HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Erstellung von Java-Programm-Traces für Programmieranfänger

Simon Müller

Konstanz, 24.08.2016

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: 24.08.2016

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:	24.08.2016
Schlagworte:	Java, ASM, JDI, Tracing, Debugging, Bytecode, Commandline tool, Klara

Ziel dieser Bachelorarbeit war die Erstellung eines Kommandozeilen-basierten Tools zur Suche von Fehlerquellen in einfachen Java-Programmen für Programmieranfänger. Hierzu wurde zunächst evaluiert, 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 abzuschätzen. 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 möglich 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, 24.08.2016

(Unterschrift)

Inhaltsverzeichnis

Abkürzungsverzeichnis

API Programmierschnittstelle, engl. Application Programming Interface

BSD Berkeley Software Distribution

DDD Data Display Debuger

EUPL European Public License

GPL GNU General Public License

IDE Integrierte Enwicklungsumgebung, engl. Integrated Development Enviroment

JDI Java Debug Interface

JDK Java Development Kit

JPDA Java Platform Debugger Architecture

JRE Java Runtime Enviroment

JVM Java Virtual Machine

LGPL GNU Lesser General Public License

PHP PHP: Hypertext Preprocessor

VM Virtuelle Maschine

1 Einleitung

Um ein fundiertes Verständnis von Java zu erreichen und um ein grundlagenorientiertes 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 der erste ausführbare Code entsteht, jedoch ist der Lernerfolg größer, da alle Schritte dorthin selbst ausgeführt wurden und damit transparent sind.

Da es gerade Anfängern schwerfällt, die Ursache von Problemen 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: 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 Framework 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 die verwendete Technologie dargelegt.

1.1 Problemstellung

Im Java Umfeld gibt es eine Reihe von Tools zum debuggen von Anwendungen, jedoch sind es 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 auch große und mächtige Hilfsmittel auf der Kommandozeile. Das wahrscheinlich bekannteste ist der Data Display Debuger (DDD). Bei moderneren Programmiersprachen ist dies jedoch zunehmend verloren gegangen, da heute nahezu jeder mit einer IDE arbeitet, welche in der Regel einen grafischen Debugger mit sich bringt.

Viele Anfänger benutzen deshalb von Anfang an eine IDE, um bei Problemen auf den Debugger der Umgebung zurückgreifen zu können, jedoch ist dies nicht zielführend. Zum einen sind die integrierten Debugger meist sehr komplex, was Anfängern auch das Finden einfacher Fehler schwer macht. Zum anderen kommt die Umgebung mit zahlreichen anderen Features, welche viele Schritte kombinieren und Anfängern nicht ersichtlich machen, was genau passiert.

Die einzige alternative dazu ist das einfügen von einer Vielzahl an Text-Ausgaben um den Verlauf des Programmes nach zu vollziehen. Dies ist aber eine mühselige Aufgabe, da sie für jedes Problem eingefügt werden müssen und sobald dieses gefunden wurde direkt wieder entfernt. Diese ist unnötige Schreibarbeit die kaum Lernerfolg bietet und automatisiert werden kann.

2 Analyse

Dieses Kapitel beschäftigt sich mit der Analyse, was genau Tracing ist und welche Anwendungen es bereits in diesem Gebiet gibt. Außerdem werden Anforderungen dargelegt und durch Szenarien weiter erläutert.

2.1 Tracing

Tracing beschreibt eine spezielle Form des Loggings, bei der Daten gesammelt werden, um den Ablauf eines Programms rekonstruieren zu können. Dies ermöglicht im Fehlerfall den Entwicklern nachvollziehen zu können, wo und wieso welches Problem aufgetreten ist. Die erzeugten Daten beim tracing werden als Programmtrace - oder kurz Trace - bezeichnet, was sich ins Deutsche als „Spur“ übersetzen lässt. Tracing stellt die umfassendste Art des Loggings dar, da bis ins kleinste Detail die Ausführung protokolliert wird. Aus diesem Grund ist sie im produktiven Betrieb auch ungeeignet, da Massen an Daten erzeugt werden.

Ein Vorteil im Vergleich zum herkömmlichen Debugging, bei dem die Ausführung angehalten wird, ist dass der Einfluss auf die Thread-Synchronisierung minimal gehalten wird. Die Ausgaben haben nur einen geringen Einfluss auf einzelne Ausführungszeiten. Der Nachteil ist wiederum, dass Programme meist erst analysiert werden, wenn deren Ausführung bereits beendet ist. Somit sind Reaktionen auf die spezifischen Zustände schwerer realisierbar.

Diese Technik ist auch einsetzbar, um Programmieranfängern den Ablauf ihrer Programme zu vermitteln, besonders wenn der textuelle Trace von grafischen Anwendungen visualisiert wird. Diese können damit ohne viel Aufwand oder Kenntnis der Werkzeuge selbständig Programmfehler nachvollziehen und nach Lösungen suchen.

2.2 Ähnliche Anwendungen

Zu Anwendungen, die solche Visualisierung bereits umgesetzt haben, zählen „Jeliot 3“ für Java und „Python Tutor“ für Python, Java, C und C++.

Jeliot ist eine Software, mit der lokale Programme analysiert werden können. Jedoch hat das Projekt seit 2007 keine Updates erhalten und ist deshalb mit Java 1.4 entwickelt worden. Moderne Programmierkonstrukte, die erst mit späteren Versionen eingeführt wurden, werden deshalb nicht unterstützt. Da das Programm zuerst ohne Interaktionsmöglichkeit ausgeführt wird, sind keine Eingaben möglich. Der gesamte Code muss in einem File sein um ausgeführt zu werden. Sind alle diese Randbedingungen erfüllt, kann damit die Ausführung schrittweise untersucht werden. Dabei werden Konstanten, der aktuelle Stack-Trace sowie Details aller Variablen und Methoden-Aufrufe visualisiert. Jeliot legt hier besonders Wert auf die Darstellung von Vererbungsstrukturen.

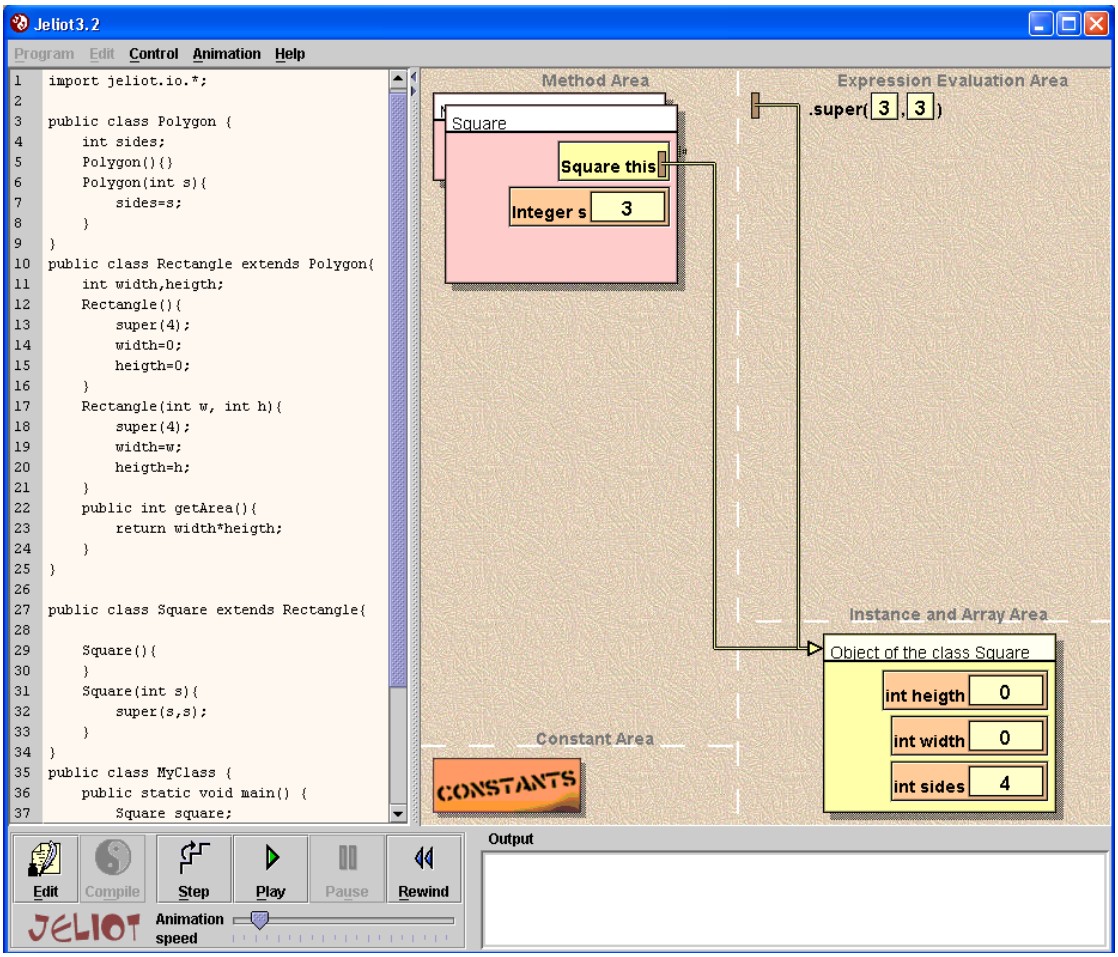

Abbildung 2.1: Oberfläche von Jeliot 3 (Quelle: http://cs.joensuu.fi/jeliot/images/jeliot3.2.png)

Python Tutor ist ein Online-Tool, welches Programme in verschiedenen Sprachen visualisieren kann. Bei Java gibt es dabei ähnliche Einschränkungen wie bei Jeliot:

- Die Ausführung kann nur mit Java Version 1.8 erfolgen
- Es ist nicht möglich, Parameter an das Programm zu übergeben oder interaktive Eingaben zu machen
- Der Code muss in einem einzelnen Textfeld eingegeben werden
- Die maximale Ausführungszeit ist auf 10 Sekunden limitiert
- Es können nur die auf dem Server verfügbaren Bibliotheken verwendet werden

Auch hier erfolgt die Anzeige in einem übersichtlichen Format, mit Visualisierung von Zeigern und den Methoden auf dem Stack. Wichtig zu beachten ist jedoch, dass die Ausführung sehr langsam erfolgt. Selbst einfache Programme benötigen meist über eine Sekunde zur Ausführung.

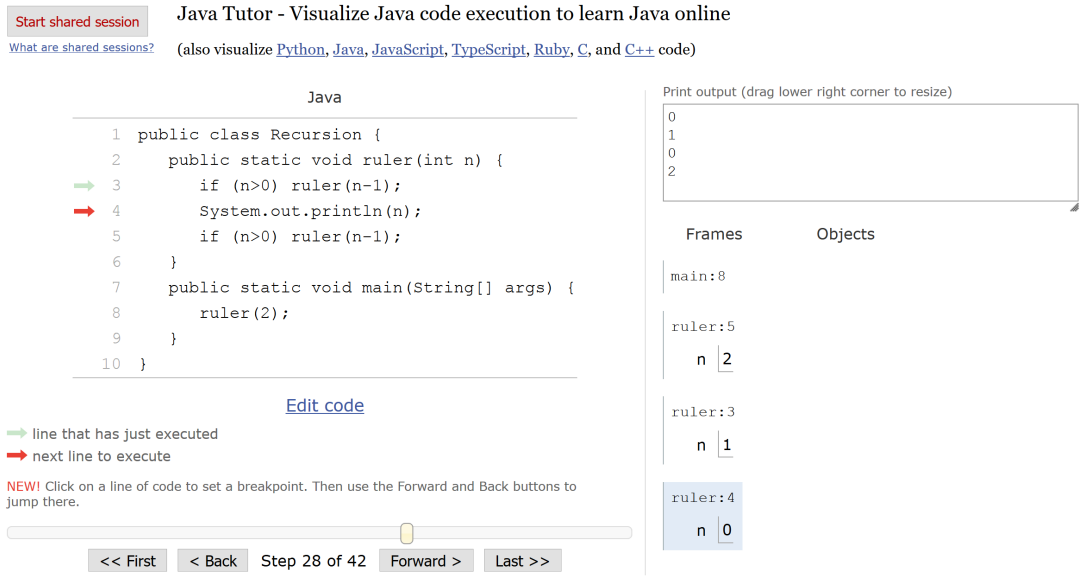

Abbildung 2.2: Oberfläche von Python Tutor (Quelle: eigener Screenshot)

2.3 Anforderungen

Ziel dieser Arbeit ist jedoch nicht eine visuelle Darstellung, sondern lediglich die Möglichkeit einer textuellen Ausgabe für simple Programme, wie sie ein Programmieranfänger entwickelt. Die wichtigsten Anforderungen sind wie folgt definiert:

- Erstellungen eines textuellen, menschlich lesbaren Programmtrace
- Keine Änderungen am Quelltext notwendig
- Kommandozeilen basiertes Interface
- Intuitive Benutzbarkeit
- Auswahlmöglichkeit von verschiedenen Optionen zur Erstellung des Trace
- Leichtgewichtig - keine Installation notwendig
- Erweiterbares Design, in das weitere Funktionen integriert werden können

Alle diese Anforderungen zielen auf ein simples Werkzeug für Anfänger ab, welches einfach zu benutzen ist, gleichzeitig aber Potenzial für komplexere Erweiterungen bietet.

2.4 Szenario

Für einen möglichen Anwendungsfall nehmen wir das Programm in Listing 2.1 an. Wie einfach zu erkennen ist, wurde bei der Bedingung ein falscher Vergleichsoperator verwendet, weshalb „7“ anstelle von „10“ ausgegeben wird.

Listing 2.1: Anfänger Programm mit Problem in Schleife

```
public class Example {
        public static void main(String[] args) {
                int r = 7;
                for (int i = 0; i > 3; i++)
                        r++;
                System.out.println(r);
        }
}
```

Hier sollte ein einfacher Aufruf an Klara genügen, um angezeigt bekommen zu können, dass nur die Zeilen 3, 4 und 6 ausgeführt werden. Dies sollte als Information genügen, um das Problem zu lokalisieren.

Ein anderes Beispiel ist in Listing 2.2 zu sehen. Hier wird eine Variable zugewiesen und dann geprüft. Jedoch erfolgt dies nicht identisch, als würde die gesamte Rechnung in einem Schritt ausgeführt, weshalb das Ergebnis 50 und nicht 22 ist.

Listing 2.2: Anfänger Programm mit Problem bei Variable

```
public class Example {
        public static void main(String[] args) {
                int r = 5;
                r *= 3 + 7;
                if (r == 22)
                        System.out.println("Erfolg");
                else
                        System.out.println("Fehler");
        }
}
```

Hier sollte von Klara ausgegeben werden, das der Variable `r` zuerst „5“ und dann „50“ zugewiesen wird. Somit sollte für einen Entwickler nachvollziehbar sein, dass direkt eine Multiplikation mit 10 stattfindet und somit ein unterschied zu `r = r * 3 + 7` besteht.

3 Technische Grundlagen

In diesem Kapitel werden die 3 wichtigen technischen Grundlagen für diese Arbeit beschrieben: Javas Bytecode, das ASM-Framework und das JDI. Bytecode ist der „übersetzte“ Java-Code. ASM bietet die Möglichkeit, diesen zu analysieren und manipulieren. Mit JDI lässt sich eine Java Virtual Machine (JVM) erstellen, von der Laufzeitinformationen ausgelesen werden können. Sowohl ASM als auch JDI bieten somit die Möglichkeit, den Ablauf eines Programms zu analysieren. Zusätzlich wird auch ein alternatives Framework zu ASM vorgestellt und beschrieben warum dieses nicht geeignet ist.

3.1 Java Bytecode

Java Bytecode ist sogenannter „intermidiate Code“, eine Maschinencode-ähnliche Form, welche jedoch nicht direkt auf einem Prozessor ausgeführt werden kann. Eine andere Programmiersprache, die diese Technik verwendet, ist C#.

Der Vorteil dieses Formats ist, dass es sich dabei bereits um eine kompaktere Form des ursprünglichen Codes handelt, welches trotzdem plattformunabhängig ist. Somit lässt sich der selbe Code auf jedem beliebigen Betriebssystem und unabhängig der Prozessorarchitektur ausführen, solange diese die passende Ausführungseinheit besitzt. Bei Java ist dies die JVM, welche Bytecode ausführen kann, bei C# ist es die .NET-Laufzeitumgebung.

Auch mehrere anderer Sprachen, wie PHP: Hypertext Preprocessor (PHP) und Python, verwenden Bytecode, obwohl sie offensichtlich den direkt interpretierten Code auszuführen scheinen. Jedoch bietet Python beispielsweise die Möglichkeit, auch den Bytecode in eine „pyc“-Datei zu speichern.

Da Bytecode eine sehr exakte Repräsentation des originalen Codes darstellt, lässt sich dieser auch wieder daraus gewinnen. Mit verschiedenen existierenden Werkzeugen ist es so möglich, den Code eines Java oder C# Programms zu erzeugen, obwohl nur der Bytecode zur Verfügung gestellt wurde. Gerade bei kommerziellen Anwendungen ist dies meistens nicht gewünscht, jedoch schwer vermeidbar.

Dies bietet jedoch auch den Vorteil, dass Bytecode sehr einfach manipuliert werden kann, nachdem er erzeugt wurde. Dies ermöglicht Änderungen, welche erst zur Laufzeit sichtbar werden, ohne den originalen Code zu kennen oder zu verändern. Auch dafür existieren Hilfsmittel, mit denen solche Manipulationen vereinfacht werden. In Java sind hierbei die beiden größten Frameworks ASM und Javassist.

3.2 ASM und Javasisst

ASM und Javasisst bieten beide Möglichkeiten, um Bytecode zu analysieren und zu manipulieren. Der wichtigste Unterschied hierbei ist jedoch die Abstraktionsebene: ASM bietet eine Assembler ähnliche direkte Darstellung, Javasisst eine am Original-Code angenäherte Form. Da natürlich beide Vorteile und Nachteile bieten, geht dieser Abschnitt genauer darauf ein und begründet die Entscheidung, welche Technik für dieses Projekt besser geeignet ist.

ASM

Der Name ASM ist keine Abkürzung im eigentlichen Sinn, sondern kommt vom dem Befehl „asm“, mit welchem sich in C direkt Assembler-Befehle ausführen lassen. Dieser Befehl wiederum ist eine Abkürzung für „Assembler“. Die erste Version wurde 2002 von Eric Bruneton veröffentlicht und seitdem weiterentwickelt, mit der zum Zeitpunkt dieser Arbeit neuesten Version veröffentlicht im März 2016.

Wie der Namensursprung vermuten lässt, ist das Framework für Manipulationen auf Bytecode-Ebene ausgelegt, welche in einer Assambler-ähnlichen Form erfolgen. Zusätzlich dazu bietet es auch die Möglichkeit, den existierenden Code zu analysieren und so dynamisch existierende Klassen zu verändern. Für die Analyse und Manipulation gibt es hier zwei verschiedene Möglichkeiten: Eine einfache Event-Chain-Struktur, welche hohe Performance bietet, aber deren Ablauf unveränderlich ist und eine Baum-Struktur, die langsamer ist, aber eine ungeordnete Modifikation erlaubt. Je nach Einsatzzweck ist es auch möglich, beide Techniken zu kombinieren, da sich eine Baum-Transformation in eine Event-Chain integrieren lässt. So sind komplexe Analysen und Transformationen möglich, sowohl statisch als auch dynamisch.

Der Nachteil dieses Frameworks ist der erhöhte Aufwand, sich mit der Bytecode-Notation vertraut zu machen, welche hier die einzige Form ist, neuen Code ein-

zufügen. Es ist nicht nötig, jedes Detail selbst als Bytecode schreiben zu können, jedoch sollte ein gutes Verständnis der Abläufe vorhanden sein. Wenn ein komplexerer Code erzeugt werden soll, so kann dieser in Java geschrieben, compiliert und dann mit ASM analysiert werden, um den dafür benötigten Bytecode zu finden. Die äußerst ausführliche Dokumentation erleichtert zudem den Einstieg gewaltig, da sie alle nötigen Details beschreibt.

Javasisst

Der Name Javasisst steht für „Java Programming Assistant“. Das Framework existiert seit 1999 und wird in Japan von Shigeru Chiba entwickelt. Der Fokus liegt bei einer Manipulation auf Sourcecode-Ebene, obwohl es auch Werkzeuge zur Bearbeitung auf Bytecode-Ebene liefert.

Der Vorteil hierbei liegt eindeutig bei der einfachen Benutzung, da Java-Code eingefügt werden kann und der Entwickler sich somit nur wenig mit Bytecode beschäftigen muss. Dies erlaubt einen schnellen Einstieg und die Möglichkeit, auch einen komplexen Code ohne großen Aufwand einzufügen.

Der Nachteil ist jedoch ähnlich offensichtlich: Dadurch, dass der Fokus der Entwicklung nicht auf dem Bytecode lag, sind die Möglichkeiten, vorhandenen Code zu analysieren, sehr beschränkt. Es kommt durchaus mit Werkzeugen dazu, jedoch nur in einer abstrakten Form, ohne Möglichkeit detaillierter zu scannen. Hier sind hauptsächlich Aussagen über den Speicher möglich, sowie über wenige grundsätzliche Kontrollflüsse. Dies hat natürlich nur bei dynamischen Transformationen Einfluss, statische sind hierbei nicht beeinträchtigt.

Entscheidung

Da die ausgeführten Manipulationen sehr generisch und dynamisch sind, sollte das Framework auch dafür ausgelegt sein. Hierbei ist die niedrige Ebene, auf der ASM arbeitet, von Vorteil, da Änderungen einfach auf Grundlage von Aktionen kleinster Granularität ausgeführt werden können. Die Abstraktion von Javassist bietet hier viel weniger Möglichkeiten. Lediglich für die grob definierten Strukturen könnten Änderungen gemacht werden.

Die initiale Hürde von ASM kann dabei nicht umgangen werden, jedoch scheint ein Verständnis des Bytecodes unerlässlich, um diesen zielführend analysieren zu können, selbst wenn Änderungen mit normalem Java-Code eingefügt werden

können. Aus diesem Grund lässt sich dies nicht vermeiden. Bei ASM macht die Dokumentation den Einstieg, sowohl in Bytecode als auch das Framework, jedoch so einfach wie möglich.

Aus diesen Gründen ist ASM die beste Alternative für eine Modifizierung des Bytecodes zur Ausführung. Zusätzlich bietet es auch ein großes Potenzial für Erweiterungen, da sowohl jegliche Art von Struktur erkannt als auch hinzugefügt oder verändert werden kann.

3.3 JDI

JDI steht für das „Java Debug Interface“. Dies ist eine Programmierschnittstelle, engl. Application Programming Interface (API) von Oracle, mit deren Hilfe eine JVM angehalten werden kann um Laufzeitinformationen aus zu lesen. Dies funktioniert sowohl für lokale als auch für entfernte virtuelle Maschinen. JDI stellt dabei die höchste Ebene der Java Platform Debugger Architecture (JPDA) dar, welche seit der Version 1.3 von Java verfügbar ist.

Dies kann dazu genutzt werden, ein Java-Programm in einer spezifisch eingestellten JVM zu starten, welche bei einer eingestellten Liste von Befehlstypen die Ausführung unterbricht und die Kontrolle an eine andere Anwendung übergibt, welche alle Laufzeitinformationen der erstellten JVM auslesen kann. Indem nach jedem ausgeführtem Befehl unterbrochen wird, lässt sich somit ein Trace des gesamten Programmablaufs erstellen. Der wichtige Vorteil ist hierbei, dass keine Manipulation des ausgeführten Codes notwendig ist.

Der Nachteil wiederum ist, dass dies nicht auf der „normalen“ JVM der Java Runtime Enviroment (JRE) möglich ist, sondern lediglich auf der des Java Development Kit (JDK). Da auf nahezu jedem System Java-Programme nur mit der JRE ausgeführt werden, würde die Benutzung eine besondere Konfiguration voraussetzen. Zudem hat JDI nahezu keine Informationen über das zu untersuchende Programm, bevor dieses nicht ausgeführt wurde.

Für diese Anwendung war die benötigte spezielle Konfiguration auf dem Ziel-System jedoch ein K.O.-Kriterium, auch wenn einfache Tests mit JDI vielversprechende Ergebnisse geliefert haben.

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.

4.1.1 Parameter basiert

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 Parametern 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 ein Programm mit ausschließlich textueller Oberfläche. Für die Implementierung existieren in Java einige Bibliotheken und Frameworks, wie beispielsweise:

- Commons CLI
- Java Gems
- JArgs

- GetOpt
- Args4J
- JCommando

Allerdings haben alle den Nachteil, dass sie nur „übliche“ Formate unterstützen. Für besondere Parameterformate muss eine eigene Verarbeitung erfolgen, welche auch zusätzlich den Vorteil bietet, dass somit keine weiteren Abhängigkeiten der Software geschaffen werden.

Die eigene Implementierung dieser Parameterverarbeitung erfolgt schleifenbasiert. 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 weiterleitet. 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 weiterzugebenden Parametern vorhanden.

Für die gesamte Implementierung gilt, dass bei invaliden Parameter-Kombinationen oder fehlenden Parametern eine entsprechende Fehlermeldung ausgegeben wird, gefolgt von der Hilfe wie in Listing 4.1 dargestellt und dass 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

```
This is the commandline interface of Klara.
Klara can be used to track bugs in Java programs.

Argument specification:
java -jar Klara.jar { -h | -H | { [{ -f regex[ lines]
    [ -f regex[ lines]]...}|{ -F regex[ -F regex]...}]
    [ -l lines ] [ -t ] [ -v ] } progname [argument[
```

```
    argument]...] }

Argument details:
  -h | -H
    Show this help message
  -f <regex> [<lines>]
    Specify a whitelist rule. Can be used multiple times
    for multiple entries. Can not be combined with -F. Can
    optionaly specify a specific lineset to debug in
    matching classes, overriding those set by -l
  -F <regex>
    Specify a blacklist rule. Can be used multiple times
    for multiple entries. Can not be combined with -f
  -l <lines>
    Specify a line set to be loged. Use minus to specify
    a range, use comma or semicolon to seperate blocks.
  -t
    Trace the exact line order by printing every line run.
  -o e | <file>
    Write output to stderr or a file.
  -v
    Trace any variable assignment. Variables will be
    printed when declared and every time they are updated.
    Will not track changes of encapsuled variables (Like
    changing the "x" value of a "Point")
  -i
    Use interactive mode instead of detailed call. Will
    ignore other arguments.

Example call:
java -jar Klara.jar -t -v -f my.cool.class.* 20-50
    my.cool.pkg.MyClass arg1 arg2



Press Enter to exit.
```

4.1.2 Zusätzliche Bedienkonzepte

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

Ein navigierbaren Menü ist besonders für komplexe Einstellungen geeignet, da es die Möglichkeit bietet, in einer übersichtlichen Form eine Reihe von Optionen zu wählen. Dies führt jedoch zu dem Nachteil, dass selbst einfache Konfigurationen vergleichsweise lange zum erstellen benötigen. Zudem ist ein Menü in der Implementierung aufwändig. Bei diesem Konzept könnte man jedoch mit Hilfe der gewählten Optionen eine Konfiguration erstellen, diese abspeichern und damit zukünftige Starts beschleunigen.

Ein interaktiver Modus hingegen ist die einsteigerfreundlichste Methode. Hierbei wird durch eine Reihe von Fragen der User nach den gewünschten Einstellungen gefragt. Dies geschieht in einer geordneten Reihenfolge und mit der Möglichkeit für ausführlichere Erklärungen der Option. Auch hier ist ein klarer Nachteil, dass einfache Konfigurationen, im Vergleich zu einem Parametrisierten-Start, länger zum Starten benötigen. Der große Vorteil ist jedoch, dass auch unerfahrene Nutzer direkt die Software benutzen können. Die definierte Reihenfolge und das prüfen von Eingaben sorgen dafür, dass Konfigurationen immer valide sind.

Aufgrund der Zielgruppe der Anwendung, nämlich Programmiereinsteigern mit einfachen Programmen, ist eine einfache Benutzung wichtiger als die Möglichkeit, komplexe Konfigurationen zu erstellen. Das dafür passendere Konzept ist der interaktive Modus.

4.1.3 Interaktiver Modus

Bei der Umsetzung des interaktiven Modus müssen eine Vielzahl von Fragen mit validierten Antworten an den User gestellt werden. Damit eine invalide Antwort nicht zum Abbruch des Programmes führt, müssen alle Fragen so lange wiederholt werden können, bis eine gültige Antwort gegeben wird. Dies lässt sich am einfachsten über eine Reihe von Schleifen realisieren, welche jeweils lediglich die Frage ausgeben und auf Antwort warten, bis diese gültig ist. Um eine bessere Lesbarkeit des Codes zu erreichen werden einfach Ja/Nein-Fragen mittels einer Hilfsmethode durchgeführt, welche die Validierung übernimmt und die Antwort

direkt als boolean liefert. Somit muss nur die Validierung von Fragen mit spezifischen Antwortmöglichkeiten, wie einer Auswahl oder einem definierten Format, in der Methode durchgeführt werden.

Diese Implementierung ist äußerst rudimentär und kann mit steigender Komplexität und Anzahl an Parametern schnell unübersichtlich werden, jedoch für den aktuellen Umfang noch ausreichend. Bei zukünftigen Erweiterungen sollte aber eine Abstraktion der Optionen in Betracht gezogen werden, welche dann von jedem Interface zur Darstellung genutzt wird. Dies würde Definition und Verarbeitung sauber trennen und Fehlerquellen minimieren.

4.2 Individueller Class-Loader

Um Bytecode manipulieren zu können, muss die jeweilige Klasse über einen selbst erstellten Class-Loader geladen werden. Dieser hat die Möglichkeit, nach dem Laden der Binärdaten von der Festplatte und vor dem Zurückgeben an die Virtuelle Maschine (VM) diese zu verändern. Somit sind jegliche Änderungen am Code, die nur die geladene Klasse betreffen, einfach zu realisieren.

Zum Erstellen eines eigenen Class-Loaders muss die Java-Klasse `ClassLoader` des `java.lang` Pakets erweitert werden. Je nach Aufgabe der eigenen Implementierung können nun verschiedene Methoden überschrieben werden. Beispielsweise könnte man hier ein eigenes Caching implementieren, wozu lediglich die `loadClass` Methode überschrieben werden muss. Eine andere Anwendung wäre das Laden von Klassen aus anderen Quellen, wie beispielsweise von einer Netzwerkressource. Bei der vorhandenen Implementierung musste sowohl `loadClass` als auch `findClass` überschrieben werden.

Um dynamisch konfigurierbar zu sein, benötigt der erstellte Class-Loader eine Reihe von Parametern:

Listing 4.2: Parameter des individuellen Class-Loader

```
public static enum FilterType {
        BLACKLIST,
        WHITELIST,
        ALL,
        NOTHING
```

```
}

public TransformingClassLoader(
        boolean cache,
        List<Class<? extends TransformationEventListener>>
                        transformers,
        FilterType filterType,
        Map<Pattern, LineSpecification> filter,
        LineSpecification defaultLineSpec,
        boolean debug)
```

Die als `tranformers` übergebene Liste an Klassen sind die für jeden Ladevorgang erstellten Event-Listener, welche die eigentliche Transformation durchführen. `filterType`, `filter` und `defaultLineSpec` spezifizieren, welche Klassen modifiziert werden sollen und welcher Zeilenbereich. Wird bei einem Filter keine Zeilenspezifikation mitgegeben sondern `null`, so wird der übergebene Standartwert verwendet.

Die Methode `loadClass` ist dafür zuständig zu evaluieren, ob die angefragte Klasse modifiziert werden soll. Wenn bereits eine gecachte Version der Klasse vorliegt, so kann diese direkt zurückgegeben werden. Ist dies nicht der Fall, muss geprüft werden, ob die Klasse überhaupt modifiziert werden soll. Dies wird über eine Hilfsmethode geprüft, welche basierend auf dem gewählten Filtertyp und Filter in Kombination mit dem Namen der zu ladenden Klasse auswertet, ob diese Klasse zu manipuliert ist. Zu modifizierende Klassen werden mit Hilfe der `findClass`-Methode geladen, alle anderen mit Hilfe der Standard-Implementierung.

Die andere überschriebene Methode, `findClass`, wiederum ist für das Finden, Laden und Modifizieren der Binärdaten verantwortlich. Dazu wird zunächst der Name der Klasse zu einem relativen Dateipfad gewandelt. Dies geschieht durch das ersetzen der Punkte, welche Pakete trennen, mit den Datei-Trennzeichen des jeweiligen Systems. Zusätzlich wird die typische Dateiendung „.class“ angehängt. Sollte die Methode `getResourceAsStream` des normalen Class-Loaders diesen finden, so werden damit die Daten geladen. Ist dies nicht möglich, so wird der Pfad als FileInputStream geöffnet. Hier würde sich auch einfach das Programm erweitern lassen, um eine Reihe verschiedener Pfade zu durchsuchen, oder in einem definierten (Unter)-Ordner nach Klassen zu suchen.

Mit der fertig geladenen, unveränderten Klasse muss nun die ASM Event-Kette aufgebaut werden. Dies erfolgt in der umgekehrten Reihenfolge wie die Events weiter gegeben werden. Zuerst muss der `ClassWriter`, welche aus den Events wieder ein Byte-Array erzeugen kann, erstellt werden. Dann wird der Tranformer (siehe Kapitel 4.3) hinzugefügt, welcher seine Events an den Writer weitergeben wird. Außerdem werden sämtliche Event-Listener für ihn erstellt und die passende Line-Specification weitergegeben. Schlussendlich wird, mithilfe der Bytes der unveränderten Klasse, ein `ClassReader` erstellt, welcher diese einliest und als Folge von Events an den Tranformer weitergibt.

Zu dieser 3-Knoten Struktur lässt sich auch ein weiterer, 4. Knoten zwischen Tranformer und Writer einbauen, welcher die erzeugten Bytecode-Befehle ausgibt. Dies ist hauptsächlich bei der Entwicklung neuer Transformation-Event-Listenern und der Weiterentwicklung des Tranformers hilfreich, wenn der erzeugte Code von der JVM nicht ausführbar ist oder funktional nicht das erwünschte Ergebnis liefert. Durch die Position in der Event-Chain kann dieser den erzeugten Code ausgeben bevor er ausgeführt wird und es zu Problemen kommen könnte.

Alternativ zum Erstellen eines eigenen Class-Loaders, welcher veränderten Bytecode lädt, lässt sich in Java auch ein sogenannter „Agent“ registrieren. Dieser muss beim Aufruf der JVM mit dem Parameter „-javaagent“ spezifiziert werden. Ein solcher Agent wird dann nach dem Laden einer Klasse aufgerufen und hat die Möglichkeit, die geladenen Bytes nach Belieben zu verändern. Somit ist die Manipulation sauber getrennt vom Laden der Klassen und eine bessere Kompatibilität mit anderen Programmen ist gewährleistet. Der große Nachteil ist hier jedoch, das die Übergabe von Parametern an den Agent sehr umständlich ist. Es kann lediglich ein einzelner String übergeben werden, der dann vom Agenten getrennt und verarbeitet werden muss. [Duncan, 2014] Ein beispielhafter Aufruf mit einem Agenten ist in Listing 4.3 dargestellt.

Listing 4.3: Beispiel Aufruf mit einem Agent

```
java -javaagent:/tracer/klara.jar=param1;param2;param3 foo.Bar
```

Die Parameterübergabe würde sich mit Hilfe eines Batch/Shell-Scripts in einem „gewohnten“ Format realisieren lassen, jedoch sind die Vorteile eher gering und es ist weniger gut erweiterbar. Aufgrund dieser Punkte fiel die Entscheidung zugunsten des Class-Loaders.

4.3 Transformer

Der Transformer stellt das Herzstück der Anwendung dar. Er führt alle Änderungen am Bytecode aus die erwünscht sind. Dies passiert über eine Event Struktur, um eine modulare Erweiterbarkeit zu erreichen und beim Aufruf eine Auswahl treffen zu können, welche Features aktiv sein sollen.

Eine andere Möglichkeit wäre, die Module direkt in die Event-Chain zu integrieren. Dies hat jedoch den großen Nachteil, dass einzelne Module keinerlei Information über die Aktivität anderer Module erhalten können. Ein weiterer Nachteil ist, dass dabei viel duplizierter Code entstehen würde. Aktionen wie das Durchlaufen der Instruction-Nodes der Methode in einer Schleife müssten von jedem Modul neu realisiert werden. Auch andere Funktionalitäten, wie das Verwalten einer Übersicht der im momentanen Kontext verfügbaren Variablen, wäre eine Aufgabe, die jedes Modul erneut implementieren müsste.

Vorteil einer Integration der Module direkt in die Event-Chain, wäre eine einfachere Struktur und weniger Hilfsklassen. Die zusätzlich geschaffene Abstraktion durch den Transformer wäre damit überflüssig und mit ihr alle dafür benötigten Komponenten. Dies würde den Code übersichtlicher und schneller verständlich machen, da man nur die Event-Struktur von ASM verstehen müsste, nicht eine weitere projektspezifische.

Jedoch können die Vorteile der Verwendung lediglich einer Struktur bei weitem nicht die Nachteile dieser ausgleichen, weshalb der Transformer implementiert wurde.

Er hält alle Informationen über den momentanen Punkt der Transformation, auf die andere Klassen zugreifen können. Zu den Daten, die er verwaltet, gehören:

- Name der Klasse
- Name der Methode
- Zeile im Original-Code
- Eine Liste der momentan verfügbaren Variablen
- Eine Liste der Variablen, welche noch in der aktuellen Methode deklariert werden

Zusätzlich bietet er eine Methode zum Einfügen von Text-Ausgaben. Diese benötigt lediglich den Anweisungs-Knoten, nach dem die Ausgabe erfolgen soll, sowie eine Anweisungs-Liste, die einen geöffneten String-Builder Text hinzufügt. Ausgaben beginnen immer mit einer Angabe der momentanen Position im Code, um eine hohe Nachvollziehbarkeit des Ablaufs zu erreichen.

Alle anderen öffentlichen Methoden des Transformers dienen zur Konfiguration oder sind Teil der `ClassNode`-Klasse, welche für Elemente der ASM Event-Chain benötigt werden. Die beiden privaten Methoden sind Hilfsmethoden von `visitEnd`.

4.3.1 Event Struktur

Die Entscheidung für eine event-basierte Struktur mit beliebig vielen Zuhörern basiert darauf, dass eine Entwicklung weiterer Module möglich sein soll. Dies hat zudem den Vorteil, dass die Auswahl, welche Module bei einer Ausführung aktiviert sein sollen, ganz einfach mit einer Liste an Zuhörern gesteuert wird.

Für diese Struktur stehen 2 verschiedene Modelle zur Auswahl: Ein großes Interface mit einer Methode für jedes Event, ähnlich der ASM Event-Chain, oder ein Interface mit lediglich einer Methode, welche unterschiedliche Implementierungen eines Event-Interface empfängt. Die zweite Variante ist bei Java-Swing verbreitet, beispielsweise als `ActionEventListener`.

Beide Varianten haben Vor- und Nachteile. Ein großes Interface hat eine klar definierte Struktur mit einer Reihe von Parametern pro Methode, die völlig individuell für jedes Event gewählt werden können. Dies birgt jedoch gleichzeitig den Nachteil, dass Klassen, die nicht auf alle Events reagieren wollen, trotzdem leere Methoden für diese implementieren müssen. Auch ist die Struktur sehr statisch und ungeeignet für Veränderung, da bei einer Änderung des Interface auch immer alle Implementierungen aktualisiert werden müssen. Eine Möglichkeit, leere Methoden zu vermeiden und Erweiterungen einfacher zu machen, sind Adapter. Dies sind Klassen, welche alle Methoden leer implementieren. Der Nachteil dabei liegt jedoch darin, dass Listener diese erweitern müssen und nicht nur implementieren, was die Einsatzmöglichkeiten einschränkt.

Ganz im Gegenteil dazu ist einer der größten Vorteile eines Interface mit nur einer Methode, dass Erweiterungen keinen Einfluss auf vorhandene Implementierungen haben. Neu erstellte Event-Klassen werden nicht verarbeitet von bestehenden

Event-Listenern, wodurch ihr Verhalten sich nicht ändert, solange diese richtig implementiert sind. Dies zeigt auch, dass lediglich jene Events verarbeitet werden müssen, welche auch benötigt werden. Der Nachteil hierbei liegt jedoch darin, dass jede Methode des großen Interface hier eine eigene Event-Klasse ist. Dies führt schnell zu Paketen voll von Klassen welche nahezu keine Logik enthalten. Wenn ein Listener auf alle Events reagieren muss, benötigt er dafür eine große Verzweigung, die auf sämtliche relevanten Event-Typen prüft. Dies führt zu einer erhöhten Code-Komplexität und langsamerer Ausführung. Eine Optimierung ist hierbei möglich, indem die Events eine Methode bieten um diese zu identifizieren.

Aufgrund der auf Erweiterung ausgelegten Architektur und der Annahme, dass die meisten Implementierungen nur auf einen kleinen Teil der Ereignisse reagieren müssen, wurde ein kleines Interface mit verschiedenen Event-Klassen realisiert.

Das erstellte Interface `TransformationEvent` definiert zwei Methoden: Eine zum identifizieren, um welches Event es sich handelt und eine weitere, welche den abstrakten Knoten liefert auf den das Ereignis bezogen ist. Dadurch kann bei Listenern häufig auf ein Type-Cast verzichtet werden, da diese beiden Informationen für die meisten Anwendungsfälle genügen.

4.3.2 Erstellte Implementierungen

Es wurden zwei Implementierungen des `TransformationEventListener` gemacht. Die erste, `VariableChangePrinter`, zur Ausgabe jeder Änderung von Klassen-Variablen, Parametern oder lokalen Variablen. Die zweite, `LineTracer`, um eine Ausgabe in jeder Zeile zu garantieren, damit der Ablauf ersichtlich ist. Beide sind in ihrer Struktur simpel gehalten. Sie können nur mit einem existierenden Transformer erstellt werden, da sie eine feste bidirektionale Bindung mit diesen initialisieren, welche unveränderlich ist.

Variable Change Printer

Für die Ausgabe aller Änderungen an Variablen sind vier Events relevant: Var-Instruction, Iinc-Instruction, Scope-Reached und Class-Variable-Changed. Die Verarbeitung aller dieser Ereignisse läuft ähnlich ab: Zuerst wird die veränderte Variable aus dem Event-Objekt extrahiert, dann eine Ausgabe des Wertes generiert und diese schlussendlich an den Transformer gegeben, um dies nach der Zuweisung einzufügen.

Die Generierung der Ausgabe ist dabei eine größere Hilfsmethode. Mit Hilfe einer selbst erstellten Abstraktion der Variablen lassen sich alle Arten gleich verarbeiten. Je nach Typ der Variable werden leicht unterschiedliche Formatierungen gewählt und deren Ausgabe als Bytecode generiert.

Line Tracer

Da der Line Tracer nur eine Ausgabe in einer Zeile machen soll, wenn kein anderer Listener bereits eine Ausgabe hinzugefügt hat, muss dieser nicht nur auf das Erreichen neuer Zeilen, sondern auch auf das Hinzufügen von Ausgaben reagieren. Die hierzu relevanten Events sind: Line-Start, Line-End und Print-Added.

Die Funktionalität ist wiederum sehr einfach: Mit Hilfe einer Variablen wird gespeichert, ob in der aktuellen Zeile noch eine Ausgabe gemacht werden muss. Wird das Zeilenende erreicht und dies ist noch nötig, wird am Beginn der Zeile eine leere Ausgabe hinzugefügt. Da Ausgaben immer die Position anzeigen, enthält diese automatisch alle nötigen Informationen.

4.4 Lizenzierung

Bei der Wahl der Lizenz für die neu entwickelte Software war es wichtig, dass diese kompatibel zu den Lizenzen der Abhängigkeiten ist. Da keine externen Bibliotheken verwendet wurden und nur ein Framework benutzt wird, musste hier nicht viel beachtet werden.

Das benutzte Framework, ASM, steht unter einer nicht benannten Lizenz, die der BSD-Lizenz sehr ähnlich ist. Der relevante Teil von dieser, welcher sich auf Verwendung und Redistribution bezieht, ist folgender:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. *Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.*

2. *Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.*

3. *Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.*

[ASM, 2016]

Somit sind die einzigen Einschränkungen, dass die Lizenz des Frameworks unverändert bleiben und mit angehängt werden muss, wenn sie gemeinsam ausgeliefert wird.

Da das Ziel der Arbeit ein Lehrmittel war, welches nicht für kommerzielle Anwendung ausgelegt ist und auch kein Unternehmen hinter der Entwicklung steht, ist eine Open-Source Lizenz für freie Nutzung hier geeignet. Hierfür gibt es eine große Auswahl. Zu den verbreitetsten Lizenzen zählen:

- GNU General Public License (GPL)
- GNU Lesser General Public License (LGPL)
- Apache-Lizenz
- Berkeley Software Distribution (BSD)-Lizenz
- European Public License (EUPL)

Alle diese Lizenzen sind sich relativ ähnlich, haben aber im Detail Unterschiede. Bei der GPL muss jede Software, die unter GPL lizenzierte Teile enthält, unabhängig ob in Originalfassung oder modifiziert, auch unter der GPL lizenziert werden, dies nennt sich „Copyleft“. [Wikipedia, 2016d] Die LGPL ist hier weniger strikt, lediglich die LGPL-Teile der Software müssen öffentlich gemacht werden und von Nutzern ersetzbar oder änderbar sein. [Wikipedia, 2016e] Die Apache-Lizenz ist der LGPL sehr ähnlich, jedoch lassen sich hier eigene weitere Urheberrechtsvermerke hinzufügen. [Wikipedia, 2016a] Auch die BSD-Lizenz ist dieser sehr ähnlich, hat jedoch in ihrer Originalform eine Werbeklausel, welche in allen Werbematerialien enthalten sein muss. [Wikipedia, 2016b] Die EUPL schlussendlich bietet den großen Vorteil, dass sie in allen 22 Amtssprachen der Europäischen Union verfügbar ist und unter Berücksichtigung des europäischen Rechts erstellt wurde. Sie ist kompatibel mit der GPL. [Wikipedia, 2016c]

Die BSD-Lizenz scheidet aufgrund ihrer Werbeklausel aus. LGPL und Apache-Lizenz sind besser für Bibliotheken und Frameworks geeignet, die in anderen Projekten eingebunden werden. Somit bleiben GPL und EUPL die passendsten Lizenzen. Im Vergleich hier liefert die Mehrsprachigkeit der EUPL sowie die spezifische Berücksichtigung europäischen Rechts eine klare Entscheidung.

Als Lizenz wurde somit die EUPL verwendet, auf Grund des passenden starken Copylefts und der internationalen Gültigkeit.

5 Fazit

Im Rahmen der Bachelorarbeit wurden die geforderten Funktionen erfolgreich umgesetzt. Das entstandene Programm bietet grundlegende Möglichkeiten zur Fehlersuche in Java-Programmen mit einer Kommandozeilen-Oberfläche. Diese ist sehr simpel gestaltet, um auch für Programmieranfänger intuitiv bedienbar zu sein.

Der höchste Aufwand der Arbeit lag bei der Entscheidung für eine Technologie, mit welcher die Aufgaben umgesetzt werden konnten. Hierbei musste ich mich in einige verschiedene Technologien einarbeiten und jeweils kleine Programme schreiben, um deren Nutzbarkeit zu evaluieren. Aufgrund der Komplexität der meisten Frameworks gestaltete sich dies als sehr zeitintensive Aufgabe, weil bei Problemen immer darauf geachtet werden musste, ob die Ursache bei der Implementierung oder der Technologie liegt. Dies hat jedoch auch einige interessante Einblicke gegeben, da alle getesteten Frameworks durchaus sinnvolle Einsatzgebiete haben.

Bei der Umsetzung lagen die größten Herausforderungen beim Finden eines Formats, in dem Optionen weitergegeben werden, sowie der Struktur, um Transformatoren modular ohne Code-Duplikation implementieren zu können. Für die Optionen wäre bei Erweiterungen eine Vereinfachung mit Hilfe von kapselnden Klassen vorstellbar, welche beispielsweise alle Filteroptionen kombiniert. Bei den Transformatoren habe ich viel Aufwand darauf verschwendet, eine Möglichkeit zu finden, diese in die bestehende Architektur von ASM zu integrieren. Viel des dafür geschriebenen Codes zeigte sich später als nutzlos, nachdem die Struktur umgebaut wurde.

Im Allgemeinen war das Thema eine persönliche Bereicherung, da es viele neue Einblicke in die Programmiersprache gegeben hat. Zwar war mir die Existenz des Bytecodes bewusst, die spezifische Umwandlung von Java-Code zu Bytecode-Befehlen und die Möglichkeit, diese direkt zur Laufzeit zu manipulieren waren jedoch gänzlich neu. Mit aktiv weiterentwickelten Projekten wie ASM sehe ich Potenzial für viele verschiedene Anwendungen in diesem Umfeld.

5.1 Erweiterungsmöglichkeiten

Mit der grundlegenden Funktionalität gegeben, sehe ich Potenzial für verschiedene Erweiterungen. Wie in Kapitel 4.1 bereits beschrieben, wäre eine Menü-Oberfläche für komplexere Konfigurationen denkbar. Somit könnte der Anwender in beliebiger Reihenfolge die Optionen wählen, bis ihm die Konfiguration gefällt. Die Menüpunkte selbst könnten dann Werte ähnlich erfragen wie es beim interaktiven Modus implementiert ist. Mit der Möglichkeit, aufwändige Konfigurationen in einem Menü zu erstellen, sollte auch eine Option zum Speichern dieser Konfiguration implementiert werden. Ein besonders einfaches Format hierfür wäre die Parameterliste, wie sie auch direkt übergeben werden kann. Wenn diese zu groß wird oder weitere Optionen hinzukommen, die nur über das Menü verfügbar sind, sollte eine XML oder JSON-Struktur verwendet werden. Da das Menü eine 3. Oberfläche darstellen würde, welche die gleichen Optionen bieten soll, wäre eine Abstraktion der Parameter sinnvoll. So könnten alle Informationen zu einer Option in einem Format gebündelt werden, das alle Oberflächen benutzen können. Somit sollten alle Informationen, von dem Argumentname als Übergabeparameter bis hin zu Wert-Validierung, enthalten sein. Dies würde sicherstellen, dass die gleiche Funktionalität über alle Oberflächen erreichbar ist und Code-Duplikationen vermeiden.

Eine zusätzlich denkbare Option ist ein gröberes Branching zu untersuchen, als Alternative zur Ausgabe jeder Zeile. Ausgegeben würde in diesem Fall beispielsweise nur, dass eine Schleife 17 Durchläufe hatte oder ein Switch-case zu Fall 3 gesprungen ist. Dies würde bei größeren Abläufen die Übersichtlichkeit erhöhen, besonders wenn lange laufende Schleifen vorhanden sind.

Weitere kleinere mögliche Verbesserungen sind ein Ant-Plugin und Support für das debuggen von jar-Archiven. Ein Ant-Plugin könnte als Integration in einer beliebigen Entwicklungsumgebung genutzt werden. Jar-Archive können momentan noch nicht untersucht werden, lediglich .class-Dateien.

Desweiteren habe ich Ideen für fünf große Erweiterungen:

Unsichtbare Überprüfungen, welche in einer Konfiguration definiert werden. Diese funktionieren ähnlich wie eine „assert“-Anweisung, sind aber nicht Teil des Codes. Dies ist syntaktisch besser, da sie lediglich Tests darstellen und nicht zum Programm gehören. Somit bleibt der Code übersichtlicher und die Ausführung wird

nur dann verlangsamt, wenn die Überprüfungen auch ausgeführt werden sollen.

Automatisches Erkennen der relevanten Klassen. Wenn eine Klasse `some.package.Foo` untersucht werden soll, gilt selbiges wahrscheinlich für andere Klassen im Paket `some.package`. Hier könnte eine Reihe von logischen Abhängigkeiten gebildet werden, um das manuelle Setzen von Filtern nahezu überflüssig zu machen.

Ähnlich dazu könnten verwendete Typen modifiziert werden, um Änderungen an den gekapselten Variablen zu erkennen. Hierbei könnte auch eine Prüfung des Stacktrace eingebaut werden, welche nur in bestimmten Fällen Ausgaben erzeugt. Damit wäre auch erkennbar, wenn ein gespeicherter Punkt nicht ersetzt wird, sondern lediglich die Koordinaten sich ändern.

Erzeugen eines maschinell lesbaren Trace, anstelle eines menschlich lesbaren. Dieser könnte viel performanter erzeugt werden als es beispielsweise bei Python Tutor der Fall ist, da keine Pausierung der VM stattfindet. Ein anderes Programm könnte diesen dann in einem für Entwickler geeigneten Format visualisieren.

Logging mit bekannten Frameworks wie Log4j erzeugen und die Ergebnisse speichern. Solche veränderten Klassen könnten auf Server zum Einsatz kommen, um bei Problemen Logging mittels des Frameworks zu aktivieren ohne etwas am Code oder Server-Setup ändern zu müssen. Da Server auf Grund der Anforderungen an Antwortzeiten und häufiger Komplexität der Umgebung nicht einfach zu debuggen sind, wäre dies eine schnellere Alternative.

5.2 Weitere Anwendungsbereiche

Zusätzlich zu möglichen Erweiterungen der bestehenden Anwendung sehe ich auch Potenzial für andere Anwendungsgebiete der Technologie. Bei allen ausgeführten Änderungen am Bytecode wurde bisher kein Konstrukt erzeugt, was nicht auch durch Änderung des Sourcecodes möglich gewesen wäre. Hier bieten sich jedoch neue Möglichkeiten.

Obfuscation beschreibt das Verändern des Codes in solcher Weise, dass der originale Code nur noch sehr schwer oder in keiner Weise wiederherstellbar ist. Dies kommt besonders bei kommerzieller Software zum Einsatz, wenn das Kompilat der Sprache Rückschlüsse auf den Code geben kann. Dies ist unter anderem bei

allen Sprachen die Bytecode verwenden der Fall. Hier könnte mit ASM das übersetzte Programm stark umgebaut werden, ohne die eigentliche Funktionalität zu ändern. Mit Hilfe von Sprüngen quer durch den Code und den potenziellen kleinen Veränderungen der Aufruf-Struktur sollten herkömmliche Decompiler nicht mehr in der Lage sein, einen sinnvollen Code daraus zu erzeugen. Allerdings muss dabei stark darauf geachtet werden, die Funktionalität wirklich nicht einzuschränken und der Speicherverbrauch sowie die Laufzeit ändern sich wahrscheinlich.

Eine der Veränderungen, die hierbei durchgeführt werden können, aber die Laufzeit potenziell verbessern, sind „Makros“. Dies könnte als Annotation an statischen Methoden realisiert werden. Aufrufe dieser werden durch den Inhalt der Methode ersetzt, um den Kontextwechsel zu vermeiden und potenziell das Laden der Utility-Klasse gänzlich unnötig zu machen.

Genau gegensätzlich dazu könnte ein unsichtbarer Code erstellt werden. Dies wären Klassen mit leeren Methoden, deren Code beim Laden generiert wird. In der Verwendung wäre ein Unterschied nicht erkennbar, jedoch bietet er eine denkbare Alternative zu klassischen Code-Generatoren.

Alle diese Mittel, welche einen schwer nachvollziehbaren Code erzeugen, können auch bei der Erstellung von Aufgaben für Wettbewerbe verwendet werden. Hier könnte eine Reversing-Aufgabe, welche sonst meist nur aus Maschinencode besteht, in Form von komplexem Bytecode erfolgen. Dieser würde weiteren Code generiert, der das eigentliche Programm darstellt. So könnte eine große Verschachtelung erstellt werden, welche eine Herausforderung zum Entschlüsseln darstellt.

6 Verfügbarkeit der Anwendung

Der gesamte Code der Anwendung ist online unter diesem Link verfügbar:

https://github.com/Monoblos/Klara/tree/V1

Die URL ist auch in gekürzte Version, sowie als eine QR-Code Repräsentation verfügbar:

https://goo.gl/QiQEqI

Abbildung 6.1: QR-Code-Darstellung der ersten kurzen URL (Quelle: Generiert von Google)

Ebenfalls sind alle Dateien zum erstellen dieser schriftlichen Ausarbeitung online verfügbar:

https://github.com/Monoblos/Bachelor-Thesis-Klara/tree/V1

https://goo.gl/gJSoTj

Abbildung 6.2: QR-Code-Darstellung der zweiten kurzen URL (Quelle: Generiert von Google)

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[ASM, 2016] ASM (2016). Asm license. http://asm.ow2.org/license.html. Letzter Zugriff am 11 August, 2016.

[Duncan, 2014] Duncan (2014). How do i pass arguments to a java instrumentation agent? http://stackoverflow.com/questions/23287228/how-do-i-pass-arguments-to-a-java-instrumentation-agent. Letzter Zugriff am 16 August, 2016.

[Guo, 2016] Guo, P. (2016). Python tutor. http://pythontutor.com/. Letzter Zugriff am 21 August, 2016.

[Oracle, 2016] Oracle (2016). Java™ debug interface. https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/. Letzter Zugriff am 21 August, 2016.

[Puls, 2014] Puls, A. (2014). Diving into bytecode manipulation: Creating an audit log with asm and javassist. https://blog.newrelic.com/2014/09/29/diving-bytecode-manipulation-creating-audit-log-asm-javassist/. Letzter Zugriff am 16 August, 2016.

[Sutinen, 2007] Sutinen, E. (2007). Jeliot 3. http://cs.joensuu.fi/jeliot/description.php. Letzter Zugriff am 21 August, 2016.

[Ullenboom, 2010] Ullenboom, C. (2010). *Java ist auch eine Insel*. Galileo Press GmbH, 8 edition.

[Wikipedia, 2015] Wikipedia (2015). Bytecode. https://de.wikipedia.org/wiki/Bytecode. Letzter Zugriff am 21 August, 2016.

[Wikipedia, 2016a] Wikipedia (2016a). Apache-lizenz. https://de.wikipedia.org/wiki/Apache-Lizenz. Letzter Zugriff am 12 August, 2016.

[Wikipedia, 2016b] Wikipedia (2016b). Bsd-lizenz. https://de.wikipedia.org/wiki/BSD-Lizenz. Letzter Zugriff am 12 August, 2016.

[Wikipedia, 2016c] Wikipedia (2016c). European union public licence. https://de.wikipedia.org/wiki/European Letzter Zugriff am 12 August, 2016.

[Wikipedia, 2016d] Wikipedia (2016d). Gnu general public license. https://de.wikipedia.org/wiki/GNU Letzter Zugriff am 12 August, 2016.

[Wikipedia, 2016e] Wikipedia (2016e). Gnu lesser general public license. https://de.wikipedia.org/wiki/GNU Letzter Zugriff am 12 August, 2016.

[Wikipedia, 2016f] Wikipedia (2016f). Tracing (software). https://en.wikipedia.org/wiki/Tracing Letzter Zugriff am 12 August, 2016.