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 [5].

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$, erreichte der Preis in den Jahren danach einen Tiefststand von 259$ [4]. 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 [3].

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, 7].

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 [6]. 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 [6]. 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, nach welcher Struktur ein Generator aufgebaut sein muss, um diesen von einer Referenzimplementierung des erwünschten Generats abzuleiten. Zu diesem Zweck wird eine beispielhafte Java Anwendung erarbeitet, welche als Ausgabeprodukt den Code der ursprünglichen Implementierung repliziert. Für die Transformation der Referenzimplementierung wird vorerst nur auf einen Teil des Java 8 Sprachschatzes Rücksicht genommen.

Anschließend wird ermittelt, welche Schritte unternommen werden müssen, um die bestehende Referenzimplementierung in ein abstraktes Modell abzuleiten. Dabei geht es vor allem darum, wie der vorhandene Quelltext bearbeitet werden muss und in welcher Form das abstrakte Modell generiert werden sollte, um es als Eingabemodell für den erarbeiteten Generator zu verwenden. Hierfür wird die erste beispielhafte Anwendung weiterentwickelt, um die Generierung von Varianten der transformierten Referenzimplementierung zu ermöglichen.

1.3. Aufbau der Arbeit

2. Grundlagen

2.1. Modellgetriebene Softwareentwicklung (MDSD)

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

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

```

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

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

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

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

[5] 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.

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

[7] 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