Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Svend-Anjes Pahl

Entwicklung einer domänenspezifischen Sprache
zur Modellierung und Validierung eines Architekturentwurfes
nach den Regeln der Quasar-Standardarchitektur

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Svend-Anjes Pahl

Entwicklung einer domänenspezifischen Sprache zur Modellierung und Validierung eines Architekturentwurfes nach den Regeln der Quasar-Standardarchitektur

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Michael Neitzke

Abgegeben am 8. August 2011

Svend-Anjes Pahl

Thema der Bachelorarbeit

Entwicklung einer domänenspezifischen Sprache zur Modellierung und Validierung eines Architekturentwurfes nach den Regeln der Quasar-Standardarchitektur

Stichworte

domänenspezifische Sprache, DSL, Quasar, Standardarchitektur, Komponenten, MPS

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Entwicklung einer domänenspezifischen Sprache zur Spezifikation von Architekturen für Informationssysteme nach der Quasar standardarchitektur von Capgemini (ehemals sd&m). Im Gegensatz zu bisher verbreiteten Spezifikationssprachen, welche ausschließlich die Möglichkeit zur Schnittstellen-Spezifikation bieten, ist es mit der hier vorgestellten DSL möglich, Architekturen auf Komponentenebene zu planen und zu entwickeln, welches neue Möglichkeiten für den Spezifikations- und Entwicklungsprozess eröffnet.

Svend-Anjes Pahl

Title of the paper

Development of a domain specific language for modeling and validation of an architecture draft according to the rules of the Quasar standard architecture

Keywords

domain specific language, DSL, Quasar, standard architecture, components, MPS

Abstract

This paper is focussed on the development of a domain specific language for the purpose of specifying a draft for information systems according to the rules of the Quasar standard architecture by Capgemini (formerly sd&m). In contrast to common specification languages which only provide the possibility for interface specification, the presented DSL allows planning and developing software architectures on component level. This offers new opportunities for the specification and development process.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

Diese Arbeit befasst sich mit der Entwicklung einer domänenspezifischen Sprache (engl. *domain specific language*, kurz: *DSL*) zur Spezifikation von Architekturen für Informationssysteme nach der Quasar-Standardarchitektur von Capgemini (ehemals sd&m). Im Gegensatz zu bisher verbreiteten Spezifikationssprachen, welche ausschließlich die Möglichkeit zur Schnittstellen-Spezifikation bieten, ist es mit der hier vorgestellten DSL möglich, Architekturen auf Komponentenebene zu planen und zu entwickeln, welches neue Möglichkeiten für den Spezifikations- und Entwicklungsprozess eröffnet.

1.1. Motivation

Obwohl der Komponentenbegriff im heutigen Software Engineering nicht mehr wegzudenken ist, wird er in keiner aktuellen Programmiersprache explizit unterstützt. Es bleibt daher dem Geschick und der Konsequenz der Entwickler überlassen, die Regeln der Komponentenorientierung im Softwareprojekt umzusetzen. Unzulässige Abhängigkeiten zwischen Komponenten bleiben so verborgen und zeigen sich erst bei einer notwendigen Erweiterung oder einem Review des Softwaresystems. Ein Refactoring zur Beseitigung der unzulässigen Abhängigkeiten ist zu diesem Zeitpunkt unter Umständen mit viel Aufwand verbunden.

Aus diesem Grund ist die Idee entstanden, unzulässige Abhängigkeiten direkt bei der Eingabe zu validieren und dem Entwickler ein unmittelbares Feedback zu geben.

1.2. Ziel der Arbeit

Im Rahmen der Bachelorarbeit soll untersucht werden, welche Möglichkeiten sich durch die direkte Unterstützung der Konzepte der Komponentenorientierung eröffnen. Hierzu soll eine textuelle, domänenspezifische Sprache entwickelt werden, welche es ermöglichen soll, die Begrifflichkeiten von Komponenten in der Außensicht zu nutzen. Durch diese zusätzlichen semantischen Informationen eröffnen sich insbesondere im Bereich der automatischen

Validierung von Abhängigkeiten zwischen Komponenten neue Möglichkeiten. Die Richtlinien der Quasar-Standardarchitektur für Abhängigkeiten zwischen Komponenten sollen dieser Validierung zugrunde liegen.

1.3. Abgrenzung

Diese Arbeit hat nicht den Anspruch eine Sprache zu entwickeln, welche sich uneingeschränkt für den produktiven Einsatz in einem Softwareprojekt eignet. Vielmehr soll ein Prototyp erstellt werden, welcher Möglichkeiten der direkten komponentenorientierten Programmierung aufzeigt. Demnach werden auch Aspekte, welche die Usability der Sprache betreffen, in dieser Arbeit nicht untersucht werden.

1.4. Erläuterungen zur Struktur

Als theoretische Grundlage dieser Arbeit dient die Quasar-Standardarchitektur, welche im zweiten Kapitel vorgestellt werden soll.

Im Anschluss befasst sich Kapitel drei mit dem Thema der domänenspezifischen Sprachen. Es beschreibt unter anderem, was man unter einer domänenspezifischen Sprache versteht, welche Gründe es für ihre Verwendung gibt und wie eine domänenspezifische Sprache grundsätzlich aufgebaut ist.

In Kapitel vier wird ein einfaches Informationssystem spezifiziert und entworfen, welches als Anschauungsobjekt in dieser Arbeit dienen soll.

Kapitel fünf widmet sich dem fachlichen Entwurf der domänenspezifischen Sprache. In diesem Kapitel sollen Schritt für Schritt Syntax und Semantik der Sprachelemente entwickelt werden.

Im Anschluss befasst sich Kapitel sechs mit der Realisierung des zuvor entwickelten Sprachentwurfs.

In Kapitel sieben soll dann das zuvor spezifizierte Informationssystem mit Hilfe der domänenspezifischen Sprache realisiert werden.

Abschließend werden dann in Kapitel acht die Ergebnisse dieser Arbeit bewertet und es soll ein Ausblick auf die Weiterentwicklungsmöglichkeiten der domänenspezifischen Sprache gegeben werden.

2. Quasar und Komponenten

Die Quasar-Standardarchitektur von Capgemini (ehemals sd&m) dient als theoretische Grundlage dieser Arbeit. Dieses Kapitel soll die Ideen und Konzepte hinter Quasar beschreiben. Da Quasar im Großen und Ganzen eine Sammlung und Konkretisierung von Architekturprinzipien darstellt, werden sie an dieser Stelle beschrieben. Des Weiteren werden die Begriffe der Komponente und der Schnittstelle in Quasar erläutert, welche eine zentrale Bedeutung für diese Arbeit haben.

2.1. Die Quasar-Standardarchitektur

Die Quasar-Standardarchitektur ist eine Softwarearchitektur, welche grundlegende Regeln für den Bau von Informationssystemen aufstellt, die so allgemein gehalten sind, dass sie für jedes Informationssystem gelten und daher standardmäßig angewandt werden können.

Unter Softwarearchitektur versteht man

> ... the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them (Bass u. a., 1998).

Die von der Quasar-Standardarchitektur beschriebenen Ideen und Konzepte stellen keine komplett neuen Ansätze für die Entwicklung von Softwaresystemen dar. Sie sind vielmehr eine Sammlung von allgemein akzeptierten Prinzipien des Software Engineerings und best practices, die im Laufe der Zeit bei Capgemini zusammengetragen und 2004 als Quasar-Standardarchitektur veröffentlicht wurden (siehe Siedersleben, 2004). Die Quasar-Standardarchitektur ist aus wissenschaftlicher Sicht unfertig, da viele Konzepte und Ideen aus praktischer Erfahrung heraus abgeleitet wurden und sich nicht auf wissenschaftlich fundierte Untersuchungen stützen. Der Grund hierfür ist, dass Quasar einen möglichst einfachen und verständlichen Leitfaden für die Praxis liefern möchte, der die Entwickler nicht mit zuviel theoretischem Hintergrund belastet.

Im Zentrum von Quasar stehen die Grundprinzipien des Software Engineerings: Die Trennung von Zuständigkeiten, sowie das Denken in Schnittstellen und Komponenten. Diese grundlegenden Prinzipien gelten nicht nur in den Standardwerken des Software Engineering

als allgemein akzeptiert und anerkannt, sondern haben sich zuvor schon in anderen Ingenieurwissenschaften, wie zum Beispiel dem Maschinenbau oder der Elektrotechnik, bewährt. In der angewandten Informatik gelten diese Prinzipien allerdings noch nicht als verbindlich, was zum Beispiel das Fehlen eines Komponentenbegriffes in Programmiersprachen zeigt.

> Erstaunlicherweise kennen die heutigen Programmiersprachen das Konzept der Komponente nicht oder nur in einer versteckten, technischen Form ... nur das Geschick des Programmierers bestimmt, ob diese technischen Möglichkeiten komponentenorientiert genutzt werden (vgl. Siedersleben, 2004, S.41).

Bei dieser Problematik setzt Quasar an und versucht ein Gesamtpaket für den Softwareentwurf zu liefern, welches das Zusammenspiel der Architekturprinzipien beschreibt. Ziel der Quasar-Standardarchitektur ist es, ein allgemein gültiges Verständnis für Softwarequalität zu schaffen, so dass diese Prinzipien auch in der Praxis vermehrt Anwendung finden. Dies soll über vier Ebenen erreicht werden:

1. Anwendungsnahe und verständliche Beschreibung besonders wichtiger Regeln und Mechanismen der Softwaretechnik.
2. Einheitliche, technikneutrale Definition von Begrifflichkeiten in Bezug auf Softwarearchitektur.
3. Syntaktische und semantische Definition von Standardschnittstellen, auf denen jedes Projekt aufbauen kann.
4. Die Bereitstellung von Standardkomponenten für einige der Standardschnittstellen.

2.2. Architekturprinzipien von Quasar

Der folgende Abschnitt beschreibt die zehn Architekturprinzipien, auf denen Quasar aufbaut, und erläutert für jede Richtlinie die Herkunft sowie die Bedeutung für die Quasar-Standardarchitektur.

2.2.1. Trennung von Zuständigkeiten

Beschreibung: Unter der Trennung von Zuständigkeiten (engl. *Separation of Concerns*) versteht man ein Softwaresystem so in Untereinheiten zu unterteilen, dass sich jeder dieser Einheiten eine genau definierte Aufgabe zuordnen lässt. Jede Aufgabe muss sich klar von denen der anderen Einheiten abgrenzen lassen, so dass sie sich so wenig wie möglich in ihrer Funktionalität überschneiden.

Herkunft: Der Ausdruck *Separation of Concerns* wurde von Edsger W. Dijkstra in seinem Paper „On the role of scientific thought“ geprägt (siehe Dijkstra, 1982, S.60 - 66).

Bedeutung für Quasar: Die Trennung von Zuständigkeiten ist eines der Kernprinzipien von Quasar, denn sie macht ein großes System leichter verständlich. Die Forderung der Trennung von Fachlichkeit und Technik, sowie der Trennung von Aspekten innerhalb der beiden Bereiche, beruht auf diesem Architekturprinzip.

2.2.2. Minimierung von Abhängigkeiten

Beschreibung: Unter der Minimierung von Abhängigkeiten (engl. *Strong Cohesion & Loose Coupling*) versteht man, die Abhängigkeiten zwischen den Bestandteilen eines Softwaresystems durch Zusammenfassen oder Entkopplung zu minimieren.

Herkunft: Diese Regel ist schon länger bekannt, wurde aber erstmals von Craig Larman in den GRASP (General Responsibility Assignment Software Patterns) systematisch beschrieben (siehe Larman, 2005).

Bedeutung für Quasar: Quasar fordert einen starken inhaltlichen Zusammenhalt innerhalb einer Softwareeinheit *(Strong Cohesion)*, die Softwareeinheiten untereinander sollten im Gegensatz dazu aber möglichst lose gekoppelt sein *(Loose Coupling)*. Dies erlaubt in Kombination mit dem Prinzip der Trennung von Zuständigkeiten die Aufteilung eines Projektauftrags in separierte Bestandteile und ermöglicht eine getrennte Bearbeitung.

Das Prinzip fördert außerdem...

...die leichtere Wartbarkeit, da Änderungen an einer Stelle im Programm nur wenige Änderungen an anderen Stellen mit sich führen.

... die Verständlichkeit des Codes, da sich sie Funktionalität an einer Stelle befindet und wenige Wechselwirkungen mit anderen Programmteilen bestehen.

... die Testbarkeit, da ein Programmteil auf Grund seiner geringen Kopplung unabhängig vom übrigen System getestet werden kann.

... eine hohe Wiederverwendbarkeit, da ein Programmteil auf Grund seiner geringen Kopplung getrennt vom übrigen System wiederverwendet werden kann.

2.2.3. Geheimnisprinzip

Beschreibung: Das Geheimnisprinzip besagt, dass jeder Softwareteil sein internes Wissen kapselt und so vor anderen Softwareteilen verbirgt.

Herkunft: Das Konzept der Datenkapselung (engl. encapsulation) ist schon länger im Zusammenhang der modularen, strukturierten Programmierung bekannt, die erstmals von Gauthier und Ponto (1970) beschrieben wurde. Der Begriff des *information hiding* wurde von David Parnas geprägt.

Bedeutung für Quasar: Das Geheimnisprinzip erweitert das Prinzip der Trennung von Zuständigkeiten und das Prinzip der Minimierung von Abhängigkeiten durch die Forderung soviel internes Wissen wie möglich zu verbergen. Die Forderung der Verwendung von Schnittstellen stützt sich unter anderem auf dieses Prinzip. Auch der Vorschlag der Kapselung des Anwendungskerns durch eine Fassade basiert auf dem Geheimnisprinzip

2.2.4. Homogenität

Beschreibung: Das Prinzip der Homogenität verlangt ähnliche Problemstellungen durch ähnliche Lösungen zu realisieren, um eine einheitliche und verständliche Architektur zu gewährleisten.

Herkunft: Die Forderung nach der Einheitlichkeit von Lösungen ist eine grundsätzliche Anforderung strukturierter Vorgehensweisen.

Bedeutung für Quasar: Das Prinzip der Homogenität überschneidet sich zum Teil mit den Prinzipien der Trennung von Zuständigkeiten und der Redundanzfreiheit, welche verlangen, dass sich Code in seiner Funktionalität so wenig wie möglich überschneidet und an genau einer Stelle umgesetzt wird. Dies führt zwangsläufig zur Homogenität der Architektur. Auch die Verwendung von Code Conventions begünstigt die Einhaltung des Prinzips der Homogenität. Quasar verlangt, dass einmal getroffene Designentscheidungen einheitlich angewandt werden sollen. Dies unterstützt in erster Linie die Forderung nach einer leicht verständlichen Architektur.

2.2.5. Redundanzfreiheit

Beschreibung: Das Prinzip der Redundanzfreiheit besagt, dass von verschiedenen Softwareteilen benötigte technische oder fachliche Funktionalität nur an genau einer Stelle umgesetzt werden sollte. Falls gemeinsame Funktionalität besteht, sollte diese ausgelagert und

so verallgemeinert werden, dass sie von allen Softwareteilen gemeinsam verwendet werden kann.

Herkunft: Der Begriff der Redundanz kommt ursprünglich aus der Informationstheorie und wurde auf das Software Engineering übertragen. Das Prinzip der Redundanzfreiheit überschneidet sich mit den Prinzipien der Trennung von Zuständigkeiten und der Homogenität.

Bedeutung für Quasar: Durch die Vermeidung von Redundanz wird eine Softwarearchitektur leichter verständlich und leichter wartbar. Änderungen an einem redundanten Softwareteil müssten ansonsten an vielen Stellen nachgezogen werden. Dies birgt ein hohes Fehlerpotenzial.

2.2.6. Unterscheidung von Software-Kategorien

Beschreibung: Das Prinzip der Unterscheidung von Software-Kategorien stellt eine Konkretisierung des Prinzips der Trennung von Zuständigkeiten dar. Jedem Softwareteil wird dabei genau eine der folgenden vier Kategorien zugeordnet:

A-Software setzt die fachlichen Anforderungen um und besitzt kein Wissen über technische Aspekte des Systems.

Zu T-Software zählen die Softwareteile, welche die technischen Aspekte des Systems realisieren und kein Wissen über die Fachlichkeit beinhalten.

0-Software ist Standardsoftware, die weder Informationen über Fachlichkeit noch über technische Aspekte beinhaltet und sich somit universell einsetzen lässt.

R-Software ist sogenannte Adapter-Software, welche weder Informationen über Fachlichkeit noch über technische Aspekte beinhaltet, diese aber referenzieren darf.

Es ist sogar möglich, diese vier Kategorien durch eigene Kategorien zu erweitern, um die Abhängigkeiten zwischen Komponenten unterschiedlicher Schichten der Architektur besser zu kontrollieren.

Herkunft: Das Prinzip der Unterscheidung von Software-Kategorien ist ein neues Quasar Architekturprinzip, das zusammen mit der Quasar-Standardarchitektur in Siedersleben (2004) veröffentlicht wurde.

Bedeutung für Quasar: Die Unterscheidung von Software-Kategorien ist ein Quasar eigenes Architekturprinzip und hat große Bedeutung für die Regelung von Abhängigkeiten zwischen Komponenten.

2.2.7. Schichtenbildung

Beschreibung: Das Architekturprinzip der Schichtenbildung (engl. *Layering*) schlägt vor, ein Softwaresystem so in Schichten zu unterteilen, dass jede Schicht Softwareteile mit ähnlichem Aufbau oder ähnlicher Grundfunktionalität zusammenfasst. Direkte Abhängigkeiten dürfen nur zwischen den Softwareteilen innerhalb einer Schicht oder darunter liegender Schichten bestehen.

Herkunft: Mehrschichtenarchitekturen sind im Softwareengineering häufig anzutreffen. Eine der ersten Mehrschichtarchitekturen war die Model-View-Controller-Architektur, die erstmals von Reenskaug (1979) beschrieben wurde.

Bedeutung für Quasar: Schichten helfen in Quasar dabei, das System in Untereinheiten zu unterteilen und die Richtung der Abhängigkeiten zwischen ihnen zu bestimmen. Damit lässt sich die Frage beantworten, auf welcher Ebene sich ein Softwareteil befinden muss, um von anderen Softwareteilen verwendet werden zu können.

2.2.8. Vertragsbasierter Entwurf

Beschreibung: Der Vertragsbasierte Entwurf (engl. *Design by Contract*) sieht vor, dass das Zusammenspiel einzelner Systemteile durch zuvor genau spezifizierte Schnittstellen geregelt wird. Die Spezifikation der Schnittstellen geht dabei über eine statische Beschreibung der Methodensignaturen hinaus. Zu jeder Methode werden alle *Vorbedingungen* aufgezählt, die der Aufrufende sicherstellen muss, um die angebotene Funktionalität in Anspruch zu nehmen. Stellt er diese sicher, werden ihm im Gegenzug alle angegebenen *Nachbedingungen* garantiert. Zusätzlich werden *Invarianten* spezifiziert, welche Konsistenzbedingungen darstellen und aus diesem Grund vor und nach einem Methodenaufruf gelten müssen. Auch mögliche Fehler gehören zur Spezifikation der Schnittstelle. Obwohl es die Aufgabe der Schnittstellenverträge ist, die angebotene Funktionalität möglichst gut zu beschreiben, so sollten diese aber gleichzeitig im Sinne des Geheimnisprinzips möglichst wenig Einzelheiten über eine zugrundeliegende Implementierung preisgeben.

Herkunft: Das Architekturprinzip *Design by Contract* wurde von Bertrand Meyer in Zusammenhang mit der von ihm entwickelten Programmiersprache *Eiffel* formuliert und 1986 veröffentlicht (siehe Meyer, 1986). Da *Design by Contract* ein geschützter Begriff ist, wird das Prinzip auch oft als *Programming by Contract*, *Contract Programming* oder *Contract-First Programming* bezeichnet.

Bedeutung für Quasar: Schnittstellenverträge regeln die „Bedingungen der Zusammenarbeit“ zwischen Softwareteilen und sind daher für wiederverwendbare Softwareteile unerlässlich. Die Quasar-Standardarchitektur definiert eine eigene Notation für Schnittstellenver-

träge, die *Quasar Specification Language*. Diese geht über die oben genannten Aspekte hinaus und erlaubt zusätzlich die Beschreibung von Methodenarten (einfache Abfragen, von abgeleitete Abfragen und Kommandos),von der Wiederholbarkeit der Ausführung, von Zustandsmodellen und von Testfällen.

2.2.9. Datenhoheit

Beschreibung: Das Prinzip der Datenhoheit (engl. *Data Sovereignty*) folgt aus dem Geheimnisprinzip und dem Prinzip der Trennung von Zuständigkeiten. Es besagt, dass es für jedes Element des Datenhaushaltes (*Entität*) genau einen Softwareteil gibt, der es verwaltet. Dieser Softwareteil ist zuständig für die Struktur der verwalteten Daten und bietet Dienste für den Zugriff auf seine Daten an. Hierfür stellt er die Integrität und Konsistenz seiner Daten sicher. Das Prinzip der Datenhoheit hilft dabei Abhängigkeiten zwischen Softwareteilen zu minimieren und die Wartbarkeit zu erhöhen.

Herkunft: Auch das Prinzip der Datenhoheit ist Quasar spezifisch.

Bedeutung für Quasar: Quasar schlägt vor, keine Entitäten aus den Komponenten, die sie verwalten, nach außen zu geben und an ihrer Stelle flache Transportobjekte zu verwenden. Dieses Vorgehen basiert auf dem Prinzip der Datenhoheit und verringert so die Abhängigkeit zwischen den Komponenten.

2.2.10. Wiederverwendung

Beschreibung: Das Prinzip der Wiederverwendung besagt, dass man Softwareteile, die auch für andere Systeme relevant sind, architektonisch so gestalten sollte, dass sie sich in diesen Systemen wiederverwenden lassen. Auf der anderen Seite bedeutet es aber auch, ein Softwaresystem so zu gestalten, dass es die Integration und den Austausch solcher Softwareteile begünstigt. Dieses Vorgehen führt zu Redundanzfreiheit, auch zwischen Softwaresystemen.

Herkunft: Der Gedanke der Wiederverwendung von Softwareteilen kam mit der modularen Programmierung auf, welche Mittel schuf um Wiederverwendbarkeit auf einfache Weise zu implementieren.

Bedeutung für Quasar: Das Prinzip der Wiederverwendung geht bei Quasar über die Benutzung, beziehungsweise Erstellung von Standardsoftwarekomponenten hinaus. Es umfasst auch die Verwendung und Bereitstellung von Entwurfsmustern, Referenzarchitekturen und Architekturprinzipien.

2.3. Komponenten in Quasar

Die Verwendung von Komponenten ist eine notwendige Folge der Architekturprinzipien von Quasar. Nur die Unterteilung der Klassen und Schnittstellen eines Softwaresystems in Untereinheiten macht dessen Komplexität kontrollierbar. Komponenten sind die grundlegenden Bausteine der *Softwarearchitektur* und müssen daher die an die Softwarearchitektur gestellten Forderungen nach Umgang mit sich ändernden Anforderungen, Parallelisierung von Entwicklungsaufgaben und Wiederverwendbarkeit realisieren.

Komponenten werden in Quasar über folgende sechs Eigenschaften definiert (vgl. Siedersleben, 2004, S. 42f)

1. Komponenten exportieren eine oder mehrere Schnittstellen. Eine Komponente *C* exportiert eine Schnittstelle *S*, wenn *C S* implementiert.
2. Eine Komponente importiert eine beliebige Anzahl von Schnittstellen. Eine Komponente *C* importiert eine Schnittstelle *S*, wenn *C* nur nach erfolgreicher Konfiguration mit einer Komponente *D*, welche *S* exportiert, lauffähig ist.
3. Eine Komponente versteckt nach dem Geheimnisprinzip ihre Implementierung und kann durch eine andere Komponente, die dieselben Schnittstellen exportiert, ersetzt werden.
4. Eine Komponente ist wiederverwendbar, da sie ihre Laufzeitumgebung nicht kennt und über diese nur minimale Annahmen macht.
5. Eine Komponente komponiert beliebig viele *Subkomponenten*.
6. Eine Komponente ist der grundlegende Baustein der Softwarearchitektur und somit die Einheit der Planung.

Nach dem Prinzip der Unterscheidung von Software-Kategorien sollte jede Komponente genau einer der definierten Softwarekategorien angehören. Könnte man einer Komponente mehr als eine Kategorie zuordnen, wäre dies ein Verstoß gegen das Prinzip der Trennung von Zuständigkeiten und sollte unbedingt vermieden werden. Eine Sonderrolle spielen hierbei die Kategorien 0 (*Standardsoftware*) und R (*Adaptersoftware*).

Jede Komponente kann *Standardsoftware* verwenden, da diese als „neutral“ bewertet wird und die Komponente damit keine zusätzliche Kategorie erhält. Als *Standardsoftware* gelten allgemein akzeptierte Komponenten und Schnittstellen, welche gut getestet und verbreitet sind. Ihre Verwendung wird daher gefördert. Sie ist global verfügbar und könnte als die „Software des Laufzeitsystems“ beschrieben werden. Die Verfügbarkeit der 0-Software ist die Annahme, welche eine Komponente über ihre Umgebung macht und ohne die sie nicht

lauffähig ist. Es muss daher gut darauf geachtet werden, welche Software man für sein System zu *Standardsoftware* erklärt. Ein Beispiel für *Standardsoftware* in einem Java-Projekt stellen die Pakete `java.lang` und `java.util` dar.

Auch durch die Verwendung von *Adaptersoftware* erhält eine Komponente keine zusätzliche Kategorie. Ihre Aufgabe ist es, die Verbindung zwischen Komponenten zweier unterschiedlicher Kategorien herzustellen. Sie führt nur die Projektion einer Schnittstelle auf eine andere durch, enthält aber kein spezielles Wissen aus einer der abzubildenden Kategorien und kann daher ohne Verstoß gegen das Prinzip der Trennung von Zuständigkeiten verwendet werden.

2.4. Schnittstellen in Quasar

Obwohl sich der Begriff der Schnittstelle (engl. *Interface*) in heutigen Programmiersprachen stärker durchsetzen konnte als der der Komponente, ist seine Bedeutung nicht eindeutig und hängt sehr stark vom Kontext ab. Schnittstellen werden grundsätzlich in zwei Kategorien aufgeteilt: Benutzerschnittstellen und Programmschnittstellen. Eine Benutzerschnittstelle stellt die Funktionalität eines Softwaresystems einem oder mehreren Benutzern zur Verfügung und lässt diesen mit dem System interagieren. Eine Programmschnittstelle verbindet im Gegensatz dazu die Komponenten eines Softwaresystems miteinander. Jede Programmschnittstelle definiert (nach Siedersleben, 2004, S. 44) Methoden mit den folgenden Eigenschaften:

- **Syntax:** Die Methodensignatur bestehend aus Argumenten (in/out) und deren Typen sowie zusätzlich dem Rückgabetyp der Methode.
- **Semantik:** Die Beschreibung der Methodenfunktionalität und deren Eigenheiten. So zum Beispiel, ob Null-Werte in den Parametern erlaubt sind oder nicht und wie damit umgegangen wird.
- **Protokoll:** Erfolgt die Ausführung der Methode synchron oder asynchron?
- **Nichtfunktionale Eigenschaften:** Genaue Beschreibung der Performance, Robustheit, Verfügbarkeit und Kosten (beispielsweise bei Web-Anwendungen).

Eine Schnittstelle in Java ist demnach keine vollständige Programmschnittstelle, da diese meist nur die Syntax der Methoden beschreibt.

2.4.1. Definition von Schnittstellen

Schnittstellen sind grundsätzlich unabhängig von ihrer Implementierung, da diese eine Funktionalität nach dem Geheimnisprinzip nur „von außen“ beschreiben sollten und nichts mit deren Realisierung zu tun haben. In der Praxis muss eine Schnittstelle aber an einer bestimmten Stelle im Softwaresystem definiert werden, was starken Einfluss auf die Wiederverwendbarkeit der implementierenden Komponenten hat. Quasar unterscheidet in diesem Kontext drei Arten von Schnittstellen: angebotene Schnittstellen, angeforderte Schnittstellen und Standardschnittstellen.

Standardschnittstellen sind 0-Software und damit global verfügbar. Standardschnittstellen eignen sich besonders für die Verwendung in der Hülle einer Schnittstelle, um die Kopplung zwischen Komponenten zu minimieren. Als Hülle bezeichnet man die Parameter- und Rückgabetypen aller Methoden einer Schnittstelle. Da Standardschnittstellen an sehr vielen Stellen in einem System verwendet werden, sollten sie besonders gut getestet und dokumentiert sein.

Angebotene und angeforderte Schnittstellen werden innerhalb einer Komponente definiert. Über eine angebotene Schnittstelle stellt eine Komponente ihre Funktionalität anderen Komponenten zur Verfügung. Eine angeforderte Schnittstelle wird hingegen von einer Komponente definiert, damit eine andere Komponente die beschriebene Funktionalität für sie realisiert. Die direkte Verwendung einer angebotenen oder angeforderten Schnittstelle bindet die referenzierende an die definierende Komponente (den Besitzer der Schnittstelle) und ist ausschließlich zusammen mit der definierenden Komponente wiederverwendbar. Möchte man diese Kopplung vermeiden, muss ein Adapter implementiert werden, welcher eine angebotene auf eine angeforderte Schnittstelle abbildet.

2.4.2. Schnittstellen-Kategorien

Definiert eine Komponente eine Schnittstelle, so erhält die Schnittstelle grundsätzlich die Kategorie der definierenden Komponente. Standardschnittstellen sind von der Kategorie 0 und können damit nur innerhalb von 0-Komponenten definiert werden. Damit eine Komponente die Kategorie 0 erhält, darf sie ausschließlich 0-Software verwenden. Die Hülle jeder Schnittstelle enthält im optimalen Fall ausschließlich Schnittstellen der Kategorie 0, da in diesem Fall die Kopplung zwischen den Komponenten minimal ist.

3. Domänenspezifische Sprachen

Ziel dieser Arbeit ist die Entwicklung einer domänenspezifischen Sprache. Dieses Kapitel soll erläutern, was eine domänenspezifische Sprache überhaupt ist und was für Gründe es für deren Verwendung gibt. Des Weiteren wird der grundlegende Aufbau einer DSL erläutert und dann am Beispiel zweier Frameworks für die Erstellung von DSLs vertieft.

3.1. Was ist eine domänenspezifische Sprache?

Domänenspezifische Sprachen werden schon seit Anbeginn der Computer verwendet, daher ist der Schöpfer des Begriffes unbekannt. Ein populärer Verfechter von domänenspezifischen Sprachen ist Martin Fowler, welcher sie wie folgt definiert:

> Domain-specific language (noun): a computer programming language of limited expressiveness focused on a particular domain (vgl. Fowler und Parsons, 2011, S.27).

Eine domänenspezifische Sprache (engl. *domain specific language*, kurz: *DSL*) ist demnach eine formale (Programmier-) Sprache, die in ihren Ausdrucksmöglichkeiten an eine spezifische Anwendungsdomäne angepasst ist. Unter einer Anwendungsdomäne (kurz: *Domäne*) versteht man einen abgeschlossenen, fachlichen Wissensbereich. Beispiele für Anwendungsdomänen sind die Medizin, das Versicherungs- oder das Bankenwesen. Diese Anwendungsdomänen können wiederum aus untergeordneten Anwendungsdomänen zusammengesetzt sein. So teilt sich die Anwendungsdomäne der Medizin beispielsweise in die Anwendungsdomänen der Human- und Veterinärmedizin und diese unterteilen sich wieder in die Anwendungsdomänen der einzelnen Spezialisierungen. Eine domänenspezifische Sprache beschränkt sich in ihrer Ausdrucksmöglichkeit auf eine genau definierte Domäne und sollte dabei so allgemein wie möglich und so speziell wie nötig sein. Das bedeutet, dass die Sprache alle Probleme der Zieldomäne darstellen kann, jedoch nichts was außerhalb der Domäne liegt. Idealerweise deckt sich ihre Ausdrucksmöglichkeit mit dem Fachvokabular der Domänenexperten und ist für diese ohne besonderes Zusatzwissen bedienbar.

Im Kontrast zu den domain specific languages stehen die general purpose languages (kurz: *GPL*), welche für viele Anwendungsfälle und Problemstellungen einsetzbar sind. Eine GPL

ist im Gegensatz zu einer DSL in der Regel turing-vollständig. Aus diesem Grund wird man kein Softwaresystem in einer DSL entwickeln. Vielmehr wird man eine GPL wie Java oder C verwenden und nur die fachlichen Aspekte spezifischer Domänen durch DSLs realisieren.

Nach Martin Fowler (Fowler und Parsons, 2011, S. 28) lassen sich DSLs in drei Hauptkategorien unterteilen:

- Eine **externe DSL** ist unabhängig von der GPL, in welcher das Softwaresystem geschrieben wird. Somit hat sie eine individuelle Syntax und benötigt einen Parser, welcher den Quelltext der DSL einliest. Beispiele für DSLs sind reguläre Ausdrücke, SQL und XML Konfigurationsdateien.
- Als **interne DSL** bezeichnet man Sprachen, die eine Untermenge an Ausdrucksmöglichkeiten der GPL verwenden und so das Gefühl einer individuellen Sprache vermitteln. Im Gegensatz zu einer externen DSL benötigt man für eine interne DSL keinen spezifischen Parser, da der Parser der Hostsprache verwendet werden kann. Beliebte Hostsprachen für interne DSLs sind Lisp, Ruby oder Skala.
- Unter einer **Language Workbench** versteht man eine IDE, welche für die Definition und das Bauen einer DSL entwickelt wurde. Eine Language Workbench bietet neben der Möglichkeit DSLs zu entwerfen meist auch einen spezifischen Editor für die fertige DSL mit Syntaxhervorhebung und Codevervollständigung.

3.2. Gründe für die Verwendung einer DSL

Softwaresysteme werden meist in einer einheitlichen Programmiersprache geschrieben, wobei die Auswahl der Sprache oft nichts mit deren Eignung zu tun hat, sondern sich vielmehr nach dem vorhandenen Know-How im Entwicklungsteams richtet. Am besten beschreibt dieses Problem ein Zitat des amerikanischen Psychologen Abraham Maslow: „If the only tool you have is a hammer, you tend to see every problem as a nail.“ Auf die Entwicklung eines Softwaresystems übertragen bedeutet es, dass die Verwendung einer einheitlichen Programmiersprache die Verständlichkeit des Codes beeinträchtigt und den Code komplexer macht als er sein müsste. Dies trifft insbesondere auf Informationssysteme zu, mit denen domänenspezifische Prozesse unterstützt werden. Für diese Prozesse sind meist spezifische Ausdrucksmöglichkeiten vorhanden, die sich innerhalb der Domäne entwickelt haben. Hierfür bietet es sich an, diese mit einer DSL zu beschreiben, was folgende Vorteile mit sich bringt:

- Die domänenspezifische Beschreibung eines Problems ist **deklarativ** und damit **kompakter** als die Beschreibung in einer GPL. Auf diese Weise lässt sich **Redundanz im Quellcode minimieren** und die **Lesbarkeit maximieren**.

- Die domänenspezifische Sprache ist **leichter zu erlernen** als eine GPL, da sie im Sprachumfang minimal ist. Daher kann die Fachlichkeit direkt von Domänenexperten modelliert werden, was den **Kommunikationsaufwand reduziert** und Missverständnisse zwischen Domänenexperten und Entwicklern gar nicht erst aufkommen lässt.
- Bei der Verwendung von externen DSLs und Language Workbenches hat man zudem die Möglichkeit, domänenspezifische, **statische Validierungen** durchzuführen, um fachliche Fehler schon zur Entwurfszeit aufzudecken.

Eine praktische Studie von drei Professoren der Old Dominion University (USA), der Georgia State University (USA) und der Helsinki School of Economics (Finnland) belegt die zuvor genannten Punkte. In dieser mussten Probanden Anforderungsänderungen in der Unified Specification Language (kurz: UML), einer General-Purpose Modeling Language, welche heutzutage als Standard für die Spezifikation von Softwaresystemen akzeptiert wird und in einer domänenspezifischen Modellierungssprache umsetzen. Sie hat ergeben, dass domänenspezifische Sprachen viel einfacher zu handhaben sind als GPLs:

> The DSM models' correctness score was about 20 percent higher than the UML model's score. ... Finally, the degree of changes in DSM is much smaller than in UML; UML diagrams required nearly twice the number of steps (Cao u. a., 2009, S. 21).

3.3. Aufbau einer domänenspezifischen Sprache

Der Aufbau einer domänenspezifischen Sprache unterscheidet sich je nach Kategorie der DSL. Folgende Unterteilung lässt sich aber bei den meisten DSLs erkennen:

- Das **syntaktische Modell** beschreibt den Aufbau der DSL. Bei einer externen DSL wird die Syntax durch eine Grammatik beschrieben, bei einer internen DSL entsteht die Syntax durch die Definition von Methoden und Symbolen innerhalb der Host-Sprache.
- Für die Verarbeitung von Programmiersprachen hat sich die Repräsentation der Daten als Baum bewährt. Bei einer externen DSL wird der Quellcode von einem Parser in einen **abstrakten Syntaxbaum (kurz *AST*)** umgeformt. Die Repräsentation der Syntax als Baum schränkt die Möglichkeiten von Referenzen zwischen den Knoten stark ein. Dieses Problem wird meist durch die Benutzung einer Symbol-Tabelle gelöst, welche sogenannte Cross-Referenzen ermöglicht. Auch die Methodenaufrufe einer internen DSL werden in eine hierarchische Struktur überführt, damit sie sich besser verarbeiten lassen.

- Im Mittelpunkt jeder domänenspezifischen Sprache steht ihr **semantisches Modell**. Das semantische Modell ist eine Datenstruktur, deren Aufgabe vergleichbar zu der des Datenmodells in einem Softwaresystem ist. Das semantische Modell sollte komplett unabhängig vom syntaktischen Modell sein, damit es möglich ist, verschiedene syntaktische Sprachausprägungen zu implementieren, die das selbe semantische Modell mit Daten befüllen. Zwei Ausdrücke haben die gleiche Semantik, wenn ihr semantisches Modell äquivalent ist. Soll kein Code aus dem semantischen Modell generiert werden, so implementiert man einen Interpreter, welcher die Daten des semantischen Modells direkt in Funktionsaufrufe umsetzt.
- Wenn aus der domänenspezifischen Sprache Code generiert werden soll, benötigt man einen **Generator**. Der Generator verarbeitet die Daten des semantischen Modells und erzeugt daraus Quellcode der Zielsprache. Die meisten Generatoren arbeiten templatebasiert. Ein Template ist eine Art Schablone, die an bestimmten Stellen Platzhalter für die Daten des semantischen Modells enthält. Auf diese Weise ist es auch nachträglich möglich, neue Templates für weitere Zielsprachen zu entwickeln.

3.3.1. Scoping und Typprüfung

Bei externen DSLs ist es notwendig, Gültigkeitsbereiche für die deklarierten Strukturen einer Sprache zu bestimmen. Diese Gültigkeitsbereiche werden als Scopes bezeichnet. Durch Scopes wird bestimmt, ob eine Struktur, wie beispielsweise eine deklarierte Variable oder Klasse, in einem bestimmten Kontext sichtbar ist. Des Weiteren wird durch Scopes geregelt, welche Strukturen durch andere verdeckt werden, sollte es mehrere Strukturen mit dem gleichen Bezeichner geben.

Meist reicht es nicht aus, nur die Sichtbarkeit von Strukturen zu regeln. In vielen Fällen ist eine Struktur zwar sichtbar, erfüllt aber bestimmte Anforderungen an den Typ nicht. Ein Beispiel hierfür ist die Multiplikation zweier zuvor definierter Variablen. Beide Variablen liegen innerhalb des Scopes, bei einer der Variablen handelt es sich aber um eine Zeichenkette. Diese Variable ist damit als Multiplikator nicht zulässig. Solche Typunzulässigkeiten werden in einem *Typesystem* definiert.

3.4. Frameworks für die Entwicklung domänenspezifischer Sprachen

Dieser Abschnitt soll den Aufbau zweier konkreter Open-Source-Frameworks für die Erstellung domänenspezifischer Sprachen erläutern. *Xtext* stellt klassische Funktionalität zur Ent-

wicklung externer DSLs, aber auch GPLs zur Verfügung, wohingegen es sich bem *Meta Programming System* (kurz: *MPS*) um eine Language Workbench handelt, die einen neuartigen Ansatz bei der Entwicklung von DSLs verfolgt.

3.4.1. Xtext

Xtext wurde erstmalig im Jahre 2006 im Rahmen des OpenArchitectureWare Projektes vorgestellt. Seit 2008 wird es bei Eclipse im Rahmen des Eclipse Modeling Projects im Bereich Textual Modeling Framework weiterentwickelt. Das Xtext-Framework stellt für den Entwickler einer DSL folgende Funktionalitäten zur Verfügung:

- Die Generierung eines Parsers auf Basis einer grammatikalischen Beschreibung der Sprache ähnlich EBNF
- Die Modellierung des semantischen Modells mit Hilfe von EMF-Modellen
- Generierung eines Eclipse-Plugins für die Sprache mit allen Eclipse-Features wie: Texteditor mit Autovervollständigung und Syntaxhervorhebung, Outline, statischer Validierungen und Quickfixes.

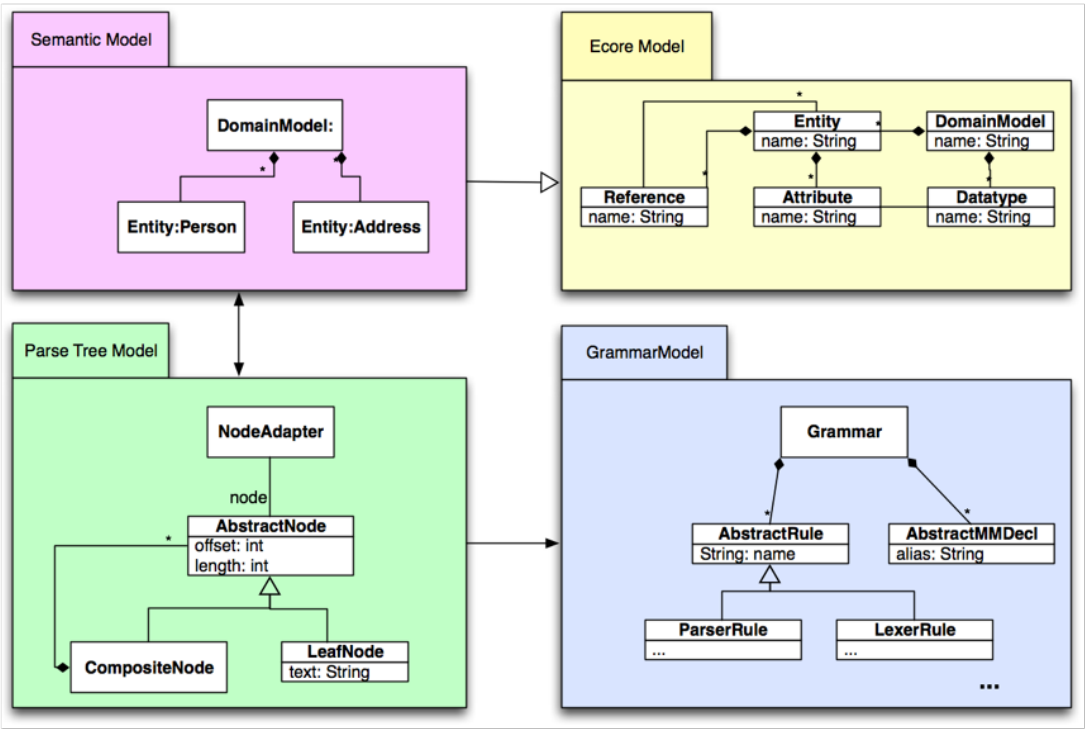

Abbildung 3.1.: Datenstrukturen in Xtext

Abbildung 3.1 zeigt die Datenstrukturen einer DSL für die Modellierung von Entitäten mit Xtext. Das semantische Modell (oben links) stellt die Semantik des vom Benutzer eingegebenen Quellcodes dar. In diesem Beispiel wird ein Domänenmodell mit den Entitäten `Person` und `Address` beschrieben. Dieses semantische Modell ist eine konkrete Instanz eines*Ecore-Modells* (oben rechts). Das *Ecore-Modell* definiert die zulässige Struktur von semantischen Modellen der Sprache und wird auch Metamodell genannt. In diesem Beispiel komponiert das `DomainModel` beliebig viele `Entity`s, die wiederum beliebig viele `Attribute`s komponieren. Die `Attribute`s sind von einem bestimmten `Datatype`, welcher im `DomainModel` definiert ist. Des Weiteren können `Entity`s beliebig viele andere `Entity`s über `Reference`s referenzieren. Das semantische Modell bzw. sein Metamodell sind komplett unabhängig von einer konkreten Syntax.

Der Zugriff auf die textuelle Repräsentation des Quellcodes erfolgt über den Syntaxbaum (unten links). Dieser enthält in den `LeafNode`s die Tokens des Quelltextes. Wandert man durch den Baum und fügt alle `LeafNode`s hintereinander, erhält man den zugrundeliegenden Quellcode. Jede Node des Syntaxbaumes enthält zudem eine Referenz auf die Regel der Grammatik, durch welche sie vom Parser verarbeitet wurde. Diese grammatikalischen Regeln sind im GrammarModel (unten rechts) enthalten.

Das *Eclipse Modeling Framework* bietet drei Sprachen an, um Operationen auf einem *Ecore-Modell* durchzuführen. Die Sprache *Xpand* ist eine statisch typisierte Templatesprache zur Codegenerierung und unterstützt unter anderem aspektorientierte Programmierung. *Check* ist eine Sprache, um Constraints für ein Modell zu definieren und *Xtend* ist eine funktionale Sprache, welche die Möglichkeit bietet Metamodelle durch Extensions um zusätzliche Logik zu erweitern. Eine der Hauptaufgaben von *Xtend* ist die Modelltransformation. Darunter versteht man die Überführung eines Modells in ein anderes Modell des gleichen oder eines anderen Metamodells.

3.4.2. Meta Programming System

Das Meta Programming System (kurz: *MPS*) startete im Jahre 2003 als Forschungsprojekt der Firma Jetbrains und ist seit 2005 öffentlich verfügbar. MPS fällt in die Kategorie der Language Workbenches und stellt eine komplette Entwicklungs- sowie Editierumgebung für domänenspezifische Sprachen zur Verfügung. Die Idee des Meta Programming Systems ist es, eine Language Workbench für modulare Spracherweiterungen zur Verfügung zu stellen. Stört einen Entwickler beispielsweise das Fehlen von *Clojures* in Java, so bietet MPS die Möglichkeit ein Modul zu implementieren, welches hierfür Sprachkonstrukte zur Verfügung stellt.

Die modulare Erweiterung von Sprachen birgt allerdings ein Problem: Es muss sichergestellt werden, dass die zugrundeliegende Grammatik der Sprache eindeutig ist, damit sie vom Par-

ser in einen eindeutigen Syntaxbaum überführt werden kann. Hat man sichergestellt, dass ein Modul A und eine Sprache S eine eindeutige Grammatik besitzen und auch ein Modul B in Kombination mit S eindeutig ist, so kann die Grammatik von A, B und S dennoch mehrdeutig sein. Dies führt dazu, dass man die Grammatik aller modularen Spracherweiterungen aufeinander abstimmen muss. Der Aufwand hierfür würde einerseits mit der Zahl an Spracherweiterungen exponentiell steigen und andererseits starke Einschränkungen bei der Wahl der Grammatik bedeuten. Aus diesem Grund geht MPS einen anderen Weg. Anstelle eines herkömmlichen Texteditors verwendet MPS einen strukturierten Texteditor. Das bedeutet, dass der Quellcode nicht in Form von Text in Dateien abgelegt, sondern ohne den Umweg über einen Parser, direkt ein abstrakter Syntaxbaum erstellt wird. Dies führt dazu, dass sich der Umgang mit MPS in folgenden Punkten gegenüber dem parserbasierten Vorgehen unterscheidet:

Strukturierter Texteditor

- Es können nur syntaktisch korrekte Eingaben gemacht werden. Was in der Strukturdefinition des Syntaxbaumes nicht vorgesehen ist, kann nicht gespeichert werden.
- Es können nur ganze Knoten des Baumes kopiert und eingefügt werden. Das Kopieren von Quellcode aus herkömmlichen Textdateien ist grundsätzlich nicht möglich.
- Das Ändern eines Schlüsselwortes im strukturierten Texteditor führt dazu, dass sich die Art des Knotens ändert und damit standardmäßig alle Unterknoten im Baum gelöscht werden. Ein anderes Verhalten bedarf einer gesonderten Behandlung.

Versionierung

- Der Quellcode kann nicht direkt unter Versionskontrolle gestellt werden, da er nur als Baum im Speicher liegt und keine direkte textuelle Repräsentation existiert. MPS speichert den Baum in einer XML basierten Struktur, für welche eine Versionierung möglich ist. Hierfür stellt MPS innerhalb der Language Workbench Funktionalität zur Verfügung.

Codegenerierung

- Die Codegenerierung in MPS wird durch eine Spracherweiterung namens *Generator-Language* durchgeführt. Da die *Generator-Language* in MPS definiert ist und zur Erstellung von Templates ein strukturierter Texteditor verwendet werden muss, ist es notwendig, dass auch die Syntax der Zielsprache in MPS beschrieben ist.

Eine Spracherweiterung wird im Meta Programming System durch neun Sprachaspekte definiert (vgl. MPS User's Guide, 2011):

Structure: Jedes Sprachkonstrukt in MPS ist ein Knoten in einem AST, dem Semantischen Modell. Die Strukturbeschreibung eines Knotens enthält drei verschiedene Elemente: Eigenschaften (properties), Kindknoten (children) und Referenzen auf andere Knoten (references) (vgl. Abbildung 3.2). Eigenschaften dienen als Instanzvariablen für einen Knoten und können Informationen wie beispielsweise einen Knotenbezeichner enthalten. Kindknoten sind die Knoten, die als direkte Kinder zulässig sind. Ein binärer mathematischer Operator hätte als Kindknoten beispielsweise seine beiden Operanden. Referenzen sind Verweise auf Knoten über die Baumstruktur hinaus. Auf diese Weise werden Cross-Referenzen ermöglicht.

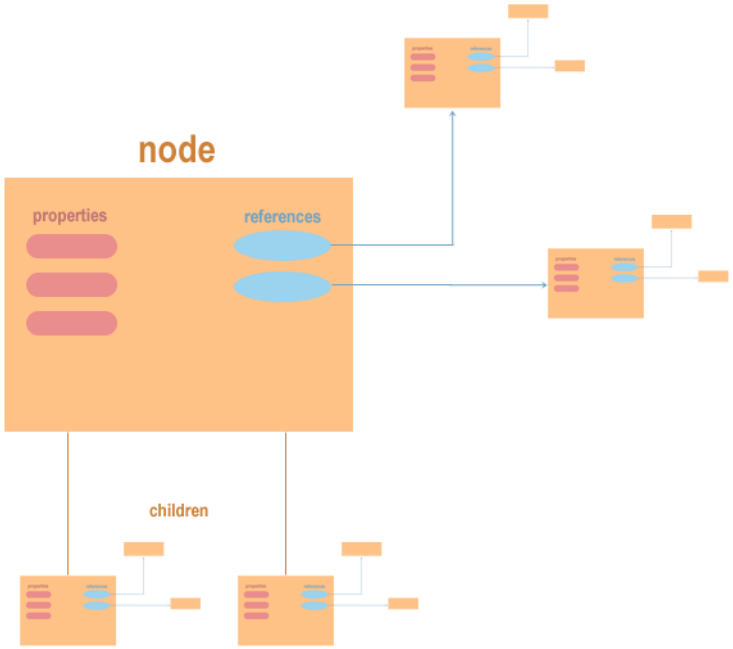

Abbildung 3.2.: Struktur eines Knotens in MPS

Editor: Über den Editor-Aspekt einer Sprache wird ihre Syntax beschrieben. An dieser Stelle werden die Schlüsselwörter einer Sprache definiert und es wird geregelt, an welcher Stelle welche Information aus dem AST angezeigt wird.

Actions: In diesem Aspekt können Besonderheiten für die Autovervollständigung konfiguriert werden.

Constraints: Über den Constraint-Aspekt können weitere Bedingungen an die strukturellen Elemente eines Knotens formuliert werden. An dieser Stelle wird auch das Scoping für referenzierte Knoten definiert.

Behavior: Über den Behavior-Aspekt lassen sich Methoden auf dem AST definieren.

Typesystem: Der Aspekt Typesystem definiert Regeln zur Berechnung von Typen der Sprache. Es kann zum Beispiel beschrieben werden, dass die Verknüpfung einer Zeichenkette mit einer Zahl als resultierenden Typen wiederum eine Zeichenkette ergibt. An dieser Stelle lassen sich auch Bedingungen formulieren, bei deren Verletzung Fehler oder Warnungen im strukturierten Texteditor angezeigt werden sollen.

Intentions: Intentions sind kontextsensitive Hilfen und Aktionen, ähnlich den Quickfixes in Eclipse.

Plugin: Unter dem Plugin-Aspekt lassen sich spezifische MPS IDE-Erweiterungen definieren.

Data Flow: Der Data Flow-Aspekt ermöglicht es, Datenflussüberprüfungen für die DSL durchzuführen mit deren Hilfe beispielsweise unerreichbare Stellen im Code gefunden werden können. Für Java lassen sich so Codebereiche nach einem `return` markieren, welche nie ausgeführt werden können. Auch der Lesezugriff auf eine uninitialisierte Variable lässt sich so aufdecken.

4. Einführung in das Beispielszenario

Bevor mit dem fachlichen Entwurf der Sprache begonnen wird, soll zunächst ein beispielhaftes Informationssystem vorgestellt werden, welches in den nachfolgenden Kapiteln als Anschauungsobjekt dienen soll. Für dieses werden zunächst Anforderungen und Prämissen spezifiziert, die im Anschluss in einen fachlichen Entwurf überführt werden sollen.

In dieser Arbeit soll ausschließlich die Fachlichkeit des Informationssystems realisiert werden. Auf die Anbindung an eine grafische Benutzerschnittstelle wird verzichtet.

4.1. Spezifikation

Bei dem Beispielsystem handelt es sich um ein einfaches News-System, das vielfältige Einsatzmöglichkeiten besitzt. In den folgenden Abschnitten sollen zunächst die Anforderungen und Prämissen an das News-System näher spezifiziert und anschließend in ein fachliches Datenmodell überführt werden.

4.1.1. Anforderungen und Prämissen

Das zu entwickelnde News-System soll die folgenden Anforderungen realisieren:

- A1 Das System soll die neusten x Nachrichten des Systems nach Datum absteigend sortiert ausgeben können
- A2 Das System soll die Anzahl der im System gespeicherten Nachrichten ausgeben können
- A3 Ein Administrator soll neue Nachrichten in das System eintragen können
- A4 Jede Nachricht besitzt eine eindeutige ID, die das System von 1 an aufsteigend vergibt
- A5 Jede Nachricht besitzt neben der ID eine Überschrift, einen Nachrichtentext, ein Erstellungsdatum und einen Autor

A6 Ein Administrator soll Nachrichten unter Angabe der Nachrichten-Id aus dem System löschen können

A7 Trägt ein Administrator eine neue Nachricht in das System ein, informiert das System alle registrierten Benutzer per Email über die neue Nachricht

Es gelten die folgenden Prämissen:

P1 Ein Benutzer- und Rechtesystem ist vorhanden und kann für das System genutzt werden. Eine detaillierte Schnittstellenbeschreibung liegt vor

P2 Eine Persistenzkomponente ist vorhanden und kann für das System über eine genau spezifizierte Schnittstelle verwendet werden

P3 Eine Emailkomponente ist vorhanden und kann für das System über eine genau spezifizierte Schnittstelle verwendet werden

4.1.2. Fachliches Datenmodell

Aus den Anforderungen des letzten Abschnitts ergibt sich das fachliche Datenmodell, welches Abbildung 4.1 zeigt.

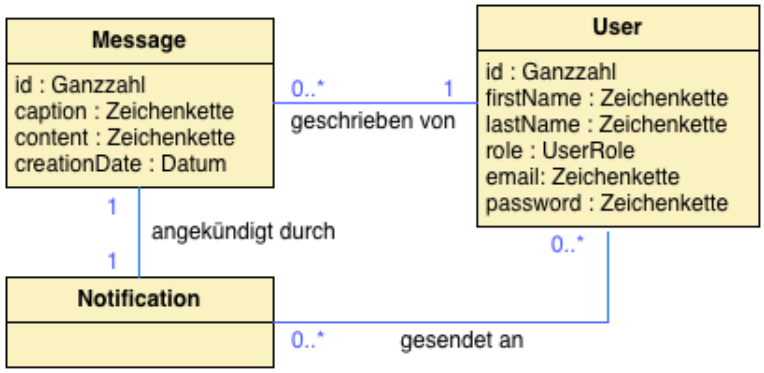

Abbildung 4.1.: Fachliches Datenmodell des News-Systems

Die `Message` ist die primäre Datenstruktur im Modell. Sie beinhaltet die Attribute, welche in den Anforderungen 4 und 5 genannt werden. Der Einfachheit halber wird auf die Einführung von spezifischen Id-Typen verzichtet. Eine `Message` wird von genau einem `User` verfasst.

Die Beschreibung der Entität `User` wurde aus der Schnittstellenbeschreibung des Benutzer- und Rechtesystems abgeleitet (siehe Prämisse 1). Die Rolle eines `User`s kann entweder „Administrator“ oder „Standarduser“ sein. Die Email-Adresse des Users wird der Einfachheit halber als Zeichenkette gespeichert und auf eine verschlüsselte Speicherung des Passwortes wird verzichtet. Ein `User` kann beliebig viele Nachrichten verfassen, vorausgesetzt er gehört der Rolle „Administrator“ an.

Eine `Message` wird durch genau eine `Notification` angekündigt und diese Bezieht sich auf genau eine `Message`. Eine `Notification` wird an beliebig viele `User` versendet (so viele wie im Benutzersystem registriert sind) und ein `User` kann beliebig viele `Notification`s erhalten.

4.2. Fachlicher Entwurf

Der Entwurf für das Beispielszenario hat nicht den Anspruch die spezifizierten Anforderungen bestmöglich umzusetzen, sondern soll vielmehr möglichst viele Aspekte der zu entwerfenden Sprache abdecken. Abbildung 4.2 zeigt das Komponentendiagramm des im letzten Abschnitt spezifizierten News-Systems.

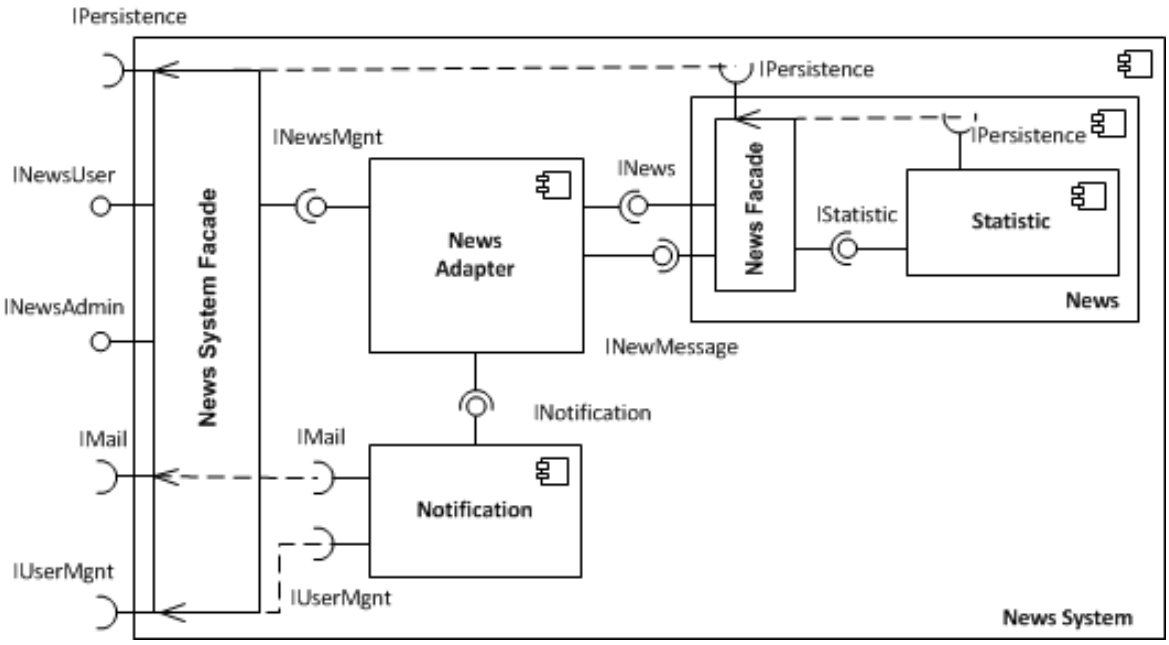

Abbildung 4.2.: Komponentendiagramm des News-Systems

Das zu entwerfende News-System besteht aus einer Komponente, welche die Schnittstellen INewsUser und INewsAdmin anbietet. Über die Schnittstelle INewsUser ist es möglich, die neusten X Nachrichten abzufragen, sowie sich über die Gesamtzahl an Nachrichten zu informieren. Die Schnittstelle INewsAdmin kann zum Hinzufügen und Löschen von Nachrichten verwendet werden.

Bevor die Komponente betriebsbereit ist, muss sie zunächst mit Implementierungen der Schnittstellen IPersistence, IMail und IUserMgnt konfiguriert werden. Bei diesen drei Schnittstellen handelt es sich um Standardschnittstellen. Die Anbindung an die in den Prämissen genannten Komponenten erfolgt später über Adapter, welche die Schnittstellen der Nachbarsysteme auf die Standardschnittstellen unseres Systems abbilden.

Die NewsSystem Komponente definiert die Schnittstelle INewsMgnt, die sie anfordert. Die Komponente NewsAdapter bildet die von der Komponente News angebotene Schnittstelle INews auf die angeforderte Schnittstelle INewsMgnt ab und entkoppelt auf diese Weise die beiden Komponenten von einander.

Die News Komponente definiert die Schnittstelle IStatistic, die sie anfordert. Diese Schnittstelle wird direkt von der Statistic Komponente realisiert. Auf diese Weise bindet sich die Komponente an die News Komponente. Die News Komponente hingegen ist von der Statistic Komponente komplett entkoppelt. Beide Komponenten importieren die IPersistence Schnittstelle, da sie Zugriff auf die Persistenz benötigen. Die einzige Aufgabe der Komponente Statistic besteht im Zählen aller im System gespeicherten Nachrichten.

Die News Komponente definiert des Weiteren die Schnittstelle INewMessage, über die die Erstellung einer neuen Nachricht an die konfigurierte Komponente signalisiert wird.

Die Notification Komponente importiert die Standardschnittstelle IMail. Die Komponente bietet die Schnittstelle INotification an, über welche Email-Benachrichtigungen verschickt werden können. Der NewsAdapter entkoppelt auch die Notification von der News Komponente, indem er die Schnittstelle INotification auf die Schnittstelle INewMessage abbildet.

5. Fachlicher Entwurf der Sprache

Vor der technischen Realisierung einer domänenspezifischen Sprache mit Hilfe eines der im letzten Kapitel vorgestellten Werkzeuge steht ihr fachlicher Entwurf. In diesem Kapitel sollen die Anforderungen an eine Sprache zur komponentenbasierten Modellierung und Validierung einer Softwarearchitektur nach den Regeln der Quasar-Standardarchitektur technikunabhängig ermittelt und beschrieben werden. Hierbei stehen die Begriffe der *Komponente*, der *Schnittstelle* und der *Klasse* im Vordergrund. Diese sollen im Kapitel als die zentralen Elemente der Sprache entwickelt werden.

5.1. Komponenten

Der Begriff der Komponente steht im Zentrum der Quasar-Standardarchitektur und hat damit auch eine zentrale Bedeutung für die zu entwerfende Sprache. Aus diesem Grund soll zunächst der Komponentenbegriff für die Sprache entworfen werden. Hierbei wird vorerst die Syntax der Sprache entwickelt, aus welcher im Anschluss das semantische Modell abgeleitet wird. Zum Schluss soll dann das Scoping und die Typprüfung für die referenzierten Typen betrachtet werden.

5.1.1. Entwicklung des syntaktischen Komponentenmodells

Analysiert man die Aufgaben und Eigenschaften einer Komponente, welche im Abschnitt Komponenten in Quasar beschrieben werden, definiert sich eine Komponente durch die folgenden Eigenschaften:

- Eine Komponente unterteilt ein Softwaresystem in Untereinheiten,
- sie hat eine eindeutige Kategorie,
- sie importiert und exportiert eine beliebige Anzahl von Schnittstellen,
- sie komponiert eine beliebige Anzahl an Subkomponenten
- und sie kapselt ihre Implementierungsdetails nach außen.

Im Folgenden soll für jeden dieser Punkte untersucht werden, wie die entsprechenden Eigenschaften für die zu entwickelnde Sprache umgesetzt werden können.

Unterteilung in Untereinheiten

Die Unterteilung eines Softwaresystems in Untereinheiten bedeutet, dass sich jeder Bestandteil eines Softwaresystems genau einer Komponente zuordnen lässt. In den meisten Programmiersprachen sind diese Bestandteile Klassen und Schnittstellen. In bisherigen Softwaresystemen besteht die einzige Strukturierungsmöglichkeit in der Organisation der Klassen und Schnittstellen in virtuellen Ordnern oder Bibliotheken. Die Information über die Komponentenzugehörigkeit ergibt sich daher nur implizit aus dem (virtuellen) Pfad zur Klasse oder Schnittstelle innerhalb des Projektes.

Der erste Schritt bei der Einführung eines Komponentenbegriffs besteht demnach darin, die Zugehörigkeit zu einer Komponente als explizite Information in die Semantik der Sprache aufzunehmen. Als Vorlage kann hierfür die Realisierung von Klassen und Schnittstellen verwendet werden. Klassen und Schnittstellen werden an einer Stelle im Quellcode mit einem eindeutigen Bezeichner definiert und dann an vielen Stellen verwendet. Es wird neben der Klasse und der Schnittstelle als weitere Struktur die Komponente eingeführt. Auf diese Weise kann jede Klasse und Schnittstelle im Softwaresystem eine Komponente als Zeichen ihrer Zugehörigkeit referenzieren.

```
1 component News {
2   ...
3 }
```

Listing 5.1: Definition einer Komponente

```
1 component News
2 public interface INews {
3   ...
4 }
```

Listing 5.2: Schnittstelle, die innerhalb von *News* definiert wurde

Kategorien von Komponenten

Die Quasar-Standardarchitektur gibt vor, dass jede Komponente genau einer Software-Kategorie angehört. Welche Kategorie eine Komponente besitzt, ist eine Entwurfsentscheigung und zur Zeit der Komponentendefinition bekannt. Es bietet sich daher an, einer Komponente neben einem eindeutigen Bezeichner auch eine Kategorie zuzuweisen.

```
1 component News of category A {
2   ...
3 }
```

Listing 5.3: Definition einer Komponente mit Kategorie

Import und Export von Schnittstellen

Die Informationen, welche eine Komponentendefinition enthält, unterscheiden sich von denen der Schnittstellen und Klassen. Innerhalb der Komponentendefinition muss angegeben werden, welche Schnittstellen von einer Komponente exportiert werden und für welche Schnittstellen die Komponente mit Implementierungen konfiguriert werden muss. Gibt man bei importierten Schnittstellen zusätzlich die Komponente an, mit der die importierende Komponente konfiguriert werden soll, besteht außerdem die Möglichkeit die Konfiguration der Komponente zu generieren. Für eine leichtere Rekonfiguration der Zielkomponente bietet es sich an, die Konfiguration der Elternkomponente zu verwenden, falls diese die selbe Schnittstelle importiert. Auf diese Weise lassen sich Implementierungsänderungen für Schnittstellen an einer Stelle vornehmen.

```
1 component News of category A {
2   export INews
3   import INewMessage [ NewsAdapter ]
4 }
```

Listing 5.4: Definition einer Komponente mit exportierten und importierten Schnittstellen

Komposition von Komponenten

Jede Komponente darf beliebig viele Subkomponenten beinhalten, welche wiederum aus Subkomponenten bestehen können. Dieses Konzept ist mit der Verhaltensvererbung von Java Interfaces vergleichbar. Hierbei ist es einem Interface möglich, von beliebig vielen anderen Interfaces zu erben, welche wiederum von beliebig vielen weiteren Interfaces erben können. Überträgt man dieses Konzept auf die Definition von Komponenten, erhält man das Ergebnis, das Listing 5.5 zeigt.

```
1 component NewsSystem of category A composes  News, Notification, NewsAdapter {
2   export INewsUser
3   ...
4   import IPersistence [ PersistenceAdapter ]
5 }
```

Listing 5.5: Definition einer Komponente mit Subkomponenten

Kapselung der Implementierungsdetails

Wie im Abschnitt Geheimnisprinzip erwähnt wurde, eignet sich die Verwendung einer Fassade zur Kapselung von Implementierungsdetails. Jede Komponente sollte daher eine Fassade beinhalten, die alle exportierten Schnittstellen implementiert und die angebotene Funktionalität der Schnittstellen durch Delegation an Subkomponenten realisiert.

```
1 component News of category A composes  Statistic {
2   facade NewsFacade {
3     export INews
4   }
5
6   import INewMessage [ NewsAdapter ]
7   ...
8 }
```

Listing 5.6: Definition einer Komponente mit Fassade

5.1.2. Entwicklung des semantischen Komponentenmodells

Aus der im letzten Abschnitt entwickelten Syntax für die Definition einer Komponente lässt sich das semantische Modell ableiten, welches Abbildung 5.1 zeigt. Die Modelle der Schnittstelle sowie der Klasse werden im nächsten Abschnitt entwickelt und sind deshalb an dieser Stelle vorerst als Platzhalter vermerkt.

5.1.3. Scoping und Typprüfung bei Komponenten

In diesem Abschnitt soll die Validierung von Referenzen innerhalb einer Komponente erläutert werden. Wie im Abschnitt Scoping und Typprüfung beschrieben, gibt es hierfür zwei Möglichkeiten: die Bildung von Scopes und die Durchführung einer Typprüfung.

Die Bildung von Scopes spielt für die Modellierung einer Architektur, nach den Regeln der Quasar-Standardarchitektur, eine entscheidende Rolle. Über die Einschränkung der Sichtbarkeit von deklarierten Komponenten, Schnittstellen und Klassen lassen sich sehr leicht Abhängigkeiten kontrollieren. Auf diese Weise ist es möglich unzulässige Referenzen auf Komponenten, Schnittstellen und Klassen gar nicht erst zuzulassen. Die meisten Entwicklungsumgebungen zeigen dem Programmierer eine Liste der sichtbaren Strukturen, welche

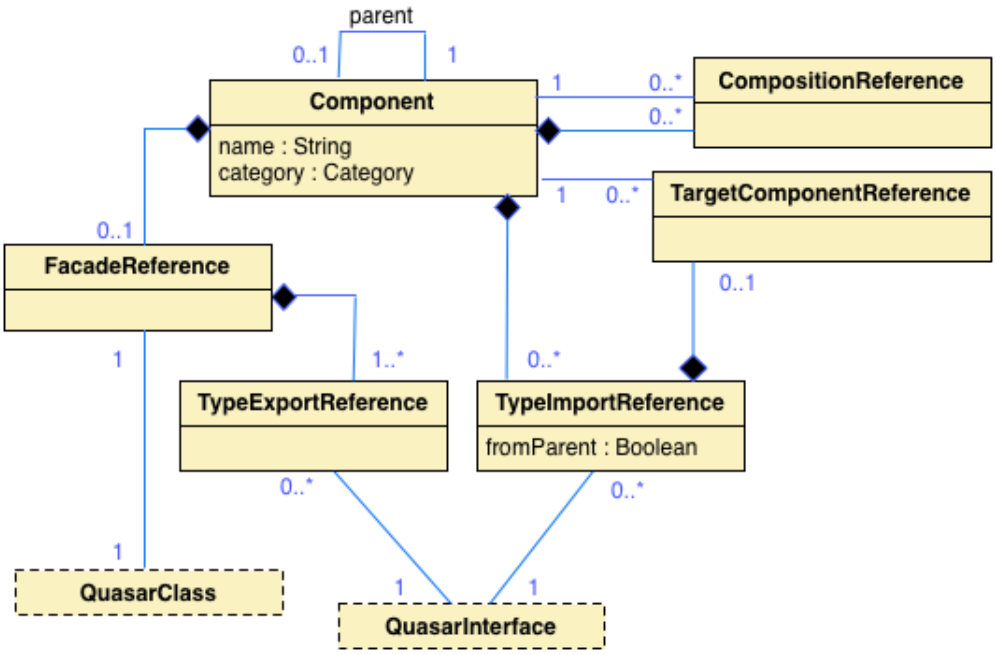

Abbildung 5.1.: Semantisches Modell der Komponente

an einer bestimmten Stelle im Quellcode referenziert werden können. Ist eine Struktur in dieser Liste nicht enthalten, so ist ihre Verwendung an dieser Stelle unzulässig. Solche Listen werden auf Basis des Scopes für den ausgewählten Context berechnet.

Im Gegensatz zur Bildung von Scopes kann bei der Typprüfung näher auf unzulässige Referenzen eingegangen werden. Schränkt man die Sichtbarkeit ein, sollte man sicher sein, dass der Programmierer den Grund kennt, wieso an einer Stelle im Code das eine Element sichtbar ist, das andere aber nicht. Ansonsten sucht der Programmierer unter Umständen eine lange Zeit nach der Ursache. Macht man auch unzulässige Referenzen sichtbar, hat man in vielen Entwicklungsumgebungen die Möglichkeit, einen Fehler mit einer genauen Beschreibung der Fehlerursache anzuzeigen. Ein weiterer Grund für die Validierung einer Referenz über Typprüfung, statt über Scopes ist in der Ausdrucksmöglichkeit der Typprüfung zu sehen. Soll beispielsweise eine Regel definiert werden, die vorschreibt, dass ein Typ nur ein einziges Mal innerhalb eines Gültigkeitsbereichs referenziert werden darf, so kann diese Regel nicht über Scopes überprüft werden. Bei der Bildung von Scopes wird die Menge aller erlaubten Referenzen ermittelt und dann überprüft, ob eine bestimmte Referenz in der Menge enthalten ist oder nicht. Aus diesem Grund ist Validierung von Kardinalitäten ausschließlich über Typprüfung möglich.

Innerhalb der Definition einer Komponente gibt es vier Arten von Referenzen:

- Referenzen auf Subkomponenten,
- Referenzen auf importierte Schnittstellen,
- Referenzen auf exportierte Schnittstellen
- und eine Referenz auf die Fassade.

Im Folgenden sollen die Regeln der Validierung für diese vier Arten von Referenzen erläutert werden.

Validierung von Komponenten für die Komposition

Wie im Abschnitt Komponenten in Quasar erläutert, muss sich jeder Komponente eine eindeutige Kategorie zuordnen lassen. Die Verwendung einer anderen als der eigenen Kategorie führt zur Vermischung und ist daher nicht zulässig. Aus diesem Grund darf eine A-Komponente keine T-Komponente als Subkomponente beinhalten und umgekehrt.

Die Benutzung von 0-Komponenten ist zwar nach den Regeln der Quasar-Standardarchitektur zulässig, macht aber für die Bildung von Subkomponenten keinen Sinn. Zweck der Bildung von Subkomponenten ist es, das Wissen um diese Komponente zu kapseln und die Subkomponente nach außen hin zu verbergen. 0-Komponenten sind Standardsoftware und daher global sichtbar. Aus diesem Grund dürfen 0-Komponenten nicht die Rolle einer Subkomponente einnehmen. Des weiteren darf eine 0-Komponente auch keine Subkomponenten beinhalten, da jede Verwendung einer anderen Kategorie außer der 0-Kategorie ihren Status als 0-Software verändern würde.

Komponenten der Kategorie R dürfen bei A-, sowie bei T-Komponenten die Rolle der Subkomponente einnehmen. Die Kategorie R ermöglicht es überhaupt erst Komponenten so zu entkoppeln, dass diese unabhängig von anderen Komponenten wiederverwendet werden können und müssen daher überall eingesetzt werden dürfen. R-Komponenten selber sollten möglichst einfach gehalten werden und im Idealfall nur eine einfache Projektion von einer Quell- auf eine Zielschnittstelle durchführen. Daher sind Subkomponenten bei Komponenten der Kategorie R nicht erlaubt.

Da die Regeln für die Validität von Kandidaten für die Komposition relativ eindeutig sind, sollen nur zulässige Komponenten für die Komposition sichtbar sein. Tabelle 5.1 verdeutlicht die erlaubten Kategorien bei der Komposition von Komponenten.

Unabhängig von der Kategorie darf jede Komponente nur maximal einmal die Rolle einer Subkomponente einnehmen. Wird eine Komponente von mehreren Komponenten benötigt, muss diese auf einer Kompositionsebene definiert werden, welche von allen importierenden Komponenten aus erreichbar ist. Die importierenden Komponenten müssen dann unter Umständen über mehrere Ebenen mit der benötigten Komponente konfiguriert werden.

	0	A	T	R
0				
A		X		X
T			X	X
R				

Tabelle 5.1.: Sichtbarkeit in Abhängigkeit der Kategorie bei der Komposition
horizontal: Kategorie der Elternkomponente
vertikal: Kategorie der Subkomponente

Validierung von Referenzen auf Schnittstellen

Für Referenzen auf zu exportierende und zu importierende Schnittstellen sind die Regeln für die Validierung zulässiger Kandidaten nicht ganz so eindeutig, wie bei der Subkomponentenbildung. Bei einer idealen Architektur, bei welcher alle Komponenten optimal von einander entkoppelt sind, dürften nur die Schnittstellen exportiert und importiert werden, welche innerhalb der Komponente oder als 0-Schnittstelle definiert worden sind. Für den Import einer Schnittstelle, welche innerhalb einer Subkomponente definiert wurde, würde dieses Vorgehen voraussetzen, dass die Subkomponente über eine R-Komponenten entkoppelt werden muss. Wendet man dieses Vorgehen durchgängig an, hätte dies eine sehr große Anzahl benötigter R-Komponenten zur Folge.

Für reale Informationssysteme ist so ein hoher Grad an Entkopplung der Komponenten nicht notwendig, da besonders bei A-Komponenten die modellierte Fachlichkeit so speziell ist, dass sie nur für ein Projekt Gültigkeit besitzt. Das Aufzwingen einer optimalen Entkopplung, welche mit zusätzlichem Aufwand verbunden ist, wäre hier der falsche Weg. Es muss daher ein Kompromiss zwischen einer ideal entkoppelten und einer praktisch angemessenen Architektur gefunden werden.

Der Ort, an dem eine Schnittstelle definiert ist, kann relativ zu einer Komponente K einer der folgenden fünf Kategorien zugeordnet werden:

- Die Schnittstelle ist innerhalb einer 0-Komponente definiert (0-Schnittstelle).
- Die Schnittstelle ist innerhalb der Komponente K definiert.
- Die Schnittstelle ist innerhalb einer Subkomponente von K definiert.
- Die Schnittstelle ist in einer Elternkomponente von K definiert.
- Die Schnittstelle ist in einer Schwesterkomponente von K definiert.

Im Folgenden sollen diese fünf Kategorien im Hinblick auf die Validität für zu exportierende und zu importierende Schnittstellen der Komponente K untersucht werden.

0-Schnittstellen: 0-Schnittstellen eignen sich sehr gut als exportierte oder importierte Schnittstelle, da diese Kategorie zwei Komponenten optimal entkoppelt. 0-Schnittstellen sind global verfügbar und erzeugen keine neuen Abhängigkeiten.

Definition innerhalb von K: Wie bereits oben erwähnt erzeugen auch Schnittstellen, welche innerhalb von K definiert wurden, keine neuen Abhängigkeiten und eignen sich damit sehr gut für den Ex- und Import.

Definition innerhalb einer Subkomponente von K: Eine Subkomponente kapselt alle Klassen und Schnittstellen ihrer Subkomponenten und verbirgt sie damit nach außen. Alle Schnittstellen der Subkomponente sind von außen sichtbar und dürfen in K importiert, jedoch auf keinen Fall ohne Indirektionsebene weiter exportiert werden. Der Export von Schnittstellen, welche innerhalb von Subkomponenten definiert sind, würde Abhängigkeiten auf die Interna von K erzeugen, was unbedingt zu vermeiden ist. Der direkte Import einer Schnittstelle, welche innerhalb einer Subkomponente definiert wurde, erzeugt eine Abhängigkeit zu dieser und führt dazu, dass sie nicht mehr ohne Weiteres ausgetauscht werden kann. Der Import der Schnittstelle einer Subkomponente ist aber als weniger kritisch zu bewerten als der Export, da sich die entstehende Abhängigkeit auf die importierende Komponente beschränkt und sich nicht, wie beim Export, auf größere Teile des Systems auswirken kann.

Definition innerhalb einer Elternkomponente von K: Der Im- und Export von Schnittstellen aus Elternkomponenten bindet die Komponente an ihre Umgebung und verhindert die Wiederverwendbarkeit der Komponente. Realisiert die Komponente einen sehr speziellen Aufgabenbereich, was eine Wiederverwendung unwahrscheinlich macht, kann diese Abhängigkeit in Kauf genommen werden. Wenn sehr viele Schnittstellen aus übergeordneten Kompositionsebenen verwendet werden, ist es empfehlenswert zu überprüfen, ob die Komponente nicht auf einer höheren Kompositionsebene definiert werden sollte.

Definition innerhalb einer Schwesterkomponente von K: Die Schwesterkomponenten von K kapseln ähnlich wie K selber ihre Interna und verbergen sie nach außen. Die Schnittstellen der Schwesterkomponenten sind allerdings von außen sichtbar und können damit importiert und exportiert werden. Auch hier bedeutet eine Referenz auf die Schnittstelle, welche innerhalb einer Schwesterkomponente definiert wurde, eine direkte Abhängigkeit zu dieser Komponente und macht sie nur gemeinsam wiederverwendbar. Insbesondere zyklische Abhängigkeiten zwischen Komponenten sollten vermieden werden.

Schnittstellen sollen für die zu entwerfende Sprache mit Hilfe von Scopes validiert werden, da keine starke Einschränkung der Sichtbarkeit durchgeführt wird:

- Sichtbar sind alle Komponentenschnittstellen der selben Kategorie oder der Kategorie 0, welche nicht von Subkomponenten gekapselt werden.

- Eine Ausnahme bilden die Komponentenschnittstellen der direkten Subkomponenten. Diese sind in der Elternkomponente sichtbar.
- Des weiteren dürfen die Schnittstellen von Subkomponenten nicht auf höhere Kompositionsebenen exportiert werden.
- Der Export von Schnittstellen mit Kopplungsgrad 4 ist nicht zulässig.

Die Bedeutung des Kopplungsgrades für Schnittstellen wird im Zusammenhang mit der Definition der Schnittstelle im nächsten Abschnitt erläutert werden.

Validierung von Klassen für die Rolle als Fassade

Die Regeln für die Sichtbarkeit von Klassen, welche als Kandidat für die Rolle als Fassade in Frage kommen sind eindeutig. Als Fassade eignen sich alle Klassen, welche innerhalb der Komponente deklariert wurden und sich nicht innerhalb von Subkomponenten befinden.

Für jede Fassade soll mittels Typprüfung ermittelt werden, ob die Klasse alle exportierten Schnittstellen der Komponente implementiert.

5.2. Schnittstellen

Neben der Komponente ist die Schnittstelle ein weiterer wichtiger Begriff im Kontext der Quasar-Standardarchitektur. Im Gegensatz zum Begriff der Komponente existiert der Begriff der Schnittstelle schon in sehr vielen Programmiersprachen. Für die Verwendung in Kombination mit Komponenten ist es jedoch notwendig das Modell der Schnittstelle zu erweitern. Hierbei soll die Verwendung der Schnittstelle in Kombination mit der Komponente im Vordergrund stehen. Elemente des Kontextes Vertragsbasierter Entwurf sollen hierbei keine Berücksichtigung finden.

Wie auch schon bei dem Entwurf der Komponente wird zunächst die Syntax entwickelt und aus dieser dann das semantische Modell abgeleitet. Zum Schluss soll dann auf die Themen Scoping und Typprüfung eingegangen werden.

5.2.1. Entwicklung des syntaktischen Schnittstellenmodells

Betrachtet man den Abschnitt Schnittstellen in Quasar im Hinblick auf die Verwendung von Schnittstellen im Kontext von Komponenten, so lässt sich eine Schnittstelle durch die folgenden Eigenschaften beschreiben:

- Eine Schnittstelle definiert Methodensignaturen und deren Rückgabetypen.
- Eine Schnittstelle wird innerhalb einer Komponente definiert,
- für welche sie eine spezifische Funktion erfüllt.
- Andere Komponenten koppeln sich bei der Verwendung an sie.

Im Folgenden soll für diese Eigenschaften untersucht werden, inwieweit sie sich in die zu entwerfende Sprache integrieren lassen.

Methodensignaturen und Rückgabewerte

Die Hauptaufgabe einer Schnittstelle besteht in der Definition von Methodensignaturen und deren Rückgabetypen, welche von implementierenden Klassen realisiert werden müssen. Die Syntax für Schnittstellen der zu entwerfenden Sprache soll auf der Syntax für die Definition von Java-Schnittstellen aufbauen, welche in dieser Arbeit nicht näher beschrieben werden soll. Die Möglichkeit zur Schachtelung von Schnittstellen soll der Einfachheit halber nicht unterstützt werden.

```
1 public interface INews extends IStatistic {
2
3   List<IMessage> getLatestMessages(int number);
4   ...
5 }
```

Listing 5.7: Definition eines Java-Interfaces

Zugehörigkeit zu Komponenten

Wie im Abschnitt Unterteilung in Untereinheiten beschrieben, muss jede Schnittstelle im Softwaresystem einer Komponente zugeordnet werden. Die Komponente, in deren Kontext die Schnittstelle definiert wurde, ist die definierende Komponente und wird auch Besitzer der Schnittstelle genannt. Listing 5.2 gibt ein Beispiel für die Zuordnung einer Schnittstelle zu einer Komponente.

Schnittstellenarten

Die Art einer Schnittstelle richtet sich nach der Funktion, welche sie erfüllt. Schnittstellen können einer von vier unterschiedlichen Schnittstellenarten angehören. In Abschnitt Definition von Schnittstellen werden die Funktion der Standardschnittstelle und die der angeforderten

/ angebotenen Schnittstelle beschrieben. Hinzu kommt eine vierte Funktion für Schnittstellen, welche keine der genannten drei Funktionen erfüllt. Diese Funktion erhält den Namen *Klassenschnittstelle*. Tabelle 5.2 gibt einen Überblick über die Schnittstellenarten.

Funktion	Komponentenschnittstelle	Schlüsselwort
Standardschnittstelle	(X)	standard
angebotene Schnittstelle	X	offer
angeforderte Schnittstelle	X	require
Klassenschnittstelle		<none>

Tabelle 5.2.: Übersicht über die Schnittstellenarten

```
1 component News
2 offer public interface INews extends IStatistic {
3
4   List<IMessage> getLatestMessages(int number);
5   ...
6 }
```

Listing 5.8: Definition einer Schnittstelle mit Funktion

Listing 5.8 zeigt ein Beispiel für die Definition einer angebotenen Komponentenschnittstelle. `INews` wird als angebotene Schnittstelle der Komponente `News` definiert. Das bedeutet, dass `News` `INews` exportiert. Durch den Export von `INews` wird auch implizit die Schnittstelle `IStatistic` exportiert. Die Schnittstelle `IMessage` wurde innerhalb von `News` definiert und ist keine Komponentenschnittstelle. Damit hat sie die Funktion Klassenschnittstelle.

Kopplungsgrad von Schnittstellen

Der Kopplungsgrad einer Schnittstelle gibt an, wie stark sich der Benutzer einer Schnittstelle an den Besitzer der Schnittstelle bindet. Die Quasar-Standardarchitektur definiert keine Skala für den Kopplungsgrad zwischen Benutzer und Besitzer einer Schnittstelle. Aus diesem Grund soll an dieser Stelle eine Skala für den Grad der Kopplung definiert werden. Tabelle 5.3 gibt einen Überblick über die fünf Kopplungsgrade.

Für jede Schnittstelle soll bei ihrer Definition der maximal zulässige Kopplungsgrad angegeben werden. Eine Schnittstelle vom Kopplungsgrad 2 darf beispielsweise auch Standardschnittstellen verwenden. Das Referenzieren von Schnittstellen, welche innerhalb der selben

Wert	Standardschnittstelle	Verwendete Typen in der Hülle
0	X	Standardschnittstellen
1		Standardschnittstellen
2		Schnittstellen höherer Kompositionsebenen
3		Vom Besitzer definierte Schnittstellen
4		Schnittstellen von Subkomponenten und Klassen

Tabelle 5.3.: Skala für den Kopplungsgrad von Schnittstellen

Komponente definiert wurden, ist ihr aber untersagt. Die Verwendung von Klassen als Typ in Schnittstellen ist nur innerhalb einer Komponente zulässig. Schnittstellen vom Kopplungsgrad 4 dürfen daher außerhalb einer Komponente nicht referenziert werden.

Listing 5.9 zeigt ein Beispiel für die Definition einer Schnittstelle mit einem maximalen Kopplungsgrad von 3.

```
1 component News
2 offer public interface INews with coupling 3 extends IStatistic {
3
4   List<IMessage> getLatestMessages(int number);
5   ...
6 }
```

Listing 5.9: Definition einer Schnittstelle mit Kopplungsgrad

5.2.2. Entwicklung des semantischen Schnittstellenmodells

Aus der im letzten Abschnitt entwickelten Syntax für die Definition einer Schnittstelle lässt sich das semantische Modell ableiten, welches Abbildung 5.2 zeigt. Die Quasar-Schnittstelle erbt von der Java-Schnittstelle, deren Semantik an dieser Stelle nicht näher beschrieben werden soll.

5.2.3. Scoping und Typprüfung bei Schnittstellen

Möchte man eine Architektur modellieren, welche zu den Regeln der Quasar-Standardarchitektur konform ist, muss darauf geachtet werden, dass nur zulässige Typen innerhalb der Hülle einer Schnittstelle referenziert werden. In diesem Abschnitt sollen die Validierungsregeln für die Referenzen einer Schnittstelle entwickelt werden.

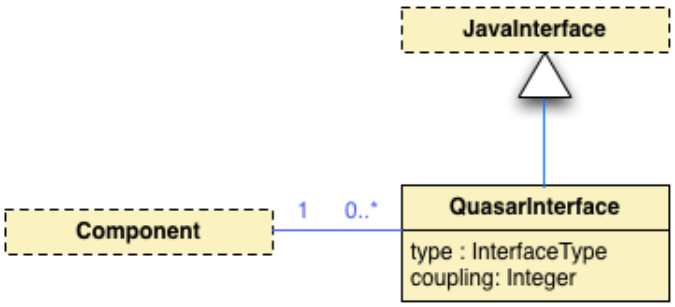

Abbildung 5.2.: Semantisches Modell der Schnittstelle

Innerhalb der Definition einer Schnittstelle gibt es zwei Arten von Referenzen:

1. Eine Referenz auf die Komponente, welche die Schnittstelle definiert.
2. Und Referenzen auf Typen innerhalb der Hülle der Schnittstelle.

Im Folgenden sollen die Regeln der Validierung für diese zwei Arten von Referenzen erläutert werden.

Validierung der definierenden Komponente

Die Kategorie der definierenden Komponente bestimmt, wie im Abschnitt Schnittstellen-Kategorien beschrieben, die Kategorie der Schnittstelle. Die Kategorie der Schnittstelle bedingt unter anderem, welche Typen in der Hülle einer Schnittstelle referenziert werden dürfen. Es gibt zwei mögliche Vorgehensweisen bei der Definition einer Schnittstelle:

1. Eine Schnittstelle wird im Kontext einer Komponente definiert. Die Zugehörigkeit einer Schnittstelle zu einer Komponente ist frei wählbar, beeinflusst aber die Auswahl an Typen, welche in der Hülle verwendet werden können.
2. Eine Schnittstelle wird definiert ohne sie von Beginn an einer Komponente zuzuweisen. Die verwendeten Referenzen innerhalb der Hülle der Schnittstelle bestimmen dann, welcher Komponente die Schnittstelle zugeordnet werden kann.

Für die zu entwerfende Sprache soll die erste Vorgehensweise umgesetzt werden, da davon ausgegangen wird, dass sich die Definition einer Schnittstelle immer auf eine konkrete Komponente bezieht, welche die zu definierende Funktionalität entweder benötigt oder anbietet. Dies führt dazu, dass eine Schnittstelle jeder Komponente zugeordnet werden kann.

Als Einschränkung wird definiert, dass Komponenten der Kategorie R keine Schnittstellen besitzen können. Die Aufgabe von Komponenten der Kategorie R ist es, Komponenten anderer Kategorien zu entkoppeln, indem die notwendigen Abhängigkeiten in ihnen gebündelt werden. Keine Komponente darf daher von einer R-Komponente abhängig sein. Würde eine R-Komponente eine Schnittstelle definieren, wäre dies aber der Fall. Aus diesem Grund wird die Definition von R-Schnittstellen verboten.

Validierung der Typen der Schnittstellenhülle

Zur Hülle einer Schnittstelle gehören alle Typen, welche in der Schnittstelle referenziert werden. Für die Entwicklung der Validierung dieser Typen werden die Referenzen zunächst in zwei Bereiche unterteilt:

- Referenzen auf die Schnittstellen, welche durch Verhaltensvererbung erweitert werden.
- Referenzen auf Rückgabetypen, Parametertypen und Generics.

Der nachfolgende Abschnitt soll die Kriterien für valide Referenzen dieser zwei Bereiche beschreiben.

Referenzen auf erweiterte Schnittstellen: Handelt es sich bei der erweiternden Schnittstelle um eine Komponentenschnittstelle, so müssen auch die Schnittstellen, von denen geerbt wird, Komponentenschnittstellen sein. Gleiches gilt für Klassenschnittstellen. Tabelle 5.4 beschreibt die zulässigen Schnittstellenarten in Relation zur erweiternden Schnittstelle.

Art der Schnittstelle	Art der Referenz
standard	standard
angeboten	angefordert / angeboten / standard
angefordert	angefordert / angeboten / standard
<none>	<none> / standard

Tabelle 5.4.: Valide Schnittstellenarten bei der Verhaltensvererbung

Neben der Schnittstellenart muss auch die Kategorie der Schnittstelle überprüft werden. Schnittstellen der Kategorie A, T und 0 dürfen jeweils Schnittstellen ihrer eigenen Kategorie erweitern. Zusätzlich dürfen 0 Schnittstellen von jeder Kategorie erweitert werden. R-Schnittstellen gibt es aus den im Abschnitt Validierung der definierenden Komponente beschriebenen Gründen nicht.

Für den Kopplungsgrad der erweiternden Schnittstelle muss gelten, dass dieser mindestens genau so hoch oder höher als der höchste Kopplungsgrad der erweiterten Schnittstellen sein muss. Wie schon im Abschnitt Kopplungsgrad von Schnittstellen erwähnt, dürfen Schnittstellen mit Kopplungsgrad 4 nur komponentenintern verwendet werden. Die Erweiterung von nicht komponenteneigenen Schnittstellen der Kategorie 4 ist daher nicht zulässig.

Die Validierung der Referenzen innerhalb einer Schnittstelle soll über Scoping erfolgen. Dies bedeutet, dass nur die Referenzen sichtbar sind, welche im entsprechenden Kontext referenziert werden können

Referenzen auf Rückgabetypen, Parametertypen und Generics: Für Referenzen auf Rückgabe- und Parametertypen, sowie Generics gelten ähnliche Regeln wie für den Im- und Export von Komponentenschnittstellen, welche im Abschnitt Validierung von Referenzen auf Schnittstellen beschrieben wurden:

- Es dürfen alle Schnittstellen der selben Kategorie oder der Kategorie 0 referenziert werden,
- welche nicht von Subkomponenten gekapselt werden
- und einen Kopplungsgrad größer gleich 3 besitzen.
- Schnittstellen von direkten Subkomponenten und Klassen dürfen nur in komponenteninternen Schnittstellen mit Kopplungsgrad 4 referenziert werden.

Die Verwendung von Typen, welche in Subkomponenten definiert wurden, führt zu einer direkten Abhängigkeit zur Subkomponente. Für den Fall, dass die Subkomponente ausgetauscht werden soll, werden damit Änderungen an der Elternkomponente notwendig. Würde die Elternkomponente die Typen der Subkomponente auf höhere Kompositionsebenen hochreichen, wären auch hier Änderungen notwendig, was auf jeden Fall vermieden werden sollte. Aus diesem Grund sind Referenzen auf Typen aus Subkomponenten nur in komponenteninternen Schnittstellen des Kopplungsgrades 4 zulässig.

Referenzen auf Komponentenschnittstellen sollten eine Ausnahme bilden, da der Austausch von Komponentenobjekten zur Laufzeit nur zum Zwecke einer dynamischen Rekonfiguration verwendet werden sollte. Eine dynamische Rekonfiguration von Subkomponenten ist nur dann möglich, wenn keine Abhängigkeiten zu den Subkomponenten bestehen.

5.3. Klassen

Neben der Komponente und der Schnittstelle soll die zu entwickelnde Sprache als drittes und letztes die Definition von Klassen unterstützen. Anders als bei der Komponente und der

Schnittstelle müssen für die Klasse keine umfangreichen Spracherweiterungen der zugrundeliegenden Java-Umsetzung vorgenommen werden. Der Vollständigkeit halber soll aber auch die Klasse nach dem selben Vorgehen, wie die Komponente und Schnittstelle entwickelt werden. Zunächst soll das syntaktische Modell entwickelt werden, woraus dann im Anschluss das semantische Modell abgeleitet wird. Zum Schluss soll auch hier auf die Themen Scoping und Typprüfung eingegangen werden.

5.3.1. Entwicklung des syntaktischen Klassenmodells

Die Klasse findet als Bestandteil einer Softwarearchitektur, anders als die Komponente und Schnittstelle, nur am Rande Erwähnung. Als Grundlage der Klassendefinition soll die Java-Umsetzung der Klasse dienen, deren Syntax in dieser Arbeit als bekannt vorausgesetzt wird. Das syntaktische Modell der Klasse soll im Folgenden um die Zugehörigkeit zu einer definierenden Komponente sowie um die Zugriffsmöglichkeit auf importierte Schnittstellen erweitert werden.

Zugehörigkeit zu Komponenten

Wie im Abschnitt Unterteilung in Untereinheiten beschrieben, muss jede Klasse im Softwaresystem einer Komponente zugeordnet werden. Die Komponente, in deren Kontext die Klasse definiert wurde, ist die definierende Komponente und wird auch Besitzer der Klasse genannt. Listing 5.10 zeigt die Definition einer Klasse innerhalb der Komponente `News`.

```
1 component News
2 public class NewsFacade implements INews {
3   ...
4 }
```

Listing 5.10: Klasse, die innerhalb von *News* definiert wurde

Zugriff auf importierte Schnittstellen

Innerhalb der Klassen wird die Anwendungslogik des zu entwickelnden Systems implementiert. Damit eine Klasse die Dienste anderer Komponenten über die importierten Schnittstellen ihrer Komponente in Anspruch nehmen kann, benötigt sie eine Zugriffsmöglichkeit auf diese. Für diesen Zweck wird der Accessor eingeführt. Listing 5.11 zeigt die Deklaration eines Accessors für die von der News-Komponente importierten Schnittstelle `IStatistic`. In der Methode `getTotalNumberOfMessages` wird dann die angebotene Funktionalität in Anspruch genommen.

```
component News
public class NewsFacade implements INews {
  accessor IStatistic statistic;
  ...
  public int getTotalNumberOfMessages() {
    statistic.getTotalNumberOfMessages();
  }
  ...
}
```

Listing 5.11: Deklaration eines Accessors innerhalb einer Klasse

5.3.2. Entwicklung des semantischen Klassenmodells

Aus der im letzten Abschnitt entwickelten Syntax für die Definition einer Klasse lässt sich das semantische Modell ableiten, welches Abbildung 5.3 zeigt. Die Quasar-Klasse erbt von der Java-Klasse, deren Semantik an dieser Stelle nicht näher beschrieben werden soll.

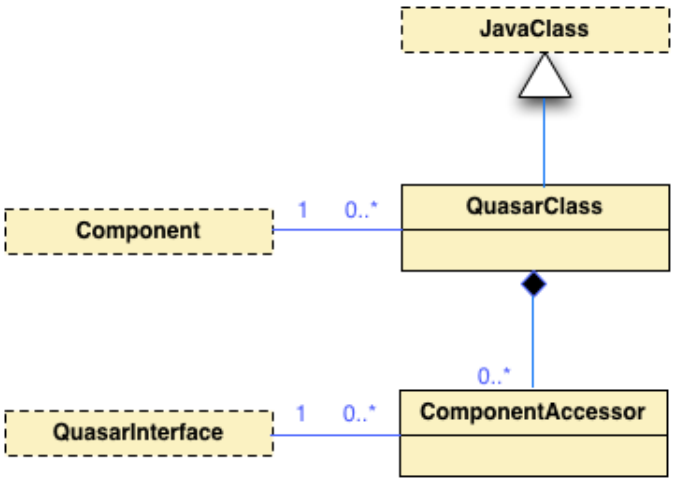

Abbildung 5.3.: Semantisches Modell der Klasse

5.3.3. Scoping und Typprüfung bei Klassen

Typ-Referenzen innerhalb einer Klasse beeinflussen die Qualität der Architektur nicht so stark wie Referenzen in Komponenten oder Schnittstellen, da Klassen nur komponentenintern verwendet werden und daher durch Typ-Referenzen keine neuen Abhängigkeiten von

anderen Komponenten zur definierenden Komponente der Klasse entstehen können. Typ-Referenzen in Klassen haben aus diesem Grund keinen Einfluss auf die Austauschbarkeit einer Komponente, sondern beeinflussen lediglich den Grad der Wiederverwendbarkeit. Vergleichbar zur Schnittstelle gibt es bei der Klasse zwei Arten von Referenzen:

- Eine Referenz auf die Komponente, welche die Klasse definiert.
- Und Typ-Referenzen innerhalb des Körpers der Klasse.

Im Folgenden sollen die Regeln der Validierung für diese zwei Arten von Referenzen erläutert werden.

Validierung der definierenden Komponente

Die Kategorie einer Klasse wird vergleichbar zur Schnittstellenkategorie durch die definierende Komponente bestimmt. Die Kategorie der Klasse bestimmt, welche Typ-Referenzen innerhalb der Klasse zulässig sind. Es wird davon ausgegangen, dass die Zugehörigkeit einer Klasse bei ihrer Definition festgelegt wird und sich danach nicht mehr ändert. Die definierende Komponente für eine Klasse kann daher frei gewählt werden. Für die definierende Komponente ist aus diesem Grund keine Validierung notwendig.

Validierung der Typ-Referenzen im Klassenkörper

Für Typ-Referenzen innerhalb einer Klassendefinition gelten folgende Regeln:

- Es dürfen alle Schnittstellen der selben Kategorie oder der Kategorie 0 referenziert werden, welche nicht von Subkomponenten gekapselt werden.
- Schnittstellen von direkten Subkomponenten dürfen referenziert werden, wenn ihr Kopplungsgrad nicht größer als 3 ist.
- Es dürfen alle Klassen der definierenden Komponente referenziert werden.

6. Technische Realisierung der Sprache

Ziel dieses Kapitels ist die Beschreibung der technischen Realisierung des fachlichen Sprachentwurfs aus dem vorherigen Kapitel. Hierfür soll zunächst eins der Frameworks, welche im Abschnitt Frameworks für die Entwicklung domänenspezifischer Sprachen vorgestellt wurden, unter Abwägung der Vor- und Nachteile ausgewählt werden. Im Folgenden soll dann die technische Realisierung der im letzten Kapitel entworfenen Sprache, mit Hilfe des ausgewählten Frameworks, erläutert werden.

6.1. Technologieauswahl

Für die Implementierung des Sprachentwurfes stehen zwei Frameworks zur Verfügung: Xtext und das Meta Programming System. Bei beiden Frameworks bestehen Vor- und Nachteile für den Einsatz zur Implementierung, die im Folgenden gegeneinander abgewägt werden sollen.

6.1.1. Gründe für den Einsatz von Xtext

Für den Einsatz von Xtext sprechen die folgenden Punkte:

- Einfache Beschreibung der Grammatik in einer EBNF ähnlichen Form .
- Xtext generiert neben einem Parser ein Plugin für Eclipse, mit dem sich die DSL komplett in Eclipse integrieren lässt.
- Die Funktionalität von Xtext lässt sich leicht erweitern und anpassen.

6.1.2. Gründe gegen den Einsatz von Xtext

Gegen den Einsatz von Xtext sprechen die folgenden Punkte:

- Es ist nicht möglich eine vorhandene Sprache modular zu erweitern. Die zu erweiternde Sprache muss komplett in Xtext rekonstruiert werden, bevor sie erweitert werden kann.
- Bei einer Spracherweiterung muss darauf geachtet werden, dass die Grammatik der erweiterten Sprache in Verbindung mit den Spracherweiterungen ihre Eindeutigkeit behält.
- Möchte man eine IDE-Unterstützung für seine Sprache erstellen, so ist man auf die Verwendung von Eclipse festgelegt.

6.1.3. Gründe für den Einsatz von MPS

Für den Einsatz von MPS sprechen die folgenden Punkte:

- MPS wurde für modulare Spracherweiterungen entworfen, weshalb diese ohne viel Aufwand realisiert werden können.
- MPS besitzt eine Implementierung der Programmiersprache Java, die als Grundlage für eine Spracherweiterung genutzt werden kann.
- Es muss keine Grammatik für die Spracherweiterung entwickelt werden, da der abstrakte Syntaxbaum direkt bei der Eingabe erzeugt wird.

6.1.4. Gründe gegen den Einsatz von MPS

Für den Einsatz von MPS sprechen die folgenden Punkte:

- MPS arbeitet nicht wie herkömmliche Frameworks zur Erstellung von DSLs auf Basis von textuellen Repräsentationen. Dies führt dazu, dass man sich an MPS als Entwicklungsumgebung bindet.
- Die Versionierung des Quellcodes erfolgt in MPS auf Basis von Modellen (vergleichbar zu Java-Packages), welche als XML-Dateien verwaltet werden. Konflikte müssen daher in MPS zwischen Packages behoben werden.
- In MPS existiert bisher nur eine Generator-Implementierung für die Programmiersprache Java. Möchte man eine andere Sprache als Java generieren, muss zuvor die Zielsprache samt Generator implementiert werden.

6.1.5. Abwägung

Bei der spezifizierten Sprache handelt es sich um eine domänenspezifische Spracherweiterung für Java. Aus den in Abschnitt 6.1.3 genannten Gründen bietet sich MPS daher als Framework für die Realisierung an. Im Folgenden soll für die in Abschnitt 6.1.4 genannten Nachteile analysiert werden, wie stark sich diese auf die Realisierung des Sprachentwurfes auswirken und ob Xtext für diese Punkte Alternativen anbietet.

Der erste Nachteil bei der Verwendung von MPS besteht in der Festlegung auf MPS als Entwicklungsumgebung. Für diesen Punkt liefert allerdings auch Xtext keine Alternative, denn auch bei der Verwendung von Xtext würde man sich an Eclipse als Entwicklungsumgebung binden, sofern eine IDE Integration benötigt wird. Es ist allerdings auch bei einer Verwendung von MPS für die Entwicklung eines Systems nicht nötig, das komplette System mit MPS als IDE zu entwickeln. Es ist auch möglich, nur einzelne Komponenten mit MPS und der zu realisierenden Sprache zu implementieren und diese anschließend in das Gesamtsystem zu integrieren.

Der nächste Punkt bezieht sich auf die Versionierung des Quellcodes. In Xtext ist es möglich einzelne Elemente der Sprache (Komponenten, Schnittstellen und Klassen) getrennt unter Versionskontrolle zu stellen, sofern sie sich in getrennten Dateien befinden. Der Nachteil von MPS, nur komplette Modelle unter Versionskontrolle stellen zu können, wiegt nicht so schwer, wenn man maximal eine Komponente pro Modell anlegt. Auch für den Fall, dass eine Komponente von mehreren Entwicklern entwickelt wird, hätte man immer noch die Möglichkeit, die Komponente auf mehrere Modelle zu verteilen, um Kollisionen zu vermeiden, oder die entstehenden Konflikte zu mergen.

Der dritte Punkt betrifft die Generierung von Code für die Zielsprache. MPS bietet bisher als Basissprache für Spracherweiterungen ausschließlich die Programmiersprache Java an. Auch bei diesem Punkt bietet Xtext keine Alternative, da es in dem Framework keine vorhandenen Sprachen zur Erweiterung gibt, sondern jede Sprache von Grund auf neu geschrieben werden muss.

Da es sich bei der zu realisierenden Sprache um eine Spracherweiterung handelt, fällt die Wahl des Frameworks eindeutig auf MPS, da es für genau diesen Zweck entwickelt wurde. Die Probleme, welche die Nutzung von MPS mit sich bringt, wiegen nicht so schwer wie die Probleme, die bei der Verwendung eines anderen Frameworks ohne die Möglichkeit zur modularen Spracherweiterung entstehen würden.

6.2. Realisierung mit MPS

In diesem Abschnitt soll die Realisierung des Sprachentwurfes mit Hilfe von MPS erläutert werden. Wie im Abschnitt Meta Programming System erwähnt, wird eine Spracherweiterung in MPS durch verschiedene Sprachaspekte beschrieben. Im Verlaufe dieses Abschnitts soll auf die wichtigsten Aspekte eingegangen werden.

6.2.1. Struktur der Sprache

Im Struktur-Aspekt wird das semantische Modell der Sprache modelliert. Dieses kann eins zu eins aus den UML-Diagrammen des fachlichen Entwurfes der Sprache übertragen werden. Eine Klasse wird dabei zu einem Knoten (Konzept), eine Komposition zu einer Kind-Knoten-Beziehung, eine Assoziation zu einer Referenz und ein Attribut zu einer Eigenschaft. Im Folgenden sollen die Strukturdefinitionen der Komponente und der Schnittstelle vorgestellt werden. Aus der Strukturdefinition der Klasse soll ausschließlich die Definition des Accessors näher erläutert werden. Zusätzlich zu den im fachlichen Entwurf spezifizierten Klassen sind einige weitere Konzepte nötig, auf die im Anschluss eingegangen werden soll.

Struktur der Komponente

Abbildung 6.1 zeigt die Strukturdefinition der Komponente in MPS, die sich kaum von dem semantischen Modell der Komponente (Abbildung 5.1) des Entwurfes unterscheidet.

```
concept Component extends     BaseConcept
                  implements  INamedConcept
                              IContainer
                              IHasQuasarCategory

  instance can be root: true

  properties:
  singleton : boolean

  children:
```

CompositionReference	composits	0..n	specializes:	<none>
FacadeReference	facade	0..1	specializes:	<none>
TypeImportReference	imports	0..n	specializes:	<none>

```
  references:
```

Component	parent	0..1	specializes:	<none>

Abbildung 6.1.: Strukturdefinition der Komponente

Der Name der Komponente und ihre Kategorie werden nicht direkt im Komponenten-Konzept definiert. Diese Eigenschaften erhält die Komponente durch die Implementierung der Konzept-Schnittstellen `INamedConcept` und `IHasQuasarCategory`.

Die Komponente erhält zusätzlich zu den im fachlichen Entwurf spezifizierten Eigenschaften die Eigenschaft *Singleton*. Ihre Funktion wird im Abschnitt Mechanismus zum Setzen der Abhängigkeiten erläutert.

Auf die Strukturen der Kind-Konzepte soll an dieser Stelle nicht weiter eingegangen werden, da diese dem semantischen Modell der Komponente aus dem fachlichen Entwurf entsprechen.

Struktur der Schnittstelle

Abbildung 6.2 zeigt die Strukturdefinition der Schnittstelle in MPS. Auch diese unterscheidet sich kaum von dem semantischen Modell der Schnittstelle (Abbildung 5.2) des Entwurfes.

```
concept QuasarInterface extends     Interface
                        implements  IHasQuasarCategory

  instance can be root: true

  properties:
  type     : interface_type
  coupling : couping_type

  children:
  QuasarClassifierType | extendedInterface | 0..n | specializes: | extendedInterface

  references:
  Component | component | 1 | specializes: | <none>

  concept properties:
  alias            = Interface
  shortDescription = quasar interface declaration
```

Abbildung 6.2.: Strukturdefinition der Schnittstelle

Das `QuasarInterface` erweitert die vorhandene Strukturdefinition des `Interface`s. Zusätzlich definiert das `QuasarInterface` Eigenschaften für Schnittstellen-Typ und Kopplungsgrad. Die Eigenschafts-Typen sind *Enums*, welche die im Entwurf genannten zulässigen Werte definieren.

Die Oberklasse der Quasar-Schnittstelle `Interface` definiert die Rolle `extendedInterface` mit dem Typ `ClassifierType`. Ein `ClassifierType` ist eine Referenz auf einen

`Classifier`. Abbildung 6.3 zeigt die Vererbungshierarchie für den `Classifier` in MPS. Möchte man bei der Erweiterung eines Konzeptes nicht alle Unterklassen eines Types für eine Rolle zulassen, hat man in MPS die Möglichkeit, den Typen einer Rolle der direkten Oberklasse einzuschränken. Das `QuasarInterface` schränkt den Typen der Rolle `extendedInterface` vom Typ `ClassifierType` auf den Typen `QuasarClassifierType` ein. Der `QuasarClassifierType` ist ein Subtyp des `ClassifierType`s. Seine Aufgabe wird im weiteren Verlauf noch näher erläutert werden.

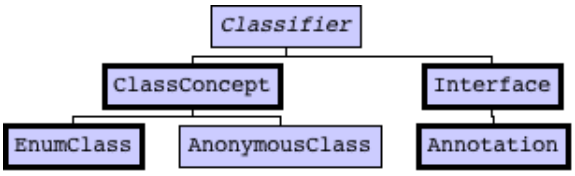

Abbildung 6.3.: Vererbungshierarchie des Classifiers

Struktur des Accessors

Abbildung 6.4 zeigt die Strukturdefinition des Accessors für importierte Schnittstellen.

```
concept ComponentAccessor extends     FieldDeclaration
                          implements  <none>

  instance can be root: false

  properties:
  << ... >>

  children:
  AccessorType  type            1     specializes:  type
  Statement     afterInitialize 0..1  specializes:  <none>

  references:
  << ... >>

  concept properties:
  dontSubstituteByDefault
```

Abbildung 6.4.: Strukturdefinition des Accessors

Der Accessor für importierte Schnittstellen ist eine spezialisierte Felddeklaration und erweitert aus diesem Grund die Struktur der `FieldDeclaration`. Das Standardverhalten des strukturierten Texteditors ist so definiert, dass an einer Position, an der ein Konzept X eingegeben werden kann, auch alle Unterklassen von X zulässig sind. Soll ein Konzept nicht automatisch substituiert werden, so kann die Konzept-Eigenschaft `dontSubstituteByDefault` gesetzt werden.

Als Typ einer `FieldDeclaration` ist jeder Typ zulässig. Da der Accessor für importierte Schnittstellen nur Quasar-Schnittstellen referenzieren darf, muss der Typ für die Rolle `type` entsprechend eingeschränkt werden. Der `AccessorType` ist ein `QuasarClassifierType`, bei welchem der Typ für die Rolle `classifier` auf Quasar-Schnittstellen eingeschränkt wurde.

Der Accessor erhält zusätzlich einen Kindknoten der Rolle `afterInitialize`. Seine Funktion wird im Abschnitt Transformation des Accessors erläutert.

Zusätzliche Konzepte

Für die Realisierung des Sprachentwurfes ist es notwendig einige weitere Konzepte anzupassen, die nicht im fachlichen Entwurf der Sprache spezifiziert werden. Diese Konzepte werden zur Integration der Spracherweiterung in die vorhandene Basissprache benötigt und dienen der Einschränkung der Gültigkeitsbereiche von Referenzen. Die Funktion der Konzepte wird im Abschnitt Constraints und Scoping näher erläutert werden.

6.2.2. Der Editor

Im Editor-Aspekt wird der strukturierte Texteditor für eine Sprache definiert. Der strukturierte Texteditor kann als „Eingabemaske“ der Sprache verstanden werden. In ihm wird geregelt, welche Eingabe des Benutzers auf welche Rolle oder Eigenschaft des abstrakten Syntaxbaumes abgebildet wird und wie eingegebene Werte dem Benutzer präsentiert werden.

Im Folgenden soll der Editor für das Konzept der Komponente exemplarisch vorgestellt werden. Abbildung 6.5 zeigt die Editordefinition der Komponente. Die Syntax der Komponente entspricht dem Entwurf aus Listing 5.6.

```
editor for concept Component
  node cell layout:
    [-
    component FE{ name } of category F{ category } composes
        F(- % composits % /empty cell: <default> -) {
      F% facade %
      <constant>
      F(- % imports % /empty cell: <constant> -)
    }
    -]

  inspected cell layout:
    [> Singleton : { singleton } <]
```

Abbildung 6.5.: Editordefinition der Komponente

Die Klammer am Anfang (**[-**) und Ende (**-]**) der Editordefinition symbolisiert, dass sich der Editor aus einer Liste von Editorelementen zusammensetzt. Das erste Editorelement in der Liste ist die Konstante `component`. Nach der Konstante soll der Name der Komponente im Editor angezeigt werden. Zu diesem Zweck wird ein Property-Mapping auf die Eigenschaft `name` der Komponente definiert. Jede Zeichenkette, die ein Benutzer an dieser Stelle angibt, wird unter der `name`-Eigenschaft der Komponente gespeichert und ein vorhandener Wert der Eigenschaft wird an dieser Stelle angezeigt. Auf die gleiche Weise wird die Eigenschaft `category` der Komponente im Editor definiert. Der Konstanten `of category` folgt ein Property-Mapping auf die Eigenschaft `category`. Da die Eigenschaft `category` ein Enum ist, kann an dieser Stelle im Editor nur ein Wert des entsprechenden Enums ausgewählt beziehungsweise angezeigt werden.

Als nächstes sollen die Subkomponenten einer Komponente eingegeben werden können. Laut dem Komponenten-Konzept, das im letzten Abschnitt vorgestellt wurde, handelt es sich bei den Subkomponenten um Kind-Knoten des Konzeptes `CompositionReference`. Da die Rolle `composits` die Kardinalität `0..n` besitzt, handelt es sich hier wiederum um eine Liste von Editorelementen. Für die Darstellung der einzelnen Elemente wird die Editordefinition des entsprechenden Kindkonzeptes verwendet.

Auch bei den Rollen `facade` und `imports` handelt es sich um Kindkonzepte. Im Gegensatz zu den `composits` besitzt die Rolle `facade` die Kardinalität `0..1`. Aus diesem Grund wird für sie keine Liste benötigt. Die `facade` ist von den `imports` durch eine Leerzeile getrennt.

Zusätzlich zum Editor ist es in MPS möglich, für jedes Konzept einen *Inspector* zu definieren. Der Inspektor eines Konzeptes wird aktiv, wenn sich der Cursor innerhalb des Editors des Konzeptes befindet. Für die Komponente wird über den Inspektor die `singleton`-Eigenschaft gesetzt.

Die Editoren der anderen Konzepte sind analog definiert. Referenzen kommen in der Editor-

definition der Komponente nicht vor. Diese verhalten sich ähnlich zu den Kindkonzepten, mit der Ausnahme, dass sie an der entsprechenden Stelle im Editor nicht neu erzeugt werden. Für Referenzen muss aus diesem Grund die Repräsentation der Referenz im Editor definiert werden. In den meisten Fällen werden Referenzen durch den Wert ihrer `name`-Eigenschaft repräsentiert.

6.2.3. Constraints und Scoping

Dieser Abschnitt soll die Funktionsweise des Scopings für die Spracherweiterung erläutern. Über den Constraint-Aspekt in MPS lassen sich Restriktionen für Knoten definieren. Zum Einen kann für jedes Konzept geregelt werden, in welchem Kontext es als Kind beziehungsweise Elternknoten zulässig ist, zum anderen können für Referenzen Suchräume (engl. *search scopes*) definiert werden. Im Folgenden sollen die Funktionsweisen dieser beiden Constraint-Arten exemplarisch erläutert werden. Im Anschluss soll dann auf die Integration der Suchräume in die Basissprache eingegangen werden.

Constraints für Eltern- / Kindknoten

Da die neuen Elemente der Spracherweiterung ausschließlich in den Root-Konzepten `Component`, `QuasarInterface` und `QuasarClass` verfügbar sein sollen, findet man in vielen Konzepten das Constraint von Abbildung 6.6. Alle Root-Konzepte der Spracherweiterung besitzen eine Quasar-Kategorie und implementieren aus diesem Grund die Konzept-Schnittstelle `IHasQuasarCategory`. In dem Constraint wird vom Elternknoten ausgehend nach einem übergeordneten Knoten mit einer Quasar-Kategorie gesucht. Wird einer gefunden, so eignet sich das Konzept im angegebenen Kontext als Kindknoten.

```
can be child
  (operationContext, scope, parentNode, link, childConcept)->boolean {
    parentNode.ancestor<concept = IHasQuasarCategory>.isNotNull;
  }
```

Abbildung 6.6.: Constraint für die Eignung als Kindknoten

Constraints für Referenzen

Für die Definition von Suchräumen für Referenzen gibt es in MPS zwei Möglichkeiten:

1. Berechnung aller Knoten des Raumes und Rückgabe der Knoten als Sequenz
2. Implementierung der Schnittstelle `ISearchScope`

Innerhalb der Basissprache wird auf die zweite Möglichkeit zurückgegriffen. MPS definiert den `VisibleClassifiersScope`, welcher den `ReachableClassifiersScope` erweitert. Der `VisibleClassifiersScope` wird bei der Initialisierung mit einem Kontext-Knoten, einem Constraint und einem Scope konfiguriert. Der Kontext-Knoten ist der Knoten im Baum, der die neue Referenz beinhalten soll. Über das Constraint wird die Funktion des zu referenzierenden Knotens bestimmt (Klasse, Schnittstelle, Annotation, Classifier, usw.). Der Scope ist die Menge aller Knoten eines Modelles und unter Umständen importierter Modelle.

Der `VisibleClassifiersScope` implementiert zwei Methoden. Eine Methode, die alle sichtbaren Classifier für den aktuellen Kontext zurückgibt und eine Methode, mit welcher überprüft werden kann, ob sich ein bestimmter Classifier im Scope befindet.

Für die Spracherweiterung ist es notwendig, die Sichtbarkeit von Knoten gemäß der im Entwurf genannten Regeln weiter einzuschränken. Aus diesem Grund wird der `QuasarClassifierScope` definiert, der den `VisibleClassifierScope` erweitert. Listing 6.1 zeigt die Implementierung zur Berechnung der sichtbaren Classifier innerhalb des `QuasarClassifierScope`.

```
1  public List<Classifier> getClassifiers() {
2    List<Classifier> classifiers = super.getClassifiers();
3    List<Classifier> result = new ArrayList<Classifier>();
4    for (Classifier classifier : classifiers) {
5      boolean isVisible = QuasarScopeUtil.isVisible(BaseAdapter.fromAdapter(
           classifier), this.myQuasarContextNode, this.getConstraint());
6      boolean isInsideVisibleComponents = isInsideVisibleComponent(BaseAdapter.
           fromAdapter(classifier));
7      if (isVisible && isInsideVisibleComponents) {
8        result.add(classifier);
9      }
10   }
11   return result;
12 }
```

Listing 6.1: Berechnung der sichtbaren Classifier

In Zeile 2 werden zunächst alle sichtbaren Classifier des aktuellen Kontextes ermittelt, indem die Anfrage an die Oberklasse, den `VisibleClassifiersScope`, delegiert wird. Die Berechnung der Sichtbarkeit eines Classifiers nach Quasar unterteilt sich in zwei Teile. Das `QuasarScopeUtil` enthält die im fachlichen Entwurf entwickelten Regeln für die Sichtbarkeit von Referenzen im Bezug auf Kategorie, Kopplungsgrad und Schnittstellentyp. In Zeile 5 wird überprüft, ob ein Classifier im Hinblick auf Kopplungsgrad, Kategorie und

Schnittstellentyp im aktuellen Kontext valide ist. In Zeile 6 wird in einem zweiten Schritt überprüft, ob sich der Classifier innerhalb einer sichtbaren Komponente befindet. Treffen beide Bedingungen zu, ist die Referenz im aktuellen Kontext zulässig und wird zur Ergebnismenge hinzugefügt.

Integration in die Basissprache

In MPS gibt es drei Mechanismen, um die Menge an gültigen Referenzen für einen bestimmten Kontext zu bestimmen:

1. In Abschnitt Struktur der Sprache wurde erwähnt, dass die Unterklasse eines Konzeptes automatisch an allen Stellen substituiert wird, an denen es selbst zulässig ist oder Oberklassen des Konzeptes zulässig sind. Über den Constraint-Aspekt eines Konzeptes kann man für jedes Konzept einen Standard-Gültigkeitsbereich definieren, welcher verwendet wird, wenn kein expliziter Gültigkeitsbereich für eine Referenz angegeben wurde.
2. Für Referenzen eines Konzeptes lassen sich explizite Gültigkeitsbereiche definieren. Dies ist nur für solche Referenzen möglich, die entweder innerhalb des Konzeptes definiert wurden oder die Referenz einer Oberklasse einschränken.
3. Die dritte Möglichkeit besteht darin, die Liste der vorgeschlagenen Konzepte manuell über den Action Aspekt der Sprache zu verändern. Da es sich beim Editor von MPS um einen strukturierten Texteditor handelt, können nur solche Konzepte eingegeben werden, die auch vorgeschlagen werden.

Im Folgenden soll die Integration des `QuasarClassifierScope` exemplarisch für Referenzen auf Classifier erläutert werden.

Referenzen auf Classifier werden in der Basissprache von MPS über den `ClassifierType` erzeugt. Damit der `QuasarClassifierScope` integriert werden kann, wird die Unterklasse `QuasarClassifierType` definiert, welche die möglichen Referenzen auf Classifier einschränkt. Hierdurch besteht nun die Möglichkeit, einen neuen Suchraum für die Classifier-Referenz zu definieren (Abbildung 6.7).

```
link {quasarClassifier}
  referent set handler:<none>
  search scope:
    (model, scope, referenceNode, linkTarget, enclosingNode)->join(ISearchScope | sequence<node< >>) {
      new QuasarClassifierScope(referenceNode.isNull ? enclosingNode : referenceNode,
          IQuasarClassifiersSearchScope.CLASSIFFIER, scope);
    }
```

Abbildung 6.7.: Definition des Suchraumes für Classifier

Da der `QuasarClassiferType` eine Unterklasse von `ClassifierType` ist, wird er automatisch an jeder Stelle substituiert, an der auch der `ClassifierType` zulässig ist. Dies bedeutet, dass nun an vielen Stellen Referenzen doppelt vorkommen: Einmal die Referenz auf einen Classifier über den `ClassifierType` und die Referenz auf den gleichen Classifier über den `QuasarClassifierType`. Da der `ClassifierType` für die Spracherweiterung nicht weiter benötigt wird, wird eine Node-Substitution-Action über den Action-Aspek definiert, mit der alle Substitutionen für den `ClassifierType` in der Spracherweiterung (Abbildung 6.8) entfernt werden.

```
substituted node: Type  // remove ClassifierType from QuasarDSL
  condition :
    (operationContext, scope, model, parentNode, childConcept, currentTargetNode, wrapped, link)->boolean {
      node<IHasQuasarCategory> parent = parentNode.ancestor<concept = IHasQuasarCategory>;
      return parent.isNotNull;
    }

  actions :
    remove concept ClassifierType
```

Abbildung 6.8.: Entfernen der Substitutionen des ClassifierType Konzeptes

6.2.4. Das Typsystem

Im Typsystem von MPS werden Typprüfungen für die Knoten im abstrakten Syntaxbaum durchgeführt. Es gibt verschiedene Arten von Typsystemregeln. Für die Spracherweiterung werden zwei von ihnen verwendet: die `SubtypingRule` sowie die `NonTypesystemRule`. Diese beiden Regelarten sollen im Folgenden exemplarisch vorgestellt werden. In MPS gibt es die Möglichkeit, Verstöße gegen Typsystemregeln durch QuickFixes zu beheben. Für die unterschiedlichen Typsystemregeln sollen in diesem Abschnitt Beispiele gegeben werden.

Die SubtypingRule

Die `SubtypingRules` dienen MPS dazu zu ermitteln, ob ein Typ ein Subtyp eines anderen Typen ist. Eine `SubtypingRule` ist eine Funktion, welche die direkten Obertypen eines Typen zurückgibt. Anwendung innerhalb der Spracherweiterung findet diese Regelart für den `QuasarClassifierType`, der ein Subtyp des `ClassifierType`s ist. Abbildung 6.9 zeigt die `SubtypingRule` für den `QuasarClassifierType`.

```
subtyping rule subtypeing_QuasarClassifierType {
  weak = false
  applicable for concept = QuasarClassifierType as quasarClassifierType

  rule {
    node<ClassifierType> classifierType = new node<ClassifierType>();
    classifierType.classifier = quasarClassifierType.quasarClassifier;
    return classifierType;
  }
}
```

Abbildung 6.9.: SubtypingRule des QuasarClassifierTypes

Die NonTypesystemRule

Die `NonTypesystemRule` wird in der Spracherweiterung verwendet, um die im Sprachentwurf beschriebenen Regeln zu validieren, welche sich nicht durch Scoping realisieren lassen. Abbildung 6.10 zeigt ein Beispiel für eine `NonTypesystemRule`. Diese überprüft, dass die Fassade einer Komponente alle exportierten Schnittstellen implementiert. Da die Fassade die exportierten Schnittstellen auch indirekt implementieren darf, werden auch alle erweiterten Schnittstellen nach den exportierten Schnittstellen durchsucht.

```
non type system rule check_FacadeReference {
  applicable for concept = FacadeReference as facadeReference
  overrides false

  do {
    sequence<node<QuasarInterface>> toImplement = facadeReference.exported.reference;
    sequence<node<QuasarInterface>> implemented = new sequence<node<QuasarInterface>>(empty);

    sequence<node<QuasarInterface>> toExpand =
        facadeReference.reference.implementedInterface.classifier.
        select({~it => it as QuasarInterface; });

    while (toExpand.isNotEmpty) {
      sequence<node<QuasarInterface>> collected = new sequence<node<QuasarInterface>>(empty);
      foreach quasarInterface in toExpand {
        collected = collected.union(quasarInterface.extendedInterface.quasarClassifier.
            select({~it => it as QuasarInterface; }));
      }
      implemented = implemented.union(toExpand);
      toExpand = collected.except(implemented);
    }

    ensure implemented.containsAll(toImplement)   reportError
        String.format("Class %s must implement all exported interfaces", facadeReference.reference)
        -> facadeReference ;
  }
}
```

Abbildung 6.10.: Typprüfung der FacadeReference

Behebung von Fehlern durch QuickFixes

Abbildung 6.11 zeigt den QuckFix, der zur Fassade einer Komponente alle noch nicht implementierten Schnittstellen hinzufügt.

```
quick fix ImplementAllExportedInterfaces

arguments:
node<FacadeReference>  reference

fields:
<< ... >>

description(node)->string {
  String.format("Let class %s implement all exported interfaces", reference.reference.name);
}

execute(node)->void {
  sequence<node<QuasarInterface>> toImplement = reference.exported.reference;
   ...
  foreach quasarInterface in toImplement {
    node<QuasarClassifierType> type = new node<QuasarClassifierType>();
    type.quasarClassifier = quasarInterface;
    reference.reference.implementedInterface.add(type);
  }
}

set selection(node, editorContext, selectionBefore)->void {
  reference.reference;
}
```

Abbildung 6.11.: QuickFix für die FacadeReference

Implementiert die Fassade einer Komponente nicht alle exportierten Schnittstellen, so wird eine entsprechende Fehlermeldung angezeigt. Neben der Fehlermeldung erscheint eine rote Glühbirne, welche einen vorhandenen QuickFix anzeigt. Bei der Ausführung des QuickFixes werden zunächst alle noch nicht implementierten Schnittstellen ermittelt. Der Algorithmus zur Bestimmung der bereits implementierten Schnittstellen entspricht dem Algorithmus des letzten Abschnittes, weshalb er an dieser Stelle ausgelassen wird. Die noch nicht implementierten Schnittstellen werden am Ende zu den implementierten Schnittstellen der Klasse hinzugefügt.

6.2.5. Intentions

Intentions verhalten sich ähnlich, wie der im letzten Abschnitt vorgestellte QuickFix, mit dem Unterschied, dass eine Intention nicht nur im Fehlerfall ausgeführt werden kann. In der Basissprache werden Intentions beispielsweise zur automatischen Generierung von *Gettern* und *Settern* für eine Klasse verwendet. Der Zugriff auf eine Intention erfolgt entweder über die gelbe Glühbirne am linken Rand oder durch die Tastenkombination ALT + Enter.

In der Spracherweiterung gibt es zwei Intentions. Die eine dient zum Ändern des Schnittstellentyps, die andere zur Generierung von Accessoren für importierte Schnittstellen.

Eine parametrisierte Intention besteht aus vier Teilen:

1. Eine Abfrage, die eine Liste von Parametern zurückliefert, auf welche die Intention angewendet werden kann.

2. Eine Funktion zur Generierung des Beschreibungstextes, die für jeden Parameter aufgerufen wird.
3. Eine Funktion, welche überprüft, ob die Intention im aktuellen Kontext aktiviert werden kann.
4. Eine Methode, die den Code enthält, der bei Aktivierung der Intention ausgeführt werden soll.

6.2.6. Integration in die IDE

Der Plugin-Aspekt einer Sprache bietet verschiedene Möglichkeiten zur Integration einer Spracherweiterung in MPS. Die Spracherweiterung definiert zwei Integrationen in die Entwicklungsumgebung. Ein Einstellungsdialog zur Konfiguration der Modelle, deren Typen innerhalb der Spracherweiterung als 0-Typen behandelt werden sollen und eine Konfiguration zur Ausführung von Komponenten direkt in MPS. Im Folgenden soll auf diese beiden Integrationen näher eingegangen werden.

0-Typen Konfiguration

Innerhalb der Spracherweiterung besteht die Möglichkeit Java-Klassen und -Schnittstellen zu verwenden. Da diese keine Kategorie besitzen, werden alle herkömmlichen Klassen und Schnittstellen standardmäßig wie Klassen und Schnittstellen der Kategorie T behandelt. Das bedeutet, dass sie nicht direkt in Komponenten der Kategorie A verwendet werden können, sondern vorher über Adapter entkoppelt werden müssen.

Für den Fall, dass Klassen und Schnittstellen direkt verwendet werden sollen, ohne sie zuvor über einen Adapter entkoppeln zu müssen, besteht die Möglichkeit einzelne Modelle explizit mit der Kategorie 0 zu konfigurieren. Dies ist über die 0-Typen Konfiguration möglich.

Es ist empfehlenswert, zumindest die Modelle `java.lang` und `java.util` als Modelle der Kategorie 0 zu konfigurieren, damit beispielsweise die Klasse `Object`, oder die Schnittstelle `List` in allen Komponenten zugänglich ist.

Die 0-Typen Konfiguration ist über die allgemeinen MPS-Einstellungen (Abbildung 6.12) erreichbar.

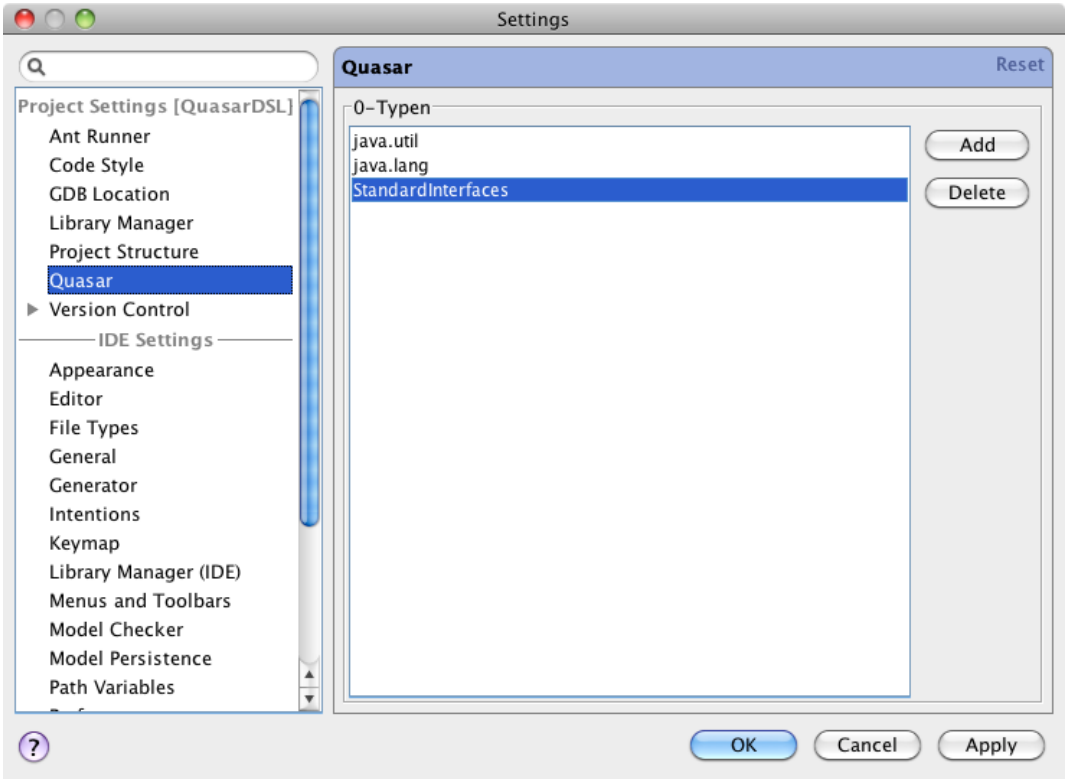

Abbildung 6.12.: 0-Typen Konfiguration

Ausführungskonfiguration für Komponenten

Zur leichteren Bedienbarkeit soll es möglich sein, Komponenten direkt in MPS auszuführen. Um eine Komponente ausführbar zu machen, muss diese die Standardschnittstelle `RootComponent` implementieren. Diese Schnittstelle definiert die Methode `run`, welche bei der Ausführung einer Komponente aufgerufen wird.

Abbildung 6.13 zeigt die Ausführungskonfiguration einer Komponente. Unter dem Punkt `Main node` kann eine Komponente ausgewählt werden, welche die Schnittstelle `RootComponent` exportiert.

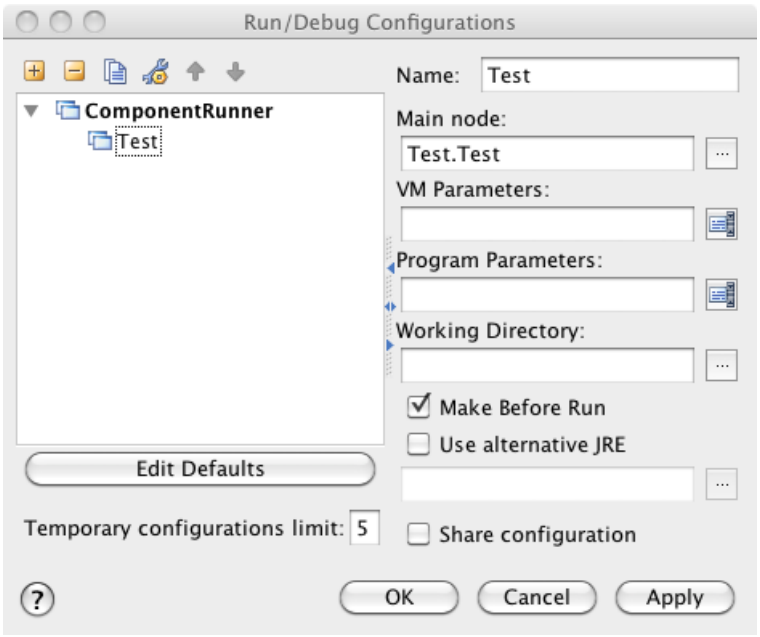

Abbildung 6.13.: Ausführungskonfiguration einer Komponente

6.2.7. Der Generator

In MPS gibt es zwei Arten von Generatoren. Generatoren zur Generierung von Textdateien auf Basis des semantischen Modells und Generatoren für Modelltransformationen. Unter einer Modelltransformation versteht man die Überführung des semantischen Modells in ein anderes semantisches Modell des selben oder eines anderen Metamodells.

Die Basissprache von MPS besitzt einen Textgenerator, der auf Basis des semantischen Modells, Java-Dateien erzeugt. Aus diesem Grund wird für die Spracherweiterung kein Textgenerator benötigt. Für sie ist nur ein Generator erforderlich, welcher eine Modelltransformation durchführt und alle Sprachkonstrukte auf Elemente der Basissprache abbildet. Des Weiteren wird ein Mechanismus benötigt, um die Abhängigkeiten für das über die Spracherweiterung entworfene System zu setzen. Im Folgenden soll der Mechanismus zum Setzen der Abhängigkeiten sowie die Modelltransformation für die Sprachkonstrukte der Spracherweiterung genauer erläutert werden.

Mechanismus zum Setzen der Abhängigkeiten

Das Setzen der Abhängigkeiten soll für die Spracherweiterung mit Google Guice umgesetzt werden. Bei Google Guice handelt es sich um ein leichtgewichtiges Dependency Injection Framework der Firma Google. In Google Guice hat man die Möglichkeit die Konfiguration der Abhängigkeiten aufzuteilen. Dies ist notwendig, da der Generator von MPS sequentiell arbeitet. Das bedeutet, dass jedes Modell der Reihe nach transformiert wird. Dabei ist es nicht möglich Konfigurationsdateien modellübergreifend zu erzeugen. Auch modellübergreifende Referenzen sind nur für die Basissprache möglich. Die sequentielle Transformation der Modelle begründen die Programmierer von MPS damit, dass es zu viel Speicher brauche, wenn alle Modelle parallel im Speicher gehalten werden müssen. Auch könne dann nicht mehr jedes Modell separat für sich generiert werden (vgl. MPS Forum, 2011).

Google Guice verwendet zur Konfiguration der Abhängigkeiten `Module`s. Für jede Komponente des semantischen Modells wird ein `Module` erzeugt, das alle Abhängigkeiten der Komponente konfiguriert.

Eine konfigurierte Abhängigkeit ist in Guice standardmäßig global gültig. Dadurch entsteht folgendes Problem: Angenommen es existiert eine Komponente `A`, welche die Schnittstelle `S` importiert. Komponente `A` wird für diese Schnittstelle mit der implementierenden Komponente `I1` konfiguriert. Auch Komponente `B` importiert Schnittstelle `S`, möchte allerdings die Implementierung `I2` verwenden. Das globale Setzen von Abhängigkeiten funktioniert in diesem Falle nicht. Aus diesem Grund werden alle injizierten Abhängigkeiten innerhalb einer Komponente mit einer Annotation versehen, welche die Komponentenzugehörigkeit angibt. Auf diese Weise lassen sich komponentenweite Gültigkeitsbereiche für Abhängigkeiten konfigurieren.

Google Guice erzeugt in seiner Standardkonfiguration jedes Mal eine neue Instanz, wenn es eine Abhängigkeit setzt. Verwendet man zustandslose Komponenten, ist diese Einstellung gut geeignet. In einigen Situationen kann jedoch die Verwendung von zustandsbehafteten Komponenten notwendig werden. Für diesen Fall besitzt die Komponente die Eigenschaft `singleton`. Ist für eine Komponente die Eigenschaft `singleton` gesetzt, wird bei der

Auflösung der Abhängigkeiten nur eine globale Instanz erzeugt und gesetzt. Aus diesem Grund sollte darauf geachtet werden, die Fassade einer Singleton-Komponente threadsicher zu programmieren.

Die Konfigurationen für importierte Komponenten werden bei Bedarf geladen. Da keine Referenzen über Modellgrenzen hinweg möglich sind, wird die Abhängigkeit zur Laufzeit über *Reflection* aufgelöst. Der Mechanismus zum Nachladen der Abhängigkeiten von anderen Komponenten besitzt eine Zyklenerkennung, um Endlosschleifen im Falle zirkulärer Abhängigkeiten zu verhindern.

Transformation der Komponente

Für jede Komponente des semantischen Modells werden verschiedene Java-Klassen und -Schnittstellen erzeugt, deren Funktionen im Folgenden näher beschrieben werden sollen:

- Eine Komponentenschnittstelle, die Getter für alle exportierten und Setter für alle importierten Schnittstellen enthält. Hierüber lassen sich die Abhängigkeiten zwischen den Komponenten leicht außerhalb von MPS verändern. Jeder Setter gibt als Ergebnis der Rekonfiguration eine neue Instanz der Komponentenschnittstelle mit der neuen Abhängigkeit zurück. Auf diese Weise lassen sich auch mehrere Setter-Aufrufe in einer Anweisung kombinieren.
- Einen Configurator, der die Fassade der Komponente erweitert und die Komponentenschnittstelle implementiert. Dieser delegiert die Getter- und Setter- Aufrufe an die Factory der Komponente.
- Die Facory kapselt die Abhängigkeit zu Google Guice. Sie konfiguriert Guice mit der im letzten Abschnitt erwähnten `DependencyConfiguration`.
- Die DependencyConfiguration enthält, wie im letzten Abschnitt erwähnt, die Konfiguration der Abhängigkeiten für die Komponente.

Transformation des Accessors

Für jeden Accessor wird ein Feld generiert, welches mit der importierten Schnittstelle konfiguriert wird. Das Setzen der Abhängigkeit erfolgt über Setter-Injection, weshalb für jeden Accessor ein Setter generiert wird.

Hierbei entsteht folgendes Problem: Es ist nicht möglich auf eine importierte Schnittstelle innerhalb des Konstruktors einer Klasse zuzugreifen, da die Abhängigkeiten erst nach erfolgreicher Konstruktion einer Objektinstanz gesetzt werden. Initialisierender Code für importierte Abhängigkeiten kann daher nicht im Konstruktor stehen. Für diesen Fall kann dem

Accessor ein Statement übergeben werden, das nach Setzen einer Abhängigkeit aufgerufen wird. Das Setzen dieses Statements ist über den Inspektor des Accessors möglich. Wird mehr als ein Statement zur Initialisierung benötigt, so kann an dieser Stelle auch ein Block von Statements übergeben werden.

Transformation der Quasar- Schnittstelle und Klasse

Die Quasar- Schnittstelle und Klasse können durch Entfernen ihrer zusätzlichen Informationen zu den entsprechenden Elementen der Basissprache transformiert werden, da die zusätzlichen Informationen nur für die Modellierung innerhalb von MPS verwendet werden und keinen Einfluss auf das Laufzeitverhalten haben.

7. Realisierung des Beispielszenarios

In diesem Kapitel soll auf die Realisierung des Beispielszenarios mit Hilfe der domänenspezifischen Spracherweiterung eingegangen werden. Zunächst sollen allgemeine Hinweise zur Inbetriebnahme der Spracherweiterung mit MPS gegeben werden. Im Anschluss werden dann ausgewählte Stellen der Realisierung des Beispielszenarios vorgestellt.

7.1. Hinweise zur Inbetriebnahme

Bei der Inbetriebnahme der Spracherweiterung mit MPS sollte auf folgende Punkte geachtet werden:

Generierung der Sprache: MPS selber ist per Bootstrapping entwickelt worden. Das bedeutet, dass der Großteil von MPS in MPS entwickelt worden ist. Die Erstellung einer Sprache mit MPS erfolgt daher auch über eine Spracherweiterung, welche vor der Benutzung generiert werden muss. Die einzelnen Sprachaspekte entsprechen dabei Modellen. Da MPS über keinen Mechanismus verfügt, welcher die beste Reihenfolge für die Generierung der Modelle festlegt, ist es erforderlich die Spracherweiterung mehrmals zu generieren, damit alle Abhängigkeiten aufgelöst werden können. Dies ist insbesondere dann notwendig, wenn die generierten Dateien durch ein Clean auf dem Projekt gelöscht wurden, oder auf Grund des initialen Checkouts aus dem Versionskontrollsystem noch nicht vorhanden sind. In diesem Fall hat es sich als hilfreich erwiesen alle Aspekte der Sprache separat zu generieren.

Konfiguration von 0-Typen: Das Beispielszenario verwendet die Bibliotheken `java.lang`, `java.util` sowie das Modell `StandardInterfaces` als 0-Typen. Diese müssen in den Einstellungen, wie im Abschnitt 0-Typen Konfiguration beschrieben, konfiguriert werden. Für Quasar-Komponenten ist die Konfiguration als 0-Typen nicht notwendig. Im Modell `StandardInterfaces` wird jedoch ein Enum definiert, welches nicht Bestandteil der Spracherweiterung ist. Aus diesem Grund muss auch das Modell `StandardInterfaces` in der 0-Typen Konfiguration hinzugefügt werden.

7.2. Definition der Komponenten

Die Definition der Komponenten kann direkt aus dem Komponentenmodell (Abbildung 4.2) übertragen werden. Der einzige Unterschied der Definition zum Komponentenmodell besteht darin, dass auch der Logger, welcher innerhalb der Komponente Verwendung findet, über einen Adapter entkoppelt wird. Listing 7.1 zeigt die Komponentendefinition der NewsSystem-Komponente.

```
1  component NewsSystem of category A composes News, Notification, NewsAdapter {
2    facade NewsSystemFacade {
3      export INewsUser
4      export INewsAdmin
5    }
6
7    import INewsMgnt [ NewsAdapter ]
8    import IUserMgnt [ ExternalMockAdapters ]
9    import IPersistence [ ExternalMockAdapters ]
10   import ILogger [ LoggerAdapter ]
11   import IMail [ ExternalMockAdapters ]
12 }
```

Listing 7.1: Definition der NewsSystem-Komponente

Die Komponente `ExternalMockAdapters` simuliert die Persistenz über eine Map, in welcher die Nachrichten gespeichert werden. Da alle Komponenten auf der gleichen Datenbasis arbeiten sollen, wird die Komponente `ExternalMockAdapters` als Singleton-Komponente definiert. Die Logger-Komponente ist ein Multiton und wird in dem Initialisierungsblock jedes Accessors mit der zu protokollierenden Klasse konfiguriert.

In der News-Komponente wird dann die Abhängigkeitskonfiguration von der Elternkomponente für die Persistenz und den Logger übernommen (siehe Listing 7.2).

```
1  component News of category A composes Statistic {
2    facade NewsFacade {
3      export INews
4    }
5
6    import IStatistic [ Statistic ]
7    import IPersistence [ <<from parent>> ]
8    import ILogger [ <<from parent>> ]
9    import INewMessage [ NewsAdapter ]
10 }
```

Listing 7.2: Definition der News-Komponente

Die Definition der übrigen Komponenten geschieht analog, weshalb auf sie nicht weiter eingegangen werden soll.

7.3. Definition des News-Adapters

Der News-Adapter entkoppelt die Komponenten NewsSystem und News, indem er die Schnittstelle `INews` der News-Komponente importiert und diese auf die Schnittstelle `INewsMgnt` der NewsKomponente abbildet. Listing 7.3 zeigt die Methode `getLatestMessages` des NewsAdapters.

```
1  public List<IMessageType> getLatestMessages(int number) {
2   logger.info(String.format("getLatestMessages: %d", number));
3   List<IMessageType> toReturn = new ArrayList<IMessageType>();
4
5   for (final IMessage message : news.getLatestMessages(number)) {
6     IMessageType messageToReturn = new IMessageType() {
7       public string getCaption() {
8         message.getCaption();
9       }
10
11      public string getContent() {
12        message.getContent();
13      }
14
15      public Date getCreationDate() {
16        message.getCreationDate();
17      }
18
19      public string getAuthor() {
20        message.getAuthor();
21      }
22
23      public int getId() {
24        message.getId();
25      }
26    };
27
28    toReturn.add(messageToReturn);
29  }
30
31  return toReturn;
32 }
```

Listing 7.3: Die Methode `getLatestsMessages` des NewsAdapters

Die Schnittstelle `INews` gibt die Entitätsschnittstelle `IMessage` der News-Komponente zurück (Zeile 5). Diese muss auf den Datentyp `IMessageType` der NewsSystem-Komponente adaptiert werden. Dies realisiert der NewsAdapter, indem er für jede zu adaptierende Nachricht eine anonyme Klasse der Schnittstelle `IMessageType` erzeugt und die Methodenaufrufe an die Instanz der Schnittstelle `IMessage` delegiert.

7.4. Ausführung des Beispielszenarios

Zum Testen der Realisierung des Beispielszenarios wird die Komponente Test definiert, welche die Schnittstellen `INewsUser` und `INewsAdmin` der NewsSystem-Komponente importiert. Damit die Komponente ausgeführt werden kann, exportiert sie, wie in Abschnitt Ausführungskonfiguration für Komponenten beschrieben, die Schnittstelle `RootComponent`.

Innerhalb der `run`-Methode wird zunächst über die Administrationsschnittstelle eine neue Nachricht eingestellt. Im Anschluss wird die Nachricht über die Benutzer-Schnittstelle abgefragt. Außerdem wird die Gesamtanzahl von Nachrichten im System abgefragt, die eins sein muss. Im Anschluss wird die Nachricht über die Administrationsschnittstelle gelöscht. Die Anzahl der Nachrichten muss danach null sein.

Der LoggerAdapter verwendet die Bibliothek log4j mit dem Appender LogFactor5. Hierbei handelt es sich um eine Swing-Oberfläche zur Visualisierung der Logausgaben (siehe Abbildung 7.1). Auf der linken Seite der Swing-Oberfläche werden unter *Categories* die Komponenten des Beispielsystems hierarchisch angezeigt. Über diese Anzeige ist es möglich, die Logausgaben für ausgewählte Komponenten zu filtern.

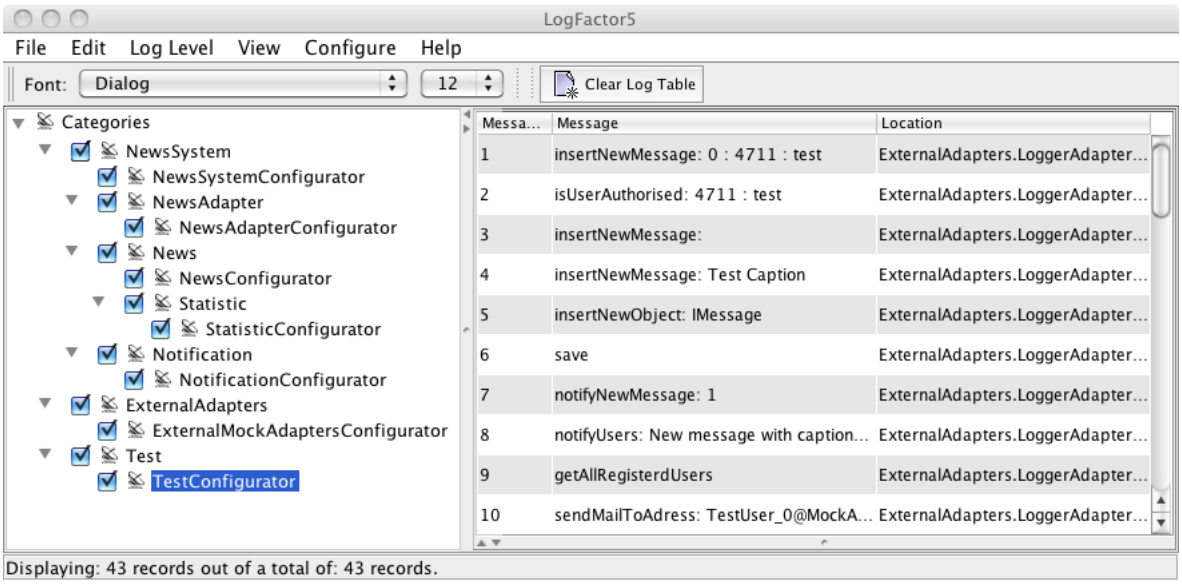

Abbildung 7.1.: Logging mit LogFactor5

7.5. Arbeiten mit dem generierten Code

Im Abschnitt Transformation der Komponente wurde beschrieben, welche Klassen und Schnittstellen aus einer Komponente generiert werden. Der generierte Code kann in eine beliebige Entwicklungsumgebung zur weiteren Bearbeitung importiert werden.

Angenommen, dass als Logging-Framework nicht *Log4J*, sondern ein anderes Framework verwendet werden soll. In diesem Fall könnte ohne Weiteres eine neue Implementierung der Schnittstelle `ILogger` erstellt und diese für das NewsSystem als Abhängigkeit gesetzt werden (siehe Listing 7.4).

```
1 NewsSystemFactory.getInstanceForInterface(INewsSystemComponent.class).
      setILogger(newLogger);
```

Listing 7.4: Ändern des Loggers für die NewsSystem-Komponente

Auch ein Austausch von mehreren Abhängigkeiten ist möglich. Listing 7.5 zeigt eine komplette Rekonfiguration der Test-Komponente.

```
1 RootComponent rootComponent = TestFactory.getInstanceForInterface(
      ITestComponent.class)
2   .setILogger(newLogger)
3   .setINewsAdmin(newNewsSystemComponent.getINewsAdmin())
4   .setINewsUser(newNewsSystemComponent.getINewsUser())
5   .getRootComponent();
6 rootComponent.run();
```

Listing 7.5: Rekonfiguration der Test-Komponente

8. Schluss

In diesem Kapittel sollen die Ergebnisse der Arbeit bewertet sowie die nächsten Schritte für eine Weiterentwicklung der Sprache aufgezählt werden.

8.1. Bewertung der Ergebnisse

Im Rahmen dieser Arbeit wurde eine domänenspezifische Spracherweiterung zur komponentenordientierten Entwicklung vorgestellt. Es wurde gezeigt, dass sich Mechanismen wie Scoping und Typprüfung zur Validierung von Abhängigkeiten zwischen Komponenten eignen und den Entwickler bei der Realisierung einer sauberen Komponentenarchitektur unterstützen können. Auf diese Weise wird die Austauschbarkeit einzelner Komponenten eines Softwaresystems stark begünstigt. Die vorgestellte Sprache zur direkten komponentenorientierten Programmierung steht noch am Anfang ihrer Entwicklung und bedarf einiger weiterer Untersuchungen, bevor sie sich für einen produktiven Einsatz eignet.

8.2. Ausblick

Bei der in dieser Arbeit entwickelten domänenspezifischen Spracherweiterung handelt es sich lediglich um einen Prototypen einer Sprache zur direkten komponentenorientierten Modellierung. Folgende Aspekte bedürfen vor dem produktiven Einsatz der vorgestellten Spracherweiterung einer genaueren Untersuchung und gegebenenfalls. Weiterentwicklung.

- Trennung von Komponentendefinition und Konfiguration der Abhängigkeiten: In der vorgestellten Spracherweiterung wurde der Einfachheit halber auf eine Trennung der Komponentendefinition und der Konfiguration der Komponenten mit den benötigten Abhängigkeiten verzichtet. Dieser Aspekt könnte besonders im Hinblick auf eine Versionierung von Schnittstellen und Komponenten notwendig werden.
- Versionierung von Komponenten und Schnittstellen: Das Thema der Versionierung von Schnittstellen und Komponenten wurde im Rahmen dieser Arbeit nicht behandelt. In einem großen Informationssystem, welches über einen langen Zeitraum in Betrieb

ist, sind Änderungen an Schnittstellen und damit den importierenden und exportierenden Komponenten unabdingbar.

- Dynamische Rekonfiguration von Komponenten: Innerhalb der Spracherweiterung besteht bisher keine Möglichkeit Komponenten im Fehlerfall mit neuen Abhängigkeiten zu Rekonfigurieren.
- Verbesserung der Dependency Injection: Im Zentrum dieser Arbeit steht die Entwicklung einer domänenspezifischen Sprache. Die Generierung von Code und das Setzen von Abhängigkeiten wurden nur zu Demonstrationszwecken in die Sprache integriert. Besonders die Rekonfiguration der Abhängigkeiten von Komponenten bedarf einer genaueren Untersuchung und eignet sich als Thema weiterer Arbeiten.
- Einsatz der Spracherweiterung mit Anwendungsservern: Professionelle Softwaresysteme werden heut zu Tage in Anwendungsservern betrieben. Es sollte eine Untersuchung durchgeführt werden, in wieweit sich die Spracherweiterung im Zusammenspiel mit Anwendungsservern betreiben lässt.
- Einführung weiterer Sprachelemente: Diese Arbeit konzentriert sich auf die Arbeit mit Komponenten in der Außensicht. Quasar definiert für die Innensicht von Komponenten Elemente wie Entitätsverwalter und Anwendungsfallklassen, die als zusätzliche Sprachelemente in die Sprache aufgenommen werden könnten.
- Feinere Unterteilung von Komponenten-Kategorien: Die vorgestellte Sprache unterscheidet zwischen den Komponenten-Kategorien A, T, 0 und R. Besonders für Komponenten der Kategorie A bietet es sich an, diese wiederum in Unterkategorien zu unterteilen, um so die Abhängigkeiten zwischen Komponenten unterschiedlicher Schichten der Architektur kontrollierbar zu machen.
- Usability-Untersuchung: Im Rahmen dieser Arbeit wurde keine Untersuchung über die Usability der domänenspezifischen Spracherweiterung durchgeführt. Auch auf diesem Gebiet bieten sich weitere Untersuchungen an.

Literaturverzeichnis

[Bass u. a. 1998] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software architecture in practice*. Reading, Mass. : Addison-Wesley, 1998 (SEI series in software engineering). – URL http://www.gbv.de/dms/bowker/toc/9780201199307.pdf. – ISBN 0201199300

[Cao u. a. 2009] CAO, Lan ; RAMESH, Balasubramaniam ; ROSSI, Matti: Are Domain-Specific Models Easier to Maintain Than UML Models? In: *IEEE Software* 26 (2009), July-Aug., Nr. 4, S. 19–21. – URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5076454. – [Online; Abruf: 2010-06-08]

[Dijkstra 1982] DIJKSTRA, Edsger W.: *Selected writings on computing: a personal perspective*. New York : Springer, 1982 (Texts and monographs in computer science). – ISBN 0387906525

[Fowler und Parsons 2011] FOWLER, Martin ; PARSONS, Rebecca: *Domain-specific languages*. Upper Saddle River, NJ : Addison-Wesley, 2011 (The Addison-Wesley signature series). – URL http://www.gbv.de/dms/tib-ub-hannover/63107046x.pdf. – ISBN 0321712943 (hardcover)

[Gauthier und Ponto 1970] GAUTHIER, Richard L. ; PONTO, Stephen D.: *Designing systems programs*. Englewood Cliffs : Prentice-Hall, 1970 (Prentice-Hall series in automatic computation). – ISBN 0132019620

[Larman 2005] LARMAN, Craig: *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. 3. ed. Upper Saddle River, NJ : Prentice Hall, 2005. – URL http://www.gbv.de/dms/bowker/toc/9780131489066.pdf. – ISBN 0131489062 (Prentice Hall : Pp.)

[Meyer 1986] MEYER, Bertrand: Design by Contract / Interactive Software Engineering Inc. 1986. – Forschungsbericht

[MPS Forum 2011] GRYAZNOV, Evgeny: *Dependencies between models within same Solution*. Website. Mai 2011. – URL http://forum.jetbrains.com/message/Meta-Programming-System-552-3. – [Online; Abruf: 2011-06-15]

[MPS User's Guide 2011] JETBRAINS (Hrsg.): *MPS User's Guide*. Website. 2011. – URL http://confluence.jetbrains.net/display/MPSD1/MPS+User%27s+Guide. – [Online; Abruf: 2011-04-10]

[Reenskaug 1979] REENSKAUG, Trygve: *MVC*. Website. Mai 1979. – URL http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. – [Online; Abruf: 2011-03-27]

[Siedersleben 2004] SIEDERSLEBEN, Johannes: *Moderne Software-Architektur: umsichtig planen, robust bauen mit Quasar*. 1. Aufl. Heidelberg : dpunkt-Verl., 2004. – ISBN 3-89864-292-5

[Xtext User Guide 2011] THEECLIPSEFOUNDATION (Hrsg.): *Xtext Dokumentation Version 1.0.1*. Website. 2011. – URL http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.html. – [Online; Abruf: 2011-04-10]

A. Grammatik

A.1. Komponente

```
componentDeclaration
    : 'component' IDENTIFIER
      'of category' ('A' | 'T' | '0' | 'R')
      ('composes' IDENTIFIER
      (',' IDENTIFIER
      )*
      )?
      componentBody
    ;

    componentBody
    : '{'
          (facadeDeclaration
          )?
          (typeImportDeclaration
          )*
      '}'
    ;

facadeDeclaration
    : 'facade' IDENTIFIER '{'
      ('export' IDENTIFIER
      )+
      '}'
    ;

typeImportDeclaration
    : 'import' IDENTIFIER
      ('[' IDENTIFIER ']'
      )?
    ;
```

A.2. Schnittstelle

```
normalInterfaceDeclaration
    :   'component' IDENTIFIER
      ('offer' | 'require' | 'standard'
      )?
      modifiers 'interface' IDENTIFIER
        (typeParameters
        )?
        'with coupling' INTLITERAL
        ('extends' typeList
        )?
        interfaceBody
    ;
```

A.3. Klasse

```
normalClassDeclaration
    :   'component' IDENTIFIER
      modifiers  'class' IDENTIFIER
        (typeParameters
        )?
        ('extends' type
        )?
        ('implements' typeList
        )?
        classBody
    ;

    accessorDeclaration
    :   'accessor'
      IDENTIFIER
      IDENTIFIER
        ';'
    ;
```

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 8. August 2011	
Ort, Datum	Unterschrift