Bachelor-Thesis

an der
w3l in Kooperation mit der FH Dortmund

im Fachbereich Informatik
im Studiengang Web- und Medieninformatik

Integration von Xtext in einen bestehenden Softwareentwicklungsprozess

Autor: Christopher Klein
Matrikel-Nr.: 7078647

Erstprüfer: Dr.-Ing. Sandra Krüger
Zweitprüfer: Prof. Dr. Heide Balzert

Abgabe am: 22. April 2013

Inhaltsverzeichnis

1 Abstrakt

In Softwareprojekten wird der Großteil des Quellcodes von Hand geschrieben, obwohl sich viele der notwendigen Artefakte aus vorhandenen Informationen automatisiert generieren ließen. Die Generierung von Quellcode reduziert die Fehleranfälligkeit innerhalb der zu entwicklenden Anwendung und sorgt dafür, dass mehr Zeit in die Implementierung der Geschäftslogik investiert werden kann.

In dieser Bachelorarbeit wird untersucht, wie ein bestehender Softwareentwicklungsprozess durch Automatismen verbessert werden kann. Diese Erkenntnisse bilden die Basis für die Entwicklung einer domänenspezifischen Sprache mit Hilfe des Werkzeugs Xtext. Neben der Anpassung der Xtext-Umgebung wird die Grundlage für einen Plug-in-Mechanismus geschaffen. Dieser Mechanismus erlaubt das einfache Einbinden von Generatoren, die aus dem Domänenmodell Artefakte generieren können. Auf diesen Mechanismus setzen die beiden implementierten Generatoren auf: Aus dem gegebenen Domänenmodell erzeugen sie daraus einerseits Mockups und andererseits eine Function Point-Analyse.

Die Nutzung von Xtext hat den Softwareentwicklungsprozess entschieden verbessert: Die Planung von zukünftigen Projekten ist nun durch die gegebene Möglichkeit einer Function Point-Analyse einfacher geworden. Während der Phase des Requirement Engineerings wurden die generierten Mockups bereits in ersten Projekten zu Demonstrationszwecken gezeigt. Besonders der einfache Plug-in-Mechanismus und die implementierten Erweiterungen für Xtext haben für eine deutliche Verbesserung des Prozesses gesorgt. Auf dieser Grundlage wurden nach kurzer Zeit weitere Generatoren entwickelt, die Quellcode für verschiedene Programmiersprachen und Anwendungsgebiete erzeugen. Die Entwickler können sich nun mehr auf die Implementierung der Geschäftslogik konzentrieren.

Abstract

In software projects most of the source code is written by hand although most of the required artifacts could be automatically generated through existing information. The generation of source codes reduces the error-proneness inside the application to be developed and ensures that more time can be invested into the development of the business logic.

This dissertation examines how an existing software development process can be improved through use of suitable automatisms. This knowledge forms the base for the development of a domain specific language through use of the tool Xtext. Besides the adaption of the Xtext environment a basis for a plug-in mechnism is created. This mechanism provides the easy integration of generators. The generators use the domain model for generating artifacts. Both of the implemented generators use this mechanism: By using the given domain model they generate on the one hand a mockup and on the other hand a function point analysis.

The usage of Xtext definitely improved the software development process: the planning for future projects is now easier with help of the function point analysis. The generated mockups have been already used in the phase of requirement engineering. Particularly the easy-to-use plug-in mechanism and the implemented extensions for Xtext showed a clearly improvement of the process. On this basis more generators for generating source code for different programming languages and use cases have been developed in a short time. Developers can now more focus on the business logic.

2 Einleitung

"Write Code That Writes Codes"[**?**] lautet einer der Ratschläge aus *The Pragmatic Programmers*, einem der bekanntesten Bücher der Softwareentwicklung. Die Idee besteht darin, dass sich wiederholende Programmieraufgaben durch geeignete Methoden und Techniken automatisieren lassen. Auf dieser Idee basiert der Ansatz der modellgetriebenen Entwicklung: die automatisierte Generierung von Quellcode auf Basis eines definierten Modells. Gerade mit der stetig steigenden Anzahl der Technologien ist es wichtig, dass sich die Softwareentwickler auf die jeweiligen Domänenprobleme konzentrieren können und nicht durch wiederkehrende Standardanforderungen und -aufgaben während des Entwicklungsprozesses Zeit verlieren. Neben proprietären Werkzeugen zur Code-Generierung existieren auch freie Alternativen.

2.1 Aufgabenstellung

Das Ziel dieser Bachelorthesis soll es sein, dass ein bestehender Softwareentwicklungsprozess durch technische Lösungen verbessert werden soll. Der Schwerpunkt liegt dabei auf der automatischen Generierung von Code-Fragmenten mit Hilfe des Xtext-Frameworks. Die resultierenden Code-Fragmente können dabei unterschiedlicher Natur sein, z.B. Quellcode für C#, PHP oder Java, Dokumentation, automatische Analysen oder Dateien mit SQL-Anweisungen

Das im Rahmen dieser Bachelorarbeit entstehende Eclipse Plug-in soll dabei so entwickelt werden, dass neue Generatoren für Code-Fragmente einfach integriert werden können.

2.2 Motivation

Auslöser für diese Bachelor-Arbeit ist ein abgeschlossenes Software-Projekt gewesen, in dem bereits Xtext im kleinen Rahmen verwendet wurde. Die dabei erstellte domänenspezifische Sprache (*Domain Specific Language*, *DSL*) beschrieb die Domänen-Objekte mit ihren Attributen, die mit Hilfe von Xtext in C#-Quellcode umgewandelt wurden. Die DSL wurde zwar nur für grundlegende Transformationen benutzt, dennoch lag der dadurch entstandene Mehrwert auf der Hand. Es entstand eine einheitliche Basis für Dokumentation und Quellcode. Die Zeit, die für alltägliche Programmierarbeiten wie dem Implementieren von Entitäten oder der Datenzugriffsschicht anfiel, wurde auf ein Minimum verringert.

2.3 Vorgehen

Zuerst wird der Softwareentwicklungsprozess bei der NeosIT GmbH vorgestellt und mögliche Verbesserungen aufgezeigt. Im darauffolgenden Kapitel wird die Grundidee hinter der modellgetriebenen Softwareentwicklung gezeigt und verschiedene Lösungsansätze und Produkte kurz erläutert. Als Einführung in die Arbeit mit Xtext werden im Kapitel 4 die notwendigen Technologien und Ideen vorgestellt, die dieses Werkzeug mit sich bringt.
Die Umsetzung der Verbesserungsvorschläge findet durch die Implementierung statt. Das Kapitel 5 erläutert die während der Bachelorarbeit entstandenen Code-Generatoren. Schlussendlich wird ein Fazit gezogen und Ausblick auf zukünftige Erweiterungen gegeben.

Während dieser Ausarbeitung gelten folgende Konventionen:

- Pfadangaben und Dateinamen werden in kursiver Schrift dargestellt, z.B. *src/plugin.xml*
- Quellcode wird in Verbatim dargestellt, z.B. `addComponent(...)`
- Innerhalb des Quellcodes bedeuten von rechts unten nach links oben stehende Schrägstriche am Ende der Zeile, dass der folgende Zeilenumbruch nur aus Platzgründen eingeführt worden ist. Der Quellcode steht regulär mit der nächsten Zeile zusammen in einer Zeile.
- Bei Verweisen auf Methoden werden die Parameterdefintionen nicht mit angegeben und durch drei Punkte ... ersetzt
- Bei Verweisen auf Klassen- oder Interfacenamen wird nicht der vollqualifizierte Name angegeben, sondern nur der Instanzname. Der vollqualifizierte Name wird als Fußnote hinterlegt. Anstatt `de.ckl.rapid.RapidRuntimeModule` wird nur `RapidRuntimeModule` geschrieben.

3 Ausgangssituation

3.1 Kontext

Die Bachelorarbeit wird innerhalb des Unternehmens NeosIT GmbH in Wolfsburg geschrieben. Die NeosIT ist ein mittelständisches IT-Unternehmen, dass in den Bereichen Softwareentwicklung und Systemadministration tätig ist. Zu den Kunden zählen unter anderem die Volkswagen AG und die Volkswagen Financial Service AG.
Neben dem Customizing bestehender Open-Source-Software wird auch Software nach Bedarf des Kundens programmiert. Die meisten Projekte werden dabei in C#, Java oder PHP realisiert und stehen dem Kunden als Webapplikationen innerhalb des Intra- oder Internets zur Verfügung.
Besondere Erfahrung haben die Mitarbeiter der NeosIT in der Entwicklung von webbasierten Management-Reporting-Systemen. Es handelt sich dabei um Software zur Auswertung von Unternehmenskennzahlen. Die Auswertung findet dabei tabellarisch oder mit Hilfe von geeigneten Charts statt. Grundlage für diese Art von Software bilden in aller Regel bereits existierende Excel-Tabellen, die in eine Webanwendung transformiert werden sollen.

3.2 Aktueller Entwicklungsprozess

Im Folgenden soll ein Einblick in den aktuell vorhandenen Softwareentwicklungsprozess bei der NeosIT GmbH gegeben werden. Eine detaillierte Beschreibung würde den Rahmen dieser Arbeit sprengen.

3.2.1 Requirements Engineering

In der Phase des Requirements Engineering werden die Anforderungen des Kunden aufgenommen. Idealerweise existiert seitens des Kunden ein detailliertes Lastenheft, das alle Anforderungen enthält. Die Vergangenheit hat aber gezeigt, dass solche idealisierten Vorstellungen sich selten bewahrheiten. Aus diesem Grund wird das Lastenheft gemeinsam durch Auftraggeber und Auftragnehmer erstellt. Nachdem die groben Anforderungen an die zu erstellende Software in ersten Meetings geklärt worden ist, werden detailierte Funktionalitäten besprochen. Je nach Anforderung des Kunden werden hier bereits mögliche Konzepte zur grafischen Gestaltung der Benutzerführung vorgestellt. Die möglichen Konzepte werden dabei mit Software wie Adobe Photoshop, Gliffy oder Balsamiq umgesetzt.
Es hat sich dabei gezeigt, dass der frühe Einsatz von Mockups deutliche Vorteile für den weiteren Projektverlauf bietet:

- Zwischen Auftraggeber/Auftragnehmer entsteht eine emotionale Beziehung

zu der zu erstellenden Software. Es handelt sich nicht mehr um etwas Abstraktes, was erst in ferner Zukunft anfassbar ist. Beide Seiten werden angeregt über eine langfristige Vision nachzudenken.
Die Entwickler können dabei oft aus den Äußerungen des Kunden erkennen, welche zusätzlichen Wünsche auftreten könnten und die zu erstellende Softwarearchitektur darauf abstimmen. Lässt sich beispielsweise bereits im Vorfeld erkennen, dass neben einer Weboberfläche eventuell eine mobile Anwendung benötigt wird, werden die öffentlichen Schnittstellen bereits entsprechend aufbereitet angeboten.

- Der Kunde kann anhand von grafischen Elementen einfacher beschreiben, wie seine internen Prozesse ablaufen. Missverständnisse zwischen Auftragnehmer und Auftraggeber können so in einer frühen Phase geklärt werden, die später zu teuren Refactoring-Maßnahmen führen würden[1].

- Die Entwickler können anhand der Ergebnisse (Anforderungen, Wünsche, Prozessabläufe) eine bessere Abschätzung zu den zu leistenden Stunden abgeben.

Im Gegensatz zu den funktionalen Anforderungen werden die nicht-funktionalen Anforderungen anhand von Formblättern aufgenommen. Hier werden messbare Größen wie z.B. gefordertes Antwortverhalten dokumentiert. Weiterhin werden unter anderem folgende Themenkomplexe abgefragt:

- Gegenwärtige Situation der IT-Landschaft - wie sieht die Netzwerkinfrastruktur aus; welche IP-Bereiche existieren; kommt Nating zum Einsatz; existieren Proxy-Server; welche Mailserver existieren etc.

- Bereits in Planung befindliche Änderungen der IT-Landschaft

- Zwingend einzusetzende Software wie z.B. DBMS, Frameworks oder Server-Applikationen inklusive der Versionen.

Die Formblätter werden in aller Regel von den zuständigen Systemadministratoren ausgefüllt. Seitens des Auftragnehmers werden in einer virtuellen Umgebung die relevanten Systeme in einem kleinen Rahmen nachgebaut. Dabei wird vor allem auf den Einsatz von identischen Versionen geachtet: Unterschiedliche Datenbankmanagementsysteme haben von Version zu Version andere Features; die zu erstellenden Dokumentation für die Systemadministration unterscheidet sich von Betriebssystem-Version zu Betriebssystem-Version etc.
Alle Anforderungen und Mockups werden im Anschluss in das unternehmensinterne Confluence-Wiki übertragen und stehen allen Mitarbeitern sowie dem Auftraggeber zur Verfügung. Im Wiki wird ebenfalls die Endanwender- und Entwickler-Dokumentation des Projektes gepflegt.

3.2.2 Analyse und Konzeptionierung

Sobald die Anforderungen an die Software definiert worden sind, werden diese analysiert. Im ersten Schritt werden die Objekte innerhalb des Anforderungskatalogs identifiziert. Die Attribute der Objekte und die Beziehung zu anderen Objekten werden dokumentiert.
Die Ergebnisse der Analyse werden direkt in ein UML-Klassendiagramm überführt. Dies gilt später als Grundlage für das zu erstellende Datenbankmodell.

[1] siehe [?] bzw. [?]

Die einzusetzende Software ergibt sich einerseits aus den Anforderungen des Kunden, andererseits aus den bisher gemachten Erfahrungen der Entwickler. Die Software-Architektur basiert dann auf der eingesetzten Software. In den meisten Fällen findet aber eine strikte Trennung zwischen Datenzugriffsschicht[2], Serviceschicht und Frontend statt. Im Frontend kommt ebenfalls in den meisten Fällen das MVC-Pattern zum Einsatz.
Je nach Plattform werden weitere Frameworks eingegliedert. Tabelle 3.1 enthält einen Auszug der favorisierten Frameworks für C# und Java.

Komponente	C#	Java
Logging	NLog oder log4net	log4j
Dependency Injection	Unity oder Spring.NET	Spring oder Guice
MVC	ASP.NET MVC 2/3	Spring MVC
Scheduling	Quartz.NET	Quartz
Messaging Queue	ActiveMQ	ActiveMQ
AOP	Unity	AspectJ
Testing	Xunit	JUnit

Tabelle 3.1: Eingesetzte Frameworks

Sobald die Software-Architektur definiert und mit dem Auftraggeber abgestimmt ist, werden die Schnittstellen zwischen den einzelnen Subsystemen definiert.

3.2.3 Realisierung

Zuerst wird das unspezifizierte Datenbankmodell auf das konkret ausgewählte DBMS übertragen. Das bedeutet, dass die DDL-Statements so erstellt werden, dass sie innerhalb des DBMS funktionieren. Je nach Datenbankmanagementsystem können sich Datentypen oder andere Optionen unterscheiden. Um zukünftige Änderungen an dem Datenbankschema einfacher handhaben zu können, werden die SQL-Statements über Tools automatisiert ausgeführt[3].

Anhand der im Vorfeld definierten Schnittstellen werden die einzelnen Komponenten von den Entwicklern programmiert. Durch eine lose Koppelung und dem Einsatz von Dependency Injection-Frameworks lässt sich noch nicht implementierte Funktionalität mit Hilfe von Mocks abbilden. Somit können mehrere Entwickler an verschiedenen Komponenten gleichzeitig arbeiten. Die Quellcodeverwaltung erfolgt dabei über Git. Als Standard-Entwicklungsumgebung wird für Java- und PHP-Projekte Eclipse eingesetzt[4], bei C#-Projekten kommt hingegen Visual Studio zum Einsatz.

In aller Regel wird ein iteratives bzw. agiles Vorgehensmodell wie Kanban oder Scrum gewählt, so dass der Kunde schnell benutzbare Software erhält. Die Qualitätssicherung geschieht dabei über die Unittest-Frameworks der einzelnen Plattformen[5] oder mit zusätzlichen Tools[6]. Die Builds werden durch einen

[2]DAL bzw. DAO-Entwurfsmuster
[3]siehe dazu https://github.com/schakko/db-migrator bzw. https://github.com/prunkstar/db-migrator.net
[4]Mit den Umgebungen JDT bzw. PDT
[5]Xunit oder JUnit
[6]z.B. Selenium

Buildserver[7] automatisiert erstellt.

3.3 Ist-Aufnahme bestehender Automatismen

Der momentane Entwicklungsprozess wird bereits durch einige Automatismen unterstützt, die folgend kurz beschrieben werden sollen.

3.3.1 Generierung der Verzeichnisstruktur eines Projekts

Zu Beginn eines Projekts wird manuell ein Projektverzeichnis angelegt, das eine standardisierte Ordnerstruktur enthält. Dieses Verzeichnis wird in das zentrale Git[8]-Repository gepusht. Jeder Entwickler arbeitet mit einer verteilten Kopie des Repositories.
Die Erstellung der standardisierten Ordnerstruktur sowie das Generieren von initialen Dateien, wie z.B. README-Dateien, erfolgt momentan über ein eigenes PowerShell-Script. Jeder Entwickler findet sich somit in neuen Projekten sofort zurecht und Vorlagen von Buildscripten können ohne Pfadanpassungen integriert werden.

3.3.2 Kontinuierliche Integration

Unter dem Begriff *Kontinuierliche Integration* versteht man das automatische Erstellen der Build-Artefakte, sobald neuer Quellcode in eine Versionverwaltung eingecheckt worden ist.
Bei der NeosIT wird dabei TeamCity[9] eingesetzt. Diese Applikation überprüft in regelmäßigen Intervallen, ob in den Git-Repositories neuer Quellcode gepusht worden ist. Bei einer Änderung wird dieser auf einen sogenannten Buildagent ausgecheckt und das durch den Administrator definierte Buildscript gestartet. Die Buildscripte enthalten jeweils Anweisungen zum Ausführen von Unittests und zur Kompilierung. Die Projektmitglieder werden per E-Mail über Fehler innerhalb des Build- oder Testprozesses informiert.
Nach erfolgreichen Build- und Testphase werden die generierten Artefakte in passende ZIP- oder MSI-Pakete transformiert und stehen allen Entwicklern und den Kunden direkt zur Verfügung.

3.3.3 Kontinuierliches Deployment

Die durch den Buildserver entstandenen Artefakte werden zum Teil automatisiert auf den Test- oder Produktivsystemen ausgerollt, so dass der Benutzer sofort die Änderungen sehen kann. Unter anderem können sich in Entwicklung befindliche Kundenwebseiten automatisch ausgerollt werden.

[7] TeamCity, http://www.jetbrains.com/teamcity/
[8] Git ist eine verteilte Versionsverwaltung; siehe http://www.git-scm.com
[9] http://www.jetbrains.com/teamcity/

3.4 Ideen zur Verbesserung

Neben den bereits bestehenden Automatismen gibt es natürlich viele weitere Anwendungsfälle, die sich automatisieren und in den bestehenden Softwareentwicklungsprozess integrieren ließen.

3.4.1 Function Point-Analyse

Bei der Function Point-Analyse handelt es sich um ein algorithmisches Schätzverfahren um den benötigten Aufwand zur Umsetzung eines Softwareprojektes zu ermitteln. Die Methode ist unter anderem in der Norm ISO/IEC 20926 standardisiert. Für das Schätzverfahren werden unterschiedliche Elemente des Anforderungskatalogs gezählt und gewichtet. Die Regeln, nach denen vorgegangen wird, hat unter anderem die International Function Point User Group[10] festgelegt [**?**]. Für jedes Projekt wird eine Function Point-Analyse durchgeführt und die dabei entstandenen Ergebnisse in einer zentraler Datenbank festgehalten. Durch die Formel $\frac{Summe\ aller\ Function\ Points}{Summe\ aller\ aufgewendeten\ Projektstunden}$ wird abgebildet, wie lange die Umsetzung eines Function Points in Stunden dauert. Es soll hier nicht verschwiegen werden, dass die Analyse nur einen groben Hinweis auf die Dauer geben kann:

- Die aufgewendeten Projektstunden müssen bereinigt sein, so dass Stunden für Meetings und Abstimmungsgespräche nicht enthalten sind.
- Bestimmte Aufgaben lassen sich mit einer Programmiersprache oder einem eingesetzten Framework einfacher umsetzen, als mit einem anderen.
- Direkten Einfluss auf die aufgewendeten Stunden hat das Projektteam. Einerseits wird die Performance eines gleichbleibenden Teams mit den selben Technologien über die Zeit besser werden. Andererseits können neue Projektmitglieder die Gesamtperformance negativ oder positiv verändern. Idealerweise sollte das selbe Team mit den selben Technologien arbeiten.

Eine automatisierte Function Point-Analyse ließe sich recht schnell mit einer verfügbaren DSL implementieren.

3.4.2 Generierung eines Mockups

Während der Phase des Requirements Engineerings könnten alle funktionalen Anforderungen sofort in einer dafür vorgesehenen Beschreibungssprache aufgenommen werden. Die dabei definierten notwendigen Attribute, Objekte und Beziehungen könnten automatisch in einen Mockup transformiert werden. Mockups sind Attrappen, die zu Präsentationszwecken gefertigt wurden und wenig bis keine Funktionalität beinhaltet. Der generierte Mockup beinhaltet eine rudimentäre Oberfläche, die die Standardaufgaben von Software enthält, wie z.B. Einträge bearbeiten oder auflisten. Die Funktionslogik selbst ist dabei nicht implementiert. Alle Eingabefelder ergeben sich aus den Datentypen der Attribute. Der Kunde kann somit bereits während der Definition der Anforderungen sehen, wie ein Teil seiner gewünschten Anwendung aussehen könnte.
Zur einfachen Erzeugung eines Prototypen bietet sich eine automatische Generierung mittels HTML und CSS an. Idealerweise kann der Anwender auswählen,

[10] IFPUG, http://www.ifpug.org

in welchem Design bzw. mit welchem Stylesheet die Anwendung erzeugt werden soll. Kunden, bei denen öfter Projekte umgesetzt werden, sehen dabei den Mockup sofort im bekannten Corporate Design.

3.4.3 Generierung einer benutzbaren Oberfläche

Sobald die Datenzugriffsschicht und die zugehörigen Entitäten existieren, könnte eine Oberfläche automatisiert generiert werden. Bei Rails- oder ASP.NET MVC-Applikationen könnte dies über Scaffolding-Funktionalitäten der Frameworks geschehen [**?**]. Im Gegensatz zum Mockup würde bei diesem Automatismus Funktionslogik zur Verarbeitung und Persistierung enthalten sein. Der Aufwand zur Umsetzung eines solchen Generators ist dementsprechend höher.

3.4.4 Generierung der Verzeichnisstruktur für den Quellcode

Innerhalb des Projektverzeichnisses existiert der Ordner *src/* der jedweden Quellcode enthält. Je nach eingesetzter Programmiersprache sind alle darin enthaltenen Unterverzeichnisse nach einem bestimmten Schema zu benennen. Bei Java-Projekten wird z.B. das Maven-Verzeichnislayout benutzt.
Sobald die Entscheidung für eine Software-Architektur getroffen wurde, ließe sich hiermit die Verzeichnisstruktur generieren.

3.4.5 Einbinden von Bibliotheken

Bibliotheken und Frameworks werden je nach Projekt über Mechanismen wie NuGet[11], Ivy[12] oder Maven[13] eingebunden.
Ebenfalls mit der Entscheidung für eine Software-Architektur ließen sich in den passenden Konfigurationsdateien die Standard-Bibliotheken aus Tabelle 3.1 einbinden.

3.4.6 Generierung der Dokumentation des Datenbankschemas

Das Datenbankdiagramm mit den Tabellen und Relationen wird im bestehenden Prozess manuell in der Online-Dokumentation hinterlegt, so dass neue Entwickler einen schnellen Überblick über die Datenstruktur haben. Bei Änderungen muss dieses Diagramm und weitere Dokumentation zum Datenbankschema aktualisiert werden.
Ein Ziel sollte es sein, dass bei einer Änderung der Anforderungen automatisch das Datenbankschema neu erstellt und die Dokumentation aktualisiert wird.

[11] http://www.nuget.org
[12] http://ant.apache.org/ivy/
[13] http://maven.apache.org

3.4.7 Generierung von Migrationen bei Änderungen am Datenbankschema

Alle Projekte des bestehenden Softwareentwicklungsprozesses nutzen bereits das Konzept von Datenbankschema-Migrationen. Sobald ein Entwickler eine Änderung am Datenbankschema der Applikation vornehmen will, muss er diese Änderung in einem eigenen SQL-Script hinterlegen. Jedes SQL-Script ist für sich genommen eine eigene Migration und enthält beliebige SQL-Statements. Die Scripte werden natürlich innerhalb des Git-Repositories versioniert.
Mit dem Erstellen der Applikation werden vom Buildscript automatisch alle ausstehenden Migrationen auf die Datenbank angewendet. Die Datenbanken der anderen Entwickler werden nach dem letzten Git-Pull und mit dem nächsten Build aktualisiert. Weiterhin lassen sich diese Migrationen packen und beim Kunden automatisiert einspielen.
Wünschenswert ist es nun, dass bei Änderungen der funktionalen Anforderungen automatisch eine passende Migration erstellt werden würde. Beispielsweise möchte der Kunde für seine Applikation ein zusätzliches Feld namens "Nachname". Es würde automatisch eine Migration mit dem Befehl

```
ALTER TABLE xyz ADD COLUMN nachname char(255);
```

erstellt werden.

3.4.8 Generierung von Quellcode für die Datenzugriffsschicht

Aus dem Klassendiagramm ergibt sich direkt, welche Entitäten bzw. Transfer-Objekte erzeugt werden. Die Entitäten enthalten alle Attribute, die das Objekt besitzt. Die Persistierung erfolgt dabei über die Datenbankzugriffsschicht. Jede Entität wird über eine DAL- bzw. DAO-Klasse [**?**] persistiert oder geladen. Jede DAO-Klasse stellt ihre Methoden über ein zugehöriges Interface bereit.
Die zu erstellenden Klassen ähneln sich und nehmen bei einer manuellen Implementierung viel Zeit in Anspruch. Durch geeignete Implementierung kann zwar der Aufwand für die CRUD-Methoden reduziert werden, die Entitäts-Klassen müssen hingegen immer händisch erzeugt werden. Eine automatisierte Generierung auf Basis des Klassendiagramms ist wünschenswert.

3.5 Ziele

Im Rahmen eines Kundenprojekts wurde das erste Mal Xtext zur automatischen Generierung von Code-Fragmenten genutzt[14].
In der Retroperspektive des Projekts wurde beschlossen, dass Xtext als Basis für zukünftige Projekte benutzt werden soll. Ein erster Schritt hin zur Umsetzung dieser Entscheidung ist die vorliegende Bachelorarbeit. Es geht dabei um folgende Ziele:

- Es muss eine formale Sprache spezifiziert werden, die den in Kapitel 3.4 beschriebenen Ideen gerecht wird. Die Spezifizierung der Sprache wird mit anderen Entwicklern und deren Anforderungen an die DSL abgestimmt.

[14]Konkret wurde die Generierung wie in Kapitel 3.4.8 beschrieben, vorgenommen

- Es muss ein Mechanismus geschaffen werden, mit dem auf einfache Art und Weise aus der formalen Sprache unterschiedliche Artefakttypen generiert werden können. Die Komplexität hinter Xtext und Eclipse soll dabei verborgen bleiben, so dass auch Webdesigner oder Personen ohne Programmierhintergrund die DSL nutzen und erweitern können.

- Es muss die Möglichkeit gegeben sein, dass der Entwickler für ein Projekt definieren kann, welche Code-Artefakte erzeugt werden sollen. Es muss also eine Konfigurationsoberfläche existieren, in der der Entwickler dediziert Generatoren aktiveren oder deaktivieren kann.

- Folgende Generatoren sollen im Rahmen der Bachelorarbeit implementiert werden:
  - Die Prototypen-Generierung aus Kapitel 3.4.2 soll umgesetzt werden. Dabei wird durch HTML und CSS ein Prototyp erzeugt, der sich durch die in den DSL definierten funktionalen Anforderungen ergibt. Es soll weiterhin die Möglichkeit bestehen, dass der Entwickler das Stylesheet für einen Prototyp auswählen kann. Webdesigner müssen ohne großen Aufwand in der Lage sein, neue Stylesheets zu integrieren.
  - Die Function Point-Analyse aus Kapitel 3.4.1 soll automatisch erzeugt werden.

Alle weiteren Anforderungen sollen im Laufe zusätzlicher Projekt- bzw. Abschlussarbeiten realisiert werden. Xtext soll eine zentrale Position bei der Umsetzung neuer Softwareprojekte innerhalb der NeosIT einnehmen.

4 Einführung in die Technologien

Aufgrund der Komplexität des Xtext-Frameworks ist es nötig, dass zuerst die im Rahmen dieser Thesis eingesetzten Technologien und Ideen kurz vorgestellt werden.

4.1 Modelle und Metamodelle

Domänenspezifische Sprachen stellen Sprachelemente bereit, um später eine bestimmte Problemdomäne, zum Beispiel eine Applikation, konkret zu beschreiben. Dabei wird für die konkrete Problemdomäne auch der Begriff *Modell* benutzt. Die verfügbaren und erlaubten Elemente des Modells werden hingegen mit Hilfe des *Metamodells* definiert. Beispielsweise kann das Metamodell definieren, dass jedes Modell über Elemente vom Typ Domäne verfügen muss und dabei jede Domäne Attribute besitzen darf. Bei der objektorientierten Programmierung wäre eine Klasse das Metamodell, eine Objektinstanz dieser Klasse hingegen das konkrete Modell. Damit nun eine domänenspezifische Sprache, das Metamodell, beschrieben werden kann, wird das *Metametamodell* eingeführt. Diese Sprachebene stellt eine Grundmenge bereit, um beliebige Metamodelle zu definieren. Innerhalb der objektorientierten Programmierung könnten die Schlüsselwörter und genutzten Zeichen der Programmiersprache als Metametamodell angesehen werden. Die Grafik 4.1 stellt die Beziehung zwischen den drei Begrifflichkeiten noch einmal dar.

4.2 Modellgetriebene Softwareentwicklung

Bei der modellgetriebenen Softwareentwicklung geht es in erster Linie um das Ableiten einer Applikation aus der Implementierung einer formalen Beschreibungssprache. Die Komplexität der dahinter liegenden Programmiersprache soll durch eine Abstrahierung der Problemebene versteckt werden. Die Erhöhung der Produktivität und kürzere Entwicklungszyklen stehen dabei im Vordergrund. Die Sprache zur Beschreibung der Problemdomäne wird *Domain Specific Language* (DSL) genannt [**?**].
Bei domänenspezifischen Sprachen wird zwischen internen und externen DSLs unterschieden. Interne DSLs werden in bereits bestehenden Programmiersprachen eingebettet. Dabei kann jede Programmiersprache als Host für eine interne DSL genutzt werden. Die Entwicklung solch einer DSL kann von jedem durchgeführt werden, der mit der Syntax der Host-Programmiersprache vertraut

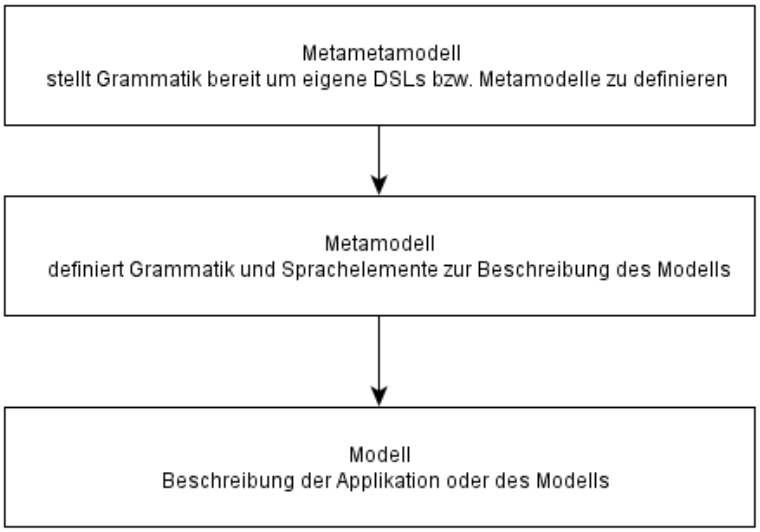

Abbildung 4.1: Modell, Metamodell und Metametamodell

ist. Häufig wird dabei das Konzept von Fluent Interfaces zur Abbildung eingesetzt. Die Programmiersprache Scala erlaubt beispielsweise eine recht hohe Anpassungsfähigkeit der Sprache, womit sich interne DSLs mit einer bestimmten Grammatik recht einfach abbilden lassen. Durch die enge Bindung an die Host-Programmiersprache werden interne Sprache in aller Regel von Softwareentwicklern eingesetzt.
Externe DSLs hingegen sind eigenständige Sprachen, deren Syntax und Sprachfeatures frei definiert werden können. Die Syntax kann so gewählt werden, dass sie auch für Endanwender ohne Kenntnisse in der Softwareentwicklung einfach und intuitiv zu bedienen sind. Neben Xtext existieren einige weitere Lösungen zum Erstellen von externen DSLs. Kommerzielle Produkte wie beispielsweise MetaEdit erlauben eine grafische Modellierung der DSL und lassen sich seit Version 5.0 über Plug-ins in Visual Studio und Eclipse einbinden. JetBrains bietet mit MPS eine Open Source Lösung an, mit der sich DSLs innerhalb einer IDE entwickeln lassen. Microsoft stellte unter dem Codenamen Oslo eine Sammlung von Werkzeugen vor, um Entwicklern bei der Erstellung von DSLs und Datenbankabfragen zu unterstützen. Oslo bestand aus den den Komponeten *Repository*, *Quadrant* und *M*. *M* diente dabei der Definition von DSLs und ähnelt in Ansätzen der Syntax von ANTLR [**?**]. Quadrant war hingegen ein grafisches Tool ur Bearbeitung von Datensätzen innerhalb eines Microsoft SQL Servers [**?**]. Nachdem die Entwicklung dieser Werkzeuge nicht weiter verfolgt worden ist, veröffentlichte Microsoft das Visualization and Modeling SDK mit dem sich ebenfalls DSLs entwickeln lassen, die innerhalb des Visual Studios eingesetzt werden können [**?**].

4.3 Eclipse

IBM begann die Entwicklung an der integrierten Entwicklungsumgebung Eclipse für Java im Jahre 1998 [**?**]. Durch die Veröffentlichung von Eclipse unter einer Open Source-Lizenz Ende 2001 bekamen auch andere Personen die Möglichkeit, Software auf Basis von Eclipse zu entwickeln. Neben den Java Development Tools entstanden zum Beispiel die PHP Development Tools zur Entwicklung von PHP-Projekten oder die Business Intelligence and Reporting Tools zur Erzeugung von

Berichten.
Der offizielle Eclipse Marketplace enthält Anfang 2013 rund 1500[15] veröffentlichte Plug-ins - viele weitere sind über weitere Quellen verfügbar. Eclipse selbst ist eine Java-Applikation, die innerhalb einer JVM läuft. Die aktuelle Eclipse-Runtime 4.2 setzt dabei auf den OSGi-Container Eclipse Equinox auf, der von der Eclipse Foundation entwickelt worden ist.

4.3.1 OSGi

Der OSGi-Container ist auf Basis der OSGi-Spezifikation entwickelt worden [**?**]. Die Spezifikation definiert unter anderem für so genannte Komponenten (Bundles) einen Lebenszyklus, wie er in Abbildung 4.2 dargestellt ist.

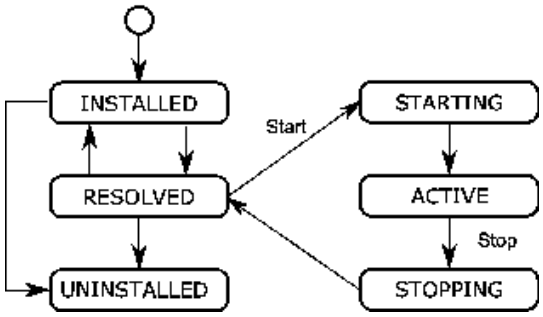

Abbildung 4.2: Lifecycle-Zustände für OSGi-Bundle nach [**?**]

In Eclipse ist jedes Plug-in eine Komponente und kann somit installiert, gestartet, gestoppt und deinstalliert werden. Über eine zentrale Service-Registy wird die Schnittstelle der Komponente veröffentlicht und kann von anderen Komponenten angesprochen werden.
Über die Datei *META-INF/MANIFEST.MF* eines Bundles wird unter anderem festgelegt, welche Klassen veröffentlicht werden und von anderen Komponenten sichtbar sind.
Neben der *MANIFEST.MF* besitzt in Eclipse jedes Plug-in eine Datei mit dem Namen *plugin.xml*, das die Erweiterbarkeit des Plug-ins definiert.

4.3.2 Extension und Extension Points

Eclipse sieht vor, dass jedes Plug-in Extension Points definieren kann. Extension Points werden anderen Plug-ins zur Verfügung gestellt, um die Funktionalität des bereitstellenden Plug-ins zu erweitern. Sobald ein Plug-in solch einen Extension Point in Anspruch nimmt, gilt das Plug-in als eine Extension. Ein Plug-in kann beliebig viele Extension Points bereitstellen und konsumieren.
Welche Extension Points bereitgestellt oder genutzt werden, wird innerhalb der *plugin.xml* definiert. Zusätzlich bietet die *Eclipse Plug-in Development Environment* die Möglichkeit, dass diese Konfiguration mit einem grafischen Editor vorgenommen werden kann.

Die Eclipse Workbench stellt von Haus aus bereits viele Extension Points zur Verfügung, so dass man z.B. Menüeinträge, Eigenschaftsseiten oder Editoren hinzufügen oder ändern kann.

[15] http://marketplace.eclipse.org/

4.3.3 Features

Sobald mehrere Plug-ins zusammengefasst werden, spricht man in Eclipse von Features. Damit ist nur gemeint, dass die Plug-ins organisatorisch zusammen gehören. Beispielsweise könnte das Feature *de.ckl.groovy.feature* eine Entwicklungsumgebung für Groovy sein und aus den einzelnen Plug-ins *de.ckl.groovy.editor* und *de.ckl.groovy.viewer* bestehen.

4.3.4 Workbench

In aller Regel erweitern die Plug-ins von Features die Extension Points der Workbench IDE UI. Die Workbench bietet unterschiedliche Extension Points an, z.B. um auf die Menüeinträge zuzugreifen oder neue Text-Editoren für einen bestimmten Dateityp zu registrieren.
Die Workbench selbst definiert unterschiedliche GUI-Elemente wie Tree-Viewer oder Editoren. Die Darstellung geschieht dabei mit Hilfe der GUI-Frameworks JFace bzw. SWT.
Über so genannte Perspektiven werden unterschiedliche GUI-Elemente von Plug-ins zusammengefasst. Dies ermöglicht das schnelle Wechseln zwischen unterschiedlichen Plug-in-Ansichten. Die Perspektive *Java* zeigt z.B. standardmäßig die logische Struktur des Java-Projekts über eine Tree-View und den Java-Quellcode-Editor an. Die Perspektive *Debug* deaktiviert hingegen die Tree-View und aktiviert das Plug-in zum Anzeigen des aktiven Stack Traces einer laufenden Java-Applikation.

4.3.5 Projekte

Anhand des Typs eines Projekts wird beim Laden desselben die zugehörige Perspektive initialisiert. Sobald ein neues Projekt durch die Eclipse IDE erstellt wird, wird innerhalb des Hauptverzeichnis des Projekts ein neuer Ordner *.settings* und die Datei *.project* erstellt.
Im Ordner *.settings* werden projektspezifische Einstellungen von Eclipse Plug-ins gespeichert. Beispielsweise enthält die Datei *.settings/org.eclipse.jdt.core.prefs* Informationen darüber, in welcher Java-Version der Quellcode vorliegt. Die Dateien werden dabei im .properties-Format gespeichert, so dass in jeder Zeile einer solchen Datei die Einstellungen im Format

```
Schlüssel=Wert
```

hinterlegt sind.

Die Datei *.project* enthält über eine definierte XML-Struktur Metainformationen über das Projekt, z.B. welche Natures oder Builder aktiviert worden sind [**?**].

Natures definieren, welche Plug-ins bzw. Features einem bestimmten Projekt zugeordnet sind [**?**]. Eine Nature kann z.B. *Java* oder *Xtext* sein. Neben den Natures wird in der *.project*-Datei hinterlegt, welche Builder existieren. Builder erzeugen, je nach Konfiguration automatisch oder manuell angesteuert, die Artefakte, die sich aus den Quellcodedateien ergeben. Der Java-Builder ruft beispielsweise für jede *.java*-Datei eines Projekts den Java-Compiler auf, der wiederum den Quellcode in eine *.class*-Datei kompiliert.

Jedem Projekt können mehrere Natures und Builder zugeordnet sein. Das Diagramm in Abbildung 4.3 stellt die grobe Beziehung zwischen den einzelnen Komponenten innerhalb der Eclipse-Umgebung grob dar.

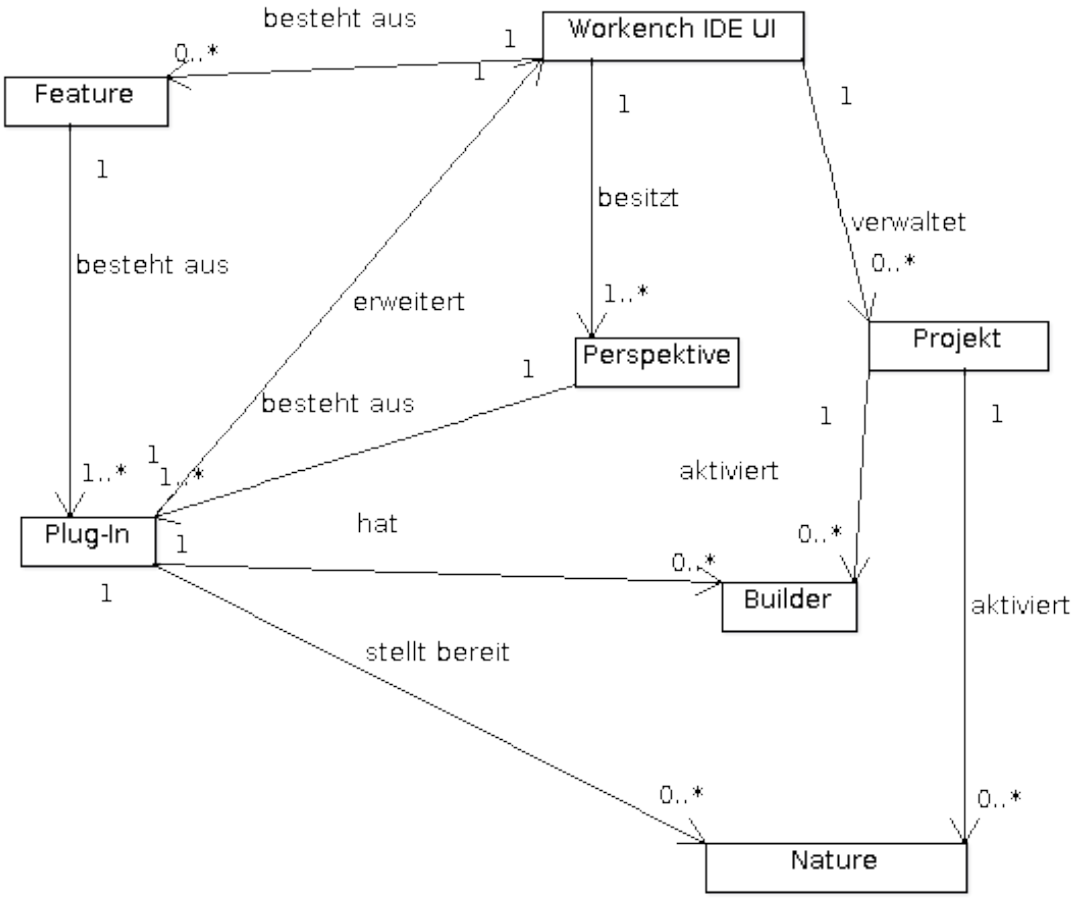

Abbildung 4.3: Beziehungen zwischen den einzelnen Eclipse-Komponenten

4.4 Eclipse Modeling Framework

Das Eclipse Modeling Framework besteht aus verschiedenen Plug-ins für die Eclipse-Plattform um eigene Metamodelle abbilden zu können. Die Erstellung dieser Metamodelle geschieht dabei über grafische Tools, die wiederum das Metamodell innerhalb eines eigenen XML-Dialekts konvertieren. Diese XML-Dateien werden mit der Endung *.ecore* gespeichert. EMF stellt weiterhin einen Generator bereit, der aus der Ecore-Datei Java-Klassen erzeugt. Diese werden von der Applikation als Grundlage für die Domänenobjekte genutzt.

4.5 ANTLR

Um Metamodelle mit Hilfe einer eigenen domänenspezifische Sprache zu definieren, sind in aller Regel zwei Komponenten nötig: Der *Lexer* kümmert sich um das Zerlegen des Eingabestreams in logisch zusammenhängende Zeichenfolgen, die sogenannten Token[16]. Die Token werden anschließend vom *Parser* als Grundlage für weitere Operationen genommen. Dazu zählt unter anderem die Überprüfung, ob die zugrundeliegende Grammatik syntaktisch gültig ist. Die Grammatik legt dabei die Regeln fest, in welcher Weise die Sprachelemente der DSL genutzt werden müssen. Für das automatische Erstellen von Lexern und Parsern existieren einige Tools wie z.B. yacc, JavaCC oder ANTLR. Xtext nutzt intern zum Parsen der DSLs bzw. Grammatiken das Tool ANTLR (*Another Tool for Language Recognition*).

4.6 Xtext

Für die vorliegende Arbeit wurde Xtext als Tool für externe DSLs ausgewählt. Xtext nutzt die Eclipse IDE als Grundlage für die Generierung. Unter anderem spielten bei der Entscheidung folgende Aspekte eine Rolle:

- Die Entwicklung von Xtext wird von der itemis AG getrieben. Die Mitarbeiter entwickeln aktiv an der Software und präsentieren ihre Ergebnisse in Java User Groups, in Blogs und bei großen Java-Konferenzen. Probleme werden per Twitter oder über das offizielle Xtext-Forum[17] von den Entwicklern sehr schnell beantwortet.

- Die Einstiegshürde in Xtext fällt durch die sehr gute Dokumentation und vorhandene Beispiele sehr einfach aus.

- Xtext erzeugt für die DSL eigene Plug-ins für Eclipse. Funktionalitäten wie Autovervollständung und Syntax Highlighting innerhalb des Editors der DSL sind somit automatisch vorhanden. Die Erweiterung um eigene Funktionalitäten ist verhältnismäßig einfach.

- Xtext ist Open Source. Bei Problemen oder Verständisschwierigkeiten von Xtext kann man mit einem Debugger die internen Abläufe innerhalb der Eclipse-Umgebung nachverfolgen.

Xtext selbst besteht aus mehreren Plug-ins für Eclipse. Eigene DSLs werden über die Xtext-Grammatik definiert. Über das Java-Tool ANTLR werden für die DLSs Lexer und Parser generiert, die entstehenden Java-Klassen werden hingegen von Xtext/XPand erzeugt. Die Xtext-Grammatik ist dabei selbst in Xtext definiert [**?**]. Xtext kann sich somit selbst erzeugen.
Xtext erzeugt aus den geschriebenen Grammatiken automatisch das Grundgerüst für eigene Eclipse Plug-ins. Weiterhin wird durch die integrierten Plug-ins des Eclipse Modeling Frameworks ein vollständiges Ecore-Metamodell der DSL generiert. Mit Hilfe von Lexer und Parser wird das Modell der DSL geparst und in ein Objektbaum auf Basis von Ecore transformiert. Über diesen Abstract Syntax Tree kann Java auf die Inhalte der DSL zugreifen.

[16] Scannerless Parser wie zum Beispiel zum Parsen von TeX sind Ausnahmen von der Regel und benötigen keinen eigenen Lexer

[17] http://www.eclipse.org/forums/index.php?t=thread&frm_id=27

Jedes Element innerhalb des Objektbaums beinhaltet die Referenzen zu Eltern- und Kindelementen. Somit kann von jedem Element aus über das komplette Metamodell iteriert werden.

4.6.1 Xbase

Xbase ist eine statisch-typisierte Programmiersprache, die selbst in einer Xtext-Grammatik definiert worden ist [**?**]. Sie stellt unter anderem Lambda-Ausdrücke bereit und kann Operatoren überladen [**?**]. Über Type Inferencing kann auf die volle Klassenhierarchie von Java zugegriffen werden.
In einer eigenen DSL kann Xbase eingesetzt werden, um diese um Ausdrücke zu erweitern. Eigene Programmlogik kann so innerhalb der DSL untergebracht werden **ohne** dass eigene Grammatiken definiert werden müssen. Xbase wird unter anderem von den Xtext-Komponenten Xtend und MWE benutzt.

4.6.2 Xtend

Die Xtext-Entwicklungsumgebung stellt die statisch-typisierte Programmiersprache Xtend bereit. Xtend unterstützt unter anderem Lambda-Expression und Extension Methods. Xtend ist durch eine Xtext-Grammatik definiert Quellen liegen unter [**?**] und nutzt Xbase.
Der Builder von Xtend erzeugt Java-Quellcode, der wiederum durch durch den Compiler zu lauffähigem JVM-Bytecode umgewandelt wird. Xtend soll als leichtgewichtige Programmiersprache und Alternative zu Java dienen. Die Programmiersprache ist fest in das Xtext-Framework integriert: neben den Standardfunktionalitäten, die Xtext für generierte DSLs bereitstellt, existiert ebenfalls Debugging Support in Eclipse.
Durch die Unterstüzung von Template Expressions [**?**] innerhalb von Textblöcken eignet sich diese Programmiersprache besonders für die Transformation der eigenen DSL in ein Ausgabeartefakt: Der Entwickler kann z.B. innerhalb eines Textsegments über ein Array iterieren:

```
var personen = ["Trinity", "Neo", "Morpheus"]
var artefakt = '''
  <ul>
  «FOREACH person : personen»
    <li>«person»</li>
  «ENDFOREACH»
  </ul>
'''
```

Strings werden durch drei Hochkommata eingeschlossen. Innerhalb dieses Blocks können XPand-Ausdrücke in Guillemot-Pfeilen genutzt werden. XPand-Ausdrücke haben dabei Zugriff auf die definierten Variablen innerhalb des Objekts oder der Methode. Eclipse unterstützt innerhalb dieser Ausdrücke Autovervollständigung.
Im Beispiel würde über das Array `personen` iteriert und der entstehende String in der Variablen `artefakt` gespeichert werden. Der entstehende String würde dann wie folgt aussehen:

```
  <ul>
    <li>Trinity</li>
    <li>Neo</li>
```

```
    <li>Morpheus</li>
  </ul>
```

4.6.3 MWE

Um wiederkehrende Arbeitsabläufe wie z.B. die Generierung von Code zu automatisieren, wird innerhalb der Xtext-Plattform die Modeling Workflow Engine genutzt. Die Modeling Workflow Engine besitzt eine eigene DSL, mit der beschrieben wird, in welcher Reihenfolge einzelne Arbeitsabläufe ausgeführt werden müssen. Die Grammatik der MWE-DSL ist dabei in einer Xtext-Grammatik definiert, die Xbase unterstützt[18]. Eigene Komponenten können durch die Implementierung des Java-Interfaces *IWorkflowComponent* bereitgestellt werden.

Xtext generiert beim Erstellen einer neuen DSL automatisch eine Workflow-Definition, die dann für die weitere Code-Generierung der DSL-Infrastruktur genutzt wird. Anhand des folgenden Quellcode-Fragments soll kurz die Grammatik und Funktionsweise von MWE beschrieben werden:

```
var projectName = "de.ckl.rapid"
var runtimeProject = "../${projectName}"
Workflow {
    // ... weitere Instruktionen
    component = DirectoryCleaner {
        directory = "${runtimeProject}/src-gen"
    }
    // ... weitere Instruktionen
}
```

Mit dem Schlüsselwort `var` werden Variablen definiert, die später wiederverwendet werden können. Auf Variablen können innerhalb von Zeichenketten mit Hilfe des Konstrukts `${...}` referenziert werden. Mit Hilfe des Wortes `Workflow` wird eine neue Instanz der Klasse `Workflow`[19] erzeugt. Hier können auch beliebig andere Klassen genutzt werden, die das Interface `IWorkflow` implementieren.

Innerhalb des `Workflow`-Blocks, gekennzeichnet durch die geschweiften Klammern, können nun Komponenten mit Hilfe des Schlüsselwortes `component` definiert werden. Diese werden der Reihe nach ausgeführt. Im obigen Beispiel wird eine neue Komponente eingefügt, indem eine neue Instanz der Klasse `DirectoryCleaner`[20] instanziert wird. In der Instanz wird die Bean-Eigenschaft `directory` zugewiesen. Komponenten müssen das Interface `IWorkflowComponent` [21] implementieren. Beim Setzen der Eigenschaft `directory` wird durch MWE automatisch die Setter-Methode `setDirectory(...)` aufgerufen.

Wie man an dem Codefragment erkennen kann, ist MWE ein gutes Beispiel für eine domänenspezifische Sprache, da sich die Grammatik allein auf die relativ einfache Problemdomäne *Sequenzielles Ausführen von Workflows* konzentriert.

[18] http://www.eclipse.org/modeling/emft/downloads/?project=mwe, SDK MWE2 (Runtime, Source), /eclipse/plugins/org.eclipse.emf.mwe2.language.source_2.3.0.v201206120758.jar, /org/eclipse/emf/mwe2/language/Mwe2.xtext

[19] org.eclipse.emf.mwe2.runtime.workflow.Workflow

[20] org.eclipse.emf.mwe.utils.DirectoryCleaner

[21] org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent

4.6.4 Guice

Das komplette Xtext-Framework nutzt das Dependency Injection-Framework Guice, das von Google entwickelt und als Open Source-Software veröffentlicht wurde. Im Gegensatz zum DI-Framework Spring ist Guice leichtgewichtiger[22]. Klassen, die das Interface `Module` implementieren, definieren die Bindings von Schnittstellen zu Klassen. Eine Konfiguration der Abhängigkeiten über XML-Dateien wie es in Spring möglich ist, ist standardmäßig nicht vorhanden.
Durch das Überschreiben von bestehenden Bindings können innerhalb des Xtext-Plug-ins eigene Klassen injiziert werden.
Neue Objektinstanzen werden nun über Guice erzeugt. Alle Abhängigkeiten werden anhand der in der Moduldefinition gebundenen Interfaces injiziert. Xtext generiert Moduldefinitionen mit dem Suffix *Module*. Innerhalb der Klasse `RapidUiModule` wird z.B. mit Hilfe der folgenden Methode das Interface `IXtextBuilderParticipant` an die konkrete Implementierung `MultipleBinderParticipant` gebunden:

```
public Class<? extends org.eclipse.xtext.builder
    .IXtextBuilderParticipant>
    bindIXtextBuilderParticipant() {
    return de.ckl.rapid.ui.builder.MultipleBuilderParticipant.class;
}
```

In jeder Klasse, die von Guice erstellt wird, bekommen nun alle Attribute mit der Annotation *@Inject* die zugehörige gebundene Instanz injiziert:

```
    @Inject
    IXtextBuilderParticipant builderParticipant;
```

4.6.5 Erstellung einer neuen DSL

- Der Benutzer erstellt innerhalb der Eclipse-Umgebung ein neues Xtext-DSL-Projekt. Dabei muss er den Namen der DSL *$dsl-name*, sowie den Paketnamen der DSL *$paket.name* definieren.
- Das Eclipse Plug-in des Xtext-Frameworks generiert daraufhin automatisch drei Eclipse Plug-ins:
  - *$paket.name*: Grundgerüst für das DSL-Backend. Dieses enthält den Parser, Lexer, Formatter, Metamodell, Scoping- und Validation-Provider. Der generierte Quellcode ist standardmäßig nicht abhängig von der Eclipse-Laufzeitumgebung und kann auch in Konsolen- oder Webanwendungen wiederverwendet werden.
  - *$paket.name*.test: Grundgerüst für das Ausführen von Unittests
  - *$paket.name*.ui: Grundgerüst für das User Interface. Dies beinhaltet unter anderem Content Assistents, Quickfixing und Outline Views. Der generierte Quellcode hängt dabei direkt von Eclipse IDE ab und kann ohne diese nicht ausgeführt werden.

  Jegliches Customizing durch den Benutzer darf nur innerhalb der *src*-Verzeichnisse in den Projektordnern geschehen. Alle Änderungen im Ordner *src-gen* werden beim Neuerstellen des Backends (z.B. bei Änderungen der Grammatik) überschrieben.

[22] guice-3.0.jar, aopalliance.jar javax.inject.jar sind kleiner als 720 KByte

- Der Benutzer definiert innerhalb des Projekts $paket-name:/src/*$paket.name*/*$dsl-name*.xtext die Grammatik der DSL
- Der Benutzer startet über den Menüpunkt *MWE2 Generate artifact* den Workflow zum Generieren des Backends
- Der MWE2-Workflow erzeugt nun im Verzeichnis *$paket.name*:/src-gen das komplette Backend. Ecore-Metamodell, Parser und Lexer werden neu generiert.
- Der Benutzer implementiert nun eigene Funktionalitäten, indem er seine eigenen Implementierungen in einer der Moduldateien bindet.

5 Realisierung

Nachdem die grundlegenden Technologien erklärt worden sind, folgt nun die Beschreibung der Realisierung. Als erstes wurde das in der Grafik **??** sichtbare UML-Diagramm einer Beispielapplikation erstellt, deren Anforderungen durch die DSL abgedeckt werden sollten. Der Einfachheit halber beschränkte es sich auf eine Forumsanwendung. Dabei wurden neben einfachen 1:n-Beziehungen auch die Möglichkeit gegeben, dass Benutzer in der Rolle *Moderator* über eine assoziative Klasse beliebig viele Foren moderieren können. Weiterhin besitzt die Klasse *Post* eine reflexive Assoziation, so dass eine Baumstruktur der Postings abgebildet werden kann. Die Methoden sind der Übersichtlichkeit halber ausgeblendet.

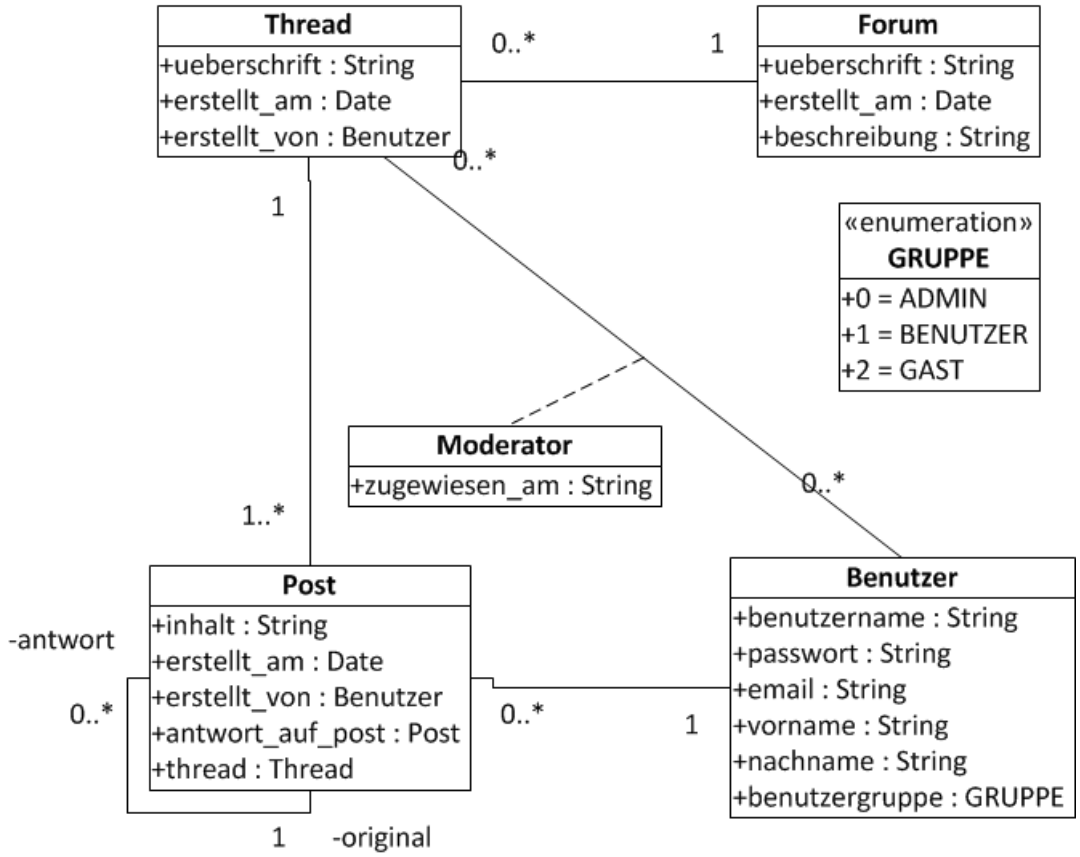

Abbildung 5.1: Beispielapplikation eines Forums

5.1 Definition und Umsetzung des Metamodells

Nach dem Design des UML-Diagramms wurde auf dieser Grundlage ein Prototyp des Metamodells erstellt.
Die Erstellung des Prototypen orientierte sich dabei an den Fragen:

- *Lässt sich die Sprache für einen Softwareentwickler intuitiv benutzen?* Bekannte Sprachkonstrukte aus verbreiteten Programmiersprachen sollten wiederverwendet werden.
- *Sind die Schlüsselwörter meiner Sprache eingängig?* Die Schlüsselwörter der Sprache müssen einerseits aussagekräftig sein, andererseits aber auch ein gewisses Maß an Abstraktion besitzen.
- *Kann ich mit dieser Sprache meine Beispielanwendung abbilden?* Die Sprache muss in der Lage sein, Kardinalitäten, Attribute, Klassen und Assoziationen abzubilden.
- *Ist der Detaillierungsgrad meiner Sprache zu hoch?* Das Entwerfen einer Sprache von Grund auf hat den Nachteil, dass man sich sehr schnell in Details verlieren kann. Angenommen, es gilt die Konvention, dass Datumswerte immer als Zeitstempel gespeichert werden, dann ist es nicht nötig, dass ein Datentyp *date* neben dem Datentyp *datetime* eingeführt wird.

Der Prototyp wurde iterativ entwickelt, bis letztendlich eine Grammatik gefunden wurde, bei der alle Fragen zufriedenstellend beantwortet werden konnten. Als Projektname für die DSL wurde *Rapid* mit dem in Grafik **??** zu sehenden Logo gewählt. Die Grammatik und Sprachelemente dieser DSL wird in den folgenden Unterkapiteln beschrieben.

Abbildung 5.2: Das Logo von Rapid

5.1.1 Allgemeiner Aufbau

Ein Modell wird in aller Regel innerhalb einer Datei definiert. Xtext erlaubt es allerdings, dass das Modell auch in mehreren Dateien zur Verfügung steht. So lassen sich bestimmte Komponenten auslagern.

- Kommentare innerhalb des Modells oder Metamodells werden standardmäßig über `//` oder `/* ... */` definiert.

- Wenn mehrere Elemente zusammengefasst werden sollen, wird dies durch eckige Klammern dargestellt. Hier wurde die Deklarierung von Arrays aus Programmiersprachen wie C oder Java übernommen.
- Innerhalb von Domänen kann das Schlüsselwort *self* benutzt werden, um auf die Domäne zu verweisen, in der sich das Element befindet.
- Zur Definition von Kardinalitäten im Metamodell werden die Token *** (0..n), *?* (0..1) und *+* (1..n) genutzt. Wurde keine Kardinalität angegeben, gilt immer, dass das Element genau einmal vorkommen muss. Die weitere Syntax kann unter [**?**] eingesehen werden.

Das Metamodell zum allgemeinen Aufbau sind folgendermaßen aus:

```
Model:
    (imports+=Import)*
    (('application' '{'
        ('name:' applicationName=STRING) &
        ('package:' package=QualifiedName)
        ('navigation:' ...
        )?
    '}'
    )?
    (enums+=Enum)*
    (types+=Type)*
    (domains+=Domain)*
    )
;

Import:
'import' importedNamespace=QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;
```

Jedes Modell kann beliebige JVM-Klassen über das Schlüsselwort *import* importieren. Diese Funktionalität ist nötig, wenn man Annotationen wie in Kapitel 5.4.2 benutzen und nicht immer den vollqualifizierten Namen der Annotation schreiben will. Eine Applikation muss einen Namen (*name*) beinhalten und einen Namensraum definieren (*package*). Nach dieser Präambel folgt die optionale Deklaration von Enumerationen, Typen und Domänen. Für die Beispielanwendung könnte folgendes Modell genutzt werden:

```
// Annotation importieren
import de.ckl.rapid.artifact.fpanalysis./*...*/.UseFunctionpointType

application {
    name: "Forumsanwendung"
    package: de.ckl.apps.forum
}

// Enumerationen
// Typen
// Domänen
```

5.1.2 Enumerationen

Eine Enumeration ist ein Datentyp zur Aufzählung von Werten. Rapid stellt Enumerationen bereit und orientiert sich dabei an der Nutzung von Aufzählungen, wie sie in C# verwendet werden:

```
Enum:
    'enum' name=ID '['
        values+=EnumValue (',' values += EnumValue)*
    ']'
;

EnumValue:
    name=ID ('=' value=INT)?
;

EnumDataType:
    'enum' '(' refEnum=[Enum] ')'(nullable?='?')?
;
```

Eine Enumeration wird dem Schlüsselwort *enum* definiert. Soll später auf diese Aufzählung verwiesen werden, muss ebenfalls *enum* benutzt werden. Danach folgt der Name der zu nutzenden Enumeration innerhalb der Klammern. Mit dem Fragezeichen *?* kann gekennzeichnet werden, dass die Nutzung dieses Datentyps NULL-Werte erlaubt. Innerhalb der Forumsapplikation können Enums zur Definition von Benutzergruppen genutzt werden:

```
// Definition der Enumeration
enum GRUPPE [0 = ADMIN, 1 = BENUTZER, 2 = GAST]

domain Benutzer {
    // Referenz zur Enumeration GRUPPE; Attribut kann null sein
    attr benutzergruppe: enum(GRUPPE)?
}
```

5.1.3 Datentypen

Rapid enthält eine Grundmenge an Datentypen wie *string*, *long*, *double*, *boolean* und *datetime*. Diese können je nach Gültigkeitsbereich NULL sein. Allerdings reichen in aller Regel diese Datentypen nicht aus und oft kommt es vor, dass man bestehende Datentypen verwenden oder neue Datentypen einführen will. Um diese Anforderung umzusetzen, wurde das Schlüsselwort *type* definiert und die Annotation *@MapToBackend* implementiert. Die formale Definition erlaubt, dass jeder neu definierte Datentyp annotiert werden kann:

```
Type:
    (annotations+=XAnnotation)*
    'type' name=ID
;

CustomDataType:
    'type' '(' refType=[Type] ')'(nullable?='?')?
;
```

Innerhalb des Modells kann nun z.B. dem Benutzer ein Attribut namens *guid* zuordnen, dass auf den Datentyp *System.Guid* aus dem .NET-Framework gemappt wird. Die Auswertung des Mappings muss dabei natürlich immer von den Generatoren erfolgen.

```
@MapToBackend("System.Guid")
type Guid

domain Benutzer {
    attr guidInActiveDirectory: type(Guid)?
}
```

5.1.4 Domänen

Domänen beschreiben letztendlich die Klassen einer Applikation:

```
Domain:
(annotations+=XAnnotation)*
'domain' name=ID
('extends' refParentDomain=[Domain]
    ('as' parentDomainLabel=STRING)?)?
    '{'
    ('label:' label=STRING)?
    ('description:' description=STRING)?
    (attributes+=Attribute)*
    (fieldgroups+=Fieldgroup)*
    (references+=HasManyReference)*
    (processes+=Process)*
    ('options' '{'
    (
        ('examples:' '['
            examples+= STRING (',' examples+=STRING)*
        ']')?
    )
    '}')?
'}'
;
```

Domänen können mit Annotationen ausgestattet werden und von anderen Domänen erben (*extends*). Die Vererbung ist optional. Zirkuläre Referenzen sind durch einen passenden Validator nicht möglich. Jede Domäne kann ein Label (*label*) und Beschreibung (*description*) besitzen. Innerhalb einer Domäne können beliebig viele Attribute, Referenzen, Prozesse und Feldgruppen definiert werden. Feldgruppen fassen mehrere Attribute zu einer Gruppe zusammen und ermöglichen beispielsweise in einer Oberfläche die Gruppierung von Eingabeelementen. Attribute sind entweder Felder von einfachen Datentypen oder Referenzen auf eigene Datentypen, Enumerationen oder andere Domänen. Eine Referenz auf eine andere Domäne kann als "gehört zu" oder "hat eine" gelesen werden. Many-To-Many-Referenzen werden in Kapitel 5.1.7 näher beschrieben.
Mit all diesen Elementen lässt sich eine Thread-Domäne der Beispielapplikation implementieren:

```
domain Thread {
    attr ueberschrift: string
```

```
    attr erstellt_am: datetime
    erstellt_von: Benutzer
    has-many posts: Post(mapped-by Post.thread)
}
```

5.1.5 Prozesse

Prozesse bilden innerhalb der Domäne User Stories ab, die während des Requirements Engineering aufgenommen worden sind:

```
Process:
    (annotations+=XAnnotation)*
    'process' name=ID ('{'
        ('label:' label=STRING)?
        ('description:' description=STRING)?
        ('view:' view=('detail' | 'edit' | 'list' | 'statistic')
            ('['
                referencesForView+=AttributeReference
                (',' referencesForView+=AttributeReference)*
             ']'
            )?
        )?
        ('delegates-to:' '['
            delegatesTo+=Transition (',' delegatesTo+=Transition)*
        ']')?
    '}')?
;
```

Neben den bereits bekannten Annotationen und Schlüsselwörtern *label* und *description* existieren innerhalb des Prozess-Kontextes die Schlüsselwörter *view* und *delegates-to*. Mit *view* wird definiert, welche Ausgabe dieser Prozess erzeugt, sobald er gestartet worden ist. Es folgen in eckigen Klammern alle Attribute, die innerhalb der Ausgabe angezeigt werden sollen. Über *delegates-to* wird gesteuert, an welche weiteren Prozesse dieser Prozess delegieren kann. Damit ließe sich beispielsweise eine Eingabemaske für neue Benutzer definieren, die nach Fertigstellung an den Prozess `listBenutzer` weiterleitet:

```
domain Benutzer {
    // Attribute
    process addBenutzer {
        label: "Neuen Benutzer hinzufügen"
        description: "Fügt einen neuen Benutzer hinzu"
        view: edit [
            self.passwort,
            self.email,
            self.vorname,
            self.nachname
        ]
        delegates-to: [ listBenutzer ]
    }

    process listBenutzer {
        label: "Alle Benutzer anzeigen"
        view: list
```

```
    }
}
```

5.1.6 Referenzen

Referenzen sind Verweise auf Attribute, Prozesse oder Multiplizitäten innerhalb der selben oder einer anderen Domäne. Ursprünglich sollte dabei folgende Grammatik genutzt werden:

```
Domaene->Attribut
bzw. Domaene->Prozess
bzw. Domaene->Multiplizitaet
```

Die Eindeutigkeit der Namen von *Attribut*, *Multiplizitaet* und *Prozess* hätten dabei über Validatoren sichergestellt werden sollen. Allerdings kann ANTLR nicht unterscheiden, welches Element nach dem Schlüsselwort "`->`" genommen werden soll. Die Problematik konnte ebenfalls nicht mit Syntactic Predicates gelöst werden. Stattdessen werden die unterschiedlichen Typen von Referenzen über das Token zwischen der Domäne und dem Namen der Referenz unterschieden:

```
Domaene.Attribut
bzw. Domaene->Prozess
bzw. Domaene*Multiplizitaet
```

5.1.7 Multiplizitäten

Ein wichtiger Teil einer jeden DSL besteht darin, Multiplizitäten zwischen einzelnen oder mehreren Domänen zu definieren. In Rapid existieren vier Möglichkeiten um Multiplizitäten zu definieren. Bei der Nutzung des Schlüsselwortes *has-many* bzw. *belongs-to* ist es immer nötig, dass der Endpunkt in der referenzierten Domäne benannt wird. Dies kann entweder ein Attribut oder eine Multiplizität sein. Dieses Vorgehen ist nötig, um eine eindeutige Zuordnung zwischen Domänen zu erstellen. Als Vorbild diente das Mapping-Konzept der Java Persistence API [**?**], [**?**]. Die Unterscheidung zwischen *has-many* und *belongs-to* wird von der DSL nicht aktiv berücksichtigt, sondern dient alleine dem Anwender zur intuitiven Bedienung der DSL.
Bei folgendem Code wird definiert, dass die Domäne `D1` der Domäne `D2` angehört. Der Domäne `D2` ist diese Verbindung nicht explizit bekannt. Es besteht eine 1:n-Beziehung zwischen D2:D1.

```
domain D1 {
    attr d2: D2
}

domain D2 {
}
```

Um nun der Domäne `D2` die Verbindung bekannt zu machen, wird das *has-many*-Attribut genutzt. Dieses Konstrukt verlangt, dass in der referenzierten Domäne der Endpunkt benannt wird:

```
domain D1 {
    attr d2: D2
}
```

```
domain D2 {
    has-many d1: D1(mapped-by D1.d2)
}
```

Soll eine m:n-Beziehung hergestellt werden, müssen beide Domänen mit *has-many*-Schlüsselwörtern ausgestattet werden. Referenzen werden mit Hilfe des Sternchens `*` von Attributen unterschieden:

```
domain D1 {
    has-many d2: D2(mapped-by D2*d1)
}

domain D2 {
    has-many d1: D1(mapped-by D1*d2)
}
```

Die letzte Möglichkeit besteht darin, dass eine m:n-Beziehung mit Hilfe einer Assoziationsdomäne hergestellt wird. Im letzten Beispiel stellt die Domäne `A1` die Verbindung zwischen den beiden Domänen her:

```
domain D1 {
    has-many d2: D2(mapped-by A1.d1)
}

domain D2 {
    has-many d1: D1(mapped-by A1.d2)
}

domain A1 {
    attr d1: D1
    attr d2: D2
}
```

Innerhalb der DSL wird durch passende Quickfixes und Validatoren der korrekte Einsatz der Multiplizitäten sichergestellt.

5.2 Erweiterung der Basisfunktionalitäten von Xtext

Werden die Xtext-Artefakte generiert, erzeugt Xtext anhand der DSL alle Klassen, die für das eigene Eclipse Plug-in nötig sind. Dieser Vorgang muss mit jeder Änderung an der DSL manuell ausgeführt werden. Bei der ersten Generierung werden alle Dateien innerhalb der Ordner *src-gen* erzeugt. Diese dürfen vom Entwickler nicht angepasst werden, da bei jeder Änderung an der Grammatik eine neue Neuerstellung dieser Dateien stattfindet. Die eigentliche Implementierung geschieht in den .java-Dateien des Ordners *src*. Xtext nutzt als Präfix für die Klassen den Namen der DSL, so dass jede automatisch generierte Klasse innerhalb der *src*-Ordner mit `Rapid` beginnt. Diese Klassen werden einmalig beim ersten Lauf der Artefakt-Generierung erzeugt und werden nicht überschrieben.

Die Architektur der gesamten Xtext-Infrastruktur erlaubt es auf einfache

Art und Weise der DSL Features hinzuzufügen, die den Anwender beim Umgang mit der Sprache unterstützen. Im Folgenden soll ein kurzer Einblick gegeben werden, welche Features angepasst worden sind.

5.2.1 Scoping

Scoping bedeutet, dass innerhalb eines bestimmten Kontextes nur die Teilmenge eines bestimmten Typs erlaubt ist. Die Auswahl der erlaubten Elemente wird also eingeschränkt.
Xtext generiert standardmäßig die Klasse `RapidScopeProvider`, in der das Scoping definiert werden kann. Die Nutzung von Annotationen[23] macht es nötig, dass zwei verschiedene Scope Provider implementiert werden:

- Die Klasse `RapidScopeProvider` erbt von `XbaseWithAnnotationsScopeProvider`[24]. Darüber wird sichergestellt, dass Annotationen per Autovervollständigung innerhalb des Editors angezeigt werden können. Das Scoping wird an die Klasse `RapidDeclarativeScopeProvider` delegiert.

- Innerhalb von `RapidDeclarativeScopeProvider` können eigene Scopings implementiert werden, indem eine neue Methode mit der benötigten Signatur `scope_$Rückgabeelement($KontextElement, EReference)` erstellt wird. Der folgende Quellcode liefert alle erlaubten Attribute zurück, die innerhalb des Elements `Fieldgroup` verfügbar sind:

  ```
  IScope scope_Attribute(Fieldgroup ctx, EReference ref) {
      return scopeFor(((Domain)ctx.eContainer()).getAttributes())
  }
  ```

In Rapid wurden Scopings definiert, um die Auswahl von Prozessen, Attributen und Referenzen einzuschränken. Folgender Auszug aus Rapid stellt eine 1-n Beziehung zwischen der Domäne `MyDomain` und `OtherDomain` her:

```
domain MyDomain {
    has-many od: OtherDomain(mapped-by: OtherDomain.belongsToMyDomain)
}

domain OtherDomain {
    attr belongsToMyDomain: MyDomain
    attr otherAttribute: string
}
```

Ohne Scoping würden bei der Autovervollständigung hinter `mapped-by: OtherDomain` die beiden Attribute `belongsToMyDomain` und `otherAttribute` vorgeschlagen werden. Da das Gegenstück einer Referenz immer vom selben Typ sein muss, in dem sich der Cursor gerade befindet, ist das Attribut `otherAttribute` nicht erlaubt. Das Scoping filtert im Beispiel alle Elemente heraus, die nicht vom Typ `MyDomain` sind und in der Autovervollständigung erscheint deshalb nur `belongsToDomain`. Sollte der Anwender manuell eine andere Referenz eintragen, wird dies als semantischer Fehler markiert. Dieser muss vom Anwender behoben werden, da sonst der Buildprozess nicht gestartet werden kann.

[23] siehe Kaptiel 5.4.2
[24] org.eclipse.xtext.xbase.annotations.scoping.XbaseWithAnnotationsScopeProvider

5.2.2 Validation

Um einzelne Elemente auf ihre Gültigkeit zu überprüfen, erzeugt Xtext die Klasse `RapidJavaValidator`. Jede dort mit *@Check* annotierte Methode wird aufgerufen, sobald eine Ausprägung der DSL gespeichert wird. Mit Hilfe der Methoden `error()`, `warning()` und `info()` können zu validierende Elemente mit einer eindeutigen Validator-ID markiert werden. Diese werden dann in der *Problem View* von Eclipse aufgelistet.
Im konkreten Fall wurden unter anderem Validatoren implementiert, um

- zu erzwingen, dass nur Annotationen benutzt werden dürfen, die selbst mit `@Attribute`, `@Process`, `@Domain`, `@HasManyReference` oder `@Type` annotiert sind.
- zu verhindern, dass zirkuläre Abhängigkeiten zwischen vererbten Domänen entstehen.
- das Benutzen des Attribut-Namens *id* zu verbieten, da dieses Attribut für jede Domäne automatisch gesetzt wird.
- doppelte Attribute-, Prozess- oder Referenznamen innerhalb von Domänen zu unterbinden.
- das doppelte Vorhandensein von Enumeration-Werten zu verhindern.
- die korrekte Benutzung von m:n-Beziehungen zu überprüfen.
- doppelte Nutzung von Attributen innerhalb von Feldgruppen zu verhindern

Einen Sonderfall nehmen die Validatoren mit dem Präfix `provide` ein. Diese markieren bestimmte Stellen innerhalb der DSL als “informativ“, so dass mit Hilfe von Quickfixes alternative Lösungswege beschritten werden können.

- `provideAddCrudOperationQuickfix(...)` sorgt dafür, dass wenn eine Domäne keinen Prozess mit dem Namen `create`, `delete` oder `update` besitzt, dieses per Quickfix automatisch erstellt werden kann.
- `provideIntroduceOfAssociationTable(...)` markiert die betreffende Stelle, wenn zwischen zwei Domänen eine direkte m:n-Beziehung existiert, ohne dass eine explizite Assoziationsdomäne hinzugefügt worden ist.

Der folgende Code überprüft beispielsweise, ob der Attributname in einer Domäne doppelt vorhanden ist:

```
@Check
public void noDuplicateAttributeNameInDomain(Attribute attribute) {
    Domain self = (Domain) attribute.eContainer();
    List<String> names = new ArrayList<String>();

    for (Attribute a : self.getAttributes()) {
        if (names.contains(a.getName())) {
            error("Das Attribut mit dem Namen \"" + a.getName()
                + "\" ist in der Domäne \"" + self.getName()
                + "\" bereits definiert worden.", attribute,
                RapidPackage.Literals.ATTRIBUTE__NAME,
                ERROR_ATTRIBUTE_NAME_ALREADY_EXISTS);
        }
```

```
            names.add(a.getName());
        }
}
```

5.2.3 Formatting

Xtext stellt über die Klasse `RapidFormatter` die Möglichkeit bereit, dass eine Ausprägung einer DSL über die Tastaturkürzel *Strg+Shift+F* formatiert werden kann. Standardmäßig ist keine Formatierung aktiviert, so dass mit Ausführen des genannten Tastenkürzels der komplette DSL-Code einfach hintereinander gereiht wird. Um den DSL-Code zu formatieren, wurde die Methode `configureFormatting(FormattingConfig c)` überschrieben und in ihr definiert, wie einzelne Codeabschnitte formatiert werden sollen. Mit Hilfe der Methode `RapidGrammarAccess.findKeywordPairs(...)` wurde die Einrückung von Blöcken definiert. Blöcke werden von den Paaren `{` und `}` sowie `[` und `]` definiert. Letztere werden zur Kennzeichnung von Aufzählungen innerhalb der DSL benutzt.
Weiterhin wurde die Methode `RapidGrammarAccess.findKeywords(...)` genutzt, um Zeilenumbrüche hinter bestimmten Schlüsselwörtern zu erzwingen und Leerzeichen zwischen Elementen zu entfernen.
Zur Einrückung von Blöcken mit eckigen Klammern wird folgender Code genutzt:

```
for (Pair<Keyword, Keyword> pair : rga.findKeywordPairs("[", "]")) {
    c.setLinewrap().after(pair.getFirst());
    c.setIndentationIncrement().after(pair.getFirst());
    c.setLinewrap().before(pair.getSecond());
    c.setIndentationDecrement().before(pair.getSecond());
}
```

Für den Benutzer der DSL bedeutet die Formatierung eine große Hilfe. Werden innerhalb einer Besprechung die Anforderungen des Kunden aufgenommen, fehlt die Zeit für manuelle Formatierungen. Der DSL-Code erhält dabei eine gut leserliche und strukturierte Form, in die man sich schnell einarbeiten kann.

5.2.4 Quickfix

Ein Quickfix wird innerhalb der Klasse `RapidQuickfixProvider` definiert, indem eine Methode mit der Annotation *@Fix($ValidatorID)* markiert wird. Sobald ein Validator ein Element mit der Validator-ID *$ValidatorID* markiert, steht dem Benutzer dieser Quickfix zur Verfügung.
Folgende Quickfixes wurden implementiert:

- Generieren von den Standard-CRUD-Prozessen
- Erstellen einer Assoziationsdomäne bei m:n-Beziehungen zwischen zwei Domänen, die wiederum zusätzliche Attribute enthalten kann
- Hinzufügen einer Referenz, falls diese in einer Assoziationsdomäne fehlt.

Weiterhin wurde die Methode `createLinkingIssueResolutions(...)` überschrieben. Diese Methode wird standardmäßig aufgerufen, wenn auf ein Element verwiesen wird, das nicht existiert. Um auf fehlende Domänen, Typen oder

Prozesse zu reagieren, wurde die Klasse `LinkIssueResolutionDelegate` implementiert. Diese delegiert an weitere Klassen, die für den jeweiligen Typ zuständig sind. `DomainModification` wird z.B. aufgerufen, wenn eine Domäne nicht existiert.
Das gewählte Vorgehen ist etwas unglücklich, da `LinkIssueResolutionDelegate` Referenzen zu den Modifikatoren enthält. Hier wäre eine Lösung mit Hilfe von Guice Multibindings deutlich einfacher und eleganter gewesen. Leider funktioniert das Multibinding mit der in Xtext genutzten Version von Google Guice nicht. Das UML-Diagramm 5.3 stellt die Beziehung zwischen den Klassen dar.

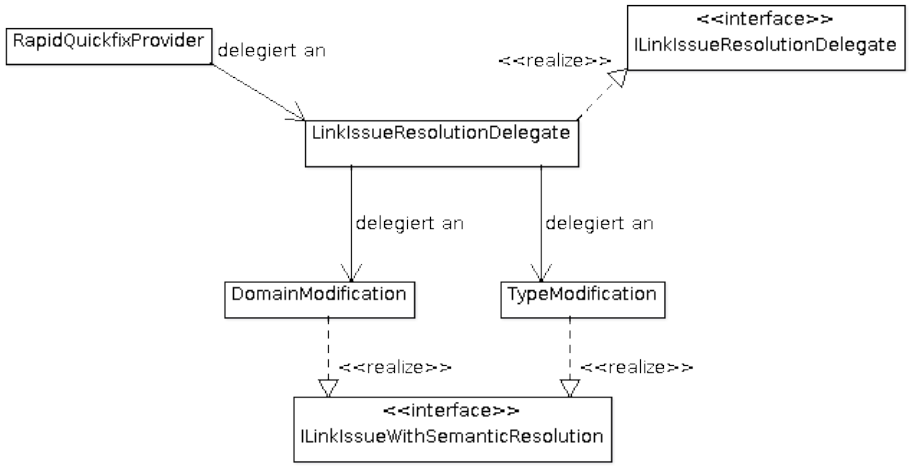

Abbildung 5.3: Delegierung von Quickfixes

Auf die Quickfixes kann innerhalb des Eclipse-Texteditor mit der Tastenkombination *Strg+1* zugegriffen werden.

5.2.5 Documentation Provider

In Java-Projekten von Eclipse wird beim Überfahren eines Elements mit dem Cursor der JavaDoc-Kommentar der Klasse, des Attributs oder der Methode angezeigt. Um den Benutzer der DSL ebenfalls die Möglichkeit an die Hand zu geben, sich Informationen über referenzierte Elemente anzuzeigen, wurde die Klasse `RapidEObjectDocumentationProvider` erweitert. Die darin enthaltene Methode `getDocumentation(EObject eObject)` wird immer aufgerufen, sobald mit dem Cursor über ein beliebiges Element innerhalb der DSL gefahren wird. Sollte es sich bei dem Methodenparameter *eObject* um eine Instanz einer Domäne, eines Attributs oder eines Prozesses handeln, wird per Java Reflection-Mechanismus überprüft, ob die Eigenschaft *label* oder *description* gesetzt ist. Aus diesen beiden Eigenschaften wird dann die Hover-Dokumentation für das

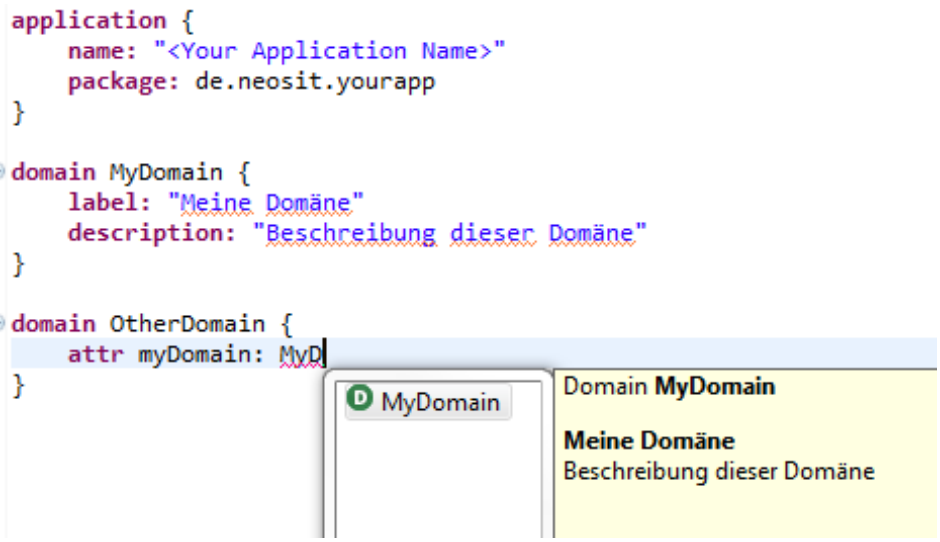

Abbildung 5.4: Die vom Documentation Provider erzeugte Dokumentation innerhalb der Umgebung

Element innerhalb von Eclipse erzeugt, wie es in Grafik 5.4 zu sehen ist.

5.2.6 Outline View

Unter der Outline View versteht man in Eclipse eine Übersicht über die wichtigsten Elemente innerhalb einer Datei. In Java-Dateien werden beispielsweise alle Attribute, Methoden, Enumerationen und ähnliches in einer Baumstruktur aufgelistet, so dass schnell zu den passenden Stellen innerhalb des Quellcodes navigiert werden kann. Xtext erzeugt ebenfalls eine Outline View, die in der Klasse `RapidOutlineTreeProvider` hinterlegt ist. Die Klasse erbt von `DefaultOutlineTreeProvider`, die wiederum die Basisfunktionalität für die Erzeugung der Baumstruktur enthält. Die Standardansicht listet alle Schlüsselwörter innerhalb der DSL hierarchisch auf. Da diese Ansicht allerdings für den Benutzer sehr unübersichtlich ist, wurden die Outline View so angepasst, dass logisch zusammenhängende Elemente der DSL gruppiert werden. Um die Darstellung aufzuwerten, wurden innerhalb der `RapidLabelProvider` für jedes Element, für das ein Icon hinterlegt, die passenden Dateinamen angegeben. Das Ergebnis ist in der Grafik 5.5 zu sehen.

5.2.7 Tests

Wie auch bei der Erstellung des Metamodells wurden die Anpassungen der Features mit Unittests abgesichert. Die Tests wurden dabei in der Sprache Xtend geschrieben, so dass Multiline-Strings verfügbar waren. Je nach Komplexität des Testfalls wurden Text-Fixtures entweder in eine eigenständige Datei ausgelagert oder durch Multiline-Strings abgebildet. Xtext bietet einen eigenen Runner für JUnit an. Außerdem existieren einige Hilfeklassen, mit denen unter anderem die Validatoren auf einfache Weise getestet werden können. Der Aufbau einer Testklasse ähnelt sich dabei:

```
@InjectWith(typeof(RapidInjectorProvider))
```

Abbildung 5.5: Outline View für Rapid innerhalb von Eclipse

```
@RunWith(typeof(XtextRunner))
class RapidDslTest extends AbstractXtextTests {
    @Inject extension ParseHelper<Model>

    @Inject extension AnnotationUtil

    override setUp() {
        super.setUp();
        with(typeof(de.ckl.rapid.RapidStandaloneSetup));
    }

    @Test
    def enumsCanBeParsed() {
        var rs = '''
enum TestEnum [
    VAL1,
    VAL2,
    VAL3=3
]
'''.parse();

        var enum = rs.enums.head
        assertEquals("TestEnum", enum.name);
   }
}
```

Mit der Annotation `@Inject` wird eine Instanz der Klasse `ParseHelper` injiziert und deren Methoden als Extension Methods für bestehende Klassen verfügbar gemacht. Mit `@Test` werden definiert, dass JUnit diese Methode überprüft. Die Extension Method `parse()` wandelt den Multiline-String in das Ecore-Modell um. Schließlich wird mit der statisch importierten Methode `assertEquals` der Soll- und Istwert verglichen.

5.3 Unterstützung von mehreren Generatoren

Die langfristige Strategie für Rapid ist, dass sie als Grundlage für die unter Kapitel 3.4 beschriebenen Verbesserungen dient. Die Architektur von Rapid soll dabei zwei Ideen folgen:

- *Als Anwender möchte ich dediziert auswählen können, welche Generatoren ich aktiviere.*
  Ein Webdesigner benötigt keine Function Point-Analyse oder einen Code-Generator für PHP, sondern möchte sich auf das Mocking der Anwendung konzentrieren.

- *Als Entwickler möchte ich auf einfache Art und Weise Rapid als DSL Grundlage nutzen und dafür mein eigenen Code-Generator schreiben können.*
  Rapid stellt das Grundgerüst für eine DSL zur Verfügung. Anwender, die zum Beispiel Applikationen in Ruby oder Python entwickeln möchten oder müssen, möchten Rapid als Basis nutzen und dafür einen Generator schreiben können.

Xtext ist darauf ausgerichtet, dass es zu einer DSL genau einen Generator gibt. Es ist nicht vorgesehen, dass mehrere Generatoren angesprochen werden können. Natürlich könnte der Generator an weitere Generatoren delegieren - die Aufgabe eines Generators ist dabei aber nicht mehr gegeben. Um nun mehrere Generatoren nutzen zu können, muss der Builder Participant erweitert werden. Dies wird im folgenden Kapitel beschrieben.

5.3.1 Implementierung eines Builder Participants

Xtext bietet den Extension Point *org.eclipse.xtext.builder.participant* an. Jedes Klasse, dass diesen Extension Point benutzen will, muss das Interface `IXtextBuilderParticipant`[25] implementieren. Sobald ein Build angestoßen wird, zum Beispiel nach dem Speichern der DSL-Ausprägung, werden alle registrierten Projekt-Builder für die Nature des Projekts ausgeführt. Xtext registriert dabei für Xtext-Natures die Klasse `XtextBuilder`[26]. Die Methode `doBuild(...)` delegiert letztendlich die Ausführung des Buildprozesses an die Klasse
`RegistryBuilderParticipant`[27] weiter. Diese liest über die `ExtensionRegistry` alle Builder aus, die für den Extension Point *org.eclipse.xtext.builder.participant* registriert sind und startet für jeden Participant den Buildprozess.

Innerhalb der *de.ckl.rapid.ui/plugin.xml* ist standardmäßig eingetragen, dass der Extension Point *org.eclipse.xtext.builder.participant* genutzt wird. Die genaue Bindung ist in der *plugin.xml* nicht hinterlegt, sondern nur, dass die Factory
`RapidExecutableExtensionFactory`[28] sich um die Bereitstellung einer solchen

[25] org.eclipse.xtext.builder.IXtextBuilderParticipant
[26] org.eclipse.xtext.builder.impl.XtextBuilder
[27] org.eclipse.xtext.builder.impl.RegistryBuilderParticipant
[28] de.ckl.rapid.ui.RapidExecutableExtensionFactory

Extension kümmert. `RapidExecutableExtensionFactory` liefert anhand des gebundenen Interface-Namens die zugehörige Instanz, die in der Guice-Konfiguration hinterlegt ist.

Zuerst wurde die `MultipleBuilderParticipant`[29] erstellt, die von `BuilderParticipant`[30] erbt. `BuilderParticipant` implementiert das Interface `IXtextBuilderParticipant` und wird in der `AbstractRapidUiModule` standardmäßig über Guice an das Interface gebunden. Damit nun der eigene `MultipleBuilderParticipant` aufgerufen wird, wurde in der `RapidUiModule` die Methode `bindIXtextBuilderParticipant()` überschrieben und die Bindung des Interfaces `IXtextBuilderParticipant` an die Klasse `MultipleBuilderParticipant` definiert.

Die einfachste Möglichkeit zum Ausführen eigener Generatoren war nun das Überschreiben der Methode `handleChangedContents(...)` innerhalb des `MultipleBuilderParticipant`. Damit sich neue Artefakt-Generatoren für die DSL registrieren können, wurde in der plugin.xml ein eigener Extension Point namens `extensionpoint.generator` innerhalb des Plug-ins *de.ckl.rapid.ui* eingeführt. Jede zu registrierende Extension muss dabei das Interface `IArtifactGenerator` implementieren. Die Methode `handleChangedContents(...)` iteriert über alle registrierten und aktivierten Artefakt-Generatoren und führt diese sequenziell aus. Im UML-Diagramm 5.6 ist die Beziehung zwischen den Klassen noch einmal dargestllt.

5.3.2 Auswahl der zu nutzenden Generatoren

Jeder Anwender sollte die Möglichkeit besitzen, einzelne Artefaktgeneratoren für das Projekt zu aktivieren oder deaktivieren. Aus diesem Grund wurde eine neue Property Page eingeführt. Property Pages werden dargestellt, wenn innerhalb der Eclipse-Umgebung die Einstellungen eines Projekts mit Rechtsklick *Project → Properties* oder Tastaturkürzel Alt+Enter aufgerufen werden und beinhalten Konfigurationseinstellungen für ein Projekt. In der *plugin.xml* wurde dazu der Extension Point `org.eclipse.ui.propertyPages` genutzt, um eine neue Property Page innerhalb des Abschnitts "Rapid"hinzuzufügen. Die entwickelte Klasse `RapidPropertyPage` erzeugt dabei mit SWT eine einfache Tabellenansicht der verfügbaren Plug-ins, die im Extension Point `extensionpoint.generator`[31] registriert sind. Durch eine Checkbox lässt sich der jeweilige Artefakt-Generator aktivieren bzw. deaktivieren. Sobald der Anwender die Einstellungen speichert, werden die Einstellungen in der Datei *.settings/de.ckl.rapid.ui* persistiert. Die Grafik 5.7 zeigt die Auswahl der Artefakt-Generatoren innerhalb von Eclipse.

5.4 Erstellen der Generatoren

Mit der Bereitstellung des Extension Points `extensionpoint.generator` war die Grundlage geschaffen, damit sich nun beliebige Generatoren in den Build-Prozess der DSL einklinken konnten. Jeder Artefakt-Generator ist dabei ein eigenes Eclipse Plug-in und kann unabhängig von anderen Generatoren ausgeliefert werden. Neue Generatoren können von Entwicklern so unabhängig entwickelt werden.

[29] de.ckl.rapid.ui.builder.MultipleBuilderParticipant
[30] org.eclipse.xtext.builder.BuilderParticipant
[31] de.ckl.rapid.ui.extensionpoint.generator

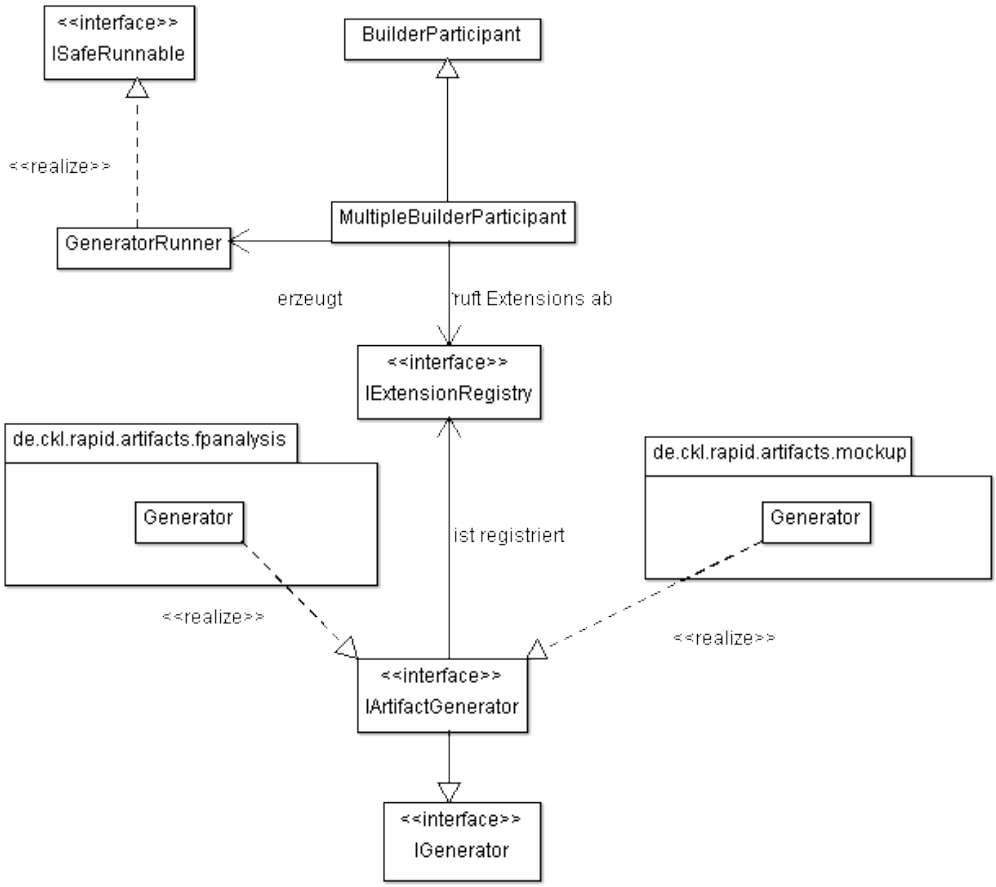

Abbildung 5.6: Mehrere Generatoren mit der Klasse MultipleBuilderParticipant ansteuern

Um einen neuen Generator zu erzeugen, muss ein neues Eclipse-Plug-in generiert werden. Generatoren sollten dabei innerhalb des Namespaces `de.ckl.rapid.artifact` liegen. Nach der Generierung des Plug-in-Grundgerüsts muss in der *plugin.xml* der Extension Point zur Registrierung dieses Generators angesprochen werden:

```
<extension
      point="de.ckl.rapid.ui.extensionpoint.generator">
   <client
       class="de.ckl.rapid.artifact.mockup\
       .GeneratorExtensionFactory:de.ckl.rapid.artifact\
       .mockup.generator.Generator">
   </client>
</extension>
```

Da die Plug-ins ebenfalls mit Guice entwickelt werden sollten, muss für jedes Plug-in eine eigene Extension Factory geschrieben werden. Dies ist nötig, da die Zugriffsbeschränkungen für Klassen zwischen den OSGi-Modulen keine weitere Abstrahierung erlaubt. Die Extension Factory kann dabei folgendermaßen aussehen:

```
public class GeneratorExtensionFactory
    extends RapidExecutableExtensionFactory {
```

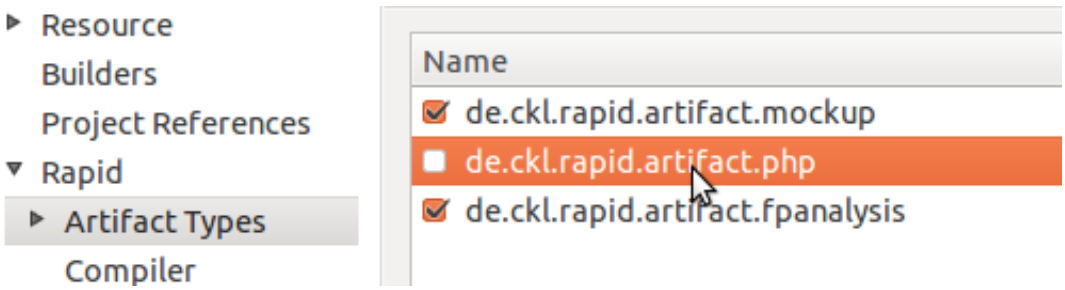

Abbildung 5.7: Property Page zur Auswahl der zu aktivierenden Generatoren

```
    @Override
    public Bundle getBundle() {
        return FrameworkUtil.getBundle(\
            GeneratorExtensionFactory.class);
    }

    private Injector generatorInjector;

    public Injector getInjector() {
        if (generatorInjector == null) {
            generatorInjector = super.getInjector()\
            .createChildInjector(
                new GeneratorModule()
            );
        }

        return generatorInjector;
    }
}
```

Wichtig ist, dass die Methode `RapidExecutableExtensionFactory.getBundle()` überschrieben wird und das ggw. Plug-in zurückliefert. Die Methode `getInjector()` wird nur benötigt, wenn ein eigenes Guice-Modul benutzt wird. Solch ein Modul könnte z.B. sein:

```
public class GeneratorModule extends AbstractGenericModule {
    public Bundle bindBundle() {
        return FrameworkUtil.getBundle(GeneratorModule.class);
    }

    public Class<? extends IJarResourceProvider>\
        bindResourceProvider() {
        return JarResourceProvider.class;
    }
}
```

Da der Aufbau eines Artefakt-Generators recht ähnlich ist, bietet sich hier eine eigene kleine DSL an, um die Infrastruktur dafür zu erzeugen.

5.4.1 Mocking

Der erste Schritt für den Mocking-Generator bestand darin, dass mit Hilfe von Twitter Bootstrap ein Dummy-Layout erstellt wurde, wie der spätere Mockup

aussehen sollte. Twitter Bootstrap bot sich als Framework an, da viele CSS-Klassen bereits vordefiniert sind. Für Softwareentwickler ohne Erfahrungen im Frontenddesign ist dies ein unschätzbarer Vorteil. Der Mockup wurde danach in eine Xtend-Klasse transformiert, die das Interface `IArtifactGenerator` implementierte und später in der *plugin.xml* an den Extension Point gebunden wurde. Im Anschluss an die Entwicklung des Code-Generators folgte die Implementierung der Property Page. Mit dieser können für den Generator folgende Einstellungen gesetzt werden:

- Das Ausgabeverzeichnis lässt sich unterhalb des Ordners *src-gen* definieren.
- Es existiert die Option, dass transiente Domains nicht in den Mockup übernommen werden, d.h. sie werden in der Navigation nicht angezeigt.
- Der Style für die zu generierende DSL konnte ausgewählt werden.

Dank der Erfahrungen, die mit der Property Page des Plug-ins zur Aktivierung der Artefakt-Generatoren gemacht worden sind, konnte die Implementierung ebenfalls schnell umgesetzt werden. Die Vorgehensweise hat sich dabei als sehr eingängig erwiesen und besitzt eine klare Struktur.

Die schwierigste Aufgabe dieses Generators bestand darin, die Auswahl der Styles zu implementieren. Styles sollen dazu dienen, dass ein Mockup auf die Bedürfnisse von Kunden zugeschnitten ist, mit denen man öfter zusammenarbeitet. Ein Webdesigner kann z.B. ein Style namens "NeosIT"definieren, der für alle Projekte des Unternehmens genutzt werden kann. Nach einigen Überlegungen wurde folgende Implementierung vorgenommen:

- Im Ordner *resource/assets* befinden sich alle Dateien, die unabhängig vom gewählten Style immer mit rekursiv in das Ausgabeverzeichnis kopiert werden.
- Alle Ordner unterhalb von *resource/styles* können innerhalb der Property Page ausgewählt werden. Beim Speichern wird der Inhalt des Unterordners rekursiv in das Ausgabeverzeichnis kopiert. Bestehende Dateien könnten damit überschrieben werden.
- Während des Generierungsprozesses werden zuerst die Dateien kopiert. Der Artefaktgenerator überprüft nach dem Kopiervorgang, welche Dateien in welchem Ordner vorhanden sind und generiert HTML-Tags für die JavaScript- und CSS-Dateien automatisch.

Zu beachten war, dass ein Plug-in in aller Regel als ZIP-Datei vorliegt. Das rekursive Kopieren von Verzeichnen aus diesem komprimierten Archiv in ein Projektverzeichnis wurde im Interface `IJarResourceProvider` definiert und in `JarResourceProvider` implementiert.

Als Feature für die Mocking-Ansicht wurde mit Hilfe der JavaScript-Klassen *jQuery Visualize*[32] und *handsontable*[33] eine editierbare Chart View eingebaut. Anhand der in der DSL hinterlegten Beispieldaten über das Schlüsselwort `examples` wird eine Tabelle hinterlegt, die sich in der generierten Ansicht editieren lässt. Das erzeugte Chart wird dabei sofort aktualisiert. Dieses Feature ist für die Entwickler interessant, da die NeosIT in erster Linie Management-Reporting-Systeme entwickelt, bei denen Charts eine wichtige Rolle spielen.

[32] http://filamentgroup.com/lab/update_to_jquery_visualize_accessible_charts_with_html5_from_designing_with/
[33] http://handsontable.com/

Folgende DSL-Definition ergibt die in der Grafik 5.8 hinterlegte, editierbare Tabelle mit Chart:

```
domain Bericht {
    attr wert1: integer {
        examples: ["2", "5", "7"]
    }
    attr wert2: integer {
        examples: ["10", "9", "2"]
    }
    attr wert3: integer {
        examples: ["4", "2", "9"]
    }

    process viewGesamtSumme {
        label: "Gesamtsumme aller Werte im Bericht"
        view: statistic [
            self.wert1,
            self.wert2,
            self.wert3
        ]
    }
}
```

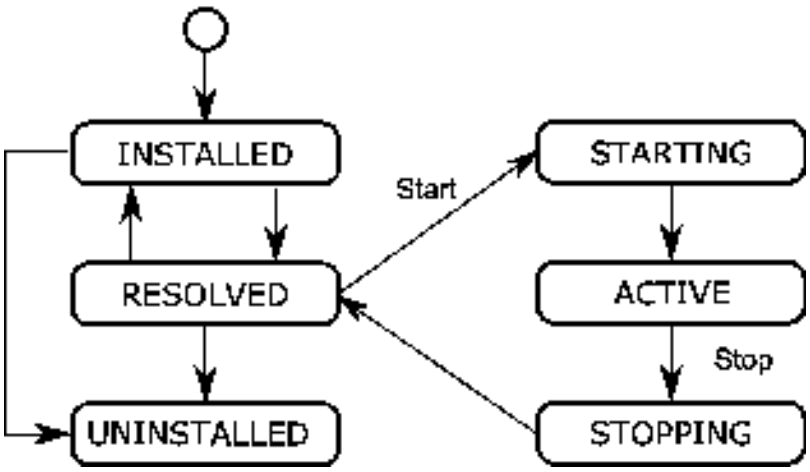

Abbildung 5.8: Editierbares Chart innerhalb des Mockups

5.4.2 Function Point-Analyse

Mit der Fertigstellung des Mocking-Generators konnten viele gewonnene Erkenntnisse und Best Practices direkt in den zweiten Generator zum automatischen Erstellen einer Function Point-Analyse (FPA) einfließen.

Am Anfang stand die Entscheidung, in welchem Format die FPA erstellt werden sollte. Die einfachste Möglichkeit bestand in der Erstellung einer CSV-Datei, die alle berechneten Werte enthielt. Allerdings wäre die Nachvollziehbarkeit der Berechnung nicht mehr gegeben, da nur noch die Ergebnisse exportiert

werden würden. Beim Export der kompletten Datensätze ist es wiederum nötig, dass eine spezielle Excel-Datei existiert, die mit Hilfe von Makros die Analyse auf Basis der CSV-Datei durchführt. Eine zweite Möglichkeit wäre die direkte Generierung einer Excel-Datei mit Hilfe des Frameworks Apache POI[34] gewesen. Aus Zeitgründen wurde davon abgesehen und schließlich wurde mit Hilfe von JavaScript/jQuery eine kleinere Webanwendung erstellt, die die Analyse anhand der Ausgangsdaten automatisiert durchführt und anzeigt. Weiterhin wurde eine Funktionalität implementiert, mit der man die Einflussfaktoren dynamisch ändern und sich die Ergebnisse ansehen kann.

Die reine Analyse der DSL erfolgt durch ein einfaches Iterieren über die Elemente der DSL. Zuerst werden alle Domänen analysiert:

- Domänen, die in der DSL mit *@ExternalService* annotiert worden sind, werden als External Logical File (ELF) angesehen. Alle anderen Domänen hingegen als Internal Logical File (ILF).
- Feldgruppen, die mit `fieldgroup` innerhalb von Domänen definiert worden sind, werden als Referenced Element Types (RET) beziehungsweise File Types Reference (FTR) gezählt.
- Attribute und `has-many`-Referenzen innerhalb einer Domäne werden als Data Element Type (DET) angesehen.

Im zweiten Schritt werden alle Prozesse innerhalb von Domänen überprüft:

- Je nach `view` wird der Prozess als Inquiry, Input oder Output gewertet. Über Annotationen [35] kann der Typ des Functionpoints geändert werden.
- Für jede `view` können die anzuzeigenden Attribute definiert werden. Werden keine Attribute spezifiziert, werden alle Attribute angezeigt. Jedes anzuzeigende Attribut wird als DET gezählt.

Aus diesen Informationen wird eine einfache HTML-Tabelle mit den ungewichteten Function Points generiert. Zusätzlich werden die für das Projekt definierten Komplexitätskriterien ausgelesen und in einer weiteren Tabelle hinterlegt. Die Berechnung der gewichteten Function Points anhand der Komplexitätskriterien erfolgt durch einfaches JavaScript, die beispielhafte Ausgabe für die Forumsapplikation aus dem Diagramm 5.1 ist in der Grafik 5.10 zu sehen. Die Grafik 5.9 stellt die Nutzung des Generators für die Function Point-Analyse noch einmal dar.

5.5 Annotationen

Da die Grammatik von Rapid fest definiert ist, können Entwickler die DSL nicht einfach um eigene Funktionalitäten erweitern. Beispielsweise entstand während der ersten Testphase die Anforderung, dass man bei der Definition der Prozesse hinterlegen kann, um welchen Functiontype (Inquiry, Input, Output) es sich handelt. Der Generator zur Function Point-Analyse sollte damit beeinflusst werden. Der erste Ansatz bestand darin, innerhalb der Rapid-Grammatik ein neues Element namens `functiontype` zu hinterlegen. Allerdings entstehen damit mehrere

[34] http://poi.apache.org/
[35] siehe 5.4.2

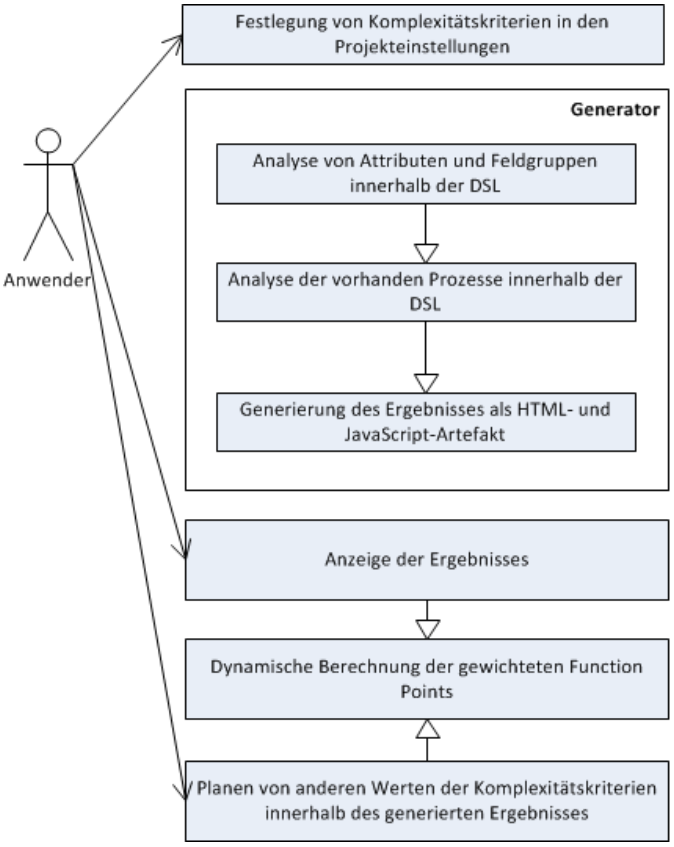

Abbildung 5.9: Nutzung der Function Point-Analyse

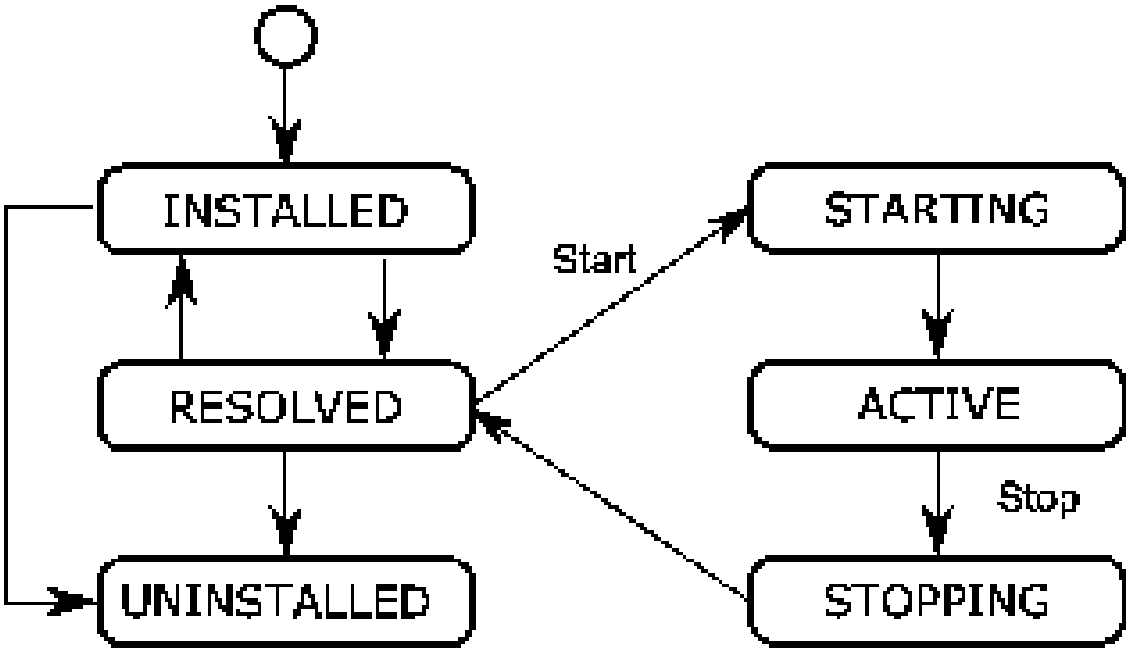

Abbildung 5.10: Ausgabe der Function Point-Analyse für die Forumsapplikation

Probleme: Bei jeder Spracherweiterung muss die Kern-Grammatik angepasst werden. Dies setzt voraus, dass der Entwickler Xtext und ANTLR kennt. Außerdem

wird die Grammatik durch diese Erweiterungen verschmutzt, da Rapid nur auf die Modellierung des Datenmodells und der Prozesse ausgelegt ist und über spezifische Eigenheiten der Generatoren keine Kenntnis haben soll. In Java und C# werden Metainformationen, wie es z.B. der Functiontype ist, in so genannten Annotationen [**?**] oder Attributen [**?**] hinterlegt. Xtext bietet die Möglichkeit, dass die eigene Grammatik auf Sprachelemente der JVM zugreifen kann. Dieses Feature wurde genutzt, um in Rapid auf Java basierte Annotationen zu integrieren. Entwickler können nun innerhalb ihres Generatores eigene Annotationen in Java definieren, die in der DSL und den Generatoren zur Verfügung stehen. Rapid wird somit von außen erweitert, ohne dass der Kern von diesen Erweiterungen Kenntnis nimmt. Die Grammatik wurde folgendermaßen umdefiniert:

```
grammar de.ckl.rapid.Rapid with
    org.eclipse.xtext.xbase.annotations.XbaseWithAnnotations

generate rapid"http://www.schakko.de/Rapid"

import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://www.eclipse.org/Xtext/Xbase/XAnnotations"
// ...
Domain:
    (annotations+=XAnnotation)*
    'domain' name=ID '{'
    // ...
    '}';

Attribute:
    (annotations+=XAnnotation)*
    'attr' name=ID ':' dataType=(OptionableDataType)
    // ...
;

Process:
    (annotations+=XAnnotation)*
    'process' name=ID ('{'
    // ...
;
```

Mit Hilfe der ersten Zeile `...XBaseWithAnnotations` wird das Element `XAnnoation` verfügbar gemacht. Jede Domäne, Attribut, Prozess oder Referenz kann mit beliebig vielen Annotationen ausgezeichnet werden. Innerhalb der `GenerateRapid.mwe2` musste das Backtracking in ANTLR aktiviert werden, da sonst zwischen den Annotationen innerhalb einer Domäne nicht hätte unterschieden werden können:

```
    fragment = parser.antlr.XtextAntlrGeneratorFragment {
        options = {
            backtrack = true
        }
    }
```

Für das Scoping wurde ein zusätzlicher Provider erstellt, der von der Klasse `XbaseWithAnnotationsScopeProvider`[36] erbt. Dieser Provider unterstützt das

[36] org.eclipse.xtext.xbase.annotations.scoping.XbaseWithAnnotationsScopeProvider

Scoping für JVM-Elemente, wie es innerhalb der JDT verfügbar ist und bekommt einen weiteren Provider injiziert, der die deklarative Definition der Scopes aus Kapitel 5.2 enthält. Dieses Vorgehen war nötig, da der `XbaseWithAnnotationsScopeProvider` keine deklarativen Scopes unterstützt. Sobald kein passender JVM-Typ gefunden wird, delegiert dieser an den deklarativen Scope-Provider weiter. Über einen zusätzlichen Validator wird definiert, welche Annotationen innerhalb der DSL genutzt werden dürfen. Jede zusätzliche Annotation muss dabei mit mindestens einem der Annotationen `@Attribute`, `@Domain`, `@HasManyReference` oder `Process` ausgezeichnet sein. Innerhalb der DSL kann über

```
@UseFunctionpointType(FunctionpointType::EXTERNAL_INPUT)
process mein_prozess {
    view: edit
}
```

der Typ spezifiziert werden. Der Generator überprüft während des Iterierens über das Modell, ob die Annotation vorhanden ist und reagiert dementsprechend darauf.

6 Fazit

Mit dieser Bachelorthesis wurde der Grundstein für die Integration von Xtext in den bestehenden Softwareentwicklungsprozess bei der NeosIT GmbH gelegt. Die definierte Grammatik der DSL namens Rapid erlaubt es auf einfache Art und Weise Kundenanforderungen zu beschreiben. Die Grammatik ist dabei auf technische Belange ausgerichtet und wird zur Generierung von beliebigen Artefakten benutzt werden. Die von Xtext generierten Plug-ins wurden dabei so erweitert, dass ein weiterer Extension Point in Eclipse bereitgestellt wurde. Die beiden erstellten Generatoren nutzen diesen Extension Point um ihre Artefakte zu generieren. Einerseits erstellt der Generator für das Mocking eine beispielhafte Anwendungsoberfläche, andererseits unterstützt der Generator zur Function Point-Analyse aktiv den Planungsprozess.
Damit die DSL von den Anwender effizient benutzt werden kann, wurden die von Eclipse bzw. Xtext bereitgestellten Features wie z.B. Outline Views, Validierung und Scoping implementiert. Um Abhängigkeiten zwischen den Generatoren und der DSL zu eliminieren, wurde der Support für Annotationen in die DSL und die Scoping Provider integriert. Diese Entscheidung trug besonders dazu bei, dass in der nachfolgenden Betaphase neue Funktionaliäten recht einfach umgesetzt werden konnten.
Die Vorstellung der DSL und den damit einhergehenden Möglichkeiten wurde von den Mitarbeitern sehr interessiert aufgenommen. Nach einer kurzen Einführung wurden die bisherigen Projekte innerhalb weniger Stunden in die Sprache von Rapid transformiert. Die erstellten Function Point-Analysen flossen zum direkten Vergleich in das unternehmensinterne Wiki ein. Die Ergebnisse dienen als Grundlage für zukünftige Planungsentscheidungen. Weiterhin entstanden von den Kollegen innerhalb von kurzer Zeit weitere Generatoren, um beispielsweise anhand der DSL die Schemadefinition für NHibernate zu erzeugen oder mit Spring Roo automatisch Applikationen zu generieren. Die in dieser Arbeit entstandenen Generatoren dienten dabei als Vorlage und Dokumentation. Während der Implementierung neuer Generatoren wurden aufgetretene Fehler und Verbesserungsvorschläge direkt umgesetzt. Weiterhin wurde durch die in dieser Arbeit gemachten Erfahrungen mit Xtext vom Autor innerhalb von zwei Stunden eine weitere DSL entwickelt, mit der sich einfache Expertensysteme abbilden lassen. Diese DSL wird bereits ebenfalls produktiv eingesetzt.

Mit der Einarbeitung in Xtext haben sich viele neue Möglichkeiten zur Verbesserung und Automatisierung aufgetan. Mit Rapid ist ein stabiles und einfach erweiterbares Framework entstanden. Das Ökosystem von Generatoren rund um die DSL unterstützt bereits jetzt erfolgreich den Softwareentwicklungsprozess der NeosIT.

7 Ausblick

In Kapitel 3.4 wurden bereits einige Ideen für Artefaktgeneratoren genannt. Die momentane Ausrichtung der NeosIT auf die Entwicklung in C# macht es nötig, dass der sich bereits im Einsatz befindende Xtext-C#-Generator in Rapid abgebildet wird. Hier wäre eine Wizard-Erweiterung für Eclipse sinnvoll, mit der sich das Grundgerüst von neuen Artefaktgeneratoren für Rapid recht einfach erstellen ließ. Diese Idee kam während der Umsetzung des Generators für die Function Point Analyse, da sich die Struktur der Generatoren doch ähnelt aber nicht weiter abstrahiert werden kann. Auch zu überlegen ist die Erzeugung von aufbereiteten POJOs, die von Apache Isis[37] weiterverarbeitet werden können. Isis ist ein Framework zum Generieren von Prototypen mit verschiedenen Ausgabeformaten, z.B. XML und HTML. Als Basis kommen dabei Java-Technologien zum Einsatz. Rapid könnte mit relativ wenig Aufwand Code für Isis erzeugen, der sofort lauffähig wäre.
Ein wichtiger Punkt ist die Integration von Rapid in den Buildserver. Das Rapid Plug-in und die Generatoren werden momentan noch manuell innerhalb von Eclipse erzeugt. Hier soll in Zukunft der Build automatisiert durch TeamCity erfolgen. Die Plug-ins sollen als Eclipse Update Site zur Verfügung stehen.

[37] Isis steht unter Apache License 2.0 und kann unter http://isis.apache.org heruntergeladen werden.

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat. Ich erkläre mich damit einverstanden/nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Datum, Unterschrift

Abkürzungsverzeichnis

API Application Programming Interface

CRUD Create, Read, Update, Delete

CSS Cascading Stylesheet

CSV Comma-separated values

DAL Data Access Layer

DAO Data Access Object

DBMS Datenbank Management System

DDL Data Definition Language

DET Data Element Type

DSL Domain Specific Language

ELF External Logical File

EMF Eclipse Modeling Framework

FTR File Type Reference

FPA Function Point-Analyse

HTML Hypertext Markup Language

IDE Integrated Development Environment

ILF Internal Logical File

JDT Java Development Toolkit

JPA Java Persistence API

JVM Java Virtual Machine

MDA Model Driven Architecture

MWE Model Workflow Engine

PDE Plug-in Development Environment

PDT PHP Development Toolkit

POJO Plain Old Java Object

RET Record Element Type

SCM Source Code Management

SWT Standard Widget Toolkit

TDD Test Driven Development

UML Unified Modeling Language

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Annotation

Annotationen dienen dazu, Metadaten innerhalb einer Programmiersprache oder DSL zu hinterlegen.

Artefakt

Ein Artefakt stellt im Rahmen dieser Arbeit ein oder mehre Quellcodedateien dar, die automatisiert erzeugt worden sind.

Artefakt-Generator

Ein Plug-in zur automatisierten Erstellung von Artefakten bzw. Quellcode. Der Generator nutzt dabei das Modell der DSL als Basis.

Domäne

Als Domäne wird der Bereich bezeichnet, in dem eine domänenspezifische Sprache eingesetzt wird.

Eclipse

Eine Entwicklungsumgebung für Java und andere Programmiersprachen

Function Point-Analyse

Methodik, die zur Aufwandsabschätzung von Softwareprojekten angewandt werden kann.

Lambda-Ausdruck

Ein Lambda-Ausdruck stellt eine anonyme Funktion dar, die an eine Methode übergeben werden kann.

Mockup

Beispielhafte Darstellung einer Anwendung ohne oder mit wenig Funktionalität.

Modell

Das Modell bildet mit Hilfe der DSL die Anforderungen einer Domäne in einer textuellen Form ab.

transient

Ein Element (Domäne, Attribut o.ä.), das zur Laufzeit nicht in einer Datenbank persistiert wird.

Literaturverzeichnis

[All12] OSGi Alliance. OSGi Alliance specification. Spezifikation für OSGi Release 5, 2012. URL: http://www.osgi.org/Specifications/HomePage.

[Art04] John Arthorne. Project builders and natures. 2004. URL: http://www.eclipse.org/articles/Article-Builders/builders.html.

[Bal09] Helmut Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 2. Auflage, 2009, Seiten 529–539.

[BCD+12] Ryan Bigg, Fredreick Cheung, Tore Darell, Jeff Dean, Mike Gunderloy, and Mikel Lindsaar. Getting started with rails. Getting Up and Running Quickly with Scaffolding, 2012. URL: http://guides.rubyonrails.org/getting_started.html#getting-up-and-running-quickly-with-scaffolding.

[Cer05] Gary Cernosek. A brief history of Eclipse. 2005. URL: http://www.ibm.com/developerworks/rational/library/nov05/cernosek/.

[Cle10] Torsten Cleff. *Basiswissen Testen von Software*. W3L-Verlag, 1. Auflage, 2010, Seiten 62–63.

[Cor09] Microsoft Corporation. The Microsoft code name M Modeling Language Specification - Introduction. 2009. URL: http://msdn.microsoft.com/en-us/library/dd285271.aspx.

[Cornta] Microsoft Corporation. Attributes (C# and Visual Basic). unbekannt. URL: http://msdn.microsoft.com/de-de/library/vstudio/z0w1kczw.aspx.

[Corntb] Microsoft Corporation. Quadrant Overview. unbekannt. URL: http://msdn.microsoft.com/en-us/library/dd857506(VS.85).aspx.

[Corntc] Microsoft Corporation. Visualization and Modeling SDK - Domain-Specific Languages. unbekannt. URL: http://msdn.microsoft.com/en-us/library/bb126259.aspx.

[Ecl07] Eclipse. The project description file. 2007. URL: http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fmisc%2Fproject_description_file.html.

[ED96] Christof Ebert and Reiner Dumke. *Software-Metriken in der Praxis*. Springer Verlag, 1. Auflage, 1996, Seite 122.

[Eff10] Sven Efftinge. Xbase - A new programming language? 2010. URL: http://blog.efftinge.de/2010/09/xbase-new-programming-language.html.

[Eff12] Sven Efftinge. Xtend documentation. 2012. URL: http://www.eclipse.org/xtend/documentation.html.

[Eff13] Sven Efftinge. Xtext documentation. 2013. URL: http://www.eclipse.org/Xtext/documentation.html.

[HT06] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley Professional, 19. Edition, 2006, Seite 103.

[iA08] itemis AG. Xtext.xtext. 2008. URL: https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext.

[iA10a] itemis AG. Xbase.xtext. 2010. URL: https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtext.xbase/src/org/eclipse/xtext/xbase/Xbase.xtext.

[iA10b] itemis AG. Xtend.xtext. 2010. URL: https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtend.core/src/org/eclipse/xtend/core/Xtend.xtext.

[Mic07a] Sun Microsystems. Java 5 API. @ManyToMany Annotation, 2007. URL: http://docs.oracle.com/javaee/5/api/javax/persistence/ManyToMany.html.

[Mic07b] Sun Microsystems. Java 5 API. @ManyToOne Annotation, 2007. URL: http://docs.oracle.com/javaee/5/api/javax/persistence/OneToMany.html.

[Moi12] Kim Moir. The Architecture of Open Source Applications - Eclipse. 2012. URL: http://www.aosabook.org/en/eclipse.html.

[Ora10a] Oracle. Annotations. 2010. URL: http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

[Ora10b] Oracle. Core j2ee patterns - data access object. 2010. URL: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html.

[Vö10] Markus Völter. Modellgetriebene, Komponentbasierte Softwareentwicklung. *JavaMagazin*, 2010. Online unter http://www.voelter.de/data/articles/MDSDandCBD-Part1.pdf.