Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Projektarbeit Informatik

Neuartiges Testkonzept mittels Bytecode-Instrumentation

Autoren	Nicholas Wright David Zolliker
Hauptbetreuung	Mark Cieliebak Karl Rege

Datum	19.12.2012

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:

……

Unterschriften:

……

……..

……

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

1 Zusammenfassung

Das Testen von Software mittels Unittests gilt heute als unerlässliche Massnahme zur Sicherstellung der Qualität. Das Schreiben dieser Tests bedeutet jedoch zusätzlichen Aufwand und wie die Software selbst, so müssen auch die zugehörigen Tests gewartet werden. In der Praxis existiert deshalb dennoch ungetesteter Code oder Code mit unbrauchbaren, weil nicht gewarteten Tests.

Es gibt bereits Verfahren, die basierend auf der Analyse des statischen Quellcodes automatisch Unittests generieren. Eine Analyse des Programmes zur Laufzeit mittels Bytecode Instrumentation könnte jedoch neue Möglichkeiten und im Endeffekt bessere Tests bieten, da mit den realen, zur Laufzeit auftretenden Daten gearbeitet werden kann. Mit dieser Arbeit wurden erste Schritte zur Umsetzung dieses neuartigen Konzeptes unternommen, sowie Lösungsansätze und mögliche Hindernisse untersucht. Sie soll eine Grundlage für weiterführende Forschung auf diesem Gebiet liefern.

Anhand von einfachen experimentellen Java-Programmen wurde untersucht, welche Laufzeitdaten erfasst werden müssen und wie die Datensätze gespeichert werden können. Im Anschluss daran wurde unter Nutzung der Javassist Bibliothek eine Architektur erstellt, die diese Datensätze zusammenstellt und zur Generierung von Unittests einsetzt. Um Objekt-Abhängigkeiten aufzulösen wurden ebenfalls auf der Basis der durch Instrumentation gewonnen Daten Mock-Klassen erstellt.

Es konnte für einfache Programmkonstrukte gezeigt werden, wie Objektzustände vor und nach einem Methodenaufruf identifiziert werden können. Weiter konnte diese definierte Zustandsänderung sowie der Rückgabewert der Methode für die automatische Formulierung von Unittests genutzt werden. Für die Isolierung der Klasse unter Test wurde ein Lösungsansatz für die Erzeugung von Mockobjekten, die das zur Laufzeit festgestellte Verhalten aufweisen, demonstriert. Bis jedoch für produktive, komplexe Programm auf diese Weise Unittests erzeugt werden können, wird noch erheblicher Forschungsaufwand nötig sein. Einer der nächsten Schritte dürfte sein, die getroffene Einschränkung auf primitive Datentypen aufzuheben. Mit geeigneten Algorithmen könnten aus den Daten Äquivalenzklassen gebildet werden,

was die Flexibilität und Performance der Tests deutlich erhöhen sollte.

2 Abstract

Software testing using unittests is nowadays regarded as an essential measure to ensure the quality of the product. However, writing these tests means additional work and like the software itself, the associated tests also need to be maintained. Thats why in reality there still exists untested code or with tests that became useless because they were not maintained.

Algorithms that generate unittest automatically based on the analysis of the static source code already exists. However, an analysis of the program to be tested at runtime using bytecode instrumentation could provide new opportunities and better suited tests because it incorporates real world data used by the application. With this project thesis, the first steps towards implementing this new concept have been taken and possible solutions and obstacles evaluated. This should serve as a basis for further research in this area.

Using simple experimental java-programs we have studied which runtime data needs to be collected and how the datasets can be stored. We then continued to develop an architecture that assembles these records and generates unit tests from them, using the Javassist library. In order to dissolve object dependencies, mock-classes have been created on the basis of the data obtained by instrumentation.

For simple java-programs it could be shown how object states can be identified before and after a method call. Furthermore, this predefined change in state and the return value of the method was successfully used for the automatic generation of unittests. To isolate the class under test, we implemented an automatic generation of mock-objects that exhibit the exact behavior demonstrated by their non-mock counterparts during the runtime of the program under test. However, for unittests of productive, complex programs to be generated using this concept, considerable future research effort will be necessary. One of the next steps should be to repeal the restriction of using only primitive datatypes in the program under test. With appropriate algorithms, equivalence classes could be formed from the data, which should increase the flexibility and performance of the tests significantly.

3 Vorwort

Diese Arbeit vereinigt zwei Aspekte, die uns als Autoren sehr interessieren. Der erste Aspekt ist das Softwaretesting. Das Schreiben von Unittests ist wohl für die meisten eine eher mühsame Pflichtarbeit. Dennoch trägt es indirekt dazu bei, das Schreiben von Software zum Genuss zu machen. Dank Unittests kann man seine Software erweitern, umstrukturieren und verbessern und sich trotzdem noch sicher sein, das die Funktionalität erhalten bleibt.

Der zweite Aspekt, die Bytecode Instrumentation, ermöglicht es uns in tiefere Ebenen der höheren Programmiersprache Java einzutauchen. So komfortabel die Abstraktionen der Sprache und der Laufzeitumgebung für den Programmierer sind, so wecken sie doch auch das Bedürfnis, einen Blick hinter die Kulissen zu werfen. Dank dieser Projektarbeit konnten wir erste Einblicke in Bytecode Instrumentation und Reflection gewinnen, zwei Thematiken, die sonst nicht Bestandteil des Lehrplans sind.

Wir möchten unseren Betreuern Karl Rege und Mark Cieliebak für ihre Unterstützung danken. Sie haben uns in vielen spannenden Diskussionen angeregt und ermutigt in dieser neuen Domäne Fuss zu fassen.

Inhaltsverzeichnis

4 Einleitung

4.1 Ausgangslage

Ein Bestandteil diese Projektes war die Recherche zu diesem Thema, zum einen, um Einblick in die automatische Testgenerierung zu erhalten, zum andern, um herauszufinden ob schon Forschung in dieser spezifischen Richtung getätigt worden ist. Auch sollte überprüft werden, ob eine kommerzielle Lösung bereits vorhanden ist.

Produkte zur automatischen Test-Erzeugung sind bereits vorhanden, jedoch basieren diese Produkte hauptsächlich auf statischer Code-Analyse. Auch gibt es Ansätze die versuchen mittels Symbolic Execution sämtliche ausführungs Pfade eines Programmes ablaufen und somit die Abdeckung eines Programmes zu erhöhen. Ein Beispiel hierzu ist KLEE, das mittels Symbolic Execution Tests erzeugen kann. [14] Eine Kombination der beiden Techniken ist auch möglich wie das Palus tool zeigt.[15]

Produkte die Laufzeit-Daten zur Erzeugung von Unittests verwenden konnten wir keine finden.

4.2 Ziel dieser Arbeit

Unittests haben die Softwareentwicklung stark verändert. Ungetestete Software gilt heute zurecht als qualitativ minderwertig bis inakzeptabel[9]. Dies gilt nicht nur in Bereichen, in denen Softwarefehler katastrophale Auswirkungen haben können, wie etwa in der Raumfahrt[16]. Denn automatisierte Tests ermöglichen es auch, Software zu erweitern, zu verbessern und umzustrukturieren (Refactoring) ohne befürchten zu müssen, dass sich neue Fehler einschleichen. Sie tragen so wesentlich zur Stabilität von Software bei und ermöglichen eine agilere Arbeitsweise.

In der Realität existiert dennoch viel ungetester Code. Das Schreiben von Tests ist aufwändig, erfordert fundiertes Verständnis des zu testenden Codes und erfüllt - auf den ersten Blick - keine Kundenbedürfnisse. So fällt es oft als erstes dem Termindruck zum Opfer.
Diese Arbeit soll die Grundlage schaffen, um zu evaluieren, welche Möglichkeiten Bytecodeinstrumentation für neuartige Testkonzepte bietet. Es soll

ein Architektur entworfen werden, mit der eine Applikation instrumentiert, ihre Laufzeitdaten gespeichert und schliesslich Unittests und Mockklassen basierend auf diesen Daten generiert werden können. Die dabei gewonnenen Erkenntnisse über bestehende Schwierigkeiten und erforderliche Erweiterungen werden als Resultate festgehalten und können zu einem späteren Zeitpunkt zum Ausbau des Konzeptes genutzt werden.

5 Theoretische Grundlagen

5.1 Unit-Testing

Unit-Testing ist eine Technik in der Software Entwicklung, bei der sämtliche Komponenten eines Projekts voneinander isoliert getestet werden. Ziel dieser Technik ist es sicherzustellen, dass die einzelnen Komponenten sich wie erwartet verhalten. Das übliche Vorgehen ist, dass eine Funktion mit definierten Parametern aufgerufen wird und der zurückgelieferte Wert mit einem erwarteten Wert überprüft wird.

5.2 Mocking

Es kommt häufig vor, dass Softwarekomponenten von anderen Softwarekomponenten abhängig sind. Dies erschwert das Testen, da abhängige Komponenten den Test beeinflussen können, ihrer Erzeugung ressourcenaufwendig ist, die Ressourcen nicht verfügbar sind, oder sie weitere Abhängigkeiten beinhalten. Zweck von Mocks ist die Entkoppelung von Abhängigkeiten. Sie können anstelle von richtigen Klassen verwendet werden und es lässt sich spezifisches Verhalten definieren. So kann z.B. ein Mock konfiguriert werden, für bestimmte Parameter einen entsprechenden Return-Wert zu liefern, oder eine Exception auszulösen. Des weiteren kann nach der Ausführung eines Tests die Interaktion mit einem Mock überprüft werden (Verifikation). Dies ermöglicht es, zu überprüfen, ob Aufrufe in einer bestimmten Reihenfolge getätigt wurden und ob sie die erwarteten Parameter verwendet haben.

5.3 Bytecode-Instrumentation

Bytecode-Instrumentation bezieht sich auf die Fähigkeit, bereits kompilierten Programmcode zur Laufzeit zu ändern. Es ist somit möglich, Programme zu schreiben, die sich dynamisch den Gegebenheiten anpassen können. Auch ist es somit möglich, das Verhalten eines Programmes zu verändern, ohne dass der Quellcode zur Verfügung steht oder das bestehende Programm verändert werden muss. Für solche Zwecke wird meist eine Bibliothek verwendet wie z.B. BCEL, ASM oder Javassist. Bytecode von Hand zu schreiben ist auch möglich, jedoch sehr aufwendig. Ein Vergleich hierzu wäre Java

zu Bytecode wie C++ zu Assembler. Meistens ist den Anwendern/Entwicklern nicht bekannt, dass Bytecode-Instrumentation verwendet wird, da diese ihr Werk im Hintergrund verrichtet. Ein paar Programme und Projekte die Bytecode-Instrumentation verwenden: VisualVM für Profiling, AspectJ.

In diesem Projekt wurde Javassist verwendet, da diese Bibliothek es ermöglicht, Java-Code zur Instrumentation zu verwenden. Dies erleichterte unsere Arbeit sehr, da die ganze Instrumentation mit Java anstelle von handgeschriebenem Bytecode gemacht werden konnte.

5.4 Reflection

Die Fähigkeit einer Programmiersprache, sich selbst zu inspizieren. Ein Programm kann so zur Laufzeit Informationen über sich selbst sammlen und gewisse Informationen verändern. Für Reflection werden keine externen Bibliotheken benötigt, da diese Funktion ein Teil der Programmiersprache ist. Dies ermöglicht es einem Programm Informationen über geladene Klassen zu sammeln (z. B. Methoden namen, Felder Namen, Sichtbarkeit) und Werte von Feldern zu ändern. Was Reflection nicht kann, ist das Verhalten von bestehenden Klassen zu verändern.

6 Vorgehen

6.1 Risiken und Vorgehen

Diese Arbeit vereinigt zwei Fachgebiete grosser Komplexität, Bytecodeinstrumentation und Testing. Während Bytecodeinstrumentation nach unserem Wissen gar nicht zum Curriculum des Informatikstudienganges an der ZHAW gehört, werden die Grundlagen des Testings parallel zu dieser Arbeit im Modul 'Softwareentwicklung 2' behandelt. Das grösste Risiko stellt deshalb die Komplexität verbunden mit der geringen bis nicht vorhandenen Expertise auf diesen Gebieten dar. Um diesem Risiko zu begegnen, müssen die behandelten Themen eingeschränkt werden. In wöchentlichen Treffen mit den Betreuern dieser Arbeit werden die aktuellen Schwierigkeiten und mögliche Vorgehensweisen besprochen.

Als erstes haben wir uns einen Überblick über die aktuelle Literatur bezüglich automatisierter Testgeneration verschafft und den zur Verfügung gestellten Demo-Code studiert, der die grundlegenden Anwendung der Javassist-Bibliothek zur Bytecodeinstrumentation demonstriert. Daraufhin haben wir unser eigenes Projekt auf einem Git-Repository aufgesetzt. Für die Automatisierung des Build- und Test-Prozesses wurden sowohl Ant- als auch Maven-Skripts verwendet.

Es folgte die Definition des Aufbaus des Method Call Records, der als Grundlage für die weiteren Schritte erforderlich war. Es galt nun einen eigenen Agenten sowie eine Instrumentationsklasse zu implementieren, mit deren Hilfe Laufzeitdaten extrahiert und in den Records gespeichert werden konnten. Als dies gelungen war, konnte die automatisierte Generierung von Unittests in Angriff genommen werden. Basierend auf einem handgeschriebenen Unittest für eine Klasse wurden Funktionen implementiert, die sich durch die aufgezeichneten Daten arbeiten und die benötigten Blöcke und Statements erzeugen. Um die Klasse unter Test(KUT) von ihrer Umwelt isolieren zu können, mussten nun noch Mockklassen erzeugt werden, die sich gegenüber der KUT genau so verhalten, wie das die echte Umgebung während der Aufzeichnungsphase tat.

Zuletzt galt es, die gewonnen Erkenntnisse zusammenzufassen, um eine Grundlage für weitere Arbeiten auf diesem Gebiet zu bieten.

6.2 Die Instrumentierung

6.2.1 Start über Launchfile

Zur einfachen Ausführung der Instrumentation wurde ein Eclipse-Launchfile erstellt (Dummy with Agent.launch), welches die Dummy Klasse mit dem Agent zusammen ausführt. Alternativ kann der Agent auch mit einem beliebigen Programm mittels der Kommandozeilen Option -javaagent: ausgeführt werden.

6.2.2 Der Agent

Agents sind eine Eigneschaft von Java die es ermöglichen das Verhalten eines Java Programmes anzupassen oder zu erweitern, ohne das der Quellcode oder die ausführbare Datei des Orginalprogramm angepasst werden muss. Java bietet die möglichkeit, einem bestehenden Programm einen oder mehrer Agents anzuhängen, die das betroffene programm instrumentieren. Damit eine Jar-Datei als Agent verwendet werden kann, muss sie gewisse bedingungen erfüllen:

- Die manifest Datei des Jar muss ein Attribut Premain-Class mit gültigem Wert haben.
- Die als Premain angegebene Klasse muss eine premain Methode implementieren.

In unserem falle implementiert die Agent Klasse die premain Methode. Auch muss beachtet werden das die verwendete JVM instrumentation unterstützt.

Das Funktionsprinzip eines Agents ist folgendermassen: Wird ein Java Programm zusammen mit einem Agent ausgeführt, wird der Agent vor dem eigentlichen Programm geladen. In diesem Agent wird die premain Methode ausgeführt, in der die möglichkeit besteht einen oder mehrer ClassFileTransformer anzugeben. Klassen die das ClassFileTranformer Interface implementieren können zur Laufzeit Klassen ändern. Dies geschieht dadurch das beim laden einer Klasse bei jedem angemeldetene Transformer die transform Methode aufgerufen wird. Innerhalb der Transform Methode bietet sich die

möglichkeit, die Klasse die aktuell geladen wird zu verändern. Die Methode liefert den Klassennamen sowie den Bytecode der Klasse als Byte Array. Die in "roher Form" gelieferte Klasse kann nun mit hilfe von Bytecode-Manipulations Bibliotheken wie BCEL, ASM oder Javassist bearbeitet werden. Nachdem alle modifikationen durchgeführt wurden, wird das modifizierte Byte Array als return Wert zurückgegeben.

In unserem Projekt verwenden wir einen Instrumentor Interface, um so einfach zwischen verschiedene Instrumentations Arten zu wechseln zu können.

Ohne irgendwelche Logik versucht die tranform Methode sämtliche Klassen zu instrumentieren, die geladen werden. Dies verursacht einige Probleme, deshalb war die verwendung einer Blacklist notwendig, um zu verhindern das problematische Klassen instrumentiert werden. In unserem Projekt können auszuschliesenden Klassen oder Pakete in einem Property File hinterlegt werden. Der Instrumentor mach zur laufzeit davon gebrauch und ignoriert Klassen und Pakete in der Blacklist.

Als problematische Klassen gelten zum einen gewisse Systemklassen, die sich nicht instrumentieren lassen, da sich die JVM (aus gutem Grunde) weigert sie zu instrumentieren. Hierzu gehören vorallem die Primitive Klassen, denn eine Veränderung dieser Klassen würde die Stabilität der JVM gefärden. [2] Es besteht die möglichkeit die Klassen in rt.jar zu diesem Zweck zu modifizieren, jedoch würde dies gegen die Nutzungsbedingungen von Oracle verstossen. [5] Zum anderen gibt es Klassen die man nicht zwingend instrumentieren will, z. B. Bibliotheken von drittanbietern. Weiter sollen auch die Klassen des Projekts nicht instrumentiert werden, da dies zu Problemen führt (z. B. StackOverflow, ConcurrentModificationException).

Der Agent verwendet standardmässig den RecordCollectorInstrumentor, der bei allen Klassen, die nicht auf der Blacklist sind, am Anfang und ende jeder Methode code einfügt. Der Code bewirkt das bei einem Methodenaufruf aktuelle informationen der Methode und der dazugehörigen Klasse aufgezeichnet wird.

Der Agent bringt auch einen ShutdownHook am System an. Hierbei handelt es sich um einen Thread, der ausgeführt wird wenn die JVM beendedt wird. Der ShutdownHook wird dazu verwendet um die gesammelten Daten

zu verarbeiten und zum Schluss die erzeugten Dateien auf einen Massenspeicher zu schreiben.

Vor der verarbeitung muss jedoch der am Anfang hinzugefügte Transformer wieder entfernt werden, da sonst für die Verarbeitung verwendete Klassen Instrumentiert, was zu Problemen führt.

6.2.3 Der Record Collector Instrumentor

Für die folgende Aufzeichnungsphase 6.2.3 werden die Klassen durch den Agent mit dem Record Collector Instrumentor instrumentiert. Seine Aufgabe ist es, die Klassen und Methoden so zu ergänzen, dass sie bei Aufruf einen MCR mit den Daten, die von Interesse sind anlegen.

Der RCI erzeugt dazu in jeder instrumentierten Klasse zwei zusätzliche Felder, eines vom Typ MethodCallRecord und eines vom Typ RecordCollection. Zu Beginn der instrumentierten Methode wird nun Code eingefügt, der einen neuen Entry-MCR erzeugt und den Method Owner, die Method Owner ID, den Methodennamen, sowie die übergebenen Parameter darin ablegt:

```
...                                                          1
codeInsertBefore = new StringBuilder("");                    2
codeInsertBefore.append(createNewEntryRecord);               3
codeInsertBefore.append(collectMethodOwner);                 4
codeInsertBefore.append(collectMethodOwnerId);               5
codeInsertBefore.append(InstrumentationUtil.                 6
        getCodeForParameterCollection(method));              7
codeInsertBefore.append(collectMethodName);                  8
                                                             9
method.insertBefore(codeInsertBefore.toString());            10
...                                                          11
```

Auf den durch die Instrumentierung zugefügten Code folgt der originale Inhalt des Methodenkörpers. Vor dem Verlassen der Methode wird wiederum zusätzlicher Code eingefügt. Dies ist erforderlich, da erst hier der Rückgabewert der Methode erfasst und im MCR gespeichert werden kann. Zusätzlich werden hier auch die Felder des Method Owners erfasst und daraus im MCR direkt der State-Hash berechnet. Der im MCR gespeicherte Objektzustand entspricht also immer dem Objektzustand nach Ausführung

der Methode, die im MCR festgehalten wurde. Dies entspricht dem Post-Call-State. Der Entry-MCR ist nun vollständig und kann zur Record Collection hinzugefügt werden. Zuletzt wird noch der Exit-MCR erstellt und ebenfalls in die Collection eingefügt.

```
...                                                          1
codeInsertAfter = new StringBuilder("");                     2
codeInsertAfter.append(collectReturnValue);                  3
codeInsertAfter.append(collectClassFields);                  4
codeInsertAfter.append(setMethodOwnerState);                 5
codeInsertAfter.append(addRecordToCollection);               6
codeInsertAfter.append(createNewExitRecord);                 7
codeInsertAfter.append(addRecordToCollection);               8
                                                             9
method.insertAfter(codeInsertAfter.toString());              10
...                                                          11
```

Die eingefügten Code-Statements werden mit Hilfe der InstrumentationUtil-Klasse generiert, die wiederum stark angelehnt ist an die JavassistHelper-Klasse, die uns zu Beginn dieser Arbeit zur Verfügung gestellt wurde.

Mögliche Erweiterung: Elegantere Codeerzeugung. Das Arbeiten mit Strings die Code-Sequenzen beinhalten ist aus mehreren Gründen unelegant. Dieser Code wird durch die IDE nicht mit Syntax-Highlighting versehen und kann auch nicht automatisiert unbenannt werden. Ein möglicher Ansatz wäre hier, diese Code-Fragmente als echten Code in einer Klasse zu schreiben und nur zu Instrumentierung in einen String umzuwandeln. Die Klasse CodestringUtil zeigt auf, wie dies funktionieren könnte, ist jedoch derzeit noch nicht eingesetzt worden.

6.3 Die Aufzeichnungsphase

Nachdem die Applikation unter Test wie im vorherigen Abschnitt beschrieben durch den Agent und Record Collector Instrumentor (RCI) instrumentiert worden ist, wird der instrumentierte Applikations-Code nun ausgeführt. Die Applikation kann dabei normal produktiv eingesetzt werden, wie es ihrem Bestimmungszweck entspricht. Im Hintergrund sammelt der RCI die Daten, die später für das automatische Generieren von Unittests

und Mock-Klassen benötigt werden. Im folgenden Abschnitt wird der Method Call Record (MCR) genauer beschrieben, der die Kapselung der Datensätze übernimmt.

Mögliche Erweiterung. Für diese Arbeit wurden die Datensätze als Java-Objekte im Speicher gehalten. Dies ermöglicht einen schnellen und einfachen Zugriff. Bei komplexeren Programmen und längerer Aufzeichnungszeit werden aber voraussichtlich Datenmengen anfallen, die eine Datenbank erforderlich machen. Die Persistenz, die eine Datenbank bietet, würde es zudem erlauben, die Unittests nach Belieben aus den aufgezeichneten Daten zu generieren. Als erster Zwischenschritt könnte dies auch über die Serialisierung der Record Collection geschehen.

6.3.1 Der Method Call Record

Wie im vorhergehenden Abschnitt beschrieben, kapselt der MCR die Daten, die bei der Instrumentierung gesammelt werden. Nachfolgend werden die Felder und Funktionen des MCR genauer erläutert.

6.3.2 Record Number

Jedem MCR wird bei der Erzeugung eine eindeutige, fortlaufende Nummer zugewiesen, die Record Number. Die Abfolge dieser Nummern entspricht exakt der zeitlichen Abfolge der Methodenaufrufe, bzw. Methodenaustritte.

Einschränkung. Dieses Verhalten ist nur für singlethreaded Anwendungen gewährleistet.

6.3.3 Method Name

Der Name der aufgerufenen Methode.

6.3.4 Method Owner

Der voll qualifizierte Name (FQN) des Objektes, auf dem die Methode aufgerufen wurde.

6.3.5 Method Caller

Der voll qualifiziert Name (FQN) des Objektes, das den Methodenaufruf ausgelöst hat. Dieser Wert hat im Moment nur informativen Charakter und wird nicht weiter verwendet.

6.3.6 Record Type

Der Record Type ist eine Enumeration, die zwei Werte annehmen kann, Entry oder Exit. Während der Entry-MCR alle hier beschriebenen Daten aufnimmt, dient der Exit-MCR lediglich als Markierung. Aus der Abfolge der beiden MCR Typen kann so genau bestimmt werden, auf welcher Verschachtelungstiefe die Methodenaufrufe erfolgt sind. Dies ist insbesondere für die Erzeugung von Mock-Klassen von Bedeutung (siehe Abschnitt 6.6.2).

6.3.7 Rückgabewerte

Für die Rückgabewerte sind neun Felder vorgesehen, je eines für jeden primitiven Datentyp in Java und eines für String-Werte. Die Ein- und Ausgabe des Wertes erfolgt als Object-Typ. Das Object wird dann auf den spezifischen Datentyp gecastet.

Erweiterung. Diese Art der Speicherung von Returnwerten ist aus dem Gedanken an Datenbank-Felder entstanden, bei denen der Typ des gespeicherten Wertes festgelegt werden muss. In der jetzigen Form liesse sich der Rückgabewert auch schlicht in einem Feld vom Typ Object speichern, was die restlichen 8 Felder obsolet machen würde.

6.3.8 Parameterwerte

Um die variable Anzahl von übergebenen Parametern aufzunehmen, stehen neun HashMaps der oben erwähnten Datentypen zur Verfügung, sowie eine Reihe von Methoden, um die gewünschten Parameter einfach aus dem MCR auszugeben. Da auf diese Weise jedoch die für eine Methode (in Verbindung mit ihrem Namen) charakteristische Reihenfolge der Parameter verloren geht, wird ein zum MCR hinzugefügter Parameter auch noch in einer Object-Liste gespeichert.

Erweiterung. Wie bei den Rückgabewerten auch wäre hier eine vereinfachte Speicherung (Verzicht auf die HashMaps) denkbar.

6.3.9 Feldvariablen

Die Feldvariablen werden wie auch die Parameter in Hashmaps festgehalten, deren Schlüsselwert vom Typ String ist und den Feldnamen trägt, während der Wertetyp einer der acht primitiven Wertetypen oder String ist und den Feld-Wert enthält.

6.3.10 Method Owner State

Der Zustand, in dem sich ein Objekt befindet, kann den Rückgabewert einer Methode beeinflussen. Dies wird sofort klar, wenn man eine z.B. eine Getter-Methode betrachtet, die ein Feld der Klasse zurückgibt. Der Rückgabewert der Methode ist direkt vom Zustand des Objekts abhängig:

```
public class Switch {                                    1
        boolean isOn = true;                             2
        public boolean isOn(){                           3
                return isOn;                             4
        }                                                5
}                                                        6
```

Anders gesagt erwarten wir genau dann, dass die Methode bei gleichen Input-Argumenten den gleichen Rückgabewert liefert, wenn der Zustand des Objekts, auf dem die Methode aufgerufen wird, derselbe ist. Um zwei Zustände einfach vergleichen zu können wird über die Felder des Objektes ein 256 Bit SHA-Hash berechnet und ebenfalls im MCR gespeichert. Hierbei handelt es sich nicht um einen sogenannten perfect hash"[8], das heisst, es ist theoretisch nicht auszuschliessen, dass zwei unterschiedliche Zustände denselben Hash-Wert liefern. Die Wahrscheinlichkeit, dass dieser Fall – Kollision genannt – eintritt ist jedoch auch bei einer grossen Anzahl von Objekten als extrem unwahrscheinlich einzustufen. Aus diesem Grund wurde in dieser Arbeit keine Kollisions-Behandlung vorgesehen. Es wäre jedoch denkbar, auf Kollisionen zu prüfen und eine entsprechende Exception zu werfen.

6.3.11 Method Owner ID

Der Object-State alleine lässt noch keine Aussage über die Identität der Objekt-Instanz zu. Zwei Instanzen derselben Klasse im selben Zustand wären nicht unterscheidbar. Im MCR wird deshalb zusätzlich zum Object-State noch ein ID-Hash des Objekts gespeichert, auf dem der Methodenaufruf erfolgt ist (methodOwnerId). Die Bestimmung dieser ID erfolgt über die die Methode System.identityHashCode(Object x). Diese Methode liefert denselben Hashcode, den Object.hashCode() zurückgibt, auch wenn ein Objekt die .hashCode() Methode überschrieben hat. Dieser Hash liefert unterschiedliche Hashes für zwei verschiedene Instanzen mit demselben State. Dieses Verhalten ist jedoch nur während einer (1) Laufzeit eines Java-Programmes zugesichert, d.h. Die ID persistiert nicht über einen Neustart des Programmes oder der Virtual Machine.
[6] "As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects."
[6] "Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer."

6.3.12 Die Record-Collection

Die Record Collection ist eine Sammlung aller aufgezeichneten Records. Zentraler Teil dieser als Singleton implementierten Klasse ist die Liste von allen aufgezeichneten Records, die sie führt.

Mit Hilfe des RecordComparators, der zwei Records nach ihrer Record Number vergleicht, lässt sich die Liste sortieren und dann z.B. über die Hilfsklasse RecordDisplay als Konsolen-Output anzeigen.

Weiter bietet die Record Collection ein Satz von Methoden an, um ein gewünschtes Subset aller Records zu erhalten. Insbesondere für die Erstellung von Unittests und Mocks können so alle Records, die zu einem bestimmten Method Owner oder einer bestimmten Methode gehören, abgefragt werden. Eine weitere Methode ermöglicht es, den Record mit dem Pre-Call-State zu finden, wenn die Record Number des Records, der den Post-Call-State beinhaltet bekannt ist.

6.4 Instrumentation von Basisklassen

Die Überlegung auch die Basisklassen von Java zu instrumentieren kam daher, dass sämtliche Aufrufe aufgezeichnet werden um alle Abhängigkeiten festzuhalten. Ein Grund hierfür wäre zum Beispiel die Verwendung der folgenden Funktionen:

```
int cores = Runtime.getRuntime().availableProcessors(1);
double rand = Math.random();                              2
```

Laut der Java Doc für Intrumentation ist das instrumentieren von Primitive Klassen und Arrays nicht möglich.[2]

Eine andere Möglichkeit Basisklassen zu instrumentieren wäre, einen alternativen Classpath anzugeben, der die modifizierten Klassen enthält (rt.jar), was jedoch gegen die Nutzungsbestimmungen von Orcale verstösst.[5]

Um zu sehen ob es noch weiter Klassen gibt, die sich nicht instrumentieren lassen, war der nächste Schritt dies selbst zu testen. Hierzu wurde Code geschrieben der nach dem Laden des Agent versucht sämtliche bereits geladenen Klassen mit Instrumentation.retransformClasses(Class ...) neu zu instrumentieren. In den ausgeführten Versuchen konnten 827 von 904 geladenen Klassen neu transformiert werden, jedoch war bei wichtigen Klassen wie String oder Integer die Transformation nicht möglich.

6.5 Erzeugen von UnitTests

Die Erzeugung der Unittests findet beim Beenden der JVM statt, im Shutdownhook wird die Erzeugung ausgelöst. Die ganze Bearbeitung und Erzeugung der Testcases findet in der TestcaseAssembler Klasse statt.

Die Klasse TestcaseAssembler koordiniert die Erzeugung von Testcases aus Method Call Records. Als erstes werden sämtliche Method Call Records nach ihren Klassen sortiert, so dass nachher jede Klasse einzeln abgearbeitet werden kann. Anschliessend wird jeder Satz von Method Call Records zu einem Testcase verarbeitet. Während der Bearbeitung ist ein Testcase eine eigene Klasse die sämtliche Daten über einen Testcase festhält (Testcase Name, Package Name, Imports, KlassenFelder und Tests). Des weiteren ist die Testcase Klasse dafür verantwortlich, die einzelnen Elemente korrekt zusammenzufügen und am Ende einen String zurückzugeben, der einen gültigen

Testcase in Java repräsentiert.

Für jede Klasse werden folgende Schritte ausgeführt:

1. Erstellung des Klassennamen
2. Erstellung der Setup Methode
3. Erstellung der Klassenfeld für die Klasse unter Test
4. Generierung der Testmethoden

Die Testmethoden stellen sich aus folgenden Elementen zusammen:

- Einem eindeutigen Test-Methodenamen (wird fortlaufend nummeriert)
- Setzen des aktuellen State, damit sich die Mock Objekte korrekt verhalten
- Setzen des States der Klasse unter Test mittels Setter-Methoden
- Assert Methoden zum Vergleich des erwarteten State der Klasse unter Test

Nachdem alle Testcases erstellt worden sind, werden sie in die entsprechende Dateien auf dem Massenspeicher geschrieben.

6.5.1 Testklasse

Für jeden Method Owner wird nun eine Testklasse erzeugt. Der Name der Testklasse besteht aus dem Namen des Owners mit angehängtem 'Test', siehe Zeile 1 des folgenden Code-Listings. Weiter wird ein Feld namens 'testee' vom Datentyp des Owners angelegt. Dieses Feld nimmt das Objekt auf, auf dem die Testfälle ausgeführt werden.

6.5.2 Setup-Methode

Für jede Test-Klasse wird eine Setup-Methode erzeugt. Sie wird mit der JUnit-Annotation '@Before' versehen, so dass sie vor jeder Ausführung einer Testmethode ausgeführt wird und die nötigen Initialisierungen durchführt. Diese Initialisierung besteht darin, dem Testklassen-Feld 'testee' jeweils eine neue Instanz zuzuweisen.

```
public class FooTest {                                  1
        Foo testee;                                     2
                                                        3
        @Before                                         4
        public void setup() {                           5
                testee = new Foo();                     6
        }                                               7
                                                        8
        ...                                             9
}                                                       10
```

6.5.3 Test-Methoden

Für jeden aufgezeichneten Methodenaufruf wird eine (1) Testmethode erstellt. Dies entspricht den Anforderungen für sauberen Testcode[18]. Ein fehlgeschlagener Test lässt sich so einfach auf ein einzelnes Assert-Statement zurückführen. Um die Einzigartigkeit der Namen der Testmethoden zu gewährleisten, wird eine Laufnummer an den Methodennamen angehängt.

Innerhalb der Testmethode muss nun der Testee in den Object-State versetzt werden, den er vor dem Methodenaufruf hatte (vergleiche 6.3.9). Um den MCR zu finden, der diesen Pre-Call-State enthält, muss derjenige Entry-MCR gesucht werden, der die grösste Record Number hat, die kleiner ist als die Record Number des MCRs, auf der die Testmethode basiert (vergleiche 6.3.1). Etwas einfacher ausgedrückt: Während der für den Test betrachtete MCR den Zustand nach dem zu betrachtenden Methodenaufruf enthält, enthält der nächst-tiefere Entry-MCR den Zustand nach der zuvor aufgerufenen Methode, also den Pre-Call-State der aktuell zu bearbeitenden Methode.

Aus den gespeicherten Feldern werden nun Setter-Statements generiert, die den Testee in den gewünschten Zustand bringen(Pre-Call-State). Als nächsten können die Assert-Statements generiert werden. Hat die Methode einen Rückgabewert, wird ein Assert-Statement eingefügt, das überprüft ob der Rückgabewert für die gespeicherten Parameter demjenigen entspricht, der in der Aufzeichnungsphase festgehalten wurde. Schliesslich wird in einem weiteren Assert-Statement geprüft, ob der Zustand des Testee nach dem

Methodenaufruf dem aufgezeichneten Post-Call-State entspricht.

Die Testmethode kann so überprüfen, dass ein Methodenaufruf auf einem Objekt mit definiertem Zustand den erwarteten Rückgabewert liefert und das Objekt in den erwarteten Post-Call-State überführt.

Anschauungs-Beispiel Die zu testende Klasse Foo besitzt ein einziges Feld namens 'state' das ihren inneren Zustand repräsentiert. Es gibt Setter- und Getter-Methoden für dieses Feld.

```
public class Foo {                                          1
        int state = 0;                                      2
                                                            3
        public int getState(){                              4
                return state;                               5
        }                                                   6
        public setState(int value){                         7
                this.state = value;                         8
        }                                                   9
}                                                           10
```

Nehmen wir an, dass ein früherer Methodenaufruf auf eine Foo-Instanz diese in den Zustand 'state = 5' gesetzt hat. Darauf folgt ein Aufruf der Methode getState() zu dem nun eine Testmethode generiert werden soll. An den Namen der Testmethode wird eine Laufnummer angehängt (Zeile 2). Die Testinstanz 'testee' von Foo wird in den Zustand vor dem Aufruf von getState() versetzt (Zeile 3). Der erwartete Wert, der in einem MCR aufgezeichnet wurde, wird mit dem tatsächlichen Rückgabewert verglichen, den testee.getState() liefert (Zeile 4). Es wird überprüft, ob 'testee' nach dem Methodenaufruf den im MCR aufgezeichneten State eingenommen hat. Der Vergleich der States erfolgt über den Vergleich der SHA-256 Hashes (Zeile 5/6).

```
@Test                                                       1
public void testgetState0() {                               2
        testee.setState(5);                                 3
        assertEquals(testee.getState(), 5);                 4
        assertEquals(TesteeHashUtil.getTesteeHash(testee),  5
                "e93fbd66b731....");                        6
}                                                           7
```

6.6 Erzeugen von Mock-Klassen

Um die einzelnen Teile der Software unabhängig zu testen, werden in der Regel Mock-Objekte verwendet, um Abhängigkeiten zu entkoppeln. Ursprünglich war geplant, die Mocks mit Hilfe von Mockito zu erzeugen, jedoch hat sich bei der PA-Sitzung vom 3.12.2013 herausgestellt, dass dieser Ansatz nicht funktionieren wird. Grund dafür ist, dass lokal in einer Methode ein Objekt erzeugt und verwendet werden kann. Mit Mockito wäre es unmöglich solche Objekte zu mocken. Es musste zwangsweise ein neuer Ansatz für das Mocking gewählt werden.

Der aktuell implementierte Ansatz beruht darauf, dass programatisch eine Java Quelltext Datei erzeugt wird, die sämtliche Methoden der Orginalklasse nachbildet. In den einzelnen Methoden wird eine Switch-Case-Struktur erzeugt, welche je nach Zustand des orginal Objektes einen anderen Return-Wert liefert. Akzeptiert die Methode Parameter, wird innerhalb des Case Blocks eine If-Else-Struktur erzeugt, welche anhand der Parameter-Kombinationen entsprechende Return-Werte liefert. Beide Strukturen werden so generiert, dass sie eine Exception auslösen wenn unbekannte Zustände oder Parameter-Kombinationen auftreten, da in solchen Fällen keine korrekten Return-Werte zurückgegeben werden können.

6.6.1 Ungelöste Probleme

Zur Zeit werden die Mock Objekte als Quelltext-Dateien abgelegt und sind in dieser Form nicht direkt für Tests zu gebrauchen.

Abhängigkeit von CurrentState Klasse, welche verwendet wird um den Mock-Objekten den aktuellen Zustand mitzuteilen. Die Klasse ist im Grunde ein statisches String Feld, auf das sämtliche Tests und Mocks zugreifen können. Ein Unittests setzt vor dem Ausführen eines Tests den entsprechenden State in der CurrentState Klasse. Mocks überprüfen den State in der CurrentState Klasse und geben den entsprechenden Return-Wert zurück.

Überladene Methoden können nicht unterschieden werden. Da Methoden nur anhand ihres Namens unterschieden werden und nicht anhand ihrer Signatur.

6.6.2 Mögliche Lösungsansätze

Mock Objekte Nach der Erzeugung der Quelltext Dateien könnten diese kompiliert werden, z.B. durch Aufruf des Java-Compilers. Dies ist jedoch fehleranfällig und setzt voraus, dass eine Java-SDK auf dem System installiert ist.

Eine bessere Alternative wäre, mit Hilfe von Javassist den Inhalt der Methoden mit dem erzeugten Code zu ersetzen und die modifizierte Klasse danach auf den Massenspeicher zu schreiben.

Eine weiteres Problem das beide Ansätze betrifft ist das Laden der modifizierten Klassen beim Ausführen der Unittests. Java erlaubt es nicht verschiedene Versionen einer Klasse gleichzeitig zu laden, somit wäre die übliche Art, alle Tests mittels TestSuits auszuführen, nicht möglich, da nachdem eine Klasse geladen und getestet wurde, das Mock dieser Klasse in einem anderen Test zu einem Laufzeitfehler führen würde. Dies passiert, weil alle Tests in der gleichen JVM ausgeführt werden und die einzige Möglichkeit Klassen garantiert zu entladen ist, den Classloader zu zerstören, der die Klasse geladen hat. Im Normalfall lädt der SystemClassloader sämtliche Klassen, jedoch führt eine Zerstörung dieses Classloaders zum Absturz der JVM. Eine Lösung hierfür wäre, jeden Test einzeln ausführen. Dies wäre jedoch für eine grössere Anzahl Tests sehr aufwändig und zeitintensiv. Eine bessere Alternative wäre die Verwendung eines eigenen Classloaders, welcher die entsprechenden Mocks lädt und am Ende des Tests ohne Bedenken wieder zerstört werden kann.

6.6.3 Verschachtelte Methodenaufrufe und lokale Variablen in Methoden

Anhand des Beispiels in Anhang19.2.2 soll aufgezeigt werden, wie die für einen Unit-Test mit Mock-Objekten benötigten Daten gesammelt werden können, wenn eine Methode lokale Variablen und weitere Methodenaufrufe enthält. Der Unittest soll für die Klasse Foo erstellt werden. Foo enthält eine Methode addTen(), die lokale Variable x sowie zwei Methodenaufrufe auf dem Objekt 'bar'. 'bar' selbst ruft auf einem dritten Objekt namens 'baz' eine Methode auf. Um die Foo-Klasse zu testen, muss ein Mock-Objekt

erzeugt werden, das sich wie die Umgebung von Foo verhält. Konkret muss das Mock also Bar und Baz simulieren. Die internen Details von Bar und Baz müssen jedoch nicht bekannt sein. Es genügt sich diejenigen von Foo anzuschauen.

Lokale Variablen in Methoden. Javassist erlaubt es nicht auf lokale Variablen in Methoden zuzugreifen [10]. Auch mit ASM ist es nur möglich via MethodVisitor.visitLocalVariable(...) auf die Deklaration der lokalen Variable zuzugreifen, nicht aber auf ihren aktuellen Wert z.B. nach Inkrementation in einer Schleife [1]. Es bliebe also noch der Weg, diese Werte direkt aus dem Methoden-Stackframe im Bytecode zu lesen. Eine Möglichkeit dies mit dem Java Debug Interface (JDI) zu lösen wurde von Wayne Adams beschrieben [7]. Die Zeilennummern, an denen die Methodenaufrufe erfolgen, liessen sich mit BCEL ermitteln, so dass die Breakpoints entsprechend gesetzt werden können [4].

Nach der Ansicht der Autoren ist es jedoch gar nicht nötig, die Werte lokaler Variablen im Methoden-Scope zu kennen. Lokale Variablen in Methoden sind bei jedem Aufruf gleich und müssen deshalb für den Unittest nicht bekannt sein. Die Betrachtung von Input und Output einer Methode wird durch die lokale Variable nicht beeinflusst.

Im Anschauungs-Beispiel im Anhang (siehe 9.2.2) ist dies gut ersichtlich. Für jeden Parameter der in die Methode gegeben wird, wird der Rückgabewert um 10 erhöht sein. Nimmt man nun z.B. x = 5 an, ist der Rückgabewert jeweils um 11 inkrementiert. Der Methodennamen ist nun zwar irreführend, aber die Methode wird wiederum ein konstantes Verhalten aufweisen und damit testbar sein.

Mögliche Erweiterung. Automatisches Laden von Mocks für Unittests. Die generierten Mock-Klassen werden derzeit als Java-Sourcecode-Files gespeichert und müssen bei Bedarf von Hand kompiliert und geladen werden. In einem weiteren Schritt wäre es möglich, vor einem Unittest über einen eigenen ClassLoader die zu mockenden Klassen durch die Mock-Klassen zu ersetzen. Dabei ist zu beachten, dass die Klasse, die getestet wird nicht ersetzt werden darf.

7 Resultate und Ausblick

Wir konnten mit dieser Arbeit zeigen, dass Bytecodeinstrumentation das Potential hat, automatisches Unittesting mit Laufzeitdaten zu realisieren. Aufgrund der zahlreichen Einschränkungen ist die Instrumentierung noch nicht auf konkrete, produktive Programme anwendbar. Die Arbeit liefert jedoch hilfreiche Erkenntnisse und Ansätze um dies künftig zu ermöglichen. So konnten wir Lösungswege für die Identifizierung von Objektinstanzen und Objektzuständen aufzeigen. Eine Architektur zur Aufzeichnung und Speicherung der benötigten Datensätze ist implementiert worden. Aus diesen Daten können automatisiert lauffähige Unittests erzeugt und gespeichert werden, die die Korrektheit von Rückgabewerten abhängig vom Objektzustand überprüfen, sowie eine etwaige Zustandsänderung, die ein Methodenaufruf auf einem Objekt auslöst. Mit der Generierung von Mock-Klassen konnten wir einen Weg aufzeigen, Abhängigkeiten in der KUT aufzulösen, und die Komplexität, die das Serialisieren oder Nachverfolgen aller Abhängigkeiten bedeutet hätte, zu vermeiden. Damit ist die Grundlage für einen wichtigen nächsten Schritt gelegt: Die Behandlung von nicht primitiven Datentypen

Nachfolgend wird genauer auf bestehende Einschränkungen und sinnvolle künftige Erweiterungen des bestehenden Projekts eingegangen.

7.1 Zugriff über Getter- und Setter-Methoden

Eine Einschränkung / Bedingung dieses Projektes ist, dass sämtliche Felder der Klassen über Setter- und Getter-Methoden zugänglich sein müssen. Dies ist notwendig, da für einen Test die KUT in einen definierten Zustand gebracht werden muss. Ein Ansatz ist die Verwendung von Reflection um auch Felder zu setzen, die nicht public sind. Da aus den Records die Feldernamen bekannt sind, sollte dies einfach zu realisieren sein.

7.2 Statische Methoden

Statische Methoden stellen zur Zeit immer noch ein Problem dar, da sie sich nicht instrumentieren lassen. Eine mögliche Ursache ist, dass wir Felder in die instrumentierten Klassen einfügen um die Daten aufzuzeichnen, das für

die Klasse zur der die statische Methode gehört, keine Instanz besitzt.

Eine Möglichkeit wäre, die Verwendung von lokalen Variabeln um die Daten aufzuzeichnen, so könnte auf die Verwendung von Klassen-Feldern verzichtet werden. Möglicherweise gibt es jedoch Limitationen mit der verwendeten Javassist Bibliothek, nämlich das man nicht auf lokale Variablen von anderen Methoden zugreifen kann.[12] Eine Alternative wäre, in den eingefügten Code-Blöcken die direkte vVrwendung eines Singleton, so könnte auf Felder sowie auf lokale Variabeln verzichtet werden.

7.3 Erweiterung für Objekte

Zur Zeit kann Projekt keine Objekte als Parameter oder Felder behandeln mit Ausnahme von Strings.

7.4 Bildung von Äquivalenzklassen

Im Moment werden die aufgezeichneten Daten direkt für die Bildung von Assert-Statements und Mockobjekten verwendet. Dies bedeutet, dass Daten, die zwar mit grosser Wahrscheinlichkeit valide sind, aber nicht in einem Record aufgezeichnet wurden, den Unittest fehlschlagen lassen. Die Tests sind damit für reale Applikationen mit realen Daten zu rigide. Hier könnte mit der Bildung von Äquivalenzklassen aus den aufgezeichneten Daten Abhilfe geschaffen werden. In Äquivalenzklassen fasst man Werte zu gültigen oder ungültigen Wertebereichen zusammen. So könnte man z.B. durch Analyse der Records für einen Rückgabewert vom Typ Integer feststellen, dass dieser stets positiv und kleiner 1'000 ist und den Test entsprechend anpassen. Das automatisierte Finden passender Äquivalenzklassen stellte eine eigene anspruchsvolle Aufgabe dar, deren Lösung etliche positive Auswirkungen auf das Testing haben könnte. Die Bildung von Äquivalenzklassen würde die Anzahl Tests stark reduzieren und damit den Testvorgang performanter machen. Weiterhin würde die Zahl der falsch-positiven Tests, also derjenigen Tests, die fehlschlagen, obwohl kein Fehler vorliegt, reduziert.

7.5 Multithreaded Applikationen

Wie in Abschnitt 6.3.1 angedeutet, wurde diese Arbeit auf singlethreaded Anwendungen eingeschränkt. Eine Erweiterung auf multithreaded Applikationen würde einige Herausforderungen mit sich bringen. In einem ersten Schritt müsste für die Records festgehalten werden, in welchem Thread ein Aufruf erfolgte. Die Zuweisung der Record Number müsste synchronisiert erfolgen oder die Records neu über einen Zeitstempel eindeutig identifiziert werden.

7.6 Überladene Methoden

Als überladene Methoden oder englisch 'method overloading' bezeichnet man Methoden, die den gleichen Namen tragen, sich jedoch in Anzahl und/oder Typ der Parameter unterscheiden[3]. Die Unterscheidung der Methoden zur Test- und Mockgeneration erfolgt bisher nur über die Methodennamen, wodurch überladene Methoden nicht korrekt behandelt werden. Diese Limitation lässt sich auf einfache Weise beheben, indem für die Unterscheidung der Methoden zusätzlich die Parameterliste (siehe 6.3.7) miteinbezogen wird.

7.7 Laufzeitüberwachung

Wie im Kurzbeschrieb(9.1.1) bereits erwähnt, könnte aufbauend auf dieser Arbeit die Überwachung einer Applikation zur Laufzeit implementiert werden. Methodenparameter, Rückgabewerte und Objektzustände würden jeweils bei einem Methodenaufruf mit den bereits aufgezeichneten Werten bzw. Äquivalenzklassen verglichen und gegebenenfalls eine Warnung bei einer auffälligen Änderung ausgegeben. Die Umsetzung einer solchen Funktionalität setzt allerdings noch viel Forschungs- und Experimentier-Arbeit voraus. Insbesondere müsste die Performance so gut sein, dass die überwachte Applikation weiterhin problemlos funktioniert. Auch die fortlaufende Bildung bzw. Anpassung der Äquivalenzklassen sowie die Erkennung von Äusreissernßind noch ungelöste Probleme.

7.8 Vererbung und Generics

Ein Themengebiet, das für die objektorientierte Sprache Java zentral ist, haben wir in dieser Arbeit ausgeklammert: Die Vererbung. Gemäss unseren Recherchen bietet Javassist einfachen Zugriff auf geerbten Felder und Methoden über die transformierte Klasse (CTClass), doch wäre es hier sicher wünschenswert, einen lauffähigen Prototypen als Proof-Of-Concept zu erstellen. Die Schwierigkeiten liegen hier eventuell im Detail. So wird zum Beispiel bei der Modifizierung einer geerbten aber nicht überschriebenen Methode in einer Subklasse auch die entsprechende Methode der Superklasse modifiziert [13].
Eine Erweiterung zur korrekten Behandlung von generischen Datentypen oder Klassen ist problemlos möglich. Die generischen Typen sind über Reflection zugänglich [17] und Javassist bietet die Möglichkeit, transformierte Klassen mit generischen Signaturen zu erzeugen [11]

8 Verzeichnisse

8.1 Glossar

Glossary

Agent Spezielle Java Programme, die andere Java Programme zur Laufzeit verändern können.. 14

ASM ASM ist eine Klassenbibliothek zur Java Bytecode Manipulation.. 15, 28

BCEL Acronym für The Byte Code Engineering Library". Eine API zur Bytecode-Manipulation von Java Klassen.. 15, 28

Classpath Ein Classpath gibt an, wo die JVM respektive der Classloader nach Klassen Dateien sucht.. 22

Entry-MCR Dieser MCR wird beim Eintritt in die instrumentierte Methode angelegt. Er enthält alle aufzuzeichnenden Daten, insbesondere auch den Object-State NACH dem Methodenaufruf.. 19

Exit-MCR Dieser MCR wird beim Verlassen der instrumentierte Methode angelegt. Er enthält keinerlei Daten sondern dient lediglich als Markierung für die Abfolge der Methodenaufrufe.. 19

FQN Acronym für Fully Qualified Name, zu deutsch: voll qualifizierter Name. Hier verwendet für Objektnamen. Objektnamen sind fully qualified wenn sie eindeutig sind, d.h. mit ihrem vollständigen Paket-Pfad angegeben sind.. 18, 19

Javassist Javassist (Java Programming Assistant) ist eine Klassenbibliothek, die es ermöglicht, Bytecodemanipulation in Java durchzuführen.. 15, 28, 30

KUT Acronym für 'Klasse unter Test'. Dies bezeichnet diejenige Klasse, die mit einem Unittest getestet werden soll.. 13, 29

MCR Acronym, siehe Method Call Record.. 18

Method Call Record Diese Klasse kapselt den Datensatz, der in der Aufzeichnungsphase über einen Methodenaufruf angelegt wird.. 13, 18, 22

Method Owner Die Objektinstanz, auf der eine Methode aufgerufen wurde.. 16, 21, 23

Post-Call-State Der interne Zustand einer Objekt-Instanz nach dem Aufruf einer Objekt-Methode. 17, 21

Pre-Call-State Der interne Zustand einer Objekt-Instanz vor dem Aufruf einer Objekt-Methode. 21, 24

Proof-Of-Concept Machbarkeitsnachweis, in der Regel durch einen Prototypen erbracht.. 32

RCI Acronym, sie Record Collector Instrumentor.. 17

Record Ein Datensatz. Hier ist, falls nichts anderes erwähnt, jeweils ein Method Call Record gemeint.. 13

Record Collection Die Record Collection verwaltet alle MCR's die während eines Aufzeichnungslaufs erstellt werden.. 18

Record Collector Instrumentor Eine Klasse die mit Hilfe des Agents andere Klassen so instrumentiert, das bei Methodenaufrufen und Methodenaustritten Records generiert werden.. 17

Record Number Laufnummer die einen Record eindeutig identifiziert.. 21

Reflection Das Auslesen und Modifizieren von Objekten und ihren Eigenschaften zur Laufzeit.. 6, 32

8.2 Literaturverzeichnis

[1] Asm documentation. http://asm.ow2.org/asm40/javadoc/user/org/objectweb/asm/MethodVisitor.html#visitLocalVariable. Accessed: 11/11/2013.

[2] Instrumentation (java platform se7). http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html#isModifiableClass%28java.lang.Class%29. Accessed: 12/12/2013.

[3] The java tutorial. overloading methods. http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html. Accessed: 10/12/2013.

[4] Javaseiten.de. 4.4 bytecode engineering library. http://asm.ow2.org/asm40/javadoc/user/org/objectweb/asm/MethodVisitor.html#visitLocalVariable. Accessed: 06/11/2013.

[5] Oracle binary code license agreement for the java se platform products and javafx. http://www.oracle.com/technetwork/java/javase/terms/license/index.html. Accessed: 12/12/2013.

[6] Oracle java^{TM} platform, standard edition 7 api specification. http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode. Accessed: 23/11/2013.

[7] Wayne adams' blog. breakpoint processing in jdi. http://wayne-adams.blogspot.ch/2011/11/breakpoint-processing-in-jdi.html. Accessed: 4/12/2013.

[8] Wikipedia article "perfect hash". https://en.wikipedia.org/w/index.php?title=Perfect_hash_function&oldid=566904486. Accessed: 16/11/2013.

[9] Martin Aspeli. Untested code is broken code: test automation in enterprise software delivery. http://www.deloittedigital.com/eu/blog/untested-code-is-broken-code-test-automation-in-enterprise-software-deliver. Accessed: 16/12/2013.

[10] Shigeru Chiba. Javassist tutorial. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial2.html`.

[11] Shigeru Chiba. Javassist tutorial. generics. `http://tutorials.jenkov.com/java-reflection/generics.html#general`. Accessed: 15/12/2013.

[12] Shigeru Chiba. Javassist tutorial. introspection and customization. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial2.html#before`. Accessed: 15/12/2013.

[13] Shigeru Chiba. Javassist tutorial. introspection and customization. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial2.html#intro`. Accessed: 15/12/2013.

[14] Daniel Dunbar et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[15] Sai Zhang et al. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 353–363. ACM, 2011.

[16] Prof Thomas Huckle. Collection of software bugs. `http://www5.in.tum.de/~huckle/bugse.html`. Accessed: 03/12/2013.

[17] Jakob Jenkov. Java reflection: Generics. `http://tutorials.jenkov.com/java-reflection/generics.html#general`. Accessed: 15/12/2013.

[18] Robert C. Martin. *Clean Code - Refactoring, Patterns, Testen und Techniken fuer sauberen Code*. mitp-Verlag, Heidelberg, Germany, 2009.

9 Anhang

9.1 Projektmanagement

9.1.1 Offizielle Aufgabenstellung

Kurzbeschrieb. Im optimalen (Lehrbuch-)Fall wurde bei der Entwicklung zu jeder Klasse eine entsprechnde Unit-Testklasse geschrieben. Leider sieht die Realität anders aus: In vielen Fällen gingen die Testklassen einfach vergessen oder sie passen nicht mehr zum aktuellen Stand der Software. In solchen Systemen Änderungen vorzunehmen wird wegen den nicht vorhersagbaren Seiteneffekten sehr riskant. Mittels Bytecode Instrumentation lässt sich der Code zur Ladezeit anpassen (wird in Java z.B. bei JPA eingesetzt). So können zum Beispiel einfach die Werte der Aufrufparameter und Rückgabewerte bestimmt werden. Einerseits lassen sich die so gesammelten Werte für automatisch generierte Testklassen verwenden und andererseits lassen sich so auch zur Laufzeit abweichendes Verhalten nach einer Codeänderung detektieren.

Es soll ein Programm geschrieben werden, das automatisch Testklassen generiert und ein bestehendes Programm zur Laufzeit überwacht. Die Arbeit kann entweder in Java oder .NET implementiert werden.

Aufgabenstellung

- Erstellen Sie einen Projektplan und überlegen Sie sich, welche Daten Sie benötigen
- Bestimmen Sie die grössten Risiken des Projekts
- Recherchieren sie den aktuellen Stand der Technik
- Bestimmen Sie einen konkreten Anwendungsfall
- Entwickeln Sie eine Architektur, mittels der Sie neue Tests in ein bestehendes Framework einfach einbeziehen können
- Zeigen Sie die Praktikabilität an einem konkreten Beispiel

- Betrachten Sie die erzielten Resultate kritisch und überlegen Sie sich mögliche Erweiterungen

9.2 Weiteres

9.2.1 CD mit vollständigem Bericht und Source-Code

Der Quellcode des im Rahmen dieser Arbeit geschriebenen Programmes sowie diese Dokumentation als PDF-Datei ist diesem Dokument in Form einer CD beigelegt.

CD Inhalt (ohne Auflistung generierter Ordner/Dateien)

```
.
|-- Dokumentation_PA13_rege3.pdf
|-- git.zip
|-- build.properties
|-- build.xml
|-- Dummy\ with\ Agent.launch
|-- Manifest_agent.mf
|-- Manifest_demoAgent.mf
|-- pom.xml
|-- settings.properties
‘-- src
    |-- main
    |   ‘-- java
    |       |-- ch
    |       |   ‘-- zhaw
    |       |       ‘-- students
    |       |           ‘-- pa13_rege_3
    |       |               |-- agent
    |       |               |   ‘-- Agent.java
    |       |               |-- dummy
    |       |               |   |-- Bar.java
    |       |               |   |-- Counter.java
    |       |               |   |-- Dummy.java
```

```
|       |               |   `-- Foo.java
|       |               |-- generator
|       |               |   |-- CurrentState.java
|       |               |   |-- MockGenerator.java
|       |               |   |-- TestcaseAssembler.java
|       |               |   |-- Testcase.java
|       |               |   `-- UnitTestGenerator.java
|       |               |-- instrumentation
|       |               |   |-- Instrumentation.java
|       |               |   |-- InstrumentingClassLoader.java
|       |               |   |-- LogInstrumentor.java
|       |               |   `-- RecordCollectorInstrumentor.java
|       |               |-- record
|       |               |   |-- FieldReflectionHelper.java
|       |               |   |-- MethodCallRecord.java
|       |               |   |-- RecordCollection.java
|       |               |   |-- RecordComparator.java
|       |               |   |-- RecordDisplay.java
|       |               |   `-- RecordType.java
|       |               |-- settings
|       |               |   |-- AbstractSettings.java
|       |               |   |-- AbstractSettingsValidator.java
|       |               |   |-- ISettingsValidator.java
|       |               |   |-- Settings.java
|       |               |   `-- SettingsValidator.java
|       |               `-- util
|       |                   |-- CodeStringUtil.java
|       |                   |-- FileUtil.java
|       |                   |-- HashUtil.java
|       |                   |-- InstrumentationUtil.java
|       |                   |-- StringUtil.java
|       |                   `-- TextFileWriter.java
|       `-- com
|           `-- runjva
```

```
|               |-- demo
|               |   `-- HelloWorld.java
|               `-- instrumentation
|                   |-- JavassistHelper.java
|                   |-- LoggerAgent.java
|                   `-- TesterAgent.java
`-- test
    `-- java
        `-- ch
            `-- zhaw
                `-- students
                    `-- pa13_rege_3
                        |-- agent
                        |   `-- AgentTest.java
                        |-- AgentTests.java
                        |-- AllTests.java
                        |-- dummy
                        |   `-- BarTest.java
                        |-- DummyTests.java
                        |-- generator
                        |   |-- MockGeneratorTest.java
                        |   |-- TestcaseTest.java
                        |   `-- UnitTestGeneratorTest.java
                        |-- GeneratorTests.java
                        |-- instrumentation
                        |   `-- TestDummy.java
                        |-- InstrumentationTests.java
                        |-- learning
                        |   |-- LoadClassForInstrumentationLearning.java
                        |   |-- LogInstrumentorLearning.java
                        |   `-- TypeMagicLearning.java
                        |-- record
                        |   |-- MethodCallRecordTest.java
                        |   |-- RecordCollectionTest.java
```

```
|   ‘-- RecordDisplayTest.java
|-- RecordTests.java
|-- settings
|   ‘-- SettingsTest.java
|-- SettingsTests.java
|-- util
|   |-- CodeStringUtilTest.java
|   |-- InstrumentationUtilTest.java
|   ‘-- StringUtilTest.java
‘-- UtilTests.java
```

9.2.2 Anhang A. Anschauungsbeispiel Code