Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Bachelorarbeit

Über die Automatisierung der Entwicklung von Software Generatoren

vorgelegt an der Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt in der Fakultät Informatik und Wirtschaftsinformatik zum Abschluss eines Studiums im Studiengang Informatik

René Ziegler

Eingereicht am: 19. März 2018

Erstprüfer: Prof. Dr. Peter Braun
Zweitprüfer: M.Sc. Tobias Fertig

Zusammenfassung

Die Automatisierung hat in vielen Bereichen, wie zum Beispiel der Automobilindustrie, zur Erhöhung der Qualität und der Wirtschaftlichkeit der Produktion geführt. Der Ansatz der modellgetriebenen Softwareentwicklung könnte ebenfalls diese Vorteile bringen. Eine große Hürde für dessen Erfolg ist die Komplexität der Entwicklung von Codegeneratoren. Im Rahmen dieser Arbeit wurde, basierend auf einer eingehenden Analyse, ein Konzept entwickelt, welches diese Komplexität durch automatisierte Erzeugung einzelner Bestandteile eines Codegenerators reduziert. Ausgehend von diesem Konzept wurde der Metagenerator Spectrum als Proof-of-Concept umgesetzt. Es wurde gezeigt, dass es möglich ist, aus einer Referenzimplementation eine domänenspezifische Sprache zur Beschreibung von definierten Variabilitäten zu erzeugen und somit den Aufwand der Entwicklung eines Codegenerators zu reduzieren.

Abstract

In many areas, such as the automotive industry, automation has led to an increase in quality and cost effectiveness in production. The model-driven software development approach could bring these benefits as well. One major obstacle to its success is the complexity of developing code generators. In the context of this thesis a concept, based on an in-depth analysis, was developed that reduces this complexity by partial automated generation of a code generator. Based on this concept, the metagenerator Spectrum was implemented as a proof of concept. It has been shown that it is possible to generate a domain-specific language from a reference implementation to describe defined variabilities, thus reducing the effort of developing a code generator.

Danksagung

An dieser Stelle möchte ich einige Worte des Dankes anbringen.

Großer Dank geht an Professor Dr. Braun, welcher mir die Chance gab, eine Arbeit zu einem so fesselnden Thema verfassen zu können und mir im Rahmen der wertvollen Betreuung dieser Thesis vieles über das wissenschaftliche Arbeiten beibrachte.

Zudem möchte ich mich bei Tobias Fertig bedanken. Er hatte immer ein offenes Ohr für mich und wies mir an einigen Stellen, an denen ich nicht sicher war, in welche Richtung ich gehen sollte, durch Diskussionen den Weg.

Ein weiteres Dankeschön möchte ich hier an Vitaly Schreibmann richten. Er half mir sehr dabei, zu verstehen was ich eigentlich tat. Es war extrem hilfreich ihm mein Konzept zu erklären, verständiges Feedback zu bekommen und dadurch tieferes Verständnis aufzubauen.

Ich möchte ganz besonders meiner Lebensgefährtin für die unendliche Geduld danken, die Sie mir entgegenbrachte, als ich nichts anderes als diese Arbeit im Kopf hatte. Vor allem, dass Sie trotzdem noch die Muse dazu hatte, Korrektur zu lesen.

Inhaltsverzeichnis

1 Einführung

Als Henry Ford 1913 die Produktion des Modell T, umgangssprachlich auch Tin Lizzie genannt, auf Fließbandfertigung umstellte, revolutionierte er die Automobilindustrie. Ford war nicht der erste, der diese Form der Automatisierung verwendete. Bereits 1830 kam in den Schlachthöfen von Chicago eine Maschine zum Einsatz, die an Fleischerhaken aufgehängte Tierkörper durch die Schlachterei transportierte. Bei der Produktion des Oldsmobile Curved Dash lies Ranson Eli Olds 1910 erstmals die verschiedenen Arbeitsschritte an unterschiedlichen Arbeitsstationen durchführen. Fords Revolution war die Kombination beider Ideen. Er entwickelte eine Produktionsstraße, auf welcher die Karossen auf einem Fließband von Arbeitsstation zu Arbeitsstation befördert wurden. An jeder Haltestelle wurden nur wenige Handgriffe von spezialisierten Arbeitern durchgeführt [24].

Fords Vision war es, ein Auto herzustellen, welches sich Menschen aller Gesellschaftsschichten leisten konnten. Durch die Reduktion der Produktionszeit der Tin Lizzie von 12,5 Stunden auf etwa 6 Stunden konnte Ford den Preis senken. Kostete ein Auto des Model T vor der Einführung der Produktionsstraße 825$, erreichte der Preis in den Jahren danach einen Tiefststand von 259$ [23]. Setzt man diesen Preis in ein Verhältnis mit dem durchschnittlichen Einkommen in den USA, das 1910 bei jährlich 438$ lag, kann man sagen, dass Fords Traum durch die eingesetzten Techniken Realität wurde [20].

Im Zuge der weiteren Entwicklung der Robotik wurden immer mehr Aufgaben, die bisher von Menschen am Fließband durchgeführt wurden, von Automaten übernommen. In der Automobil-Industrie war General Motors der erste Hersteller, bei welchem die Produktionsstraßen im Jahr 1961 mit 66 Robotern des Typs Unimation ausgestattet wurden. Bis zur Erfindung des integrierten Schaltkreises in den 1970ern waren die Roboter jedoch ineffizient. Der Markt für industrielle Roboter explodierte in den Folgejahren. Im Jahr 1984 waren weltweit ungefähr 100.000 Roboter im Einsatz [6, 29].

Die industrielle Revolution prägte die Autoindustrie: von der Erfindung auswechselbarer Teile 1910 bei Ransom Olds, über die Weiterentwicklung des Konzepts unter der Verwendung von Fließbändern bei Ford im Jahr 1913, bis hin zur abschließenden Automatisierung mit Industriellen Robotern in den frühen 1980ern [6].

1.1 Motivation

> „If you can compose components manually, you can also automate this process.“

Das hervorgehobene Zitat nennen Czarnecki und Eisenecker die Automation Assumption. Diese allgemein gehaltene Aussage lässt die Parallelen, die die beiden Autoren zwischen der Automatisierung der Automobilindustrie und der automatischen Code Generierung sehen, erkennen. Dafür müssten die einzelnen Komponenten einer Softwarefamilie derart gestaltet werden, dass diese austauschbar in eine gemeinsame Struktur integriert werden können. Des weiteren müsste klar definiert sein, welche Teile eines Programms konfigurierbar sind und welche der einzelnen Komponenten in welcher Konfiguration benötigt werden. Setzt man dieses definierte Wissen in Programmcode um, könnte ein solches Programm eine Software in einer entsprechenden Konfiguration generieren [6].

Konkret bedeutet dies, dass entweder eine vorhandene Implementation in Komponenten zerlegt oder eine für die Zwecke der Codegenerierung vorgesehene Referenzimplementation geschrieben werden muss. Codeabschnitte, die in Ihrer Struktur gleich sind, sich jedoch inhaltlich unterscheiden, müssen formal beschrieben werden [27]. Ein solches abstraktes Modell wird dann mit Daten befüllt. Schlussendlich wird ein Generator implementiert, der den Quellcode für unterschiedliche Ausprägungen eines Programms einer Software-Familie, auf Basis des konkreten Modells, generieren kann [7].

Sowohl bei der Umsetzung von einzigartigen Anwendungen, als auch bei der Verwirklichung von Software mit mehreren Varianten, kann die Verwendung von bereits verfügbaren oder die Entwicklung eigener Codegeneratoren vorteilhaft sein. Die Entwicklungsgeschwindigkeit könnte erhöht, die Softwarequalität gesteigert und Komplexität durch Abstraktion reduziert werden [27]. Allgemein wird weniger Zeit benötigt, um eine größere Vielfalt an ähnlichen Programmen zu entwickeln [6].

Bisher müssen fast alle Teilaufgaben bei der Umsetzung eines Codegenerators manuell durchgeführt werden. Werkzeuge wie Language Workbenches können Quelltext bis zu einem gewissen Grad automatisiert generieren oder interpretieren. Sie haben aber in erster Linie die Aufgabe, den Entwickler beim Design von externen domänenspezifischen Sprachen zu unterstützen und dienen als Entwicklungsumgebung für die Arbeit mit der Sprache [7].

1.2 Zielsetzung

Um die zuvor genannten Vorteile der modellgetriebenen Softwareentwicklung noch besser ausnutzen zu können, wird in dieser Arbeit untersucht, ob und wie die Entwicklung eines Codegenerators automatisiert werden kann. Ein generativer Ansatz auf einer höheren Ebene der Indirektion könnte das komplexe Thema der modellgetriebenen Softwareentwicklung weiter vereinfachen und somit Codegenerierung wirtschaftlicher machen.

Im speziellen wird analysiert, wie ein Metagenerator zur Erhöhung der Wirtschaftlichkeit modellgetriebener Softwareentwicklung umgesetzt werden könnte. Zu diesem Zweck wird ein Konzept entwickelt und daraus eine beispielhafte Java-Anwendung erarbeitet, welche es ermöglichen soll, aus vorhandenem Java-Quelltext einen Metagenerator zu erzeugen.

Ein Metagenerator bezeichnet hierbei einen Softwaregenerator, welcher auf Basis einer Referenzimplementation, die Entwicklung anderer Softwaregeneratoren vollständig oder in Teilen automatisiert.

1.3 Aufbau der Arbeit

Die sieben Kapitel dieser Bachelorarbeit sollen Stück für Stück an das komplexe Thema der Metagenerierung heranführen. Da es nicht möglich ist, im Rahmen einer vergleichsweise kurzen Thesis wie dieser sämtliche Grundlagen der Informatik zu beschreiben, wird ein solides Fundament aus Vorwissen, wie man es zum Beispiel in einem Bachelorstudium erwerben kann, vorausgesetzt.

Eingeleitet wird diese Arbeit mit einem Kapitel zur Motivation und Zielsetzung, in welchem aufgezeigt werden soll, warum es sinnvoll ist, sich mit dem Thema der modellgetriebenen Softwareentwicklung zu beschäftigen.

Das zweite Kapitel behandelt die erweiterten, grundlegenden Kenntnisse, die zum Verständnis des Textes notwendig sind. Hier wird sowohl auf die modellgetriebene Softwareentwicklung, als auch auf die Softwarearchitektur und die Verwendung von Design Pattern objektorientierter Programmierung eingegangen.

Aufbauend auf dem vorhergehenden Abschnitt sollte das dritte Kapitel, die Analyse, gut verständlich sein. Hier wird zuerst der wirtschaftliche Aufwand konventioneller Softwareprojekte mit dem Aufwand von Projekten, welche ein modellgetriebenen Ansatz verfolgen, verglichen. Danach werden die Probleme der einzelnen Teilschritte bei der Automatisierung der Entwicklung eines Codegenerators untersucht.

Das Konzept-Kapitel erläutert den im Proof-of-Concept verfolgten Lösungsansatz der analysierten Probleme. Zur Veranschaulichung kommen hier lediglich Diagramme zum Einsatz, dadurch sollte es möglich sein, die allgemeine Idee hinter der Implementation leichter zu verstehen und den Ansatz losgelöst von der Umsetzung weiter zu verfolgen.

Im fünften Kapitel werden zum einen die verwendeten externen Bibliotheken kurz vorgestellt, zum anderen wird genau auf die Architektur der entwickelten Anwendung eingegangen. Die Funktionsweise und der Aufbau jeder Programmkomponente wird anhand von Quelltextauszügen genau erläutert; mithilfe von Beispielen wird die Verwendung der einzelnen Einheiten demonstriert.

Das vorletzte Kapitel der Arbeit evaluiert sowohl das Konzept als auch die Umsetzung der Module. Die Grenzen des Lösungsansatzes werden umfassend diskutiert und mögliche Antworten auf die hieraus entstehenden Fragen angesprochen.

Im siebten Kapitel werden Inhalt und Erkenntnisse der Arbeit noch einmal in gebündelter Form zusammengefasst. Ein ausführlicher Ausblick soll die vielen möglichen Anknüpfungspunkte dieser Thesis aufzeigen.

2 Grundlagen

Um die grundlegenden Zusammenhänge zu verstehen, wird im folgenden auf die verschiedenen Aspekte und Teilschritte der modellgetriebenen Softwareentwicklung eingegangen. Begriffe und Konzepte, die in der Arbeit zur Anwendung kommen, werden eingeführt und definiert. Danach wird auf einige Grundlagen der Softwarearchitektur, vor allem auf die Prinzipien der Softwaretechnik, eingegangen. Da in der Implementation auf einige Design Pattern objektorientierter Programmierung zurückgegriffen wird, werden deren Zweck, Aufbau und Anwendbarkeit abschließend in diesem Kapitel erläutert.

2.1 Modellgetriebene Softwareentwicklung (MDSD)

> „Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."

Die obige Definition stammt aus dem Buch *Modellgetriebene Softwareentwicklung* von Thomas Stahl und Markus Völter [27, S. 11]. Sie lässt sich gut erläutern, wenn man sie in drei Teile zerlegt.

Zunächst einmal ist der Ausdruck formales Modell zu erläutern. Damit ist ein Modell gemeint, welches einen Teil einer Software vollständig beschreibt. Jedoch soll das nicht heißen, dass dieses Modell allumfassend ist. Es muss eindeutig reguliert sein, was genau von diesem Modell beschrieben wird [27, S. 11f.]. Weiterhin bezieht sich die Definition darauf, dass lauffähige Software erzeugt wird. Wird das formale Modell nur zur Dokumentation verwendet oder dient es als Information zur händischen Umsetzung, so kann man das laut Stahl und Völter nicht als modellgetriebene Softwareentwicklung bezeichnen [27, S. 12]. Der letzte Teil der Definition, den die Autoren explizit erläutern, ist, dass die Umwandlung von Modell zu ausführbar Software automatisiert erfolgen soll. Insbesondere soll der Quelltext nicht nur einmal generiert und dann manuell verändert und weiterentwickelt werden, sondern das Modell soll anstelle des Quelltextes treten. Der Quelltext wird aus den geänderten Modellen generiert, dadurch kann aktueller und einheitlicher Quellcode gewährleistet werden [27, S. 13].

In der Literatur findet sich auch die alternative Bezeichnung Model Driven Development (MDD) [25, 2]. Diese Thesis wird jedoch durchgängig die Bezeichnung MDSD verwenden.

Auch der Model Driven Architecture (MDA) Ansatz der Object Management Group (OMG) beschäftigt sich mit modellgetriebener Softwareentwicklung [18]. Dieser Ansatz beschreibt detailliert und umfassend den Gesamtprozess von der Analyse bis hin zur Implementation und führt eigene Standards ein.

Czarnecki und Eisenecker verwenden den Ausdruck Generative Programming, definieren ihn jedoch in den wesentlichen Punkten vergleichbar zu MDSD. Generative Programming strebt nach Czarnecki und Eisenecker nach vollständiger Automation und soll ein vollständiges Zwischen- oder Endprodukt erzeugen [6, S. 5]. Für MDSD merken Stahl und Völter an, dass die verwendeten Modelle nicht unbedingt das vollständige System abbilden. Ein komplettes System enthalte sowohl manuell implementierte als auch automatisch generierte Anteile [27, S. 13].

Durch MDSD soll, wie bereits in der Einleitung dieser Arbeit beschrieben, die Qualität der entstandenen Software gesteigert werden. Dies wird durch den resultierenden einheitlichen Code und die erhöhte Wiederverwertbarkeit erreicht. Außerdem kann potentiell mithilfe der zusätzlichen Abstraktion eine erhöhte Entwicklungsgeschwindigkeit erzielt werden. Ein Bonus von MDSD ist es, dass die Software immer durch aktuelle Modelle beschrieben und somit zumindest in Teilen dokumentiert wird [27, S. 13ff.]. Die Vorteile von Generative Programming [6, S. 13ff.] und MDA [18] werden sehr ähnlich beschrieben.

2.1.1 Domäne

Bei der modellgetriebenen Softwareentwicklung sind, wie bereits erwähnt, Modelle Dreh- und Angelpunkt des Entwicklungsprozesses. Um jedoch ein Modell bilden zu können, muss zuerst untersucht werden, was mit diesem Modell abgebildet wird. Diese Untersuchung ist Teil des Domain Engineerings.

Domain Engineering umfasst die Analyse, das Design und die Implementation einer Domäne. Design und Implementation beziehen sich bereits auf die Zusammensetzung des Systems und dessen technische Umsetzung [6, S. 21f.]. Zur Modellbildung ist für diese Arbeit vor allem die Domänenanalyse interessant.

2.1.1.1 Definition

Eine Domäne beinhaltet laut Czarnecki und Eisenecker das fachliche Wissen über ein Problem-Themengebiet. Jedoch geht die Domäne im Domain Engineering noch darüber hinaus. Sie umfasst nicht nur das fachliche Wissen, sondern auch Informationen darüber, wie Anwendungen für diesen Themenbereich aufgebaut sind. Wichtig ist hierbei, dass sich die Domäne aus dem übereinstimmenden Wissen der beteiligten Stakeholder ergibt. Als Stakeholder werden alle Personen bezeichnet, die ein Interesse an einer bestimmten Domäne haben [6, S. 33].

2.1.1.2 Domänenanalyse

Bereits im 1990 erschienenen Paper *Domain Analysis: An Introduction* zählt der Autor Rubén Prieto-Díaz die drei Grundschritte zur Analyse einer Domäne auf [22]:

1. identification of reusable entities
2. abstraction or generalization
3. classification and cataloging for further reuse

Diese drei Punkte sind auch heute noch zutreffend. In der Analyse wird zunächst beim Domain Scoping die Domäne eingegrenzt. Dies ist notwendig, um eine konsistente Umsetzung möglich zu machen und alle Anforderungen konkret und klar zu definieren [27, S. 239]. Danach werden Entitäten, Operationen und Beziehungen zwischen diesen identifiziert. Sobald dies geschehen ist, können diese auf Ähnlichkeiten und Unterschiede versucht werden. Durch die gefundenen Gemeinsamkeiten, beziehungsweise Variabilitäten, kann nun abstrahiert und generalisiert werden. Festgehalten werden diese Informationen in einem umfassenden Domänenmodell [6, S. 24ff.].

Als Domänenmodell bezeichnen Czarnecki und Eisenecker eine ausdrückliche Darstellung aller Gemeinsamkeiten und Variabilitäten eines Systems in einer Domäne, der Bedeutung dieser Domänenkonzepte und den Abhängigkeiten zwischen den unterschiedlichen Eigenschaften [6, S. 23f.].

2.1.1.3 Feature Modelling

Neben der Domänendefinition, dem Domänenlexikon, welches den verwendeten Glossar auflistet, und konzeptionellen Modellen können auch Featuremodelle Teil des Domänenmodells sein.

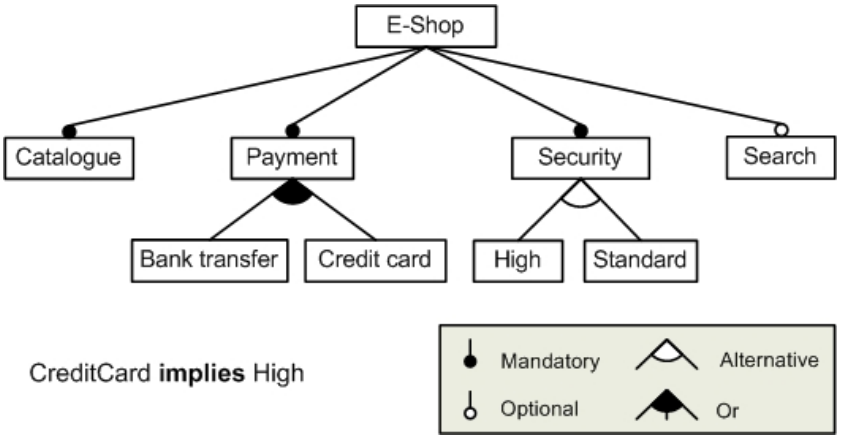

Abbildung 2.1: Darstellung eines Featurediagramms für ein konfigurierbares E-Shop System von Segura09 in der Wikipedia auf Englisch (Transferred from en.wikipedia) [Public domain], via Wikimedia Commons.

Das Feature Modelling erlaubt es, Gemeinsamkeiten und Variabilitäten von Systemen zu dokumentieren. Neben einer genauen Beschreibung aller Features in schriftlicher Form kann man Zusammenhänge und Regeln zur Komposition der Features in einem Featurediagramm darstellen.

Ursprünglich wurde das Featurediagramm als Teil der Feature-Oriented Domain Analysis (FODA) definiert [13]. Es ermöglicht, Merkmale hierarchisch darzustellen und erlaubt es, Beziehungen zwischen ihnen genauer festzulegen. Weiterhin kann unterschieden werden, ob genau eine oder mehrere Eigenschaften der Alternativen Anwendung finden. Zudem können noch weitere Einschränkungen definiert werden, welche es erlauben auch weiter voneinander entfernte Knotenpunkte zueinander in Beziehung zu setzen [27, S. 240f.]. Die Abbildung 2.1 stellt beispielshaft ein Featurediagramm für einen E-Shop dar.

2.1.2 Metamodell

Die Analyse des Problemfelds und Modellierung der Merkmale soll in erster Linie die Grundlage für die Definition des Metamodells der Domäne liefern [27, S. 200].

2.1.2.1 Definition

Zur Beschreibung, welche Mittel zur Definition eines bestimmten Modells zur Verfügung stehen, kann man wiederum ein Modell definieren. Dieses, über dem beschriebenen Modell stehende Modell, nennt man Metamodell. Stahl und Völter bezeichnen es als Be-

schreibung der möglichen Struktur von Modellen. Dazu zählen sie die Konstrukte der Modellierungssprache, auf welche Weise diese zueinander in Beziehung stehen, sowie vorhandene Regeln bezüglich der Gültigkeit beziehungsweise der Modellierung [27, S. 59].

Ein Unified Modeling Language 2.0 (UML) Diagramm, welches alle Java-Sprachkonzepte, wie beispielsweise die Existenz von Klassen, Attributen und Instanzen, beschreibt, ist eine Darstellung für das Java-Metamodell. Geschriebener Java-Quelltext wäre somit eine Instanz dieses Modells, kann aber wiederum instanziiert werden. Handelt es sich nämlich um eine Definition einer Java-Klasse, so können hiervon Objekte mit konkreten Werten existieren.

Es ist noch hervorzuheben, dass mit dem Begriff Metamodell nicht einfach nur eine Abstraktion gemeint ist. Die Silbe Meta kann hier als „die Definition von“ verstanden werden. Ein Modell kann ein anderes Modell abstrahieren, in dem es beispielsweise Informationen auslässt, falls diese für ein Anwendungsfall nicht relevant oder implizit sind. Das ist im obigen Sinne dann jedoch noch kein Metamodell. Es ist also nur bedingt sinnvoll, zu sagen, ein Metamodell sei das Modell eines Modells [28, S. 27].

2.1.2.2 Abstrakte Syntax und Konkrete Syntax

Die in diesem Beispiel definierten Sprachkonzepte sind die abstrakte Syntax für Java. In Abgrenzung hierzu dient eine konkrete Syntax zur Beschreibung eines Modells. Möchte man einen Datentypen definieren, könnte man das mit einem UML Klassendiagramm. Da man den Datentypen auch mit Java-Code beschreiben könnte, sieht man, dass es für dasselbe Metamodell mehr als eine konkrete Syntax geben kann [27, S. 59f.].

2.1.3 Domänenspezifische Sprache (DSL)

Wurde das Metamodell für eine Domäne definiert, das heißt die abstrakte Syntax ist bekannt, wird eine Möglichkeit benötigt, eine Instanz des Metamodells zu bilden. Durch Anreichern des Modells mit Informationen wird eine konkrete Ausprägung des Metamodells geschaffen. Das Mittel hierfür ist eine domänenspezifische Programmiersprache.

2.1.3.1 Definition

Stahl und Völter definieren eine solche Programmiersprache mit: [27, S. 30]. „Eine DSL ist nichts anderes als eine Programmiersprache für eine Domäne.“Martin Fowler hat in

seinem Buch zu diesem Thema eine umfangreichere Definition geschaffen und erläutert diese mit vier Schlüsselelementen [7, S. 27f.].

> „Domain-specific Language (noun): a computer programming language of limited expressiveness focused on a particular domain.“

Der Ausdruck „Computer programming language“ bedeutet, dass eine DSL vom Menschen verwendet wird, um einem Computer Befehle zu geben. Also sollte sie wie jede moderne Programmiersprache durch ihre Struktur sowohl vom Menschen leicht verständlich als auch vom Computer ausführbar sein.

Laut Fowler ist ein weiteres Element die „Language nature“. Als Programmiersprache sollte die Ausdrucksstärke einer DSL nicht nur von den einzelnen Ausdrücke kommen, sondern auch wie diese zusammengesetzt werden.

Mit „Limited expressiveness“ meint Fowler, dass eine DSL, im Vergleich zu allgemeinen Programmiersprachen, nur das für die Beschreibung der Domäne notwendige Minimum an Funktionalität aufweist.

Im letzten Teil seiner Ausführung zu dieser Definition geht Fowler auf den „Domain focus“ ein. Eine beschränkte Sprache sei nur sinnvoll, wenn sie klar auf eine Domäne eingegrenzt ist.

2.1.3.2 General Purpose Language (GPL)

Die von Fowler angesprochenen allgemeinen Programmiersprachen bezeichnet man als General Purpose Language oder auch abgekürzt als GPL. Im Gegensatz zu DSLs sind GPLs nicht auf eine bestimmte Domäne zugeschnitten, sondern können breiter eingesetzt werden. Moderne Programmiersprachen wie Java und C# sind solche Universalsprachen. Durch ihre Turing-Vollständigkeit, kann man sie untereinander austauschen [12, S. 111]. Da sich einzelne Sprachen durch zusätzliche besondere Sprachfeatures voneinander abheben, gibt es mehr als eine GPL [28, S. 27]. Durch die Unterstützung von Pointern in C ist es zum Beispiel potenziell möglich Datenstrukturen besonders speicheroptimiert anzulegen [14, S. 93ff.].

2.1.3.3 Interne DSLs

Bettet man die Befehle zur Beschreibung einer Instanz des Metamodells auf geeignete Weise in eine Universalsprache ein, kann man dies interne DSL nennen. Der Aufbau und

die Folge von Befehlen soll sich bei einer internen DSL, soweit möglich, wie eine eigene Sprache anfühlen [7, S 28].

Für die Implementation von internen DSLs eignen sich dynamische Sprachen wie Ruby und Lisp gut, da sie hilfreiche Features wie die Definition von Makros unterstützen. Je mehr in die normale Syntax einer Sprache eingegriffen werden kann, desto eigenständiger lässt sich die Syntax der internen DSL formen [27, S. 98]. Hierdurch grenzt sich diese immer weiter von einem einfachen Application Programming Interface (API) ab. Gerade wenn das Metamodell bei objektorientierter Programmierung in Form von Klassen vorliegt, kann eine interne DSL eher wie eine einfache Schnittstelle wirken. Die Abgrenzung von internen DSLs zu APIs ist nicht eindeutig und stellt eine Grauzone dar [7, S. 67].

Eine erwähnenswerte Sonderform der intern DSLs sind fragmentierte DSLs. Bei diesem Spezialfall wird die Hostsprache, also die Programmiersprache, mit der die interne DSL ausgedrückt wird, an einzelnen Stellen mit Informationen angereichert. Reguläre Ausdrücke, die als Bruchstücke in einer GPL verwendet werden, kann man hier als Beispiel anführen [7, S. 32].

2.1.3.4 Externe DSLs

Eine externe DSL hat genau wie eine GPL eine eigene Syntax. Laut Fowler ist es jedoch auch nicht unüblich, dass die Syntax einer anderen Sprache, beispielsweise XML, verwendet wird. Sprachen wie die Structured Query Language (SQL) oder Cascading Style Sheets (CSS) sind externe DSLs [7, S. 28]. SQL erlaubt es, mit syntaktisch einfachen Mitteln Anfragen an eine Datenbank zu schicken und mit CSS kann die grafische Repräsentation von HTML-Seiten manipuliert werden.

Im Gegensatz zu internen DSLs, welche auf der Struktur der zugrunde liegenden GPL beruhen, erlauben externe DSLs ihre Syntax weitestgehend frei zu gestalten. Für ihre Implementation kann grundsätzlich auf einen reichen Schatz an Techniken, die zur Verarbeitung von Programmiersprachen seit Jahren Verwendung finden, zugegriffen werden [7, S. 89].

2.1.3.5 Language Workbenches

Martin Fowler nennt bei der Kategorisierung von DSLs als weitere, dritte Kategorie Language Workbenches. Diese sind hochspezialisierte Entwicklungsumgebungen zur Definition dömanenspezifischer Sprachen. Zusätzlich zu der Fähigkeit, in einer solchen Entwicklungsumgebung DSLs zu definieren, dienen sie auch als Entwicklungsumgebung zur Verwendung dieser Sprachen [7, S.28].

2.1.4 Parser

Ein großer Unterschied zwischen internen und externen DSLs besteht in ihrer Verarbeitung. Während interne DSLs eine Datenstruktur direkt mit Informationen befüllen, muss ein Ausdruck einer externen DSL, genau wie bei einem Ausdruck einer GPL, zuerst auf seinen semantischen Informationsgehalt analysiert werden.

2.1.4.1 Einlesen des Quelltextes

Normalerweise liegt der Quelltext, wie der Name schon sagt, in Textform vor. Wenn im folgenden von Quelltext die Rede ist, so ist damit der Source Code von externen DSLs sowie der von GPLs gemeint.

Zur Verarbeitung liest ein Reader den Quelltext ein. Dieser wird dann meist in einem ersten Schritt zeilenweise von einem Lexer in seine Bestandteile (Token) zerlegt. Diese aufgespaltenen Codezeilen können dann auf syntaktische Korrektheit geprüft werden und in eine Übergangsrepräsentation überführt werden [21, S. 29f.]. Eine Form hierfür kann ein Abstract Syntax Tree (AST) sein.

2.1.4.2 Abstract Syntax Tree (AST)

Für jeden wichtigen Token wird im AST ein Knotenpunkt mit für den weiteren Prozess wesentlichen Daten angelegt [21, S. 23]. Die grafische Darstellung des AST für die Prüfung der Parität einer natürlichen Zahl auf Abbildung 2.2 zeigt diese knotenbasierte Strukturierung des eingelesenen Codes. Die Knoten eines AST kennen ihre eigene Position im Baum und somit kann dieser mit geeigneten Strategien durchlaufen werden [21, S. 24].

Eine Möglichkeit ist die Anwendung des Visitor Patterns, welches in dieser Arbeit in einem späteren Unterkapitel behandelt wird.

2.1.4.3 Weitere Verarbeitung des eingelesenen Quelltextes

Nachdem die syntaktische Korrektheit des Quelltextes sichergestellt ist und dieser in einer geeigneten Datenstruktur vorliegt, stehen mehrere Möglichkeiten zur Verfügung. Die jetzt in Baumstruktur vorliegenden Anweisungen sind durch Interpretation von einer Plattform ausführbar. Alternativ könnte auch eine Übersetzung in eine andere Sprache vorgenommen oder direkt zu ausführbaren Maschinencode kompiliert werden.

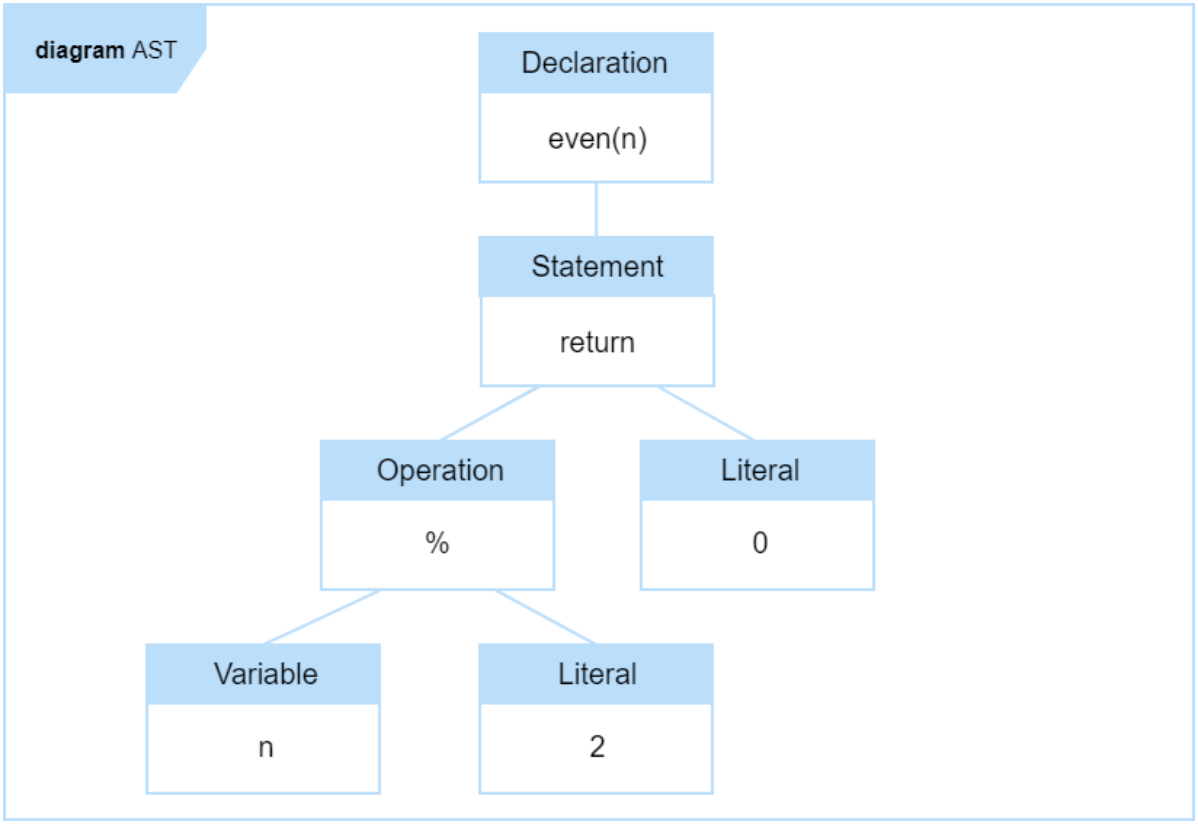

Abbildung 2.2: Darstellung einer Methode zur Prüfung der Parität einer natürlichen Zahl als AST

Bei Java wird der Quelltext beispielsweise eingelesen und zu Byte-Code transformiert, welcher dann von der Java Virtual Machine (JVM) interpretiert werden kann [15]. Die Übersetzung in eine andere Sprache bietet sich sowohl bei externen DSLs als auch bei GPLs an. Allgemein könnte diese Transformation sinnvoll sein, wenn man seinen Code für eine bestimmte Plattform kompatibel zu einer anderen Plattform machen möchte. Ein Beispiel hierfür ist der Kompilierprozess einer in C# geschriebenen Unity3D Anwendung zu einer WebGL App. Hierbei wird der C# Code zuerst in C++ und dann in JavaScript umgewandelt [19]. Native GPLs wie C werden direkt zu Maschinensprache kompiliert [14]. Externe DSLs werden von einem Generator häufig zu GPL Quelltext transformiert [28, S. 26].

2.1.5 Codegenerator

Soll der vom Parser eingelesene Quelltext oder das von einer DSL beschriebene Modell nicht interpretiert, das heißt als Anweisungen an den Computer ausgeführt werden, kann ein Codegenerator zum Einsatz kommen.

2.1.5.1 Definition

Nach Czarnecki und Eisenecker ist ein Generator ein Programm, welches eine Spezifikation von Software auf einer höheren abstrakten Ebene entgegennimmt und dessen Implementation als Artefakt ausgibt [6, S. 333].

Das ausgegebene Artefakt kann, muss aber nicht zwingend, ein gesamtes lauffähiges Softwaresystem sein. Es könnten auch nur einzelne Klassen oder Funktionen generiert werden [6, S. 333]. Sogar in einem noch kleineren Scope kommt Codegenerierung zum Einsatz. Integrierte Entwicklungsumgebungen, wie JetBrains IntelliJ oder Microsofts Visual Studio, erlauben es auf Knopfdruck Code-Snippets, wie zum Beispiel Konstruktoren für eine Klasse, zu generieren.

In der modellgetriebenen Softwareentwicklung dreht sich der gesamte Automatisierungsprozess um das Modell. Es bildet den Ausgangspunkt jeder weiteren Umwandlung. Wie bereits im Abschnitt zu MDSD beschrieben, sollen Änderungen nur am zugrunde liegenden Modell vorgenommen werden und nicht an hieraus entstandenen Generaten. Der Einsatz eines Codegenerators setzt dieses Vorgehen jedoch nicht zwingend voraus.

2.1.5.2 Techniken zur Generierung von Code

Herrington teilt Codegeneratoren in seinem Buch *Codegeneration in Action* in die zwei Hauptkategorien aktiv und passiv ein [11, S. 28].

Passive Codegeneratoren werden eingesetzt, um einmal Quelltext zu generieren, zum Beispiel zur Unterstützung des Programmierers in einer integrierten Entwicklungsumgebung. Der Programmierer verarbeitet diesen generierten Code dann nach Belieben weiter. Im Gegensatz hierzu stehen aktive Codegeneratoren, welche immer wieder zur Ausführung kommen und den generierten Quelltext, abhängig von den Eingabeparametern des Generators, immer wieder regenerieren. Die für MDSD verwendeten Generatoren kann man dieser Kategorie zuordnen, die Eingabeparameter sind hier das konkrete Modell.

Im Zusammenhang mit MDSD gibt es zwei Möglichkeiten der Generierung von Code aus dem konkreten Modell. Fowler nennt diese zwei Formen der Generierung Transformer Generation und Templated Generation [7, S. 124f.].

Bei der Transformer Generation werden aus einem konkreten Modell Statements einer Zielsprache generiert. Hierfür wird das Modell traversiert und der Generator setzt aus den im Modell vorliegenden Informationen zum Aufbau des Codes Ausdrücke zusammen. Templated Generation basiert auf Textersetzung. Es wird eine Codevorlage, das Template, geschrieben, in welcher Variabilitäten auf besondere Weise markiert sind. Aus den Daten des konkreten Modells werden Werte für diese Variabilitäten vom Generator ausgelesen und im Template eingesetzt.

Diese Ansätze sind nicht nur voneinander getrennt verwendbar. Beim Einsatz von Transformer Generation ist es möglich, dass der ausgegebene Quelltext zwar komplett neu generiert wird, aber einzelne Teile dieses Quelltextes auf Basis von kleinen Templates erstellt werden [7, S. 125].

2.1.5.3 Zusammenhang mit Transformatoren

Es ist denkbar, ein von einer DSL beschriebenes Modell durch weitere Umwandlungsschritte weiter zu verarbeiten, bevor abschließend Quelltext generiert wird. Dies kann Sinn machen, um ein Modell so aufzubereiten, dass ein bestimmter Generator es verarbeiten kann [27, S. 195]. Eine Komponente, die diese Transformation durchführt, nennt man Transformator. Wird ein Modell zu Text transformiert, so nennt man das Codegenerierung, durchgeführt von einem Generator [28, S. 271].

Ein konkreter Anwendungsfall wäre, wenn ein Generator, der eine konkrete Instanz

eines bestimmten Metamodells zu Code verarbeiten kann, bereits implementiert vorliegt, jedoch das von einer DSL beschriebene Metamodell von dem für den Generator verständlichen Metamodell abweicht. Eine Modell-zu-Modell-Transformation könnte hier eine Brücke schlagen und somit den bereits vorhandenen Generator nutzbar machen. Dies könnte unter Umständen, vor allem wenn die Metamodelle bereits eine ähnliche Struktur aufweisen, den Arbeitsaufwand reduzieren.

Die Komplexität der Umwandlung von einem Metamodell in das andere könnte durch mehrere Zwischentransformationen reduziert werden. Ein denkbarer Extremfall hierfür wäre die Reduktion der Transformationen auf nur eine einzelne Transformationsregel pro Modell-Transformationsschritt. Dies ist aber nichtmehr zwingend eine Vereinfachung.

2.1.5.4 Abgrenzung zu Compilern

In der Literatur wird ein Generator stellenweise auch Compiler genannt [28, S. 26]. Das Erzeugen von Quelltext nennt sich auch kompilieren [7, S. 19].

Bei genauerem Hinsehen ist die Abgrenzung von Generatoren zu Compilern im Zusammenhang mit MDSD möglich. Bei der modellgetriebenen Softwareentwicklung wandeln Generatoren DSL-Code in konkreten Code einer GPL um. Normalerweise wird aus DSL-Code nicht direkt Bytecode oder Maschinencode generiert, da wir durch den Schritt über GPL-Code den Compiler der GPL mit all seinen Features verwenden können [28, S. 11].

Wie beschrieben, wandelt ein Compiler GPL-Code zu Bytecode oder Maschinencode um. Dabei nimmt er Optimierungen, wie zum Beispiel zur Verbesserung der Geschwindigkeit oder des Speicherverbrauchs eines Programms, vor. Das vom Compiler erzeugte Artefakt kann auf der Zielplattform direkt ausgeführt werden [6, S. 345ff.].

Da eine Beschreibung des Kompilierprozesses und die verwendeten Techniken im Compilerbau den Rahmen dieser Arbeit sprengen würden, wird hier nicht tiefer auf das Thema eingegangen, sondern auf das „Drachenbuch“ als weiterführende Literatur verwiesen [1].

2.2 Software Engineering

Software Engineering, auch bekannt unter der deutschen Bezeichnung Softwaretechnik, wird von Helmut Balzert in seinem Lehrbuch der Softwaretechnik folgendermaßen definiert [3, S.17]:

> Softwaretechnik: Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen fürdie arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen. Zielorientiert bedeutet dieBerücksichtigung z. B. von Kosten, Zeit, Qualität.

Es geht also um die wirtschaftliche Anwendung von erprobten Lösungen bei der Umsetzung großer Softwaresysteme. Außerdem soll dies systematisch geschehen, das heißt es soll begründet werden, wieso ein Prinzip, eine Methode oder ein Werkzeug zur Lösung eines bestimmten Problems verwendet wurde. In der Softwaretechnik geht es nicht darum, das Rad jedes Mal neu zu erfinden. Vielmehr soll der vorhandene Stand der Technik strukturiert angewandt werden.

Für den Bereich der Prinzipien in der Softwaretechnik hat Balzert einige grundlegende Konzepte zusammengefasst und diese in Relation zueinander gestellt. Da diese Prinzipien einen wichtigen Teil der Softwareentwicklung ausmachen und für diese Arbeit systematisch eine Software entwickelt wurde, sind diese auch hier sehr relevant.

2.2.1 Prinzipien der Softwaretechnik

Ein Prinzip liegt als Basis dem zugrunde, was wir tun. Sie sind weder konkret, noch beziehen sie sich auf Spezialfälle [3, S. 25].

2.2.1.1 Abstraktion

Abstrahieren bedeutet, etwas zu verallgemeinern. Das bedeutet, sich von besonderen Eigenschaften einer Sache zu lösen und wesentliche Dinge und gleiche Merkmale, die mehrere Instanzen dieser Sache haben, hervorzuheben. Im Gegensatz zur Abstraktion steht die Konkretisierung [3, S. 26].

In der Softwareentwicklung, im Rahmen objektorientierter Programmierung, ist eine Form der Abstraktion der Aufbau von Vererbungsstrukturen. Hier werden allgemeine Eigenschaften in eine Oberklasse abstrahiert und andere Klassen erben hiervon und fügen eigene, spezielle Eigenschaften hinzu.

Abstraktion und Modellbildung treten häufig gemeinsam auf. Möchte man in einer Anwendung beispielsweise Personen verwalten, so ist es kaum möglich, alle Eigenschaften, die eine reale Person mitbringt, als Modell abzubilden.

Bei einer Kundenverwaltung sind für eine Anwendung vermutlich Attribute wie der

Name, das Passwort und die Adresse eines Kunden relevant. Zusätzliche Merkmale dieser Person, wie zum Beispiel die Haarfarbe, werden nicht abgebildet. Der Grund hierfür könnte sein, dass dies für eine Kundenverwaltung nicht relevant ist. Die Haarfarbe wurde abstrahiert.

Selbst wenn ein Modell so genau wie möglich sein soll, muss an einer Stelle ein Schlussstrich gezogen werden. Ist die Haarfarbe relevant, könnte man hier rot oder blond als Haarfarbe abbilden. Man könnte jedoch auch noch Abstufungen unterscheiden. Statt der Abbildung des vereinfachenden blond könnte man die Farbwerte einpflegen. Es kann festgehalten werden, dass Abstraktion ab einem gewissen Punkt notwendig ist.

2.2.1.2 Modularisierung

Das Prinzip der Modularisierung steht in direktem Zusammenhang mit der Abstraktion. Modularisierung bedeutet, ein System in Module zu zerlegen. Dieses Konzept kommt zum Beispiel besonders häufig bei Elektrogeräten vor. Ein Computer hat mehrere, einzeln entwickelte, Bauteile. Der Arbeitsspeicher ist von einem anderen Hersteller als die Grafikkarte. Über eine definierte Schnittstelle, die Anschlüsse auf der Hauptplatine, können Arbeitsspeicher und Grafikkarte gemeinsam in einem System betrieben werden.

Ein Modul stellt eine funktionale Einheit oder eine auf semantische Weise verbundenen Funktionsgruppe dar. Ebenso ist ein Modul weitgehend Kontext unabhängig, das heißt, es kann für sich stehend verstanden, entwickelt, geprüft und gewartet werden. Gekoppelt sind die Module über eine klar erkennbare Schnittstelle [3, S. 41].

2.2.1.3 Weitere Prinzipien und Abhängigkeiten zwischen den Prinzipien der Softwaretechnik

Es gibt neben den genannten Prinzipien noch einige weitere, die hier nicht tiefer behandelt werden. Das Prinzip der Strukturierung besagt, dass ein Softwaresystem in einer bestimmten definierten Struktur vorliegt [3, S. 34ff.]. Bindung und Kopplung beschreiben strukturelle Eigenschaften eines Softwaresystems. Bindung bezeichnet hierbei die Kompaktheit einer Systemkomponente. Ist ein System stark gebunden, so haben die einzelnen Komponenten ihre klar definierten Verantwortlichkeiten. Im Gegensatz zur Bindung steht die Kopplung, diese ist die Assoziation und Vererbungsstruktur zwischen verschiedenen Klassen und Paketen. Je höher der Kopplungsgrad ist, desto größere Auswirkungen haben Änderungen in einer Komponente auf andere, hiermit gekoppelte, Komponenten [3, S. 37f.]. Wenn ein System nach einer Rangordnung angeordnet ist, so besitzt es eine Hierarchie. Dies ist eine stärkere Form der Strukturierung und verhindert, dass ein Softwaresystem eine chaotische Struktur entwickelt [3, S. 39ff.]. Unter Geheim-

nisprinzip versteht man eine schärfere Version der Modularisierung. Für den Anwender soll eine Systemkomponente keinen Einblick in die eigene Funktionsweise bieten, sondern nur korrekt funktionieren [3, S. 42ff.]. Lokalität bedeutet, dass für eine Aufgabe wichtige Informationen nicht zusammengesucht werden müssen, sondern zum Beispiel als Felder in einer Klasse bereits vorliegen [3, S. 45f.]. Mit dem Prinzip der Verbalisierung ist unter anderem gemeint, dass Komponenten, wie Klassen, Methoden und Felder eine aussagekräftige Namensgebung haben, da dies bis zu einem gewissen Grad zu einer Selbstdokumentierung des Codes führt [3, S. 46ff.].

Die Prinzipien der Softwaretechnik stehen nicht losgelöst im Raum, sondern sie bedingen sich teilweise oder stehen zueinander in Wechselwirkung. Bindung und Kopplung benötigen zum Beispiel Modularisierung beziehungsweise Strukturierung. Das Geheimnisprinzip kann ebenfalls nur auf Basis von Modularisierung Anwendung finden. Ein Graph, der diese Zusammenhänge abbildet, findet sich bei Balzert [3, S. 49].

2.2.1.4 Zusammenhang zwischen den Prinzipien der Softwaretechnik und der Entwicklung von Codegeneratoren

Wie bereits eingangs erwähnt, haben die in vorherigen Abschnitten beschriebenen Prinzipien einen direkten Zusammenhang mit der Entwicklung von Codegeneratoren.

Insbesondere sticht die für die modellgetriebene Softwareentwicklung unerlässliche Abstraktion durch Modellbildung heraus. Ein Metamodell stellt die Verallgemeinerung der relevanten Konzepte einer Domäne dar. Eine verwendete DSL zur Beschreibung des Metamodells abstrahiert demnach Konzepte einer GPL oder einer Domäne.

Modularisierung findet bei MDSD durch die Einteilung der einzelnen Arbeitsschritte in einzelne Softwarekomponenten statt. Ein Metamodell hat eine definierte abstrakte Syntax. Eine hieraus gebildete konkrete Syntax ist eine Schnittstelle für das Metamodell. Ein Parser, als weitere Komponente, liest eigenständig Quelltext ein und bildet diesen auf einen AST ab. Mit einer Schnittstelle kann der eingelesene Quelltext weiterverarbeitet werden. Der Codegenerator ist hier unabhängig vom konkret gewählten Parser, er verarbeitet nur den AST und ein anderes, aus dem Parseprozess hervorgegangenes, Modell. Auch für den Parser ist es nicht relevant, wie der einzulesende Quelltext entstanden ist, er muss lediglich syntaktisch korrekt sein.

2.3 Design Pattern objektorientierter Programmierung

Das viel beachtete Werk *Design Patterns: Elements of Reusable Object-Oriented Software* stellt einen Katalog auf, welcher Beschreibungen für Muster liefert, die allgemeine Designprobleme in einem bestimmten Kontext lösen [8]. Diese Muster, von den Autoren des Werkes Pattern genannt, haben grundsätzlich vier Elemente:

Das erste Element ist der Name, welcher verwendet werden kann, um auf das Pattern zu referenzieren. Dies ermöglicht, allein schon durch die Erweiterung des Vokabulars, Software mit erhöhter Abstraktion zu entwerfen. Des Weiteren gehört zu jedem dieser Muster eine Beschreibung, bei welcher Art von Problem es anwendbar sein könnte. Als dritter Teil eines Patterns werden die Elemente, welche verwendet werden, um das beschriebene Problem zu lösen, abstrakt erläutert. Diese Lösungen beschreiben kein konkretes Design oder eine konkrete Implementation, da das Muster in verschiedenen Situationen Anwendung finden kann. Jedes Muster bringt Vor- und Nachteile. Diese Konsequenzen sind wichtig zum Verständnis eines Patterns und werden daher explizit aufgeführt [9, S.30 f.].

Modellgetriebene Softwareentwicklung stellt den Anwender im Zusammenhang mit objektorientierter Programmierung vor Probleme, für deren Lösung bestimmte Design Pattern hilfreich sein können. Die Beschreibung dieser Entwurfsmuster folgt in dem oben zitierten Werk einem konsistenten Aufbau. Die folgenden Abschnitte dieser Arbeit orientieren sich an dieser Vorlage. Zunächst wird das Pattern benannt, dann wird kurz der Zweck, also was mit der Verwendung des Patterns erreicht werden soll, zusammengefasst. Anschließend wird erklärt, wann das Pattern anwendbar sein könnte und zum Abschluss wird in Form eines Diagramms das behandelte Muster dargestellt. Erklärungen zur verwendeten Notation können auf den ersten Seiten der aktuellen deutschen Übersetzung des Design Pattern Werkes entnommen werden [9, S. 8]. Der genaue Aufbau und der Ablauf der Kommunikation zwischen den Akteuren des Patterns wird in dieser Arbeit nur oberflächlich erläutert. Die Intention dieses Abschnitts ist, dass die aufgezeigten Muster, wenn diese später erwähnt werden, in ihrer grundlegenden Funktion bekannt sind.

Das Factory Method Pattern bezeichnet hier nicht das von Gamma et al. vorgeschlagene Factory Method Muster [9, S. 173ff.], sondern bezieht sich auf die von Bloch in *Effective Java* beschriebene Static Factory Method [4, S. 5ff.].

2.3.1 Visitor Pattern

Das Visitor Pattern oder auch Besuchermuster ist ein objektbasiertes Verhaltensmuster [9, S. 480].

2.3.1.1 Zweck

Dieses Muster erlaubt es, Operationen auf Elementen einer Objektstruktur zu definieren, ohne die eigentlichen Klassen der Elemente ändern zu müssen [9, S. 480].

2.3.1.2 Anwendbarkeit

Das Besuchermuster kann Anwendung finden, wenn eine Struktur aus Objekten mit unterschiedlichen Schnittstellen vorliegt und auf diesen Objekten, abhängig von ihrer konkreten Klasse, Operationen auszuführen sind. Dies schützt vor allem davor, dass die Klassen dieser Objekte für jede Anwendung die jeweiligen Operationen implementieren müssen. Wird die Objektstruktur verwendet, so können die notwendigen Operationen lokalisiert definiert werden. Das Besuchermuster erlaubt es leicht, neue Operation hinzuzufügen, sollte sich jedoch die Objektstruktur häufiger ändern, so müssen jedes Mal die Visitor-Schnittstellen neu definiert werden. Hier könnte es besser sein, die Operationen in den betroffenen Klassen zu definieren [9, S. 484].

Ein Anwendungsbeispiel für das Visitor Pattern bei modellgetriebener Softwareentwicklung ist die Verarbeitung der Nodes eines AST. Der Baum stellt hier die erwähnte Objektstruktur dar und dessen Knoten die Elemente [9, S. 480].

2.3.1.3 Struktur

Die Abbildung 2.3 zeigt den Aufbau des Visitor Patterns. Sind die entsprechenden Klassen auf diese Weise implementiert, so kann jederzeit, wenn eine Operation auf den Elementen einer Datenstruktur ausführbar gemacht werden soll, ein konkreter Visitor implementiert werden, welcher diese Operation ausführt. Traversiert man die einzelnen Objekte der Struktur, kann auf jedem dieser Objekte die Accept-Operation aufgerufen werden. Diese führt dann die entsprechende Visit-Operation auf dem übergebenen konkretenen Visitor aus. Es ist möglich, dass nicht eine einzigartig benannte Visit-Operation für jedes konkrete Element existiert, sondern, dass die Operation durch Funktionsüberladung einer aussagekräftig benannten Visit-Methode implementiert wird [9, S.485 ff.].

2.3.2 Builder

Das Pattern Builder oder im deutschen auch Erbauer ist ein objektbasiertes Erzeugungsmuster [9, S. 159].

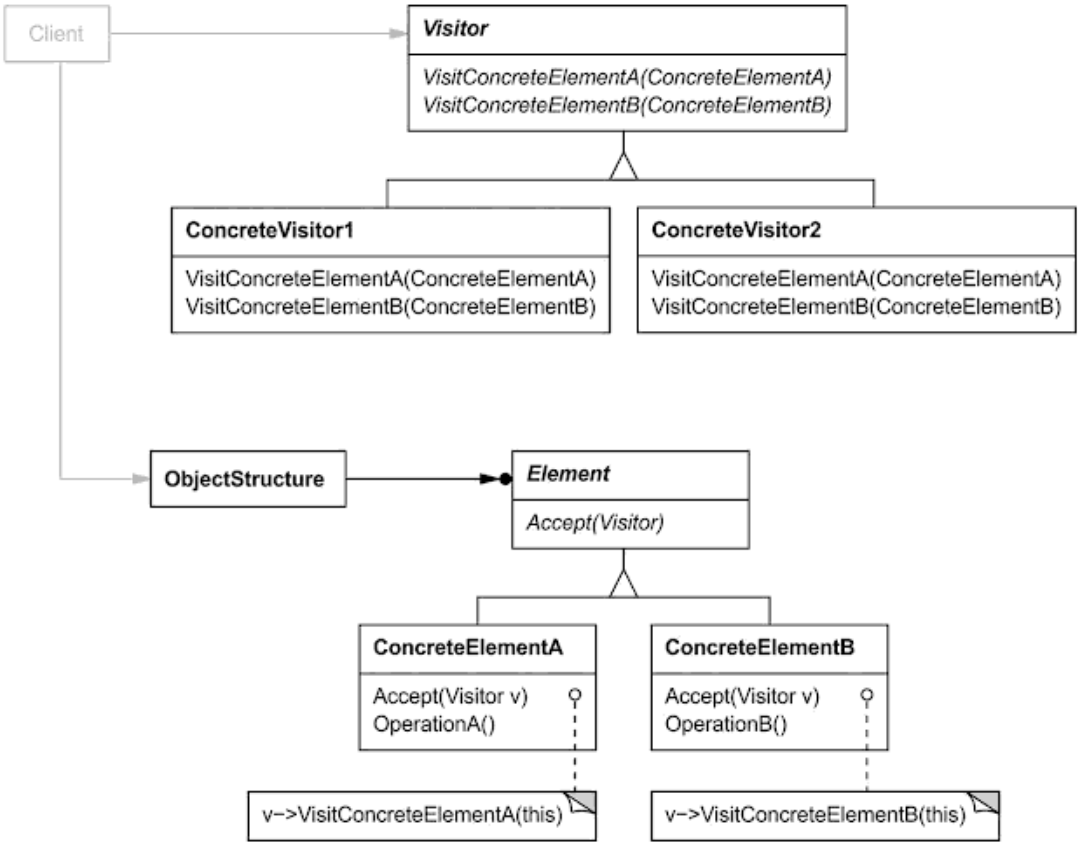

Abbildung 2.3: Darstellung der Struktur des Visitor Design Patterns, entnommen aus [9, S. 485].

2.3.2.1 Zweck

Ein Erbauer soll die Erzeugung eines komplexen Objektes von dessen Repräsentation trennen. Somit ist es möglich, unterschiedlich aufgebaute Instanzen einer Klasse zu erzeugen [9, S. 159]. Eine spezielle Form des Builders, die Joshua Bloch in *Effective Java* vorschlägt, dient dazu, komplexe Objekte mit vielen optionalen Parametern vereinfacht zu konstruieren [4, S. 10ff.].

2.3.2.2 Anwendbarkeit

Soll die Logik, die bei der Erzeugung eines komplexen Objektes verwendet wird, losgelöst sein von dem eigentlichen Objekt oder soll das komplexe Objekt optionale Parameter ermöglichen, könnte man einen Builder verwenden.

Ein Anwendungsfall für die Verwendung eines Builders stellt die Implementation einer internen DSL dar. Das beschriebene komplexe Objekt ist hier die konkrete Instanz des Metamodells [7, S. 343ff.].

2.3.2.3 Struktur

Wie bei vielen der von Gamma et al. beschriebenen Pattern, definiert eine Abstrakte Klasse eine Schnittstelle mit den zur Erzeugung verwendbaren Methoden, welche dann von einer konkreten Klasse implementiert werden. Hier sind das die Klassen Builder und ConcreteBuilder. Ein Director verwendet diese Schnittstelle und erzeugt damit ein konkretes Produkt [9, S. 162f.]. Eine mögliche Abwandlung davon wäre die von Bloch beschriebene Erweiterung der BuildPart-Methoden, um eine Referenz auf das aktuelle Builder Objekt als Rückgabewert. Der Builder hält den aktuellen Zustand des erzeugten Objekts und erst wenn bei der dadurch entstehenden Methodenkaskade eine abschließende Build-Methode aufgerufen wird, gibt der Builder das erzeugte Objekt zurück [4, S. 13ff].

2.3.3 Factory Method

Obwohl die Factory Method, oder auch Fabrikmethode, in dieser Form nicht von Gamma et al. in ihrem Katalog beschrieben wurde, könnte man sie als ein objektbasiertes Erzeugungsmuster bezeichnen, da sie ebenfalls Objekte einer Klasse instanziiert.

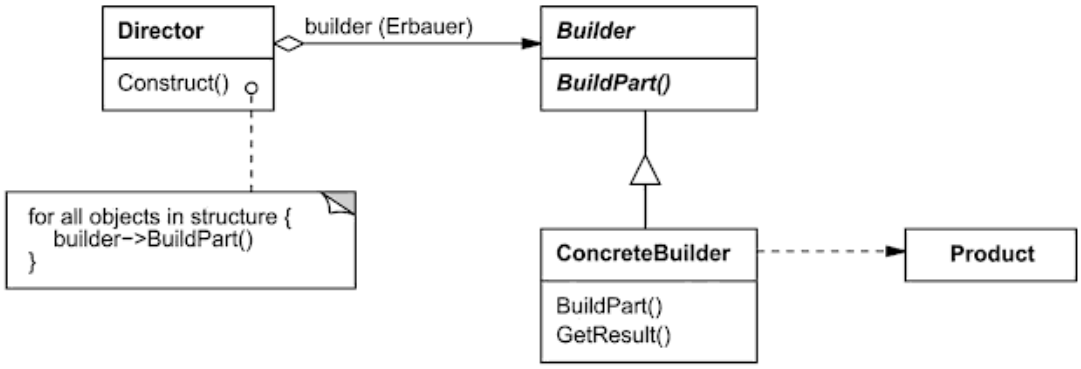

Abbildung 2.4: Darstellung der Struktur des Builder Design Patterns, entnommen aus [9, S. 162].

2.3.3.1 Zweck

Ziel einer Factory Method ist es, anstelle von speziellen Konstruktoren, statische Methoden zur Erzeugung von Objekten bereitzustellen. Da eine Methode im Gegensatz zu Konstruktoren benannt werden kann, ist es möglich die erzeugte Instanz eindeutiger zu beschreiben [4, S. 5f].

2.3.3.2 Anwendbarkeit

Die Verwendung von Fabrikmethoden kann sinnvoll sein, wenn die Parameter eines Konstruktors das erzeugte Objekt nicht eindeutig beschreiben. Hier kann eine sprechend benannte statische Fabrikmethode eine genauere Beschreibung bieten und die Lesbarkeit des Quelltextes erhöhen. Es ist nicht möglich, Konstruktoren mit gleicher Signatur aber unterschiedlichem Verhalten bereitzustellen. Statische Fabrikmethoden können solche Konstruktoren ersetzen. Außerdem können sie Objekte eines Subtypen zurückgeben, somit kann die eigentliche instanziierte Klasse versteckt werden [4, S. 5ff].

In Verbindung mit einem Enum oder String als Parameter der statischen Fabrikmethode, kann die Erzeugung eines komplexen Objekts mit einer aussagekräftigen Benennung gesteuert werden.

2.3.3.3 Struktur

Da für die statische Fabrikmethode nach Bloch kein Gegenstück im Katalog Design Pattern von Gamma et al. existiert [4, S. 5] und die Struktur lediglich durch das Hinzufügen

weiterer Methoden zu einem Objekt besteht, wird hier auf eine Abbildung verzichtet.

Eine statische Fabrikmethode ist eine von außen zugängliche statische Methode einer Klasse mit beliebigen Parametern, die, wie ein Konstruktor, ein Objekt dieser Klasse zurückgibt. Sie kann die Konstruktoren einer Klasse vollständig ersetzen oder die Klasse nur erweitern [4, S. 5f]. Die statische Fabrikmethode könnte auch in eine extra Fabrik Klasse ausgelagert werden, um so Logik für die Instanziierung verschiedener konfigurierter Objekte einer Klasse lokal zu sammeln.

3 Analyse

> „Wie kann, ausgehend von bestehendem Java-Code, die Entwicklung eines Generators zur Erhöhung der Wirtschaftlichkeit modellgetriebener Softwareentwicklung automatisiert werden?“

Auf Basis der Grundlagen ist es jetzt möglich, diese Fragestellung eingehend zu analysieren. Das folgende Kapitel wird zuerst auf einen Vergleich des Aufwands bei konventionellen Softwareprojekten und modellgetriebener Softwareentwicklung eingehen. Es soll deutlich werden, wo die wirtschaftlichen Hürden bei MDSD liegen. Der zweite Abschnitt dieser Analyse beschäftigt sich Schritt für Schritt mit den Schwierigkeiten, die bei der Umsetzung eines Proof-of-Concept zur Beantwortung der obigen Fragestellung auftreten können.

Zu jedem analysierten Problem werden mögliche Lösungswege aufgeführt und ihre Vor- und Nachteile erörtert. Das Ziel dieses Abschnitts ist es, dass im nachfolgendem Konzept-Kapitel auf diese Analyse zurückgegriffen werden kann und somit der im Konzept gewählte Ansatz begründ- und nachvollziehbar ist. Wenn in den folgenden Abschnitten im Zusammenhang mit Generierung und Automatisierung von Apps, Anwendungen und Produktfamilien die Rede ist, so ist dies beispielhaft gemeint. Die beschriebenen Probleme beziehen sich auch auf einen kleineren Skopus, zum Beispiel für Module oder Klassen, fallen in diesen Zusammenhängen aber entsprechend kleiner aus.

3.1 Aufwand eines konventionellen Softwareprojektes im Vergleich zu MDSD

Die Umsetzung einer Anwendung hat unterschiedliche Etappen. Zu Beginn der Entwicklung wird spezifiziert, was entwickelt werden soll, also wie die Anforderungen an die Anwendung aussehen. Diese ist gefolgt von einer Entwurfsphase, in welcher beschrieben wird, wie die Anwendung aussieht und aus welchen Bausteinen sie besteht. Als letzter Schritt wird das in der Entwurfsphase erarbeitete Konzept implementiert [3, S. 62]. Abhängig vom verwendeten Entwicklungsprozess, wird parallel zu dieser Entwicklung oder im Anschluss daran die Qualität durch Tests gesichert. Wurden diese Etappen

durchlaufen, ist die App bereit für eine Veröffentlichung.

Die Idee der modellgetriebenen Softwareentwicklung ist es, dass man vereinfacht und ab einem gewissen Punkt wirtschaftlich, ganze Anwendungen oder Anwendungsteile einfacher, schneller und wiederverwertbar generieren kann [27, S. 14f.]. Ein Beispiel hierfür wäre die Umsetzung verschiedener konfigurierter Ausprägungen einer Softwarefamilie [27, S. 237ff.]. Wie aus dem vorherigen Kapitel Grundlagen hervorgeht, hat auch MDSD grundsätzlich mehrere Entwicklungsschritte. Um Software mit einem Modell beschreiben und daraus generieren zu können, muss bekannt sein, wie diese aufgebaut ist und funktioniert. Ein erster Schritt bei MDSD kann also sein, eine Ursprungs-App zu entwickeln, welche als Referenz für den weiteren Entwicklungsprozess dient. Danach kann diese Referenzimplementation analysiert werden. Hierunter fällt auch das in Grundlagen beschriebene Domain Engineering und die Definition eines Metamodells. Hierauf aufbauend kann zum einen eine domänenspezifische Sprache und zum anderen ein Generator entwickelt werden, welcher aus einer Instanz des Metamodells verschiedene Ausprägungen der Software generieren kann.

Für die Umsetzung einer Referenzimplementation muss der gleiche Prozess durchlaufen werden, wie der für das konventionelle Entwickeln einer App [27, S. 219f.]. Da bei MDSD noch eine umfangreiche Analysephase und die Entwicklung des Generators durchgeführt werden müssen, kann modellgetriebene Softwareentwicklung frühestens mit der Entwicklung einer zweiten Anwendung einer Softwarefamilie oder vielleicht sogar erst zu einem späteren Zeitpunkt wirtschaftlich werden. Die Abbildung 3.1 stellt diesen Vergleich grafisch vereinfacht dar.

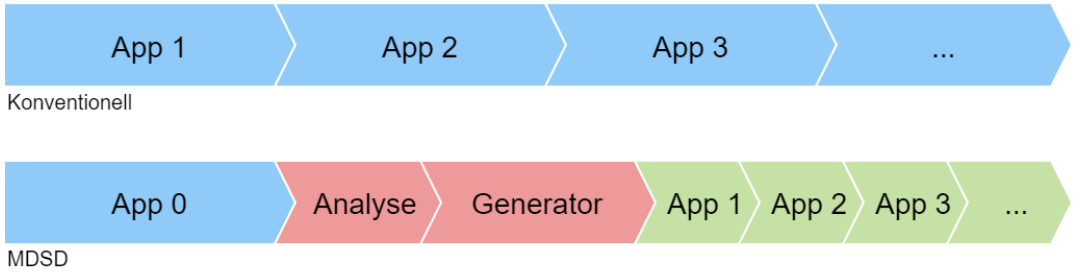

Abbildung 3.1: Vereinfachte Darstellung des Aufwandsvergleichs eines konventionellen Softwareprojektes und MDSD.

Ein zusätzlicher Faktor, der die Wirtschaftlichkeit von MDSD beeinträchtigen könnte, ist, dass sich ein Entwicklungsteam bei der Verwendung eines modellgetriebenen Ansatzes normalerweise aufteilt. Ein Teil der Entwickler erweitert DSLs und Generatoren und der andere Teil verwendet diese. Diese Abhängigkeit muss in der Ablaufplanung des Projektes berücksichtigt werden. Da sich bei MDSD nicht um konventionelle Softwareentwicklung handelt, kann es zusätzlich sein, dass die Konzepte einem Entwicklungsteam erst näher gebracht und etwaige Kompetenzen erst aufgebaut werden müssen [28, S. 44ff.].

Betrachtet man den MDSD Entwicklungsprozess strukturiert von Anfang bis Ende, ist gut zu sehen, an welcher Stelle eine Erhöhung der Wirtschaftlichkeit denkbar wäre.

Die Entwicklung einer Referenzimplementation kann, wie bereits beschrieben, kaum gekürzt werden. Eine vollständige Analyse ist nur möglich, wenn auch eine zu analysierende Anwendung existiert. Da mit dem Generator konkreter Quelltext erzeugt werden soll, reicht auch eine reine Kozeptionierung der Referenzimplementation kaum aus. Diese sollte codiert vorliegen. Es könnte möglich sein, mithilfe von zum Beispiel Feature Modelling einen großen Teil einer Domäne losgelöst von einer konkreten Implementation zu analysieren. Spätestens, wenn es darum geht, die zu generierende Architektur zu untersuchen, ist die Referenzimplementation eine wichtige Hilfestellung und Ausgangspunkt [27, S. 123f.].

Da die Analyse der Domäne und die Entwicklung des Metamodells sich nicht nur auf die Referenzimplementation stützt, sondern sich auch auf semantische, in der Referenzimplementation nicht zwingend festgehaltene, Zusammenhänge bezieht, ist sie nur sehr schwer und wenn überhaupt nur in Teilen automatisierbar.

Im Gegensatz dazu steht die geradezu mechanisch anmutende Generierung von konkretem Quelltext aus einer Instanz eines beschriebenen Metamodells. Ist durch die Referenzimplementation und Analyse bekannt, wie die allgemeine Architektur für eine Anwendung aussieht und durch das Metamodell welche Variabilitäten vorliegen, so folgt die Erzeugung von Quelltext eindeutig vorher definierbaren Regeln. Konkrete Instanzen von Modellen, die für einen solchen Generator verständlich sind, müssten also dann alle auf dem gleichen Metamodell beruhen. Eine DSL zur Instanziierung eines solchen Metamodells könnte dann entweder immer gleich aussehen oder davon abhängen, welche Variabilitäten, basierend auf einer Referenzimplementation, überhaupt beschreibbar sein müssen. Unabhängig hiervon besteht in diesem Schritt somit Potenzial den MDSD Prozess durch Automatisierung zu verkürzen. Die folgende Abbildung 3.2 zeigt dieses Potenzial in vereinfachter grafischer Darstellung noch einmal auf.

Der Aufwand des letzten Schritts, der eigentlichen Anwendung des Generators, hängt von den oben bereits erwähnten Kompetenzen der Entwickler und vor allem mit der zur Modellbeschreibung verwendeten DSL zusammen. Je einfacher die DSL verwendet werden kann, je eindeutiger und eingängig ihre Syntax und je stärker die durch die DSL ermöglichte Abstraktion ist, desto produktiver kann mit ihr gearbeitet werden. Um dies bei DSLs zu erreichen haben Stahl und Voelter einige Best-Practices zur Definition von DSLs formuliert [27, S. 113]. Einsparungen können hier also im Grunde nur im Vorfeld, also bei der Analyse unter Umsetzung des Generators ausgelöst werden. Welche Ausprägung einer Anwendung genau generiert werden soll, muss ein Entwickler immer noch manuell schreiben.

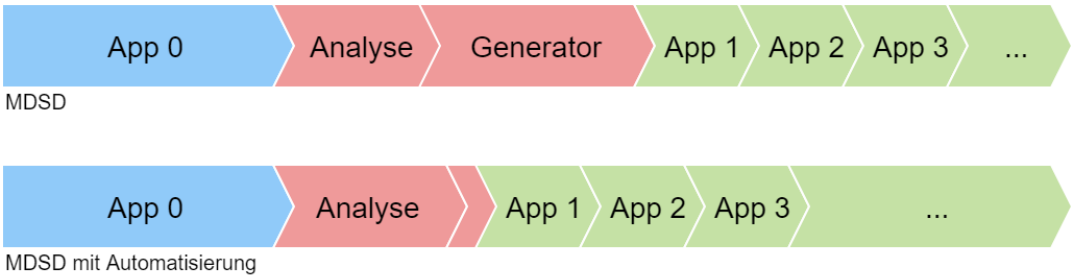

Abbildung 3.2: Vereinfachte Darstellung des Einsparungspotentials im Generator Implementations-Schritt bei MDSD.

3.2 Automatisierung der Entwicklung eines Codegenerators

Wie aus dem vorherigen Abschnitt entnommen werden kann, besteht ein großes Automatisierungspotenzial darin, den gesamten Generator oder Teile davon automatisiert zu erzeugen. Hierbei treten einige Schwierigkeiten auf, welche im Folgenden genau analysiert werden. Im Rahmen dieser Arbeit wird davon ausgegangen, dass eine Referenzimplementation bereits in Form von Java-Quelltext vorliegt.

3.2.1 Java-Code als Ausgangsmodell

Um die existierende Referenzimplementation weiter verarbeiten und hieraus ableiten zu können, welche Variabilitäten in Generaten auf Basis dieser Referenz existieren, müssen Informationen hierzu festgehalten und für den Metagenerator zugänglich gemacht werden. Das bedeutet, dass zum Beispiel zur Implementation einer Java-Klasse zusätzlich die Information vorliegen muss, dass in einer erzeugten Variante dieser Klasse der Identifier variabel bestimmt werden kann.

3.2.1.1 Anreicherung des Java-Codes mit Informationen

Es gibt mehrere Möglichkeiten, zusätzliche Informationen zum vorhandenen Java-Quelltext anzubringen. Ein Ansatz hierfür wäre, dies als zusätzliches Modell, zum Beispiel in einer XML Datei [5], zu formulieren. Hier könnten dann unter der Verwendung von XML-Tags und Attributen Referenzen zu bestehenden Klassen gebildet werden. Beim Einlesen des Java-Quelltextes würde dann auch das zugehörige XML-Dokument geparsed und nach korrespondierenden Informationen durchsucht werden. Mit XML-Schema-Definitionsdateien (XSD) [10] und Unterstützung von geeigneten APIs wie JAXB [16]

könnte sehr genau beschrieben werden, wie dieses Modell aufgebaut sein dürfte. Alle Informationen zu Erweiterung einer Java-Quelltextdatei wären an einer Stelle zu finden. Außerdem müsste ein Benutzer des Metagenerators, anstatt nur einzelne Stellen in seiner Referenzimplementation zu markieren, eine zusätzliche XML Datei nach den Regeln der zugehörigen XSD Datei aufstellen.

Als weiterer Ansatz könnten Stellen im Quelltext, die mit Informationen angereichert werden sollen, zeilenweise mit Kommentaren versehen werden. Die genaue textuelle Form der Kommentare ist hier frei wählbar. Es wäre sogar denkbar, hier eine eigenständige DSL zu verwenden. Der Informationsgehalt und dessen semantische Bedeutung müssten nach oder beim Parsen des Java-Quelltextes in einem zusätzlichen Schritt analysiert werden. Kommentare könnten an jeder Stelle des Codes, also auch innerhalb von Methoden, theoretisch sogar innerhalb von einzelnen Ausdrücken, verwendet werden. Eine Einschränkung, welcher Kommentar an welche Stelle des Codes gesetzt werden darf, kann jedoch nicht ohne zusätzliche Hilfsmittel gemacht werden.

Eine dritte Variante wäre es, den Quelltext mit Java-Annotationen zu versehen. Abhängig von der Information die eine solche Annotation anbringen soll, könnte eine semantisch passende Benennung gewählt werden. Durch zusätzliche Target-Annotationen an den Annotations-Definitionen könnte noch genauer bestimmt werden, an welcher Stelle im Code welche Annotation angebracht werden darf. Mit Hilfe von Annotations-Parametern könnten noch weitere Informationen übermittelt werden. Da es nicht möglich, ist Quelltext innerhalb von Methoden zu annotieren, könnten dort auf diesem Weg keine zusätzlichen Informationen angegeben werden.

3.2.1.2 Parsen des Java-Codes

Der mit Informationen angereicherte Java-Code muss eingelesen werden. Ein Ansatz hierfür wäre es, einen Java-Parser von Grund auf neu zu implementieren oder auf bereits vorhandene Bibliotheken zurückzugreifen. Eine Neu-Implementation würde volle Kontrolle über das Modell, in welches der Java-Code eingelesen wird, liefern. Wird zur Informationsanreicherung der XML Ansatz verfolgt, könnte die Zusammenführung der Daten in ein gemeinsames Modell dann bereits beim Parsen durchgeführt werden. Die im Grundlagen-Kapitel beschriebenen Schritte zum Verarbeiten und Analysieren von Quelltext müssten implementiert werden. Da Java eine syntaktisch sehr umfangreiche Sprache ist, ist der Aufwand hier sehr groß. Bei der Verwendung einer externen Bibliothek zum Einlesen des Codes hat dies meist den Vorteil, dass diese Schritte hinter einer API abstrahiert sind und man ohne Zusatzaufwand Hilfsmittel wie ein implementiertes Visitor Pattern für den AST zur Verfügung hat. Je nach verwendeter Bibliothek kann man auch damit rechnen, dass der Parser weitestgehend fehlerfrei funktioniert und regelmäßig aktualisiert wird.

3.2.2 Abstrakte Darstellung von Java-Code als Modell

Wurden der Java-Quelltext und die angebrachten Informationen eingelesen und zusammengeführt, muss eine Möglichkeit gefunden werden, konkrete Ausprägungen der Variabilitäten definieren zu können. Das Modell, welches diese Daten abbildet, ist das Metamodell, auf dessen Basis der Metagenerator Quelltext erzeugt.

3.2.2.1 Anforderungen an das Metamodell

An das Metamodell werden mehrere Anforderungen gestellt. Grundsätzlich muss es möglich sein, die, zur späteren Generierung notwendigen Strukturinformationen aus dem Java-Quelltext, zu halten. Außerdem müssen Instanzen dieses Metamodells gebildet werden können, die in ihrem Aufbau die nicht variablen Teile des Ursprungs-Codes widerspiegeln und über eine Schnittstelle mit Informationen zu den variablen Teilen befüllt werden können.

Da beschreibbar sein muss, welche Variabilitäten im Quelltext existieren, reicht die Verwendung des beim Parsen entstandenen AST als solcher nicht aus. Es wäre jedoch denkbar, das Metamodell des AST zu übernehmen und diesen mit weiteren Informationen zu den Nodes zu versehen. Je nach verwendeter Bibliothek könnte es hier möglich sein, diese Erweiterung durch den Aufbau einer Vererbungshierarchie zu den vorhandenen Klassen der Knotenpunkte vorzunehmen. Wird auf den von einer Bibliothek gelieferten AST zurückgegriffen, könnte es sein, dass dieser viel mehr Informationen enthält, als für den konkreten Erzeugungsprozess notwendig wären und somit zusätzliche Komplexität einführt.

Eine weitere Möglichkeit wäre es, ein eigenes Metamodell zu definieren. Hierdurch können Struktur und Inhalt des Metamodells genau an die Aufgabe des Metagenerators angepasst werden. Mit Einführung eines unabhängigen Metamodells würde die Modularität der Anwendung zusätzlich gesteigert. Dieser Ansatz würde zudem ein iteratives Vorgehen erlauben. Parser, Metamodell und Generator könnten schrittweise gemeinsam um zusätzliche Funktionalität erweitert werden.

3.2.2.2 Schnittstellen zur Instanziierung des Metamodells

Um konkrete Ausprägungen des Metamodells bilden zu können, wird eine Schnittstelle benötigt. Im Zusammenhang mit modellgetriebener Softwareentwicklung ist dies normalerweise eine DSL. Es ist jedoch auch denkbar, auf das Metamodell mit einer Command-Query-API zuzugreifen [7, S. 343ff.]. Das bedeutet, dass die einzelnen Felder einer Instanz des Metamodells mit Methoden, wie zum Beispiel klassischen Settern, gesetzt werden.

Greift man auf eine domänenspezifische Sprache zurück, könnten hier gleich mehrere Ansätze verfolgt werden. Bei Verwendung einer externen DSL müsste diese entweder selbst definiert und implementiert werden. Das würde bedeutet, dass auch ein Parser für diese geschaffen werden müsste. Alternativ kann auf eine bereits existierende DSL zurückgegriffen werden. In diesem Fall muss dafür Sorge getragen werden, dass das Metamodell und die DSL miteinander kompatibel sind. Zusätzlich hätte die Verwendung einer bereits existierenden DSL den Vorteil, dass Hilfsmittel zum Parsen und Weiterverarbeiten der DSL normalerweise schon gegeben sind und somit der Implementationsaufwand reduziert wird. Stattdessen könnte auf eine Language Workbench zurückgegriffen werden. So stünde eine Entwicklungsumgebung zum Schreiben von DSL-Code zur Verfügung. Dies würde aber die Bindung an ein Tool bedeuten.

Da der Metagenerator im für diese Thesis relevanten Fall, unabhängig von seinem sonstigen Konzept, Java-Quelltext als Referenzimplementation einlesen wird und nach erfolgter Modellbeschreibung durch die DSL wieder Java-Quelltext generiert, könnte die Verwendung einer internen DSL vorteilhaft sein. Ein Anwender der DSL könnte potenziell im Java-Sprachraum bleiben und müsste somit, zumindest bezüglich der verwendeten Syntax, weniger neue Grundlagen lernen.

3.2.3 Generierung von Java-Quelltext

Eine konkrete Ausprägung des Metamodells soll eingelesen und aus diesem Java-Quelltext generiert werden.

3.2.3.1 Verwendung vorhandener Bibliotheken zur Java-Codegenerierung

Wie auch beim ersten Schritt des Metagenerator-Prozesses, dem Parsen, muss abermals entschieden werden, ob die Komponente, welche den Java-Quelltext in Form von Zeichenketten generiert, vollständig neu umgesetzt wird oder ob auf vorhandene Bibliotheken zurückgegriffen werden soll. Es gelten die gleichen Vorteile wie schon bezüglich des Parsers erörtert. Verwendet man bereits existierende Komponenten, spart man sich die Implementationsarbeit und hat meist den Vorteil eines zuverlässigeren und aktuell gehaltenen Moduls.

Sowohl bei Verwendung einer externen Bibliothek als auch bei einer Neuimplementation muss entschieden werden, auf welche Weise der Code zu generieren ist. Wie bereits im Grundlagen-Kapitel erklärt, gibt es hier im Allgemeinen zwei Ansätze. Der Quelltext könnte aus Vorlagen heraus erzeugt werden. Dies ist vor allem dann von Vorteil, wenn große Teile des erzeugten Codes gleich sind und nur Schlüsselelemente, wie

zum Beispiel Identifier, ausgetauscht werden sollen [7, S. 125]. Werden die ersetzten Teile des erzeugten Quelltextes zu komplex oder zu umfangreich, so könnte das Generat unüberschaubar werden. Ab einer gewissen geforderten Variabilität des Generats, könnte templatebasierte Generierung gar nicht mehr anwendbar sein. In solchen Fällen kann die von Fowler sogenannte Transformer Generation in Betracht gezogen werden [7, S. 125]. Sie gibt die volle Kontrolle über den erzeugten Code. Da der Ziel-Quelltext in diesem Fall mit Java-Statements definiert wird, zum Beispiel unter Verwendung eines Builders, ist er im Vergleich zur vorlagenbasierten Generierung möglicherweise schwieriger verständlich. Bei einer vorlagenbasierten Generierung liegt der erzeugte Quelltext quasi in seiner endgültigen Form bereits vor und könnte somit leichter erfasst werden.

3.2.3.2 DSL oder Generator als alternative Erzeugnisse des Metagenerators

Eine weiteres Problem stellt die Wahl dar, welches Artefakt der Metagenerator erzeugen soll. Wie zu Beginn dieses Kapitels beschrieben, besteht vor allem in der Phase der Implementation des Generators Einsparpotenzial durch Automatisierung. Händisch müssen hier normalerweise zwei Bestandteile umgesetzt werden: Die DSL zur Beschreibung des Metamodells und der Generator, welcher aus einer Instanz des Metamodells konkreten Java-Quelltext erzeugt.

Das automatische Erzeugen der DSL könnte nur eingespart werden, wenn das beschriebene Metamodell immer gleich wäre und die DSL von vornherein die gleiche abstrakte und konkrete Syntax hat. In diesem Fall wäre die DSL zwar umfangreicher, da sie grundsätzlich alle möglichen Variabilitäten des Metamodells beschreiben können muss, müsste aber von einem Anwender nur einmal erlernt werden. Da wir davon ausgehen, dass wir Variabilitäten in der Konfiguration konkreter Java-Klassen mit der DSL ausdrücken wollen, sollte diese keine Möglichkeit geben, zum Beispiel den Datentyp eines Feldes zu ändern, wenn dies nicht auch durch die mit Informationen angereicherte Referenzimplementation vorgesehen ist.

Theoretisch könnte man auch den eigentlichen Quelltextgenerator erzeugen. Abhängig von der konfigurierten Ausprägung des Metamodells, könnte dieser dann nur Java-Quelltext mit der dadurch bestimmten Struktur erzeugen. Dem gegenüber steht eine Variante des Generators, der grundsätzlich jede Art von gültigem Java-Quelltext erzeugen kann. Auf Basis der Informationen in der Instanz des Metamodells könnte dieser dann nur gewünschten Quelltext generieren. Damit ein solcher Generator funktioniert, müssen die von ihm eingelesenen Daten stets gleich aufgebaut sein, das heißt aus einem gemeinsam Metamodell erzeugt worden sein. Von außen betrachtet würde dies für den Anwender des Metagenerators keinen Unterschied machen. Ein Entwickler würde immer noch sein Metamodell mit der DSL beschreiben und dann würde aus dieser Instanz Quelltext generiert werden.

4 Konzept

Ausgehend von der im vorherigen Kapitel durchgeführten Analyse, ist es jetzt möglich, ein umfassendes Konzept für die vorliegende Problemstellung zu beschreiben.

In den folgenden Abschnitten wird zunächst auf den allgemeinen Aufbau des Metagenerators eingegangen, um nachfolgend dessen Funktionsweise genauer zu erläutern. Die für das Konzept getroffenen Entscheidungen beruhen auf der bereits beschriebenen Analyse und werden auf Basis dieser Untersuchung begründet. Mehrere der Ansätze sind mit Blick auf das Prinzip der Modularisierung entstanden. Der Hintergedanke hieran ist, auch wenn ein bestimmter Weg für dieses Konzept gewählt wurde, es leicht möglich sein soll, an verschiedenen Stellen des Metagenerators Komponenten zu verändern oder ganz auszutauschen und somit für zukünftige Arbeiten an dem umgesetzten Prototypen so viele Anschlussmöglichkeiten wie möglich bereitzustellen.

Der im vorangegangenen Kapitel durchgeführte Aufwandsvergleich zeigt, dass, wenn die Entwicklung eines Codegenerators durch Automatisierung wirtschaftlicher gemacht werden soll, diese nicht in der Analyse, sondern im eigentlichen Implementationsschritt für den Generator eingeführt werden könnte. Hier setzt auch das Konzept dieses Generators an. Durch Verwendung eines einheitlichen Metamodells kann die Entwicklung der beschreibenden DSL automatisiert werden. So kann anstelle eines spezialisierten Generators ein allgemeiner Generator treten.

4.1 Allgemeine Struktur

Wie auf Abbildung 4.1 zu erkennen ist, besteht der Java-Metagenerator im Grunde aus zwei Teilen. Der DSL-Generator bietet dem Anwender die Möglichkeit Java-Quelltext durch Annotationen mit Informationen anzureichern. Der annotierte Java-Quelltext wird dann geparsed und daraus ein Annotations-Modell instanziiert. Dieses beinhaltet alle nötigen Informationen, um eine DSL zur Beschreibung des CodeUnit-Modells zu generieren. Das konkrete Endprodukt des DSL-Generators sind mehrere Builder-Klassen, die dann als interne DSL verwendet werden können. Der zweite Teil, der Java-Quelltextgenerator, transformiert eine Instanz des CodeUnit-Modells in ein allgemeines Java-Modell. Aus einer Instanz des Java-Modells wird mit einem allgemeinen

Java-Generator konkreter Java-Quelltext generiert.

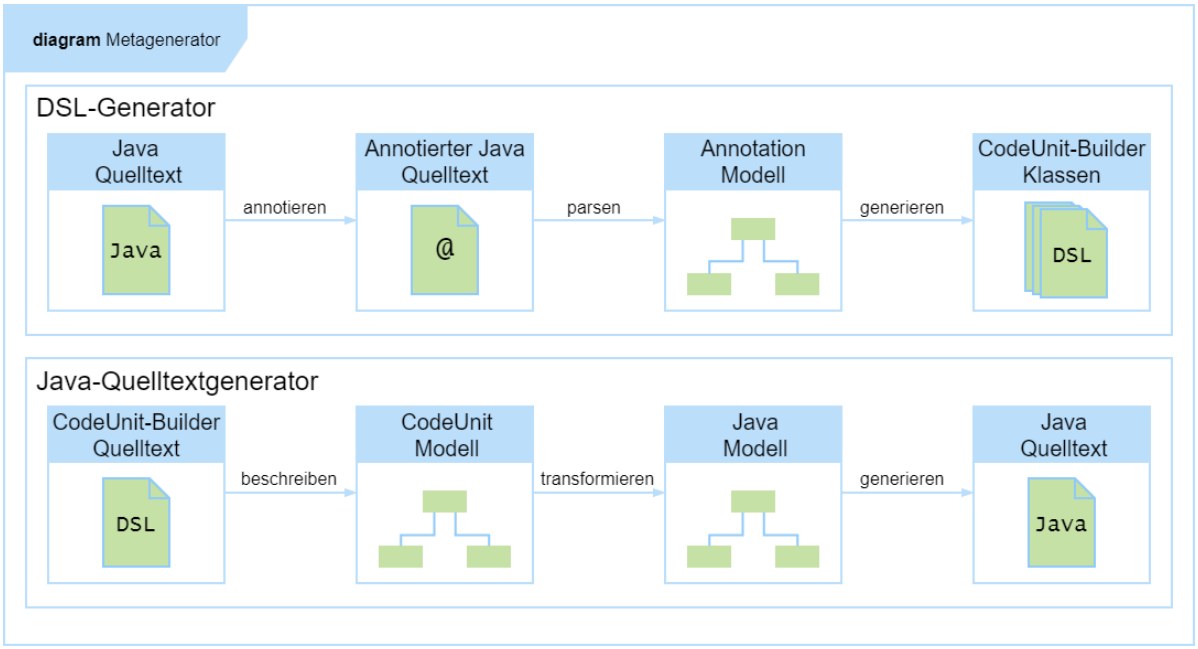

Abbildung 4.1: Darstellung der Prozessstruktur des Java-Metagenerators.

4.2 Funktionsweise des Metagenerators

Das Konzept der einzelnen Schritte und der hierfür verwendeten Ansätze wird im folgenden, orientiert an der obigen Prozessstruktur, detailliert erläutert.

4.2.1 Von Java-Quelltext zum Annotations-Modell

Ausgehend von einer Referenzimplementation muss diese, wie bereits in der Analyse beschrieben, mit Informationen angereichert werden, um Variabilitäten zur Generierung einer DSL ausdrücken zu können.

Das Konzept sieht vor, dass Java-Elemente entweder als statisches oder als variables Element markiert werden können. Da nur für variable Elemente ein Builder erzeugt wird, müssen statische Elemente stets innerhalb eines anderen variablen Elements stehen. Das bedeutet beispielsweise, dass eine Klasse als variabel gekennzeichnet wird und dadurch einzelne Felder innerhalb der Klasse als statisch oder variabel markiert werden können. Elemente, die weder als statisch noch als variabel gekennzeichnet wurden und nicht implizit zu einem übergeordneten Element gehören, werden beim Parsen nicht beachtet.

Ein statisches Java-Element soll hierbei als vordefiniert verstanden werden. Ein Beispiel hierfür wäre ein ID Feld einer Klasse.

Als Alternative zur expliziten Markierung einzelner Elemente als statisch, wäre es stattdessen denkbar gewesen, alle nicht markierten Elemente als statisch zu interpretieren. Dies hätte zur Folge gehabt, dass es nicht möglich gewesen wäre, Elemente der Referenzimplementation zu ignorieren. Damit wären Fälle, in denen zwei gleichartige Elemente unter einer CodeUnit mit einer bestimmten Variabilität zusammengefasst werden sollen, nicht ohne weiteres umsetzbar. Es sei denn, es würde eine weitere Markierung eingeführt, welche dafür sorgt, das das Element explizit ignoriert wird. Hierunter könnte die Lesbarkeit des markierten Quelltextes leiden. Zudem hätte für die Markierung ein zusätzliches Schlüsselwort eingeführt werden müssen.

4.2.1.1 Annotationen als Mittel zur Informationsanreicherung

Im letzten Kapitel wurden drei Ansätze aufgezeigt, um Java-Quelltext mit zusätzlichen Informationen zu hinterlegen. Diese sind die Verwendung einer externen Modelldatei mit Referenzen zum Java-Quelltext, Kommentierung von Quelltextzeilen beziehungsweise -abschnitten mit den zusätzlichen Daten in einer speziellen Syntax oder Markierung von Codeelementen mit Annotationen.

Für dieses Konzept sind Annotationen das Mittel der Wahl. Sie kombinieren einige Vorteile der beiden anderen Ansätze. Da ein Anwender des Metagenerators den Schritt der Informationsanreicherung manuell durchführen muss, bietet ihm die Einschränkungsmöglichkeit der Anwendbarkeit von Annotationen auf bestimmte Elemente des Codes eine Hilfestellung und reduziert mögliche Fehler. Die Annotationen sind hier eine verteilte interne DSL und somit stellt jede Annotationen ein Schlüsselwort dieser internen DSL dar. Im Gegensatz zur Verwendung von Kommentaren hat ein Entwickler also rudimentäre IDE Unterstützung.

Der Vorteil einer externen XML-Modelldatei mit entsprechender XSD ist, dass die Struktur klar definiert und eingeschränkt werden kann. Diese Eigenschaft kommt zwar nur beim Parsen der XML-Daten zum tragen, bietet aber keinerlei Hilfe bei der Definition der Variabilitäten durch einen Entwickler. Zudem müsste hierfür ein System zur Referenzierung einzelner Codezeilen eingeführt werden. Durch Annotationen ist die Zugehörigkeit zu einem bestimmten Codeelement eindeutig.

Sowohl bei der Verwendung von Kommentaren, als auch von externen Modell-Dateien müsste ein Entwickler unter Umständen ein komplett neuartiges Konzept lernen. Für Kommentare müsste eine extra Syntax verwendet werden und bei externen Modell-Dateien zum Beispiel XML. Annotationen erlauben es dem Entwickler, im Java-Sprachraum zu bleiben und reduzieren somit den zusätzlichen Lernaufwand.

4.2.1.2 Parsen des annotierten Codes

Für dieses Konzept wird der annotierte Java-Quelltext mithilfe einer externen Bibliothek geparsed. Diese Entscheidung wurde getroffen, da ein möglichst großer Teil des Java-Sprachschatzes vom Metagenerator abgedeckt sein sollte. Während der Entwicklung hätte eine eigene Implementation des Parsers zur Folge gehabt, dass dieser sukzessiv mit dem Metagenerator hätte erweitert werden müssen, welches ein signifikant höherer Arbeitsaufwand gewesen wäre.

Die konkrete Wahl ist auf die Bibliothek JavaParser in Kombination mit dessen Erweiterung JavaSymbolSolver in der Version 0.6.3 gefallen. Beides ist unter http://javaparser.org/ zu finden und dort mit Anleitungen und Codebeispielen versehen. Das aktive GitHub-Repository mit mehreren tausend Commits weckt den Anschein, dass JavaParser ständig aktualisiert und weiterentwickelt wird. Außerdem kann der von der Bibliothek bereitgestellte AST leicht mithilfe des Visitor Patterns weiterverarbeitet werden. JavaSymbolSolver kommt zum Einsatz, da an einigen Stellen im Proof-of-Concept voll qualifizierte Datentypen benötigt werden, welche aus dem normalen AST nicht ohne Zusatzaufwand gewonnen werden konnten. Genauere Informationen zur Verwendung von JavaParser und JavaSymbolSolver werden im Kapitel zur tatsächlichen Umsetzung dieses Konzepts bereitgestellt.

4.2.1.3 Zweck des Annotations-Modells

Mit dem Visitor Pattern wird der von JavaParser bereitgestellte AST untersucht. Je nach Annotation und annotiertem Java-Ausdruck wird aus diesen gewonnenen Informationen ein oder mehrere Annotations-Modelle instanziiert. Eine Instanz des Annotations-Modells enthält dabei immer alle Informationen zu den statischen und variablen Bestandteilen, die später mit dem CodeUnit-Modell abgebildet werden, beziehungsweise mit den generierten CodeUnit-Buildern beschrieben werden können.

Das Annotations-Modell wurde als Zwischenschritt von AST zur konkreten DSL konzeptioniert. Da dies losgelöst vom Parser, im Sinne der Modularisierung ermöglicht, den aktuellen DSL Generator durch eine andere Komponente zu ersetzen. Durch das Annotations-Modell liegen hierfür bereits alle notwendigen Informationen abstrahiert vor und müssen nicht erneut aus dem AST ausgelesen werden.

4.2.2 Das CodeUnit-Modell

Ein wichtiger Teil dieses Konzepts ist das CodeUnit-Modell. Instanzen dieses Modells werden von den generierten CodeUnit-Buildern beschrieben und somit ist es das Metamodell für die vom Metagenerator erzeugte DSL. Mit einer spezifischen CodeUnit-Annotation und unter Angabe einer Benennung kann ein Java-Element so markiert werden, dass hieraus ein Annotations-Modell geparsed wird. Aus den so entstandenen Annotations-Modellen wird in einem nachgelagerten Schritt die interne DSL erzeugt.

4.2.2.1 Aufbau des Modells

Das CodeUnit-Modell setzt sich aus einer Struktur mehrerer CodeUnits zusammen, diese sind dabei hierarchisch geordnet. An jede CodeUnit können SubCodeUnits angehängt werden, aber es gibt stets genau eine CodeUnit, welche das Wurzelelement des darunterliegenden Baumes ist. Der Aufbau jedes CodeUnit-Modells ist gleich, einzelne CodeUnits unterscheiden sich nur in ihren konkreten Parametern.

Diese Struktur wurde gewählt, da sich Quelltext in einer solchen hierarchischen Folge gut ausdrücken lässt. Bei Java gibt es beispielsweise Klassen, welche Felder und Methoden haben. Dies wird im CodeUnit-Modell als eine CodeUnit für die Klasse mit SubCodeUnits für Felder und Methoden abgebildet. Die CodeUnit für eine Methode könnte wiederum SubCodeUnits für Parameter und den Methodenkörper haben. Dieser beschriebene Zusammenhang ist beispielhaft in Abbildung 4.2 dargestellt.

4.2.2.2 Spezialisierung durch ein Typ-Feld

Um später eine konkrete Instanz des CodeUnit-Modells zu einer Java-Modell-Instanz transformieren zu können, müssen wichtige semantische Informationen vorliegen. Eine hiervon ist, welche Art von Java-Element mit dieser CodeUnit abgebildet wird. Jede CodeUnit hat einen CodeUnit-Typen, an welchem der Transformator später erkennt, auf welche Weise die spezifische CodeUnit in das Java-Modell transformiert werden muss.

Für das Konzept wurde hierfür kein objektorientierter Ansatz gewählt. Bei diesem würde für jeden CodeUnit-Typ eine eigene CodeUnit-Klasse implementiert werden. Das Argument gegen spezielle CodeUnit-Klassen ist, dass die CodeUnit-Klasse hier ein Datenmodell darstellt, welches kein eigenes Verhalten hat. Somit wäre jede von CodeUnit abgeleitete Klasse durch ihre Benennung nur eine semantische Erweiterung, aber ansonsten funktionslos. Zudem reduziert der gewählte Ansatz den Aufwand bei der Erweiterung der CodeUnit-Struktur um neue CodeUnit-Typen.

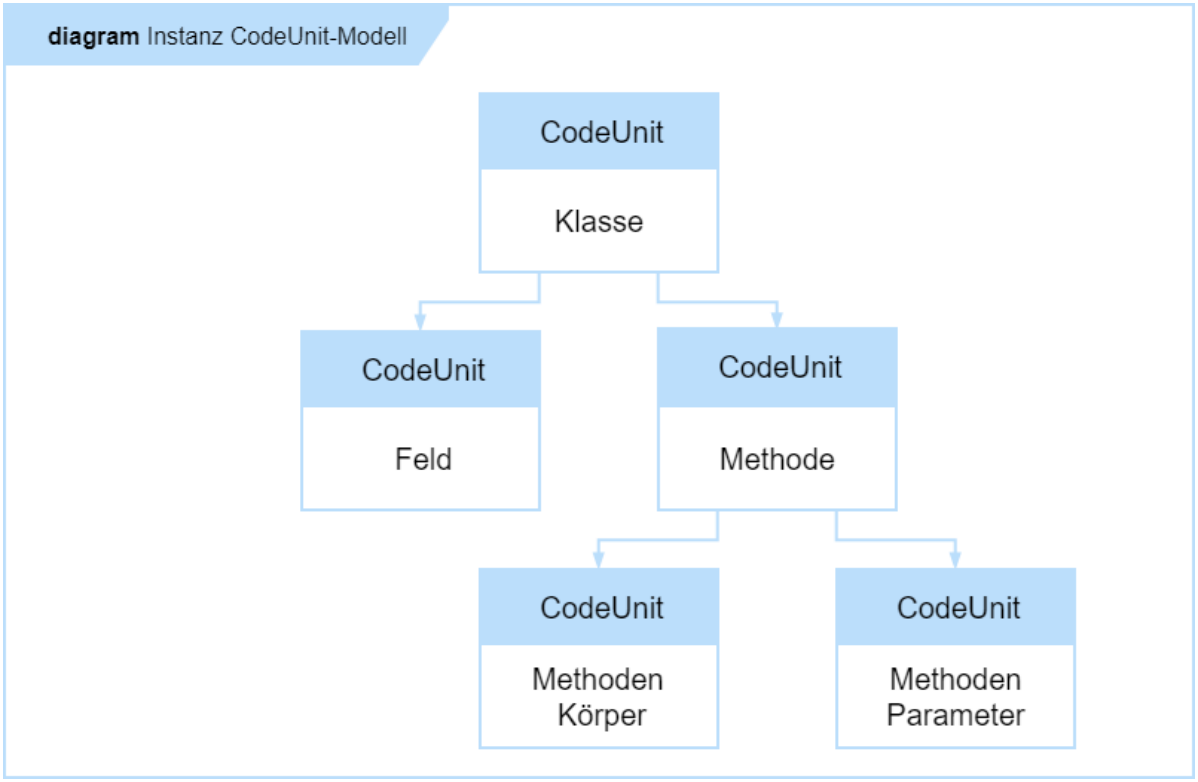

Abbildung 4.2: Beispielhafte Darstellung der Struktur eines CodeUnit-Modells.

Als Ausnahme könnte man CodeUnits, welche mit Buildern beschreibbar sind, zählen. Da mit einem Builder für eine Klassen-CodeUnit unter Umständen eine Feld-CodeUnit als SubCodeUnit angehängt wird, hätte man hier durch die Einführung von eigenen Datentypen die Möglichkeit einzuschränken, welche Typen von CodeUnits als SubCodeUnits verwendbar sind. Diese Klassen würden dann nur für diese Eigenschaft existieren keine weiteren Funktionen haben.

4.2.2.3 Parametrisierung durch generische Datenstruktur

Wie beschrieben wird jeder CodeUnit-Typ in seiner Struktur gleich dargestellt. Konfigurationsdaten zu einer CodeUnit werden in einer allgemein gehaltenen Datenstruktur gespeichert. Dies ermöglicht es, jede Art von Datum in einer CodeUnit zu hinterlegen. Mögliche Daten wären hier zum Beispiel der Identifier einer Klasse oder der Java-Datentyp eines Feldes.

Durch diesen Ansatz in Kombination mit dem Typ der CodeUnit, kann das CodeUnit-Modell sehr einfach erweitert oder abgeändert werden, ohne in dessen zugrunde liegende Struktur eingreifen zu müssen. Änderungen an den Schnittstellen für Transformator und DSLs werden vermieden.

Beispiele für solche Parameter wären der Identifier-Parameter, welchem als Datum eine Zeichenkette zugeordnet ist und der Modifier-Parameter, zu welchem ein Array aus Java-Modifier Schlüsselwörtern angegeben wird. Ein solches Datum kann auch zur Markierung einer Eigenschaft der zugehörigen CodeUnit verwendet werden, in dem, unter einer geeigneten Parameterbenennung, zum Beispiel ein Boolescher Wert hinterlegt wird.

4.2.3 Generierung von Buildern als interne DSL aus dem Annotations-Modell

Grundsätzlich wäre, wie in der Analyse beschrieben, für dieses Konzept sowohl eine externe DSL als auch eine interne DSL denkbar gewesen. Der für eine externe DSL notwendige Mehraufwand beziehungsweise die Toolbindung bei Verwendung einer Language Workbench waren hier jedoch ausschlaggebend für die Entscheidung zur internen DSL. Außerdem gilt, genau wie bei dem Argument für Annotationen, dass bei der Verwendung einer internen DSL auf Java-Basis die Anwendung potentiell einfacher ausfällt. Durch das Annotations-Modell wäre es jedoch denkbar aus diesem eine externe DSL zu generieren.

Die konkrete Umsetzung einer internen DSL bietet mehrere Ansätze. Zum einen könnte man hier auf Sequenzen aus Funktionen zurückgreifen. Damit hätte der Anwender aber keine Kontrolle darüber, in welcher Reihenfolge der Entwickler die verschiedenen Statements aufruft. Unter Verwendung eines Kontextobjekts wäre es lediglich möglich, bei einer unerlaubten Sequenz Fehler auszugeben oder Exceptions zu werfen [7, S. 351ff.]. Alternativ hierzu könnte ein Builder Pattern mit Method Chaining zum Einsatz kommen. Dies würde große Kontrolle darüber ermöglichen, welche Statements in welcher Reihenfolge ausführbar sind [7, S. 343ff.]. Eine weitere Möglichkeit zur Implementation einer internen DSL wäre die Verwendung von geschachtelten Methoden. Hierdurch kann ebenso stark kontrolliert werden, in welcher Reihenfolge Methoden einer Sequenz anwendbar sind. Da die verschachtelten Funktionen von innen nach außen evaluiert werden, muss man hierauf besondere Rücksicht nehmen [7, S. 357ff.].

Für dieses Konzept wurde, wie schon mehrfach erwähnt, auf ein Builder Pattern mit Method Chaining zurückgegriffen. Hauptargument hierfür war die mögliche Kontrolle über die Reihenfolge der ausführbaren Statements. Da sowohl verschachtelte Methoden als auch Builder ähnliche Vorteile haben, wurde zwischen diesen auf Basis persönlicher Präferenz entschieden. Wird an eine CodeUnit eine SubCodeUnit angehängt, ist diese ein Parameter der aufgerufenen Builder Methode und somit handelt es sich hier im kleinen Rahmen um eine Verschachtelung.

4.2.3.1 Erzeugte Builder als Schlüsselwörter der internen DSL

Aus jedem Annotations-Modell wird ein konkreter Builder erzeugt. Jedes dieser Modelle ist ursprünglich durch eine spezifische CodeUnit-Annotationen entstanden und hat hierdurch auch eine Benennung erhalten. Die Idee dahinter ist, dass es möglich sein soll, bestimmte Attribute eines Java-Elements nicht variabel zu machen und somit eine gewisse Form der Abstraktion zu bieten. Mit einer aussagekräftigen Benennung der beschreibbaren CodeUnit, kann hieraus auch ein eindeutig identifizierbarer Builder generiert werden.

Man nehme beispielsweise den Fall, dass ein Java-Feld als CodeUnit mit einem Builder beschreibbar sein soll. Unter Verwendung entsprechender Annotationen könnte diese CodeUnit immer als öffentlich mit variablem Datentyp markiert werden. Hinzu käme eine sprechende Benennung wie zum Beispiel PublicVariable. Hieraus könnte dann ein PublicVariableBuilder generiert werden, welcher immer ein öffentliches Feld erzeugt und eine Funktion bietet, um dessen Datentyp zu setzen.

Werden auf diese Weise mehrere Builder zur Beschreibung unterschiedlich konfigurierter CodeUnits mit unterschiedlichem CodeUnit-Typ erzeugt, so entwickelt sich dadurch ein DSL-Sprachraum mit konkreter Semantik. Die einzelnen Builder könnte man in diesem Sprachraum als Schlüsselwörter der internen DSL betrachten.

4.2.3.2 Komposition der Builder aus benötigten Builder-Methoden

Abhängig davon, welcher CodeUnit-Typ von einem konkreten Builder beschrieben werden kann und welche Variabilitäten in Form von Annotationen in der Referenzimplementation definiert wurden, hat ein CodeUnit-Builder unterschiedliche Methoden.

Ein Builder zur Beschreibung des im letzten Abschnitt gegebenen Beispiels eines öffentlichen Feldes mit variablem Datentyp, benötigt beispielsweise keine Methode zum Setzen eines Parameters für den Java-Modifier. Jedoch benötigt jeder Builder eine Methode zum Starten der Methodenkette und eine um diese abzuschließen.

Der Ansatz hierfür ist komponentenbasiert. Die unterschiedlichen Methoden, die ein konkreter Builder haben kann, werden diesem komponentenweise während der Erzeugung hinzugefügt. Dieses Konzept wurde gewählt, da die Alternative hierfür wäre, dass jeder Builder genau die gleichen Methoden implementiert. Hierbei müsste, um das Setzen bestimmter Werte zu verbieten, spezielles Error Handling betrieben werden. Der komponentenbasierte Ansatz schränkt die konkrete Syntax der internen DSL weiter ein und bietet somit dem Anwender eine Hilfestellung bei der Formulierung von Ausdrücken.

4.2.3.3 Übertragung vorgegebener Informationen in einen Builder

Einige in der Referenzimplementation vorliegende Daten müssen als solches in den am Ende der Kette generierten Java-Quelltext übertragen werden. Hierunter fallen alle unveränderlichen Parameter eines als CodeUnit annotierten Java-Elements, die nicht durch den Typen oder den Kontext der CodeUnit impliziert werden und jedes als statisch annotierte Java-Element.

Hierfür sieht dieses Konzept vor, bereits beim Parsen des annotierten Java-Quelltextes eine Ursprungs-CodeUnit zu jedem Annotations-Modell zu erzeugen. Diese CodeUnit dient dann dem aus dem Annotations-Modell generierten Builder als Ausgangspunkt bei der Beschreibung einer konkreten CodeUnit.

Ein alternativer Ansatz wäre gewesen, eine zusätzliche Datenstruktur zum Halten dieser Informationen zu verwenden. Im Aufbau würde dieses zusätzliche Modell jedoch sehr dem der CodeUnit ähneln, da gleichartige Daten hinterlegt werden würden. Daher kann auch direkt und ohne Mehraufwand das bereits bestehende CodeUnit-Modell verwendet werden.

4.2.3.4 Verwendung von Pattform-Code zur Generierung von vordefinierten CodeUnits und Fehlermanagement

Bei der Anwendung eines Builders kann es vorkommen, dass vordefinierte CodeUnits in das vom Builder beschriebene CodeUnit-Modell eingefügt werden müssen. Als konkretes Beispiel dienen hier Getter- und Setter-Methoden für ein Feld. Ein solches könnte mit einer CodeUnit und einer HasGetter-Annotation versehen werden. Dies würde semantisch bedeuten, dass immer, wenn diese Feld-CodeUnit mit einem Builder beschrieben und an eine Klassen-CodeUnit angehängt wird, ebenfalls eine auf dieses Feld zugreifende Getter-CodeUnit erzeugt werden soll.

Auf solche vordefinierten CodeUnits kann aus vielen Buildern zugegriffen werden. Sie unterscheiden sich teilweise gar nicht oder nur in einzelnen Parametern, wie zum Beispiel bei einem Getter dem Feldnamen. Deswegen kommt hier Pattform-Code zum Erzeugen dieser CodeUnits zum Einsatz. Dieser bildet folglich eine Abhängigkeit zur Verwendung von generierten Buildern.

Pattform-Code wird ebenfalls verwendet, um eine CodeUnit zu prüfen, bevor ihr Builder abgeschlossen wird.

4.2.3.5 Auflösung von Referenzen in vordefinierten CodeUnits als nachgelagerter Verarbeitungsschritt

Es ist möglich, dass ein als statisch annotiertes Feld den selben Datentyp hat, wie dessen zugehörige Klasse. Eine solche Referenz auf den eigenen Typen kommt zum Beispiel beim Instanz-Feld eines Singletons vor. Wird jetzt eine CodeUnit basierend auf einer solchen Situation beschrieben, sollte auch die entsprechende Referenz auf den angegebenen neuen Identifier der zugehörigen Klasse verweisen.

Der früheste Zeitpunkt, an dem bei der Verwendung eines Builders für eine Klassen-CodeUnit klar ist, welche SubCodeUnits diese hat, ist, wenn die Method Chain beendet wird. Vor der Rückgabe der endgültigen CodeUnit ist ein letzter Verarbeitungsschritt vorgesehen.

Jede SubCodeUnit der Klasse wird nach Referenzen auf den ursprünglichen Datentyp aus der Referenzimplementation durchsucht und an dessen Stelle tritt durch Textersetzung der neue, tatsächliche Identifier der erzeugten Klassen-CodeUnit. Die Information hierzu wird, wenn benötigt, bei der Erzeugung des Annotations-Modells in der Ursprungs-CodeUnit als Parameter hinterlegt.

Der aktuelle Proof-of-Concept ersetzt in diesem Nachbearbeitungsschritt nur Referenzen auf die Ursprungs-Klasse und erzeugt Standard-CodeUnits, wie zum Beispiel Getter und Setter. Erweiterungen dieser nachgelagerten Phase sind jedoch durchaus denkbar.

4.2.4 Erzeugung von Java-Code aus einem befüllten CodeUnit-Modell

In diesem Konzept wird aus dem instanziierten CodeUnit-Modell nicht direkt Java-Quelltext erzeugt. Stattdessen wird das CodeUnit-Modell in ein allgemeines Java-Modell überführt und dieses wird als Eingabe für ein allgemeinen Java-Codegenerator verwendet. Diese zusätzliche Schicht wurde in den Generator eingebaut, um, wie auch bereits die Entscheidung zum Annotations-Modell begründet wurde, die Modularität des gesamten Konzeptes zu erhöhen. Denn hiervon ausgehend ist es jetzt möglich, eine CodeUnit auch in andere Sprachen, beziehungsweise Modelle, zu transformieren. Liegt bereits ein Codegenerator für eine Sprache vor, so könnte dieser wiederverwendet werden, in dem man das CodeUnit-Modell zu einem für diesen Generator verständlichen Modell überführt.

4.2.4.1 Transformation des CodeUnit-Modells zum Java-Modell

Für die Umwandlung des CodeUnit-Modells zum Java-Modell wird, ausgehend vom Wurzelelement, eine CodeUnit anhand von Transformationsregeln iterativ umgewandelt. Die Wahl der Transformationsregel basiert auf dem CodeUnit-Typen.

Das verwendete Java-Modell stellt nur eine Teilmenge des Java-Metamodells dar und bildet nur Java-Sprachfeatures ab, welche mit den generierten DSLs in einer CodeUnit beschreibbar sind. Mit jeder Funktionserweiterung des Proof-of-Concept geht auch eine Erweiterung des verwendeten Java-Modells einher. Das Java-Modell soll keine abstrakte Darstellung von Code sein, sondern gezielt und so genau wie möglich die Struktur von Java-Quelltext abbilden.

4.2.4.2 Erzeugung von Quelldateien

Zur Generierung von Java-Quelltext wird bei diesem Konzept Transformer Generation verwendet. Wenn sich viele Teile des Generats wiederholen hat eine templatebasierte Quelltexterzeugung den Vorteil, dass der Aufwands reduziert wird. Bei der vorliegenden Problemstellung kann dieser Vorteil nicht ausgenutzt werden, denn die vom Metagenerator erzeugten Java-Quelltext Dateien können in ihrem Inhalt praktisch vollständig voneinander abweichen.

Zur konkreten Java-Quelltexterzeugung wurde im Proof-of-Concept auf eine externe Bibliothek zugegriffen. Zum Einsatz kommt hier JavaPoet in der Version 1.9.0 für welches unter [17] eine Informationsseite hinterlegt ist. JavaPoet wurde gewählt, da es eine auf Method Chaining basierende, gut verständliche API zum Erstellen spezifischer Java-Quelltextteile bereitstellt. Zudem wird JavaPoet regelmäßig aktualisiert.

5 Lösung: Spectrum (Proof-of-Concept)

Parallel zur Kozeptionierung und um schlussendlich das Konzept auch grundsätzlich zu validieren, wurde für diese Arbeit ein Proof-of-Concept umgesetzt. Der hierbei entstandene Metagenerator hat die Bezeichnung Spectrum.

Im folgenden Kapitel werden zuerst die im letzten Kapitel bereits erwähnten, in Spectrum eingesetzten externen Bibliotheken mit kurzen Anwendungsbeispielen vorgestellt. Gefolgt wird dies von einer allgemein Architekturübersicht über die Komponenten von Spectrum, welche dann in den darauf folgenden Abschnitten genau erläutert werden. Neben einer Beschreibung der Funktionalität der Komponenten wird auch deren Umsetzung näher betrachtet. Wichtige Stellen der Implementation und bei Bedarf die Anwendung einer Komponente werden anhand von Codebeispielen detaillierter dargestellt.

5.1 Eingesetzte externe Bibliotheken

Wie bereits im Konzept erläutert, wurde sowohl zum Einlesen der Referenzimplementation als auch zur späteren Java-Quelltexterzeugung auf externe Bibliotheken zugegriffen. Um diese Abhängigkeiten leicht handhaben zu können, wurde für die Implementation Apache Maven verwendet, welches unter https://maven.apache.org/ bezogen werden kann. Moderne integrierte Entwicklungsumgebungen, wie beispielsweise IntelliJ, unterstützen Maven ohne zusätzliche Installationsschritte.

5.1.1 JavaParser mit JavaSymbolSolver

Die Entwickler von JavaParser und JavaSymbolSolver beschreiben ihre Bibliothek als Möglichkeit zur Interaktion mit Java-Quelltext in Form einer Objekt-Repräsentation, womit hier ein AST gemeint ist. Zusätzlich soll mit Unterstützung des Visitor Patterns ein praktischer Mechanismus zur Traversierung des AST zur Verfügung gestellt werden.

Außerdem könnte mit dem JavaParser der AST manipuliert und wieder in eine Datei überführt werden [26, S. 1]. Dieses Feature wird in Spectrum jedoch nicht verwendet.

5.1.1.1 Anwendung

Listing 5.1: Beispielhafte Initialisierung und Verwendung des JavaParsers und des JavaSymbolSolvers

```
1  class ExampleParser {
2  public void handle(File file) {
3  //parse sourcecode file
4  CompilationUnit cu = JavaParser.parse(file);
5
6  //inject TypeSolver
7  TypeSolver ts = new ReflectionTypeSolver();
8  JavaSymbolSolver jss = new JavaSymbolSolver(ts);
9  jss.inject(cu);
10
11 //visit with VariableVisitor
12 new VariableVisitor().visit(cu, null);
13 }
14 }
15
16 private static class VariableVisitor extends VoidVisitorAdapter<Void> {
17 @Override
18 public void visit(VariableDeclarator declarator, Void arg) {
19 //resolve declarator and get qualified identifier
20 String qualifiedVariableName = declarator
21 .resolve()
22 .asReferenceType()
23 .getQualifiedName();
24
25 //do something with qualifiedVariableName;
26 }
27 }
```

Um mit JavaParser arbeiten zu können, wird zuerst eine vorhandenen Java-Quelltextdatei geparsed. Das Endprodukt des Parseprozesses heißt bei JavaParser CompilationUnit. Hierunter sind in einer hierarchischen Ordnung alle Knotenpunkte des AST gesammelt. Zur Traversierung der CompilationUnit können Visitor implementiert werden. Diese ermöglichen es, den Baum gezielt nach Knotenpunkten mit bestimmten Eigenschaften zu durchlaufen. Hierfür wird eine Klasse mit einer Visit-Methode implementiert, welche einen von JavaParser zur Verfügung gestellten VisitorAdapter erweitert. Der Typpa-

rameter des erweiterten VisitorAdapters ist in der Signatur der Visit-Methode wiederzufinden. Über den zweiten Parameter dieser Methode kann, wenn ein Typparameter angegeben wird, ein Kontextobjekt übergeben werden.

Um mit JavaSymbolSolver Symbole auflösen zu können, damit beispielsweise auf den vollständig qualifizierenden Namen einer Klasse zugegriffen werden kann, muss ein JavaSymbolSolver Objekt instanziiert werden. Dafür wird eine Instanz eines TypeSolvers benötigt. Diese werden in *JavaParser: Visited* genauer erklärt [26, S. 39ff.].

Der genaue Typ des TypeSolvers steuert, wo nach Typreferenzen gesucht wird. Ein JarTypeSolver sucht beispielsweise nur innerhalb einer bestimmten JAR-Datei. Ein ReflectionTypeSolver sucht, wie der Name schon sagt, nach Klassenreferenzen via Reflection. Das Listing 5.1 zeigt die in diesem Abschnitt erläuterten Features von JavaParser und JavaSymbolSolver in einer beispielhaften Anwendung.

5.1.1.2 Installation mit Maven

Listing 5.2: XML-Code zum Einbinden von JavaParser und JavaSymbolSolver als Maven-Dependency.

```
1 <dependency>
2 <groupId>com.github.javaparser</groupId>
3 <artifactId>java-symbol-solver-core</artifactId>
4 <version>0.6.3</version>
5 </dependency>
```

Bei der kombinierten Verwendung von JavaParser und JavaSymbolSolver, muss lediglich die Abhängigkeit zu JavaSymbolSolver in die pom.xml eingetragen werden. JavaSymbolSolver beinhaltet automatisch die aktuellste kompatible JavaParser Version. Der XML-Code zum Einbinden des JavaSymbolSolver als Maven-Dependency wird in Listing 5.2 dargestellt.

5.1.2 JavaPoet

JavaParser hätte zwar grundsätzlich die Möglichkeit zur Erzeugung von Java-Quelltext Dateien geboten, im Gegensatz zu JavaParser hat JavaPoet jedoch zusätzlich eine gut lesbare API zur Beschreibung des zu erzeugenden Quelltextes.

5.1.2.1 Anwendung

Listing 5.3: Java-Quelltext zur Erzeugung eines „Hallo, Welt!“-Beispiels, angelehnt an das „Hello, JavaPoet!“ Beispiel in der README.md von [17].

```
class HelloWorldGenerator {
    public String generateHelloWorld() {
        MethodSpec mainMethodSpec = MethodSpec.methodBuilder("main")
            .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
            .addParameter(String[].class, "args")
            .addStatement("$T.out.println($S)", System.class, "Hello,
   World!")
            .returns(void.class)
            .build();

        TypeSpec exampleTypeSpec = TypeSpec.classBuilder("Example")
            .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
            .addMethod(mainMethodSpec)
            .build();

        JavaFile javaFile = JavaFile
            .builder("example.package.name", exampleTypeSpec)
            .build();

        String javaCodeString = javaFile.toString();
    }
}

//above generateHelloWorld()-Method returns the following String
package example.package.name;

public final class Example {
public static void main(String[] args) {
System.out.println("Hello, World!");
}
}
```

Auf Basis von mehreren Buildern werden die verschiedenen Teile einer Java-Klasse zuerst modelliert, um dieses Modell dann später als Java-Quelltext auszudrücken. Die Ausnahme bilden hier Methodenkörper, diese werden bei JavaPoet in Form von Strings verwaltet.

Soll in einzelnen Ausdrücken ein bestimmtes Datum in einer Zeichenkette verwendet werden, müssen diese nicht mit einzelnen Operatoren konkateniert werden. Um die API lesbarer zu gestalten, wurde in JavaPoet eine an Javas String.format() angelehnte, Kurzschreibweise entwickelt. Hierdurch können in einer Zeichenkette Platzhalter verwendet werden, deren Inhalt zusätzlich als Parameter an die entsprechende Funktion übergeben wird. Dies ist vor allem hilfreich, wenn diese Parameter dynamisch sind. Hierunter fällt

zum Beispiel der Platzhalter $L für Literale, welche direkt und unverändert in den einbettenden String übernommen werden. Ein weiterer Platzhalter ist $T. Dieser kann verwendet werden, um eine Referenz auf einem bestimmten Datentypen zu hinterlegen. Es gibt noch einige weitere dieser Platzhalter. Diese werden gut zusammen mit weiteren Features, in der JavaPoet README.md des zugehörigen GitHub Repositories erklärt [17]. Listing 5.3 zeigt die Verwendung von JavaPoet anhand eines einfachen Beispiels.

5.1.2.2 Installation mit Maven

Listing 5.4: XML-Code zum Einbinden von JavaPoet als Maven-Dependency.

```
1 <dependency>
2 <groupId>com.squareup</groupId>
3 <artifactId>javapoet</artifactId>
4 <version>1.9.0</version>
5 </dependency>
```

Zur Einrichtung der Maven-Dependency von JavaPoet kann der in Listing 5.4 dargestellte XML-Code verwendet werden.

5.2 Architekturübersicht

Die allgemeine Architektur von Spectrum orientiert sich an der Funktionalität einzelner Komponenten. Insgesamt besteht Spectrum aus fünf Java-Packages: Amber, Cherry, Jade, Scarlet und Violet. Jede dieser Komponenten hat eine klare Aufgabe im Metagenerator-Gesamtprozess. Die Grafik 5.1 zur Architektur von Spectrum, welche sich an der Abbildung 4.1 aus dem Konzept-Kapitel orientiert, zeigt unter anderem die im Prozess entstehenden Artefakte und welche Komponenten jeweils mit diesen in Berührung kommen. Die genaue Aufgabe und Implementation der Bestandteile von Spectrum werden in den folgenden Abschnitten erläutert.

5.2.1 Amber

Die Bereitstellung der Annotationen zur Informationsanreicherung der Referenzimplementation und die Überführung des annotierten Quelltextes in das Annotations-Modell sind die Aufgaben von Amber. Hierfür ist Amber in vier Packages unterteilt. Diese sind jeweils sprechend nach ihrer Funktion benannt. Abbildung 5.2 stellt die Packages grafisch dar.

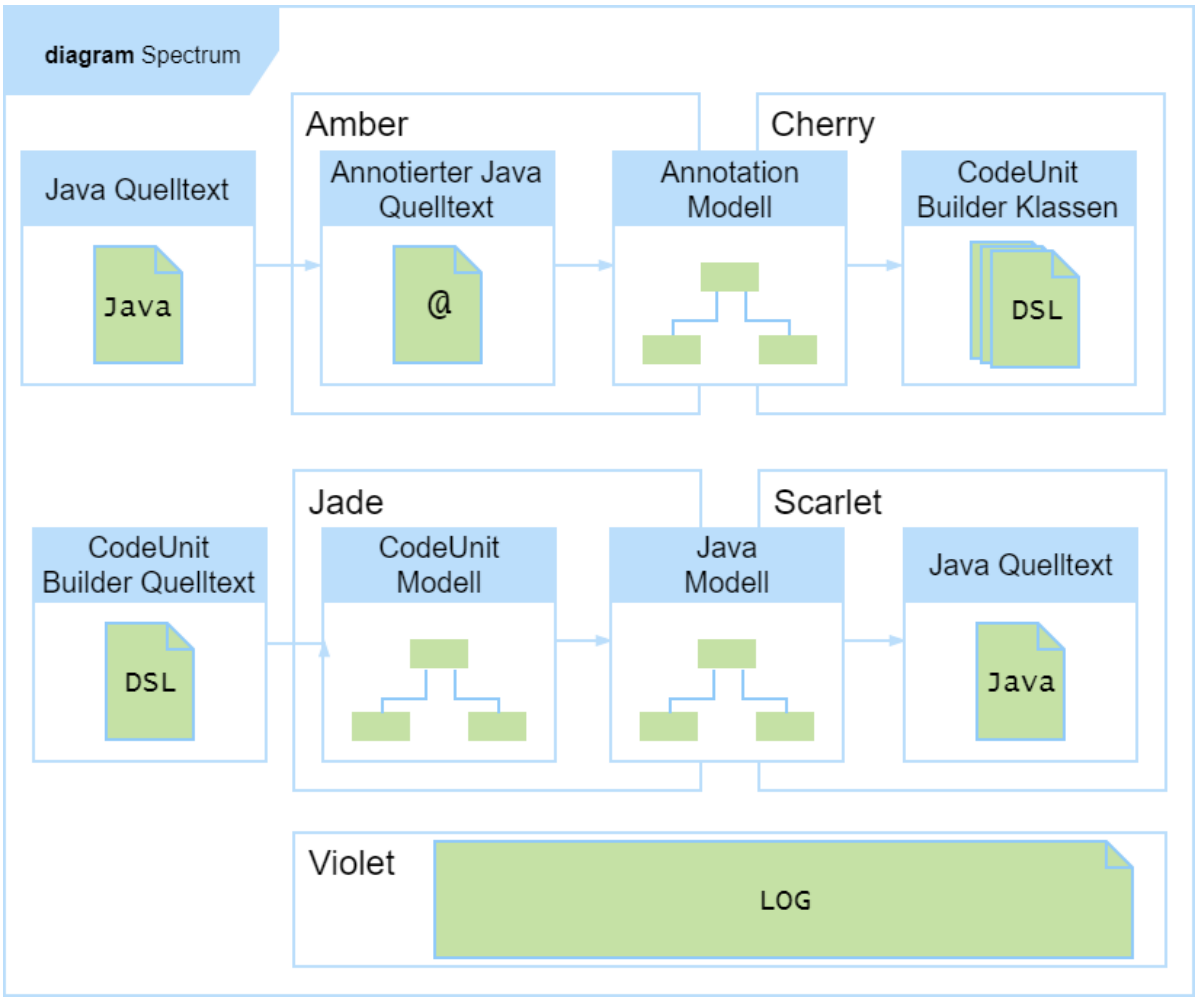

Abbildung 5.1: Darstellung der Komponenten von Spectrum.

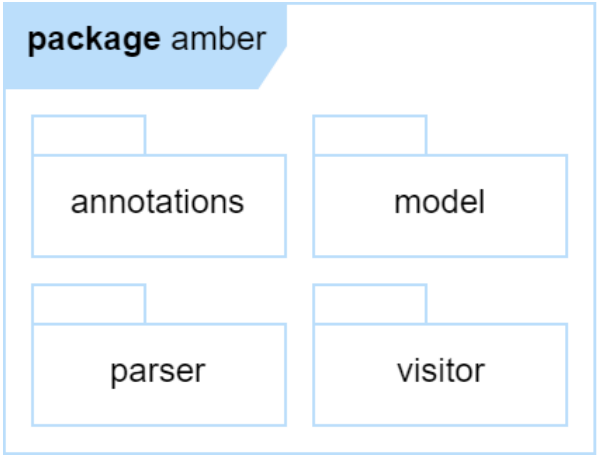

Abbildung 5.2: Darstellung der Unter-Packages im amber Package.

5.2.1.1 Implementierte Annotationen

Die Annotationen bilden die Schlüsselwörter der verteilten DSL zur Informationsanreicherung einer Referenzimplementation. In der aktuellen Version des Proof-of-Concept können Klassen, Felder, Methoden und Konstruktoren annotiert werden. Ist eines davon als CodeUnit markiert und wird als Annotations-Parameter ein String übergeben, so wird hieraus ein Annotations-Modell geparsed. Cherry erzeugt dann aus jeder Instanz des Annotations-Modells einen Builder. Als FixedCodeUnit annotierte Elemente werden in die Ursprungs-CodeUnit des logisch darüber liegenden Annotations-Modells übernommen.

Alle zur Informationsanreicherung verfügbaren Annotationen sind im Package `annotations` zu finden. Grundsätzlich gibt es für diese Annotationen mehrere Einschränkungen. Durch eine Target-Annotation wird gesteuert, an welche Teile des Java-Quelltextes die Annotationen überhaupt angebracht werden dürfen. Diese Einschränkung wird also normalerweise von der verwendeten IDE erzwungen. Abbildung 5.3 zeigt auf, wo die jeweiligen Annotationen stehen dürfen. Es existieren noch weitere, nicht erzwungene Restriktionen, welche durch den Parseprozess eingehalten werden. Allgemein sind die einzigen beiden Annotationen, welche eigenständig verwendet werden dürfen, die CodeUnit- und FixedCodeUnit-Annotationen, alle anderen können nur in Kombination mit diesen beiden stehen. Einige der Einschränkungen beruhen auf semantischen Zusammenhängen, denn beispielsweise eine HasGetter-Annotation macht nur an einem Feld Sinn. Andere Limitierungen sind auf den aktuellen Entwicklungsstand zurückzuführen.

	Class	Field	Method	Constructor
@CodeUnit				
@FixedCodeUnit				
@HasGetter				
@HasSetter				
@VariableModifier				
@VariableType				
@VariableParams				

Abbildung 5.3: Darstellung der legalen Ziele für die von Amber bereitgestellten Annotationen. Eine grüne Markierung kennzeichnet ein Ziel als legal.

5.2.1.2 Bedeutung und Limitationen der Annotationen und deren Auswirkung auf die erzeugten Builder

In den nachfolgenden Absätzen werden die unterschiedlichen Annotations-Ziele und die Wirkung einer bestimmten Annotationen hierauf zusammengefasst. Zudem wird auf durch den aktuellen Stand der Implementation bedingte, Einschränkungen eingegangen.

Listing 5.5: Darstellung der Annotations-Möglichkeiten an einer Klasse.

```
1 @CodeUnit("ExampleClass") @VariableModifier
2 public class ExampleRef {
3 //...
4 }
```

Listing 5.5 zeigt die Annotations-Möglichkeiten an einer Klasse. Diese kann nur als CodeUnit und optional mit VariableModifier markiert werden. Hierdurch wird ein Builder für die Klasse mit dem in der CodeUnit übergebenen Namen erzeugt. Eine Klasse darf dabei jede Kombination von Java-Modifiern haben. Liegt eine VariableModifier-Annotation vor, müssen gewünschte Modifier explizit mit dem später erzeugten Builder gesetzt werden. An der Referenzklasse vorhandene Modifier werden standardmäßig ignoriert. Es werden keine weiteren Informationen der Klasse, zum Beispiel Interfaces und generische Typparameter, berücksichtigt. Diese müssten jedoch lediglich im Nachgang implementiert werden. Details zum Erweiterungsprozess werden im Anschluss an dieses Kapitel beschrieben.

Listing 5.6: Darstellung der Annotations-Möglichkeiten an einem Feld.

```
1 @CodeUnit("ExampleClass")
2 public class ExampleRef {
3 @FixedCodeUnit @HasGetter @HasSetter
4 private int id;
5
6 @FixedCodeUnit
7 protected static float f;
8
9 @CodeUnit("PublicString") @HasGetter @HasSetter
10 public String s;
11
12 @CodeUnit("Variable") @VariableModifier @VariableType
13 Object o;
14 }
```

Das Listing 5.6 zeigt die Annotations-Möglichkeiten an einem Feld. Fast jede der Annotationen, außer VariableParams, kann hier angebracht werden. Genau wie bei einer

Klasse wird aus einem Feld, welches als CodeUnit annotiert ist, ein Annotations-Modell gebildet. Ist dieses als FixedCodeUnit markiert, wird das Feld in die Ursprungs-CodeUnit des zugehörigen Klassen-Annotations-Modells übernommen. Bei einer Annotation mit HasGetter oder HasSetter in Kombination mit FixedCodeUnit wird automatisch zum Feld noch ein Getter beziehungsweise Setter erzeugt. Werden die beiden Annotationen zusammen mit CodeUnit verwendet, so wird jedes Mal, wenn der für die CodeUnit generierte Builder genutzt wird, auch ein, dem beschriebenen Feld entsprechender, Getter beziehungsweise Setter generiert.

Wird für das Feld ein variabler Modifier oder Typ annotiert, so verhält sich das wie die VariableType-Annotation an einer Klasse: Vorhandene Daten werden ignoriert, wird eine Variabilität annotiert, müssen Werte hierfür später explizit gesetzt werden. Wurde ein variabler Datentyp nicht im Builder gesetzt, so wird bei der abschließenden Prüfung der CodeUnit ein Fehler ausgegeben und als Standardwert der Datentyp Object benutzt.

Bezüglich der Annotation von Feldern gibt es noch nicht implementierte Features. Werden in einer Zeile mehrere Felder in der Form `int a, b, c;` deklariert, wird nur das erste dieser Felder beachtet. Genauso wird eine Initialisierung bei Deklaration eines Feldes im aktuellen Proof-of-Concept ignoriert. Wird ein Feld als Array deklariert, wird die Deklaration als einfaches nicht-Array Feld interpretiert.

Listing 5.7: Darstellung der Annotations-Möglichkeiten von Konstruktoren und Methoden.

```
1  @CodeUnit("ExampleClass")
2  public class ExampleRef {
3  @FixedCodeUnit
4  private ExampleRef() {
5  //...
6  }
7
8  @CodeUnit("Constructor") @VariableModifier @VariableParams
9  public ExampleRef(int i) {
10 //...
11 }
12
13 @FixedCodeUnit
14 public static void ExampleStaticMethod() {
15 //...
16 }
17
18 @CodeUnit("Method") @VariableModifier @VariableParams
19 void ExampleMethod() {
20 //..
21 }
```

```
22 }
```

Methoden und Konstruktoren können als CodeUnit oder FixedCodeUnit markiert werden. Bei einer Annotation als FixedCodeUnit wird die Methode oder der Konstruktor, genau wie bei einem Feld, in die Ursprungs-CodeUnit des Annotations-Modells der übergeordneten Klasse übernommen. Entsprechend wird bei der Verwendung der CodeUnit-Annotation ein Annotations-Modell für die Methode oder den Konstruktor gebildet. Der Körper der Methode oder des Konstruktors muss beim Bauen immer gesetzt werden, ansonsten wird ein leerer Standard-Körper verwendet. Der Rückgabewert kann ebenso nicht als Variabilität annotiert werden, da dieser als standardmäßig variabel angesehen wird. Als Defaultwert wird hier, bei fehlender Angabe, void angenommen. Listing 5.7 zeigt die unterschiedlichen Annotations-Möglichkeiten für Methoden und Konstruktoren auf.

Da vor dem Parsen der Referenzimplementation keine Prüfung der Annotationen durchgeführt wird, sind Kombinationen wie ein static-Feld mit einer HasGetter Annotation möglich. In der aktuellen Version wird in diesem Fall ein Getter erzeugt, welcher auf eine nicht vorhandene Instanzvariable verweist. Es gibt mehrere solcher potentieller Fehlerquellen. Es macht jedoch wenig Sinn diese Fehler an einer späteren Stelle des Erzeugungsprozesses zu handhaben, für die verteilte interne DSL müsste im Vorfeld eine Syntaxprüfung erfolgen. Dies wurde in diesem Proof-of-Concept aus Aufwandsgründen nicht umgesetzt.

5.2.1.3 Visitor & Parser

Mithilfe von Visitor-Klassen werden die Knotenpunkte des als AST eingelesenen Quelltextes verarbeitet. Abhängig vom Typ des Knotenpunkts wird ein Parser aufgerufen, welcher prüft, ob eine CodeUnit- oder FixedCodeUnit-Annotation angebracht wurde. Die jeweiligen Parser-Klassen steuern die weitere Verarbeitung der Node im entsprechenden Kontext. Neben der Verarbeitung der Konkreten Information zur annotierten Klasse ruft der ClassAnnotationParser noch weitere Visitor auf. Diese sorgen dann für die Verarbeitung etwaiger innerhalb der annotierten Klasse befindlicher FixedCodeUnits. Der Umkehrschluss hieraus ist, dass FixedCodeUnit-Annotationen nur dann berücksichtigt werden, wenn diese innerhalb einer als CodeUnit annotierten Klasse stehen.

5.2.1.4 Annotation Modell

Für jede CodeUnit-Annotation erzeugt Amber ein AnnotationModel. Neben einem Identifier und der im Konzept bereits erläuterten Ursprungs-CodeUnit, enthält dieses zusätzlich

zwei Sets aus AnnotationType-Einträgen. AnnotationType ist hier ein Enum zur Identifikation von in der Referenzimplementation gemachten Annotationen. Eines der Sets enthält Informationen zu Variabilitäten, das heißt, ob beispielsweise der Modifier veränderlich ist. Das andere beinhaltet Daten zu Erweiterungen. Mit Erweiterung ist hier gemeint, dass es zu einem Feld automatisch einen Getter, Setter oder vergleichbares gibt. Momentan sind dies die einzigen Erweiterungen, die durch Annotationen getätigt werden können. Es wäre jedoch auch denkbar, zum Beispiel eine IsSingleton-Annotation einzuführen, welche dann eine Erweiterung der Klasse wäre.

Im Generator für die Builder werden abhängig von den Variabilitäts-Annotationen passende Builder-Methoden als Komponenten ausgewählt und an den Builder angehängt. Zudem wird die Information, welche Erweiterungs-Annotationen existieren, in den Builder übertragen, damit bei der Anwendung des Builders später entsprechende Verarbeitungsschritte durchgeführt werden können.

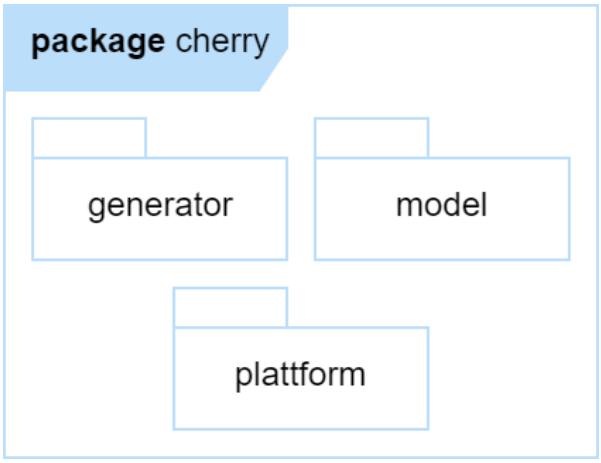

Abbildung 5.4: Darstellung der Unter-Packages im cherry Package.

5.2.2 Cherry

Aus den aus der Referenzimplementation entstandenen Annotations-Modellen werden von Cherry im darauffolgenden Schritt die später verwendbaren Builder erzeugt. Zudem ist im Package `model` das von dieser generierten internen DSL beschriebene CodeUnit-Modell definiert. Der im Konzept erwähnte Pattform-Code ist in Cherry im Package `platform` zu finden. Abbildung 5.4 zeigt noch einmal die erwähnten Packages.

In den folgenden Abschnitten werden die Funktionsweise und Aufgabe von Cherry genau erläutert. Im Anschluss daran wird die Verwendung der erzeugten Bilder noch einmal erklärt und mit Beispiel verdeutlicht.

5.2.2.1 CodeUnit-Modell

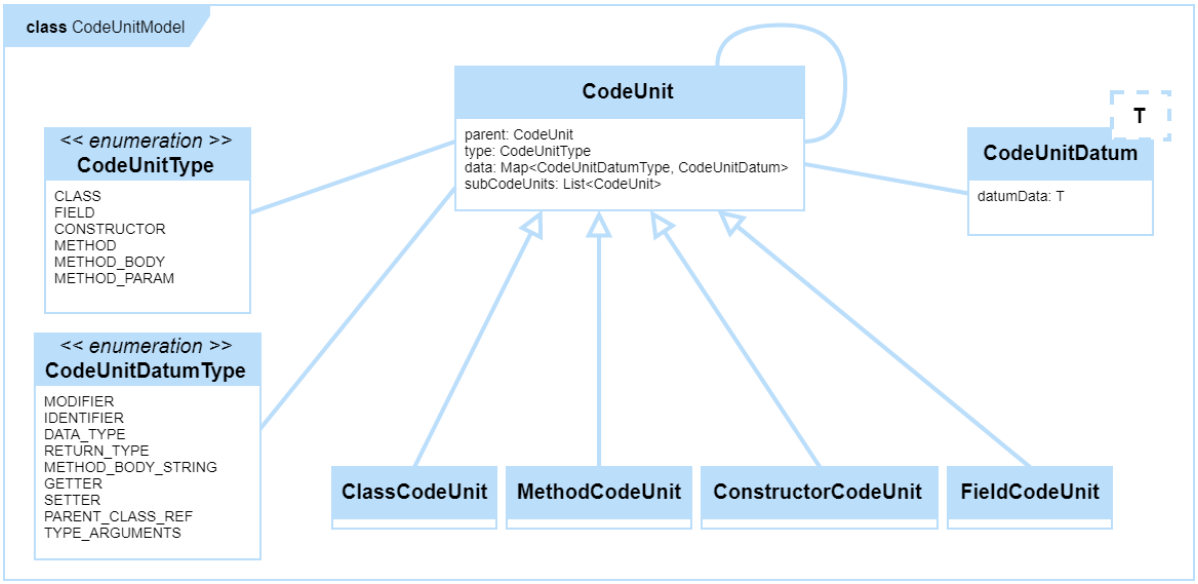

Abbildung 5.5: Klassen Diagramm des CodeUnit-Modells.

Wie bereits zuvor erwähnt, könnte man das CodeUnit-Modell als das Kernstück von Spectrum ansehen. Eine Instanz des CodeUnit-Modells wird durch Verwendung der erzeugten internen DSL beschrieben und aus dieser kann dann zum Beispiel wieder konkreter Java-Quelltext erzeugt werden.

Das Diagramm auf Abbildung 5.5 zeigt das CodeUnit-Modell und seine Hilfsklassen. Die Grundidee des CodeUnit-Modells wurde im Konzeptkapitel bereits erläutert. Konkret wird der Zusammenhang zwischen CodeUnit und SubCodeUnits dadurch dargestellt, dass eine CodeUnit eine Liste mit weiteren CodeUnits hält. Der Typ der CodeUnit wird in einem Feld eines Wertes des CodeUnitType-Enums gehalten. Es gibt nicht immer zwingend zu jedem CodeUnitType auch einen möglichen Builder. Die CodeUnitTypes METHOD_BODY und METHOD_PARAM existieren nur als SubCodeUnits des CodeUnitTypes METHOD.

Besonderes Augenmerk fällt hier darauf, dass in der Implementation, entgegen dem Konzept, von CodeUnit abgeleitete Klassen existieren. Wie bereits beschrieben, wird die Identifikation des Typs einer CodeUnit anhand des CodeUnitType durchgeführt. Die Ableitungen von CodeUnit haben keine eigene Funktion und dienen nur zur Fehlerreduzierung bei der Verwendung der Builder. Durch spezielle CodeUnit-Klassen ist es möglich, Builder-Methoden zu generieren, die einschränken, welche Art von CodeUnit als SubCodeUnit übergeben werden darf. Beispielsweise kann mithilfe der MethodCodeUnit-Klasse eine Methode für ein Builder der eine ClassCodeUnit beschreibt erzeugt werden, welche ausschließlich die Übergabe einer MethodCodeUnit erlaubt. Konkret wäre das die Methode withMethod.

Das Konzept der Parametrisierung durch eine generische Datenstruktur wurde als Map mit dem Schlüssel CodeUnitDatumType und Einträgen des Typs CodeUnitDatum realisiert. CodeUnitDatumType ist abermals ein Enum, welches eine semantische Bedeutung hat und somit später erlaubt, den Parameter korrekt auszulesen. Der Parameter selbst, also das CodeUnitDatum, ist generisch parametrisiert und kann somit einzelne Werte oder auch Arrays halten. Allgemein kann hier jedoch jeder Datentyp stehen. Für den CodeUnitDatumType IDENTIFIER wird hier beispielsweise ein String als Typparameter verwendet. Zur Verdeutlichung zeigt Listing 5.8 die Implementation von CodeUnitDatum. Exemplarisch wird in Listing 5.9 dargestellt, wie einer CodeUnit ein CodeUnitDatum hinzugefügt werden kann und aud welche Weise dieses später ausgelesen werden muss.

Die Methode addCodeUnitDatum folgert den hinterlegten Datentyp anhand des als zweiten Parameter übergebenen Datums. Mit der Methode getCodeUnitDatum kann das unter einem CodeUnitDatumType hinterlegte Datum wieder ausgelesen werden. Da die Information, welcher generische Typparameter in der Map hinterlegt wurde, verloren geht, muss stets explizit auf den korrekten Typ umgewandelt werden. Dies stellte kein Problem dar, da unter einem bestimmten CodeUnitDatumType mit einem eindeutigen Datentyp zu rechnen ist.

Listing 5.8: Implementation der Klasse CodeUnitDatum.

```
1  public class CodeUnitDatum<T> implements Serializable {
2  private final T datumData;
3
4  public CodeUnitDatum(T datumData) {
5  this.datumData = datumData;
6  }
7
8  public T getDatumData() {
9  return datumData;
10 }
11
12 @Override
13 public String toString() {
14 if(datumData.getClass().isArray())
15 return Arrays.toString((Object[]) datumData);
16
17 return datumData.toString();
18 }
19 }
```

Listing 5.9: Beispiel zum konkreten hinterlegen und auslesen eines CodeUnitDatums.

```
1  void exampleDatumWriter(String identifier) {
2  CodeUnit cu = new CodeUnit();
```

```
3 cu.addCodeUnitDatum(CodeUnitDatumType.IDENTIFIER, identifier);
4 }
5
6 String exampleDatumReader(CodeUnit cu) {
7 return (String)
     cu.getCodeUnitDatum(CodeUnitDatumType.IDENTIFIER).getDatumData();
8 }
```

5.2.2.2 DSL Generator

Der BuilderGenerator folgt dem Konzept, dass jeder Builder komponentenbasiert ist. Die Komponenten sind hierbei die Builder-Methoden. Zum einen sind diese durch die im Annotations-Modell hinterlegten Variabilitäten vorgegeben und zum anderen aus dem Typ der von dem Builder beschriebenen CodeUnit ableitbar. Im Generator wird ein Builder dementsprechend auf Grundlage dieser Daten Stück für Stück zusammengebaut.

Im ersten Schritt wird die Referenzimplementation zu einer CompilationUnit geparsed. Diese wird dann von Amber verarbeitet, woraus eine Liste von Annotations-Modellen entsteht. Aus jedem dieser Modelle wird dann ein Builder erzeugt. Der Klassenname des Builders setzt sich aus dem Identifier des Annotations-Modells und dem Suffix UnitBuilder zusammen. Im Anschluss hieran werden, abhängig von der zu beschreibenden CodeUnit, aus einer Hilfsklasse, dem BuilderMethodTypeProvider, die Informationen abgefragt, welche Standardmethoden der Builder benötigt. Mit der BuilderMethodFactory wird aus dieser BuilderMethodType-Information eine entsprechende JavaPoet-MethodSpec zusammen gebaut. Danach werden auf gleichem Weg die MethodSpecs zur Beschreibung der Variabilitäten erzeugt und angehängt. Der Identifier der generierten Methode ist im BuilderMethodType-Enum als methodName hinterlegt und kann hier auch angepasst werden.

Um die in der defaultCodeUnit hinterlegten Daten in die Ursprungs-CodeUnit des Builders zu übertragen, wird die defaultCodeUnit zu einem Byte-Array serialisiert und als Literal in der erzeugten initializeDefaultCodeUnit-Methode hinterlegt. Diese Methode wird im Konstruktor des erzeugten Builders aufgerufen und sorgt dafür, dass aus dem Byte-Array wieder ein CodeUnit-Objekt deserialisiert wird. Die Erzeugung dieses MethodSpecs und die von JavaParser aus dem MethodSpec erzeugte Builder-Methode werden in Listing 5.10 gezeigt.

Listing 5.10: Quelltext zur Erzeugung des MethodSpecs für die Builder-Methode initializeDefaultCodeUnit und die daraus resultierende konrekte Methode in einem Builder.

```
1
```

```
2  //Erzeugung des MethodSpec in der BuilderMethodFactory
3  private MethodSpec createInitDefCodeUnitMethod() {
4  CodeUnit sourceCodeUnit = annotationModel.getDefaultCodeUnit();
5
6  byte[] serializedCodeUnit =
       SerializationUtils.serialize(sourceCodeUnit);
7
8  String codeUnitArrayLiteral = Arrays
9  .toString(serializedCodeUnit)
10 .replace("[","{")
11 .replace("]","}");
12
13 return MethodSpec.methodBuilder("initializeDefaultCodeUnit")
14 .addComment("Initializes this builder's data with default data encoded
       into a byte[]")
15 .addModifiers(Modifier.PRIVATE)
16 .addStatement("byte[] serializedCodeUnit = new byte[] $L",
       codeUnitArrayLiteral)
17 .addStatement("this.codeUnit = $T.deserialize(serializedCodeUnit)",
       SerializationUtils.class)
18 .build();
19 }
20
21 //Erzeugte konrete Methode in einem Builder
22 private void initializeDefaultCodeUnit() {
23 // Initializes this builder's data with default data encoded into a
       byte[]
24 byte[] serializedCodeUnit = new byte[] {-84, -19, 0/* more data */};
25 this.codeUnit = SerializationUtils.deserialize(serializedCodeUnit);
26 }
```

5.2.2.3 Abhängigkeit der erzeugten Builder zu Plattform-Code

Um die erzeugte DSL verwenden zu können, muss das Package `platform` zur Verfügung stehen. Hierin befinden sich Klassen zur Auflösung von Referenzen und zur Fehlerbehandlung. Da Getter und Setter erst zur Laufzeit, während die Builder bereits im Einsatz sind, erzeugt werden können, findet sich hier auch der DefaultCodeUnitProvider. Im aktuellen Stand von Spectrum ist die Funktionalität der Plattformklassen noch sehr beschränkt. Vor allem die Fehlerhandhabung ist nur exemplarisch implementiert. Momentan werden nur Klassenreferenzen auf die beschriebene Klasse selbst aufgelöst, um für den Proof-of-Concept die Definition von Singletons mithilfe der internen DSL möglich zu machen. Da die Referenz auf Basis eines CodeUnitDatums aufgelöst wird, könnten

selbst verständlich noch weitere Referenzen im Parseprozess hinterlegt werden und somit auch vom CodeUnitReferenceResolver gehandhabt werden.

5.2.2.4 Verwendung der erzeugten Builder

Listing 5.11 stellt die Anwendung der generierten Builder dar. Anzumerken ist, dass dieses Beispiel eine kaum sinnhafte CodeUnit beschreibt und lediglich die verschiedenen Möglichkeiten und wie man die einzelnen Builder-Methoden anwendet, aufzeigt.

Modifier werden mit einem Enum gehandhabt. Methoden, welche einen Datentypen als Parameter erwarten, bekommen diesen als String übergeben. Ein einfacher Weg, diesen zu erhalte, ist über die Klasse mit .class auf die Methode getName zuzugreifen. Der Körper einer Methode wird hier, genau wie bei JavaPoet, als String übergeben.

Ein zusammenhängendes Beispiel, wie aus einer Referenzimplementation mit Annotationen Builder generiert werden, wie man diese dann benutzen kann und wie daraus generierter Java-Quelltext aussieht, folgt am Ende dieses Kapitels.

Listing 5.11: Quelltext zur Verwendung verschiedener erzeugter Builder.

```
1  CodeUnit cu = ExampleClassUnitBuilder
2  .createWithIdentifier("ExampleIdentifier")
3  .withModifiers(CodeUnitModifier.PROTECTED)
4  .withField(VarUnitBuilder
5  .createWithIdentifier("exampleField")
6  .withModifiers(CodeUnitModifier.PRIVATE)
7  .withDataType(Object.class.getName())
8  .end())
9  .withMethod(MethodUnitBuilder
10 .createWithIdentifier("exampleMethod")
11 .withModifiers(CodeUnitModifier.PUBLIC)
12 .withParameter("param", String.class.getName())
13 .withMethodBody("return param")
14 .withReturnType(String.class.getName())
15 .end())
16 .withConstructor(ConstructorUnitBuilder
17 .create()
18 .withParameter("param", String.class.getName())
19 .withModifiers(CodeUnitModifier.PUBLIC)
20 .withMethodBody("//body als String")
21 .end())
22 .end();
```

5.2.3 Jade

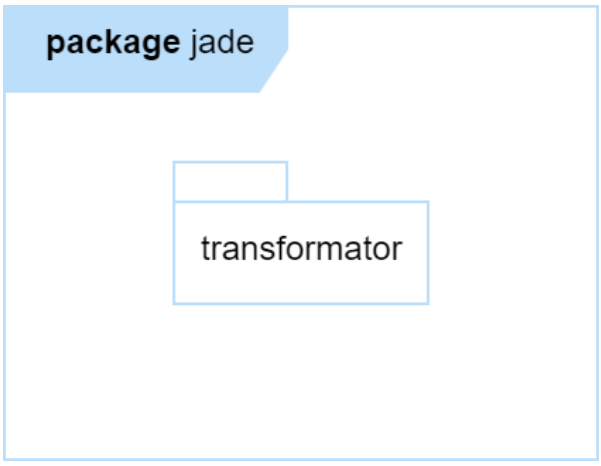

Abbildung 5.6: Darstellung des Unter-Packages im jade Package.

Die eher kleine Komponente Jade hat die Aufgabe, fertig beschriebene CodeUnits in ein für Scarlet verarbeitetbares Modell zu transformieren. Jade umfasst nur das Package `transformator`, in welchem auch nur die Klasse CodeUnitTransformator liegt. Dies ist auf Abbildung 5.6 zu sehen.

5.2.3.1 Transformator

Für die unterschiedlichen Elemente in Scarlets Java-Modell sind im CodeUnitTransformator Transformationsregeln in Form von Methoden definiert. Da das CodeUnitModell und somit auch die CodeUnitTypes ursprünglich aus Java-Quelltext abgeleitet wurden, ist die Umwandlung vergleichsweise unkompliziert. Dies ist am Beispiel der Transformation von Feldern in Listing 5.12 gut zu sehen. Mithilfe von Java-Streams werden hier die SubCodeUnits einer Klassen-CodeUnit gefiltert und dann die einzelnen Felder umgewandelt.

Listing 5.12: Umwandlung der Felder in den SubCodeUnits einer CodeUnit. Auszug aus der Klasse CodeUnitTransformator.

```
1 private List<JavaField> transformFields(CodeUnit cu) {
2 return cu.getSubCodeUnits()
3 .stream()
4 .filter(this::isField)
5 .map(this::transformField)
6 .collect(Collectors.toList());
7 }
8
```

```
9  private JavaField transformField(CodeUnit cu) {
10 JavaField jField = new JavaField();
11 jField.modifiers = this.transformModifier(cu);
12 jField.identifier = this.transformIdentifier(cu);
13 jField.type = this.transformType(cu);
14 jField.typeParams = this.transformTypeArguments(cu);
15
16 return jField;
17 }
```

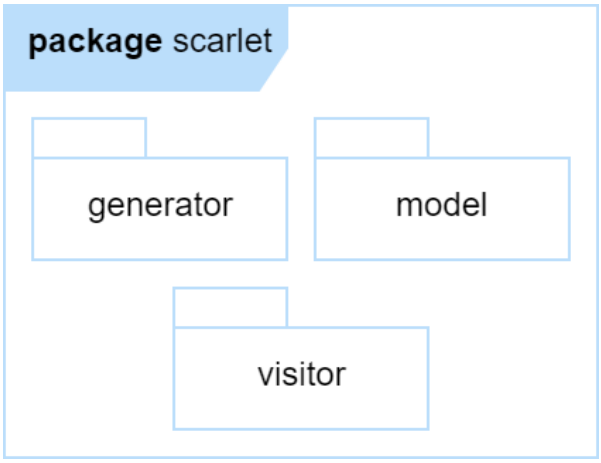

Abbildung 5.7: Darstellung des Unter-Packages im scarlet Package.

5.2.4 Scarlet

Die Komponente Scarlet stellt ein Modell zur Verfügung, welches die Struktur und den Aufbau einer Java-Klasse widerspiegelt. Die zweite Aufgabe von Scarlet ist es, aus diesem Java-Modell konkreten Java-Quelltext zu erzeugen. Die in Scarlet definierten Visitor finden keine Verwendung mehr. Ursprünglich haben diese Java-Quelltext eingelesen und in das Java-Modell überführt. Alle Unter-Packages von Scarlet sind auf Abbildung 5.7 dargestellt.

5.2.4.1 Allgemeiner Java-Klassengenerator

Der Generator ist vergleichsweise einfach aufgebaut. Der konkrete Quelltext wird abermals mit JavaPoet erzeugt. Für die unterschiedlichen im Java-Modell darstellbaren Java-Elemente sind in der Klasse JavaClassGenerator jeweils generate-Methoden implementiert. Diese Methoden erzeugen die für JavaPoet notwendigen Objekte. Mithilfe des Ja-

vaPoetTypeMappers werden die als Zeichenketten vorliegenden Klassennamen zu einem JavaPoet-Typen umgewandelt.

5.2.4.2 Modell zur Abbildung von Java-Klassen

Das Modell auf dessen Basis Scarlet den Java-Quelltext erzeugt, ist ein Teilmodell des Java-Metamodells. Soll Spectrum um neue Generiermöglichkeiten erweitert werden, so werden auch meist das Scarlet Java-Modell und der Java-Klassengenerator ergänzt.

5.2.5 Violet

Als letzte und auch kleinste Komponente stellt Violet einen Logger zur Verfügung. Dies ermöglicht es, vereinfacht eine Info- oder Fehlernachricht auf der Konsole auszugeben. Er sollte leicht um Ausgabe in einer Log-Datei erweiterbar sein.

5.3 Erweiterung von Spectrum

Im letzten Abschnitt dieses Kapitels wird beschrieben, wie ein beispielhafter Erweiterungsprozess für Spectrum aussehen könnte. Hierdurch sollen Anhaltspunkte gegeben werden, welche Komponenten und Klassen für eine Ausweitung des Funktionsumfangs bearbeitet werden müssen.

Allgemein gilt bei der Erweiterung von Spectrum immer das gleiche Prinzip. Soll eine neue Annotation hinzugefügt werden, muss in Amber diese Annotation definiert werden und im Modell muss das AnnotationType-Enum erweitert werden. In jedem Fall muss der entsprechende AnnotationParser die gewünschten neuen Daten aus dem AST in das Annotations-Modell übertragen.

Der einfachste Fall ist die Erweiterung der möglichen Ziele für FixedCodeUnit. Zuerst müssen die möglichen Targets der FixedCodeUnit entsprechend erweitert werden. Ist dies geschehen, sollten ein neuer Visitor und ein neuer Parser für das neue Ziel erstellt werden, hierbei kann man sich gut an der vorhandenen Struktur orientieren. Intern sind die Parser sehr ähnlich aufgebaut. Für die Annotation direkt und Annotationen, die in Abhängigkeit der Haupt-Annotation stehen, gibt es jeweils parse Methoden. Für die FixedCodeUnit wird eine parseFixedCodeUnitAnnotation-Methode implementiert, welche prüft, ob die entsprechende Annotation vorhanden ist. Ist dies der Fall wird die

vorhandene defaultCodeUnit des Modells um die neuen FixedCodeUnit-Daten erweitert. Bei Bedarf können neue CodeUnitTypes und CodeUnitDatumTypes angelegt werden.

Da es sich hier um eine FixedCodeUnit handelt, müssen keine neuen Builder-Methoden erzeugt werden. Um später Java-Quelltext generieren zu können, muss zuerst das Java-Modell von Scarlet um entsprechende Klassen oder Felder erweitert werden. Danach können im JavaClassGenerator neue generate-Methoden implementiert werden. Der letzte Schritt ist dann in Jade neue Transformationsregeln zu definieren, welche die Daten der CodeUnit in Java-Modell-Daten umwandeln.

Bei der Entwicklung von Spectrum wurde die Erfahrung gemacht, dass für eine Erweiterung fast jedes Mal ein Element sowohl als FixedCodeUnit annotierbar, als auch als Variabilität von einem Builder beschreibbar sein soll. Da bei einer FixedCodeUnit keine Erweiterung des BuilderGenerators notwendig ist, ist es sehr hilfreich, zuerst den gesamten Erweiterungsprozess für eine FixedCodeUnit durchzuführen und sich dann erst um die Implementation der fehlenden Teile für die CodeUnit-Annotation zu kümmern.

Handelt es sich um eine neue Variabilität, die mit der internen DSL ausgedrückt werden können soll, muss zuerst die BuilderMethodFactory um die neue Methode erweitert werden. Parallel dazu wird ein neuer BuilderMethodType hinzugefügt. Jetzt kann dafür Sorge getragen werden, dass Amber im entsprechenden Parser die für den Generator notwendige Information, dass eine entsprechende Methode generiert werden soll, als Variabilitäts-Annotation aufnimmt. Wurden die oben erwähnten Schritte für die FixedCodeUnit bereits einmal implementiert, wäre die Erweiterung jetzt abgeschlossen.

Die einzelnen Schritte können, vor allem Zusammenhang mit Erweiterungs-Annotationen, mehr oder weniger komplex ausfallen. Es könnten auch grundsätzliche Änderungen an Spectrum notwendig sein. Diese Fälle können in dieser Arbeit jedoch nicht alle abgehandelt werden, da diese kaum absehbar sind.

5.4 Zusammenhängendes Beispiel: Von der Referenzimplementation zum erzeugten Java-Quelltext

Die folgenden Quelltextausschnitte sollen als zusammenhängendes Beispiel verdeutlichen, welche Builder aus einer beziehungsweise mehreren Referenzimplementationen erzeugt werden, wie diese Builder dann beispielsweise eingesetzt werden können und wie der aus diesem Einsatz resultierende, erzeugte Quelltext konkret aussieht.

Listing 5.13 und 5.14 zeigen zwei Referenzimplementationen mit Annotationen. Die An-

notationen wurden so gewählt, dass mit der erzeugten DSL ein Logger beschrieben werden kann. Cherry erzeugt aus diesen annotierten Referenzimplementationen für jede CodeUnit einen Builder. Zum Einsatz kommen diese Builder im Listing 5.15. Hier werden ein Logger und ein rudimentärer LoggerTester beschrieben. Mit Jade werden die konkreten Instanzen der CodeUnits dann zu Java-Modellen transformiert. Abschließend erzeugt Scarlet daraus den in Listing 5.16 und 5.17 gezeigten Java-Quelltext.

Listing 5.13: Quelltext einer annotierten Referenzimplementation für eine Singleton Klasse.

```
1  @CodeUnit("Singleton")
2  public class SingletonClass {
3  @FixedCodeUnit
4  private static SingletonClass instance;
5
6  @FixedCodeUnit
7  private SingletonClass() { }
8
9  @FixedCodeUnit
10 public static SingletonClass getInstance() {
11 if(instance == null) {
12 instance = new SingletonClass();
13 }
14
15 return instance;
16 }
17 }
```

Listing 5.14: Quelltext einer annotierten Referenzimplementation für eine Klasse mit den für den Logger benötigten Annotationen.

```
1  @CodeUnit("Clazz")
2  public class ReferenceClass {
3  @CodeUnit("GetVar") @HasGetter @VariableType
4  private int getvar;
5
6  @CodeUnit("PublicMethod") @VariableParams
7  public void publicMethod() {}
8
9  @CodeUnit("PrivateMethod") @VariableParams
10 private void privateMethod() {}
11
12 @CodeUnit("PublicStaticMethod") @VariableParams
13 public static void publicStaticMethod() {}
14 }
```

Listing 5.15: Quelltext zur Erzeugung des DataLoggers und des zugehörigen Testers.

```
1  void GenerateDataLogger() {
2  CodeUnit dataLogger = BuildDataLogger();
3  CodeUnit dataLoggerTester = BuildDataLoggerTester();
4
5  System.out.println(dataLogger);
6  System.out.println(dataLoggerTester);
7
8  WriteToFile(dataLogger);
9  WriteToFile(dataLoggerTester);
10 }
11
12 private CodeUnit BuildDataLogger() throws IOException {
13 return SingletonUnitBuilder
14 .createWithIdentifier("DataLogger")
15 .withField(GetVarUnitBuilder
16 .createWithIdentifier("logCount")
17 .withDataType(int.class.getName())
18 .end())
19 .withMethod(PublicMethodUnitBuilder
20 .createWithIdentifier("logInfo")
21 .withParameter("message", String.class.getName())
22 .withMethodBody("this.log(\"[info]\", message);")
23 .end())
24 .withMethod(PublicMethodUnitBuilder
25 .createWithIdentifier("logError")
26 .withParameter("message", String.class.getName())
27 .withMethodBody("this.log(\"[error]\", message);")
28 .end())
29 .withMethod(PrivateMethodUnitBuilder
30 .createWithIdentifier("log")
31 .withParameter("prefix", String.class.getName())
32 .withParameter("message", String.class.getName())
33 .withMethodBody("System.out.println(prefix + \" \" + message);\n" +
34 "logCount++;")
35 .end())
36 .end();
37 }
38
39 private CodeUnit BuildDataLoggerTester() {
40 return ClazzUnitBuilder
41 .createWithIdentifier("DataLoggerTester")
42 .withMethod(PublicStaticMethodUnitBuilder
```

```
43 .createWithIdentifier("main")
44 .withMethodBody("DataLogger logger = DataLogger.getInstance();\n" +
45 "logger.logInfo(\"Hello, World!\");\n" +
46 "logger.logError(\"Don't Panic!\");\n" +
47 "int c = logger.getLogCount();\n" +
48 "logger.logInfo(\"Logged Messages: \" + c);")
49 .end())
50 .end();
51 }
52
53 private void WriteToFile(CodeUnit cu) throws IOException {
54 JavaFile jf = TransformToJavaFile(cu, "scarlet.generated");
55 jf.writeTo(new File("src-generated/"));
56 }
57
58 private JavaFile TransformToJavaFile(CodeUnit cu, String packageName)
      throws IOException {
59 CodeUnitTransformator cut = new CodeUnitTransformator();
60 JavaClassFile j = new JavaClassFile();
61 j.javaClass = cut.transformClassCodeUnit(cu);
62 j.packageName = packageName;
63
64 JavaClassGenerator jcg = new JavaClassGenerator();
65 return jcg.generateJavaFileFromModel(j);
66 }
```

Listing 5.16: Quelltext der erzeugten DataLogger-Klasse.

```
1 package scarlet.generated;
2
3 import java.lang.String;
4
5 public class DataLogger {
6 private static DataLogger instance;
7
8 private int logCount;
9
10 private DataLogger() {
11
12 }
13
14 public static DataLogger getInstance() {
15 if (instance == null) {
16 instance = new DataLogger();
17 }
```

```
18
19 return instance;
20 }
21
22 public void logInfo(String message) {
23 this.log("[info]", message);
24 }
25
26 public void logError(String message) {
27 this.log("[error]", message);
28 }
29
30 private void log(String prefix, String message) {
31 System.out.println(prefix + " " + message);
32 logCount++;
33 }
34
35 public int getLogCount() {
36 return logCount;
37 }
38 }
```

Listing 5.17: Quelltext der erzeugten DataLoggerTester-Klasse.

```
1 package scarlet.generated;
2
3 class DataLoggerTester {
4 //String[] args fehlt, da Arrays noch nicht implementiert sind
5 public static void main() {
6 DataLogger logger = DataLogger.getInstance();
7 logger.logInfo("Hello, World!");
8 logger.logError("Don't Panic!");
9 int c = logger.getLogCount();
10 logger.logInfo("Logged Messages: " + c);
11 }
12 }
```

6 Evaluierung

In den folgenden Absätzen wird das Konzept und die Implementation des Proof-of-Concept evaluiert. Zudem ist ein Abschnitt der Beurteilung der Wirtschaftlichkeit des Ansatzes gewidmet. Für jede Komponente von Spectrum wird noch einmal auf das Konzept Bezug genommen und bewertet ob dieses grundlegend implementiert wurde. Bei Bedarf wird das Konzept oder die Implementation einzelner Teile der Komponenten noch einmal genauer diskutiert. Abgeschlossen wird dieses Kapitel mit einer Erläuterung der bisher bekannten vermutlichen Grenzen des Lösungsansatzes.

6.1 Wirtschaftlichkeit

Spectrum ermöglicht es, mit der aktuellen Implementation die händische Entwicklung des Generators einzusparen. Ob die Definition einer Referenzimplementation mit Annotationen und die anschließende Formulierung der konkreten Klassen mit der erzeugten DSL tatsächlich messbare Vorteile bringt, kann im Rahmen dieser Arbeit nur schwer festgestellt werden. Dies liegt zum einen daran, dass zur Umsetzung kompletter Projekte noch viele Features von Spectrum fehlen. Zum anderen müssen hierfür verschiedene Projekte miteinander verglichen werden. Dies wäre Gegenstand einer eigenen weiterführenden Arbeit. Im Rahmen dieser Thesis kann nur die Aussage getroffen werden, dass eine Einsparung zu vermuten ist.

6.2 Amber

Amber erfüllt die im Konzept in Abschnitt 4.2.1 beschriebenen Aufgaben. Das Konzept sieht vor, dass in einer Referenzimplementation durch Annotationen Variabilitäten definiert werden können. Zudem ist es möglich, nicht-variable, im Konzept sogenannte statische Elemente, zu annotieren, welche als Standarddaten, beispielsweise einer Klasse, übernommen werden. Das Annotations-Modell wurde als Abstraktionsschicht zur Vereinfachung eingeführt.

Das bedeutet, dass die in diesem Teil des Konzepts beschriebenen Ansätze grundsätzlich umsetzbar waren. Trotzdem gibt es noch einige Punkte im Konzept und bei der Implementation, die auf andere Weise angegangen werden könnten und hierdurch potenziell verbesserbar wären.

6.2.1 Namenskonflikt der CodeUnit-Annotation

Die Annotationen erfüllen ihren Zweck und scheinen in ihrer Benennung verständlich zu sein. Ein Problem könnte hier die Annotationen CodeUnit, beziehungsweise FixedCodeUnit, darstellen, da hier eine Art Namenskonflikt vorliegt. Auf der einen Seite bedeutet die Annotation CodeUnit, dass aus einem Java-Element im Verlauf des Erzeugungsprozesses ein Builder generiert wird. Auf der anderen Seite hat auch das von einem solchen Builder beschriebene Modell den Namen CodeUnit. Dies könnte im Sprachgebrauch für Verwirrungen sorgen.

Abhilfe würde hier grundsätzlich eine Umbenennung entweder des Metamodells von Cherry oder der Annotationen von Amber schaffen. Einfacher wäre es vermutlich, die Annotationen umzubenennen. Für das Modell scheint die bestehende Namenswahl zutreffend zu sein.

Eventuell wäre hier auch der richtige Weg, die CodeUnit-Annotation vollständig zu kürzen. Informationen zum Identifier des erzeugten Builders würden dann aus dem bestehenden Namen des annotierten Elements gewonnen werden. Sobald ein Element mit einer Variabilität gekennzeichnet wurde, würde für dieses dann auch ein Builder erzeugt. Die FixedCodeUnit-Annotation könnte dann zum Beispiel, passend zu den anderen Annotationen, zu IsFixed umbenannt werden.

6.2.2 Ursprungs-CodeUnit im Annotations-Modell

Auch das momentane Verfahren zur Übergabe von Standarddaten der CodeUnit eines Builders ist aktuell zweckmäßig, sollte jedoch auch noch einmal genau auf den Prüfstand gestellt werden. Aktuell werden besagte Standarddaten in Form eines CodeUnit-Objekts im Annotations-Modell gehalten und dann als serialisierter Byte-Array bei Verwendung eines erzeugten Builders wieder initialisiert. Dieser Vorgang wird in Abschnitt 5.2.2.2 beschrieben und anhand eines Listings 5.10 erläutert.

In diesem Abschnitt geht es erst einmal grundsätzlich darum, dass durch die Verwendung eines CodeUnit-Objektes für das Halten dieser Ursprungsdaten das Annotations-Modell mit dem CodeUnit-Modell gekoppelt wird. Insbesondere mit Blick auf eine sauberere und leichter verständlichere Lösung zum späteren re-initialisieren der CodeUnit, sollte

überlegt werden, ob hier nicht eine andere zusätzliche Datenstruktur im Annotations-Modell hinterlegt werden könnte. Dies würde die Modularisierung der Komponenten von Spectrum noch weiter erhöhen und somit die Schnittstelle von Amber zu Cherry weiter schärfen.

6.2.3 Verbesserte Strukturierung des Parsers durch Vererbung oder Generics

Große Quelltextanteile innerhalb der verschiedenen Parser wiederholen sich. Grundsätzlich sollte hier durch Refaktorierung sich wiederholender Code in eine Basisklasse übertragen werden. Der von JavaParser ausgegebene AST basiert auf einem Baum aus Knotenpunkten, welche keine eigenen Klassen haben. Stattdessen implementieren die Knotenpunkte Interfaces mit generischen Typparametern. Daher kann nicht ohne weiteres in den Parse-Methoden von Amber auf gemeinsame Basistypen zurückgegriffen werden. Ein möglicher Lösungsansatz könnte hier sein, die AnnotationParser ebenfalls mit Java-Generics zu versehen und auf diese Weise eine Verallgemeinerung zu ermöglichen.

6.3 Cherry

Das Konzept zur Komposition und Erzeugung der Builder wurde im Proof-of-Concept umgesetzt und hat sich auch während des Entwicklungsprozesses als gut erweiterbar herausgestellt. Die im Konzept beschriebenen Funktionen konnten gut umgesetzt werden. Die erzeugte interne DSL scheint praktikabel zu sein.

6.3.1 Objektorientierter Ansatz für das CodeUnit-Modell

Obwohl das CodeUnit-Modell den Anforderungen im Abschnitt 3.2.2 der Analyse genügt und entsprechend dem Konzept umgesetzt wurde, sollte der nicht verwendete objektorientierte Ansatz trotzdem für eine zukünftige, alternative Umsetzung des Konzeptes in Betracht gezogen werden.

Das CodeUnit-Modell würde als vollständige objektorientierte Struktur implementiert werden. Für jeden CodeUnit-Typen wäre dies eine eigene CodeUnit-Klasse. Für dessen Parameter bräuchte es keine generische Datenstruktur, sondern spezifische Felder. Der größte Vorteil hieran wäre, dass der Aufbau des CodeUnit-Metamodells leichter nachzuvollziehen wäre. Dies würde potenziell den gesamten Erweiterungsprozess von Spectrum

vereinfachen. Der Arbeitsaufwand ist vermutlich vergleichbar, müsste aber in einem konkreten Projekt gemessen werden.

6.3.2 Verwendung von Pattform-Code zur Übertragung der Urspungs-CodeUnit in einen Builder

Wie in der Evaluierung zu Amber erwähnt, wird die Ursprungs-CodeUnit innerhalb des Builders durch Deserialisierung aus einem Byte-Array instanziiert. Dieser Lösungsansatz ist äußerst undurchsichtig. Es gibt praktisch keine Möglichkeit, im Nachhinein festzustellen, welche Daten genau für einen Builder initialisiert werden. Der größte Vorteil an diesem Ansatz war, dass jede Art von CodeUnit, unabhängig von der genauen Zusammensetzung, instanziiert werden konnte. Somit musste auch bei jeder Erweiterung von Spectrum keine Rücksicht auf diese Initialisierung genommen werden.

Entgegen dem ursprünglichen Konzept, wurde in der Implementation, wie in Abschnitt 5.2.2.3 beschrieben, auf Pattform-Code zurückgegriffen. Wenn es gelingt, die Initialisierung einer CodeUnit allgemein zu formulieren, könnte auch hierfür Pattform-Code angelegt werden. Dieser würde dann aus einer konkreten CodeUnit-Instanz eine Reihe von Statements erzeugen. Die Statements würden den Java-Code beinhalten, der notwendig wäre, um die CodeUnit zu initialisieren. Der erzeugte Java-Code würde dann als weitere Komponente in den Builder-Quelltext einfließen.

6.4 Jade

Der implementierte Transformator erfüllt seine Aufgaben und lässt sich dadurch, dass er für jedes Modellelement eine eindeutige Transformationsfunktion angibt, leicht warten und erweitern.

6.4.1 Verarbeitung des CodeUnit-Modells mit einem Visitor Pattern

Das CodeUnit-Modell wird mit Java-Streams traversiert und ermöglicht auf diese Weise, potenziell eine einfache Umstellung auf parallelisierte Verarbeitung. Selbstverständlich müsste hierfür noch eine genaue Analyse der Datenabhängigkeiten innerhalb des Modells durchgeführt werden, um herauszufinden, an welcher Stelle eine Parallelisierung überhaupt machbar ist. Unter Umständen ist eine Anpassung des momentan sequentiellen Algorithmus notwendig. Zudem ist allgemein fraglich, ob eine Parallelisierung in

diesem Anwendungsfall überhaupt Performancegewinne zur Folge hätte, da die Menge der verarbeiteten CodeUnits vermutlich eher gering bleibt.

Sollte das CodeUnit-Modell umgestellt werden und der in der Evaluierung von Cherry erwähnte objektorientierte Ansatz zur Anwendung kommen, könnte für die Verarbeitung des CodeUnit-Modells auch ein Visitor Pattern in Betracht kommen. Gerade wenn immer mehr Transformationsregeln implementiert werden müssen, könnte dieses Muster den Quelltext hier strukturierter und übersichtlicher machen.

6.5 Scarlet & Violet

Wie im Konzept in Abschnitt 4.2.4.2 beschrieben, erzeugt Scarlet aus dem Java-Modell konkreten Java-Quelltext. Auch hier lassen sich Erweiterungen, durch die im Lösungskapitel beschriebene Struktur, mit geringem Aufwand einpflegen.

Violet war im ursprünglichen Konzept nicht vorgesehen, erfüllt die Aufgabe des Loggings als Querschnittsfunktion aber einwandfrei und könnte lediglich noch um weitere Features, wie zum Beispiel Logging in eine Datei, erweitert werden.

6.6 Grenzen des Lösungsansatzes

Für diesen Abschnitt muss eine klare Trennung vollzogen werden. Der erste Teil wird sich mit Grenzen der aktuellen Implementation von Spectrum beschäftigen. Diese Limitationen können vermutlich vollständig aufgelöst werden, wenn die entsprechende Arbeitskraft zu Erweiterung von Spectrum aufgebracht wird. Im zweiten Teil dieses Abschnittes werden Themen angesprochen, welche sich nur in Teilen oder vielleicht gar nicht realisieren lassen.

6.6.1 Limitationen aufgrund des aktuellen Implementationsstandes

Viele Sprachfeatures von Java können von Spectrum noch nicht annotiert oder geparsed werden. In den meisten Fällen liegt dies daran, dass die Implementation dieser Funktionen mit Arbeitsaufwand verbunden ist, der zeitlich nicht mehr im Rahmen dieser Thesis machbar war. Im Lösungs-Kapitel nicht erwähnt, aber in Teilen implementiert, ist der Support für generische Typparameter. Felder, die als FixedCodeUnit annotiert sind, können bereits mit Parametern versehen werden und werden dann von Scarlet auch als Java-Quelltext erzeugt.

Zu den noch nicht unterstützten Features gehören unter anderem der Support für Arrays, die Verarbeitung von Informationen zur Vererbung, die Initialisierung von Feldern bei Deklaration und die Verarbeitung von bestehenden Annotationen, die nicht zu Spectrum gehören. Natürlich ist dies nur ein kleiner Auszug der möglichen Erweiterungen. Allgemein gilt durch die CodeUnit-Struktur, dass fast alles als CodeUnit mit Parametern darstellbar ist. Voraussetzung hierfür ist, dass die notwendigen Daten im AST liegen oder auf andere Weise beschafft werden können.

6.6.2 Allgemeine Limitationen des Konzepts

Da dieses Konzept auf ein Informationsanreicherung mithilfe von Annotation aufbaut, also davon ausgeht, dass die Referenzimplementation in Java geschrieben ist, gibt es hier grundsätzliche Limitationen. Einige Teile des Java-Quelltextes können nicht annotiert werden. Dazu gehören Lambdas und einzelne Ausdrücke im Körper einer Methode. Somit ist es zum Beispiel nicht möglich, eine lokale Variable gezielt zu annotieren. Natürlich sind hierfür Workarounds denkbar, wie zum Beispiel eine Annotation mit einer Zeichenkette als Parameter. Als solcher könnte dann beispielsweise der Name, der zu referenzieren Variable, übergeben werden. Hier müsste dann der Annotations-Parser den Teil-AST der Methode ablaufen und die entsprechende Variablendeklaration suchen und anschließend verarbeiten.

Sicherlich gibt es noch weitere Limitationen, welche sich erst im Laufe der Zeit beziehungsweise wenn die Notwendigkeit für ein bestimmtes Feature entsteht, zeigen. Aber wie oben bereits beschrieben wird, sind beispielsweise Probleme die auf der Annotierbarkeit von einem Teil des Java-Quelltextes beruhen, wahrscheinlich, wenn auch mit Umständen verbunden, lösbar. Nach dem bisherigen auf dieser Arbeit beruhenden Kenntnisstand, scheinen die meisten Java Features umsetzbar oder zumindest durch einen Workaround ersetzbar.

7 Abschluss

Zum Abschluss dieser Bachelorthesis werden im folgenden Kapitel die aus dieser Arbeit gewonnenen Erkenntnisse noch einmal zusammengefasst und ein Ausblick auf die mögliche zukünftige Weiterentwicklung dieses Konzepts gegeben. Außerdem werden mögliche Schnittstellen für potentielle nachfolgende Arbeiten umrissen.

7.1 Zusammenfassung

> „Wie kann, ausgehend von bestehendem Java-Code, die Entwicklung eines Generators zur Erhöhung der Wirtschaftlichkeit modellgetriebener Softwareentwicklung automatisiert werden?“

Auf Basis der ausführlichen Einführung in die Grundlagen der modellgetriebenen Softwareentwicklung in Kapitel 2 und den damit zusammenhängenden Themen aus dem Software Engineering wurden im Analyse-Kapitel die mit der zu Beginn des Kapitels gestellten Frage verbundenen Problemstellungen untersucht. Die aufgezeigten Optionen wurden ausführlich diskutiert. Fußend auf dieser Diskussion wurde im vierten Kapitel ein umfassendes Konzept erarbeitet und beschrieben. Im fünften Kapitel wurde die konkrete Umsetzung im Detail erläutert, um dann im sechsten Kapitel evaluiert zu werden.

Im Rahmen dieser Arbeit wurde festgestellt, dass das Generieren einer internen DSL aus annotiertem Java-Quelltext unter Verwendung eines allgemeinen Metamodells möglich ist. Zudem konnte gezeigt werden, dass eine Instanz dieses Metamodells geeignet ist, um wieder konfigurierten, konkreten Java-Quelltext zu erzeugen. Unter der Annahme, dass das Konzept vollständig für alle Features der Programmiersprache Java implementiert wird, könnte es möglich sein, den Implementationsaufwand eines Generators vollständig einzusparen.

Lediglich der Arbeitsaufwand für die Definition der Variabilitäten einer Referenzimplementation, also im weitesten Sinne eine Domänenanalyse, und die Konfiguration an sich, das heißt das Schreiben des DSL-Codes zur Definition konkreter Ausprägungen, konnte nicht reduziert werden. Im Rückblick auf das Analysekapitel zeigt Abbildung 7.1 noch

einmal zusammengefasst die vereinfachte Darstellung des Aufwandsvergleich von konventioneller Softwareentwicklung, modellgetriebene Softwareentwicklung und MDSD in Kombination mit dem in dieser Thesis erarbeiteten Konzept.

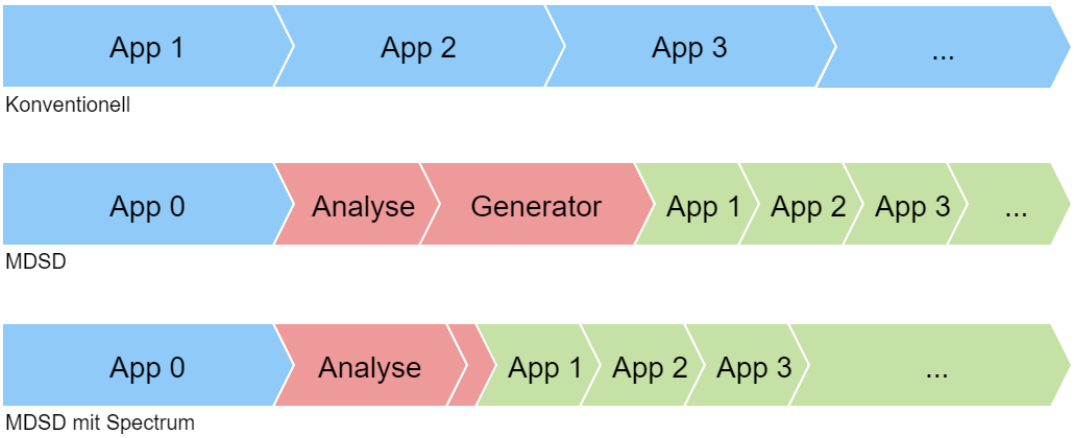

Abbildung 7.1: Zusammengefasste vereinfachte Darstellung des Einsparungspotentials von Spectrum im Vergleich zu konventioneller und modellgetriebener Softwareentwicklung.

Zusammenfassend kann die Aussage getroffen werden, dass eine mögliche Antwort auf die obige Frage gefunden wurde. Wie auch schon in der Evaluierung beschrieben, können die tatsächlichen wirtschaftlichen Auswirkungen von Spectrum zu diesem Zeitpunkt nur vermutet, jedoch noch nicht validiert werden.

7.2 Ausblick

Spectrum bietet eine Vielzahl möglicher Anknüpfungspunkte für zukünftige Arbeiten. An erster Stelle stünde die offensichtliche Erweiterung des Funktionsumfangs. Wie bei der Evaluierung bereits beschrieben wurde, fehlen noch fundamentale Features. Allein um vergleichsweise einfache POJOs vollständig beschreiben zu können, fehlt zum Beispiel der Support für Arrays und Feldern mit generischen Typparametern.

Um den Aufwand bei der Verwendung der erzeugten DSL weiter zu reduzieren, könnten zudem weitere abstrahierende Annotationen eingeführt werden. Die HasGetter- und HasSetter-Annotationen stellen hier nur den Anfang des machbaren dar. Man könnte beispielsweise eine IsSingleton-Annotation definieren, mit welcher eine Klasse annotiert würde. Bei der späteren Erzeugung dieser Klasse könnten automatisiert die fehlenden Bestandteile für das Singleton Pattern generiert werden. Aufwendiger, aber vermutlich ebenso machbar, wäre eine IsBuildable-Annotation für Felder. Bei Erzeugung eines so markierten Feldes, könnte eine zugehörige Builder-Methode, die zum Setzen des Feldes

verwendet werden kann und die Instanz des zu zugehörigen Klassenobjektes zurückgibt, generiert werden.

Zur Reduktion von Fehlern bei der Verwendung der Annotationen, könnte ein Werkzeug zur statischen Codeanalyse implementiert werden. Hiermit könnten Fehler, wie beispielsweise die Annotation eines static-Feldes mit einer HasSetter-Methode, frühzeitig erkannt werden.

Anstelle einer internen DSL könnte aus der annotierten Referenzimplementation eine externe DSL erzeugt werden. Hierdurch wäre es möglich, die auf die Syntax von Java zurück zu führenden Störelemente zu reduzieren.

Auch Jade und daraus folgend Scarlet könnten ausgetauscht werden. An deren Stelle könnte aus Instanzen des CodeUnit-Modells Quelltext anderer Programmiersprachen erzeugt werden. Somit könnte Spectrum möglicherweise als Crosscompiler Anwendung finden.

Allgemein stellt sich natürlich auch die Frage, ob es nicht angebracht wäre, mit den durch diese Arbeit gewonnenen Erkenntnissen, Spectrum von Grund auf neu zu implementieren. Auf diese Weise könnte ein stellenweise sicherlich notwendiges Refactoring und weitere Abstrahierung vermutlich sauberer erzielt werden.

Abseits der genannten Punkte wäre es auch denkbar, eine Reihe kleinerer Untersuchungen um Spectrum aufzubauen, um herauszufinden, was bereits jetzt alles mit Spectrum möglich ist oder wie es genau erweitert werden müsste, um eine bestimmte Aufgabe zu erfüllen. Zum Beispiel könnten die Pattern objektorientierter Programmierung von Gamma et al. als Referenzimplementation verwendet werden, um zu analysieren, wie diese zu annotieren wären, um daraus eine möglichst komfortable interne DSL zu erzeugen. Hierdurch würden auch weitere Limitationen des Konzepts oder Spectrums erkannt und dadurch potenziell behebbar.

Abbildungsverzeichnis

Listings

Literatur

[1] A. Aho u. a. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] C. Atkinson und T. Kühne. „Model-Driven Development: A Metamodeling Foundation“. In: *IEEE Softw.* 20.5 (Sep. 2003), S. 36–41. ISSN: 0740-7459.

[3] H. Balzert. *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*. 3., Aufl. Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-1705-3.

[4] J. Bloch. *Effective Java (3rd Edition)*. 3. Aufl. Addison-Wesley Professional, 2017. ISBN: 978-0134685991.

[5] T. Bray u. a. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, Recommendation REC-xml-20081126. 2008. URL: `http://www.w3.org/TR/2008/REC-xml-20081126`.

[6] K. Czarnecki und U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000. ISBN: 978-0-201-30977-5.

[7] M. Fowler. *Domain Specific Languages*. Addison-Wesley, 2011. ISBN: 978-0-321-71294-3.

[8] E. Gamma u. a. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[9] E. Gamma u. a. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. MITP-Verlags GmbH & Co. KG, 2015. ISBN: 9783826699047.

[10] S. Gao u. a. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, REC-xmlschema11-1-20120405. 2012. URL: `https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`.

[11] J. Herrington. *Code Generation in Action*. Manning Publications Co., 2003. ISBN: 1930110979.

[12] J. Hromkovič. *Theoretische Informatik: Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie*. Springer Fachmedien, 2014. ISBN: 978-3-658-06433-4.

[13] K. Kang u. a. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Pittsburgh, PA, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[14] B. Kernighan und D. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[15] T. Lindholm u. a. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 9780133905908.

[16] E. Ort und B. Mehta. *Java Architecture for XML Binding (JAXB)*. 2003. URL: http://www.oracle.com/technetwork/articles/javase/index-140168.html.

[17] o.V. *JavaPoet*. GitHub repository. Version dfba6d7272931991e56b0d0ac4b969c17cd3f334. 2017. URL: https://github.com/square/javapoet/.

[18] o.V. *OMG MDA Guide rev. 2.0*. 2014. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[19] o.V. *Unity User Manual (2017.3)*. Version 2017.3b-001X. Unity Technologies. Vienna, Austria, 2018. URL: https://docs.unity3d.com/Manual/index.html.

[20] o.V. *Zahlen & Fakten: Einkommen und Preise 1900 - 1999*. o.D. URL: https://usa.usembassy.de/etexts/his/e_g_prices1.htm (besucht am 16. 03. 2018).

[21] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 9781934356456.

[22] R. Prieto-Díaz. „Domain Analysis: An Introduction“. In: *SIGSOFT Softw. Eng. Notes* 15.2 (Apr. 1990), S. 47–54. ISSN: 0163-5948.

[23] J. Reichle. *100 Jahre Ford T-Modell: Schwarze Magie*. 2010. URL: http://www.sueddeutsche.de/auto/jahre-ford-t-modell-schwarze-magie-1.702183.

[24] G. Sager. *Erfindung des Ford Modell T: Der kleine Schwarze*. 2008. URL: http://www.spiegel.de/einestages/100-jahre-ford-modell-t-a-947930.html.

[25] B. Selic. „The Pragmatics of Model-Driven Development“. In: *IEEE Softw.* 20.5 (Sep. 2003), S. 19–25. ISSN: 0740-7459.

[26] N. Smith, D. van Bruggen und F. Tomassetti. *JavaParser: Visited - Analyse, transform and generate your Java code base*. 2017. URL: http://leanpub.com/javaparservisited.

[27] T. Stahl und M. Völter. *Modellgetriebene Softwarentwicklung: Techniken, Engineering, Management*. 2. Aufl. dpunkt.verlag, 2007. ISBN: 978-3-89864-881-3.

[28] M. Völter u. a. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0.

[29] W. Wallén. *The history of the industrial robot*. Techn. Ber. Division of Automatic Control at Linköpings universitet, 2008. URL: http://liu.diva-portal.org/smash/get/diva2:316930/FULLTEXT01.pdf.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

René Ziegler, am 19. März 2018

Zustimmung zur Plagiatsüberprüfung

Hiermit willige ich ein, dass zum Zwecke der Überprüfung auf Plagiate meine vorgelegte Arbeit in digitaler Form an PlagScan (www.plagscan.com) übermittelt und diese vorrübergehend (max. 5 Jahre) in der von PlagScan geführten Datenbank gespeichert wird, sowie persönliche Daten, die Teil dieser Arbeit sind, dort hinterlegt werden.

Die Einwilligung ist freiwillig. Ohne diese Einwilligung kann unter Entfernung aller persönlichen Angaben und Wahrung der urheberrechtlichen Vorgaben die Plagiatsüberprüfung nicht verhindert werden. Die Einwilligung zur Speicherung und Verwendung der persönlichen Daten kann jederzeit durch Erklärung gegenüber der Fakultät widerrufen werden.

René Ziegler, am 19. März 2018