HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Erstellung von Java-Programm-Traces für Programmieranfänger

Simon Müller

Konstanz, 17.08.2016

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Angewandte Informatik

Thema: **Erstellung von Java-Programm-Traces für Programmieranfänger**

Bachelorkandidat: Simon Müller, Neue Straße 4/1, 73732 Esslingen

1. Prüfer: Prof. Dr. Heiko Drachenfels
2. Prüfer: Prof. Dr. Oliver Bittel

Ausgabedatum: 17.05.2016
Abgabedatum: 17.08.2016

Zusammenfassung (Abstract)

Thema:	Erstellung von Java-Programm-Traces für Programmieranfänger
Bachelorkandidat:	Simon Müller
Firma:	HTWG
Betreuer:	Prof. Dr. Heiko Drachenfels Prof. Dr. Oliver Bittel
Abgabedatum:	17.08.2016
Schlagworte:	Java, ASM, JDI, Tracing, Debugging, Bytecode, Commandline tool

Ziel dieser Bachelorarbeit war die Erstellung eines Kommandozeilen-basierten tools zum suchen von Fehlerquellen in einfachen Java-Programmen für Programmieranfänger. Hierzu musste zunächst evaluiert werden welche Technologien hierfür vorhanden sind und welche von diesen am besten geeignet ist. Teil dieser Auswertung war auch die Definierung und Abgrenzung der Aufgaben für welche es eingesetzt werden sollte, um deren Machbarkeit ab zu schätzen. Auch die detailierte definierung der Ein- und Ausgabe waren Teil der Arbeit. Bei der Umsetzung wurde besonders auf Intuitive benutzbarkeit und Erweiterbarkeit geachtet, um den einstieg für Programmieranfänger so einfach wie mäglich zu gestalten.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Simon Müller, geboren am 04.05.1993 in Esslingen*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Erstellung von Java-Programm-Traces für Programmieranfänger

bei der HTWG unter Anleitung von Prof. Dr. Heiko Drachenfels selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 17.08.2016

(Unterschrift)

Inhaltsverzeichnis

Abkürzungsverzeichnis

JDI Java Debug Interface

IDE Integrierte Enwicklungsumgebung, engl. Integrated Development Enviroment

DDD Data Display Debuger

1 Einleitung

Für fundiertes Verständnis von Java und um ein Grundlagen-orientiertes lernen zu ermöglichen hilft es anfänglich nur mit einem normalen Texteditor Code zu schreiben und diesen dann selbst mittels Konsole zu kompilieren. Mit diesem Ansatz dauert es möglicherweise etwas länger bis erster ausführbarer Code entsteht, jedoch ist der Lernerfolg größer da alle Schritte dorthin selbst Ausgeführt wurden und damit transparent sind.

Da gerade Anfängern es schwer fällt die Ursache von auftretenden Problemen bei der Ausführung zu finden, welche nicht bereits vom Compiler erkannt werden können, kann dies jedoch schnell mühsam werden. Selbst simple Probleme, wie eine falsch definierte Bedingung in einer Schleife oder Verzweigung, stellen hier ein ernstes Hindernis dar.

Hierbei soll „Klara“ zum Einsatz kommen. Es hilft dabei, den Ablauf des Programms zu verstehen, ohne den Code dafür ändern zu müssen oder eine schwergewichtige Integrierte Enwicklungsumgebung, engl. Integrated Development Enviroment (IDE) zu verwenden.

In dieser Thesis wird dazu zuerst einmal auf die Problemstellung eingegangen. Hierbei wird beleuchtet, warum die Entwicklung nötig war und welche Rolle das Programm einnehmen soll. Im zweiten Kapitel beschäftigt sich diese Arbeit mit der Analyse von Umgebung und Anforderungen. Hier wird auch darauf eingegangen was man genau unter tracing versteht und welche Einsatz-Szenarien möglich sind. Im darauffolgenden Abschnitt werden die 3 wichtigen technischen Grundlagen genauer dargelegt: Java-Bytecode, ASM und Java Debug Interface (JDI). Da somit die Rahmenbedingungen der Arbeit klar sein sollten, wird in Kapitel 4 die Umsetzung genau erklärt.

Die wichtigen Aspekte hierbei sind:

- Das Konsolen-Interface, über das der Nutzer die Anwendung bedient
- Der individuelle ClassLoader, welcher eine Manipulation des Ausgeführten Codes ermöglicht
- Der Transformer, welcher modulare Änderungen ausführen kann
- Die Lizenzierung, um Konform zu der Lizenz des verwendeten Framwork zu bleiben

Schlussendlich wird in einem Fazit das Ergebnis der Arbeit reflektiert, ein Ausblick auf mögliche weitere Zukünftige Funktionen gegeben und es werden andere potenzielle Anwendungsbereiche für diese Technologie dargelegt.

1.1 Problemstellung

Im Java Umfeld gibt es eine Reihe von Tools zum debuggen von Anwendungen, jedoch sind diese alles große und meist komplexe Programme. Zudem sind diese auch häufig Teil einer IDE und nicht ohne diese einsetzbar. In anderen Programmiersprachen, wie beispielsweise C, gibt es hierzu große und mächtige Hilfsmittel, als wahrscheinlich bekanntesten der Data Display Debuger (DDD). Auch modernere Sprachen wie beispielsweise Python bieten hierfür Programme.

2 Analyse

2.1 Tracing

2.2 Ähnliche Anwendungen

2.3 Anforderungen

2.4 Idee

2.5 Szenario

3 Technische Grundlagen

3.1 Java Bytecode

3.2 ASM

3.3 JDI

4 Umsetzung

In diesem Kapitel wird detailliert die Umsetzung der in Kapitel 2.3 dargelegten Anforderungen an die Software beschrieben. Hierbei werden auch verschiedene andere mögliche Umsetzungen vorgestellt und erklärt warum diese nicht zum Einsatz gekommen sind.

Wichtig ist hierbei auch der letzte Punkt der Lizenzierung, da diese konform zu der Lizenz des verwendeten Frameworks „ASM“ sein musste.

4.1 Konsolen Interface

Bei der Realisierung des Konsolen Interface war besonders die intuitive Benutzbarkeit und Übersichtlichkeit wichtig. Sowohl bei der Wahl der Parameter für die Ausführung, als auch bei der erzeugten Ausgabe.

Das intuitivste Bedienkonzept für eine Konsolen-Anwendung ist der Aufruf mit Parametern. Durch diese werden alle Optionen gesetzt und das Programm benötigt keine Interaktion während der Ausführung. Der Nachteil ist allerdings ganz offensichtlich: Der Nutzer muss zum starten des Programms zuerst die Liste der verfügbaren Parameter und deren Bedeutung kennen. Zudem können Aufrufe sehr schnell sehr lang werden, was die Übersichtlichkeit einschränkt.

Trotz dieser Nachteile ist die Möglichkeit eines parametrisierten Aufrufs unverzichtbar für eine Programm mit ausschließlich textueller Oberfläche. Für die Implementierung existieren in Java einige Bibliotheken und Framerworks, wie Beispielsweise:

- Commons CLI
- Java Gems
- JArgs
- GetOpt
- Args4J
- JCommando

Allerdings haben alle den Nachteil, das sie nur „übliche“ Formate unterstützen. Für besondere Parameterformate muss eine eigene verarbeiten erfolgen, welche

auch zusätzlich den Vorteil bietet, dass somit keine weiteren Abhängigkeiten der Software geschaffen werden.

Die eigene Implementierung dieser Parameterverarbeitung erfolgt Schleifen-basiert. Über einen if-else-Baum wird jeder einzelne Parameter verarbeitet. Da sich die Verarbeitung im Variablenkontext der Schleife befindet, kann diese die Parameterliste durchsuchen und, wenn benötigt, beliebig viele weitere Parameter, die als Argumente dienen, verwenden. Auch die Abbruchbedingung für das verarbeiten von Parametern kann frei gewählt werden. So könnte Beispielsweise eine End-Option eingeführt werden, welche das verarbeiten der Parameter beendet und die verbleibenden an das Haupt-Programm weiter leitet. In der Gewählten Implementierung bricht die Verarbeitung ab sobald ein Parameter gefunden wird der nicht mit einem '-' beginnt. Dieser wird als Identifikator für das zu ladende Hauptprogramm interpretiert. Sollten weitere Parameter vorhanden sein werden diese als „args“ weitergegeben. Somit ist eine klare Trennung der zu verarbeitenden und der weiter zu gebenden Parametern vorhanden.

Für die gesamte Implementierung gilt das bei invaliden Parameter-Kombinationen oder fehlenden Parametern eine entsprechende Fehlermeldung ausgegeben wird, gefolgt von der Hilfe wie in Listing 4.1 dargestellt und das Programm dann mit einem Fehlercode abbricht. Alle Fehlercodes sind als Konstanten deklariert um eine Nachvollziehbarkeit von Code zu Ursache zu erzeugen.

Listing 4.1: Hilfe Ausgabe mit Parametererklärung

```
java -jar Klara.jar -i
```

Weitere Mögliche Bedienkonzepte sind ein navigierbares Menü und ein interaktiver Modus.

Bei dem navigierbaren Menü könnte man dann mit Hilfe der einzelnen Optionen eine Konfiguration erstellen, diese potenziell abspeichern und damit starten. Dieses Konzept ist besonders für komplexe Einstellungen geeignet, da es die Möglichkeit bietet, in einer übersichtlichen Form eine reihe von Optionen zu wählen. Dies kommt jedoch mit dem Nachteil, das selbst einfache Konfigurationen vergleichsweise lange zum erstellen benötigen. Zudem ist ein Menü in der Implementierung aufwändig.

Ein interaktiver Modus hingegen ist die Einsteiger freundlichste Methode. Hierbei wird durch eine Reihe von Fragen der User nach dem gewünschten Einstellungen gefragt, in einer geordneten Reihenfolge und mit der Möglichkeit für ausführlichere Erklärungen der Option. Auch hier ist ein klarer Nachteil das einfache

Konfigurationen, im Vergleich zu einem Parametrisierten-Start, länger zum starten benötigen. Der große Vorteil ist jedoch, das auch unerfahrene Nutzer ohne

4.2 Individueller Class-Loader

4.3 Transformer

4.3.1 Event Struktur

4.3.2 Erstellte Implementierungen

Referenz aufs Codebeispiel 4.2.

Listing 4.2: Codebeispiel Java

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

4.4 Lizenzierung

ASM Lizens bla blub [asm, 2016]

Also alles easy, geht fast jede Lizenz wenn man LIZENZ.txt in die jar packt.

5 Fazit

5.1 Erweiterungsmöglichkeiten

Umbau der gewrappten Variablen das diese tiefe Änderungen tracken können

Ausgabe des Branchings anstelle Zeilen (-b)

Invisible Asserts (Fehler-case definieren und passend konfigurieren)

Auto-Detect of relevant classes (Not just the class containing main)

Support for jars?

Menü zum konfigurieren, ähnlich interactive, siehe 4.1

Speichern von Konfigurationen erstellt mit Menü zum einfachen wiederholen von Debug-vorgängen.

5.2 Weitere Anwendungsbereiche

Heavy Obfuscation after compile (goto and stuff)

Invisible Code generators

„Makro“ calls (Replace „method“ class with specific code)

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literaturverzeichnis

[asm, 2016] (2016). Asm license.