Institut für Informatik
der Technischen Universität München

Introspektive modellgetriebene Softwareentwicklung

Thomas Büchner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. B. Brügge, Ph. D.
Prüfer der Dissertation:

1. Univ.-Prof. Dr. F. Matthes
2. Univ.-Prof. Dr. Dr. h.c. M. Broy

Die Dissertation wurde am 21.08.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.10.2007 angenommen.

Zusammenfassung

In dieser Arbeit wird ein alternativer Ansatz zur modellgetriebenen Softwareentwicklung vorgestellt, der die Integration von Modellen mit dem darunter liegenden System zum Ziel hat. Dieser Ansatz wird introspektive modellgetriebene Softwareentwicklung genannt und beruht auf der Entwicklung von introspektiven Frameworks, deren Erweiterungsmöglichkeiten direkt im Quellcode annotiert sind. Eine introspektive Analyse der Erweiterungsmöglichkeiten liefert das Metamodell des Frameworks. Darauf aufbauend ist es möglich, eine integrierte domänenspezifische Sprache zu realisieren.

Es werden zwei Arten von introspektiven Frameworks vorgestellt. Introspektive Blackbox Frameworks realisieren domänenspezifische Sprachen durch externe Modellrepräsentationen, die jedoch jederzeit mit den Erweiterungsmöglichkeiten des Frameworks integriert sind.

In introspektiven Whitebox Frameworks werden Modelle intern durch Quellcode der Basisprogrammiersprache repräsentiert. Durch Introspektion lassen sich diese Modelle extrahieren sowie auf einem hohen Abstraktionsniveau darstellen und bearbeiten. Aufgrund der internen Repräsentation sind Modelle introspektiver Whitebox Frameworks mit dem darunter liegenden System integriert.

Es wurden Werkzeuge entwickelt, die das Erstellen und Modellieren von introspektiven Frameworks ermöglichen. Diese Werkzeuge basieren auf der Programmiersprache Java und sind als Plugins in die Entwicklungsumgebung Eclipse integriert. Für beide Arten von Frameworks werden Metametamodelle vorgestellt, auf deren Basis introspektive Frameworks mit integrierter domänenspezifischer Sprache effizient entwickelt werden können.

Um den Ansatz zu evaluieren, wurde die introspektive Webplattform Toro entwickelt, die aus introspektiven Whitebox Frameworks besteht. Ein Bestandteil dieser Frameworks ist jeweils eine maßgeschneiderte domänenspezifische Sprache, die verständliche Modellierungssichten auf die zu entwickelnde Webanwendung ermöglicht. Es werden Frameworks mit Modellperspektiven zur Lokalisierung von multilingualen Nachrichten, zur Datenmodellierung und zur Spezifikation von Webinteraktion und Webvisualisierung vorgestellt. Ein bestehendes industrielles Wissensmanagementsystem (100000 Lines-of-Code) wurde auf Toro portiert, um die praktische Anwendbarkeit zu prüfen. Die introspektiven Modellierungssichten erhöhen die Verständlichkeit des Systems und vereinfachen die Komplexitätsbeherrschung. Da die Modelle mit dem System integriert sind, ist die Konsistenz des Gesamtsystems gewährleistet.

Danksagung

An erster Stelle möchte ich mich bei meinem Doktorvater Herrn Prof. Matthes und den Mitarbeitern seines Lehrstuhls für das angenehme Arbeitsumfeld, in dem diese Arbeit entstand, bedanken. Insbesondere der Gedankenaustausch mit Prof. Matthes hat meine Arbeit kreativ beflügelt und zu vielen in der Arbeit verwirklichten Ideen geführt. Mein Dank gilt auch Herrn Prof. Broy, der sich bereit erklärte, das Zweitgutachten für meine Arbeit zu erstellen.

Mein ganz besonderer Dank gilt jedoch meiner Familie und meiner Freundin Susan für die Unterstützung, die sie mir in der Phase der Erstellung der Dissertation gegenüber aufgebracht haben. Ohne ihren Rückhalt wäre diese Arbeit nicht möglich gewesen.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Modelle in der Softwareentwicklung

Die Entwicklung von Software-Systemen ist mittlerweile eine Ingenieursdisziplin, in der aufgrund der großen Komplexität der zu entwickelnden Systeme Modelle eine immer größere Rolle spielen. Im Folgenden wird ein grundsätzlicher Überblick über die Verwendung von Modellen im Software-Engineering gegeben.

Modelle bieten die Möglichkeit, durch das Ausblenden irrelevanter Details die Verständlichkeit von Systemen zu erhöhen. Durch Modelle mit klar definierter Syntax und Semantik ist es möglich, verständlichere Sichten auf Systeme zu erlangen und ihre Komplexität zu beherrschen.

Die Verwendung von Modellen im Software-Engineering kann dahin gehend unterschieden werden, ob die Modelle lediglich unterstützende Artefakte sind, deren Erstellung nicht direkt Einfluss auf das zu erstellende System hat, oder ob Modelle konstruktiv bei der Realisierung von Systemen genutzt werden. Die unterstützende Verwendung wird im Folgenden *modellbasierte* Softwareentwicklung genannt. Hierbei werden Modelle einerseits in frühen Stadien des Entwicklungsprozesses eingesetzt, um bestimmte Aspekte eines Systems zu spezifizieren. Andererseits werden Modelle in modellbasierten Verfahren zur Dokumentation von Softwaresystemen erstellt. Mit unterstützender Nutzung ist gemeint, dass die erstellten Modelle keinen direkten Einfluss auf das abgebildete Softwaresystem haben. Es existiert keine explizite Abbildung zwischen Modellen und Implementierungsartefakten. Es gibt eine große Zahl von etablierten Modellierungssprachen, die zur modellbasierten Softwareentwicklung in den verschiedenen Phasen des Entwicklungsprozesses zum Einsatz gebracht werden können [Ob05a].

Im Gegensatz zur unterstützenden Nutzung wird bei aktiver Modellnutzung das System direkt durch die erstellten Modelle beeinflusst. Dieser Ansatz wird im Folgenden *modellgetriebene* Softwareentwicklung genannt. Hierbei sind die erstellten Modelle Implementierungsartefakte von der gleichen Wichtigkeit wie der Quellcode einer klassischen Programmiersprache. Die in modellgetriebenen Ansätzen genutzten Modellierungssprachen sind meist so genannte domänenspezifische Sprachen (engl.: Domain Specific Language – DSL) [DKV00]. Dabei handelt es sich um für eine bestimmte Anwendungsdomäne maßgeschneiderte Abstraktionen von eingeschränkter Ausdrucksmächtigkeit, durch die die intellektuelle Distanz zwischen Systemspezifikation und tatsächlicher Problemstellung verringert werden soll. Dies ist ein grundlegendes Prinzip des Software-Engineering [Fa85 S. 3] und hat sowohl eine Erhöhung der Produktivität als auch der Qualität zum Ziel.

Modellbasierte und modellgetriebene Softwareentwicklung schließen sich nicht gegenseitig aus, sondern können komplementär in unterschiedlichen Phasen eines Entwicklungsprozesses zum Einsatz kommen. Der Schwerpunkt dieser Arbeit liegt auf der Nutzung von Modellen in

modellgetriebenen Softwareentwicklungsprozessen und den speziellen Anforderungen an eine Umsetzung dieses Ansatzes.

1.2 Integration von Modellen

Im Speziellen motiviert sich diese Arbeit aus der Forderung nach domänenspezifischen Sprachen, die mit dem darunter liegenden System *integriert* sind. Das darunter liegende System ist im Allgemeinen ein Framework, welches in der Basisprogrammiersprache programmiert ist. Durch Modelle domänenspezifischer Sprachen ist es möglich, Frameworks auf einem hohen Abstraktionsniveau anzupassen. Es besteht eine enge Verbindung zwischen Erweiterungsmöglichkeiten des Frameworks und Konzepten der domänenspezifischen Sprache. Etablierte Ansätze zur modellgetriebenen Softwareentwicklung realisieren domänenspezifische Abstraktionen ohne direkte Verbindung zum instrumentalisierten Framework und somit ohne Integration von Modellen und Framework-Kern (Details hierzu siehe Abschnitte 4.8 und 5.6.3). Dadurch ist es schwer, die Konsistenz des Gesamtsystems sicherzustellen. Weiterentwicklungen müssen stets parallel sowohl an der domänenspezifischen Sprache als auch am Framework-Kern vorgenommen werden. Aufgrund der fehlenden Integration ist eine Unterstützung durch Werkzeuge bei diesen Prozessen nicht möglich.

Weitere, über diesen elementaren Integrationsaspekt hinausgehende Anforderungen ergeben sich aus folgendem Szenario modellgetriebener Softwareentwicklung: In vielen Fällen wird ein Softwaresystem durch mehrere domänenspezifische Sprachen sowie durch Quellcode einer Basisprogrammiersprache spezifiziert.

Mehrere DSLs sind im Allgemeinen notwendig, da Softwaresysteme oft verschiedene Aspekte umfassen, die sich durch unterschiedliche Konzepte beschreiben lassen. Beispielsweise besteht ein Aspekt einer Anwendung durch ein Datenmodell, welches die verwalteten Geschäftsobjekte verzeichnet. Ein anderer Aspekt besteht in der Präsentation der Daten, ein weiterer Aspekt beschreibt die Geschäftslogik der Anwendung. Jeder Aspekt kann durch eigene domänenspezifische Termini beschrieben werden.

Quellcode der Basisprogrammiersprache ist zur Systementwicklung weiterhin erforderlich, da domänenspezifische Sprachen typischerweise über eine eingeschränkte Ausdrucksmächtigkeit verfügen. Dies ist einerseits eine gewollte Stärke von DSLs, da hiervon die Verständlichkeit der Modelle profitiert. Andererseits lassen sich bestimmte Aspekte von Softwaresystemen nur unter Benutzung einer Turing-vollständigen Programmiersprache spezifizieren.

In diesem Szenario, welches auf eine große Klasse von Softwaresystemen zutrifft, besteht der Bedarf, alle beteiligten Artefakte miteinander zu integrieren, um die Konsistenz des Gesamtsystems sicherzustellen. Existierende Ansätze zur modellgetriebenen Softwareentwicklung ermöglichen diese weitergehenden Integrationsanforderungen ebenfalls nicht.

1.3 Ziele und Lösungsansatz der Arbeit

In dieser Arbeit wird ein alternativer Ansatz zur modellgetriebenen Softwareentwicklung vorgestellt, dessen Ziel darin besteht, die geschilderten Integrationsanforderungen umzusetzen.

Die Grundidee des vorgeschlagenen Ansatzes besteht darin, domänenspezifische Sprachen direkt aus den instrumentalisierten Frameworks zu extrahieren. Dies wird erreicht, indem die Erweiterungsmöglichkeiten des zu modellierenden Frameworks explizit durch Annotationen im Framework-Kern gekennzeichnet werden. Ein derartig annotiertes Framework wird *introspektives Framework* genannt und ermöglicht die Extraktion des Metamodells der vom Framework definierten domänenspezifischen Sprache durch Metamodell-Introspektion. Der vorgeschlagene Ansatz wird *introspektive* modellgetriebene Softwareentwicklung genannt (engl.: Introspective Model-Driven Development – IMDD). Im Unterschied zu existierenden Ansätzen zur modellgetriebenen Softwareentwicklung zeigt die Haupt-Abbildungsrichtung von detaillierteren Implementierungsartefakten zu abstrakteren Modellen und ist somit umgekehrt. Dadurch sind die extrahierten Modelle stets mit dem darunter liegenden System integriert, und eine Realisierung der vorgestellten Integrationsanforderungen ist möglich. Des Weiteren ermöglicht der vorgeschlagene Ansatz die Umsetzung von Roundtrip-Engineering.

Ein weiteres Ziel introspektiver modellgetriebener Softwareentwicklung besteht darin, die Entwicklung von domänenspezifischen Sprachen basierend auf Frameworks zu vereinfachen. Im einfachsten Fall ist es hierfür lediglich notwendig, die Erweiterungsmöglichkeiten des Frameworks zu annotieren.

Zusätzlich zur konzeptuellen Idee wird in dieser Arbeit eine konkrete Realisierung von IMDD erstellt, die die Konstruktion von introspektiven Frameworks und introspektiven domänenspezifischen Sprachen ermöglicht. Als Basisprogrammiersprache wurde Java gewählt. Die Realisierung von IMDD basiert auf der Eclipse IDE, einer weit verbreiteten integrierten Entwicklungsumgebung, welche eine mächtige Basis für introspektive Analysen darstellt. Die entwickelten Werkzeuge sind als Plugins in Eclipse integriert und fügen sich somit nahtlos in bestehende Entwicklungsprozesse ein.

Um eine Bewertung der Kosten und Nutzen introspektiver modellgetriebener Softwareentwicklung durchführen zu können, soll der vorgeschlagene Ansatz in realistischen Szenarien angewandt werden. Ein weiteres Ziel dieser Arbeit besteht somit in der Entwicklung von introspektiven Frameworks und darauf basierenden Anwendungen.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die grundlegenden Abstraktionsebenen vorgestellt, auf denen IMDD basiert. Dies sind objektorientierte Programmiersprachen und Frameworks. Des Weiteren werden die konzeptuellen Grundlagen modellgetriebener Softwareentwicklung detailliert vorgestellt.

In Kapitel 3 wird der Ansatz der introspektiven modellgetriebenen Softwareentwicklung vorgestellt. Es gibt zwei Arten von introspektiven Frameworks: introspektive Blackbox und Whitebox Frameworks. Im folgenden Kapitel 4 wird zunächst die Realisierung von introspektiven Blackbox Frameworks und anschließend in Kapitel 5 von introspektiven Whitebox Frameworks besprochen.

In Kapitel 6 wird die in dieser Arbeit entwickelte Realisierung von IMDD vorgestellt, welche auf Basis der Programmiersprache Java und der Entwicklungsumgebung Eclipse entwickelt wurde.

Zur Validation des vorgeschlagenen Ansatzes wurde eine vollständig introspektive Plattform zur Entwicklung von Webanwendungen entwickelt, welche in Kapitel 7 vorgestellt wird. Diese Plattform besteht aus verschiedenen introspektiven Frameworks, welche in diesem Kapitel detailliert besprochen werden. Eine Bewertung der durch Introspektion erreichten Integration wird anhand von Anwendungen vorgenommen, die auf Basis der introspektiven Plattform erstellt wurden.

Eine Zusammenfassung des Erreichten und ein Ausblick auf neu entstandene Fragestellungen bilden den Schluss in Kapitel 8.

2 Grundlagen

In diesem Kapitel wird der Ansatz der modellgetriebenen Softwareentwicklung motiviert und vorgestellt. Die Notwendigkeit modellgetriebener Verfahren ergibt sich aus der Komplexität moderner Informationssysteme. Zum Beherrschen der Komplexität hat sich eine Reihe von aufeinander aufbauenden Abstraktionen etabliert. In Abschnitt 2.1 werden objektorientierte Programmiersprachen als Basisabstraktion vorgestellt. Aufbauend auf objektorientierten Programmiersprachen heben Frameworks das Abstraktionsniveau, indem sie Problemlösung durch Anpassen an definierten Erweiterungspunkten ermöglichen (siehe Abschnitt 2.2).

Ein gravierender Nachteil objektorientierter Frameworks besteht darin, dass das Anpassen auf dem niedrigen Abstraktionsniveau der objektorientierten Programmiersprache stattfindet. Obendrein sind die vom Framework zur Verfügung gestellten Anpassungsmöglichkeiten nicht explizit erkennbar. Als Konsequenz daraus sind Frameworks schwer verständlich und erlernbar. Diese Nachteile werden durch den Ansatz der modellgetriebenen Softwareentwicklung adressiert, welcher das Konzept der domänenspezifischen Sprachen als Kernidee in sich trägt. Die Anpassungsmöglichkeiten eines Frameworks werden hierbei explizit durch eine domänenspezifische Sprache spezifiziert. Dies ermöglicht Anpassen auf einem hohen Abstraktionsniveau. In Abschnitt 2.3 werden zunächst die Kernkonzepte modellgetriebener Softwareentwicklung vorgestellt. Anschließend werden Realisierungsmöglichkeiten sowie Anforderungen an eine Realisierung modellgetriebener Softwareentwicklung vorgestellt.

2.1 Programmiersprachen als Basisabstraktion

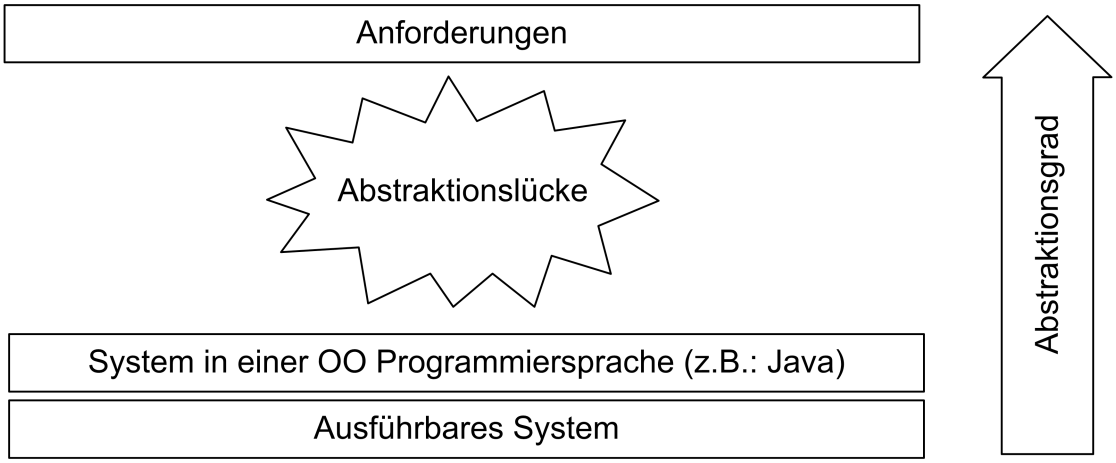


Abbildung 1: Eine objektorientierte Programmiersprache als Basisabstraktion

Die Basisabstraktion bei der Entwicklung einer betrieblichen Anwendung ist die Programmiersprache, in der das System entwickelt wird. Aufgrund der Vielfältigkeit der Anforderungen an

ein betriebliches Informationssystem handelt es sich dabei um so genannte *General Purpose Languages* (GPLs). Am weitesten verbreitetet sind heutzutage objektorientierte Programmiersprachen wie zum Beispiel Java und C#. Dabei handelt es sich um Programmiersprachen der dritten Generation, die bereits Details des ausführbaren Systems vor dem Entwickler verbergen und somit einen Beitrag zur Überwindung der Abstraktionslücke leisten (siehe Abbildung 1). Eine Programmiersprache ist eine formale Sprache und lässt sich dementsprechend beschreiben.

Exkurs: Anatomie einer formalen Sprache

Eine formale Sprache wird durch die folgenden Dinge definiert [Gr04]:

1. *Syntax:*
   - Die *abstrakte Syntax* beschreibt die Konstrukte der Sprache und die Beziehungen zwischen ihnen.
   - Die *konkrete Syntax* ist von Menschen bearbeitbar und stellt die Schnittstelle zum Benutzer der Sprache dar.
   - Die *Serialisierungssyntax* dient zum Persistieren der Programme sowie als Austauschformat zwischen Tools.
2. *Semantik*: Die Semantik einer Sprache beschreibt die Bedeutung der in der Sprache ausgedrückten Programme.

Abstrakte Syntax	Java ist eine kontextfreie Grammatik, die in Backus-Naur-Form in [Go05] spezifiziert ist.
Konkrete Syntax	Java-Programme werden in textueller Form erstellt und bearbeitet.
Serialisierungssyntax	Die erstellten textuellen Programme sind gleichzeitig sowohl konkrete Syntax als auch Serialisierungssyntax.
Semantik	Informelle Beschreibung in [Go05].

Tabelle 1: Sprachanatomie von Java

Die Anatomie der Programmiersprache Java stellt sich beispielsweise wie in Tabelle 1 dargestellt dar. Es gibt verschiedene Ansätze, die abstrakte Syntax einer formalen Sprache zu definieren:

- kontextfreie Grammatiken (werden im Allgemeinen in Backus-Naur-Form [Ba60] dargestellt),
- Metamodelle.

Kontextfreie Grammatiken sind zur Definition der abstrakten Syntax textueller Sprachen besser geeignet, während die Beschreibung durch Metamodelle für graphische Sprachen vorteilhaft ist. Eine eingehendere Abwägung der Vor- und Nachteile von kontextfreien Grammatiken und Metamodellen ist in [Gr04] zu finden.

Ebenso existieren mehrere Möglichkeiten, die konkrete Syntax einer formalen Modellierungssprache zu realisieren. Für einen Benutzer der Modellierungssprache stellt dies einen wesentlichen Aspekt dar. Die am weitesten verbreiteten Arten der Darstellung einer Modellierungssprache sind in textueller sowie in graphischer Form [CL04]. Eine weitere Art besteht in der Darstellung in einer Baumstruktur (siehe [Si95]).

Beispielsweise wird wie in Tabelle 2 dargestellt, die abstrakte Syntax der graphischen Modellierungssprache „UML-Klassendiagramm“ durch Metamodelle definiert.

Abstrakte Syntax	Die abstrakte Syntax von UML-Klassendiagrammen wird durch ein Metamodell, welches in [Ob05a] spezifiziert ist beschrieben.
Konkrete Syntax	Klassendiagramme werden zur Bearbeitung graphisch visualisiert. Es existiert eine Vielzahl von Editoren.
Serialisierungssyntax	Zum Austausch von UML-Diagrammen zwischen Tools verschiedener Hersteller existiert der Standard XMI [Ob05b].
Semantik	Die Semantik von UML-Klassendiagrammen ist in [Ob05a] größtenteils informell spezifiziert. Einige wenige Teile sind mittels OCL [Ob06] formal beschrieben.

Tabelle 2: Sprachanatomie von UML-Klassendiagrammen

2.1.1 Entwurf objektorientierter Systeme

Das grundlegende Abstraktionsmittel objektorientierter Programmiersprachen ist die Definition von Klassen. Klassen beschreiben Eigenschaften (Attribute) und Verhalten (Methoden) von Objekten. Das Konzept der Vererbung ermöglicht die Definition neuer Klassen genannt Subklassen auf der Basis existierender Klassen, genannt Superklassen. Für eine weiterführende Einführung in objektorientierte Programmiersprachen sei auf [Se05] verwiesen.

Der Prozess des Entwurfs objektorientierter Systeme wird in die Phasen *objektorientierte Analyse* (OOA), *objektorientiertes Design* (OOD) und *objektorientiertes Programmieren* (OOP) unterteilt [Bo94]. Aufgabe der objektorientierten Analyse ist die Analyse der Anforderungen aus der Perspektive von Klassen und Objekten. Darauf aufbauend wird ein objektorientiertes Design erstellt, welches schließlich als Basis für eine objektorientierte Implementierung dient. Somit besteht das Ziel der Methodik in der Abbildung der Anforderungen an das System auf geeignete Klassenhierarchien und somit eine objektorientierte Implementierung. Die diesem Ansatz zugrunde liegende Annahme besteht darin, dass sich die zu modellierenden Dinge relativ nahtlos auf Klassen abbilden lassen.

Objektorientierte Programmiersprachen sind eine mächtige Basisabstraktion zum Erstellen komplexer Systeme. Die Leistungsfähigkeit einer konkreten Programmiersprache hängt hierbei nicht nur von der Anatomie der Sprache ab. Weitere wichtige Faktoren sind vorhandene Bibliotheken, Werkzeuge sowie weiche Faktoren wie die Verfügbarkeit gut ausgebildeter Programmierer.

Das wichtigste Werkzeug bei der Erstellung objektorientierter Systeme ist die integrierte Entwicklungsumgebung (Integrated Development Environment, IDE). Eine neuartige und für diese Arbeit wichtige Art der Entwicklungsumgebung ist die der Post-IntelliJ-IDEs.

2.1.2 Post-IntelliJ-IDEs

Der Terminus Post-IntelliJ-IDE wurde von Martin Fowler geprägt [Fo04b]. Er bezeichnet damit alle integrierten Entwicklungsumgebungen, die bereits während der Bearbeitung des Programms eine interne Repräsentation erzeugen. Mit Hilfe dieser internen Repräsentation – im Allgemeinen ein abstrakter Syntaxbaum – ist die IDE in der Lage, den Quelltext tiefer zu erschließen und zugänglich zu machen. Dies ist im besonderen Maße für *statisch typisierte* Programmiersprachen möglich. In dieser Klasse von Programmiersprachen sind Typinformationen bereits zur Kompilierzeit vorhanden, und ihre Konsistenz wird ebenfalls zur Kompilierzeit überprüft. Aufgrund der statischen Verfügbarkeit der Typinformationen sind Post-IntelliJ-IDEs in der Lage, folgende Funktionalitäten zur Verfügung zu stellen:

- Navigation ermöglicht das Wandern entlang von Strukturen. Beispielsweise ist es möglich, von einem Methodenaufruf direkt zum Ort der Deklaration der Methoden zu springen.
- Code Assist ist die Unterstützung bei der automatischen Vervollständigung bestimmter Konstrukte. Da der Typ von Variablen bekannt ist, können alle vorhandenen Methoden und Felder vorgeschlagen werden.
- Suche nach Vorkommnissen ermöglicht die Suche nach der Verwendung von Klassen, Methoden und Attributen.
- Refactoring ermöglicht die Strukturverbesserung von Programmcode unter Beibehaltung des Verhaltens [Fo99]. Aufgrund der vorhandenen Typinformationen ist es in einer statisch typisierten Programmiersprache relativ einfach möglich, Refactoring automatisch zu unterstützen.

In Abbildung 2 sind die vorgestellten Funktionalitäten in einem Screenshot der Post-IntelliJ-IDE *Eclipse* dargestellt [Ec06b]. Im oberen Bereich befindet sich der Quelltext der bearbeiteten Klasse. Über das Kontextmenü ist es möglich, zur Deklaration der Methode `getAssociatedAsset` zu navigieren. Durch Code Assist werden mittleren Bereich der Abbildung drei Methoden vorgeschlagen, die zur bisherigen Eingabe passen. Im unteren Bereich sind im Plugin *Search* alle Referenzen auf die Methode `updateStatistics` aufgelistet. Soll die Methode umbenannt werden, so ist dies über ein Refactoring möglich. Dabei werden alle Referenzen automatisch angepasst.

Die Bezeichnung Post-IntelliJ-IDE ergibt sich daraus, dass *IntelliJ* [Je06] die erste verfügbare Java-IDE mit den genannten Funktionalitäten war. Bereits vor IntelliJ verfügte Microsofts Entwicklungsumgebung Visual Studio über ähnliche Funktionalitäten, die unter dem Namen *IntelliSense* zusammengefasst werden. Mittlerweile existiert eine Reihe von Entwicklungsumgebungen, die die genannten Eigenschaften aufweisen. Die genannten Vorteile haben zu einer breiten Verwendung dieser Art der Entwicklungsumgebung und ihrer Funktionalitäten geführt [MKF06].

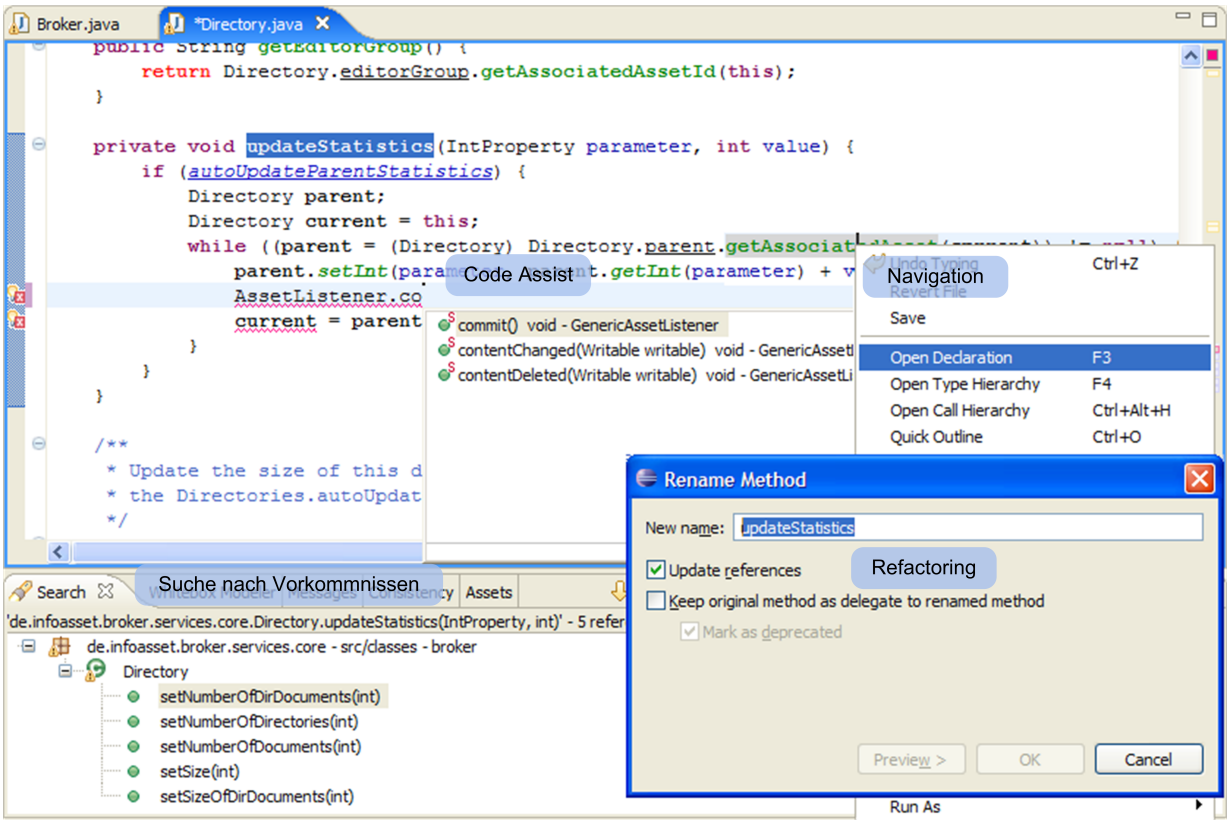


Abbildung 2: Post-IntelliJ-IDE

Für diese Arbeit von besonderer Relevanz ist die Kerneigenschaft von Post-IntelliJ-IDEs – die Verfügbarkeit des abstrakten Syntaxbaums zur Bearbeitungszeit. Aufgrund der Verbreitung, der freien Lizenzierung und der modularen Architektur wurde Eclipse als Basis für diese Arbeit ausgewählt.

2.1.3 Imperative und deklarative Programmierstile

Laut [We89] existieren zwei Programmierparadigmen: deklarative und imperative Programmierung. Imperative Programmierung spezifiziert die Art und Weise, wie eine Berechnung durchgeführt werden soll. Dies geschieht durch eine Reihe von Operationen, die den Zustand von Variablen explizit verändern. Dementsprechend verfügen imperative Programmiersprachen über Konzepte wie Variablen, Zuweisungen, Schleifen. Imperatives Programmieren spezifiziert den Weg, der zum Finden der Lösung einzuschlagen ist. Die Lösung an sich wird nicht spezifiziert.

Das imperative Paradigma basiert auf der vorherrschenden von Neumann Prozessorarchitektur. Der Grund für die weite Verbreitung imperativer Programmiersprachen ist zunächst darin zu finden, dass aufgrund der konzeptuellen Nähe eine Abbildung auf ausführbare Konstrukte vergleichsweise einfach ist. Allerdings ist auch anzumerken, dass der imperative Programmierstil von vielen Programmierern mittlerweile als natürliche Art der Programmierung angesehen

wird. Die Aussage, dass imperative Programmierung nur aufgrund technischer Unzulänglichkeiten existenzfähig ist (siehe [Ba78]) wird nur von einer kleinen Gruppe geteilt.

Im deklarativen Programmierparadigma wird die gewünschte Lösung spezifiziert, ohne den Weg, die Lösung zu finden, anzugeben. Beispielsweise spezifiziert die deklarative Anfragesprache SQL [Da87] die Eigenschaften der gewünschten Resultate. Die Aufgabe des Findens der Ergebnisse wird dem Datenbanksystem überlassen. Somit befinden sich deklarative Programme auf einem höheren Abstraktionsniveau, da sie Details der Ausführung verbergen. Programmiersprachen, die das deklarative Paradigma unterstützen lassen sich wiederum in funktionale und logische Programmiersprachen unterteilen.

Im Kontext dieser Arbeit wird mit *deklarativem Programmieren* der Programmierstil gemeint, der gewünschte Eigenschaften einer Lösung beschreibt und vom konkreten Finden der Lösung abstrahiert. Die Anwendung eines deklarativen Programmierstils ist auch mit den Mitteln einer imperativen Programmiersprache möglich:

```
new Criteria(Project.class)
  .addEq("name", "Testprojekt")
    .addLike("description", "interessant")
      .execute();
```

In diesem Beispiel wird deklarativ eine Anfrage mit zwei Kriterien an ein System beschrieben. Die Art und Weise der Ausführung ist der Implementierung überlassen.

Zusammenfassend dienen objektorientierte Programmiersprachen und die mit ihnen verbundenen Plattformen als mächtige Basis für weitere darauf aufbauende Abstraktionsschichten. Leider ist das Abstraktionsniveau objektorientierter Programme aufgrund des zugrunde liegenden imperativen Paradigmas relativ niedrig.

Es soll darauf hingewiesen werden, dass die Mächtigkeit objektorientierter Programmiersprachen keineswegs als lediglich störend angemerkt wird. In vielen Fällen ist diese Mächtigkeit erforderlich, um ein komplexes Problem lösen zu können. Allerdings ist ein unreflektierter Umgang mit der zur Verfügung stehenden Mächtigkeit in vielen Fällen hinderlich und verringert die Verständlichkeit der Programme [PBK07]. In diesen Fällen, in denen die volle Ausdrucksstärke einer objektorientierten Programmiersprache nicht erforderlich ist, stellt sich die Frage nach Möglichkeiten, den Abstraktionsgrad anzuheben.

2.2 Frameworks

Objektorientierte Analyse und Design ermöglicht die Entwicklung komplexer Systeme, die Probleme in der realen Welt lösen. Es gibt Klassen von Problemen, die sich grundsätzlich ähneln und deren Lösungen sich an den immer gleichen Stellen voneinander unterscheiden. Anstatt für jedes Problem ein eigenes System zu entwickeln, muss das Ziel darin bestehen, die Gemeinsamkeiten auszunutzen und lediglich die Unterschiede zu adressieren. Ein solches "Rahmen-System", welches sich an das jeweilige Problem anpassen lässt, wird Framework genannt:

> *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems (, and supports reuses at a larger granularity than classes).* [JF88]

Ein Framework löst eine Klasse von Problemen. Um ein konkretes Problem zu lösen, ist eine Anpassung des Frameworks notwendig. Ein gut entworfenes Framework zeichnet sich einerseits dadurch aus, dass eine große Menge des vorhandenen Designs wieder verwendet werden kann. Andererseits muss es möglich sein, das Framework flexibel an unterschiedliche Probleme anzupassen.

2.2.1 Arten von Frameworks

Es gibt zwei Arten von Frameworks, die sich in der Art und Weise in der sie an konkrete Problemstellungen angepasst werden, unterscheiden [Pr97b]. Blackbox Frameworks basieren auf fertig implementierten Klassen, die zur Problemlösung lediglich konfiguriert werden müssen. Wie der Begriff „Blackbox“ bereits impliziert, sind zur Anpassung keine detaillierten Kenntnisse über Implementierungsdetails der existierenden Klassen notwendig. Im Unterschied dazu werden Whitebox Frameworks durch Implementieren von speziellen dafür vorgesehenen Methoden angepasst. Hierbei wird Vererbung als grundlegendes Anpassungsmittel eingesetzt.

Die Punkte, an denen Anpassung erforderlich und möglich ist, werden für beide Arten von Framworks *Erweiterungspunkte* oder *Hot-Spots* genannt. Durch das Anpassen eines Frameworks wird das gewünschte Verhalten an den Erweiterungspunkten spezifiziert.

2.2.2 Lebenszyklus von Frameworks

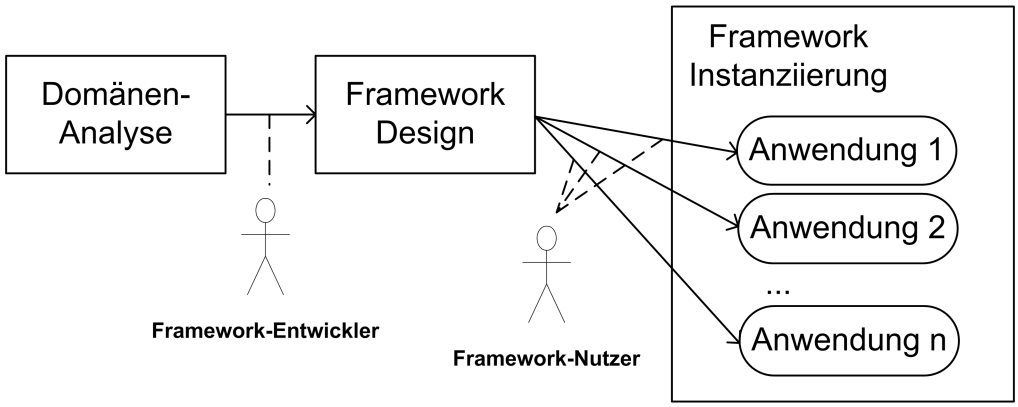


Abbildung 3: Prozess der Entwicklung und Anpassung eines Frameworks [ML01]

Der Lebenszyklus eines Frameworks besteht im Wesentlichen aus zwei Phasen, in denen unterschiedliche Entwicklergruppen involviert sind [Fr99] (siehe auch Abbildung 3):

- Erstellen und Weiterentwickeln des Frameworks durch *Framework-Entwickler*, sowie
- Anpassen des Frameworks zum Lösen konkreter Probleme durch *Framework-Nutzer*.

Die Unterscheidung, ob es sich um ein Blackbox oder Whitebox Framework handelt, hängt von der Art der Anpassung, die der Framework-Nutzer vornimmt, ab. Falls der Framework-Nutzer zur Anpassung lediglich aus vorhandenen Klassen auswählt und diese konfiguriert, handelt es sich um ein Blackbox Framework. Die Bezeichnung Blackbox drückt aus, dass zur Anpassung nicht zwangsläufig genaue Kenntnisse des Designs des Frameworks notwendig sind. Muss der Framework-Nutzer zur Anpassung an ein konkretes Problem jedoch eigene Subklassen implementieren, so handelt es sich um ein Whitebox Framework. In diesem Fall sind im Allgemeinen genauere Kenntnisse des Designs des Frameworks notwendig. Die Benutzung eines Frameworks ohne vorherige Anpassung ist nicht möglich.

2.2.3 Bestandteile eines angepassten Frameworks

Die Bestandteile einer unter Anpassung eines Frameworks erstellten Anwendung lassen sich wie in Abbildung 4 dargestellt unterteilen. Der *Framework-Kern* besteht aus den Klassen des Frameworks, welche nicht durch Anpassung verändert werden. Besondere Bedeutung kommt den Erweiterungspunkten zu, die Teil des Framework-Kerns sind und in der Abbildung durch die Umrandung visualisiert werden. Diese sind für den Framework-Nutzer relevant, da sie die Anpassungsmöglichkeiten vorgeben. Im Falle eines Whitebox Frameworks sind dies abstrakte Klassen, die in der Phase der Anpassung vom Framework-Nutzer implementiert werden müssen. Im Falle eines Blackbox Frameworks sind dies vollständig implementierte Klassen, die lediglich instanziiert und konfiguriert werden müssen.

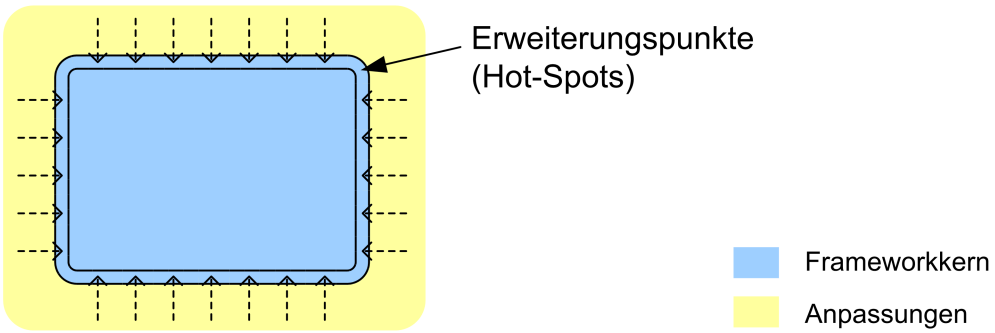


Abbildung 4: Bestandteile eines angepassten Frameworks

Der mit *Anpassungen* bezeichnete Teil umfasst Quellcode, der das Framework an eine konkrete Aufgabenstellung anpasst. Dies können im Falle eines Whitebox Frameworks konkrete Klassen sein, die von Klassen des Framework-Kerns abgeleitet sind. Im Falle eines Blackbox Frameworks ist dies Quellcode, der die Instanziierung des Systems steuert. Wie in der Abbildung dargestellt, hängen Anpassungsartefakte jeweils von bestimmten Erweiterungspunkten ab.

Der Quellcode zur Anpassung eines Blackbox Frameworks instanziiert dafür vorgesehene Klassen des Framework-Kerns, konfiguriert diese und setzt sie in Beziehung zueinander. Die Erweiterungspunkte eines Blackbox Frameworks sind also spezielle zur Konfiguration zu instanziierende Klassen des Framework-Kerns, deren Eigenschaften und Verknüpfungsmöglichkeiten.

Quellcode zur Anpassung eines Whitebox Frameworks erweitert existierende abstrakte Klassen des Framework-Kerns und implementiert bestimmte Methoden. Diese abstrakten Klassen und die zu implementierenden Methoden stellen die Erweiterungspunkte eines Whitebox Frameworks dar.

2.2.4 Blackbox vs. Whitebox Frameworks

Es stellt sich die Frage, für welche Problemklassen Blackbox bzw. Whitebox Frameworks besser geeignet sind. Die beiden Arten von Frameworks unterscheiden sich in der Art und Weise, wie Anpassung an konkrete Probleme stattfindet. Im Fall eines Blackbox Frameworks findet Anpassen durch Auswählen und Konfigurieren statt. Dies setzt einerseits voraus, dass für jedes denkbare konkrete Problem adäquate Implementierungen vorhanden sind und dass diese Implementierungen entsprechend konfigurierbar sind. Andererseits entspricht das Anpassen durch Auswählen und Konfigurieren einem deklarativen Programmierstil. Durch Auswählen und Konfigurieren werden Eigenschaften der Lösung spezifiziert. Die bereitgestellte Implementierung spezifiziert die Art und Weise, wie die Lösung konkret zur Ausführung umgesetzt wird. Lässt sich die Problemdomäne also durch Bereitstellen von einer absehbaren Anzahl verschiedener Implementierungen abdecken *und* ist es weiterhin möglich, durch deklaratives Konfigurieren an das konkrete Problem anzupassen, so ist ein Blackbox Framework die geeignete Wahl.

Ist dies nicht gegeben, so lässt sich die Problemklasse nur mit einem Whitebox Framework abdecken. In diesem Fall findet die Anpassung durch Implementieren von Subklassen und somit Methoden statt. Innerhalb dieser Methoden kann der Framework-Nutzer die volle Mächtigkeit der imperativen objektorientierten Programmiersprache ausnutzen. Nachteilig an dieser Lösung ist, dass dies die Anforderungen an den Framework-Nutzer erhöht.

2.2.5 Vor- und Nachteile von Frameworks

Die Verwendung von Frameworks bringt die folgenden Vorteile mit sich:

1. Konzeptueller Abstraktionsgrad: Das Anpassen eines gut entworfenen Frameworks verlangt Entscheidungen auf einem der Problemdomäne angemessenen Abstraktionsgrad. Diese Aussage bezieht sich auf die konzeptuelle Struktur der Erweiterungspunkte und ist somit analog zur abstrakten Syntax einer formalen Sprache (siehe Seite 22) zu verstehen.
2. Wiederverwendungsgrad: Frameworks ermöglichen Wiederverwendung von sowohl Implementierung als auch Design [FSJ99 S. 8].
3. Wartungsaufwand: Aufgrund der Wiederverwendung verringert sich der Wartungsaufwand für Elemente des Framework-Kerns, da nur eine Codebasis gepflegt werden muss.
4. Qualität: Eine weitere Folge der Wiederverwendung besteht darin, dass die Qualität der geteilten Codebasis mit der Anzahl der Verwendung durch Anwendungen anwächst [Fr99].

Den genannten Vorteilen objektorientierter Frameworks stehen die folgenden Nachteile gegenüber:

1. Aufwändig: Das Erstellen eines Frameworks ist ein aufwändiger Prozess. Aufgrund der gewünschten Wiederverwendbarkeit müssen beim Entwurf eines Frameworks viele Dinge beachtet werden, die beim Erstellen einer konkreten Anwendung nicht relevant sind. Es ist also wesentlich teurer, ein Framework zu erstellen, als eine einzelne Anwendung. Die Entwicklung eines Frameworks lohnt sich nur, wenn eine Vielzahl von gleichgearteten Problemen gelöst werden sollen [FSJ99 S. 11].
2. Abhängigkeiten zwischen Anwendung und Framework: Die Kopplung zwischen Framework und abgeleiteten Anwendungen ist einerseits von Vorteil, da das Framework zentral gepflegt werden kann. Andererseits können durch eine Weiterentwicklung des Frameworks Abhängigkeiten zu existierenden Anwendungen betroffen sein und verletzt werden. Demzufolge lohnt sich die Entwicklung eines Frameworks in einer sich wenig verändernden Problemdomäne am meisten.

Die folgenden Nachteile ergeben sich direkt aus der Tatsache, dass sowohl der Kern eines objektorientierte Frameworks als auch die Anpassungen in einer GPL geschrieben sind und somit direkt als Quellcode der Basisprogrammiersprache vorliegen:

3. Schwer zu verstehen: Es existiert keine allgemein anerkannte Art und Weise der Dokumentation objektorientierter Frameworks [Bo99]. Das Verstehen eines komplexen Frameworks ist ein zeitaufwändiger Prozess. Insbesondere sind die Erweiterungspunkte eines Frameworks oft nicht einfach erkennbar. Vorgeschlagene Ansätze zur Kennzeichnung von Erweiterungspunkten in UML-Klassendiagrammen haben sich in der Praxis nicht durchgesetzt [FPR01].
4. Konkrete Anpassung auf niedrigem Abstraktionsniveau: Quellcode zur direkten Anpassungen von Frameworks befindet sich auf dem niedrigen Abstraktionsniveau der Basisprogrammiersprache. Der Zusammenhang zwischen den vorzunehmenden Anpassungen und den Konzepten der Problemdomäne ist hierbei oft nicht offensichtlich, da diese von der Syntax der Basisprogrammiersprache überschattet werden.
5. Unerwünschte Mächtigkeit der Basisprogrammiersprache: Im Allgemeinen ist zur Anpassung eines Frameworks eine Sprache erforderlich und wünschenswert, die wesentlich weniger ausdrucksstark als die verwendete Basisprogrammiersprache ist. Wird das Framework direkt durch Konstrukte der Basisprogrammiersprache angepasst, ist es jederzeit möglich auf deren volle Mächtigkeit zuzugreifen. Dies ist in vielen Fällen nicht erwünscht und verringert die Verständlichkeit.

Die drei letztgenannten Punkte adressieren Nachteile von Frameworks in Bezug auf das Schließen der Abstraktionslücke. Einerseits sind zum Anpassen eines gut entworfenen Frameworks Entscheidungen auf einem angemessenen Abstraktionsniveau erforderlich. Andererseits ist die konkrete Art und Weise, in der diese Entscheidungen manifestiert werden (durch Quellcode der Basisprogrammiersprache), auf einem niedrigen Abstraktionsniveau angesiedelt. Das Anpassen eines Frameworks hat viele Gemeinsamkeiten mit dem Programmieren in einer domänenspezifischen Programmiersprache [Ko01]. Unter Benutzung dieser Analogie verfügt die abstrakte Syntax (siehe Seite 22) eines gut entworfenen Frameworks über die für die Problemdomäne relevanten Konzepte. Die Kritikpunkte richten sich gegen die konkrete Syntax, mit der der Framework-Nutzer direkt in Berührung kommt.

2.2.6 Frameworks zur Entwicklung betrieblicher Informationssysteme

An betriebliche Informationssysteme wird eine Vielzahl von immer wiederkehrenden nicht-funktionalen Anforderungen gestellt. Es existiert eine Reihe von Frameworks, die bei der Erfüllung der Anforderungen helfen. Aufgrund des typischerweise dreischichtigen Aufbaus eines betrieblichen Informationssystems [Ec95] gibt es eine Vielzahl von Frameworks zur Realisierung der für die jeweilige Schicht relevanten nicht-funktionalen Anforderungen. Die Entwicklung eines betrieblichen Informationssystems besteht also im Wesentlichen aus den folgenden Schritten:

1. Auswahl von geeigneten Frameworks,
2. Anpassen der Frameworks und somit Lösen des Problems.

Das eigentliche Lösen des Problems findet also größtenteils durch Anpassen von Frameworks statt. Dementsprechend wichtig ist dieser Aspekt und steht im Fokus dieser Arbeit. Eine unter Benutzung von Frameworks entwickelte Anwendung besteht jeweils aus den beteiligten Frameworks sowie den vorgenommenen Anpassungen. Frameworks stellen eine Möglichkeit zur Verringerung der Abstraktionslücke dar. Durch die Benutzung eines Frameworks ist zur Problemlösung lediglich die Kenntnis der Erweiterungspunkte des Frameworks notwendig. Es werden Details der Implementierung vor dem Framework-Nutzer verborgen und somit wird das Abstraktionsniveau angehoben (siehe Abbildung 5).

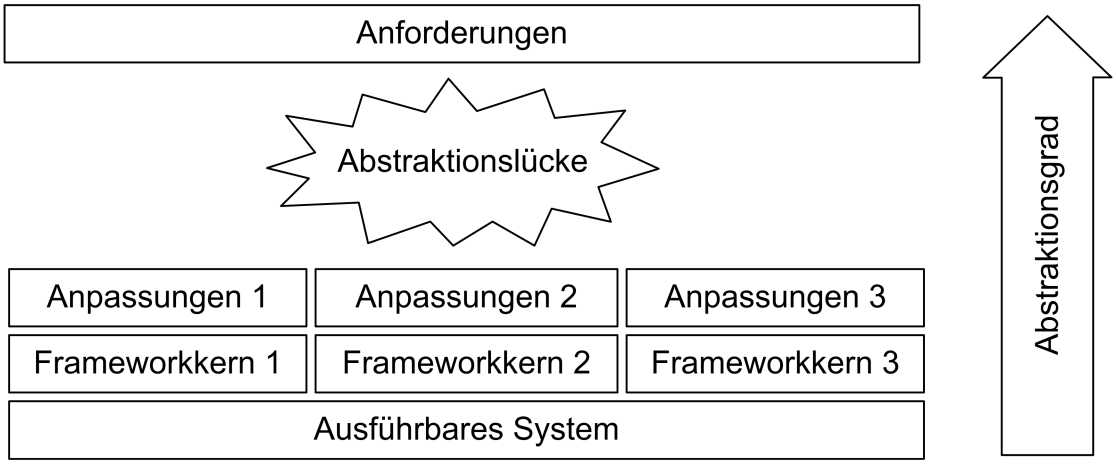


Abbildung 5: Frameworks zur Verringerung der Abstraktionslücke

Es stellt sich heraus, dass die drei letztgenannten Nachteile objektorientierter Frameworks ihre gravierendsten Nachteile sind. Zusammenfassend lässt sich sagen, dass objektorientierte Frameworks in ihrer bisherigen Form nicht geeignet sind, das Problem der Komplexität betrieblicher Informationssysteme vollständig zu lösen. Das Abstraktionsniveau, auf dem das konkrete Anpassen von Frameworks stattfindet, ist nicht der Problemdomäne angemessen.

2.3 Modellgetriebene Softwareentwicklung

Die Defizite von Frameworks bzgl. des Schließens der Abstraktionslücke werden durch den Ansatz der modellgetriebenen Softwareentwicklung (englisch: Model Driven Software Development – MDSD) adressiert. Das Ziel dieses Ansatzes ist das Überbrücken der Abstraktionslücke durch deklarative modellhafte Beschreibung.

Modellgetriebene Softwareentwicklung wird im Rahmen dieser Arbeit komplementär zu existierenden Frameworks gesehen und eingesetzt. Die verwendeten Modelle machen Anpassungen an den verwendeten Frameworks explizit und ermöglichen somit das Erstellen von Anwendungen auf einem hohen Abstraktionsniveau. Die zur Beschreibung der Anpassungen verwendeten maßgeschneiderten Modellierungssprachen werden domänenspezifische Sprachen (englisch: Domain Specific Languages – DSLs) genannt. Die Syntax einer DSL spiegelt die Erweiterungsmöglichkeiten eines Frameworks wider und es besteht eine direkte Beziehung zwischen einem Framework und einer domänenspezifischen Sprache (siehe Abbildung 6).

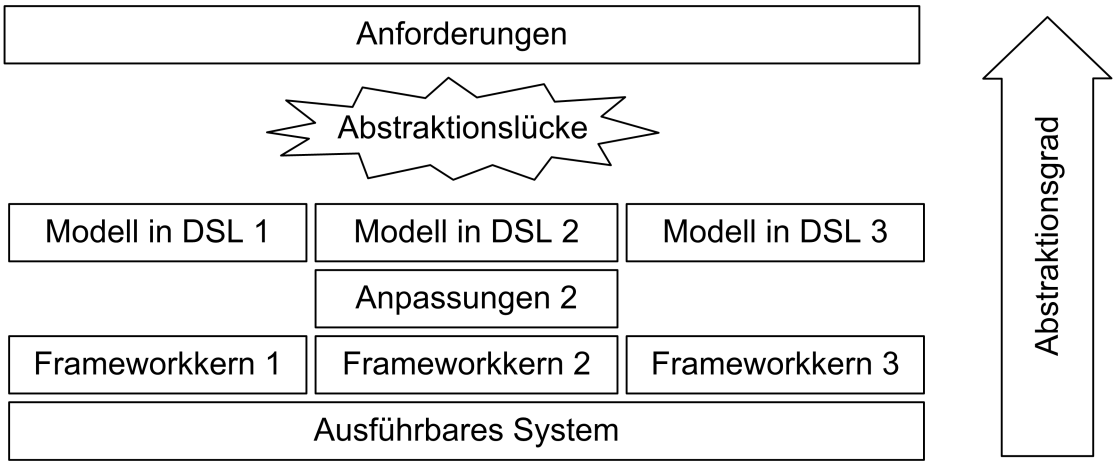


Abbildung 6: DSLs zur Verringerung der Abstraktionslücke

Das Erstellen maßgeschneiderter Modellierungssprachen passend zu den Anpassungsmöglichkeiten eines Frameworks ist somit ein wesentlicher Bestandteil modellgetriebener Softwareentwicklung.

Die Beschreibung durch Modelle soll dabei die Bereiche umfassen, in denen eine deklarative Anpassung sinnvoll ist. Dies umfasst laut [Gr04] circa 70% einer Anwendung. Zur Spezifikation der verbleibenden Bereiche wird die volle Ausdrucksstärke der verwendeten objektorientierten Programmiersprache benötigt. Das Ziel modellgetriebener Softwareentwicklung kann demzufolge realistischerweise nicht in einer vollständigen modellhaften Beschreibung bestehen, sondern muss vielmehr in einer Integration der verschiedenen Abstraktionsgrade bestehen. In Abbildung 6 ist dies durch das Vorhandensein von Anpassungsartefakten für Framework 2 dargestellt. Dieses Framework lässt sich also nur teilweise durch deklarative Modelle anpassen. Gewisse Anpassungen erfordern die volle Ausdrucksmächtigkeit der Basisprogrammiersprache.

Damit unterscheidet sich der Abstraktionsschritt durch domänenspezifische Sprachen von den bisher im Rahmen der Entwicklung von Programmiersprachen erreichten Anhebungen des Abstraktionsniveau dadurch, dass das eingeführte Abstraktionsniveau nicht die Konstrukte auf darunter liegenden Abstraktionsgraden vollständig transparent macht. Zum Einen sind die Framework-Kerne in der Basisprogrammiersprache programmiert, zum Anderen sind an bestimmten Stellen weiterhin Anpassungen in der Basisprogrammiersprache erforderlich.

In diesem Abschnitt werden die für das Verständnis modellgetriebener Softwareentwicklung zentralen Konzepte eingeführt, näher erläutert und in Beziehung gesetzt. Hervorzuheben sei hier, dass der in diesem Abschnitt vorgestellte Ansatz keine Aussagen über die konkrete Umsetzung enthält. Dies ist insofern wichtig, als eine Unterscheidung zwischen Konzept und Umsetzung nicht immer eindeutig durchgeführt wird. So wird modellgetriebene Softwareentwicklung oft mit dem weit verbreiteten Ansatz *generativer* modellgetriebener Softwareentwicklung gleichgesetzt (siehe Abschnitt 2.3). Im Rahmen dieser Arbeit wird *introspektive* modellgetriebene Softwareentwicklung als alternative Realisierungsmöglichkeit modellgetriebener Softwareentwicklung vorgeschlagen, so dass eine klare Unterscheidung zwischen Konzept und Realisierung für das Verständnis von entscheidender Bedeutung ist.

2.3.1 Modelle

Gemäß [St73] verfügen Modelle über die drei folgenden Hauptmerkmale:

1. Abbildungsmerkmal: *Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können*. Besonders sei hierbei auf die Transitivität des Modellbegriffs hingewiesen.
2. Verkürzungsmerkmal: Modelle erfassen im Allgemeinen nur bestimmte für die Modellerschaffer relevante Merkmale des repräsentierten Originals und abstrahieren somit von für das Modell irrelevanten Eigenschaften.
3. Pragmatisches Merkmal: Modelle werden zu einem bestimmten Zweck erstellt.

Ein Modell beschreibt also wesentliche Eigenschaften eines Gegenstandes der Betrachtung. Modelle bestehen aus Modellelementen. Aufgrund des Verkürzungsmerkmals von Modellen ist jedes Modell eine Abstraktion des abgebildeten Originals. Durch Modellbildung lassen sich also verständlichere Sichten auf komplexe Systeme erzeugen, die für eine bestimmte Betrachtungsweise irrelevante Details weglassen. Modelle sind geeignet, die Abstraktionslücke zwischen Anforderungen und ausführbarem System zu schließen.

2.3.2 Domänenspezifische Sprachen

Eine domänenspezifische Sprache (DSL – Domain Specific Language) ist eine formale Sprache (siehe Exkurs Seite 22), die auf die Probleme einer bestimmten Domäne zugeschnitten ist [DKV00]. DSLs sind zumeist von deklarativem Charakter und verfügen nicht über die volle Ausdrucksmächtigkeit einer Turing-vollständigen General Purpose Language. Domänenspezifische Sprachen können in Bezug auf die Basisprogrammiersprache entweder intern oder ex-

tern realisiert werden. Eine eingehendere Diskussion dieses Aspektes erfolgt in Abschnitt 2.3.10.

Ein Paradigma, in dem domänenspezifische Sprachen von großer Bedeutung sind, ist das *Language Oriented Programming* – LOP [Wa94, Di04]. Hierbei werden komplexe Softwaresysteme durch das Zusammenspiel von domänenspezifischen Sprachen beschrieben. Der Entwicklungsprozess besteht dabei in einem Schritt des Sprachdesigns und einem Schritt der Spezifikation des Systems unter Benutzung der entworfenen Sprachen.

Exkurs: Modellieren vs. Programmieren

Traditionell wird zwischen Programmiersprachen und Modellierungssprachen unterschieden. Diese Unterscheidung ist jedoch historisch bedingt [CL04, Gr04] und legt lediglich unterschiedliche Schwerpunkte auf letztendlich äquivalente Konzepte. Sowohl Programmier- als auch Modellierungssprachen verfügen über Syntax sowie Semantik und weisen alle Merkmale der auf Seite 22 vorgestellten Anatomie einer formalen Sprache auf. Im weiteren Verlauf dieser Arbeit werden beide Begriffe als synonym betrachtet.

Somit sind die Begriffe „modellieren“ und „programmieren“, sowie „Modell“ und „Programm“ ebenfalls synonym. Ein Programm einer Programmiersprache weist bei genauerer Betrachtung alle Hauptmerkmale des vorgestellten Modellbegriffs auf. Die Pragmatik eines Programms besteht in der Steuerung der Ausführung eines Computers. Das Verkürzungsmerkmal besteht darin, dass von bestimmten im Moment der Ausführung relevanten Aspekten abstrahiert wird, um eine Bearbeitung auf einem höheren Abstraktionsniveau zu ermöglichen. Die Abbildung auf die für die Ausführung benötigten Dinge wird letztendlich durch einen Compiler oder Interpreter durchgeführt.

2.3.3 Modellbasierte vs. modellgetriebene Prozesse

In [SV05] wird zwischen *modellbasierten* und *modellgetriebenen* Entwicklungsprozessen unterschieden. Danach dienen in modellbasierten Prozessen Modelle zur passiven Dokumentation von Softwaresystemen. Es existiert in diesem Fall lediglich eine gedankliche Verbindung zwischen Modellen und der abgebildeten Implementierung. So werden in der objektorientierten Analyse und Design (siehe Seite 23) in frühen Phasen Modellierungssprachen eingesetzt. Die spätere Abbildung auf Implementierungskonstrukte erfolgt jedoch im Allgemeinen händisch und die Beziehung zwischen Modell und Implementierung geht verloren. Modell und Implementierung werden sozusagen entkoppelt. Um den Vorteil des erhöhten Abstraktionsniveaus auch in der Implementierungsphase realisieren zu können, wäre eine händische Pflege der Modelle notwendig. Da dies mit viel manueller Arbeit verbunden ist, wird das Pflegen von Modellen häufig als Belastung empfunden und dementsprechend nicht durchgeführt.

Im Gegensatz dazu sind Modelle in modellgetriebenen Entwicklungsprozessen „First-Class Objects“ und es existiert eine explizite Abbildung zwischen Modellkonstrukten und Implementierung. Die sich aus der Verwendung von Modellen ergebenden Vorteile lassen sich in diesem Fall vollständig realisieren.

2.3.4 Ziele modellgetriebener Softwareentwicklung

Das vordergründigste Ziel modellgetriebener Softwareentwicklung ist die Verbesserung der Übersichtlichkeit und Verständlichkeit der entwickelten Systeme. Es soll die intellektuelle Distanz zwischen dem zu lösenden Problem und der tatsächlichen Implementierung so weit wie möglich überwunden werden. Hierbei handelt es sich um ein grundlegendes Anliegen des Software-Engineering (siehe [Fa85 S. 3]). Die Verwirklichung dieses Ziels wird durch das Verkürzungsmerkmal von Modellen erreicht, welches das Ausblenden irrelevanter Details beinhaltet und eine kompaktere und übersichtlichere Beschreibung von Systemen ermöglichen.

Dies ermöglicht es einerseits, komplexe Systeme schneller und somit mit geringeren Kosten zu entwickeln. Andererseits wirkt sich eine gesteigerte Verständlichkeit positiv auf die Qualität der entwickelten Software aus [ER03 S. 72]. Diese Effekte ergeben sich aus der Beschränkung des menschlichen Kurzzeitgedächtnisses auf sieben plus-minus zwei Elemente [Mi56]. Eine kompaktere Repräsentation ermöglicht es Entwicklern, größere Bereiche des zu erstellenden Systems zu bearbeiten. Eine weitere Erklärung für die positiven Effekte ergibt sich daraus, dass unter Benutzung einer weniger kompakten Repräsentation Details auf einem niedrigen Abstraktionsniveau manuell erstellt werden müssen. Dies ist zeitaufwändig und fehleranfällig. Durch modellgetriebene Softwareentwicklung wird dies automatisiert.

Eine weitere Phase im Entwicklungsprozess, in der sich eine durch Modelle gesteigerte Verständlichkeit betriebswirtschaftlich positiv auswirkt, ist die Wartungsphase. In dieser Phase werden Fehler behoben und das System wird an geänderte Umweltbedingungen und Anforderungen angepasst. In vielen Fällen werden diese Änderungen von Personen durchgeführt, die nicht an der Systementwicklung beteiligt waren. Der durch verständliche Modelle verringerte Einarbeitungsaufwand führt somit zu einer Verringerung der in dieser wichtigen Phase anfallenden Kosten.

Ein weiteres Ziel modellgetriebener Softwareentwicklung kann darin bestehen, die Problemlösung unabhängig von konkret existierenden Technologien zu realisieren. Die Modelle abstrahieren in diesem Fall von konkreten Technologien und ermöglichen die Wiederverwendung einer fachlichen Problemlösung unter Benutzung unterschiedlicher Basistechnologien. Dadurch werden Abhängigkeiten reduziert.

2.3.5 Metamodellierung

Ein Hauptbestandteil modellgetriebener Softwareentwicklung ist die Möglichkeit, maßgeschneiderte Modellierungssprachen zu definieren. Dies wird durch Metamodellierung erreicht. Wie in Abschnitt 2.3.1 dargestellt, können Modelle wiederum Repräsentationen von anderen Modellen sein. Wenn dies der Fall ist, spricht man von Metamodellen. Metamodelle beschreiben also wesentliche Eigenschaften von Modellen, die wiederum rekursiv Metamodelle für andere Modelle sein können. Alle Modelle, welche von einem Metamodell beschrieben werden, sind Instanzen des Metamodells. Dies äußert sich darin, dass alle Modellelemente des Modells Instanzen von Modellelementen des Metamodells sind. Zur Beschreibung von Modellierungssprachen wurden von der Object Management Group (OMG) vier Metaebenen definiert [Ob04b]:

- M3 – Meta-Metamodellebene: Auf dieser Ebene befinden sich Metametamodelle, die die zur Definition von Metamodellen notwendigen Konstrukte zur Verfügung stellen.
- M2 – Metamodellebene: Auf der Ebene M2 werden durch Metamodelle die in der Modellebene (M1) existierenden Modelle spezifiziert. Zum Beispiel werden auf dieser Ebene UML-Klassendiagramme spezifiziert.
- M1 – Modellebene: Auf dieser Ebene befinden sich die für den Benutzer von Modellierungssprachen relevanten konkreten Modelle, zum Beispiel ein UML-Klassendiagramm. Alle Modellelemente dieser Ebene sind Instanzen des dazugehörigen Metamodells von Ebene M2.
- M0 – Instanzebene: Auf dieser Ebene befinden sich die Laufzeitobjekte und somit Instanzen der jeweiligen Modelle.

In Abbildung 7 sind die vier Metaebenen anhand eines Beispiels dargestellt. Im Kontext modellgetriebener Softwareentwicklung ist Metamodellierung unter Anwendung von Metametamodellen wichtig, um effizient maßgeschneiderte Modellierungssprachen entwerfen zu können [GH05]. So lassen sich unter Verwendung eines gemeinsamen Metametamodells eine große Anzahl von Modellierungssprachen spezifizieren. Aufgrund des gemeinsamen Metametamodells existiert eine gemeinsame Methodik zur Definition aller für eine formale Sprache Aspekte relevanten Aspekte (siehe Seite 22). Dies vereinfacht das Erstellen maßgeschneiderter Modellierungssprachen wesentlich.

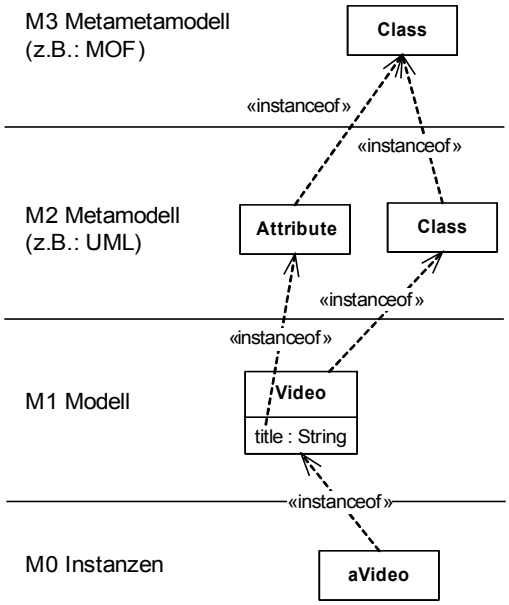


Abbildung 7: Vier Metaebenen der OMG

Es existiert eine Reihe von standardisierten Metametamodellen, auf deren Basis somit das Erstellen von formalen Sprachen möglich ist. Beispielsweise sei hier die *Meta Object Facility* (MOF) [Ob04a] der OMG erwähnt, auf deren Basis die *Unified Modeling Language* (UML) [Ob05a] definiert ist. Ein weiteres Metametamodell ist das von der Eclipse Foundation entwickelte *Ecore* Metametamodell, welches vom *Eclipse Modeling Framework* (EMF) [Bu03] ver-

wendet wird. Für beide Metametamodelle existiert Werkzeugunterstützung, die bei der Metamodellierung hilft.

2.3.6 DSLs und Frameworks

Wie bereits in Abbildung 6 dargestellt, werden domänenspezifische Sprachen im Rahmen dieser Arbeit komplementär zu Frameworks gesehen [De97, FSJ99 S. 16]. Die abstrakte Syntax einer DSL spiegelt die Struktur der Erweiterungspunkte eines Frameworks wider. Die Semantik der Sprache wird durch die Implementierung des Frameworks bestimmt. Die konkrete Syntax der DSL wird vom Anwendungsentwickler zur Anpassung des Frameworks wahrgenommen.

Analog zum Lebenszyklus eines Frameworks (siehe Seite 27) lässt sich modellgetriebene Softwareentwicklung in zwei Phasen untergliedern, und ebenfalls analog gibt es zwei verschiedene Rollen von Entwicklern, die die jeweiligen Tätigkeiten ausführen. In der ersten Phase wird die domänenspezifische Sprache basierend auf einem Framework-Kern entworfen. In der zweiten Phase kann die DSL zur Lösung von konkreten Problemen genutzt werden. In Abbildung 8 sind die in den jeweiligen Phasen beteiligten Elemente farblich gekennzeichnet.

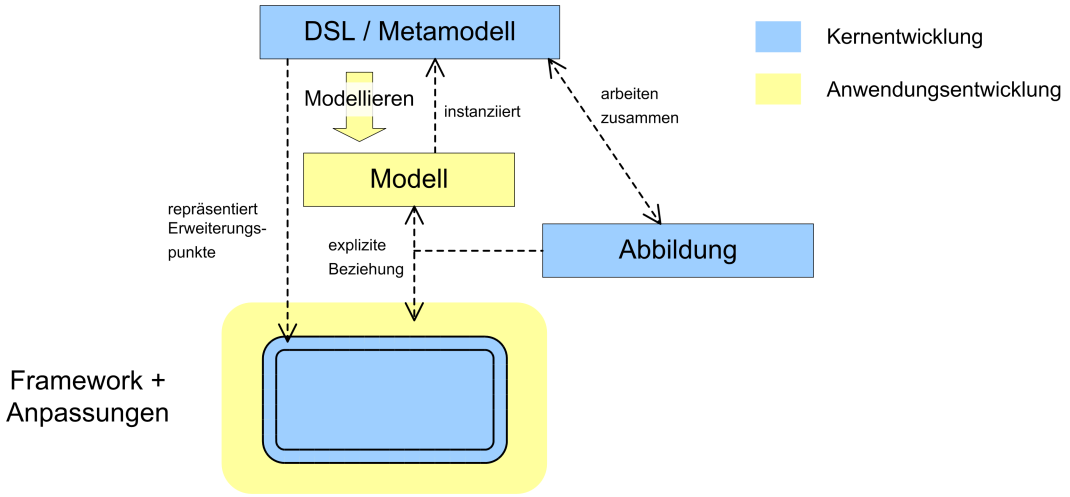


Abbildung 8: Schema modellgetriebener Softwareentwicklung

2.3.7 Integration von domänenspezifischen Sprachen

Bisher wurden die Grundideen modellgetriebener Softwareentwicklung unabhängig von einer konkreten Realisierung dargestellt. In diesem Abschnitt wird eine von existierenden Ansätzen oft vernachlässigte Anforderung an eine Realisierung vorgestellt. Es handelt sich um die Forderung nach *Integration* von domänenspezifischen Modellen sowohl mit dem darunter liegenden System als auch untereinander. Zwei Artefakte sind miteinander integriert, wenn es eine explizite Verbindung zwischen ihnen gibt, die für ein automatisiertes Werkzeug erkennbar ist. Integration ermöglicht Werkzeugunterstützung, welche das Bearbeiten der Artefakte erleichtert sowie die Konsistenz des Gesamtsystems überprüft.

Mit *Artefakt* werden im Kontext dieser Arbeit Dokumente oder Teile von Dokumenten bezeichnet, die von Menschen erstellt und bearbeitet werden. Dies ist in Abgrenzung zu abgeleiteten Konstrukten zu verstehen, die beispielsweise von Compilern erstellt werden. Im Kontext modellgetriebener Softwareentwicklung sind dies meist Dokumente, die auf der Basis einer formalen Notation erstellt wurden.

Ein Beispiel welches die gewinnbringende Nutzung vorhandener Integration demonstriert sind die in Abschnitt 2.1.2 vorgestellten Post-IntelliJ-IDEs für statisch typisierte objektorientierte Programmiersprachen. Aufgrund der statischen Typisierung ist der Programmcode zu einem hohen Grad integriert. Diese Integration ermöglicht die vorgestellte Werkzeugunterstützung.

Im Kontext modellgetriebener Softwareentwicklung gibt es drei Integrationsszenarien, die in Abbildung 9 dargestellt sind. Die wichtigste Integrationsform bezieht sich auf die Integration zwischen Modellartefakten und dem darunter liegenden Framework-Kern. Konkret bedeutet dies, dass jedes Modellelement eine direkte Verbindung mit dem korrespondierenden Element des Framework-Kerns aufweist. Es soll für ein Werkzeug möglich sein, die Konsistenz zwischen Modellelementen und Framework-Kern zu überprüfen.

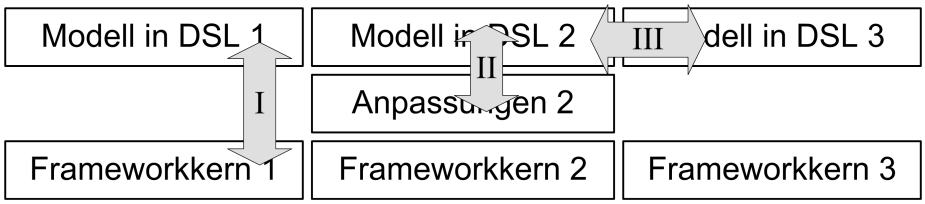


Abbildung 9: Integrationsanforderung an modellgetriebene Softwareentwicklung

Eine weitere Integrationsforderung betrifft die Beziehung zwischen Modellartefakten und Anpassungsartefakten, die direkt in der Basisprogrammiersprache verfasst wurden. In vielen Fällen bestehen Abhängigkeiten zwischen diesen beiden Artefaktklassen. Die Forderung nach Integration verlangt nach einer Realisierung dieser Abhängigkeiten, die für ein Werkzeug zugänglich ist. Das dritte Integrationsszenario bezieht sich auf Abhängigkeiten zwischen Artefakten verschiedener domänenspezifischer Sprachen.

Eine Umsetzung modellgetriebener Softwareentwicklung, die domänenspezifische Sprachen integriert realisiert, ermöglicht Werkzeugunterstützung und überbrückt die Lücke zwischen Modellierung in DSLs und Programmierung in der Basisprogrammiersprache. Dies ist für Akzeptanz und weite Verbreitung modellgetriebener Softwareentwicklung von großer Bedeutung.

2.3.8 Roundtrip-Engineering

Die Kernidee modellgetriebener Softwareentwicklung besteht darin, dass eine explizite und automatisierte Abbildung zwischen Modellen und Implementierungsartefakten existiert. Daraus ergibt sich die Fragestellung, welches Artefakt sich aus welchem ergibt, und an welcher Stelle somit Änderungen erlaubt sind. Falls die Abbildungsrichtung vom abstrakteren Modell zur detaillierteren Implementierung zeigt, spricht man von *Forward Engineering* [Fr03]. In diesem Fall sind Änderungen nur am Modell und nicht an der Implementierung erlaubt.

Der umgekehrte Fall wird *Reverse Engineering* genannt. In diesem Fall wird versucht, das Modell aus der Implementierung abzuleiten. Änderungen im Modell haben hierbei keine Auswirkungen. Das Forschungsgebiet des Reverse Engineerings versucht diese Abbildung möglichst generisch für verschiedenste Arten von Systemen zu bewerkstelligen [CC90].

Da eine vollständig deklarative modellhafte Beschreibung eines komplexen betrieblichen Informationssystems weder möglich noch wünschenswert ist, ist eine Kombination aus beiden Ansätzen anstrebenswert. Es existieren also Artefakte auf verschiedenen Abstraktionsniveaus, zwischen denen eine Abbildung existiert. Es soll möglich sein, auf beiden Abstraktionsniveaus Änderungen vorzunehmen. Die von den Änderungen betroffenen Artefakte sollen automatisch entsprechend aktualisiert werden. Dies wird *Roundtrip-Engineering* genannt [Fr03]. Das Umsetzen von Roundtrip-Engineering ist mit einer Reihe von Schwierigkeiten verbunden. Insbesondere lässt sich wahres Roundtrip-Engineering nicht durch bloßes Hintereinanderausführen von Forward Engineering und Reverse Engineering erreichen. Für eine genauere Diskussion sei auf [SK04] verwiesen. Ein Großteil der Schwierigkeiten ergibt sich aus der Tatsache, dass sowohl Original als auch Modell Artefaktcharakter haben.

Roundtrip-Visualisierungen

Eine besondere Umsetzung von Roundtrip-Engineering sind so genannte *Roundtrip-Visualisierungen* [CTM03]. Dieser besonders im Umfeld von WYSIWYG-Editoren für HTML-Dokumente weit verbreitete Ansatz (z.B. [Ad06] basiert auf Roundtrip-Visualisierungen) zeichnet sich dadurch aus, dass das Modell lediglich transient zum Zwecke der Modellierung erzeugt wird. Daraus ergibt sich, dass die Konsistenz zwischen Original und Modell stets gewährleistet ist. Eine Manipulation des Originals resultiert sofort in einem angepassten Modell. Veränderungen am Modell führen direkt zunächst zu Veränderungen am Original, was somit ein angepasstes Modell zur Folge hat. Bei derartigen Modellrepräsentationen handelt es sich nicht um Artefakte, die gepflegt werden müssen. Es entsteht kein zusätzlicher Verwaltungsaufwand durch die Verwendung von Modellen.

2.3.9 Migration von Modellen

Eine weitere Herausforderung an eine Realisierung modellgetriebener Softwareentwicklung entsteht, wenn sich das darunter liegende Framework weiterentwickelt. Verändern sich die Erweiterungsmöglichkeiten, so ändert sich das Metamodell der domänenspezifischen Sprache. Es stellt sich die Frage, wie existierende Modelle an dieser Weiterentwicklung teilhaben können.

Wünschenswert ist eine werkzeuggestützte automatische Anpassung der Modelle sobald das Framework verändert wird. Vorbild hierbei ist die Unterstützung von Refactoring durch moderne Post-IntelliJ-IDEs. Eine Realisierung dieses Szenarios ist daran gebunden, dass die domänenspezifische Sprache mit dem Framework integriert ist. Nur in diesem Fall kann ein Werkzeug Schlüsse von Veränderungen am Framework auf durchzuführende Anpassungen an Modellen durchführen. Ist keine Migration gegeben, so müssen Modelle manuell an veränderte Metamodelle angepasst werden (siehe [SK04]).

2.3.10 Realisierungsmöglichkeiten

Es stellt sich nun die Frage, wie modellgetriebene Softwareentwicklung konkret realisiert werden kann. In Abbildung 10 ist eine Klassifikation von existierenden Realisierungsmöglichkeiten graphisch dargestellt.

Die grundlegende Unterscheidung besteht in der Realisierung als interne oder externe DSL [DKV00, Fo05a]. Dies bezieht sich auf die Frage, ob die Sprachelemente der DSL auch gleichzeitig Elemente der Basisprogrammiersprache sind oder nicht. Ist dies der Fall, spricht man von einer *internen* domänenspezifischen Sprache. Dies wird oft auch als *eingebettete* DSL bezeichnet [Hu96]. Eine interne DSL wird erstellt, indem die Syntax der Basisprogrammiersprache auf die zu lösende Problemdomäne angepasst wird. Dieser Ansatz verfügt über eine Reihe von positiven Eigenschaften. So kann die für die Basisprogrammiersprache vorhandene Infrastruktur (Interpreter, Debugger, IDE, ...) wieder verwendet werden, was das Erstellen einer internen DSL erheblich vereinfacht. Des Weiteren sind die Modellelemente automatisch mit dem darunter liegenden System integriert.

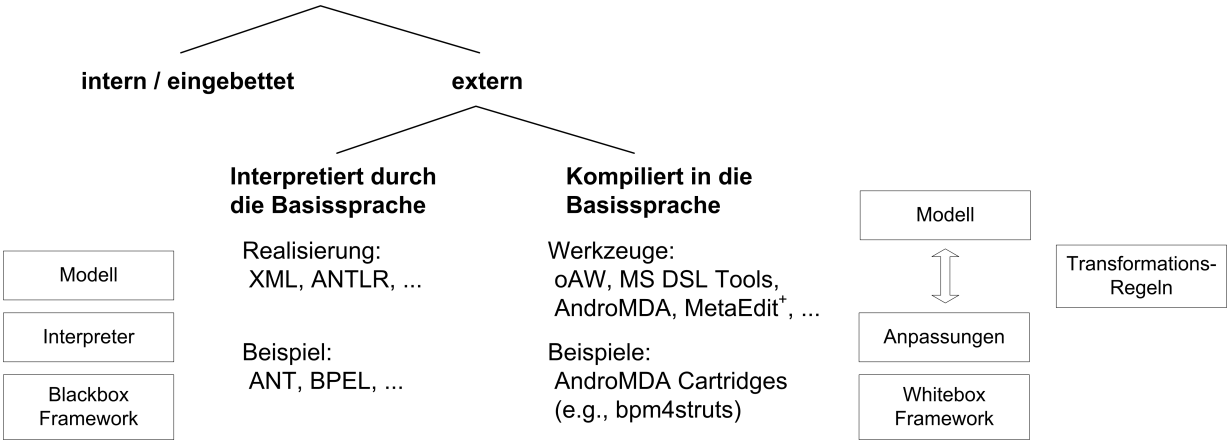


Abbildung 10: Realisierungsmöglichkeiten für domänenspezifische Sprachen

Die vorgestellten Integrationsforderungen lassen sich also mit einer internen domänenspezifischen Sprache erfüllen. Den genannten Vorteilen stehen Nachteile dieser Realisierungsmöglichkeit gegenüber. Wie bereits dargestellt, wird eine interne DSL durch flexible Anpassung der Syntax der Basisprogrammiersprache erstellt. Die hauptsächlich für die Erstellung betrieblicher Informationssysteme verwendeten Programmiersprachen verfügen über sehr beschränkte Möglichkeiten diesbezüglich. Ein weiterer Nachteil besteht darin, dass der Nutzer der domänenspezifischen Sprache zusätzlich zu den Sprachmerkmalen der DSL stets die volle Mächtigkeit der Basisprogrammiersprache zur Verfügung hat (siehe Abschnitt 2.2.5, Nachteil 5). Die syntaktische Korrektheit lässt sich also mit den direkt vorhandenen Werkzeugen nur in Bezug auf die Basisprogrammiersprache überprüfen.

Wie in Abschnitt 2.2.3 dargestellt, können Frameworks direkt durch Quellcode der Basisprogrammiersprache angepasst werden. Dieser Anpassungsquellcode hängt von den existierenden domänenspezifischen Erweiterungspunkten des Frameworks ab. Dies kann bereits als interne domänenspezifische Sprache aufgefasst werden.

Alternativ kann eine domänenspezifische Sprache auch *extern* realisiert werden. In diesem Fall sind die Sprachelemente der DSL nicht Konstrukte der Basisprogrammiersprache. Dies hat den Vorteil, dass die Syntax der DSL frei gewählt werden kann. Externe DSLs sind nicht von der syntaktischen Anpassbarkeit der Basisprogrammiersprache abhängig und lassen sich somit für jede beliebige Basisprogrammiersprache definieren. Diesem Vorteil steht ebenfalls eine Reihe von konzeptuellen Nachteilen gegenüber. So kann für eine externe DSL die gesamte für die Basisprogrammiersprache vorhandene Infrastruktur nicht wieder verwendet werden. Dementsprechend ist es erheblich aufwändiger, eine externe DSL zu erstellen. Ein weiterer Nachteil ist die fehlende Integration zwischen den Artefakten der DSL und dem darunter liegenden System. Es besteht eine *symbolische Barriere* (engl.: symbolic barrier, siehe [Fo05a]) zwischen Konstrukten der DSL und Konstrukten der Basisprogrammiersprache.

Für externe domänenspezifische Sprachen existieren wiederum zwei unterschiedliche Realisierungsmöglichkeiten. Die allgemeine Problemstellung besteht darin, dass das in einem externen Artefakt spezifizierte Modell in Aufrufe der Basisprogrammiersprache überführt werden muss. Eine Möglichkeit besteht darin, dass das Modell von einer Komponente der Basisprogrammiersprache interpretiert wird. Diese Interpreter-Komponente versteht die Syntax der DSL und überführt die Modellelemente in Aufrufe des Framework-Kerns. Dieser Schritt findet zur Laufzeit statt. In diesem Fall handelt es sich bei dem instrumentalisierten Framework im Allgemeinen um ein Blackbox Framework, da durch die deklarative DSL Komponenten des Frameworks ausgewählt und konfiguriert werden (siehe Abschnitt 2.2.1).

Eine weitere Möglichkeit besteht darin, die Modellelemente direkt in Ausdrücke der Basisprogrammiersprache zu übersetzen. Dieser Übersetzungsschritt findet zur Entwicklungszeit statt und wird durch Transformationsregeln gesteuert. Die Anpassungen des Frameworks liegen in diesem Fall in Form von Programmiersprachenkonstrukten vor.

2.3.11 Zusammenfassung

Modellgetriebene Softwareentwicklung versucht, die Abstraktionslücke durch den Einsatz von Modellen zu schließen. Dabei werden maßgeschneiderte domänenspezifische Sprachen entwickelt, die jeweils die Anpassungsmöglichkeiten eines Frameworks abdecken. Somit ist Anpassung auf hohem Abstraktionsniveau möglich. Im Unterschied zu modellbasierten Verfahren beeinflussen Modelle das Verhalten des Systems direkt. Eine Anforderung an eine Realisierung modellgetriebener Softwareentwicklung besteht in der Integration der Modelle mit dem instrumentalisierten Framework-Kern.

Da domänenspezifische Sprachen größtenteils deklarativen Charakter haben, strebt modellgetriebene Softwareentwicklung eine Abdeckung der Anwendungsteile an, die sich deklarativ ausdrücken lassen. Für die verbleibenden Anwendungsteile wird die volle Ausdrucksmächtigkeit der objektorientierten Programmiersprache benötigt. Eine Anforderung bei der Umsetzung modellgetriebener Softwareentwicklung besteht somit in der Integration dieser beiden Bestandteile.

3 Introspektive modellgetriebene Softwareentwicklung

In diesem Kapitel werden die Kernkonzepte introspektiver modellgetriebener Softwareentwicklung vorgestellt. Die in Abschnitt 3.2 vorgestellte Grundidee des vorgeschlagenen Ansatzes besteht darin, Erweiterungspunkte, die für eine deklarative Modellierung relevant sind, im Framework-Kern explizit zu kennzeichnen. Die abstrakte Syntax der vom Framework definierten domänenspezifischen Sprache lässt sich somit durch Introspektion direkt ablesen. Die Entwicklung der DSL fällt sozusagen mit der Entwicklung des Frameworks zusammen. Dies führt zu Modellen, die mit dem darunter liegenden Framework integriert sind.

Entsprechend den zwei fundamental verschiedene Arten von Frameworks gibt es introspektive Blackbox und Whitebox Frameworks, die sich in der Art und Weise, wie Introspektion angewandt wird unterschieden. Dies wird in den Abschnitten 3.3 und 3.4 vorgestellt.

Im Abschnitt 3.6 wird ein Überblick über die im Rahmen dieser Arbeit entwickelte Realisierung introspektiver modellgetriebener Softwareentwicklung gegeben. Abschnitt 3.7 schließt dieses Kapitel mit einem Überblick über verwandte Arbeiten ab.

3.1 Introspektion in Abgrenzung zu Reflektion

Der Vorgang, mit dem aus entsprechend gekennzeichneten Implementierungsartefakten Modellinformationen gewonnen werden, wird im Folgenden *Introspektion* (engl.: *Introspection*) genannt. Der Begriff Introspektion leitet sich vom lateinischen Verb *introspicere* ab, was soviel wie „hineinschauen“ bedeutet. Besondere Betonung soll auf der Abgrenzung zum in der Informatik weit verbreiteten Begriff der *Reflektion* (engl.: *Reflection*) gelegt werden. Reflektion ist in [Ma87] folgendermaßen definiert:

> *Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.*

Reflektion findet also zur Laufzeit des Programms statt und wird von modernen Programmiersprachen unterstützt [Mc98]. Im Gegensatz dazu findet Introspektion im Verständnis dieser Arbeit zur Entwicklungszeit statt und ermöglicht die Extraktion von Modellinformationen aus Implementierungsartefakten.

3.2 Grundidee introspektiver modellgetriebener Softwareentwicklung

Mit dem Terminus *introspektiv* wird etwas bezeichnet, das Introspektion unterstützt. Darin ist die Idee enthalten, dass Introspektion nur an bestimmten gekennzeichneten Stellen möglich ist. Durch Introspektion werden Modelle aus detaillierteren Implementierungsartefakten extrahiert. Aufgrund der Abbildungsrichtung handelt es sich also um einen Reverse Engineering Ansatz mit der speziellen Eigenschaft, dass das zu analysierende System dies aktiv unterstützt. Die

durch Introspektion extrahierten Modellelemente stehen jederzeit mit den originalen Implementierungsartefakten in Verbindung, es besteht Integration zwischen ihnen. Bei den Artefakten, auf denen Introspektion durchgeführt wird, handelt es sich im Folgenden um Frameworks und Anpassungen an diesen.

Aufgrund der großen Schwierigkeiten, Design aus beliebigen Implementierungsartefakten zu rekonstruieren, ist es erforderlich, dass die zu analysierenden Frameworks ein introspektives Programmiermodell einhalten. Frameworks, in denen Erweiterungspunkte durch Annotationen gekennzeichnet sind, und deren Programmiermodell eine introspektive Analyse ermöglicht, werden im Folgenden *introspektive Frameworks* genannt. Die Kernbestandteile introspektiver modellgetriebener Softwareentwicklung können folgendermaßen zusammengefasst werden: Code + Annotationen + Struktur.

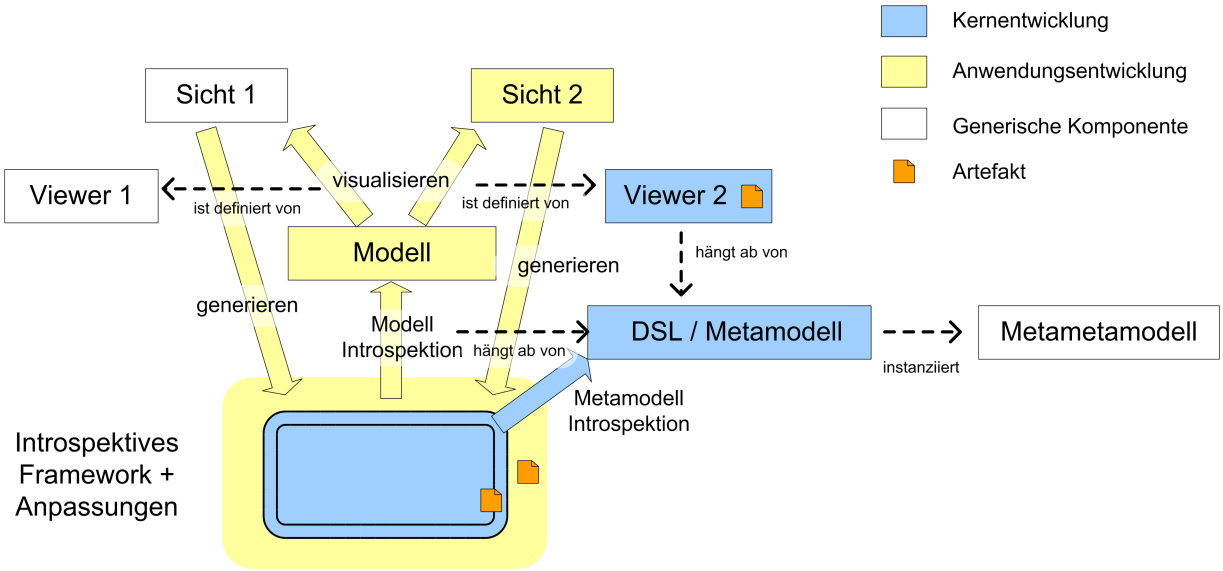


Abbildung 11: Schema introspektiver modellgetriebener Softwareentwicklung

In Abbildung 11 sind die wichtigsten beteiligten Elemente sowie deren Beziehungen schematisch dargestellt. Ausgangspunkt der Kernentwicklung ist ein introspektives Framework, genauer ein introspektiver Framework-Kern. Im Unterschied zur Entwicklung eines nicht-introspektiven Frameworks besteht eine zusätzliche Aufgabe des Framework-Entwicklers darin, die Erweiterungspunkte durch Annotationen zu kennzeichnen, und somit die Struktur des Frameworks einer introspektiven Analyse zugänglich zu gestalten. Als Ergebnis dessen lässt sich die Struktur der domänenspezifischen Sprache, die durch die Erweiterungsmöglichkeiten des Frameworks spezifiziert wird, durch Introspektion ermitteln. Es existiert ein generisches Metametamodell, welches dem Frameworkentwickler Konstruktionsmöglichkeiten introspektiver Frameworks vorgibt. Die Extraktion des Metamodells des Frameworks wird Metamodell-Introspektion genannt. Da zur Konstruktion introspektiver Frameworks ein vorgegebenes generisches Metametamodell benutzt wird, kann der Vorgang der Metamodell-Introspektion ebenfalls generisch durch ein Werkzeug unterstützt werden und ist nicht vom konkreten Framework abhängig. Eine Weiterentwicklung des Frameworks führt also automatisch zu einem angepassten Metamodell der domänenspezifischen Sprache.

In der Anwendungsentwicklungsphase soll mit Hilfe des Frameworks ein Problem gelöst werden. Dies geschieht durch Anpassen des Frameworks. Die Anpassungen sind deklarativer Natur und somit durch die domänenspezifische Sprache darstellbar. Sie werden durch Modellelemente repräsentiert, die das Metamodell der DSL instanziieren.

Die Modelle werden durch Sichten dargestellt, welche durch Viewer erstellt werden. Dies repräsentiert die konkrete Syntax der DSL. Der links dargestellte „Viewer 1“ und die sich daraus ergebende „Sicht 1“ visualisieren Modelle auf eine generische Art und Weise und müssen nicht vom Frameworkentwickler erstellt werden. Soll die konkrete Syntax von dieser generischen Visualisierung abweichen, können vom Kernentwickler maßgeschneiderte Viewer erstellt werden (siehe „Viewer 2“ und „Sicht 2“ in Abbildung 11).

Die Modelle können in den Sichten manipuliert werden. Dies führt zu veränderten Anpassungsartefakten, was sofort zu einem angepassten Modell und somit zu einer aktuellen Sicht führt. Dabei handelt es sich um eine Roundtrip-Visualisierung (siehe Abschnitt 2.3.8).

Besonderes Augenmerk soll auf den Artefaktcharakter der vorgestellten Elemente gelegt werden. Die beim Ansatz der introspektiven modellgetriebenen Softwareentwicklung maßgeblichen Artefakte sind der Framework-Kern und die vorgenommenen Anpassungen. Die Besonderheit des introspektiven Ansatzes besteht darin, dass sowohl das Metamodell, als auch das Modell sowie die zur Visualisierung erzeugten Sichten keinen Artefaktcharakter besitzen. Sie werden bei Bedarf „on-the-fly“ erzeugt und sind somit keine Artefakte des Entwicklungsprozess. Daraus ergibt sich die Möglichkeit, sowohl Modell als auch abgebildete Anpassungsartefakte voneinander unabhängig zu manipulieren und somit echtes Roundtrip-Engineering zu ermöglichen.

Entsprechend den zwei Arten von Frameworks werden introspektive Blackbox Frameworks und introspektive Whitebox Frameworks unterschieden.

Exkurs: Annotationen

Moderne objektorientierte Programmiersprachen verfügen über die Möglichkeit, bestimmte Konstrukte durch Metadaten zu kennzeichnen. Diese Kennzeichnungen werden in Java *Annotationen* genannt [BL04]. Mit Annotationen ist es möglich, Klassen, Interfaces, Methoden und Attribute zu kennzeichnen. Die Kennzeichnung verändert die Semantik des gekennzeichneten Elements nicht direkt, gleichwohl ist es möglich dies automatisiert durch Werkzeuge zu tun [KM05].

Es existiert eine Reihe von vordefinierten Annotationstypen. So kennzeichnet beispielsweise die Annotation `Override` Methoden, von denen explizit verlangt wird, dass sie eine Methode einer Superklasse überschreiben:

```
@Override
public boolean equals(Object that) { ... }
```

Es ist möglich, eigene Annotationstypen zu definieren. Annotationen können über Attribute verfügen, für die Standardwerte angegeben werden können.

3.3 Externe integrierte DSLs durch introspektive Blackbox Frameworks

Anpassung eines Blackbox Frameworks findet statt, indem existierende Klassen ausgewählt, ihre Eigenschaften konfiguriert und zueinander in Beziehung gesetzt werden. Dies sind die allgemeinen, immer wiederkehrenden Grundkonzepte zur Anpassung von Blackbox Frameworks. Konkret befinden sich im Quellcode des Framework-Kerns Stellen, an denen Eigenschaften und Beziehungen anpassbar sind. Diese Erweiterungsmöglichkeiten werden in einem introspektiven Blackbox Framework durch Annotationen explizit gekennzeichnet. Entsprechend den allgemeinen Grundkonzepten gibt es Annotationstypen zum Kennzeichnen von anpassbaren Eigenschaften und Beziehungen. Diese Annotationstypen bilden das Metametamodell der Blackbox Introspektion und geben die Möglichkeiten des Kernentwicklers, introspektive Blackbox Frameworks zu entwickeln, vor. In dieser Arbeit wurde ein konkretes Metametamodell entworfen, welches im Abschnitt 4.2 vorgestellt wird.

Der Kern eines introspektiven Blackbox Frameworks lässt sich analysieren, und das Metamodell der vom Framework spezifizierten domänenspezifischen Sprache kann durch Metamodell-Introspektion extrahiert werden. Das extrahierte Metamodell instanziiert das generische Metametamodell. Basierend auf dem extrahierten Metamodell kann die zum Framework gehörige domänenspezifische Sprache visualisiert und zum Anpassen genutzt werden. Die erstellten Anpassungen werden in Form eines XML-Dialekts extern abgespeichert. Diese Zusammenhänge sind in Abbildung 12 schematisch dargestellt. Die erstellten Anpassungen sind mit dem Framework-Kern integriert, da jederzeit eine explizite Verbindung zum annotierten Erweiterungspunkt aufrechterhalten wird.

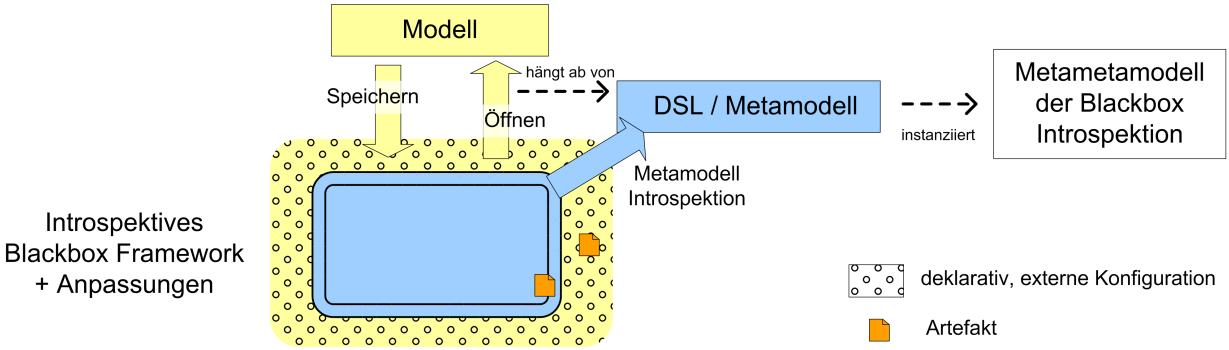


Abbildung 12: Introspektive Blackbox Frameworks

Eine Integration von händisch erstellten Implementierungsartefakten mit Modellen eines Blackbox Frameworks ist nicht möglich, da die Modelle nicht in Form von Programmiersprachenkonstrukten repräsentiert werden. Introspektive Blackbox Frameworks werden detailliert in Kapitel 4 vorgestellt.

3.4 Interne integrierte DSLs durch introspektive Whitebox Frameworks

Im Falle eines Whitebox Frameworks findet Anpassung an ein konkretes Problem durch Implementieren von Subklassen statt. Wie im Abschnitt 2.2.4 beschrieben, besteht eine Motivati-

on für die Nutzung eines Whitebox Frameworks darin, die volle Ausdrucksmächtigkeit der Basisprogrammiersprache an bestimmten Stellen zu nutzen. Whitebox Frameworks verfügen also oft einerseits über Aspekte, die sich in einer deklarativen Art und Weise durch eine domänenspezifische Sprache modellieren lassen sowie andererseits über Aspekte, die aufgrund der notwendigen Ausdrucksmächtigkeit, direkt durch Konstrukte der Basisprogrammiersprache spezifiziert werden. Anpassung findet also auf unterschiedlichen Abstraktionsebenen statt. Eine Integration dieser beiden Modellierungsaspekte ist von großer Bedeutung, da nur so die Konsistenz des Systems automatisch überprüft werden kann. Diese Forderung lässt sich nur durch eine interne domänenspezifische Sprache realisieren. Die Grundidee introspektiver Whitebox Frameworks besteht darin, domänenspezifische Modelle direkt als Quellcode der Basisprogrammiersprache zu repräsentieren. Durch Introspektion ist eine Visualisierung auf hohem Abstraktionsniveau möglich.

Konkret betrachtet bestehen die Erweiterungspunkte eines Whitebox Frameworks aus abstrakten Methoden, die zur Anpassung implementiert werden (siehe Abschnitt 2.2.3). In einem introspektiven Whitebox Framework werden diejenigen Methoden, deren Anpassung deklarativ durch eine domänenspezifische Sprache stattfinden soll, durch Annotationen gekennzeichnet. Diese Methoden werden introspektive Methoden genannt, da sie eine Analyse der Anpassungskonstrukte ermöglichen. Im Rahmen dieser Arbeit wurden immer wiederkehrende Arten introspektiver Methoden identifiziert, die das Metametamodell introspektiver Whitebox Frameworks bilden. Die Gesamtheit aller annotierten introspektiven Methoden eines Frameworks formt das Metamodell der vom Framework spezifizierten domänenspezifischen Sprache, welches durch Metamodell-Introspektion gewonnen wird. Konkrete Anpassungen implementieren die annotierten introspektiven Methoden unter Benutzung des vorgegebenen introspektiven Programmiermodells, welches eine introspektive Analyse ermöglicht.

Eine Analyse der konkreten Anpassungsartefakte führt zu einem Modell, welches dann visualisiert und manipuliert werden kann. Dieser Vorgang wird Modellintrospektion genannt. Modelle werden in Form von Konstrukten der Basisprogrammiersprache repräsentiert, es handelt sich hierbei also um eine Realisierung in Form einer internen domänenspezifischen Sprache, die jedoch durch Roundtrip-Visualisierung auf einem hohen Abstraktionsniveau zugänglich gemacht wird (siehe hierzu auch Abschnitt 5.4). Dies kann auch durch die einfache Formel „*Code is Model*“ ausgedrückt werden (siehe hierzu [Pi05]).

Neben den einer Introspektion zugänglichen Aspekten kann die Anpassung eines Whitebox Frameworks auch aus Methodenimplementierungen bestehen, die die volle Ausdrucksmächtigkeit der Basisprogrammiersprache nutzen. Aufgrund der internen Modellrepräsentation in Form von Programmiersprachkonstrukten ist eine Integration von deklarativen Aspekten, die Teil der domänenspezifischen Sprache sind, mit händisch implementierten Aspekten direkt gegeben. In Abbildung 13 sind die geschilderten Zusammenhänge graphisch dargestellt.

Zusammenfassend können Anpassungen von Whitebox Frameworks aus sowohl imperativ als auch deklarativ formulierten Bestandteilen bestehen. Der introspektive Ansatz besteht darin, die deklarativ kodifizierten Aspekte zu analysieren und als Modell zu repräsentieren. Die Stellen, an denen ein deklaratives Programmiermodell gewählt wurde werden durch Annotationen direkt im Framework gekennzeichnet und auf dieser Basis findet die Analyse des Modells statt. Introspektion findet auf verschiedenen Metaebenen statt (siehe Seite 35).

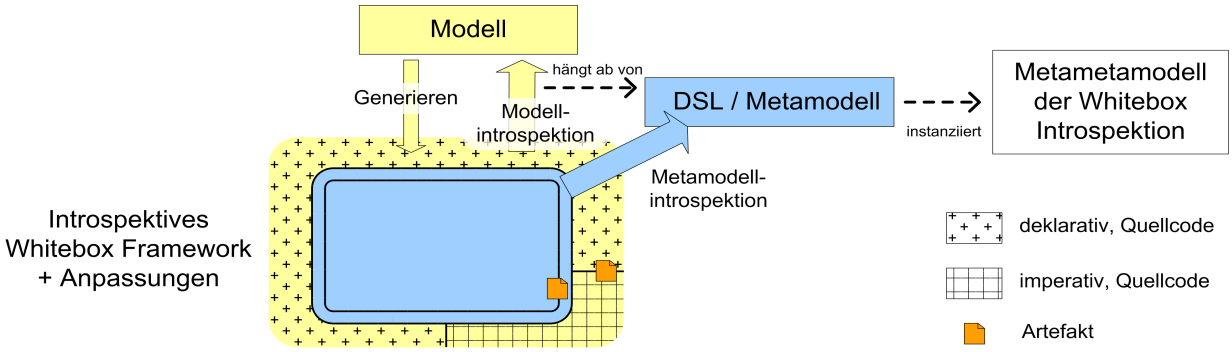


Abbildung 13: Introspektive Whitebox Frameworks

Introspektive Whitebox Frameworks werden detailliert in Kapitel 5 diskutiert.

3.5 Ziele von IMDD

Introspektive modellgetriebene Softwareentwicklung ist eine Realisierung modellgetriebener Softwareentwicklung und verfolgt somit wie in Abschnitt 2.3.4 dargestellt das Ziel, durch eine kompaktere und verständlichere Systembeschreibung sowohl die Entwicklungszeit zu verkürzen, als auch die Qualität zu verbessern. Das Erreichen dieser Ziele hängt davon ab, wie Modelle technisch realisiert werden. Der besondere Schwerpunkt des in dieser Arbeit vorgeschlagenen introspektiven Ansatzes besteht darin, Modelle mit dem darunterliegenden System *integriert* zu realisieren (siehe Abschnitt 2.3.7). Durch die Integration der Modellrepräsentationen kann die Konsistenz des Gesamtsystems werkzeuggestützt sichergestellt werden. Existierende Ansätze, in denen Modelle nicht-integriert realisiert werden, erfordern manuellen Aufwand zur Sicherstellung der Konsistenz. Dieser zusätzliche Aufwand schmälert die angestrebten Produktivitätsgewinne und stellt eine Fehlerquelle dar.

Ein positiver Effekt, der sich durch die Anwendung introspektiver modellgetriebener Softwareentwicklung ergibt, ist eine Strukturverbesserung des erstellten Quellcodes. Wie in Abschnitt 3.2 dargestellt, erfordert IMDD die Einhaltung gewisser struktureller Vorgaben an den erstellten Quellcode. Introspektive Modellbildung ist nur möglich, wenn diese strukturellen Vorgaben eingehalten werden. Eine Realisierung introspektiver modellgetriebener Softwareentwicklung überprüft die Einhaltung dieser strukturellen Vorgaben während des Entwicklungsprozesses. Wie in [ER03 S. 74] dargestellt, führt eine verbesserte Strukturierung zu Software, die weniger Fehler enthält und einfacher zu warten ist. Dies ergibt sich aus der verbesserten Lesbarkeit und Verständlichkeit.

Im Unterschied zu existierenden Ansätzen ist eine technologieunabhängige Problembeschreibung nicht Ziel introspektiver modellgetriebener Softwareentwicklung. Dies widerspricht dem Ziel einer möglichst nahtlosen Integration von Modellen und darunterliegenden Frameworks.

3.6 Realisierung von IMDD

Bisher wurde die Idee introspektiver modellgetriebener Softwareentwicklung unabhängig von einer konkreten Realisierung vorgestellt. Um den praktischen Wert des Ansatzes bewerten zu können, wurde im Rahmen dieser Arbeit eine Plattform zur Unterstützung introspektiver Modellierung entwickelt.

Die Kernaufgabe, die von einem Werkzeug zur Unterstützung des Ansatzes geleistet werden muss, ist der Vorgang der Introspektion. Dabei handelt es sich – technisch betrachtet – um statische Codeanalyse. Die durch Annotationen gekennzeichneten Strukturen müssen zunächst gefunden werden. Anschließend werden Modelle aus den analysierten Codeartefakten erzeugt. Die Darstellung der Modelle erfolgt durch Sichten „on-the-fly“. Es soll stets möglich sein, zwischen Quelltext und abgeleiteten Modellen hin und her zu navigieren. Daraus folgt, dass eine enge Integration in eine integrierte Entwicklungsumgebung wünschenswert ist. Ein weiterer Punkt, der dies nahe legt, ist die weite Verbreitung moderner Post-IntelliJ-IDEs (siehe Seite 24). Diese unterhalten stets eine interne Repräsentation der Programme als abstrakten Syntaxbaum. Auf Basis dieser Datenstruktur ist eine statische Codeanalyse zum Erzeugen introspektiver Modelle leicht realisierbar.

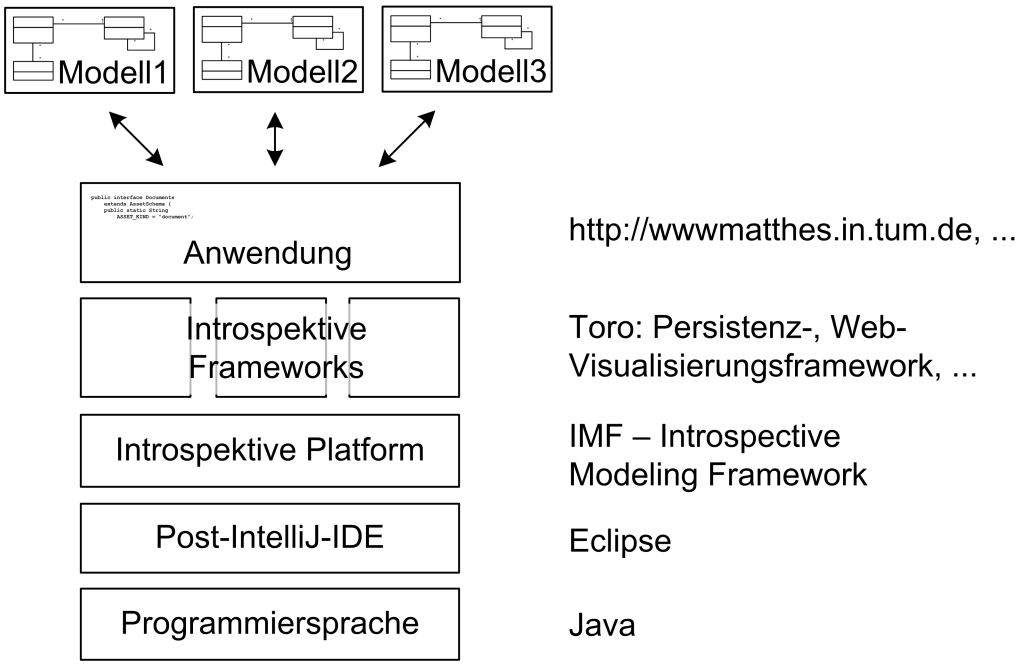


Abbildung 14: Technologie-Stack von IMDD

Wie bereits im Exkurs zum Thema Post-IntelliJ-IDEs auf Seite 24 erwähnt, wurde das im Rahmen dieser Arbeit entwickelte Werkzeug auf Basis der Eclipse-IDE entwickelt. Dabei handelt es sich um das *Introspective Modeling Framework – IMF*, welches in Kapitel 6 detailliert vorgestellt wird. Technisch betrachtet ist das IMF ein Framework, für das jedoch bereits Anpassungen an häufig benötigte Funktionalitäten existieren. So unterstützt IMF sowohl die Introspektion von Blackbox, als auch von Whitebox Frameworks. Es sei jedoch darauf hingewie-

sen, dass der konzeptuelle Ansatz der introspektiven modellgetriebenen Softwareentwicklung keinesfalls mit der konkreten Realisierung durch das IMF gleichzusetzen ist.

In Abbildung 14 ist der Technologie-Stack dargestellt, dem eine Realisierung von IMDD genügt. Auf der linken Seite der Abbildung sind die Technologiekategorien eingetragen, rechts sind die in dieser Arbeit vorgestellten konkreten Realisierungen aufgezeigt.

3.7 Verwandte Arbeiten

In diesem Abschnitt wird ein Überblick über verwandte Arbeiten gegeben, auf die teilweise an anderer Stelle tiefergehend eingegangen wird. Da der vorgestellte Ansatz eine Realisierungsform modellgetriebener Softwareentwicklung ist, lassen sich zur Einordnung hier Arbeiten auf diesem Gebiet aufführen. In Abschnitt 2.3.10 wurden bereits konzeptuelle Möglichkeiten der Realisierung modellgetriebener Softwareentwicklung vorgestellt. Wie dargestellt, sind interne domänenspezifische Sprachen aufgrund der Inflexibilität der hauptsächlich genutzten objektorientierten Sprachen wenig verbreitet. Existierende Ansätze lassen sich in die Kategorien externe interpretierte oder externe kompilierte Realisierungsform klassifizieren.

In Abschnitt 4.8 werden verwandte Ansätze diskutiert, die modellgetriebene Softwareentwicklung durch externe interpretierte Sprachen realisieren. Verwandte Ansätze, die DSLs auf eine externe kompilierte Art und Weise realisieren, werden detailliert in Abschnitt 5.6 vorgestellt und anschließend in Abschnitt 5.6.3 diskutiert. Beide Realisierungsformen erfüllen die in Abschnitt 2.3.7 aufgestellten Integrationsforderungen nicht.

Ein aktueller Ansatz, der ein ähnliches Ziel wie diese Arbeit verfolgt, erstellt so genannte *Frameworks-Specific Modeling Languages* (FSML) [An06, AC06]. Das Ziel dieses Ansatzes besteht darin, domänenspezifische Sprachen auf Basis beliebiger vorhandener Frameworks zu erstellen und Roundtrip-Engineering zu ermöglichen. Im Unterschied zu bisherigen Ansätzen werden Transformationen in beide Richtungen – vom Modell zum Quellcode und anders herum – erstellt, und es existieren Mechanismen zur Konfliktbehebung. Dies soll Roundtrip-Engineering ermöglichen, sowohl Modelle als auch Quellcode sollen unabhängig voneinander bearbeitbar sein. Eine konkrete FSML besteht aus einer abstrakten Syntax, die die Erweiterungsmöglichkeiten des Frameworks abdeckt, Abbildungen zwischen Modell und Quellcode sowie einer konkreten Syntax. In diesem Ansatz wird Integration zwischen Framework und domänenspezifischer Sprache durch Abbildungen in beide Richtungen versucht zu erreichen. Nachteilig an diesem Ansatz ist der mit der Erstellung der Abbildungsvorschriften verbundene Aufwand. Beide Abbildungsrichtungen müssen sowohl miteinander, als auch mit dem Framework-Kern und der abstrakten Syntax der Sprache konsistent sein. Diese Konsistenz muss händisch aufrechterhalten werden. Als Ergebnis dessen sind zwar die Modelle mit dem Framework-Kern integriert, die Modellierungssprache als solche (abstrakte Syntax, Abbildungsvorschriften) ist jedoch nicht integriert.

Eine weitere große Gruppe verwandter Arbeiten versucht Erkenntnisse über bestehenden Quellcode durch Analysen zu gewinnen und durch geeignete Visualisierungen sichtbar zu machen. Dieses Forschungsgebiet wird *Software Visualization* genannt und hat zum Ziel, die Verständlichkeit des analysierten Systems zu erhöhen. In [PBS93] und [RC93] werden Taxonomi-

en für Arbeiten in diesem Bereich vorgestellt. Im Unterschied zur vorliegenden Arbeit besteht der Anspruch dieses Ansatzes darin, Aussagen über beliebigen Quellcode zu treffen. Aufgrund der Ausdrucksmächtigkeit der zu untersuchenden Programmiersprachen ist dies mit großen Schwierigkeiten verbunden – so existiert beispielsweise kein Algorithmus der eine scheinbar einfache Aussage darüber treffen kann, ob ein vorgegebenes Programm terminiert oder nicht. Ein weiterer Unterschied zur vorliegenden Arbeit besteht darin, dass die im Gebiet der Software Visualization genutzten Abstraktionen meist generischer Natur sind und sich nicht auf eine bestimmte Domäne beziehen.

Trotzdem eine große Anzahl von Arbeiten auf dem Gebiet der Software Visualization stattgefunden hat, haben die daraus hervorgehenden Werkzeuge und Methoden wenig Eingang in die Praxis der Softwareentwicklung gefunden. Eine Diskussion dieses Sachverhalts wird in [Re05] durchgeführt.

4 Introspektive Blackbox Frameworks

In diesem Kapitel wird das Konzept der introspektiven Blackbox Frameworks ausführlich vorgestellt. Zur Veranschaulichung wird ein Beispielframework genutzt, welches im folgenden Abschnitt eingeführt wird. In Abschnitt 4.2 wird das zur Konstruktion von introspektiven Blackbox Frameworks zur Verfügung stehende Metametamodell detailliert vorgestellt. Anschließend wird in Abschnitt 4.3 die Nutzersicht und somit die konkrete Syntax der integrierten domänenspezifischen Sprache eines introspektiven Blackbox Frameworks vorgestellt. Die mit dem Einlesen existierender Konfigurationen verbundenen Aspekte werden in Abschnitt 4.4 besprochen. Anschließend wird der Ablauf der Instanziierung eines introspektiven Blackbox Frameworks basierend auf einer erstellten Konfiguration vorgestellt. In Abschnitt 4.7 werden die Eigenschaften der vorgestellten integrierten domänenspezifischen Sprache zur Konfiguration von introspektiven Blackbox Frameworks zusammenfassend dargestellt. Das Kapitel wird durch eine Diskussion verwandter Ansätze zur modellgetriebenen Softwareentwicklung auf Basis von Blackbox Frameworks abgeschlossen.

4.1 Einführung

In diesem Abschnitt wird die Grundidee introspektiver Blackbox Frameworks anhand eines Beispielsframeworks vorgestellt.

4.1.1 Beispiel: Webserver-Framework

Als Beispielframework zur Veranschaulichung der Grundkonzepte introspektiver Blackbox Frameworks wird in diesem Abschnitt das Webserver-Framework der in Kapitel 7 vorgestellten Webplattform *Toro* vorgestellt. Die Aufgabe des Webserver-Frameworks ist das Entgegennehmen, Interpretieren und Beantworten von HTTP-Requests. Die für die weitere Diskussion relevanten Klassen sind in einem implementierungsnahen UML-Klassendiagramm in Abbildung 15 dargestellt.

Die Kernkomponente des gesamten Systems ist die Klasse `Broker`. Da es sich um eine Webanwendung handelt, verfügt ein Broker über die Eigenschaft `host` die spezifiziert, auf welcher Maschine der Server installier ist. Ein Objekt vom Typ `Broker` kann über mehrere Webserver verfügen, die durch die Klasse `WebServer` realisiert werden. Die Klasse `SslWebServer` ist eine Spezialisierung der Klasse `WebServer`, die Inhalte verschlüsselt ausliefert. Ein Webserver verfügt über eine Reihe von Eigenschaften, die als Attribute verwaltet werden. Für Eigenschaften, die einfache Zugriffsszenarien des Webservers betreffen, existiert die Klasse `Configuration`. Durch die Klasse `HttpLog` wird spezifiziert, auf welche Art und Weise Requests des Servers geloggt werden.

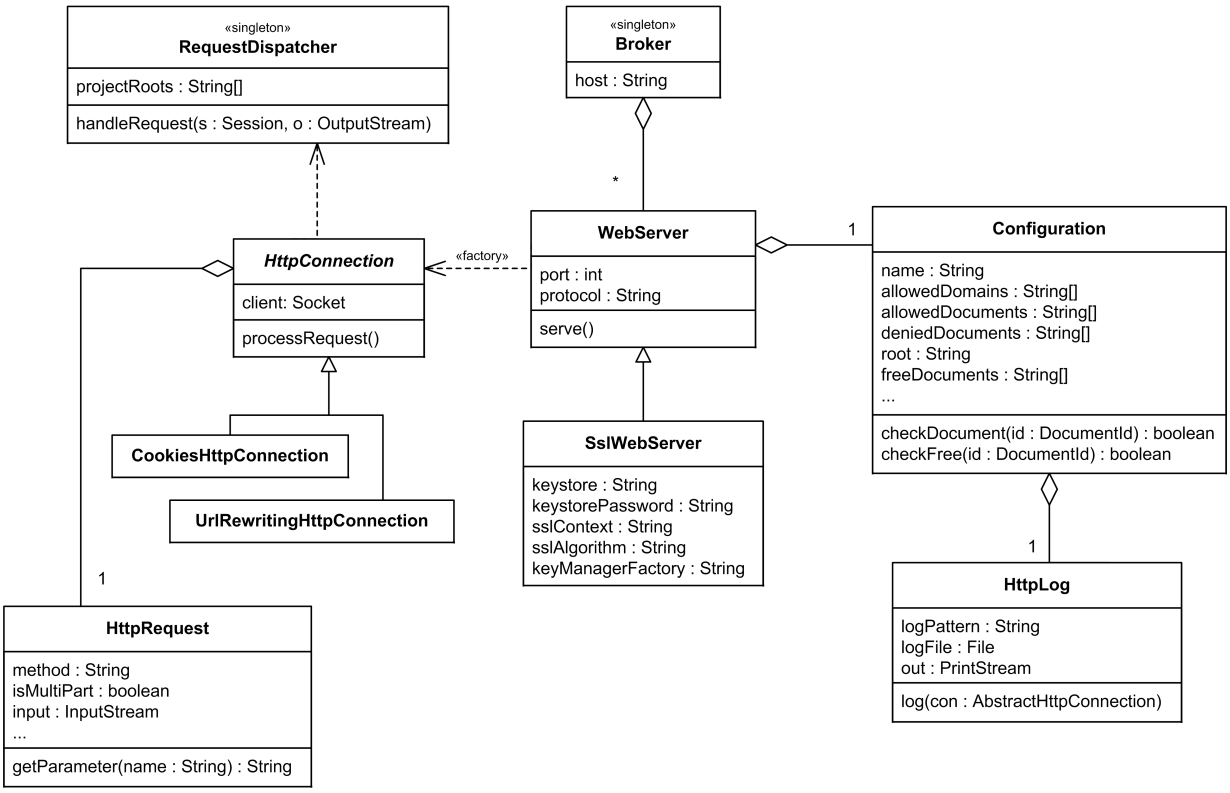


Abbildung 15: Klassendiagramm des Webserver-Frameworks

Als Reaktion auf einen eingehenden HTTP-Request erzeugt ein Webserver ein Objekt vom Typ `HttpConnection`. Es existieren zwei Implementierungen, die sich in der Art des Session-Trackings unterscheiden: `CookiesHttpConnection` und `UrlRewritingHttpConnection`. Die Eigenschaften eines Requests werden von Objekten des Typs `HttpRequest` verwaltet. Zur weitergehenden dynamischen Beantwortung von Requests existiert pro Instanz des Frameworks ein Singleton vom Typ `RequestDispatcher`.

Das vorgestellte Webserver-Framework ist ein Blackbox-Framework. Die Komponenten des Frameworks liegen vollständig implementiert vor. Um eine lauffähige Instanz des Frameworks zu erhalten müssen lediglich konkrete Implementierungen ausgewählt und konfiguriert werden. Die Stellen, an denen zum Zwecke der Instanziierung Konfigurationen stattfinden müssen sind in Abbildung 16 rot markiert. Diese Stellen werden Hot-Spots bzw. Erweiterungspunkte genannt.

Im Beispiel muss zunächst der Host, auf dem das System sich befindet, konfiguriert werden. Anschließend muss pro zu instanziierenden Server eine Implementierung (`WebServer` oder `SslWebServer`) ausgewählt werden. Für jeden Webserver müssen nun die rot eingefärbten elementaren Eigenschaften konfiguriert werden. Dies sind der Port auf dem der Server läuft sowie das verwendete Protokoll. Falls der ausgewählte Webserver vom Typ `SslWebServer` ist müssen zusätzliche Eigenschaften konfiguriert werden. Nachdem der Webserver konfiguriert ist, muss ein Objekt vom Typ `Configuration` ausgewählt und konfiguriert werden. An-

schließend muss der `HttpLog` konfiguriert werden. All dies sind Entscheidungen, die in der Sprache der Domäne „Webserver“ formuliert sind.

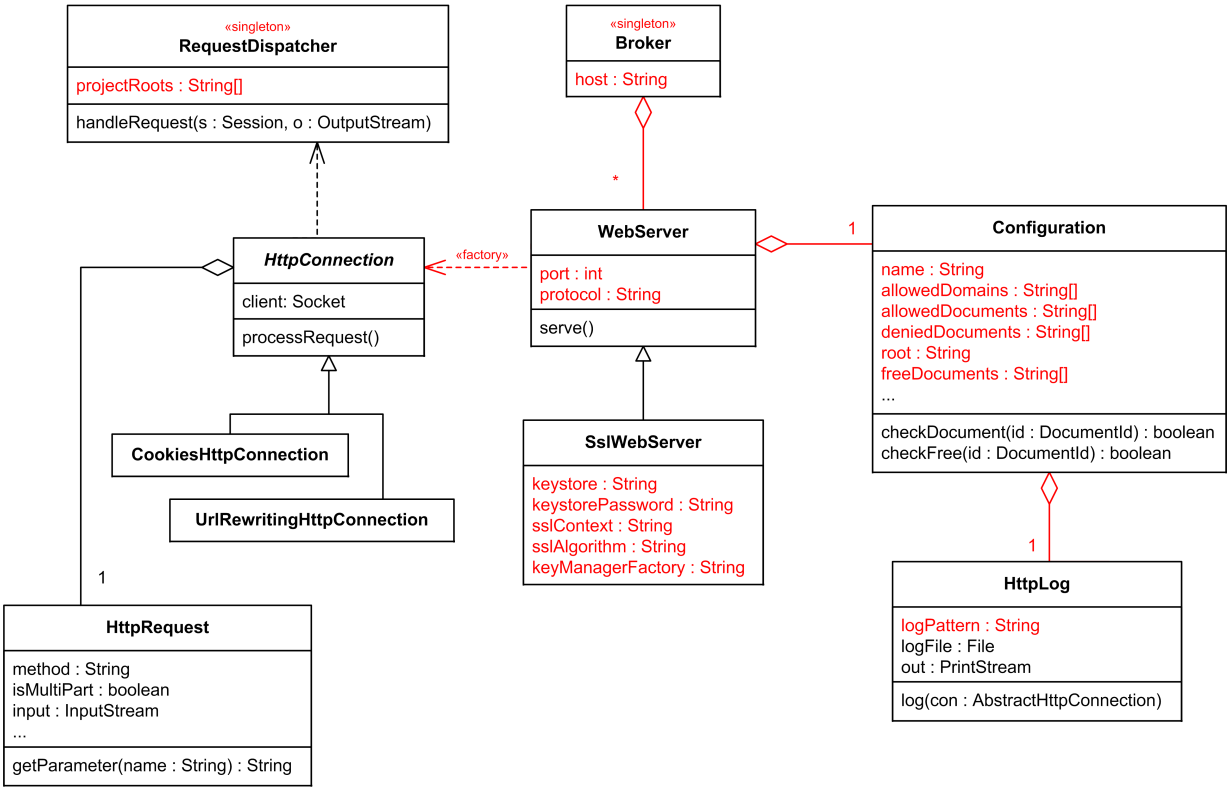


Abbildung 16: Klassendiagramm des Webserver-Frameworks mit Hot-Spots

Die Anpassung des Frameworks kann durch Java-Quellcode erfolgen:

```
Broker b = new Broker();
b.setHost("wwwmatthes.in.tum.de");
WebServer w = new WebServer();
b.addWebServer(w);
w.setPort(80);
...
```

Dies stellt bereits eine einfache Form einer internen domänenspezifischen Sprache dar. Der Abstraktionsgrad dieser DSL ist jedoch sehr gering, da die domänenspezifische Syntax nicht explizit ist (siehe Abschnitt 2.2.5, Nachteile 3 bis 5). Dass zur Konfiguration nur die in Abbildung 16 rot eingefärbten Erweiterungsmöglichkeiten zulässig sind ist nicht ersichtlich. Weiterhin sind zur Anpassung Kenntnisse der Syntax von Java notwendig – auch wenn es sich in diesem Fall um sehr stereotypen Code handelt, der nur eine kleine Teilmenge der Mächtigkeit von Java ausnutzt.

Die Nachteile dieser Form der Anpassung werden durch den Ansatz der modellgetriebenen Softwareentwicklung adressiert, der darin besteht, den Vorgang der Konfiguration durch eine formale domänenspezifische Sprache zu unterstützen. Die Konzepte der DSL sind auf die Pro-

blemdomäne zugeschnitten und verbergen die zur tatsächlichen Instanziierung notwendigen Details vor dem Programmierer. Dies führt zu einer Verringerung der Komplexität, die bei der Konfiguration wahrgenommen wird.

Die abstrakte Syntax einer DSL zur Konfiguration des Webserver-Frameworks umfasst also die in Abbildung 16 rot dargestellten Elemente. Wie in Abschnitt 2.1 auf Seite 22 dargestellt, wird eine formale Sprache zusätzlich durch eine konkrete Syntax, eine Serialisierungssyntax sowie eine Semantik charakterisiert. Die Semantik besteht in der beschriebenen Instanziierung des Webserver-Frameworks. Offen bleibt die Frage nach der konkreten Syntax, sowie der Serialisierungssyntax. Eine einfache Möglichkeit, die DSL textuell zu repräsentieren ist beispielsweise folgendermaßen:

```
host = wwwmatthes.in.tum.de

webserver.class = de.infoasset.core.server.WebServer
webserver.port = 80
webserver.protocol = http

configuration.name = internet
configuration.internet.allowedDomains = *
configuration.internet.httpLog.logFile = httplog.txt
...
```

In diesem Fall ist die konkrete Syntax gleich der Serialisierungssyntax. Es handelt sich hierbei um eine externe DSL (siehe Abschnitt 2.3.10). Zur Instanziierung des Systems wird eine zusätzliche Komponente benötigt, die die enthaltenen Informationen interpretiert und die entsprechenden Objekte instanziiert und konfiguriert. Die Vorteile der Verwendung einer DSL liegen in der Einfachheit der Konfiguration. Die Konstrukte der DSL sind auf die Domäne maßgeschneidert und ermöglichen eine fokussierte Problemlösung.

Nachteilig an der Benutzung einer externen DSL ist der initial mit dem Erstellen der DSL verbundene Aufwand. Ein weiterer Nachteil externer DSLs liegt in der fehlenden Integration mit den Konstrukten der Programmiersprache – und somit mit den Konstrukten des Frameworks. Inwiefern eine externe DSL die Erweiterungsmöglichkeiten des zu modellierenden Frameworks konsistent repräsentiert lässt sich nicht werkzeugunterstützt überprüfen.

4.1.2 Ansatz der Blackbox Introspektion

Jedes Blackbox Framework definiert implizit durch seine Erweiterungsmöglichkeiten ein Metamodell für mögliche Konfigurationen. Das Konfigurieren des Frameworks entspricht dem Modellieren entsprechend diesem Metamodell. Die Idee der Introspektion in Bezug auf Blackbox Frameworks besteht darin, Erweiterungspunkte explizit im Quelltext des Frameworks zu kennzeichnen. In einem derart gekennzeichneten Framework kann das Metamodell direkt aus dem annotierten Quelltext gewonnen werden. Ein Blackbox Framework, in dem die Erweiterungspunkte explizit gekennzeichnet sind, wird *introspektives* Blackbox Frameworks genannt. Zur Kennzeichnung werden Java-Annotationen benutzt.

Es werden also bestimmte Konstrukte der Programmiersprache mit Annotationen versehen. Beispielsweise verfügt die Klasse `Broker` über eine konfigurierbare Eigenschaft `host` und eine konfigurierbare Beziehung zu Objekten vom Typ `WebServer`. Dies manifestiert sich in einem introspektiven Blackbox Framework folgendermaßen:

```
class Broker {
  @Property(example="wwwmatthes.in.tum.de")
  String host;

  @Association
  List<WebServer> webservers;

  ...
}
```

Aus dem annotierten Quelltext lässt sich das zum Konfigurieren benötigte Metamodell extrahieren. Das Metamodell enthält sowohl Information die sich aus der Annotation ergeben, als auch Informationen über das annotierte Konstrukt. Im Beispielfall handelt es sich um eine Zeichenketteneigenschaft, die den Namen `host` hat. Diese Informationen ergeben sich aus der Signatur des Feldes. Weiterhin wird durch die Annotation der Beispielwert `wwwmatthes.in-.tum.de` beigesteuert.

Zur expliziten Definition von Erweiterungspunkten in introspektiven Blackbox Frameworks stehen bestimmte Annotationstypen in Verbindung mit annotierbaren Programmiersprachkonstrukten zur Verfügung. Dadurch wird ein Metametamodell definiert, welches die Struktur der Erweiterungsmöglichkeiten aller möglichen introspektiven Blackbox Frameworks beschreibt. Dieses Metametamodell soll im Folgenden grob umrissen werden und wird in Abschnitt 4.2 vollständig vorgestellt.

Eigenschaften und Assoziationen

Als Ergebnisse einer konzeptuellen Analyse der in Abbildung 16 markierten Hot-Spots lassen sich zwei Arten von konfigurierbaren Elementen unterscheiden:

1. **Properties**: konfigurierbare Attribute von Objekten, sowie
2. **Associations**: konfigurierbare Beziehungen zwischen Objekten.

Daraus ergibt sich ein vereinfachtes konzeptuelles Metametamodell der Erweiterungsmöglichkeiten in Blackbox Frameworks wie in Abbildung 17 dargestellt. Eine Konfiguration (`Configuration`) besteht aus genau einer konfigurierbaren Klasse (`ConfigurableClass`), die die Wurzel der Konfiguration bildet. Konfigurierbare Klassen ihrerseits können konfigurierbare Eigenschaften (`Property`) mit einem Wert und Assoziationen (`Association`) zu anderen konfigurierbaren Klassen haben. Durch Assoziationen wird eine Baumstruktur aufgespannt. Die Elemente des Metametamodells sind mit Icons versehen, die zur übersichtlichen Kennzeichnung unter anderem im Konfigurationswerkzeug verwendet werden.

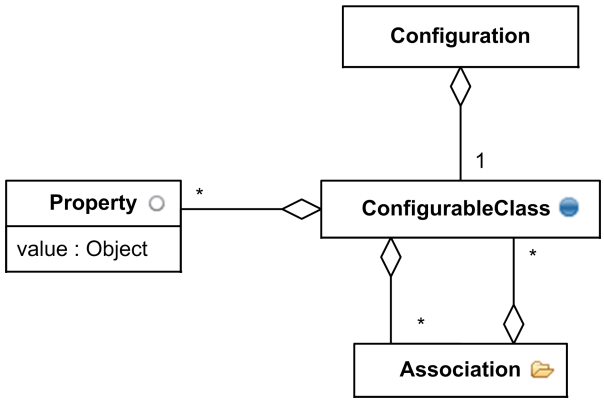


Abbildung 17: Vereinfachtes Metametamodell der Blackbox Introspektion

Durch Introspektion wird der annotierte Quelltext analysiert, und es entsteht ein Metamodell des zu konfigurierenden Frameworks. Dieses Metamodell enthält entsprechend dem Metametamodell alle konfigurierbaren Klassen, sowie deren zu konfigurierenden Assoziationen und Eigenschaften. Das Metamodell des Webserver-Frameworks ist in Abbildung 18 dargestellt. Hierin sind alle für die Konfiguration relevanten Elemente enthalten.

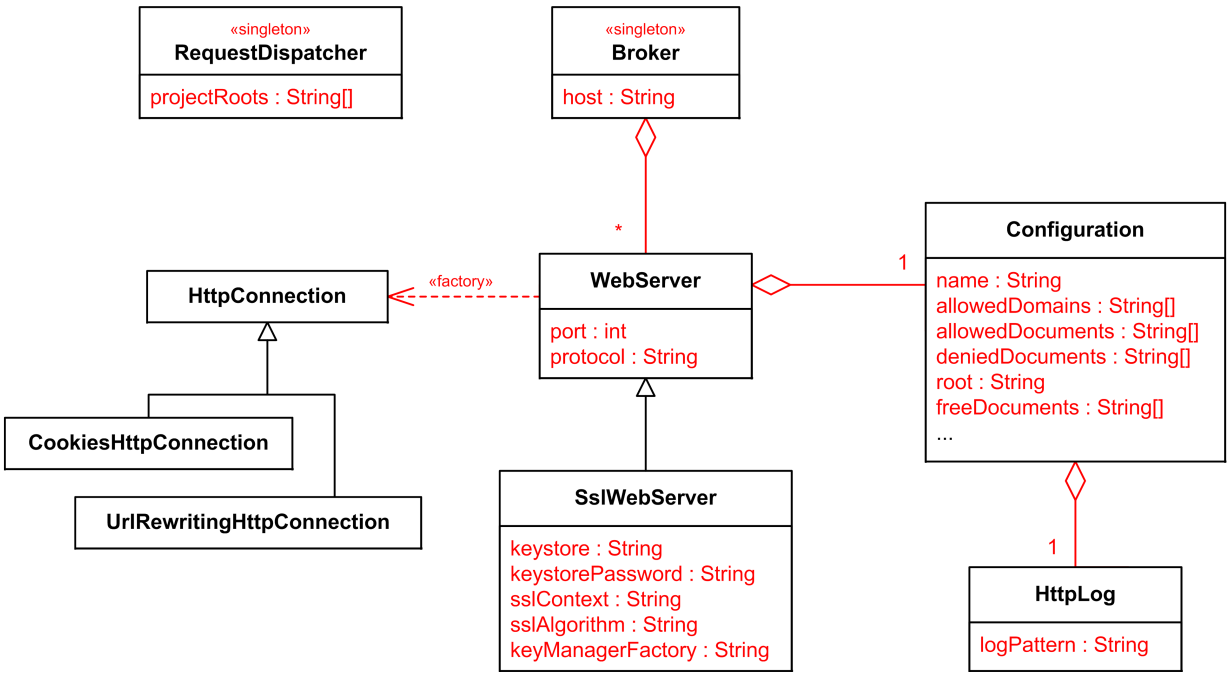


Abbildung 18: Konfigurationsmetamodell des Webserver-Frameworks

Auf Basis des Metamodells kann durch Konfiguration ein Modell erstellt werden, welches dem Metamodell entspricht und somit eine valide Instanziierung des Blackbox-Frameworks ermöglicht. Ziel des gesamten Prozesses ist ein instanziierter Objektgraph.

4.1.3 Vorgehen im Überblick

Im Rahmen dieser Arbeit wurden Werkzeuge entwickelt, die den kompletten Vorgang der Konfiguration und Instanziierung eines introspektiven Blackbox Frameworks zu unterstützen. Diese Werkzeuge basieren auf der existierenden Post-IntelliJ-IDE Eclipse (siehe hierzu den Exkurs auf Seite 24). Dies ist sinnvoll, da diese Art von Entwicklungsumgebung die Möglichkeit liefert, auf den abstrakten Syntaxbaum des Programms zuzugreifen. Es wird ein introspektives Blackbox Framework, in dem die Hot-Spots entsprechend annotiert sind, benötigt. Der gesamte Ablauf der Blackbox Introspektion ist in Abbildung 19 dargestellt. Zusammenfassend lässt sich der Ablauf in folgende Teilschritte unterteilen:

1. Gewinnen des Metamodells durch Metamodell-Introspektion,
2. Konfigurieren und speichern des Modells,
3. Öffnen und instanziieren des Objektgraphs.

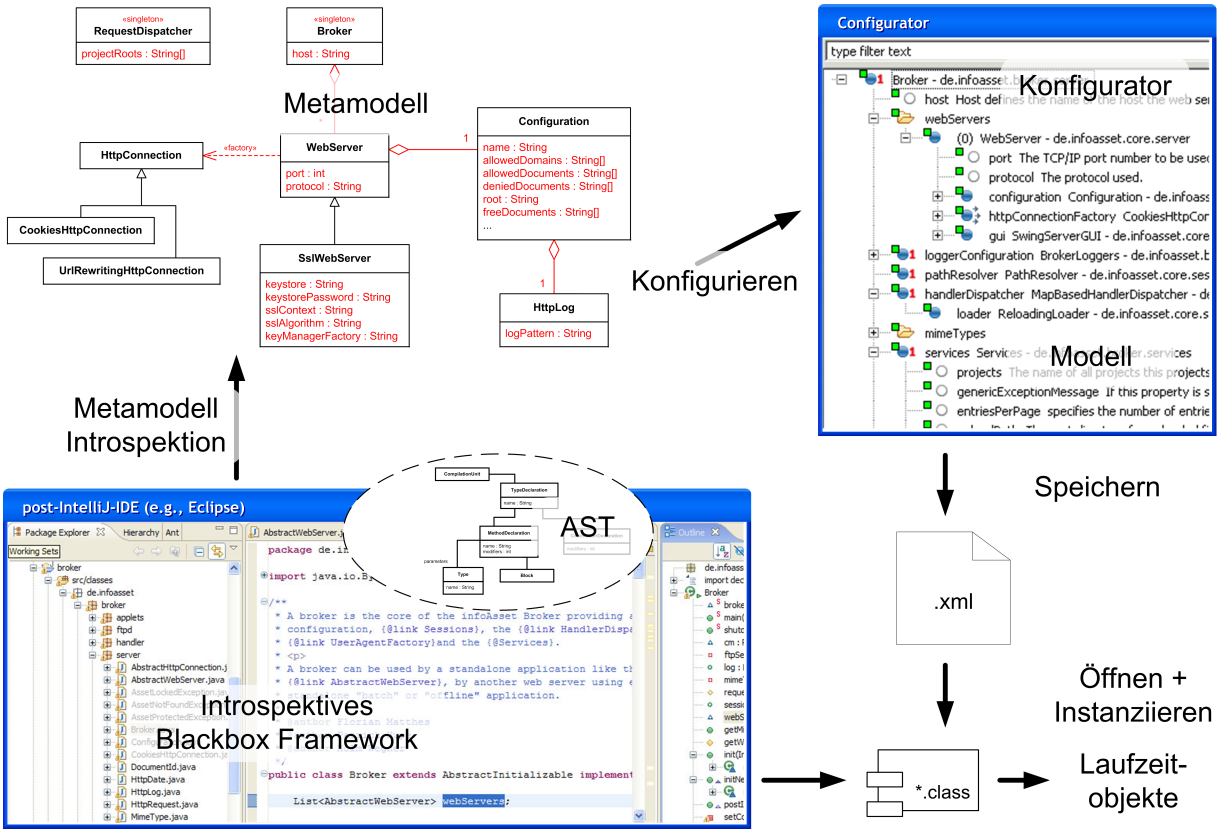


Abbildung 19: Überblick: Blackbox Introspektion

Diese Teilschritte werden im Folgenden näher vorgestellt. Zunächst wird im folgenden Abschnitt das vollständige Metametamodell der Blackbox Introspektion vorgestellt.

4.2 Metametamodell der Blackbox Introspektion

Wie bereits in Abbildung 17 dargestellt, enthält das Metametamodell der Blackbox Introspektion konfigurierbare Klassen und deren konfigurierbare Eigenschaften und Beziehungen. Die konfigurierbaren Elemente werden im Quelltext direkt durch Annotationen gekennzeichnet. Entsprechend den vorgestellten Primitiven existieren zwei Annotationstypen: `Property` und `Association`. In diesem Abschnitt werden Signatur und Semantik der eingeführten Annotationstypen detailliert vorgestellt.

Eine vereinfachte Version des Metametamodells wurde in [BM06] vorgestellt.

4.2.1 Eigenschaften

Die einfachste Art der Konfiguration ist das Spezifizieren einer elementaren Eigenschaft. Mit elementar ist gemeint, dass der Wert der Eigenschaft sich nicht wiederum aus mehreren Werten zusammensetzt. Konkret wird die Konfiguration aller primitiven Java-Datentypen sowie von Werten des Typs `java.lang.String` unterstützt.

Wie bereits erwähnt, werden die zu konfigurierenden Attribute durch eine Annotation vom Typ `Property` gekennzeichnet. Die Verwendung der Annotation wird durch Introspektion analysiert und es wird ein Metamodell aufgebaut. Entsprechend dem Metamodell werden Werte konfiguriert und zur Instanziierung gesetzt. Das Setzen der Werte geschieht durch Reflection [Su99a] direkt auf dem annotierten Element und kann auf zwei verschiedene Arten geschehen. Entweder wird der Wert eines Attributs direkt gesetzt. In diesem Fall wird also ein Feld einer Klasse annotiert:

```
@Property
int host;
```

Im anderen Fall wird eine Methode mit dem konfigurierten Wert als Parameter auf dem zu konfigurierenden Objekt aufgerufen. In diesem Fall wird also die Methode (typischerweise eine Setter-Methode) annotiert:

```
int host;

@Property
void setHost(int host) {
  this.host = host;
}
```

Beide Arten sind äquivalent. Soll lediglich wie im Beispiel gezeigt (ohne Seiteneffekt) eine Eigenschaft konfiguriert werden, ist sicherlich die Annotation des Attributes direkt vorzuziehen. Im Falle der Konfiguration über eine Setter-Methode besteht jedoch die Möglichkeit, zusätzliche Aktionen zum Zeitpunkt des Setzens der Eigenschaft auszuführen. So ist es zum Beispiel einfach möglich, komplexere Datentypen aus einfachen Strings zu erhalten. Ein Feld von Strings ließe sich beispielsweise folgendermaßen konfigurierbar gestalten:

```
String[] projects

@Property
void setProjects(String value) {
  StringTokenizer tokenizer = new StringTokenizer(value,
",");
  projects = new String[tokenizer.countTokens()];
  for(int i=0; i<projects.length; i++) {
    projects[i] = tokenizer.nextToken();
  }
}
```

Im aufgeführten Beispiel ist sofort ersichtlich, dass es wünschenswert ist, bestimmte Meta-Informationen für denjenigen Benutzer, der eine Konfiguration durchführt, zusätzlich zur Signatur des zu konfigurierenden Attributes zu hinterlegen. Dass es sich nämlich in diesem Fall nicht um eine elementare Eigenschaft, sondern um die Elemente eines durch "," separierten Felds handelt, ist für die introspektive Analyse nicht ersichtlich. Somit ist diese Information zum Konfigurationszeitpunkt ebenfalls nicht verfügbar. Zu diesem Zweck gibt es die Möglichkeit, Annotationen mit Werten zu versehen [BL04]. Eine Anreicherung der Annotation `Property` um die Eigenschaft `description` ermöglicht eine informelle Beschreibung der konfigurierten Eigenschaft:

```
@Property(description=
  "specify all projects separated by \",\" here")
void setProjects(String value) {...
```

Die angeführte Beschreibung ist zum Konfigurationszeitpunkt verfügbar. Das Anreichern der Konfigurationsannotationen ist optional. Alle im Folgenden vorgestellten zusätzlichen Meta-Informationen sind mit geeigneten Werten vorbelegt (siehe Tabelle 3).

Eine weitere wünschenswerte Meta-Information ist das Spezifizieren von Beispielwerten:

```
@Property(example="80",
  description="the port this server runs on")
int port;
```

Hierbei stellt sich die Frage, inwieweit man die angegebenen Beispielwerte als Defaultwert ansieht, und ob man eine Bestätigung des Beispielwerts zum Konfigurationszeitpunkt erzwingt. Dies lässt sich ebenfalls über die Annotationseigenschaft `exampleIsDefault` steuern:

```
@Property(example="80",
  exampleIsDefault=true,
  description="the port this server runs on")
int port;
```

Die Annotationseigenschaft `exampleIsDefault` ist standardmäßig auf `false` gesetzt.

In Tabelle 3 sind alle Eigenschaften der Annotation `Property` zusammengetragen. Interessant ist die Spalte "Wodurch definiert?". Die bisher vorgestellten Meta-Informationen wurden durch als Wert der Annotation spezifiziert. Im Unterschied dazu werden der Name sowie Typ des annotierten Elements direkt aus der Signatur gewonnen. Der Wert der Eigenschaft, welcher zur Laufzeit zugewiesen werden soll wird wiederum erst zum Konfigurationszeitpunkt zugewiesen.

Name	***Typ***	***Wodurch definiert?***	***Default-Wert***	***Kommentar***
`description`	`String`	Wert der Annotation	Leerer String	Ist rein deskriptiv.
`example`	`String`	Wert der Annotation	Leerer String	
`exampleIsDefault`	`boolean`	Wert der Annotation	`false`	Entscheidet, ob der Beispielwert als `value` übernommen wird.
`name`	`String`	Signatur des annotierten Elements	–	Wird für die Instanziierung benötigt.
`type`	`String`	Signatur des annotierten Elements	–	Z.B.: String, int, boolean, ...
`isMethod`	`boolean`	Signatur des annotierten Elements	–	
`value`	`Object`	Konfiguration	–	Umwandlung in Zieltyp erfolgt durch automatische Typanpassung.
`isObservable`	`boolean`	Wert der Annotation	`false`	Entscheidet, ob die Eigenschaft zur Laufzeit beobachtet und gesetzt werden kann (siehe Abschnitt 4.6.1).

Tabelle 3: Spezifikation der Annotation `Property`

4.2.2 Assoziationen

Zur Anpassung eines Blackbox Frameworks müssen neben elementaren Eigenschaften auch Assoziationen zwischen Klassen konfiguriert werden. Im Rahmen dieser Arbeit werden ausschließlich binäre Assoziationen betrachtet, da es lediglich für diesen Assoziationstyp eine un-

mittelbare Implementierungsmöglichkeit in Java gibt. Assoziationen werden in Java durch Attribute, die Referenzen auf Objekte enthalten, implementiert.

Zur Kennzeichnung einer konfigurierbaren Assoziation existiert die Annotation `Association`. Analog zur Konfiguration von Eigenschaften gibt es einerseits die Möglichkeit, das Attribut direkt zu annotieren:

```
class WebServer {
  @Association
  Configuration configuration;
  ...
```

Die andere Möglichkeit besteht darin, eine entsprechende Setter-Methode zu kennzeichnen:

```
Configuration configuration;

@Association
void setConfiguration(Configuration c) {
  this.configuration = c;
}
```

In beiden Fällen soll eine Beziehung zu genau einem Objekt vom Typ `Configuration` hergestellt werden. Die Multiplizität der Assoziation ist in diesem Fall offensichtlich. Falls eine Beziehung zu mehreren Objekten definiert werde soll muss, müssen diese Objekte in einer Menge aufbewahrt werden. Zu diesem Zweck existiert in Java die so genannte Collection-API. Unter Benutzung eines Containers vom Typ `List` lässt sich eine Assoziation zu mehreren Objekten beispielsweise so definieren:

```
class Broker {
  @Association
  List<WebServer> webServers;
}
```

In diesem Fall wird eine Liste von Objekten des Typs `WebServer` verwaltet. Eine konfigurierbare Assoziation kann also genau eine oder mehrere Referenzen enthalten. Im Falle einer mehrwertigen Assoziation ist es sinnvoll, die Multiplizität des beteiligten Assoziationspartners genauer zu spezifizieren. Zu diesem Zweck verfügt die Annotation `Association` über die Eigenschaften `lowerBound` und `upperBound`. Eine Liste von Webservern, die mindestens ein Element enthält wird also folgendermaßen spezifiziert:

```
@Association(lowerBound=1)
List<WebServer> webServers;
```

Des Weiteren kann man den Namen der Assoziation, sowie Namen der Assoziationsenden über die Eigenschaften `name`, `thisEndName` und `thatEndName` spezifizieren. Dies ist sinnvoll, wenn man das durch Introspektion gewonnene Metamodell in einem UML-Klassendiagramm graphisch darstellen möchte.

Exkurs: Assoziationen in UML

Binäre Assoziationen werden in UML durch einfache Verbindungslinien zwischen den beteiligten Klassen dargestellt. Um den inhaltlichen Gehalt des Modells zu erhöhen kann man der Assoziation einen Namen zuordnen. Des Weiteren können die Enden der Assoziation mit Namen versehen werden. Eine wesentliche Eigenschaft einer Assoziation ist, wie viele Objekte der beteiligten Klassen an ihr teilnehmen können. Dies wird durch die Multiplizität der Assoziationsenden spezifiziert. Für eine detaillierte Diskussion von Assoziationen in UML sei auf [Hi05] verwiesen.

Eine wichtige Art von Assoziationen ist die *Aggregation*. Aggregationen werden landläufig auch als „Teil-Ganzes-Beziehungen“ oder „Besteht-aus-Beziehung“ bezeichnet. Es besteht also ein asymmetrisches Verhältnis zwischen nicht gleichwertigen Partnern. Aggregationen werden in UML durch eine Raute auf Seite des Ganzen (der Aggregatklasse) graphisch dargestellt. Aggregationen können benutzt werden, um zusammengesetzte Objekte, die auch selbst wieder als Teil in zusammengesetzten Objekten vorkommen können, zu modellieren [Hi05]. Eine weitere für diese Arbeit wichtige Eigenschaft einer Aggregation besteht darin, dass auf der Ausprägungsebene keine Zyklen auftreten dürfen. Ein Objekt kann also keine Komponente von sich selbst sein. In Abbildung 22 handelt es sich beispielsweise bei der Beziehung zwischen den Klassen `Broker` und `WebServer` um eine Aggregationsbeziehung.

Die exakte Semantik von Assoziationen im Unterschied zu Aggregationen in UML ist Gegenstand umfangreicher Diskussionen [HB99, Al03]. Im Rahmen dieser Arbeit stellt sich die Frage, wie Assoziationsbeziehungen bzw. Aggregationsbeziehungen auf der Ebene der konkreten objektorientierten Programmiersprache Java realisiert werden.

Im Konfigurationsmetamodell des Webserver-Frameworks (Abbildung 18) wurden ausschließlich Aggregationsbeziehungen verwendet. Dies ist im Sinne der Modellierung zusammengesetzter Objekte zu verstehen. Die Modellierung des Systems beginnt bei der Klasse `Broker`, die ihrerseits aus Instanzen der Klassen `WebServer` und `MimeType` zusammengesetzt ist. Durch Aggregationsbeziehungen wird zur Laufzeit eine Baumstruktur erzeugt, welche die Eigenschaft der Zyklenfreiheit einhält. Es besteht ebenfalls eine klare Unterscheidung zwischen den an der Beziehung beteiligten Partnern. Objekte vom Typ der Aggregatklasse enthalten das annotierte Element und verwalten Teile vom Typ des Aggregationspartners. Genauer betrachtet, handelt es sich bei den bisher vorgestellten durch einen Annotationstyp `Association` gekennzeichneten Beziehungen also um Aggregationsbeziehungen mit der Eigenschaft, dass pro konfigurierte Teil-Komponente eine neue Instanz erzeugt wird.

ObjectFactory

Wie der aufmerksame Leser bereits gemerkt hat, ist die Beziehung zwischen `WebServer` und `HttpConnection` im Metamodell in Abbildung 18 durch den Stereotyp `factory` gekennzeichnet. Konzeptuell ist diese Beziehung von allen anderen bezüglich des Zeitpunktes der Instanziierung der Komponentenobjekte verschieden. Im Fall einer „normalen“ Aggregationsbeziehung werden die konfigurierten Objekte zum Zeitpunkt der Instanziierung des Frameworks erzeugt.

Im Fall der Beziehung zwischen `WebServer` und `HttpConnection` ist dies nicht das gewünschte Verhalten. Vielmehr benötigt der Webserver zum Zeitpunkt der Instanziierung des Frameworks keine Instanz vom Type `HttpConnection`, sondern eine Factory [Ga95], mit deren Hilfe jederzeit neue Instanzen erzeugt werden können. In Abbildung 20 ist das Klassendiagramm der generischen Factory dargestellt.

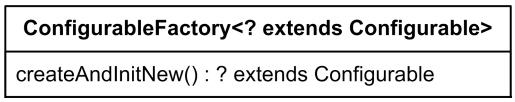


Abbildung 20: Klassendiagramm: `ConfigurableFactory`

Die Beziehung wird also folgendermaßen im Quelltext spezifiziert:

```
@Association
ConfigurableFactory<HttpConnection> connectionFactory;
```

Das Aggregat erhält also zur Laufzeit nicht direkt ein Objekt vom gewünschten Typ, sondern eine Factory mit der sich jederzeit Objekte neu erzeugen lassen. Das Erzeugen von Objekten vom Typ `HttpConnection` erfolgt dann beispielsweise folgendermaßen:

```
void run() {
  try {
    ServerSocket server = new ServerSocket(port);
    for(;;) {
      connectionFactory.createAndInitNew().init(this);
    }
  }
}
```

Die Unterscheidung, ob es sich um eine Factory oder eine „normale“ Beziehung handelt ist anhand der Signatur des zu konfigurierenden Elements möglich.

Alle Eigenschaften der Annotation `Association` sind in Tabelle 4 zusammengefasst, sowie im vollständigen Metametamodell in Abbildung 21 dargestellt.

Name	*Typ*	*Wo definiert?*	*Default-Wert*	*Kommentar*
`name`	`String`	Wert der Annotation	Leerer String	rein deskriptiv
`thisEndName`	`String`	Wert der Annotation	Leerer String	rein deskriptiv
`thatEndName`	`String`	Wert der Annotation	Leerer String	rein deskriptiv
`lowerBound`	`int`	Wert der Annotation	`0`	
`upperBound`	`int`	Wert der Annotation	Im Falle einer Liste ∞; ansonsten `1`.	
`type`	`String`	Signatur des annotierten Elements	–	
`isList`	`boolean`	Signatur des annotierten Elements	–	
`isFactory`	`boolean`	Signatur des annotierten Elements	–	

Tabelle 4: Spezifikation der Annotation `Association`

4.2.3 Konfigurierbare Klassen

Eine konfigurierbare Klasse ist prinzipiell eine gewöhnliche Klasse mit konfigurierbaren Eigenschaften oder Beziehungen. Um eine effiziente Analyse aller konfigurierbaren Klassen zu ermöglichen ist es jedoch von Vorteil, eine einfachere Identifikation zu ermöglichen. Zu diesem Zweck existiert das Interface `Configurable`, welches alle konfigurierbaren Klassen implementieren müssen. Es handelt sich hierbei um ein reines Markerinterface, welches keine eigenen Methoden definiert.

Exkurs: Singleton-Muster

Bei der Organisation komplexer objektorientierter Softwaresysteme spielt das Singleton-Muster eine große Rolle [Ga95]. Es stellt einerseits sicher, dass es genau ein Objekt von einem bestimmten Typ gibt und stellt andererseits einen globalen Zugriffspunkt auf das Objekt zur Verfügung. Die Umsetzung des Singleton-Musters erfolgt in Java unter Verwendung öffentlicher statischer Variablen:

```
public class Broker {
  public static Broker INSTANCE() {
    return instance;
  }

  private static Broker instance;
  ...
```

Die Unterstützung des Singleton-Musters ist für die Konfiguration von Blackbox Frameworks essentiell. Im Webserver-Framework sind beispielsweise die Klassen `Broker` und `RequestDispatcher` als Singleton realisiert.

Zu diesem Zweck muss es zunächst möglich sein, eine konfigurierbare Klasse als Singleton zu identifizieren. Dies ist leicht möglich, indem nach einer statischen Methodenimplementierung, welche ein Objekt vom Typ der Klasse zurückliefert gesucht wird. Zum Instanziierungszeitpunkt wird ein privates statisches Feld vom entsprechenden Typ gesucht, welches mit dem entsprechend konfigurierten Objekt initialisiert wird. Darauf wird im Abschnitt „Ablauf der Instanziierung von introspektiven Blackbox Frameworks“ näher eingegangen. Das somit vollständige Metametamodell der Blackbox Introspektion ist in Abbildung 21 dargestellt.

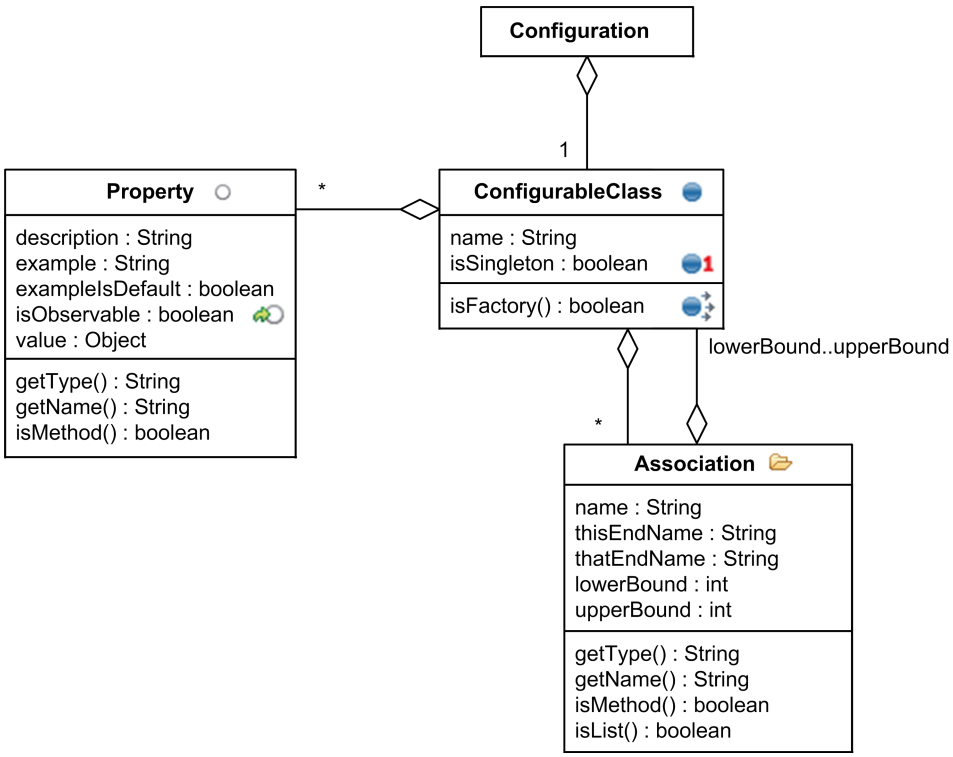


Abbildung 21: Vollständiges Metametamodell der Blackbox Introspektion

4.3 Konfigurator

In diesem Abschnitt wird der Vorgang der eigentlichen Konfiguration eines Blackbox Frameworks besprochen. Als Basis zur Erstellung einer Konfiguration dient das durch Introspektion erzeugte Metamodell des jeweiligen Frameworks (siehe beispielsweise Abbildung 18 auf Seite 58). Hierauf aufbauend kann werkzeuggestützt eine Konfiguration erstellt werden, die dem Metamodell entspricht und somit eine valide Instanziierung des Frameworks ermöglicht.

Die Betonung liegt auf dem *werkzeuggestützten* Konfigurieren. Dieses Werkzeug wird im Folgenden „Konfigurator“ genannt. Der Konfigurator existiert in verschiedenen Ausprägungen. Er

existiert sowohl als Plugin integriert in die Eclipse-IDE, als auch Standalone-Anwendung. Das Hauptaugenmerk der folgenden Beschreibung liegt auf der in Eclipse integrierten Version. Ein Screenshot des Plugins ist in Abbildung 23 dargestellt. Dargestellt ist eine Konfiguration der Klasse `Broker`.

Der Bildschirmbereich des Konfigurators ist in drei Bereiche unterteilt. Oben befindet sich die Toolbar des Konfigurators. Im linken unteren Bereich ist eine Baumansicht angeordnet, die die gesamte aktuelle Konfiguration darstellt. Im rechten unteren Bereich befindet sich die Elementanzeige, eine Detailansicht auf das aktuelle Element. In der Detailansicht ist ein Zugriff auf alle Aktionen zur Manipulation der Konfiguration möglich.

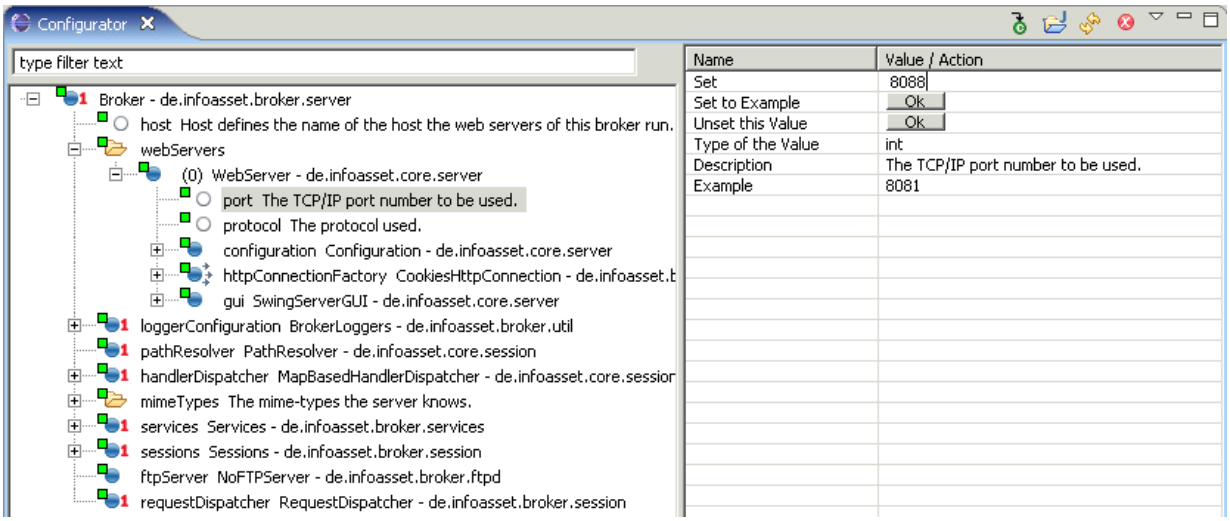


Abbildung 22: Konfigurator-Plugin

4.3.1 Baumansicht und Elementanzeige

Die Baumansicht zeigt eine Sicht auf die komplette bearbeitete Konfiguration an. Entsprechend dem Metametamodell der Blackbox Introspektion (siehe Abbildung 21) handelt es sich hier um Elemente vom Typ `ConfigurableClass`, `Property` und `Association`. Die Struktur des Baumes wird von den in Aggregationen beteiligten Elementen aufgespannt (siehe Abschnitt 4.2.2). Zur Verbesserung der Übersichtlichkeit wurden die Elemente mit den im Metametamodell dargestellten Symbolen gekennzeichnet. So handelt es sich beispielsweise beim Wurzelelement in Abbildung 22 um ein Element vom Typ `ConfigurableClass`. Genauer liegt ein Singleton vor, was durch die rote „1“ visualisiert wird. Der Klassenname des Elements ist `Broker` und es liegt im Package `de.infoasset.broker.server`.

In Abbildung 23 ist das Element `port` der Klasse `WebServer` in der Baumansicht selektiert und es sind alle verfügbaren Informationen und Aktionen in der Elementanzeige sichtbar. Man erfährt, dass es sich um eine Eigenschaft vom Typ `int` handelt, die den TCP/IP-Port auf dem die Webserver läuft konfiguriert und dass der Beispielwert `8081` ist. Mittels der verfügbaren

Aktionen kann man den Wert der Eigenschaft eingeben und setzen, den Wert des Beispiels übernehmen, sowie das Setzen der Eigenschaft rückgängig machen.

Eine besondere Eigenschaft des Konfigurators besteht darin, dass es jederzeit möglich ist, aus der Konfigurationssicht zurück in den Quelltext zu navigieren. Jedes Element der Baumansicht entspricht einer durch Annotation gekennzeichneten Stelle im Quelltext des zu konfigurierenden Blackbox Frameworks. Dementsprechend ist es stets möglich, zu der Stelle im Code zu navigieren, die das entsprechende Element spezifiziert. In Abbildung 23 wurde beispielhaft zur Definition des Attributs `port` der Klasse `WebServer` navigiert.

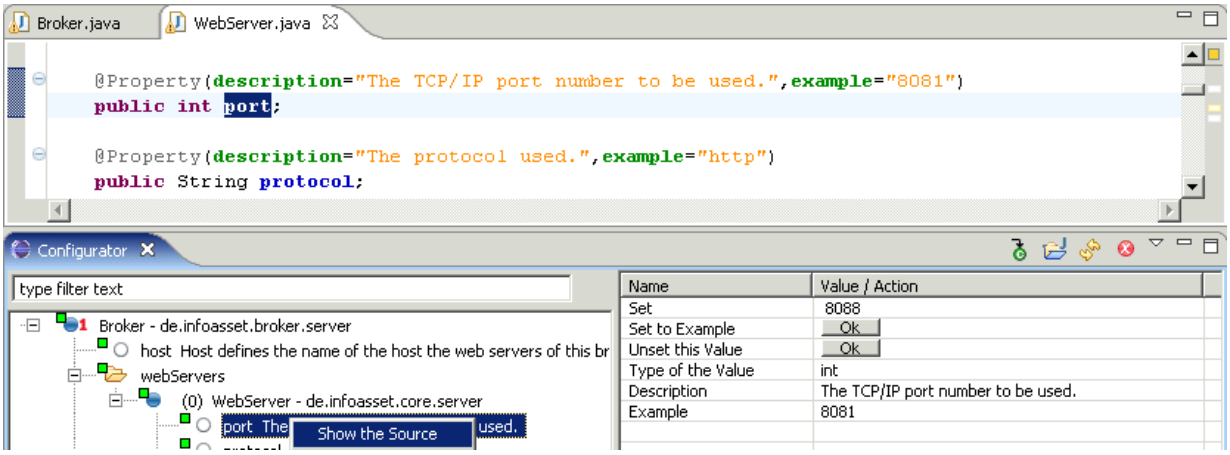


Abbildung 23: Konfigurator: Property mit Aktionen

Alle bisher dargestellten Modelle wurden bereits vollständig konfiguriert. Dies ist an den grünen Dekorationen erkennbar. Zu Beginn einer Konfiguration liegen die Modellelemente unkonfiguriert vor, was durch rote Dekorationen wie in Abbildung 24 dargestellt visualisiert wird.

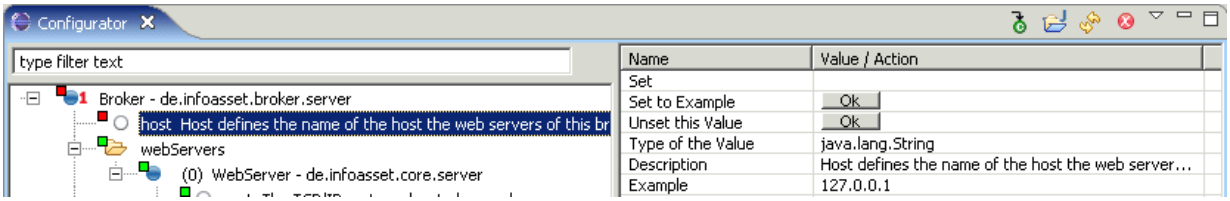


Abbildung 24: Unkonfigurierte Property

In Abbildung 25 ist die Konfigurationsansicht der „one-to-many“-Beziehung `webServers` zu sehen. Durch die Aktion *New Configurable Class* auf der rechten Seite können neue Komponenten hinzugefügt werden. Das annotierte Element des Frameworks spezifiziert eine Liste von Elementen vom Typ `WebServer`. Es wurden zwei konkrete Typen gefunden, die diesem Typ genügen, welche beide in einer Drop-down-Liste zur Auswahl angeboten werden. Da die Beziehung mindestens ein Element enthalten muss, ist sie als unvollständig dekoriert.

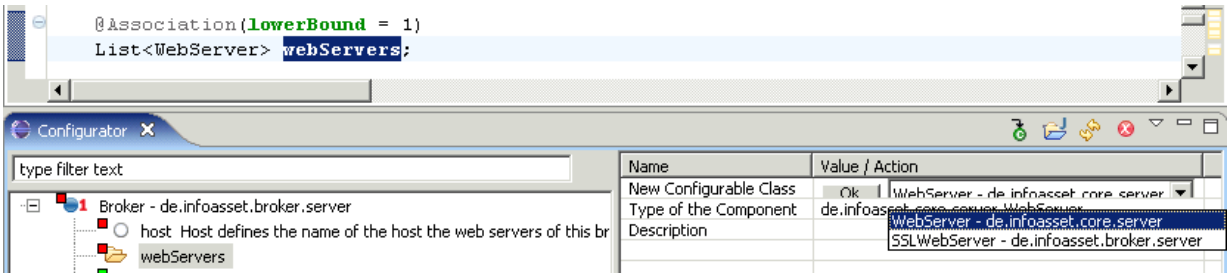


Abbildung 25: "One-to-many"-Beziehung

Wurde wie dargestellt, eine neue Komponente vom Typ `WebServer` eingefügt, so ergibt sich die in Abbildung 26 dargestellte Situation. Die Komponente ist an Position 0 und kann durch die Aktion *Remove This* wieder entfernt werden.

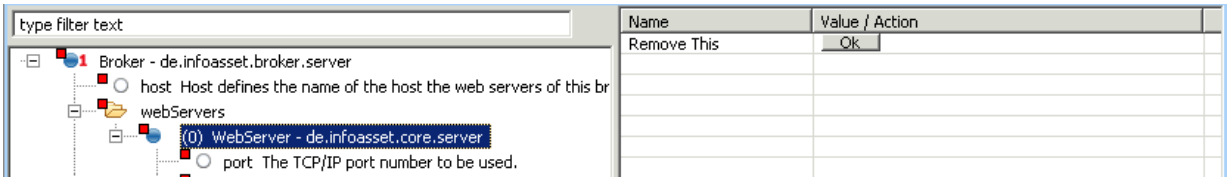


Abbildung 26: Ein Element einer "One-to-many"-Beziehung

Generell werden zur Visualisierung der Elemente die im Metametamodell in Abbildung 21 eingeführten Icons verwendet. So handelt es sich in Abbildung bei der Klasse `Broker` um ein Singleton, welches durch das entsprechende Icon veranschaulicht wird. In Abbildung 27 ist die Konfiguration einer Factory zur Erzeugung von Objekten vom Typ `CookiesHttpConnection` dargestellt.

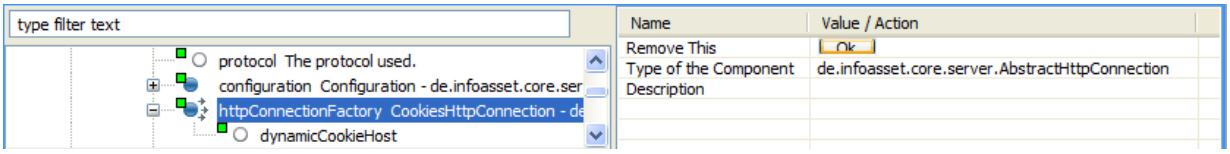


Abbildung 27: Visualisierung einer `ConfigurableFactory`

Eine wesentliche Eigenschaft des Konfigurators besteht darin, dass man stets auf einen Blick die syntaktische Validität der aktuellen Konfiguration überprüfen kann. Handelt es sich um einen gültige Konfiguration, so ist das Wurzelelement der Baumansicht grün dekoriert. Die Regeln zur Berechnung der Validität sind recht einfach. Generell kann ein Element nur gültig sein, wenn alle Kinderelemente ihrerseits gültig sind. Zusätzlich gilt: eine Eigenschaft ist gültig, wenn irgendein Wert konfiguriert wurde. Eine Assoziation ist gültig wenn die Anzahl der Kinderelemente der durch `lowerBound` und `upperBound` spezifizierten Multiplizität genügt.

4.3.2 Toolbar

Zusätzlich zur direkten Manipulation einer Konfiguration in der Baumansicht und Elementanzeige verfügt das Plugin über weitere Funktionalitäten, die durch eine Toolbar zur Verfügung gestellt werden (siehe Abbildung 28). Dabei stehen die folgenden Funktionen zur Verfügung (von links nach rechts):

- Analyse der aktuell im Editor geöffneten Klasse,
- Analyse aller konfigurierbaren Klassen des aktuellen Projekts und Speichern des Metamodells,
- Löschen des aktuellen Modells und Freigabe der beanspruchten Ressourcen,
- Markieren aller im aktuellen Modell enthaltenen Fehler.

Im Aufklappmenü, welches ebenfalls in Abbildung 28 dargestellt ist, befinden sich Funktionen zum Speichern und Öffnen von existierenden Konfigurationen.

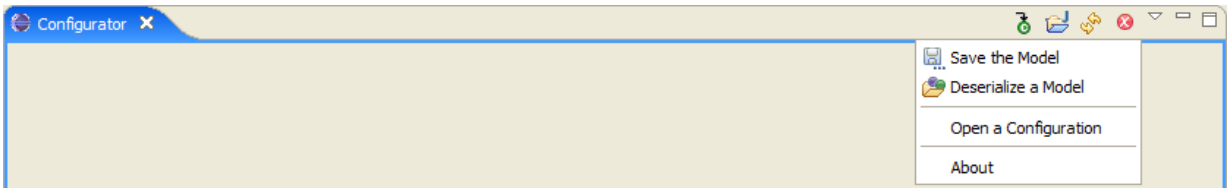


Abbildung 28: Toolbar des Werkzeugs

4.4 Einlesen von abgespeicherten Konfigurationen

Mit dem vorgestellten Konfigurator lassen sich Modelle erstellen, die das im Framework-Kern annotierte Metamodell instanziieren. Alle dargestellten Modellelemente sind Ergebnis einer Metamodell-Introspektion und sind somit mit dem Framework-Kern integriert. Die im Konfigurator durchgeführten Konfigurationsschritte setzen elementare Werte oder veranlassen die Introspektion weiterer Klassen.

Wie in Abbildung 28 dargestellt, lassen sich diese Modelle abspeichern und zu einem späteren Zeitpunkt wieder öffnen. Zum Öffnen existieren zwei Funktionen. Einerseits lässt sich ein existierendes Modell *deserialisieren*, was die originalgetreue Wiederherstellung des Modells veranlasst. In diesem Fall wird keine Introspektion durchgeführt. Das deserialisierte Modell ist in diesem Fall nicht mit dem Framework-Kern integriert – es lassen sich auch Modelle deserialisieren, die nicht mehr existierende Klassen und Eigenschaften repräsentieren. Die zweite existierende Funktion ermöglicht, gespeicherte Konfigurationen zu öffnen und neu auf den existierenden Framework-Kern anzuwenden. In diesem Fall wird Metamodell-Introspektion durchgeführt und die Werte der gespeicherten Konfiguration werden auf die durch Introspektion erzeugten Modellelemente angewandt. Intuitiv kann man sich diesen Vorgang so vorstellen, dass der Reihe nach die Eigenschaften und Assoziationen des abgespeicherten Modells im

Konfigurator angewandt werden. Dies führt zu einem mit dem Framework integrierten Konfigurationsmodell, welches die Entscheidungen des abgespeicherten Modells repräsentiert.

4.5 Migration von Konfigurationen

Im Konfigurator geöffnete Modelle sind mit dem Framework integriert. Werden Modelle gespeichert und zu einem späteren Zeitpunkt geöffnet, ist es möglich, dass sich die Konfigurationsmöglichkeiten des Frameworks in der Zwischenzeit weiterentwickelt haben. Das Framework definiert somit ein verändertes Metamodell. Dies führt zu der Fragestellung, wie mit Inkonsistenzen die sich aus diesem Szenario ergeben, umgegangen wird (siehe Abschnitt 2.3.9).

Aufgrund des vorgestellten Mechanismus, durch den abgespeicherte Konfigurationen geöffnet werden, ist zunächst sichergestellt, dass nach dem Öffnen eine syntaktisch korrekte Konfiguration vorliegt, die mit dem Framework integriert ist. Wurden beispielsweise Konfigurationsmöglichkeiten entfernt, so ist ein Konfigurationsmodell ohne manuelle Änderungen nach dem Öffnen wieder mit dem Framework konsistent. Wurden Konfigurationsmöglichkeiten hinzugefügt, so sind diese Stellen leicht erkennbar und müssen konfiguriert werden. Komplizierter sind Fälle, in denen Konfigurationsmöglichkeiten verändert wurden. Dies umfasst das Umbenennen oder Verschieben von Eigenschaften oder Assoziationen. Für den Konfigurator sind diese Operationen nicht als solche erkennbar und erscheinen wie das voneinander unabhängige Löschen und Hinzufügen von Elementen. Demzufolge ist die in der abgespeicherten Konfiguration enthaltene Entscheidung nicht an der neuen Stelle verfügbar und wird durch eine fehlende Konfiguration sichtbar. Handelt es sich um wenige Modelle in denen Eigenschaften umbenannt wurden, ist es sicherlich sinnvoll, diese fehlenden Konfigurationen manuell nachzutragen. Eine andere Möglichkeit besteht darin, Veränderungen direkt in der XML-Repräsentation vorzunehmen. Wurde beispielsweise eine Assoziation umbenannt, so kann der komplette Teilbaum unterhalb dieses Elements nicht geöffnet werden. In diesem Fall ist es sinnvoll, die Umbenennung in der gespeicherten Konfiguration vorzunehmen.

Zusammenfassend hilft die momentan existierende Werkzeugunterstützung beim Finden von Inkonsistenzen, falls sich das Framework weiterentwickelt hat. Dies ist aufgrund der Integration der domänenspezifischen Sprache möglich. Gefundene Inkonsistenzen müssen manuell behoben werden – entweder im Konfigurator oder direkt in der XML-Repräsentation. Theoretisch ist eine Weiterentwicklung der Werkzeugunterstützung möglich, die Modelle anpasst, falls das Framework verändert wird. Die Grundidee hierbei besteht darin, Veränderungen der annotierten Erweiterungsmöglichkeiten automatisch auf existierende Modelle anzuwenden.

4.6 Ablauf der Instanziierung von introspektiven Blackbox Frameworks

Mit Hilfe des Konfigurators lassen sich Konfigurationen erstellen, die dem durch Introspektion gewonnenen Metamodell des Frameworks entsprechen. In einem letzten Schritt wird das Framework entsprechend der Konfiguration instanziiert. Dies wird von der Klasse `Initializer` realisiert, die in einem Klassendiagramm in Abbildung 29 dargestellt ist. Pro Konfiguration wird eine Instanz dieser Klasse erzeugt, die das fertig instanziierte Wurzelobjekt zurückliefert und alle konfigurierten Objekte verwaltet.

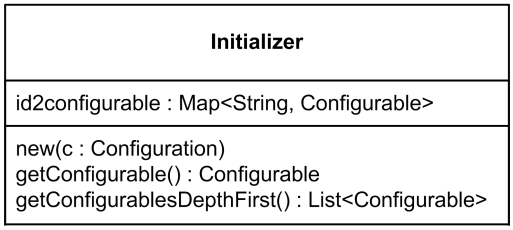


Abbildung 29: Klassendiagramm: `Initializer`

Der grundlegende Ablauf dieses Instanziierungsprozesses wird durch folgenden Codeausschnitt verdeutlicht:

```
public Initializer(ConfigurationModel model) {
  id2configurable = new HashMap<String, Configurable>();
  for (ModelElement me : model.depthFirstPreEnumeration()) {
    if (me instanceof ConfigurableClass) {
      ConfigurableClass cc = (ConfigurableClass) me;
      Class clazz = getClassLoader().loadClass(cc.className);
      Configurable conf = (Configurable) clazz.newInstance();
      id2configurable.put(cc.getId(), conf);
      for (PropertyElement pe :
          cc.getChildElements(PropertyElement.class)) {
        setValue(pe.getMethodEncoding(), clazz, conf,
          pe.value);
      }
    }
  }
  for (ModelElement me :
      model.depthFirstPreEnumeration()) {
    if (me instanceof AssociationElement) {
      AssociationElement ae = (AssociationElement) me;
      List<Object> refs = new ArrayList<Object>();
      for (ConfigurableClass cc :
          ae.getChildElements(ConfigurableClass.class)) {
        refs.add(id2configurable.get(cc.getId()));
      }
      Configurable conf = id2configurable.get(
        ae.getParent().getId());
      setValue(
        ae.getMethodEncoding(), conf.getClass(), conf, refs);
    }
  }
}
```

Der Ablauf ist in zwei Teilschritte untergliedert. Im ersten Schritt wird ein Objekt pro konfigurierbare Klasse erzeugt und die Eigenschaften werden gesetzt. Im zweiten Schritt werden die Beziehungen zwischen den Objekten hergestellt.

Die tatsächliche Realisierung der Instanziierung ist um die vorgestellten Funktionalitäten, die über das sehr einfache Grundmodell von Objekten und Beziehungen hinausgehen, erweitert.

4.6.1 Beobachtbare Eigenschaften

Eine häufig auftretende Anforderung an Informationssysteme besteht darin, den Zustand von Komponenten zur Laufzeit des Systems überwachen und anpassen zu können [FL02]. Dieses Szenario wird durch *beobachtbare Eigenschaften* unterstützt. Eine konfigurierbare Eigenschaft kann durch das Feld `isObservable` des Annotationstyps `Property` (siehe Tabelle 3) als beobachtbare Eigenschaft gekennzeichnet werden:

```
class Logger {
  @Property(isObservable=true)
  String logLevel;
  ...
```

Dies führt dazu, dass zur Instanziierung des Objektgraphen pro beobachtbare Eigenschaft ein Objekt vom Typ `ObservableProperty` erzeugt wird (siehe Abbildung 30). Über dieses Objekt lässt sich der aktuelle Wert der Eigenschaft zur Laufzeit des Systems abfragen und verändern. Eine Eigenschaft wird durch ihren Pfad im Konfigurationsbaum identifiziert.

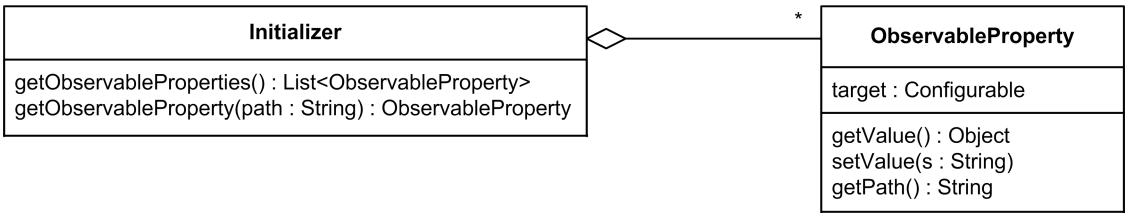


Abbildung 30: Beobachtbare Eigenschaften

4.7 Anatomie der DSL eines introspektiven Blackbox Frameworks

Bei domänenspezifischen Sprachen die durch Blackbox Introspektion realisiert werden, handelt es sich um formale Sprachen, die über die in Abschnitt 2.1 dargelegten Eigenschaften verfügen. In Tabelle 5 ist dieser Aspekt dargestellt.

Abstrakte Syntax	Die Konzepte der DSL eines introspektiven Blackbox Frameworks werden durch ein Metamodell, welches das in Abschnitt 4.2 beschriebene Metametamodell instanziiert, beschrieben. Zur Definition werden die Erweiterungsmöglichkeiten des Frameworks direkt annotiert.
Konkrete Syntax	Die Modelle lassen sich in Form einer domänenspezifischen Sprache im Konfigurator bearbeiten.
Serialisierungssyntax	Zum Bearbeitungszeitpunkt werden die Modelle als Instanzen des Metamodells vom Konfigurator verwaltet. Zur Speicherung werden sie in Form eines XML-Dialekts serialisiert, welche alle Informationen über die repräsentierten Erweiterungsmöglichkeiten enthält.
Semantik	Die Modelle führen zu Instanzen des Frameworks, wie in Abschnitt 4.6 dargestellt.

Tabelle 5: Anatomie der DSL eines introspektiven Blackbox Frameworks

4.8 Abgrenzung zu bestehenden Ansätzen

In diesem Abschnitt werden existierende Ansätze zur Konfiguration von Blackbox Frameworks vorgestellt. Diese Ansätze ermöglichen keine Integration der domänenspezifischen Sprache mit dem darunter liegenden Framework-Kern.

4.8.1 Konfiguration durch maßgeschneiderte textuelle Syntax

Ein weit verbreiteter Ansatz zur Konfiguration eines Blackbox Frameworks besteht darin, eine maßgeschneiderte textuelle domänenspezifische Sprache mit dazugehöriger Interpreter-Komponente zu erstellen. Eine solche Lösung besteht aus den folgenden Bausteinen:

- Sprachspezifikation mit Editoren,
- Interpreter-Komponente und
- Blackbox Framework

Die am häufigsten genutzte Technologie zur Erstellung der ersten beiden Bausteine ist XML. Die Sprachspezifikation kann entweder durch eine DTD oder durch ein XML-Schema erfolgen. In beiden Fällen existiert umfangreiche Werkzeugunterstützung. Ebenfalls existiert Unterstützung beim Erstellen der Interpreter-Komponente durch XML-Parser.

Eine Alternative zu XML ist die Verwendung von Parser-Generatoren, wie beispielsweise ANTLR [PQ95]. Hierbei wird die Grammatik der Sprache durch eine formale Spezifikation (z.B.: Backus-Naur-Form [Ba60]) vorgegeben, und hieraus wird eine Komponente erzeugt, die Programme der Sprache analysieren kann.

Der Kritikpunkt an diesem Ansatz besteht darin, dass die Sprachspezifikation sich nicht direkt aus dem Framework-Kern ergibt. Stattdessen wird die Sprache „neben“ dem Framework entworfen. Die domänenspezifische Sprache ist somit nicht mit dem Framework-Kern integriert. Das Erstellen einer DSL ist mit zusätzlichem Aufwand verbunden und die Konsistenz zwischen Sprache und Framework-Kern ist nicht automatisch überprüfbar. Wird der Framework-Kern weiterentwickelt, muss die Sprache stets manuell gepflegt werden.

Ein weiterer Schwachpunkt dieses Ansatzes ist die maßgeschneiderte Interpreter-Komponente. Zusätzlich zur Entwicklung der Sprachspezifikation muss auch diese manuell entworfen und weiterentwickelt werden.

4.8.2 Dependency Injection

Ein weiterer aktueller Ansatz zur Konfiguration von Blackbox Frameworks wird „Dependency Injection“ genannt [Fo04a]. Das ursprüngliche Ziel dieses Ansatzes ist die Verringerung von Abhängigkeiten zwischen Komponenten. Dies wird erreicht, indem Beziehungen zwischen Komponenten durch eine generische Initialisierungskomponente erstellt werden. Die beteiligten elementaren Komponenten sind während des Initialisierungsprozesses passiv und bekommen alle benötigten Abhängigkeiten „injiziert“. Diese Idee ist auch für elementare Eigenschaften anwendbar, so dass sich eine komplette Anwendung durch Dependency Injection initialisieren lässt. Ein populäres Framework zur Dependency-Injection ist das Spring Framework [In07]. Die Instanziierung des auf Seite 55 dargestellten Beispiels wird durch folgende Konfiguration erreicht:

```
<beans>
  <bean id="broker" class="de.infoasset.server.Broker">
    <property name="host" value="wwwmatthes.in.tum.de"/>
    <property name="webServers">
      <list>
        <ref bean="webServer"/>
      </list>
    </property>
  </bean>
  <bean id="webServer" class="de.infoasset.server.WebServer">
    <property name="port" value="80"/>
  </bean>
</beans>
```

Der Vorteil dieser Konfigurationsform besteht darin, dass die Interpreter-Komponente generisch und somit vom Framework-Kern unabhängig ist. Problematisch ist jedoch, dass die zu injizierenden Elemente zu großen Teilen nicht explizit gekennzeichnet sind und somit die vom Framework-Kern definierte domänenspezifische Sprache ebenfalls nicht explizit vorliegt. Der in dieser Arbeit vorgeschlagene Ansatz der Blackbox Introspektion erweitert die Idee der Dependency Injection um diesen Aspekt und ermöglicht so die Extraktion der domänenspezifischen Sprache.

5 Introspektive Whitebox Frameworks

In diesem Kapitel wird das Konzept der introspektiven Whitebox Frameworks detailliert vorgestellt. Introspektive Whitebox Frameworks realisieren domänenspezifische Sprachen intern unter Benutzung von introspektiven Programmiermodellen. Dies wird in Abschnitt 5.1 eingeführt. Im Rahmen dieser Arbeit wurden konkrete introspektive Programmiermodelle identifiziert, die das in Abschnitt 5.2 vorgestellte Metametamodell der Whitebox Introspektion bilden. Unter Benutzung dieses Metametamodells lassen sich effizient introspektive Whitebox Frameworks erstellen. Die entwickelte Werkzeugunterstützung ermöglicht das Anpassen dieser Frameworks durch eine domänenspezifische Sprache.

In Abschnitt 5.3 wird ein schematischer Überblick über das vorgestellte Vorgehen gegeben. In Abschnitt 5.4 werden domänenspezifische Sprachen introspektiver Whitebox Frameworks gemäß der im Exkurs auf Seite 22 vorgestellten Betrachtungsweise diskutiert. Zum Schluss des Kapitels wird der verwandte Ansatz der generativen modellgetriebenen Softwareentwicklung vorgestellt und diskutiert.

5.1 Introspektive Programmiermodelle

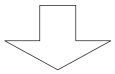
Introspektive Whitebox Frameworks realisieren domänenspezifische Modelle durch eine interne Repräsentation als Konstrukte der Basisprogrammiersprache, die explizit einer introspektiven Analyse zugänglich ist. Die Anpassungskonstrukte gehorchen bestimmten introspektiven Programmiermodellen. Zur Verdeutlichung dieses Konzepts wird im folgenden Abschnitt zunächst ein Beispielframework eingeführt, welches im gesamten Kapitel zur Demonstration verwendet wird.

5.1.1 Beispiel: Textdatei-Leser

Die im Folgenden eingeführte Problemstellung basiert auf einem Artikel von Martin Fowler [Fo05a]. Es soll der Inhalt einer Textdatei Zeile für Zeile ausgewertet werden. Mit „#“ beginnende Zeilen enthalten deskriptive Kommentare und werden nicht ausgewertet. In Abhängigkeit vom Zeileninhalt sollen verschiedene Aktionen ausgeführt werden. Die auszuführende Aktion wird von den ersten vier Zeichen einer jeden Textzeile bestimmt. Im Beispiel von Abbildung 31 wird durch den Zeilenanfang `SVCL` eine Aktion mit Namen `ServiceCall` ausgelöst. Die Parameter zur näheren Spezifikation eines `ServiceCall` befinden sich in bestimmten Bereichen der Textzeile. So beginnt der Parameter `customerName` in der Konfiguration von Abbildung 31 beispielsweise an Position 4 und endet an Position 10. Der Parameter `customerId` startet an Position 11 und endet an Position 15. Aktionen werden also durch die ersten vier Zeichen der Zeile identifiziert und durch Daten in bestimmten Abschnitten der Textzeile parametrisiert. Das Auslösen von konkreten Aktionen geschieht durch Funktionsauf-

rufe auf dem in Abbildung 32 dargestellten Interface. Dies soll stellvertretend für Programmlogik stehen, die einen beliebigen imperativen Programmstil wünschenswert macht.

```
#12345678901234567890123456789012345678901234567890123456789012345678901234567890
SVCLFOWLER 10101MS0120050313.......................
USGE10301TWO x50214..7050329.......................
```



```
ServiceCall(customerName=„Fowler“,
customerID=„10101“, callTypeCode=„MS01“,
dateOfCall=2005/03/13)

Usage(customerName= “TWO          x“,
customerID “10301“, cycle=7, readDate 05/03/29)
```

Abbildung 31: Beispieldaten

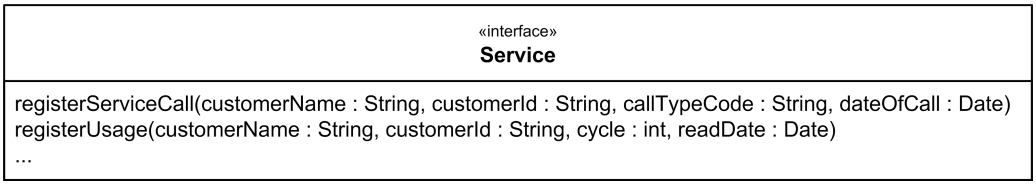


Abbildung 32: Service-Interface zum Auslösen von Aktionen

Um flexibel auf ein geändertes Format reagieren zu können, soll die Aufgabenstellung modellgetrieben umgesetzt werden. Die Parametrisierung einer konkreten Konfiguration sollen also auf einem hohen Abstraktionsniveau eingesehen und bearbeitet werden können. Die abstrakte Syntax einer domänenspezifischen Sprache, die Konfigurationen des Problems beschreibt, ist in Abbildung 33 dargestellt.

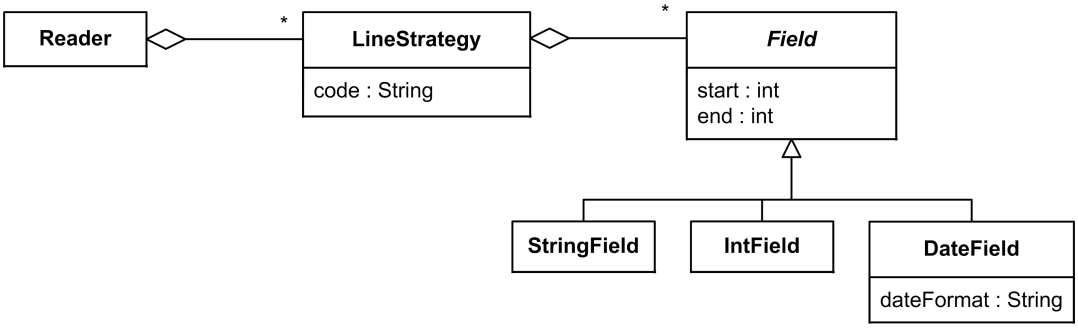


Abbildung 33: Abstrakte Syntax einer DSL zum Lesen der Textdatei

Wie bereits erwähnt, besteht ein weiterer Teil des Szenarios darin, dass Aktionen durch imperative Funktionsaufrufe registriert werden. Eine spezielle Herausforderung an eine modellgetriebene Umsetzung besteht also in der Integration zwischen den deklarativen Modellierungsar-

tefakten zur Parametrisierung der Aktionen und den imperativen Codefragmenten zum Registrieren der Aktionen.

5.1.2 Schablonen- und Einschubmethoden

Für das Verständnis von Whitebox Frameworks sind zwei Arten von Methoden wichtig [Pr97a]:

- Schablonenmethoden (template methods) und
- Einschubmethoden (hook methods).

Schablonenmethoden spezifizieren generisches Verhalten, welches allen vom Framework abgeleiteten Anwendungen gemeinsam ist. Dementsprechend werden Schablonenmethoden in den Klassen des Framework-Kerns implementiert und nicht durch Anpassung verändert. Im Gegensatz dazu fungieren Einschubmethoden als Platzhalter, deren Implementierung von Subklassen angepasst werden kann. Einschubmethoden werden in abstrakten Frameworkklassen als Platzhalter definiert und in konkreten Subklassen entsprechend überschrieben. Zusammenfassend spezifizieren Schablonenmethoden das von allen Anwendungen wieder verwendete Verhalten und rufen an bestimmten Stellen Einschubmethoden auf, die angepasstes Verhalten spezifizieren. Die Erweiterungsmöglichkeiten eines Whitebox Frameworks werden somit durch die vorhandenen Einschubmethoden definiert.

Im vorgestellten Framework zur Interpretation von Textdateien wird der komplette Vorgang des Einlesens von einer Schablonenmethode der Klasse `Reader` realisiert:

```
public final void readFile(String filename) {
  String all = Util.getFileAsString(filename);
  StringTokenizer tok = new StringTokenizer(all, "\r\n");
  List<LineStrategy> strategies = getStrategies();
  while (tok.hasMoreTokens()) {
    String line = tok.nextToken();
    if(!line.startsWith("#")) {
      String code = line.substring(0, 4);
      findStrategy: {
        for (LineStrategy strategy : strategies) {
          if (strategy.getCode().equals(code)) {
            strategy.read(line);
            break findStrategy; }
        }
        throw new IllegalStateException("line: " + line);
      }
    }
  }
}
```

Der in dieser Methode spezifizierte generelle Ablauf ist für alle Konfigurationen des Frameworks gleich und kann von Subklassen nicht verändert werden. Die Stellen, an denen Variabilität eingebaut ist, sind rot gekennzeichnet. Dabei handelt es sich um Aufrufe von Einschubmethoden, deren Verhalten nicht im Framework-Kern festgelegt wurde und somit von Subklassen angepasst werden kann.

In Abbildung 34 sind die Klassen des Framework-Kerns in einem UML-Klassendiagramm dargestellt. Die Erweiterungsmöglichkeiten, die durch Einschubmethoden realisiert sind, sind rot gekennzeichnet. Um das Framework auf eine konkrete Problemstellung anzupassen, werden vom Anwendungsentwickler konkrete Subklassen mit entsprechenden Einschubmethoden implementiert.

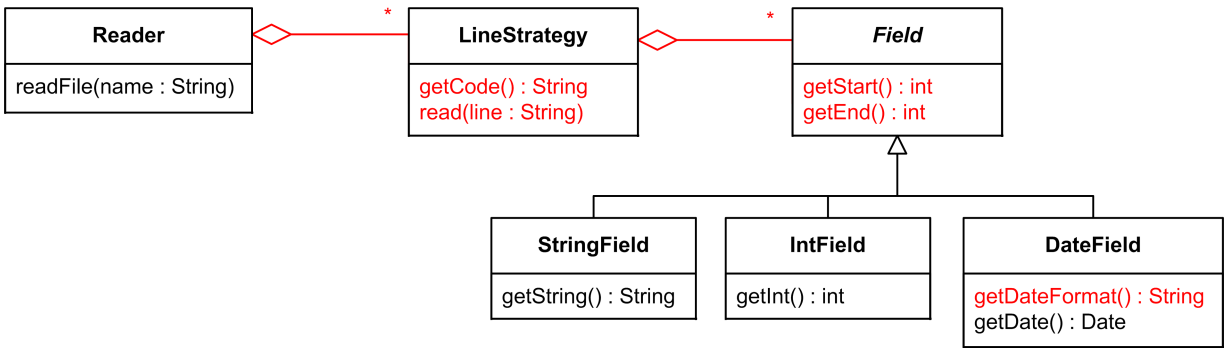


Abbildung 34: Konzeptuelles Klassendiagramm mit Erweiterungsmöglichkeiten

5.1.3 Introspektive Einschubmethoden

Die zur Anpassung zu implementierenden Einschubmethoden lassen sich wiederum in zwei Kategorien unterteilen. Wie bereits erwähnt, lassen sich bestimmte Aspekte des Frameworks deklarativ anpassen, für andere ist ein imperativer Programmierstil wünschenswert. Dementsprechend unterscheiden sich die Einschubmethoden bezüglich des zur Anpassung gewählten Programmierstils. Der in dieser Arbeit vorgestellte Ansatz der introspektiven modellgetriebenen Softwareentwicklung besteht darin, deklarativ implementierte Anpassungen introspektiv zu analysieren und durch ein Modell zu repräsentieren (siehe Seite 46). Dementsprechend wird im Folgenden zwischen introspektiven und nicht-introspektiven Einschubmethoden unterschieden.

Das Implementieren von introspektiven Einschubmethoden erfolgt in einem deklarativen Programmierstil. Aufgrund des einfachen, gut verstandenen Programmiermodells sind einerseits eine Analyse und andererseits eine Synthese von Anpassungskonstrukten möglich. Eine Anforderung an einen introspektiven Programmierstil besteht also in der Einfachheit der Analyse. Dies bezieht sich auf syntaktische Eigenschaften der zu analysierenden Codefragmente.

Eine zweite Forderung bezieht sich auf die Semantik des gewonnenen Modells, welches durch statische Analyse introspektiver Einschubmethoden entsteht. Es handelt sich hierbei um eine so genannte Übersetzersemantik. Die Semantik der introspektiv gewonnenen Modelle ergibt sich aus der Semantik der abgebildeten introspektiven Konstrukte der Basisprogrammiersprache –

im konkreten Fall Java. Die Modelle werden durch statische Analyse gewonnen. Wünschenswert ist eine definitive Aussage über das Verhalten zur Laufzeit eines Systems. Daraus ergibt sich die Forderung an ein introspektives Programmiermodell, dass die Semantik der Konstrukte, aus denen das Modell gewonnen wird, unabhängig von Laufzeitbedingungen ist. Diese Forderung wird im Folgenden *Reproduzierbarkeit* der Modelleigenschaften genannt.

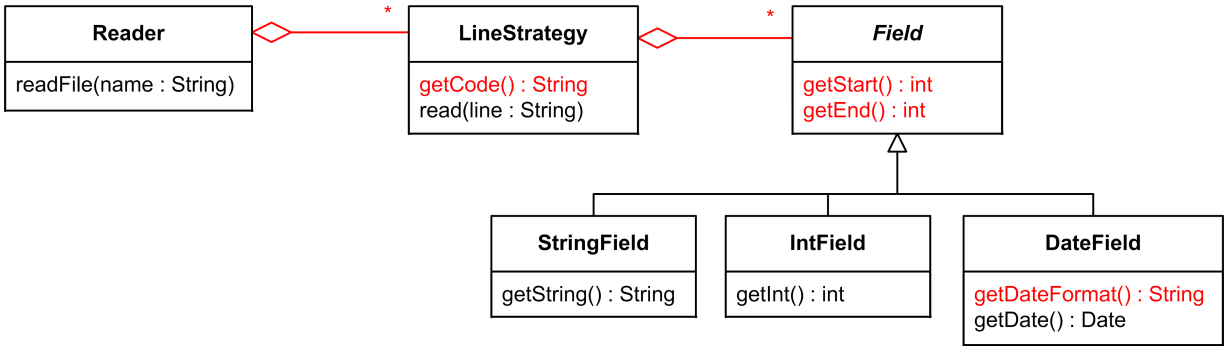


Abbildung 35: Introspektive Erweiterungsmöglichkeiten

Im Klassendiagramm in Abbildung 35 sind alle introspektiven Erweiterungsmöglichkeiten des Beispielframeworks, die durch introspektive Einschubmethoden realisiert werden, farbig gekennzeichnet. Diese Erweiterungsmöglichkeiten beschreiben eine domänenspezifische Sprache und sind strukturgleich mit der in Abbildung 33 dargestellten abstrakten Syntax. Beispielsweise wird der Erweiterungspunk `code` der Klasse `LineStrategy` im Framework-Kern durch die abstrakte Methode `getCode` spezifiziert:

```
@Introspective
public abstract String getCode();
```

Introspektive Einschubmethoden werden durch den Annotationstyp `Introspective` explizit gekennzeichnet. Das introspektive Programmiermodell besteht in diesem konkreten Fall darin, dass der implementierenden Methodenrumpf aus genau einem Return-Statement besteht, welches ein String-Literal zurückgibt:

```
public String getCode() {
  return "SVCL";
}
```

Dies ermöglicht eine Analyse und eine anschließende Modellbildung. In Abbildung 36 sind sowohl Quelltext als auch introspektive Modellansicht des vorgestellten Quelltextes im Whitebox Modeler in einem Screenshot dargestellt.

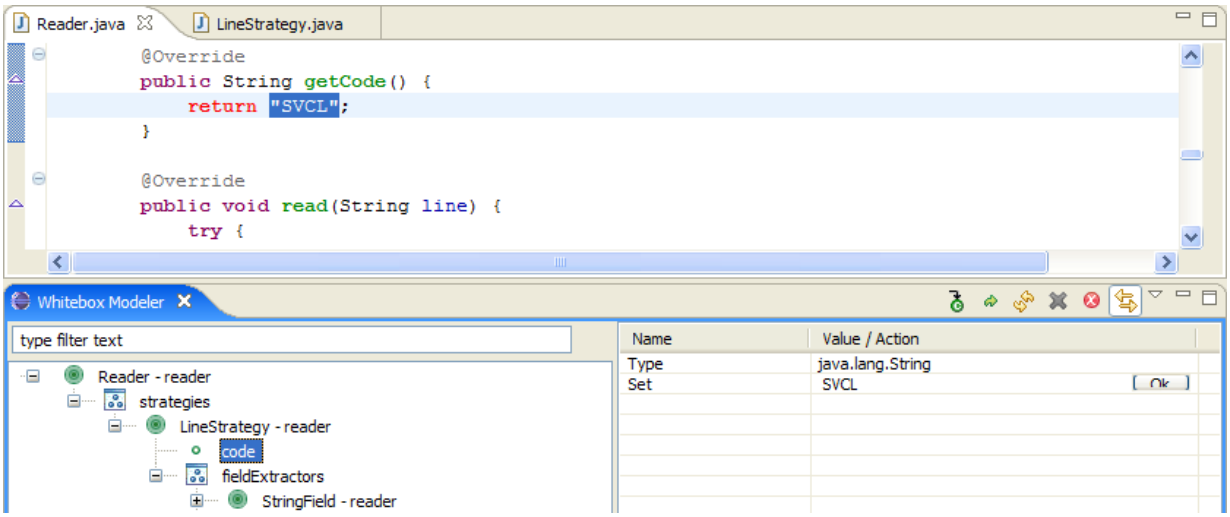


Abbildung 36: Beispiel einer introspektiven Methode im Whitebox Modeler

Die Methode `read` der Klasse `LineStrategy` ist nicht-introspektiv. In dieser Methode wird entsprechend den in der entsprechenden Zeile enthaltenen Daten eine Aktion erzeugt. Dies geschieht durch einen Aufruf auf der in Abbildung 32 dargestellten Schnittstelle:

```
public void read(String line) {
  Service.INSTANCE().registerServiceCall(
    this.customerName.getString(line),
    this.customerId.getString(line),
    this.callTypeCode.getString(line),
    this.dateOfCall.getDate(line));
}
```

Es existieren verschiedene Arten von introspektiven Methoden, die jeweils mit einem bestimmten deklarativen Programmiermodell verbunden sind. Beim Entwurf eines introspektiven Whitebox Frameworks werden die deklarativ zu modellierenden Aspekte durch introspektive Methoden anpassbar und somit introspektiv modellierbar gestaltet. Dabei wird das Programmiermodell festgelegt. Die introspektiven Methoden werden in den abstrakten Frameworkklassen durch Annotationen gekennzeichnet.

5.2 Metametamodell der Whitebox Introspektion

In diesem Abschnitt werden die von dieser Arbeit unterstützten Arten von introspektiven Methoden vorgestellt. Diese definieren ein Metametamodell, welches in einem Metamodellierungsschritt konzeptuell vom Frameworkentwickler zur Entwicklung introspektiver Frameworks genutzt werden kann und wofür Werkzeugunterstützung zur Modellierung existiert. Ebenso beinhaltet jede introspektive Methodenart ein bestimmtes Programmiermodell, an welches sich der Anwendungsentwickler halten muss, um eine introspektive Analyse zu ermöglichen. Das vorgestellte Werkzeug erstellt ein Modell und überprüft hierbei die Einhaltung des Programmiermodells.

In [BM06] wurde eine vereinfachte Version des Metametamodells vorgestellt.

5.2.1 Introspektive Klasse

Ein Kernkonzept des hier vorgestellten Metametamodells der Whitebox Introspektion ist das der *introspektiven Klasse*. Eine introspektive Klasse ist eine Klasse mit mindestens einer introspektiven Methode. Klassen werden nicht explizit als introspektiv deklariert, sondern vielmehr implizit durch das Vorhandensein introspektiver Methoden. Wie in Abbildung 37 dargestellt, besteht ein durch Introspektion gewonnenes Modell aus introspektiven Klassen. Wie bereits beim Metametamodell der Blackbox Introspektion, werden auch im Folgenden Icons zur visuellen Markierung der Metametamodellelemente benutzt.

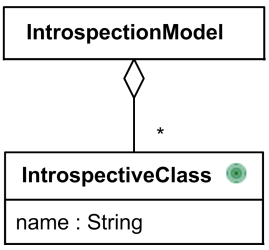


Abbildung 37: Metametamodell: Introspektive Klasse

Aufgrund der Eigenschaft der geforderten Reproduzierbarkeit des Ergebnisses introspektiver Methodenaufrufe sind die im Modell repräsentierten Eigenschaften tatsächlich Eigenschaften der jeweiligen analysierten Klasse und somit von der tatsächlich vorliegenden Instanz unabhängig.

5.2.2 Value-Methode

Die einfachste Art introspektiver Methoden ist die so genannte *Value-Methode*. Dabei handelt es sich um Methoden, deren Signatur keine Parameter enthält und deren Rückgabewert von elementarem Typ ist. Elementare Typen sind in diesem Zusammenhang alle primitiven Typen sowie der Typ `java.lang.String`. Das introspektive Programmiermodell, das von konkreten Anpassungen eingehalten werden muss, beinhaltet die direkte Rückgabe eines Literals vom entsprechenden Typ.

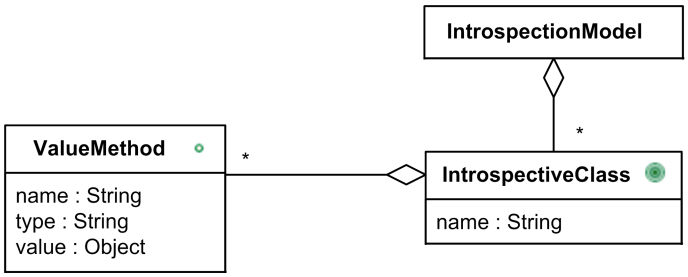


Abbildung 38: Metametamodell mit Value-Methode

In Abbildung 38 ist das um das Element „Value-Methode“ erweiterte Metametamodell der Whitebox Introspektion dargestellt. Name und Typ der Value-Methode ergeben sich aus der Signatur der Methode. Der Wert ergibt sich als Ergebnis einer konkreten Analyse.

Spezifikation by-value vs. by-constant

Wie bereits beispielhaft auf Seite 81 dargestellt, lässt sich das Feld `code` der Klasse `LineStrategy` mit einer Value-Methode realisieren. Dies trifft auch auf die Eigenschaften `start` und `end` der Klasse `Field`, sowie `dateFormat` der Klasse `DateField` zu. Das bereits im Beispiel zum Anpassen verwendete Programmiermodell liefert das Ergebnis direkt als String-Literal zurück. In manchen Fällen ist es wünschenswert, den Rückgabewert zusätzlich an anderen Stellen explizit referenzieren zu können. Für diesen Fall bietet sich die Deklaration einer Variablen an, die dann von der Value-Methode zurückgeliefert wird:

```
String SERVICE_CALL_CODE = "SVCL";

public String getCode() {
  return SERVICE_CALL_CODE;
}
```

In diesem Fall ist jedoch die Reproduzierbarkeit des Ergebnisses nicht gegeben. Eine Veränderung des Wertes der Variable `SERVICE_CALL_CODE` zur Laufzeit kann zu einem vom Modell abweichenden Ergebnis führen. Dementsprechend führt eine introspektive Analyse zu einer Fehlermeldung, wie in Abbildung 39 dargestellt.

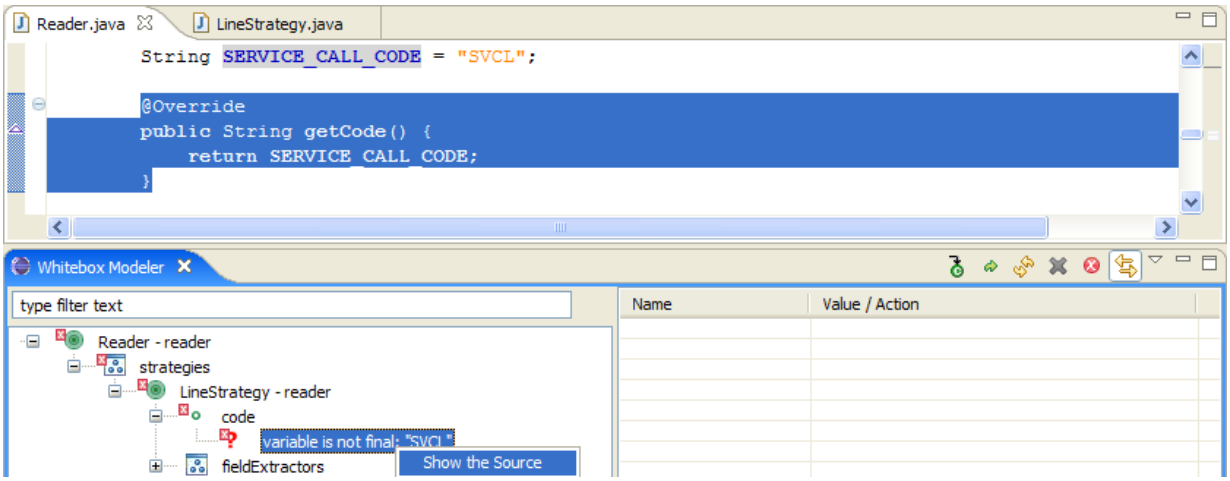


Abbildung 39: Modellsicht einer Value-Methode mit nicht-finaler Variablenreferenz

Die Lösung dieses Problems besteht darin, die Variable als `final` zu deklarieren, was eine Veränderung nach der initialen Zuweisung verhindert:

```
final String SERVICE_CALL_CODE = "SVCL";

public String getCode() {
  return SERVICE_CALL_CODE;
}
```

In vielen Fällen ist die direkte Spezifikation des Wertes wünschenswert. Diese Art der Spezifikation wird im Folgenden „*by-value*“ genannt. Ein großer Vorteil der Modellierung durch Whitebox Introspektion besteht jedoch darin, dass symbolische Integration von deklarativen und imperativen Anpassungskonstrukten möglich ist. Dies lässt sich dadurch erreichen, dass introspektive Methoden Referenzen auf finale und somit konstante Variablen zurückliefern. Diese konstanten Variablen können ebenfalls in nicht-introspektiven Methoden zugegriffen werden. Dies wird im Folgenden „*by-constant*“ genannt.

In der Modelldarstellung sind beide Arten der Spezifikation nicht unterscheidbar. Dies ist insofern problematisch, da sich die Semantik von *by-value* und *by-constant* möglicherweise leicht unterscheidet. Dieser Unterschied entsteht, wenn durch introspektive Methoden Objekte erzeugt werden. Im Fall von *by-value* werden bei jedem Aufruf neue Objekte erzeugt. Ein durch *by-constant* zurückgeliefertes Objekt wird im Unterschied dazu nur bei der Deklaration der Variable erzeugt und bei anschließenden Aufrufen wird die immergleiche Objektinstanz zurückgeliefert.

Dieser subtile Unterschied muss bei der Konstruktion introspektiver Whitebox Frameworks bedacht werden. Die durch das introspektive Modell getroffenen Aussagen beziehen sich auf Gleichheit der Werte. Eventuelle Gleichheit der tatsächlichen Referenzen lässt sich aus dem Modell direkt nicht ablesen.

Kontextfreie Grammatiken zur Beschreibung des Programmiermodells

Eine introspektive Modellbildung ist nur möglich, wenn beim Implementieren introspektiver Einschubmethoden das gewählte Programmiermodell eingehalten wurde. Für jede Art introspektiver Methoden werden bestimmte Programmiermodelle unterstützt. Im Fall der Value-Methode handelt es sich dabei um die Rückgabe eines statisch im Quelltext definierten Wertes – sowohl direkt *by-value* als auch *by-constant* über eine finale Variable. Die vollständige Ausdrucksmächtigkeit der Programmiersprache Java wird hierbei zugunsten der Möglichkeit der Introspektion eingeschränkt. Zum Anpassen eines introspektiven Whitebox Frameworks ist ein detailliertes Verständnis der unterstützten Programmiermodelle von großer Bedeutung. Zu diesem Zweck ist eine formalere Beschreibung wünschenswert.

Die Syntax und Semantik der Programmiersprache Java wird in [Go05] beschrieben. Bei der Syntax von Java handelt es sich um eine kontextfreie Grammatik. Eine kontextfreie Grammatik wird durch die folgenden vier Komponenten beschrieben:

1. Eine Menge von *Terminalen*. Terminale lassen sich nicht weiter auflösen und sind somit die elementaren Symbole der Sprache.
2. Eine Menge von *Nichtterminalen*. Nichtterminale sind die noch nicht aufgelösten Zwischensymbole der Sprache.
3. Eine Menge von *Produktionen*. Durch Produktionen werden gültige Sätze der Sprache beschrieben. Auf der linken Seite einer Produktion steht ein Nichtterminal, welches durch auf der rechten Seite befindliche Terminale und Nichtterminale ersetzt werden kann.
4. Ein *Startsymbol*. Das Startsymbol ist ein Nichtterminal, welches den Startpunkt aller gültigen Sätze der Sprache darstellt.

Für eine detailliertere Einführung in kontextfreie Grammatiken sei auf [ASU86] verwiesen. Zur Beschreibung der introspektiven Programmiermodelle sollen im Folgenden ebenfalls kontextfreie Grammatiken benutzt werden. Allerdings soll die Beschreibung nicht auf der Sprachbeschreibung in [Go05] basieren, sondern auf der vom Eclipse-Projekt Java Development Tools (JDT) definierten. Wie bereits im Post-IntelliJ-IDEs auf Seite 24 erwähnt, ist Eclipse eine so genannte Post-IntelliJ-IDE. Demzufolge verfügt Eclipse über einen inkrementellen Compiler, der stets den Quelltext der bearbeiteten Projekte kompiliert und eine Repräsentation als abstrakten Syntaxbaum vorhält. Diese Teilaufgabe innerhalb der Entwicklungsumgebung wird vom Eclipse-JDT wahrgenommen. Zu diesem Zweck existiert eine Beschreibung von Java als kontextfreie Grammatik und ebenfalls eine objektorientierte Implementierung dieser Grammatik [Ec05]. In dieser Implementierung werden Nichtterminale durch Klassen modelliert. Beispielsweise wird die Deklaration einer Methode durch die folgende Produktion beschrieben:

MethodDeclaration:
{ *IExtendedModifier* }
[**<** *Type* { **,** *Type* } **>**] (*Type* | **void**) *SimpleName* **(**
[*SingleVariableDeclaration* { **,** *SingleVariableDeclaration* }] **)**
{ **[ ]** } [**throws** *Name* { **,** *Name* }] (*Block* | **;**)

Dabei sind Nichtterminale *kursiv* dargestellt. Terminale sind nicht-proportional in **`fett`** gedruckt. In diesem Beispiel besteht eine *MethodDeclaration* aus mehreren *IExtendedModifier*-Elementen, optionalen *Types* als generische Typparameter, einem *Type* oder **`void`** als Rückgabetyp, einem *SimpleName* als Methodenname, mehreren durch Komma getrennten *SingleVariableDeclarations* als Parameter, optionalen jeweils durch einen *Name* spezifizierten Exceptions, sowie einem optionalen *Block*, der den Methodenrumpf repräsentiert.

Um eine formale Spezifikation der unterstützten introspektiven Programmiermodelle zu erlangen, werden im Folgenden jeweils Produktionen für die Programmiermodelle angegeben. Da es sich um introspektive Methoden handelt, basieren diese auf der im Beispiel vorgestellten *MethodDeclaration* und reglementieren diese auf das jeweils erlaubte Programmiermodell. Dabei werden neue Nichtterminale eingeführt, die typographisch ***fett und kursiv*** dargestellt werden.

Die Spezifikation einer introspektiven Value-Methode wird durch das Nichtterminal ***ValueMethode_M2*** beschrieben:

ValueMethode_M2:
`@Introspective {` *Modifier* **`}`** ***ValueMethodType*** *SimpleName* **`( ) ;`**

Durch den Nachsatz ***_M2*** wird ausgedrückt, dass sich das Konstrukt auf der Metaebene M2 auswirkt und somit ein Element des Metamodells beschreibt. Es wird eine Methoden-Deklaration beschrieben, die mit der Annotation `Introspective` gekennzeichnet ist, mehrere *Modifier* enthalten kann, den Rückgabetyp ***ValueMethodType*** hat und keine Parameter spezifiziert. Das Nichtterminal ***ValueMethodType*** erlaubt die folgenden Typen:

ValueMethodType:
`byte`
`short`
`char`
`int`
`long`
`float`
`double`
`boolean`
StringType

Das Nichtterminal ***StringType*** repräsentiert einen *Type*, der an die Klasse `java.lang.String` gebunden ist. Eine Value-Methode hat also als Rückgabewert entweder einen primitiven Typ oder einen String.

Das im Kontext des Anpassens einer Value-Methode erlaubte Programmiermodell wird durch das Nichtterminal ***ValueMethod_M1*** repräsentiert:

ValueMethod_M1:
`{` *Modifier* **`}`** ***ValueMethodType*** *SimpleName* **`( ) {`**
`return` ***AbstractValue*** **`; }`**

Der Rumpf einer Value-Methode besteht also stets aus genau einem Return-Statement. Der erlaubte Inhalt des Return-Statements wird durch das Nichtterminal ***AbstractValue*** spezifiziert:

AbstractValue:
    Value
    SimpleName$_{ValueVariableName}$

Hierbei kann es sich also um das Nichtterminal ***Value*** oder einen Identifier, der durch das Nichtterminal *SimpleName* spezifiziert wird, handeln. Das Nichtterminal ***Value*** umfasst alle Möglichkeiten der Spezifikation eines Wertes und ist folgendermaßen definiert:

Value:
    BooleanLiteral
    CharacterLiteral
    NumberLiteral
    StringLiteral
    NullLiteral

Interessant ist, dass das Nichtterminal *SimpleName* in der Definition von ***AbstractValue*** durch den tiefergesetzten Nachsatz $_{ValueVariableName}$ gekennzeichnet ist. Rein syntaktisch bezeichnet *SimpleName* einen beliebigen Identifier. In diesem Zusammenhang muss jedoch eine Bindung zwischen dem Identifier und einer Variablen-Deklaration bestehen:

FieldDeclaration:
    [*Javadoc*] ***FinalModifiers***
        ValueMethodType *SimpleName*$_{ValueVariableName}$ **=** ***Value*** **;**

Das Nichtterminal ***FinalModifiers*** spezifiziert hierbei mehrere *Modifier*, die mindestens **`final`** enthalten.

Vier Metaebenen der Whitebox Introspektion

In Abbildung 40 sind die vier von der OMG definierten Metaebenen (siehe Seite 35) mit ihren jeweiligen Ausprägungen im Falle der Whitebox Introspektion abgebildet. Auf Ebene M3 befindet sich das Metametamodell. Ein Whitebox Framework instanziiert Elemente des Metametamodells, indem es abstrakte introspektive Klassen mit introspektiven Einschubmethoden definiert. Durch Introspektion lassen sich diese analysieren und somit das Metamodell gewinnen. Metamodellierung findet also implizit durch den Entwurf des Frameworks statt.

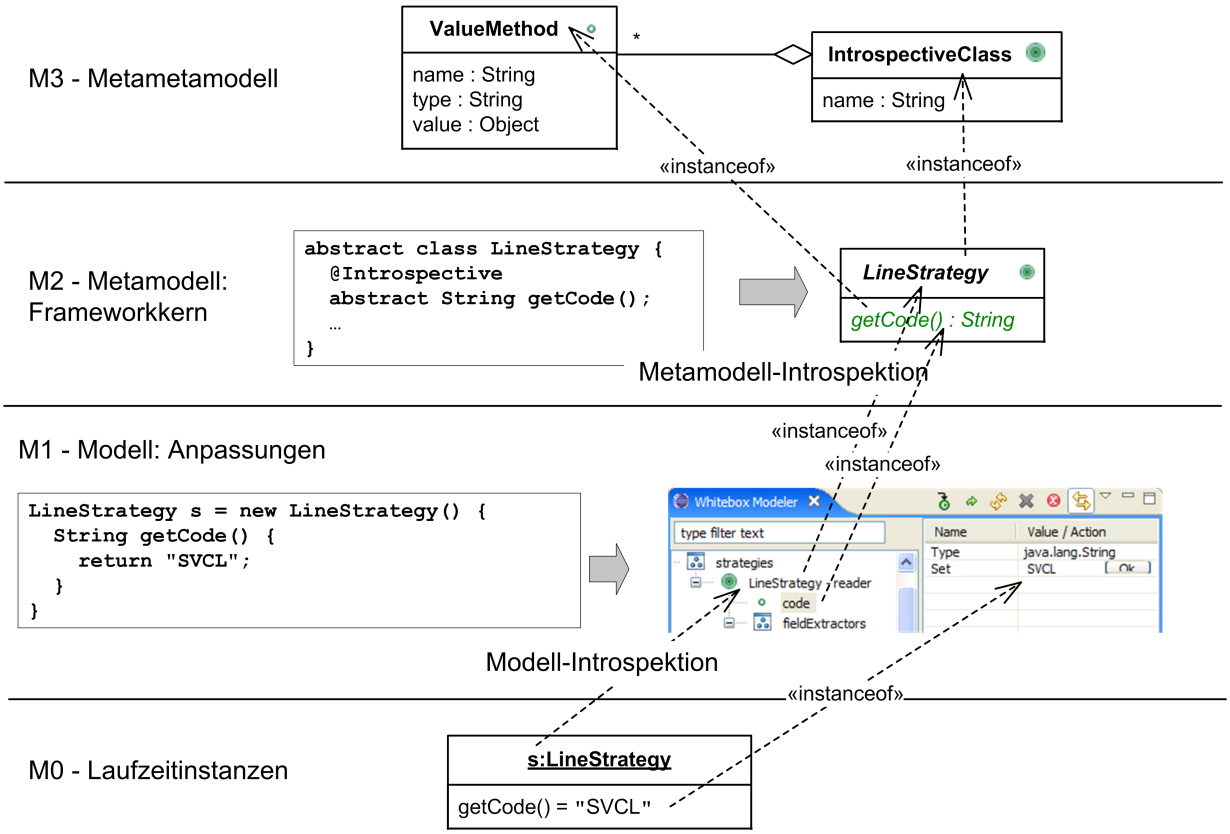


Abbildung 40: Vier Metaebenen der Whitebox Introspektion

Im nächsten Schritt wird das Framework zur Lösung eines Problems angepasst. Dies manifestiert sich in konkreten Klassen, die die abstrakten Frameworkklassen – und somit deren Methoden – implementieren. Durch Introspektion lassen sich diese Klassen analysieren und als Modelle darstellen. Modell und Anpassung sind äquivalent. Anpassung kann entweder direkt im Quelltext oder in der modellhaften Sicht erfolgen. Anpassungen instanziieren das vom Frameworks vorgegebene Metamodell..

Wird das System zur Laufzeit instanziiert, werden Objekte von den modellierten Klassen erzeugt. Diese Objekte weisen aufgrund der Eigenschaft der Reproduzierbarkeit der introspektiven Programmiermodelle die modellierten Eigenschaften auf.

Jedes Element des Metametamodells wirkt sich auf verschiedenen Metaebenen aus. So wird das Element „Value-Methode“ auf der Ebene M2 durch die Deklaration einer Methode mit entsprechender Signatur und Kennzeichnung durch den Annotationstyp `Introspective` definiert. Rein konzeptuell lassen sich durch ein Metamodell-Element vom Typ „Value-Methode“ konfigurierbare elementare Eigenschaften spezifizieren.

Auf der Ebene M1 ergibt sich im Quelltext durch die Verwendung einer Value-Methode ein bestimmtes introspektives Programmiermodell, welches eingehalten werden muss. In der modellhaften Darstellung sind mit einem Element vom Typ Value-Methode bestimmte Aktionen verbunden.

Auf den Ebenen M1 und M2 besteht also jeweils eine Dualität zwischen Eigenschaften auf Seite des Quelltextes und Eigenschaften auf Seite der modellhaften Repräsentation.

5.2.3 Objects-Methoden

Neben elementaren Eigenschaften muss ein Metamodellierungsansatz das Erstellen von Beziehungen ermöglichen. Die einfachste Möglichkeit, dies in einem introspektiven Whitebox Framework zu tun, ist mit Hilfe von *Objects-Methoden*. Dabei handelt es sich um Methoden, deren Signatur keine Parameter enthält und die einzelne oder eine Menge von Objekten introspektiver Klassen zurückliefern. Das entsprechend erweiterte Metametamodell ist in Abbildung 41 dargestellt. Die angegebenen Eigenschaften wie Name, Typ und Kardinalität ergeben sich aus der Signatur der abstrakten Objects-Methode im Framework-Kern.

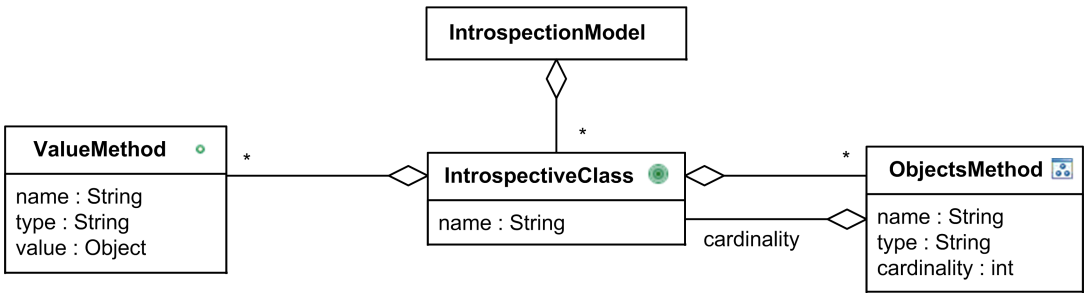


Abbildung 41: Metametamodell mit Objects-Methode

Im Unterschied zur Value-Methode gibt es verschiedene Programmiermodelle, die das Konzept der introspektiven Objects-Methode realisieren. Zunächst gibt es zwei Arten von Objects-Methoden, die sich in der Kardinalität der herzustellenden Beziehung unterscheiden:

1. Methoden, die genau ein Objekt zurückliefern und
2. Methoden, die mehrere Objekte zurückliefern können.

SingleObject-Methode

Die einfachste Art der Objects-Methode ist die sogenannte *SingleObject-Methode*. Als Beispiel soll hier die Kardinalität der Beziehung zwischen den Klassen `Reader` und `LineStrategy` verändert werden (siehe Abbildung 42). Ein `Reader` unterhält demnach eine Beziehung zu genau einem Objekt vom Typ `LineStrategy`.

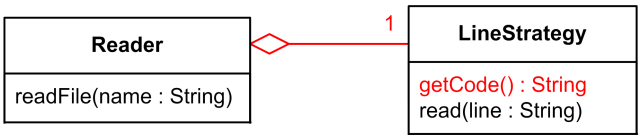


Abbildung 42: Beispiel: Kardinalität one zu Demonstrationszwecken

Auf der Metaebene M2 wird eine SingleObject-Methode durch eine abstrakte Methode, die keine Parameter erhält und ein Objekt vom Typ des Beziehungspartners zurückliefert, definiert. Die Beispielbeziehung wird folgendermaßen definiert:

```
@Introspective
public abstract LineStrategy getStrategy();
```

Die Spezifikation der Methode auf Metaebene M2 ist folgendermaßen mittels einer kontextfreien Grammatik definiert:

SingleObjectMethod_M2:
`@Introspective` { *Modifier* } ***IntrospectiveType*** *SimpleName* **`( ) ;`**

Das Nichtterminal ***IntrospectiveType*** steht dabei für einen *Type*, der an eine introspektive Klasse gebunden ist.

Das Programmiermodell auf der Metaebene M1 zum Anpassen des Frameworks erlaubt im Methodenrumpf genau ein Return-Statement, das eine Referenz auf ein erzeugtes Objekt zurückliefert:

```
public LineStrategy getStrategy() {
  return new LineStrategy() {
    public String getCode() {
      return "SVCL";
    };
  }
}
```

Eine Ansicht des introspektiven Modells im Whitebox Modeler ist in Abbildung 43 dargestellt.

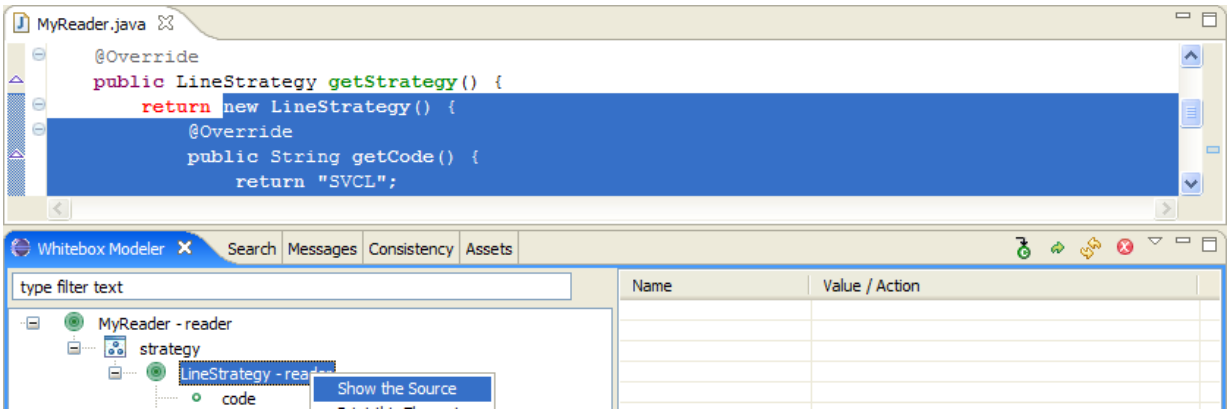


Abbildung 43: Beispiel: Introspektive Modellsicht einer Objects-Methode

Analog zur Argumentation bei Value-Methoden ist es in einigen Fällen erwünscht, die zurüc gelieferten Objekte unabhängig von der introspektiven Methode zu referenzieren. Dies wird durch die Verwendung von finalen konstanten Variablen unterstützt, was der bereits erwähnten Spezifikation *by-constant* entspricht. Die Spezifikation des Beipiels *by-constant* ergibt den folgenden Quelltext:

```
final LineStrategy STRATEGY = new LineStrategy() {
  public String getCode() {
    return "SVCL";
  };
}

public LineStrategy getStrategy() {
  return STRATEGY;
}
```

Im Unterschied zur Spezifikation *by-value* ist das Objekt auch außerhalb des Methodenaufrufs direkt zugreifbar. Es existieren Anwendungsfälle, in denen diese Art der symbolischen Integration von großer Bedeutung ist. Der direkte Zugriff auf die introspektiven Objekte ist in diesen Fällen essenziell, so dass stets die Spezifikation *by-constant* zur Anwendung kommt.

Die formale Spezifikation des zur Anpassung benutzten Programmiermodells auf Metaebene M1 ist folgendermaßen:

SingleObjectMethod_M1:
{ *Modifier* } ***IntrospectiveType*** *SimpleName* **( ) {**
return ***NewIntrospectiveObject*** **; }

NewIntrospectiveObject:
ClassInstanceCreation
$SimpleName_{IntrospectiveObjectVariableName}$

Beim Nichtterminal ***NewInstrospectiveObject*** handelt es sich also entweder um eine direkt erzeugte Instanz oder um einen finalen Identifier, der an eine Variable gebunden ist, die eine neue Instanz erzeugt:

IntrospectiveObjectDeclaration:
[*Javadoc*] ***FinalModifiers***
IntrospectiveType $SimpleName_{IntrospectiveObjectVariableName}$ **=**
ClassInstanceCreation **;**

Exkurs: Anonyme innere Klassen

Im auf Seite 92 eingeführten Nichtterminal ***NewIntrospectiveObject*** wird die Erzeugung eines Objekts vom Typ einer introspektiven Klasse durch ein Nichtterminal vom Typ *ClassInstanceCreation* beschrieben. Introspektive Klassen enthalten per Definition introspektive Methoden, für die konkrete Klassen Implementierungen bereitstellen müssen. Es ergeben sich zwei Möglichkeiten, an welcher Stelle diese Implementierungen vorgenommen werden. Einerseits wird eine eigenständige Klasse definiert, die die abstrakten introspektiven Methoden implementiert. Im Nichtterminal *ClassInstanceCreation* wird eine Instanz vom Typ dieser Klasse erzeugt. Beispielsweise ist die Klasse `TestStrategy` eine konkrete Klasse vom Typ `LineStrategy`, die alle introspektiven Methoden implementiert:

```
public class TestStrategy extends LineStrategy {
  public String getCode() {
    return "SVCL";
  }
}
```

Die Klasse kann nun instanziiert werden:

```
final LineStrategy STRATEGY = new TestStrategy();
```

Eine weitere Möglichkeit besteht darin, direkt „vor Ort“ Instanz vom Typ `LineStrategy` zu erzeugen:

```
LineStrategy STRATEGY = new LineStrategy() {
  public String getCode() {
    return "SVCL";
  }
};
```

In diesem Fall wird eine so genannte *anonyme innere Klasse* erzeugt, die die abstrakten Einschubmethoden implementiert [Go05]. Auf der Ebene des durch Introspektion erzeugten Modells sind beide Möglichkeiten gleichwertig.

Das vorgestellte Programmiermodell *by-constant* beinhaltet zu einem gewissen Grad Redundanz. Konzeptuell betrachtet soll durch eine finale Variable ein Objekt erzeugt werden, das ebenfalls von einer introspektiven Objects-Methode zurückliefert wird. Die Existenz der Varia-

ble ist zwingend, da das Objekt durch diese Variable zugegriffen werden kann. Die Implementierung der Methode jedoch ist stereotyp und beinhaltet Redundanz. Dies stellt einen Verstoß gegen das DRY-Prinzip (Dont Repeat Yourself) dar [TC02].

ObjectByName-Methode

Eine Alternative stellt hier die Verwendung von Programmierkonventionen und Reflektion (engl.: Reflection) dar. Die Verwendung von Konventionen hat in der letzten Zeit großen Anklang im Zusammenhang mit dem Webvisualisierungs-Framework „Ruby on Rails“ gefunden [TH05]. Ein Leitsatz bei der Entwicklung von Anwendungen mit Rails ist „Convention over Configuration“. Durch Reflektion ist ein generischer Zugriff auf Methoden und Variablen von Objekten zur Laufzeit möglich [Mc98]. Im Zusammenhang introspektiver Objects-Methoden besteht die Idee darin, introspektive Objekte durch Variablen zu definieren, die bestimmten Konventionen unterliegen und so durch Reflektion zur Laufzeit gefunden werden.

Im Beispiel lässt sich die folgende Methode in der Frameworkklasse spezifizieren, die das von einer Variable mit Namen `STRATEGY` definierte Objekt zurückliefert:

```
abstract class Reader {

  private LineStrategy strategy;

  @Introspective
  public final LineStrategy getStrategy() {
    if(strategy == null) {
      strategy = FieldFinder.getFieldByName(
        this, LineStrategy.class, "STRATEGY");
    }
    return strategy;
  }
```

In der Subklasse muss nun lediglich eine finale Variable vom entsprechenden Typ definiert werden. Das Ergebnis des Methodenaufrufs ist identisch mit dem Ergebnis einer manuell in der Subklasse implementierten Methode. Anpassung findet in diesem Fall nicht durch Implementieren einer Methode statt, sondern durch Deklarieren von finalen Variablen. Dieses introspektive Programmiermodell wird *ObjectByName-Methode* genannt.

Auf der Metaebenen M2 wird eine ObjectByName-Methode formal durch die folgende kontextfreie Grammatik beschrieben:

***ObjectByNameMethod_M2*:**

FinalModifiers IntrospectiveType *SimpleName* **( ) {**
if (*SimpleName*$_{ReturnedVariableName}$ **== null) {**
SimpleName$_{ReturnedVariableName}$ **= FieldFinder.getFieldByName (**
this, *SimpleName*$_{ReturnedVariableName}$ **. class ,** " *StringLiteral* " **) ;**
} return *SimpleName*$_{ReturnedVariableName}$**; }**

Durch das erste Nichtterminal *StringLiteral* wird der Name der Variablen spezifiziert. Eine weitere Bedingung, die nicht durch die Grammatik abgedeckt wird, ist die Existenz einer privaten Variable, welche lediglich im Methodenrumpf der ObjectByName-Methode verwendet wird. Diese Variable wird durch das Nichtterminal $SimpleName_{ReturnedVariableName}$ repräsentiert.

Anpassungen auf der Metaebene M1 wurden formal bereits auf Seite 93 durch das Nichtterminal ***IntrospectiveObjectDeclaration*** beschrieben. In diesem Zusammenhang ist der Name der Variablen entsprechend zu wählen. In Abbildung 44 ist das introspektive Modell einer Anpassung im Whitebox Modeler dargestellt.

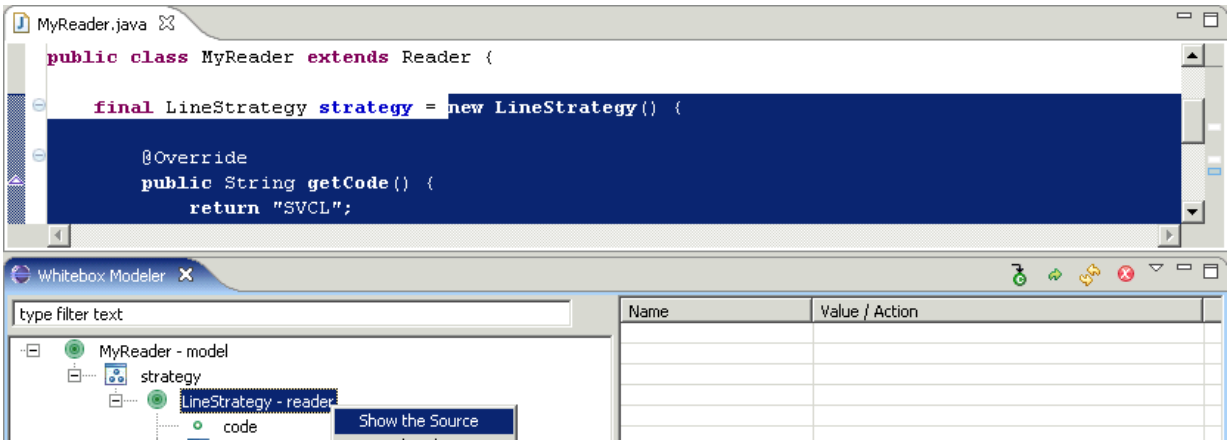


Abbildung 44: Modellierungssicht einer ObjectByName-Methode

Im Folgenden sollen introspektive Objects-Methoden, die Beziehungen zu mehreren Objekten herstellen können, besprochen werden. Als Beispiel sei hier die Beziehung zwischen den Klassen `Reader` und `LineStrategy` mit ihrer tatsächlichen Kardinalität gewählt (siehe Abbildung 35 auf Seite 81). Auf der Metaebene M2, auf der die Objects-Methode durch Deklaration einer gekennzeichneten Methode spezifiziert wird, gibt es im Wesentlichen zwei Realisierungsmöglichkeiten: einerseits mittels eines Arrays, andererseits mit Klassen der Collections-API [Ec06a].

ObjectsArray-Methode

Unter expliziter Verwendung eines Arrays wird die Beziehung auf der Metaebene M2 folgendermaßen definiert:

```
@Introspective
abstract LineStrategy[] getStrategies();
```

Eine formale Spezifikation der Deklaration der Methode ist wie folgt:

ObjectsArrayMethod_M2:
`@Introspective` { *Modifier* } ***IntrospectiveType*** `[ ]`
SimpleName `( ) ;`

Das zur Anpassung unterstützte Programmiermodell ist im Falle des Beispiels folgendermaßen:

```
LineStrategy[] getStrategies() {
  return new LineStrategy[] {
    new TestStrategy1(), new TestStrategy2()};
}
```

Formal lässt sich dies in der folgenden Form spezifizieren:

***ObjectsArrayMethod_M1**:*
{ *Modifier* } ***IntrospectiveType*** **[ ]**
SimpleName **( ) {**
return ***ArraySpecification*** **}**

Auch in diesem Fall ist eine Spezifikation des zurückgegebenen Arrays in der Form *by-value* und *by-constant* möglich:

***ArraySpecification**:*
new ***IntrospectiveType*** **[ ] {** { ***NewIntrospectiveObject*** } **} ;**

Bezüglich der Reproduzierbarkeit der Modellsicht im Falle der Spezifikation *by-constant* sei folgendes angemerkt. Das von der Methode zurückgelieferte Array kann anschließend verändert werden und weist dann nicht mehr die vom Modell repräsentierten Eigenschaften auf. Die Forderung nach Reproduzierbarkeit bezieht sich jedoch darauf, dass das Ergebnis des Methodenaufrufes unabhängig von Laufzeitbedingungen sein muss. Dies ist gegeben und somit ist die Rückgabe *by-constant* ein valides introspektives Programmiermodell.

ObjectsList-Methode

Die Verwendung der Collections-API führt für das Beispiel zu folgender Spezifikation der Beziehung:

```
@Introspective
abstract List<LineStrategy> getStrategies();
```

Das in diesem Fall zu verwendende Programmiermodell zum Anpassen stellt sich beispielsweise folgendermaßen dar:

```
List<LineStrategy> getStrategies() {
  List<LineStrategy> result = new ArrayList<LineStrategy>();
  result.add(new TestStrategy1());
  result.add(new TestStrategy2());
  return result;
}
```

Leider handelt es sich dabei um kein besonders elegantes Programmiermodell. Im Beispielcode wurden die einzelnen Objekte *by-value* spezifiziert, *by-constant* ist ebenfalls möglich. Eine

vollständige formale Spezifikation der ObjectsList-Methode auf den Metaebenen M1 und M2 wird durch die folgenden Nichtterminale erreicht:

***ObjectsListMethod_M2*:**
`@Introspective` { *Modifier* } ***IntrospectiveType*** **`[ ]`**
SimpleName **`( ) ;`**

***ObjectsListMethod_M1*:**
`{` *Modifier* **`}`** **`List <`** ***IntrospectiveType*** **`>`**
SimpleName **`( ) {`**
`List <` ***IntrospectiveType*** **`>`** $SimpleName_{ListVariableName}$ **`=`**
`new ArrayList <` ***IntrospectiveType*** **`> ( ) ;`**
{ $SimpleName_{ListVariableName}$ **`. add ( NewIntrospectiveObject ) ;`** }
`return` $SimpleName_{ListVariableName}$ **`; }`**

Auch in diesem Fall trifft die bereits für ObjectsArray-Methoden getroffene Aussage bezüglich der Reproduzierbarkeit zu.

ObjectsByType-Methode

Die bisher vorgestellten Programmiermodelle ObjectsArray-Methode und ObjectsList-Methode zum introspektiven Herstellen einer mengenwertigen Assoziation lassen ein wenig die von introspektiven Programmiermodellen geforderte deklarative Eleganz vermissen. Ebenso entsteht bei Rückgabe *by-constant* Redundanz, was zur Einführung der ObjectByName-Methode geführt hat (siehe Seite 94). Dieselbe Argumentation führt zum introspektiven Programmiermodell der *ObjectsByType-Methode*, die eine elegante Spezifikation von Mengen von Objekten durch einfache Variablendeklaration ermöglicht. Auf der Metaebene M2 wird eine ObjectsByType-Methode beispielsweise folgendermaßen spezifiziert:

```
private List<LineStrategy> strategies;

@Introspective
public final List<LineStrategy> getStrategies() {
  if(strategies == null) {
    strategies = FieldFinder.getFieldsByType
      (this, LineStrategy.class);
  }
  return strategies;
}
```

Als Ergebnis dieses Methodenaufrufs wird eine Liste zurückgeliefert, die alle in dieser Klasse definierten finalen Variablen vom Typ `LineStrategy` enthält. Zur Anpassung müssen Subklassen also lediglich konstante finale Variablen definieren. Formal ist die ObjectsByType-Methode auf der Metaebene M2 folgendermaßen spezifiziert:

ObjectsByTypeMethod_M2:

FinalModifiers **`List <`** ***IntrospectiveType*** **`>`** *SimpleName* **`( ) {`**
`if (` $SimpleName_{ReturnedListVariableName}$ **`== null ) {`**
$SimpleName_{ReturnedListVariableName}$ **`= FieldFinder.getFields`**
`( this ,` ***IntrospectiveType*** **`. class ) ; }`**
`return` $SimpleName_{ReturnedListVariableName}$ **`; }`**

Soll beispielsweise wie in Abbildung 31 dargestellt, das Lesen von ServiceCalls realisiert werden, werden unter Anderem zwei String-Felder `customerName` und `customerId` von Positionen 4 bis 10 und 11 bis 15 eingelesen werden, so ergibt dies unter Verwendung einer ObjectsByType-Methode folgenden Quelltext:

```
public class Reader {
  final LineStrategy serviceCall = new LineStrategy() {
    final StringField customerName = new StringField() {
      public int getEnd() {
        return 10;
      }

      public int getStart() {
        return 4;
      }
    };

    final StringField customerId = new StringField() {
      public int getEnd() {
        return 15;
      }
      public int getStart() {
        return 11;
      }
    };
  ...

    public void read(String line) {
      Service.INSTANCE().registerServiceCall(
        this.customerName.getString(line),
        this.customerId.getString(line), ...);
    }
  };
}
```

Eine introspektive Modellsicht hierauf ist in Abbildung 45 dargestellt.

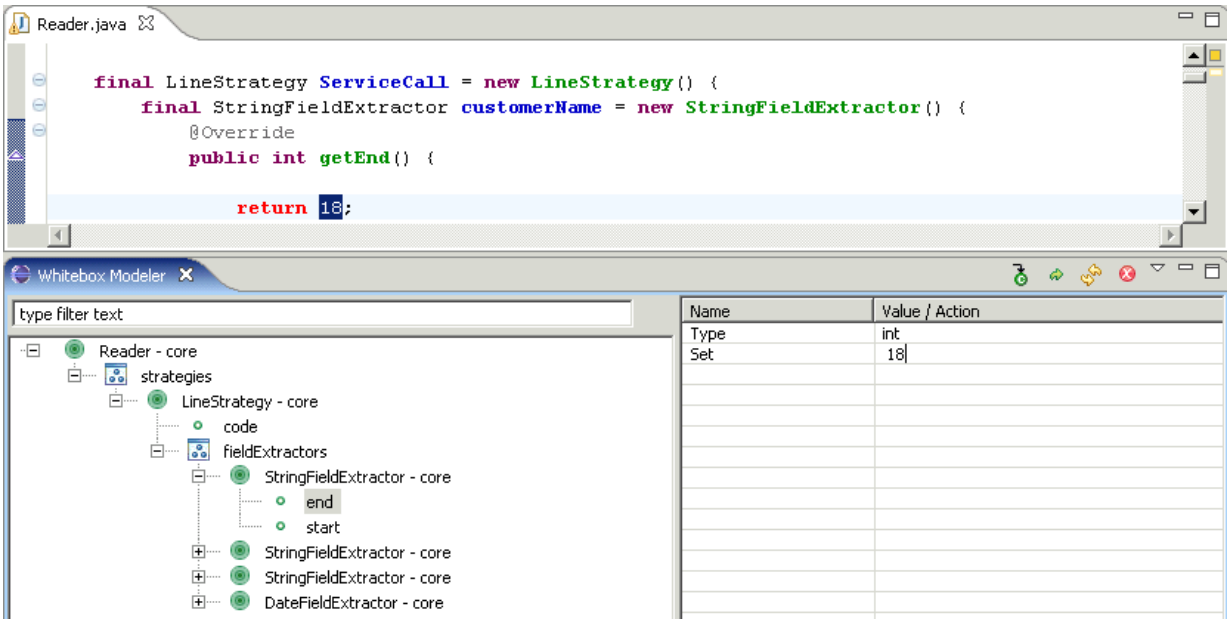


Abbildung 45: Modellsicht einer ObjectByType-Methode

5.2.4 Recordable-Methode

Mit den bisher aufgezeigten introspektiven Methoden verfügt die Whitebox Introspektion bereits über die für einen Metamodellierungsansatz grundlegende Mächtigkeit. So lassen sich mithilfe der Value-Methode elementare Eigenschaften modellieren. Beziehungen lassen sich durch Objects-Methoden herstellen. Die Modelle manifestieren sich direkt durch Quelltexte einer objektorientierten Programmiersprache. Durch Introspektion ist es möglich, die Modelle auf einem höheren Abstraktionsniveau einzusehen und zu manipulieren.

Das konkrete Programmiermodell, in dem die Modellierungsinformationen spezifiziert werden, ist von großer Bedeutung, da das direkte Manipulieren des Quelltextes möglich ist und so gut wie möglich unterstützt werden soll. In einigen Anwendungsfällen entspricht das Programmiermodell der bisher vorgestellten introspektiven Methoden nicht der intuitiven Vorgehensweise. Eine weitere Möglichkeit stellt die *Recordable-Methode* dar. Im Folgenden wird diese Art des introspektiven Programmierens an einem Beispiel aus dem Bereich des Dateilese-Frameworks motiviert und vorgestellt.

Wie bereits in Abschnitt 5.1.1 auf Seite 77 dargestellt, besteht das Auslesen der Textdatei im Wesentlichen aus dem Identifizieren von typisierten Feldern an bestimmten Stellen einer Textzeile. Wie im konzeptuellen Klassendiagramm in Abbildung 35 auf Seite 81 dargestellt, besteht eine Möglichkeit der Realisierung dieser Anforderung in einer expliziten Modellierung der Felder durch eine Klassenhierarchie. Die Beziehung wird in diesem Fall durch eine Objects-Methode realisiert. Elementare Eigenschaften der Felder (z.B. `start` und `end`) werden durch Value-Methoden realisiert. In diesem konkreten Fall ist unter Berücksichtigung der bisher vorgestellten Programmiermodelle die zuletzt vorgestellte Realisierung mittels einer ObjectsByType-Methode und mehreren Value-Methoden am elegantesten.

Alternativ realisiert die folgende, wesentlich kompaktere Spezifikation das gleiche konzeptuelle Modell:

```
public class Reader {
  final LineStrategy serviceCall = new LineStrategy() {
    private String customerName;
    private String customerId;
    ...

    public void readLine(FieldExtractor extractor) {
      customerName = extractor.getString(4, 10);
      customerId = extractor.getString(11, 15);
      ...
    }

    public void doBusinessLogic() {
      Service.INSTANCE().registerServiceCall(
        customerName, customerId, callTypeCode, dateOfCall);
    }
  };
```

In diesem Fall werden introspektiv analysierbare Informationen als Parameter von Methodenaufrufen spezifiziert. Im Klassendiagramm in Abbildung 46 ist im oberen Teil die Realisierung mit ObjectsByType- und Value-Methoden dargestellt. Im unteren Teil ist die äquivalente Realisierung durch eine Recordable-Methode dargestellt. Objekte wurden durch Methodenaufrufe ersetzt, primitive Werte durch Parameter.

Dieses introspektive Programmiermodell wird im Folgenden *Recordable-Methode* genannt. Eine Recordable-Methode wird auf der Metaebene M2 als Methode mit mindestens einem Parameter spezifiziert:

***RecordableMethod_M2**:*
`@Introspective` { *Modifier* } **`void`** *SimpleName*
`(` ***RecordableType*** *SimpleName*$_{RecordableVariableName}$
{ **`,`** *FormalParameter* } **`)`** **`;`**

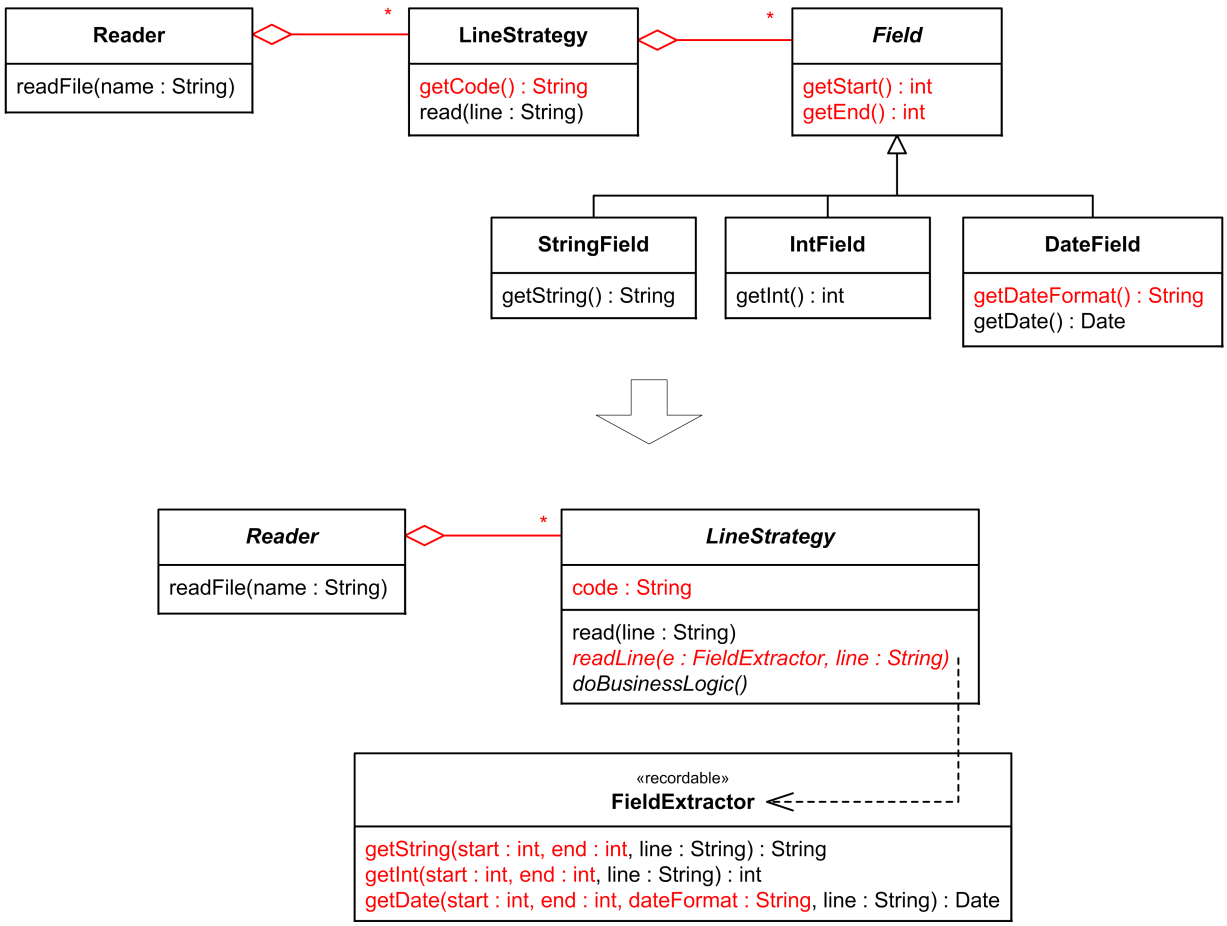


Abbildung 46: Auslesen von Parametern mittels einer Recordable-Methode

Der erste Parameter ist ein ***RecordableType***, der durch eine Annotation vom Typ `Recordable` gekennzeichnet ist und Methoden definiert, deren Verwendung für eine introspektive Realisierung relevant ist:

```
@Recordable
public class FieldExtractor {

  public String getString(int start, int end) {
    return line.substring(start, end + 1).trim();
  }
  ...
```

Als Anpassung auf der Metaebene M1 sind im Methodenrumpf lediglich Aufrufe auf dem übergebenen Objekt erlaubt:

***RecordableMethod_M1**:*
{ *Modifier* } **void** *SimpleName*
(***RecordableType*** *SimpleName*$_{RecordableVariableName}$ **) {**
{ ***RecordableCall*** }
}

Diese Methodenaufrufe werden durch das Nichtterminal ***RecordableCall*** spezifiziert:

RecordableCall:
SimpleName **=** ***MethodInvocationOnRecoridble***
MethodInvocationOnRecoridble
RecordableMethodInvocation

***MethodInvocationOnRecordable**:*
SimpleName$_{RecoridbleVariableName}$ **.**
SimpleName **(**
[***RecordableElementaryParameter***
{ **,** ***RecordableElementaryParameter*** }] **) ;**

RecordableElementaryParameter:
AbstractValue
NewInstrospectiveObject

***RecordableMethodInvocation**:*
SimpleName **(** *SimpleName*$_{RecordableVariableName}$ **) ;**
Name **.** *SimpleName* **(** *SimpleName*$_{RecordableVariableName}$ **) ;**
ThisExpression **.** *SimpleName* **(** *SimpleName*$_{RecordableVariableName}$ **) ;**

Das um das Konstrukt der Recordable-Methode erweiterte Metametamodell der Whitebox Introspektion ist in Abbildung 47 dargestellt. Interessant ist hierbei, dass es nun verschiedene Möglichkeiten gibt, ein konzeptuelles Metamodell durch ein introspektives Whitebox Framework zu realisieren. Beziehungen zwischen Objekten lassen sich sowohl durch Objects-Methoden als auch durch Recordable-Methoden realisieren. Elementare Eigenschaften lassen sich durch Value-Methoden und durch Parameter von Recordable-Methoden realisieren.

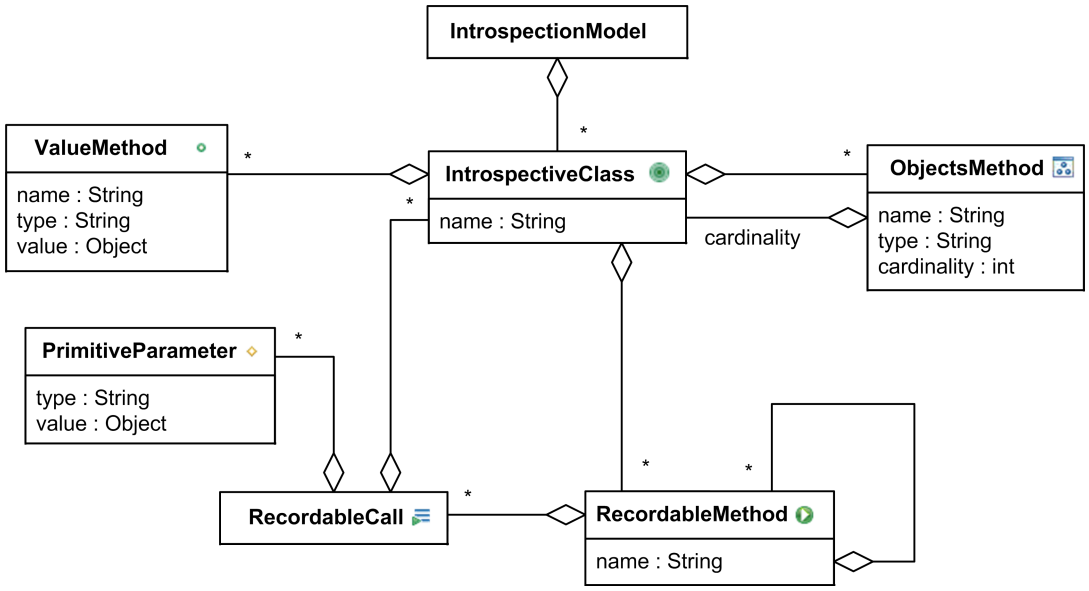


Abbildung 47: Vollständiges Metametamodell der Whitebox Introspektion

5.3 Vorgehen im Überblick

Der gesamte Ablauf der Whitebox Introspektion ist in Abbildung 48 schematisch dargestellt. Grundvoraussetzung zur Durchführung introspektiver Analysen ist das Vorhandensein einer Post-IntelliJ-IDE. Dies ermöglicht direkten Zugriff auf den abstrakten Syntaxbaums des bearbeiteten Systems. Handelt es sich um ein System, das durch Anpassen eines introspektiven Whitebox Frameworks erstellt wurde ist es möglich das Metamodell des Frameworks durch Metamodellintrospektion zu ermitteln. Das Metamodell wird durch die Struktur der introspektiven Frameworkklassen und der verwendeten introspektiven Methoden bestimmt.

Auf Basis des Metamodells können nun konkrete Klassen mit konkreten Anpassungen introspektiert werden. In diesem Fall handelt es sich um Modell-Introspektion, da es sich beim Ergebnis der Introspektion um das Modell handelt. Das Modell kann auf verschiedene Art und Weise visualisiert und bearbeitet werden. Einzelne Bearbeitungsschritte führen dazu, dass Änderungen im Quelltext vorgenommen werden, die wiederum zu einer Aktualisierung des Modells und der Visualisierung führen.

Das Endresultat besteht in „normalen“ Programmiersprachkonstrukten, die wiederum vom Compiler zu ausführbaren Konstrukten weiterverarbeitet werden und ausgeführt werden können.

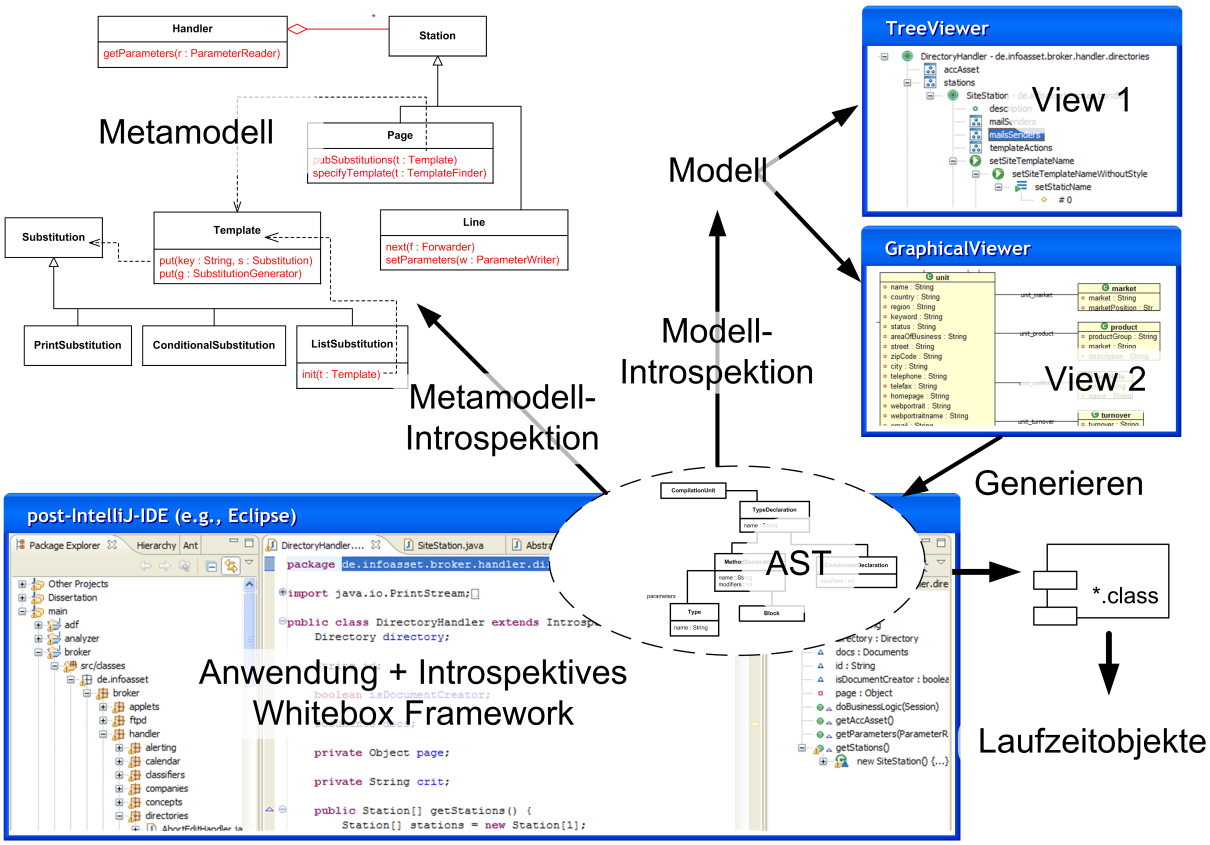


Abbildung 48: Überblick: Whitebox Introspektion

5.4 Anatomie der DSL eines introspektiven Whitebox Frameworks

Bei domänenspezifischen Sprachen, die durch Whitebox Introspektion realisiert werden, handelt es sich um formale Sprachen. In Tabelle 6 ist eine Beschreibung der Sprachen entsprechend den in Abschnitt 2.1 vorgestellten Kriterien dargestellt.

Abstrakte Syntax	Die abstrakte Syntax der DSL eines introspektiven Whitebox Frameworks wird durch ein Metamodell, welches das in Abschnitt 5.2 beschriebene Metametamodell instanziiert, beschrieben. Die Definition der abstrakten Syntax erfolgt durch annotierte introspektive Methoden.
Konkrete Syntax	Die Modelle lassen sich einerseits in Form einer domänenspezifischen Sprache im Whitebox Modeler bearbeiten. Es sind auch spezielle maßgeschneiderte Visualisierungen möglich. Andererseits ist das direkte Bearbeiten als Quelltext möglich.
Serialisierungssyntax	Die Modelle werden textuell als Quelltext der Basisprogrammiersprache repräsentiert und auch in dieser Form serialisiert.
Semantik	Die Semantik der Modelle ergibt sich aus der Semantik der darunter liegenden Basisprogrammiersprache.

Tabelle 6: Anatomie der DSL eines introspektiven Whitebox Frameworks

5.5 Migration von Modellen

Wie in Abschnitt 2.3.9 dargestellt, entsteht eine spezielle Herausforderung an eine Realisierung modellgetriebener Softwareentwicklung, wenn sich die Anpassungsmöglichkeiten des Frameworks weiterentwickeln. Wünschenswert ist eine automatische Anpassung der existierenden Modelle. Aufgrund der internen Modellrepräsentation introspektiver Whitebox Frameworks werden bestimmte Migrationsszenarien direkt durch existierende Refactorings in Post-IntelliJ-IDEs realisiert. Beispielsweise soll eine Eigenschaft, die durch eine Value-Methode realisiert wurde, umbenannt werden. In diesem Fall muss die Value-Methode im Framework-Kern und in allen implementierenden Subklassen umbenannt werden. Dies ist ein existierendes Standard-Refactoring.

Zur Umsetzung komplexerer Szenarien lassen sich moderne Post-IntelliJ-IDEs instrumentalisieren und ermöglichen einen Zugriff auf den abstrakten Syntaxbaum des Quellcodes. Ein Beispiel hierfür ist das Projekt Jackpot der IDE Netbeans [My07].

5.6 Generative modellgetriebene Softwareentwicklung

Die momentan vorherrschende Realisierungsform modellgetriebener Softwareentwicklung ist der Ansatz der *generativen* modellgetriebenen Softwareentwicklung [SV05]. Dabei werden aus Modellen durch Transformationen ausführbare Konstrukte generiert. Im Verständnis dieser Arbeit handelt es sich bei diesen Konstrukten um Frameworkanpassungen. Ausgangspunkt generativer modellgetriebener Softwareentwicklung ist also ein vorhandenes Framework mit entsprechenden Erweiterungsmöglichkeiten. Durch Metamodellierung wird eine domänenspezifische Sprache entworfen, die den Erweiterungsmöglichkeiten des Frameworks entspricht. Der Frameworknutzer nutzt zur Problemlösung die erstellte domänenspezifische Sprache, welche

sich auf einem angemessenen Abstraktionsniveau befindet und kommt somit nicht mit den Anpassungskonstrukten direkt in Berührung.

5.6.1 Zwei Phasen generativer modellgetriebener Softwareentwicklung

Analog zum Lebenszyklus eines Frameworks (siehe Seite 27) gibt es zwei Phasen, in denen generative modellgetriebene Softwareentwicklung betrieben wird, und ebenfalls analog gibt es zwei verschiedene Rollen von Entwicklern, die die Tätigkeiten ausführen.

In der ersten Phase, die der Entwicklung des Frameworks entspricht, wird das Metamodell der DSL entworfen, das die Erweiterungsmöglichkeiten des Frameworks widerspiegelt. Wie im Abschnitt 2.3.5 auf Seite 35 dargestellt, lässt sich das Erstellen von DSLs durch die Verwendung eines gemeinsamen Metametamodells unterstützen. Ebenfalls in diese Phase gehört das Spezifizieren der Abbildung von Modellen auf konkrete Implementierungsartefakte. Zu diesem Zweck werden im Allgemeinen Templates und Generatoren benutzt. In [SV05] wird diese Phase *Domänenarchitekturentwicklung* genannt. Die in dieser Phase aktiven Rollen sind die des *Sprachdesigners* und des *Transformationsentwicklers*.

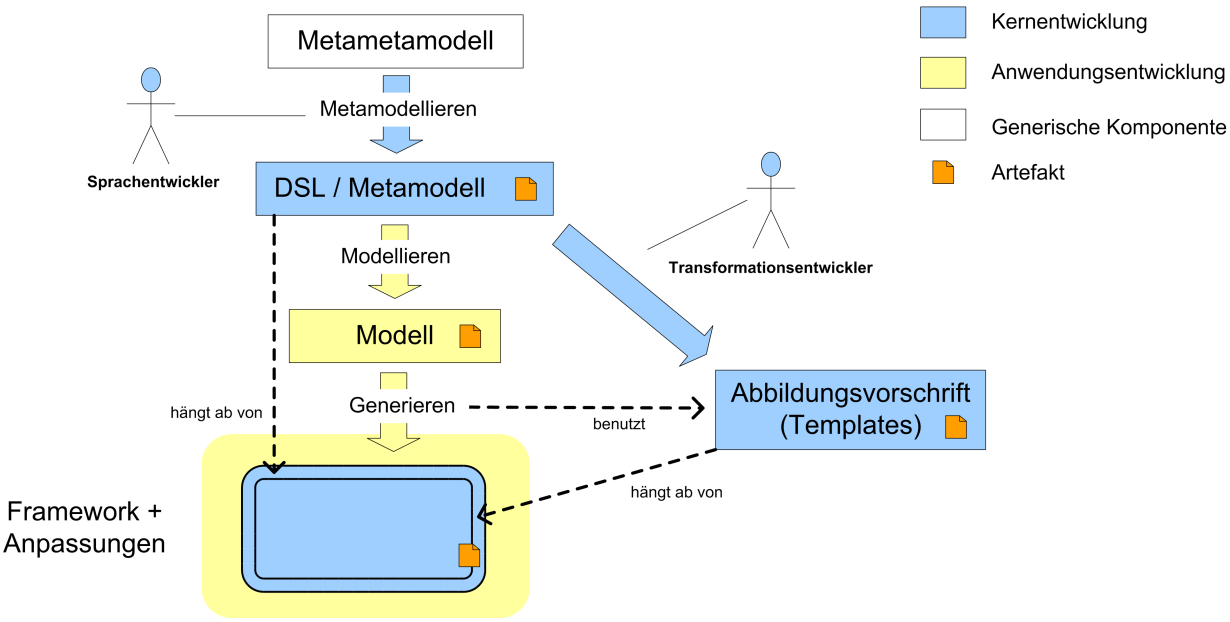


Abbildung 49: Zwei Phasen generativer modellgetriebener Softwareentwicklung

In der zweiten Phase, die der Anpassungsphase von Frameworks entspricht, werden die Anforderungen an ein zu erstellenden Softwaresystem unter Verwendung der domänenspezifischen Sprache modelliert. In [SV05] wird diese Phase *Anwendungsentwicklung* genannt. Hierbei ist die konkrete Syntax der DSL von entscheidender Bedeutung. Analog zur Framework-Terminologie wird diese Phase von Framework-Nutzern durchgeführt, die Anpassungen mit Hilfe der DSL anstatt durch händisches Erstellen von Anpassungsartefakten durchführen. Als Ergebnis dieser Phase werden die erstellten Modelle auf Anpassungen des Frameworks abgebildet. Der gesamte Ablauf ist in Abbildung 49 dargestellt.

Es existiert eine Reihe von konkurrierenden Ansätzen, die dem vorgestellten schematischen Ablauf entsprechen und sich lediglich durch die verwendeten Metametamodelle und Transformationssprachen unterscheiden. All diese Ansätze werden im Rahmen dieser Arbeit unter der Bezeichnung generative modellgetriebene Softwareentwicklung geführt.

5.6.2 Beispiel: Entwicklung einer DSL mit openArchitectureWare

Die in allen Ansätzen vorhandenen Kernkonzepte werden im Folgenden anhand des Generator-Frameworks *openArchitectureWare* [Op06] illustriert. Dabei handelt es sich um ein Open-Source Projekt, das sich selbst als „Werkzeug zum Erstellen von Werkzeugen zur modellgetriebenen generativen Softwareentwicklung“ bezeichnet. Hierfür wird der Prozess der generativen modellgetriebenen Softwareentwicklung am eingeführten Beispiel des Textdatei-Lesers vorgestellt. Das Ziel der Kernentwicklungsphase besteht darin, einerseits ein Framework zu erstellen, welches das Problem löst. Hierfür wird das von [Fo05b] bereitgestellte Framework benutzt. Andererseits soll vom Sprachentwickler eine externe domänenspezifische Sprache erstellt werden, deren Modelle in Anpassungsartefakte der Basisprogrammiersprache transformiert werden können. OpenArchitectureWare benutzt zum Erstellen vom domänenspezifischen Sprachen Metamodelle, die mit Hilfe des Frameworks *Eclipse Modeling Framework* (EMF) [Bu03] erstellt werden. Das Metametamodell von EMF ist Ecore. Es existiert ein Eclipse-Plugin, mit dessen Hilfe Ecore-Modelle erstellt werden können. In Abbildung 50 ist ein Ausschnitt der Bearbeitung des für das Beispiel erstellten Ecore-Metamodells dargestellt, welches die in Abbildung 33 dargestellten Konzepte realisiert.

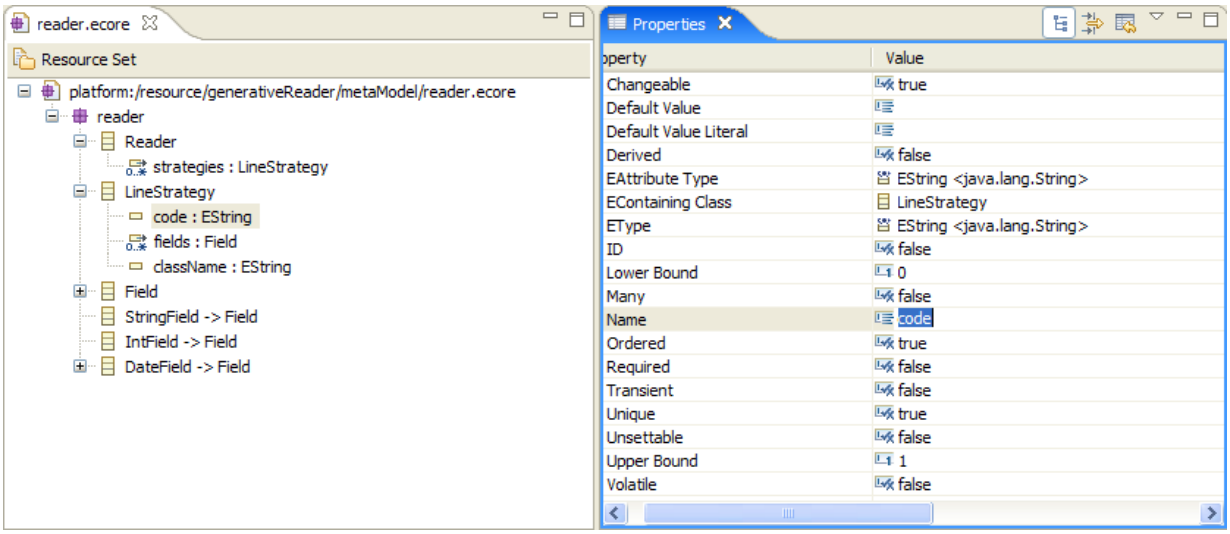


Abbildung 50: Metamodellierung mit EMF

In einem zweiten Schritt generativer modellgetriebener Softwareentwicklung werden in der Kernentwicklungsphase Abbildungsvorschriften, die die Abbildung der vom Framework-Nutzer erstellten Modelle auf konkrete Anpassungen spezifizieren, erstellt. Da es sich hierbei um so genannte *Codetransformationen* (model-to-code) handelt [CH03] wird die Abbildung unter Benutzung von Generatoren und Templates bewerkstelligt. OpenArchitectureWare stellt zu diesem Zweck die Template-Sprache *Xpand* zur Verfügung. Hierfür wird im Allgemeinen zu-

nächst Beispielcode erstellt, der als Anschauungsobjekt zum Erstellen der Transformationen verwendet wird. In unserem Beispielframework kann die in Abbildung 31 dargestellte Konfiguration durch folgenden Quellcode konfiguriert werden:

```
public class MyReader extends AbstractReader {
  protected void configure() {
    LineStrategy s = new LineStrategy("SVCL",
      ServiceCall.class);
    s.addField(new StringField("customerName", 4, 10));
    s.addField(new StringField("customerID", 11, 15));
    s.addField(new StringField("callTypeCode", 16, 19));
    s.addField(new DateField("dateOfCall", "", 20, 27));
    addStrategy(s);
    ...
```

Hieraus lässt sich folgendes Xpand-Template ableiten:

```
«IMPORT reader»
«DEFINE Root FOR Reader»

«FILE "MyReader.java"»
import util.*;

public class MyReader {
  protected void configure() {
  «FOREACH strategies AS s»
    LineStrategy s = new LineStrategy("«s.code»", «s.classNa-
me».class);
    «FOREACH s.fields AS f»
      «IF StringField.isInstance(f)»
    s.addField(new StringField("«f.name»", «f.start»,
«f.end»));
      «ENDIF»
      «IF IntField.isInstance(f)»
    s.addField(new IntField("«f.name»", «f.start», «f.end»));
      «ENDIF»
      «IF DateField.isInstance(f)»
    s.addField(
      new DateField("«f.name»", "«f.dateFormat»", «f.start»,
«f.end»));
      «ENDIF»
    «ENDFOREACH»
    addStrategy(s);
  «ENDFOREACH»
«ENDFILE»
«ENDDEFINE»
```

Basierend auf dem Ecore-Metamodell können Plugins generiert werden, die einen Editor realisierten, welcher die Instanziierung des Metamodells unterstützt. Hiermit ist die Kernentwicklungsphase abgeschlossen, und das Framework sowie die domänenspezifische Sprache können von Anwendungsentwicklern zur Problemlösung verwendet werden. Die Modellierungsansicht ist in einem Screenshot in Abbildung 51 dargestellt.

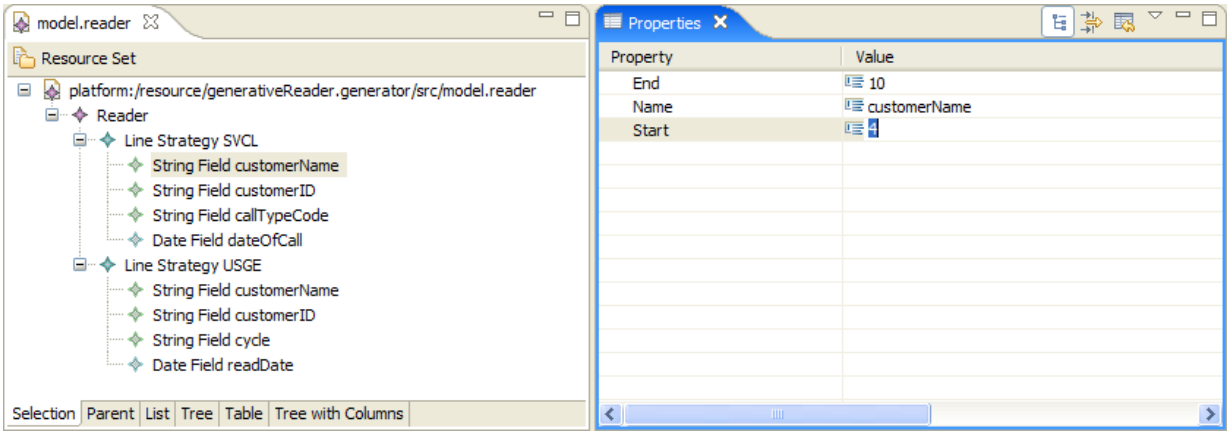


Abbildung 51: EMF – Modellierungssicht

5.6.3 Diskussion generativer modellgetriebener Softwareentwicklung

Generative modellgetrieben Softwareentwicklung ist eine Möglichkeit, modellgetriebene Softwareentwicklung zu betreiben. Es werden alle Vorteile des modellgetriebenen Ansatzes realisiert. Anpassungen können auf einem hohen Abstraktionsniveau durchgeführt werden. Irrelevante Details der Lösungsimplementierung werden vor dem Framework-Nutzer versteckt und somit wird die wahrgenommene Komplexität reduziert.

Dem erreichten Vorteil stehen Nachteile gegenüber. Der im Rahmen dieser Arbeit vorstechendste Nachteil ist die fehlende Integration zwischen den Modellierungsartefakten und dem darunter liegenden System. Die Entwicklung des Framework-Kerns und der domänenspezifischen Sprache erfolgen getrennt voneinander, beide Dinge stehen in keiner expliziten Beziehung zueinander. Die abstrakte Syntax der domänenspezifischen Sprache repräsentiert die Erweiterungsmöglichkeiten des Frameworks, diese Beziehung ist jedoch implizit. Falls sich beispielsweise der Framework-Kern weiterentwickelt, so muss die domänenspezifische Sprache händisch nachgezogen werden. Dies betrifft die bereits erstellten Modelle, das Metamodell sowie die Abbildungsvorschriften.

Ein weiteres Problem besteht in der Integration von Modellen mit händisch erstelltem Programmcode (siehe Abschnitt 2.3.7). Da es sich um externe domänenspezifische Sprachen handelt, ist dieser bedeutende Integrationsaspekt aufgrund der symbolischen Distanz kaum zu realisieren.

Zusammenfassend ist die Entwicklung eines Frameworks mit dazugehöriger domänenspezifischer Sprache unter Benutzung des generativen Ansatzes aufwändig und erfordert großen ma-

nuellen Aufwand. Dieser Aufwand kann durch Werkzeugunterstützung reduziert werden, das konzeptuelle Problem der fehlenden Integration bleibt jedoch bestehen. Ein weiteres Problem besteht darin, dass sich keine Konvergenz bzgl. der eingesetzten Werkzeuge abzeichnet. Dies ist umso schwerwiegender, da es sich um komplexe, schwergewichtige Werkzeuge handelt.

6 IMF – Eine Realisierung von IMDD

In den beiden vorangegangenen Kapiteln wurden bereits Werkzeuge zur Realisierung introspektiver modellgetriebener Softwareentwicklung vorgestellt. In diesem Kapitel wird die technische Realisierung dieser Werkzeuge vorgestellt. Entsprechend den zwei vorgestellten Frameworkarten existiert Unterstützung zur Introspektion von Blackbox und Whitebox Frameworks. Bei der Entwicklung dieser Werkzeuge hat sich herausgestellt, dass ein großer Teil der benötigten Funktionalitäten von beiden Introspektionsarten benötigt wird. Dem Rechnung tragend wurde ein Framework entwickelt, das die in beiden Fällen benötigten Szenarien unterstützt. Der Name dieses Frameworks ist *Introspective Modeling Framework* – IMF. Die konkreten Werkzeuge zur Unterstützung von Whitebox und Blackbox Introspektion sind Anpassungen von IMF an die jeweils konkreten Anforderungen (siehe Abbildung 52).

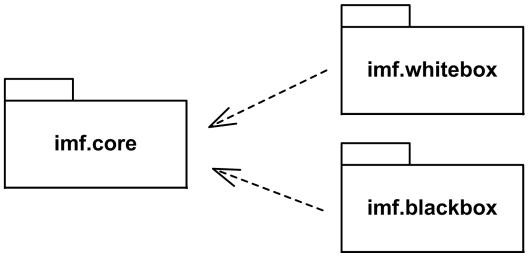


Abbildung 52: IMF und Anpassungen an Whitebox und Blackbox Introspektion

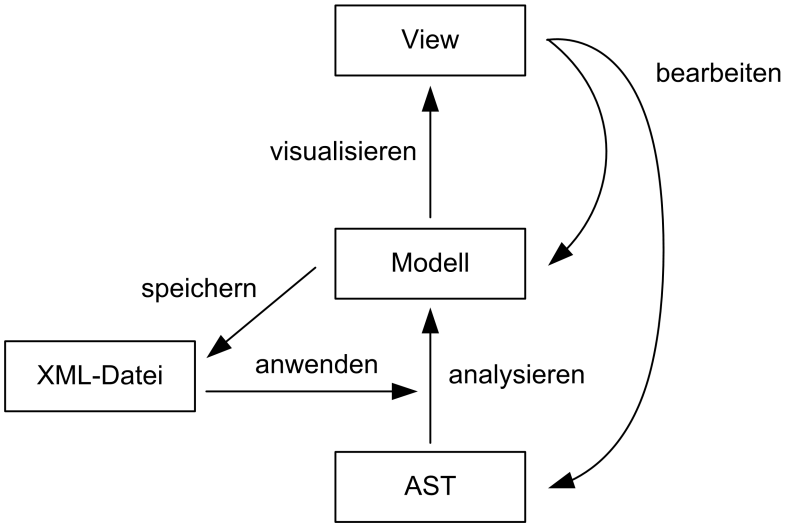


Abbildung 53: IMF: Überblick

In diesem Kapitel werden Architektur und Funktionsweise des IMF vorgestellt. In Abbildung 53 sind die wesentlichen von IMF unterstützten Entitäten und Aktivitäten überblicksartig dar-

gestellt. IMF ist eingebettet in die Post-IntelliJ-IDE Eclipse und hat somit über einen abstrakten Syntaxbaum (engl.: Abstract Syntax Tree – AST) Zugriff auf den Quelltext der Programme. Auf Basis des AST werden Modelle durch introspektive Analyse gewonnen. Diese lassen sich in einem XML-Format abspeichern und können somit als Eingabe für eine neuerliche Analyse dienen. Zur Visualisierung der Modelle dienen Sichten (engl.: *Views*), die das Manipulieren sowohl des Modells als auch des AST – und somit des Quelltexts – ermöglichen.

Die sich aus den unterstützten Aktivitäten ergebenden Anforderungen an die Repräsentation der Modelle, sowie deren Umsetzung werden in Abschnitt 6.1 näher erläutert. Das Erstellen der Modelle wird von so genannten *Anlyzern* durchgeführt. Deren Architektur und Funktionsweise wird in Abschnitt 6.2 besprochen. In Abschnitt 6.3 wird die technische Realisierung der interaktiven Sichten zur Manipulation der Modelle diskutiert. Abschließend werden die Anpassungen zur Unterstützung der Blackbox und Whitebox Introspektion vorgestellt.

6.1 Modellelemente

Als zentrales Artefakt der introspektiven modellgetriebenen Softwareentwicklung werden Modelle erstellt. Modelle werden in IMF durch Objekte vom Typ `ModelElement` repräsentiert. Diese bilden eine Baumstruktur und verfügen jeweils über einen eindeutigen Identifier. `ModelElement` ist eine abstrakte Klasse, die lediglich Basisfunktionalitäten bereitstellt und nicht direkt instanziiert werden kann. Ein konkretes Modell besteht aus typisierten Subklassen, die ihrerseits anwendungsspezifische Attribute und Methoden hinzufügen. Modellelemente sind hierarchisch in einer Baumstruktur angeordnet, welche generisch durch die Eltern-Kind-Beziehung der Klasse `ModelElement` hergestellt (siehe Abbildung 54).

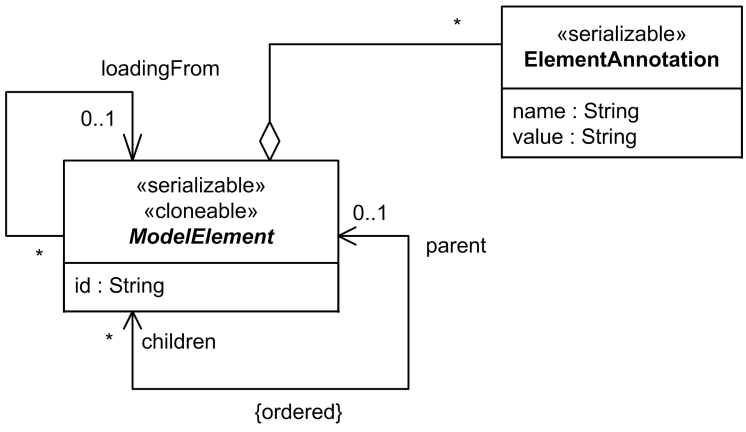


Abbildung 54: Klassendiagramm: `ModelElement`

Eine wichtige Eigenschaft aller Modellelemente besteht in der Serialisierbarkeit in Form eines XML-Streams. Dies geschieht, indem pro Modellelement ein XML-Element mit dem Klassennamen gebildet wird. Die Werte der zu serialisierenden Attribute werden als XML-Attribute serialisiert. Alle Kindsobjekte werden entsprechend als XML-Elemente kodiert.

In einigen Anwendungsfällen ist die Anpassung durch Subtypisierung nicht ausreichend. Ein Beispiel hierfür sind Positionsangaben für Modellelemente, die graphisch visualisiert werden.

Verschiedene Visualisierungen verwalten unterschiedliche Daten. Hierfür wird ein flexibler Mechanismus benötigt, mit dem beliebige Modellelemente mit Metadaten versehen werden können. Zu diesem Zweck können generisch Beziehungen zu Objekten vom Typ `ElementAnnotation` hergestellt werden. Diese stellen ein Name-Wert-Paar dar und sind ein sehr flexibler Mechanismus zur Informationsanreicherung von Modellelementen.

Eine weitere Anforderung besteht darin, Introspektion auf Basis eines existierenden abgespeicherten Modells durchzuführen. Diese Anforderung ergibt sich aus zwei Szenarien. In einem ersten Szenario sollen Verortungsinformationen für graphische Visualisierungen wiederhergestellt werden und im Ergebnis der Introspektion vorhanden sein. Dieses Szenario ist für Modelle von introspektiven Whitebox Frameworks relevant. Das abgespeicherte Quellmodell beeinflusst den Ablauf der Introspektion in diesem Szenario nicht, sondern sorgt lediglich dafür, dass die graphische Anordnung des Introspektionsergebnisses mit den Informationen aus dem Quellmodell initialisiert wird. In einem zweiten Szenario steuern abgespeicherte Modelle Teile des Introspektionsvorgangs. Dies ist beim Einlesen einer Konfiguration eines Blackbox Frameworks der Fall (siehe Abschnitt 4.4). Beide Szenarien werden realisiert, indem jedes Modellelement eine Beziehung zu einem Modellelement eines Quellmodells unterhalten kann. Während des Introspektionsvorgangs wird diese Beziehung Schritt für Schritt mit Referenzen auf das Quellmodell gesetzt.

In Abbildung 55 sind alle wesentlichen Methoden der Klasse `ModelElement` dargestellt. Interessant ist hierbei der typisierte Zugriff auf Eltern- bzw. Kindsobjekte durch die Methoden `getNextParent` und `getChildElements`.

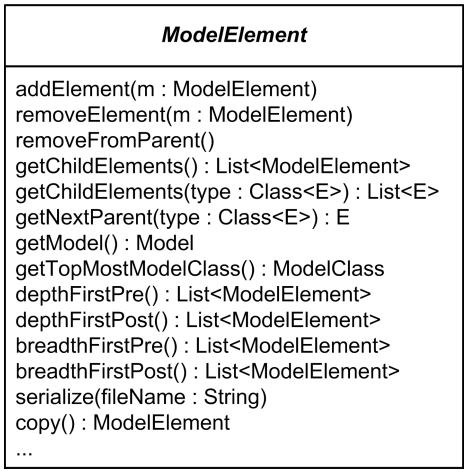


Abbildung 55: Methoden von `ModelElement`

Die in Abbildung 56 dargestellten Subklassen `MethodNode`, `ModelClass` und `Model` sind ebenfalls Teil des generischen Frameworks IMF. Objekte vom Typ `ModelClass` repräsentieren Klassen des zu analysierenden Quelltextes. Die in der Abbildung dargestellten konkreten Klassen `ConfigurableClass` und `IntrospectiveClass` sind Teil der Anpassungen von IMF an die Analyse von Blackbox und Whitebox Frameworks.

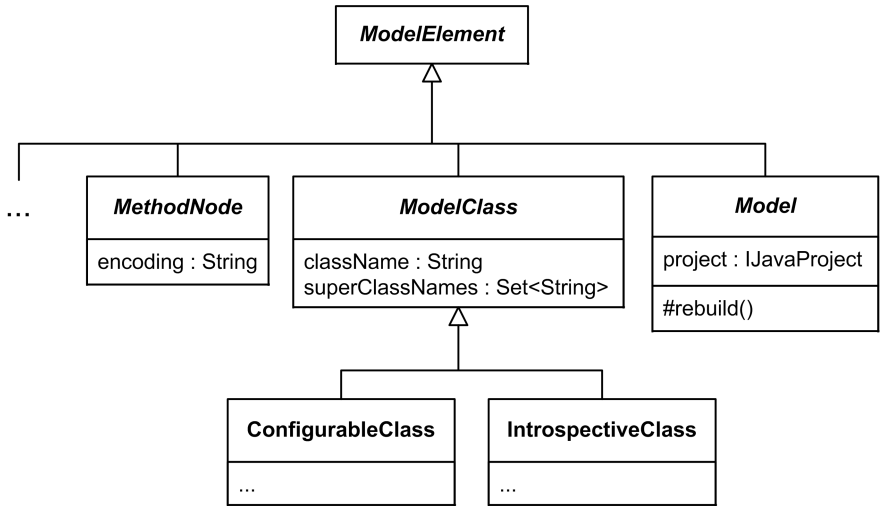


Abbildung 56: Subklassen von `ModelElement`

Subklassen von `Model` repräsentieren jeweils die Wurzel konkreter Modelle. Von jedem Modellelement aus ist durch die Methode `getModel` ein Zugriff auf das jeweilige Modellobjekt möglich. Dementsprechend eignet sich das Modellobjekt als Zugriffsmöglichkeit für an vielen Stellen benötigte Ressourcen, die pro Modell eindeutig sind. So wird zum Beispiel das Attribut `project` vom Typ `IJavaProject` benötigt, um auf die von Eclipse angebotenen Dienste zuzugreifen. Ein Zugriff auf diese Ressource ist von jedem Modellelement aus möglich:

```
IJavaProject p = this.getModel().getJavaProject();
```

6.2 Analyse

In diesem Abschnitt wird die eigentliche introspektive Analyse besprochen, die auf Basis des abstrakten Syntaxbaums der zu analysierenden Klassen Modelle erstellt. Wie bereits im Abschnitt über kontextfreie Grammatiken auf Seite 86 erwähnt, findet der Zugriff auf den AST über eine objektorientierte API statt, die jeweils Nichtterminale der Produktionen durch Objekte repräsentiert. Diese API wird vom Eclipse-Teilprojekt JDT bereitgestellt. Die abstrakte Superklasse aller Elemente des AST ist die Klasse `org.eclipse.jdt.core.dom.ASTNode`.

Den Startpunkt einer introspektiven Analyse bildet stets der Quellcode einer oder mehrerer Klassen bzw. deren Repräsentation als abstrakter Syntaxbaum. Zunächst werden als introspektiv gekennzeichnete Elemente der Klasse oder einer ihrer Superklassen gesucht. Diese introspektiven Sprachelemente werden anschließend analysiert und es wird ein Modell erstellt, das bestimmte Eigenschaften der analysierten Elemente widerspiegelt. Dabei muss stets gewährleistet sein, dass die analysierten Elemente dem dazugehörigen introspektiven Programmiermodell entsprechen.

Die Analyse von gefundenen introspektiven Sprachkonstrukten besteht demzufolge aus den folgenden Bestandteilen:

- Abgleich der Konstrukte mit dem introspektiven Programmiermodell,
- Extraktion der relevanten Daten und Aufbau des Modells.

So soll beispielsweise die folgende Implementierung der introspektiven Value-Methode zur Konstruktion eines Modellelements vom Typ `ValueMethod` mit dem Wert `"SVCL"` führen (siehe Abschnitt 5.2.2):

```
public String getCode() {
  return "SVCL";
}
```

Das Programmiermodell wurde auf Seite 87 durch das Nichtterminal ***ValueMethod_M1*** spezifiziert, welches im Methodenrumpf lediglich ein Return-Statement zulässt.

Die Realisierungsidee der introspektiven Analyse besteht in der Spezifikation des introspektiven Programmiermodells durch Schablonen (engl.: Templates) und dem Vergleich mit dem zu analysierenden Sprachkonstrukt. Eine Schablone für die im Beispiel vorliegende Value-Methode ist folgendermaßen:

```
StringOrPrimitive yyy1() {
  return null;
}
```

Die Methode `yyy1` fungiert als Schablone für den zu analysierenden Quelltext. Ein Vergleich zwischen Schablone und Quelltext lässt sich durch Objekte vom Typ `org.eclipse.jdt.core.dom.ASTMatcher` realisieren. Dieser Mechanismus wird als Teil von Eclipse-JDT zur Verfügung gestellt und stellt eine Variante des Entwurfsmusters *Visitor* dar (siehe [Ga95]). Zwei Knoten eines AST werden durch die Methode `subtreeMatch` verglichen (siehe Abbildung 57). Das Resultat des Methodenaufrufs trifft eine Aussage über die Gleichheit der Knoten bezüglich des verwendeten konkreten Matchers.

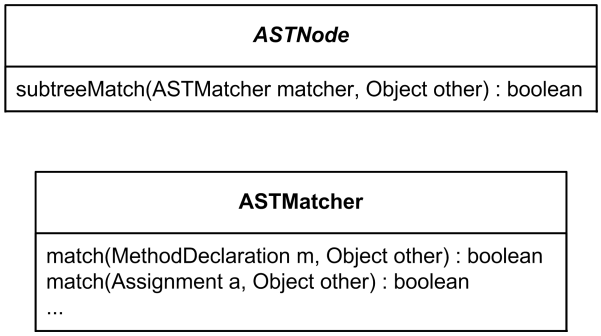


Abbildung 57: Vergleich von abstrakten Syntaxbäumen mittels `ASTMatcher`

Die durch die Klasse `ASTMatcher` zur Verfügung gestellte Implementierung prüft auf vollständige Isomorphie der verglichenen Teilbäume. Dieses Verhalten lässt sich durch Subklas-

senbildung anpassen. So wird im Beispielfall eine Subklasse von `ASTMatcher` realisiert, die den Vergleich zwischen den Typen `StringOrPrimitive` des Templates mit allen primitiven Typen und dem Typ `String` positiv bescheidet. Ebenfalls wird durch den Methodennamen `yyy1` ausgedrückt, dass die Vergleichsmethode einen beliebigen Namen haben darf.

Des Weiteren ist es im Verlauf des Vergleichs möglich, bestimmte Elemente zu speichern und somit relevante Konstrukte zu extrahieren. So extrahiert der Matcher zur Analyse der beispielhaft vorgestellten Value-Methode den Rückgabewert folgendermaßen:

```
public class ValueMatcher_M1 extends ASTMatcher {
  public Expression expression;

  public boolean match(NullLiteral node, Object o) {
    if (o instanceof Expression) {
      expression = (Expression) o;
      return true;
    }
    return false;
  }...
```

Die Umsetzung der skizzierten Realisierungsidee erfolgt durch Objekte vom Typ `MatchingAnalyzer`. Ein konkreter `MatchingAnalyzer` stellt sowohl eine Schablone, als auch einen Matcher für Vergleich und Extraktion zur Verfügung. Der Vergleich erfolgt durch die Methode `matchAndAnalyze`, welche zurückliefert ob der übergebene Teilbaum dem entsprechenden Programmiermodell entspricht. Ist dies der Fall wurde die Analyse bereits teilweise durchgeführt und die relevanten Informationen sind als Attribute des konkreten verwendeten Matchers verfügbar. In der Einschubmethode `createModel` werden diese Informationen zum Erzeugen von Modellelementen verwendet. In einigen Fällen ist eine weitere Analyse erforderlich, die durch weitere Analyzer erfolgt. Dies geschieht in der Einschubmethode `createFollowUps`. Ist die Analyse nicht erfolgreich, so wird ein spezielles Modellelement erzeugt, welches den für den Abbruch verantwortlichen Teil des AST referenziert. Dadurch ist eine Lokalisation des Fehlers möglich.

In Abbildung 58 ist die Klasse `Analyzer` in einem UML-Klassendiagramm dargestellt. Die Methode `matchAndAnalyze` ist folgendermaßen implementiert:

```
public final boolean matchAndAnalyze(ASTNode node) {
  if (getTemplate().subtreeMatch(getMatcher(), node)) {
    setNode(node);
    try {
      createModel();
      createFollowUps();
    } catch (Exception e) {
      new ErrorAnnotationAnalyzer(
        e, getNode(), getTreeElement()).analyze();
    }
    return true;
  } else {return false;}
}
```

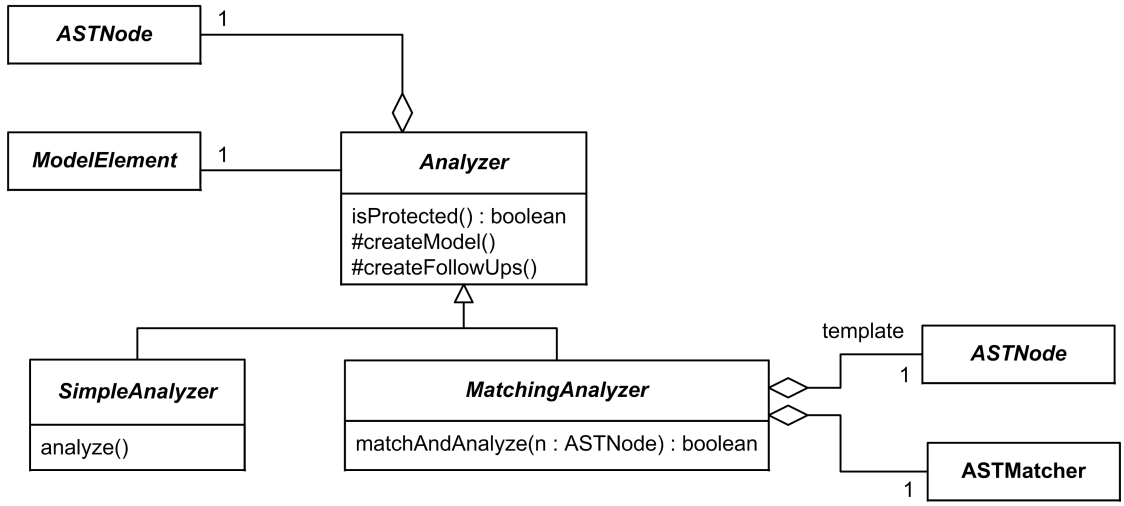


Abbildung 58: Klassendiagramm: `Analyzer`

In einigen Fällen ist es einfacher, die introspektive Analyse „von Hand“ durchzuführen. In diesem Fall können Subklassen vom Typ `SimpleAnalyzer` erzeugt werden, in denen die Methoden `createModel` und `createFollowUps` direkt auf dem dazugehörigen abstrakten Syntaxbaum arbeiten. Ein Beispiel hierfür sind die Analyzer, die zu Beginn eine Klasse nach introspektiven Methoden absuchen. In diesem Fall ist die Verwendung einer Schablone nicht geeignet.

6.3 Sichten

Mit den bisher besprochenen Bestandteilen des IMF ist das Erzeugen eines Modells, das Eigenschaften von introspektiven Konstrukten widerspiegelt, möglich. In diesem Abschnitt wird die Umsetzung der Visualisierung des Modells besprochen.

Das Framework IMF unterstützt zwei Visualisierungsarten. Einerseits ist es möglich, Modelle in einer Baumstruktur zu visualisieren, andererseits wird eine graphische Visualisierung unterstützt. Eine wichtige Anforderung besteht darin, ein und dasselbe Modell auf verschiedene Art

und Weise zu visualisieren. Dies wird durch Objekte vom Type `View` unterstützt. Wie in Abbildung 59 dargestellt, ist einem Modell ein Viewpoint zugeordnet, welcher seinerseits mehrere Views verwaltet. Zu jedem Viewpoint gibt es genau einen voreingestellten View, der standardmäßig verwendet wird. Es werden zwei Arten von Views unterstützt: einerseits die Visualisierung in einer Baumstruktur und andererseits eine graphische Visualisierung. Eine dritte Art ermöglicht die gleichzeitige baumartige und graphische Visualisierung. Im Abschnitt 6.3.2 wird die Umsetzung der baumartigen Visualisierung vorgestellt. Anschließend wird im Abschnitt 6.3.3 die graphische Darstellung vorgestellt.

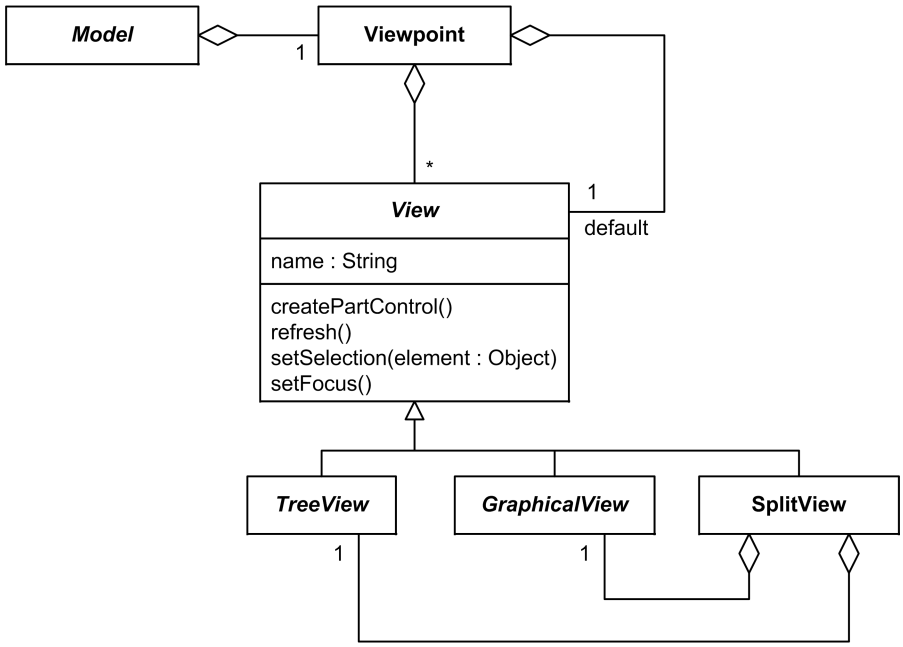


Abbildung 59: Klassendiagramm: `Viewpoint`*,* `View`

6.3.1 Bearbeiten durch Sichten

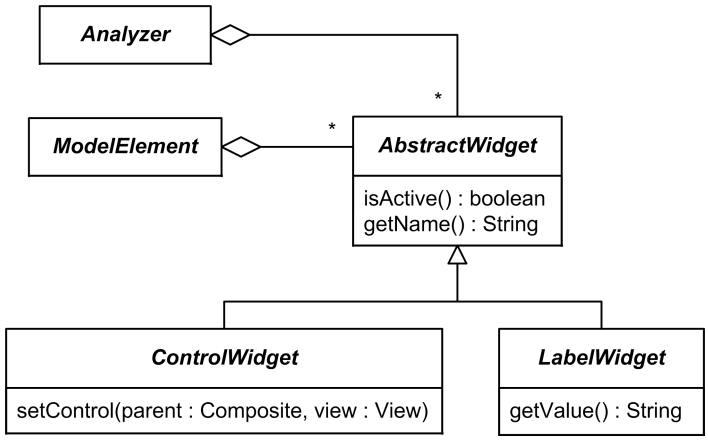


Abbildung 60: Klassendiagramm: `AbstractWidget`

Wie in Abbildung 53 auf Seite 111 dargestellt, soll eine Visualisierung direkte Veränderungen des Modells beziehungsweise des analysierten Quelltextes ermöglichen. Eine naive Umsetzung würde die Verantwortlichkeit zum Manipulieren der konkreten Visualisierung überlassen. Typische Bearbeitungsszenarien sind das Verändern von Werten, sowie das Hinzufügen und Löschen von Elementen. Dies sind generische Bearbeitungsszenarien, die unabhängig von der konkreten Visualisierung sind und somit mehrfach redundant realisiert würden. Als Alternative werden alle Manipulationsmöglichkeiten visualisierungsneutral direkt an die zu manipulierenden Elemente geknüpft. Die zu manipulierenden Elemente sind in diesem Fall Modellelemente und Analyzer, welche ihrerseits direkt für ein Element des abstrakten Syntaxbaums stehen. Konkret werden Manipulationen durch Objekte vom Typ `ControlWidget` realisiert (siehe Abbildung 60). Durch Objekte vom Typ `LabelWidget` werden können Analyzer und Modellelemente Informationen über sich zur Verfügung stellen. Die Verantwortlichkeit der konkreten Visualisierung beschränkt sich darauf, die angebotenen Informationen und Manipulationsmöglichkeiten in geeigneter Art und Weise aufzubereiten und anzubieten.

6.3.2 Baumansicht

Die von IMF unterstützte Visualisierung von Modellen in einer Baumstruktur besteht aus zwei Bereichen. Im linken Bereich werden durch eine Baumstruktur Sichten des Modellbaums visualisiert. Dieser Bereich wird im folgenden *Baumansicht* genannt. Im rechten Bereich befindet sich die *Elementanzeige*, in der die Eigenschaften des aktuell ausgewählten Modellelements dargestellt (siehe Abbildung 61).

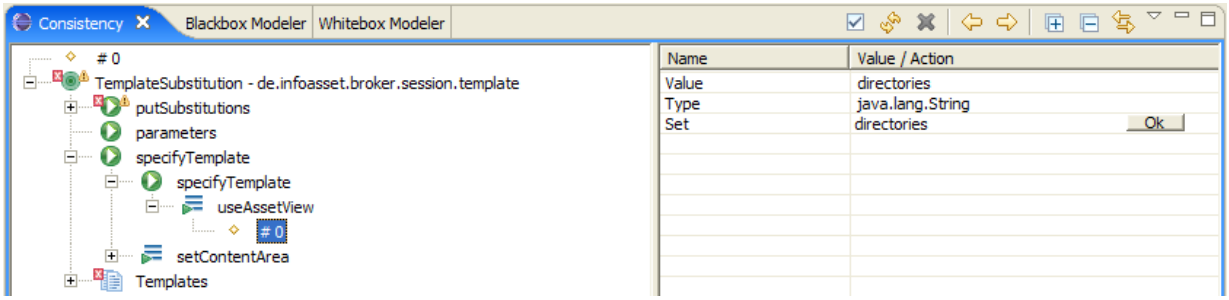


Abbildung 61: Visualisierung durch Baumansicht und Elementanzeige

Konkrete Baumansichten sollen sich in der Auswahl der darzustellenden Modellelemente, den verwendeten Icons zur Kennzeichnung der Elemente, den Texten mit denen die Elemente beschriftet werden, sowie evtl. verwendeten Dekorationsicons unterscheiden können. Eine Anpassung der Elementanzeige soll durch Selektion der angezeigten Eigenschaften und Manipulationsmöglichkeiten möglich sein.

Zur technischen Realisierung dieser Forderungen, stellt IMF ein kleines Whitebox Framework zur Verfügung, welches durch Subklassenbildung an konkrete Visualisierungsanforderungen angepasst werden kann. Ausgangspunkt für Visualisierung durch eine Baustruktur ist die Klasse `TreeView` (siehe Klassendiagramm in Abbildung 62). Diese Klasse aggregiert jeweils eine Instanz der Klasse `TreeContentProvider`, `TreeLabelProvider`, `TreeLabelDecorator` und `TableContentProvider`, die die aufgeführten Aspekte der Anpassbarkeit abdecken. Um eine angepasste Sicht zu erzeugen, müssen also jeweils Subklassen implementiert werden, die das gewünschte Verhalten aufweisen.

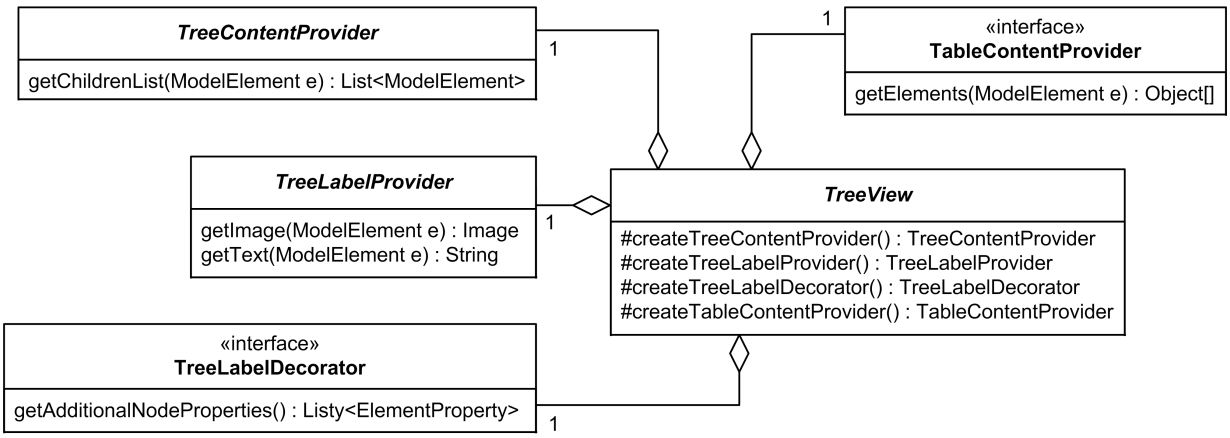


Abbildung 62: `TreeView` zur Visualisierung einer Baumstruktur

6.3.3 Graphische Visualisierung

Die Unterstützung einer graphischen Visualisierung in IMF basiert auf dem *Graphical Editor Framework* (GEF) von Eclipse [Ec06c]. GEF unterstützt das Erstellen graphischer Editoren, indem es eine *Model-View-Controller* (MVC) Architektur realisiert [KP88, Bu96]. Dabei wird

die Funktionalität einer interaktiven graphischen Benutzeroberfläche durch drei Komponenten realisiert. Das Model repräsentiert die jeweils aktuelle logische Sicht der darzustellenden Daten. Diese Daten werden durch den View visualisiert. Für die Interaktion mit dem Benutzer ist der Controller verantwortlich. Als Folge einer Interaktion muss der Controller das Model aktualisieren. Daran anschließend aktualisiert sich der View entsprechend dem veränderten Model.

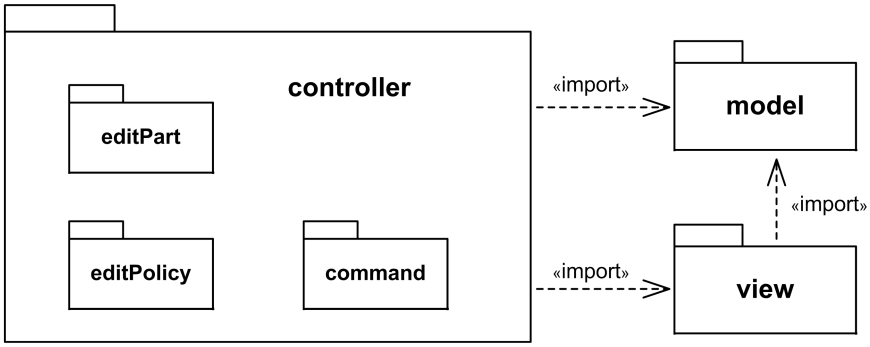


Abbildung 63: Paketdiagramm für GEF

In der Realisierung von GEF wird die Controller-Komponente in drei weitere Subkomponenten unterteilt. Als zentrale Komponente fungieren so genannte *EditParts*. Deren Verantwortlichkeit besteht im Erzeugen des Views basierend auf dem aktuellen Model. Des Weiteren erzeugen EditParts *EditPolicies*, die als Reaktion auf Aktionen des Benutzers *Commands* erzeugen. Die Aufgabe der Commands besteht im Verändern des Modells und anschließendem Aktualisieren des Views. Alle vorgestellten Komponenten und ihre Abhängigkeiten sind in einem Paketdiagramm in Abbildung 63 dargestellt.

Die Realisierung eines konkreten graphischen Editors mit GEF erfolgt also durch Implementieren der vorgestellten Komponenten. Wie bereits in Abbildung 59 dargestellt, wird ein graphischer Editor in IMF durch ein Objekt der Klasse `GraphicalView` repräsentiert. Dabei handelt es sich um eine abstrakte Klasse, die im Zuge der Anpassung des Frameworks vom Framework-Nutzer implementiert werden muss. Konkret müssen Implementierungen der Interfaces `EditPartFactory` und `ModelFactory` bereitgestellt werden (siehe Abbildung 64). Durch Objekte vom Typ `ModelFactory` wird, basierend auf einem Introspektionsmodell, ein Model im Sinne des Graphical Editing Framework bereitgestellt. Objekte vom Typ `EditPartFactory` erzeugen in Abhängigkeit von einem konkreten Model-Objekt EditParts. Diese erzeugen Views und EditPolicies, welche wiederum Commands erzeugen.

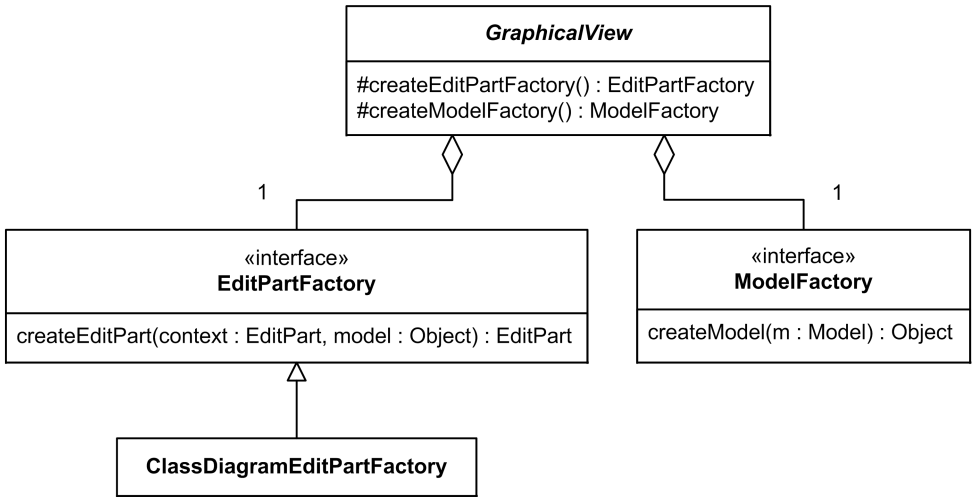


Abbildung 64: Klassendiagramm: `GraphicalView`

Im Vergleich zum Erstellen einer maßgeschneiderten baumartigen Ansicht, ist das Erstellen einer graphischen Ansicht mit deutlich größerem Aufwand verbunden. Dies ist auf die der Problemdomäne inhärente Variabilität und Komplexität zurückzuführen. Die bisher vorgestellte Unterstützung beschränkt sich auf die Möglichkeit des Einbindens eines beliebigen zu erstellenden GEF-Editors. Es wird im Unterschied zur baumartigen Visualisierung kein Rahmen vorgegeben, in dem dann nur noch Anpassungen vorzunehmen sind.

Die Syntax von UML-Klassendiagrammen lässt sich leicht auch jenseits der Visualisierung von Klassen einer objektorientierten Programmiersprache und deren Beziehungen nutzen. Dem Rechnung tragend existiert eine Implementierung des Interfaces `EditPartFactory` als Teil von IMF, die eine graphische Visualisierung durch UML-Klassendiagramme realisiert.

6.4 Realisierung des Konfigurators mit IMF

In diesem Abschnitt werden die Anpassungen vorgestellt, mit denen IMF zur Introspektion von Blackbox Frameworks anpasst wurde. Das Ergebnis dieser Anpassungen ist der bereits in Abschnitt 4.3 vorgestellte Konfigurator. Die Anpassungen umfassen die drei vorgestellten Bereiche der Modellelemente, der Analyzer und der Visualisierung.

Die zur Introspektion von Blackbox Frameworks erforderlichen Modellelemente korrespondieren mit dem in Abschnitt 4.2 vorgestellten Metametamodell. Die Erzeugung dieser Modellelemente wird jeweils durch Analyzer realisiert. Dabei handelt es sich um Klassen vom Typ `SimpleAnalyzer`.

Die wichtigste Visualisierung des Konfigurator wurde unter Verwendung des `TreeView` realisiert. Um eine bessere Übersichtlichkeit der Modellvisualisierung zu erreichen, versteckt die angepasste Sicht bestimmte Elemente des Modells. Zudem werden die im Metametamodell verwendeten Icons zur Kennzeichnung der Elemente benutzt.

Im Verlauf der Anpassung eines introspektiven Blackbox Frameworks werden die durch Introspektion gewonnenen Modellelemente konfiguriert und miteinander in Beziehung gesetzt. Dabei werden die Modellelemente manipuliert. Realisiert ist dies, indem jedes Modellelement jeweils alle möglichen Manipulationen durch `ControlWidget`s spezifiziert. In der Visualisierung werden lediglich alle vorhandenen Widgets angezeigt.

Die Kennzeichnung von „vollständig konfigurierten“ Modellelementen wird durch eine Klasse vom Typ `TreeLabelDecorator` realisiert.

7 Toro – Eine introspektive Webplattform

Um Nutzen und Kosten des Ansatzes der introspektiven modellgetriebenen Softwareentwicklung evaluieren zu können, wurde im Rahmen dieser Arbeit ein umfangreiches Softwaresystem nach diesem Prinzip gestaltet. Als Basis wurde das in [We02] beschriebene Portalsystem für das Wissensmanagement gewählt. Dabei handelt es sich um eine integrierte Webanwendung, die über ein eigenständig entwickeltes Persistenz- und Webvisualisierungs-Framework verfügt. Diese existierenden Frameworks wurden jeweils in introspektive Frameworks umgewandelt, welche in diesem Kapitel detailliert vorgestellt werden. Dabei hat sich herauskristallisiert, dass eine Trennung von fachlicher Umsetzung des existierenden Wissensmanagementsystems und entstandener introspektiver Plattform sinnvoll und wünschenswert ist. Diese Plattform wird im Folgenden *Toro* genannt.

In Abschnitt 7.1 werden zunächst die dem Entwurf von Toro zugrunde liegenden Prinzipien dargelegt. Aufgrund der introspektiven Realisierung lassen sich auf Basis von Toro entwickelte Anwendungen durch verschiedene in Abschnitt 7.2 vorgestellte Modellperspektiven veranschaulichen und modellieren. In den folgenden Abschnitten werden drei introspektive Whitebox Frameworks vorgestellt, die jeweils eine Modellperspektive realisieren. Ein einfaches Framework zur Realisierung von multilingualen Nachrichten wird in Abschnitt 7.3 vorgestellt. In Abschnitt 7.4 wird ein introspektives Persistenzframework vorgestellt. In Abschnitt 7.5 wird ein introspektives Framework zur Webvisualisierung vorgestellt.

In Abschnitt 7.6 wird ein Überblick über die Entwicklung von Webanwendungen mit Toro gegeben. Zu Schluss dieses Kapitels gibt Abschnitt 7.7 eine Bewertung der vorgestellten Plattform anhand von kommerziell eingesetzten Anwendungen sowie anhand eines nicht-geplanten Wiederverwendungsszenarios, in dem die erreichte Integration sich positiv auswirkt.

7.1 Entwurfsprinzipien von Toro

Die entstandene introspektive Plattform Toro soll die effiziente Entwicklung und Pflege von Webanwendungen unterstützen. Hierbei sind zwei Szenarien von besonderem Interesse. Ein erstes unterstütztes Szenario beinhaltet die effiziente Neuentwicklung einer Anwendung. In diesem Fall werden ausgehend von Kundenwünschen Benutzeroberflächen, ein entsprechendes Datenmodell und Geschäftslogik neu entwickelt. Eine derartige Neuentwicklung basiert direkt auf Toro als Plattform und profitiert von den introspektiven Technologien.

In einem zweiten Szenario soll es mit geringem Aufwand möglich sein, Familien von ähnlichen Anwendungen zu erstellen und zu pflegen. Beispielsweise soll das in [We02] beschriebene existierende Wissensmanagementsystem effizient an abweichende Kundenanforderungen angepasst werden können. Diese Abweichungen können beispielsweise das Layout der Benutzeroberfläche, das Datenmodell der Anwendung oder das Rechtemanagement betreffen. Die

vom Kernsystem abgeleiteten Anwendungen sollen mit überschaubarem Aufwand gepflegt werden können und von Weiterentwicklungen des Kernsystems profitieren.

Zur Realisierung dieser Anforderungen werden *Projekte* als Modularisierungseinheiten verwendet. Ein Projekt besteht aus Artefakten, die bestimmte Funktionalitäten realisieren. Dabei handelt es sich beispielsweise um Java-Quelldateien, abgespeicherte Konfigurationsmodelle, HTML-Templates oder JavaScript-Dateien. Ein Grundprinzip bei der Entwicklung von Toro ist die Einhaltung des Prinzips „Convention over Configuration“ [TH05]. Dies wird unter Anderem dadurch umgesetzt, die Artefakte in jedem Projekt in einer bestimmten vorgegebenen Struktur anzuordnen. Dies erleichtert den gedanklichen Aufwand bei der Neuerstellung von Projekten und ermöglicht die direkte Nutzung von Konventionen.

Die Kernkonzepte der Plattform sind in einem Projekt realisiert, welches von anderen Projekten erweitert werden kann. Unter Benutzung der Erweiterungsbeziehung werden neue Artefakte hinzugefügt. Soll ein Projekt – wie im ersten Szenario beschrieben – neu entwickelt werden, wird ein Projekt erstellt, welches das Kernprojekt von Toro direkt erweitert. Die erstellten Artefakte werden entsprechend den Konventionen von Toro im Projekt angeordnet,

Zur Unterstützung des zweiten Szenarios ist eine andere Art der Beziehung zwischen Projekten notwendig. Soll ein bestehendes Projekt angepasst werden, ist eine einfache additive Erweiterungsbeziehung nicht ausreichend. Soll beispielsweise das Datenmodell angepasst werden, was dann Auswirkungen auf das Rechtemanagement hat, ist eine Anpassung existierender Artefakte notwendig. Eine Möglichkeit hierbei wäre, eine Kopie des anzupassenden Projekts anzulegen und diese dann anzupassen. In diesem einfachen Fall existiert keine explizite Beziehung zwischen Original und angepasstem Projekt. Werden nach Anfertigung der Kopie Änderungen im originalen Projekt durchgeführt (z.B.: Fehlerbeseitigungen), so propagieren sich diese Änderungen nicht automatisch in das kopierte Projekt. Eine simultane Weiterentwicklung beider Projekte ist mit großem manuellem Aufwand verbunden. Zu diesem Zweck ermöglichen Versionskontrollwerkzeuge die Erzeugung von so genannten *Verzweigungen* (engl.: Branch) eines Projekts [Le00]. Für den Entwickler stellt sich ein verzweigtes Projekt wie eine Kopie des Originals dar, allerdings existiert eine explizite Verbindung zwischen den originalen und verzweigten Artefakten. Aufgrund dieser expliziten Verbindung ist eine werkzeuggestützte Integration von Änderungen am Original-Projekt möglich. Dies ermöglicht die effiziente simultane Entwicklung und Pflege von Familien ähnlicher Systeme.

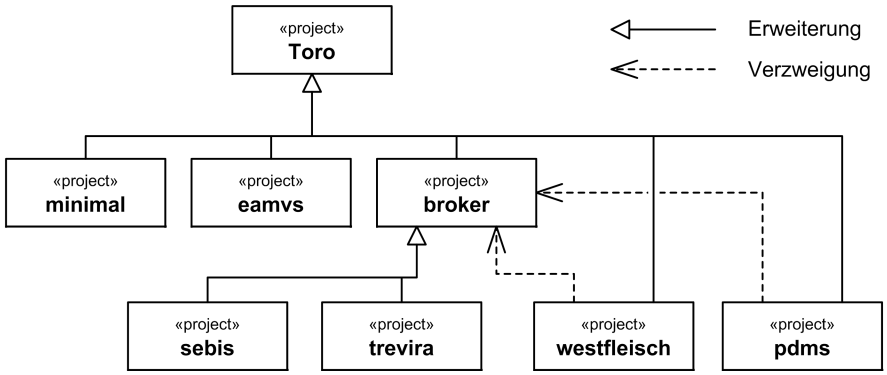


Abbildung 65: Toro-Projekte und ihre Beziehungen

In Abbildung 65 sind existierende Projekte und ihre Beziehungen beispielhaft graphisch dargestellt. Die Projekte `minimal`, `eamvs` und `broker` sind Entwicklungen, die direkt auf der Plattform aufsetzen. Die Projekte `sebis` und `trevira` sind einfache Erweiterungen des Projekt `broker`. Im Gegensatz dazu sind die Projekte `westfleisch` und `pdms` durch Verzweigung aus dem Projekt `broker` entstanden.

Wie bereits aufgeführt, handelt es sich bei Toro um eine *introspektive* Plattform. Konkret bedeutet dies einerseits, dass jede auf Toro basierende Anwendung als Ganzes ein introspektives Blackbox Framework ist. Dies ermöglicht die effiziente integrierte Konfiguration von Toro-Anwendungen. Andererseits besteht ein komplexes betriebliches Informationssystem im Allgemeinen aus mehreren Frameworks, die gemeinsam die gewünschte Funktionalität erbringen. Teile dieser zu integrierenden Frameworks lassen sich in einem deklarativen Programmierstil anpassen, andere Teile erfordern einen imperativen Programmierstil (siehe Abschnitt 2.2.4). Das Zusammenspiel und die Integration aller beteiligten Frameworks sind essenziell. Diese Anforderungen lassen sich gut mit introspektiven Whitebox Frameworks umsetzen. Das Erstellen einer auf Toro basierenden Anwendung erfolgt also durch das Anpassen von introspektiven Whitebox Frameworks. Die durch Introspektion gewonnenen Modelle erhöhen die Verständlichkeit und Übersichtlichkeit der Anwendungen und ermöglichen Anpassungen auf einem hohen Abstraktionsniveau.

7.2 Modellperspektiven einer Webanwendung

Toro ermöglicht die Entwicklung von Webanwendungen unter Benutzung von introspektiven Modellen, welche die Anwendung auf einem hohen Abstraktionsniveau repräsentieren. Es gibt eine Reihe von Ansätzen zur modellgetriebenen Entwicklung von Webanwendungen, welche geeignete domänenspezifische Sprachen vorschlagen [Co99, CFB00, Ce03, BG04]. Dabei handelt es sich um generative top-down Ansätze, die Anwendungscode auf der Basis der erstellten Modelle generieren (siehe Abschnitt 5.6).

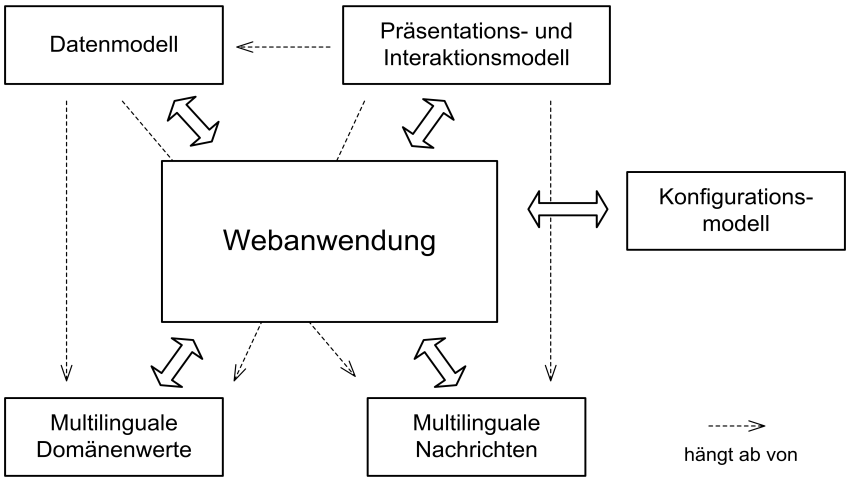


Abbildung 66: Introspektive Modellperspektiven in Toro

Die im Rahmen dieser Arbeit identifizierten und introspektiv realisierten Modellperspektiven sind in Abbildung 66 dargestellt. Die Modelle im unteren Teil der Abbildung beschreiben einfache multilinguale Übersetzungen von Meldungen des Systems sowie von vordefinierten Domänenwerten. Im oberen Teil befindet sich das Datenmodell, welches die von der Anwendung verwalteten Geschäftsobjekte beinhaltet, sowie das Präsentations- und Interaktionsmodell, welches die Benutzeroberfläche definiert. Diese Modelle werden durch in den folgenden Abschnitten detailliert beschriebene introspektive Whitebox Frameworks realisiert. Es existieren Abhängigkeiten zwischen diesen Modellen, welche ebenfalls in Abbildung 66 dargestellt sind.

Das Konfigurationsmodell nimmt eine Sonderposition ein. Keines der anderen Modelle hängt vom Konfigurationsmodell ab. Des Weiteren ist das Erstellen eines Konfigurationsmodells nicht spezifisch für Webanwendungen, sondern repräsentiert einen Querschnittsaspekt, der typisch für eine große Klasse von Systemen ist.

7.3 Eine introspektive Realisierung multilingualer Nachrichten

Eine typische Anforderung an moderne Softwaresysteme besteht darin, dass es einfach möglich sein soll, sie in verschiedenen Regionen einzusetzen. Dies umfasst Dinge wie unterschiedliche Sprachen, Maßeinheiten, Datumsangaben. Der Vorgang der Erstellung von Software, die mit wenig Aufwand an verschiedene Regionen angepasst werden kann wird *Internationalisierung* genannt [CD01]. Die Anpassung an konkrete Regionen wird *Lokalisierung* genannt.

Die Erstellung von internationalisierter Software erfordert das Einführen von Indirektionen an Stellen, die regionsabhängige Aspekte betreffen. Im Folgenden wird der einfache Aspekt multilingualer Nachrichten betrachtet. An bestimmten Stellen im Programmcode werden Nachrichten produziert, die dem Benutzer des Systems angezeigt werden. So soll bei einer falschen Passworteingabe beispielsweise eine sinnvolle Fehlermeldung ausgegeben werden. In einem nicht internationalisierten System ist der textuelle Inhalt dieser Fehlermeldung direkt im Quellcode enthalten:

```
out.println("Das eingegebene Passwort war falsch.");
```

Eine Anpassung an verschiedene Sprachen ist in diesem Fall nicht einfach möglich, da der auszugebende Text fest eincodiert ist und nicht dynamisch ausgewählt werden kann. Ein weiteres Problem besteht darin, dass alle zu übersetzenden Texte über das gesamte System verstreut sind. Wie bereits erwähnt, besteht die Lösung dieses Problems darin, eine Indirektion einzuführen, die den Text von der Nachricht trennt. Alle Texte sollen an einer zentralen Stelle zugänglich und somit einfach übersetzbar sein.

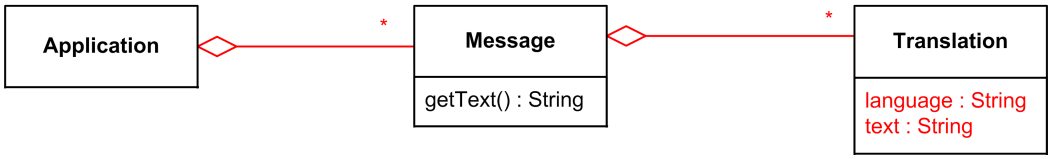


Abbildung 67: Konzeptuelles Klassendiagramm multilingualer Nachrichten

Ein konzeptuelles Klassendiagramm multilingualer Nachrichten ist in Abbildung 67 dargestellt. In dieser Abbildung sind bereits die Erweiterungspunkte farbig gekennzeichnet. Eine Anwendung beinhaltet viele multilinguale Nachrichten. Eine multilinguale Nachricht verwaltet Übersetzungen, welche durch ihre Sprache und den jeweiligen Text gekennzeichnet sind. Die aktuelle Übersetzung einer Nachricht kann durch die Methode `getText` abgefragt werden.

Der traditionelle Weg, Java-Programme zu internationalisieren besteht darin, Nachrichten durch einen Textschlüssel zu identifizieren und durch so genannte *Resource Bundles* zu lokalisieren. Ein Resource Bundle speichert Übersetzungen für eine bestimmte Region in einem einfachen Textformat. Der Zugriff erfolgt durch die in Abbildung 68 ausschnitthaft in einem Klassendiagramm dargestellte Klasse `ResourceBundle`. Der Zugriff auf die Nachricht erfolgt also beispielsweise folgendermaßen:

```
out.println(bundle.getString("wrong_login");
```

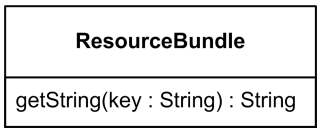


Abbildung 68: Klassendiagramm: `ResocureBundle`

Die Lokalisierung erfolgt durch Textdateien im Format:

```
key = value
```

Die Beispielnachricht wird für die deutsche Sprache also beispielsweise in einer Datei namens `messages_de_DE.properties` folgendermaßen lokalisiert:

```
wrong_login = Das eingegebene Passwort war falsch.
```

Dadurch ist es möglich, Übersetzungen für verschiedene Regionen an einer zentralen Stelle durchzuführen. Aufgrund des einfachen Formats, in dem die Übersetzungen gespeichert wer-

den, ist dies auch für Nicht-Programmierer einfach möglich. Es handelt sich hierbei um eine einfache domänenspezifische Sprache, die das in Abbildung 67 dargestellte konzeptuelle Klassendiagramm multilingualer Nachrichten realisiert. Aufgrund der externen Speicherung der Übersetzungen handelt es um eine externe DSL mit den damit verbundenen Problemen (siehe Abschnitt 2.3.10). So ist die Konsistenz zwischen Programmcode und den übersetzten Nachrichten nicht automatisch überprüfbar. Hierbei sind zwei Arten von Inkonsistenzen denkbar: Einerseits kann es übersetzte Nachrichten geben, die nicht mehr benutzt werden. Dies verursacht unnötige Mehrarbeit bei zusätzlichen Lokalisierungen und führt langfristig zu einem aufgeblähten System. Die zweite Art von Inkonsistenz besteht in Nachrichten, die vom Programmcode genutzt werden, für die aber keine Übersetzungen existieren. Dabei handelt es sich um einen Fehler. Beide Inkonsistenzszenarien sind nicht automatisch auffindbar, die Konsistenz muss händisch aufrechterhalten werden.

Im Rahmen dieser Arbeit wurde eine introspektive Realisierung multilingualer Nachrichten entwickelt, die die beschriebenen Konsistenzprobleme adressiert. Dabei handelt es sich um ein introspektives Whitebox Framework, welches das in Abbildung 67 dargestellte Klassendiagramm realisiert. Die Grundidee besteht darin, multilinguale Nachrichten und ihre Übersetzungen explizit wie folgt zu spezifizieren:

```
class Messages extends GenericMessages {
  public static final Message wrongLogin = new Message() {
    String de = "Das eingegebene Passwort war falsch.";
    String en = "The submitted password is wrong.";
  };
  ...
```

Pro multilinguale Nachricht wird ein öffentliches, statisch zugreifbares Objekt deklariert, welches Übersetzungen als Felder definiert. Der Zugriff auf multilinguale Nachrichten erfolgt dann folgendermaßen:

```
out.println(Messages.wrongLogin.getText());
```

Die Nachricht wird typisiert benutzt. Beide vorgestellten Inkonsistenzszenarien sind leicht automatisch überprüfbar. Pro Nachricht lässt sich leicht mit Mitteln einer Post-IntelliJ-IDE überprüfen, wie oft sie benutzt wird. Der Zugriff auf eine nicht definierte Nachricht wird sofort vom Compiler beanstandet. Da die Nachrichten direkt als Programmiersprachkonstrukte spezifiziert werden, sind sie mit dem Programmcode der sie benutzt, integriert. Es handelt sich hierbei um eine interne domänenspezifische Sprache. Problematisch an der bisher vorgeschlagenen Lösung ist jedoch, dass das Lokalisieren von Nachrichten für Nicht-Programmierer erheblich schwieriger geworden ist. Die Syntax der internen domänenspezifischen Sprache ist hierfür zu komplex, da eine Teilmenge von Java verwendet wurde. Dies ist der Punkt, an dem durch Introspektion eine einfachere konkrete Syntax der DSL zur Verfügung gestellt wird.

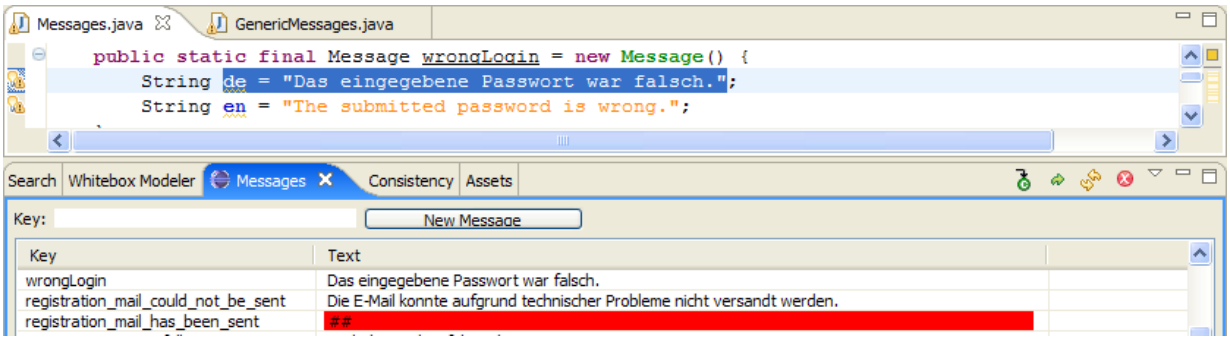


Abbildung 69: Introspektive Lokalisierungsansicht multilingualer Nachrichten

In Abbildung 69 ist die zur Lokalisierung bereitgestellte introspektive Bearbeitungsansicht dargestellt. Die hier sichtbare konkrete Syntax ist auf einem hohen Abstraktionsniveau und ist auch für Nicht-Programmierer verständlich. Die dritte Nachricht in Abbildung 69 ist nicht übersetzt, was durch die rote Einfärbung dargestellt wird. Das Lokalisieren ist durch direkte Manipulation möglich, was zum transparenten Erzeugen eines neuen Java-Felds führt. Es werden einerseits die Vorteile einer internen DSL in Bezug auf Integration realisiert, andererseits abstrahiert die konkrete für den Benutzer sichtbare Syntax von Details der darunter liegenden Programmiersprache.

Die vorgestellte Realisierung benutzt das introspektive Programmiermodell der ObjectByType-Methode (siehe Abschnitt 5.2.3). Die Analyse der Übersetzungen erfolgt durch einen maßgeschneiderten Analyzer (siehe Abschnitt 6.2) und wird durch einen maßgeschneiderten Viewer visualisiert.

7.4 Ein introspektives Persistenzframework

In diesem Abschnitt wird ein introspektives Whitebox Framework vorgestellt, welches die persistente Repräsentation von Geschäftsobjekten ermöglicht.

7.4.1 Persistenzframeworks

Moderne betriebliche Informationssysteme sind häufig als dreischichtige Client-Server-Architektur aufgebaut, wobei die Aufgabe der untersten Schicht in der persistenten Speicherung der vom System verwalteten Geschäftsobjekte besteht [Ec95, Bu96]. Mit *persistenter* Speicherung ist eine dauerhafte Datenhaltung gemeint, die die verwalteten Daten über die Lebenszeit von Betriebssystem-Prozessen hinaus speichert. Die in diesem Kontext am weitesten verbreitete Technologie zur persistenten Speicherung sind relationale Datenbanken, die Daten in Form von benannten Relationen speichern [Co70, KE06]. Zum Zugriff auf relationale Datenbanken existiert die standardisierte Anfragesprache *SQL* (*Structured Query Language*) [Da87].

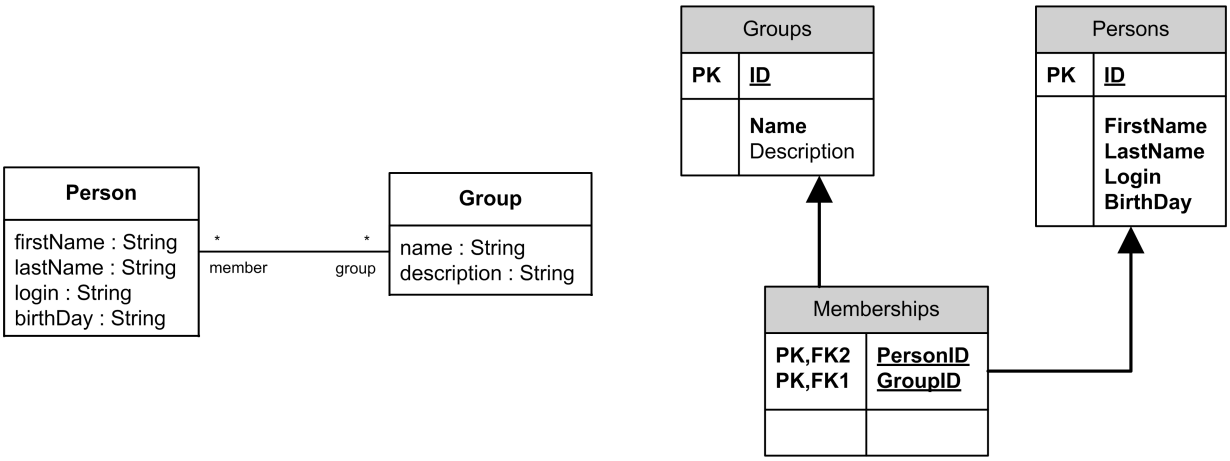


Abbildung 70: Beispiel einer objektrelationalen Abbildung

Es gibt eine Reihe von Gründen, relationale Datenbanken aus den über der Datenschicht befindlichen Teilen eines betrieblichen Informationssystems nicht direkt durch SQL-Anfragen zuzugreifen, sondern ein so genanntes *Persistenzframework* zum Zugriff zu benutzen. Als erstes implementieren die existierenden Datenbanksysteme leicht unterschiedliche Dialekte von SQL, so dass bei direkter Benutzung eine sehr enge Koppelung zwischen Anwendungscode und tatsächlich verwendetem Datenbanksystem besteht. Ein zweites Argument besteht in der großen intellektuellen Distanz zwischen dem relationalen Modell und dem zur Implementierung der Anwendung verwendeten Programmiermodell. Typischerweise werden Informationssysteme in einer objektorientierten Programmiersprache entwickelt. Ein Persistenzframework führt also ein so genanntes o*bjekt-relationales Mapping* [Fu97] durch. Dabei werden Objekte einer objektorientierten Programmiersprache möglichst transparent auf Tupel einer relationalen Datenbank abgebildet. In Abbildung 70 ist beispielhaft ein objekt-relationales Mapping zweier Klassen, die durch eine Beziehung der Kardinalität many-to-many verbunden sind, dargestellt.

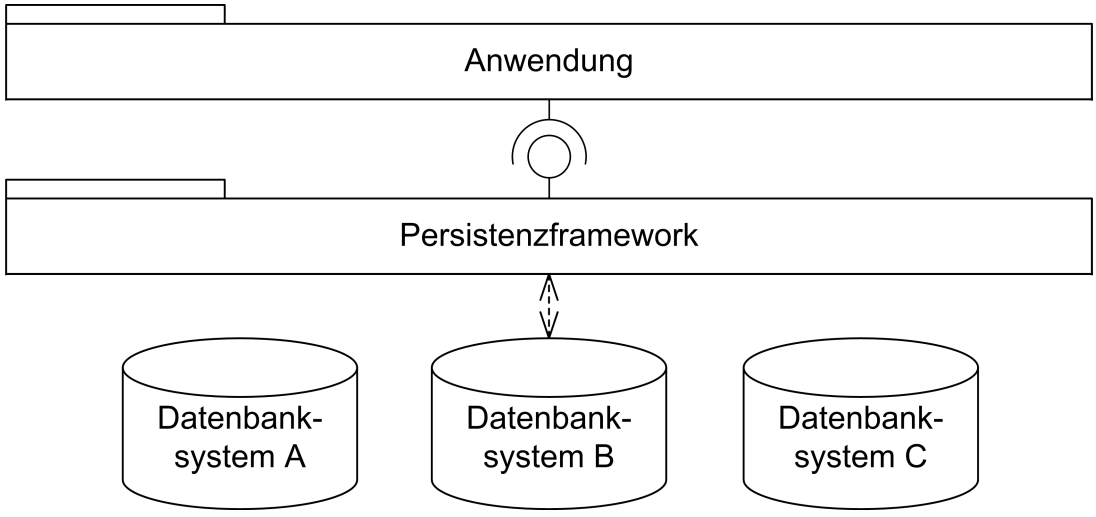


Abbildung 71: Persistenzframework als Schicht eines Informationssystems

Durch die Verwendung eines Persistenzframeworks soll es dem Anwendungsentwickler mög-

lich sein, Geschäftsobjekte möglichst transparent persistent zu verwalten. Die konsequente Anwendung eines Persistenzframeworks führt wie bereits dargestellt, zu einer in Abbildung 71 dargestellten Schichtenarchitektur. Alle persistenten Operationen werden mit Hilfe des Persistenzframeworks bewerkstelligt. Dadurch ist es im Persistenzframework möglich, Unterschiede zwischen Datenbanksystemen zu berücksichtigen. Der Anwendungscode ist somit vom konkreten Datenbanksystem unabhängig.

Unsere Realisierung eines Persistenzframeworks trennt die Umsetzung der beiden vorgestellten Hauptaufgaben voneinander (siehe Abbildung 72). Eine erste Schicht realisiert eine Basisabstraktion, die einen vom konkret verwendeten Datenbanksystem unabhängigen Zugriff auf Relationen ermöglicht. Die Kernabstraktionen dieser Schicht entsprechen konzeptuell denen einer relationalen Datenbank. Hauptaufgabe dieser Basis-Persistenzabstraktion besteht in einem herstellerneutralen Zugriff auf relationale Daten. Die Abbildung auf objektorientierte Konzepte und die Repräsentation von Geschäftsobjekten findet in einer zweiten Schicht statt, deren Realisierung auf den Ideen der introspektiven modellgetriebenen Softwareentwicklung basiert.

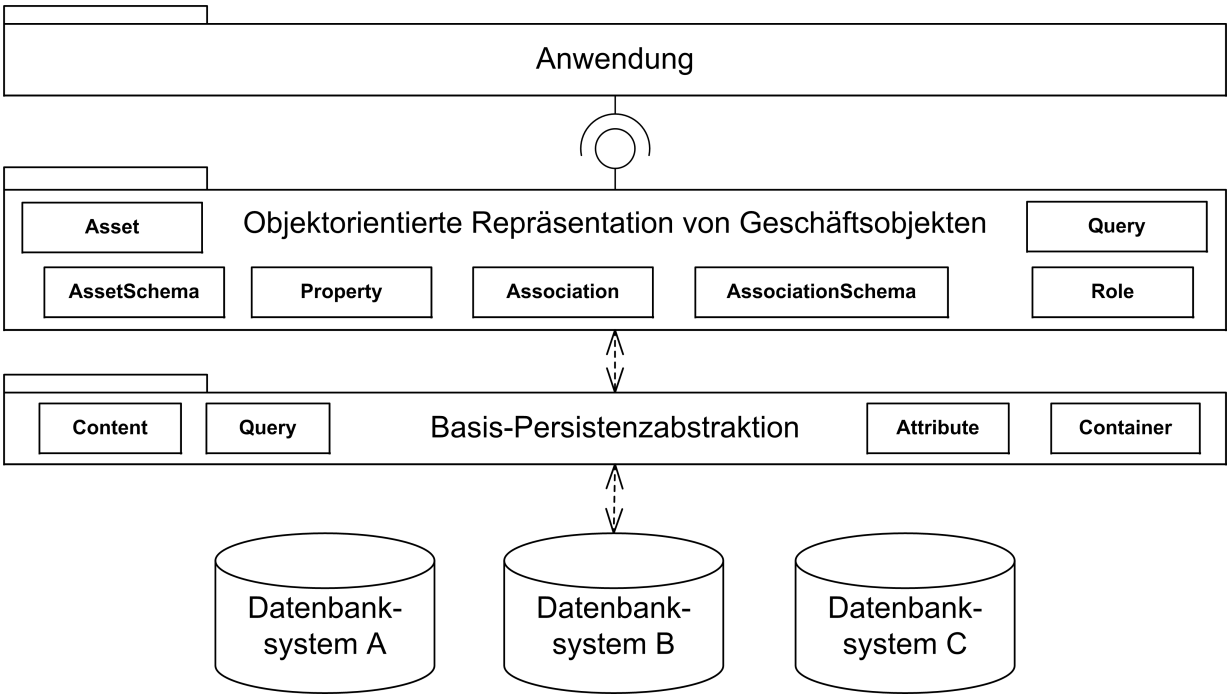


Abbildung 72: Trennung von Basisabstraktion und objektorientierter Sicht

7.4.2 Basis-Persistenzabstraktion

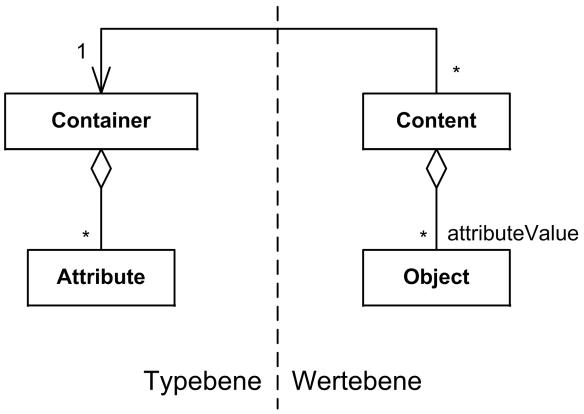


Abbildung 73: Metamodell der Basis-Persistenzabstraktion

Große Teile der grundlegenden Basis-Persistenzabstraktion wurden bereits als Teil von [We02] entwickelt und beschrieben. Das Metamodell der Kernkonzepte dieser Schicht ist in Abbildung 73 dargestellt. `Container` und `Content` sind objektorientierte Repräsentierungen von relationalen Tabellen und deren Zeilen. Die Spalten einer Tabelle werden durch Objekte vom Typ `Attribute` repräsentiert. Durch Methoden von `Container` können Inhalte erstellt, gelöscht und gefunden werden. Die Klasse `Content` ermöglicht typisierten Zugriff auf die Werte der Attribute.

Entsprechend den in relationalen Datenbanken verbreiteten Attributtypen existieren verschiedene Subtypen von `Attribute`, wie in Abbildung 74 dargestellt. Interessant sind hierbei vor allem der Attributtyp `ReferencingAttribute`, welcher einen Fremdschlüssel repräsentiert, sowie der Attributtyp `IdAttribute`, welcher den Primärschlüssel einer Relation repräsentiert.

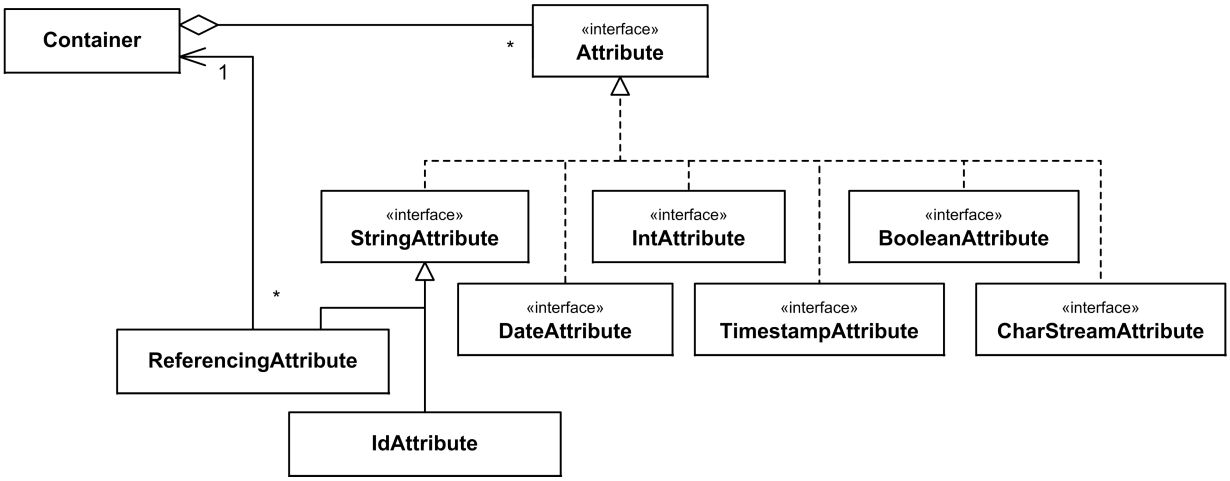


Abbildung 74: Unterstützte Attributtypen als Subtypen von `Attribute`

7.4.3 Objektorientierte Repräsentation von Geschäftsobjekten

Die bisher vorgestellte Basis-Persistenzabstraktion stellt eine API zur Verfügung, die einen herstellerneutralen Zugriff auf relationale Daten ermöglicht. Das eigentliche objekt-relationale Mapping wird von einer zweiten Schicht bewerkstelligt (siehe Abbildung 72). Dem Entwurf dieser Abbildung liegen zwei Entscheidungen zugrunde. Eine grundsätzliche Entscheidung bestand darin, Vererbungsbeziehungen zwischen Typen von Geschäftsobjekten nicht zu unterstützen. Des Weiteren unterstützt das Framework ausschließlich binäre Assoziationsbeziehungen. Diese Entscheidungen wurden getroffen, um den Rahmen dieser Arbeit einhalten zu können.

Eine weitere grundlegende Entscheidung betrifft die generelle Beziehung zwischen den aufeinander abzubildenden Elementen. Genauer stellt sich die Frage, welche Seite bei der Entwicklung im Vordergrund steht, und somit „führend“ ist. Im allgemeinen Fall können sowohl die objektorientierten Klassen als auch die relationalen Tabellen unabhängig voneinander erstellt und anschließend miteinander verbunden werden. Dies ist jedoch nur in einigen Fällen wünschenswert, da in diesem Fall Redundanz entsteht. Im Fall des vorgestellten Persistenzframeworks ist stets die objektorientierte Seite „führend“. Die Abbildung auf relationale Tabellen erfolgt auf Basis der definierten Klassen nach einem feststehenden Schema.

Die Kernkonzepte des Persistenzframeworks sind in Abbildung 75 dargestellt. Dabei wird zwischen Wert- und Typebene unterschieden. Auf Wertebene werden Geschäftsobjekte durch die Klasse `Asset` repräsentiert. Assets haben Eigenschaften und stehen mit anderen Assets in Beziehung. Diese Beziehungen werden durch Objekte vom Typ `Association` repräsentiert.

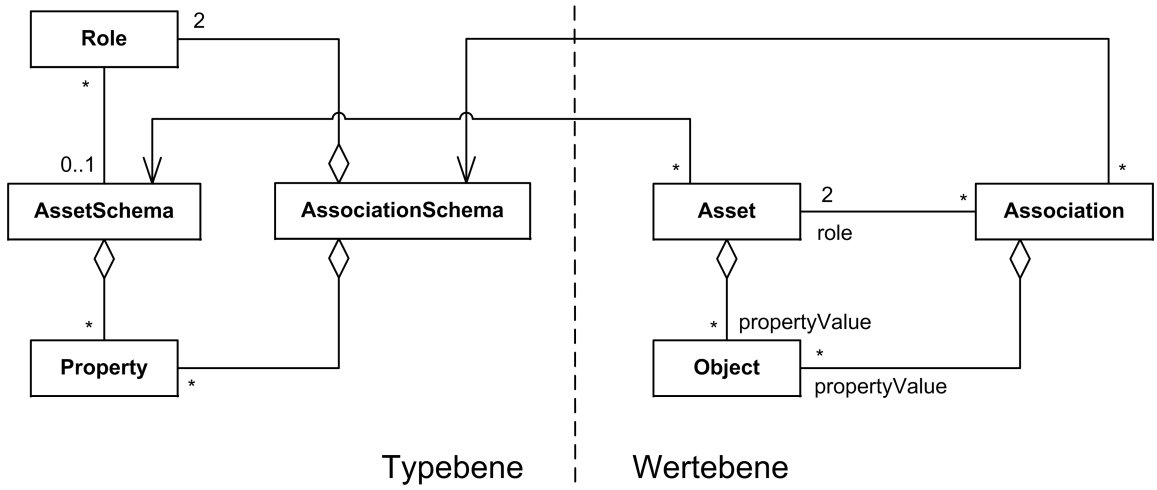


Abbildung 75: Kernkonzepte des Persistenzframeworks

Auf der linken Seite der Abbildung ist das Metamodell des Persistenzframeworks dargestellt. Typen von Geschäftsobjekten werden durch die Klasse `AssetSchema` repräsentiert. Durch Objekte vom Typ `Property` werden die elementaren Attribute des Assettyps repräsentiert. Die Spezifikation möglicher Beziehungen erfolgt durch Objekte vom Typ `Association-Schema`. Ein Assoziationstyp hat zwei Assoziationsenden, welche durch die Klasse `Role` repräsentiert werden. Es gibt zwei Arten von Assoziationsenden. Einerseits ist der Typ des an der

Assoziation teilnehmenden Assets bekannt. Andererseits kann am Ende der Assoziation ein Asset von beliebigem Typ teilnehmen.

Durch Instanzen des auf der linken Seite von Abbildung 75 dargestellten Metamodells werden Typen von Geschäftsobjekten modelliert. Die grundlegende Idee der introspektiven Modellierung im Kontext des Persistenzframeworks besteht in der expliziten introspektiven Instanziierung des Metamodells. Ein konkretes Datenmodell wird durch statisch definierte Instanzen der Metamodellklassen spezifiziert. Durch introspektive Analyse des Quellcodes ist es möglich, das Datenmodell auf einem hohen Abstraktionsniveau darzustellen und zu manipulieren. Andererseits wird die explizite Typdefinition zum typisierten Zugriff genutzt. Eine introspektive Repräsentation der in Abbildung 70 dargestellten Assettypen `Person` und `Group` ist folgendermaßen:

```
public class Person extends Asset {
  final public static StringProperty firstName =
    new StringProperty() {};
  final public static StringProperty lastName =
    new StringProperty() {};
  final public static StringProperty login =
    new StringProperty() {};
  final public static DateProperty birthDay =
    new DateProperty() {};

  final public static ManyRole<Group> groups =
    new ManyRole<Group>() {
      protected Role otherRole() {
        return Group.members;
      }
  };
}

public class Group extends Asset {
  final public static ManyRole<Person>
    members = new ManyRole<Person>() {
      protected Role otherRole() {
        return Person.groups;
      }
  };
  ...
```

Durch Introspektion ist es möglich, das Modell zu gewinnen und anschließend verschiedenartig zu visualisieren und zu bearbeiten. In Abbildung 76 ist eine Visualisierung als Baumansicht dargestellt.

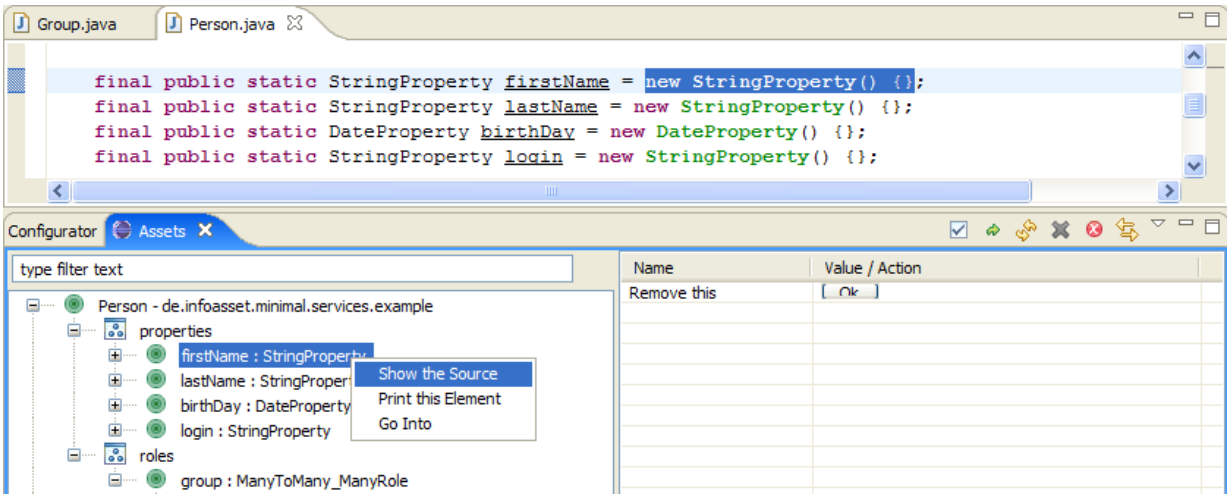


Abbildung 76: Introspektionsmodell in Baumansicht

Eine graphische Visualisierung in der Notation von UML-Klassendiagrammen ist in Abbildung 77 zu sehen. Einerseits kann also aufgrund der introspektiven Modellierung das Modell der Geschäftsobjekttypen und deren Beziehungen extrahiert werden. Andererseits werden die Metamodellinstanzen zum typisierten Zugriff auf die Geschäftsobjekte genutzt. Der Zugriff auf die Eigenschaft `firstName` erfolgt beispielsweise folgendermaßen:

```
firstName = person.getString(Person.firstName);
```

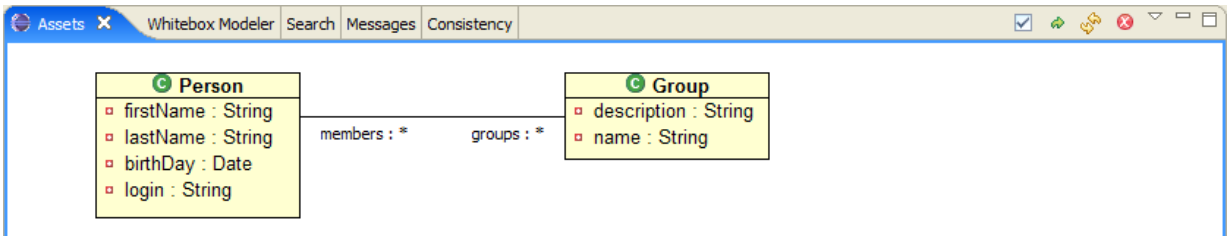


Abbildung 77: Introspektionsmodell in Klassendiagramm-Notation

Auf die gleiche Art und Weise wird eine Eigenschaft gesetzt:

```
person.setString(Person.firstName, "otto");
```

Die explizit definierten Metamodellinstanzen werden in diesem Programmiermodell zur Identifikation der zugegriffenen Eigenschaften genutzt (siehe Klassendiagramm in Abbildung 78).

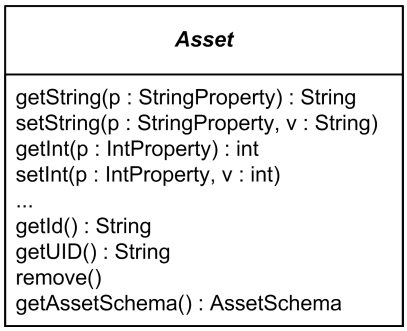


Abbildung 78: Klassendiagramm: `Asset`

In Abbildung 79 sind alle unterstützten Eigenschaftstypen dargestellt.

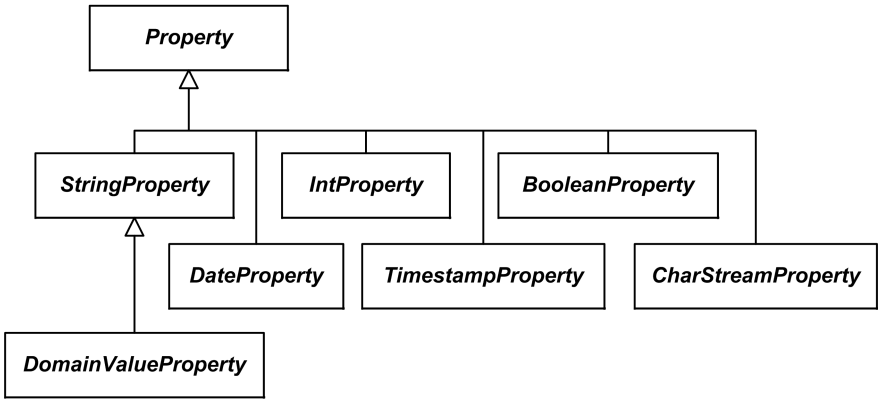


Abbildung 79: Klassenhierarchie: `Property`

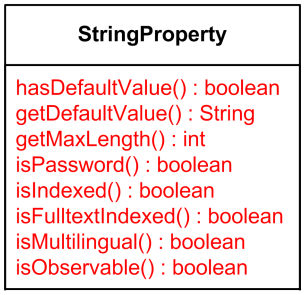


Abbildung 80: Introspektive Parametrisierung von `StringProperty`

Die Eigenschaften eines Geschäftsobjekts verfügen zusätzlich zu ihrem Namen und Typ über weitere für die Persistenzabbildung wichtige Parameter. Diese werden in unserem Persistenzframework introspektiv modelliert. In Abbildung 80 sind alle Parametrisierungsmöglichkeiten des Eigenschaftstyps `StringProperty` dargestellt. Eine Parametrisierung der Eigenschaft `firstName` des eingeführten Assettyps `Person` ist folgendermaßen möglich:

```
public class Person extends Asset {
  final public static StringProperty firstName =
    new StringProperty() {
      public boolean isFulltextIndexed() {
        return true;
      }

      public int getMaxLength() {
        return 1024;
      }
  }
  ...
```

Aufgrund des introspektiven Programmiermodells ist es möglich, die Parametrisierung im introspektiven Modellierungswerkzeug vorzunehmen (siehe Abbildung 81). Hier sind alle vorhandenen Parametrisierungsmöglichkeiten übersichtlich dargestellt.

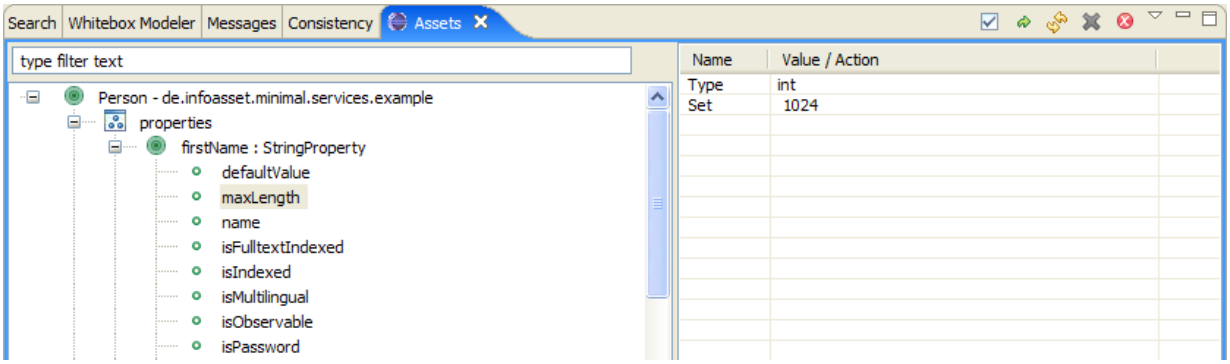


Abbildung 81: Introspektive Modellierungsansicht

Analog zum Zugriff auf Eigenschaften, erfolgt das Navigieren von Beziehungen ebenfalls unter Benutzung der statisch definierten Metamodellinstanzen. Beziehungen werden durch introspektive Instanziierung der Assoziationsenden definiert. Diese werden durch Objekte vom Typ `Role` repräsentiert.

Wie in Abbildung 82 dargestellt, gibt es zwei Arten von Assoziationsenden, die sich bezüglich der Kardinalität und Navigationsmöglichkeiten unterscheiden. Beispielsweise lassen sich alle Gruppen, in denen eine Person Mitglied ist, folgendermaßen auffinden:

```
Iterator<Group> groups =
  Person.groups.getAssociatedAssets(person);
```

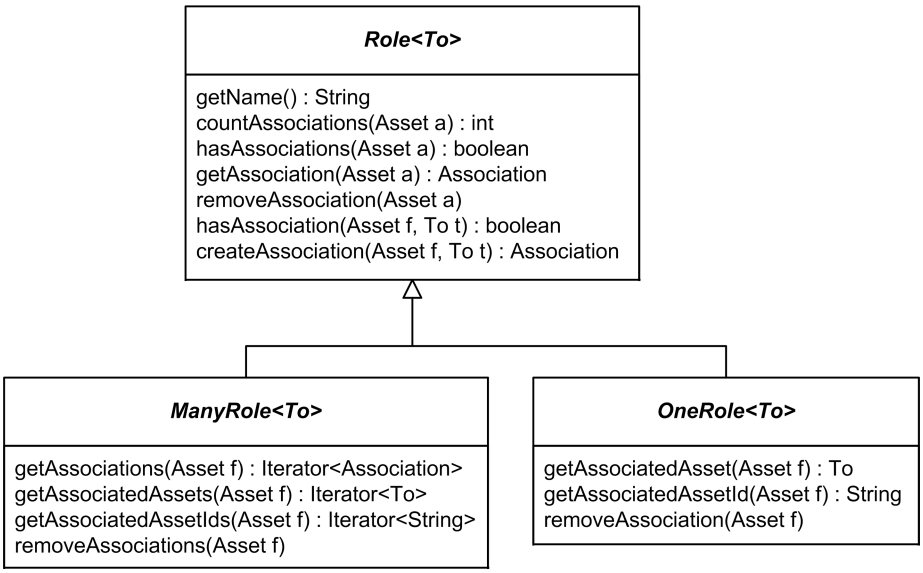


Abbildung 82: Klassenhierarchie: Role

Zusammengefasst wird ein Geschäftsobjekttyp durch Erstellen einer Subklasse der abstrakten Klasse `Asset` erzeugt. Entsprechend den gewünschten Eigenschaften des Typs werden statische finale Instanzen der Eigenschaftstypen erstellt. Beziehungen zwischen Assettypen werden durch Instanzen von Objekten vom Typ `Role` erstellt. Die erstellten Instanzen können nun zum Zugriff auf die Eigenschaften und zur Navigation der Beziehungen genutzt werden. Aufgrund des introspektiven Programmiermodells kann das Datenmodell der Geschäftsobjekte extrahiert und visualisiert werden.

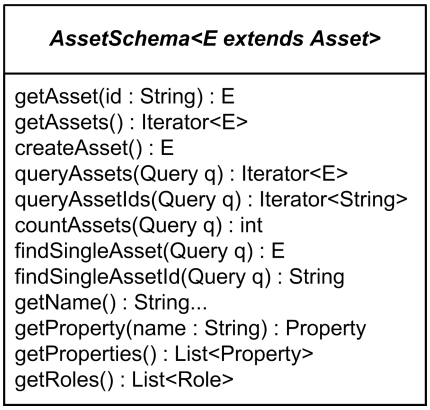


Abbildung 83: Klassendiagramm: `AssetSchema`

Alle bisher vorgestellten Zugriffsmechanismen bezogen sich auf einzelne, bereits bekannte Assets. Eine weitere wesentliche Anforderung an ein Persistenzframework ist die Suche nach Geschäftsobjekten. Diese Anforderung wird von der in Abbildung 83 dargestellten Klasse `AssetSchema` realisiert. Zur Laufzeit des Systems existiert genau eine Instanz von `AssetSchema` pro Assettyp. Eine Aufgabe von `AssetSchema` ist das Erzeugen von neuen Geschäftsobjekten durch die Methode `createAsset`. Jedes Asset hat eine innerhalb des Asset-

schemas eindeutige ID, welche durch die Methode `getId` abgefragt werden kann. Falls die ID eines Assets bekannt ist, kann es durch die Methode `getAsset` von `AssetSchema` zugegriffen werden. Alle Assets eines Assetschemas können durch die Methode `getAssets` abgerufen werden, welche einen typisierten Iterator zurückliefert.

Zur Suche nach Assets mit bestimmten Eigenschaften existiert die Methode `queryAssets`. Die Selektionskriterien werden durch ein Objekt vom Typ `AbstractQuery` übermittelt, welches einen Ableitungsbaum einer Anfrage repräsentiert. Die Klassenhierarchie in Abbildung 84 gibt einen Überblick über die syntaktischen Möglichkeiten, eine Anfrage zu stellen. Die einfachste Anfrage stellt durch Objekte vom Typ `QueryCondition` eine Bedingung an den Wert einer Eigenschaft eines Assets. Durch Objekte vom Typ `QueryOperator` können Anfragen logisch miteinander verknüpft werden. Eine Anfrage über mehrere miteinander durch Fremdschlüsselbeziehungen verknüpfte Relationen ist durch Objekte vom Typ `AssetJoin` möglich. Letztendlich kann die Vereinigung mehrerer Anfragen gebildet werden.

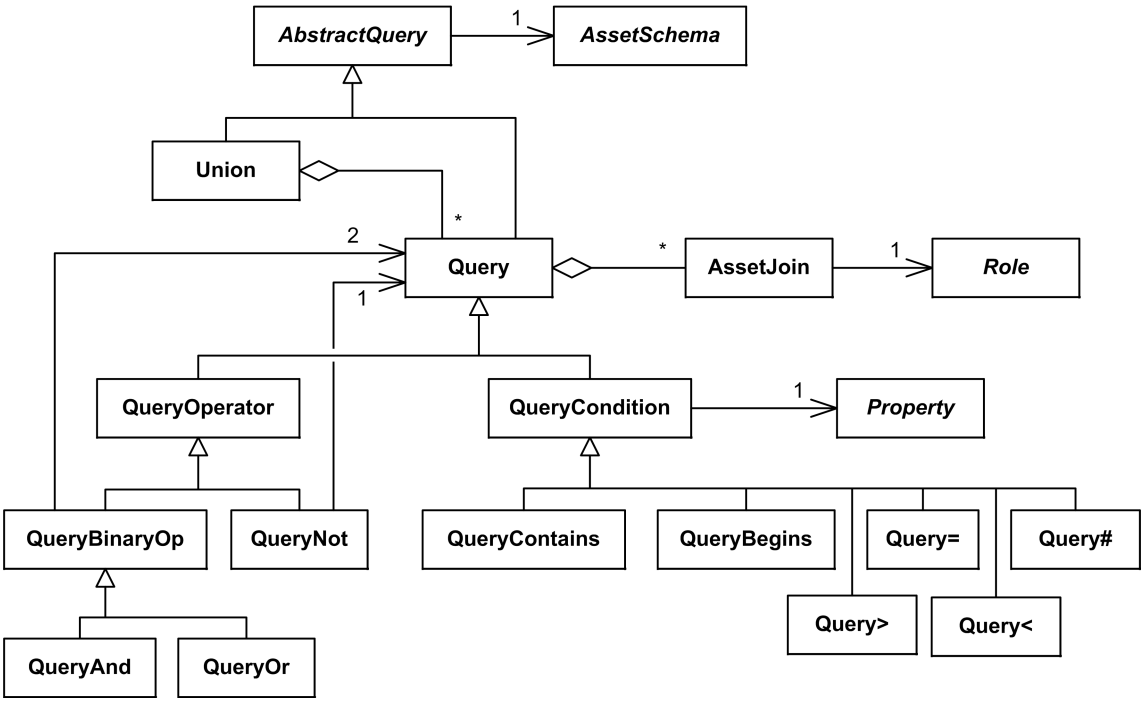


Abbildung 84: Typisierte Query-API

Beispielsweise lässt sich eine Suche nach allen Personen, die Mitglied in einer Gruppe mit Namen „Redakteure“ sind, folgendermaßen formulieren:

```
Query q = new QueryEquals(Group.name, "Redakteure");
q.addJoin(new AssetJoin(Group.members));
Iterator<Person> found = Person.SCHEMA().queryAssets(q );
```

Besonderer Wert sei hierbei darauf gelegt, dass die Suchanfrage typisiert formuliert wurde und direkt die verwendeten Eigenschaften und Assoziationen referenziert. Dies ist aufgrund des in-

trospektiven Programmiermodells möglich, durch welches direkt statische Instanzen des Metamodells erzeugt wurden.

Eine weitere Aufgabe der Klasse `AssetSchema` besteht darin, zur Laufzeit des Systems generischen Zugriff auf das Metamodell der Geschäftsobjekte zu ermöglichen. Von der Methode `getProperties` wird ein Iterator über alle Eigenschaften und von der Methode `getRoles` über alle Assoziationsenden eines Assettyps geliefert. Diese Möglichkeit ist eine Voraussetzung zur Entwicklung von Komponenten, die generische Dienste für beliebige Typen von Geschäftsobjekten anbieten.

7.4.4 Vergleich mit existierenden Persistenzframeworks

In diesem Abschnitt wird unsere introspektive Realisierung eines Persistenzframework dem momentan vorherrschenden POJO-Ansatz gegenübergestellt. POJO ist eine Abkürzung für *Plain Old Java Objects* und steht dafür, persistente Geschäftsobjekte durch einfache JavaBeans zu repräsentieren [Ri06]. In diesem Ansatz werden Eigenschaften von Geschäftsobjekten durch private Felder mit einer öffentlichen Getter- und Setter-Methode repräsentiert. Beziehungen werden durch mengenwertige Felder mit Getter- und Setter-Methoden repräsentiert. Dieser Ansatz wird beispielsweise von Hibernate [BK04a], JDO [JR03] und der neuen EJB-3-Spezifikation [DK06] verfolgt. Der beispielhaft eingeführte Geschäftsobjekttyp `Person` wird auf diese Art und Weise folgendermaßen repräsentiert:

```
public class Person {
  @Indexed
  @Column(length=1024)
  private String firstName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  @ManyToMany
  private Set<Group> groups;

  public Set<Group> getGroups() {
    return groups;
  }

  public void setGroups(Set<Group> groups) {
    this.groups = groups;
  };
}
```

Die Modellierung erfolgt in diesem Fall durch Konventionen (private Felder und öffentliche Getter- und Setter-Methoden) sowie durch Annotationen. In einem gewissen Sinne ist dies bereits eine Form der introspektiven Modellierung, da die Modellinformationen durch Java-Code ausgedrückt werden. Allerdings findet ein großer Teil der Modellierung durch maßgeschneiderte Annotationen statt.

Die Modellierungsansätze (introspektive Instanziierung des Metamodells und POJOs mit Annotationen) führen zu unterschiedlichen Eigenschaften der Frameworks. Diese Auswirkungen werden im Folgenden untersucht. Der Schwerpunkt liegt demzufolge nicht auf funktionalen Eigenschaften der Persistenzframeworks wie Performanz, Transaktionalität, Unterstützung von Datenbankherstellern, usw., sondern lediglich auf Aspekten die direkt von den unterschiedlichen Modellierungsarten betroffen sind.

Es lassen sich die folgenden Rollen unterscheiden, die von den vorgestellten unterschiedlichen Modellierungsansätzen betroffen sind:

- Geschäftsobjektmodellierer spezifizieren die zu persistierenden Geschäftsobjekttypen, deren Eigenschaften und Beziehungen.
- Anwendungsprogrammierer entwickeln Anwendungen, die persistente Geschäftsobjekte verwalten.
- Frameworkentwickler entwickeln Persistenzframeworks.

Jeder der folgenden Vergleichsdimensionen ist für einen Teil der aufgeführen Entwicklergruppen von Bedeutung.

Modellierung

Dieser Aspekt ist für Geschäftsobjektmodellierer von Bedeutung. Es lassen sich zwei Modellierungsaspekte unterscheiden. Einerseits werden die Geschäftsobjekttypen und deren Beziehungen modelliert. Andererseits muss die Abbildung auf die darunter liegende relationale Datenbank spezifiziert werden. POJO-basierte Ansätze verwenden Programmierkonventionen zur elementaren Spezifikation der Geschäftsobjekttypen. Dies ist eine intuitive Art der objektorientierten Modellierung, auch wenn dies eine große Menge Redundanz erfordert. So erfordert beispielsweise die aufgezeigte Spezifikation der Eigenschaft `firstName` die siebenfache Wiederholung des Namens der Eigenschaft. Dieses Problem wird durch moderne IDEs gelindert.

Für alle weitergehenden Modellierungsaspekte werden in einem POJO-basierten Ansatz Annotationen verwendet. Programmierung findet also zu einem großen Teil durch Annotationen statt. Das Wissen darüber, welche Annotationen an welchen Stellen sinnvoll und erlaubt sind, ist von großer Bedeutung. Es existiert kein generischer Sprachmechanismus, der bei der Beantwortung dieser Fragen hilfreich ist.

Der in dieser Arbeit vorgeschlagene Ansatz nutzt zur Modellierung durchgängig introspektive Metamodellinstanzen. Die Modellierungsmöglichkeiten sind durch die überschreibbaren Methoden der instanziierten Metamodellklassen offensichtlich. Dies ist ein generischer Mechanismus einer objektorientierten Programmiersprache. Diese Fragen werden zusätzlich durch die

vorhandene Werkzeugunterstützung beantwortet. Ein Nachteil des vorgeschlagenen Ansatzes besteht darin, dass die Geschäftsobjekttypen einen vorgegebenen Basistyp erweitern müssen.

Programmiermodell zum Zugriff auf die Geschäftsobjekte

Dieser Aspekt ist für die Rolle der Anwendungsprogrammierer relevant und betrifft den Zugriff auf die Eigenschaften und das Navigieren von Beziehungen von Geschäftsobjekten. Hierbei kann lesender und schreibender Zugriff unterschieden werden. POJO-basierte Ansätze verwenden hierzu öffentliche Getter- und Setter-Methoden. Dies ist ein in der objektorientierten Programmierung übliches und sehr natürliches Programmiermodell. Es ermöglicht den intuitiven und typsicheren Zugriff auf Eigenschaften und Beziehungen.

Das vorgeschlagene introspektive Persistenzframework realisiert den Zugriff unter Benutzung der introspektiven Metamodellinstanzen. Dies ist auf den ersten Blick weniger intuitiv, nach einer kurzen Eingewöhnungsphase stellt dies jedoch kein Problem mehr dar. Beziehungen können typsicher navigiert werden, da bei der Definition Java-Generics verwendet wurden. Bei der Definition von Eigenschaften wurde bewusst auf Generics verzichtet, um die Lesbarkeit zu erhöhen. Als Konsequenz ist für den Compiler eine statische Typüberprüfung nicht möglich. Theoretisch ist es also möglich, auf nicht existierende Eigenschaften zuzugreifen, ohne dass bereits zur Entwicklungszeit ein Fehler erscheint:

```
document.getString(Person.firstName);
```

Erfahrungsgemäß ist dies eine zu vernachlässigende Fehlerquelle. Von größerer Bedeutung ist die vorhandene Werkzeugunterstützung beim Erstellen von Zugriffscode, wie in Abbildung 85 dargestellt.

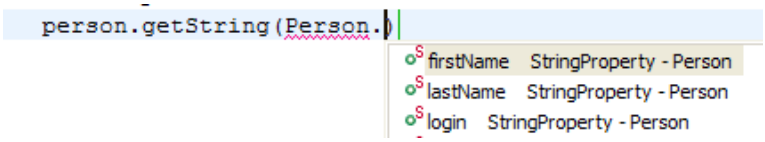


Abbildung 85: Code Assist

Ein weiterer interessanter Aspekt betrifft nur den Zugriff auf Beziehungen. Diese werden im POJO-basierten Ansatz durch Java-Collections und Referenzen definiert und zugegriffen. Dies ist zwar einerseits intuitiv, wirft aber in Bezug auf die Persistenzabbildung einige Fragen auf. So ist bei existierenden Realisierungen stets nur eine Seite der Beziehung aktiv, das heißt Änderungen auf dieser Seite werden in die Datenbank übernommen. Eine Veränderung auf der inaktiven Seite bleibt ohne Folgen. Dies ist eine gravierende Fehlerquelle für beidseitig navigierbare Beziehungen. Mit dem vorgeschlagenen introspektiven Ansatz kann dieser Aspekt transparent realisiert werden, so dass beide Seiten als aktiv erschienen.

Ein zweiter Punkt betrifft die Frage der Materialisierung von Beziehungen. Soll eine mengenwertige Beziehung über das POJO-Programmiermodell zugegriffen werden, muss die Collection vollständig materialisiert werden. Der vorgeschlagene Ansatz ist in diesem Punkt flexibler, da er keine zwingende Materialisierung erfordert. Sollen alle in einer Beziehung beteiligten

Elemente zugegriffen werden, so kann das beispielsweise durch einen `Iterator` geschehen, was keine vollständige Materialisierung erfordert.

Stellen von Anfragen

Ein wichtiger Aspekt eines Persistenzframeworks ist der Zugriff auf Objekte anhand bestimmter meist deklarativ formulierter Kriterien. Diese Anfragen werden im Allgemeinen in SQL-Anfragen an die darunter liegende Datenbank übersetzt und ausgeführt. Dieser Aspekt ist für Anwendungsprogrammierer relevant. Die formulierten Anfragen beziehen sich auf die modellierten Eigenschaften und Beziehungen der Geschäftsobjekte.

In POJO-basierten Ansätzen werden die Kriterien unter Benutzung des Namens von Eigenschaften und Beziehungen formuliert. Eine Anfrage nach allen Personen mit dem Vornamen `Thomas` kann beispielsweise folgendermaßen formuliert werden:

```
Criteria crit = session.createCriteria(Person.class);
crit.add(Expression.eq("firstName", "Thomas"));
List<Person> result = crit.list();
```

Die vollständig untypisierte Identifikation von Eigenschaften und Beziehungen anhand ihres Namens ist ein klarer Nachteil POJO-basierter Ansätze. Im in dieser Arbeit vorgeschlagenen Ansatz können wie auf Seite 141 dargestellt, Anfragen typsicher unter direkter Benutzung der Metamodell-Instanzen formuliert werden. Dies erhöht die Konsistenz und ermöglicht Werkzeugunterstützung bei der Formulierung von Anfragen.

Zugriff auf Metainformationen

Ein weiterer für den Anwendungsentwickler relevanter Aspekt ist der Zugriff auf Metainformationen über Geschäftsobjekte zur Laufzeit des Systems. Dies ermöglicht das Erstellen und Benutzen von Komponenten, die immer wiederkehrende Probleme generisch lösen, wie beispielsweise in [Pa02] beschrieben. Beispielsweise werden Passwörter als Strings repräsentiert, an die Darstellung und Repräsentation werden jedoch spezielle Anforderungen gestellt. In einem POJO-basierten Ansatz ist der Zugriff auf Metadaten nur durch Reflection möglich. Dies ist umständlich und nutzt keine domänenspezifischen Abstraktionen wie Eigenschaften oder Beziehungen. Im vorgeschlagenen Ansatz können die zum Zugriff benutzten introspektiven Metamodellinstanzen jederzeit direkt zugegriffen werden. Wie in Abbildung 80 dargestellt, verfügt die Klasse `StringProperty` über eine Methode `isPassword`, die sowohl zur Spezifikation als auch zum Zugriff zur Laufzeit genutzt wird.

Frameworkentwicklung und Entwicklung von Werkzeugunterstützung

Aufgrund der expliziten Modellierung der Problemdomäne ist die Entwicklung eines Persistenzframeworks unter Benutzung einer introspektiven Modellierungsmethode vergleichsweise einfach. Elementare Werkzeugunterstützung zur deklarativen Modellierung in Form einer domänenspezifischen Sprache ist aufgrund des introspektiven Programmiermodells sofort vor-

handen. Darauf aufbauen können weitere Werkzeuge wie beispielsweise die in Abbildung 77 dargestellte Visualisierung als UML-Klassendiagramm konsistent entwickelt werden.

7.5 Ein introspektives Framework zur Webvisualisierung

Die Präsentationsschicht eines betrieblichen Informationssystems ist für die Interaktion mit dem Benutzer des Systems zuständig. Moderne Informationssysteme sind größtenteils als Webanwendung realisiert. In diesem Abschnitt wird ein introspektives Framework vorgestellt, das die Entwicklung von Webanwendungen unterstützt.

7.5.1 Schematischer Ablauf

Die grundlegenden architekturellen Eigenschaften des World Wide Web wurden in [Fi00] beschrieben und sind für den großen Erfolg dieses Systems verantwortlich. Eine wesentliche Eigenschaft des World Wide Web stellt die zustandslose Kommunikation dar, die über das Request-Response basierte Protokoll HTTP [BFF96] realisiert wird. Eine weitere Eigenschaft ist die uniforme Identifikation und Repräsentation von Ressourcen. Zur Identifikation von Ressourcen werden URLs verwandt, zur Repräsentation wird im allgemeinen HTML benutzt.

Diese grundlegenden architekturellen Eigenschaften stellen den Rahmen dar, in dem der Entwurf von Webanwendungen stattfindet. In [Co99] wird eine Webanwendung als ein Websystem definiert, in dem der Zustand von Geschäftsobjekten durch Benutzerinteraktionen (Navigation und Dateneingaben) verändert wird. Die gesamte Interaktion zwischen einem Benutzer und einer Webanwendung besteht also – technisch betrachtet – aus dem Schicken von Requests und deren Beantwortung durch HTML-Dokumente. Die Aufgabe eines Webframeworks besteht darin, diesen Vorgang zu unterstützen.

In Abbildung 86 ist dieser Vorgang schematisch dargestellt. Als Startpunkt schickt der Webbrowser des Benutzers einen HTTP-Request an eine URL des Webservers der Webanwendung. Der Webserver identifiziert zunächst die Sitzung, zu der der Request gehört. Dieser Schritt ist aufgrund der bereits erwähnten Zustandslosigkeit des Übertragungsprotokolls notwendig. Da betriebliche Informationssysteme im Allgemeinen über ein Rechtesystem verfügen, welches die erlaubten Aktionen reglementiert, ist die Kenntnis des Benutzers essenziell.

Als nächstes identifiziert der Webserver in Schritt 3 anhand der URL des Requests die Komponente, die für die Beantwortung zuständig ist. In unserem introspektiven Webframework heißen diese Komponenten *Handler*. Ein Handler repräsentiert einen von außen zugreifbaren Dienst einer Webanwendung. Der für den jeweiligen Request verantwortliche Handler liest nun in Schritt 4 die von ihm benötigten Parameter aus dem Request aus. In einem nächsten Schritt wird geprüft, ob der aktuelle Benutzer für die auszuführende Aktion autorisiert ist. Ist diese Prüfung positiv ausgefallen, wird die eigentliche mit der Aktion verbundene Geschäftslogik ausgeführt. In einigen Fällen können sich als Ergebnis der Geschäftslogik unterschiedliche Möglichkeiten der Weiterverarbeitung ergeben. Eine Möglichkeit besteht darin, die Anfrage an einen weiteren Handler weiterzuleiten. Dies wird durch Schritt 7a visualisiert. Die zweite Möglichkeit – Schritt 7b – beinhaltet das Senden einer Antwort zum Benutzer. Die Antwort wird in

Form eines HTML-Dokuments repräsentiert, welches im Allgemeinen unter Benutzung von Vorlagen erstellt wird.

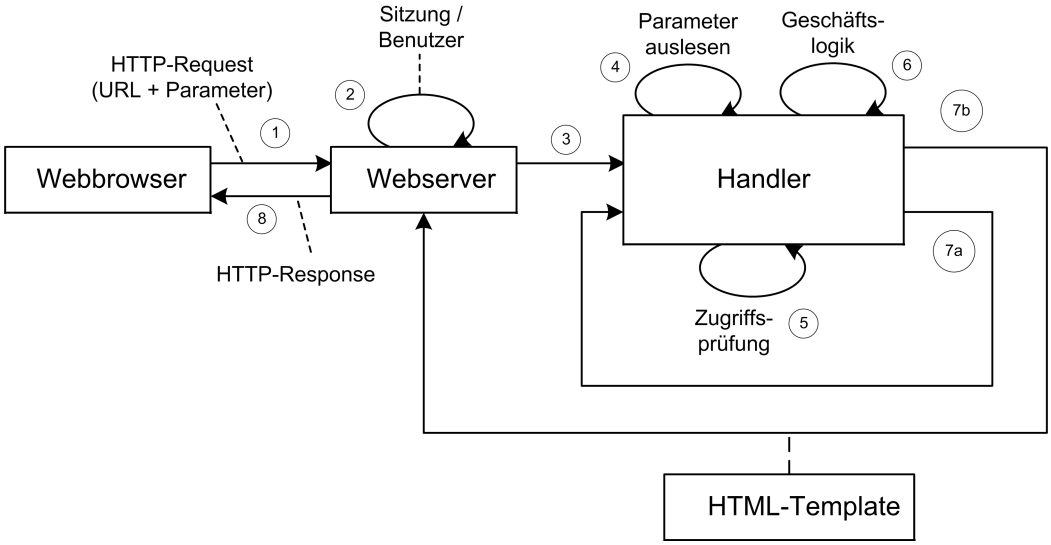


Abbildung 86: Schematischer Ablauf der Beantwortung eines HTTP-Requests

Schritte 1, 2 und 8 werden generisch vom Framework realisiert und werden im Folgenden nicht näher beschrieben.

7.5.2 Entwurfsziele und -entscheidungen

Gemäß den in Abschnitt 7.1 vorgestellten Entwurfsprinzipien besteht die Grundidee des vorgeschlagenen Webframeworks darin, den gesamten vorgestellten Ablauf durch ein introspektives Whitebox Framework zu realisieren. Die zentrale Abstraktion des Webframeworks sind die bereits angesprochenen Handler, welche durch introspektive Java-Klassen repräsentiert werden. Alle deklarativ modellierbaren Aspekte der Beantwortung einer Anfrage werden durch introspektive Methoden realisiert. Aspekte, deren Modellierung nicht wünschenswert ist, werden durch nicht-introspektive Methoden realisiert, was die Nutzung der vollen Ausdrucksmächtigkeit der Basisprogrammiersprache ermöglicht.

Aufgrund der bereits angesprochenen architekturellen Eigenschaften von Websystemen spielen die in Schritt 7b benutzten HTML-Templates eine große Rolle beim Erstellen einer Webanwendung. Wesentliche Eigenschaften der Anwendung werden durch sie spezifiziert (z.B.: Look&Feel, Zugriff auf Ressourcen, Navigation). Das Verhalten der Anwendung wird also einerseits durch größtenteils introspektive Java-Klassen und andererseits durch HTML-Templates spezifiziert. Die Konsistenz und das Zusammenspiel dieser beiden unterschiedlichen Artefakte sind von großer Bedeutung. Da es sich um introspektiven Java-Quellcode handelt, ist eine Modellbildung und ein anschließender Abgleich mit den HTML-Templates möglich. Die Konsistenz lässt sich somit automatisch überprüfen.

Eine weitere Anforderung und Designentscheidung beruht ebenfalls auf der Wichtigkeit der HTML-Artefakte. Analog zur exzellenten Toolunterstützung beim Bearbeiten von Java-Quell-

code ist eine Unterstützung beim Bearbeiten der HTML-Templates wünschenswert. Die wichtigsten geforderten Funktionalitäten sind das Bearbeiten im WYSIWYG-Modus sowie das automatische Überprüfen der Konsistenz der Links. Besonders das Prüfen der Link-Konsistenz stellt einen wichtigen Beitrag zur Qualitätssicherung dar. Ein Entwurfsziel unseres introspektiven Webframeworks bestand also darin, möglichst „freundlich“ mit existierenden HTML-Editoren zu interagieren.

7.5.3 Identifikation von Handlern

In Schritt 3 des vorgestellten Ablaufs wird der für die Beantwortung des Requests zuständige Handler identifiziert. Diese Identifikation erfolgt anhand der URL des Requests. Insbesondere ist der *Pfad* der URL relevant, welcher jenseits von Protokoll, Name des Servers und Port die aufgerufene Ressource einer Webanwendung identifiziert. In folgender URL:

```
http://wwwmatthes.in.tum.de/documents/search
```

ist der Pfad `documents/search`. Es findet also eine Abbildung von einem Pfad auf eine Klasse eines Handlers statt. Entsprechend den in Abschnitt 7.1 vorgestellten Entwurfsprinzipien von Toro wird für diese Abbildung das Prinzip „Convention over Configuration“ genutzt. Konkret bedeutet das, dass sich der Klassenname des Handlers schematisch aus dem Pfad ableiten lässt. Diese Abbildung muss also nicht zusätzlich konfiguriert werden, sondern ergibt sich per Konvention aus dem vorhandenen Klassennamen. Im konkreten Beispielfall wird ein Request mit dem Pfad `documents/search` von einem Handler mit dem Klassennamen `de.infoasset.broker.handler.documents.SearchHandler` beantwortet.

Einen Sonderfall, der nicht durch Konvention behandelt werden kann, stellen so genannte *dynamische* Pfade dar. So soll beispielsweise durch einen Request mit der URL

```
http://wwwmatthes.in.tum.de/document/123
```

das Dokument mit der ID `123` angezeigt werden. Da sich der zweite Teil des Pfades in Abhängigkeit vom konkreten Dokument ändert, ist eine Abbildung durch Konvention nicht möglich. Stabil ist in diesem Fall lediglich der erste Teil des Pfades. Die Identifikation des Handlers basiert also in diesem dynamischen Fall auf dem Präfix des Pfades. Ein konzeptuelles Modell dieser Abbildung in Form eines Klassendiagramms ist in Abbildung 87 dargestellt.

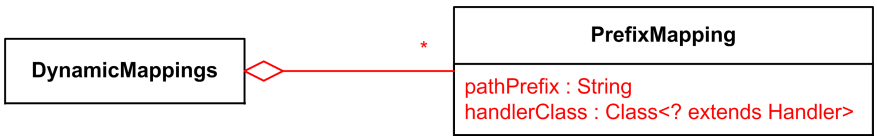


Abbildung 87: Konzeptuelles Modell der Abbildung von dynamischen Pfaden

Eine Realisierungsmöglichkeit dieser einfachen Abbildung besteht in der Spezifikation in einer Textdatei. Der Eintrag für den vorgestellten dynamischen Pfad wäre folgendermaßen:

```
document/* =
de.infoasset.broker.handler.documents.DocumentHandler
```

Dabei handelt es sich um eine sehr einfache domänenspezifische Sprache. Der Nachteil dieser Art der Spezifikation besteht darin, dass sich die Gültigkeit des Verweises auf eine Handlerklasse mit einfachen Mitteln nicht überprüfen lässt. Es besteht keine symbolische Integration zwischen dem Eintrag und der existierenden Klasse. Um diese Quelle von Inkonsistenzen zu vermeiden, wurde die Abbildung durch introspektiven Quellcode realisiert. Eine introspektive Spezifikation der im Beispiel vorgestellten Abbildung ist folgendermaßen:

```
mappings.add("document/", DocumentHandler.class);
```

In diesem Fall ist die Beziehung zwischen der vorgenommenen Abbildung und der Klasse des Handlers explizit und wird vom Compiler überprüft.

Zusammenfassend gibt es zwei Arten von Pfaden, die auf unterschiedliche Art zur Identifikation eines Handlers führen. Zunächst wird geprüft, ob ein konkreter Pfad mit einem Präfix eines registrierten dynamischen Pfades beginnt. Ist dies der Fall, behandelt der entsprechend registrierte Handler die Anfrage. Falls es sich nicht um einen dynamischen Pfad handelt, wird per Konvention generisch ein Handler mit entsprechendem Klassennamen instanziiert.

7.5.4 Auslesen von Parametern

Wie in Abbildung 86 angedeutet, wird ein HTTP-Request durch seine URL und eine Menge von Parametern repräsentiert. Nachdem die entsprechende Handlerklasse identifiziert ist und eine Instanz erzeugt wurde, werden in Schritt 4 die Werte der Parameter des Requests ausgelesen. Ein Parameter ist ein Name-Wert-Paar und wird im Allgemeinen anhand seines Namens identifiziert. Ein Request kann mehrere Parameter mit dem Selben Namen enthalten.

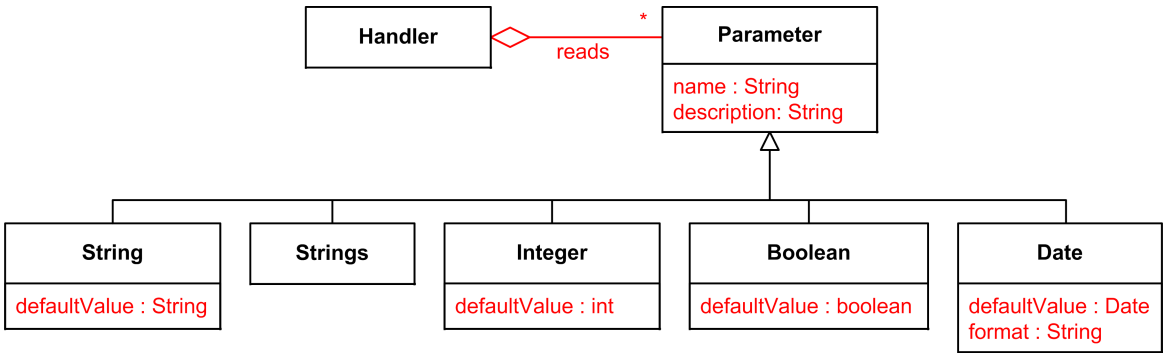


Abbildung 88: Konzeptuelles Modell des Auslesens von Parametern

Ziel dieses Schrittes ist die Umwandlung von Parametern des Requests in Werte von Attributen des Handlers. Technisch betrachtet sind alle Parameter im Request als Zeichenketten repräsentiert. Auf konzeptueller Ebene sind typisierte Parameter wünschenswert. Ein konzeptuelles Modell des Auslesens von Parametern ist in Abbildung 88 dargestellt. Ein Parameter kann über eine textuelle Beschreibung verfügen und von verschiedenem Typ sein. Weiterhin können Pa-

rameter einen typisierten Standardwert angeben. Im Falle eines `DateParameter` ist zusätzlich das Format, in dem das Datum formatiert sein muss, konfigurierbar.

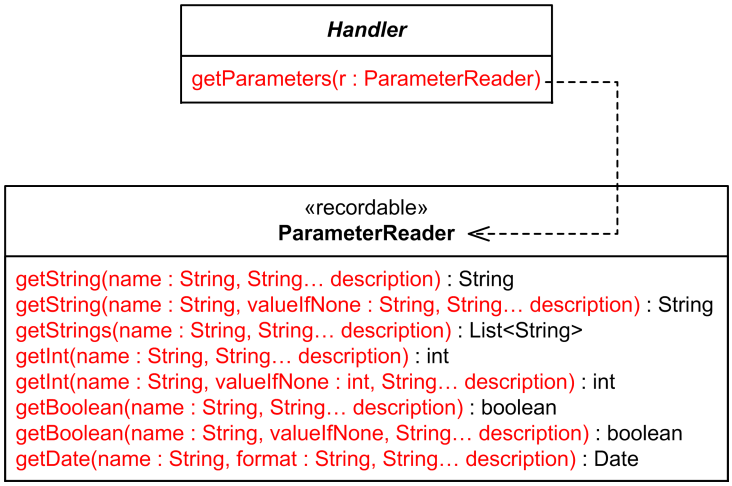


Abbildung 89: `ParameterReader` *zum introspektiven Auslesen von Parametern*

Gemäß den in Abschnitt 7.1 vorgestellten Entwurfsprinzipien wird das Auslesen von Parametern introspektiv realisiert. Konkret werden die Parameter in der introspektiven Recordable-Methode `getParameters` ausgelesen. Dies ist in Abbildung 89 in einem Klassendiagramm dargestellt (siehe hierzu auch Abschnitt 5.2.4). Interessant ist hierbei, dass das in Java 5 eingeführte Sprachelement der *Varargs* einen eleganten Mechanismus zur optionalen Bereitstellung einer textuellen Beschreibung ermöglicht. Das folgende Codefragment liest zwei Parameter aus:

```
id = reader.getString("id");
isNew = reader.getBoolean("isNew",
  "tells, whether this is a new asset");
```

7.5.5 Zugriffsprüfung und Geschäftslogik

Auf Basis der eingelesenen Parameter überprüft der Handler in Schritt 5, ob der angemeldete Benutzer autorisiert ist, die jeweilige Aktion durchzuführen. Dies wird im Allgemeinen realisiert, indem zunächst die durch die Parameter referenzierten Geschäftsobjekte materialisiert werden. Anschließend wird auf dieser Basis die Autorisierung des Benutzers geprüft. Beispielsweise materialisiert das folgende Codefragment ein Objekt vom Typ `Directory` und überprüft durch die Methode `mayRead` die Leseberechtigung des aktuellen Benutzers:

```
public void checkAccess() {
  directory = Directories.INSTANCE().getAsset(id);
  if(directory == null)
    throw new ObjectIsNullException();
  if(!directory.mayRead())
    throw new AccessException();
}
```

Diese Methode ist nicht-introspektiv und kann somit unter Nutzung der vollen Ausdrucksmächtigkeit von Java implementiert werden.

Ist diese Überprüfung positiv ausgefallen, wird die Geschäftslogik des Handlers ausgeführt. Dieser Aspekt wird durch die nicht-introspektive Einschubmethode `doBusinessLogic` der Klasse `Handler` realisiert. Ein Handler kann über verschiedene Möglichkeiten der Weiterverarbeitung verfügen. Dies ist in Abbildung 86 durch die Schritte 7a (Weiterleiten an einen weiteren Handler) und 7b (Generieren einer Antwortseite) dargestellt. Diese Möglichkeiten werden in unserem Webframework durch introspektive Objekte vom Typ `Line` und `Page` realisiert, welche ihrerseits den Typ `Station` erweitern. Ein Handler kann mehrere Stationen definieren, die unterschiedliche Weiterverarbeitungswege repräsentieren. Die Auswahl einer konkreten Station hängt vom Ergebnis der Geschäftslogik ab. Die Methode `doBusinessLogic` gibt also die ausgewählte Station direkt zurück. In Abbildung 90 ist ein Klassendiagramm dargestellt, welches diesen Zusammenhang graphisch darstellt.

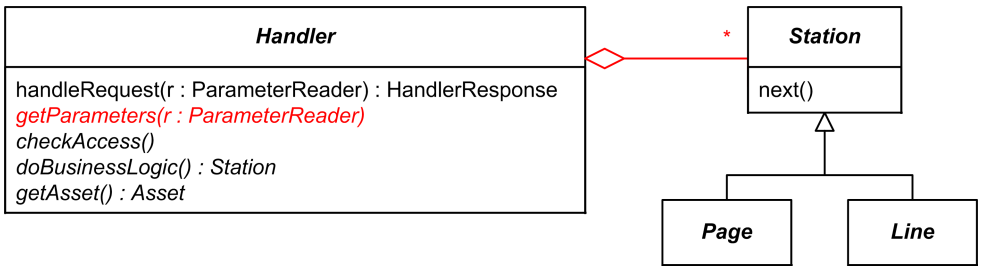


Abbildung 90: Klassendiagramm: `Handler` *und* `Station`

7.5.6 Weiterleitung

Die introspektive Beziehung zwischen `Handler` und `Station` wird durch eine ObjectsByType-Methode realisiert. Ein Handler spezifiziert also alle möglichen Weiterverarbeitungswege durch Deklaration von finalen Variablen durch anonyme innere Klassen. Beispielsweise wird durch folgende Variable eine Weiterleitung an die Klasse `DirectoryDetailHandler` spezifiziert:

```
class DirectoryDetailHandler extends Handler {
  final Station SHOW_DETAIL = new Line() {
    public void setParameters(ParameterWriter p) {
      p.setString("id", directory.getId());
    }

    public void next(Forwarder f) {
      f.go(DirectoryDetailHandler.class);
    }
  };
  ...
```

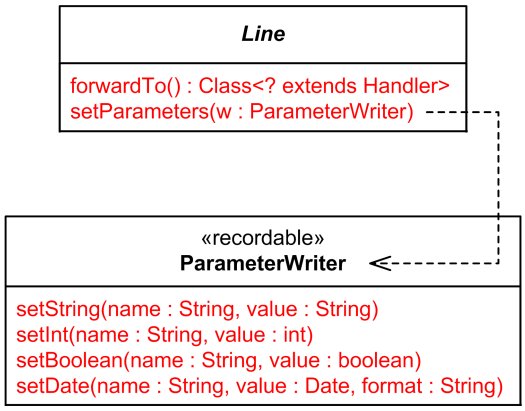


Abbildung 91: Klassendiagramm: `Line`

Durch die introspektive Methode `forwardTo` wird die Klasse des weiterverarbeitenden Handlers spezifiziert. Dies geschieht typisiert und wird vom Compiler überprüft. Weiterhin kann eine `Line` Parameter für den folgenden Handler setzen. Dies geschieht introspektiv in der Recordable-Methode `setParameters` (siehe Abbildung 91).

7.5.7 Antwortgenerierung

Der noch fehlende Schritt zur Beantwortung eines Requests ist das Generieren einer Antwortseite. Wie bereits in Abschnitt 7.5.2 dargestellt, ist dieser Schritt von großer Wichtigkeit. Die Antwortseite ist ein vom Zustand des Systems abhängiges HTML-Dokument. Wie ebenfalls bereits erwähnt, geschieht dies durch das Zusammenspiel von Java-Code einerseits und HTML-Templates andererseits. Templates sind Dokumente mit Platzhaltern, die dynamisch durch Ergebnisse von Auswertungen des Programmcodes ersetzt werden. Die Ersetzung wird von einer so genannten *Template-Engine* durchgeführt. Eine Template-Engine ist ein wichtiger Bestandteil eines jeden Webframeworks. Ihre Hauptaufgabe ist die Trennung von Geschäftslogik und Darstellungsaspekten.

Die Grundelemente der von unserem Webframework genutzten Template-Engine wurden bereits in [We02] beschrieben. HTML-Templates werden unter Benutzung von drei elementaren Platzhaltern spezifiziert, welche beliebig ineinander geschachtelt werden können. Dies sind

skalare Substitutionen, bedingte Substitutionen und Listen-Substitutionen. Jedes dieser Grundelemente wird auf eine bestimmte Art und Weise einerseits in HTML-Template und andererseits im Java-Code spezifiziert. Dies wird im Folgenden jeweils kurz vorgestellt.

Durch skalare Substitutionen lassen sich dynamisch erzeugte Textfragmente an beliebigen Stellen in einem Dokument einfügen. Syntaktisch werden skalare Substitutionen im Template durch einen mit einem Dollarzeichen beginnenden und endenden Bezeichner spezifiziert. Dies lässt sich durch eine kontextfreie Grammatik wie folgt formalisieren:

PrintSubstitution:
$ *Tag* **$**

Im dazugehörigen Java-Code wird die Ersetzung durch eine anonyme innere Klasse vom Typ `PrintSubstitution` spezifiziert. Diese Klasse verfügt über eine Methode `print`, welche den einzufügenden Inhalt zurückgibt. Ein Beispiel dafür ist in Tabelle 7 dargestellt. In der ersten Zeile sind Template und das daraus erzeugte Fragment dargestellt. Die zweite Zeile enthält den dafür verantwortlichen Java-Code.

`Name: $name$`	`Name: Thomas Büchner`

`template.put("name", new PrintSubstitution() {`
`  public String print() {`
`  return person.getName();`
`  }};`	

Tabelle 7: Beispiel: Skalare Substitution im Template und Java-Code

Durch bedingte Substitutionen kann die Ausgabe von Blöcken in Abhängigkeit von dynamisch ausgewerteten Bedingungen gesteuert werden. Die folgende Grammatik beschreibt, mit welcher Syntax bedingte Substitutionen im Template spezifiziert werden:

ConditionalSubstitution:
$[*Tag* **$** *Content* [**$]** *Tag* **[$** *Content*] **$** *Tag* **]$**
$[! *Tag* **$** *Content* [**$]** *Tag* **[$** *Content*] **$** *Tag* **]$**

Die erste Regel gibt den vorderen Teil aus wenn die Bedingung wahr ist. Der optionale hintere Teil wird ausgegeben wenn die Bedingung falsch ist. Die zweite Regel verhält sich genau anders herum. Im Java-Code erfolgt die Auswertung der Bedingung durch die Methode `test` der Klasse `ConditionalSubstitution`. In Tabelle 8 sind Template und Java-Code einer bedingten Substitution beispielhaft dargestellt.

`Die Bedingung ist`
`$[condition$`
`  wahr.`
`$]condition[$`
`  falsch.`
`$condition]$`
	`Die Bedingung ist wahr.`

```
template.put("condition", new ConditionalSubstitution() {
  public boolean test() {
    return true;
  }
});
```

Tabelle 8: Beispiel: Bedingte Substitution im Template und Java-Code

Als drittes Grundelement dienen Listen-Substitutionen zur wiederholten Ausgabe bestimmter Blöcke. Dabei sollen diese Blöcke nicht identisch wiederholt werden, sondern der jeweiligen Position angepasste Inhalte enthalten. Innerhalb eines Blocks befinden sich also Substitutionen, die sich auf das aktuelle Listenelement beziehen. Zur Unterscheidung von „globalen“ Substitutionen und solchen, die sich auf ein Listenelement beziehen, wird in der Definition einer Listen-Substitution ein Namensqualifikator angegeben, der dann im Listenblock benutzt werden kann. Die Syntax von Listen-Substitutionen lässt sich mittels einer kontextfreien Grammatik folgendermaßen beschreiben:

ListSubstitution
$[*Tag Qualifier* **$** *Content* [**$|$** *Content*] **$** *Tag* **]$**

In einigen Fällen ist es sinnvoll, das letzte Element einer Liste anders darzustellen. Dies ist beispielsweise der Fall, wenn die Listenelemente durch Komma voneinander getrennt werden sollen. Nach dem letzten Element soll allerdings kein Komma erscheinen. Zu diesem Zweck kann die letzte Zeile einer Listen-Substitution separat durch `$|$` getrennt angegeben werden. In Tabelle 9 sind die drei Inkarnationen einer Listen-Substitution beispielhaft dargestellt.

`$[list e$`
`  $e.print$ ,`
`$\|$`
`  $e.print$`
`$list]$`
	`erstens, zweitens, drittens`

```
template.put("list", new ListSubstitution() {

  Iterator<Integer> i;
  Integer current;

  public void start() { i = list.iterator(); }
  public boolean hasNext() { return i.hasNext(); }
  public void next() { current = i.next(); }

  public void putSubstitutions(Template t) {
    t.put("print", new PrintSubstitution() {
      public String print() {
        return current.toString();
      }
    });
  }
});
```

Tabelle 9: Beispiel: Listen-Substitution im Template und Java-Code

Die Möglichkeiten der Spezifikation einer Antwortseite beschränken sich also im Wesentlichen auf die drei vorgestellten Grundelemente. Es handelt sich hierbei um eine Template-Engine mit bewusst beschränkter Ausdrucksmächtigkeit. Im Gegensatz dazu verfügt ein Großteil der weit verbreiteten Template-Engines über die volle Turing-Mächtigkeit. Es ist also möglich und dementsprechend auch gebräuchlich, direkt im Template komplexe Programmlogik zu spezifizieren. Dies führt zu einer Verwischung der Trennung von Geschäftslogik und Darstellungsaspekten und reduziert die Lesbarkeit und Wartbarkeit der Anwendung. Ein weiterer Vorteil von bewusst einfachen Templates besteht darin, dass eine Bearbeitung durch Webdesigner möglich ist. Für eine weiterführende Diskussion der Vorteile von Template-Engines mit beschränkter Mächtigkeit sei auf [Pa04] verwiesen.

Introspektive Konsistenzüberprüfung

Bei jeder Template-Engine, die ein Dokument durch das Zusammenspiel getrennter Artefakte (Templates und Programmcode) erzeugt, können Probleme durch Inkonsistenzen zwischen den beteiligten Artefakten entstehen. Zu jedem Platzhalter muss eine entsprechende Substitution im Programmcode existieren. Eine manuelle Überprüfung der Konsistenz ist insbesondere bei großen Projekten mit vielen hunderten Templates sehr aufwändig. Ein Alleinstellungsmerkmal unserer Template-Engine besteht darin, diese Konsistenzüberprüfung automatisch durchführen

zu können. Dies ist möglich, da einerseits die Templates über eine einfach analysierbare Struktur verfügen. Andererseits wird der Java-Code introspektiv programmiert und kann somit ebenfalls analysiert werden. Es ist also möglich, zur Entwicklungszeit Modelle beider Artefakte zu erstellen und diese dann zur Konsistenzüberprüfung zu verwenden.

Wie in Abbildung 90 dargestellt, werden Antwortseiten in unserem Webframework durch Objekte vom Typ `Page` generiert. Um die Konsistenz überprüfen zu können, muss einerseits das verwendete Template identifiziert werden können. Andererseits werden alle bereitgestellten Substitutionen gebraucht. Die introspektive Spezifikation des zu verwendenden Templates wird durch die Methode `specifyTemplate` realisiert. Dabei handelt es sich um eine Recordable-Methode, in der das Template durch Methodenaufrufe auf einem Objekt vom Typ `TemplateFinder` spezifiziert wird. Der folgende Methodenrumpf identifiziert ein Template mit dem Namen `directory/directoryDetail`:

```
public void specifyTemplate(TemplateFinder f) {
  f.staticName("directory/directoryDetail");
}
```

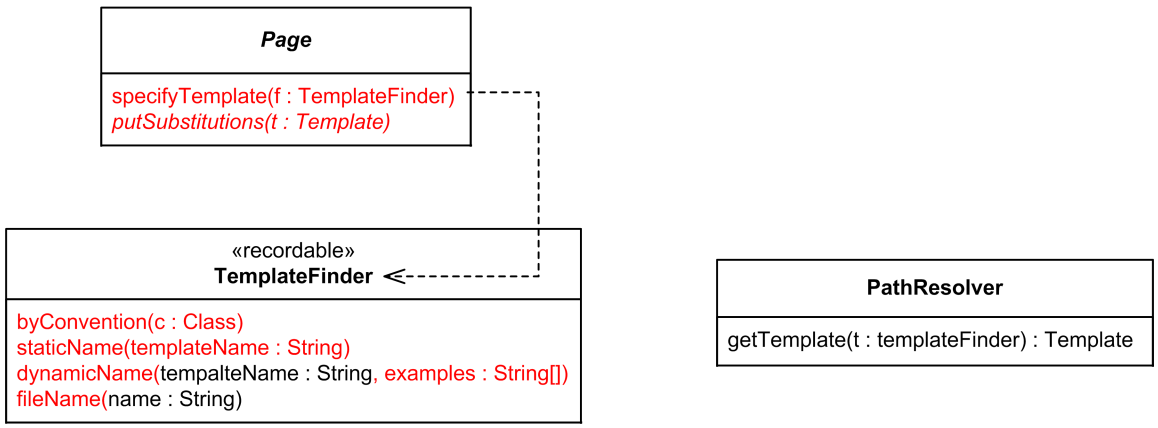


Abbildung 92: Introspektive Spezifikation von Templates

In Abbildung 92 sind die zur introspektiven Identifikation von Templates benötigten Klassen in einem Diagramm dargestellt. Zusätzlich zur Angabe eines statischen Namens ist die Übergabe der Klasse, die das Template benötigt, möglich. In diesem Fall wird das Prinzip „Convention over Configuration“ angewendet. Die in der abstrakten Basisklasse `Page` vorhandene Standardimplementierung nutzt diese Art der Spezifikation. Nur in Ausnahmefälle ist also das Überschreiben der Methode `specifyTemplate` überhaupt notwendig. Ist eine statische Angabe des Template-Namen nicht möglich, kann die Methode `dynamicName` genutzt werden. Um Introspektion und damit Konsistenzüberprüfung dennoch zu ermöglichen, ist in diesem Fall die Angabe von exemplarischen Namen möglich. Dies ist sinnvoll, wenn zur Laufzeit dynamisch aus einer Menge von Namen ausgewählt werde soll.

Um eine Konsistenzüberprüfung durchführen zu können, ist zusätzlich zur Identifikation des benutzten Templates ein Modell aller von einer Page gesetzten Substitutionen notwendig. Dies ist zur Entwicklungszeit möglich, da alle Substitutionen in der introspektiven Methode `put-`

`Substitutions` definiert werden. Auch hier wurde das Programmiermodell der Recordable-Methode gewählt (siehe Abbildung 93).

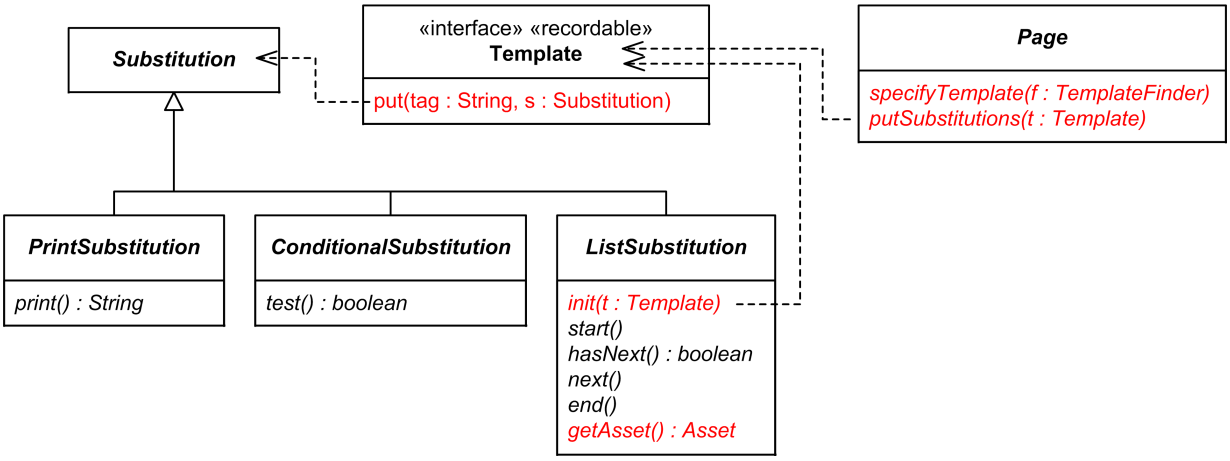


Abbildung 93: Introspektive Spezifikation von Substitutionen

Zur Visualisierung des introspektiven Modells eines Handlers sowie zur Überprüfung der Konsistenz wurde im Rahmen dieser Arbeit ein Werkzeug entwickelt, welches den in IMF vorhandenen Whitebox Modeler erweitert. Dieses Werkzeug wird im Folgenden *Consistency Checker* genannt [Kä05]. In Abbildung 94 ist ein Screenshot der Hauptansicht des Consistency Checker dargestellt. Im oberen Bereich sind die bereitgestellten Substitutionen dargestellt, es handelt sich dabei um die in den Beispielen eingeführten. Im unteren Bereich ist eine Analyse der im Template definierten Platzhalter abgebildet. Die Hauptaufgabe des Consistency Checkers besteht darin, beide Artefakte miteinander abzugleichen und eine integrierte Sicht anzubieten. So ist es möglich, zwischen korrespondierenden Artefakten zu navigieren. Im abgebildeten Fall ist die in eine List eingebettete skalare Substitution ausgewählt. Einerseits ist es möglich, in das Template zum Platzhalter zu springen. Andererseits ist eine Navigation zur dazugehörigen Substitution in der Page möglich. Dabei kann einerseits zur Repräsentation der Substitution im Consistency Checker gesprungen werden. Andererseits ist es möglich, direkt zum Quellcode zu navigieren. Im abgebildeten Beispiel wurde direkt zum Java-Code gesprungen.

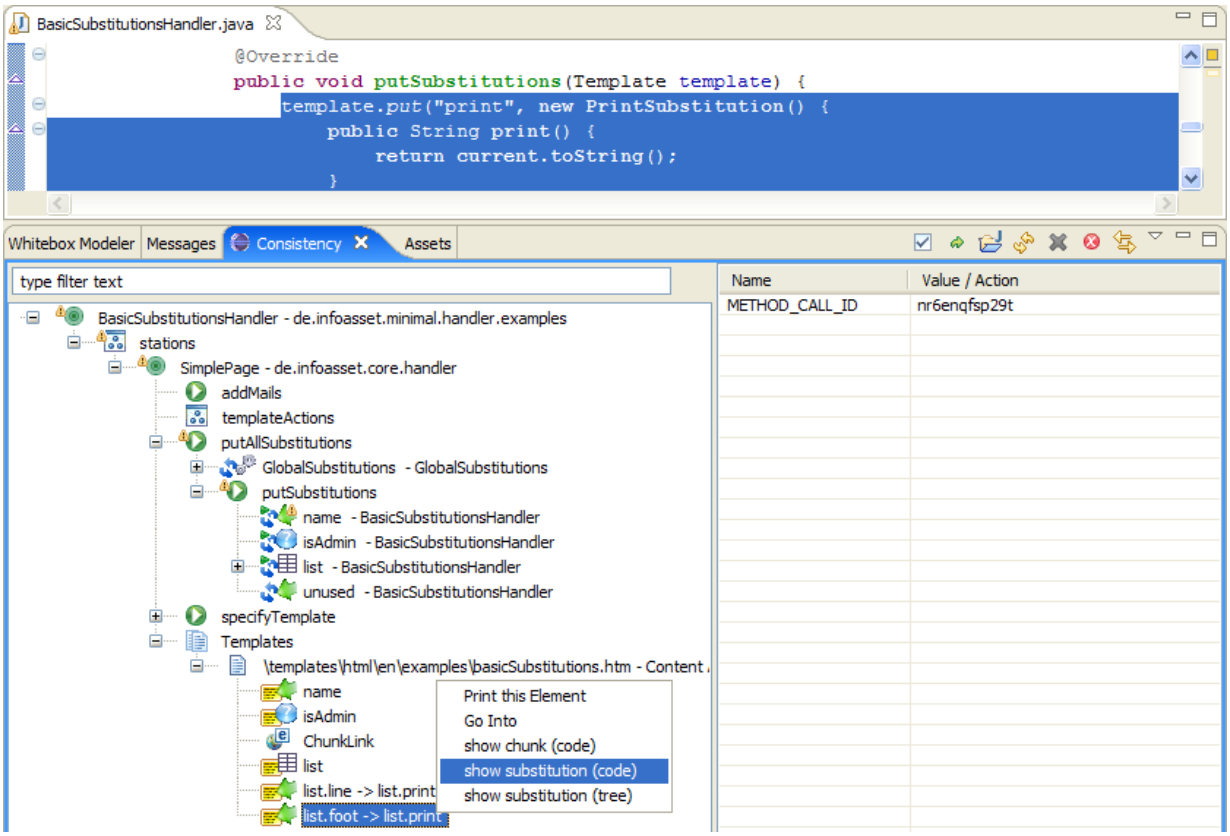


Abbildung 94: Introspektive Konsistenzüberprüfung

Eine weitere Funktionalität des Consistency Checker besteht darin, die Verwendung von definierten Substitutionen zu visualisieren. Die im oberen Bereich visualisierten Substitutionen `name`, `isAdmin` und `list` wurden vom Template verwendet. Dies wird durch die Dekoration mit einem grünen Pfeil dargestellt. Die Substitution `unused` wird an keiner Stelle im Template benutzt und ist demzufolge nicht dekoriert.

Der offensichtlichste Anwendungsfall des Consistency Checkers ist die Überprüfung der Konsistenz zwischen Java-Code und Templates. Ein Handler ist konsistent, wenn zu jedem Platzhalter im Template eine Substitution in der Page existiert. Inkonsistenzen werden im Consistency Checker durch eine rote Dekoration des nicht bedienten Platzhalters visualisiert. Dies ist in Abbildung 95 dargestellt. In diesem Fall ist es möglich, eine Rumpf-Substitution an die passende Stelle einzufügen.

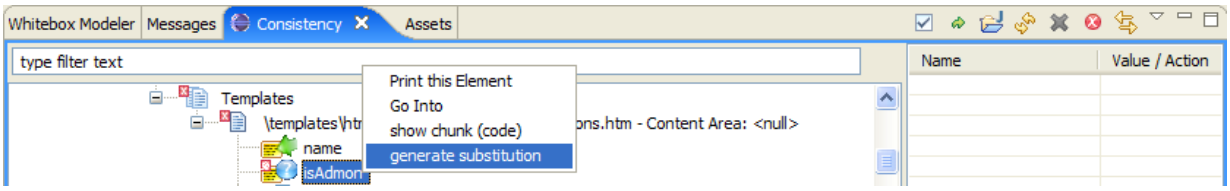


Abbildung 95: Gefundene Inkonsistenz

Polymorphe Substitutionen

Im Folgenden werden Strategien vorgestellt, die die Anzahl der im Java-Code zu definierenden Substitutionen minimieren helfen. Ein typisches Szenario beinhaltet die Ausgabe von Inhalten in Abhängigkeit vom Vorhandensein eines skalaren Wertes, welcher durch eine `PrintSubstitution` zur Verfügung gestellt wird. Beispielsweise soll, falls der Wert der eingeführten skalaren Substitution `name` leer ist, eine Meldung ausgegeben werden. Dies lässt sich mit den vorgestellten Mitteln nur durch eine zusätzliche `ConditionalSubstitution` bewerkstelligen. Um dieses Szenario besser zu unterstützen, sind die vorgestellten skalaren und Listen-Substitutionen polymorph und können im Template als bedingte Substitution verwendet werden. So ist es einfach möglich, in Abhängigkeit vom Vorhandensein von Elementen einer Listen-Substitution, Abschnitte ein- und auszublenden.

Wiederverwendung von Substitutionen

Webanwendungen beinhalten of komplexe wiederkehrende Elemente an unterschiedlichen Stellen. Ein Beispiel hierfür ist die Darstellung eines Warenkorbs an verschiedenen Stellen einer Webanwendung. Um Redundanz zu vermeiden ist es notwendig, Visualisierungselemente wieder verwenden zu können. Dies muss auf eine Art und Weise erfolgen, die eine Introspektion und somit Integration und Konsistenzüberprüfung ermöglicht.

Zu diesem Zweck existieren zwei weitere Substitutionsarten. Objekte vom Typ `SubstitutionAggregator` ermöglichen die zentrale Definition von Substitutionen, die mehrfach verwendet werden sollen. Hiermit ist lediglich eine Wiederverwendung von Java-Code möglich (siehe Abbildung 96). In Abbildung 94 wurde ein solches Element mit dem Namen `GlobalSubstitutions` verwendet.

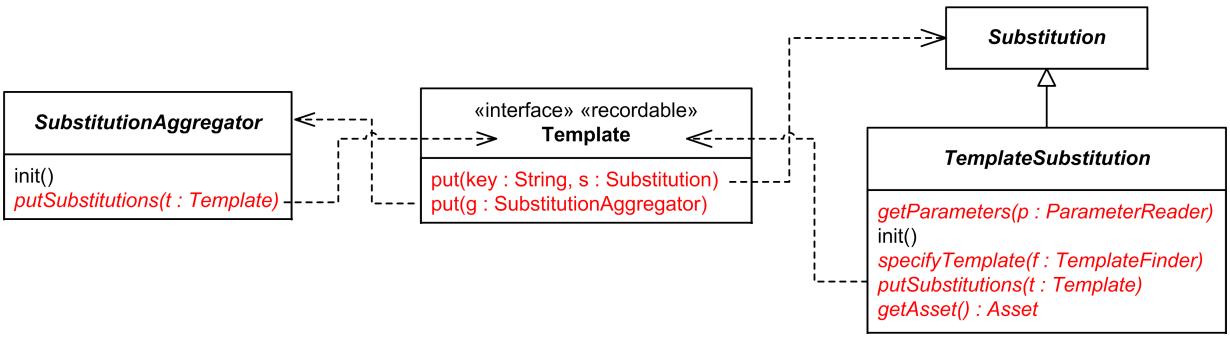


Abbildung 96: Wiederverwendung von Substitutionen

Die Wiederverwendung von sowohl Substitutionen als auch dazugehörigen Templates wird durch Objekte vom Typ `TemplateSubstitution` realisiert. Template-Substitutionen können Parameter auslesen, definieren Substitutionen und spezifizieren ein zu verwendendes Template. All dies geschieht introspektiv, so dass die Überprüfung der Konsistenz möglich ist.

Substitutions-Funktionen

Wie bereits dargestellt, ermöglichen Webanwendungen das Bearbeiten von Geschäftsobjekten über Webtechnologien. Eine dabei immer wiederkehrende Anforderung ist das Anzeigen und Bearbeiten von Geschäftsobjekten. Dabei werden alle Eigenschaften eines Geschäftsobjekts visualisiert. Die Umsetzung dieses Szenarios mit den bisher vorgestellten Elementen erfordert die stereotype Definition einer großen Menge von Substitutionen im Java-Code. Wünschenswert ist in diesem Fall die alleinige Spezifikation im Template. Dies ist durch *Substitutions-Funktionen* möglich. Diese ermöglichen den im Template parametrisierten Zugriff auf serverseitige Funktionen.

Eine besondere Rolle kommt hierbei Funktionen, die auf dem Zustand von Geschäftsobjekten arbeiten zu. So wird beispielsweise der Platzhalter `$firstName.propertyValue()$` durch den Wert der Eigenschaft `firstName` des aktuellen Assets ersetzt. Substitutions-Funktionen werden im Java-Code durch Subklassen von `PrintFunction` realisiert. Es existieren vier in Abbildung 97 dargestellte Arten von Substitutions-Funktionen, welche sich bezüglich des Kontexts in dem sie agieren, unterscheiden. Die unterstützten Kontexte sind ein konkretes Asset, eine Eigenschaft oder eine Rolle eines Assets, sowie der globale Kontext.

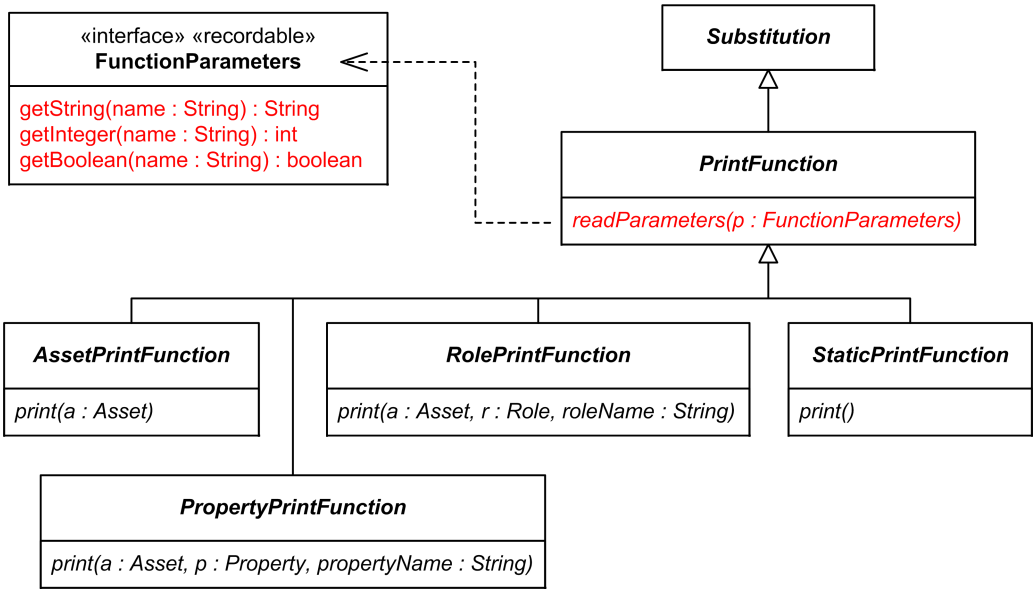


Abbildung 97: Klassendiagramm: Substitutions-Funktionen

Im Template wird der Zugriff auf Substitutions-Funktionen durch folgende Grammatik spezifiziert:

PrintFunction
$ [*Identifier* { **.** *Identifier* } **.**] *FunctionName*
([*Parameter* { **;** *Parameter* }] **)$**

Parameter werden folgendermaßen angegeben:

Parameter
    ParameterName **=** *ParameterValue*

Die semantische Interpretation der im Template angegebenen Identifier erfolgt nach den in Tabelle 10 angegebenen Regeln. In Abhängigkeit von der Anzahl vorhandener Identifier werden die in der Tabelle angegebenen Optionen nacheinander angewandt. Ist beispielsweise ein Identifier vorhanden, so wird zunächst überprüft ob es eine umschließende Listen-Substitution mit diesem Namen gibt. Ist dies der Fall, so wird zunächst nach einer korrespondierenden `StaticPrintFunction`, anschließend nach einer `AssetPrintFunction` gesucht. Falls keine Listen-Substitution gefunden wurde, wird der Identifier als Name einer Eigenschaft des aktuellen Assets interpretiert. Existiert eine solche Eigenschaft, wird nach einer korrespondierenden `PropertyPrintFunction` gesucht. Falls keine Eigenschaft mit diesem Namen existiert, wird der Identifier als Rollenname des Assets interpretiert.

Identifier-Anzahl	Bedeutung von Identifier[0]	Bedeutung von Identifier[1]
0 [*static* | *asset*]	-	-
1	Listen-Qualifier [*static* | *asset*]	-
	Eigenschaftsname [*property*]	-
	Rollenname [*role*]	-
2	Listen-Qualifier	Eigenschaftsname [*property*]
		Rollenname [*role*]

Tabelle 10: Kontext von Substitutions-Funktionen

Der vorgestellte Ablauf der Interpretation von Substitutions-Funktionen ist bereits zum Entwurfszeitpunkt vollständig durchführbar. Dabei wird falls notwendig ein Modell des Typs des aktuellen Assets erstellt (siehe Abschnitt 7.4.3) und in die Analyse mit einbezogen. Dies ist in Abbildung 98 dargestellt. Im mittleren Teil des Screenshots ist ein Modell des Templates dargestellt, welches die Substitution `$firstName.value()$` enthält. Der Identifier `firstName` wird in diesem Fall als Name einer Eigenschaft interpretiert. Das Modell des aktuellen Geschäftsobjektes `Person` ist im unteren Teil zu sehen. Die korrespondierende `PropertyPrintFunction` `value` ist im oberen Teil markiert. Die introspektive Analysesicht ermöglicht Navigation zwischen den beteiligten Artefakten. In diesem dargestellten Fall handelt es sich um eine Integration von Modellen verschiedener domänenspezifischer Sprachen (siehe Abschnitt 2.3.7).

Wie im oberen Teil von Abbildung 98 dargestellt, existiert eine Reihe von generischen Substitutions-Funktionen als Teil der Plattform. Besonders soll hier die Eigenschafts-Substitution `propertyInput` hervorgehoben werden, welche abhängig vom Typ und Wert der Eigenschaft eine Visualisierung als Webcontrol einfügt. Dies ermöglicht die effiziente Realisierung von Bearbeitungsmasken.

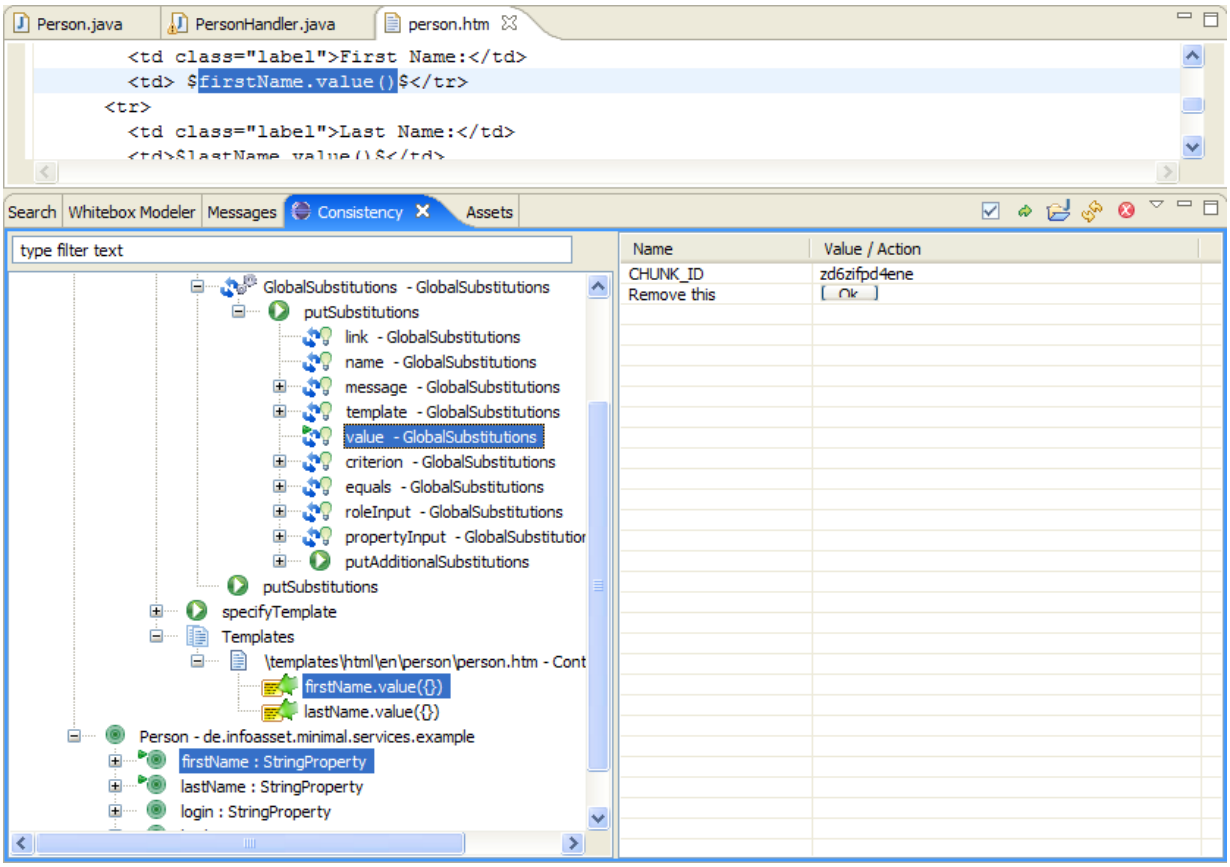


Abbildung 98: Introspektive Analyse von Substitutions-Funktionen

7.5.8 Vergleich mit existierenden Technologien zur Webvisualisierung

In diesem Abschnitt wird das vorgestellte introspektive Webvisualisierungs-Framework den auf Java basierten vorherrschenden Technologien gegenübergestellt. Dies sind einerseits Servlets [Hu01] zur Realisierung der Schritte 3 bis 7a des in Abbildung 86 dargestellten schematischen Ablaufs sowie *JavaServer Pages* (JSP) zur Realisierung der in Abschnitt 7.5.7 beschriebenen Erzeugung von Antwortseiten [Be03]. Die Trennung in diese beiden Technologien resultiert aus der in [Pa04] motivierten Trennung von Präsentationsbeschreibung und Anwendungslogik.

Abbildung 99: Klassendiagramm: HttpServlet

Servlets werden zur Realisierung der Anwendungslogik benutzt. In Abbildung 99 ist die Klasse `HttpServlet` in einem Klassendiagramm dargestellt, die von konkreten Servlets erweitert wird. Diese implementieren die abstrakte Methode `service` und spezifizieren auf diesem

Weg das gewünschte Verhalten. Die als Parameter übergebenen Objekte vom Typ `HttpServletRequest` und `HttpServletResponse` ermöglichen einen objektorientierten Zugriff auf Eigenschaften von Anfrage und Antwort. So ist beispielsweise das Auslesen von Parametern der Anfrage durch die Methode `getParameter` von `HttpServeltRequest` möglich. Im Unterschied zum vorgestellten introspektiven Webvisualisierungs-Framework werden die verschiedenen durch die Schritte 3 bis 7a in Abbildung 86 repräsentierten Aspekte bei dieser Implementierungsform nicht voneinander getrennt realisiert. Die in dieser Arbeit vorgeschlagene Trennung erhöht einerseits die Verständlichkeit und ermöglicht andererseits das Extrahieren von Modellinformationen. So liefert eine introspektive Analyse eines Handlers alle von ihm ausgelesenen Parameter sowie mögliche Weiterleitungen an andere Handler. Diese Informationen können beispielsweise zur Dokumentation verwendet werden.

Ist die Anwendungslogik abgeschlossen, wird eine Antwortseite erzeugt. Die hierfür am weitesten verbreitete Technologie sind JavaServer Pages. Prinzipiell handelt es sich dabei gemäß der in [Pa04] eingeführten Terminologie um eine *unrestricted* Template-Sprache, die das Einbetten von beliebigem Java-Code ermöglicht. Aufgrund der damit verbundenen Probleme (siehe [Pa04]) werden im Allgemeinen so genannte *Tag Libraries* verwendet, die über einen eingeschränkten Funktionsumfang verfügen. Diese Tag Libraries verfügen über einen vergleichbaren Funktionsumfang wie die in Abschnitt 7.5.7 vorgestellten Platzhalter und Funktionen, so dass das Erstellen einer JavaServer Page mit dem Erstellen eines in Abschnitt 7.5.7 vorgestellten Templates vergleichbar ist.

Beide Ansätze unterscheiden sich grundsätzlich darin, wie das Zusammenspiel von dynamischem Java-Code und den in den Templates definierten Platzhaltern stattfindet. Das in diesem Kapitel vorgestellte Framework realisiert dynamische Substitutionen introspektiv, so dass ein Modell erstellt werden kann. Mit diesem Modell kann die Konsistenz zwischen den im Template definierten Platzhaltern und den im Java-Code definierten dynamischen Substitutionen überprüft werden. Dies ist unter Benutzung der klassischen Technologien nicht möglich. Beide Artefakte (Servlets und JavaServer Pages) sind nicht miteinander integriert und somit ist Werkzeugunterstützung bei der Suche nach Inkonsistenzen nicht möglich.

7.6 Entwicklung von Webanwendungen mit Toro

Webanwendungen werden in einem iterativ-inkrementellen Prozess entworfen, in dem zunächst das Design der Webseite spezifiziert wird [DLH07]. Der Prozess gliedert sich in sieben Schritte:

1. *Discovery*. In diesem Schritt werden die durch die Webseite verfolgten Ziele aufgenommen und es wird ein Verständnis für die adressierten Kunden aufgebaut.
2. *Exploration*. Als nächstes werden erste grobe Entwürfe für das Design der Anwendung entworfen, von denen am Ende dieser Phase ein Entwurf für die weitere Entwicklung ausgewählt wird.
3. *Refinement*. Design und Navigationsstruktur des ausgewählten Entwurfs werden verfeinert.
4. *Production*. In diesem Schritt wird ein vollständig interaktiver Prototyp, der das endgültige Design repräsentiert fertig gestellt.

5. *Implementation*. Erst in diesem Schritt wird die eigentliche Dynamik der Webanwendung realisiert.
6. *Launch*. Die fertig entwickelte Webanwendung wird in diesem Schritt für die Benutzer freigeschaltet.
7. *Maintenance*. In dieser oft vernachlässigten Phase werden Fehler beseitigt, wird das Funktionieren der Anwendung sichergestellt und es findet der Betrieb der Anwendung statt.

In den Schritten 1 bis 4 wird das eigentliche Design einer Webanwendung erstellt. Dies wird in vielen Fällen von darauf spezialisierten Agenturen durchgeführt, die statische HTML-Seiten als Ergebnis dieses Schritts abliefern. Auf Basis dieser statischen Seiten entsteht in Schritt 5 eine dynamische Anwendung durch „dynamisieren“ der statischen HTML-Seiten. Alle Phasen werden jeweils mehrfach iterativ durchlaufen und erzeugen in jedem Durchlauf ein bestimmtes Inkrement.

7.6.1 Rapid Application Development mit Toro

Die in dieser Arbeit vorgestellte Webplattform Toro unterstützt den dargestellten kundenzentrierten Entwicklungsprozess, indem ein nahtloser Übergang von statischen HTML-Seiten zur dynamischen Realisierung unterstützt wird. Die wesentlichen Eigenschaften einer Webanwendung werden in Toro durch die folgenden Artefakte beschrieben:

- Templates spezifizieren das Aussehen der Anwendung.
- Assets spezifizieren alle verwalteten Geschäftsobjekttypen.
- Handler realisieren die Geschäftslogik.

In den Phasen 1 bis 4 werden die entwickelten statischen HTML-Seiten zunächst von einem Standard-Handler ausgeliefert. Anschließend wird ein einfacher Handler pro zu dynamisierender Seite erstellt, der zunächst keinerlei Substitutionen enthält. An den Stellen, an denen Dynamik erforderlich ist, werden als nächstes im Template Platzhalter eingefügt. Durch den Consistency Checker lassen sich daraus die zur Implementierung der Dynamik im Java-Code notwendigen Substitutionsrümpfe generieren. An den generierten Stellen muss nun lediglich vom Entwickler das gewünschte Verhalten realisiert werden.

Eine häufig auftretende Besonderheit stellen Templates dar, in denen Geschäftsobjekte visualisiert werden. Die erstellten statischen HTML-Seiten enthalten Bearbeitungsmasken wie in Abbildung 100 dargestellt.

Diese Masken beschreiben Modellierungsinformationen über die von der Anwendung zu verwalteten Geschäftsobjekttypen in einer für Benutzer des Systems verständlichen Form. Wie im Abschnitt 7.5.7 dargestellt, wird die fertige Realisierung zu diesem Zweck die Eigenschafts-Substitution `propertyInput` verwenden. Dies setzt jedoch die Existenz des Geschäftsobjekttyps mit einer entsprechenden Eigenschaft voraus. Um die Modellierung von Geschäftsobjekttypen zu vereinfachen, kann diese Eigenschafts-Substitution auch wie in Abbildung 101 dargestellt parametrisiert verwendet werden. Die Eigenschaften des fehlenden Geschäftsob-

jekttyps werden durch Parameter angegeben und die vom System ausgelieferte Antwortseite enthält trotz fehlender Eigenschaft die in Abbildung 100 dargestellten Eingabemasken.

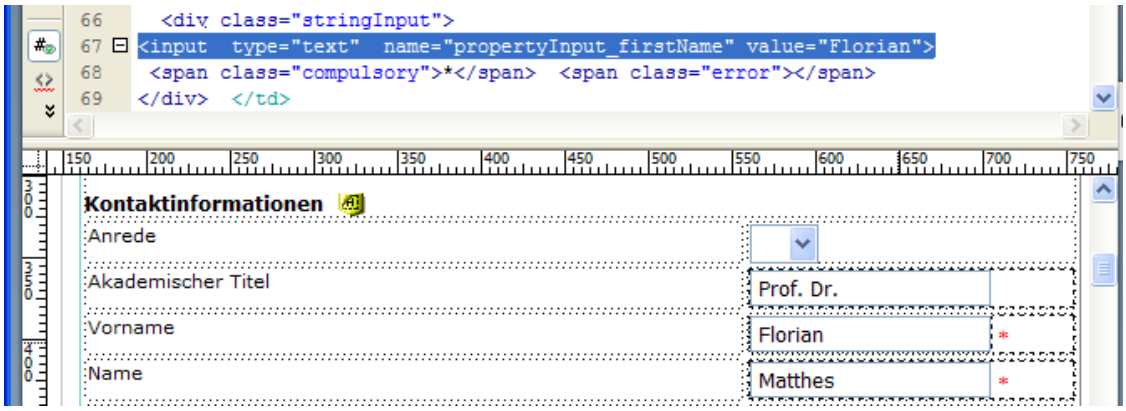


Abbildung 100: Statische Bearbeitungsmaske

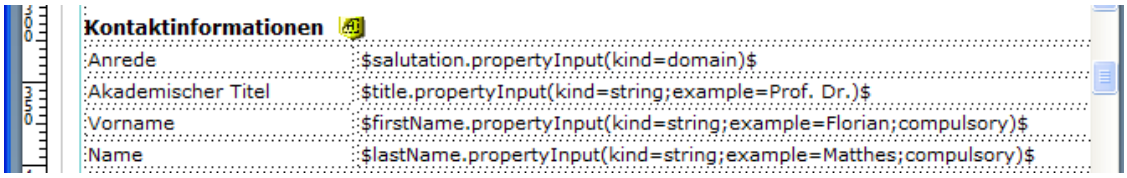


Abbildung 101: Statische Definition von Bearbeitungsmasken

Eine introspektive Analyse der Seite liefert Fehler, die sich auf die fehlenden Eigenschaften beziehen. Auf den zur Verfügung gestellten Parametern aufbauend kann Modellierungscode generiert werden, der den durch die Parameter beschriebenen Eigenschaften entspricht (siehe Abbildung 102). So ist es möglich, durch statisch definierte parametrisierte Eingabemasken, die vom System verwalteten Geschäftsobjekttypen in einer für Anwender verständlichen Art und Weise zu modellieren.

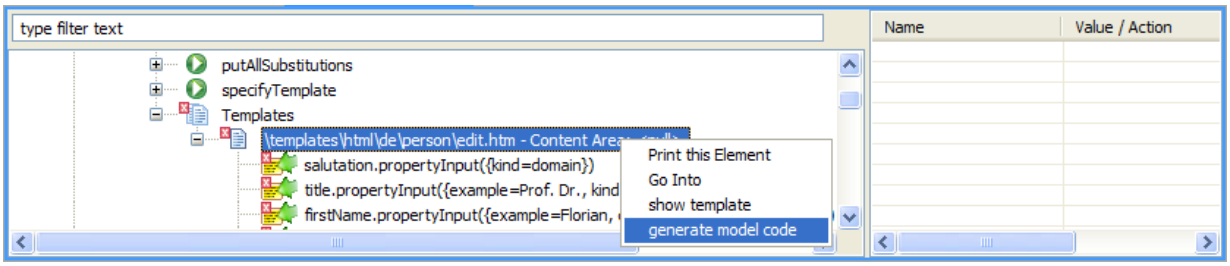


Abbildung 102: Generierung des Datenmodells

7.6.2 Integrierte Modellperspektiven in Toro

In Abbildung 103 sind die bereits in Kapitel 7.2 eingeführten Modellierungsperspektiven einer auf Basis von Toro entwickelten Webanwendung mit den erstellten Werkzeugen und den reali-

sierenden Artefakten dargestellt. Die Modelle sind entsprechend ihren Abhängigkeiten untereinander integriert. Es existieren zwei verschiedene Integrationsmechanismen. Die Integration von Templates mit dem Quellcode der Handler sowie mit dem Datenmodell erfolgt durch das Werkzeug Consistency Checker. In allen weiteren Fällen findet Integration mit Mitteln der Basisprogrammiersprache statt und wird durch die verwendete Post-IntelliJ-IDE Eclipse überwacht.

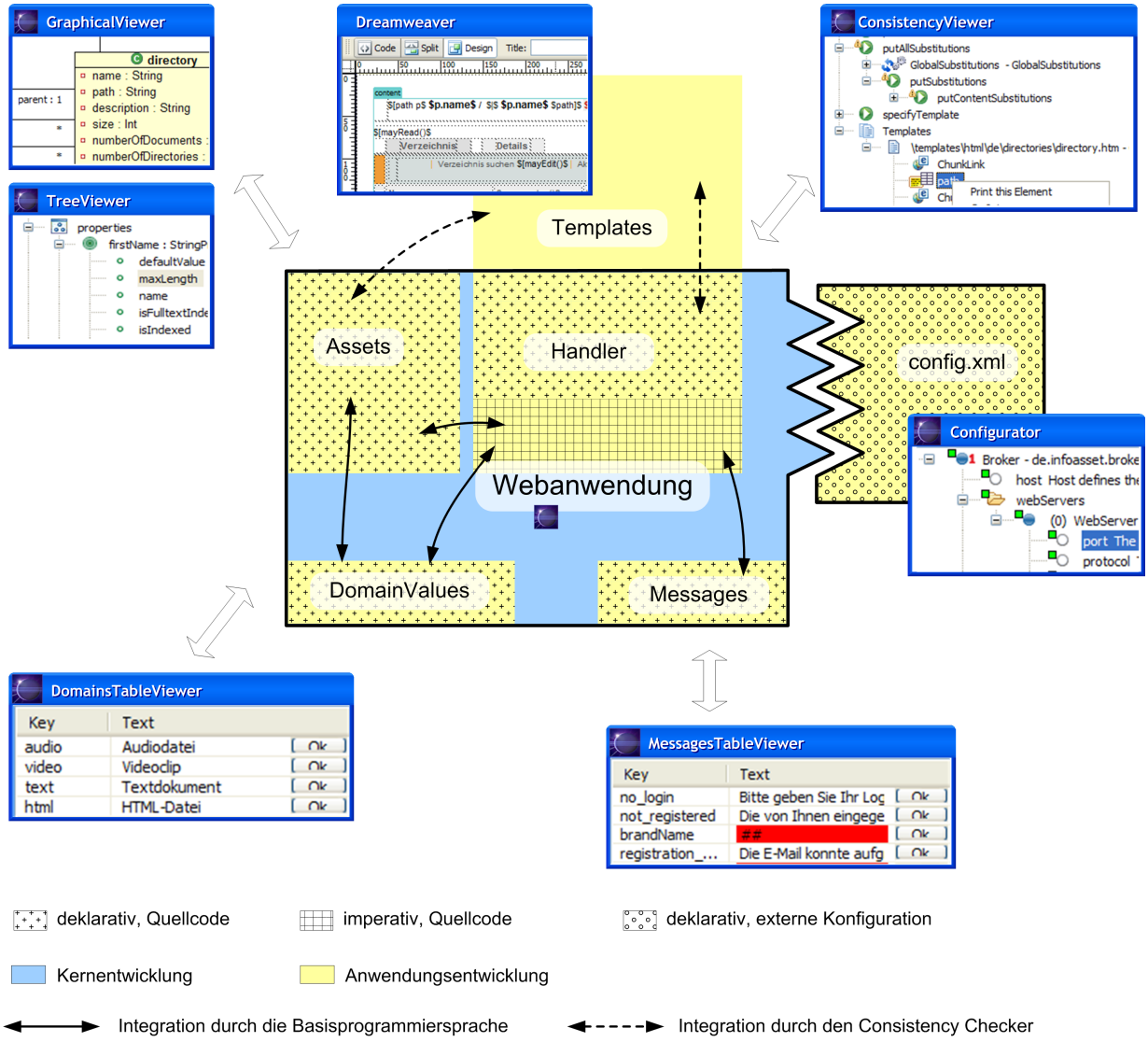


Abbildung 103: Integrierte Modellierungsperspektiven in Toro

7.7 Bewertung der introspektiven Plattform

In diesem Abschnitt wird die vorgeschlagene introspektive Plattform bewertet. Dabei wird insbesondere auf die Eigenschaften eingegangen, die sich aus der Anwendung des Ansatzes der introspektiven modellgetriebenen Softwareentwicklung ergeben. Zunächst werden in Abschnitt

7.7.1 auf Basis von Toro realisierte Anwendungen sowie die hierbei gesammelten Erfahrungen vorgestellt. Anschließend wird in Abschnitt 7.7.2 ein nicht-geplantes Wiederverwendungsszenario besprochen, in dem eine existierende, auf Basis von Toro realisierte Anwendung, auf veränderte Kundenanforderungen angepasst werden soll.

7.7.1 Realisierte Anwendungen

Basierend auf Toro wurden in den vergangenen Jahren zahlreiche Webanwendungen realisiert, welche größtenteils in kommerziellen Szenarien eingesetzt werden. Diese Anwendungen wurden hauptsächlich basierend auf dem in [We02] beschriebenen Portalsystem für das Wissensmanagement realisiert. Dieses System wurde technologisch auf Toro portiert und verfügt somit über die vorgestellten Modellierungsperspektiven. Der gesamte Quelltext des Systems umfasst 100000 Zeilen Java-Code.

Die Portierung des Systems hin zur vermehrten Nutzung introspektiver modellgetriebener Softwareentwicklung wurde schrittweise vorgenommen. In einem ersten Schritt wurde der auf Textdateien und der Klasse `java.util.Properties` basierende Konfigurationsmechanismus durch den in Kapitel 4 vorgestellten Mechanismus der Blackbox Introspektion ersetzt. Diese Portierung verursachte einen Aufwand von 15 Personentagen (siehe [BK04b]). In Anhang B ist ein vollständiges Konfigurationsmodell des Kernsystems dargestellt. Es umfasst 130 konfigurierte Klassen mit 200 konfigurierten Eigenschaften und 120 Beziehungen. Da die Konfiguration mit dem System integriert ist, kann die Konsistenz zwischen Konfiguration und Anwendung werkzeuggestützt aufrechterhalten werden. Dadurch wurden häufig auftretende Fehler, die durch inkonsistente Konfigurationen verursacht wurden, vollständig eliminiert. Weiterhin liefert das introspektive Konfigurationsmodell eine übersichtliche hierarchische Sicht auf diesen Aspekt des Systems und erhöht dadurch die Verständlichkeit des Gesamtsystems.

In einem zweiten Schritt wurden die zunächst nicht-introspektiven Handler auf das vorgestellte introspektive Programmiermodell portiert. Das gesamte System umfasst 390 Handler, die aufrufbare Services realisieren. Die visuelle Repräsentation dieser Services wird durch 300 HTML-Templates spezifiziert. Um diese Portierung durchzuführen wurden maßgeschneiderte Refactorings erstellt, die einen großen Teil der Umstellung automatisiert unterstützt haben. Als Ergebnis der Portierung sind die Platzhalter der Templates mit den in Handlern definierten Substitutionen durch das Werkzeug *Consistency Checker* integriert. Diese Integration erhöht einerseits die Verständlichkeit, da es nun möglich ist zwischen den Artefakten zu navigieren. Weiterhin ermöglicht Integration, die Konsistenz der Visualisierungsschicht sicherzustellen. Bestimmte Fehler, die sich ansonsten nur durch aufwändiges Testen aufspüren ließen, lassen sich somit werkzeuggestützt zur Entwicklungszeit zu finden. Ein weiterer positiver Effekt besteht darin, dass durch das Werkzeug nicht verwendete Substitutionen kenntlich gemacht werden und somit gefahrlos gelöscht werden können. Dies verhindert das so genannte *Huckepack-Syndrom*, bei welchem große Mengen toten Codes die Wartbarkeit eines Softwaresystems beeinträchtigen [RH06 S. 134].

Weiterhin wurde das in [We02] beschriebene Datenmodellierungsframework auf das in Abschnitt 7.4 vorgestellte introspektive Whitebox Framework portiert. Hierbei wurden zusätzlich

zum Framework auch das existierende Datenmodell und der gesamte Anwendungscode auf das introspektive Programmiermodell umgestellt. Das Gesamtsystem verwaltet 32 persistente Geschäftsobjekttypen mit 500 Eigenschaften und 100 Beziehungen. Einen Überblick über die Geschäftsobjekttypen der Anwendung gibt das vollständig in Anhang A dargestellte introspektive Datenmodell der Wissensmanagementanwendung. Auch diese Modellierungsperspektive erhöht die Verständlichkeit des Systems. Aufgrund des in Abschnitt 7.4.3 vorgestellten Programmiermodells ist das Datenmodell mit allen Anwendungsteilen, die darauf zugreifen, integriert. Damit ist gemeint, dass beispielsweise alle Zugriffe auf eine Eigenschaft eines Geschäftsobjekttyps werkzeuggestützt nachverfolgt werden können. Dadurch ist es möglich, die Konsistenz des Gesamtsystems zu überprüfen und sicherzustellen.

Unter Nutzung der in Abschnitt 7.1 dargestellten Prinzipien war es möglich, auf spezielle Kundenwünsche zugeschnittene Varianten des Wissensmanagementsystems zu erstellen. Beispielsweise wurde eine auf die speziellen Bedürfnisse von Patentanwälten zugeschnittene Version des Wissensmanagementsystems entwickelt und eingesetzt.

Eine weitere zunächst separate Entwicklung unterstützt das Erstellen von so genannten Weblogs [He04]. Dabei handelt es sich Webseiten, deren Inhalte sich häufig ändern und auf denen Einträge in umgekehrt chronologischer Reihenfolge aufgelistet werden. Diese Entwicklung wurde eigenständig kommerziell eingeständig eingesetzt und wurde anschließend in das Wissensmanagementsystem integriert.

7.7.2 Nicht-geplante Wiederverwendung

In diesem Abschnitt wird ein Szenario vorgestellt, in dem die Vorteile, die sich aus der Integration der Modellperspektiven ergeben, in besonderem Maße zum Tragen kommen. Das Szenario betrifft die Anpassung einer auf Toro basierten Webanwendung an neu formulierte Anforderungen. Wie bereits dargestellt, wurde auf Basis von Toro eine speziell auf die Bedürfnisse von Patentanwälten zugeschnittene Anwendung erstellt. Diese Anwendung soll typische Abläufe und Dokumente einer Anwaltskanzlei elektronisch repräsentieren und unterscheidet sich somit stark von dem existierenden Wissensmanagementsystem.

Eine Möglichkeit der Realisierung besteht darin, ein komplett neues System zu erstellen, welches genau die speziellen Anforderungen erfüllt. Eine weitere Möglichkeit besteht darin, das existierende Kernsystem mit dem in Anhang A dargestellten Datenmodell als Basis zu wählen und an die speziellen Anforderungen der Verwaltung von Patenten anzupassen. Hierbei besteht ein großes Wiederverwendungspotenzial, falls sich Konzepte aus beiden Domänen aufeinander abbilden lassen. So weisen Patentdokumente eine große konzeptuelle Ähnlichkeit mit der bestehenden Klasse `Document` auf. Zur Repräsentation von Mandanten eines Patentanwalts kann die Klasse `Unit` als Basis gewählt werden. Die Benutzerverwaltung des existierenden Wissensmanagementsystems kann beispielsweise völlig ohne Änderungen übernommen werden (dies betrifft die Klassen `Person`, `Group` und `Membership`). In diesem Änderungsszenario wird also ein existierendes System modifiziert, um neuen Anforderungen gerecht zu werden.

Wie in [En88] dargestellt, lassen sich durch Wiederverwendung Produktivität und Qualität bei der Entwicklung von Softwaresystemen steigern. Diese positiven Effekte sind von den mit einer Anpassung verbundenen Kosten abhängig. Im vorgestellten Fall handelt es sich um so genannte *nicht-geplante* Wiederverwendung, da bei der Entwicklung des ursprünglichen Systems nicht an eine spätere Wiederverwendung gedacht wurde. Dies erschwert im Allgemeinen die spätere Wiederverwendung und wirft eine Reihe von Problemen auf. Die Probleme betreffen zunächst die Frage, inwieweit sich die existierende Funktionalität mit der benötigten Funktionalität deckt. Eine große Hilfe bei der Beantwortung dieser Frage ist das introspektive Datenmodell. Lassen sich die Konzepte der Domäne des neu zu erstellenden Systems wie dargestellt zu großen Teilen auf Konzepte der existierenden Domäne abbilden, kann diese Frage positiv beantwortet werden und Wiederverwendung sollte in Betracht gezogen werden. Weitere Fragen in Bezug auf nicht-geplante Wiederverwendung betreffen allgemein die Verständlichkeit und Übersichtlichkeit des existierenden Codes. Diese Aspekte werden direkt durch die existierenden introspektiven Modelle adressiert und vereinfachen somit nicht-geplante Wiederverwendung.

Wie in Abschnitt 7.6.1 dargestellt, werden auf Basis von Toro entwickelte Webanwendungen durch die Artefakte Templates, Assets und Handler spezifiziert. Diese Artefakte müssen in einem Wiederverwendungsszenario auf die neuen Anforderungen angepasst werden. In einem wie in Abschnitt 7.6 dargestellten kundenzentrierten Entwicklungsprozess steht eine Veränderung der Templates am Beginn des Anpassungsprozesses. So werden in den existierenden Templates Eigenschaften von Geschäftsobjekten umbenannt, neu erstellt oder entfernt. Diese Änderungen beziehen sich konzeptuell betrachtet auf die Definition der Geschäftsobjekttypen. Aufgrund der Integration können sich aus Änderungen ergebende Inkonsistenzen durch den Consistency Checker zur Entwicklungszeit effizient gefunden und durch Änderungen am Datenmodell behoben werden. Nach einer Änderung des Datenmodells liefert eine introspektive Analyse des gesamten Projekts alle eventuell vorhandenen Inkonsistenzen. Der Anpassungsprozess besteht somit aus mehreren Iterationen dieser drei Schritte:

1. Änderungen an einem Template, welches ein Geschäftsobjekt visualisiert,
2. Konsistenzprüfung und Nachvollziehen der Änderungen am Datenmodell und
3. Konsistenzprüfung der gesamten Anwendung und Beheben von gefundenen Inkonsistenzen.

Dieses systematische und effiziente Vorgehen ist aufgrund der in Abbildung 103 dargestellten Integration aller Modellperspektiven möglich. Integration sowie die erhöhte Verständlichkeit und Übersichtlichkeit ermöglichen nicht-geplante Wiederverwendung und erhöhen dadurch die Produktivität des Softwareentwicklungsprozesses.

8 Schluss

Zum Schluss dieser Arbeit werden in Abschnitt 8.1 zunächst die Ergebnisse zusammengefasst und bewertet. In Abschnitt 8.2 werden die Vor- und Nachteile des in dieser Arbeit vorgestellten Ansatzes diskutiert. Anschließend gibt Abschnitt 8.3 einen Ausblick auf Fragen und möglichen Folgearbeiten, die sich aus dieser Arbeit ergeben.

8.1 Ergebnisse

In dieser Arbeit wurde ein alternativer Ansatz modellgetriebener Softwareentwicklung vorgestellt, welcher auf den etablierten und weit verbreiteten Basisabstraktionen objektorientierte Programmiersprachen und Frameworks basiert. Die spezielle Motivation für den vorgestellten Ansatz ergibt sich aus der Forderung nach *Integration* zwischen Modellierungsartefakten und darunter liegendem System. Es wurden drei Integrationsszenarien identifiziert. Das elementarste Szenario betrifft die Integration zwischen domänenspezifischer Sprache und Framework-Kern. Ein weiteres Szenario ergibt sich aus der Erkenntnis, dass einerseits domänenspezifische Abstraktionen zur Komplexitätsbeherrschung notwendig sind, andererseits wird für bestimmte Aspekte weiterhin die Mächtigkeit der Basisprogrammiersprache benötigt. Dies erfordert eine Integration von Modellierungsartefakten mit Konstrukten der Basisprogrammiersprache. Ein drittes Szenario folgt aus der Tatsache, dass Systeme im Allgemeinen durch mehrere domänenspezifische Sprachen für verschiedene Aspekte spezifiziert werden. Dies erfordert die Integration verschiedener domänenspezifischer Sprachen untereinander. Eine Realisierung modellgetriebener Softwareentwicklung muss also einerseits domänenspezifische Abstraktionen bereitstellen, andererseits ist eine Integration dieser domänenspezifischen Sprachen in das darunter liegende System von großer Bedeutung.

Es wurde aufgezeigt, dass existierende Ansätze zur modellgetriebenen Softwareentwicklung die vorgestellten Integrationsforderungen nicht erfüllen. Die Konsistenz zwischen Modellierungsartefakten und Artefakten der Basisprogrammiersprache muss hierbei manuell aufrechterhalten werden. Dies ist aufwändig und fehleranfällig.

Dem in dieser Arbeit vorgestellten Ansatz liegt die Beobachtung zugrunde, dass die Erweiterungspunkte eines objektorientierten Frameworks bereits domänenspezifische Abstraktionen realisieren. Der in dieser Arbeit verfolgte Ansatz besteht darin, diese Abstraktionen direkt aus dem annotierten Framework-Kern zu extrahieren und darauf aufbauend integrierte domänenspezifische Sprachen zu realisieren. Dieser Extraktionsprozess wird *Introspektion* genannt. Annotierte Frameworks, die diesen Prozess unterstützen, werden *introspektive Frameworks* genannt und ermöglichen die automatische Extraktion des Metamodells der vom Framework definierten domänenspezifischen Sprache. Dieser Ansatz heißt *introspektive modellgetriebene Softwareentwicklung*. In dieser Arbeit wurden die grundlegenden Konzepte introspektiver modellgetriebener Softwareentwicklung vorgestellt sowie eine Realisierung entwickelt, die die

Konstruktion introspektiver Frameworks ermöglicht. Die Realisierung basiert auf der Programmiersprache Java und erweitert die integrierte Entwicklungsumgebung Eclipse.

Es wurden zwei Arten von introspektiven Frameworks identifiziert, die sich in der Art und Weise der Realisierung von integrierten domänenspezifischen Sprachen unterscheiden: introspektive Blackbox und Whitebox Frameworks.

Introspektive Blackbox Frameworks realisieren domänenspezifische Sprachen in einer externen Form. Integration wird erreicht, da zu jedem Zeitpunkt eine Verbindung zwischen Modellelementen und korrespondierenden Elementen des Framework-Kerns erhalten bleibt. Dadurch werden inkonsistente Modelle bereits zur Entwicklungszeit gefunden und können anschließend korrigiert werden. Es wurde ein Metametamodell sowie Werkzeugunterstützung zur Entwicklung und Modellierung von introspektiven Blackbox Frameworks entworfen und realisiert.

Introspektive Whitebox Frameworks realisieren modellgetriebene Softwareentwicklung durch interne domänenspezifische Sprachen. Hierbei werden Modelle durch Quellcode der Basisprogrammiersprache repräsentiert, der einem definierten introspektiven Programmiermodell folgt. Aufgrund des introspektiven Programmiermodells ist es möglich, die repräsentierten Modelle zu extrahieren und auf einem hohen Abstraktionsniveau zu visualisieren und zu verändern. Es wurden introspektive Programmiermodelle identifiziert, die die Konstruktion komplexer introspektiver Whitebox Frameworks ermöglichen. Da es sich um eine interne domänenspezifische Sprache handelt, sind die Modelle introspektiver Whitebox Frameworks automatisch mit dem Framework-Kern integriert. Zusätzlich ermöglicht die interne Repräsentation die Integration von Modellen mit Anpassungsartefakten, die direkt unter Benutzung der Basisprogrammiersprache erstellt wurden. Weiterhin ist die Integration verschiedener domänenspezifischer Sprachen mit Mitteln der Basisprogrammiersprache möglich. Die vorgestellte Werkzeugunterstützung ermöglicht die Konstruktion und Modellierung introspektiver Whitebox Frameworks unter Benutzung des identifizierten Metametamodells.

In Kapitel 6 wurde die softwaretechnische Realisierung der entwickelten Werkzeugunterstützung beschrieben. Auf der Eclipse IDE basierte Lösung ermöglicht das Erstellen von speziellen Visualisierungen sowie von maßgeschneiderten introspektiven Frameworks, die über die vorgestellten generischen Metametamodelle hinausgehen

Um Kosten und Nutzen des Ansatzes der introspektiven modellgetriebenen Softwareentwicklung bewerten zu können, wurden im Rahmen dieser Arbeit mehrere introspektive Frameworks erstellt. Diese Frameworks bilden zusammen *Toro* – eine Plattform zur Entwicklung von Webanwendungen. In Kapitel 7 wird die Plattform sowie drei introspektive Whitebox Frameworks beschrieben. Die gesamte Plattform ist als introspektives Blackbox Framework realisiert und kann auf diese Art und Weise integriert konfiguriert werden.

Das erste vorgestellte introspektive Whitebox Framework realisiert multilinguale Nachrichten. In diesem einfachen Fall soll es Nicht-Programmierern ermöglicht werden, vom System verwendete Nachrichten in verschiedene Sprachen zu übersetzen. Klassischerweise wird diese Aufgabenstellung durch eine einfache externe domänenspezifische Sprache realisiert. Aufgrund der fehlenden Integration muss die Konsistenz aufwändig manuell aufrechterhalten werden. Es handelt sich hierbei um das zweite Integrationsszenario, welches sich auf die Integration zwischen domänenspezifischen Sprachen und händisch erstellten Quellcode der Basispro-

grammiersprache bezieht. Aufgrund der vorgestellten Realisierung als interne DSL durch ein introspektives Whitebox Framework ist die Integration gegeben und die Konsistenz kann mit Mitteln der Basisprogrammiersprache überprüft werden. Die Verständlichkeit für Nicht-Programmierer ist durch die introspektive Visualisierung gegeben.

Das zweite vorgestellte Framework realisiert eine objekt-relationale Persistenzabbildung. Hierbei werden Geschäftsobjekttypen durch introspektive Metamodell-Instanzen spezifiziert. Die Modellierung kann vollständig durch die introspektive domänenspezifische Sprache des Frameworks erfolgen. Zur besseren Verständlichkeit wurde eine maßgeschneiderte Visualisierung in Form von UML-Klassendiagrammen entwickelt. Die Modelle sind vollständig mit manuell erstelltem Quellcode, welcher auf die Geschäftsobjekte zugreift, integriert.

Zuletzt wurde ein introspektives Webvisualisierungs-Framework vorgestellt. Hierbei handelt es sich um eine Problemdomäne, welche in bestimmten Aspekten deklarativ durch eine domänenspezifische Sprache modelliert werden kann; andere Aspekte benötigen jedoch die volle Ausdrucksmächtigkeit der Basisprogrammiersprache. Dies lässt sich einfach durch ein introspektives Whitebox Framework realisierten. Die deklarativen Aspekte werden durch ein introspektives Programmiermodell spezifiziert und können demzufolge durch eine domänenspezifische Sprache repräsentiert werden. Zur Spezifikation der verbleibenden Aspekte kann die volle Ausdrucksmächtigkeit der Basisprogrammiersprache genutzt werden. Eine Integration beider Aspekte ist gegeben. Eine weitere Anforderung an Webvisualisierungs-Frameworks ergibt sich aus der Notwendigkeit, Präsentationsaspekte durch HTML-Templates zu spezifizieren. Eine Integration von Templates und steuerndem Quellcode ist wünschenswert. Das vorgestellte Webvisualisierungs-Framework ermöglicht dies aufgrund der introspektiven Realisierung. Es wurde ein in die Entwicklungsumgebung integriertes Werkzeug entwickelt, welches beide Artefakte miteinander integriert und Inkonsistenzen feststellt sowie Vorschläge zur Behebung macht.

Um die Anwendbarkeit der introspektiven Webplattform Toro bewerten zu können, wurde eine existierende industrielle Wissensmanagementanwendung auf Toro portiert. Die als Teil von Toro vorliegenden introspektiven Modellierungssichten erhöhen einerseits die Verständlichkeit des Systems. Andererseits ermöglicht die Integration aller Modellierungssichten die werkzeugunterstützte Sicherstellung der Konsistenz des Gesamtsystems. Aufgrund dieser Eigenschaften ist es effizient möglich, existierende Anwendungen an neue Anforderungen anzupassen und somit wiederzuverwenden.

8.2 Vor- und Nachteile introspektiver modellgetriebener Softwareentwicklung

Zusammenfassend ermöglicht introspektive modellgetriebene Softwareentwicklung die Konstruktion von Systemen, die sich durch domänenspezifische Sprachen auf einem hohen Abstraktionsniveau spezifizieren lassen. Die domänenspezifischen Sprachen sind mit dem darunter liegenden System integriert. Dadurch ist es möglich, die Konsistenz zwischen den domänenspezifischen Abstraktionen und dem darunter liegenden System sicherzustellen. Da Modelle lediglich eine transiente Sicht auf Implementierungsartefakte sind, wird Roundtrip-Engineering ermöglicht.

Verglichen mit existierenden Ansätzen ist introspektive modellgetriebene Softwareentwicklung ein leichtgewichtiger Ansatz zur modellgetriebenen Softwareentwicklung. Zur Entwicklung von Frameworks mit Modellierungsunterstützung ist lediglich ein Verständnis der vorgestellten Metametamodelle notwendig. Darauf aufbauend können integrierte domänenspezifische Sprachen effizient realisiert werden. Der Ansatz passt sich nahtlos in etablierte codezentrierte Entwicklungsprozesse ein. Die entwickelte Werkzeugunterstützung erweitert die weit verbreitete Entwicklungsumgebung Eclipse und passt sich somit ebenfalls gut in existierende Prozesse ein.

Den genannten positiven Eigenschaften introspektiver modellgetriebener Softwareentwicklung stehen die folgenden Einschränkungen gegenüber. Aufgrund der starken Integration mit den darunter liegenden Frameworks wird in diesem Ansatz durch Modellierung keine Technologieabhängigkeit erreicht, wie sie beispielsweise beim Ansatz der Model-Driven Architecture angestrebt wird. Die erstellten Modelle hängen in starkem Maße mit den darunter liegenden Technologien zusammen.

Die in dieser Arbeit erstellte Realisierung introspektiver modellgetriebener Softwareentwicklung wurde auf Basis der Programmiersprache Java und der Entwicklungsumgebung Eclipse entwickelt und ist somit vom Vorhandensein dieser Technologien abhängig. Allerdings ist die vorgestellte konzeptuelle Idee für beliebige Basisprogrammiersprachen anwendbar.

Introspektive modellgetriebene Softwareentwicklung basiert auf introspektiven Frameworks und ist somit nicht auf Basis der existierenden Framework-Infrastruktur möglich.

Zur Modellierung introspektiver Whitebox Frameworks müssen Teile des Quellcodes vorliegen.

8.3 Ausblick

Aufbauend auf der vorliegenden Arbeit sind Weiterentwicklungen auf zwei verschiedenen Ebenen denkbar. Einerseits ist es denkbar, den Ansatz der introspektiven modellgetriebenen Softwareentwicklung in weiteren Domänen zur Anwendung zu bringen und dadurch weiter zu evaluieren. Hierbei sind prinzipiell alle Anwendungsdomänen denkbar, in denen modellgetriebene Softwareentwicklung sinnvoll ist. Die Vorteile des introspektiven Ansatzes lassen sich nur realisieren, wenn entweder existierende Frameworks zu introspektiven Frameworks umgewandelt werden, oder entsprechende Frameworks von Grund auf neu entwickelt werden.

Eine weitere Ebene auf dieser Arbeit aufsetzender Tätigkeiten kann auf der existierenden introspektiven Webplattform Toro aufsetzen und diese weiterentwickeln.

A Beispiel: Datenmodell

In diesem Abschnitt wird das Datenmodell des auf Basis von Toro entwickelten Wissensmanagementsystems vorgestellt. Die Geschäftsobjekttypen wurden mit dem in Abschnitt 7.4 vorgestellten introspektiven Persistenzframework modelliert. Dies ermöglicht die introspektive Extraktion des Modells und die anschließende Visualisierung in Form von UML-Klassendiagrammen. Das gesamte Datenmodell umfasst 35 Klassen und wurde zur besseren Übersichtlichkeit in fünf Bereiche aufgespalten, die in den Abbildung 104 bis 108 dargestellten sind.

Es existieren Beziehungen, die über die Grenzen der dargestellten Bereiche hinwegreichen. Dies wird durch „Platzhalter-Klassen“ repräsentiert, die durch den Nachsatz `(P)` gekennzeichnet sind. So existieren beispielsweise in Abbildung 104 zahlreiche Beziehungen zur Klasse Group, die vollständig in Abbildung 105 dargestellt ist. Es sind jeweils nur die Beziehungen zwischen Klassen eines Diagramms und nicht enthaltenen Platzhalter-Klassen dargestellt, für die eine explizit benannte Rolle zur Navigation existiert.

In Abbildung 104 sind die für die Dokumentverwaltung zuständigen Klassen des Systems dargestellt. Kernklassen sind hierbei die Klassen `Document` und `Directory`. Die Klassen `Concept` ermöglicht es, ein Begriffsnetzwerk aufzubauen, dem Dokumente zugeordnet werden können.

In Abbildung 105 sind die Klassen dargestellt, die mit der Benutzerverwaltung des Systems vertraut sind. Personen sind durch Objekte vom Typ `Membership` Gruppen zugeordnet.

In Abbildung 111 sind generische Dienste dargestellt, die für beliebige Geschäftsobjekttypen zur Verfügung stehen. Die Klasse `AnyAsset` repräsentiert beliebige Geschäftsobjekttypen. So können beispielsweise durch Objekte vom Type `AssetTask` beliebigen Objekten Aufgaben zugeordnet werden.

In den Abbildungen 107 und 108 sind Klassen dargestellt, die ein Weblog-System und ein Wettbewerber-Informationssystem realisieren.

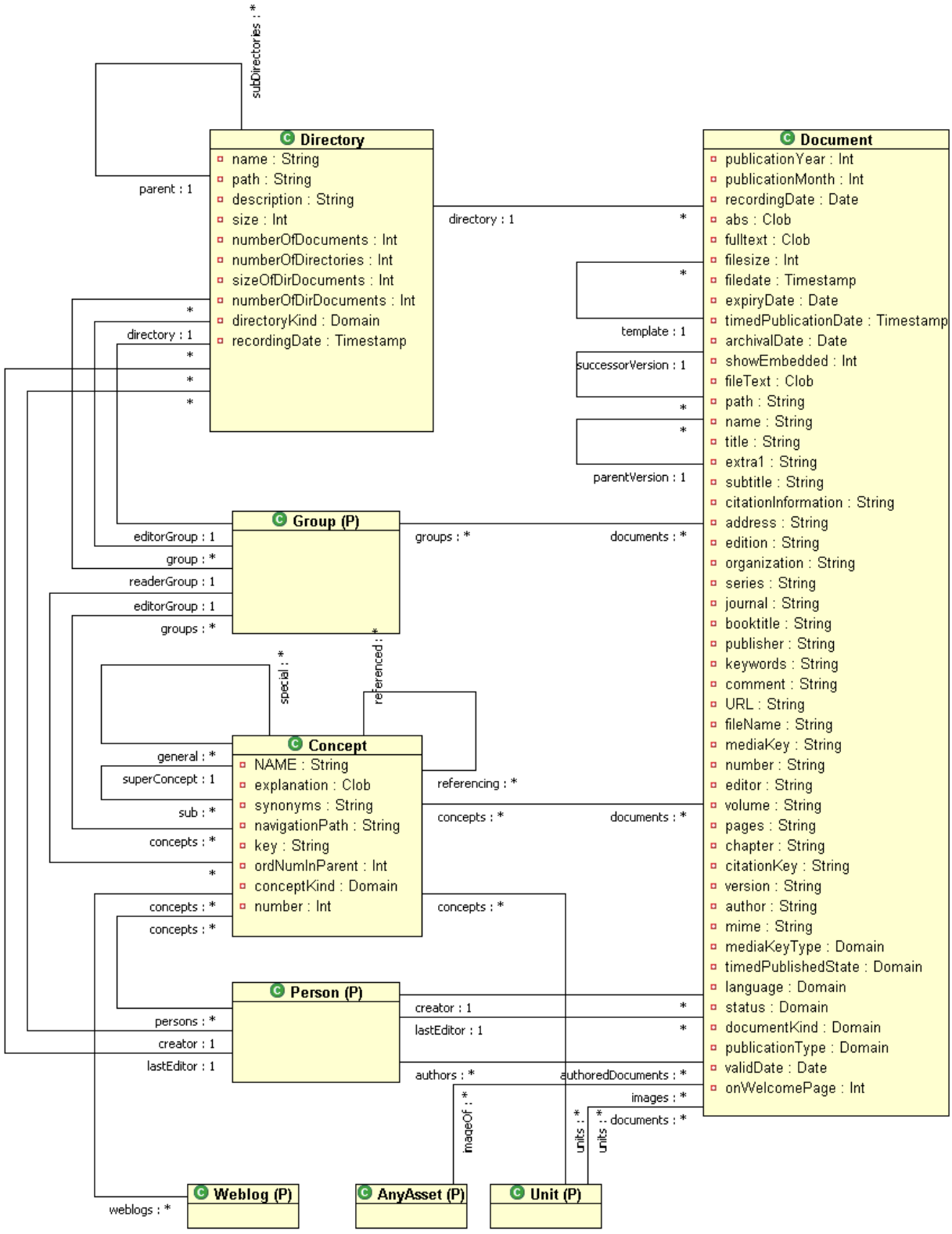


Abbildung 104: Datenmodell – Dokumentverwaltung

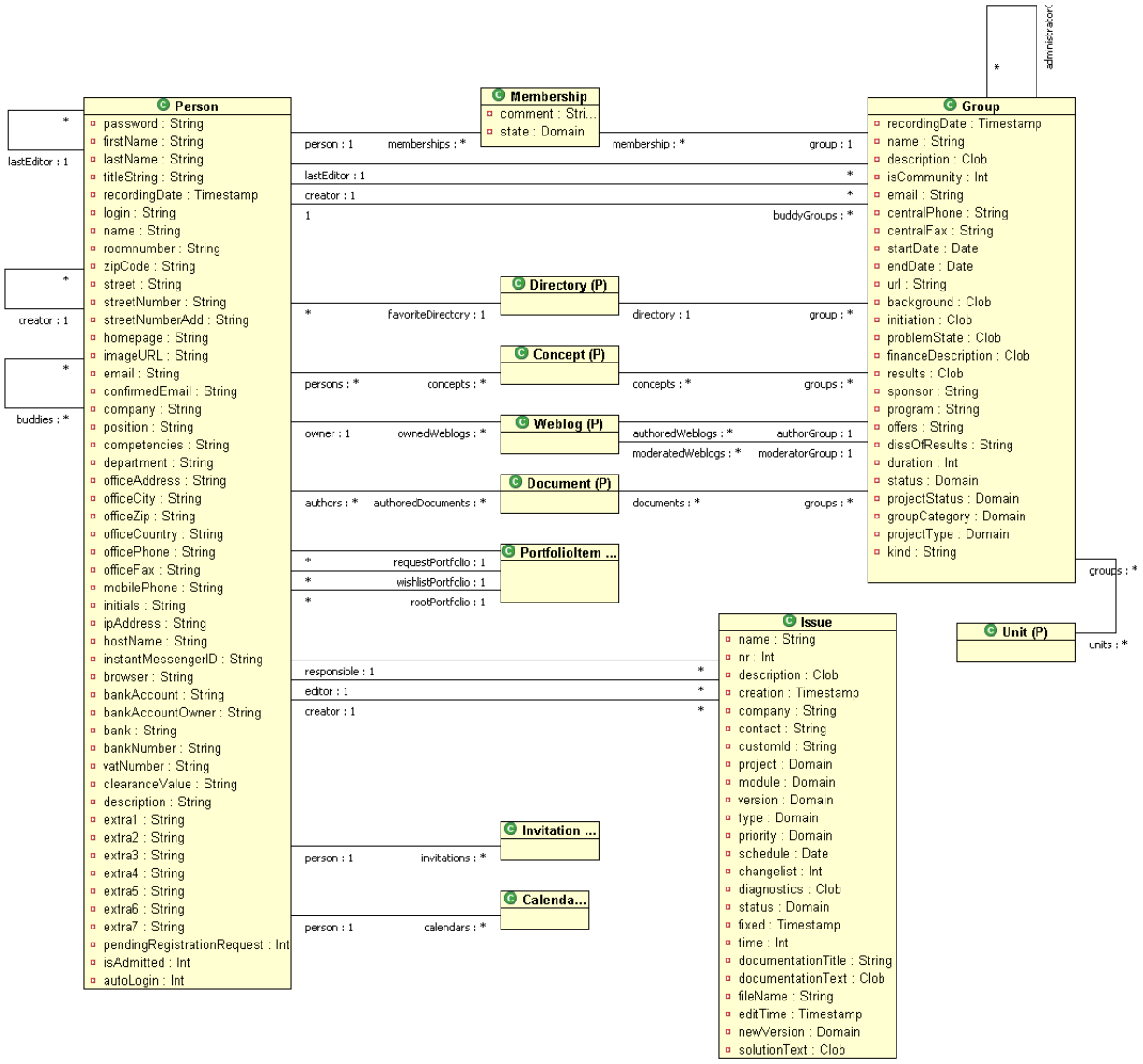


Abbildung 105: Datenmodell – Benutzerverwaltung

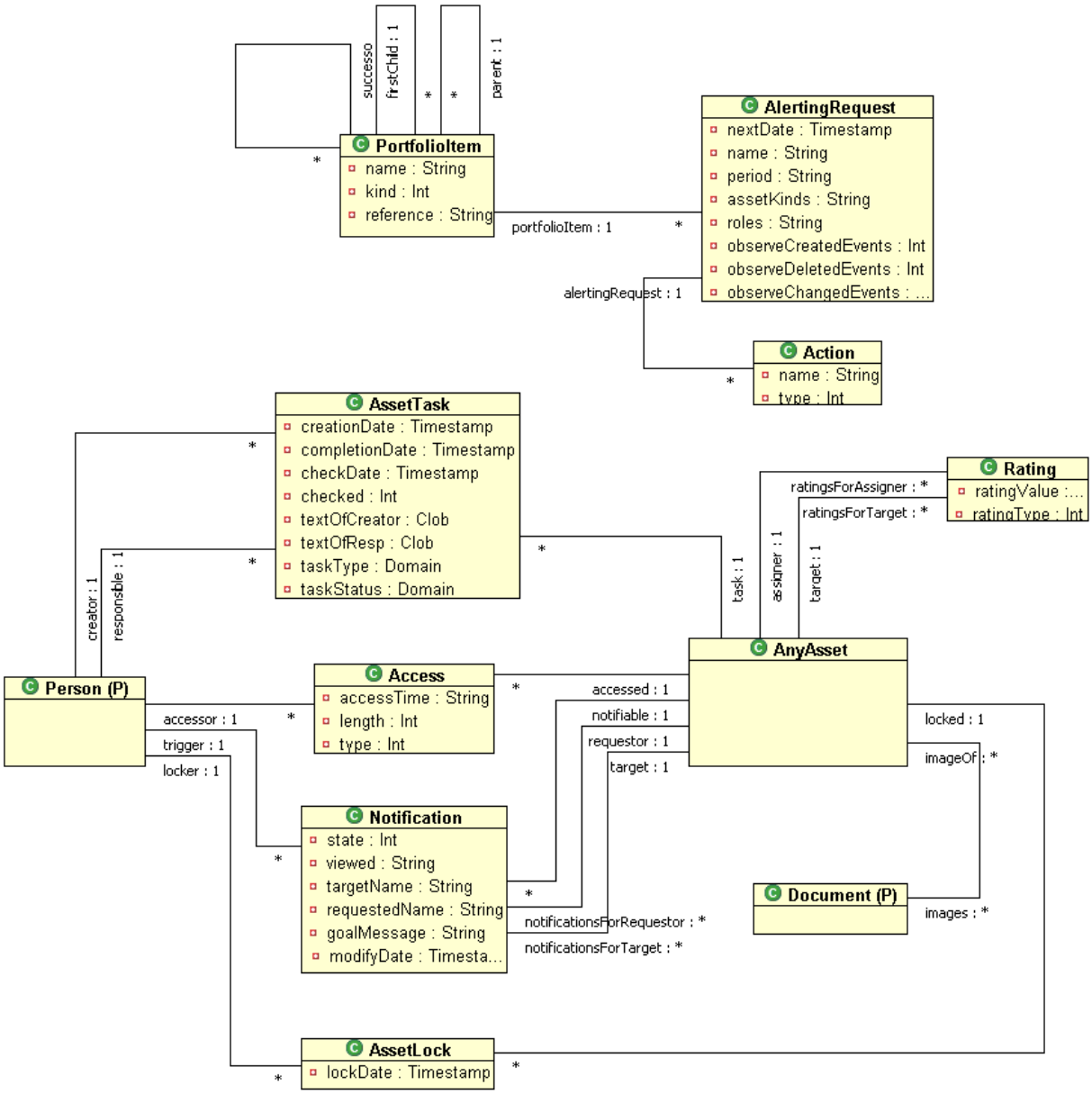


Abbildung 106: Datenmodell – Generische Services

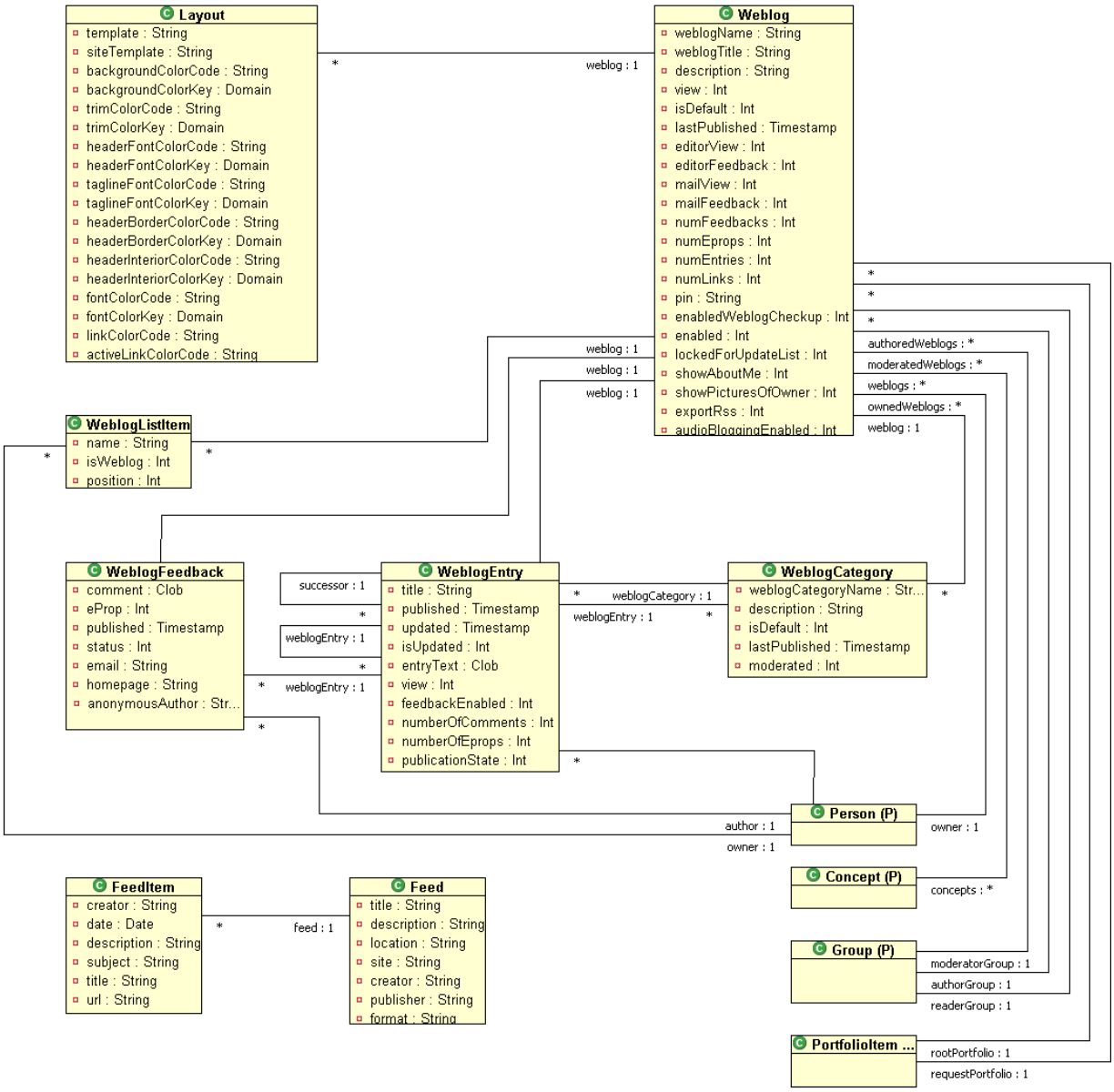


Abbildung 107: Datenmodell – Weblogsystem

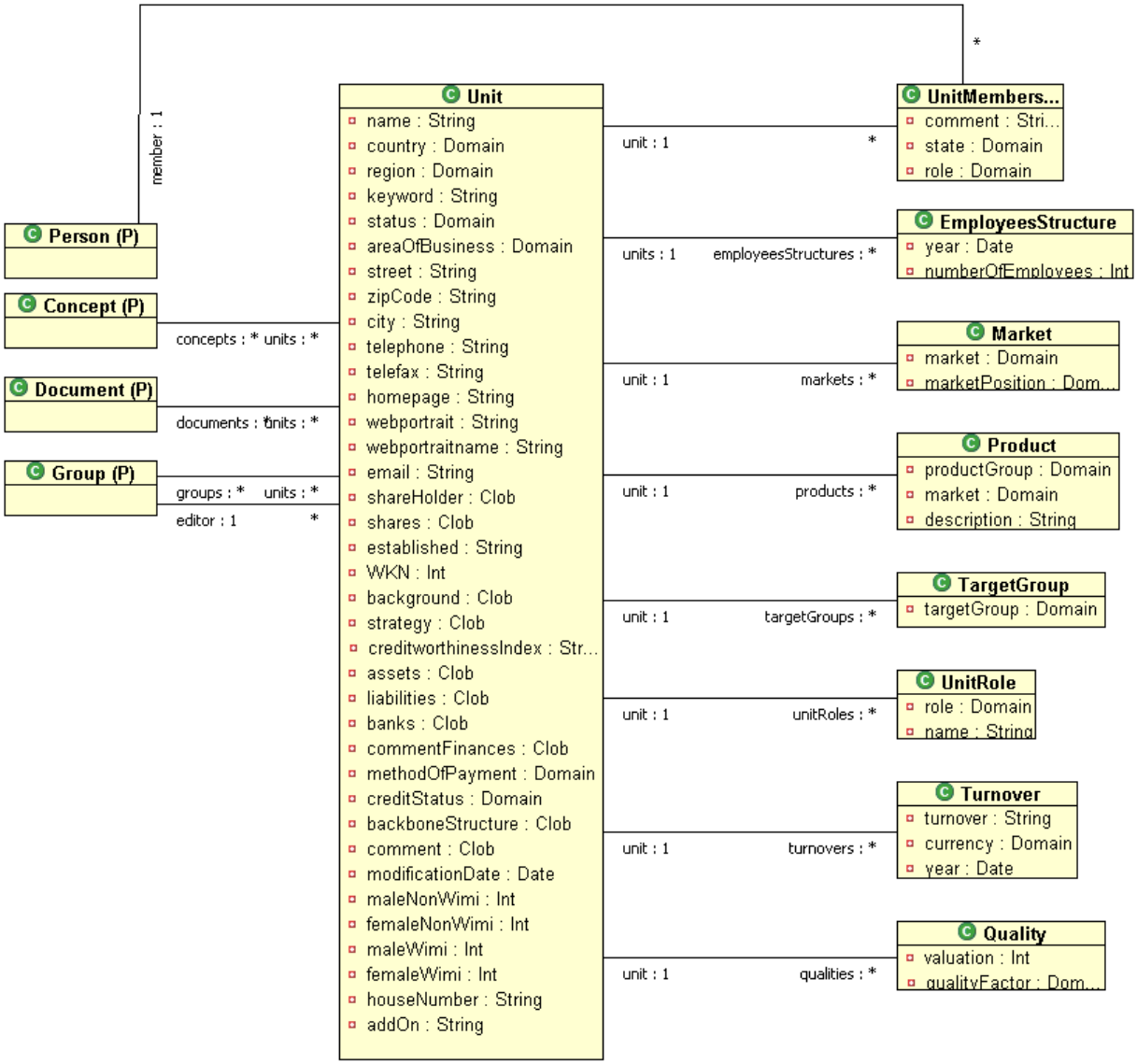


Abbildung 108: Datenmodell – Wettbewerber-Informationssystem

B Beispiel: Konfigurationsmodell

Wie dargestellt, ist Toro als Ganzes als introspektives Blackbox Framework realisiert und kann auf diese Art und Weise an konkrete Kundenumgebungen angepasst werden. In den Abbildungen 109 bis 110 ist ein vollständiges Konfigurationsmodell des auf Basis von Toro entwickelten Wissensmanagementsystems dargestellt. Es handelt sich hierbei um Screenshots des in Abschnitt 4.3 detailliert vorgestellten Werkzeugs „Konfigurator“. Die benutzte Symbolik entspricht der in Abbildung „Vollständiges Metametamodell“ auf Seite 67 dargestellten Ikonographie.

Wie bereits in Abschnitt 7.7 dargestellt, umfasst das Konfigurationsmodell 130 konfigurierte Klassen mit 200 Eigenschaften und 120 Beziehungen.

Abbildung 109: Konfigurationsmodell - Abschnitt 1

Abbildung 110: Konfigurationsmodell - Abschnitt 2

Abbildung 111: Konfigurationsmodell - Abschnitt 3

Abbildung 112: Konfigurationsmodell - Abschnitt 4

Literaturverzeichnis

[AC06] Michal Antkiewicz und Krzysztof Czarnecki, *Eclipse Workbench Part Interaction FSML*, Electrical & Computer Engineering Department, University of Waterloo 2006.

[Ad06] Adobe Systems Incorporated, *Dreamweaver 8*, `http://www.macromedia.com/software/dreamweaver/` aufgerufen am 2007-02-06

[Al03] Manoli Albert, Vicente Pelechano, Joan Fons, Marta Ruiz und Oscar Pastor, *Implementing UML Association, Aggregation, and Composition. A Particular Interpretation Based on a Multidimensional Framework*, Lecture Notes in Computer Science, vol. 2681, pp. 143 - 158, 2003.

[An06] Michal Antkiewicz, *Round-Trip Engineering of Framework-Based Software using Framework-Specific Modeling Languages*, Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), 2006.

[ASU86] Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullman, *Compilers*, Addison Wesley, 1986.

[Ba60] J.W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden und M. Woodger, *Revised Report on the Algorithmic Language ALGOL 60*, Communications of the ACM, vol. 3, pp. 299-314, 1960.

[Ba78] John Backus, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Communications of the ACM, vol. 21, pp. 613-641, 1978.

[Be03] Hans Bergsten, *JavaServer Pages*, O'Reilly Media, 2003.

[BFF96] T. Berners-Lee, R. Fielding und H. Frystyk., *Hypertext Transfer Protocol — HTTP/1.0*, Internet RFC 1945, May 1996.

[BG04] Matthias Book und Volker Gruhn, *Modeling Web-Based Dialog Flows for Automatic Dialog Control*, Proceedings of the 19th IEEE international conference on Automated software engineering, 2004.

[BK04a] Christian Bauer and Gavin King, Hibernate in Action, Manning Publications, 2004.

[BK04b] Katharina Brendebach and Katy Kirsche, Introspektive Konfiguration eines Portalsystems für das Wissensmanagement, Systementwicklungsprojekt, Technische Universität München, 2004.

[BL04] Joshua Bloch, *JSR 175: A Metadata Facility for the Java Programming Language*, `http://www.jcp.org/en/jsr/detail?id=175` aufgerufen am 2006-01-02

[BM06] Thomas Büchner and Florian Matthes, Introspective Model-Driven Development, Proc. of Third European Workshop on Software Architectures (EWSA 2006), LNCS 4344, Springer, Nantes, France, 2006.

[Bo94] Grady Booch, *Objektorientierte Analyse und Design*, Addison-Wesley, 1994.

[Bo99] Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson und Mohamed E. Fayad, *Framework problems and experiences*, in Building Application Frameworks, Douglas C. Schmidt Mohamed Fayad, Ralph E. Johnson, Ed.: Wiley Computer Publishing, 1999, pp. 55-82.

[Bu03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick und Timothy J. Grose, *Eclipse Modeling Framework*, Addison-Wesley Professional, 2003.

[Bu96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons Ltd., 1996.

[CC90] Elliot J. Chikofsky und James H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy.*, IEEE Software, vol. 7, 1990.

[CD01] David Czarnecki und Andy Deitsch, *Java Internationalization*, O'Reilly Media, 2001.

[Ce03] Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai und Maristella Matera, *Designing Data-Intensive Web Applications*, Morgan Kaufman Publishers, 2003.

[CFB00] Stefano Ceri, Piero Fraternali und Aldo Bongio, *Web Modeling Language (WebML): A Modeling Language for Designing Web Sites*, Proceedings of the 9th international World Wide Web conference on Computer networks, 2000.

[CH03] Krzysztof Czarnecki und Simon Helsen, *Classification of Model Transformation Approaches*, OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, USA, 2003.

[CL04] Tony Clark, Andy Evans, Paul Sammut und James Willans, *Applied Metamodelling: A Foundation for Language Driven Development*, `http://www.xactium.com` aufgerufen am 2005-11-01

[Co70] E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, vol. 13, pp. 377-387, 1970.

[Co99] Jim Conallen, *Modeling Web application architectures with UML*, Communications of the ACM, vol. 42, pp. 63-70, 1999.

[CTM03] Stuart M. Charters, Nigel Thomas und Malcolm Munro, *The end of the line for Software Visualization?*, VISSOFT 2003: 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis, Amsterdam, September 2003.

[Da87] C. J. Date, *A guide to the SQL standard: A user's guide to the standard relational language SQL*, Addison-Wesley Pub. Co, 1987.

[De97] Arie van Deursen, *Domain-specific languages versus object-oriented frameworks: A financial engineering case study*, Smalltalk and Java in Industry and Academia, Ilmenau Technical University, 1997.

[Di04] Sergey Dimitriev, *Language Oriented Programming: The Next Programming Paradigm*, JetBrains 2004.

[DK06] Linda DeMichiel und Michael Keith, *JSR 220: Enterprise JavaBeans 3.0*, `http://jcp.org/en/jsr/detail?id=220` aufgerufen am 2007-07-26

[DKV00] Arie van Deursen, Paul Klint und Joost Visser, *Domain-specific Languages: An Annotated Bibliography*, ACM SIGPLAN Notices, vol. 35, pp. 26-36, 2000.

[DLH07] Douglas K. van Duyne, James A. Landay und Jason I. Hong, *The Design of Sites*, Prentice Hall, 2007.

[Ec05] Eclipse Foundation, *Eclipse JDT - API Doc*, `http://help.eclipse.org/help31/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html` aufgerufen am 2006-03-14

[Ec06a] Bruce Eckel, *Thinking in Java*, Prentice Hall, 2006.

[Ec06b] Eclipse Foundation, *Eclipse*, `http://www.eclipse.org` aufgerufen am 2006-01-11

[Ec06c] Eclipse Foundation, *Graphical Editing Framework*, `http://www.eclipse.org/gef/` aufgerufen am 2006-06-02

[Ec95] Wayne W. Eckerson, *Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications.*, Open Information Systems, vol. 10 no. 1, 1995.

[En88] Albert Endres, *Software-Wiederverwendung: Ziele, Wege und Erfahrungen*, Informatik Spektrum, vol. 11, pp. 85-95, 1988.

[ER03] Albert Endres und Dieter Rombach, *A Handbook of Software and Systems Engineering*, Pearson Education, 2003.

[Fa85] Richard E. Fairley, *Software Engineering Concepts*, McGraw-Hill Companies, 1985.

[Fi00] Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000.

[FL02] Marc Fleury und Juha Lindfors, *JMX: Managing J2EE Applications with Java Management Extensions*, Sams, 2002.

[Fo04a] Martin Fowler, *Inversion of Control Containers and the Dependency Injection Pattern*, `http://www.martinfowler.com/articles/injection.html` aufgerufen am 2005-12-29

[Fo04b] Martin Fowler, *PostIntelliJ*, `http://www.martinfowler.com/bliki/PostIntelliJ.html` aufgerufen am 2006-01-11

[Fo05a] Martin Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, `http://www.martinfowler.com/articles/languageWorkbench.html` aufgerufen am 2005-11-23

[Fo05b] Neal Ford, *Language Workbench Sample Ported to Java*, `http://memeagora.blogspot.com/2005/07/language-workbench-sample-ported-to.html` zugegriffen am 2007/07/11

[Fo99] Martin Fowler, Kent Beck, John Brant, William Opdyke und Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

[FPR01] Marcus Fontoura, Wolfgang Pree und Bernhard Rumpe, *The UML Profile for Framework Architectures*, Addison-Wesley Professional, 2001.

[Fr03] David S. Frankel, *Model Driven Architecture - Applying MDA to Enterprise Computing*, Wiley Publishing, Inc., 2003.

[Fr99] Garry Froehlich, H. James Hoover, Ling Liu und Paul Sorenson, *Designing Object-Oriented Frameworks*, in Handbook of Object Technology, Saba Zamir, Ed.: CRC Press, 1999.

[FSJ99] Mohamed E. Fayad, Douglas Schmidt und Ralph E. Johnson, *Building Application Frameworks*, Wiley, 1999.

[Fu97] Mark L. Fussell, *Foundations of Object-Relational Mapping*, `http://www.chimu.com/publications/objectRelational/objectRelational.pdf` aufgerufen am 2006-04-21

[Ga95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[GH05] Ralf Gitzel und Tobias Hildenbrand, *A Taxonomy of Metamodel Hierarchies*, Universität Mannheim, Lehrstuhl für Wirtschaftsinformatik III 2005.

[Go05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha, *The Java Language Specification*, Addison-Wesley, 2005.

[Gr04] Jack Greenfield, Keith Short, Steve Cook und Stuart Kent, *Software Factories*, Wiley Publishing, Inc., 2004.

[HB99] Brian Henderson-Sellers und Franck Barbier, *Black and White Diamonds*, Lecture Notes in Computer Science, vol. 1723, pp. 550 - 565, 1999.

[He04] Susan C. Herring, Lois Ann Scheidt, Sabrina Bonus und Elijah Wright, *Bridging the Gap: A Genre Analysis of Weblogs*, Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), 2004.

[Hi05] Martin Hitz, Gerti Kappel, Elisabeth Kapsheimer und Werner Retschitzegger, *UML@Work*, dpunkt.verlag, 2005.

[Hu01] Jason Hunter, *Java Servlet Programming*, O'Reilly Media, 2001.

[Hu96] Paul Hudak, *Building Domain-Specific Embedded Languages*, ACM Computing Surveys, vol. 28, pp. 196, 1996.

[In07] Interface21, *Spring Framework*, `http://www.springframework.org/` aufgerufen am 2007-06-29

[Je06] Jetbrains, *IntelliJ IDEA*, `http://www.jetbrains.com/idea/` aufgerufen am 2006-11-01

[JF88] Ralph E. Johnson und Brian Foote, *Designing reusable classes*, Journal of Object-oriented Programming, vol. 1(2), pp. 22-35, 1988.

[JR03] David Jordan und Craig Russell, *Java Data Objects*, O'Reilly Media, 2003.

[Kä05] Stefan Käck, *Introspektive Techniken zur Sicherung der Konsistenz zwischen Webpräsentationsvorlagen und Anwendungsdiensten*, Diplomarbeit, Technische Universität München, 2005.

[KE06] Alfons Kemper und André Eickler, *Datenbanksysteme*, Oldenbourg Verlag München Wien, 2006.

[KM05] Gregor Kiczales und Mira Mezini, *Separation of Concerns with Procedures, Annotations, Advice and Pointcuts*, ECOOP 2005 - Object-Oriented Programming, Glasgow, 2005.

[Ko01] Kalle Korhonen, *Motivation and hypothesis for comparison between component framework and DSL paradigms*, OOPSLA 2001 - Workshop on Domain-Specific Visual Languages, 2001.

[KP88] Glenn E. Krasner und Stephen T. Pope, *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, vol. 1, pp. 26 - 49, 1988.

[Le00] Alexis Leon, *A Guide to Software Configuration Management*, Artech House Computing Library, 2000.

[Ma87] Pattie Maes, *Concepts and Experiments in Computational Reflection*, Conference proceedings on Object-oriented programming systems, languages and applications, Orlando, Florida, United States, 1987.

[Mc98] Glen McCluskey, *Using Java Reflection*, `http://java.sun.com/developer/technicalArticles/ALT/Reflection/` aufgerufen am 2006-06-04

[Mi56] George A. Miller, *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*, The Psychological Review, vol. 1963, pp. 81 - 97, 1956.

[MKF06] Gail C. Murphy, Mik Kersten und Leah Findlater, *How Are Java Software Developers Using the Eclipse IDE?*, IEEE Software, vol. 23, pp. 76-83, 2006.

[ML01] Marcus Eduardo Markiewicz und Carlos J.P. de Lucena, *Object Oriented Framework Development*, ACM Crossroads Xrds7-4, 2001.

[My07] Adam Myatt, *Pro NetBeans IDE 5.5 Enterprise Edition*, Apress, 2007.

[Ob04a] Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification*, ptc/04-10-15, 2004

[Ob04b] Object Management Group, *Unified Modeling Language: Infrastructure*, ptc/04-10-14, 2004

[Ob05a] Object Management Group, *Unified Modeling Language: Superstructure*, formal/05-07-04, 2005

[Ob05b] Object Management Group, *MOF 2.0/XMI Mapping Specification, v2.1*, formal/05-09-01, 2005

[Ob06] Object Management Group, *Object Constraint Language*, formal/06-05-01, 2006

[Op06] openArchitectureWare Group, *openArchitectureWare*, `http://www.eclipse.org/gmt/oaw/` aufgerufen am 2006-02-01

[Pa02] Richard Pawson, *Naked Objects*, IEEE Software, vol. 19, pp. 81-83, 2002.

[Pa04] Terence John Parr, *Enforcing Strict Model-View Separation in Template Engines*, Proceedings of the 13th international conference on World Wide Web, New York, NY, USA, 2004.

[PBK07] Torsten Posch, Klaus Birken und Michael Gerdom, *Basiswissen Softwarearchitektur*, dpunkt.verlag, 2007.

[PBS93] Blaine A. Price, Ronald M. Baecker und Ian S. Small, *A Principled Taxonomy of Software Visualization*, Journal of Visual Languages and Computing, vol. 4, pp. 211-266, 1993.

[Pi05] Harry Pierson, *Code is Model*, `https://blogs.msdn.com/devhawk/archive/2005/10/05/477529.aspx` aufgerufen am 2006-01-03

[PQ95] T. J. Parr und R. W. Quong, ANTLR: *A Predicated-LL(k) Parser Generator*, Software - Practice & Experience, vol. 25, pp. 789 - 810, 1995.

[Pr97a] Wolfgang Pree, *Essential Framework Design Patterns*, Object Magazine, vol. 7, pp. 34-37, 1997.

[Pr97b] Wolfgang Pree, *Komponentenbasierte Softwareentwicklung*, dpunkt Verlag, 1997.

[RC93] Gruia-Catalin Roman und Kenneth C. Cox, *A Taxonomy of Program Visualization Systems*, IEEE Computer, vol. 26, pp. 11-24, 1993.

[Re05] Steven P. Reiss, *The Paradox of Software Visualization*, Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, Budapest, Hungary, 2005.

[RH06] Ralf Reussner and Wilhelm Hasselbring, Handbuch der Software-Architektur, dpunkt.verlag, 2006.

[Ri06] Chris Richardson, *Untangling Enterprise Java*, ACM Queue, vol. 4, pp. 36 - 44, 2006.

[Se05] Robert W. Sebesta, *Concepts of Programming Languages*, Addison Wesley, 2005.

[Si95] Charles Simonyi, *The Death Of Computer Languages, The Birth Of Intentional Programming*, NATO Science Committee Conference, 1995.

[SK04] Jonathan Sprinkle und Gabor Karsai, *A Domain-Specific Visual Language For Domain Model Evolutio*n, Journal of Visual Languages and Computing, vol. 15, 2004.

[St73] Herbert Stachowiak, *Allgemeine Modelltheorie*, Springer-Verlag, 1973.

[Su99a] Sun Microsystems, *Reflection*, `http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html` aufgerufen am 2007-05-02

[SV05] Thomas Stahl und Markus Völter, *Modellgetriebene Softwareentwicklung*, dpunkt.verlag, 2005.

[TC02] David Thomas und Ward Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 2002.

[TH05] Dave Thomas und David Hansson, *Agile Web Development with Rails: A Pragmatic Guide*, Pragmatic Bookshelf, 2005.

[Wa94] M. P. Ward, *Language Oriented Programming, Software - Concepts and Tools*, vol. 15, pp. 147-161, 1994.

[We02] Holm Wegner, *Analyse und objektorientierter Entwurf eines integrierten Portalsystems für das Wissensmanagement*, Dissertation, Technische Universität Hamburg-Harburg, 2002.

[We89] Peter Wegner, *Introduction to programming language paradigms*, ACM Computing Surveys, vol. 21, pp. 253-258, 1989.