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. 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 [6].

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$, erreichte der Preis in den Jahren danach einen Tiefststand von 259$ [5]. 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 [4].

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 [1, 8].

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 [1].

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 [1].

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 [7]. 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 [2].

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 [7]. Allgemein wird weniger Zeit benötigt, um eine größere Vielfalt an ähnlichen Programmen zu entwickeln [1].

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 [2].

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 Code Generierung für mehr Entwickler zugänglicher und somit auch wirtschaftlicher machen.

Im speziellen wird analysiert, wie ein Meta-Generator zur Erhöhung der Wirtschaftlichkeit modellgetriebener Softwareentwicklung umgesetzt werden könnte. Zu diesem Zweck wird eine beispielhafte Java Anwendung erarbeitet, welche es ermöglichen soll, vorhandenen Java Quelltext so mit Informationen anzureichern, dass aus diesem ein Meta-Generator abgeleitet werden kann.

1.3. Aufbau der Arbeit

Die sieben Kapitel dieser Bachelorarbeit versuchen den Leser Stück für Stück an das komplexe Thema der Meta-Generierung 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 [7, 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 [7, 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 [7, 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 [7, S. 13].

Auch der Model Driven Architecture (MDA) Ansatz der Object Management Group (OMG) beschäftigt sich mit modellgetriebener Softwareentwicklung [3]. 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 ähnlich wie MDSD [1, S. 5].

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 [7, S. 13ff.]. Die Vorteile von Generative Programming [1, S. 13ff.] und MDA [3] werden sehr Ähnlich beschrieben.

2.1.1. Domäne

2.1.1.1. Definition

2.1.1.2. Domänenanalyse

2.1.1.3. Feature Modelling

2.1.2. Metamodell

2.1.2.1. Definition

2.1.2.2. Abstraktes Modell

2.1.2.3. Konkretes Modell

2.1.3. Domänenspezifische Sprache (DSL)

2.1.3.1. Definition

2.1.3.2. General Purpose Language (GPL)

2.1.3.3. Interne DSLs

2.1.3.4. Externe DSLs

2.1.3.5. Parser

2.1.4. Code Generator

2.1.4.1. Definition

2.1.4.2. Techniken zur Generierung von Code

2.1.4.3. Zusammenhang zu Transformatoren

2.1.4.4. Abgrenzung zum Compilerbau

2.2. Software Architektur

2.2.1. Prinzipien der Softwaretechnik

3. Analyse

3.0. Fragestellung: Wie kann, ausgehend von bestehendem Java-Code, ein Meta-Generator zur Erhöhung der Wirtschaftlichkeit Modellgetriebener Softwareentwicklung umgesetzt werden?

3.1. Aufwand eines konventionellen Softwareprojektes im Vergleich zu MDSD

3.2. Automatisierung der Entwicklung eines Code Generators

3.2.1. Java Code als Ausgangsmodell

3.2.1.1. Anreicherung des Java Codes mit Informationen

3.2.1.2. Parsen des Java Codes

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 Meta-Generator

4.2. Funktionsweise des Meta-Generators

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

```
1
2 public void foo() {
3     // Kommentar
4 }
```

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

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

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

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

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

[6] 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.

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

[8] 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. 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 19. Februar 2018