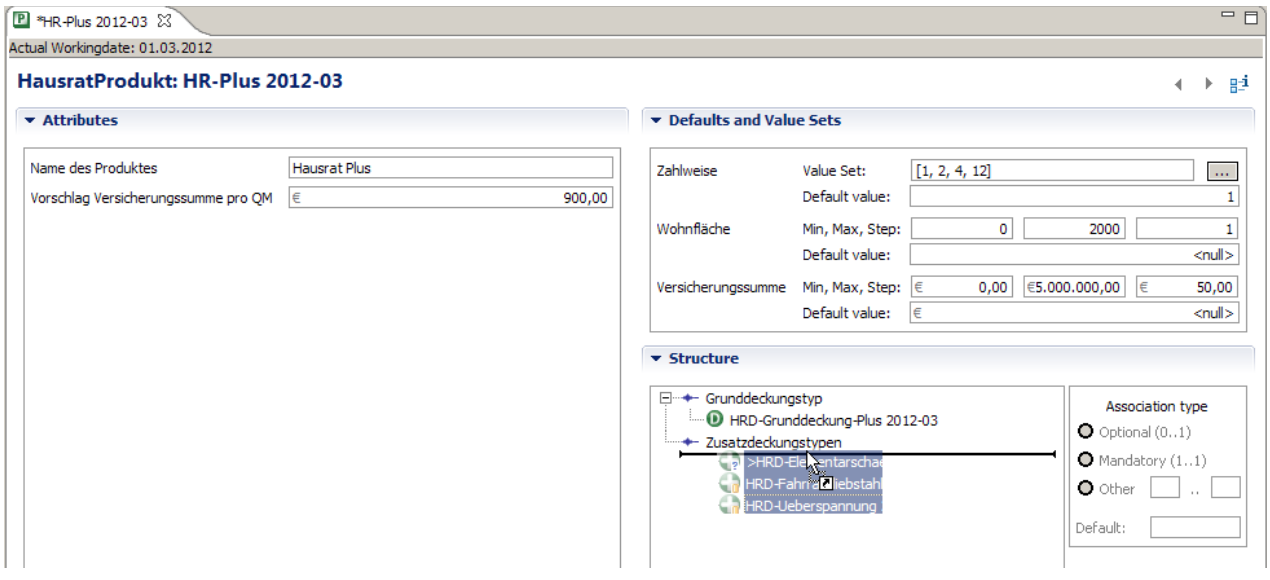

Abbildung 8: Zusatzdeckungen dem Produkt per Drag&Drop zuordnen

Um die Änderungen im Angebotssystem zu sehen, starten wir die Anwendung neu. Wir bekommen nun drei Produkte zur Auswahl, darunter das eben angelegte *Hausrat Plus*(Abbildung 9).

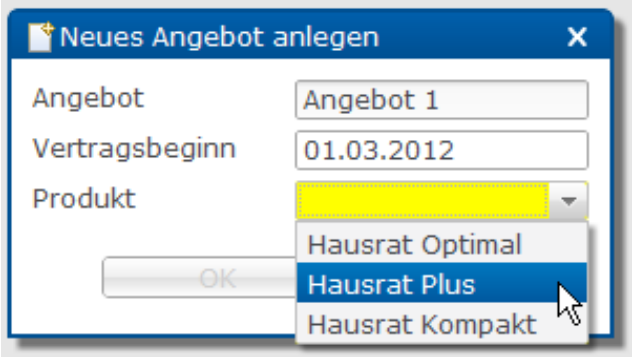

Abbildung 9: neues Produkt steht im Angebotssystem zur Auswahl

Wählen wir unser neues Produkt und bestätigen mit OK, so sehen wir auf der folgenden Angebotserfassungsmaske die Eigenschaften des Produktes (z.B. den Produktnamen in der Überschrift, den Vorschlag der Versicherungssumme pro m²) und die neu angelegte und dem

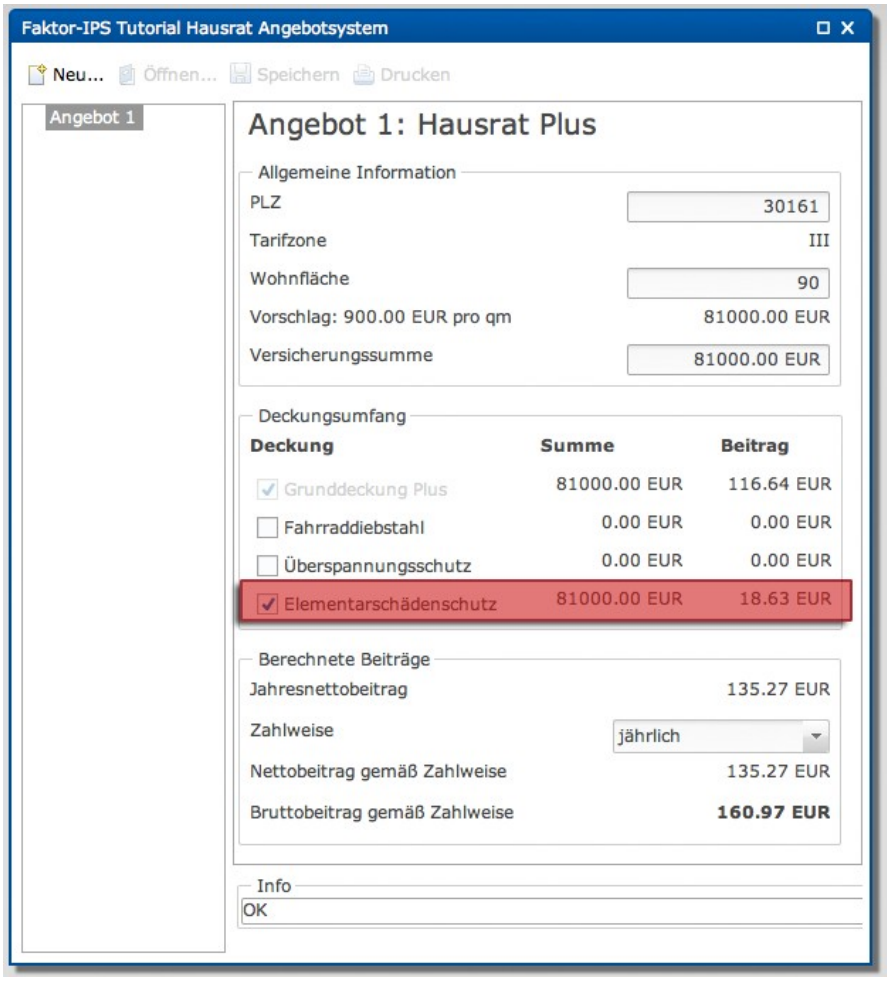

Abbildung 10: Das neue Produkt Hausrat Plus im Dialog des Angebotssystem

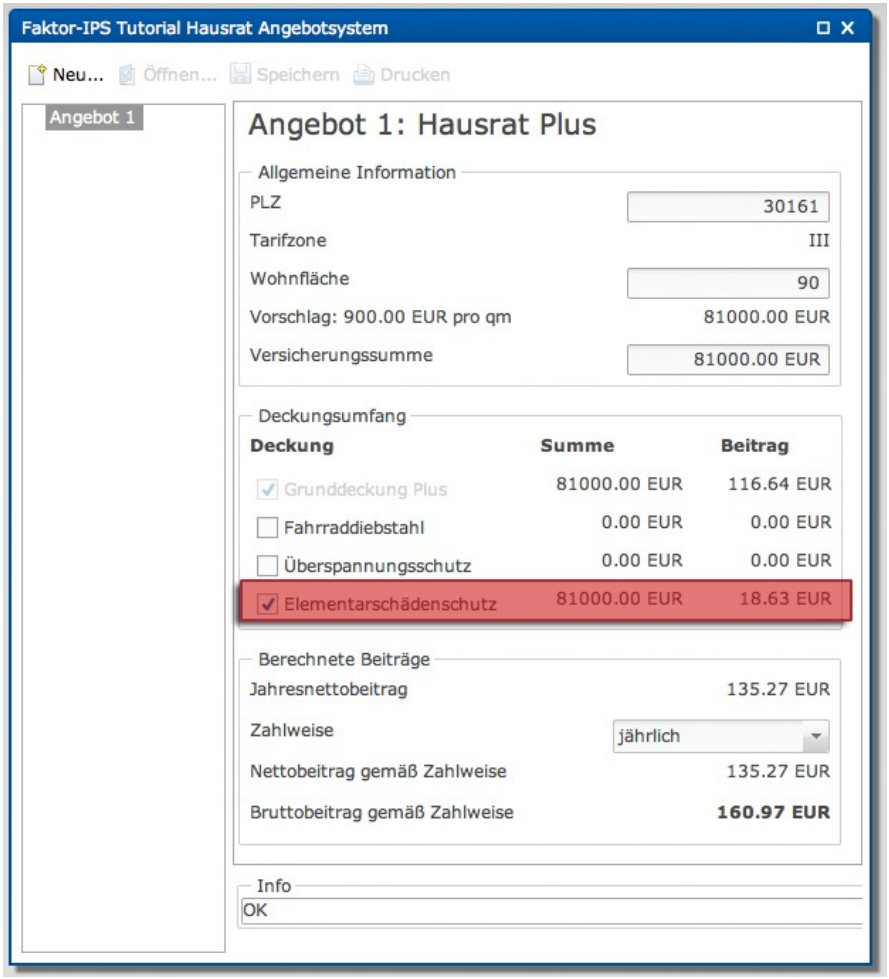

Abbildung 10: Das neue Produkt Hausrat Plus im Dialog des Angebotssystem

Produkt zugeordnete Zusatzdeckung Elementarschäden-Schutz (Abbildung 10).

Wir sehen, dass die Versicherungssumme der Elementarschädendeckung mit der eingegebenen Versicherungssumme übereinstimmt (Versicherungssummenfaktor =1) und dass der Basisbeitrag der Deckung 0,23 € je 1000 € des gesamten Jahresnettobeitrags ausmacht.

Architektur der Beispielanwendung

AJAX-Webanwendung mit Eclipse RAP

Die bisher gezeigten Screenshots haben es noch nicht verraten: die Beispielanwendung ist implementiert als AJAX-Webanwendung. Als UI-Toolkit / Framework wurde dabei Eclipse RAP (Rich AJAX Platform) eingesetzt [3].

Eclipse RAP besteht – vergleichbar mit anderen AJAX-Frameworks – zum einen aus einer Komponente, die im Webbrowser läuft und anderseits aus einer (Java-)Webserver-seitigen Komponente. Die im Web-Browser laufende AJAX-Engine kümmert sich dabei v.a. um die Event-

Behandlung, die UI-Updates und die Kommunikation mit dem Server.

Die serverseitige Komponente bietet dem Anwendungsentwickler letztendlich ein von der „normalen“ Client-Entwicklung her gewohntes Programmiermodell, d. h. es gibt UI-Widgets wie Buttons und Listboxes mit entsprechender Event-Behandlung. Im Fall von Eclipse RAP sind dies die gewohnten SWT-Komponenten, die von RAP auf die AJAX Plattform umgesetzt wurden.

Anwendungsschichten

Unser Tarifrechner setzt auf der beschriebenen Eclipse RAP auf[1] und läuft in einem Servlet Container. Das Angebotssystem verwendet die RAP-SWT-Komponenten zur Implementierung der Benutzerschnittstelle. Die UI-Schicht arbeitet i.d.R. direkt mit dem Domain Model. Für bestimmte Modellklassen gibt es ein Presentation Model auf UI-Seite, das die Darstellung und Verhaltensweise einer Domain Klasse auf UI-Seite kapselt. Zum Domain Model zählt hier auch die Faktor-IPS-Runtime, durch die z. B. auf die mit Faktor-IPS konfigurierten Produktdaten zugegriffen wird.

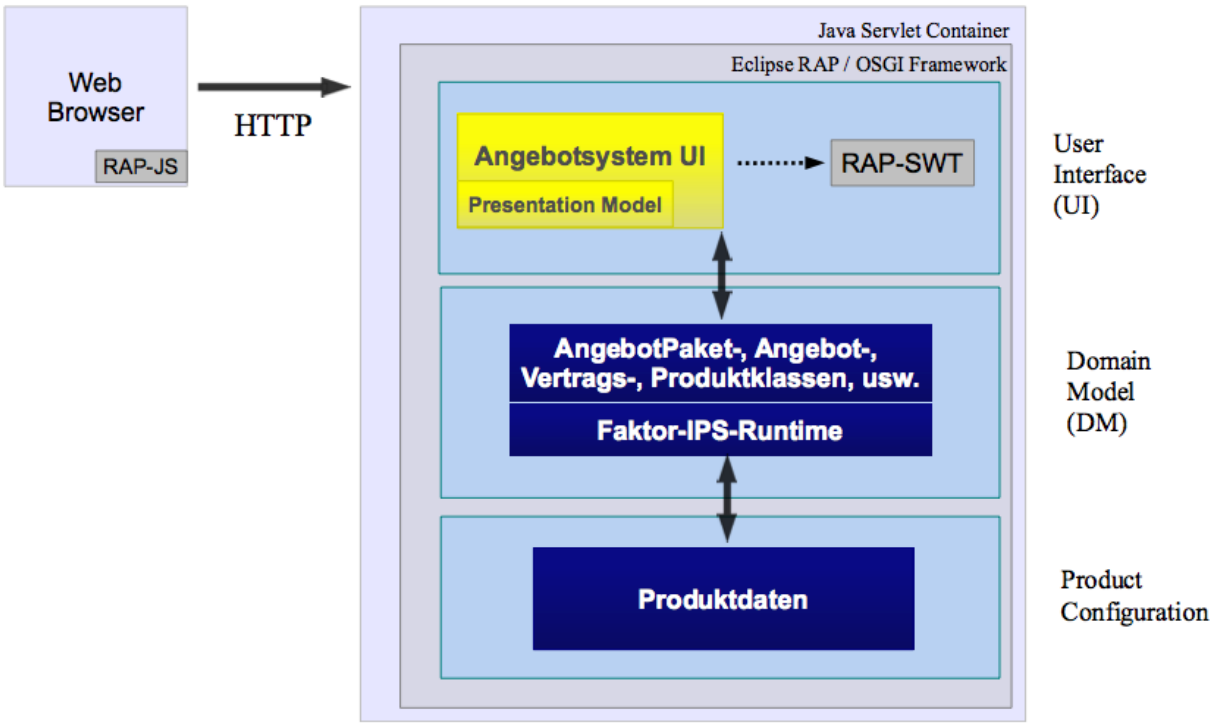

Abbildung 11: Architektur der Beispielanwendung

Abbildung 11 zeigt die Bestandteile des Systems und die Schichtung der Anwendung im Überblick.

1 Eclipse RAP wiederum verwendet das OSGi Framework

Domain Model

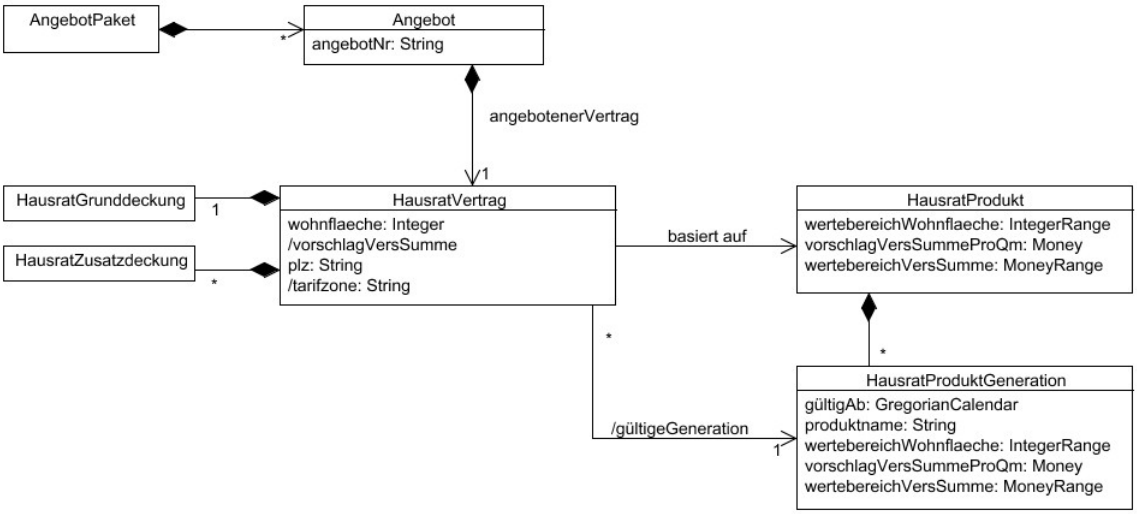

Abbildung 12: Domain Model des Angebotssystems

Die im Faktor-IPS Einführungs-Tutorial erstellten Klassen dienen als Grundlage für das Domainmodel des Angebotssystems. Wenn Sie das Tutorial durchgeführt haben, können Sie mit dem Eclipse-Projekt zu diesem Tutorial sehen, wie ihr Hausratmodell und die modellierten Produkte in einem Angebotssystem aussehen können. Für das Angebotssystem wird das Modell um die Klassen Angebot und AngebotPaket erweitert.

Zentrale Klasse des Angebotssystems ist das Angebot. Ein Angebot referenziert einen HausratVertrag-Objekt in der Rolle „angebotenerVertrag“ und hat als Attribut eine ID (angebotNr), mit deren Hilfe ein Angebot eindeutig identifiziert werden kann.

Im GUI können mehrere Angebote „nebeneinander“ angelegt werden. Diese werden durch die Klasse AngebotPaket zusammengefasst und durch eine angebotNr identifiziert.

Fachliche Methoden des Domain Modells

Das Domain Modell enthält die folgenden fachlichen Methoden:

- Ermittlung Tarifzone
  Die Tarifzone wird auf Ebene des Vertrags anhand der PLZ bestimmt. Die Tarifzonen sind in einer Tarifzonentabelle abgelegt, die vom Vertrag referenziert wird. Wird die PLZ nicht in der Tabelle gefunden, wird die Tarifzone „I“ verwandt. (s. Implementierung `HausratVertag.getTarifzone()` im Faktor-IPS-Tutorial)
- Berechnung Vorschlag für die Versicherungssumme
  Die Versicherungssumme des Vertrags wird berechnet aus der vorgeschlagene Versicherungssumme pro qm (definiert auf Produktebene) multipliziert mit der eingegebenen Wohnfläche (implementiert in `Hausratvertrag.getVorschlag-VersSumme()`).

- Beitragsberechnung
  Der Jahresbasisbeitrag (Beitrag p.A. ohne Steuer, ohne Ratenzahlungszuschlag) eines Vertrags ergibt sich aus der Summe der Jahresbasisbeiträge aller eingeschlossenen Deckungen. Der Beitrag wird gecached und durch den Aufruf der Methode `Vertrag.berechneBeitrag()` aktualisiert. Dabei wird jeweils die Methode `Deckung.berechneBeitrag()` aller referenzierten Deckungen aufgerufen und danach der Jahresbasisbeitrag der Deckungen summiert.

Konzepte

Bei der Entwicklung einer Anwendung, in der Faktor-IPS verwendet wird, kommen verschiedene Konzepte zum Einsatz. Im Folgenden werden die wichtigsten Konzepte anhand der Beispielanwendung vorgestellt.

Anwendung als Eclipse Plug-In

RAP verfolgt das Ziel, die Entwicklung von AJAX-Webanwendungen im Stil von Eclipse zu ermöglichen: auf RAP basierende Anwendungen werden grundsätzlich als Eclipse Plug-in[2] entwickelt. Dies hat eine wichtige Entscheidung zur Strukturierung der Eclipse-Projekte zur Folge.

Unsere Beispielanwendung benötigt zur Compile- und zur Laufzeit die von Faktor-IPS generierten Klassen aus dem Basis- und dem Hausrat-spezifischen Projekt. Als weitere Abhängigkeit kommt das Produktdaten-Projekt hinzu. Es gibt mehrere Möglichkeiten, um die abhängigen Projekte zum Build und Runtime-Classpath eines Eclipse Plug-ins hinzuzufügen.

Der erste Lösungsansatz, nämlich das Erstellen und Kopieren von JAR-Dateien mit den Klassen/Dateien der abhängigen Projekte in das eigene Plug-in-Projekt, wurde verworfen. Dieser Ansatz wäre sinnvoll, wenn es in den abhängigen Projekte sehr selten Änderungen gibt, wie z. B. bei der Einbindung von Third-Party-Software mit definierten Release-Zyklen. Während der Entwicklung des Angebotssystems wurden jedoch gleichzeitig immer wieder Anpassungen in den abhängigen Projekten gemacht, so dass das häufige Erstellen und Kopieren der JAR-Dateien die Entwicklung etwas aufwendiger gemacht hätte.

Stattdessen wurden schließlich alle abhängigen Projekte selbst zu Plug-in-Projekten gemacht. Vorteil dieser Lösung ist, dass die Build- und Runtime-Abhängigkeiten zu einem anderen Plug-in-Projekt einfach im Plug-in-Manifest eingetragen werden. Eclipse setzt dann Build- und Runtime-Classpath entsprechend dieser Konfiguration.

Anmerkung
Die beschriebene Umsetzung als Eclipse-Plug-in ist bedingt durch den Einsatz des Eclipse RAP AJAX-Frameworks. Selbstverständlich können Anwendungen, die Faktor-IPS verwenden, auch anders gebaut werden, z. B. mit Java Swing. Die Anwendungsentwicklung ist somit auch von Eclipse unabhängig.

2 Es handelt sich um eine Anwendung für das Equinox OSGi Framework

Das Faktor-IPS Runtime-Repository

Das Angebotssystem lässt den Benutzer bei der Erstellung eines neuen Angebots aus der Liste der verfügbaren Produkte wählen (Abbildung 4). Es werden dabei Methoden der Faktor-IPS-Runtime benutzt, um die zu Grunde liegende Produktdaten auszulesen.

Intern sind die Produktdaten im Faktor-IPS-Runtime-Repository gespeichert, das die Produktdaten (derzeit) in Form von XML-Dateien enthält: für jeden Produktbaustein und jede Tabelle gibt es jeweils eine eigene XML-Datei. Eine spezielle XML-Datei, faktorips-runtime-toc.xml, enthält das Verzeichnis aller Produktbausteine und Tabellen und referenziert die zugehörigen XML-Dateien.

Zugriff auf das Runtime-Repository

Der Zugriff auf das Runtime-Repository erfolgt über Methoden des Interfaces `IRuntimeRepository`. Eine Implementierung des Interfaces für den Zugriff auf das im Dateisystem gespeicherte Repository erhält man durch Aufruf der Factory-Methode `ClassloaderRuntime-Repository.create(String tocResource)` wie im folgenden Beispiel gezeigt wird:

```
String REPOSITORY_TOC_XML = "org/faktorips/tutorial/produktdaten/"
                      + "internal/faktorips-repository-toc.xml";

IRuntimeRepository repository =
       ClassloaderRuntimeRepository.create(REPOSITORY_TOC_XML);
```

Vorsicht bei mehreren Classloadern

Das `ClassloaderRuntimeRepository` lädt über den Classloader zunächst die angegebene Inhaltsverzeichnis-Datei und bei Bedarf weitere XML-Dateien aus dem Dateisystem.

Wenn das System mehrere Classloader benutzt, um z. B. Komponenten besser voneinander abzuschirmen, muss man sicherstellen, dass der verwendete Classloader auch diese Dateien finden kann. Dies trifft insbesondere bei Eclipse Plug-ins zu: Eclipse lädt Plug-ins in separaten Classloadern und lässt nur Zugriffe auf exportierte Java-Packages zu[3].

Um sicherzustellen, dass der richtige Classloader zum Laden der Dateien benutzt wird, haben wir in der Beispielanwendung eine Factory-Klasse im Produktdatenprojekt abgelegt.

3 Eclipse kann außerdem bei Verwendung des Plug-in-Manifest-Editors nur Packages eines Plug-ins exportieren, in dem sich tatsächlich Java-Klassen befinden.

```
package org.faktorips.tutorial.produktdaten;

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

public class RuntimeRepositoryFactory {

	/** Pfad zum Faktor-IPS repository. */
	static final String REPOSITORY_TOC_XML = "org/faktorips/tutorial/produktdaten/"
			+ "internal/faktorips-repository-toc.xml";
	/**
	 *
	 * Gibt das Faktor-IPS Runtime-Repository zurück, dass die im Tutorial
	 * erstellte Produktkonfiguration enthält.
	 *
	 * @return The {@code IRuntimeRepository}.
	 */
	public static IRuntimeRepository createRuntimeRepository() {
		IRuntimeRepository repository = ClassloaderRuntimeRepository.create(
				RuntimeRepositoryFactory.REPOSITORY_TOC_XML,
				RuntimeRepositoryFactory.class.getClassLoader());

		return repository;
	}
}
```

`RuntimeRepositoryFactory.createRuntimeRepository()` verwendet den Classloader, der die Factory-Klasse selbst geladen hat. Da sich diese im gleichen Plug-in wie die Produktdaten befindet, kann dieser Classloader auch auf die Produktdaten zugreifen.

Auslesen der Produktbausteine

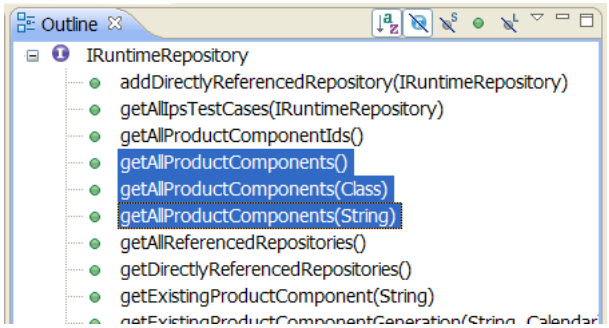

Abbildung 13: Methoden des IRuntimeRepository Interfaces

Das Runtime-Repository kann nun mit den Methoden des Interfaces `IRuntimeRepository` ausgelesen werden (Abbildung 13). Die Methode `getAllProductComponents()` liefert z. B. die Liste aller Produktbausteine.

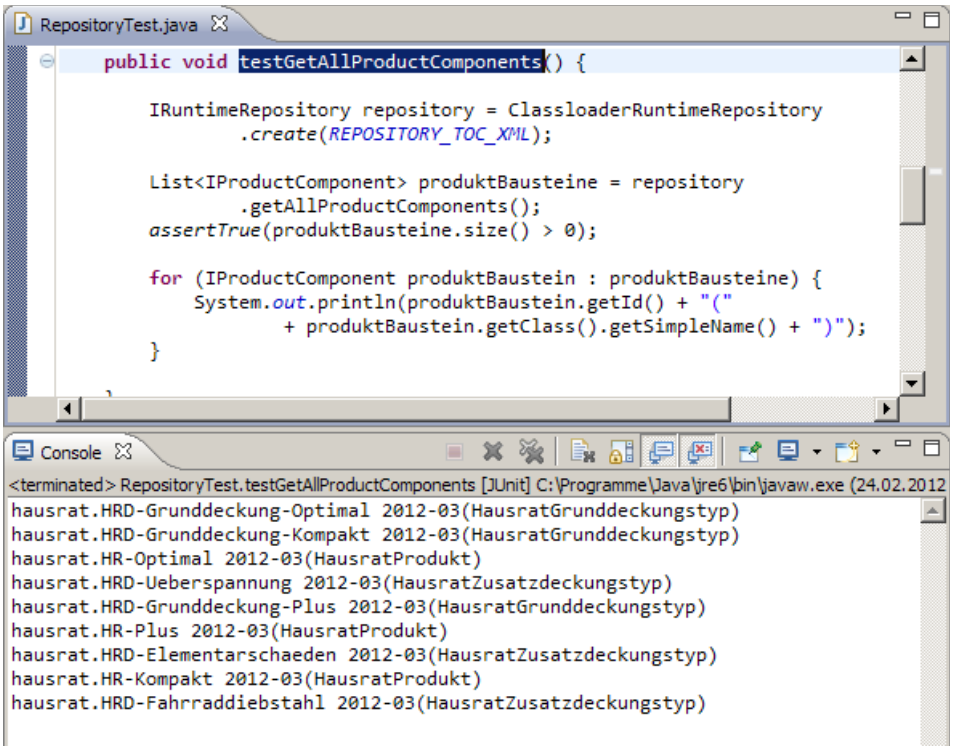

Abbildung 14: Aufruf von getAllProductComponents()

Die Methode `getAllProductComponents()` liefert z. B. die Liste aller Produktbausteine. Abbildung 14 zeigt die Ausgabe von `getAllProductComponents()` für unsere Beispiel-Produktdaten.

Da `getAllProductComponents()` grundsätzlich alle Produktbausteine (Interface `IProduct-`

`Component`) liefert, müssen wir auf unsere Hausratversicherung Produkte einschränken. Dazu testen wir, ob der Produktbaustein das gemeinsame published Interface unserer Hausratprodukte implementiert. Die Methode `getAllProductComponents(Class productComponentType)` implementiert genau diese Filterung.

Typsicheres Runtime-Repository

Die beschriebene Einschränkung auf Hausratprodukte kapseln wir in einer Hausrat-spezifischen Klasse `HausratProduktRepository`, wo wir zudem prüfen, ob das Produkt zum ausgewählten Zeitpunkt verfügbar ist. Des weiteren verwenden wir als Rückgabetyp nicht mehr eine mit `IProductComponent`-typisierte Liste, sondern eine vom Typ `IHausratProdukt`.

```
package org.faktorips.tutorial.hausrat.Angebotssystem.model;
// ...
public class HausratProduktRepository {
	// ...

/**
 * Sucht alle im Faktor-IPS-Repository vorhandenen und zum spezifizierten
 * Wirksamkeitsdatum verwendbaren Hausratprodukte.
 *
 * @param wirksamkeitsdatum
 *
 * @return die verwendbaren Produkte.
 */
public List<IHausratProdukt> getAvailableProducts(Calendar wirksamkeitsdatum) {
	List<IProductComponent> alleProdukte = repository
			.getAllProductComponents(IHausratProdukt.class);
	List<IHausratProdukt> verwendbareProdukte = new ArrayList<IHausratProdukt>();

	for (IProductComponent einProdukt : alleProdukte) {
		IHausratProdukt hausratProdukt= (IHausratProdukt)einProdukt;
		IHausratProduktGen gen = hausratProdukt.getHausratProduktGen(wirksamkeitsdatum);
		if (gen != null){
			verwendbareProdukte.add(hausratProdukt);
		}
	}
		return verwendbareProdukte;
	}
}
```

`HausratProduktRepository` ist als Singleton implementiert. Die Instanz wird geholt durch die hier nicht gezeigte Methode `HausratProduktRepository.getRepository()`.

Füllen der Produkte-Combobox

Der in Abbildung 4 gezeigte `NeuesAngebotDialog` kann nun die Produkt-Combobox mit den Produktnamen füllen, die aus der zum Stichtag geltenden Produktgeneration ausgelesen werden.

```
List<IHausratProdukt> hausratProdukte = HausratProduktRepository
                .getRepository().getAvailableProducts(vertragsbeginn);
int i = 0;
for (IHausratProdukt hausratProdukt : hausratProdukte) {
        String produktName = hausratProdukt.getHausratProduktGen(
                        vertragsbeginn).getProduktname();
        produktCombo.add(produktName, i);
        produktCombo.setData(produktName, hausratProdukt);
        if (currentProduktName.equals(produktName)) {
                produktCombo.select(i);
                setOkButtonEnabled(true);
        }
        i++;
}
```

Aufbau einer initialen Vertragsstruktur

Nach Auswahl des gewünschten Produkts erzeugt das Angebotssystem eine initiale Angebotsstruktur und füllt die Pflichtattribute und andere Felder, die zu diesem Zeitpunkt bekannt sind.

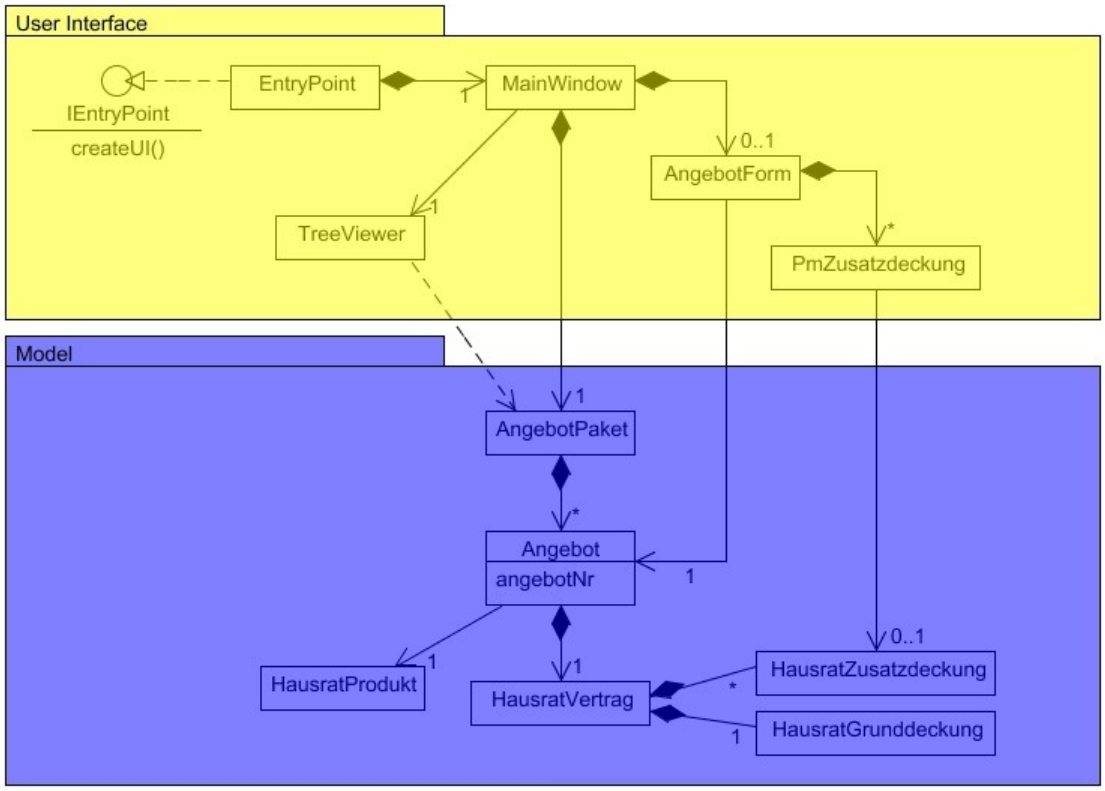

Abbildung 15: Klassendiagramm User Interface und Modell

Die Klasse `Angebot` erzeugt einen `HausratVertrag` und eine `HausratGrunddeckung`, die dem Vertrag hinzugefügt wird. Folgendes Code-Beispiel zeigt dies.

```
public class Angebot implements Comparable<Angebot> {

private String angebotNr;
private IHausratVertrag hausratVertrag;
/**
 * Erstellt ein neues Angebot {@code Angebot} für das gewählte
 * Hausrat-Produkt zu dem angegeben Wirksamkeitsdatum.
 *
 * @param angebotNr
 *            eine Angebotsnummer.
 * @param hausratProdukt
 *            ein Hausratprodukt.
 * @param wirksamkeitsdatum
 *            das Wirksamkeitsdatum.
 */
public Angebot(String angebotNr, IHausratProdukt hausratProdukt,
        GregorianCalendar wirksamkeitsdatum) {
        this.angebotNr = angebotNr;

                // Initiale Vertragsstruktur aufbauen
        hausratVertrag = hausratProdukt.createHausratVertrag();
        hausratVertrag.setWirksamAb(wirksamkeitsdatum);

        IHausratProduktGen hausratProduktGen = (IHausratProduktGen) hausratProdukt
                .getProduktGen(wirksamkeitsdatum);
        // Grunddeckung erzeugen
        IHausratGrunddeckungstyp hausratGrunddeckungstyp = (IHausratGrunddeckungstyp)
        hausratProduktGen.getHausratGrunddeckungstyp();
        IHausratGrunddeckung hausratGrunddeckung =
                hausratGrunddeckungstyp.createHausratGrunddeckung();
        hausratVertrag.setHausratGrunddeckung(hausratGrunddeckung);

        }
    // ...
}
```

Allgemein werden Instanzen von Vertragsklassen, die durch eine entsprechende Produktklasse konfiguriert werden, mit einer Factory-Methode der Produktklasse erzeugt: die `HausratVertrag`-Instanz wird von `IHausratVertrag.createHausratVertrag()` erzeugt, die `HausratGrunddeckung`–Instanz von `IHausratGrunddeckungstyp.createHausrat-Grunddeckung()`.

Nach Erzeugen der `HausratVertrag`-Instanz kann auf die Attribute der Klasse mit `get*()` und `set*()` zugegriffen werden. Obiges Beispiel setzt als Erstes das Pflichtattribut `wirksamAb`. Anschließend wird die `HausratGrunddeckung`-Instanz erzeugt; dazu wird erst die zum Stichtag geltende Produktgeneration geholt und von dieser die einzige[4] `HausratGrunddeckungstyp`-Instanz. Die `HausraGrunddeckung`-Instanz wird dann analog zum `HausratVertrag` durch die Factory-Methode `IHausratGrunddeckungstyp.createHausratGrunddeckung()` erzeugt.

4 Die Beziehung zwischen HausratVertrag und HausratGrunddeckung ist eine 1-zu-1-Beziehung. In der Produktkonfiguration ist daher jedem Produkt genau ein HausratGrunddeckungtyp zugeordnet.

Auslesen von Wertebereichen und Defaultwerten

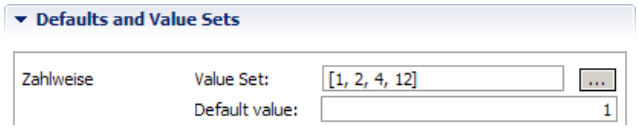

Abbildung 16: Konfiguration der Zahlweise im Produkt Hausrat Optimal in Faktor-IPS

In der Konfiguration unserer Beispielprodukte *Hausrat Kompakt* und *Hausrat Optimal* ist jeweils festgelegt, welche Werte für das Attribut *zahlweise* erlaubt sind und was der Defaultwert ist (Abbildung 16).

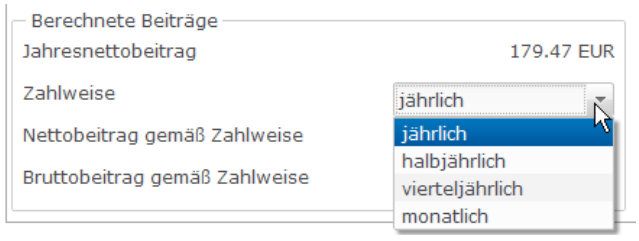

Abbildung 17: Combobox für die Zahlweise im Angebotssystem

Diese Konfiguration wollen wir aus dem Faktor-IPS-Runtime-Repository lesen, um im User Interface die zugehörige Combobox richtig zu füllen (Abbildung 17).

Der Wertebereich und der Defaultwert für ein produktseitig konfiguriertes Attribut können aus der Produktgeneration gelesen werden. Folgendes Code-Beispiel zeigt dies:

```
zahlweiseCombo = createCombo(beitraegeGroup, hausratProduktGen
                             .getAllowedValuesForZahlweise(null).getValues(true),
                             hausratProduktGen.getDefaultZahlweise(),
                             "QuoteForm.calculatedPremiums.modeOfPayment.", gridDataRight);
//...
/**
 * Erzeugt eine ComboBox, füllt sie und wählt den Default-Wert
 *
 * @param composite
 *            das übergeordnete {@code Composite}.
 * @param set
 *            die Werte, mit denen die ComboBox gefüllt wird.
 * @param vorbelegung
 *            Vorbelegung
 * @param messagePrefix
 *            Prefix, mit dem nach dem Muster "Prefix+itemValue" die Texte
 *            aus der messages.properties gelesen werden
 * @param layoutData
 *            Layoutdata
 *
 * @return {@code Combo} box.
 */
private Combo createCombo(Composite composite, Set<Integer> set,
                Object vorbelegung, String messagePrefix, Object layoutData) {
        Combo combo = new Combo(composite, SWT.DROP_DOWN | SWT.READ_ONLY);
                int i = 0;
        for (Object value : set) {
                String text = Messages.getString(messagePrefix + value);
                combo.add(text, i);
                combo.setData(text, value);
                if (value.equals(vorbelegung)) {
                        combo.select(i);
                }
                i++;
        }
//...
        return combo;
}
```

Zunächst wird die zu gewählten Stichtag gültige Produktgeneration geholt. Mit den von der Faktor-IPS-Runtime bereitgestellten Methoden `getAllowedValuesForZahlweise()` und `getDefaultZahlweise()` lesen wir die Produktkonfiguration aus. Erstere Methode liefert ein `OrderedValueSet<Integer>`; die darin enthaltenen Werte erhalten wir von `getValues()` als Integer-typisiertes Set.

Im Angebotssystem übersetzen wir für die Oberfläche die erlaubten Werte[5] mit Hilfe einer Property-Datei, die alle internationalisierbaren Strings enthält, in lesbaren Text.

Presentation Model für Zusatzdeckungen

Abbildung 18: Zusatzdeckungen an der Oberfläche

Das Modell der Klassen `HausratVertrag` und `HausratGrunddeckung` deckt sich im Wesentlichen

5 1, 2, 4, 12 für jährlich, halbjährlich, quartalsweise, monatlich, usw.

mit der Darstellung im User Interface. Für jedes änderbare Attribute gibt es ein Eingabefeld (oder eine Combobox) in der Oberfläche, für jedes abgeleitete Attribut ein Textlabel zur Anzeige. Die Routinen, die das UI kontrollieren, arbeiten deshalb direkt mit dem zu Grunde liegenden Domain Modell. Dabei müssen lediglich einfache Typkonvertierungen durchgeführt werden, wie z. B. die Umwandlung des eingegebenen Textes in ein `Money`-Objekt.Im Fall der `HausratZusatzdeckung` unterscheidet sich das Modell jedoch von der Darstellung an der Oberfläche etwas stärker. Im User Interface werden die optionalen Deckungen immer gezeigt und können mit einer Checkbox aktiviert und deaktiviert werden (Abbildung 18).

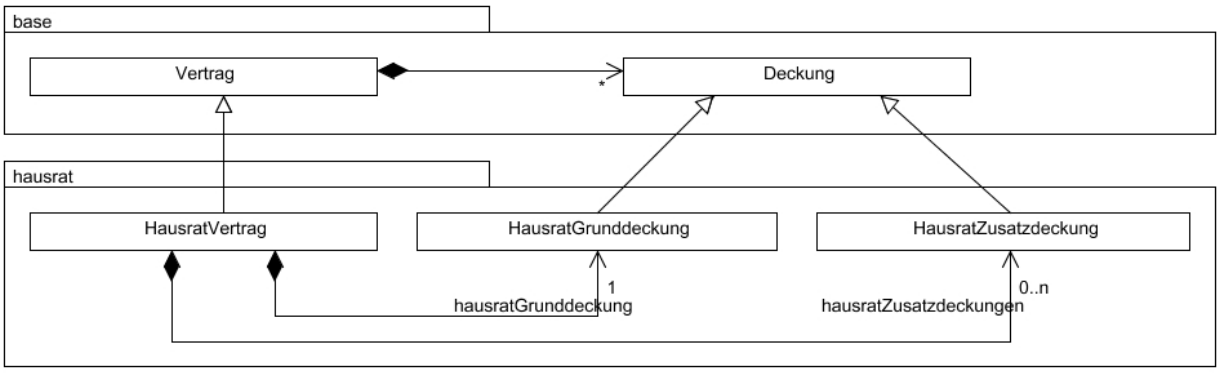

Abbildung 19: HausratZusatzdeckungen im Modell

Im Modell hingegen gibt es eine Kompositionsbeziehung zwischen *HausratVertrag* und *HausratZusatzdeckung* (Abbildung 19).

Wenn die Checkbox einer optionalen Zusatzdeckung an der Oberfläche angehakt wird, muss eine eine entsprechende *HausratZusatzdeckung* erzeugt und an den *HausratVertrag* gehängt werden. Umgekehrt wird beim Abwählen der Checkbox das zugehörige *HausratZusatzdeckung* Objekt aus der Collection der *HausratZusatzdeckungen* entfernt.

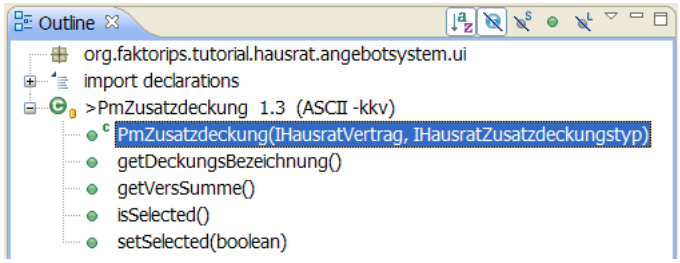

Abbildung 20: Methoden des Presentation Model für Zusatzdeckungen

Wir führen ein Presentation Model [4] ein, um das beschriebene Verhalten der Oberfläche in eine eigene Klasse zu verschieben, die Bestandteil der UI-Schicht ist. Die resultierende Klasse `PmZusatzdeckung` arbeitet mit den Modellklassen `IHausratVertrag` und `IHausratZusatzdeckungstyp`. Für die UI-Schicht implementiert das Presentation Model Methoden zum Lesen und Setzen des Zustands, in unserem Fall die Versicherungssumme und ein Flag, das angibt, ob die Zusatzdeckung ausgewählt ist (Abbildung 15). Die UI-Schicht verwendet

diese Methoden, um View und Modell zu synchronisieren (Abbildung 20).

Für jeden gemäß Produktkonfiguration möglichen `HausratZusatzdeckungstyp` wird ein `PmZusatzdeckung`-Objekt erzeugt, das die Grundlage für eine Zeile in der UI-Ansicht darstellt.

Wie die möglichen `HausratZusatzdeckungstypen` aus der Produktkonfiguration gelesen werden, wird im folgenden Abschnitt gezeigt.

Auslesen der HausratZusatzdeckungstypen

Die Implementierung der Methode `IHausratProduktGen.getHausratZusatzdeckungstypen()` liefert die für die jeweilige Produktgeneration gültigen HausratZusatzdeckungstypen.

```
/**
 * füllt die Zusatzdeckungen ins Presentation Model
 */
private void initPmZusatzdeckungen() {
    pmZusatzdeckungen = new ArrayList<PmZusatzdeckung>();

    List<IHausratZusatzdeckungstyp> hausratZusatzdeckungstypen = hausratProduktGen
            .getHausratZusatzdeckungstypen();

    for (IHausratZusatzdeckungstyp hausratZusatzdeckungstyp : hausratZusatzdeckungstypen) {
          PmZusatzdeckung pmz = new PmZusatzdeckung(hausratVertrag,
                          hausratZusatzdeckungstyp);
          //..
          pmZusatzdeckungen.add(pmz);
    }
}
```

Erzeugen von HausratZusatzdeckungen

HausratZusatzdeckungen werden auf gleiche Weise erzeugt wie *HausratGrunddeckungen* (siehe „Aufbau einer initialen Vertragsstruktur“).

```
IHausratZusatzdeckung hausratZusatzdeckung =
hausratZusatzdeckungstyp.createHausratZusatzdeckung();

hausratVertrag.addHausratZusatzdeckung(hausratZusatzdeckung);
```

Analog zu `IHausratVertrag.setHausratGrunddeckung()` hängt `IHausratVertrag.addHausratZusatzdeckung()` eine Zusatzdeckung an den Vertrag.

Die Beitragsberechnung

Sobald der Benutzer die Versicherungssumme eingibt oder ändert, berechnet das System den Versicherungsbeitrag neu und zeigt ihn an. Mit anderen Worten: es gibt keinen „Beitrag berechnen“-Button, sondern die Beitragsberechnung wird implizit ausgeführt.

In der Beispielanwendung wird ein ModifyListener installiert, der bei Änderungen in Eingabefeldern die Inhalte der UI-Controls in das Modell überträgt, die Beitragsberechnung ausruft und schließlich die berechneten Werte vom Modell wieder in das User Interface schreibt. Die Beispielanwendung unterscheidet dabei nicht, welches Eingabefeld tatsächlich geändert wurde – es werden immer alle Eingabefelder ins Modell übertragen. In einer echten Anwendung würde man die Ein- und Ausgabefelder jedoch feingranularer auslesen und schreiben.

Die Anwendung ruft die Methode `IVertrag.berechneBeitrag()` auf, welche die Versicherungsbeiträge für die einzelnen Deckungen und den Vertrag als Ganzes berechnet. Anschließend werden die berechneten Werte über die entsprechenden `get*Beitrag*()`-Methoden abgefragt und im User Interface angezeigt.

Teil 2: Dynamische Anzeige von Attributen

Motivation

Im ersten Teil dieses Tutorials haben wir gesehen, wie wir eine Anwendung implementieren, die dynamisch auf Produktänderungen (z.B. durch die Fachabteilung) reagiert und mit der wir in der Lage sind, neue Produkte zu veröffentlichen, ohne den Programmcode anpassen zu müssen. Die Flexibilität beschränkt sich jedoch auf ein definiertes Klassenmodell. Wird dieses geändert oder erweitert, zum Beispiel, wenn für eine neue Produktversion neue Tarifierungsmerkmale eingeführt werden, muss der Programmcode der Beispielanwendung angepasst werden.

Anpassungen am Angebotssystem

Um nun in unserer Anwendung auf neue Attribute im Domain-Modell reagieren zu können, werden wir sie ein wenig anpassen. Wir nutzen dazu die Möglichkeit der Runtime-API von Faktor-IPS, das Domain-Modell nach seinen Eigenschaften zu fragen, um daraus dynamisch die Darstellung zur Laufzeit zu erzeugen. Dabei halten wir uns an einen teilgenerischen Ansatz, d.h., wir stellen nur den Teil (die Klassen) des Modells dynamisch dar, der potentiell einer höherer Änderungshäufigkeit bezüglich seiner Attribute unterliegt. Damit bleiben wir an den nötigen Stellen flexibel und können an allen übrigen Stellen die Benutzerschnittstelle weiterhin optimal benutzbar gestalten.

Auslesen von Attributen der Modellklasse HausratVertrag

Die Runtime-API von Faktor-IPS bietet Methoden zur Abfrage von Modelleigenschaften zur Laufzeit (Klassen, Attribute, Beziehungen und deren Eigenschaften). Mit der Methode des `getModelType(Class<?> modelObjectClass)` können wir das RuntimeRepository nach den Meta-Informationen zu unserem Modellobjekt HausratVertrag fragen:

```
IModelType modelTypeVertrag = repository.getModelType(HausratVertrag.class);
List<IModelTypeAttribute> attributesVertrag = modelTypeVertrag.getAttributes();
for (IModelTypeAttribute modelTypeAttribute : attributesVertrag) {
       System.out.println(modelTypeAttribute.getName() + ": "
       + modelTypeAttribute.getAttributeType() + ", "
       + modelTypeAttribute.getDatatype());
}
```

Ausgabe:

```
plz: changeable, class java.lang.String
tarifzone: derived, class java.lang.String
wohnflaeche: changeable, class java.lang.Integer
vorschlagVersSumme: derived, class org.faktorips.values.Money
versSumme: changeable, class org.faktorips.values.Money
```

Wir wollen aber nicht die Eigenschaften unseres Modell-Objektes auf der Konsole ausgeben, sondern sie für die Darstellung auf der GUI unseres Angebotssystems nutzen. Dazu ändern wir die Implementierung der Methode `createUI()` in der Klasse `AngebotForm`. Wir ersetzten die Implementierung der einzelnen Aufrufe der Methoden zur Erzeugung von UI-Komponenten durch eine generische Implementierung. Diese iteriert über alle Attribute und ließt für jedes zunächst mit der Methode `Object getAttributeValue(IModelTypeAttribute, Object)` den aktuellen Wert

des Attributs aus dem darzustellenden `hausratVertrag`. Die Erzeugung der Eingabeelemente wird an die Methode `createControlForTypeAttribute(...)` der Klasse `ControlFactory` delegiert.

```
private void renderHausratVertrag(Group g1, GridData gridDataLeft,
        GridData gridDataRight, IModelType modelTypeVertrag) {
   List<IModelTypeAttribute> attributesVertrag = modelTypeVertrag.getAttributes();
   for (IModelTypeAttribute modelTypeAttribute : attributesVertrag) {
        // Label fuer Attribut erzeugen
        ControlFactory.createLabel(g1, getLabelText(modelTypeAttribute,
                    hausratVertrag.getClass()), gridDataLeft);
        // Eingabecontrol erzeugen
        Control control = ControlFactory.createControlForTypeAttribute(g1,
                    hausratVertrag, modelTypeAttribute, gridDataRight);
        // ...
   }
}
```

Die Methode `createControlForTypeAttribute(...)` sorgt dafür, dass Eingabeelemente gemäß der Metaeigenschaften des übergebenen ModelTypeAttribute erzeugt werden. Hat das Attribut die Eigenschaft CHANGABLE, wird ein entsprechendes Text-Eingabecontrol erzeugt, ist es nicht CHANGEABLE, wird stattdessen ein Label erzeugt.
EnumValues werden durch eine Combobox repräsentiert, die neben dem aktuellen Wert den möglichen Wertevorrat des EnumsTypes darstellt:

```
public static Control createControlForTypeAttribute(Composite composite,
              Object bean, IModelTypeAttribute attribute, Object layoutData) {
       AttributeType attributeType = attribute.getAttributeType();
       Object value = getAttributeValue(bean, attribute);
       String valueString = StringUtils.getString(value);

       // wenn das Attribut aenderbar ist, entsprechendes Eingabe-Control
       // rendern, sonst Label
       if (AttributeType.CHANGEABLE.equals(attributeType)) {
              // EnumValues als Combobox rendern, andere als TextBox
              if (value instanceof EnumValue) {
                     Combo enumCombo = createCombo(composite, (EnumValue) value,
                                   layoutData);
                     return enumCombo;
              } else {
                     Text textBox = createText(composite, valueString, layoutData);
                     textBox.setData(value);
                     return textBox;
              }
       } else {
              Label label = createLabel(composite, valueString, layoutData);
              return label;
       }
}
```

Neben der dynamischen Darstellung der Attribute müssen wir noch dafür sorgen, dass Modell und View synchron bleiben. Hierzu nutzen wir Eclipse Databinding, indem wir alle erzeugten UI-Elemente mit den Properties des hausratVertrag im Modell verbinden:

```
private void renderHausratVertrag(Group g1, GridData gridDataLeft,
                GridData gridDataRight, IModelType modelTypeVertrag) {
        // ...
        for (IModelTypeAttribute modelTypeAttribute : attributesVertrag) {
                // ...
                IObservableValue modelElement = BeansObservables.observeValue(
                                hausratVertrag, modelTypeAttribute.getName());
                IObservableValue uiElement = observeControl(control);
                // ...
                dataBindingCtx.bindValue(uiElement, modelElement,
                                t2MUpdateStrategy, m2TUpdateStrategy);
        }
}
```

Die Methode `observeControl(Control)` erzeugt abhängig vom Typ des übergebenen Controls ein passendes Observable:

```
private IObservableValue observeControl(Control control) {
        if (control instanceof Combo) {
                return SWTObservables.observeSingleSelectionIndex(control);
        }
        if (control instanceof Text) {
                return SWTObservables.observeText(control, SWT.Modify);
        }
        return SWTObservables.observeText(control);
}
```

Beispiel: Hinzufügen von Attribut Versicherungsort zu Hausratvertrag

Mit diesen Vorarbeiten ist unser Angebotssystem nun bereit, dynamisch auf Änderungen im Modellobjekt Hausratvertrag zu reagieren. Probieren wir es aus, indem wir das neue Attribut Versicherungsort hinzufügen. Versicherungsort soll durch einen EnumType definiert sein und je nach Auswahl des Versicherungsortes soll auf den Basisbeitrag ein Zuschlag oder Nachlass gewährt werden[6]. Um den Datentyp Versicherungsort anzulegen, erstellen wir eine Faktor-IPS Tabelle auf Basis der Tabellenstruktur EnumValue.

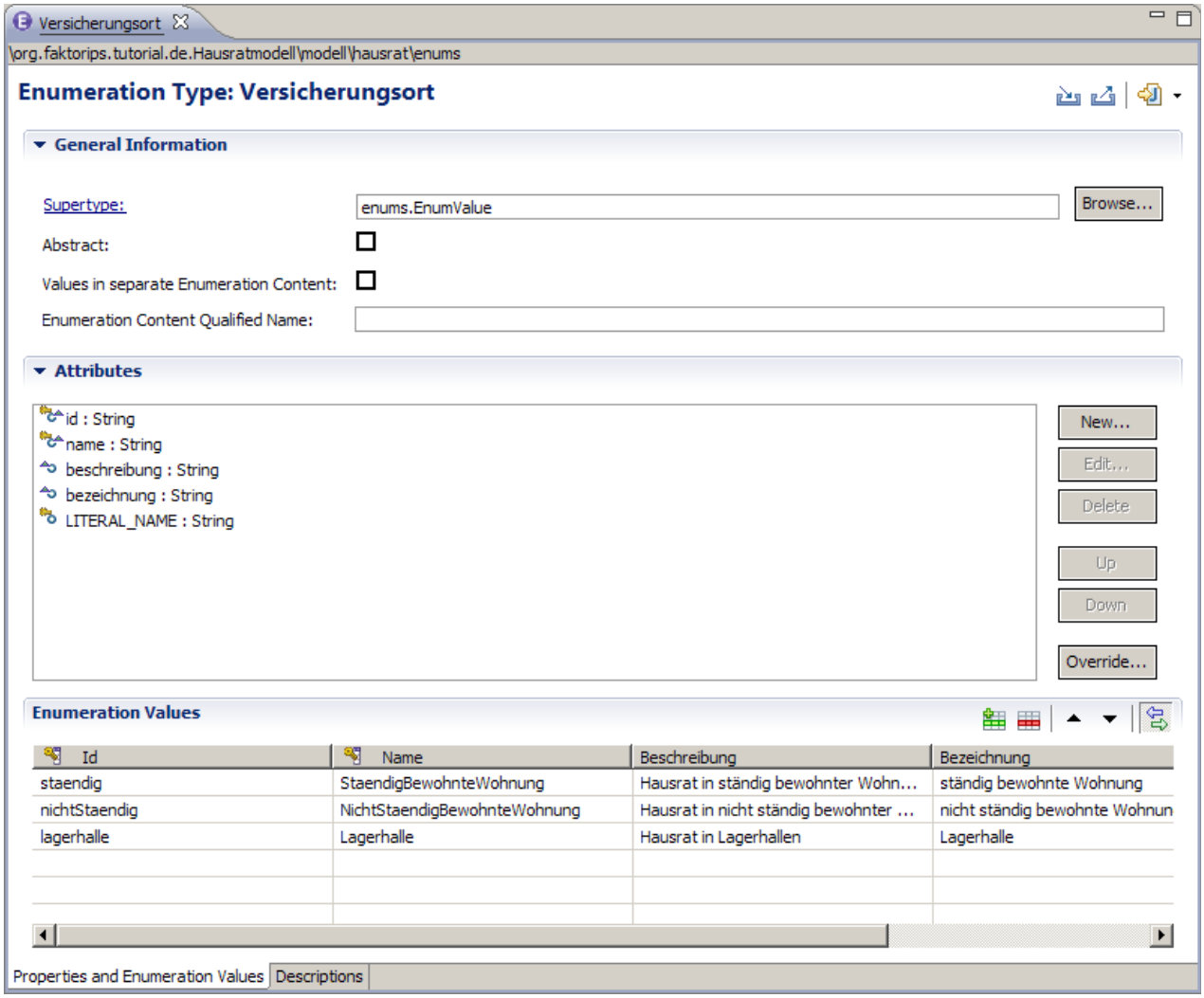

Abbildung 21: Definition des Enums Versicherungsort mit Faktor-IPS

Nun öffnen wir die Modellklasse HausratVertrag und legen das neue Attribut versicherungsort hinzu:

6 Auf die konkrete Implementierung der Beitragsberechnung unter Berücksichtigung des Versicherungsorts wollen wir hier nicht eingehen, sie ist beispielhaft implementiert und kann im Sourcecode nachvollzogen werden.

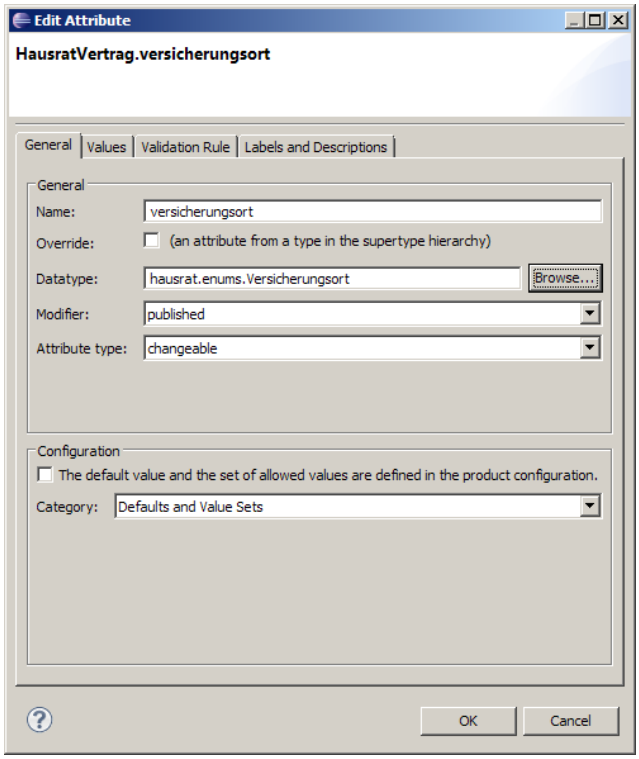

Abbildung 22: Anlegen des neuen Attributs Versicherungsort

Nach dem Neustart des Servers sehen wir, dass das neue Attribut im Bereich „Allgemeine Informationen“ auf der Oberfläche als Combobox erscheint. Die Konstanten des Enums stehen als mögliche Auswahlwerte zur Verfügung. Wenn wir den Versicherungsort ändern, sehen wir, dass der Beitrag sich ändert. Die Synchronisation von View und Modell funktioniert hier also auch.

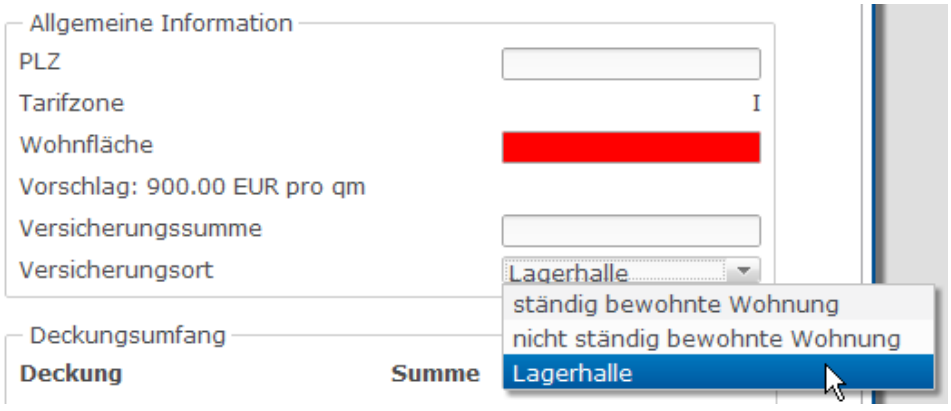

Abbildung 23: Attribut Versicherungsort als Eingabeelement

Inbetriebnahme der Beispielanwendung

Die hier vorgestellte Beispielanwendung wird auf der Faktor-IPS-Website zum Download[7] zur Verfügung gestellt. Da die Anwendung auf dem im Faktor-IPS-Tutorial erstellten Modell aufsetzt, werden auch die dort erstellten Projekte benötigt. Die fertigen Projekte können ebenfalls von der Faktor-IPS-Community-Website heruntergeladen werden[8].

Die Projekte werden als .zip-Archiv bereitgestellt und können in Eclipse über Import ► Existing Projects into Workspace importiert werden.

Im Folgenden wird beschrieben, wie die Beispielanwendung in der eigenen Eclipse-Entwicklungsumgebung und in einem alleinstehenden Webserver eingerichtet und gestartet werden kann. Es wird ein installiertes Eclipse 3.7 und Faktor-IPS 3.6 vorausgesetzt wie im Tutorial [1] beschrieben.

Installation von Eclipse RAP

Die Beispielanwendung verwendet Eclipse RAP als UI-Toolkit. Ein kurze Beschreibung, wie RAP in Eclipse eingebunden wird, befindet sich auch auf der RAP-Download-Seite [2]. Im Folgenden werden die notwendigen Installationsschritte wiedergegeben.

- In den Eclipse-Preferences (Window ► Preferences) auf der Seite Plug-in Development ► Target Platform eine neue Targetplatform mit Add... hinzufügen.
- Initialize the target definition with: Nothing wählen.
- Auf der nächsten Seite Add... Software Site wählen:

7 http://faktorzehn.org/_media/fips:tutorial-angebotsystem.zip
8 http://faktorzehn.org/_media/fips:tutorial-projekte.zip

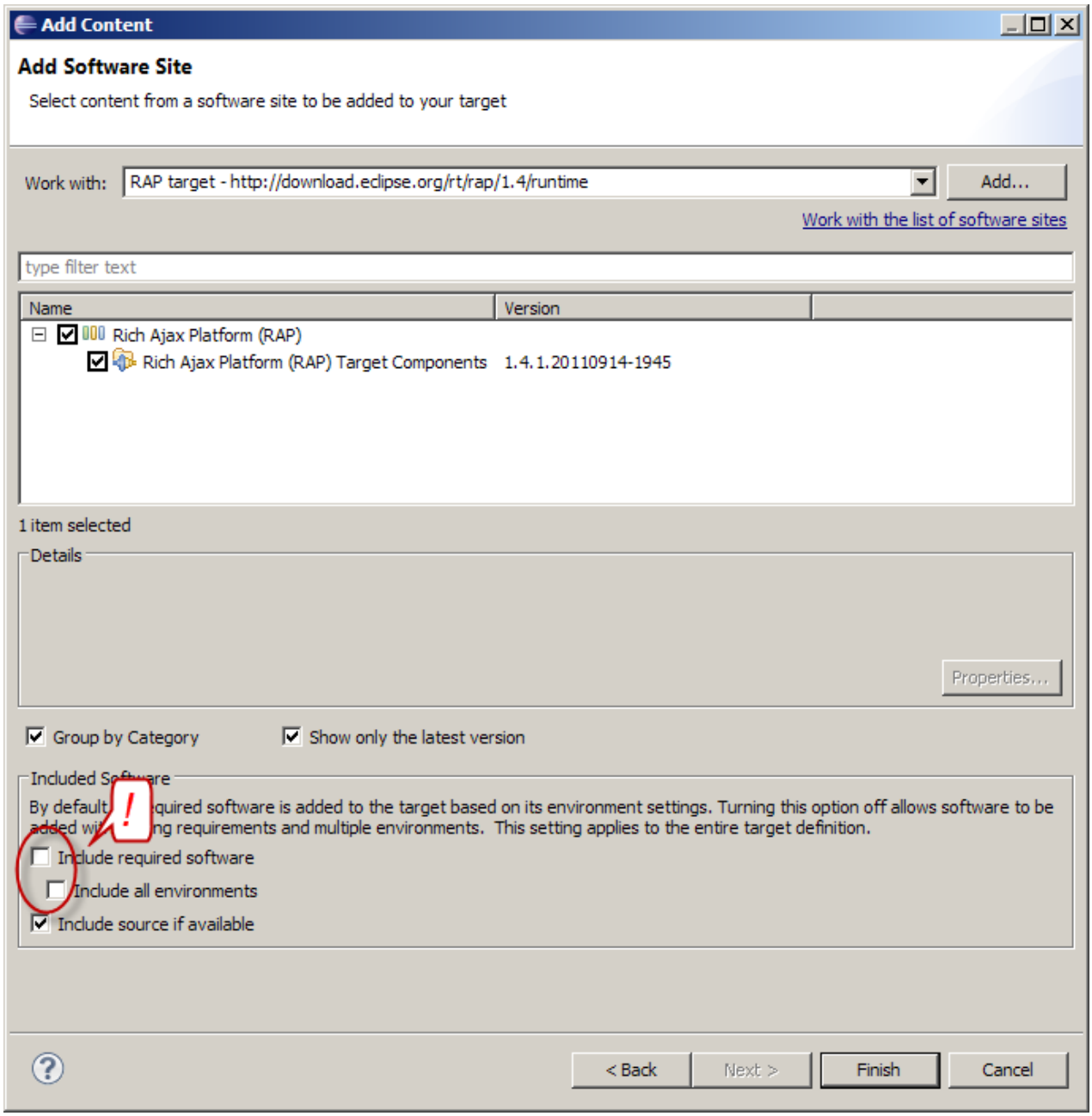

Abbildung 24: Target Platform von Software-Site installieren

- In Work with das RAP-Repository angeben: http://download.eclipse.org/rt/rap/1.4/runtime und mit Enter bestätigten.
- RAP Ajax Platform (RAP) auswählen, Include required software deaktivieren. Finish klicken. Nachdem der Download beendet ist, erneut Finish clicken.
- Nun das neu eingerichtete Target auf der Target Platform Preference Page auswählen.
- Ein Neustart von Eclipse und ein Clean/Build ist u.U. erforderlich.

Starten der Anwendung

In Eclipse unter Run ► Run Configurations... die Run Configuration Faktor-IPS Tutorial Angebotsystem auswählen und auf Run klicken (Abbildung 25).

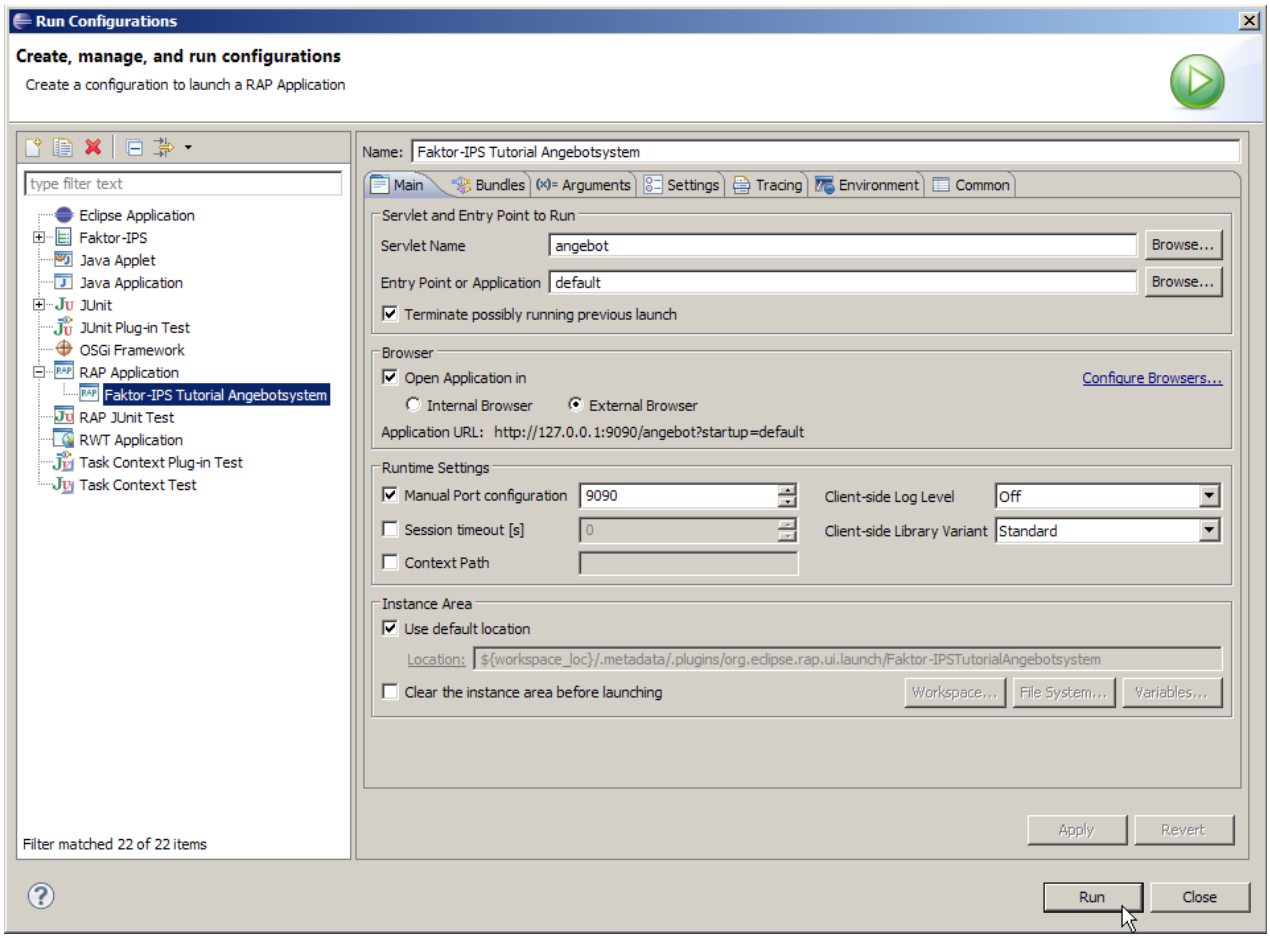

Abbildung 25: Auswahl der Launch Configuration

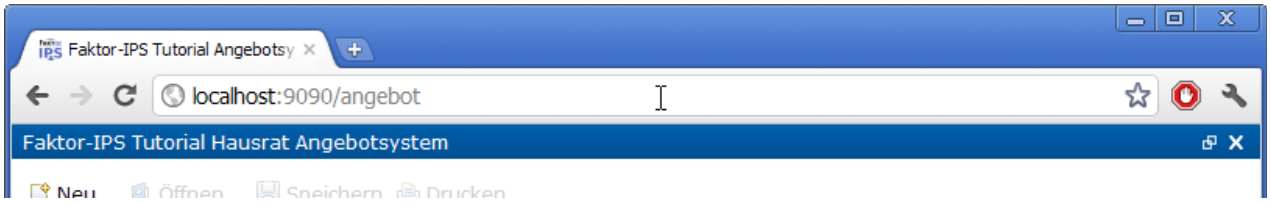

Abbildung 26: Aufruf der Anwendung im Browser

Die Anwendung läuft jetzt und kann vom Webbrowser über die URL http://localhost:9090/rap aufgerufen werden (Abbildung 26).

Inhalt

Quellen

[1] Faktor Zehn AG, Faktor-IPS Tutorial, http://www.faktorzehn.org/fips:tutorial

[2] RAP Projects – Downloads, http://www.eclipse.org/rap/downloads/

[3] Rich Ajax Platform (RAP) Project, http://www.eclipse.org/rap/

[4] Martin Fowler, Presentation Model, http://martinfowler.com/eaaDev/PresentationModel.html