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: 27. Februar 2018

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

Zusammenfassung

TODO

Abstract

TODO

Danksagung

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 [23].

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$, erreichte der Preis in den Jahren danach einen Tiefststand von 259$ [22]. 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 [19].

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 ineffizient. Der Markt für industrielle Roboter explodierte jedoch in den Folgejahren. Im Jahr 1984 waren weltweit ungefähr 100.000 Roboter im Einsatz [6, 27].

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 seien 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 Implementierung in Komponenten zerlegt werden muss oder eine für die Zwecke der Codegenerierung vorgesehene Referenzimplementierung geschrieben wird. Codeabschnitte, die in Ihrer Struktur gleich sind, sich jedoch inhaltlich unterscheiden, müssen formal beschrieben werden [25]. 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 Code Generatoren oder die Entwicklung eigener Code Generatoren vorteilhaft sein. Die Entwicklungsgeschwindigkeit könnte erhöht, die Softwarequalität gesteigert und Komplexität durch Abstraktion reduziert werden [25]. Allgemein wird weniger Zeit benötigt, um eine größere Vielfalt an ähnlichen Programmen zu entwickeln [6].

Bisher müssen fast alle Teilaufgaben bei der Umsetzung eines Code Generators manuell durchgeführt werden. Werkzeuge wie Language Workbenches können Code 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].

Soll ein Projekt Modellgetrieben entwickelt werden, so lohnt sich dies wirtschaftlich gesehen erst, wenn auf Basis des entwickelten Modells mehrere Programme entwickelt wurden []. Einer der Teilschritte dieses Entwicklungsprozesses ist die Planung und Implementation des Code Generators. Durch ihre hohe Komplexität, ist diese Aufgabe sehr zeitaufwendig [].

ÜBERARBEITEN

1.2. Zielsetzung

In dieser Arbeit soll untersucht werden, ob und wie die Entwicklung eines Codegenerators automatisiert werden kann. Eine zusätzliche Ebene der Indirektion könnte das komplexe Thema der Modellgetriebenen Softwareentwicklung weiter vereinfachen und somit Codegenerierung auch wirtschaftlicher machen.

Im speziellen wird analysiert, wie ein Metagenerator zur Erhöhung der Wirtschaftlichkeit modellgetriebener Softwareentwicklung umgesetzt werden könnte. Zu diesem Zweck wird 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, vollständig oder in Teilen, die Entwicklung anderer Softwaregeneratoren automatisiert.

1.3. Aufbau der Arbeit

Die sieben Kapitel dieser Bachelorarbeit versuchen den Leser Stück für Stück an das komplexe Thema der Metagenerierung heranzuführen. 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 modellgetriebene 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 Software Architektur 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 des Codegenerators untersucht.

Das Konzept Kapitel erläutert nun 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 jetzt 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 Quelltext Auszügen 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 aller Module im Detail. Außerdem wird die allgemeine Softwarequalität des entwickelten Generators nach bewährten Kriterien untersucht. Besonders umfangreich werden die Grenzen des Lösungsansatzes diskutiert und mögliche Antworten auf die hieraus entstehenden Fragen angesprochen.

Mit dem siebten Kapitel werden Inhalt und Erkenntnisse der Arbeit noch einmal in gebündelter Form zur Verfügung gestellt. 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 [25, S. 11]. Sie lässt sich gut erläutern wenn man sie in drei Teile zerlegt.

Zunächst einmal wäre dort der Ausdruck "formales Modell". Damit ist ein Modell gemeint welches einen Teil einer Software vollständig beschreibt. Jedoch soll das nicht heißen, dass dieses Modell allumfassend ist, sondern was genau von diesem Modell beschrieben wird muss eindeutig reguliert sein [25, S. 11f.]. Weiterhin bezieht sich die Definition darauf dass lauffähige Software erzeugt wird. D. h. 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 [25, 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 [25, S. 13].

In der Literatur findet sich auch die alternative Bezeichnung Model Driven Development (MDD) [24, 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 [17]. 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. Wobei Generative Programming nach Czarnecki und Eisenecker nach vollständiger Automation strebt und ein vollständiges Zwischen- oder Endprodukt erzeugen soll [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 [25, 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 [25, S. 13ff.]. Die Vorteile von Generative Programming [6, S. 13ff.] und MDA [17] 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 interesannt.

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 die Domäne sich aus dem übereinstimmenden Wissen der beteiligten Stakeholder ergibt. Wobei hier alle Personen die ein Interesse an einer bestimmten Domäne haben als Stakeholder bezeichnet werden [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 [21]:

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 [25, S. 239]. Danach werden im allgemeinen Entitäten, Operationen und Beziehungen zwischen diesen identifiziert. Sobald dies geschehen ist, können diese auf Ähnlichkeiten und Unterschiede versucht werden. Auf Basis der gefundenen Gemeinsamkeit bzw. 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.

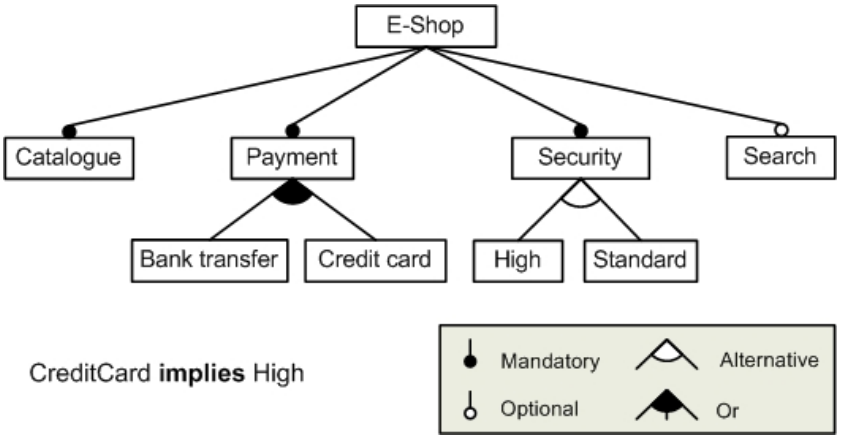

Abbildung 2.1.: Darstellung eines Featurediagramms für ein konfigurierbares E-Shop System von Segura09 in der Wikipedia auf Englisch (Transferred from en.wikipedia) [Public domain], via Wikimedia Commons.

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 darf bzw. dürfen. Zudem können noch weitere Einschränkungen definiert werden welche es erlauben auch weiter voneinander entfernte Knotenpunkte zueinander in Beziehung zu setzen [25, S. 240f.]. Die Abbildung 2.1 stellt oben genannte Features beispielhaft 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 [25, 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 Voelter bezeichnen es als Beschreibung 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 bzw. Modellierung [25, 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. Zu sagen ein Metamodell ist das Modell eines Modells, ist also nur bedingt sinnvoll [26, 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. [25, S. 59f.].

2.1.3. Domänenspezifische Sprache (DSL)

Wurde das Metamodell für eine Domäne definiert, also ist die abstrakte Syntax 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: [25, 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 die Ausdrücke 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.

DSLs können in zwei Hauptkategorien unterteilt werden, bezeichnet als interne und externe DSLs.

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]. Mehr als eine GPL gibt es, da sich einzelne Sprachen durch zusätzliche besondere Sprachfeatures voneinander abheben [26, 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 dynamisch zu bezieht 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 [25, 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 internal DSL zu API 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 oder auch Annotationen in Java 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 zum verarbeiten von Programmiersprachen seit Jahren verwendet werden, zugegriffen werden [7, S. 89].

2.1.3.5. Language Workbenches

Martin Fowler nennt bei der Kategorisierung von DSLs noch eine weitere, dritte Kategorie. Diese Language Workbenches sind hochspezialisierte Entwicklungsumgebungen zur Definition domainspezifischer 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 zuerst ein und meist wird dieser dann in einem ersten Schritt zeilenweise von einem Lexer in seine Bestandteile (Token) zerlegt. Diese aufgespaltenen Code Zeilen können dann auf syntaktische Korrektheit geprüft werden und in eine Übergangsrepräsentation überführt werden[20, 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 wichtigen Daten, angelegt [20, S. 23]. Die grafische Darstellung des AST für den euklidischen Algorithmus auf Abbildung 2.2 zeigt diese Knoten-basierte Strukturierung des eingelesenen Codes. Die Knoten eines AST kennen ihre eigene Position im Baum und somit kann dieser mit geeigneten Strategien durchlaufen werden [20, S. 24].

Eine Möglichkeit ist die Anwendung des Visitor Patterns, welches in dieser Arbeit in einem späteren Unterkapitel der Grundlagen 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 jetzt mehrere Möglichkeiten zur Verfügung.

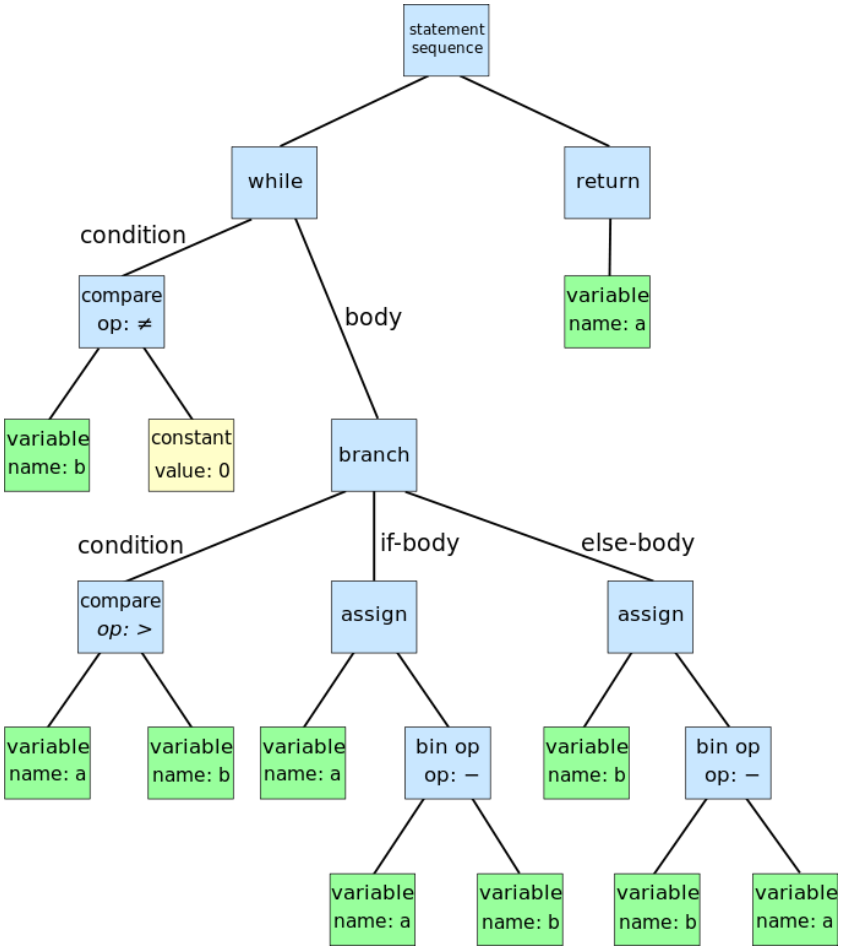

Abbildung 2.2.: Darstellung des euklidischen Algorithmus als AST von Dcoetzee (Eigenes Werk) [CC0], via Wikimedia Commons.

Die jetzt in Baumstruktur vorliegenden Anweisungen sind jetzt durch Interpretation von einer Plattform ausführbar. Alternativ könnte auch eine Übersetzung in eine andere Sprache vorgenommen oder direkt zu ausführbaren Maschinencode kompiliert werden.

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 [18]. Native GPLs wie C werden direkt zu Maschinensprache kompiliert [14]. Externe DSLs werden von einem Generator häufig zu GPL Quelltext transformiert [26, S. 26].

2.1.5. Code Generator

Soll der vom Parser eingelesene Quelltext oder das von einer DSL beschriebene Modell nicht interpretiert, d. h. als Anweisungen an den Computer ausgeführt werden, kann ein Code Generator 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 konkrete 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 denkbar, dass der Ausgabe 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 [25, S. 195]. Eine Komponente die diese Transformation durchführt nennt man Transformator. Wird ein Modell zu Text transformiert, so nennt man das Code Generierung, durchgeführt von einem Generator [26, 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 ins andere könnte durch mehrere Zwischentransformationen reduziert werden. Ein denkbarer Extremfall hierfür wäre die Reduktion der Transformationen auf nur eine einzelne Transformationsregel pro Modell Transformations Schritt.

2.1.5.4. Abgrenzung zu Compilern

In der Literatur wird ein Generator stellenweise auch Compiler [26, S. 26] bzw. Code generieren auch kompilieren genannt [7, S. 19].

Jedoch kann man bei genauerem hinsehen Generatoren von Compilern im Zusammenhang mit MDSD voneinander gut abgrenzen. 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 [26, 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 für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen. Zielorientiert bedeutet die Berücksichtigung z. B. von Kosten, Zeit, Qualität.

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

Für den Bereich der Prinzipien in der Softwaretechnik hat Balzert einige grundlegenden 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, d. h. 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 ist eine Form der Abstraktion der Aufbau von Vererbungsstrukturen in der objektorientierten Programmierung. 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 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. Die Haarfarbe mag relevant sein, also könnte man hier für die Haarfarbe rot oder blond abbilden, aber dies ginge noch genauer, man könnte noch Abstufungen unterscheiden. Vielleicht sollte man Haarfarbe an sich noch genauer festhalten, denn die Frage aus welchen rot, grün und blau Werten sich die Farbe genau zusammensetzt bleibt trotzdem noch offen. Abstraktion ist also irgendwann notwendig.

2.2.1.2. Modularisierung

Im direkten Zusammenhang mit Abstraktion steht das Prinzip der Modularisierung. 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, d. h. 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 verhin-

dert es, dass ein Softwaresystem eine chaotischer Struktur entwickelt [3, S. 39ff.]. Unter Geheimnisprinzip versteht man eine schärfere Version der Modularisierung, für ein 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 benötigt zum Beispiel Modularisierung bzw. Strukturierung das Geheimnisprinzip kann auch 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 Kapiten beschriebenen Prinzipien einen direkten Zusammenhang mit der Entwicklung von Codegeneratoren.

Insbesondere sticht die für die modellgetriebene Software Entwicklung unerlässliche Abstraktion durch Modellbildung heraus. Vor allem ein Metamodell stellt die Verallgemeinerung der relevanten Konzepte eine 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 Modul Metamodell. Ein Parser als weitere Komponente liest eigenständig Quelltext ein und bildet diesen auf einen AST ab. Mit einer Schnittstelle kann der eingelesen Quelltext weiterverarbeitet werden. Codegenerator ist hier unabhängig vom konkret gewählten Parser, er verarbeitet nur den AST und ein anderes aus dem Parsing Prozess 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 ein bestimmten Kontext lösen [8]. Diese Muster, von den Autoren des Werks Pattern genannt, haben grundsätzlich vier Elemente:

Zum einen 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. Zum anderen 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 beschrieben. Diese Lösungen beschreiben kein konkretes Design oder eine konkrete Implementation da das Muster in verschiedenen Situation Anwendung finden kann. Jedes Muster bringt Vor- und Nachteile, diese Konsequenzen sind wichtig zum Verständnis eines Patterns und sollten daher explizit aufgeführt werden [9, S.30 f.].

Modellgetriebene Software Entwicklung stellt uns im Zusammenhang mit objektorientierter Programmierung vor Probleme für welche 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. Erst wird das Pattern benannt, dann wird kurz der Zweck, also was mit der Verwendung des Patterns erreicht werden soll, zusammengefasst. Dies wird gefolgt von einer Erklärung 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 sinnvoll 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

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].

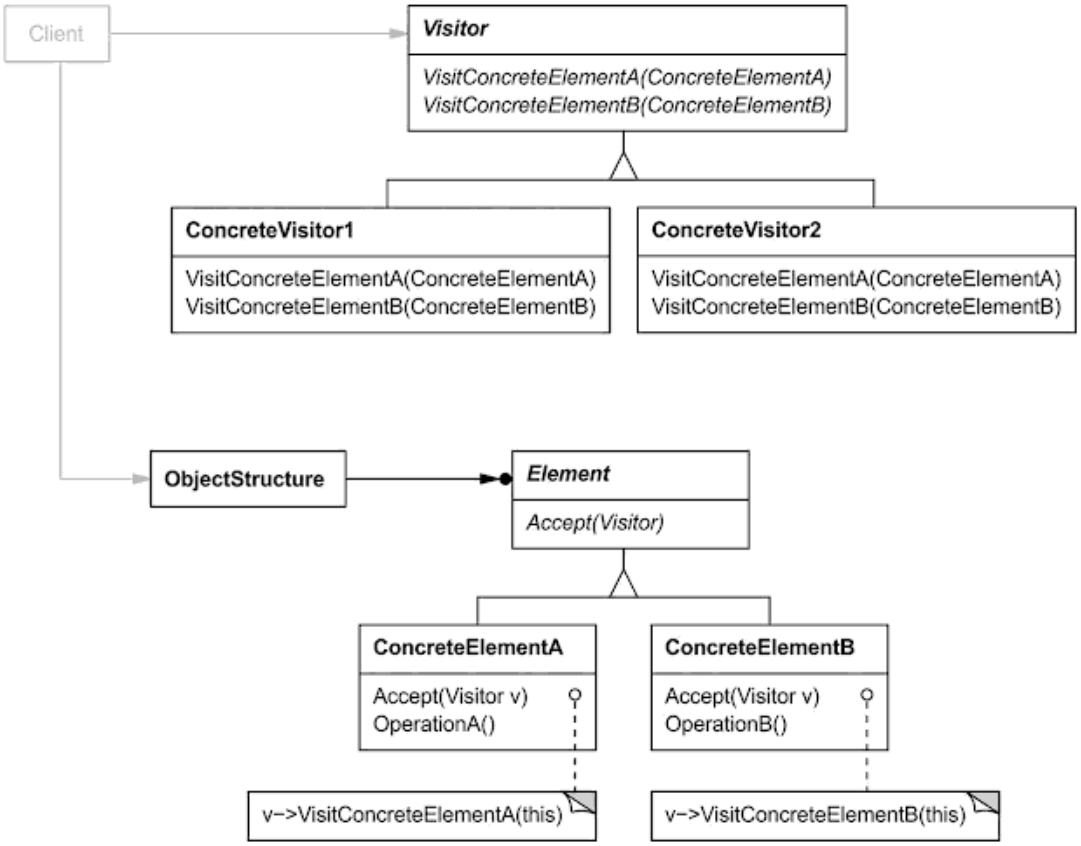

Abbildung 2.3.: Darstellung der Struktur des Visitor Design Patterns, entnommen aus [9, S. 485].

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 Abst rakte 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.

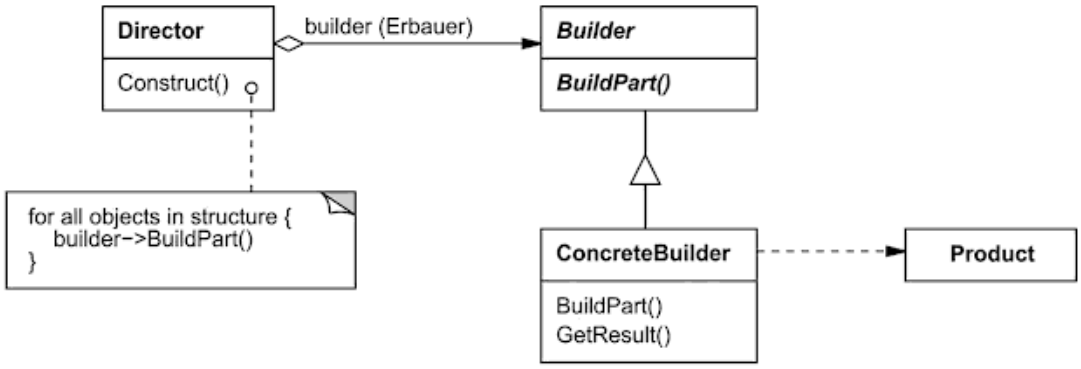

Abbildung 2.4.: Darstellung der Struktur des Builder Design Patterns, entnommen aus [9, S. 162].

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. Statische Fabrikmethoden können 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.

Eine Möglichkeit zur Erzeugung eines Generats ist die Zusammenstellung aus Komponenten. Man könnte statische Methoden zur Erstellung der jeweiligen Komponenten verwenden.

2.3.3.3. Struktur

Da es für die statische Fabrikmethode nach Bloch kein Gegenstück im Katalog Design Pattern 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 erweitern [4, S. 5f]. Die statische Fabrikmethode könnte auch in eine extra Fabrik Klasse ausgelagert werden um so Logik für die Instanziierung verschieden 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 Hauptfragestellung dieser Arbeit eingehend zu analysieren. Das folgende Kapitel wird zuerst auf einen Vergleich des Aufwands bei konventionellen Softwareprojekten und modellgetriebener Softwareentwicklung eingehen. Es soll klargemacht 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

Abhängig von der gewählten Methode für den Entwicklungsprozess, hat die Umsetzung einer Anwendung unterschiedliche Etappen. Zu Beginn der Entwicklung wird spezifiziert was entwickelt werden soll, also welche Anforderungen die Anwendung gestellt werden. Gefolgt von einer Entwurfsphase, in welcher beschrieben wird wie die Anwendung aussieht und aus welchen Bausteinen sie besteht. Als letzter Schritt wird das in Entwurfsphase erarbeitete Konzept implementiert [3, S. 62]. Abhängig vom verwendeten Prozess, wird parallel zu dieser Entwicklung oder im Anschluss daran die Qualität

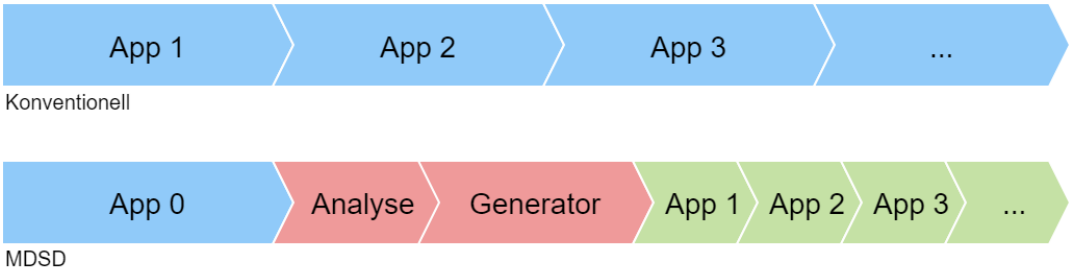

Abbildung 3.1.: Vereinfachte Darstellung des Aufwandsvergleichs eines konventionellen Softwareprojektes und MDSD.

der Anwendung durch Tests gesichert. Sind diese Etappen alle erledigt, kann die App veröffentlicht werden.

Die Idee der modellgetriebenen Software Entwicklung ist es, dass man vereinfacht und ab einem gewissen Punkt wirtschaftlich, ganze Anwendungen oder Teile derer einfacher, schneller und wiederverwertbar generieren kann [25, S. 14f.]. Ein Beispiel hierfür wäre die Umsetzung verschieden konfigurierter Ausprägungen einer Anwendung, basierend auf einer Softwarefamilie [25, S. 237ff.]. Wie aus den im vorherigen Kapitel beschriebenen Grundlagen hervorgeht, hat auch MDSD grundsätzlich mehrere Entwicklungsschritte. Um Software mit einem Modell beschreiben und daraus generieren zu können, muss bekannt sein wie diese Software aufgebaut ist und funktioniert. Ein erster Schritt bei MDSD kann also sein, eine Ursprungs App zu entwicklen, 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 entwickelt werden, zum anderen ein Generator, welcher aus einer Instanz des Metamodells verschiedene Ausprägungen der Software generieren kann.

Bereits für die Umsetzung einer Referenzimplementation muss der gleiche Prozess durchlaufen werden, wie der für das konventionelle umsetzen einer App [25, S. 219f.]. Da bei MDSD noch eine umfangreiche Analysephase und die Entwicklung des Generators durchgeführt werden muss, kann modellgetriebene Softwareentwicklung sich frühestens mit der Entwicklung einer zweiten App einer Softwarefamilie oder vielleicht sogar erst zu einem späteren Zeitpunkt wirtschaftlich rechnen. Abbildung 3.1 stellt diesen Vergleich grafisch vereinfacht dar.

Ein zusätzlicher Faktor der die Wirtschaftlichkeit von MDSD beeinträchtigen könnte ist, dass der Entwicklungsprozess bei der Verwendung eines modellgetriebenen Ansatzes ein Entwicklungsteam 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 konven-

tionelle Softwareentwicklung handelt, kann es zusätzlich sein, dass die Konzepte einem Entwicklungsteam erst nähergebracht und etwaige Kompetenzen erst aufgebaut werden müssen [26, 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.

Das umsetzen einer Referenzimplantation kann, wie bereits beschrieben, kaum gekürzt werden. Eine vollständige Analyse ist nur möglich wenn auch eine zu analysierende Anwendung exisitiert. Da wir mit dem Generator konkreten Quelltext erzeugen wollen, reicht auch eine reine Konzeptionierung der Referenzimplementation kaum aus, sie sollte codiert vorliegen. Es könnte möglich sein mithilfe von zum Beispiel Feature Modelling den großteil einer Domäne losgelöst von einer konkreten Implementation zu analysieren. Spätestens wenn es darum geht die zu generierende Architektur zu untersuchen, ist die Referenzimplementierung eine wichtige Hilfestellung und Ausgangspunkt hierfür [25, 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 Variabilität 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 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 [25, 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

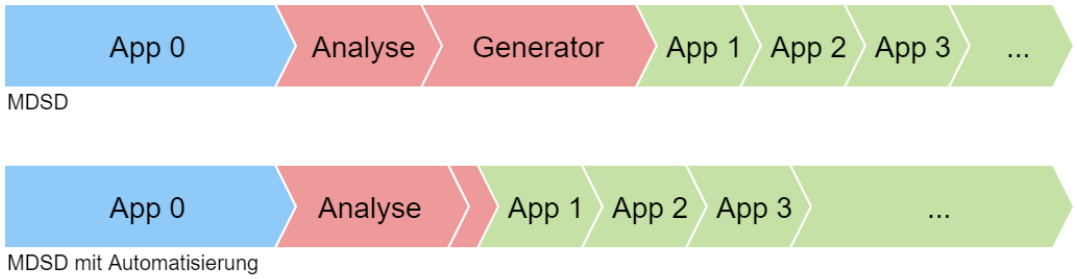

Abbildung 3.2.: Vereinfachte Darstellung des Einsparungspotentials im Generator Implementations-Schritt bei MDSD .

noch manuell schreiben.

3.2. Automatisierung der Entwicklung eines Code Generators

Wie aus dem vorherigen Absatz 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 Informationen vorliegen muss, dass in einer generierten 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. Eine Möglichkeit hierfür wäre dies in Form eines zusätzlichen Modells, zum Beispiel in einer XML Datei [5], zu tun. Hier würden dann unter der Verwendung

von XML Tags und Attributen Referenzen zu bestehenden Klassen gebildet. Beim einlesen des Java Quelltextes würde dann auch das zugehörige XML Dokument geparsed und nach korrespondierenden Informationen durchsucht werden. Mit XML Schema Definitions Dateien (XSD) [10] und Unterstützung von geeigneten APIs wie JAXB [16] könnte sehr genau beschrieben werden wie dieses Zusatzinformationsmodell aufgebaut sein dürfte. Alle Informationen zu Erweiterung einer Java Quelltext Datei 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. Der Informationsgehalt und dessen semantische Bedeutung müssten nach dem 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 wird 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 an Annotations Definitionen könnte noch genauer bestimmt werden an welcher Stelle im Code welche Annotation angebracht werden kann. Durch Annotation Parameter könnten noch weitere Informationen übergeben 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 parsen 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 parsen 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

3.2.2.1. Anforderungen an das Modell

3.2.2.2. Schnittstellen zur Befüllung des Modells

(DSLs intern / extern)

3.2.3. Generierung von Java Code

3.2.3.1. Techniken zur Code Generierung

3.2.3.2. Vorhandene Frameworks zur Java Code Generierung

4. Konzept

4.1. Einsparung der Implementation des Code Generators durch einen Metagenerator

4.2. Funktionsweise des Metagenerators

4.2.1. Von Java Code zum Annotations-Modell

4.2.1.1. Annotationen als Mittel zur Informationsanreicherung

4.2.1.2. Parsen des annotierten Codes

4.2.1.3. Zweck des Annotations-Modells

4.2.2. Das CodeUnit-Modell

4.2.2.1. Baumstruktur des Modells

4.2.2.2. Spezialisierung durch ein Typ-Feld

4.2.2.3. Parametrisierung durch generische Datenstruktur

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

4.2.3.1. Komposition der Builder aus benötigten Builder-Methoden

4.2.3.2. Übertagung vorgegebener Informationen in einen Builder

4.2.3.3. Verwendung von Plattform Code zur Generierung von vordefinierten CodeUnits

4.2.3.4. Auflösung von Referenzen in vordefinierten CodeUnits als nachgelagerter Verarbeitungsschritt

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

5. Lösung: Spectrum (Proof of Concept)

5.1. Verwendete Bibliotheken

5.2. Verwendeter Glossar

5.2.1. JavaParser mit JavaSymbolSolver

5.2.2. JavaPoet

5.3. Architekturübersicht

5.3.1. Amber

5.3.1.1. Annotationen

5.3.1.2. Parser

5.3.1.3. Modell

5.3.2. Cherry

5.3.2.1. Generator

5.3.2.2. Modell

5.3.2.3. Plattform

5.3.2.4. Generierte Builder

5.3.3. Jade

5.3.3.1. Transformator

5.3.4. Scarlet

Listing 5.1: Beispiel für einen Quelltext

```

public void foo() {
    // Kommentar
}
```

6. Evaluierung

6.1. Kozept & Implementation

6.1.1. Amber

6.1.2. Cherry

6.1.3. Jade

6.1.4. Scarlet

6.2. Softwarequalität

Nach Balzert S.111

6.2.1. Functionality

6.2.2. Maintainability

6.2.3. Performance

6.2.4. Usability

6.3. Grenzen des Lösungsansatzes

7. Abschluss

7.1. Zusammenfassung

7.2. Ausblick

A. Dokumentation

A.1. Verwendung der Annotationen

A.2. Verwendung der generierten CodeUnit-Builder

A.3. Klassendokumentation

A.3.1. Amber

A.3.2. Cherry

A.3.3. Jade

A.3.4. Scarlet

Abbildungsverzeichnis

Tabellenverzeichnis

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. URL: http://dx.doi.org/10.1109/MS.2003.1231149.

[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.

[6] K. Czarnecki und U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[7] M. Fowler. *Domain Specific Languages*. Addison-Wesley, 2011.

[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.

[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*. Techn. Ber. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 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: 013390590X, 9780133905908.

[16] E. Ort und B. Mehta. *Java Architecture for XML Binding (JAXB)*. 2003.

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

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

[19] o.V. *Zahlen & Fakten: Einkommen und Preise 1900 - 1999*. o.D. URL: https://usa.usembassy.de/etexts/his/e_g_prices1.htm.

[20] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 193435645X, 9781934356456.

[21] R. Prieto-Díaz. „Domain Analysis: An Introduction“. In: *SIGSOFT Softw. Eng. Notes* 15.2 (Apr. 1990), S. 47–54. ISSN: 0163-5948. URL: http://doi.acm.org/10.1145/382296.382703.

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

[23] 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.

[24] B. Selic. „The Pragmatics of Model-Driven Development“. In: *IEEE Softw.* 20.5 (Sep. 2003), S. 19–25. ISSN: 0740-7459. URL: http://dx.doi.org/10.1109/MS.2003.1231146.

[25] T. Stahl und M. Völter. *Modellgetriebene Softwarentwicklung: Techniken, Engineering, Management*. 2. Aufl. dpunkt.verlag, 2007.

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

[27] 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 27. Februar 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 27. Februar 2018