Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit Informatik

Automatische Generierung von Unit-Tests

Autor	Peter Strelecki
Hauptbetreuung	Mark Cieliebak Karl Rege
Datum	06.06.2014

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

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

Mit der Abgabe dieser Bachelorarbeit 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 Bachelorarbeit 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 Bachelorarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Zusammenfassung

Die Erstellung von Unittests während der Entwicklung gilt heute bei objektorientierter Software als Standard. Bei bestehender Software sind jedoch oftmals keine Tests vorhanden, obschon diese bei der Wartung und Pflege sehr hilfreich wären. Die nachträgliche Erstellung von Tests gestaltet sich aber schwierig, denn das Verhalten einzelner Codeteile ist meist nicht genau bekannt; der Code müsste zunächst arbeitsintensiv untersucht werden. Dies macht es interessant, Unittests automatisiert zu generieren. Die Testerzeugung kann sich dabei auf eine statische Quellcode-Analyse oder auf das dynamische Verhalten des Programms zur Laufzeit abstützen. Während der statische Ansatz gut untersucht ist, ist der Nutzen eingeschränkt, da hier nicht mit realen Daten gearbeitet wird, wie sie zur Laufzeit anfallen. Eine Analyse des laufenden Programms verspricht hier bessere Tests. Man stützt sich dabei auf die Idee, dass ein bestehendes Programm korrekt arbeitet und sein Verhalten dabei aufgezeichnet werden kann. Nach einer Änderung des Codes wird geprüft, ob das neue Programm gleiches Verhalten zeigt.

In dieser Arbeit wird untersucht, wie dieser Ansatz bei Java-Programmen realisiert werden kann. Die Instrumentierung des bestehenden Bytecodes ermöglicht eine Erfassung des Programmverhaltens und der dabei auftretenden Zustände. Dazu wird ein Execution Trace erstellt, der Methodenaufrufe und die Inhalte der involvierten Objekte festhält. Aus diesen Daten können später Unittests erzeugt werden, welche die Einhaltung dieser Zustände überprüfen. Durch Mocking können zudem die getesteten Klassen von ihrer Umgebung getrennt werden. Dieses Konzept wurde bereits in einer Projektarbeit an der ZHAW geprüft; auch ein Proof of Concept wurde dabei erstellt, der einfache Objekte mit primitiven Datentypen überwachen und testen sollte. Ziel dieser Bachelorarbeit war es, das Verfahren ausführlicher zu analysieren und die Voraussetzungen zu bestimmen, unter denen es tatsächlich funktionieren kann. Des weiteren sollten Probleme aufgezeigt werden, die einer vollständigen Umsetzung im Wege stehen können. Basierend auf bereits bestehenden Code-Teilen sollte eine Implementierung geschaffen werden, die mit komplexeren Begebenheiten wie etwa nicht-primitiven Datentypen umgehen kann. Dies ist ansatzweise gelungen.

Die primäre Erkenntnis dieser Arbeit ist, dass eine vollständig automatisierte Generierung nützlicher Tests nur unter besonderen Bedingungen möglich sein kann. Zu vielfältig sind die Einschränkungen, die bei den hier erprobten Mechanismen gelten müssen. Die Beziehungen von Objekten untereinander, die oft zu nicht persistierbaren und damit nicht reproduzierbaren States führen, sowie die Ausführung von Systemfunktionen erschweren oder verunmöglichen das Testen von Klassen im Rahmen des gesamten Programms. Die Einführung von Mock-Klassen zur Aufhebung solcher Beziehungen kann dieses Problem teilweise entschärfen, bringt aber neue Herausforderungen und weitere Einschränkungen mit sich. In jedem Falle ist zu beachten, dass nur spezifische, behutsame Änderungen an den Methoden der zu testenden Klasse erfolgreich über einen Execution Trace validiert werden können.

Dies bedeutet aber nicht, dass dieses Testkonzept hinfällig ist – die erzeugten Testfälle können eine wertvolle Hilfe sein, wenn sie manuell geprüft und zu konventionellen Unittests umgearbeitet werden. Weitere Arbeit auf diesem Gebiet wird notwendig sein, um die gefundenen Hürden besser zu verstehen. Offen bleibt insbesondere, wie die Schwierigkeiten bei der automatisierten Mock-Erzeugung und im Umgang mit Systemaufrufen überwunden werden könnten.

Summary

Today, writing unit tests during software development is common practice when creating object oriented programs. But for older, pre-existing software, such tests might often not be available, even though they would be particularly useful for maintenance tasks. Creating tests for legacy software is difficult, as exact code behaviour may not be fully known; it could result in expensive and cumbersome code studying. Therefore, automatic generation of unit tests seems appealing. Test generation can be based on static source code analysis or on dynamic program behaviour during runtime. While the static approach is well researched, its usefulness is limited, since the real-world data handled during runtime is not considered. Analysing a program in execution promises better results. This relies on the idea that the existing piece of software does work correctly and that its behaviour can be monitored and recorded. After changing the code, it can be checked whether the new program behaves in the same manner.

In this work, the described approach is considered for Java programs. Byte code instrumentation offers a way to capture program behaviour and the states that are encountered. This is achieved by creating an execution trace which contains method call information and the contents of the involved objects. From this data, unit tests can be derived that check for adherence to the expected states. Mocking allows separating the tested classes from their environment. This concept has already been looked at in a previous project thesis at ZHAW, and a proof of concept implementation has been created in an attempt to monitor and test simple Java objects containing primitive data types. The goal of this bachelor thesis was to analyse the process more thoroughly and to determine the conditions under which it can actually work. A further goal was to discover problems standing in the way of a full-featured solution. Based on existing code bits, an implementation was to be created that could cope with more complex scenarios like non-primitive data types. This has succeeded to some extent.

The primary insight from this work is that fully automated test generation can only succeed under fairly limited conditions, as there are many constraints to be applied for the attempted mechanisms. The relationships of objects will often

lead to non-persistable and therefore non-reproducible states, while the execution of system functions will hinder testing of classes within their normal surroundings. The introduction of mock classes to remove these relationships can help somewhat, but it does bring along new challenges and more limitations. In any case, only specific and slight changes within the methods of a tested class will ever successfully validate by means of an execution trace.

However, this does not entirely defeat the testing concept – the generated test cases can still be valuable if they are manually inspected and transformed into conventional unit tests. More work in this area will be required to fully understand the problems at hand, and it remains to be seen if the difficulties with automated mock creation and handling of system calls can be surpassed.

Inhalt

Kapitel 1

Einführung

1.1 Aufgabe

Auch wenn das Schreiben von Unittests bei der Software-Entwicklung in Java heutzutage als Normalfall anzusehen ist, gibt es doch viel bestehende Software, die noch ohne diese Tests ausgeliefert wurde. Auch ist denkbar, dass die Tests zwar einst geschrieben wurden, aber durch zahlreiche Änderungen am Programmcode im Laufe der Jahre nicht mehr sinnvoll zu verwenden sind. Da die nachträgliche Entwicklung solcher Tests sehr aufwändig ist, erscheint die Idee interessant, sich Unittests automatisch durch eine Software erzeugen zu lassen.

Um dies zu ermöglichen, soll ein neuartiger Ansatz erforscht werden. Er stützt sich auf die Annahme, dass der bestehende Programmcode – auch wenn er keinen Unittests unterzogen wurde – durch seine Reife zuverlässig funktioniert. Lässt man diesen Code laufen und protokolliert dabei sein Verhalten, kann später modifizierter Code gegen dieses Protokoll geprüft werden, um kritische Seiteneffekte zu finden.

Dazu muss im Protokoll möglichst jede Zustandsänderung erfasst werden, die eine zu testende Java-Klasse an ihren Objekten und an ihrer Umgebung durchführt. Um dies zu erreichen, kann Bytecode-Instrumentierung verwendet werden. Sie erlaubt es, Java-Klassen beim Laden in die Anwendung abzufangen und mit zusätzlicher Funktionalität auszurüsten.

In einer Projektarbeit an der ZHAW [PA13] wurden die Grundzüge dieses Testverfahrens bereits untersucht. Dabei wurde auch ein Proof of Concept entwickelt, der einfach aufgebaute Klassen mit ausschliesslich primitiven Datentypen testen sollte. Die Aufgabe zu dieser Arbeit war, das Verfahren genauer zu analysieren, grundlegende Überlegungen zu möglichen Herausforderungen und Problemen anzustellen und seine Funktionsfähigkeit in komplexeren Szenarien zu erörtern.

Dazu sollte die Umsetzung erweitert werden, etwa um Unterstützung für Objekttypen.

1.2 Ergebnisse

Eine Prüfung unterschiedlicher Szenarien mit mehr und weniger komplexen Klassen im Hinblick auf die Anwendbarkeit der Testmethodik hat ergeben, dass dem Verfahren in vielerlei Hinsicht Grenzen gesetzt sind. So kann die Erzeugung funktionierender Testfälle schon fehlschlagen, wenn die an der Testklasse vorgenommenen Modifikationen sich damit nicht vertragen. Daher sind nur kleinere Eingriffe in den Quellcode vertretbar. Weitere Einschränkungen gelten bei der Erfassung von Objektzuständen, denn diese können nicht-persistierbar ausfallen. Ein anschliessendes Aufsetzen beliebiger Zustände für den Testbetrieb wird dadurch erschwert, wenn dabei die echten Klassen ausgeführt werden.

Eine automatische Erzeugung von Mock-Klassen, die anstelle der Originale mit der Testklasse interagieren, kann hier abhelfen. Die Mocks haben keinen eigenen Zustand und stützen sich stattdessen auf den Zustandsübergang, der auf dem Testobjekt erwartet wird. Ein geeignetes Funktionsprinzip dafür konnte formuliert werden. Doch leider lösen die Mocks nicht alle Probleme. Je komplexer das Beziehungsgeflecht einer Testklasse, desto grösser die Gefahr, dass eine Klasse nicht wie erforderlich instrumentiert und gemockt werden kann.

Letztlich ist zu erwarten, dass in vielen Fällen manuelle Eingriffe in die so generierten Tests und Mocks notwendig sein werden, sei es um den Wirkungsbereich der Zustandsprüfungen adäquat zu beschränken, Rückgabewerte anzupassen oder bessere Mocks einzufügen. Im Sinne einer halbautomatischen Testgenerierung, deren Ausgabe überprüft und angepasst werden muss, kann eine solche Lösung für den Entwickler durchaus nützlich sein. Die State-basierten Tests dürften in Fällen, in denen das Verhalten einer Klasse nahezu unbekannt ist, eine gute Vorlage für das Ausformulieren von Unittests abgeben.

Einige Probleme technischer Art, die in [PA13] gefunden wurden, konnten geprüft und gelöst werden. Die Instrumentierung des statischen Kontext ist möglich, wenn die notwendigerweise eingefügte Feldvariable statisch bleibt. Der Zugriff auf Felder des Testobjekts beim Test-Setup kann über das Reflection-API statt über Getter- und Setter-Methoden erfolgen.

1.3 Zielpublikum

Für das Verständnis dieser Arbeit werden lediglich solide Java-Kenntnisse benötigt sowie Vorwissen über die Funktionsweise von Unittests. Vorkenntnisse zur Bytecode-Instrumentierung oder zu Javassist sind nicht notwendig; eine kurze Einführung in die funktionsweise wird in Kapitel 2 gegeben.

1.4 Bekannte Ansätze

Ein aus dem Bereich der dynamischen Code-Analyse bekannter Ansatz ist die Verfolgung gültiger Ausführungspfade und auftretender Datenflüsse eines Programms. Ein solches Verfahren wird in [Dharam] vorgestellt. Hier geht es primär darum, alle Verhaltensweisen, die ein Programm erwartet oder unerwartet hervorbringen könnte, zu erkennen.

Das Palus-Projekt [Palus] ist mit dieser Arbeit sehr nah verwandt. Palus arbeitet sowohl mit dynamischer als auch mit statischer Code-Analyse; der Ansatz wird in [Zhang] erklärt. Die statische Analyse dient der Ermittlung von Methoden-Abhängigkeiten, um festzustellen, wie eng einzelne Methoden mit ihrer Umgebung gekoppelt sind. Die dynamische Analyse erfolgt wie in dieser Arbeit mit Bytecode-Instrumentierung und dient der Ermittlung eines Ablaufmodells für die beobachteten Methoden. Aus den gewonnenen Daten werden Random Tests generiert, die eine möglichst gute Abdeckung aller Ausführungspfade gewährleisten sollen, was das erklärte Hauptziel des Projekts ist.

Leider liegt zu Palus nur wenig Dokumentation vor; auf der zugehörigen Webseite [Palus] sind nur ein paar Angaben zur Ausführung wie etwa benötigte Kommandozeilenparameter zu finden. Eine Untersuchung der wesentlichen Kernklassen, die für Instrumentierungsaufgaben verwendet werden, zeigt, dass Palus die Zustände von Objekten nur in abstrahierter Form erfasst. So werden etwa bei ganzzahligen Feldvariablen nicht die genauen Werte, sondern lediglich der vorgefundene Bereich – positiv, Null oder negativ – gesichert. Bei Arrays und Collections wiederum wird bloss zwischen leer und nichtleer unterschieden.

Ein interessantes Produkt ist der Chronon Time Travelling Debugger [Chronon]. Die Software ist laut Hersteller in der Lage, den Zustand einer kompletten Anwendung fortlaufend aufzuzeichnen und nach Belieben wiederherzustellen. Leider ist das Programm nicht quelloffen, so dass die benutzten Techniken nicht einsehbar sind.

Kapitel 2

Grundlagen der Instrumentierung

In diesem Kapitel soll kurz das Funktionsprinzip der Bytecode-Instrumentierung mit Javassist [JVA] vorgestellt werden.

Der Startpunkt aller Instrumentierungsmassnahmen ist die Agent-Klasse. Hierbei handelt es sich um eine Klasse, die von der Java VM auf Anweisung in den Startvorgang eingebunden wird und über ein passendes Interface die Möglichkeit erhält, jede danach von der VM geladene class-Datei einzusehen und zu verändern.

Dies könnte grundsätzlich auch manuell erfolgen, indem direkt der rohe Bytecode der kompilierten Klasse als Byte-Strom manipuliert wird. Dies ist jedoch nicht erforderlich, da Bibliotheken wie Javassist hierbei hervorragende Dienste leisten. Sie bilden die rohe Klassendatei auf eine Java-Klasse CtClass ab, die ähnlich dem Reflection-API bedient werden kann.

Sie stellt Methoden zur Verfügung, mit denen die Felder und Methoden der geladenen Klasse aufgezählt und untersucht werden können. Die gefundenen Methoden werden wiederum auf eine Klasse CtMethod abgebildet.

An dieser Stelle kann bequem die Instrumentierung erfolgen. CtMethod bietet die Methoden insertBefore() und insertAfter() an, die Java-Code als String entgegennehmen, diesen umgehend zu Bytecode kompilieren und am Anfang beziehungsweise am Ende der instrumentierten Methode einfügen.

Ebenso einfach können neue Felder in die untersuchte Klasse eingebettet werden. So kann eine Instrumentierungsklasse in eine Reihe geladener Klassendateien eingefügt werden, während die Java-Anwendung hochfährt.

Allerdings lassen sich nicht alle Klassen auf diese Weise verändern. Bei systemeigenen Klassen ist dies häufig unmöglich, wie bereits in [PA13] dargelegt wurde. Das Verändern systemnaher Klassen könnte die korrekte Funktion der ganzen Anwendung gefährden und wird daher nicht zugelassen.

Kapitel 3

Analyse der Anwendungsfälle

In diesem Kapitel werden die Eigenschaften möglicher Testklassen untersucht, um zu erörtern, unter welchen Voraussetzungen das Testkonzept korrekt funktionieren kann und welche Besonderheiten bei verschiedenen Szenarien zu beachten sind, wenn entsprechende Tests automatisch generiert werden sollen. Dabei wird vorerst davon ausgegangen, dass allfällige Beziehungen der Testklasse zu anderen Klassen bei der Testausführung bestehen bleiben. Benachbarte Klassen werden hier also ausgeführt, anstatt sie durch Mocks zu ersetzen.

3.1 Zustandslose Objekte mit autarken Methoden

In einem ersten einfachen Szenario soll die Grundidee des Testverfahrens betrachtet werden. Die Testklasse habe hier noch keine Feldvariablen, sondern allein Methoden. Die Methoden einer solchen Klasse werden typischerweise als *static* ausgewiesen sein.

Weiterhin nehmen diese Methoden nur Argumente an, die primitiven Typs sind und geben auch nur solche als Rückgabewerte. Innerhalb der Methoden werden zudem keine anderen Methoden aufgerufen; jede Methode arbeitet also rein autark.

Aus diesen – offensichtlich noch sehr praxisfremden – Einschränkungen folgt leicht, dass der Rückgabewert der Methode nur von den erhaltenen Argumenten abhängt. Eine Aufzeichnung dieser Argumente und der Rückgabe beschreibt das Verhalten der Methode bereits komplett. Erfasst man diese Werte im Execution Trace, ist eine Generierung und Ausführung von Tests auf Grundlage dieser Daten im Prinzip möglich.

Eine einfache Überlegung zeigt jedoch, dass dieser Ansatz beim Testen einer veränderten Methodenfassung durchaus scheitern kann. Die Methode kann beispielsweise im Rahmen eines Bugfix bewusst so geändert werden, dass sie

manchmal oder immer andere Rückgabewerte liefert. Dann funktioniert das Testkonzept zumindest nicht mehr vollständig.

Abhilfe könnte hier eine automatische Bildung von Äquivalenzklassen schaffen. Da aber ein Testgenerator die Semantik der beobachteten Werte nicht kennt, kann er hier nur basierend auf dem Execution Trace annahmen treffen, die richtig oder falsch sein können. So könnte man etwa bei Ganzzahlen auf kleiner, gleich, oder grösser Null verallgemeinern.

Schon hier zeigt sich also, dass etwaige Automatismen klare Grenzen haben werden. Im weiteren Verlauf dieses Kapitels sollen möglichen Testklassen zusätzliche Eigenschaften verliehen und die Folgerungen daraus vorgestellt werden.

3.2 Einfache Objekte mit State

Die oben angeführten Beschränkungen für den Code einer Testklasse sollen zunächst dahingehend gelockert werden, dass Feldvariablen deklariert sein dürfen – jedoch vorerst nur mit primitivem Datentyp. Objekte dieser Klasse haben damit einen Zustand, der innerhalb der meisten Methoden irgendwie verwendet oder verändert werden dürfte, was das Ergebnis bei Ausführung solcher Methoden bestimmen wird.

Das offensichtlichste Beispiel für diesen Vorgang stellen Zugriffsfunktionen dar. Der Rückgabewert von Getter-Methoden hängt nur von den zugehörigen Feldvariablen ab; demgegenüber haben Setter-Methoden keine Rückgabewerte, verändern aber unmittelbar den Zustand des Objekts, indem sie dessen Feldvariablen mutieren. Um die korrekte Funktion einer zu testenden Methode zu prüfen, muss somit der Zustand des Objekts vor und nach Ausführung der Methode erfasst werden. Die angeführten Zugriffsfunktionen sind freilich trivialer Art und werden solche Betrachtungen nicht rechtfertigen; bei komplexeren Methoden hingegen können die Zustandsänderungen schon umfangreicher ausfallen.

Der Zustand eines Testobjekts vor der Ausführung einer zu testenden Methode soll fortan als Pre-State des Objekts bezeichnet werden, jener nach der Ausführung als Post-State. Durch die Instrumentierung der Testklasse wird der Pre-State zu Beginn und der Post-State beim Verlassen jeder Methode im Execution Trace erfasst. Die Methoden können nun als Zustandsübergänge betrachtet werden, die zusätzlich über ihre Argumente parametrisiert werden. Ein bestimmter Methodenaufruf mit bestimmten Parametern in einem spezifischen Pre-State wird im

Trace zu einem eindeutigen Post-State führen, und dieser Übergang lässt sich hinterher zusammen mit dem Rückgabewert der Methode als Testfall zur Validierung neuen Codes benutzen. Um alle erkannten Übergänge nacheinander testen zu können, muss vor jedem Test der passende Pre-State korrekt aufgesetzt werden; es müssen somit alle Feldvariablen die entsprechenden Werte erhalten.

Dabei gilt die gleiche Einschränkung wie im vorigen Abschnitt. Gezielte Änderungen am Verhalten einer Methode untergraben leicht das Absolvieren solcher Zustands-Tests. Neu kommt hinzu, dass der Zustand eines Objekts nur für feste Konfigurationen seiner Felder vergleichbar ist. Schon die Einführung einer einzelnen, neuen Feldvariablen macht bestehende Traces hinfällig, ebenso das blosse Umbenennen. Und selbst wenn sich äusserlich an der Felddeklaration nichts ändert, kann eine Umstellung der Verwendungsweise eines Feldes stattfinden, die sich faktisch wie ein neues Feld auswirkt.

3.3 Interne Methodenaufrufe

Als nächste Änderung der Vorgaben sollen Methodenaufrufe erlaubt sein, vorerst jedoch nur innerhalb der betrachteten Klasse. Hat eine Klasse zwei Methoden foo() und bar(), kann die erste die zweite ausführen. Für den Execution Trace bedeutet dies, dass ein Aufruf-Stack entsteht, der bei der späteren Auswertung bedacht werden muss. Auf den Eintritt in foo() folgt der Eintritt in bar(), bevor beide in umgekehrter Folge wieder verlassen werden. Ein State-Test für foo() ist nur dann sinnvoll, wenn der gesamte Ablauf bis zum Austritt aus foo() als einzelner Zustandsübergang angesehen wird.

In diesem Zusammenhang kann auch die Frage erörtert werden, ob lokale Variablen in foo() für das Konzept von Bedeutung sind. Offenbar sind sie nicht relevant, da sie nur im Scope der Methode foo() existieren und nicht Bestandteil des Pre-State für foo()-Testfälle sind. Sie werden – in diesem Szenario – aus ebendiesem Pre-State abgeleitet. Diese Folgerung wurde auch schon in [PA13] vollzogen. Werden lokale Werte an bar() weitergereicht, muss dies entweder als Argument beim Aufruf oder durch vorherige Änderung einer Feldvariablen erfolgen; beide Fälle werden auch im Execution Trace erfasst, sobald bar() ausgeführt wird.

Dass interne Aufrufe das Testkonzept nicht behindern, lässt sich auch daran erkennen, dass sie gedanklich einfach aus dem Ablauf entfernt werden können –

der Code von bar() kann prinzipiell aufgelöst und inline in foo() untergebracht werden.

3.4 Arrays

Als erste Abkehr von rein primitiven Typen sollen nun Arrays betrachtet werden. Diese sind offenkundig in ihrer Gesamtheit ein Bestandteil des Objektzustands. Sie lassen sich iterativ auflösen, so dass jeder einzelne Wert im Array beim Aufbau des Execution Trace erfasst wird.

Damit lässt sich spätestens hier erkennen, dass beim Anfertigen des Trace Überlegungen im Hinblick auf den Speicherverbrauch notwendig sein werden. Es sollte offensichtlich vermieden werden, grosse Arrays, deren Inhalte sich womöglich über lange Zeit nicht verändern, bei jedem Methodenaufruf im zugehörigen Objekt vollständig in den Trace zu schreiben.

3.5 Externe Beziehungen

Nun soll die von Beginn des Kapitels an geltende Einschränkung abgelegt werden, dass nur primitive Datentypen erlaubt sind. In echten Anwendungen werden Objekte offensichtlich Referenzen auf andere Objekte halten. Da diese wiederum Referenzen auf weitere Objekte haben können, entsteht ein Objekt-Graph, der in vollem Umfang als Teil des Zustands angesehen werden muss.

3.5.1 Beziehungen zu Objekten ohne Systemanbindung

Zunächst soll für alle direkt und indirekt in diesem Graph verfügbaren Objekte, ähnlich der Forderung nach autarken Methoden im ersten Fallbeispiel, die starke Einschränkung gelten, dass keine Systemanbindung vorhanden sein darf. Konkret bedeutet dies, das jegliche direkte oder indirekte Nutzung der Java-Klassenbibliothek – oder einer sonstigen Bibliothek – für diese Betrachtung vorerst ausgeklammert bleibt. Das gleiche gilt an dieser Stelle für die Nutzung statischer Variablen in eigenen Klassen. Diese beiden Aspekte werden in den nachfolgenden Abschnitten erörtert.

Die Methoden, die in den angenommenen Testklassen dieses Szenarios definiert sind, können sich aber über den Objekt-Graph nach Belieben gegenseitig aufrufen. Dabei können nun auch Objektreferenzen als Argumente und Rückgabewerte zum Einsatz kommen.

Wird dieser Graph nun vollständig durchlaufen, um die Zustände aller Einzelobjekte im Execution Trace aufzuzeichnen, lässt sich ein konsistenter Gesamtzustand ermitteln, sowohl als Pre-State für eine bestimmte Methode wie auch als Post-State. Dies funktioniert selbst dann, wenn die Bezugsklassen der Testklasse selbst nicht instrumentiert werden. Ihre Methoden erscheinen dann zwar nicht im Aufruf-Stack, ihre Felder werden aber über die verfügbaren Referenzen zum Bestandteil des Pre- und Post-State aller Methoden der Testklasse. Auf ähnliche Weise können auch Argumente und Rückgabewerte durch rekursives Auflösen aller Objektbeziehungen korrekt festgehalten werden, wobei aber in diesem Fall die aufgerufene Methode instrumentiert sein muss.

Auch wenn das Testkonzept hier grundsätzlich durchführbar scheint, sind dennoch einige Schwierigkeiten zu beachten. Die erste liegt darin, dass der beschriebene Graph Zyklen enthalten kann. Ein Objekt foo kann also auf ein Objekt bar verweisen, welches wiederum auf foo zurückverweist. Bei der Erfassung aller Objekte für einen State muss daher für jedes von ihnen geprüft werden, ob es bereits erfasst wurde. Dabei muss jedes Objekt als erfasst gelten, bevor seine Referenzen aufgelöst werden, da die rekursive Suche sonst keine Zyklen vermeiden kann.

Ein anderes Problem wurde weiter oben beim Thema Arrays bereits angesprochen. Bei grossen Objekt-Graphen dürften der Speicherverbrauch und die Laufzeit des Instrumentierungscodes zu einem ernsthaften Problem werden. Da die Erfassung eines States stets die komplette Prüfung des ganzen Graphen umfassen muss, ist hier wenig Potential für Performance-Optimierungen. Um den Speicherbedarf zu begrenzen, sollten Objekte für jeden auftretenden State höchstens einmal gespeichert werden.

3.5.2 Systemaufrufe ohne Zustandsänderungen

Nun sollen erstmals Aufrufe der Java-Klassenbibliothek betrachtet werden, allerdings nur solche, deren Benutzung keine Zustandsänderungen an der laufenden Anwendung vornimmt.

Ein gutes Beispiel dafür ist eine Methode wie Math.sin(). Solche Methoden liefern nur Rückgabewerte, die allein von den Aufrufparametern abhängen.

Analog dazu können auch Systemaufrufe betrachtet werden, die den Zustand des Systems zwar ändern, aber den Zustand der Testklassen-Instanzen und ihrer Um-

gebung nicht beeinflussen. Dazu gehören etwa Ausgabeoperationen über System.out.println() oder über einen Logger. Diese Aktionen laufen in Bezug auf das Testverfahren neutral ab, da mit ihnen keine Zustandsinformationen von aussen eingebracht werden.

3.5.3 Zustand aus dem statischen Kontext

Ein weiteres Szenario findet sich bei nunmehr beliebigen Methoden, die im statischen Kontext ablaufen. Diese können grundsätzlich genau wie Objekte einen Zustand halten; dieser Zustand wird hier aber nicht über Objektreferenzen in Feldvariablen indirekt als Pre-State erfasst. Als triviales Beispiel kann hier eine Zählerklasse betrachtet werden, der mit einer statischen Variable arbeitet:

```
public class NumberGenerator {
      private static int num = 0;
      public static int getSequenceNumber() {
            num++;
            return num;
      }
}
```

Eine solche Klasse kann prinzipiell auch instrumentiert werden; Aufrufe der Methode getSequenceNumber() erscheinen dann zusammen mit dem State des Zählers num im Execution Trace. Doch dieser State ist nicht automatisch Teil des Pre-State einer Testmethode, die getSequenceNumber() aufruft.

Dennoch ist die Information im entsprechenden Aufruf-Stack vorhanden. Bei der Auswertung des Trace könnte ein Testgenerator feststellen, dass eine statische Methode aufgerufen wurde aus einer Klasse, die wiederum statischen State hat. Dieser statische Pre-State von getSequenceNumber() müsste dann mit dem Pre-State der Testmethode verknüpft werden, damit beim Test-Setup beide States korrekt vorbereitet werden können.

3.5.4 Beliebige Beziehungen und Aufrufe

Nach Betrachtung aller Vereinfachungen können nunmehr alle Einschränkungen ignoriert werden. Für beliebige Klassen, die unter anderem auf beliebige Funktionalität der Java-Klassenbibliothek zurückgreifen, lässt sich generell feststellen, dass Zustandsinformationen von aussen in die Testumgebung eingebracht werden. Dies ist ein Problem für das State-basierte Testverfahren, denn diese In-

formationen können prinzipbedingt nicht vollständig über einen Execution Trace aufgezeichnet werden – schon deshalb nicht, weil sich nicht alle Klassen der Klassenbibliothek instrumentieren lassen.

Hinzu kommt, dass nicht mehr alle involvierten Objekte nach Belieben in gewünschte Pre-States gebracht werden können, um darauf aufbauend diverse Testfälle durchzuspielen. Dies liegt daran, dass manche Objekte transiente Eigenschaften aufweisen. Was dies bedeutet, lässt sich an einem typischen I/O-Vorgang aufzeigen, beispielsweise dem Lesen einer Datei.

Um eine Datei von der Festplatte lesen zu können, muss die Java VM über plattformabhängige, native Implementierungen auf die Schnittstellen des Betriebssystems zugreifen. Um die Datei zu öffnen, muss ein passender Deskriptor oder ein Handle eingesetzt werden, der sowohl dem System als auch der Anwendung bekannt ist. Die Eigenschaften dieses Deskriptors werden als transient bezeichnet, da sie nur für diesen einen Lesevorgang gültig sind. Der Versuch, den Zustand dieses Deskriptors zu erfassen und zu einem späteren Zeitpunkt für einen Testlauf wiederherzustellen, wird offensichtlich scheitern, selbst wenn die gewünschte Datei dann noch immer verfügbar ist.

Der Zustand solcher Systembeziehungen ist also nicht persistierbar. Dies ist vergleichbar mit Java-Objekten, die aufgrund solcher Eigenschaften nicht serialisierbar sind. Neben offenen Dateien sind auch offene Sockets typische Problemfälle sowie alles andere, was einen Bezug zu einem äusseren Vorgang hat, der nicht mehr Teil der Anwendung selbst ist. Damit ist vor einem State-Test kein definierter Zustand mehr herstellbar.

3.5.5 Vererbung, Generics und Multithreading

Da Java-Klassen, die nicht implizit von Object vererbt sind, zusätzliche Felder aufweisen können, die von ihren Elternklassen stammen, muss dies bei der Erfassung von States berücksichtigt werden. Erbt eine Klasse von einer Systemklasse, kann dies dazu führen, dass sie indirekt nicht-persistierbare Eigenschaften annimmt.

Generics sollten auf die Anwendung des Testkonzepts keinen Einfluss haben. Der Java-Compiler entfernt die zusätzlichen Typinformationen bei der Erzeugung des Bytecodes in einem Vorgang, der als Type Erasure bezeichnet wird [Gen]. Sie sind zum Zeitpunkt der Instrumentierung somit nicht mehr vorhanden.

Kommt bei der zu testenden Anwendung Multithreading zum Einsatz, ergeben sich neue Schwierigkeiten, die mit parallelen Zugriffen auf den Execution Trace lediglich ihren Anfang nehmen. Wesentlich problematischer ist, dass die zur Laufzeit erfassten Objekt-States unter Umständen nicht mehr konsistent sind, wenn mehrere Threads auf die gleichen Objekte zugreifen und deren Zustände beeinflussen. So können Zustandsänderungen nicht mehr mit Gewissheit bestimmten Methoden der Testklasse zugeordnet werden. Multithreading wird daher im Rahmen dieser Arbeit vollständig ausgeklammert.

3.6 Fazit

Bei der Erfassung der Objektzustände und Objekt-Graphen sowie bei der Vorbereitung von Testfällen müssen, etwa aufgrund von nicht-persistierbaren Zuständen, zahlreiche Probleme erwartet werden. Der Zustand des Gesamtsystems kann nicht korrekt verfolgt oder beliebig manipuliert werden.

Das State-basierte Testkonzept hat damit bei einer Ausführung aller Klassen zur Testzeit erhebliche Mängel. Ob einige dieser Probleme mit Mocks gelöst werden können, soll im nächsten Kapitel untersucht werden.

Kapitel 4

Mocking

In diesem Kapitel soll geprüft werden, ob sich aus dem aufgezeichneten Verhalten einiger Klassen automatisch Mock-Klassen erzeugen lassen, die es erlauben, die Testklasse von ihrer Laufzeitumgebung zu isolieren.

Das Ziel dieser Isolation ist es, Abhängigkeiten zu entfernen und der Testklasse eine vereinfachte, stabile Testumgebung zu verschaffen. Dazu werden möglichst viele Umgebungsklassen durch Mock-Klassen ersetzt. Diese müssen sich so verhalten, wie es die Testklasse zur Laufzeit mit dem Execution Trace festgehalten hatte. Sind die Mock-Klassen geladen, können die aus dem Trace bekannten Testklassen-Methodenaufrufe als Testfälle ausgeführt werden.

Als Beispiel seien zwei Klassen Foo und Bar angeführt. Foo ist die Testklasse und konsumiert Methoden der Klasse Bar; diese wird nun durch ein Mock ersetzt. Der kombinierte State von Foo und Bar, wie er im Kapitel 3 noch betrachtet wurde, ist hier aber nicht mehr relevant, da Bar als Mock keine Felder und keinen State mehr besitzt. Dies bedeutet, dass die erfassten States von Foo-Objekten ohne das referenzierte Bar-Objekt aus dem Trace gelesen werden müssen; die Datensätze im Trace müssen dies entsprechend unterstützen.

4.1 Ausgestaltung des Mock-Mechanismus

Für die Mock-Methoden sind zwei grundlegend verschiedene Architekturen denkbar; diese sollen nachfolgend kurz vorgestellt werden.

4.1.1 Statische Mock-Methoden

Die Methoden der Mock-Klasse werden vom Mock-Generator schon vor dem Test so erzeugt, dass sie mit umfangreichen switch- und if-Anweisungen zu jedem bekannten Pre-State jede bekannte Parameter-Kombination abdecken können. Wird keine passende Kombination gefunden, wird eine Exception geworfen,

die den Testfall mit einem entsprechenden Hinweis beendet. In [PA13] wurde damit begonnen, diesen Ansatz für primitive Datentypen zu implementieren. Problematisch ist daran, dass lange Traces mit sehr vielen States und Aufrufen zu extrem grossen Klassen führen, die Informationen zu allen möglichen Testfällen mitbringen.

4.1.2 Dynamische Mock-Methoden

Dynamisch arbeitende Mock-Methoden können prüfen, mit welchem Pre-State der Aufrufer beim Test-Setup aufgesetzt wurde. Dann durchsuchen sie den Execution Trace nach einem passenden Aufruf, der in diesem State gemacht wurde und die gleichen Argumente aufwies. So finden sie den zugehörigen Rückgabewert. Unbekannte Kombinationen werden auch hier mit einer Exception quittiert.

4.2 Korrekte Zuordnung der protokollierten Aufrufe

Die im vorigen Abschnitt genannte Suche nach einem passenden Aufruf nimmt ein Kernproblem dieser Konstruktion bereits vorweg – wenn es mehrere passende Aufrufe gibt, welcher ist dann der richtige? Die Unsicherheit kommt daher, dass die nun gemockten Objekte ursprünglich mit States behaftet waren, die in vielen Fällen auf Rückgabewerte Einfluss nahmen. Bei Getter-Methoden ist dies offensichtlich der Fall. Für das oben vorgeschlagene Verfahren wurde aber die Annahme getroffen, dass diese Rückgabewerte und somit der Post-State des Testobjekts nur von seinem Pre-State abhängig sind. Diese Annahme ist allerdings ungültig.

Um die Wirkung der Rückgabewerte zu honorieren, muss tatsächlich der gesamte Zustandsübergang von einem spezifischen Pre-State in den zugehörigen Post-State als Testfall betrachtet werden. Die Methoden der Mock-Klasse müssen also ihre Rückgabewerte so wählen, dass sie zum vorbereiteten Pre-State, zu den erhaltenen Argumenten und zum erwarteten Post-State des Test-Objekts passen.

Werden diese Voraussetzungen eingehalten, kann in den meisten Fällen genau ein passender Rückgabewert bestimmt werden. Dies schlägt nur dann fehl, wenn die Testklasse innerhalb einer ihrer Methoden einen gleichartigen Mock-Aufruf mehrmals hintereinander durchführt und dabei immer neue Werte zurückerhält. Ein triviales Beispiel dafür ist der Zähler, welcher bereits in Abschnitt 3.5.3 vorgestellt wurde. In diesem Fall kann nur eine Einhaltung der Aufrufsequenz Ab-

hilfe schaffen; dafür müsste die Mock-Klasse aber eine entsprechende Vorrichtung enthalten, mit der die Anzahl der erfolgten Aufrufe festgestellt werden kann.

4.3 Diskussion

Der Rückgabetyp einer Mock-Methode kann ein Objekttyp sein. Die Methode muss also ein passendes Objekt anlegen und übergeben, was kein Problem ist, wenn die jeweilige Klasse auch durch eine Mock-Klasse substituiert worden ist. Denn ein echtes Objekt der Originalklasse mit sinnvoll präpariertem Zustand kann durch eine automatisch generierte Mock-Methode nicht ohne weiteres angelegt werden. Die dazu potentiell notwendigen Beziehungen zu weiteren Anwendungsklassen werden beim Mocken bewusst verworfen. Möglich wäre dies höchstens dann, wenn die im vorigen Kapitel erläuterte Einschränkung gälte, dass das betreffende Objekt und alle seine Abhängigkeiten einwandfrei persistierbar wären. Allerdings liefe dies der Grundidee entgegen, die Testklasse möglichst gut durch Mocks zu isolieren.

Der Typ eines Mock-Methodenarguments kann ebenfalls ein Objekttyp sein. Allein daraus ergibt sich noch nicht zwingend die Bedingung, dass dieser Typ wiederum selbst als Mock vorliegen muss. Grundsätzlich wäre es möglich, hier echte Laufzeit-Objekte zu verwenden. Warum dies keine gute Idee ist, soll nachfolgend weiter erörtert werden.

Generell lässt sich feststellen, dass das Ersetzen vereinzelter Nachbarsklassen der Testklasse durch Mocks zu Problemen führen kann, wenn gleichzeitig andere Beziehungen zu echten Laufzeitklassen erhalten bleiben. Diese Probleme liegen darin begründet, dass zwischen den gemockten und den nicht gemockten Klassen Abhängigkeiten bestehen könnten, die unweigerlich zu Fehlfunktionen der nicht gemockten Klassen führen würden. Dies liegt daran, dass sich die Schnittstelle einer Mock-Klasse nur darauf ausrichtet, die Testklasse als Konsumenten zu haben und nur deren Aufrufe und Zustände zu berücksichtigen. Dazu kommt, dass auch umgekehrt von einer Mock-Klasse keine Aufgaben wahrgenommen werden, die der echten Klasse normalerweise zufallen würden. Inkonsistenzen und Fehler in ungemockt ablaufenden Klassen der Anwendung wären die Folge.

All diese Faktoren führen zur Schlussfolgerung, dass bei Umsetzung dieses Testkonzepts grundsätzlich alle Abhängigkeiten der Testklasse gemockt werden sollten.

Rückblickend auf die Beispielklassen Foo und Bar lässt sich zudem sagen, dass die Aufrufe von Bar-Methoden nur im Execution Trace erscheinen, wenn die Klasse Bar instrumentiert wurde. Alle zu mockenden Klassen müssen also zuvor auch instrumentiert worden sein.

Nun lassen sich aber manche Klassen der Java-Klassenbibliothek nicht instrumentieren und demzufolge mit diesem Konzept auch nicht durch Mocks ersetzen.

Diese beiden Anforderungen stehen einander offenbar entgegen. Je mehr Beziehungen die Testklasse zu anderen Klassen unterhält, desto grösser ist die Wahrscheinlichkeit, dass sich nicht alle davon durch automatisch aus dem Execution Trace erzeugte Mocks versehen lassen. Das untersuchte Mocking-Konzept stösst hier an eine Grenze, die mit der vorgestellten Methode zur Instrumentierung und Ablaufverfolgung vermutlich nicht überwunden werden kann.

Kapitel 5

Aspekte der Implementierung

In diesem Kapitel werden einige Detail-Überlegungen in Bezug auf den Proof of Concept-Code vorgestellt. Als Basis für die Versuche diente der Demonstrationscode von Javassist und der Code, der im Rahmen der Projektarbeit [PA13] entwickelt wurde. Die grundlegende Architektur aus Agent-Klasse und Instrumentierungsklasse ist demnach auch hier beibehalten worden.

5.1 Einfügen der Instrumentierungsklasse

In [PA13] wurde festgestellt, dass der statische Kontext mit Javassist nicht instrumentierbar sei. Dies lag offenbar daran, dass versucht wurde, eine Instanz der Instrumentierungsklasse in jede Zielklasse als Feld einzufügen. Wird stattdessen eine statische Referenz eingesetzt, funktioniert die Instrumentierung zuverlässig.

Um mögliche Kollisionen mit bereits existierenden Feldern der Zielklasse auszuschliessen, ist es sinnvoll, ein ungewöhnliches Präfix wie etwa einen doppelten Unterstrich mit dem Feldnamen zu benutzen.

Da der Instrumentierungscode beim Auslesen aller Feldvariablen zur Erfassung des Objektzustands auch dem hier eingefügten Feld begegnet, muss darauf geachtet werden, es dabei stets auszublenden.

5.2 Instrumentierung von Konstruktoren

Java kennt drei Typen von Methoden, die bei der Initialisierung von Objekten und Klassen verwendet werden. Für Objekte existieren Konstruktoren sowie anonyme Initializer, für Klassen nur anonyme Initializer mit dem Keyword static. Letztere werden nur beim Laden einer Klasse im statischen Kontext ausgeführt und sind für die Zustandserfassung einer laufenden Anwendung daher uninteressant. Anonyme Objekt-Initializer wiederum existieren nur im Quellcode einer Java-Klasse und werden vom Compiler nicht als Methoden in den Bytecode ge-

schrieben. Der enthaltene Code wird stattdessen bei den Konstruktoren vorangestellt. Somit genügt es, alle mit getDeclaredConstructors() aus CtClass erhältlichen Konstruktoren zu instrumentieren.

5.3 Exceptions

Da das Verlassen einer Methode durch werfen einer Exception prinzipiell auch eine anzutreffende Variante ist, um aus einer aufgerufenen Methode auszutreten, ist es sinnvoll, diese Art des Austritts im Execution Trace zu erfassen. Dazu kann jede instrumentierte Methode mit Hilfe von CtMethod.addCatch() um eine catch-Klausel erweitert werden, die allfällige Exceptions aus dem Methodenkörper abfängt. Diese können ausgewertet und erneut geworfen werden.

5.4 Hashberechnung

Aufgrund der Speicherproblematik, die in Kapitel 3 angesprochen wurde, ist es sinnvoll, Objekte und Arrays nur einmal pro State im Execution Trace abzulegen. Die Berechnung eines Hashwerts über den gesamten Inhalt, wie schon in [PA13] demonstriert, ist daher unumgänglich, da sonst kein einfacher Lookup von States ausgeführt werden kann.

Auf der anderen Seite führen solche Massnahmen leicht zu einer signifikanten Verlängerung der Programmlaufzeit, da laufend Hashes berechnet werden müssen. Eine Optimierung des Speicherverbrauchs steht damit einer Optimierung der Performance entgegen.

Da die getHash()-Methode, die in jedem Objekt zur Verfügung steht, in Kindklassen oft nicht richtig überschrieben wird oder falsch implementiert ist, muss sie als unzuverlässig für die Erkennung verschiedener States betrachtet werden. Eine Hashberechnung mit einem kryptografischen Algorithmus wie in [PA13] ist daher empfehlenswert.

Kapitel 6

Weiteres Vorgehen

Das in dieser Arbeit vorgeschlagene Testkonzept bietet noch zahlreiche Ansatzpunkte für weitere Untersuchungen.

Im vierten Kapitel wurden die Beschränkungen dargelegt, denen das bisherige Konzept zu unterliegen scheint und die mit den bislang eingesetzten technischen Mitteln nicht einfach aufgehoben werden können. Daher ist es naheliegend, für eine weitere Erforschung alternative Ansätze bei der Instrumentierung von Testklassen ins Auge zu fassen.

Die reine Aufzeichnung von States nur beim Eintritt in und Austritt aus Methoden ist vermutlich unzureichend; eine direkte Manipulation von Calls im Methodenkörper könnte hier neue Wege eröffnen. Ein offenkundiger Kandidat dafür ist das Java Debug Interface [JDI]. Dieses ermöglicht beispielsweise das Setzen von Haltepunkten innerhalb der Methode. Wenn es gelingen sollte, problematische Aufrufe, etwa auf Systemfunktionen der Klassenbibliothek, direkt zu beeinflussen, könnte die Testklasse bei der Ausführung besser isoliert und in die gewünschten States manövriert werden.

Ein anderer diskussionswürdiger Aspekt ist die Genauigkeit der Zustandsübergange an sich. Hier müsste untersucht werden, ob ein gewisses Mass an Abstraktion nicht von Vorteil wäre. So mag es vielleicht beim Testen nicht immer wichtig sein, wie ein bestimmtes Objekt mit seinen Feldvariablen genau beschaffen ist; eine Überprüfung des Typs und eine Kontrolle, ob die Referenz nicht *null* ist, könnte in vielen Fällen genügen. Die blosse Länge eines Arrays zu kennen, statt jedes Element darin einzeln zu validieren, ist als Vereinfachung möglicherweise hinnehmbar.

Auch interessant ist die Frage, wie die automatische Erzeugung von Testfällen und Mock-Klassen genau gestaltet werden müsste, damit diese eine möglichst nützliche Vorlage für den Entwickler abgeben könnten, um daraus manuell richti-

ge Unittests herzustellen. Dies bleibt auch dann wichtig, wenn ein umfangreicher Satz automatisch generierter und funktionierender Testfälle auf State-Basis bereits erstellt werden konnte.

Des weiteren sind einige nebenläufige Fragen offen geblieben, etwa wie ein solches Testkonzept künftig mit Multithreading in Einklang gebracht werden könnte. Dabei müsste aber zuerst untersucht werden, ob der vermutlich geringe Nutzen den hohen Aufwand überhaupt rechtfertigen kann.

Literaturverzeichnis

[Chronon] Chronon Time Travelling Debugger, Chronon Systems, URL: http://chrononsystems.com/products/chronon-time-travelling-debugger (Jun 2014).

[Dharam] R. Dharam, S. G. Shiva: Runtime Monitoring - A Post-deployment Security Testing Technique, IEEE Hong Kong, 2012, URL: http://gtcs.cs.memphis.edu/pubs/Ramya-IRWAIST-2012.pdf (Jun 2014).

[Gen] Oracle Java Documentation – Generics Type Erasure, 2014, URL: http://docs.oracle.com/javase/tutorial/java/generics/erasure.html (Jun 2014).

[JDI] Oracle Java Documentation – Java Debug Interface, 2014, URL: http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/ (Jun 2014).

[JVA] S. Chiba: Getting Started With Javassist, 2012, URL: http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial.html (Jun 2014).

[PA13] N. Wright, D. Zolliker: Neuartiges Testkonzept mittels Bytecode-Instrumentation, Zürcher Hochschule für angewandte Wissenschaften, 2013.

[Palus] Palus Project Homepage – A Hybrid Automated Unit Test Generation Tool for Java Programs, URL: https://code.google.com/p/tpalus/ (Jun 2014).

[Zhang] S. Zhang: Palus – A Hybrid Automated Unit Test Generation Tool for Java Programs, ICSE 201, 2011, URL: http://dbonline.igroupnet.com/ACM.TOOLS/Rawdata/Acm1106/fulltext/1990000/1986036/p1182-zhang.pdf (Jun 2014).

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Automatische Generierung von Unit-Tests

BA14_ciel_4

BetreuerInnen:	Mark Cieliebak, ciel Karl Rege, rege
Fachgebiete:	Software (SOW)
Studiengang:	IT
Zuordnung:	Institut für angewandte Informationstechnologie (InIT)
Gruppengrösse:	2

Kurzbeschreibung:
In dieser Arbeit erweitern Sie ein Framework, mit dem automatisch Unit-Tests für ein bestehendes Software-System generiert werden können.

Hintergrund: Im optimalen (Lehrbuch-) Fall wurde bei der Entwicklung zu jeder Klasse eine entsprechende 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 einer Methode bestimmt werden.

Diese Werte können verwenden werden, um automatisch Unit-Tests zu erzeugen. Es gibt bereits ein Framework, das solche Tests für sehr einfache Java-Programme erzeugt.

Aufgabe: Das bestehende Framework soll erweitert werden, sodass auch komplexere Programmen analysiert und getestet werden können.

Voraussetzungen:
- Gute Programmierkenntnisse in Java
- Erfahrung mit Bytecode-Instrumentation von Vorteil