Codegenerierung

Table of contents

FaktorIPS: Codegenerierung

1.

2. Rahmenbedingungen

Das Dokument beschreibt welcher Java Sourcecode aus den Vertragskomponentenklassen und den Produktkomponenten generiert wird. An den generierten Sourcecode werden dabei zwei Anforderungen gestellt:

1. Er sollte typsicher sein, damit Fehler insbesondere auch mit Sourcecode der die generierten Klassen verwendet, durch den Java Compiler gefunden werden.
2. Er sollte leicht verständlich sein. Das bedeutet vor allem, dass das konzeptionelle Modell (manifestiert in den Vertragskomponenten) sehr einfach im Code wiederzufinden ist.

3. Vertragskomponentenklassen und Produktkomponenten als Grundlagen der Generierung

Über das Eclipse-Plugin von FaktorIPS ist es möglich, ein Modell zu erstellen, das die Basis für die Generierung des Java Sourcecodes ist. Dieses Modell besteht aus zwei Arten von Elementen, den Vertragskomponentenklassen und den Produktkomponenten. Jeder Vertragskomponenteklasse kann eine oder mehrere Produktkomponenten zugeordnet werden. Umgekehrt gehört zu einer Produktkomponente maximal eine Vertragskomponentenklasse.

Grafik3

Zu jeder Vertragskomponentenklasse können Attribute, Beziehungen und Methoden definiert werden. Es gibt 3 Attributarten: Konstant, änderbar oder berechnet (constant, changeable, computed).

Wenn konstante oder änderbare Attribute einer Vertragskomponentenklasse als produktrelevant gekennzeichnet sind, dann werden sie von FaktorIPS auch als Attribute der zugeordneten Produktkomponente(n) angelegt. Dort können für diese Attribute über den Modelleditor Werte eingegeben werden (Konstanter Wert oder Default-Wert).

Wenn ein berechnetes Attribut über den Modelleditor erfasst wird, dann kann gleichzeitig eine Parameterliste für die zur Berechnung benötigte Methode spezifiziert werden. Die Formel zur Ausführung der Berechnung muss in den zugeordneten Produktkomponenten

definiert werden.

Auch zwischen Produktkomponenten können Beziehungen modelliert werden. Die möglichen Partnerproduktkomponenten für eine Beziehung sind aber durch die Beziehungen der zugehörigen Vertragskomponentenklasse eingeschränkt, wie das folgende Bild zeigt. Zu jeder Beziehung zwischen Produktkomponenten gibt es eine entsprechende Beziehung zwischen den zugeordneten Vertragskomponentenklassen.

Die Modellinformation auf Seite der Vertragskomponentenklassen beinhaltet die Information, dass zu einem Hausratvertrag je eine Leistungsvereinbarung für Wasserschaden oder Fahrraddiebstahl gehören kann. Die Beziehungen auf der Seite der Produktkomponenten zeigen, dass das Produkt HausratvertragBasis keine Leistungsvereinbarung für Fahrraddiebstahl enthält, das ist dem Produkt HausratvertragPlus vorbehalten. Eine Anwendung könnte diese Information z.B. nutzen, um gesteuert über die Produktdaten anzuzeigen, welche Leistungsvereinbarungen das gewählte Produkt bietet.

Grafik4

Zwischen Vertragskomponentenklassen können Vererbungsbeziehungen bestehen. Für Produktkomponenten gilt dies derzeit noch nicht. Bei produktrelevanten Attributen hat die Vererbung auf der Seite der Vertragskomponentenklassen aber auch Auswirkungen auf die zugeordneten Produktkomponenten, da die oben genannten Übertragungsregeln für Attribute von Vertrags- auf Produktkomponenten auch für geerbete produktrelevante Attribute einer Vertragskomponenteklasse gelten.

Grafik5

4. Generierungsregeln

4.1. Interfaces und Implementierungsklassen

Ein Beispiel

Die Generierung wird am Beispiel einer Vertragskomponentenklasse Hausratvertrag mit folgenden Eigenschaften erläutert:

- konstantes Attribut Versicherungssteuersatz, produktrelevant, published
- änderbares Attribut Zahlweise, nicht produktrelevant, published
- berechnetes Attribut Bruttoprämie, produktrelevant, published
- Methode neuberechnung(), published

Der Vertragskomponentenklasse Hausratvertrag sind zwei Produktkomponenten HausratvertragPlus und HausratvertragBasis zugeordnet.

Nur Attribute und Methoden die published sind, sollen für einen Client sichtbar sein. Diese Forderung wird durch Generierung von Interfaces umgesetzt.

Das folgende UML Diagramm zeigt die generierten Interfaces und Klassen im Überblick:

Grafik1

Vertragsseite

Für jede Vertragskomponentenklasse wird ein Interface und eine Klasse generiert (Hausratvertrag und HausratvertagImpl), die Teile von konkreten Versicherungsverträgen repräsentieren. Das Interface definiert die für Clients nutzbare Schnittstelle der Vertragskomponentenklasse (und wird deswegen auch published interface genannt). Die Klasse implementiert das Interface. Innerhalb des Modells kann auch die Klasse referenziert werden, es muss nicht notwendigerweise nur das Interface referenziert werden. Das Interface dient nur dazu, die Schnittstelle für Clients wie UI oder Batchprogramm zu definieren. Ziel des published Interfaces ist es nicht, verschiedene Implementierungen austauschen zu können. (Natürlich kann die Implementierung geändert werden, ohne dass sich das Interface ändert, aber es gibt immer genau eine Implementierung). Interface und Klasse liegen in unterschiedlichen Packages. Der Client sollte nur das Package (die Packages) mit den Interfaces referenzieren.

Produktseite

Für jede Vertragskomponentenklasse wird ein Interface und eine Implementierungsklasse generiert (HausratvertragPk und HausratvertragPkImpl).

Enthält eine Vertragskomponentenklasse ein berechnetes, produktrelevantes Attribut (=>Produktkomponenten enthalten dann also eine Formel), so ist die Klasse, die das Produktinterface implementiert, abstrakt und enthält für jedes berechnete Attribut eine abstrakte Berechnungsmethode. In diesem Fall wird für jede in FaktorIPS definierte Produktkomponente, die der Vertragskomponentenklasse zugeordnet ist, eine eigene Javaklasse generiert, die von der abstrakten Implementierungsklasse abgeleitet ist (HausratvertragBasisPk und HausratvertragPlusPk). Enthält eine Vertragskomponentenklasse kein berechnetes Attribut, so ist die Implementierungs-klasse nicht abstrakt und für die einzelnen Produktkomponenten wird keine weitere Klasse generiert.

Clients können und müssen auch direkt auf Produktkomponenten zugreifen. Interfaces von Produktkomponenten werden deshalb in dasselbe Package generiert, wie die Interfaces für die entsprechenden Vertragskomponentenklassen. Analoges gilt für die Implementierungsklassen.

Übersicht über die generierten Interfaces und Klassen

Interface für Vertragskomponentenklasse (Published Interface)
Enthält alle als published markierten Methoden sowie Zugriffsmethoden (getter/setter) für die als published markierten Attribute. Enthält Navigationsmethoden zu anderen Vertragsklasssen.
Standardname: Name der Vertragskomponentenklasse, also etwa Hausratvertrag

Implementierungklasse für Vertragskomponentenklasse (Impl-Klasse)
Implementiert das published interface. Enthält Membervariablen für die Attribute und Beziehungen. Enthält die Methodenimplementierung für die in der Vertragskomponentenklasse definierten Methoden sowie Zugriffsmethoden für die Attribute und Beziehungen.
Im Konstruktor wird eine Produktkomponente erwartet (s.u.)
Standardname: Name der Vertragsklasse + “Impl”, also etwa HausratvertragImpl

Interface für Produktkomponenten zu einer Vertragskomponentenklasse (Pk-Interface)
Enthält alle Methoden für berechnete Attribute sowie Zugriffsmethoden (getter/setter) für die Member. Enthält Navigationsmethoden zu anderen Produktkomponenten.
Standardname: Name der Vertragskomponentenklasse + “Pk”, also etwa HausratvertragPk

Implementierungklasse für Produktkomponenten zu einer Vertragskomponentenklasse (Pk-Klasse)
Enthält die Vertragskomponentenklasse mindestens ein berechnetes Attribut, ist die Klasse abstrakt.
Standardname: Name der Vertragsklasse + “PkImpl”, also etwa HausratvertragPkImpl

Abgeleitete Klassen für Produktkomponenten mit berechneten Attributen (abgeleitete Pk-Klasse)

Standardname: Name der Produktkomponente

(muss m.E. überprüft werden, sobald Vererbung zwischen Produktkomponenten eingeführt ist)

4.2. Attribute und Methoden

Die Methoden für die berechneten Attribute werden in den abgeleiteten Pk-Klassen implementiert und enthalten die in Java übersetzten Formeln.

Enthält eine Vertragskomponentenklasse ein konstantes, produktrelevantes Attribut (=>Produktkomponenten enthält das gleiche Attribut), dann wird bei der Impl- und PkImpl-Klasse eine entsprechende Getter-Methode generiert. Nur die PkImpl-Klasse hat ein entsprechendes Member, die Getter-Methode der Impl-Klasse ruft die Getter-Methode der PkImpl-Klasse über das Pk-Interface. Die PkImpl-Klasse gibt ihr Member zurück. Der Wert des Members wird über einen entsprechenden Parameter im Konstruktor der PkImpl-Klasse gesetzt.

Attributtyp	Nicht produktrelevant	produktrelevant
Konstant	Impl-Klasse:	Impl-Klasse:PkImpl-Klasse:Wirklich im Konstruktor, wird wahrscheinlich etwas unübersichtlich! Vielleicht besser über setter().
Changeable	Impl-Klasse:	Impl-Klasse:PkImpl-Klasse:wie bei konstanten Attributen
Computed	Impl-Klasse:Es muss weitere Methode geben, die Wert des Members setzt. Generator kennt diesen Zusammenhang aber nicht.	Impl-Klasse:Es muss weitere Methode geben, die Berechnungsmethode der PkImpl-Klasse aufruft. Generator kennt diesen Zusammenhang aber nicht.PkImpl-Klasse:
Derived	Später noch zu spezifizierenJeder Getter-Aufruf führt zu Neuberechnung.	

Table 1: Tabelle1

4.3. Vererbung

Es wurde bereits erwähnt, dass zwischen Vertragskomponentenklassen Vererbungsbeziehungen bestehen können. Wie das folgende Bild zeigt, finden sich diese Vererbungsbeziehungen analog in den generierten Interfaces und Pk-Interfaces, sowie in den

Impl- und PkImpl-Klassen wieder.

Member für produktrelevante Attribute (im Bild “a1”) gehören zu der PkImpl-Klasse in der gleichen Hierarchieebene wie die Impl-Klasse mit der zugehörigen Getter-Methode auch wenn der Wert für dieses Attribut in einer Produktkomponente definiert wird, die einer abgeleiteten Vertrags-komponentenklasse zugeordnet ist (siehe Kapitel “Vertragskomponentenklassen und Produktkomponenten als Grundlagen der Generierung”). Von der abgeleiteten Pk-Klasse wird der Wert über Konstruktorparameter zum Member transportiert.

Methoden für berechnete Attribute (im Bild “a2”) werden analog dazu bei dem Pk-Interface und der PkImpl-Klasse spezifiziert (abstrakt), das auf der gleichen Hierarchieebene wie die Impl-Klasse mit der rufenden Methode liegt. Kursive Namen deuten abstrakte Klassen bzw. Methoden an.

Grafik2

4.4.

4.5.

4.6.

4.7.

4.8.

4.9.

(muss m.E. überprüft werden, sobald Vererbung zwischen Produktkomponenten eingeführt ist)

4.10. Erzeugen von Vertragskomponenten

Jedes Pk-Interface und jede PkImpl-Klasse besitzt eine Factorymethode zur Erzeugung einer neuen Vertragskomponente (siehe UML-Diagramm zum Beispiel). Jede so erzeugte Instanz einer Vertragskomponentenklasse kennt die Produktkomponente, auf deren Basis sie erzeugt

wurde. (Beipiel: Jan's Hausrat-Vertrag basiert auf HausratvertragPlus).

Die neu erzeugten Vertragskomponenten besitzen zunächst keine Beziehungen zu anderen Vertragskomponenten. Zur Erzeugung einer kompletten Vertragsstruktur kann der Modellentwickler zusätzlich eine Faktoryklasse schreiben, die z. B. eine Hausratvertrag-Instanz mit den obligatorischen Vertragsteilen und dem versicherten Objekt Hausrat erzeugt. Diese Klasse wird nicht generiert.

Offener Punkt: Vertragskomponenten ohne zugeordnete Produktkomponenten

4.11. Erzeugen von Produktkomponenten

Das Erzeugen von Produktkomponenten (zur Laufzeit: Instanzen von nicht abstrakten PkImpl-Klassen oder abgeleiteten Pk-Klassen) erfolgt für alle Typen über die ProductComponentRegistry. FaktorIPS definiert ein Interface für die ProductComponentRegistry und stellt eine Default-Implementierung zur Verfügung. Von den Implementierungen des Interfaces wird folgendes Verhalten erwartet.

Die Registry weiß, wo sie die Informationen über die definierten Produktkomponenten und deren Beschreibungen findet (könnten z.B. auch in einer Datenbank abgelegt sein). Die Beschreibungen der Produktkomponenten speichert der IPS-Modelleditor ja schon persistent ab (XML). Der Generator holt zusätzlich aus den IPS-Modellklassen die Liste aller Produktkomponenten, ergänzt jedes Element um die qualifizierten Namen der Java-Klassen, die er für die jeweilige Produktkomponenten und ihre Vertragskomponentenklasse generiert, und speichert dies ergänzte Liste in einem XML-Format ab.

Jede Produktkomponente wird durch einen qualifizierten Namen eindeutig identifiziert. Über diesen Name kann von der Registry eine Produktkomponente angefordert werden: “getProductComponent(String qn)”. Die Registry erzeugt daraufhin ein neues Produktkomponentenobjekt oder gibt ein bereits vorhandenes Objekt zurück.

Zum Erzeugen neuer Objekte holt sich die Registry aus der oben genannten Liste der Produktkomponenten die Information über die benötigte Java-Klasse. Dann ruft die Registry zunächst per reflection den Konstruktor dieser Klasse auf und an der erzeugte Instanz die Methode zur Initialisierung, der als Parameter die Beschreibung der Produktkomponente mitgegeben wird.

Die Initialisierung erfolgt durch eine generierte Methode. Diese setzt die Attributwerte und die Verweise auf die referenzierten Produktkomponenten aus der übergebenen Beschreibung

(XML-Document).

Die referenzierten Produktkomponenten werden mit den qualifizierte Namen initialisiert und erst beim Zugriff mit Hilfe der Registry instanziert. Deshalb merkt sich jede Produktkomponente die im Konstruktor übergeben Registry-Referenz als Member (der Oberklasse).

Zu beachten ist, dass FaktorIPS benutzerdefinierte Datentypen unterstützt z.B. Aufzählungstypen wie “Zahlungsweise”. Damit der Generator die Konvertierung von der XML-Repräsentation der entsprechenden Werte in die Instanzen des benutzerdefinierten Datentyps erbringen kann, muss für jeden solchen Datentyp dem Generator eine Hilfsklasse zur Verfügung gestellt werden, die das Interface DataTypeHelper implementiert. Dieses Interface sieht z.B. die Methode “newInstance” vor, die ein Codefragment für diese Typkonvertierung erstellt (derzeitige Version muss noch geändert werden).

4.12. Wertebereiche

Über den IPS-Modelleditor können an Produkt-komponenten nicht nur Konstanten und Default-Werte sondern auch zulässige Wertebereiche für produktrelevante Attribute definiert werden. Die zulässigen Zahlungsweisen sind z.B. häufig vom gewählten Produkttyp abhängig. Clients benötigen die Festlegung der Wertebereiche, um Benutzereingaben zu validieren oder dem Benutzer nur die korrekten Werte zur Eingabe anzubieten z.B. über eine Select-Box mit den Bezeichnern für die zulässigen Zahlungsweisen.

Zu klären:

Welche Methoden werden generiert?

Vorschlag: je änderbarem, produktrelevantem Attribut:

- Prüfmethoden
- für Datentypen bei denen das sinnvoll ist (endlicher Wertebereich ist notwendige Bedingung) Getter für zulässiges ValueSet mit Bezeichnern zur Anzeige
- Methoden zur Abfrage von Min- und Max-Werten, falls definiert.

Benötigt Generator Unterstützung durch DataTypeHelper?

Wie sind Wertebereiche im IPS-Modell repräsentiert?

Werden die Methoden nur an den Produktkomponenten generiert oder auch an den

Vertragskomponentenklassen?

Vorschlag: an beiden, die Methoden an den Vertragskomponentenklassen rufen die Methoden an den Produktkomponenten auf, sind notwendig um bei Bedarf manuell nachbearbeitet werden zu können, um den Objektzustand zu berücksichtigen. Also z.B. nicht getErlaubteZahlungsweisen und getMoeglicheZahlungsweisen sondern

- vertragskomponente.getZahlungsweisen liefert die für dieses Objekte zulässigen Zahlungsweisen
- produktkomponente.getZahlungsweisen liefert die für die Produktkomponte zulässigen Zahlungsweisen

Bei änderbaren Attributen mit einem entsprechenden Datentyp würde an der Vertragskomponentenklasse auch nur eine Getter-Methode für den zulässigen Wertebereich generiert werden, das passt zu dem Vorschlag.

4.13. Beziehungen

Im Kapitel “Vertragskomponentenklassen und Produktkomponenten als Grundlagen der Generierung” ist ein UML-Diagramm enthalten, das den Zusammenhang von Beziehungen zwischen Vertrags- und Produktkomponenten zeigt. Die Regeln für die Generierung von entsprechenden Methoden bei den Impl-Klassen und ihren Interfaces sind in der folgenden Tabelle dargestellt. Wegen der Typsicherheit wird nicht Collection sondern Array verwendet.

	Zu-1-Beziehung	Zu-n-Beziehung
Getter	Ergebnistyp: Published Interface des Zieltyps	Ergebnistyp: Array[Published Interface des Zieltyps]
Setter	Parametertyp: Published Interface des Zieltyps	-
Add	-	Parametertyp: Published Interface des Zieltyps
Remove	-	Parametertyp: Published Interface des Zieltyps
GetAnzahl	-	Ergebnis: Anzahl der Elemente im Array

Table 1: Tabelle2

Im folgenden UML-Diagramm wird dies für das genannte Beziehungsbeispiel umgesetzt.

Grafik8

Setter, Add- und Remove-Methoden aktualisieren auch die Rückrichtung der Beziehungen.

Beziehungen zwischen Produktkomponenten sind immer Zu-1-Beziehungen. Da Produkt-komponenten-Objekte des gleichen Typs immer den gleichen Zustand haben, machen Zu-n-Beziehungen keinen Sinn. Wegen der Invarianz des Objektzustandes gibt es auch keine Setter für Beziehungen. Alle referenzierten Objekte müssen im Konstruktor übergeben werden!

4.14. Spezialisierung von Beziehungen

Betrachten wir noch einmal das obige Beziehungsbeispiel. Aufgrund des fachlichen Zusammenhangs sollte eine Leistungsvereinbarung, die mit der Methode setLvbWasserschaden oder setLvbFahrraddienstahl gesetzt wurde, in dem Array enthalten sein, das die Methode getLvbSachschaden liefert. Umgekehrt sollten die Methoden addLvbSachschaden und removeLvbSachschaden Auswirkungen auf die Ergebnisse der Methoden getLvbWasserschaden oder getLvbFahrraddienstahl haben, falls eine Leistungsvereinbarung vom Typ LvbWasserschaden bzw. LvbFahrraddienstahl hinzugefügt oder entfernt wird.

Um entsprechenden Code zu generieren, muss dieser fachliche Zusammenhang im IPS-Modell bekannt gemacht werden. Dies erfolgt dadurch, wie unten dargestellt, dass den Beziehungen zwischen den abgeleiteten Vertragskomponentenklassen die Beziehung zwischen den Super-Vertragskomponentenklassen als “Super-Relation” zugeordnet wird.

Grafik7

Für die Umsetzung dieser Information im generierten Sourcecode gibt es z.B. Die beiden folgenden Möglichkeiten.

todo: Fände, es schön, wenn wir die akutelle Hausratdemo als Grundlage aller Beispiele nehmen.

Spezialisierung von Beziehungen mit abstrakter Oberklasse

Grafik9

4.15.

4.16.

4.17.

4.18.

4.19.

4.20.

4.21.

4.22.

```
public void addSachvertrag(Sachvertrag sachvertrag) {
ArgumentCheck.notNull(sachvertrag);
if (sachvertrag instanceof HausratvertragImpl) {
setHausratvertrag((Hausratvertrag) sachvertrag);
return;
}
if (sachvertrag instanceof GlasvertragImpl) {
setGlasvertrag((Glasvertrag) sachvertrag);
return;
}
throw new IllegalArgumentException(
"Eine Hausratpolice kann keinen Vertrag der Klasse "
+ sachvertrag.getClass() + " enthalten.");
}
```

```
public void removeSachvertrag(Glasvertrag sachvertrag) {
ArgumentCheck.notNull(sachvertrag);
if (sachvertrag == hausratvertrag) {
hausratvertrag.setSachpolice(null);
hausratvertrag = null;
return;
}
if (sachvertrag == glasvertrag) {
glasvertrag.setSachpolice(null);
glasvertrag = null;
return;
}
}

public Sachvertrag[] getSachvertraege() {
ArrayList vertraege = new ArrayList();
if (glasvertrag != null) {
vertraege.add(glasvertrag);
}
if (hausratvertrag != null) {
vertraege.add(hausratvertrag);
}
return (SachvertragImpl[]) vertraege
.toArray(new SachvertragImpl[vertraege.size()]);
}
```

Spezialisierung von Beziehungen mit unveränderter Oberklasse

Bei der zweiten Alternative bleibt die Impl-Klasse der Super-Vertragskomponentenklasse unverändert. Es werden nur die Methoden addLvbSachschaden und removeLvbSachschaden in der abgeleiteten Impl-Klasse überschrieben, außerdem werden die Methoden setLvbWasserschaden und setLvbFahrraddienstahl erweitert.

```
public void addSachvertrag(Sachvertrag sachvertrag) { ... wie in erster Alternative }

public void removeSachvertrag(Glasvertrag sachvertrag) {
ArgumentCheck.notNull(sachvertrag);
if (sachvertrag == hausratvertrag) {
hausratvertrag.setSachpolice(null);
hausratvertrag = null;
super.removeSachvertrag(sachvertrag);
return;
}
if (sachvertrag == glasvertrag) {
glasvertrag.setSachpolice(null);
glasvertrag = null;
super.removeSachvertrag(sachvertrag);
return;
}
}

public void setGlasvertrag(Glasvertrag glasvertrag) {
this.glasvertrag = (GlasvertragImpl) glasvertrag;
this.glasvertrag.setSachpolice(this);
this.glasvertrag.setHausrat(hausrat);
```

super.addSachvertrag(glasvertrag);

}

Die zweite Alternative hat den Vorteil, dass durch das Einführen einer Super-Relation Klassen nicht zwingend abstrakt werden müssen.

Wie entscheiden wir Jan?

4.23. Validierung

Die Klasse PolicyComponentImpl, von der alle Impl-Klassen direkt oder indirekt abgeleitet sind, implementiert die Methoden validate, validateSelf und validateDependencies. Die Methode validate hat nur die Aufgabe validateSelf und validateDependencies aufzurufen. Die Methodenrümpfe von validateSelf und validateDependencies sind leer und müssen von den Subklassen bei Bedarf überschrieben werden. ValidateSelf soll den Zustand des Objektes selbst also die Attributwerte und die Existenz notwendiger Referenzen überprüfen, validateDependencies bei abhängigen Objekten wiederum validate aufrufen.

Einziger Parameter ist bei allen Methoden eine MessageList, in die Fehlernachrichten für die Anzeige eingefügt werden.

In der ersten Stufe wird der Generator bei den Vertragskomponentenklassen, die Methodenrümpfe für validateSelf und validateDependencies und in validateDependencies die Weiterleitung des Aufrufs an abhängige Objekte (Composition) erzeugen.

Später dann auch Aufruf der im IPS-Modelleditor spezifizierten Validation Rules. Der dazugehörige Fehlertext wird auch im Modelleditor als Kommentar eingegeben.

Generierung von Code für Überprüfung von Attributwerten und Existenz notwendiger Referenzen ist noch zu klären, manuell muss auf jeden Fall der Fehlertext ergänzt werden.

5. Der generierte Code

Nichtterminale und Ausdrücke in spitzen Klammern, Terminale fett.

InterfacePolicyCmptType(PolicyCmptType pc) ::=

<InterfaceJavadoc(pc)>

```
<InteracePackage(pc)>
<InterfaceImports(pc)>
public interface <pc.name> extends <pc.supertype> {
<InterfaceGetterAndSetterAttribute(pc.attributes[i])>*
<InterfaceGetterAndSetterRelation(pc.relations[i])>*
<InterfacePublishedMethod(pc.methods[i])>*
}

ImplPolicyCmptType(PolicyCmptType pc) ::=
<ImplJavadoc(pc)>
<ImplPackage(pc)>
<ImplImports(pc)>
public class <pc.name>Impl extends <pc.supertype>Impl implements <pc.interfacename> {
<ImplAttributeMembers(pc.attributes[i])>*
<ImplRelationMembers(pc.attributes[i])>*
<ImplGetterAndSetterAttribute(pc.attributes[i])>*
<ImplGetterAndSetterRelation(pc.relations[i])>*
<ImplMethod(pc.methods[i])>*
}

InterfaceGetterAndSetterAttribute(Attribute a) ::=
if(a.modifier == Modifier.PUBLISHED) {
<SignatureGetterAttribute(a)>;
if(a.attributeType == AttributeType.CHANGABLE) {
<SignatureSetterAttribute(a)>;
}
```

```
}

SignatureGetterAttribute(Attribute a) ::=
<JavadocGetterAttribute(a)>
public <a.datatype> get<Capitalise(a.name)> ()

SignatureSetterAttribute(Attribute a) ::=
<JavadocSetterAttribute(a)>
public void set<Capitalise(a.name)> (<a.datatype> newValue)

InterfaceGetterAndSetterRelation(Relation r) ::=
<SignatureGetterRelation(r)>;
if(r.is1ToMany()) {
<SignatureAddRelation(r)>;
<SignatureRemoveRelation(r);
} else // 1 to 1 {
<SignatureSetterRelation(r)>;
}
<SignatureNumOfRelation(r)>;

SignatureGetterRelation(Relation r) ::=
<JavadocGetterRelation(r)>
if(r.is1ToMany()) {
<r.modifier> <r.target.javatype>[] getAll<r.name>( )
} else // 1 to 1 {
<r.modifier> <r.target.javatype> get<r.name>( )
}
SignatureSetterRelation(r) ::=
```

<JavadocSetterRelation(r)>
public void set<r.name>(<r.target.javatype> refObject)

SignatureAddRelation(r) ::=
<JavadocAddRelation(r)>
public void add<r.name>(<r.target.javatype> refObject)

SignatureRemoveRelation(r) ::=
<JavadocRemoveRelation(r)>
public void remove<r.name>(<r.target.javatype> refObject)

InterfacePublishedMethod(Method m) ::=
if(m.modifier == Modifier.PUBLISHED) {
<SignatureMethod(m)>;
}

SignatureMethod(Method m) ::=
<JavadocPublishedMethod(m)>
<m.modifier> <m.datatype> <m.name> (<ParameterDecl(m.parameter[i])>*)

ParameterDecl(Parameter p) ::=
<p.datatype> <p.name>

ImplAttributeMembers ::=

ImplRelationMembers ::=

ImplGetterAndSetterAttribute ::=

ImplGetterAndSetterRelation ::=

ImplMethod ::=

6. Die Generatorklassen

Grafik6

7. Offene Punkte:

7.1. Vererbung zwischen Produktkomponenten

Idee war erst mal, dass man der Fachabteilung so etwas kompliziertes wie Vererbung nicht zumutet.

Szenario: Einer Vertragskomponentenklasse sind 20 Produktkomponenten zugeordnet (Generationen noch gar nicht berücksichtigt). Die Vertragskomponentenklasse hat eine konstantes Attribut K und ein berechnetes B. K hat für jede Produktkomponenten einen anderen Wert, für B gibt es nur zwei unterschiedliche Formeln.

Problem: Wenn Formel in jede abgeleitet Pk-Klasse kopiert wird, dann ist die Wartung der Formeln nicht gerade state of the art.

Vorschlag: Produktkomponenten können Super-Produktkomponente haben, von der sie Attributdefinitionen und Formeln erben. Überschreiben ist natürlich möglich aber nicht zwingend nötig.

Regel: Super-Produktkomponente muss der gleichen Vertragskomponentenklasse oder einer Super-Vertragskomponentenklasse zugeordnet sein.

Vorteil: Formeln und Attributdefinitionen müssen nur einmal bei der entsprechenden Supe-Produktkomponente definiert werden.

Auswirkung auf generierten Code ist zu prüfen. Bauchgefühl: Pk-Interface bleibt, Pk-Impl fliegt raus. Einsparen von Pk-Klassen ohne unterschiedliches Verhalten (berechnete Attribute) kann analog durch die Pk-Klassen für die Super-Produktkomponenten geleistet werden.

7.2. Generationen und Anpassungsstufen

Im PM gibt es Generationen und Anpassungsstufen (Achtung begriffen werden verdreht gebraucht). Warum kommen wir nur mit Generationen aus?

Lösung: neue Produktkomponenten anlegen, später über IPS Modelleditor unterstützen

7.3. Bestimmung der relevanten Generation:

Wenn ich zum Beispiel Jans Hausratvertrag erzeugt habe und den nach dem Versicherungssteuersatz frage, dann benötige ich ein Datum zu dem ich das tun will, da es in unterschiedlichen Generationen unterschiedliche Steuersätze geben könnte. Wie komme ich an dieses Datum?

7.4. Generierung von Generationen:

Die Produktkomponente gibt Zugriff auf ihre Generationen. Sie kann sowohl alle Generationen zurückliefern als auch die zu einem Stichtag gültige Generation.

Produktkomponentenklasse

Enthält eine Methode, die für ein Gültigkeitsdatum die passende Generation zurückgibt (s.u.).
Enthält eine Methode, die alle Generationen der Komponente zurückgibt.

Produktkomponentengenerationsklasse
Enthält eine Membervariable, die den Gültigkeitsbeginn enthält.
Enthält für jedes konstante Attribut eine Membervariable.
Enthält für jede Membervariable eine getter Methode.
Hat einen Konstruktor, der alle Membervariable setzt.
Enthält für jede Beziehung eine Methode, die die entsprechenden referenzierten Produktkomponenten zurückgibt. TODO: Wie kommt man an die Daten?
Enthält die Vertragskomponentenklasse mindestens ein berechnetes Attribut, ist die Klasse abstrakt und enthält für jedes berechnete Attribute eine abstrakte Berechnungsmethode.
Standardname: Name der Vertragsklasse + “PkGen”, also etwa HausratvertragPkGen

7.5. Wertebereiche

Wie kann bei produktrelevanten Attributen, der Wertebereich produktspezifisch gesetzt werden und wie wird das bei der Generierung umgesetzt? Wie erfolgt Überprüfung der Wertebereiche?

Überprüfung kann zwar generiert werden, aber der fehlertext?? vorschlag: unterstuetzen, benutzer muss noch ran.

7.6. Einbindung der generierten Klassen in eine Anwendung

Bevor es zur eigentlichen Codegenerierung geht, anbei noch einige Vorstellungen von mir zu dem Thema Modellobjekte.

Verwendung der Modellobjekte im Application Server

Modellobjekte sind reine POJOs, und sind unabhängig von einer konkreten Technologie wie z. B. EJBs. Die Modellobjekte sollten aber in einem Application Server inkl. Persistenz ablauffähig sein. Da könnte man sich einmal das Springframework ansehen, die bekommen das wohl ganz gut hin. (Möchte ein Kunde direkt z. B. Entitybeans generieren, kann das durch Anpassung des Generators erfolgen. Interessant wäre hier auch noch was in EJB 3.0 kommt, mit den EntityBeans ist man ja nicht so recht glücklick.)

Persistenz der Modellobjekte

Hier habe ich die Vorstellung, dass in den Vertragskomponentenklassen gleichzeitig notwendige Persistenzinformationen abgelegt werden. Da diese vom verwendeten Persistenzmechanismus abhängt, sollte FaktorIPS an dieser Stelle entsprechende Customizingmöglichkeiten (in Form von eigenes ExtensionPoints) bereitstellen. Dazu sollte es nach Möglichkeit eine Implementierung geben, zum Beispiel für Hibernate.

Historisierung der Modellobjekte

Noch nicht berücksichtigt.

Verwendung der Modellobjekte

Es wird davon ausgegangen, dass die Modellobjekte über LPCs aufgerufen werden, nicht über RPCs. Ist es erforderlich die Modellobjekte remote aufzurufen, so muss die Remoteschnittstelle programmiert werden, sie wird nicht generiert.

7.7. Umbenennung

Wird derzeit nicht unterstützt. Vorschlag IPS-Modelleditor sollte Umbenennung oder Änderungen des Attributtyps vorerst nicht zulassen. Muss erstmal über Löschen und Neuanlegen gemacht werden.

7.8. Customizingmöglichkeiten

Meine Vorstellung ist, dass die Art&Weise wie der Code generiert wird in einem Projekt

durch Customizing geändert werden kann. Dabei sollten einmal folgende Varianten möglich sein:

1. Die generelle Struktur der generierten Klassen bleibt unverändert, aber die Art&Weise wie z.B. Getter oder Setter Methoden generiert werden wird angepasst. Das kann man wahrscheinlich gut über eine Art Templatemechanismus abbilden. Idee wäre, den in Eclipse verwendeten Mechanismus zu benutzen.
2. Es werden andere Klassen und Interfaces auf Basis der Vertragskomponentenklassen und Produktkomponenten erzeugt. In diesem Fall wird der komplette Generator ausgetauscht (wobei durchaus Teile des existierenden wiederverwendet werden können).