



**Hochschule Ingolstadt**

**Fakultät Elektrotechnik und Informatik**

**Studiengang Allgemeine Informatik**

**Bachelorarbeit**

Thema:

Refactoring im Kontext von MDSD am Beispiel von Faktor-IPS

Vor- und Zuname:

Alexander Weickmann

ausgegeben am: 24.11.2009

abgegeben am: 15.02.2010

Erstprüfer: Prof. Dr. Hans-Michael Windisch

Zweitprüfer: Prof. Dr. Christian Facchi



# Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ingolstadt, \_\_\_\_\_

Alexander Weickmann

# Danksagung

Zunächst möchte ich mich bei Herrn Jan Ortmann für die hervorragende Unterstützung vonseiten der Faktor Zehn AG bedanken.

Ich möchte mich ebenfalls bei Prof. Dr. Hans-Michael Windisch und Prof. Dr. Christian Facchi für das Betreuen dieser Arbeit bedanken.

Außerdem möchte ich mich bei meinen Eltern Marianne und Friedrich Weickmann bedanken, die mich während meines Studiums immer unterstützt haben.

Mein ganz besonderer Dank gilt jedoch meiner Freundin Teresa, die mir während der Erstellung dieser Bachelorarbeit immer wieder neue Kraft gegeben hat.

# Abstract

Model-Driven Software Development is a discipline in modern Software Engineering that is becoming increasingly important in the industry. MDSD has not yet been combined satisfyingly with Refactoring however, a technique that has matured over many years. The main problem in this regard is to properly update the not-generated source code of the application to be developed. This thesis aims to map out an implementation concept for Refactoring in the context of MDSD as well as to develop an integration for the open source MDSD tool Faktor-IPS, which is based on the Eclipse Rich Client Platform.

The created refactoring support is based on the Eclipse LTK framework. The Java source code of the target application is updated by sequentially applying conventional refactorings to generated code elements. These conventional refactorings are provided by the Eclipse JDT plug-in. For this to work an MDSD tool needs to fulfill certain functional and infrastructural requirements that might need to be established first. The great amount of effort pays off given the enormous potentials in productivity and quality that can be uncovered by automated refactorings.

# Abstrakt

Modellgetriebene Softwareentwicklung ist eine Disziplin in modernem Software Engineering, die zunehmend an Bedeutung in der Industrie gewinnt. Das Verfahren wurde bislang noch nicht in befriedigendem Ausmaß mit der über viele Jahre gereiften Technik Refactoring kombiniert. Die Aktualisierung des nicht-generierten Quellcodes der zu entwickelnden Anwendung ist dabei das zentrale Problem. Das Ziel dieser Arbeit ist es, ein Lösungskonzept für Refactoring im Kontext von MDSD zu entwerfen. Darüber hinaus soll eine Integration für das auf der Eclipse Rich Client Plattform basierende Open Source MDSD-Werkzeug Faktor-IPS entwickelt werden.

Die entstandene Refactoring-Unterstützung setzt auf dem Eclipse LTK Framework auf. Der Java Quellcode der Zielanwendung wird durch sequentielle Ausführung mehrerer konventioneller Refactorings aktualisiert, welche vom Eclipse JDT Plug-in zur Verfügung gestellt werden. Ein MDSD-Werkzeug muss dazu über eine Reihe von funktionalen und infrastrukturellen Voraussetzungen verfügen, die gegebenenfalls erst geschaffen werden müssen. Der hohe Aufwand lohnt sich angesichts der enormen Potentiale in Produktivität und Qualität, die durch automatisierte Refactorings freigelegt werden können.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Ausgangssituation.....	1
1.2	Aufgabenstellung.....	2
1.3	Aufbau der Arbeit.....	3
<b>2</b>	<b>Grundlagen</b>	<b>4</b>
2.1	Modellgetriebene Softwareentwicklung.....	4
2.1.1	Modelle in der Softwareentwicklung.....	4
2.1.2	Der modellgetriebene Ansatz.....	5
2.1.3	Vorteile und Ziele von MDSD.....	7
2.1.4	Probleme und Risiken von MDSD.....	8
2.2	Refactoring.....	9
2.2.1	Begriffsklärung und Herkunft.....	9
2.2.2	Ein Katalog von Refactorings.....	11
2.2.3	Vorteile und Ziele von Refactoring.....	12
2.2.4	Probleme und Risiken von Refactoring.....	13
2.3	Eclipse RCP.....	14
2.3.1	Überblick.....	14
2.3.2	Extension Point Mechanismus.....	16
2.3.3	Vorteile der Eclipse RCP.....	17
2.4	Faktor-IPS.....	18
2.4.1	Überblick.....	18
2.4.2	Modellierung.....	19
2.4.3	Codegenerierung.....	21
<b>3</b>	<b>Analyse und Konzept</b>	<b>23</b>
3.1	Problemanalyse.....	23
3.1.1	Modellseitige Referenzen.....	23
3.1.2	Nicht-generierter Quellcode.....	24

<b>3.2 Voraussetzungen für eine Integration.....</b>	<b>25</b>
3.2.1 Referenzsuche und Referenzaktualisierung auf Modellebene.....	25
3.2.2 Ermittlung generierter Codeelemente.....	28
3.2.3 Referenzsuche und Referenzaktualisierung auf Codeebene.....	30
<b>3.3 Architektur.....</b>	<b>32</b>
3.3.1 Anforderungen.....	32
3.3.2 Eclipse LTK Framework.....	32
<b>4 Umsetzung und Test</b>	<b>35</b>
<b>4.1 Codegenerator: Auskunft über generierte Codeelemente.....</b>	<b>35</b>
4.1.1 Identifikation der erzeugten Codeelemente.....	35
4.1.2 Instanziierung der Java Elemente.....	36
<b>4.2 Core: Kern der Refactoring-Unterstützung.....</b>	<b>39</b>
4.2.1 Übersicht.....	40
4.2.2 Überprüfung von Vorbedingungen.....	41
4.2.3 Änderung des Modells.....	43
4.2.4 Laden von Refactoring-Teilnehmern.....	46
<b>4.3 Codegenerator: Aktualisierung von nicht-generiertem Quellcode.....</b>	<b>47</b>
4.3.1 Übersicht.....	47
4.3.2 Initialisierung.....	49
4.3.3 Überprüfung von Vorbedingungen.....	50
4.3.4 Starten der JDT Refactorings.....	51
<b>4.4 UI: Die Refactoring-Benutzeroberfläche.....</b>	<b>52</b>
4.4.1 Übersicht.....	52
4.4.2 Starten der Faktor-IPS Refactorings.....	53
4.4.3 Faktor-IPS Refactoring-Wizards.....	54
<b>4.5 Test.....</b>	<b>55</b>
<b>5 Schlussbetrachtung</b>	<b>56</b>
5.1 Fazit.....	56
5.2 Ausblick.....	57
<b>Literaturverzeichnis</b>	<b>59</b>



## Abbildungsverzeichnis

Abbildung 1: Forward Engineering.....	5
Abbildung 2: Reverse Engineering.....	5
Abbildung 3: Round-trip Engineering.....	5
Abbildung 4: Abstraktionslücke zwischen Modell und fachlichen Anforderungen.....	6
Abbildung 5: Bestandteile einer DSL im Kontext einer MDSD-Umgebung.....	7
Abbildung 6: Kostenkurve abgebildet auf den generierten Anteil des Quellcodes (grob).....	8
Abbildung 7: Bestandteile der Eclipse RCP.....	14
Abbildung 8: Erweiterung der Eclipse Workbench.....	16
Abbildung 9: Extension Point Mechanismus.....	17
Abbildung 10: Einordnung von Faktor-IPS in die Komponentenstruktur der Eclipse IDE.....	18
Abbildung 11: Ausschnitt aus dem Faktor-IPS Metamodell.....	19
Abbildung 12: Zusammenhang zwischen Modellklassen und Produktinstanzen.....	20
Abbildung 13: Faktor-IPS mit geöffnetem Model Explorer und Vertragsteilklassen-Editor.....	21
Abbildung 14: Von Faktor-IPS automatisch generierter Java Quellcode.....	22
Abbildung 15: Grundsätzlich notwendige Aktivitäten für Refactoring im Kontext von MDSD.....	23
Abbildung 16: Modellseitige Referenzen am Beispiel eines Hausratmodells.....	23
Abbildung 17: Invalide Modellreferenzen nach Umbenennen eines Modellelements.....	24
Abbildung 18: Das Interface IReference.....	26
Abbildung 19: Referenzsuche und Referenzaktualisierung auf Modellebene für Faktor-IPS.....	27
Abbildung 20: Architektur des Faktor-IPS Codegenerators.....	28
Abbildung 21: Ausschnitt der Klassenhierarchie des JDT Java Metamodells.....	29
Abbildung 22: Auskunft über generierte Java Codeelemente in Faktor-IPS.....	29
Abbildung 23: Schichtenarchitektur von Faktor-IPS.....	32
Abbildung 24: Zyklus von Abhängigkeiten zwischen Kern und Codegenerator.....	32
Abbildung 25: Die wichtigsten Klassen des Eclipse LTK Framework.....	33
Abbildung 26: Architektur der Faktor-IPS Refactoring-Integration.....	34
Abbildung 27: Grober Ablauf eines Faktor-IPS Refactorings.....	34
Abbildung 28: Aufbau des Faktor-IPS Codegenerators mit ergänzten Methoden.....	36
Abbildung 29: Aus einem Faktor-IPS Modell generierte Java Pakete und Dateien.....	36
Abbildung 30: Kern der Faktor-IPS Refactoring-Integration.....	40

---

Abbildung 31: Refactoring-Teilnehmer des Faktor-IPS Codegenerators.....	47
Abbildung 32: Extensions des Codegenerators zum Registrieren der Refactoring-Teilnehmer.....	48
Abbildung 33: Klassen der Faktor-IPS Refactoring-Benutzeroberfläche.....	52
Abbildung 34: Kontextmenü im Faktor-IPS Model Explorer.....	53
Abbildung 35: Kontextmenü im Vertragsteilklassen-Editor bzw. Produktbausteinklassen-Editor....	54
Abbildung 36: Faktor-IPS Refactoring-Wizards.....	54

## Listings

Listing 1: Code vor dem Refactoring Replace Temp With Query.....	10
Listing 2: Code nach dem Refactoring Replace Temp With Query.....	10
Listing 3: Programmcode vor Umbenennen eines Faktor-IPS Attributes.....	25
Listing 4: Programmcode nach Umbenennen eines Faktor-IPS Attributes.....	25
Listing 5: Methoden zum Abfragen und Setzen der Supertype-Eigenschaft in Faktor-IPS.....	26
Listing 6: Signatur einer Methode zur Erzeugung einer Instanz von IReference.....	26
Listing 7: Signatur einer Methode zum Abfragen von Referenzen auf andere Modellelemente.....	27
Listing 8: Signatur einer Methode zur Aktualisierung von modellseitigen Referenzen.....	27
Listing 9: Signatur einer Methode zur Ermittlung generierter Java Elemente.....	29
Listing 10: Programmcode vor Anwendung konventioneller Java Refactorings.....	31
Listing 11: Programmcode nach Anwendung konventioneller Java Refactorings.....	31
Listing 12: Instanziierung von Java Typen in der Klasse JavaSourceFileBuilder.....	37
Listing 13: Die Klasse JavaSourceFileBuilder delegiert Anfragen an Subklassen.....	37
Listing 14: Die Klasse AbstractTypeBuilder leitet Anfragen an die Klasse GenType weiter.....	38
Listing 15: Öffentliche Komponenten-Schnittstelle zur Abfrage generierter Java Elemente.....	39
Listing 16: Überprüfung von Final Conditions in IpsRefactoringProcessor.....	43
Listing 17: Methodensignatur zur Beschreibung von Änderungen.....	44
Listing 18: Faktor-IPS verwendet den Change-Mechanismus des LTK Frameworks nicht.....	45
Listing 19: Anpassung des Faktor-IPS Modells in der Klasse RenameTypeMoveTypeHelper.....	45
Listing 20: Laden von Move Participants in der Klasse IpsMoveProcessor.....	47
Listing 21: Initialisierung von Refactoring-Teilnehmern in RefactoringParticipantHelper.....	49
Listing 22: Ermittlung von Target Java Elements für ein Faktor-IPS Attribut.....	50
Listing 23: Überprüfung der Vorbedingungen aller durchzuführenden JDT Refactorings.....	50
Listing 24: Starten der JDT Refactorings.....	51

## Abkürzungsverzeichnis

Abb.	Abbildung
AG	Aktiengesellschaft
API	Application Programming Interface
bzw.	beziehungsweise
CVS	Concurrent Versions System
DSL	Domain Specific Language
EPL	Eclipse Public License
et al.	et alii (und andere, weitere)
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IPS	Insurance Product System
IT	Information Technology
JDT	Java Development Tools
Kap.	Kapitel
LTK	Language ToolKit
MDSD	Model-Driven Software Development
OSGi	Open Services Gateway Initiative
RCP	Rich Client Platform
S.	Seite
SVN	Subversion
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
usw.	und so weiter
XML	Extensible Markup Language

# 1 Einleitung

## 1.1 Ausgangssituation

*Refactoring* ist eine Technik, welche sich während des letzten Jahrzehnts als wichtiges Werkzeug für professionelle Softwareentwickler herausgestellt hat. Durch Refactoring lässt sich die interne Struktur eines Programms verändern, ohne dabei die Funktionalität zu beeinflussen. Dies kann so einfach sein wie das Umbenennen einer Variable oder so komplex wie das Zusammenführen von zwei Klassenhierarchien. Das Ziel ist eine Verbesserung der Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Quellcodes. Automatisierte Refactorings sind heute fester Bestandteil moderner *IDEs*. Sie ermöglichen effiziente und vor allem fehlerfreie Strukturänderungen am Programmcode.

Gleichzeitig gewinnt ein neues Verfahren immer mehr an Bedeutung, bei der Modelle den selben Stellenwert wie Quellcode einnehmen. Dieses Verfahren heißt *modellgetriebene Softwareentwicklung* oder auch *MDSD* (engl.: *Model-Driven Software Development*). Das Design von Software wird zwar schon lange mit Hilfe von *modellbasierten* Ansätzen beschrieben, dabei werden Modelle aber nur als Dokumentation des eigentlichen Programmcodes angesehen und separat davon gepflegt. In der Praxis führt dieser Umstand häufig zu Problemen, weil es sehr aufwändig ist die Dokumente auf aktuellem Stand zu halten. Beim MDSD-Ansatz dagegen wird ein Großteil des Quelltextes mit einem Codegenerator automatisch erzeugt. Die Modelle sind immer auf aktuellem Stand, da sie direkt in das Produkt einfließen. Das Erstellen und Bearbeiten von formalen Modellen rückt stärker in den Vordergrund, wohingegen die Arbeiten am Programmcode weniger werden.

Die Vorteile automatisierter Refactorings sollen auch beim Modellieren ausgeschöpft werden. Eine MDSD-Lösung muss Entwicklern derartige Funktionalität aber erst zur Verfügung stellen. Ein primäres Ziel von MDSD ist die Steigerung der Entwicklungseffizienz. Es widerspricht diesem Ziel, wenn Softwareentwickler bereits automatisierte Arbeitsschritte im Rahmen von MDSD manuell durchführen müssen.

Bei der Entwicklung einer Refactoring-Unterstützung im Kontext von MDSD ist das zentrale Problem, dass in der Regel nicht 100% des Quellcodes automatisch erzeugt werden. Somit ist es nicht ausreichend, wenn durch das MDSD-Werkzeug nur Änderungen am Modell vorgenommen werden. Beim erneuten Generieren des Programmcodes würden mit großer Wahrscheinlichkeit nicht-generierte Programmstücke fehlerhaft sein, da sie Code referenzieren, der nicht länger in

dieser Form existiert. Die MDSD-Lösung muss auch Modifikationen am Quellcode vornehmen. Das kann unter Umständen einen erheblichen Entwicklungsaufwand bedeuten.

Eine derartige Refactoring-Unterstützung wird von den aktuell verfügbaren MDSD-Werkzeugen kaum oder gar nicht angeboten. Diese Arbeit beschäftigt sich mit dem Thema, da durch automatisierte Refactorings große Potentiale bezüglich Produktivität und Qualität freigelegt werden können. Neben einer Erläuterung der grundlegenden Technologien und Techniken sowie der Vorstellung eines Lösungskonzepts, sollen für ein Open Source MDSD-Werkzeug einige automatisierte Refactorings entwickelt werden.

## 1.2 Aufgabenstellung

Die Motivation der Arbeit begründet sich aus der Anforderung heraus, das Open Source Produkt *Faktor-IPS* [FIPS] um eine Refactoring-Unterstützung zu erweitern. Dieses Programm ermöglicht die modellgetriebene Entwicklung von Versicherungssoftware im Sinne von MDSD. Die *Faktor Zehn AG* entwickelt das Werkzeug seit dem Jahr 2006 und modernisiert damit die IT namhafter Versicherungen in Deutschland und Österreich.

Faktor-IPS wird auf Basis der *Eclipse RCP* [Eclipse] entwickelt, einer in der Programmiersprache *Java* geschriebenen, quelloffenen und erweiterbaren Plattform für Software jeglicher Art.

Das Ziel dieser Arbeit ist es, eine Refactoring-Unterstützung in Faktor-IPS zu integrieren. Dazu soll zunächst eine Problemanalyse und eine Untersuchung der Voraussetzungen durchgeführt werden. Dabei erfolgt die Entwicklung eines Lösungskonzepts. Im Konzept wird beschrieben, wie die vorausgesetzten Teilfunktionen umgesetzt werden können und wie diese im Rahmen einer Refactoring-Integration kombiniert werden müssen. Anschließend sollen einige automatisierte Refactorings in Faktor-IPS implementiert werden. Dabei soll auch darauf eingegangen werden, wie die entwickelte Funktionalität getestet wurde.

Es gilt zu berücksichtigen, dass innerhalb einer (sinnvoll angelegten) MDSD-Umgebung nur der Codegenerator darüber Auskunft geben kann, welche Codeelemente für ein gegebenes Modellelement erzeugt werden. Daher muss die während eines Refactorings notwendige Anpassung des Quellcodes im Verantwortungsbereich des Generators liegen. Dieser stellt eine gekapselte Komponente dar und unterstützt möglicherweise mehrere Ziel-Programmiersprachen. Die im Rahmen dieser Arbeit entworfene Architektur muss diesen Gegebenheiten gerecht werden und darf insbesondere keine Abhängigkeiten zum Codegenerator erzeugen, um dessen Austauschbarkeit zu gewährleisten.

### 1.3 Aufbau der Arbeit

In Kapitel 2 werden die grundlegenden Technologien und Techniken vorgestellt, auf denen die im Rahmen dieser Arbeit entwickelte Refactoring-Integration aufbaut: modellgetriebene Softwareentwicklung und Refactoring. Außerdem enthält das Kapitel eine Einführung in die Eclipse Rich Client Platform und das MDSD-Werkzeug Faktor-IPS. Dies soll dem besseren Verständnis der folgenden Kapitel dienen.

Im dritten Kapitel wird erklärt, welche Probleme sich bei Refactoring im Kontext von MDSD ergeben. Aus diesen Problemen werden die funktionalen und infrastrukturellen Voraussetzungen für eine Integration abgeleitet, über die ein MDSD-Werkzeug vorab verfügen muss. Dabei werden auch Lösungsansätze zur Realisierung dieser Funktionen in Faktor-IPS vorgestellt. Das Kapitel wird mit dem Entwurf einer geeigneten Architektur abgeschlossen. Besonderes Augenmerk liegt dabei auf dem Aspekt der losen Kopplung von beteiligten Modulen.

Das vierte Kapitel beschäftigt sich mit der konkreten Umsetzung und dem Test der Faktor-IPS Refactoring-Integration. Im ersten Unterkapitel wird der Faktor-IPS Codegenerator um die Auskunft über generierte Codeelemente erweitert. Anschließend wird erklärt, wie der Kern der Refactoring-Unterstützung aufgebaut ist und wie das Ändern des Modells vonstatten geht. Die Aktualisierung von nicht-generiertem Quellcode ist Thema des dritten Unterkapitels. Im vierten Abschnitt wird die Faktor-IPS Refactoring-Benutzeroberfläche vorgestellt. Einige Worte zum Test der entwickelten Funktionalität runden das Kapitel ab.

Im Rahmen einer Schlussbetrachtung werden die Ergebnisse der Arbeit und alle erworbenen Kenntnisse in einem kurzen Fazit zusammengefasst und bewertet. Zum Abschluss wird ein Ausblick auf Themen gegeben, die auf dieser Arbeit aufbauen können. Dabei wird insbesondere aufgezeigt, wie sich die Faktor-IPS Refactoring Integration in Zukunft weiterentwickeln sollte.

## 2 Grundlagen

### 2.1 Modellgetriebene Softwareentwicklung

#### 2.1.1 Modelle in der Softwareentwicklung

Die Struktur einer komplexen Software kann nicht überblickt werden, wenn lediglich Programmcode betrachtet wird. Zu viele Details verschleiern das zugrunde liegende Design. Mit Hilfe von *Modellen* können Systeme auf ein höheres Abstraktionsniveau abgebildet werden. So lassen sich Beziehungen zwischen Programmmodulen übersichtlich darstellen und die für den Gesamtkontext unbedeutenden Details ausblenden. Softwaremodelle ermöglichen somit den Blick auf Konstruktionskonzepte, auf deren Grundlage Systeme entwickelt werden.

*„Als **Modellierung** wird in der Informatik im Allgemeinen eine abstrakte Abbildung eines – zu entwickelnden – Softwaresystems verstanden.“ [Pi07 S.45]*

Menschen können visuelle Darstellungen schneller erfassen als Text. Modelle werden daher in der Regel grafisch in Form von Diagrammen dargestellt. Der Einsatz von Diagrammen ermöglicht eine verbesserte Kommunikation unter Entwicklern. Verschiedene Diagramme können unterschiedliche Sichten auf das selbe Modell bieten. Je nach Anforderung der Situation können Elemente ausgeblendet oder besonders detailliert dargestellt werden. So ist es auch möglich, mit Personen, die nicht in den Programmcode eingearbeitet sind, über ein System zu kommunizieren.

Zur Anfertigung solcher Modelle und Diagramme existieren verschiedene *Modellierungssprachen*. Die heute bekannteste und am weitesten verbreitete Modellierungssprache ist die *UML* [UML].

Modelle werden genutzt, um zu Beginn eines Projekts die grundlegende Softwarearchitektur festzulegen. Während der Entwicklung wird mit Hilfe von Modellen das Softwaredesign dokumentiert und kommuniziert. Diese Vorgehensweise nennt man *modellbasiert*. Es gibt dabei verschiedene Probleme:

- 1) Es ist aufwändig die Modelle auf aktuellem Stand zu halten.
- 2) Die Modelle werden als Dokumentation betrachtet und fallen schnell Termindruck zum Opfer.
- 3) Viele Entwickler empfinden es als anstrengend die Modelle anzufertigen, da sie nicht zu unmittelbarem Entwicklungsfortschritt führen.



Strukturelle Änderungen müssen zweifach vorgenommen werden, nämlich im Quelltext und im Modell. Man versucht daher, aus den im Modell enthaltenen Informationen die Struktur des Quellcodes automatisiert zu generieren. Diese Technik nennt man *Forward Engineering*. Bei einer Neugenerierung werden im Quellcode vorgenommene Strukturänderungen gelöscht. Anderenfalls wären Modell und Code nicht mehr konsistent.



Abbildung 1: Forward Engineering

*Reverse Engineering* löst dieses Problem, indem das Modell aus dem Quellcode abgeleitet wird. Die Entwickler modifizieren bei dieser Technik nur den Programmcode. Strukturänderungen können nur im Quellcode vorgenommen werden.



Abbildung 2: Reverse Engineering

*Round-trip Engineering* kombiniert *Forward Engineering* und *Reverse Engineering* miteinander.

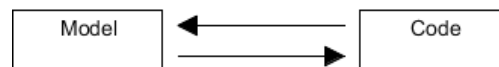


Abbildung 3: Round-trip Engineering

### 2.1.2 Der modellgetriebene Ansatz

*Modellgetriebene Softwareentwicklung* ist eine Weiterentwicklung von Forward Engineering. Wie beim Forward Engineering wird aus Modellen Quellcode generiert.

*„Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ [St07 S.11]*

Die in Kapitel 2.1.1 vorgestellten Techniken betrachten Modelle stets als Dokumentation von Quellcode und motivieren sich aus der Notwendigkeit heraus, Code und Modell synchron zu halten. Modelliert wird dabei die exakte Struktur des Programmcodes. Es existiert immer noch eine relativ große Abstraktionslücke zwischen dem Modell und den fachlichen Anforderungen einer Software.

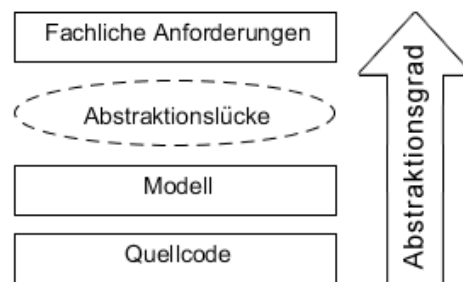


Abbildung 4: Abstraktionslücke zwischen Modell und fachlichen Anforderungen  
(angelehnt an [Bü07 S.21])

MDSD überwindet die Abstraktionslücke, indem fachspezifische Konstrukte als Modellelemente eingesetzt werden. Diese Konstrukte werden durch die *Domäne* einer Anwendung vorgegeben. Der Begriff Domäne wird von Stahl et al. [St07 S.28] als begrenztes Interessen- bzw. Wissensgebiet definiert.

Die Menge der fachlichen Konstrukte mit den dazugehörigen Regeln bezüglich Syntax und Semantik wird als *Domain Specific Language (DSL)* bezeichnet [vgl. St07 S.30].

Zentraler Bestandteil jeder DSL ist das sogenannte *Metamodell*. Es umfasst die *abstrakte Syntax* und die *statische Semantik* der Sprache. Die Begriffe werden von Stahl et al. [St07 S.29f] wie folgt definiert:

*„Im Kontext von MDSD geht es darum, Inhalte einer Domäne formal zu beschreiben [...] Diese formalisierte Beschreibung [...] nennt man **Metamodell**.“*

*„Die Metamodellelemente und ihre Beziehungen untereinander bezeichnet man auch als **abstrakte Syntax**.“*

*„Die **statische Semantik** einer Sprache legt Bedingungen fest, die ein Modell erfüllen muss, damit es »wohlgeformt« ist.“*

Wenn im Verlauf dieser Arbeit der Begriff Metamodell eingesetzt wird, so bezieht sich dies insbesondere auf die abstrakte Syntax.

Weitere Bestandteile einer DSL sind zudem eine *konkrete Syntax* und eine *dynamische Semantik*. Die konkrete Syntax beschreibt, wie die auf Grundlage der DSL erstellten Modelle aussehen. Die Bedeutung jedes Modellelements wird durch die dynamische Semantik definiert [vgl. St07 S.29f].

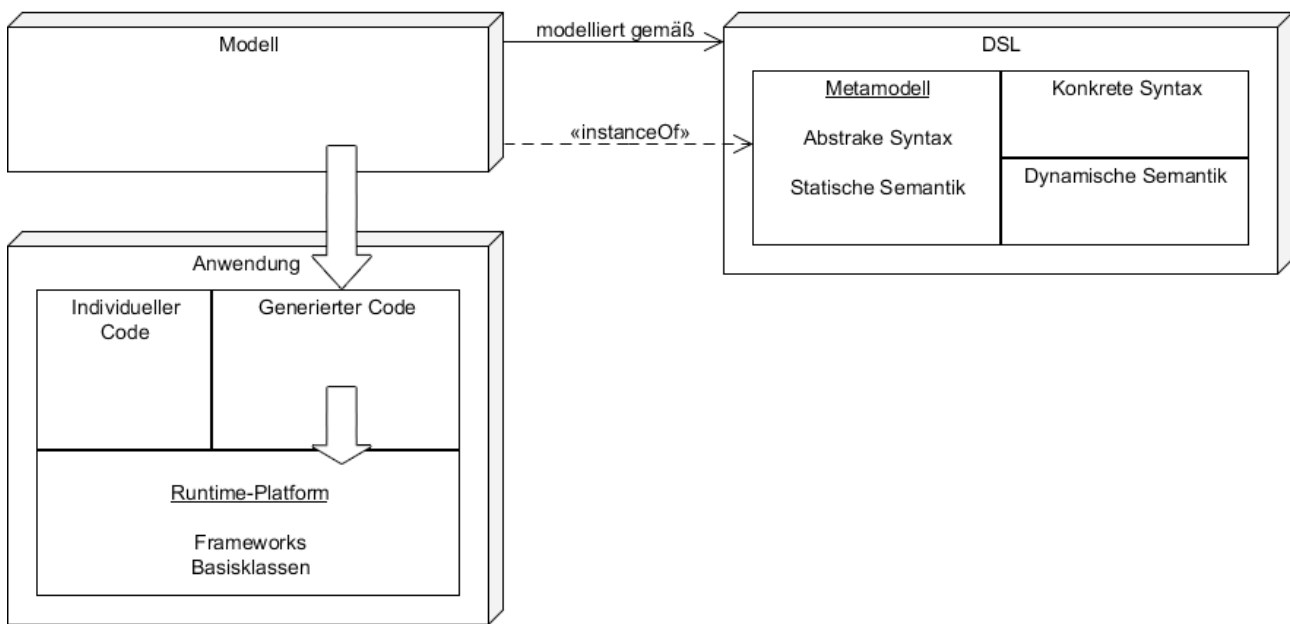


Abbildung 5: Bestandteile einer DSL im Kontext einer MDSD-Umgebung  
(angelehnt an [FZ08 S.13])

### 2.1.3 Vorteile und Ziele von MDSD

*„Es geht bei MDSD zunächst einmal um die Verbesserung von Softwarequalität und Wiederverwendbarkeit sowie die Steigerung der Entwicklungseffizienz.“ [St07 S.13]*

Durch die höhere Abstraktion in den Modellen kann wesentlich mehr Quellcode generiert werden als beim Forward Engineering. Die Modelle entsprechen keiner 1-zu-1 Abbildung der Codestruktur mehr. Insbesondere wird von Konstruktionsparadigmen der Zielprogrammiersprache abstrahiert. Diese können vom Codegenerator auf Grundlage der dynamischen Semantik aus den Modellen abgeleitet werden. Auch das Generieren von Teilen der Programmlogik oder anderen Artefakten wie zum Beispiel Konfigurationsdateien ist möglich. Der Anteil an Routinearbeit wird für Entwickler erheblich reduziert. Bei modellgetriebener Softwareentwicklung nehmen deswegen Modelle den selben Stellenwert wie Quellcode ein.

Der Programmcode wird stets in einer einheitlichen Formatierung und anhand vorgegebener Konstruktionsprinzipien erzeugt. Dadurch ist sichergestellt, dass Architekturentscheidungen im Code umgesetzt sind. Die Architektur wird an einem zentralen Ort – dem Codegenerator – festgelegt und „verwässert“ nicht mit der Zeit. Änderungen an dieser Stelle erreichen den gesamten generierten Anteil des Quellcodes. Eine Veränderung der Softwarearchitektur kann bei MDSD somit wesentlich einfacher als bei konventioneller Entwicklung vorgenommen werden.

Eine Steigerung der Entwicklungsgeschwindigkeit ergibt sich vor allem, wenn die zu produzierende Software eine gewisse Größe erreicht oder eine lange Lebensdauer hat. Auf Grundlage der entwickelten Werkzeuge und DSL können mehrere Softwareprojekte entwickelt werden, da diese Bestandteile einer MDSD-Lösung wiederverwendbar sind.

#### 2.1.4 Probleme und Risiken von MDSD

Bei modellgetriebener Softwareentwicklung wird eine Fülle an Werkzeugen benötigt, um DSL und Modelle erstellen und bearbeiten zu können. Ein Codegenerator ist erforderlich, um aus Modellen automatisiert Quellcode zu generieren. Zum Bearbeiten und Ergänzen des individuellen Codes ist eine traditionelle IDE notwendig. Das größte Risiko bei MDSD ist, dass diese Werkzeugkette den qualitativen Ansprüchen nicht genügt oder nicht wie erwartet funktioniert. Die Effizienz der Entwickler hängt zudem stark von der Qualität der verwendeten Werkzeuge ab.

Es ist zu berücksichtigen, dass Modelle von mehreren Entwicklern zur gleichen Zeit bearbeitet werden können. Modellartefakte müssen außerdem mit Versionsverwaltungssystemen verträglich sein. Es wäre in diesem Zusammenhang sehr unvorteilhaft, wenn das komplette Modell einer Anwendung in einer einzelnen Datei gespeichert wird.

Eine weitere Gefahr ist, dass Sonderfälle in der DSL erfasst werden, welche sich nur äußerst geringfügig auf den generierten Code auswirken. Änderungen an der DSL sind relativ teuer und müssen einen Mehrwert erzielen. Aus wirtschaftlicher Sicht kann nicht 100% der Anwendung automatisch aus formalen Modellen generiert werden.

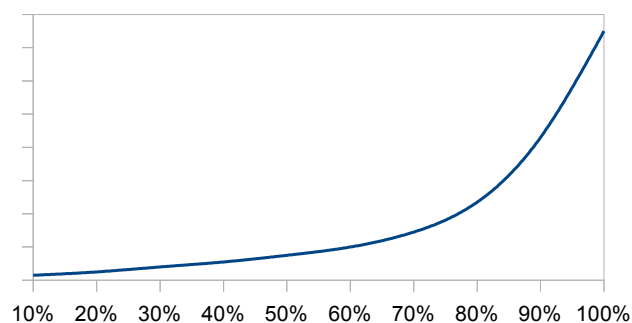


Abbildung 6: Kostenkurve abgebildet auf den generierten Anteil des Quellcodes (grob)

Nicht-generierte Codefragmente müssen mit generiertem Quellcode verbunden werden. Wenn es die Architektur erlaubt, kann das Problem dadurch gelöst werden, dass nicht-generierte Codeteile in eigene Artefakte ausgelagert werden. Ist das nicht möglich, so müssen bei jeder Neugenerierung die nicht-generierten Programmteile mit dem generierten Anteil fusioniert werden. Diese Technik wird als *Merging* (vom engl.: verschmelzen) bezeichnet.

## 2.2 Refactoring

### 2.2.1 Begriffsklärung und Herkunft

Es ist selten, dass die Struktur von einmal geschriebenem Programmcode nie wieder verändert werden muss. Oft stellt sich Monate oder Jahre nach dem Erstverfassen heraus, dass das Quellcodegerüst eine nun benötigte Erweiterung nicht tragen kann, weil dies nicht vorhergesehen und der Code nicht darauf ausgelegt wurde. Es könnte sich aber auch herausstellen, dass Teile des vorhandenen Quellcodes komplexer sind, als sie sein müssten. Das passiert wenn versucht wird, möglichst viele Eventualitäten abzudecken, welche in der Praxis nur mit geringer Wahrscheinlichkeit eintreten werden. Durch *Refactoring* kann Programmcode gezielt und schrittweise so modifiziert werden, dass er neuen Gegebenheiten besser entspricht, ohne dabei existierende Funktionalität zu zerstören.

Die Pioniere der Technik sind *Ward Cunningham* und *Kent Beck*. Sie verwendeten Refactoring bereits in den 80er Jahren bei der Entwicklung von Software mit *Smalltalk*, einer damals populären, objektorientierten Programmiersprache. Das Thema wurde laut Fowler [Fo99 S.71] im Jahr 1992 zum ersten Mal in der Doktorarbeit „*Refactoring Object-Oriented Frameworks*“, [Op92] von *William F. Opdyke* wissenschaftlich behandelt. In den folgenden Jahren wurde die Technik ausschließlich von wenigen Experten angewandt, da es keine Einführung zu Refactoring in einer für alle Programmierer verständlichen Form gab. Dies änderte sich im Jahr 1999, als das von *Martin Fowler* verfasste Buch „*Refactoring – Improving the Design of Existing Code*“ [Fo99] erschien. Heute ist Refactoring ein wichtiger Bestandteil von *Extreme Programming*, einem beliebten Vorgehensmodell in der Softwaretechnik.

Es ist unklar, woher der Begriff Refactoring ursprünglich stammt. Internetquellen wie Wikipedia [Wi10] behaupten, dass ursprünglich ein Werkzeug entwickelt werden sollte, welches Quelltext automatisch verbessert. Dieses Werkzeug wurde als *Re-Factory* bezeichnet und der Prozess der Verbesserung demnach Refactoring. Im Rahmen dieser Arbeit konnte aber kein Indiz dafür gefunden werden, dass diese Behauptung wahr ist.

Zum gegenwärtigen Zeitpunkt ist die Auffassung weit verbreitet, dass der Begriff an die Mathematik angelehnt ist. In der Mathematik können Zahlen *faktoriert* werden:

$$8 * 2 = 16 \quad \Rightarrow \quad 2 * 2 * 2 * 2 = 16$$

Beide Ausdrücke liefern das selbe Ergebnis, sind jedoch von der Struktur her unterschiedlich. Der linke Term kommt mit weniger Zahlen und Operanden aus, wohingegen der rechte Term kein Wissen über die Zahl 8 erfordert. Betrachtet man beispielsweise das Refactoring *Replace Temp with Query* [Fo99 S.120], bei dem eine temporäre Variable durch eine Abfrage ersetzt wird, so kann man eine Ähnlichkeit zum Faktorisieren in der Mathematik erkennen:

**Listing 1:** Code vor dem Refactoring Replace Temp With Query

```
1    public double computePrice(int quantity, double itemPrice) {
2        double basePrice = quantity * itemPrice;
3        double discountFactor = (basePrice > 1000) ? 0.95 : 0.98;
4        return basePrice * discountFactor;
5    }
```



**Listing 2:** Code nach dem Refactoring Replace Temp With Query

```
1    public double getPrice(int quantity, double itemPrice) {
2        double discountFactor =
3            (getBasePrice(quantity, itemPrice) > 1000) ? 0.95 : 0.98;
4        return getBasePrice(quantity, itemPrice) * discountFactor;
5    }
6
7    private double getBasePrice(int quantity, double itemPrice) {
8        return quantity * itemPrice;
9    }
```

Auch hier liefern beide Varianten das selbe Ergebnis, obwohl die interne Struktur vollkommen unterschiedlich ist.

Fowler definiert den Begriff Refactoring folgendermaßen:

*„Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.“*

*[Fo99 S.xvi]*

Diese Definition gilt heute als allgemein anerkannt.

### 2.2.2 Ein Katalog von Refactorings

Martin Fowler beschreibt in seinem Buch einen „*Katalog von Refactorings*“ für objektorientierte Programmiersprachen. Ein *Refactoring* ist dabei ein Muster, welches sich schrittweise auf Quellcode anwenden lässt und hat zum Ziel, einen bestimmten Aspekt des Quellcodes umzugestalten.

Das Substantiv Refactoring ist nicht zu verwechseln mit dem Verb, welches der Technik ihren Namen gibt und in Kapitel 2.2.1 definiert wurde. Daher werden in [Fo99 S.53f] zwei weitere Definitionen eingeführt:

*“**Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.”*

*“**Refactor** (verb): to restructure software by applying a series of refactorings without changing its observable behaviour.”*

Ein einzelnes Refactoring wird folgendermaßen beschrieben [vgl. Fo99 S.103]:

- **Name**

Der Name ist wichtig, um mit anderen Entwicklern über das Refactoring kommunizieren zu können. Er lässt bereits erkennen, um was es bei dem Refactoring geht. Namen für Refactorings setzen sich immer aus einer Tätigkeit und einem Programmelement zusammen.

- **Zusammenfassung**

In einer Zusammenfassung wird beschrieben, in welcher Situation das Refactoring gebraucht wird und was die grundlegenden Aufgaben des Refactorings sind. Ein Minimalbeispiel kommuniziert die Absicht noch deutlicher.

- **Motivation**

Dieser Abschnitt erklärt, warum das Refactoring durchgeführt werden sollte. Außerdem wird darauf eingegangen, unter welchen Voraussetzungen das Refactoring nicht durchgeführt werden sollte.

- **Mechanik**

Im Abschnitt Mechanik wird mit Hilfe einer präzisen Schritt-für-Schritt Anleitung erklärt, wie das Refactoring durchzuführen ist.

Die im Katalog definierten Refactorings können auf den eigenen Programmcode angewandt werden, indem man den im Abschnitt Mechanik definierten Schritten des jeweiligen Refactorings

Folge leistet. Moderne IDEs wie Eclipse sind in der Lage, diese Arbeitsanweisungen automatisiert abuarbeiten. Das beschleunigt den Entwicklungsprozess und vermeidet Fehler bei der Ausführung.

### 2.2.3 Vorteile und Ziele von Refactoring

Das Design einer Software verliert an Integrität, wenn Programmcode modifiziert wird, um kurzfristige Ziele zu erreichen. Die Ursache hierfür ist, dass bei derartigen Änderungen häufig nicht die Struktur des Gesamtsystems berücksichtigt wird. Durch regelmäßiges Refactoring kann verhindert werden, dass das Design einer Software mit der Zeit zerfällt.

Refactoring bewirkt, dass Quellcode lesbarer und einfacher zu verstehen wird. Durch eine bessere Aufteilung von komplexen Methoden oder Klassen in beherrschbare Teilstücke kann Intention und Implementierung getrennt voneinander kommuniziert werden. Das Aufteilen einer komplexen Methode in mehrere Subroutinen impliziert, dass jede Subroutine über einen eigenen Namen und eine eigene Dokumentation verfügt. Durch die Verwendung von sinnvollen Bezeichnern erklärt sich ein Programmstück oft von selbst.

Ein wichtiges Ziel von Refactoring ist es, duplizierten Code zu eliminieren. So wird sichergestellt, dass im Quelltext jedes Problem nur genau einmal gelöst wird. Das ist die Essenz von gutem Design und ermöglicht die gemeinsame Nutzung von Programmlogik [vgl. Fo99 S.56].

*„Bei der Refaktorisierung können wir unser gesamtes Wissen über gutes Software-Design einsetzen. Wir können die Kohäsion verstärken, die Kopplung verringern, Zuständigkeiten trennen, Systemaufgaben in Module auslagern, unsere Funktionen und Klassen verkleinern, bessere Namen verwenden usw. An dieser Stelle wenden wir auch die anderen drei Regeln des einfachen Designs an: Duplizierten Code eliminieren, die Ausdruckskraft des Codes verbessern und die Anzahl der Klassen und Methoden minimieren.“ [Ma09 S.211]*

Regelmäßiges Refaktorisieren erhöht die Geschwindigkeit, mit der Software entwickelt werden kann. Es wird einfacher für alle beteiligten Entwickler, den Quellcode zu verstehen und zu überblicken. Somit können Fehler schneller gefunden und beseitigt sowie neue Funktionen schneller hinzugefügt werden.

*“Programs have two kinds of value: what they can do for you today and what they can do for you tomorrow.” [Fo99 S.60]*



### 2.2.4 Probleme und Risiken von Refactoring

Die meisten Refactorings sind einfache Einzelschritte wie zum Beispiel das Umbenennen einer Variable. Viele Programmierer denken daher, dass Refactoring sich nicht lohnt. Ihnen fällt es schwer, die Vorteile zu erkennen. Einige Manager vertreten die Ansicht, dass jegliche Zeit, die ein Entwickler für Refactoring verwendet, vergeudet sei. In der Praxis kann es sich als schwierig herausstellen, solche Personen vom Gegenteil zu überzeugen. In in diesem Fall rät Fowler [Fo99 S.61] dazu, heimlich zu refaktorisieren.

Auf der anderen Seite kann es gefährlich sein, wenn Refactoring von Entwicklern überbewertet wird. Ein dringender Liefertermin kann ernsthaft gefährdet werden, wenn kurz vor Ablauf der Frist refaktorisiert wird. In diesem Szenario ist es ratsam, nach Einhaltung des Termins die angesammelte Refactoring-Schuld zu begleichen. Problematisch ist dabei, dass dies häufig nicht mehr erledigt wird.

Es besteht weiterhin die Gefahr, dass durch Refactoring *veröffentlichte Schnittstellen* verändert werden. Laut Fowler [Fo99 S.64] ist eine Schnittstelle als veröffentlicht zu betrachten, wenn vom Entwickler nicht sichergestellt werden kann, dass alle Klienten der Schnittstelle vom Refactoring erreicht und entsprechend aktualisiert werden können. In diesem Fall darf beim Refactoring die alte Schnittstelle nicht verändert werden. Stattdessen muss eine neue Schnittstelle eingeführt und die alte Schnittstelle als *deprecated* gekennzeichnet werden. Mit *automatisierten Modultests* können durch Refactoring entstandene Fehler aber schnell erkannt und behoben werden.

Oft wird behauptet, Refactoring wirke sich negativ auf die Performance eines Programms aus. Dies ist meistens nicht der Fall, da zusätzliche Code-Infrastruktur nur eine vernachlässigbare Auswirkung auf die Gesamtperformance hat. Fowler [Fo99 S.69] legt nahe, dass diese Auswirkungen nur in Systemen mit harten Echtzeitanforderungen eine Rolle spielen. Es wirkt sich in den meisten Fällen auch nicht negativ aus, wenn nach einem Refactoring Programmlogik doppelt so oft aufgerufen wird (wie im Beispiel auf Seite 10). Eine Ausnahme stellen Methoden dar, in denen teure Operationen wie Datenbank- oder Dateizugriffe durchgeführt werden. Refactoring wirkt sich insgesamt positiv auf die Performance aus, da Werkzeuge zur Performance-Analyse in häufig refaktoriertem Quellcode auf kleinere Programmabschnitte verweisen. Somit können Programmteile, die häufig aufgerufen werden – sogenannte *Hot Spots* – schneller gefunden, isoliert und angepasst werden.

## 2.3 Eclipse RCP

### 2.3.1 Überblick

*Eclipse* wurde von *IBM* entwickelt und war ursprünglich als *Open Source Java IDE* konzipiert. Die IDE war qualitativ hochwertig und benutzerfreundlich, wodurch sie sich unter Programmierern schnell verbreitete. Im Jahr 2004 gründete *IBM* die eigenständige *Eclipse Foundation*, welche seitdem mit der Weiterentwicklung des Projekts beauftragt ist.

Entwickler hatten bald den Wunsch, das Werkzeug um eigene Funktionen zu erweitern. Daher wurde ein Mechanismus implementiert, mit dem die Software um eigene *Module* erweitert werden konnte. Diese Module bezeichnet man auch als *Plug-ins*. So wurden beispielsweise Erweiterungen veröffentlicht, welche die Entwicklung von *C++* Programmen oder Webapplikationen ermöglichten. Dadurch wurde die IDE noch vielseitiger und bekannter.

Seit der im Jahr 2004 erschienenen Version 3 besteht *Eclipse* selbst nur noch aus einem kleinen Kern, der die installierten Plug-ins lädt. Zu diesem Zweck wurden die einzelnen Bestandteile der IDE ebenfalls zu Plug-ins umgestaltet. Der Kern zusammen mit einigen grundlegenden Framework-Modulen wird als *Eclipse RCP* bezeichnet.

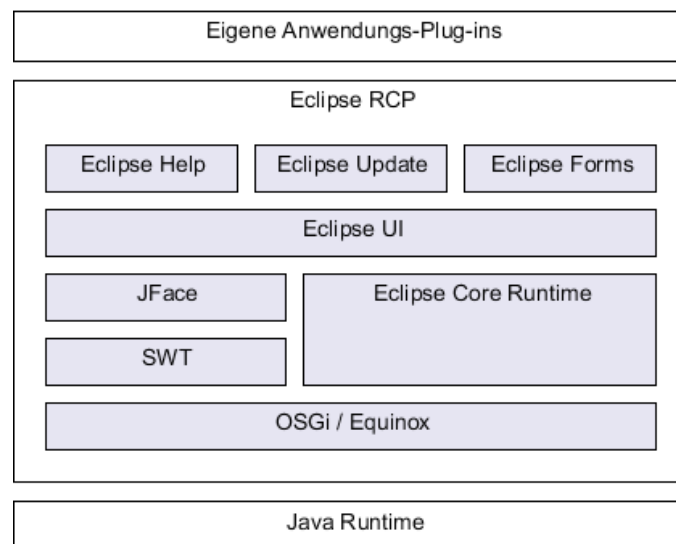


Abbildung 7: Bestandteile der Eclipse RCP

[Eb09] (Grafik überarbeitet)

Somit ist es möglich, auf Basis der Eclipse RCP Client Anwendungen aller Art zu entwickeln. Die einzelnen Bestandteile der Eclipse RCP seien an dieser Stelle kurz erläutert:

- **OSGi / Equinox**

Die *OSGi Service Platform* [OSGi] ist die Spezifikation eines dynamischen Modulsystems für Java. Das Plug-in Equinox [Equinox] ist eine Implementierung dieser Spezifikation. Das Modul ist zuständig für das Installieren, Aktivieren, Deaktivieren und Deinstallieren aller anderen Plug-ins. Dies kann (mit Einschränkungen) zur Laufzeit geschehen.

- **Eclipse Core Runtime**

Dieses Modul stellt APIs zur Verfügung, welche die Plattform selbst betreffen. Das Plug-in ist außerdem für den Start und die Initialisierung der Anwendung verantwortlich.

- **SWT**

*SWT* [SWT] ist ein Framework zur Programmierung grafischer Oberflächen. Dabei wird auf die nativen UI-Elemente des Betriebssystems zurückgegriffen, um die Performance zu verbessern und das *Look-and-Feel* des verwendeten Betriebssystems zu gewährleisten.

- **JFace**

*JFace* ist ein Framework, welches auf Grundlage von SWT Basisklassen für Dialoge, *Wizards* und vielem mehr zur Verfügung stellt. Wizard ist die Bezeichnung für spezielle Dialoge, die Nutzer über mehrere sogenannte Wizardseiten durch komplexe Vorgänge führen.

- **Eclipse UI**

Dieses Plug-in stellt die *Eclipse Workbench* bereit. Das ist ein leerer Anwendungsrahmen, der von anderen Modulen mit *Views*, *Editoren* und *Perspektiven*, den primären Bestandteilen der Eclipse Benutzeroberfläche, angereichert werden kann. Mit Editoren können Dateien bearbeitet werden, wohingegen Views die Navigation durch Hierarchien, das Öffnen von Editoren oder die Anzeige von Eigenschaften bezüglich der geöffneten Datei erlauben. Perspektiven ermöglichen das Organisieren sowie Anordnen von Views und Editoren.

- **Eclipse Help**

*Eclipse Help* stellt ein Hilfesystem inklusive Suchfunktion zur Verfügung.

- **Eclipse Update**

Dieses Modul ermöglicht das Herunterladen und Aktualisieren von Plug-ins.

- **Eclipse Forms**

*Eclipse Forms* ermöglicht die Entwicklung plattformunabhängiger Formulare wie sie in Webanwendungen vorkommen.

Die Software steht unter der *Eclipse Public License* [EPL]. Diese Open Source Lizenz erlaubt auch die kommerzielle Nutzung.

### 2.3.2 Extension Point Mechanismus

Ein Plug-in muss gegebenenfalls Code aufrufen, der erst durch andere Plug-ins bereit gestellt werden soll. Das Modul Eclipse UI muss zum Beispiel Klassen anderer Module laden, welche die Workbench um einzelne Views, Editoren und Perspektiven erweitern.

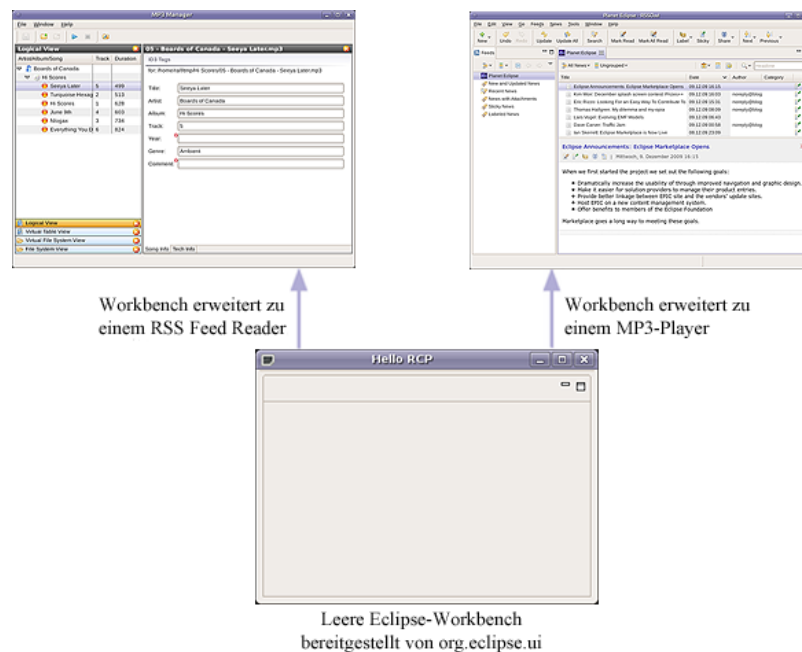


Abbildung 8: Erweiterung der Eclipse Workbench  
[Eb09] (Grafik überarbeitet)

Oft möchten die Entwickler eines Plug-ins es anderen Modulen ermöglichen, sich in bestimmte Abläufe einzuhängen. Ein Plug-in kann daher sogenannte *Extension Points* definieren. Diese können Einstiegspunkte für Quellcode anderer Module bieten. Ein Plug-in, welches einen durch ein anderes Plug-in zur Verfügung gestellten Extension Point nutzen möchte, registriert zu diesem Zweck eine *Extension*. Extension Points und Extensions werden mit Hilfe von XML [XML] definiert.

„**Extension Point:** Defines the possibility to add functionality“ [Vo09]

„**Extension:** Uses the extension point to provide functionality, also known as contributions.“ [Vo09]

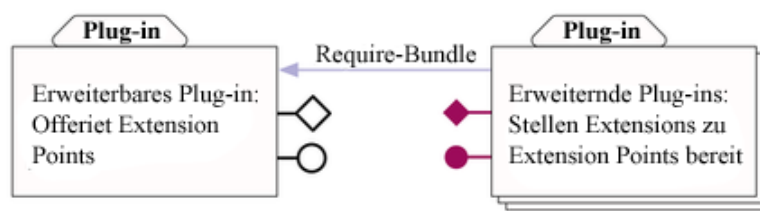


Abbildung 9: Extension Point Mechanismus

[Eb09]

### 2.3.3 Vorteile der Eclipse RCP

Anwendungen auf Basis der Eclipse RCP zu entwickeln bietet mehrere Vorteile:

- Es kann auf eine Vielzahl fertiger Module zurückgegriffen werden, welche Basistechnologien wie Versionsverwaltung integrieren.
- Die Plattform bringt mit SWT und JFace mächtige Frameworks zur Entwicklung grafischer Oberflächen mit.
- Der Eclipse Update-Mechanismus und das Eclipse Hilfesystem kann für eigene Anwendungen genutzt werden.
- Durch die modulare Architektur ist es einfach, verschiedene Distributionen eines Programms bereitzustellen.
- Anwender, die bereits mit einer Eclipse RCP Applikation gearbeitet haben, finden sich dank der einheitlichen Bedienung schnell zurecht.
- Durch den Extension Point Mechanismus und den Plug-in Mechanismus ist es ohne weiteren Aufwand möglich, eigene Anwendungen modular und erweiterbar zu gestalten.

## 2.4 Faktor-IPS

### 2.4.1 Überblick

*Faktor-IPS* ist ein Open Source Werkzeug zur modellgetriebenen Entwicklung von Versicherungssoftware und wird von der *Faktor Zehn AG* seit 2006 entwickelt. Der Name Faktor-IPS setzt sich aus dem Firmennamen und der Abkürzung *IPS* für *Insurance Product System (Versicherungs-Produkt-System)* zusammen. Faktor-IPS ist in die Eclipse IDE integriert.

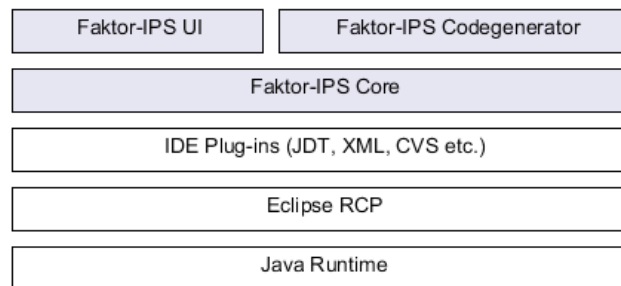


Abbildung 10: Einordnung von Faktor-IPS in die Komponentenstruktur der Eclipse IDE

Das Programm zielt darauf ab, als Fundament in der IT-Landschaft von Versicherungen eingesetzt zu werden. Um diesem Anspruch gerecht zu werden, muss grundsätzlich zwischen zwei verschiedenen *Anwender-Rollen* unterschieden werden:

- **Anwendungsentwickler** nutzen Faktor-IPS, um das Geschäftsmodell der Versicherung zu modellieren und um die Geschäftslogik zu implementieren.
- **Facharbeiter** pflegen auf Grundlage des Geschäftsmodells Produktdaten und definieren fachliche Testfälle.

Problematisch ist dabei, dass Facharbeiter sicher mit den Funktionen überfordert wären, welche das Programm den Anwendungsentwicklern bietet. Da Faktor-IPS auf der Eclipse RCP basiert, ist es mit wenig Aufwand möglich, spezielle Distributionen des Programms für Facharbeiter bereitzustellen. Diese beinhalten nur die für die Rolle relevanten Komponenten.

Zur Persistenz der Modelle und Produktdaten wird das Dokumentformat XML eingesetzt. Projekte werden über gängige Versionsverwaltungssysteme wie *CVS* [CVS] oder *SVN* [SVN] synchronisiert. Die Eclipse Community unterstützt diese Technologien, wodurch Entwicklungsaufwand eingespart werden kann.

Faktor-IPS steht unter einer speziellen Open Source Lizenz, welche durch die Faktor Zehn AG verabschiedet wurde. Das Open Source Modell bringt Vorteile für Hersteller und Kunden:

- Kunden müssen kein Geld investieren, um die Software testen und evaluieren zu können.
- Von Weiterentwicklungen, die durch einen Kunden initiiert und finanziert werden, profitieren alle anderen Kunden ebenso.
- Für die Faktor Zehn AG gestaltet es sich einfacher Neukunden zu gewinnen, da zunächst keine Investitionen von Seiten des potentiellen Kunden notwendig sind.

Die Faktor Zehn AG finanziert sich über Dienstleistungen wie Beratung, Schulung und Weiterentwicklung von Faktor-IPS. Bei der Umstellung der IT auf das neue System unterstützen Mitarbeiter der Faktor Zehn AG die Anwendungsentwickler vor Ort.

## 2.4.2 Modellierung

Das Faktor-IPS Metamodell ist stark an das Java Metamodell angelehnt. Es gibt genau wie in Java Klassen, die wiederum Attribute, Methoden und Beziehungen enthalten können. Vererbung und abstrakte Klassen sind ebenso möglich. Im Wesentlichen führt das Faktor-IPS Metamodell zwei Spezialisierungen des Konstrukts *Klasse* ein: *Vertragsteilklass*e und *Produktbausteinklass*e.

Das folgende Diagramm zeigt einen Ausschnitt aus dem Faktor-IPS Metamodell. Dieser Ausschnitt enthält nur die für diese Arbeit relevanten Elemente des Modells. In Wirklichkeit ist das Faktor-IPS Metamodell noch weit umfangreicher.

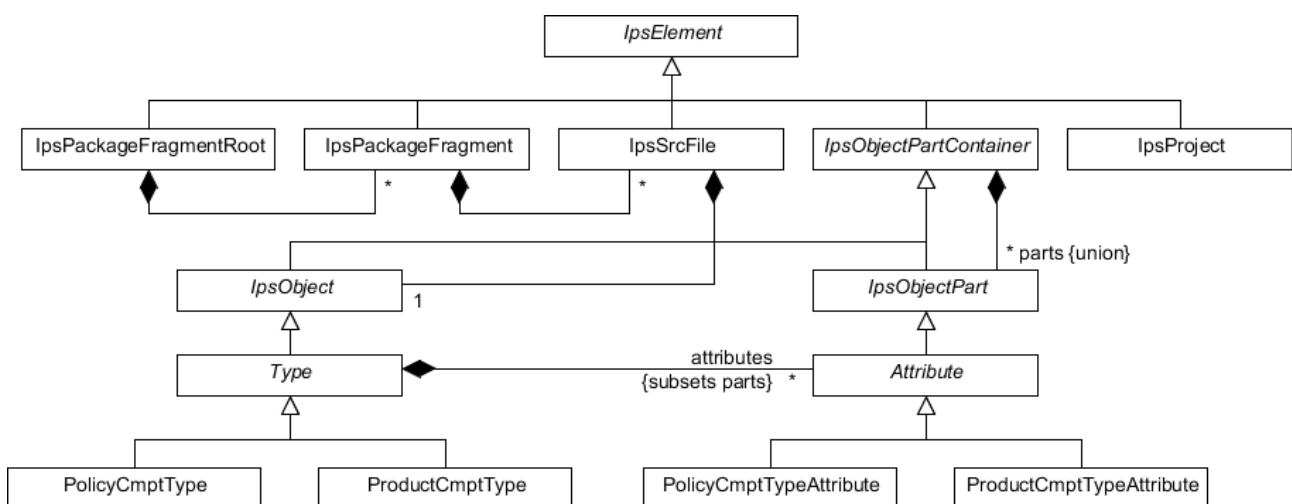


Abbildung 11: Ausschnitt aus dem Faktor-IPS Metamodell

Die Bezeichnungen der Elemente wurden nicht übersetzt, daher sei angemerkt, dass das Element *Type* mit *Klasse* ins Deutsche übersetzt wird. Auf den ersten Blick könnte es verwirrend sein, dass die Klasse *IpsObjectPart* von der Klasse *IpsObjectPartContainer* abgeleitet ist. Das Design

begründet sich auf der Tatsache, dass eine Methode wiederum Parameter enthalten kann. Dieser Umstand ist zur Vereinfachung im Diagramm aber nicht abgebildet.

Um die Bedeutung dieser Konstrukte verstehen zu können, ist es zunächst notwendig, das grundlegende Geschäftsmodell von Versicherungsunternehmen zu betrachten. Bei einem Versicherungsunternehmen werden *Verträge* mit Kunden abgeschlossen. In einem Vertrag werden Eigenschaften wie Beitragssumme, Adressdaten oder Wirksamkeitsdatum definiert. Verträge wiederum basieren auf *Produkten*, in denen zum Beispiel festgelegt wird, welche Schadensfälle von der Versicherung übernommen werden. Ein Produkt kann die Eigenschaften eines Vertrags weiter einschränken und so beispielsweise eine minimale und maximale Beitragssumme definieren. Produkte sind somit Konfigurationen von Verträgen. In Faktor-IPS sind Produkte immer Instanzen einer Produktbaustein-Klasse.

Die folgende Abbildung soll die Zusammenhänge am Beispiel einer stark vereinfachten Hausratversicherung verdeutlichen.

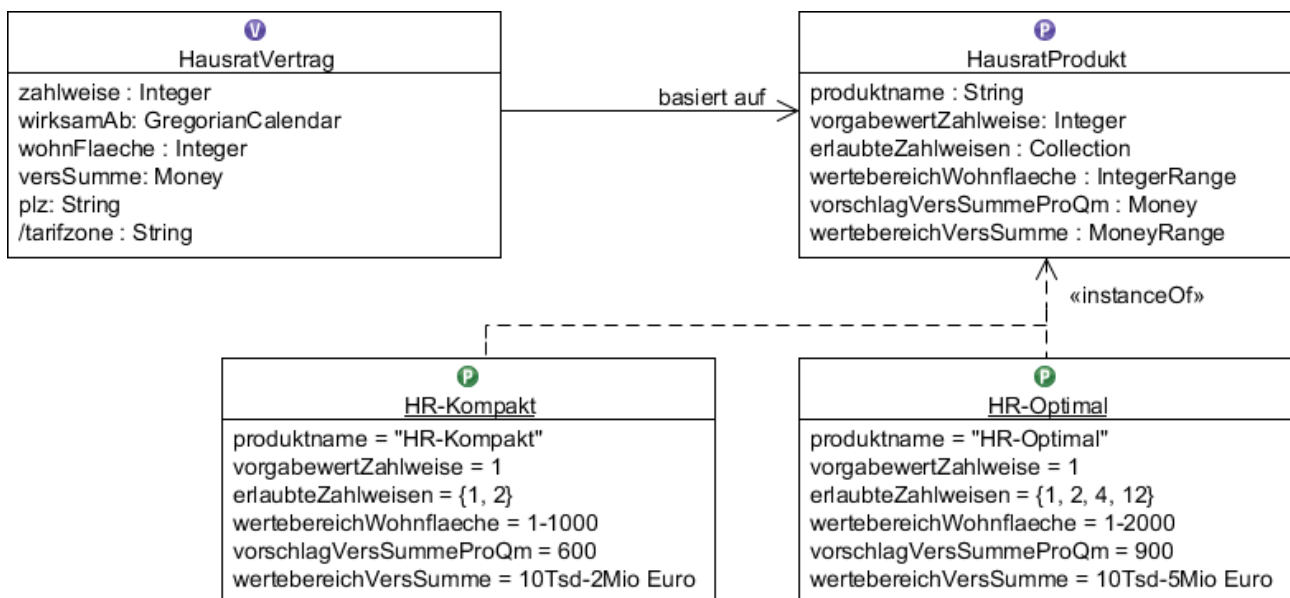


Abbildung 12: Zusammenhang zwischen Modellklassen und Produktinstanzen

[Or09 S.18] (Grafik überarbeitet)

Vertragsinstanzen werden nicht mit Faktor-IPS gepflegt. Das ist Aufgabe der Vertragsverwaltung bzw. des Bestandssystems des Versicherungsunternehmens. Solche Systeme setzen in der Regel auf einer Datenbank auf. Mit Faktor-IPS lassen sich solche Bestandssysteme aber entwickeln.

Zum Erstellen und Bearbeiten der Modelle bietet Faktor-IPS spezielle *Views*, *Editoren* und *Perspektiven* an (siehe Eclipse UI in Kap. 2.3.1, S. 15).



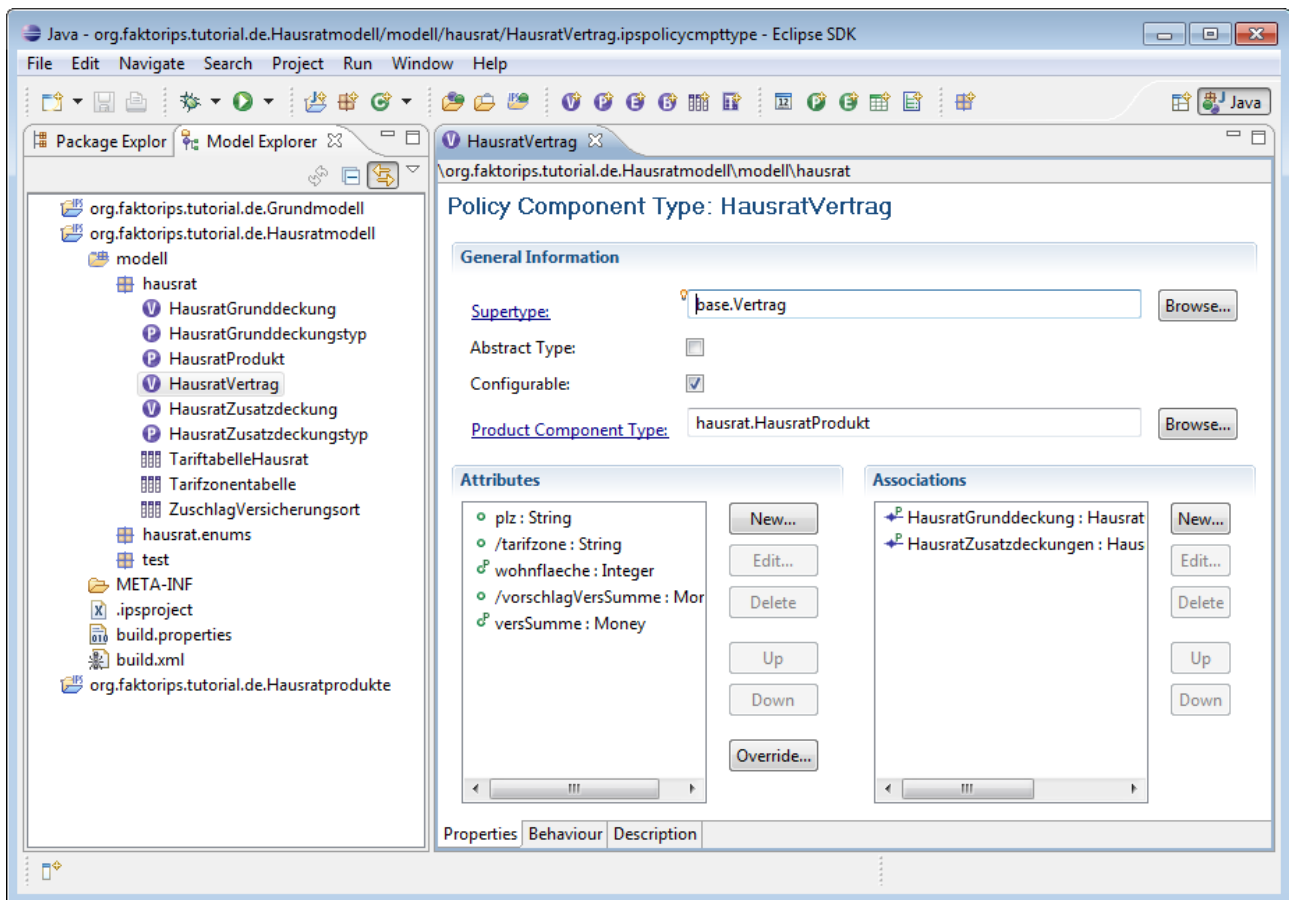


Abbildung 13: Faktor-IPS mit geöffnetem Model Explorer und Vertragsteilklassen-Editor

### 2.4.3 Codegenerierung

Aus den mit Faktor-IPS erzeugten Modellen wird nach dem Prinzip von MDSD automatisch Quellcode generiert. Diese Aufgabe übernimmt der Faktor-IPS Codegenerator. Zum gegenwärtigen Zeitpunkt unterstützt das Werkzeug nur Java als Zielprogrammiersprache.

Welche Programmfragmente beim Prozess der Codegenerierung für ein gegebenes Modellelement erzeugt werden, ist in der Programmlogik des Codegenerators verankert. Dadurch wird im Generator jedem Modellelement eine formal spezifizierte Bedeutung zugewiesen (siehe dynamische Semantik in Kap. 2.1.2, S. 6).

Um den Codegenerator zu starten, registriert Faktor-IPS über einen Extension Point einen sogenannten *Incremental Builder*, zu deutsch inkrementellen Erbauer.

„**Builders** take raw materials and produce some output based on those materials. [...]“  
[Ar03]

Ein inkrementeller Erbauer erzeugt nicht bei jedem Bauvorgang die komplette Ausgabe von Grund auf neu. Stattdessen werden nur diejenigen Teile neu gebaut, welche sich seit dem letzten Bauvorgang verändert haben. Inkrementelle Erbauer werden von der Plattform bereits gestartet, sobald eine Datei modifiziert wurde.

Bevor die Codegenerierung gestartet wird, erfolgt eine Überprüfung der statischen Semantik des Modells (siehe Definition in Kap. 2.1.2, S. 6). Dadurch wird festgestellt, ob das Modell in einem validen Zustand ist. Das wäre zum Beispiel nicht der Fall, wenn einem Attribut kein Datentyp zugewiesen wurde. Dieser Vorgang wird *Validierung* genannt. Rote Problemmarker zeigen dem Benutzer die bei diesem Prozess erkannten Fehler im Modell auf.

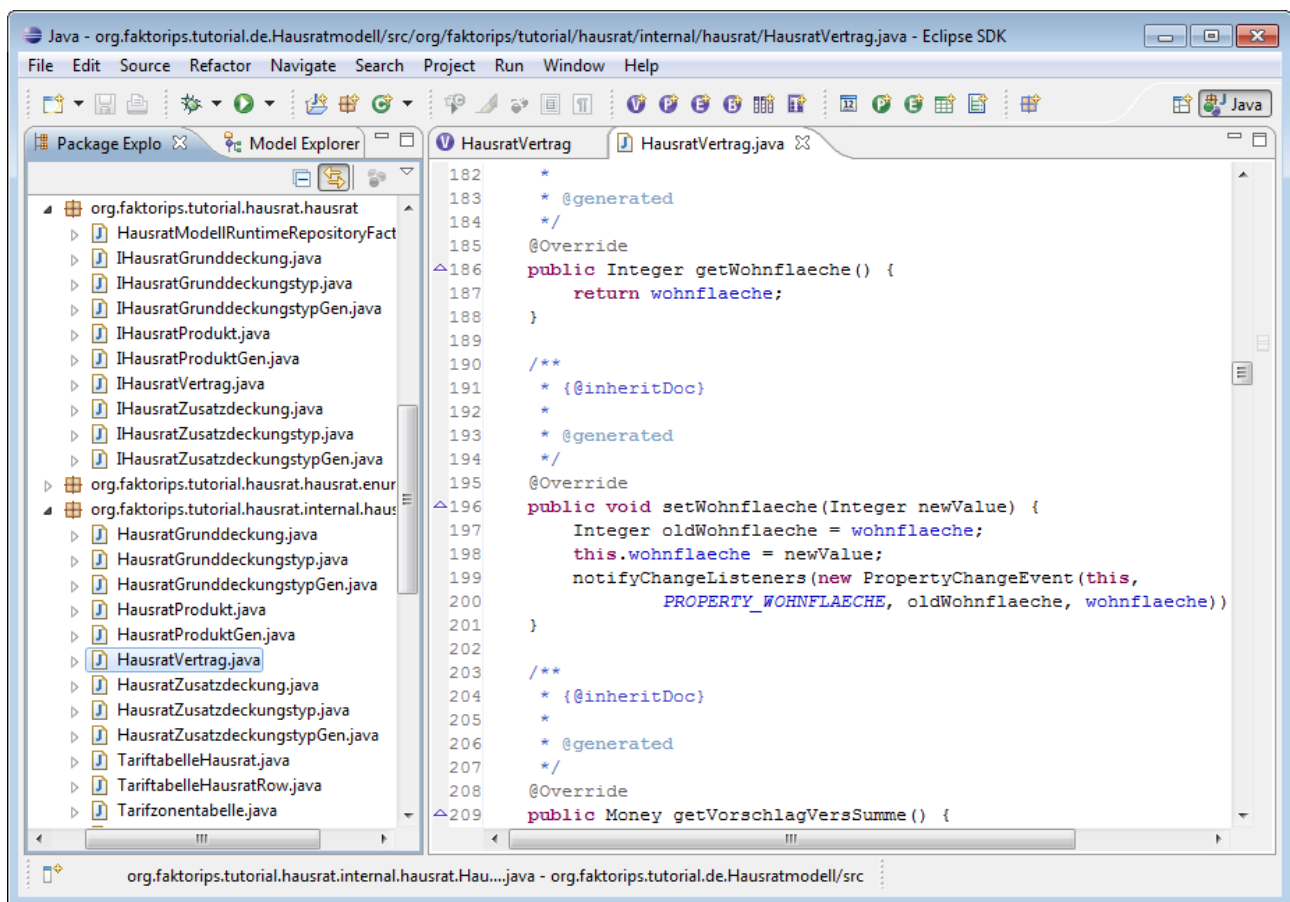


Abbildung 14: Von Faktor-IPS automatisch generierter Java Quellcode

## 3 Analyse und Konzept

### 3.1 Problemanalyse

Im Folgenden wird die Problematik von Refactoring im Kontext von MDSD erklärt. Abbildung 15 zeigt die grundsätzlich notwendigen Schritte, die eine Integration im Rahmen eines Refactorings leisten muss.

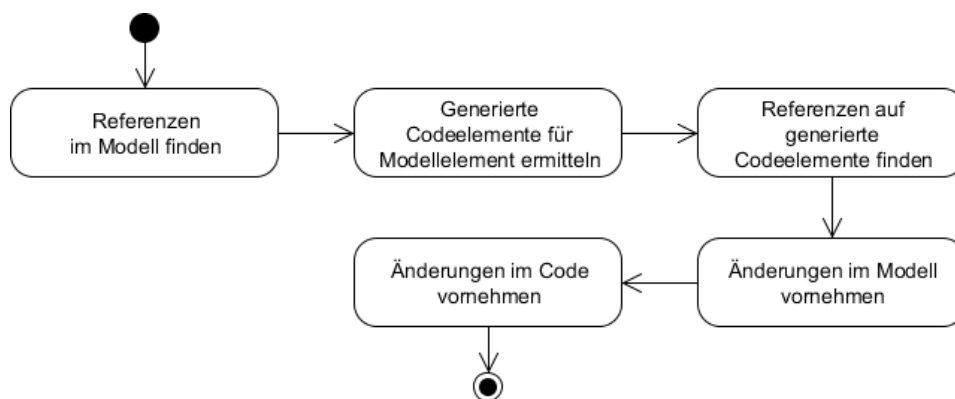


Abbildung 15: Grundsätzlich notwendige Aktivitäten für Refactoring im Kontext von MDSD

#### 3.1.1 Modellseitige Referenzen

Soll im Kontext von MDSD ein Modellelement automatisch refaktorisiert werden, so müssen zunächst alle Referenzen auf das Element im Modell gefunden werden. Diese sind nach einer Modifikation des Modellelements nicht mehr aktuell und müssen daher aktualisiert werden.

Abbildung 16 zeigt zur Verdeutlichung das Modell einer vereinfachten Hausratversicherung mit vier Klassen: *HausratVertrag* und *Vertrag* sowie *HausratDeckung* und *Deckung*.

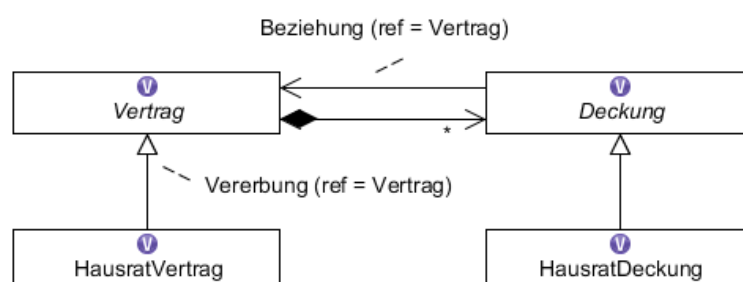


Abbildung 16: Modellseitige Referenzen am Beispiel eines Hausratmodells

Wird die Klasse *Vertrag* umbenannt oder verschoben, so müssen die Vererbungsreferenz von *HausratVertrag* und die Beziehungsreferenz von *Deckung* aktualisiert werden.

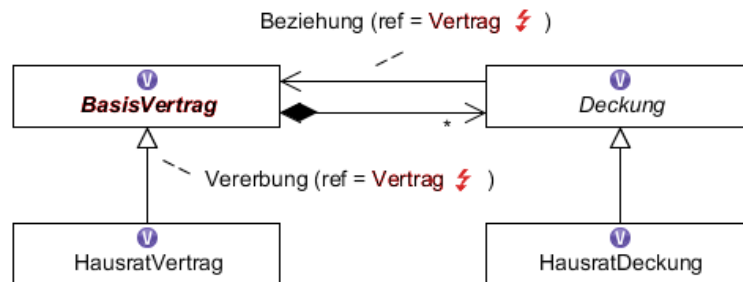


Abbildung 17: Invalide Modellreferenzen nach Umbenennen eines Modellelements

Abhängig davon, ob der Quellcode des MDSD-Tools über die notwendige Infrastruktur (siehe Kap. 3.2.1) verfügt oder nicht, kann eine derartige Referenzsuche und Referenzaktualisierung unterschiedlich aufwändig sein.

### 3.1.2 Nicht-generierter Quellcode


Nach dem Durchführen der notwendigen Änderungen im Modell ist es nicht ausreichend, eine Neugenerierung des Quellcodes anzustoßen, damit sich die Auswirkungen des Refactorings dort widerspiegeln. Das eigentliche Problem bei Refactoring im Kontext von MDSD ist der nicht-generierte Anteil des Quellcodes.

Aus den in Kapitel 2.1.4 genannten Gründen werden in der Regel nicht 100% des Programmcodes einer Anwendung vom MDSD-Werkzeug generiert. Daher werden durch eine Neugenerierung auch nicht alle Teile des Quellcodes vom Refactoring erreicht. Um den nicht-generierten Anteil zu aktualisieren, benötigt man eine Liste aller Codeelemente, die für das veränderte Modellelement ursprünglich – also noch bevor das Refactoring angestoßen wurde – generiert wurden. Der nicht-generierte Quellcode muss anschließend nach Referenzen zu diesen Codeelementen abgesucht und entsprechend angepasst werden.

Das folgende Codebeispiel illustriert das Problem des nicht-generierten Quellcodes anhand dem Umbenennen eines Attributs *plz*, welches zu *postLeitZahl* umbenannt werden soll. Es werden zwei Methoden betrachtet, wobei beide Methoden über die Annotation *@generated* verfügen. Bei der zweiten Methode *printPlz()* steht hinter der Annotation allerdings ein *NOT*, welches diese Methode als nicht-generiert markiert. Sie wird demnach bei einer Neugenerierung vom Codegenerator nicht modifiziert.

**Listing 3:** Programmcode vor Umbenennen eines Faktor-IPS Attributes

```
1  /** @generated */
2  private String plz;
3
4  /** @generated */
5  public String getPlz() {
6      return plz;
7  }
8
9
10 /** @generated NOT */
11 public void printPlz() {
12     System.out.print(plz);
13 }
```

Um-  
benennen  
  
Neu-  
generieren

**Listing 4:** Programmcode nach Umbenennen eines Faktor-IPS Attributes

```
1  /** @generated */
2  private String postLeitZahl;
3
4  /** @generated */
5  public String getPostLeitZahl()
6  {
7      return postLeitZahl;
8  }
9
10 /** @generated NOT */
11 public void printPlz() {
12     System.out.print(plz);
13 }
```

Die **rot** markierte Referenz ist ein Syntaxfehler (kein gültiger Variablenbezeichner) und bewirkt, dass das Programm nicht mehr kompiliert werden kann. Das **grün** markierte Vorkommnis *plz* im Namen der Methode *printPlz()* verursacht keine unmittelbaren Schwierigkeiten, ist aber semantisch nicht korrekt und könnte auf Dauer zu Verständnisproblemen führen.

## 3.2 Voraussetzungen für eine Integration

### 3.2.1 Referenzsuche und Referenzaktualisierung auf Modellebene

Um modellseitige Referenzen zu aktualisieren ist es möglich, während dem Refactoring über alle Modellelemente zu iterieren, welche Referenzen enthalten könnten. Dabei wird nach dem Namen des zu verändernden Elements gesucht und bei einem Fund das referenzierende Element über eine Setter-Methode angepasst.

Wesentlich besser ist es aber, die Referenzsuche und Referenzaktualisierung in die Implementierung der Metamodell-Elemente zu integrieren. Dies hat verschiedene Vorteile:

- Eine Referenzsuche lässt sich sicher auch an anderen Stellen des Programms sinnvoll verwenden.
- Das Design wird klarer und die Funktion ist getrennt vom Refactoring-Aspekt testbar.
- Bei einer Erweiterung oder Änderung des Metamodells muss der für Refactoring zuständige Code nicht angepasst werden.

In dieser Arbeit wurde der erste Ansatz gewählt, da eine Suche und Aktualisierung von Referenzen in Faktor-IPS gegenwärtig nicht implementiert ist und im Rahmen der Arbeit nicht umsetzbar gewesen wäre. Dennoch soll an dieser Stelle ein Designvorschlag zur Entwicklung dieses Features für Faktor-IPS gegeben werden. Der Entwurf sollte auf andere MDSD-Werkzeuge übertragbar sein.

In Faktor-IPS sind Referenzen als *Strings* realisiert, welche den qualifizierten Namen des referenzierten Modellelements beinhalten. Ein qualifizierter Name ist eine Zusammensetzung aus dem Namen eines Modellelements und dem Namen des beinhaltenden Pakets. Der Supertyp einer abgeleiteten Klasse kann beispielsweise mit folgenden Methoden abgefragt und gesetzt werden:

**Listing 5:** Methoden zum Abfragen und Setzen der Supertype-Eigenschaft in Faktor-IPS

```
1  /** Returns the qualified name of this type's supertype. */
2  public abstract String getSupertype();
3
4  /** Sets the qualified name of this type's supertype. */
5  public abstract void setSupertype(String supertype);
```

Das Problem dabei ist, dass für eine allgemein gültige Referenzaktualisierung eine einheitliche Schnittstelle benötigt wird. Daher müssen alle *Strings*, die der Referenzierung anderer Modellelemente dienen, zu Objekten des Typs *IReference* umgestaltet werden. Referenzen dieses Typs können dann über eine einheitliche Schnittstelle aktualisiert werden.

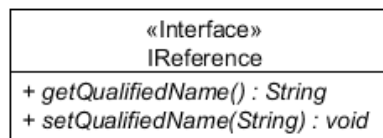


Abbildung 18: Das Interface *IReference*

Das Projekt, in welchem ein *IpsObjectPartContainer* (siehe Abb. 11 auf S. 19) gespeichert ist, kann von diesem aus erreicht werden. Das entsprechende Interface heißt *IProject* und muss um folgende Methode erweitert werden.

**Listing 6:** Signatur einer Methode zur Erzeugung einer Instanz von *IReference*

```
1  /** Creates a new instance of IReference. */
2  public abstract IReference getReference(String qualifiedName);
```

Somit ist die Erzeugung von *IReference*-Instanzen an einer zentralen Stelle positioniert, was unter anderem den Austausch der darunterliegenden Implementierung vereinfacht.

Jede konkrete, von *IpsObjectContainer* abgeleitete Klasse muss folgende Methode implementieren, welche alle Referenzen zu einem anderen *IpsObjectContainer* liefert.

**Listing 7:** Signatur einer Methode zum Abfragen von Referenzen auf andere Modellelemente

```
1  /** Returns any references to the given IpsObjectPartContainer. */
2  public abstract List<IReference> getReferences (
3      IIpsObjectPartContainer otherObjectPartContainer);
```

Weiterhin muss die Schnittstelle der Klasse eine Methode anbieten, welche die Suche und Aktualisierung aller Referenzen bewerkstelligt. Abbildung 19 illustriert den Ablauf dieser Methode.

**Listing 8:** Signatur einer Methode zur Aktualisierung von modellseitigen Referenzen

```
1  /**
2   * Updates all references to this IpsObjectPartContainer.
3   * @throws CoreException If an error occurs while searching the file system.
4   */
5  public abstract void findAndUpdateReferencingObjects (
6      String newQualifiedNames,
7      IProgressMonitor progressMonitor) throws CoreException;
```

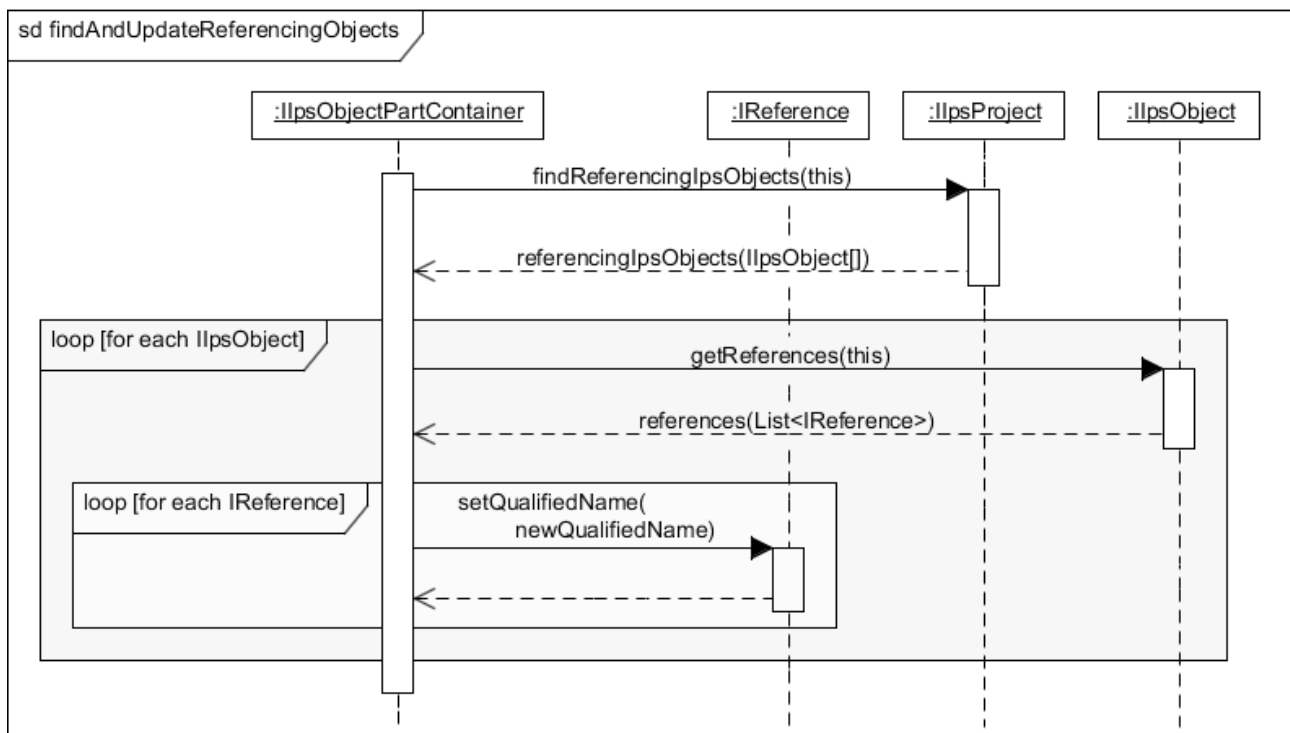


Abbildung 19: Referenzsuche und Referenzaktualisierung auf Modellebene für Faktor-IPS

Dieser Lösungsvorschlag sollte allerdings nur als erster Denkanstoß verstanden werden. Die konkrete Ausarbeitung muss an das jeweilige MDSD-Werkzeug angepasst werden. Es werden

Änderungen an den Implementierungen aller Metamodell-Elemente notwendig. Daher erfordert die Entscheidung, wie die Referenzsuche und Referenzaktualisierung auf Modellebene letztendlich zu realisieren ist, eine Abstimmung unter allen Entwicklern.

### 3.2.2 Ermittlung generierter Codeelemente

Wie bereits in Kapitel 3.1.2 erklärt, wird beim Refactoring eine Liste aller Codeelemente benötigt, welche für ein Modellelement generiert werden. Erst dann ist es möglich, in nicht-generierten Programmteilen nach Referenzen auf diese Codeelemente zu suchen.

Die benötigten Informationen liegen im Codegenerator verborgen. Für Refactoring im Kontext von MDSD muss der Generator in der Lage sein, zu einem gegebenen Modellelement eine Liste der generierten Codeelemente zu liefern.

Der Faktor-IPS Codegenerator nutzt ein Set mehrerer Klassen vom Typ *IpsArtefactBuilder* (im Folgenden auch als Erbauer bezeichnet). Dieses Set wird als *BuilderSet* bezeichnet. *IpsArtefactBuilder* sind dafür zuständig, aus einem gegebenen Modellelement ein bestimmtes Ziel-artefakt zu erzeugen.

Faktor-IPS liefert ein *StandardBuilderSet* aus, welches Java Quellcode generiert. Ein *JavaSourceFileBuilder* erzeugt dabei eine Java Klasse oder ein Java Interface aus einem gegebenen Modellelement. Innerhalb eines *JavaSourceFileBuilder* wird diese Aufgabe unter mehreren Objekten vom Typ *JavaGeneratorForIpsPart* aufgeteilt. Jeder dieser Generatoren erzeugt einen Teil des Java Programmcodes aus einem Teil des Modellelements. Der Begriff Generator ist in diesem Zusammenhang nicht mit der Codegenerator-Komponente als Ganzes zu verwechseln.

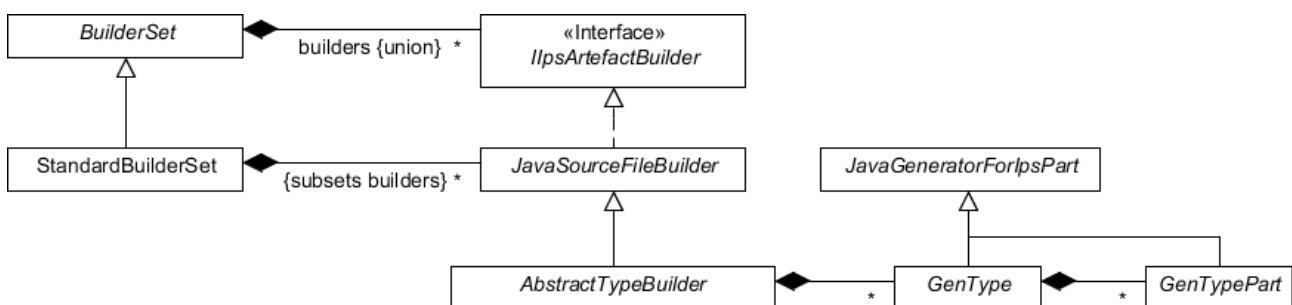


Abbildung 20: Architektur des Faktor-IPS Codegenerators

Die Klasse *StandardBuilderSet* muss um eine öffentliche Methode ergänzt werden, welche eine Beschreibung aller generierten Codeelemente für ein gegebenes Faktor-IPS Modellelement liefern kann:



**Listing 9:** Signatur einer Methode zur Ermittlung generierter Java Elemente

```

1  /**
2   * Returns a list containing all Java elements this builder set generates for
3   * the given IPS Element.
4   */
5  public List<IJavaElement> getGeneratedJavaElements(IIPsElement ipsElement);

```

Das Interface *IJavaElement* stammt vom Eclipse Plug-in *JDT* [JDT]. Das Modul stellt das Java Metamodell zur Verfügung und bietet ein API zur Erzeugung von Java Modellelementen an. Diese Elemente implementieren das Interface *IJavaElement*. In Abbildung 21 ist der für diese Arbeit relevante Ausschnitt der Klassenhierarchie abgebildet.

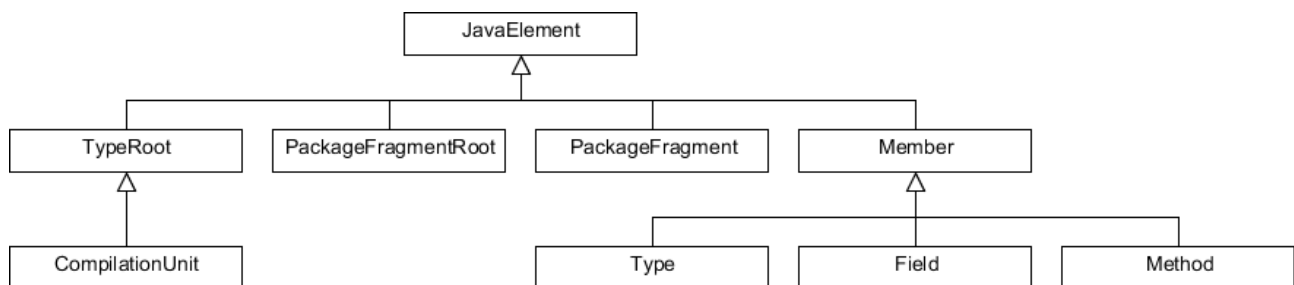


Abbildung 21: Ausschnitt der Klassenhierarchie des JDT Java Metamodells

Das *BuilderSet* delegiert die Anfrage an die zugehörigen *JavaSourceFileBuilder*. Diese wiederum beziehen die Information von den angehörigen Generatoren.

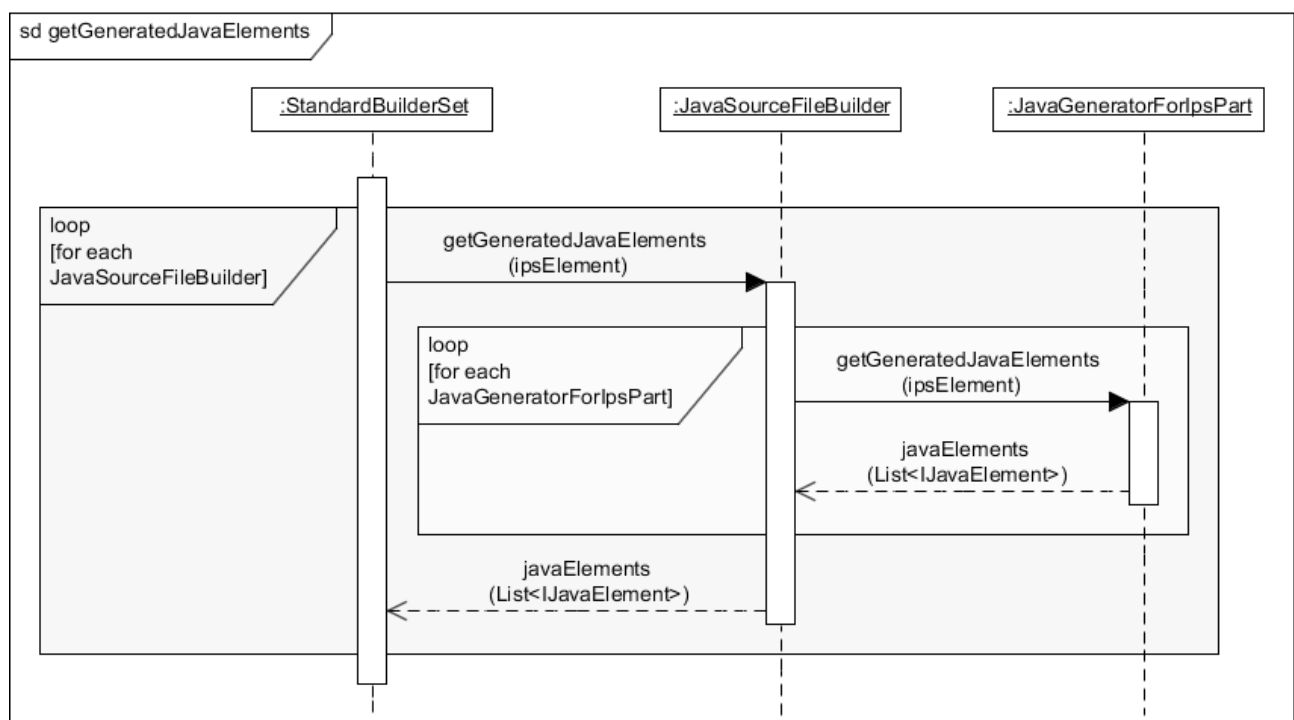


Abbildung 22: Auskunft über generierte Java Codeelemente in Faktor-IPS

### 3.2.3 Referenzsuche und Referenzaktualisierung auf Codeebene

Nach dem Modifizieren eines Modellelements ändert sich bei MDSD der generierte Anteil des Quellcodes. Der nicht automatisch erzeugte Programmcode wird somit fehlerhaft, wenn dort generierte Codeelemente referenziert werden (siehe Kap. 3.1.2).

Die Referenzsuche und Referenzaktualisierung auf Codeebene muss den nicht-generierten Programmcode nach Referenzen auf die aus einem Modellelement erzeugten Codeelemente durchsuchen. Diese Referenzen müssen so angepasst werden, dass sie dem neuen Stand des generierten Quellcodes entsprechen.

Für ein einzelnes Codeelement lösen konventionelle Refactorings dieses Problem bereits. Diese können im Rahmen einer Programmiersprachen-Unterstützung von einer modernen IDE automatisiert durchgeführt werden. Das Gesamtproblem lässt sich durch sequentielles Ausführen mehrerer konventioneller Refactorings lösen, welche auf die generierten Codeelemente angewandt werden. Es wird nicht mehr weiter zwischen generiertem und nicht-generiertem Programmcode unterschieden.

Bei dieser Lösung sollte unbedingt auf existierende Funktionalität zurückgegriffen werden, die von der IDE bzw. von einer Bibliothek zur Verfügung gestellt wird. Eine Eigenentwicklung an dieser Stelle ist aufwändig und teuer. Vor der Entwicklung einer Refactoring-Integration sollte daher überprüft werden, ob ein API zum programmatischen Starten von konventionellen Refactorings für die Zielprogrammiersprache verfügbar ist.

Faktor-IPS wird auf Basis der Eclipse RCP entwickelt und die Zielprogrammiersprache ist Java. Daher kann bei Faktor-IPS zum Refaktorisieren des Java Quellcodes auf das JDT Plug-in zurückgegriffen werden. Dieses Modul bietet ein API zum programmatischen Start von Java Refactorings an.

**Listing 10:** Programmcode vor Anwendung konventioneller Java Refactorings

```
1  /** @generated */
2  private String plz;
3
4
5
6  /** @generated */
7  public String getPlz() {
8      return plz;
9  }
10
11
12  /** @generated NOT */
13  public void printPlz() {
14      System.out.print(plz);
15  }
16
```



Rename  
Field



Rename  
Method

**Listing 11:** Programmcode nach Anwendung konventioneller Java Refactorings

```
1  /** @generated */
2  private String postLeitZahl;
3
4
5
6  /** @generated */
7  public String getPostLeitZahl()
8  {
9      return postLeitZahl;
10 }
11
12  /** @generated NOT */
13  public void printPlz() {
14      System.out.print(
15          postLeitZahl);
16  }
```

Das Java Refactoring *Rename Field* erreicht angewandt auf das Attribut *plz* auch die nicht-generierte Referenz in der Methode *printPlz()*. Analog würde das Java Refactoring *Rename Method* alle Referenzen auf die Methode *getPlz()* erreichen und diese so anpassen, dass sie von nun an auf die Methode *getPostLeitZahl()* verweisen. Nur noch das grün markierte Vorkommnis *plz* im Namen der Methode *printPlz()* weist darauf hin, dass das Attribut *postLeitZahl* ursprünglich einmal *plz* geheißen hat.

Die vom JDT angebotenen Java Refactorings können so konfiguriert werden, dass auch Übereinstimmungen in Bezeichnern und Kommentaren ersetzt werden. Das ist aber gefährlich, weil auch Vorkommnisse ersetzt werden, die mit dem veränderten Codeelement nichts zu tun haben. Der semantische Fehler sollte daher besser manuell korrigiert werden. Dazu kann vom MDSD-Werkzeug nach dem automatisierten Teil des Refactorings ein Dialog gestartet werden, der den Entwickler bei diesem Arbeitsschritt unterstützt.

### 3.3 Architektur

#### 3.3.1 Anforderungen

Die zentrale Anforderung an die Architektur einer Refactoring-Integration im Kontext von MDSD ist, dass keine Abhängigkeit vom Kern des MDSD-Werkzeugs zum Codegenerator erzeugt werden darf. Nur so kann sichergestellt werden, dass die Codegenerator-Komponente austauschbar bleibt.

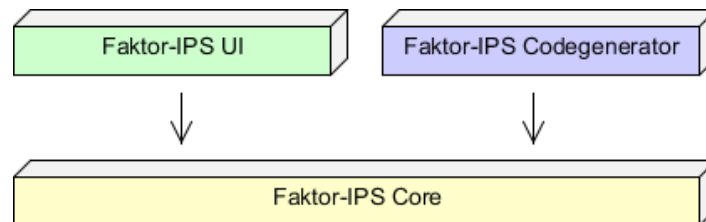


Abbildung 23: Schichtenarchitektur von Faktor-IPS

Die automatisierte Refactoring-Unterstützung wird im Kern entwickelt, aber nur der Codegenerator kann Auskunft über generierte Codeelemente geben. Dieser Umstand kann schnell zu einem Zyklus von Abhängigkeiten führen.

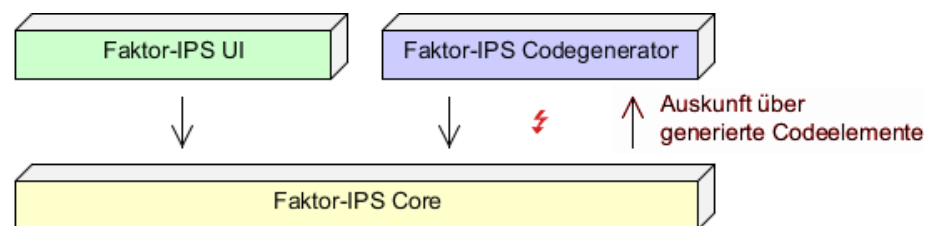


Abbildung 24: Zyklus von Abhängigkeiten zwischen Kern und Codegenerator

Im Kern darf zudem keine Annahme darüber getroffen werden, welche Zielprogrammiersprache eingesetzt wird. Es ist daher unmöglich, die in Kapitel 3.2.3 angesprochenen konventionellen Refactorings zur Aktualisierung des nicht-generierten Quellcodes von dort aus zu starten.

#### 3.3.2 Eclipse LTK Framework

Die Referenzsuche und Referenzaktualisierung auf Codeebene muss im Verantwortungsbereich des verwendeten Codegenerators liegen. Der Prozess muss vom Kern während dem Refactoring eines Modellelements angestoßen werden, ohne dass dabei eine Abhängigkeit zum Codegenerator erzeugt wird. Daher registriert der Codegenerator verschiedene *Refactoring-Teilnehmer* bei der Plattform. Während dem Refactoring werden alle registrierten Refactoring-Teilnehmer aufgerufen.

Die beschriebene Architektur wird bereits vom Eclipse Modul *LTK* (*Language ToolKit*) zur Verfügung gestellt. Das Plug-in ist aus der Java Refactoring-Unterstützung des JDT entstanden. Die von einer konkreten Programmiersprache unabhängigen Bestandteile wurden dazu vom JDT abgespalten.

Eclipse LTK bietet ein Framework mit einigen Basisklassen und Extension Points an. Es bildet somit die Basis aller automatisierten Refactorings für die Eclipse IDE. Die in Abbildung 25 dargestellten Klassen sind hierbei von zentralem Interesse.

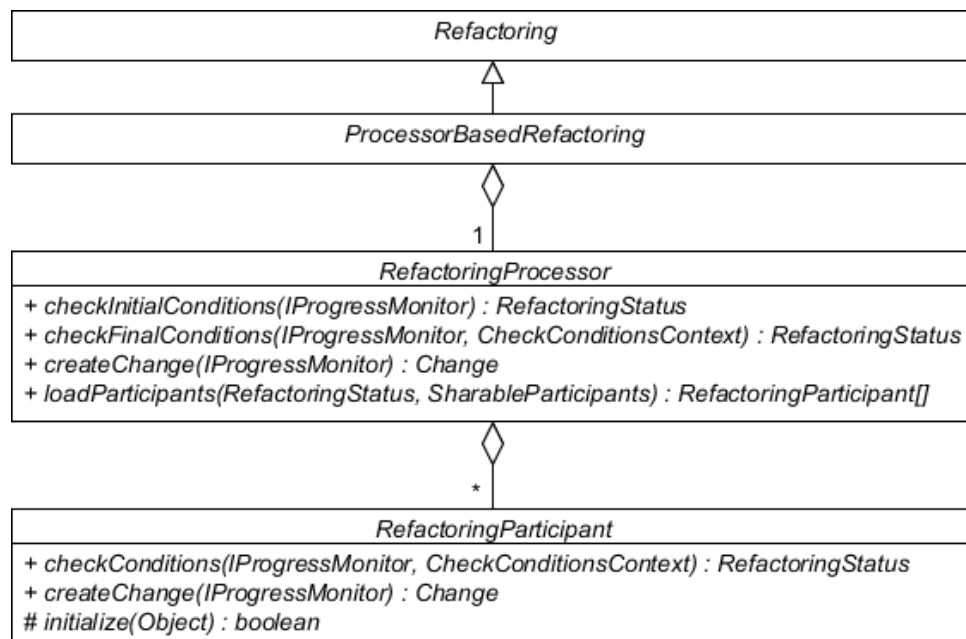


Abbildung 25: Die wichtigsten Klassen des Eclipse LTK Framework

- **Refactoring**

Die abstrakte Klasse *Refactoring* bildet die Basis für alle automatisierten Refactorings innerhalb der Eclipse IDE.

- **ProcessorBasedRefactoring**

Prozessorbasierte Refactorings sind in einen Refactoring-Prozessor (*RefactoringProcessor*) und beliebig viele Refactoring-Teilnehmer (*RefactoringParticipant*) aufgeteilt.

- **RefactoringProcessor**

Ein Refactoring-Prozessor ist dafür verantwortlich, das ausgewählte Element zu refaktorisieren. Dies beinhaltet das Überprüfen von Vorbedingungen, welche erfüllt sein müssen, damit das Refactoring korrekt ausgeführt werden kann. Das Laden aller registrierten Teilnehmer ist eine weitere Aufgabe des Refactoring-Prozessors.

### • RefactoringParticipant

Refactoring-Teilnehmer können sich bei der Überprüfung von Vorbedingungen und am Refaktorisieren des ausgewählten Elements beteiligen. Somit stellen sie Erweiterungen von Refactorings dar.

Zusätzlich sind Basisklassen für Wizards und Dialoge im Umfang des LTK Frameworks enthalten. Diese sind speziell für den Anwendungsfall Refactoring entworfen worden und implementieren beispielsweise das Starten oder Abbrechen eines Refactorings auf Knopfdruck.

Auf Grundlage der vorgestellten Klassen wurde die Architektur der Faktor-IPS Refactoring-Integration entworfen. Der grobe Ablauf eines Refactorings ist in Abbildung 27 dargestellt.

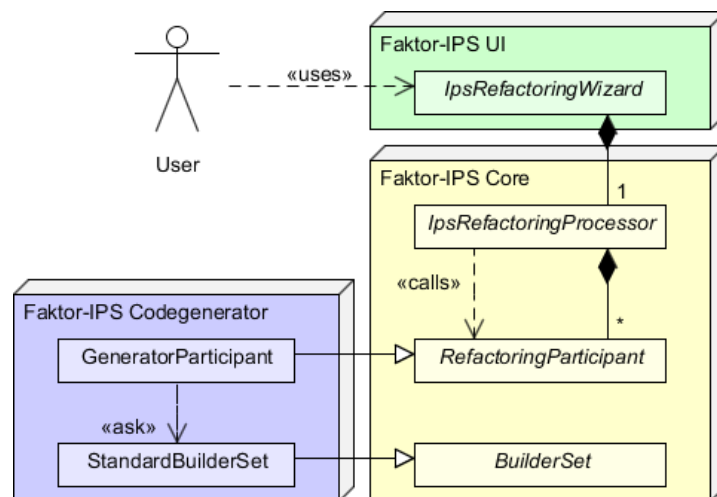


Abbildung 26: Architektur der Faktor-IPS Refactoring-Integration

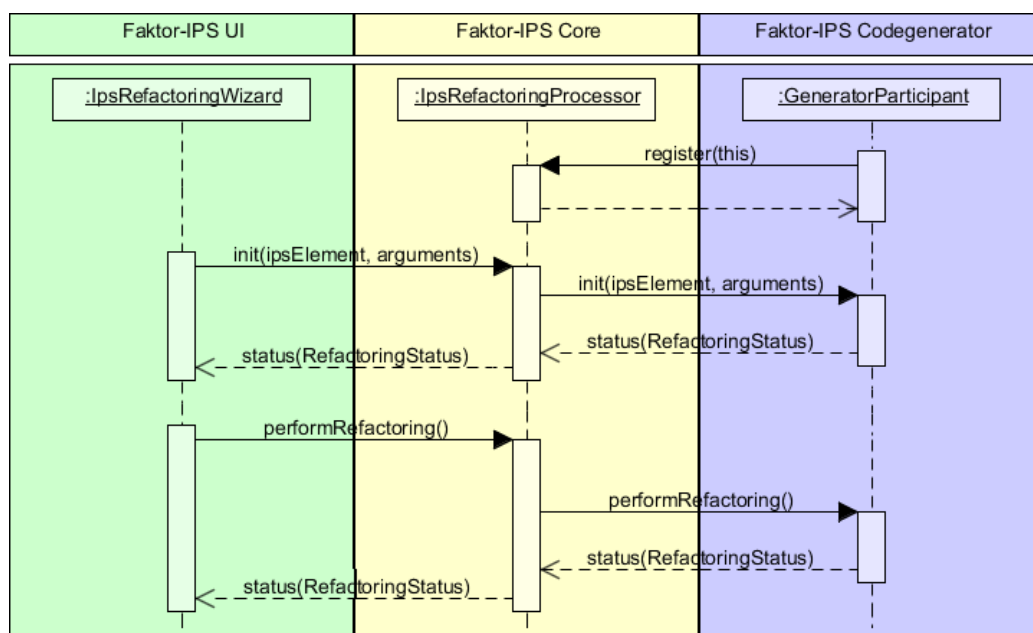


Abbildung 27: Grober Ablauf eines Faktor-IPS Refactorings

## 4 Umsetzung und Test

In diesem Abschnitt wird beschrieben, wie die Implementierung der automatisierten Refactorings *Rename Attribute*, *Rename Type* und *Move Type* in Faktor-IPS erfolgte.

Bei der Programmierung wurde schrittweise vorgegangen. Der erste Schritt bestand darin, den Faktor-IPS Codegenerator um die in Kapitel 3.2.2 beschriebene Auskunft über generierte Codeelemente zu erweitern. Anschließend wurden die Refactorings in der genannten Reihenfolge einzeln implementiert. So konnte die Entwicklungsarbeit in überschaubare Teilaufgaben mit jeweils lauffähigem Ergebnis aufgeteilt werden.

Dem Lesefluss halber werden in diesem Kapitel die verschiedenen Refactorings als einzelnes Arbeitspaket betrachtet. Eine Trennung erfolgt stattdessen anhand der in den Modulen Faktor-IPS Core, Faktor-IPS Codegenerator und Faktor-IPS UI anfallenden Arbeitsschritte sowie dem Test der entwickelten Funktionalität.

### 4.1 Codegenerator: Auskunft über generierte Codeelemente

Zur Beschreibung generierter Codeelemente in Faktor-IPS konnte das vom Eclipse Plug-in JDT zur Verfügung gestellte API verwendet werden. Dieses erlaubt die Erzeugung von Instanzen des Typs *IJavaElement* (siehe dazu auch Kap. 3.2.2).

#### 4.1.1 Identifikation der erzeugten Codeelemente

Der aus einem Modellelement generierte Java Quellcode ist in Faktor-IPS nicht dokumentiert. Deshalb mussten zunächst die aus den Modellelementen *PolicyCmptType*, *ProductCmptType*, *PolicyCmptTypeAttribute* und *ProductCmptTypeAttribute* (siehe Abb. 11 auf S. 19) generierten Java Codeelemente identifiziert werden, um diese anschließend mit Hilfe der Eclipse JDT Programmierschnittstelle beschreiben zu können.

Das Identifizieren der erzeugten Codefragmente geschah durch Untersuchen des für die Generierung zuständigen Programmcodes und des generierten Quelltextes. Die Ergebnisse wurden in tabellarischer Form festgehalten, da der für ein Modellelement erstellte Code von dessen Konfiguration abhängig ist.

### 4.1.2 Instanziierung der Java Elemente

In Abbildung 28 ist zum besseren Verständnis dieses Abschnitts der Aufbau des Codegenerators nochmals dargestellt. Die ergänzten Methoden sind im Klassendiagramm ebenfalls aufgeführt. Die abgebildeten Klassen sind nur ein Ausschnitt des im Rahmen der Arbeit modifizierten Programmcodes. Das gesamte Diagramm kann aus Platzgründen hier nicht gezeigt werden.

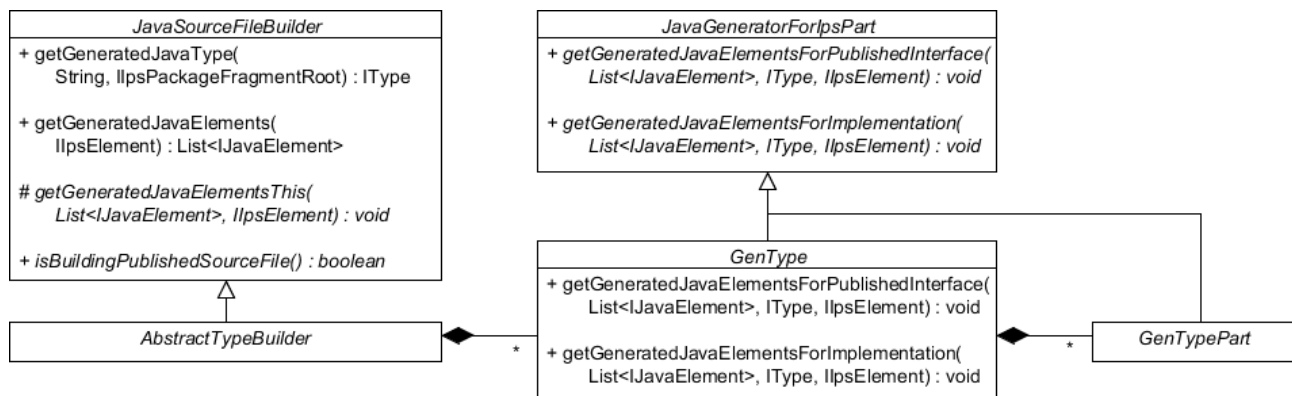


Abbildung 28: Aufbau des Faktor-IPS Codegenerators mit ergänzten Methoden

Der erste Schritt zur Instanziierung der Java Elemente bestand darin, den von einem *JavaSourceFileBuilder* erzeugten Java Typ zu ermitteln (Typ ist in Java ein Oberbegriff für Klasse oder Interface). Diese Aufgabe übernimmt die Methode *getGeneratedJavaType(String, ITypePackageFragmentRoot)*. Die Funktion muss dazu zuerst die vom *JavaSourceFileBuilder* erzeugte Java Quelldatei (*Compilation Unit*) bestimmen, von der wiederum der beinhaltete Typ abgefragt werden kann.

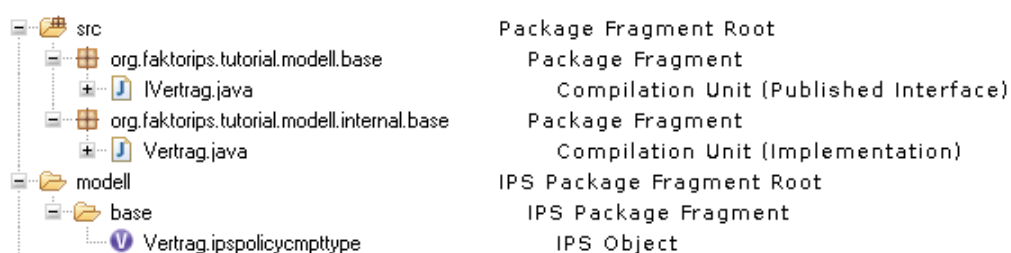


Abbildung 29: Aus einem Faktor-IPS Modell generierte Java Pakete und Dateien

Aufgrund technischer Details, auf welche hier nicht näher eingegangen werden soll, können Informationen über generierte Dateien in Faktor-IPS nur zum Zeitpunkt des Build-Prozesses abgefragt werden. Daher muss die Funktion die Zieldatei neu ermitteln.

Die Methode ist hier auf das Wesentliche reduziert abgebildet und soll die Erzeugung einer Instanz eines Java Typs veranschaulichen. Auf Basis des zurückgegebenen Java Typs können zugehörige Attribute und Methoden definiert werden.



**Listing 12:** Instanziierung von Java Typen in der Klasse *JavaSourceFileBuilder*

```
1      public final IType getGeneratedJavaType(String qualifiedNameIpsObject,
2          I IpsPackageFragmentRoot ipsPackageFragmentRoot) {
3          // ...
4
5          IFolder targetSourceFolder;
6          // Ziel-Quellordner aus gegebenem I IpsPackageFragmentRoot ermitteln
7          // ...
8
9          IJavaProject javaProject = getIpsProject().getJavaProject();
10         IPackageFragmentRoot javaPackageFragmentRoot =
11             javaProject.getPackageFragmentRoot(targetSourceFolder);
12
13         String javaFileName, javaPackageName;
14         // Namen der erzeugten Java Datei und des erzeugten Java Packages
15         // aus gegebenem qualifizierten Namen des Modellelements ermitteln
16         // ...
17
18         IPackageFragment javaPackageFragment =
19             javaPackageFragmentRoot.getPackageFragment(javaPackageName);
20         ICompilationUnit javaCompilationUnit =
21             javaPackageFragment.getCompilationUnit(javaFileName);
22
23         return compilationUnit.getType(typeName);
24     }
```

In der Implementierung der Methode *getGeneratedJavaElements(I IpsElement)* werden Anfragen an konkrete Subklassen delegiert. Dabei wird das *Collecting Parameter* Entwurfsmuster angewandt, bei dem eine Liste als Parameter weitergereicht wird. Jede aufgerufene Funktion kann der Liste Elemente hinzufügen.

**Listing 13:** Die Klasse *JavaSourceFileBuilder* delegiert Anfragen an Subklassen

```
1      public List<IJavaElement> getGeneratedJavaElements(I IpsElement ipsElement) {
2          List<IJavaElement> javaElements = new ArrayList<IJavaElement>();
3          getGeneratedJavaElementsThis(javaElements, ipsElement);
4          return javaElements;
5      }
```

In Abbildung 29 ist zu erkennen, dass aus einer Faktor-IPS Klasse zwei Java Quelldateien generiert werden: eine *öffentliche Schnittstelle (Published Interface)* und eine *interne Implementierung (Implementation)*. Für jedes Artefakt ist jeweils ein *JavaSourceFileBuilder* zuständig, beide Erbauer

verwenden aber die selben Generatoren (der Zusammenhang zwischen Erbauern und Generatoren wurde in Kap. 3.2.2 erläutert). Daher verfügen Generatoren über zwei separate Methoden zur Abfrage generierter Codeelemente, die je nach Anwendungsfall von konkreten Erbauern aufgerufen werden können:

- *getGeneratedJavaElementsForPublishedInterface(List<IJavaElement>, IType, IIpsElement)*
- *getGeneratedJavaElementsForImplementation(List<IJavaElement>, IType, IIpsElement)*

Die Funktionen erwarten neben der Liste generierter Codeelemente (Collecting Parameter) und dem Faktor-IPS Modellelement zusätzlich den vom Erbauer erzeugten Java Typ. Die Delegation an den Generator *GenType* könnte theoretisch bereits in der Klasse *AbstractTypeBuilder* (siehe Abb. 28) durch den nachfolgend abgebildeten Code installiert werden.

**Listing 14:** Die Klasse *AbstractTypeBuilder* leitet Anfragen an die Klasse *GenType* weiter

```
1  @Override
2  protected void getGeneratedJavaElementsThis(
3      List<IJavaElement>, IIpsElement ipsElement) {
4
5      IType type = null;
6      if (ipsElement instanceof IType) {
7          type = (IType)ipsElement;
8          // Hier wird eigentlich ein Interface ITypePart benötigt
9      } else if (ipsElement instanceof IAttribute) {
10         type = ((IAttribute)ipsElement).getType();
11     } else if (ipsElement instanceof IMethod) {
12         type = ((IMethod)ipsElement).getType();
13     } else {
14         return;
15     }
16
17     GenType genType = getGenType(type);
18     IIpsPackageFragmentRoot root = type.getIpsPackageFragment().getRoot();
19     org.eclipse.jdt.core.IType javaType =
20         getGeneratedJavaType(type.getQualifiedName(), root);
21     if (isBuildingPublishedSourceFile()) {
22         genType.getGeneratedJavaElementsForPublishedInterface(javaElements,
23             javaType, ipsElement);
24     } else {
25         genType.getGeneratedJavaElementsForImplementation(
26             javaElements, javaType, ipsElement);
27     }
28 }
```

Die Klassenhierarchie des Codegenerators erlaubt das zum gegenwärtigen Zeitpunkt aber nicht. Deswegen wurde die Methode in Subklassen dupliziert eingefügt. Das Problem wurde im Programm als *TODO* vermerkt.

In der Klasse *GenType* wird die Anfrage abermals weitergeleitet, nämlich an die zugehörigen Generatoren vom Typ *GenTypePart*. Das wird so lange fortgesetzt, bis jeder relevante Generator die Chance hatte, der weitergereichten Liste Java Elemente hinzuzufügen. Alle im vorherigen Arbeitsschritt identifizierten Codeelemente wurden an den entsprechenden Programmstellen mittels des JDT API beschrieben.

Die Komplexität bleibt Nutzern der Codegenerator-Schnittstelle verborgen. Aufrufer interagieren ausschließlich mit der Klasse *StandardBuilderSet*. Deren Schnittstelle veröffentlicht lediglich die Methode *getGeneratedJavaElements(IpsElement)*.

**Listing 15:** Öffentliche Komponenten-Schnittstelle zur Abfrage generierter Java Elemente

```
1  /**
2   * Returns a list containing all Java elements this builder set
3   * generates for the given IPS element.
4   */
5  public List<IJavaElement> getGeneratedJavaElements(IpsElement ipsElement) {
6      ArgumentCheck.notNull(ipsElement);
7      List<IJavaElement> javaElements = new ArrayList<IJavaElement>();
8      for (IIpsArtefactBuilder builder : getArtefactBuilders()) {
9          if (!(builder instanceof JavaSourceFileBuilder)) {
10             continue;
11          }
12          JavaSourceFileBuilder javaBuilder = (JavaSourceFileBuilder)builder;
13          javaElements.addAll(
14              javaBuilder.getGeneratedJavaElements(ipsElement));
15      }
16      return javaElements;
17  }
```

## 4.2 Core: Kern der Refactoring-Unterstützung

Nachdem der Faktor-IPS Codegenerator um die Beschreibung generierter Codeelemente erweitert wurde, konnte mit der eigentlichen Entwicklung der Refactoring-Integration begonnen werden. Das Ziel des Refactoring-Kerns ist es, die notwendigen Änderungen am Modellelement vorzunehmen und dabei alle modellseitigen Referenzen zu aktualisieren.

### 4.2.1 Übersicht

Der Refactoring-Kern besteht aus den in Abbildung 30 dargestellten Klassen und Schnittstellen.

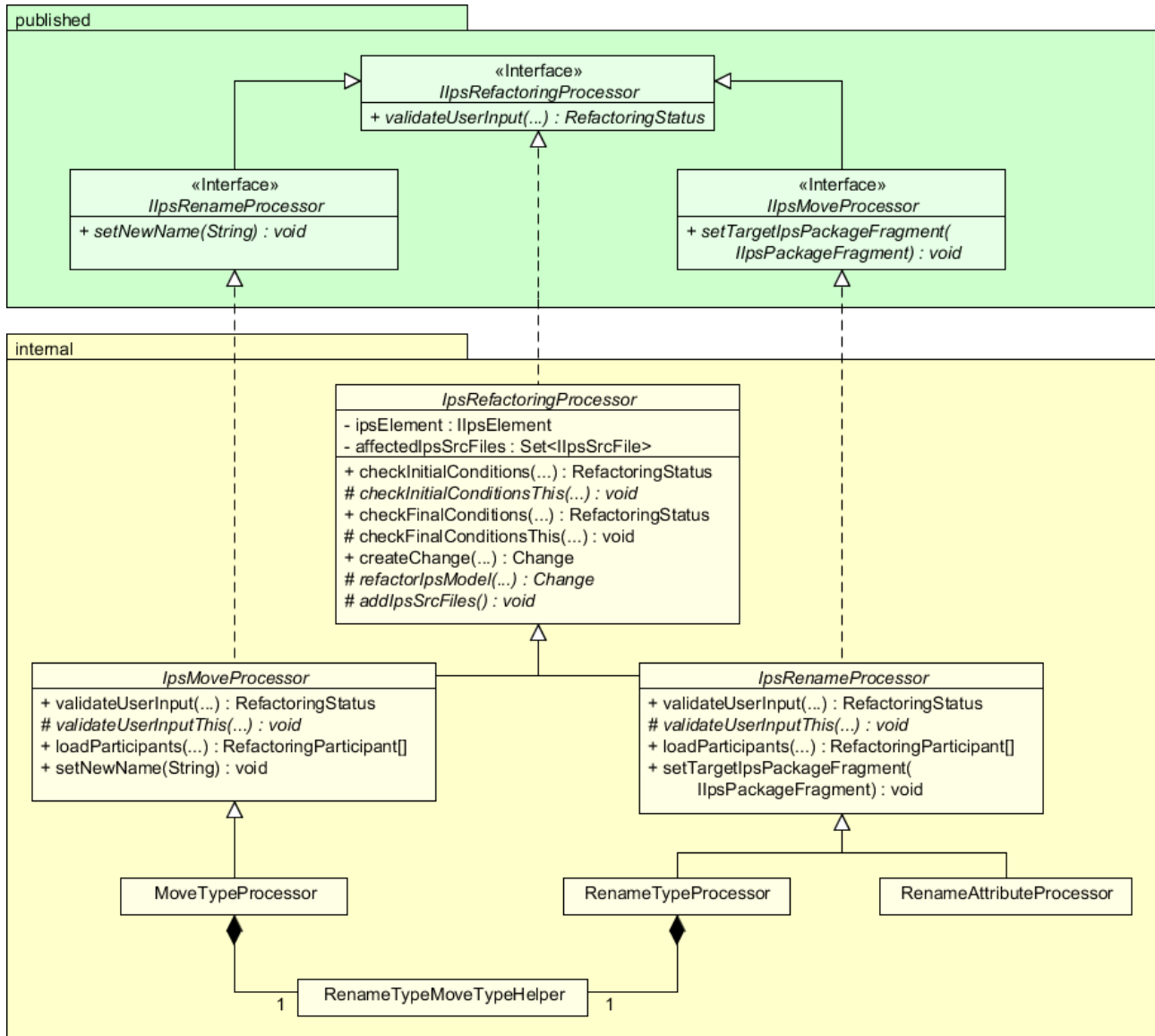


Abbildung 30: Kern der Faktor-IPS Refactoring-Integration

Die öffentliche Schnittstelle (Published Interface) des Refactoring-Kerns besteht aus den drei Java Interfaces *IpsRefactoringProcessor*, *IpsRenameProcessor* und *IpsMoveProcessor*. Andere Plug-ins – wie zum Beispiel das Modul Faktor-IPS UI – kommunizieren ausschließlich über das Published Interface mit dem Refactoring-Kern.

Die interne Implementierung kann somit wesentlich einfacher verändert werden, da sie vor externen Zugriffen geschützt ist. Die Bedeutung der einzelnen Klassen wird im Folgenden kurz zusammengefasst.

- **IpsRefactoringProcessor**

Die Klasse *IpsRefactoringProcessor* ist die Basisklasse für alle Faktor-IPS Refactoring-Prozessoren. Sie überprüft bereits einige grundlegende Vorbedingungen und stellt Subklassen nützliche Hilfsmethoden zur Verfügung.

- **IpsMoveProcessor**

Von *IpsMoveProcessor* sind alle Refactoring-Prozessoren abgeleitet, die Elemente verschieben. Die Klasse führt eine grundlegende Validierung der Benutzereingabe durch und lädt alle Refactoring-Teilnehmer.

- **IpsRenameProcessor**

Alle Refactoring-Prozessoren, die Elemente umbenennen, sind von dieser Basisklasse abgeleitet. Wie bei der Klasse *IpsMoveProcessor* erfolgt hier eine Validierung der Benutzereingabe und das Laden von Refactoring-Teilnehmern.

- **MoveTypeProcessor**

Ein *MoveTypeProcessor* ist für das Verschieben einer Faktor-IPS Klasse verantwortlich.

- **RenameTypeProcessor**

Ein *RenameTypeProcessor* hat das Umbenennen einer Faktor-IPS Klasse zur Aufgabe.

- **RenameAttributeProcessor**

Das Umbenennen eines Faktor-IPS Attributs wird von einem *RenameAttributeProcessor* übernommen.

- **RenameTypeMoveTypeHelper**

Die Faktor-IPS Refactorings *Rename Type* und *Move Type* unterscheiden sich in ihrer Implementierung nur minimal. Die gemeinsame Funktionalität wurde daher in die Helfer-Klasse *RenameTypeMoveTypeHelper* ausgelagert, um duplizierten Code zu vermeiden.

#### 4.2.2 Überprüfung von Vorbedingungen

Es muss sichergestellt werden, dass bestimmte Vorbedingungen (*Conditions*) erfüllt sind, bevor ein Refactoring gestartet wird. Anderenfalls ist es wahrscheinlich, dass während der Ausführung Fehler auftreten oder das Refactoring zu einem inkonsistenten Zustand führt. Das LTK Framework beschreibt eine zweistufige Überprüfung von Vorbedingungen:

- **Check Initial Conditions**

Noch vor dem Öffnen eines Dialogs oder Wizards werden sogenannte *Initial Conditions* überprüft. Dieser Check muss sehr schnell vonstatten gehen, da während der Ausführung die Benutzeroberfläche blockiert wird. Mit diesem ersten Schritt soll sichergestellt werden, dass das Refactoring prinzipiell durchführbar ist. Erst dann kann das Refactoring vom Benutzer konfiguriert werden.

- **Check Final Conditions**

Im zweiten Schritt werden die *Final Conditions* verifiziert. Wie der Name schon andeutet, wird dieser Vorgang unmittelbar vor der tatsächlichen Ausführung des Refactorings gestartet. Zu diesem Zeitpunkt ist das Refactoring vollständig konfiguriert und startbereit. Das Überprüfen der *Final Conditions* darf im Gegensatz zum Schritt *Check Initial Conditions* lange dauern. Dem Benutzer wird durch einen Fortschrittsbalken angezeigt, dass das Programm noch arbeitet.

In Faktor-IPS ist die Überprüfung der *Initial Conditions* wenig spektakulär. Es wird sichergestellt, dass das zu refaktorisierende Modellelement physikalisch existiert und in einem fehlerfreien Zustand ist (Validierung).

Wesentlich aufwändiger ist der Schritt *Check Final Conditions*. Die Aufgabe wird zu einem großen Teil von der Klasse *IpsRefactoringProcessor* übernommen. In der Implementierung der entsprechenden Methode wird wie nachfolgend beschrieben vorgegangen.

- 1) Zunächst wird die von Subklassen zu realisierende Validierung der Benutzereingaben angestoßen. Bei diesem Prozess wird überprüft, ob das Modellelement nach Anwendung der Benutzerdaten immer noch in einem fehlerfreien Zustand ist. Das Ergebnis der Validierung wird in den Status der Operation aufgenommen.
- 2) Anschließend müssen dem Refactoring alle erreichten Faktor-IPS Quelldateien bekannt gemacht werden. Dieser Schritt ist ebenfalls von Subklassen zu realisieren.
- 3) Es muss sichergestellt werden, dass alle erreichten Faktor-IPS Quelldateien mit dem Dateisystem synchronisiert sind. Die Ausführung des Refactorings würde sonst zu inkonsistenten Daten führen. Für jede nicht synchronisierte Quelldatei wird ein *Fatal Error* registriert. Die Fehlermeldung wird mit Hilfe des Eclipse *Native Language Support* lokalisiert.
- 4) Falls bis zu diesem Zeitpunkt alle Vorbedingungen erfüllt sind, wird Subklassen die Möglichkeit gegeben, weitere Checks durchzuführen.

**Listing 16:** Überprüfung von Final Conditions in *IpsRefactoringProcessor*

```
1  @Override
2  public final RefactoringStatus checkFinalConditions(
3      IProgressMonitor pm, CheckConditionsContext context)
4      throws CoreException, OperationCanceledException {
5
6      RefactoringStatus status = new RefactoringStatus();
7
8      // 1
9      RefactoringStatus validationStatus = validateUserInput(pm);
10     status.merge(validationStatus);
11
12     // 2
13     addIpsSrcFiles();
14
15     // 3
16     for (IIpsSrcFile ipsSrcFile : ipsSrcFiles) {
17         IResource resource = ipsSrcFile.getCorrespondingResoucre();
18         if (!(resource.isSynchronized(IResource.DEPTH_ZERO))) {
19             String errorMsg = NLS.bind(
20                 Messages.ipsSrcFileOutOfSync, resource.getFullPath());
21             status.addFatalError(errorMsg);
22         }
23     }
24
25     // 4
26     if (status.isOk()) {
27         checkFinalConditionsThis(status, pm, context);
28     }
29
30     return status;
31 }
```

### 4.2.3 Änderung des Modells

Nachdem die Vorbedingungen überprüft und das tatsächliche Refactoring gestartet wurde, muss das Faktor-IPS Modell entsprechend den Vorgaben des Refactorings verändert werden. Das Eclipse LTK Framework definiert in der Basisklasse *RefactoringProcessor* die in Listing 17 abgedruckte Methodensignatur zur Beschreibung von Änderungen.

**Listing 17:** Methodensignatur zur Beschreibung von Änderungen

```
1  /**
2   * Creates a Change object describing the workspace modifications the processor
3   * contributes to the overall refactoring.
4   *
5   * @throws CoreException If an error occurred while creating the Change.
6   * @throws OperationCanceledException If the condition checking got canceled.
7   */
8  public abstract Change createChange(IProgressMonitor pm)
9      throws CoreException, OperationCanceledException;
```

Idealerweise beschreibt eine Implementierung alle durchzuführenden Änderungen mit Hilfe eines *Change*-Objekts. Das LTK Framework stellt zu diesem Zweck verschiedene, von *Change* abgeleitete Klassen zur Verfügung, zum Beispiel:

- **ResourceChange**

Ein *ResourceChange* kann verwendet werden, um Dateien zu löschen, zu kopieren, zu verschieben oder umzubenennen. Dazu existieren die Subklassen *DeleteResourceChange*, *CopyResourceChange*, *MoveResourceChange* und *RenameResourceChange*.

- **TextEditBasedChange**

Mit einem *TextEditBasedChange* kann der Inhalt von Dateien bzw. Dokumenten durch Objekte vom Typ *TextEdit* modifiziert werden.

Durch die Verwendung eines *CompositeChange* können mehrere *Change*-Instanzen in einem einzelnen Objekt gebündelt werden. Ein *CompositeChange* kann wiederum andere *CompositeChange*-Instanzen enthalten. So ist es möglich, komplexe Änderungen zu beschreiben.

Dem Benutzer kann auf Grundlage der gesammelten *Change*-Objekte zunächst optional ein Preview angezeigt werden. Sollte der Anwender feststellen, dass die Änderungen nicht seinen Wünschen entsprechen, kann er das Refactoring abbrechen. In jedem Fall wird vor der Durchführung ein *Undo-Change* erzeugt, mit dem die Aktion später rückgängig gemacht werden kann. Sowohl das Preview, als auch der *Undo-Change* werden vom LTK Framework realisiert.

Die Faktor-IPS Refactoring-Integration nutzt diesen Mechanismus jedoch nicht. Dies liegt darin begründet, dass das Entwickeln von Faktor-IPS Modellen im Gegensatz zum Programmieren mit einer Programmiersprache nicht aus dem Bearbeiten von Textdateien besteht. Die Abbildung von Modellelementen auf Textsegmente innerhalb von Faktor-IPS Quelldateien ist nicht sinnvoll, da der



Text lediglich die Persistenz der tatsächlichen Elemente darstellt. Die vom LTK angebotenen *Change*-Objekte agieren aber auf Dateiebene und sind speziell für Textmanipulationen geeignet.

Die Änderung des Modells erfolgt daher anhand der aus den Quelldateien erzeugten Laufzeit-Objekte, die mit Hilfe von Setter-Methoden manipuliert werden. Dieser Prozess wird in der Implementierung der vom LTK Framework vorgeschriebenen Methode angestoßen. Anschließend wird ein *NullChange* zurückgegeben. Das ist ein *Change*-Objekt, welches keine Änderungen enthält.

**Listing 18:** Faktor-IPS verwendet den Change-Mechanismus des LTK Frameworks nicht

```
1      @Override
2      public final Change createChange(IProgressMonitor pm)
3          throws CoreException, OperationCanceledException {
4
5          refactorIpsModel(pm);
6          saveIpsSourceFiles(pm);
7          return new NullChange();
8      }
```

Die Methode *refactorIpsModel(IProgressMonitor)* ist in den Subklassen *RenameTypeProcessor*, *MoveTypeProcessor* und *RenameAttributeProcessor* realisiert. Erstere verlassen sich dabei auf die Implementierung der Helfer-Klasse *RenameTypeMoveTypeHelper*, die in Listing 19 abgedruckt ist.

**Listing 19:** Anpassung des Faktor-IPS Modells in der Klasse *RenameTypeMoveTypeHelper*

```
1      public void refactorIpsModel(IIpsPackageFragment targetIpsPackageFragment,
2          String newName, IProgressMonitor pm) throws CoreException {
3
4          copySourceFileToTargetLocation(targetIpsPackageFragment, newName, pm);
5          if (type instanceof IPolicyCmptType) {
6              updateConfiguringProductCmptTypeReference(
7                  targetIpsPackageFragment, newName);
8              updateTestCaseTypeParameterReferences(
9                  targetIpsPackageFragment, newName);
10         } else {
11             updateConfiguredPolicyCmptTypeReference(
12                 targetIpsPackageFragment, newName);
13             updateProductCmptReferences(targetIpsPackageFragment, newName);
14         }
15         updateMethodParameterReferences(targetIpsPackageFragment, newName);
16         updateAssociationReferences(targetIpsPackageFragment, newName);
17         updateSubtypeReferences(targetIpsPackageFragment, newName);
18         deleteOldIpsSourceFile(pm);
19     }
```

Faktor-IPS hält geladene Quelldateien in einem Cache, so dass bei weiteren Zugriffen die bereits geladenen Dateien nicht erneut gelesen werden müssen. Dieser Cache kommt durcheinander, sollte eine Datei umbenannt oder verschoben werden. Deshalb wird bei den Refactorings *Rename Type* und *Move Type* die Faktor-IPS Quelldatei kopiert und abschließend die ursprüngliche Quelldatei gelöscht.

Der Rest der Methode beschäftigt sich mit der Aktualisierung modellseitiger Referenzen. In Zukunft sollte die Referenzaktualisierung auf Modellebene wie in Kapitel 3.2.1 beschrieben überarbeitet werden.

Den *Change*-Mechanismus zu umgehen hat den Nachteil, dass kein Preview für ein Refactoring angezeigt werden kann. Dieses Feature hat in Faktor-IPS aber nur niedrige Priorität. Die Umsetzung würde sich sowieso aufwändiger gestalten, denn von textuellen Darstellungen wie XML soll in Faktor-IPS abstrahiert werden (Faktor-IPS Quelldateien werden im XML-Format abgespeichert). Es ist daher nicht sinnvoll, dem Benutzer eine Liste von Änderungen zu präsentieren, die sich auf XML-Dateien zur Persistenz von Modellelementen beziehen.

Ein wirklicher Nachteil ist jedoch, dass Refactorings nicht rückgängig gemacht werden können. Dieses Manko wurde für eine erste Version der Refactoring-Integration in Kauf genommen. Sowohl die Entwicklung einer *Undo*-Möglichkeit, als auch die eines aussagekräftigen Preview-Features sind Themen, die an diese Arbeit anknüpfen können

#### 4.2.4 Laden von Refactoring-Teilnehmern

Zu den Aufgaben des Refactoring-Kerns gehört neben dem Überprüfen von Vorbedingungen und dem Anpassen des Faktor-IPS Modells auch das Laden von Refactoring-Teilnehmern. Diese registrieren sich über einen Extension Point beim LTK Framework. Das Framework bietet mit dem *ParticipantManager* eine Fassade an, mit der das Laden der registrierten Extensions wesentlich vereinfacht wird. Somit genügt ein einzelner Methodenaufruf, bei dem einige Argumente übergeben werden.

**Listing 20:** Laden von Move Participants in der Klasse IpsMoveProcessor

```

1  @Override
2  public RefactoringParticipant[] loadParticipants(RefactoringStatus status,
3      SharableParticipants sharedParticipants) throws CoreException {
4
5      RefactoringProcessor processor = this;
6      Object elementToMove = getIpsElement();
7      MoveArguments arguments = new MoveArguments(
8          targetIpsPackageFragment, true);
9      String[] affectedNatures = new String[] { IIpsProject.NATURE_ID };
10     return ParticipantManager.loadMoveParticipants(status, processor,
11         elementToMove, arguments, affectedNatures, sharedParticipants);
12 }

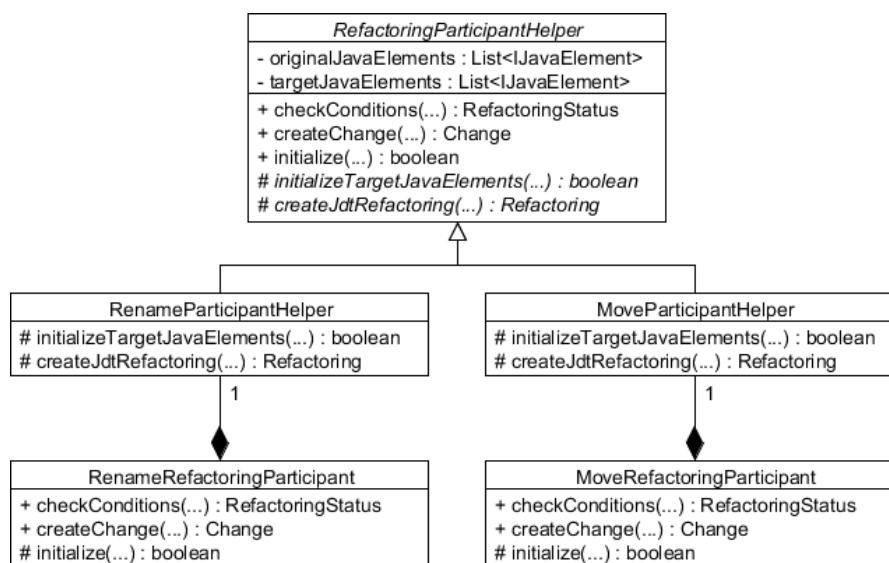
```

### 4.3 Codegenerator: Aktualisierung von nicht-generiertem Quellcode

Der Codegenerator beteiligt sich am Refactoring eines Faktor-IPS Modellelements. Das Ziel der vom Codegenerator registrierten Refactoring-Teilnehmer ist sicherzustellen, dass nicht-generierte Codeelemente nach dem Refactoring automatisch generierten Code korrekt referenzieren. Dazu werden mit Hilfe des JDT Moduls konventionelle Java Refactorings auf die generierten Codeelemente angewandt.

#### 4.3.1 Übersicht

Die vom Codegenerator registrierten Refactoring-Teilnehmer und ihre unterstützenden Klassen sind in Abbildung 31 dargestellt.

**Abbildung 31:** Refactoring-Teilnehmer des Faktor-IPS Codegenerators

- **RefactoringParticipantHelper**

Die Klasse *RefactoringParticipantHelper* ist die abstrakte Basisklasse für die Helfer-Klassen der Refactoring-Teilnehmer. Der Großteil des Programmcodes befindet sich hier. Insbesondere das Überprüfen von Vorbedingungen wird bereits vollständig implementiert.

- **RenameParticipantHelper**

Diese Helfer-Klasse wird vom *RenameRefactoringParticipant* benötigt.

- **MoveParticipantHelper**

Die Helfer-Klasse *MoveParticipantHelper* wird vom *MoveRefactoringParticipant* benötigt.

- **RenameRefactoringParticipant**

Der *RenameRefactoringParticipant* registriert sich bei der Plattform, um beim Umbenennen eines Faktor-IPS Modellelements gestartet zu werden.

- **MoveRefactoringParticipant**

Analog zum *RenameRefactoringParticipant* registriert sich der *MoveRefactoringParticipant* bei der Plattform, um beim Verschieben eines Faktor-IPS Modellelements gestartet zu werden.

Der Entwurf ergab sich aus dem Umstand, dass Refactoring-Teilnehmer, die Elemente umbenennen, von der LTK Basisklasse *RenameParticipant* abgeleitet werden sollten. Analog sind Refactoring-Teilnehmer, welche Elemente verschieben, von der Klasse *MoveParticipant* abzuleiten. Der Aufbau einer eigenen Klassenhierarchie wird dadurch verhindert, denn es gibt in Java keine Mehrfachvererbung. Die Refactoring-Teilnehmer des Codegenerators unterscheiden sich jedoch in der Implementierung nur geringfügig. Daher wurden Helfer-Klassen eingeführt, die von einer gemeinsamen Basisklasse abgeleitet werden konnten.

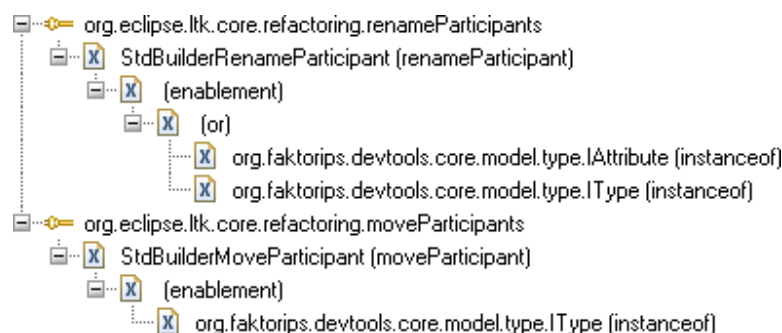


Abbildung 32: Extensions des Codegenerators zum Registrieren der Refactoring-Teilnehmer

### 4.3.2 Initialisierung

Bei der Initialisierung der Refactoring-Teilnehmer kommt die in Kapitel 4.1 entwickelte Auskunft über generierte Codeelemente zum Einsatz. Das zu refaktorisierende Modellelement wird dem Teilnehmer als Argument übergeben. Die generierten Java Elemente werden vom *StandardBuilderSet* abgefragt und in einer Liste zur späteren Verwendung zwischengespeichert (siehe Zeile 11 in Listing 21).

Falls ein Refactoring-Teilnehmer nicht initialisiert werden kann, nimmt er nicht am Refactoring teil.

**Listing 21:** Initialisierung von Refactoring-Teilnehmern in *RefactoringParticipantHelper*

```
1  public final boolean initialize(Object element) {
2      if (!(element instanceof IIpsElement)) {
3          return false;
4      }
5
6      IIpsElement ipsElement = (IIpsElement)element;
7      IIpsProject ipsProject = ipsElement.getIpsProject();
8      StandardBuilderSet builderSet =
9          (StandardBuilderSet)ipsProject.getIpsArtefactBuilderSet();
10
11     originalJavaElements = builderSet.getGeneratedJavaElements(ipsElement);
12     boolean success = initializeTargetJavaElements(ipsElement, builderSet);
13     return success;
14 }
```

Ein Problem beim Refaktorisieren der Java Elemente besteht darin, die neuen Namen für jedes einzelne Element zu ermitteln. Die Namen von Codeelementen entsprechen in den seltensten Fällen den Namen der Modellelemente, aus denen sie generiert wurden. Aus einem Attribut *plz* wird beispielsweise unter anderem eine Getter-Methode *getPlz()* erzeugt.

Die benötigte Information liegt abermals im Codegenerator verborgen. Die Auskunft über generierte Codeelemente kann geschickt zur Lösung dieses Problems eingesetzt werden. Dazu übergibt man dem Codegenerator das zu refaktorisierende Modellelement in dem Zustand, in welchem es nach dem Refactoring vorliegen wird. So erhält man die als *Target Java Elements* bezeichneten Codeelemente. Die für das Modellelement ursprünglich generierten Java Elemente werden als *Original Java Elements* bezeichnet.

Die in Listing 22 abgedruckte Methode der Klasse *RenameParticipantHelper* ermittelt auf diese Weise die *Target Java Elements* für ein Faktor-IPS Attribut.

**Listing 22:** Ermittlung von Target Java Elements für ein Faktor-IPS Attribut

```
1  private boolean initializeTargetJavaElementsForAttribute(  
2      IAttribute attribute, StandardBuilderSet builderSet) {  
3  
4      String oldName = attribute.getName();  
5      attribute.setName(getArguments().getNewName());  
6      setTargetJavaElements(builderSet.getGeneratedJavaElements(attribute));  
7      attribute.setName(oldName);  
8      return true;  
9  }
```

### 4.3.3 Überprüfung von Vorbedingungen

Refactoring-Teilnehmer erweitern die im Refactoring-Prozessor implementierte Überprüfung von Vorbedingungen. Ist eine von einem Refactoring-Teilnehmer geforderte Bedingung nicht erfüllt, so gilt das gesamte Refactoring als nicht durchführbar.

**Listing 23:** Überprüfung der Vorbedingungen aller durchzuführenden JDT Refactorings

```
1  public final RefactoringStatus checkConditions(IProgressMonitor pm,  
2      CheckConditionsContext context) throws OperationCanceledException {  
3  
4      RefactoringStatus status = new RefactoringStatus();  
5      for (int i = 0; i < originalJavaElements.size(); i++) {  
6          IJavaElement javaElement = originalJavaElements.get(i);  
7          // The refactoring may be executed without present Java code.  
8          if (!(javaElement.exists())) {  
9              continue;  
10         }  
11         try {  
12             Refactoring jdtRefactoring = createJdtRefactoring(  
13                 javaElement, targetJavaElements.get(i), status);  
14             if (jdtRefactoring != null) {  
15                 status.merge(jdtRefactoring.checkAllConditions(pm));  
16             }  
17         } catch (CoreException e) {  
18             RefactoringStatus errorStatus = new RefactoringStatus();  
19             errorStatus.addFatalError(e.getLocalizedMessage());  
20             return errorStatus;  
21         }  
22     }  
23     return status;  
24 }
```

Die vom Codegenerator registrierten Refactoring-Teilnehmer fordern, dass die Vorbedingungen aller durchzuführenden JDT Refactorings erfüllt sind. Das bedeutet zum Beispiel auch, dass alle relevanten Java Quelldateien mit dem Dateisystem synchronisiert sind. Zu beachten ist dabei, dass ein Faktor-IPS Refactoring auch gestartet werden kann, wenn überhaupt kein generierter Java Quellcode vorliegt.

#### 4.3.4 Starten der JDT Refactorings

Auch Refactoring-Teilnehmer verfügen über den in Kapitel 4.2.3 beschriebenen *Change*-Mechanismus. Alle Änderungen sollten mit *Change*-Objekten zunächst beschrieben werden, so dass sie anschließend vom Framework ausgeführt werden können.

Es ist möglich, die von den einzelnen JDT Refactorings erzeugten *Change*-Instanzen abzufragen. Probleme bereitet allerdings die Vereinigung dieser in einen *CompositeChange*. Nachdem die Änderungen eines JDT Refactorings durchgeführt wurden, sind die übrigen *Change*-Objekte nicht mehr gültig, denn diese basieren auf Instanzen der Klasse *TextEdit*. Die zur Änderung markierten Textstellen sind nach Anwendung eines anderen JDT Refactorings aber nicht mehr die Gleichen.

Aus diesem Grund wurde der Mechanismus auch bei den Refactoring-Teilnehmern umgangen. Die JDT Refactorings werden direkt zur Ausführung gebracht. Es galt abermals zu beachten, dass ein Faktor-IPS Refactoring auch ausgeführt werden kann, wenn kein Java Quellcode vorliegt.

**Listing 24:** Starten der JDT Refactorings

```
1      public final Change createChange(IProgressMonitor pm)
2          throws CoreException, OperationCanceledException {
3
4          for (int i = 0; i < originalJavaElements.size(); i++) {
5              IJavaElement javaElement = originalJavaElements.get(i);
6              if (!(javaElement.exists())) {
7                  continue;
8              }
9              Refactoring jdtRefactoring =
10                  createJdtRefactoring(originalJavaElements.get(i),
11                                      targetJavaElements.get(i), new RefactoringStatus());
12              if (jdtRefactoring != null) {
13                  performRefactoring(jdtRefactoring, pm);
14              }
15          }
16          return new NullChange();
17      }
```

Die Überprüfung auf Existenz eines Java Elements (Listing 24, Zeile 6-8) löst noch ein weiteres Problem: Das Java Refactoring *Rename Method* angewandt auf eine Methode innerhalb einer Java Klasse passt auch die Methodensignatur im *Published Interface* an und umgekehrt. Der Codegenerator beschreibt diese jedoch separat voneinander (Auskunft über generierte Codeelemente). Würde man das Refactoring auf beide Elemente anwenden, so käme es beim zweiten Element zu einem Fehler, da dieses nicht mehr existiert.

## 4.4 UI: Die Refactoring-Benutzeroberfläche

Zum Konfigurieren und Starten der automatisierten Refactorings stehen Nutzern der Faktor-IPS Refactoring-Integration komfortable Wizards zur Verfügung.

### 4.4.1 Übersicht

Die Faktor-IPS Refactoring-Benutzeroberfläche wird von den in Abbildung 33 dargestellten Klassen realisiert.



Abbildung 33: Klassen der Faktor-IPS Refactoring-Benutzeroberfläche

- **IpsRefactoringWizard**

Die Klasse *IpsRefactoringWizard* ist die Basisklasse für alle Faktor-IPS Refactoring-Wizards.

- **RenameRefactoringWizard**

Der *RenameRefactoringWizard* kommt zum Einsatz, wenn ein Modellelement umbenannt werden soll.

- **MoveRefactoringWizard**

Der *MoveRefactoringWizard* wird gestartet, wenn ein Modellelement verschoben werden soll.

- **IpsRefactoringUserInputPage**

Ein Refactoring-Wizard besteht aus beliebig vielen Wizardseiten, die der Konfiguration eines Refactorings durch den Benutzer dienen. Die Klasse *IpsRefactoringUserInputPage* ist die Basisklasse für alle Faktor-IPS Refactoring-Wizardseiten.



- **MoveUserInputPage**

Diese Wizardseite wird vom *MoveRefactoringWizard* verwendet, um dem Benutzer zu ermöglichen, einen Zielort für das zu verschiebende Modellelement auszuwählen.

- **RenameUserInputPage**

Die *RenameUserInputPage* bietet dem Benutzer ein Textfeld, in welches der neue Name für ein Modellelement eingegeben werden kann. Der *RenameRefactoringWizard* benötigt diese Wizardseite.

#### 4.4.2 Starten der Faktor-IPS Refactorings

Die Faktor-IPS Refactorings *Rename Type* und *Move Type* können über das Kontextmenü des Faktor-IPS *Model Explorer* gestartet werden. Der Zugriff ist auch über Tastenkombinationen möglich. Diese können über die Eclipse Einstellungen (*Window* → *Preferences* → *General* → *Keys*) individuell konfiguriert werden. Das Refactoring *Move Type* kann darüber hinaus innerhalb des Faktor-IPS *Model Explorer* durch *Drag and Drop* aktiviert werden.

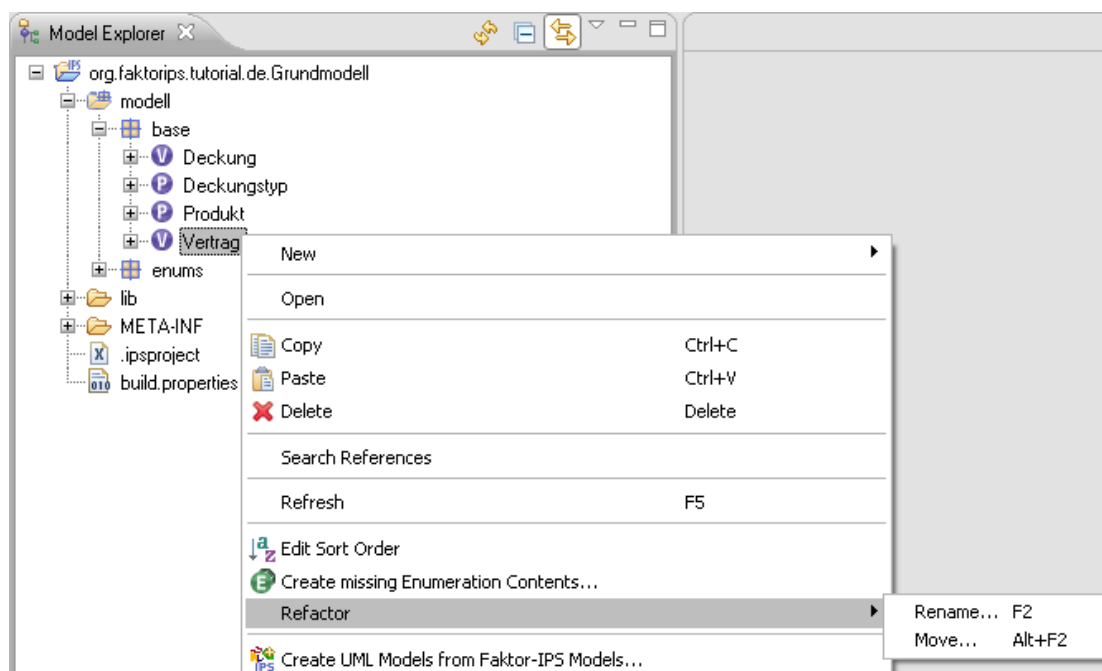


Abbildung 34: Kontextmenü im Faktor-IPS Model Explorer

Das Faktor-IPS Refactoring *Rename Attribute* wird über das Kontextmenü der entsprechenden Sektion im Vertragsteilklassen-Editor bzw. Produktbausteinklassen-Editor angestoßen. Auch dort ist die Anwendung der konfigurierten Tastenkombination möglich.

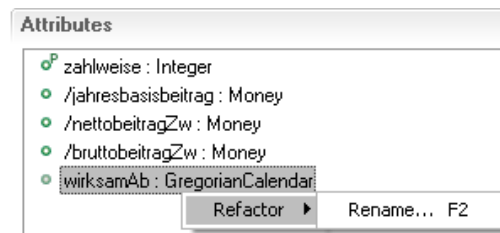


Abbildung 35: Kontextmenü im Vertragsteilklassen-Editor bzw. Produktbausteinklassen-Editor

#### 4.4.3 Faktor-IPS Refactoring-Wizards

Das LTK Framework sieht zwei Darstellungsmöglichkeiten für die Refactoring-Oberfläche vor: Wizard-basiert und Dialog-basiert. Die Refactoring-Oberfläche der Java IDE ist Dialog-basiert. Für Faktor-IPS wurde der Wizard-basierte Ansatz gewählt. So können Anwender auf den ersten Blick erkennen, dass ein Faktor-IPS Refactoring angestoßen wurde.

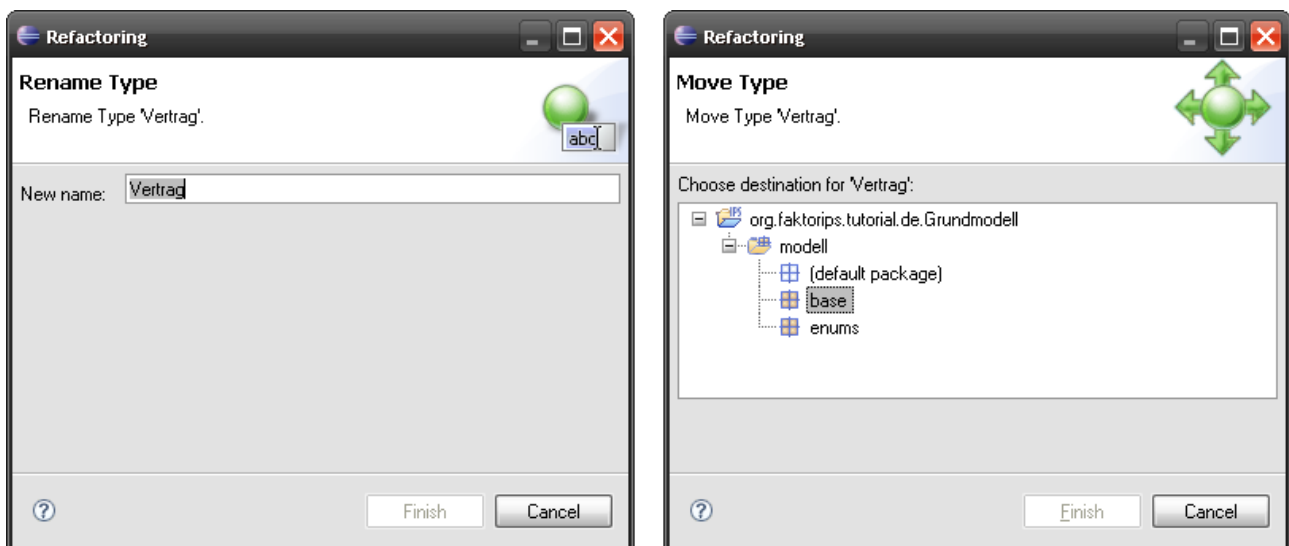


Abbildung 36: Faktor-IPS Refactoring-Wizards

Die Benutzereingabe wird vorab validiert. Der *Finish* Button kann nicht aktiviert werden, so lange keine gültige Eingabe getätigt wurde. Der Benutzer wird über fehlerhafte Angaben informiert.

Während der Ausführung eines Refactorings erscheint ein Fortschrittsbalken im unteren Teil des Wizards. Der *Cancel* Button kann währenddessen nicht betätigt werden, da ein Faktor-IPS Refactoring nicht abgebrochen werden darf. In diesem Fall müssten alle bereits ausgeführten Änderungen rückgängig gemacht werden. Das ist nicht möglich, weil diese nicht über den vom LTK Framework vorgesehenen *Change*-Mechanismus beschrieben wurden (siehe Kap. 4.2.3 und 4.3.4).

Sollte die Überprüfung der *Final Conditions* (siehe Kap. 4.2.2 und 4.3.3) fehlschlagen, so bricht der Wizard den Refactoring-Prozess ab und zeigt eine entsprechende Fehlerseite an.

## 4.5 Test

Die Programmierung der Faktor-IPS Refactoring-Integration erfolgte testgetrieben. Mit Hilfe von *JUnit* [JUnit] wurden insgesamt 81 automatisierte Testfälle erstellt.

<u><i>Funktion</i></u>	<u><i>Anzahl Testfälle</i></u>
Auskunft über generierte Codeelemente	47
Änderung des Modells	28
Aktualisierung von nicht-generiertem Quellcode	6

Bei der Entwicklung von Faktor-IPS wird das Prinzip der kontinuierlichen Integration angewandt, das heißt nach jeder Änderung im Quellcode wird das Programm auf einem Server neu gebaut. Dabei werden auch alle automatisierten Testfälle ausgeführt.

Solange die für die Refactoring-Integration entwickelten Testfälle erfolgreich bestanden werden, kann davon ausgegangen werden, dass die Funktion zumindest grundsätzlich korrekt funktioniert. Es ist aber nahezu unmöglich alle Eventualitäten mit automatisierten Tests abzudecken. Das ist auch nicht erforderlich, denn bevor eine neue Version von Faktor-IPS veröffentlicht wird, durchläuft diese immer zuerst mehrere *Release Candidates*. Kritische Fehler in der Refactoring-Integration sollten während dieser schrittweisen Veröffentlichung auffallen.

Berichtete Fehler werden mit einem weiteren Testfall zuerst nachgestellt, bevor sie ausgebessert werden. So wird man sich in Zukunft immer besser darauf verlassen können, dass die Refactoring-Integration korrekt funktioniert, sollten alle Testfälle bestanden werden.

Faktor-IPS verfügt noch nicht über automatisierte Benutzeroberflächen-Tests. Aus diesem Grund sollte die Refactoring-Integration vor der Veröffentlichung einer neuen Faktor-IPS Version nochmals manuell getestet werden.

## 5 Schlussbetrachtung

### 5.1 Fazit

Im Rahmen dieser Arbeit ist es gelungen, Refactoring im Kontext von MDSD in Faktor-IPS zu realisieren. Das Projekt durchlief die Phasen Problemanalyse, Konzeptentwicklung, Umsetzung und Test. Die Probleme, die sich bei der Entwicklung einer Refactoring-Unterstützung im Kontext von MDSD ergeben, wurden dabei zunächst möglichst unabhängig von einem konkreten MDSD-Werkzeug inklusive entsprechender Lösungsansätze dargestellt.

Mit den automatisierten Refactorings *Rename Type*, *Move Type* und *Rename Attribute* wurde der Grundstein für eine umfassende Refactoring-Unterstützung in Faktor-IPS gelegt. Die umgesetzten Refactorings werden häufig von Entwicklern benötigt und mussten von diesen bislang manuell durchgeführt werden. Durch die Refactoring-Integration können alle Einzelschritte automatisiert ausgeführt werden, inklusive der Aktualisierung des nicht-generierten Quellcodes. Die resultierende Produktivitäts- und Qualitätssteigerung wird als hoch eingeschätzt. Bemessen lässt sie sich aber erst im Laufe der Zeit.

Abschließend lässt sich sagen, dass die Integration von Refactoring im Kontext von MDSD aufwändig zu realisieren ist. Insbesondere die Referenzsuche und Referenzaktualisierung auf Modellebene erfordert die Umsetzung eines durchgängigen Konzepts quer durch die gesamte Implementierung des Metamodells. Der eingesetzte Codegenerator muss darüber hinaus in der Lage sein, zur Laufzeit über den für ein gegebenes Modellelement generierten Code Auskunft zu geben. Eine wichtige Voraussetzung ist außerdem, dass ein API zur programmatischen Anwendung konventioneller Refactorings auf den Ziel-Quellcode zur Verfügung steht.

Durch das LTK Framework bietet Eclipse eine durchdachte Architektur für die Entwicklung automatisierter Refactorings an. Dabei steht vor allem die lose Kopplung einzelner Module im Vordergrund. Das JDT Plug-in macht es möglich, konventionelle Refactorings programmatisch auf Java Quellcode anzuwenden. Die Eclipse Plattform stellt somit essentielle Bestandteile einer MDSD Refactoring-Integration bereit.

Im Laufe der Entwicklung ergaben sich keine gravierenden Probleme. Nur ein Preview der von einem Refactoring durchgeführten Änderungen und die Möglichkeit, ausgeführte Refactorings

rückgängig zu machen, konnten noch nicht umgesetzt werden. Das Entwickeln dieser Funktionen hätte den Zeitrahmen des Projekts gesprengt. Die Refactoring-Integration wird ab der Programmversion 2.5 in Faktor-IPS enthalten sein.

## 5.2 Ausblick

Der dringendste Punkt, den es bei der Faktor-IPS Refactoring-Integration umzugestalten gilt, ist die Referenzsuche und Referenzaktualisierung auf Modellebene. Die während dieser Arbeit notgedrungen eingesetzte Lösung erzeugt redundanten Code, ist überaus fehleranfällig und nur schwer wartbar. In Kapitel 3.2.1 wurde ein Lösungsansatz vorgeschlagen, der im Rahmen eines Meetings mit den Architekten der Faktor Zehn AG diskutiert wurde. Dabei wurde der Entwurf noch weiter für Faktor-IPS angepasst, so dass die notwendigen Eingriffe quer durch den Programmcode möglichst gering ausfallen. Bei der entwickelten Lösung bleiben die zur Referenzierung eingesetzten Zeichenketten bestehen. Zur Referenzsuche wird ein in Faktor-IPS bereits existierender *Dependency-Graph* eingesetzt. Die Aktualisierung der Referenzen erfolgt über *Reflection*. Im Zuge dieser Änderungen wird das Abstraktionsniveau der Refactorings angehoben. Aus dem Refactoring *Rename Type* wird das Refactoring *Rename Object Part Container*, *Move Type* wird zu *Move Object Part Container* und *Rename Attribute* wird zu *Rename Object Part* (siehe Abb. 11 auf S. 19). Nach dieser Umgestaltung können praktisch alle Modellelemente umbenannt, und alle Elemente des Typs *IpsObjectContainer* verschoben werden.

Darüber hinaus kann die Refactoring-Unterstützung um weitere Refactorings erweitert werden. Eine Umfrage ergab, dass die Entwickler in folgenden, potentiell umsetzbaren Refactorings großen Nutzen sehen:

- **Move Object Part**

Mit diesem Refactoring soll das Verschieben von Elementen des Typs *IpsObjectPart* zu einem anderen *IpsObjectPartContainer* ermöglicht werden.

- **Pull Up Object Part**

Das Refactoring *Pull Up Object Part* soll das Verschieben eines oder mehrerer *IpsObjectPart* in Richtung Superklasse ermöglichen.

- **Push Down Object Part**

Analog zu *Pull Up Object Part* soll das Refactoring *Push Down Object Part* das Verschieben eines oder mehrerer *IpsObjectPart* in Richtung Subklasse ermöglichen.

- **Extract Object Part To New Subclass**

Mit dem Refactoring *Extract Object Part To New Subclass* soll es möglich werden, eine beliebige Anzahl von Elementen des Typs *IpsObjectPart* innerhalb eines *IpsObjectPartContainer* in eine neu erstellte Subklasse zu extrahieren.

Früher oder später wird der Wunsch aufkommen, ausgeführte Refactorings wieder rückgängig machen zu können. Die entsprechende Umsetzung könnte sich als schwierig herausstellen und erfordert zunächst die Durchführung einer Machbarkeitsstudie. Gleiches gilt für die Realisierung einer Vorschau über die von einem Refactoring durchgeführten Änderungen. Hier muss zusätzlich der Nutzen gegen die Kosten abgewogen werden.

Die Faktor-IPS Refactoring-Integration ist schon jetzt ein großer Erfolg. Weitere Arbeit wird das Feature in Zukunft noch besser und umfangreicher machen.

## Literaturverzeichnis

- [Ar03] John Arthorne, *Project Builders and Natures*,  
<http://www.eclipse.org/articles/Article-Builders/builders.html>,  
aufgerufen am 09.01.2010
- [Bü07] Thomas Büchner, *Introspektive modellgetriebene Softwareentwicklung*, Dissertation,  
Technische Universität München, 2007
- [CVS] Free Software Foundation, *CVS - Open Source Version Control*,  
<http://www.nongnu.org/cvs/>, aufgerufen am 11.02.2010
- [Eb09] Ralf Ebert, *Anwendungsentwicklung mit Eclipse RCP*,  
<http://www.ralfebert.de/rcpbuch/>, aufgerufen am 15.01.2010
- [Eclipse] Eclipse Foundation, *Eclipse.org home*, <http://www.eclipse.org/>, aufgerufen am  
11.02.2010
- [EPL] Eclipse Foundation, *Eclipse Public License - Version 1.0*,  
<http://www.eclipse.org/legal/epl-v10.html>, aufgerufen am 11.02.2010
- [Equinox] Eclipse Foundation, *Equinox*, <http://www.eclipse.org/equinox/>, aufgerufen  
am 11.02.2010
- [FIPS] Faktor Zehn AG, *Faktor Zehn.org*, <http://www.faktorips.org/>, aufgerufen am  
11.02.2010
- [Fo99] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring -  
Improving the Design of Existing Code*, *The Addison-Wesley Object Technology  
Series*, Addison-Wesley Longman, Inc, Boston, 1999

- [FZ08] Faktor Zehn AG, *Modellgetriebene Softwareentwicklung mit Faktor-IPS*, Dokumentversion 652, München, 2008
- [JDT] Eclipse Foundation, *Eclipse Java development tools (JDT)*, <http://www.eclipse.org/jdt/>, aufgerufen am 11.02.2010
- [JUnit] Object Mentor, *Welcome to JUnit.org! | JUnit.org*, <http://www.junit.org/>, aufgerufen am 11.02.2010
- [Ma09] Robert C. Martin, *Clean Code - Testen und Techniken für sauberen Code*, Deutsche Ausgabe, mitp-Verlag, Heidelberg, 2009
- [Op92] William F. Opdyke, *Refactoring Object-Oriented Frameworks*, Dissertation, University of Illinois at Urbana-Champaign, 1992
- [Or09] Jan Ortmann, Gunnar Tacke, *Faktor-IPS Tutorial - Teil 1: Modellierung und Produktkonfigurierung*, Dokumentversion 16, München, 2009
- [OSGi] OSGi Alliance, *OSGi Alliance | Main / Alliance*, <http://www.osgi.org/>, aufgerufen am 11.02.2010
- [Pi07] Georg Pietrek, Jens Trompeter, Juan Carlos Flores Beltran, Boris Holzer, Thorsten Kamann, Michael Kloss, Steffen A. Mork, Benedikt Niehues, Karsten Thoms, *Modellgetriebene Softwareentwicklung*, entwickler.press, Software & Support Verlag GmbH, Frankfurt, 2007
- [St07] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase, *Modellgetriebene Softwareentwicklung*, 2. aktualisierte und erweiterte Auflage, dpunkt.verlag, Heidelberg, 2007
- [SVN] Apache Software Foundation, *Apache Subversion*, <http://subversion.apache.org/>, aufgerufen am 11.02.2010



- [SWT] Eclipse Foundation, *SWT: The Standard Widget Toolkit*,  
<http://www.eclipse.org/swt/>, aufgerufen am 11.02.2010
- [UML] Object Management Group, *Object Management Group - UML*,  
<http://www.uml.org/>, aufgerufen am 11.02.2010
- [Vo09] Lars Vogel, *Eclipse Extension Points and Extensions - Tutorial*,  
<http://www.vogella.de/articles/EclipseExtensionPoint/>, aufgerufen am 21.01.2009
- [Wi10] *Refactoring - Wikipedia*, <http://de.wikipedia.org/wiki/Refactoring>,  
aufgerufen am 22.01.2010
- [XML] W3C, *Extensible Markup Language (XML)*, <http://www.w3.org/XML/>, aufgerufen  
am 11.02.2010