Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit Informatik

Nie mehr Unit-Tests schreiben!

Autor	Thomas Moser
Hauptbetreuung	Mark Cieliebak
Nebenbetreuung	Walter Eich
Datum	10.06.2016

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

Bei der Überarbeitung von Legacy-Software ist es aufgrund fehlender Dokumentation und spärlicher Kommentierung des Codes oft schwierig zu ermitteln, ob die bearbeiteten Stellen noch das gewünschte Verhalten aufweisen. In der Regel existieren auch keine Unit-Tests, was diese Validierung zusätzlich erschwert. Es wäre deshalb wünschenswert, die Analyse dieses Verhaltens zu automatisieren.

Die Idee, die in dieser Arbeit untersucht wurde, ist es, das Programmverhalten vor dem Refactoring aufzuzeichnen. Mit diesen aufgezeichneten Daten können Tests erzeugt und auf die überarbeitete Software angewendet werden. Auf diese Weise kann ermittelt werden, ob in der Software geändertes Verhalten auftritt und, falls dies der Fall ist, wie weit sich dieses auf die Software auswirkt.

Um dieses Ziel für Java-Programme zu erreichen wurden mit Hilfe von Bytecode-Instrumentierung zur Laufzeit des Programmes alle Konstruktor- und Methodenaufrufe inklusive der Parameter und Rückgabewerte aufgezeichnet. Diese Daten wurden anschliessend verwendet, um die geänderte Software zu validieren. Dazu wurden die bekannten Objekte einzeln instanziiert. Wurden auf diesen Instanzen die bekannten Methoden ausgeführt, konnte ihr Verhalten validiert werden (Rückgabewerte, Aufrufe andere Methoden). Im ersten Schritt wurden die Objekte durch Mocking aller Abhängigkeiten isoliert geprüft. Wurden so Änderungen festgestellt, wurde schrittweise die Anzahl gleichzeitig instanziierter Objekte erhöht, um so die Ausbreitung einer Änderung feststellen zu können.

Im Rahmen dieser Arbeit konnte ein Prototyp entwickelt werden, welcher für eine einfache Beispiel-Applikation erfolgreich Verhaltensänderungen finden und deren Ausbreitung feststellen kann. Für den Praxiseinsatz ist dieser Prototyp jedoch noch nicht geeignet, da er vielen Einschränkungen unterliegt. Beispielsweise können nur primitive Datentypen oder selbst definierte Objekte als Parameter verwendet werden.

Für eine Weiterentwicklung muss die Frage geklärt werden, ob und wie mit Aufrufen von Funktionen aus der Java-Klassenbibliothek oder von Third-Party-Libraries umgegangen werden kann. Ein weiterer Punkt, der untersucht werden muss, ist, wie das Programmverhalten vor und nach dem Refactoring verglichen werden kann, wenn sich Methodensignaturen der verwendeten Klassen ändern.

Abstract

When revising legacy software, it is often difficult to determine whether the edited parts of the software still behave as expected due to a lack of documentation and the absence of comments in the code. In the majority of cases there are no unit tests, which makes the validation even more challenging. It would thus be a worthwhile goal if the analysis of the software behaviour could be automated.

The approach that was taken in this thesis was to record the behaviour of the software before refactoring. This recorded data is then used to generate tests that can be applied to the revised software. In this way it is possible to determine if the behaviour of the software has changed. If this is the case, it is also possible to identify the impact of the changes.

To accomplish this task for java applications, all constructor and method calls including their parameters and return values were recorded during runtime using bytecode instrumentation. This recorded data was then used to validate the modified software. In order to do so, the known objects were instantiated individually. By executing the recorded method calls on these instances it was possible to evaluate their behaviour (return values, calls of other methods). In a first step, all dependencies on other objects were mocked to enable isolated testing of the objects. If differences were detected the number of simultaneously instantiated objects was increased to determine which parts of the software a change impacts.

In the context of this thesis a prototype was developed that enables the identification of behaviour modifications and their impact. For regular use, however, this prototype is not yet suitable due to several constraints such as no support of non-primitive objects except the ones that are defined in the code itself.

For further development of this software, the access to the java runtime or third party libraries needs to be resolved. Another crucial point that needs investigation is to ascertain how the behaviour of a software before and after refactoring can be compared if there were changes in the method signatures of the classes used.

Vorwort

Bei der Wahl meiner Bachelorarbeit war es mir wichtig, ein Thema zu finden, für welches Forschungsarbeit notwendig ist und bei dem ich auch neue Aspekte der Programmierung kennenlerne. Mein Ziel war es, mich nicht mit einer Aufgabenstellung auseinandersetzen zu müssen, die sich in reinem Engineering erschöpft und deren Umsetzung letztendlich das Abarbeiten eines vorgegebenen Pflichtenheftes bedeutete.

Mit dieser Arbeit konnten meine Anforderungen erfüllt werden. Es war unklar, wieviel überhaupt erreicht werden kann. Mit dieser Frage musste ich mich während der ganzen Arbeitsdauer auseinandersetzen. Weiter konnte ich meine Java-Kenntnisse im Bereich der Bytecode-Instrumentierung respektive der aspektorientierten Programmierung erweitern. Das Testing von Software ist ein Bereich, der oft als notwendiges Übel angesehen wird. Trotzdem bin ich der Meinung, dass es sehr wichtig ist, Software testbar zu gestalten und diese Tests auch zu schreiben. Nur so kann gewährleistet werden, dass die Software gut wartbar ist und gegebenenfalls auch überarbeitet werden kann.

Natürlich ist das Ziel, das Testing von Legacy-Software obsolet zu machen, noch etwas hoch gegriffen. Trotzdem war es für mich eine spannende Herausforderung, in diesem Gebiet einen Beitrag zu leisten.

An dieser Stelle möchte ich mich bei meinen Betreuern Mark Cieliebak und Walter Eich für die Unterstützung und viele wertvolle und hilfreiche Inputs während der gesamten Projektdauer danken. Dank gilt auch Daniel Schutzbach und Fatih Uzdilli, die mich bei der Einarbeitung in ihr Projekt «Augest» massgeblich unterstützt haben.

Inhaltsverzeichnis

1 Einleitung

Um bestehende Software effizient warten, erweitern und «reparieren» zu können, ist es unumgänglich, Tests zu dieser Software zu schreiben. Die Absicht dieser Tests ist unter anderem, vor und nach einem Refactoring verifizieren zu können, ob der Code die gewünschte Wirkung erzielt. So kann sichergestellt werden, dass Fehler korrekt behoben und keine neuen Fehler eingeführt werden.

Bei moderner Software, welche nach Test-Driven-Development-Prinzipien entwickelt wurde, ist das Erstellen von Tests bereits elementarer Bestandteil des Entwicklungsprozesses. Dabei werden die Tests erstellt, um das korrekte Verhalten der Software wiederholt verifizieren zu können.

Bei Legacy-Software ist es jedoch häufig der Fall, dass keine Tests existieren. Lückenhafte Dokumentation erschwert es zudem, die Aufgaben einzelner Software-Komponenten auszumachen. Doch gerade bei dieser Art von Software ist es wichtig, dass der Code überarbeitet und optimiert werden kann. Sei es für die Ergänzung mit neuen oder als Optimierung der bisherigen Funktionalitäten. Damit bei diesem Vorgang versichert werden kann, dass keine unerwünschten Änderungen eingebaut werden, ist es wünschenswert, wenn der Entwicklungsvorgang mit einem Tool unterstützt werden kann, welches aufzeigt, wie die Änderungen/Erweiterungen den Programmablauf beeinflussen.

1.1 Ausgangslage

Für die automatische Generierung von Unit-Tests gibt es verschiedene Lösungen, die auf statischer Code-Analyse beruhen. Ein Beispiel für eine solche Lösung ist der AgitarOne JUnit-Generator [1]. Solche Tools können durchaus beim Refactoring von Legacy-Software eingesetzt werden. Da sie keine Kenntnisse vom Programmablauf haben, können sie die Auswirkungen von Änderungen jedoch nicht ausmachen. Damit ermittelt werden kann, wie sich ein Programm zur Laufzeit verhält und auf Änderungen reagiert, sind dynamische Daten erforderlich.

An der School of Engineering der ZHAW wurden bereits verschiedene Arbeiten zum Thema automatisches Testing mithilfe von Bytecode-Instrumentierung durchgeführt. In einer Projektarbeit [2] wurde untersucht, wie aus Laufzeitdaten automatisch Unit-Tests erstellt werden können. Dieser Ansatz wurde in einer Bachelorarbeit [3] weiterverfolgt. Bei beiden Arbeiten wurden starke Einschränkungen festgestellt, welche die Testerzeugung erschweren. Beide Arbeiten verfolgten das Ziel, aus den aufgenommenen Daten JUnit-Tests zu erstellen und dabei Abhängigkeiten zu mocken. Die JUnit-Tests werden so zwar aus dynamischen Daten erstellt, ermöglichen aber im Anschluss keine vertiefte Auswertung.

Am Institut für angewandte Informationstechnologie (InIT), ZHAW, wurde mit «Augest» ein Projekt gestartet, das sowohl die Aufzeichnung des Programmverhaltens vor und die Validierung des Programms nach einem Refactoring übernehmen soll. Es werden dabei keine Unit-Tests erstellt. Stattdessen stellt Augest ein eigenständiges Testtool dar. Bei Augest wurden bisher die Aufzeichnung des Programmverhaltens und erste Ansätze für die Validierung realisiert. Zu diesem Projekt gibt es keine Dokumentation und auch der Code ist nur spärlich kommentiert.

Meine Bachelorarbeit basiert auf der Software des Augest-Projektes. Dabei wird untersucht, wie Augest weiterentwickelt werden kann, um eine detaillierte Analyse von Änderungen des Programmverhaltens vorzunehmen. Zudem dient diese Arbeit auch als Dokumentation von Augest, damit dieses Tool in Zukunft weiterentwickelt werden kann.

Es gibt bereits Tools wie zum Beispiel Chronon [4], die Time-Travelling-Debugging ermöglichen. Dafür wird Bytecode-Instrumentierung verwendet, um ein Programm zur Laufzeit aufzuzeichnen. Anhand dieser Informationen kann der Programmablauf später vorwärts und rückwärts rekapituliert werden. Im Artikel «Back to the Future» [5] wurde die Thematik genauer auf Stärken und Schwächen untersucht. In diesem Artikel wurde bereits die Möglichkeit thematisiert, diese Aufzeichnungsdaten für die Generierung von Unit-Tests zu verwenden. Analog dazu könnten Daten, die von Chronon aufgezeichnet werden, als Grundlage für die in dieser Arbeit behandelte Programmanalyse dienen.

1.2 Zielsetzung / Anforderungen

Ziel dieser Arbeit ist es, aufbauend auf dem Augest-Projekt ein Tool zu entwickeln, welches es ermöglicht, das Verhalten einer Software vor und nach einem Refactoring zu vergleichen. Es soll ausserdem nach dem Refactoring ermitteln können, wie gross die Tragweite festgestellter Änderungen ist.

Zusätzlich soll eine einfache Beispiel-Applikation entwickelt werden, anhand welcher die Funktion des Tools verifiziert und demonstriert werden kann.

Es soll mit diesem Tool möglich sein, sehr einfache Programme zu validieren. Bisherige Lösungen basieren vorallem auf statischer Code-Analyse. Daher kann nicht auf eine Grundlage für den untersuchten Ansatz zurückgegriffen werden. Aus diesem Grund wird in dieser Arbeit ein Tool mit starken Einschränkungen entwickelt. Eckpunkte für die zu testende Software:

- Es werden keine Multithreaded-Programme analysiert
- File-, Datenbank-, Konsolen-, Webservice-Zugriffe werden nicht berücksichtigt
- Es erfolgt keine Userinteraktion
- In der zu analysierenden Software dürfen keine Exceptions auftreten
- Zugriffe auf Instanzvariablen anderer Objekte erfolgen nur mittels Getter-/Setter-Methoden
- als Parameter und Rückgabewerte werden nur primitive Typen, Strings und in der Software selbst definierte Klassen verwendet
- Parameter und Rückgabewerte dürfen nur deterministische Werte enthalten

Das zu entwickelnde Tool soll und kann keine Entscheidung darüber treffen, ob sich das Programm *korrekt* oder *falsch* verhält. Es soll lediglich die Änderungen aufzeigen, damit diese von der entwickelnden Person selbst beurteilt werden können.

2 Vorgehen

Die verschiedenen Punkte, die für die Entwicklung des geforderten Tools erarbeitet wurden, werden in diesem Kapitel beschrieben.

2.1 Softwaretypen

Grundsätzlich kann zwischen zwei Softwaretypen unterschieden werden: Software die eine oder mehrere Aufgaben abarbeitet und sich danach beendet (zum Beispiel anhand von Datenwerten eine Kalkulation durchführen) und Software, welche kontinuierlich arbeitet und nur durch ein externes Signal beendet werden kann (zum Beispiel ein Webserver, ein Textverarbeitungsprogramm oder ein Datenlogger). In beiden Fällen ist es möglich, dass das Programm interaktiv läuft (beispielsweise Eingabe von Parametern oder Text) oder ohne Userinteraktion funktioniert (Datenlogger, Kalkulation aus gegebenen Werten, etc.).

Abhängig von der Art der Software stellen sich unterschiedliche Probleme bei der Aufzeichnung/Validierung ein: Sowohl bei terminierender als auch kontinuierlich arbeitender Software können sehr grosse Datenmengen entstehen. Der Aufwand für die Validierung steigt mit zunehmender Datenmenge kontinuierlich an. Bei Software, die interaktiv arbeitet oder zum Beispiel auf Sensorwerte zugreift, ist der Programmablauf zudem nicht mehr deterministisch.

Im Rahmen dieser Arbeit wird daher nur von Software ausgegangen, die weder kontinuierlich läuft, noch Daten von externen Quellen verwendet.

2.2 Aufzeichnung

Damit das Verhalten einer geänderten Software analysiert werden kann, muss zuerst festgehalten werden, wie sich die ursprüngliche Software verhält. Ziel der Aufzeichnung ist es, alle Informationen zu speichern, die für die spätere Analyse benötigt werden. Je nach Art der Analyse können die benötigten Daten variieren.

Nachfolgend ein nicht abschliessender Auszug möglicher Aufzeichnungsdaten:

- Konstruktoraufrufe (mit/ohne Parameterliste)
- Methodenaufrufe (mit/ohne Parameterliste/Rückgabewert)
- Lese-/Schreibzugriff auf Variablen/Felder
- Exceptions
- Zugriffe auf Filesystem
- Datenbankzugriffe
- Zeitstempel/Reihenfolge von Zugriffen/Aufrufen
- Struktur von Aufrufen (zum Beispiel als Baum)
- Nutzung von Drittsoftware (Libraries/Frameworks)

Die Aufzeichnung muss in einer Form gespeichert werden, die später vom Validierungstool gelesen und verarbeitet werden kann. Es wäre beispielsweise möglich, die Daten in einer Datenbank abzulegen, oder als Baumstruktur in einer Datei zu speichern.

2.3 Validierung

Um die Daten aus einer Aufzeichnung auszuwerten, können verschiedene Vorgehensweisen verfolgt werden.

Ein Ansatz ist, das Verhalten der überarbeiteten Software erneut aufzuzeichnen und die neue Aufzeichnung mit der alten zu vergleichen. Dieser Vergleich kann abhängig vom Aufzeichnungsformat mit verschiedenen Tools erfolgen. Wurde beispielsweise ein geeignetes Textformat gewählt, können für einen manuellen Vergleich Tools wie `diff` verwendet werden, die auch für den Vergleich von Sourcecode eingesetzt werden können. Es ist jedoch ebenfalls möglich, den Vergleich von zwei Aufzeichnungen zu automatisieren. So kann der anwendenden Person ein besser lesbares Ergebnis präsentiert werden. Der Vorteil dieser Methode liegt darin, dass beide Aufzeichnungen mit dem gleichen Tool durchgeführt werden können. Für die anschliessende Auswertung werden nur die beiden Aufzeichnungen benötigt, nicht aber die getestete Software. Diese Methode bringt aber auch verschiedene Nachteile mit sich: Damit die Aufzeichnungen verglichen werden können, müssen sie unter den gleichen Voraussetzungen entstanden sein. Das heisst, die Software muss zweimal auf die gleiche Art und Weise ausgeführt werden. Das kann insbesondere bei interaktiver Software, oder bei Software, deren Aufzeichnung einen längeren Zeitraum gedauert hat, problematisch sein. Monatelange Aufzeichnungen zu wiederholen ist zudem nicht praktikabel. Aus diesen Gründen wird dieser Ansatz in meiner Arbeit nicht mehr weiterverfolgt.

Ein zweiter Ansatz ist, die überarbeitete Software dynamisch anhand der aufgezeichneten Daten zu validieren. In diesem Fall werden die einzelnen Objekte instanziiert. Mit den aufgezeichneten Aufrufen auf dem jeweiligen Objekt kann nun kontrolliert werden, ob sich das Objekt noch gleich verhält (gleicher Rückgabewert bei identischen Parametern, die selben Aufrufe von Methoden anderer Objekte, gleiche Nebeneffekte, etc.). Dieser Vorgang kann nicht manuell vorgenommen werden und muss von einem Tool umgesetzt werden. Es ist zudem möglich, einzelne Komponenten zu überprüfen und iterativ weitere Komponenten hinzuzufügen, um so die Tragweite einer Änderung zu ermitteln (dieses Vorgehen wird genauer im Kapitel 2.6.2 Iterative Analyse beschrieben). Dieser Ansatz wird in meiner Arbeit weiter vertieft.

Unter der Voraussetzung, dass eine Aufzeichnung sowohl von fehlerhafter als auch korrekt arbeitender Software stammen kann, ist es bei der Validierung nicht möglich eine Aussage über richtiges oder falsches Verhalten der analysierten Software zu treffen. Es kann ausschliesslich ermittelt werden, ob sich die Software abweichend zur Aufzeichnung verhält oder nicht. Es liegt daher bei der anwendenden Person des Tools, die Validierungsergebnisse situationsgerecht auszuwerten.

2.4 Sequenzdiagramme

In dieser Arbeit werden viele Beispiele anhand von Sequenzdiagrammen beschrieben. Damit diese richtig gelesen werden können erfolgt hier eine kleine Erläuterung.

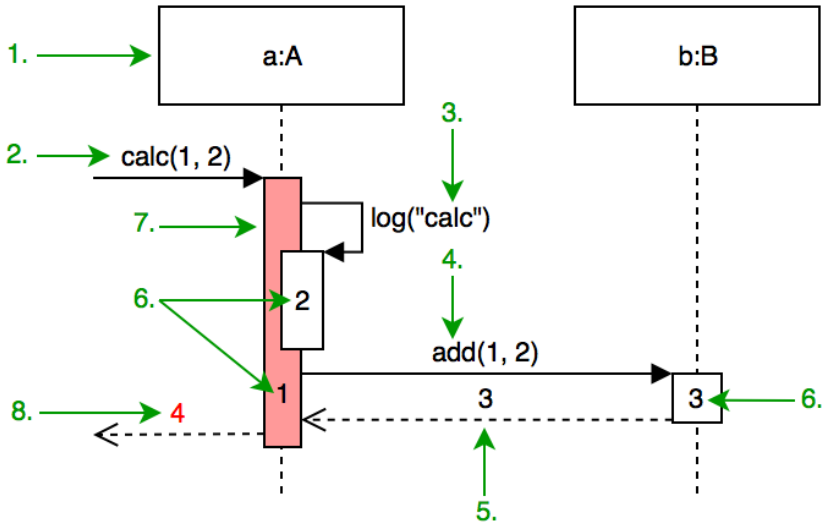


Abbildung 1 Legende Sequenzdiagramme

1. Bezeichnet sowohl den Namen der Objektinstanz (vor dem Doppelpunkt) als auch den Namen der Klasse (nach dem Doppelpunkt). Die senkrechte, gestrichelte Linie ist die «Lebenslinie» des Objekts. Darauf werden alle Konstruktor- und Methodenaufrufe (auch Sequenzen genannt) eingezeichnet, die auf diesem Objekt erfolgen.
2. Kennzeichnet den Aufruf einer Methode. Dabei werden der Name der Methode (`calc`), als auch allfällige Parameter (in diesem Fall zwei Ganzzahlen 1 und 2) angegeben.
3. Das Objekt ruft eine eigene Methode auf. Diese Methode kann einen Rückgabewert haben, dieser wird aber auf dem Diagramm nicht eingezeichnet.
4. Aufruf einer Methode eines anderen Objekts.
5. Der Rückgabewert einer Methode. Hat eine Methode keinen Rückgabewert, wird ein Pfeil ohne Text gezeichnet.
6. Kennzeichnet die Sequenznummer dieses Aufrufs.
7. Eine Sequenz, bei der abweichendes Verhalten zur Aufzeichnung detektiert wurde (genaueres im Kapitel 2.7 Umsetzung in Software).
8. Ein Rückgabewert, der von dem aufgezeichneten Rückgabewert abweicht.

2.5 Änderungen

Von einer Änderung wird gesprochen, wenn sich der aufgezeichnete Ablauf einer Software im Vergleich zur anschliessenden Analyse des geänderten Codes abweichend verhält. Um eine Änderung genauer beschreiben zu können, werden die in diesem Kapitel definierten Änderungstypen verwendet.

2.5.1 Änderungstypen

Nebeneffekte

Unter Nebeneffekten werden alle Aktionen/Reaktionen verstanden, welche die Folge eines Konstruktor-/Methodenaufrufs sind.

Eine Auswahl von möglichen Nebeneffekten:

- Aufrufe von Konstruktoren/Methoden weiterer Klassen/Objekte oder Aufruf von objekteigenen Methoden
- geänderte Reihenfolge der Aufrufe
- Userinteraktion (Ausgaben auf Konsole/GUI, Eingabe des Users, etc.)
- Filezugriff (lesen/schreiben/löschen)
- Datenbankzugriff
- Exceptions

Statusänderungen

Eine weitere Folge von Konstruktor-/Methodenaufrufen können geänderte Instanzvariablen und somit eine Änderung des Zustandes eines Objektes sein. Genau genommen sind die Statusänderungen den Nebeneffekten zuzuordnen. Statusänderungen können aber das Verhalten eines Objektes beeinflussen. So kann beispielsweise eine Methode nach einer Statusänderung einen anderen Rückgabewert haben als zuvor mit den gleichen Parametern. Aus diesem Grund werden die Statusänderungen als Spezialfall der Nebeneffekte separat definiert.

Interne Änderungen

Die interne Struktur einer Methode hat sich geändert, hat aber die selben Nebeneffekte und den gleichen Rückgabewert zur Folge.

Um diese Art von Änderungen zu analysieren muss eine Aufzeichnung/Validierung mit sehr feiner Granularität erfolgen. Auf dieser Detailstufe werden bei einem Refactoring üblicherweise viele Änderungen vorgenommen. Diese Änderungen würden das Ergebnis einer Validierung schnell sehr unübersichtlich gestalten. Da aber in dieser Arbeit primär das Verhalten von Methoden untersucht wird, wird auf diese Art von Änderungen nicht mehr weiter eingegangen.

Funktionelle Änderung

Wird eine Methode mit bekannten Parametern aufgerufen, resultieren andere Rückgabewerte.

Diese Art der Änderung kann sehr einfach festgestellt werden, indem die bekannten Methodenaufrufe aus einer Aufzeichnung auf instanziierten Objekten erneut mit den aufgezeichneten Parametern aufgerufen werden. Weicht der nun erhaltene Rückgabewert von demjenigen der Aufzeichnung ab, liegt eine funktionelle Änderung vor.

Geänderte Methodensignatur

Die Signatur einer Methode hat sich geändert, oder die Methode ist nicht mehr verfügbar.

Hat sich eine Methodensignatur geändert oder wurde die Methode komplett entfernt, kann das detektiert werden, da die Methode mit der ursprünglichen Signatur nicht mehr verfügbar ist.

Geänderte Nebeneffekte

Wird eine Methode aufgerufen, treten andere Nebeneffekte auf, als es bei der Aufzeichnung der Fall war.

Grundsätzlich können geänderte Nebeneffekte detektiert werden. Es ist jedoch massgeblich von der Aufzeichnungstiefe abhängig, bis zu welchem Grad Nebeneffekte erkannt werden können. Werden beispielsweise Aufrufe von Methoden der Java Plattform nicht aufgezeichnet, können geänderte Aufrufe von `System.out.println(...)` nicht ausgemacht werden.

2.5.2 Änderungen nach einem Refactoring

Nach einem Refactoring können sich Änderungen von Sequenzen aus mehreren der oben beschriebenen Änderungstypen zusammensetzen. Dabei ist sowohl eine einzelne Änderung möglich, als auch mehrere Änderungen gleichzeitig.

Beispiele

Im Folgenden werden einige Beispiele von Änderungen gemacht.

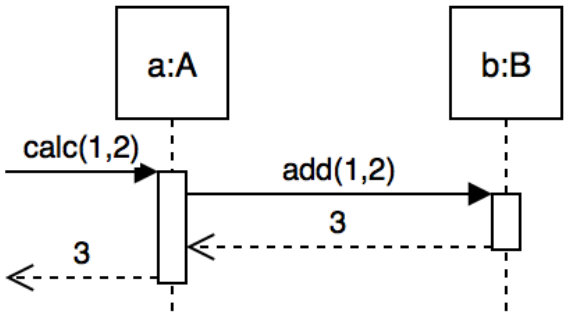


Abbildung 2 Sequenzdiagramm der unbearbeiteten Software

Abbildung 2 zeigt dabei das Sequenzdiagramm der unbearbeiteten Software, mit welcher die Sequenzdiagramme der Beispiele verglichen werden.

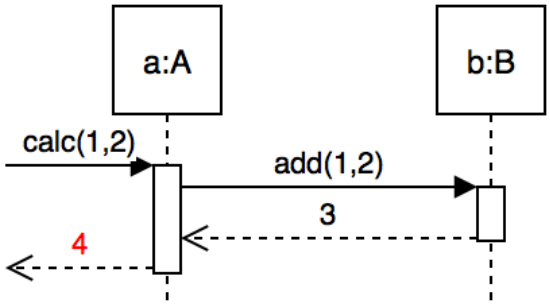


Abbildung 3 Sequenzdiagramm bei einer funktionellen Änderung

In Abbildung 3 ist das Sequenzdiagramm nach einer funktionellen Änderung zu sehen. Die Methode `calc(1, 2)` hat bei gleichbleibenden Nebeneffekten einen geänderten Rückgabewert.

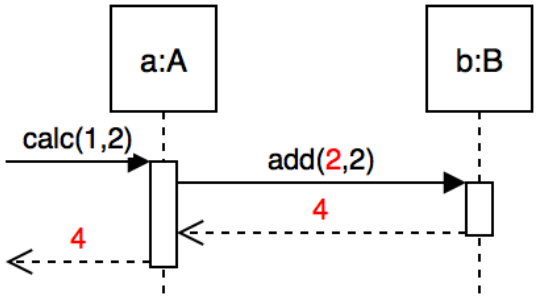


Abbildung 4 Sequenzdiagramm mit verschiedenen Änderungen

Abbildung 4 zeigt das Sequenzdiagramm der überarbeiteten Software, die sowohl eine funktionelle Änderung (Rückgabewert 4 anstatt 3) als auch geänderte Nebeneffekte (die Methode `add` wird mit anderen Parametern aufgerufen als es bei der Aufzeichnung der Fall war) aufweist.

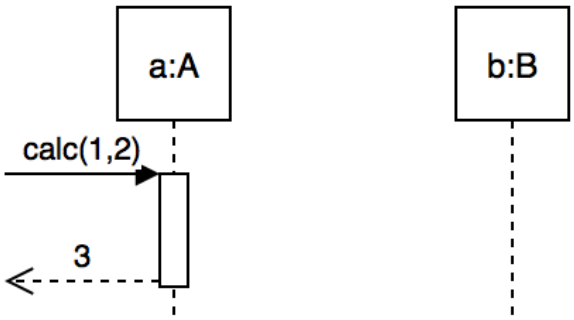


Abbildung 5 Sequenzdiagramm bei geänderten Nebeneffekten

Im Falle von Abbildung 5 handelt es sich um das Sequenzdiagramm bei geänderten Nebeneffekten, da hier der Aufruf `add(1, 2)` entfällt.

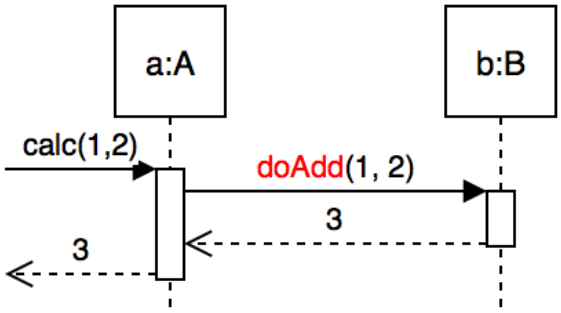


Abbildung 6 Sequenzdiagramm bei geänderter Signatur

Im Sequenzdiagramm in Abbildung 6 weist die Methode `calc` des Objektes a sowohl geänderte Nebeneffekte auf (Aufruf der neuen Methode `doAdd`), während die Methode `add` des Objektes b eine geänderte Signatur aufweist.

2.5.3 Auswirkung von Änderungen eingrenzen

Um den Einfluss einer Änderung eingrenzen zu können, ist das Ziel aus den über- und/oder untergeordneten Methodenaufrufen einen Block zu bilden, welcher sich gleich verhält, wie bei der Aufzeichnung. Innerhalb des Blockes darf das Verhalten von der Aufzeichnung abweichen. Dieses Prinzip wird hier kurz Anhand zweier Beispiele erläutert.

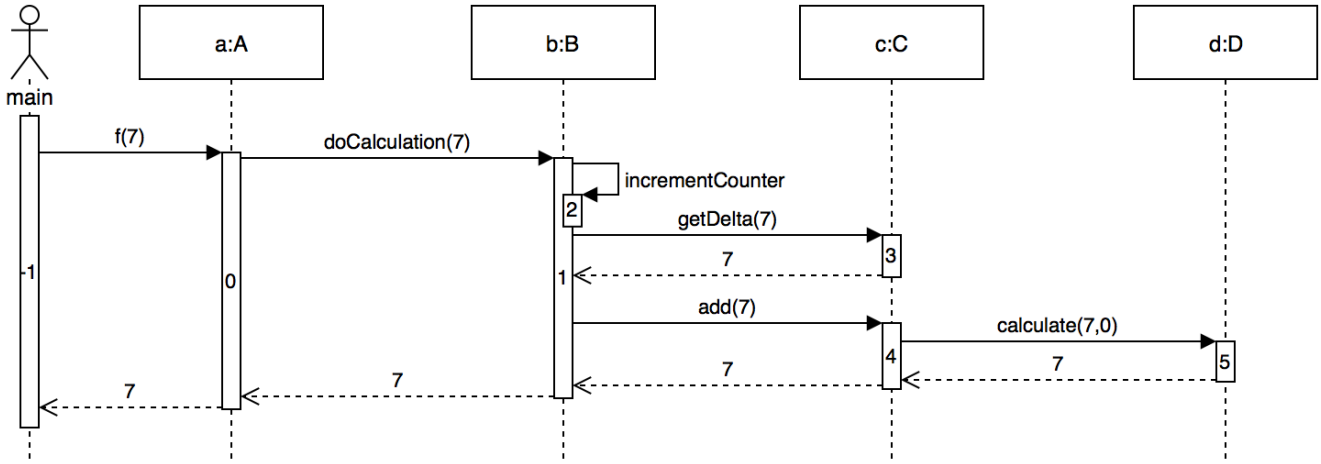


Abbildung 7 Aufgezeichnetes Sequenzdiagramm

Beispiel 1

Abbildung 8 zeigt das Sequenzdiagramm der geänderten Software. In diesem Fall wurden die Methoden mit der Sequenz #1 und #4 editiert. Bei Sequenz #4 wird 1 zu dem erhaltenen Resultat aus #5 addiert, während bei #1 wieder -1 gerechnet wird, um diese Anpassung auszugleichen.

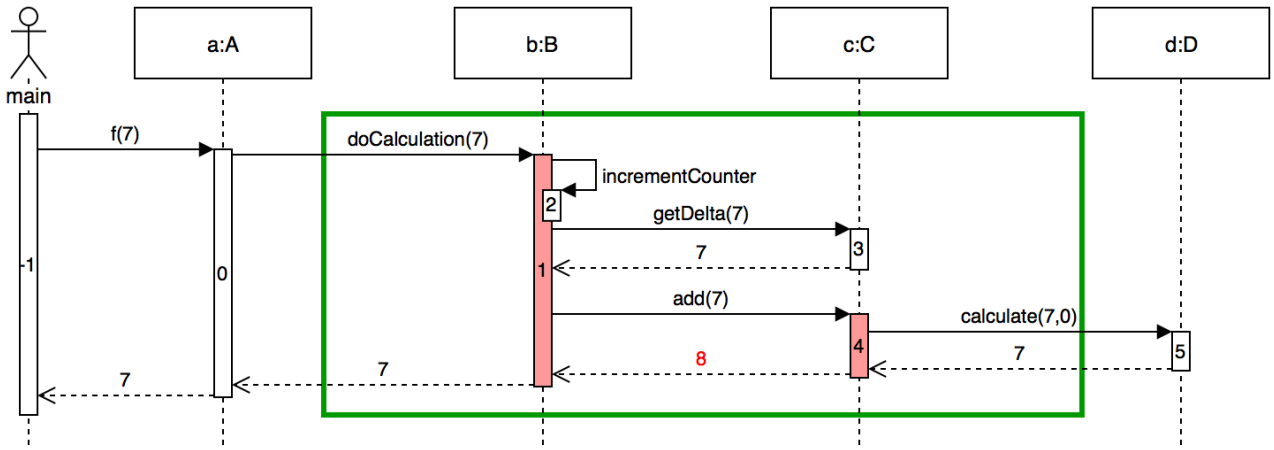


Abbildung 8 Block, der sich nach aussen gleich verhält

Somit konnte in diesem Fall ein Block ermittelt werden, der sich von aussen betrachtet gleich verhält wie in der Aufzeichnung. Das heisst, dass bei einem Aufruf der Methode `doCalculation(7)` des Blocks sowohl die erwartete Methode `calculate(7, 0)` des Objektes d aufgerufen wird, als auch der erwartete Rückgabewert 7 vom Block zurückgegeben wurde. Innerhalb des Blockes weicht das Verhalten von der Aufzeichnung ab.

Beispiel 2

In Abbildung 9 wird eine Änderung der Sequenz #5 dargestellt, die von keiner anderen Änderung ausgeglichen wird.

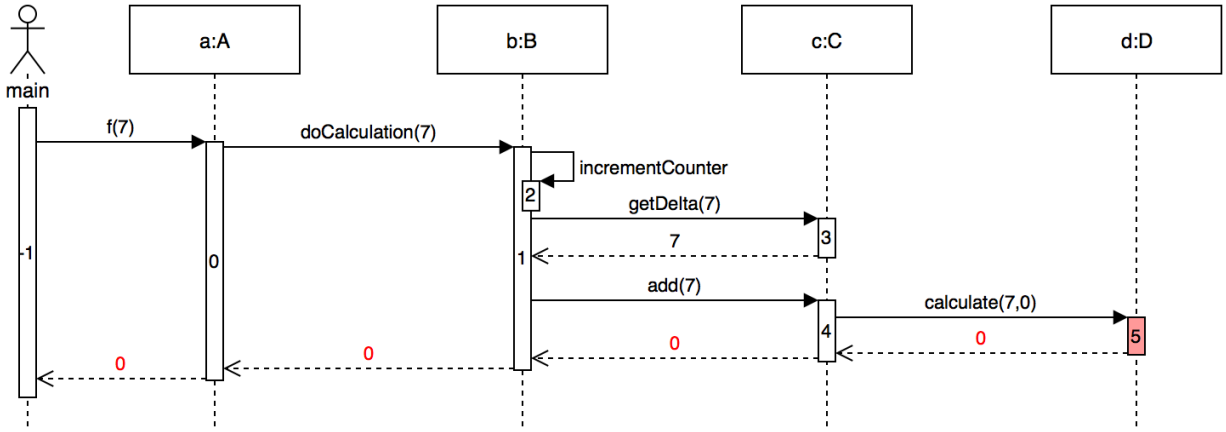


Abbildung 9 Uneingeschränkte Änderungsausbreitung

Statt der erwarteten 7 wird beim Aufruf `calculate(7, 0)` der Objektinstanz d 0 zurückgegeben. Die weiteren Sequenzen in der Aufrufkette verwenden diesen Wert ohne weitere Anpassungen, wie das bereits bei der Aufzeichnung der Fall war. Somit breitet sich diese Änderung bis zum ersten Aufruf aus, welcher in der main-Methode erfolgt. Folglich konnte in diesem Fall kein Block gebildet werden.

2.6 Grundidee für die Analyse von Programmverhalten

Es wurde ein Vorgehen entworfen, welches auf zwei Phasen beruht: der Vorbereitungsphase und der anschliessenden iterativen Analyse.

2.6.1 Vorbereitungsphase

Im ersten Durchlauf der Analyse werden alle bekannten Objekte isoliert von den anderen Objekten auf ihr Verhalten geprüft. Dazu werden die bekannten Objekte nacheinander abgearbeitet. Diese Analyse besteht aus den nachfolgenden Schritten.

1. Der Konstruktor des aktuellen Objektes wird aufgerufen. Dafür werden die selben Parameter verwendet, die auch bei der Aufzeichnung festgehalten wurden. Das so erzeugte Objekt wird anschliessend für die weiteren Schritte verwendet.
2. Auf dem erzeugten Objekt werden nun alle Methodenaufrufe durchgeführt, die während der Aufzeichnung festgehalten wurden. Hier werden die Parameter verwendet, die aufgezeichnet wurden. Nach dem Methodenaufruf wird der zurückgegebene Wert mit dem aufgezeichneten Wert verglichen. Es wird während der Ausführung der Methode überwacht, welche Methoden von anderen Objekten mit welchen Parametern aufgerufen werden. Stimmen sowohl der Rückgabewert als auch die beobachteten Aufrufe von Methoden/Konstruktoren anderer Objekte mit der Aufzeichnung überein, wurde für diese Methodenausführung keine Abweichung festgestellt.
3. Wurden alle aufgezeichneten Methoden eines Objekts ausgeführt und sind dabei keine Abweichungen aufgetreten wird dieses Objekt als «erwartungsgemäss» markiert. Das bedeuted, dass es sich während der gesamten Analyse gleich verhalten hat wie während der Aufzeichnung. Sind Abweichungen aufgetreten, so werden die betroffenen Sequenzen für die genauere Analyse festgehalten.

Während den Schritten 1. und 2. werden noch weitere Punkte berücksichtigt.

- Um sicherzustellen, dass ein Objekt unabhängig von anderen Objekten geprüft werden kann, werden alle anderen Objekte, die während der Prüfung benötigt werden, gemockt. Es ist so möglich, die Umgebung des geprüften Objektes zu simulieren und zu überwachen. Es können für bekannte Parameter (aus der Aufzeichnung) die entsprechenden Rückgabewerte zurückgegeben werden. Ausserdem ist es möglich, zu überpüfen, welche Methoden auf den Mocks aufgerufen wurden und in welcher Reihenfolge diese Aufrufe erfolgt sind. Wird auf einem Mock eine unbekannte Methode aufgerufen (unbekannter Methodenname oder nicht aufgezeichnete Parameterkombination), kann kein Rückgabewert ermittelt werden. Es wird eine Exception erzeugt und die aktuell getestete Methode wird abgebrochen.
- Wird während der Ausführung eines Konstruktors oder einer Methode ein weiterer Konstruktor aufgerufen, wird dieser abgefangen. Statt nun ein neues Objekt zu erstellen wird ein Mock generiert und zurückgegeben.
- Wird als Parameter einer Methode ein Objekt benötigt, das bisher noch nicht verwendet wurde, wird davon ebenfalls ein Mock generiert. Dieser Mock wird nun als Parameter verwedet.

Wurden nun alle bekannten Objekte nach diesem Muster geprüft, kann mit der iterativen Analyse begonnen werden.

2.6.2 Iterative Analyse

Die nun durchgeführte Analyse erfolgt für die Sequenzen, die während der Vorbereitungsphase als abweichend markiert wurden. Diese Sequenzen werden nun nacheinander abgearbeitet. Es wird dabei nach einem ähnlichen Muster gearbeitet wie bei der Vorbereitungsphase, wobei sich einige Punkte unterscheiden.

- Es werden nicht automatisch alle anderen Objekte, die für eine Methodenausführung benötigt werden gemockt. Bei jeder weiteren Iteration wird eine zusätzliche Ebene des Aufzeichnungsbaumes ausgeführt. Dies bedeutet, dass für jeden Aufruf ein weiteres Objekt instanziiert wird. Bei jeder weiteren Iteration wird das aufrufende Objekt der zuvor gescheiterten Sequenz zusätzlich instanziiert. Auf dieser zusätzlichen Instanz wird dann die Methode aufgerufen, innerhalb welcher der Methodenaufruf aus der vorherigen Iteration erfolgt.
- Vor jeder Sequenz, die geprüft wird, muss festgestellt werden, welche Objekte instanziiert werden müssen. Anschliessend muss dafür gesorgt werden, dass diese Objekte entsprechend instanziiert und von den richtigen Objekten verwendet werden. Bei der Vorbereitungsphase wurde jedes Objekt, das zusätzlich benötigt wurde gemockt. Diese zwei Techniken (abfangen von Konstruktoraufrufen, übergeben von Mocks als Parameter) müssen nun so angepasst werden, dass gegebenenfalls kein Mock sondern eine Objektinstanz erzeugt wird.
- Handelt es sich beim Aufrufer einer Sequenz um die main-Methode, ist die Ausbreitung der Änderung nach oben maximal. Die Änderung wird dementsprechend markiert und es müssen keine weiteren Iterationen mehr dafür durchgeführt werden.
- Da die Sequenzen in dieser Phase nicht zwangsläufig aufeinanderfolgende Methoden eines Objektes sind, kann der Test in beliebiger Reihenfolge durchgeführt werden. Da die Objekte einen Zustand haben, der sich durch Aufruf verschiedener Methoden verändern kann, ist es notwendig, alle instanziierten Objekte in den selben Zustand zu bringen, in welchem sie sich während der Aufzeichnung bei der Ausführung der geprüften Sequenz befunden haben. Um dies zu gewährleisten wird dafür gesorgt, dass alle Methodenaufrufe der instanziierten Objekte, welche vor der zu testenden Sequenz liegen, in der richtigen Reihenfolge aufgerufen werden.
- Wurde bei einem Refactoring eine Methode geändert, kann es sein, dass eine weitere Methode ebenfalls an diese Änderung angepasst wurde. In diesem Fall werden in der Vorbereitungsphase beide Sequenzen als abweichend markiert. Typischerweise werden während der iterativen Analyse im Laufe der Iterationen beide Sequenzen geprüft. In diesem Fall macht es keinen Sinn Tests der «übergeordneten» Sequenz durchzuführen, da diese von der Änderung einer tiefer gelegenen Sequenz abhängig ist und somit auch implizit geprüft wird.
- Verhält sich der Block, welcher durch das Instanziieren mehrere Objekte entsteht, gleich, wie es bei der Aufzeichnung der Fall war, wurde eine Ausbreitungsgrenze der Änderung gefunden und es müssen keine weiteren Tests mehr durchgeführt werden. Ist das nicht der Fall, wird der Block für eine weitere Iteration als abweichend markiert.

Dieser iterative Vorgang wird solange durchgeführt, bis keine Sequenzen mehr vorhanden sind, deren Ausbreitung nicht fertig bestimmt ist.

Beispiel 1

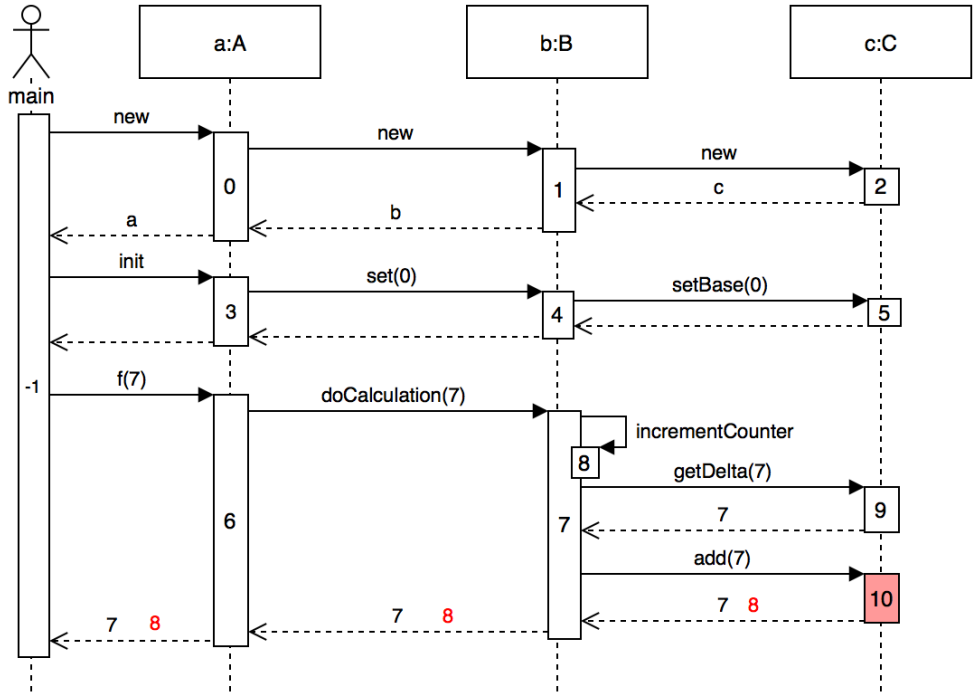


Abbildung 10 Sequenzdiagramm mit unbeschränkter Änderungsausbreitung

Annahme: In der Vorbereitungsphase wurde die Sequenz #10 als abweichend markiert. Rot sind die Rückgabewerte, wie sie tatsächlich von der geänderten Methode zurückgegeben werden. Die schwarzen Werte sind diejenigen, welche anhand der Aufzeichnung erwartet werden.

Vorgehen:

1. Für die erste Iteration muss sowohl das Objekt c als auch das Objekt b instanziiert werden. Dafür wird der Konstruktor `new B()` aufgerufen. Innerhalb dieses Konstruktors wird gemäss Sequenzdiagramm der Konstruktor `new C()` aufgerufen. In beiden Fällen wird eine Instanz erzeugt und zurückgegeben. Bevor das Verhalten der Methode `doCalculation` (Sequenz #7) geprüft werden kann, müssen die Objekte b und c in den richtigen Zustand überführt werden. Dafür müssen die Sequenzen #4 und #5 ausgeführt werden. Da #5 innerhalb von #4 aufgerufen wird, reicht es die Methode `set(0)` aufzurufen, um den richtigen Zustand beider Objekte sicherzustellen. Nun kann die Methode `doCalculation(7)` geprüft werden. Da der Rückgabewert 8 nicht der Aufzeichnung entspricht, muss eine weitere Iteration durchgeführt werden.
2. In der zweiten Iteration müssen alle Objekte instanziiert werden. Dazu reicht es den Konstruktor `new A()` aufzurufen (`new B()` und `new C()` werden dadurch implizit ausgeführt). Nun muss der Zustand der Objekte wiederhergestellt werden. Dafür muss die Methode `init()` (Sequenz #3) des Objektes a aufgerufen werden. Der folgende Aufruf der Methode `f(7)` gibt mit 8 nicht den erwarteten Wert zurück, weshalb nochmals eine Iteration angestossen wird.
3. In der dritten Iteration wird das Tool feststellen, dass keine weiteren Objekte involviert sind. Da mit der main-Methode die obere Grenze der Aufruf-Hierarchie erreicht wurde, können keine Tests durchgeführt werden. Die Ausbreitung der Änderung wird dementsprechend vermerkt und die Iteration wird beendet.

Beispiel 2

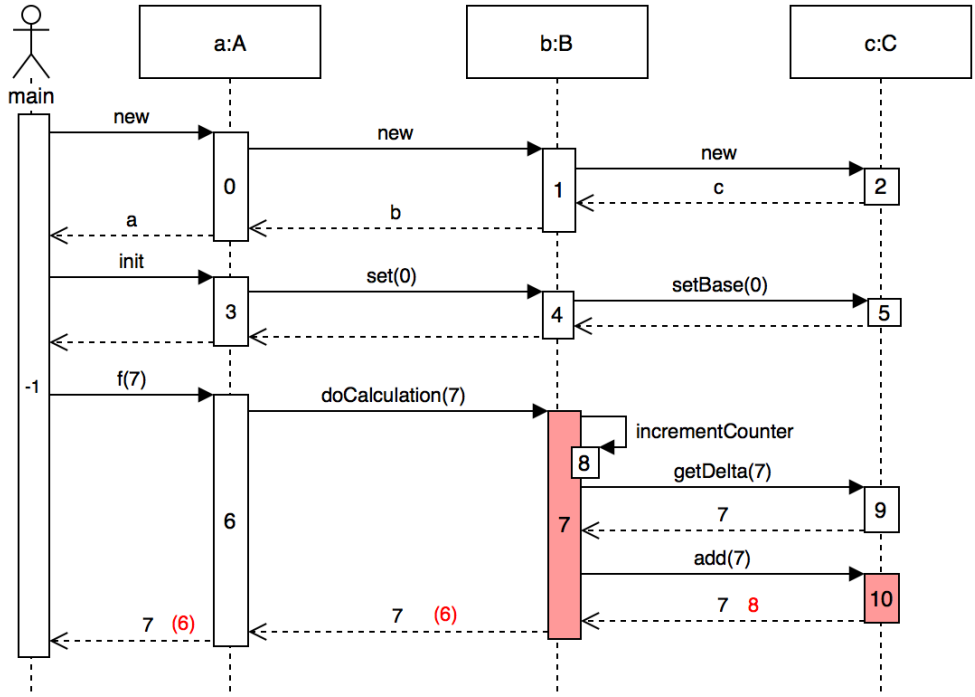


Abbildung 11 Sequenzdiagramm mit eingegrenzter Änderungsausbreitung

Annahme: Während der Vorbereitungsphase wurden die Sequenzen #7 und #10 als abweichend markiert. Rot sind die Rückgabewerte, wie sie tatsächlich von der geänderten Methode zurückgegeben werden. Die schwarzen Werte sind diejenigen, die anhand der Aufzeichnung erwartet werden.

Vorgehen:

1. Da zwei Sequenzen als abweichend markiert wurden, wird geprüft, ob eine Sequenz bereits eine geänderte, untergeordnete Sequenz hat. Ist das wie bei Sequenz #7 der Fall, wird diese Sequenz zurückgestellt. Eine zurückgestellte Sequenz wird nur dann geprüft, wenn sie von der Ausbreitung der tiefer gelegenen geänderten Sequenz nicht betroffen ist.
2. Um die Änderung der Sequenz #10 zu analysieren müssen die Objekte b und c instanziiert werden. Das Vorgehen für die Instanziierung und die Zustandswiederherstellung ist dabei identisch mit dem Vorgehen bei der ersten Iteration des vorherigen Beispiels. Der Aufruf von Methode `doCalculation(7)` gibt nun wie erwartet 7 zurück. Somit wurde eine obere Grenze der Änderungsausbreitung gefunden und die Analyse kann beendet werden.
3. Da in diesem Beispiel die zurückgestellte Sequenz #7 bereits im Verbund mit Sequenz #10 geprüft und für erwartungsgemäss befunden wurde müssen keine weiteren Tests mehr erfolgen.

Anmerkung: Würde die Analyse der Sequenz #7 nicht zurückgestellt, müsste in der ersten Iteration das der Block bestehend aus Sequenz #7 und #6 ebenfalls validiert werden. Die Methode `add(7)` würde mit dem aufgezeichneten Rückgabewert 7 gemockt. Das wiederum würde dazu führen, dass die Methode `doCalculation(7)` und somit auch die Methode `f(7)` den falschen Wert 6 (in Klammern) zurücklieferte. So würde eine falsche Änderungsausbreitung detektiert werden.

2.7 Umsetzung in Software

Um dieses Prüfkonzept als Tool umzusetzen stand als Basis das Projekt «Augest» zur Verfügung, welches am InIT von Fatih Uzdilli und Daniel Schutzbach entwickelt wurde. Dieses Tool wurde bereits erfolgreich eingesetzt, um das Verhalten einer Software aufzuzeichnen. Es existierte zudem bereits ein Ansatz um das Verhalten der modifizierten Software zu verifizieren. Dieser Ansatz war jedoch noch nicht einsatzfähig und konnte keine Eingrenzung der Änderungsauswirkung vornehmen.

Während beim Aufzeichnungstool nur geringfügige Anpassungen vorgenommen werden mussten, wurde der Validierungsteil - die sogenannte «Validation» - in weiten Teilen überarbeitet/repariert oder umstrukturiert. Ein weiterer Teil, der mit nicht allzu grossen Änderungen weiterhin eingesetzt werden konnte, ist der Mechanismus um Mocks von Objekten mithilfe von CgLib zu erstellen.

Um den Code besser zu modularisieren und so wartbarer zu gestalten, wurden in allen Teilen der Software grössere Refactorings ausgeführt. Ein grosser Teil der Refactorings umfasste das Extrahieren von Methoden oder ganzen Klassen. Der ursprüngliche Code war auf wenige Klassen mit relativ schwacher Kohäsion verteilt. Ein Ziel von mir war es, diese Kohäsion zu verbessern. Dazu wurden viele Hilfsklassen erstellt. Einige Beispiele:

- **TraceTreeHandler.java**: Diese Klasse stellt verschiedene Methoden zur Verfügung um einen Aufzeichnungsbaum abzuarbeiten. So können Beispielsweise alle Konstruktoraufrufe des Baumes extrahiert werden oder alle Sequenzen zu einer bestimmten Objekt-ID.
- **Executions.java:** Sammlung von Hilfsmethoden für den Umgang mit `Executions` (`ConstructorExecution`, `MethodExecution`: aufgezeichnete Konstruktor- respektive Methodenaufrufe). Damit ist es möglich, die Parameter einer `Execution` oder den Rückgabewert einer `MethodExecution` abzufragen (dabei werden die tatsächlichen Objekte/Werte zurückgegeben und nicht die für die Aufzeichnung verwendeten Wrapper).
- **Verifications.java:** Stellt Vergleichsmethoden zur Verfügung, um Objekte oder Arrays von Objekten miteinander zu vergleichen. Diese Methoden wurden extrahiert, damit primitive- und Objekt-Typen miteinander verglichen werden können, ohne dabei auf `null`-Werte achten zu müssen. Zudem können Parameter-Arrays von `MethodExecutions` verglichen werden.
- **Mocking.java:** Das gesamte Erstellen von Mocks erfolgt in dieser Klasse. Es gibt auch die Möglichkeit, Mocks auf erwartete `MethodExecutions` vorzubereiten und im nachhinein zu überprüfen, welche Methoden tatsächlich auf den Mocks aufgerufen wurden oder ob bei den Aufrufen Fehler aufgetreten sind.
- **ObjectProvider.java:** Die Verwaltung von Objekten, die für die Validierung benötigt werden erfolgt in dieser Klasse. Dabei kann festgelegt werden, welche Objekte instanziiert und welche gemockt werden sollen. Wird ein Objekt zu einer besimmten ID benötigt, kann es beim `ObjectProvider` abgefragt werden. Falls das Objekt noch nicht existiert, wird es je nach Einstellung entweder gemockt oder instanziiert. Es ist mit dem `ObjectProvider` möglich, den Zustand der einzelnen instanziierten Objekte herzustellen, der zu einem bestimmten Zeitpunkt der Aufzeichnung vorlag.

2.7.1 Bytecode-Instrumentierung

Sowohl für die Aufzeichnung als auch die Validierung wurde Bytecode-Instrumentierung mit AspectJ verwendet.

Bytecode-Instrumentierung ist ein Werkzeug, um bereits vorhandene, kompilierte Java Programme zu modifizieren. Dabei muss der Sourcecode dieser Programme nicht bearbeitet werden oder vorliegen.

Es ist möglich, vor, nach oder während vorhander Methoden eigenen Code einzufügen und auszuführen. So können beispielsweise die Parameter oder Rückgabewerte von Methoden gespeichert oder eigener Code anstelle der Methoden ausgeführt werden. So können zusätzliche generische Funktionen (wie zum Beispiel Logging) getrennt von der Geschäftslogik in eine Software integriert werden. Diese Programmiertechnik wird aspektorientierte Programmierung genannt.

Direkt mit Bytecode zu arbeiten ist sehr umständlich und fehleranfällig. Um dies zu umgehen wird AspectJ eingesetzt. In AspectJ ist es möglich, den Code, der eingefügt werden soll (sogenannte Aspekte) mit Java ähnlicher Syntax zu erstellen.

Pointcuts

Die Aktionen, die modifiziert werden sollen, können mit sogenannten Pointcuts beschrieben werden. So können die Codestellen, die ergänzt werden sollen, sehr stark eingegrenzt oder auch sehr allgemein gehalten werden.

```
pointcut starter(): target(Car) && call(void startEngine());
```

Listing 1 stark eingrenzender Pointcut

Dieser Pointcut bezieht sich genau auf die Fälle, wenn auf einer Instanz von `Car` eine parameterlose Methode ohne Rückgabewert namens `startEngine` aufgerufen wird.

```
pointcut any(): call(* *(..));
```

Listing 2 allgemein gehaltener Pointcut

Dieser Pointcut hingegen wird auf *alle* Aufrufe bezogen, unabhängig von Rückgabetyp, Parameter, oder ob es sich um einen Konstruktor oder eine Methode handelt.

Advice

Ein Pointcut für sich alleine hat noch keine Auswirkung auf den Programmablauf. Um das zu ändern werden sogenannte Advices eingesetzt.

```
before(): starter() {
  System.out.println("Engine is about to be started");
}
```

Listing 3 Advice wird jeweils vor dem Pointcut `starter` eingefügt

Mit diesem Advice wird nun vor jedem Aufruf der Methode `startEngine()` auf einer Instanz von `Car` die Meldung «Engine is about to be started» auf der Konsole ausgegeben.

Weaving

Damit die Instrumentierung verwendet werden kann, müssen die definierten Aspekte in den Code «eingewoben» werden. Dies kann vor der Ausführung durch den AspectJ Compiler ausgeführt werden. In diesem Fall wird der zusätzliche Code fest in das Java Programm eingefügt und ab diesem Zeitpunkt immer mit ausgeführt. Eine andere Möglichkeit, die in diesem Projekt eingesetzt wird, ist Load-Time-Weaving. Dabei wird das bestehende Java Programm nicht geändert. Stattdessen werden die Aspekte zur Laufzeit in eine Klasse eingefügt, sobald diese das erste Mal geladen wird.

Damit dies möglich ist, muss die Java Virtual Machine (JVM) mit einem sogenannten Java-Agent gestartet werden. Auf der Kommandozeile kann dies mit dem folgenden Aufruf ausgeführt werden:

```
java -javaagent:jarpath[=options] -jar path-to-application-jar
```

Listing 4 Start der JVM mit einem Java-Agent [6]

Im Falle von AspectJ muss als Agent der AspectJ-Weaver angegeben werden und die verwendeten Jar-Files müssen im Classpath angegeben werden. Ist der AspectJ-Weaver vorhanden, die Aspekte kompiliert und als Jar-File verpackt, kann ein Java Programm mit «eingewobenen» Aspekten gestartet werden:

```
java -javaagent:aspectjweaver.jar -classpath "aspects.jar:app.jar"
ch.zhaw.aspectj.demo.Main
```

Listing 5 Start einer Java-Applikation mit dem AspectJ-Weaver

Mit `-classpath "aspects.jar:app.jar"` werden alle verwendeten Jar-Files eingebunden. Dabei können auch Wildcards eingesetzt werden.

`ch.zhaw.aspectj.demo.Main` bezeichnet die Klasse, welche die zu startende main-Funktion enthält.

2.7.2 Aufzeichnung

Mit AspectJ ist es nun möglich, zur Laufzeit vor und nach Konstruktor-/Methodenaufrufen eigenen Code einzufügen. So können die aufgerufenen Methoden mit den verwendeten Parametern inklusive Parameter erfasst werden.

Damit die Aufzeichnung später verwendet werden kann, ist es notwendig, dass die aufgezeichneten Aufrufe einem Objekt zugeordnet werden können. Um dies zu ermöglichen wird eine Objekt-ID eingesetzt. Diese Objekt-ID ist eine Laufnummer, die in der Reihenfolge der Konstruktoraufrufe inkrementiert wird. Damit ein Objekt eindeutig identifiziert werden kann, wird im Wesentlichen die Objektidentität verwendet. Dabei handelt es sich typischerweise um die umgerechnete Objekt-Adresse [7], [8].

Bei der Aufzeichnung werden nicht die eigentlichen Objekte gespeichert, sondern eine Abstraktion davon. Dabei wird unterschieden zwischen Werte- und Referenztypen. Als Wertetypen werden die primitiven Typen verstanden (boolean, byte, char, short, int, long, float, double), ebenfalls als Wertetypen behandelt werden Strings, void (bei Methoden ohne Rückgabewert), sowie die Objekt-Repräsentationen der primitiven Typen. Bei den Wertetypen wird der Wert in der Aufzeichnung festgehalten. Die Referenztypen beinhalten alle verbleibenden Objekte. Bei den Referenztypen wird kein Wert, sondern lediglich die ID der Objekte gespeichert. Bei beiden Typen wird stets der Klassen- oder Typname festgehalten. So ist es möglich die Objekte bei der Validierung wieder herzustellen.

Alle Konstruktor-/Methodenaufrufe werden als `Executions` gespeichert. Eine `Execution` beinhaltet die Information auf welchem Objekt der Aufruf erfolgt ist, ein Array mit den Parametern für den Aufruf, sowie eine Sequenznummer. Die Sequenznummer ist eine Laufnummer, die in der Reihenfolge der Aufrufe inkrementiert wird. Damit ist es möglich, in einem Sequenzdiagramm die entsprechende Sequenz zu lokalisieren und die zeitliche Abfolge der Aufrufe festzuhalten.

Die festgehaltenen `Executions` werden anschliessend in einer Baumstruktur gespeichert. Dadurch ist es möglich die Hierarchie der verschiedenen Aufrufe festzuhalten. Dieser Baum repräsentiert das beobachtete Verhalten einer Software und wird später als Grundlage für die Validierung der überarbeiteten Software verwendet. Der Baum könnte auch dazu eingesetzt werden, um automatisiert ein Sequenzdiagramm der beobachteten Software zu erzeugen.

Wurde die beobachtete Software terminiert, wird das Baum-Objekt serialisiert und als Datei abgespeichert. Diese Trace-Datei muss für die Validierung wieder geladen werden.

2.7.3 Validierung

Für die Validierung muss zunächst ein Trace-Baum geladen werden. Es ist nun möglich alle vorhandenen Objekt-IDs zu validieren (es ist auch möglich, nur einzelne IDs zu prüfen, dabei wird jedoch nur ein isolierter Test und keine iterative Analyse durchgeführt). Dabei wird das zuvor beschriebene Prinzip der iterativen Analyse eingesetzt.

Mit AspectJ ist es möglich ganze Konstruktor-/Methodenkörper durch eigenen Code zu ersetzen. Dieser Umstand wird bei der Validierung ausgenutzt, um Konstruktoraufrufe abzufangen. Im so eingesetzten Code wird dann je nach Situation entweder ein Mock generiert oder eine Objektinstanz erstellt (im zweiten Fall wird vom eigenen Code der echte Konstruktor des Objektes aufgerufen).

Um von der Aufzeichnung abweichende Ausführungen für die spätere Analyse festzuhalten, werden `FailedExecution` Objekte verwendet. Ein solches Objekt enthält die ursprüngliche fehlgeschlagene `Execution`. Im Laufe der Iterativen Analyse ist es möglich, weitere «Ebenen» fehlgeschlagener `Executions` hinzuzufügen. Diese werden intern in einem Stack verwaltet. Aus einer `FailedExecution` ist es nun möglich, die IDs aller beteiligten Objekte oder einen Trace der enthaltenen Sequenzen zu extrahieren. Erstere werden benötigt, damit während der iterativen Analyse konfiguriert werden kann, welche Objekte instanziiert werden müssen. Mit dem Sequenz-Trace ist es möglich am Schluss der Analyse anzuzeigen, wie stark sich eine Änderung ausgewirkt hat, respektive welche Sequenzen davon betroffen sind.

Während der Vorbereitungsphase der Analyse werden die fehlgeschlagenen `Executions` in einer Liste gespeichert. Bei der weiteren Analyse sollen nun nur diejenigen Sequenzen geprüft werden, welche nicht im Zuge der Ausbreitung tiefer gelegener `FailedExecutions` implizit geprüft werden. Daher werden vor der iterativen Analyse alle `FailedExecutions` aus der Liste entfernt, die untergeordnete `Executions` haben, welche in der Vorbereitungsphase als abweichend markiert wurden.

Die Ergebnisse der Analyse werden in einem Resultat-Objekt gespeichert. Nach der abgeschlossenen Analyse können diese Ergebnisse auf der Konsole ausgegeben werden. Es wäre auch möglich die Resultate für die spätere Verwendung in einem geeigneten Format (zum Beispiel JSON oder XML) abzuspeichern.

2.7.4 Kernstück: Objektverwaltung

Ein zentrales Element der Realisierung ist die Objektverwaltung. Dieses Element wurde in der Klasse `ObjectProvider` implementiert und verwaltet die verwendeten Objekte während der Validierung.

Im einfachsten Fall werden die Objekte mit der ID beim `ObjectProvider` angefordert. Innerhalb des `ObjectProviders` wird nun geprüft, ob das Objekt bereits existiert. Ist das der Fall, wird das Objekt zurückgegeben. Existiert das Objekt noch nicht, wird geprüft, ob das Objekt instanziiert oder gemockt werden muss und die entsprechende Aktion wird ausgeführt. Das erzeugte Objekt wird für die weitere Verwendung in einem internen Objekt-Pool gespeichert und zurückgegeben. Für die Instanziierung von Objekten werden jeweils die Konstruktoraufrufe verwendet, die bei der Aufzeichnung festgehalten wurden.

Es ist möglich, festzulegen, zu welchen IDs eine Instanz erzeugt werden muss. Anschliessend kann das Objekt angefordert und die Instanz erzeugt werden.

Auf diese Art ist es möglich, alle verwendeten Objekte dann zu erzeugen, wenn sie gerade benötigt werden. Das ist beispielsweise der Fall, wenn ein Konstruktor abgefangen wurde oder Objekte als Parameter für andere Aufrufe verwendet werden müssen.

2.8 Herausforderungen

Bei der Umsetzung sind verschiedene Schwierigkeiten aufgetreten. Manche davon waren konzeptioneller Art, andere ergaben sich beim Erstellen der Beispiel-Software. Nicht abschliessend werden hier einige Schwierigkeiten erläutert und Lösungsansätze präsentiert.

2.8.1 Statische Methoden

Statische Methoden können direkt auf einer Klasse aufgerufen werden. Es muss also kein Objekt instanziiert werden. Solche Methoden werden oft in Hilfsklassen eingesetzt, um beispielsweise Berechnungen, die häufig und immer gleich erfolgen auszulagern. Ein Beispiel aus meiner Arbeit ist die Hilfsklasse `Verifications`, die es ermöglicht zwei Objekte miteinander zu vergleichen. Ein weiteres Beispiel für eine Klasse mit statischen Methoden ist die Klasse `java.lang.Math`, welche unter anderem Methoden für trigonometrische Berechnungen bereitstellt.

Mit dem aktuellen Stand des Tools werden keine statischen Aufrufe aufgezeichnet. Grundsätzlich können mit AspectJ auch statische Aufrufe abgefangen und somit aufgezeichnet werden. Da aber das Konzept darauf beruht instanziierte Objekte zu validieren, ist mein Ansatz mit statischen Methoden nicht kompatibel.

Um diese Schwierigkeit zu beheben, müsste das Prüfprinzip so erweitert werden, dass auch statische Aufrufe berücksichtigt werden können. Dabei kann eine weitere Schwierigkeit auftreten. Typischerweise haben statische Methoden keinen Zugriff auf Instanzvariablen. Somit hat ein «statisches Objekt» keinen Zustand. Es ist aber möglich, statische Klassenvariablen zu verwenden. Dies wiederum würde statischen Methoden ermöglichen den Zustand der Klasse zu ändern. Dieser Umstand muss berücksichtigt werden, wenn das Prüfprinzip erweitert wird.

Werden mit dem aktuellen Tool statische Methoden verwendet, erscheinen diese nicht im Trace. Ändert sich nun das Verhalten einer statischen Methode, kann sich das auf das Verhalten der aufrufenden Methode auswirken. In diesem Fall wird die Methode, welche den statischen Aufruf getätigt hat, als die eigentliche Ursache der Änderung angesehen.

2.8.2 Singletons

Bei Singletons ist es üblicherweise nicht möglich, den Konstruktor aus anderen Klassen aufzurufen. Typischerweise steht eine statische Methode zur Verfügung, mit welcher die Instanz des Singletons angefordert werden kann. Ist zu diesem Zeitpunkt noch keine Instanz vorhanden, wird diese neu erzeugt.

Dieses Prinzip bringt verschiedene Probleme mit sich. Für das Tool sind sowohl der Konstruktoraufruf, wie auch die verwendeten Methoden sichtbar. Da per Reflection auch private Konstuktoren aufgerufen werden können, kann der Singleton problemlos auf sein Verhalten isoliert von Abhängigkeiten geprüft werden. Es ist für das Tool allerdings nicht möglich herauszufinden, mit welcher statischen Methode der Singleton angefordert werden kann (typischerweise heisst diese `getInstance()` oder ähnlich, der Name kann jedoch beliebig sein).

Ein Singleton wird während der Laufzeit eines Programmes nur einmal instanziiert. Dadurch kann zwar der erste Konstruktoraufruf während der Validierung abgefangen werden um beispielsweise einen Mock zu erzeugen. Wird aber von mehreren Objekten oder während der iterativen Analyse auf den Singleton zugegriffen, ist es nicht mehr möglich, die Instanz durch ein neues Objekt zu ersetzen, da unbekannt ist, wo die Referenz angepasst werden muss. Das ist allerdings notwendig, um gegebenfalls eine Instanz oder einen Mock zu verwenden, oder um den Zustand des Singletons für einen bestimmten Zeitpunkt herzustellen.

Um dies zu umgehen, müsste das Validierungstool in der Lage sein, den Namen der Klassenvariablen zu ermitteln, welche die Instanz des Singletons enthält (da dieser Name beliebig sein kann, ist das praktisch nicht umsetzbar). Eine weitere Möglichkeit wäre, Objekt-Proxies einzusetzen, die beim ersten Konstruktoraufruf erzeugt würden. Die Klassenvariable würde nun auf den Proxy zeigen. Innerhalb des Proxies könnte nun das Objekt ersetzt werden, an welches alle Methodenaufrufe weitergeleitet werden.

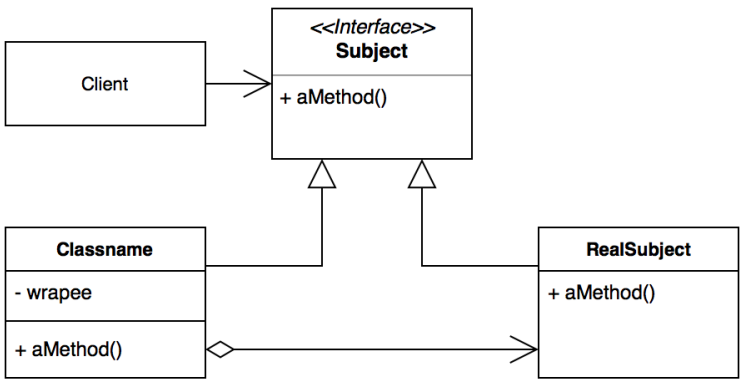


Abbildung 12 Proxy-Pattern [9]

Solche Proxies könnten mithilfe von CgLib automatisch generiert werden.

2.8.3 Enums / finale Klassen

Enums sind in Java implizit als finale Subklasse von java.lang.Enum ausgelegt [10]. Werden nun in einer Software, die analysiert werden soll Enums eingesetzt, wird bei der Validierung versucht, diese Enums zu mocken. CgLib erstellt Mocks, indem es eine Subklasse der zu mockenden Klasse erzeugt. Im Fall von Enums (oder anderen finalen Klassen) kann somit keine Subklasse und dadurch auch kein Mock erstellt werden. Um dies zu umgehen, müsste ein Tool gefunden werden, dass es ermöglicht auch von finalen Klassen dynamisch Mocks zu erzeugen. Eine andere Möglichkeit wäre es, den Schutz von `final` mithilfe von Bytecode-Manipulation zu umgehen. Dafür müsste ein zusätzlicher Java Agent verwendet werden, der diese Aufgabe übernimmt.

2.8.4 Zustand von Objekten

Werden während der iterativen Analyse Sequenzen geprüft, bei welchen mehrere instanziierte Objekte involviert sind, ist es notwendig, dass sich diese Objekte im richtigen Zustand befinden. Dafür muss für alle Objekte der Zustand hergestellt werden, in welchem sie sich bei der Aufzeichnung vor der zu prüfenden Sequenz befunden haben.

Dieses Problem wurde mit dem `ObjectProvider` gelöst. Mit dem `ObjectProvider` ist es möglich, die Zustände für einen bestimmten Zeitpunkt in der Aufzeichnung wiederherzustellen. Dafür muss zuerst festgelegt werden, welche Objekte instanziiert werden müssen. Anschliessend kann der Zustand *vor* einer bestimmten Sequenznummer hergestellt werden. Dafür wird der Aufzeichnungsbaum traversiert um zu bestimmen, welche Methoden ausgeführt werden müssen. Dabei wird darauf geachtet, dass nur diejenigen Methoden ausgeführt werden, welche tatsächlich benötigt werden. Wird zum Beispiel von einem instanziierten Objekt eine Methode eines anderen ebenfalls instanziierten Objekts aufgerufen, muss diese zweite Methode nicht mehr aufgerufen werden, da sie implizit im ersten Aufruf enthalten ist. Erfolgt ein Aufruf auf einem gemockten Objekt, der gemäss Sequenzdiagramm den Aufruf einer Methode auf einem instanziierten Objekt zur Folge hätte, muss dieser letzte Aufruf separat ausgeführt werden.

Beispiel

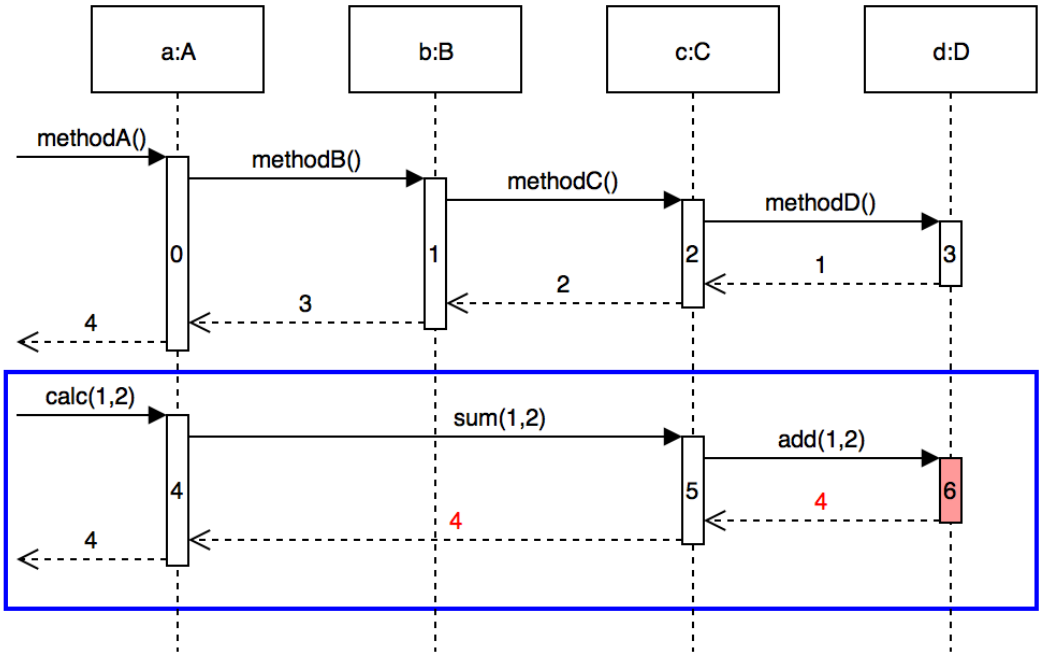


Abbildung 13 Sequenzdiagramm für die Statuswiederherstellung

Abbildung 13 stellt das Sequenzdiagramm eines zu validierenden Programmes dar. In der aktuellen Iteration soll nun der blau markierte Block auf sein Verhalten hin analysiert werden. Bevor die Methode `calc(1, 2)` aufgerufen werden kann, muss der Zustand aller instanziierter Objekte wiederhergestellt werden. Da für diese Iteration die Objekte a, c und d instanziiert sind, müssen die Sequenzen #0, #2 und #3 ausgeführt werden um den Zustand zu rekonstruieren.

Der `ObjectProvider` prüft dazu zuerst die Methode `methodA()`. Da die untergeordnete Methode `methodB()` auf einem Mock ausgeführt wird (b wird nicht instanziiert), kann `methodA()` ausgeführt werden. Das Objekt a befindet sich nun bereits im richtigen Zustand für die Analyse. Um auch die Objekte c und d in den richtigen Zustand zu versetzen, prüft der ObjectProvider als nächstes die Methode `methodC()`. Diese Methode ruft `methodD()` auf dem ebenfalls instanziierten Objekt d auf. Somit genügt es, auf c `methodC()` aufzurufen, da `methodD()` implizit aufgerufen wird. Anschliessend befinden sich die Objekte c und d ebenfalls im Zustand vor dem Aufruf von `calc(1, 2)`.

2.9 Beispiel-Software

Um die Funktion des Aufzeichnungs- und Validationstools prüfen und demonstrieren zu können, habe ich eine einfache Software erstellt. Das Verhalten dieser Software kann aufgezeichnet werden.

Wird nun der Code bearbeitet, so kann er basierend auf der zuvor erstellten Aufzeichnung validiert werden, um so die Änderungen lokalisieren und eingrenzen zu können.

In der erstellten Beispiel-Software ist ausserdem bereits ein Mechanismus integriert, mit welchem verschiedene vorbereitete Szenarien durchgespielt werden können ohne den Code bearbeiten zu müssen. Aus Gründen der Lesbarkeit wurde dieser Mechanismus jedoch in den nachfolgenden Code-Listings entfernt.

Abgesehen davon, verschiedene Möglichkeiten für Code-Manipulationen zu bieten erfüllt diese Beispiel-Software keinen weiteren Zweck.

Der Code der Beispiel-Software im Detail:

```
public class A {
  private B b = new B();

  public int f(int arg) {
    int result = b.doCalculation(arg);
    b.set(result);
    return result;
  }

  public B getB() {
    return b;
  }
}
```

Listing 6 Klasse A der Beispiel-Software

```
public class B {
  private int callCounter;
  private C c = new C();

  public C getC(D d) {
    doSomething(d);
    return c;
  }

  public void set(int arg) {
    c.setBase(arg);
  }

  private synchronized void incrementCounter() {
    callCounter++;
  }

  public int getCallCounter() {
    return callCounter;
  }

  public int doCalculation(int arg) {
    incrementCounter();
    int delta = c.getDelta(arg);
    return c.add(delta);
  }

  private void doSomething(D d) {
    c.setD(d);
    d.getCallCounter();
  }
}
```

Listing 7 Klasse B der Beispiel-Software

```
public class C {
  private int base;

  private D d;

  public void setBase(int base) {
    this.base = base;
  }

  public int getDelta(int arg) {
    return Math.abs(base - arg);
  }

  public void setD(D d) {
    this.d = d;
  }

  public int add(int arg) {
    return d.calculate(arg, base);
  }

  public int getBase() {
    return base;
  }
}
```

Listing 8 Klasse C der Beispiel-Software

```
public class D {
  private int callCounter = 0;

  public int getCallCounter() {
    return callCounter;
  }

  public int calculate(int a, int b) {
    callCounter++;
    return a + b
  }
}
```

Listing 9 Klasse D der Beispiel-Software

```
public class Main {
  public static void main(String[] args) {
    A a = new A();
    B b = a.getB();

    D d = new D();
    C c = b.getC(d);
    c.getBase();
    a.f(7);
    a.f(5);
    b.getCallCounter();
    c.getBase();
    d.getCallCounter();
  }
}
```

Listing 10 Main-Klasse der Beispiel-Software

Wird dieses Programm ausgeführt, resultiert daraus das folgende Sequenzdiagramm:

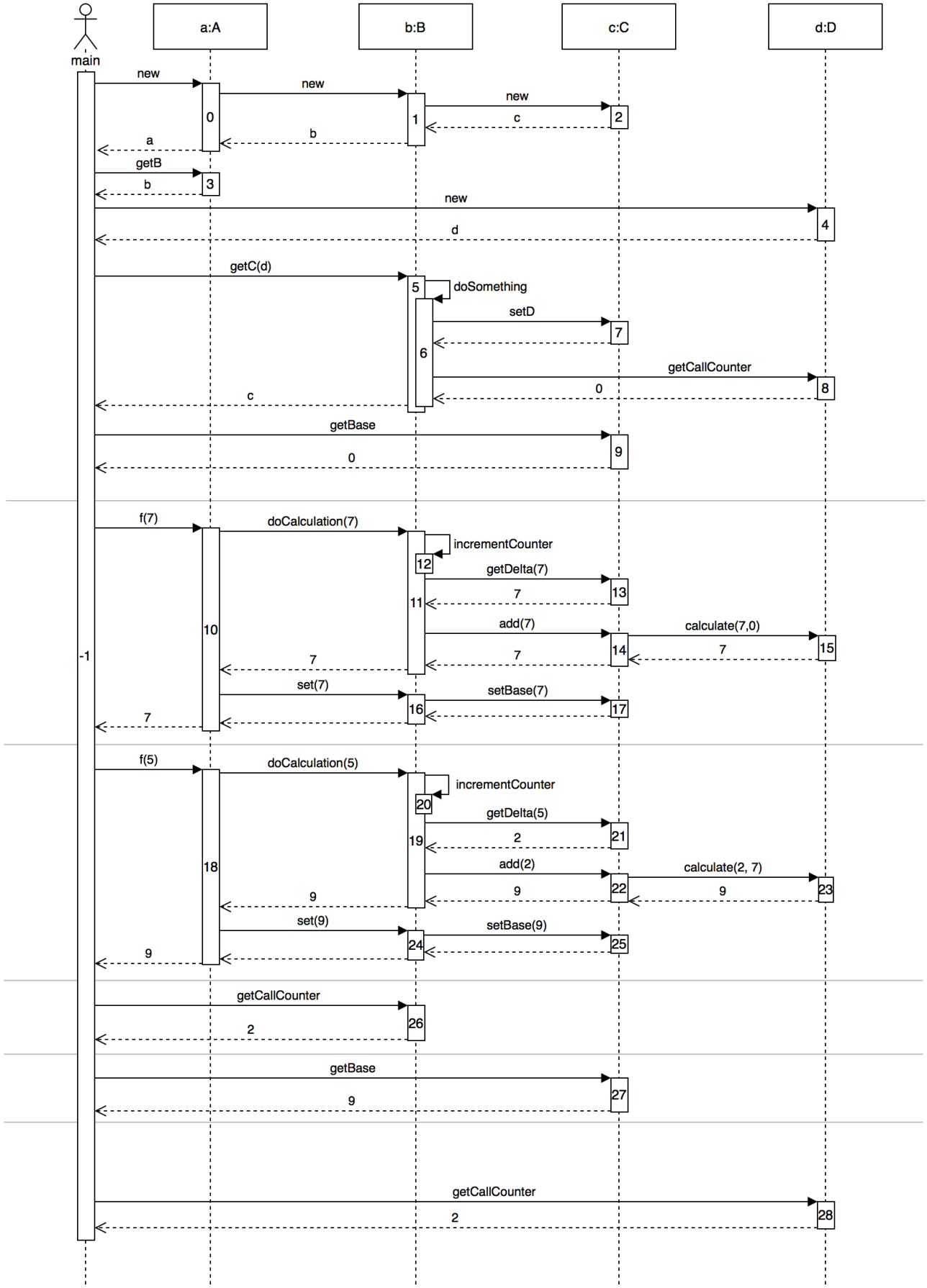


Abbildung 14 Sequenzdiagramm der Beispiel-Applikation

3 Resultate / Fazit

In dieser Arbeit wurde untersucht, ob ein Tool realisiert werden kann, das aufgrund aufgezeichneter Laufzeitdaten ein Programm vor und nach einem Refactoring analysieren und validieren kann. Auf der Basis des Augest-Projekts konnte ein solches Tool entwickelt werden. Mit diesem Tool kann bei sehr einfachen Programmen nach einem Refactoring untersucht werden, welche Objekte sich nach dem Refactoring gleich verhalten wie davor. Für Objekte, die nach dem Refactoring ein geändertes Verhalten aufweisen, kann zudem ermittelt werden, wie weit sich diese Änderung nach oben ausbreitet.

Für diese Grundfunktion konnte bewiesen werden, dass es möglich ist, dieses Prinzip umzusetzen. Die Verhaltensänderung kann in der Beispiel-Applikation für verschiedene Fälle durchgeführt werden. Für einen realen Einsatz müssen noch verschiedene Probleme gelöst werden. Insbesondere wenn das Tool eingesetzt werden soll, um das Verhalten einer Software vor und nach einem umfangreichen Refactoring zu vergleichen, ist das Tool auf der aktuellen Entwicklungsstufe ungeeignet. Zu zahlreich sind die Einschränkungen.

3.1 Ändernde Signatur

Ändert sich die Signatur eines Konstruktors oder einer Methode, kann nicht mehr zugeordnet werden, welche Methode der ursprünglich aufgezeichneten entsprechen würde. Selbst wenn das möglich wäre, ist es bei zusätzlichen Parametern nicht möglich zu ermitteln, welche Werte dafür verwendet werden müssen.

Bei einem Refactoring werden typischerweise häufig Signaturen geändert. Oft werden auch Methoden oder ganze Klassen extrahiert, was ebenfalls dazu führen kann, dass eine Methode mit der bekannten Signatur nicht mehr verfügbar ist.

3.2 Änderungen im Sequenzdiagramm

Das vorliegende Tool geht für die Analyse davon aus, dass sich das Sequenzdiagramm, also der Programmablauf, zwischen Aufzeichnung und Validierung nicht verändert hat. Das bedeutet, dass sowohl bei der Aufzeichnung als auch bei der Validierung die gleichen Objekte in der gleichen Reihenfolge mit den selben Methoden verwendet werden, wie es bei der Aufzeichnung der Fall war. Würde sich das Diagramm ändern, kann der neue Ablauf nicht sinnvoll mit dem aufgezeichneten verglichen werden. Aus den selben Gründen, wie sich die Signatur oft ändert (Extraktion von Methoden/Klassen) wird das Sequenzdiagramm nach einer Überarbeitung häufig anders ausfallen.

Auch abweichende Aufrufe von Methoden auf Mocks können nur beschränkt detektiert werden. Wurde eine Abweichung festgestellt, kann keine weitere mehr detektiert werden, da sich das Tool nicht mehr mit den weiteren erwarteten Aufrufen synchronisieren kann.

3.3 «Verlorene» Abweichungen

Nach der Vorbereitungsphase werden alle `FailedExecutions` welche eine tieferliegende Sequenz mit abweichendem Verhalten haben, aus der Liste der weiter zu analysierenden Sequenzen entfernt. So kann es passieren, dass auch `FailedExecutions` entfernt werden, deren geändertes Verhalten nicht im Zusammenhang mit einer untergeordneten Sequenz liegt.

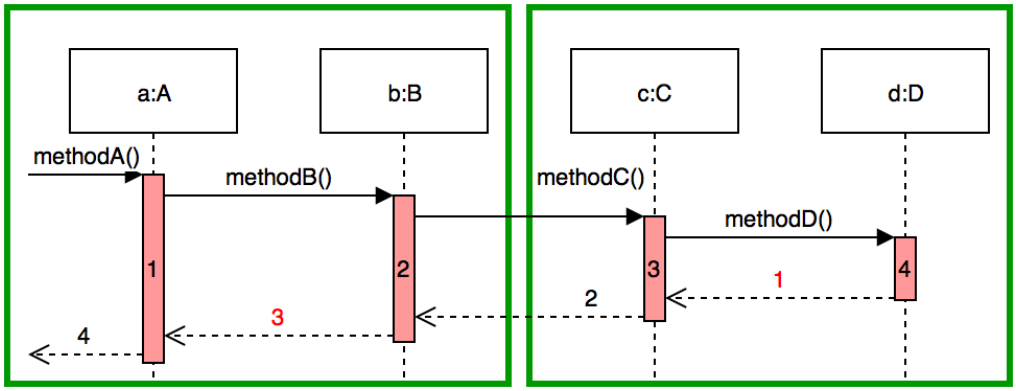


Abbildung 15 Zwei unabhängige Blöcke mit validem Verhalten

Im Beispiel wurden alle Sequenzen in der Vorbereitungsphase als abweichend markiert. Sequenz #4 hat einen veränderten Rückgabewert. Sequenz #3 wurde so angepasst, dass diese Änderung wieder ausgeglichen wird. Der Block bestehend aus den Objekten c und d verhält sich also von aussen betrachtet gleich, wie es bei der Aufzeichnung der Fall war. Das gleiche Prinzip kann sinngemäss auf den Block bestehend aus den Objekten a und b angewandt werden. Dabei resultiert aus Sequenz #2 ein veränderter Rückgabewert, der von Sequenz #1 ausgeglichen wird.

Bei dem realisierten Tool werden nach der Vorbereitungsphase die Sequenzen #1, #2 und #3 aus der Liste der weiter zu analysierenden Sequenzen entfernt, da die tiefer liegende Sequenz #4 ebenfalls als abweichend markiert ist. Bei der iterativen Analyse von Sequenz #4 wird nun c ebenfalls instanziiert und die Methode `methodC()` aufgerufen. Diese verhält sich entsprechend der Aufzeichnung und die weitere Analyse wird abgebrochen. Deshalb wird die Analyse von Sequenz #1 und #2 nicht mehr durchgeführt.

3.4 Statusänderungen

Werden auf einem Objekt Methoden aufgerufen, ist es möglich, dass sich der Zustand dieses Objekts ändert. Diese Statusänderungen können einen Einfluss auf das spätere Verhalten eines Objektes haben. Das Validierungstool kann zwar feststellen, wenn Methoden anders aufgerufen werden, als es bei der Aufzeichnung der Fall war, es ist momentan aber nicht möglich, geändertes Verhalten, welches später erfolgt, auf diese geänderten Aufrufe zurückzuführen. Das heisst, es wird sowohl der geänderte Aufruf als auch das spätere, geänderte Verhalten festgestellt und analysiert. Diese Analysen erfolgen jedoch isoliert voneinander, weshalb kein Zusammenhang zwischen den Änderungen hergestellt werden kann.

3.5 Aufrufe eigener Methoden

Natürlich kann ein Objekt nicht nur Methoden anderer Objekte aufrufen, sondern auch eigene. Auf einem Sequenzdiagramm sieht ein solcher Aufruf wie folgt aus:

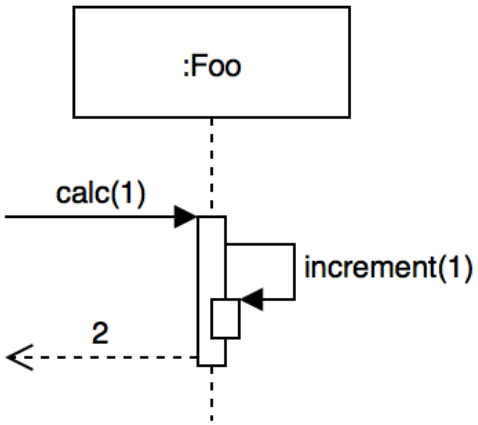


Abbildung 16 Aufruf der eigenen Methode `increment`

```
public class Foo {
  // ...
  private int increment(int arg) {
    return arg + 1;
  }

  public int calc(int arg) {
    return increment(arg);
  }
  // ...
}
```

Listing 11 Aufruf einer eigenen Methode in Java

Ein solcher *interner* Aufruf wird bei der Validierung nicht detektiert und überprüft. Analog wird bei der iterativen Analyse nicht aufgezeichnet, wenn instanziierte Objekte Methoden anderer ebenfalls instanziierter Objekte aufrufen. Wenn sich nun die internen Aufrufe ändern, was je nach Implementierung zu Statusänderungen führen kann, so kann das nicht auf den geänderten Aufruf zurückgeführt werden.

3.6 Package

Damit bei der Validierung nicht Third-Party-Libraries oder Bestandteile der Java Runtime geprüft werden, müssen die Aspekte so definiert sein, dass die Bytecode-Instrumentierung nur innerhalb bestimmter Packages erfolgt.

```
pointcut myClass(Object obj): this(obj) && within(ch.zhaw.augestdemo..*);
```

Listing 12 Pointcut der nur innerhalb eines bestimmten Packages zutrifft

Dieses Package ist momentan fest einprogrammiert. Das bedeutet, dass sich entweder die zu prüfende Software an die Package-Konvention halten muss, die vom Tool vorgegeben ist, oder das Validierungstool je nach Software angepasst und neu kompiliert werden müsste.

4 Ausblick

Der untersuchte Ansatz kann nur mit sehr starken Einschränkungen umgesetzt werden. Die meisten der im Fazit erwähnten Einschränkungen eignen sich als weitere Forschungsziele. Einige mögliche nächste Schritte werden nachfolgend erläutert:

- **Systemaufrufe:** Um die Einschränkung von Systemaufrufen zu entschärfen wäre ein möglicher weiterer Schritt, zu untersuchen, wie eine funktionierende Systemumgebung simuliert werden könnte, ohne dabei sämtliche Objekte abspeichern zu müssen. In einem nächsten Schritt könnten Third-Party-Libraries in diese Überlegung miteinbezogen werden.
- **Grafische Oberfläche/Konfigurationsmöglichkeit:** Erfolgt ein Refactoring in sehr kleinen Schritten und werden die Einschränkungen dabei berücksichtigt (das heisst das Validierungsergebnis wird dementsprechend interpretiert), kann auf dem bisher geprüften Ansatz aufgebaut werden. Für den Einsatz eines solchen Tools muss die Bedienbarkeit verbessert werden. Vorstellbar ist eine grafische Oberfläche, mit welcher die Konfiguration und Visualisierung der Aufzeichnung und des Ergebnisses vorgenommen werden kann. Es ist möglich, einen aufgezeichneten TraceTree als Sequenzdiagramm darzustellen. In diesem Diagramm könnten nach der Analyse die festgestellten Änderungen mit ihren Auswirkungen eingezeichnet werden. Ein solches Tool sollte auch über verschiedene Konfigurationsmöglichkeiten für die Aufzeichnung und die Validierung verfügen. Wurden grosse Bereiche des Codes überarbeitet könnten so ganze Klassen bereits im Vorfeld von der Analyse ausgeschlossen werden, um ein brauchbares Ergebnis zu erzielen. Auch muss es möglich sein, zu konfigurieren, in welchem Package (oder welchen Packages) sich der zu analysierende Code befindet. Informationen über die Position der Änderung im Code (Datei, Zeilennummer) wären für eine Auswertung hilfreich und sollten in eine Weiterentwicklung integriert werden.
- **Statusänderungen:** Um Statusänderungen zu detektieren und vor allem auf eine Ursache zurückführen zu können, müsste ein entsprechendes Konzept ausgearbeitet werden. Ein solches Konzept könnte darauf aufbauen, dass von einem Objekt alle Änderungen stets mit dem Wissen um andere Änderungen dieses Objekts analysiert werden. Technisch könnte das mit moderatem Engineeringaufwand umgesetzt werden. Um die Zusammenhänge herzustellen müsste allerdings ein heuristischer Ansatz verfolgt werden, da nie ganz klar ist, was innerhalb einer Methode genau geschieht. Somit könnte nur eine Vermutung über die Herkunft der Statusänderung angestellt werden. Dieses Problem kann eventuell entschärft werden, indem auch Instanzvariablen überwacht werden, was jedoch erheblichen Mehraufwand mit sich bringt.
- **Abweichungen im Sequenzdiagramm:** Treten nach einem Refactoring Änderungen im Sequenzdiagramm auf, kann untersucht werden, inwiefern trotzdem ein Vergleich mit der Aufzeichnung möglich ist. Ähnlich wie bei den Statusänderungen ist es vorstellbar, hier einen heuristischen Ansatz zu verfolgen. Weiter wäre es auch möglich, gemeinsame Teile der Diagramme zu finden, und die Validierung nur für diese Stellen auszuführen. Auch der Umgang mit geänderten Methodensignaturen gehört zu diesem Punkt.

- **Statische Aufrufe:** Auch statische Aufrufe sollten in der Validierung berücksichtigt werden. Die Aufzeichnung dieser Aufrufe sollte keine grosse Hürde darstellen. Es müssten lediglich die entsprechenden Aspekte ergänzt werden. Es muss jedoch ein Konzept ausgearbeitet werden, wie diese statischen Aufrufe in Einklang mit dem besprochenen Ansatz der Validierung gebracht werden können.
- **Interne Methodenaufrufe:** Es kann überprüft werden, wann die Berücksichtigung von internen Methodenaufrufen bei der Validierung Sinn macht. Eine solche Analyse wäre in den bereits vorhandenen Ansatz gut integrierbar und auch die vorhandenen Aspekte können ohne allzu grosse Anpassungen verwendet werden. Es muss dabei noch erarbeitet werden, wie diese Aufrufe im aktuellen Prüfvorgang berücksichtigt werden können.
- **Performanz:** Gerade bei grossen Aufzeichnungen kann es sein, dass die Analyse lange Zeit in Anspruch nimmt. Um die Performanz einer Analyse zu steigern, könnte untersucht werden, ob und wie der dynamische Ansatz mit statischer Code-Analyse kombiniert werden kann. So wäre es möglich, die Analyse nur an denjenigen Stellen auszuführen, an denen Codemodifikationen erfolgt sind. Es kann auch untersucht werden, wie die iterative Analyse optimiert werden kann, damit nicht jedesmal alle Zustände wiederhergestellt werden müssen. Ein weiterer Untersuchungspunkt kann die Prüfreihenfolge der fehlgeschlagenen Ausführungen sein. Eventuell kann durch geschickte Wahl der Reihenfolge die Performanz verbessert werden.
- **Alternative:** Eine weitere Anwendungsmöglichkeit der Aufzeichnungsdaten, welche bereits von N. Wright und D. Zolliker erwähnt wurde [2], ist Unit-Tests zu erzeugen. Interessant ist hier vorallem der erwähnte Ansatz, der vorschlägt aus den Daten Äquivalenzklassen zu bilden.
- **Aufzeichnung optimieren:** Um Langzeitaufzeichnungen umsetzen und vorallem verwenden zu können, müsste untersucht werden, wie das Volumen von Aufzeichnungen klein gehalten werden kann und somit die Validierung in nützlicher Zeit erfolgt. Wird die Aufzeichnung nicht von einem externen Tool vorgenommen, wäre auch interessant, zu untersuchen, wie die Aufzeichnung die Performanz der Programmausführung beeinflusst.
- **Objekttypen:** Für den Einsatz in einer realen Umgebung muss ein Konzept gefunden werden, wie auch Objekttypen aus der Java Laufzeitumgebung oder von Third-Party-Libraries in der Aufzeichnung erfasst werden können. Mögliche Ansätze dafür sind, alle Objekte für jeden Zustand zu serialisieren und in der Aufzeichnungsdatei zu speichern oder auch für diese Objekte alle Aufrufe aufzuzeichnen, damit die verschiedenen Zustände jederzeit wiederhergestellt werden können.

5 Verzeichnisse

5.1 Literaturverzeichnis

[1] Agitar Technologies (2009). *AgitarOne JUnit Generator* [Online]. URL: http://www.agitar.com/pdf/AgitarOneJUnitGeneratorDatasheet.pdf [Stand: 30.05.2016]

[2] D. Zolliker, N. Wright, «Neuartiges Testkonzept mittels Bytecode-Instrumentation», Zürcher Hochschule für angewandte Wissenschaften, Winterthur, 19.12.2012

[3] P. Strelecki, «Automatische Generierung von Unit-Tests», Zürcher Hochschule für angewandte Wissenschaften, Winterthur, 06.06.2014

[4] Chronon Systems. *Chronon | DVR for Java* [Online]. URL: http://chrononsystems.com/ [Stand: 30.05.2016]

[5] D. Schutzbach, F. Uzdilli, M. Cieliebak «Back to the Future. Time-Travelling-Debugger als Alternative zu klassischen Debuggern», JavaMagazin, S. 18-22, 12.2015

[6] Oracle. j*ava.lang.Instrument (Java Platform SE 7)* [Online]. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html [Stand: 30.05.2016]

[7] Oracle. *System (Java Platform SE 8)* [Online]. URL: http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#identityHashCode-java.lang.Object [Stand: 30.05.2016]

[8] Oracle. *Object (Java Platform SE 8)* [Online]. URL: http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode [Stand 30.05.2016]

[9] SourceMaking. *Proxy Design Pattern* [Online]. URL: https://sourcemaking.com/design_patterns/proxy [Stand: 30.05.2016]

[10] Oracle. Java SE Specifications – Chapter 8. Classes [Online]. URL: https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.9 [Stand: 30.05.2016]

[11] Code Generation Library. *GitHub - cglib/cglib: cglib - Byte Code Generation Library is high level API to generate and transform Java byte code. It is used by AOP, testing, data access frameworks to generate dynamic proxy objects and intercept field access* [Online]. URL: https://github.com/cglib/cglib [Stand: 08.06.2016]

5.2 Glossar

AspectJ	Java Erweiterung für die aspektorientierte Programmierung.
Augest	Software, die vom InIT als Grundlage für die Bachelorarbeit zur Verfügung gestellt wurde.
CgLib	Die Code Generation Library ist eine Java-Library, die eingesetzt werden kann um zur Laufzeit Code zu erzeugen. So können Klassen erweitert oder Interfaces implementiert werden [11].
Legacy Software	Altsoftware, die seit Langem in Betrieb ist. Typischerweise ist Legacy Software während ihrer Lebensdauer stetig erweitert worden und daher nur schwer zu ersetzen.
Methodensignatur	Definiert die Schnittstelle einer Methode. Die Methodensignatur besteht aus dem Namen der Methode, den Typen der Parameter und deren Reihenfolge sowie dem Typ des Rückgabewertes.
Mock	Test-Objekt, das üblicherweise in Unittests eingsetzt wird, um andere Objekte zu simulieren. Damit kann die Abhängigkeit von den realen Objekten aufgelöst werden.
Refactoring	Überarbeiten von bestehendem Sourcecode, ohne dabei die Funktionalität zu ändern. In der Regel in der Absicht, die Lesbarkeit, Wartbarkeit oder Erweiterbarkeit zu verbessern.
Time Travelling Debugging	Debugging-Technik, bei der es möglich ist, sich auf der Zeitachse vorwärts und rückwärts zu bewegen. Bei herkömmlichen Debuggern ist es nur möglich, sich in der Zeit vorwärts zu bewegen. Für Time-Travelling-Debugging sind spezielle Tools notwendig.
TraceTree	Aufzeichnung eines Programmverhaltens.
Wildcards	Platzhalter in Ausdrücken, um den Ausdruck allgemeiner auszudrücken.

5.3 Abbildungsverzeichnis

5.4 Listings

6 Anhang

6.1 Projektmanagement

6.1.1 Offizielle Aufgabenstellung

Was wäre, wenn man Unit-Tests automatisch generieren könnte?

Ein guter Software-Ingenieur schreibt immer Tests für seinen Code. Theoretisch. In der Realität gibt es aber Unmengen Java-Code, für die keine Tests existieren. Zum Beispiel weil der Code uralt ist, der Entwickler unter Zeitdruck war oder einfach keine Lust hatte.

Deswegen haben wir das Forschungsprojekt Augest gestartet. Die Idee ist, dass man automatisch Tests aus Protokoll-Dateien vom Laufzeitverhalten einer Software generiert. Dazu wird zunächst die Interaktion aller Software-Komponenten mit ihrem Umfeld im Betrieb aufgenommen und protokolliert. Diese Recordings können später verwendet werden, um das gesamte Programm nochmal ablaufen zu lassen. Damit kann man feststellen, ob und wo sich das Verhalten verändert hat (z.B. durch ein Refactoring). Diese Idee ist angelehnt ans Behavior Driven Testing. Wir haben bereits eine Basisversion implementiert, die den Programmablauf einer Java-Applikation aufnimmt und für unveränderte Schnittstellen wieder abspielen kann. In dieser Projektarbeit soll diese Basisversion erweitert werden, sodass sie verifizieren kann, ob die Software nach einem komplexeren Code-Refactoring noch dasselbe Verhalten zeigt wie vorher.

Die wichtigsten Teilaufgaben sind:

- Einarbeiten in die Thematik (automated) Software Testing
- Konzept basierend auf existierender Software für automatisiertes Testing
- Implementierung Prototyp
- Demonstration anhand einer Beispiel-Applikation

6.1.2 Zeitplan

KW	INHALT
8	• Einarbeitung (Aufgabenstellung)
9	• Einarbeitung (Aufgabenstellung, Augest) • Erstellen Zeitplan
10	• Einrichtung Entwicklungsumgebung (IDE, Plug-Ins, Git, ...) • Augest in lauffähigen Zustand bringen
11	• Augest in lauffähigen Zustand bringen • Grobe Gliederung Bericht
12	• Refactoring Augest • Beispiel-Software erstellen
13	• Refactoring Augest • Tests zu Software erstellen • Maven Umgebung einrichten (saubere Projektstruktur)
14	• Mock Handling • Objektverwaltung
15	• Konzept für Änderungsdetektion ausarbeiten
16	• Änderungsdetektion implementieren
17	• Änderungsdetektion implementieren
18	• Groben Inhalt des Berichts erstellen • Übersicht über noch notwendige Punkte erstellen
19	• Arbeit an Bericht • Startskript für einfache Bedienung erstellen • Refactoring und letzte Fehler beheben bei Software
20	• Arbeit an Bericht • Refactoring der Software
21	• Arbeit an Bericht
22	• Arbeit an Bericht • Bereinigung Software
23	• Zusammenfassung hochladen: 7.6.2016 • Abgabe für Druck: 9.6.2016 • Abgabe: 10.06.2016 16:00 Uhr

6.2 Verwendete Software

6.2.1 Arbeitsumgebung

Für die Entwicklung wurde unter Mac OS X gearbeitet. Für die Entwicklung in Eclipse wurden die AspectJ Development Tools (AJDT) verwendet. Es wurde die folgende Umgebung eingesetzt:

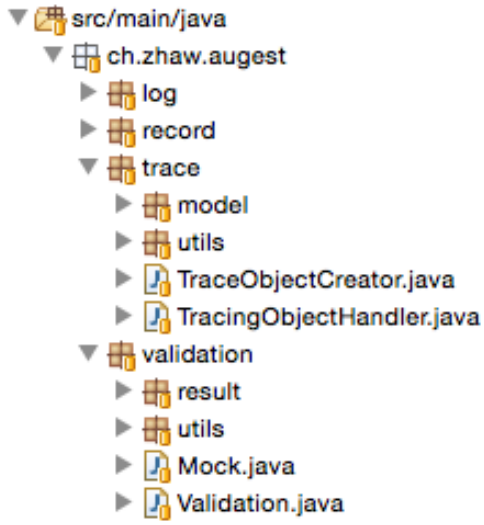
- Mac OS X 10.10.5
  - Java: 1.8.0u40
  - Maven: 3.3.9
  - Eclipse: Mars (4.5.0)
  - AJDT: 2.2.4
  - Python: 3.5.0

Der Einsatz von Maven, Startscript und Eclipse mit AJDT wurde zudem auf folgenden Plattformen geprüft:

- Windows 10
  - Java: 1.8.0u91
  - Maven: 3.3.9
  - Eclipse: Mars.2 (4.5.2)
  - AJDT: 2.2.4
  - Python: 3.5.1
- Ubuntu 16.04 LTS
  - Java: 1.8.0u91
  - Maven: 3.3.9
  - Eclipse: Mars.2 (4.5.2)
  - AJDT: 2.2.4
  - Python: 3.5.1+

6.3 Beschreibung Projektstruktur

Um das Projekt wartbar zu gestalten wurden der Code in verschiedene Packages aufgeteilt.



`ch.zhaw.augest.log`: Enthält Klassen für die Verwaltung der Konsolenausgabe.

`ch.zhaw.augest.record`: Enthält die Klassen, welche für die Aufzeichnung des Code-Verhaltens benötigt werden.

`ch.zhaw.augest.trace`: Trace-Komponenten, welche sowohl bei der Aufzeichnung als auch der Validierung verwendet werden.

`ch.zhaw.augest.trace.model`: Model-Klassen für die Trace-Darstellung.

`ch.zhaw.augest.trace.utils`: Hilfs-Klassen für die Behandlung des Trace.

`ch.zhaw.augest.validation`: Klassen, die für die Validierung eingesetzt werden.

`ch.zhaw.augest.validation.result`: Klassen für die Darstellung eines Validierungs-Resultates.

`ch.zhaw.augest.validation.utils`: Hilfsklassen, die für die Validierung benötigt werden. Dazu gehört beispielsweise der `ObjectProvider`.

6.4 Projekt mit Maven kompilieren

Das Projekt kann auf der Kommandozeile mithilfe von Maven kompiliert werden. Dafür muss im Verzeichnis `Augest/` der Befehl `mvn clean package` ausgeführt werden. Das Goal `clean` ist dabei optional und gibt an, dass vor der Kompilation alle bereits kompilierten Files gelöscht werden sollen.

6.5 Beispiel-Software

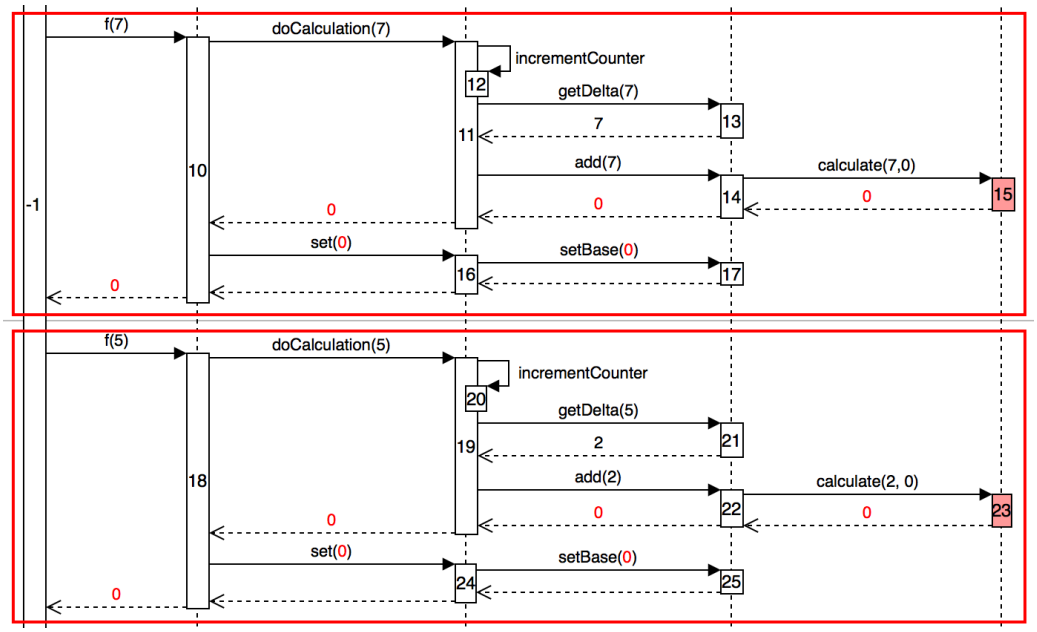
Für Demonstrationszwecke wurde in der Beispiel-Software die Möglichkeit eingebaut, das Verhalten zu konfigurieren. Es gibt drei Verhaltensoptionen. Diese können beim Einsatz des Startscripts als Option konfiguriert werden.

6.5.1 Standardverhalten

Wird beim Start die Option `DEFAULT` gewählt, verhält sich die Beispiel-Software gemäss dem Sequenzdiagramm in 2.9 Beispiel-Software.

6.5.2 Änderung mit uneingeschränkter Ausbreitung

Mit der Option `CASE_1` weist das Beispiel-Programm geändertes Verhalten auf, welches sich über die ganze Aufrufbreite erstreckt. Das folgende Sequenzdiagramm zeigt die geänderten Stellen.

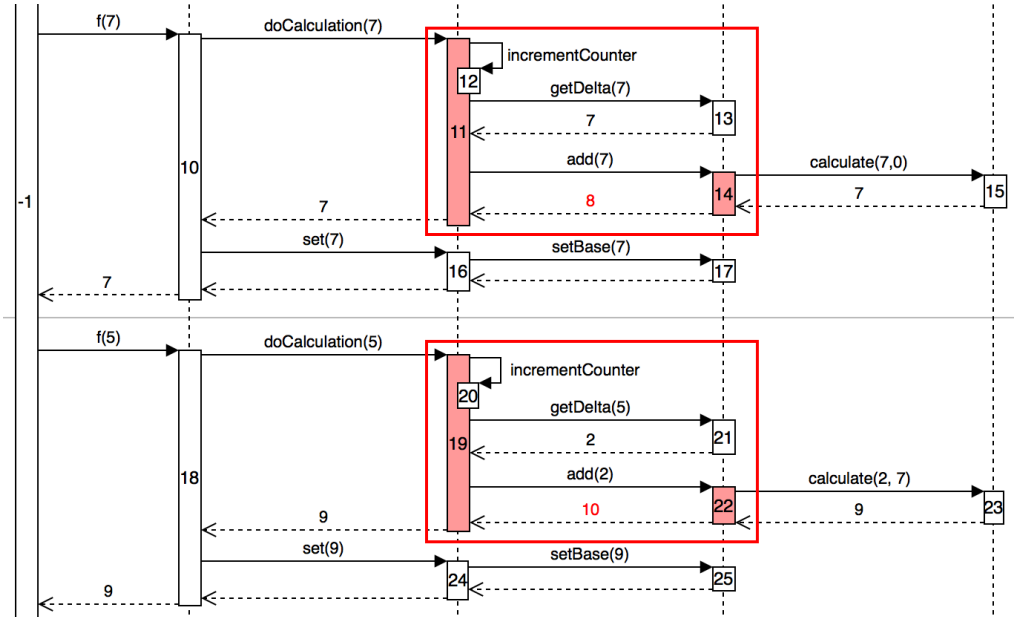


Dabei wurde lediglich die Methode `calculate` (Sequenz #15 und #23) geändert. Diese Änderung wird an keiner anderen Stelle ausgeglichen. Daher wirkt sich die Änderung bis zum Aufruf in `main` (Sequenz #-1) aus.

Da der instanziierte Block als Blackbox betrachtet wird, werden die geänderten Aufrufe `set` und `setBase` bei den Sequenzen #16, #17 resp. #24, #25 nicht detektiert. Diese haben aber eine Statusänderung des Objektes c zur Folge.

6.5.3 Änderung mit eingeschränkter Ausbreitung

Wird die Option `CASE_2` gewählt, weist die Beispiel-Software geändertes Verhalten auf, das auf einen Block eingeschränkt werden kann. Die Änderungen im Sequenzdiagramm sind nachfolgend ersichtlich.



Die Änderungen, welche in den Sequenzen #14 respektive #22 vorgenommen wurden, werden in diesem Fall in den Sequenzen #11 respektive #19 wieder ausgeglichen. Daher kann die Ausbreitung auf den rot eingerahmten Block begrenzt werden.

6.6 Startscript

Um die Bedienung des Tools zu vereinfachen, wurde ein Startscript entwickelt, welches mit der Beispiel-Software eingesetzt werden kann. Damit das Startscript eingesetzt werden kann, ist eine aktuelle Version von Python 3 erforderlich.

Vor der Verwendung des Startscripts muss das Projekt mit Maven kompiliert werden, da andernfalls die notwendigen Programme nicht verfügbar sind.

Das Startscript wird auf der Kommandozeile aufgerufen und kann sowohl unter unixoiden Betriebssystemen als auch Windows gestartet werden. Unter unixoiden Betriebssystemen wird das Script mit `./augest.py` gestartet, während es unter Windows genügt `augest.py` einzugeben.

6.6.1 Betriebsmodi

Das Startscript kann für zwei Betriebsmodi eingesetzt werden.

- `record`: In diesem Modus kann das Verhalten einer Software aufgezeichnet werden.
- `validate`: Die Validation des Softwareverhaltens kann mit diesem Modus vorgenommen werden.

Die verschiedenen Modi können mit den Befehlen

```
./augest.py validate
```

respektive

```
./augest.py record
```

ausgeführt werden.

6.6.2 Optionen

Für das Startscript sind verschiedene Optionen verfügbar.

`-h`, `--help`	Zeigt die verfügbaren Optionen an.
`-t`, `--trace`	Erlaubt die Angabe eines Dateinamens, der für die Speicherung der Aufzeichnung verwendet wird (`record`), respektive, welcher auf die Aufzeichnung zeigt, die für die Validierung verwendet werden soll (`validate`).
`-c`, `--democase`	Mit dieser Option kann gewählt werden, welches Beispiel durchgespielt werden soll (siehe 6.5 Beispiel-Software). Diese Option ist nur im Validationsmodus verfügbar. Ohne Angabe wird `DEFAULT` verwendet.
`-l`, `--loglevel`	Legt die Granularität der Konsolenausgabe fest. Mögliche Level sind: `TRACE`, `DEBUG`, `INFO`, `WARNING`, `RESULT`, `ERROR`. Ohne Angabe wird `INFO` verwendet.
`-p`, `--showlocation`	Wird dieses Flag gesetzt, wird bei der Ausgabe zusätzlich angegeben, an welcher Stelle im Code eine Nachricht erzeugt wurde.

6.6.3 Beispiele

Beispiel 1

```
./augest.py record -t trace.ser
```

Aufzeichnung des Programmverhaltens. Die Aufzeichnung wird in die Datei «trace.ser» gespeichert.

Beispiel 2

```
./augest.py validate -t trace.ser -l RESULT
```

Validierung des Programmverhaltens anhand der Aufzeichnung «trace.ser». Es werden nur Nachrichten mit dem log-Level `RESULT` ausgegeben:

Ausgabe:

```
[RESULT ] Verification result
[RESULT ]
[RESULT ] Object with id 0 behaved as expected: true
[RESULT ] Object with id 1 behaved as expected: true
[RESULT ] Object with id 2 behaved as expected: true
[RESULT ] Object with id 3 behaved as expected: true
```

Beispiel 3

```
./augest.py validate -t trace.ser -c CASE_1 -l RESULT -p
```

Validierung der geänderten Software mit dem Verhalten CASE_1. Es werden nur Resultate ausgegeben. Zusätzlich wird der jeweilige Ursprung der Log-Nachricht angegeben:

```
[RESULT ] Verification result (VerificationResult.java:31)
[RESULT ]
[RESULT ] Object with id 0 behaved as expected: true (VerificationResult.java:36)
[RESULT ] Object with id 1 behaved as expected: true (VerificationResult.java:36)
[RESULT ] Object with id 2 behaved as expected: true (VerificationResult.java:36)
[RESULT ] Object with id 3 behaved as expected: false (VerificationResult.java:36)
[RESULT ]   Trace: -1, 10, 11, 14, 15 (VerificationResult.java:49)
[RESULT ]   Trace: -1, 18, 19, 22, 23 (VerificationResult.java:49)
```

6.7 Konfiguration Eclipse

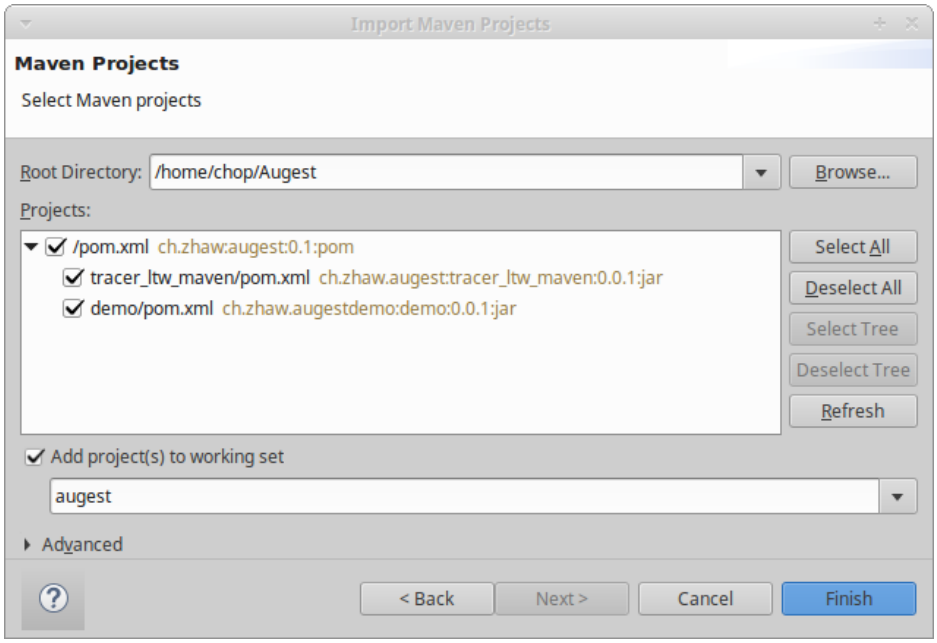
In diesem Kapitel werden die Schritte beschrieben, die notwendig sind, damit in das Projekt in Eclipse verwendet werden kann.

6.7.1 AspectJ Development Tools

Damit das Projekt in Eclipse bearbeitet werden kann, müssen die AJDT installiert werden. Installationshinweise können dem README.md auf der CD entnommen werden.

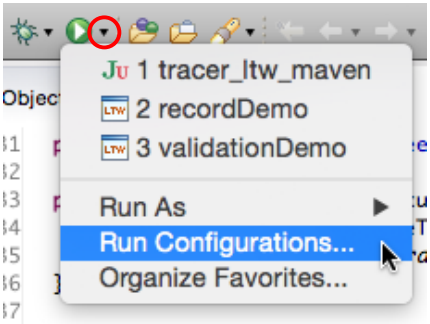
6.7.2 Maven-Import

Das Projekt kann als Maven-Projekt in Eclipse importiert werden. Dabei sollte als Import-Verzeichnis «Augest» gewählt werden. Alle drei enthaltenen Projekte sollten importiert werden.

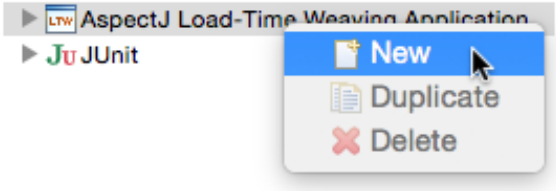


6.7.3 Run Configuration

Damit das Projekt mit «eingewobenen» Aspekten gestartet werden kann, muss eine entsprechende Run Configuration erstellt werden.

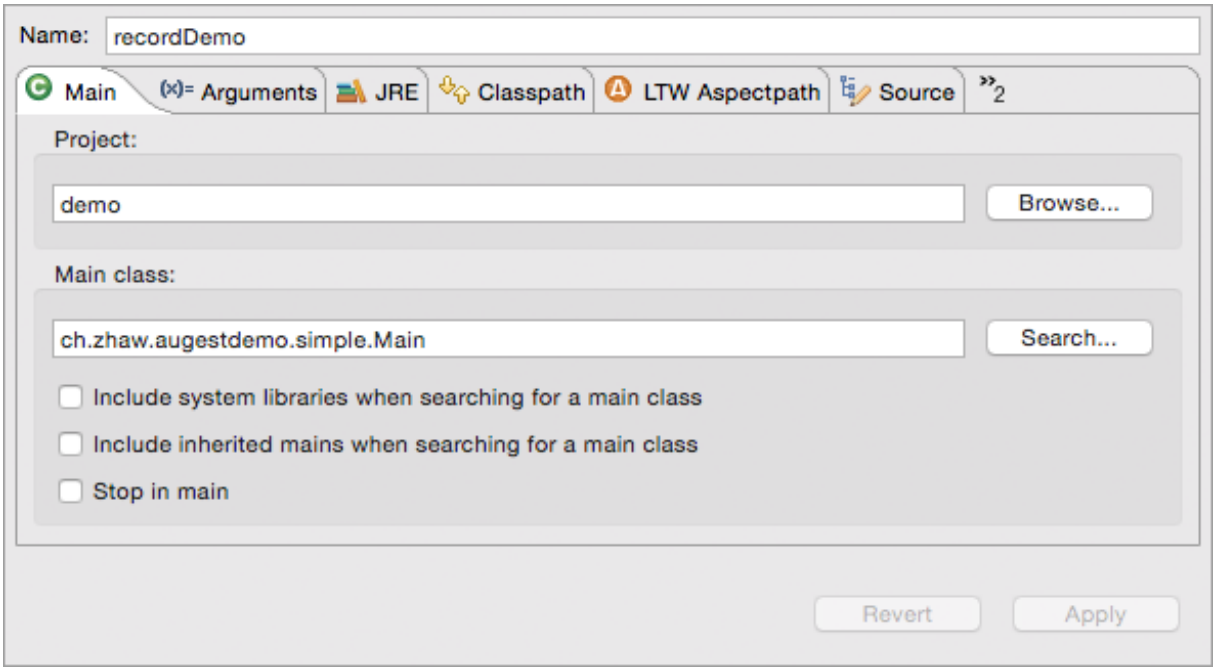


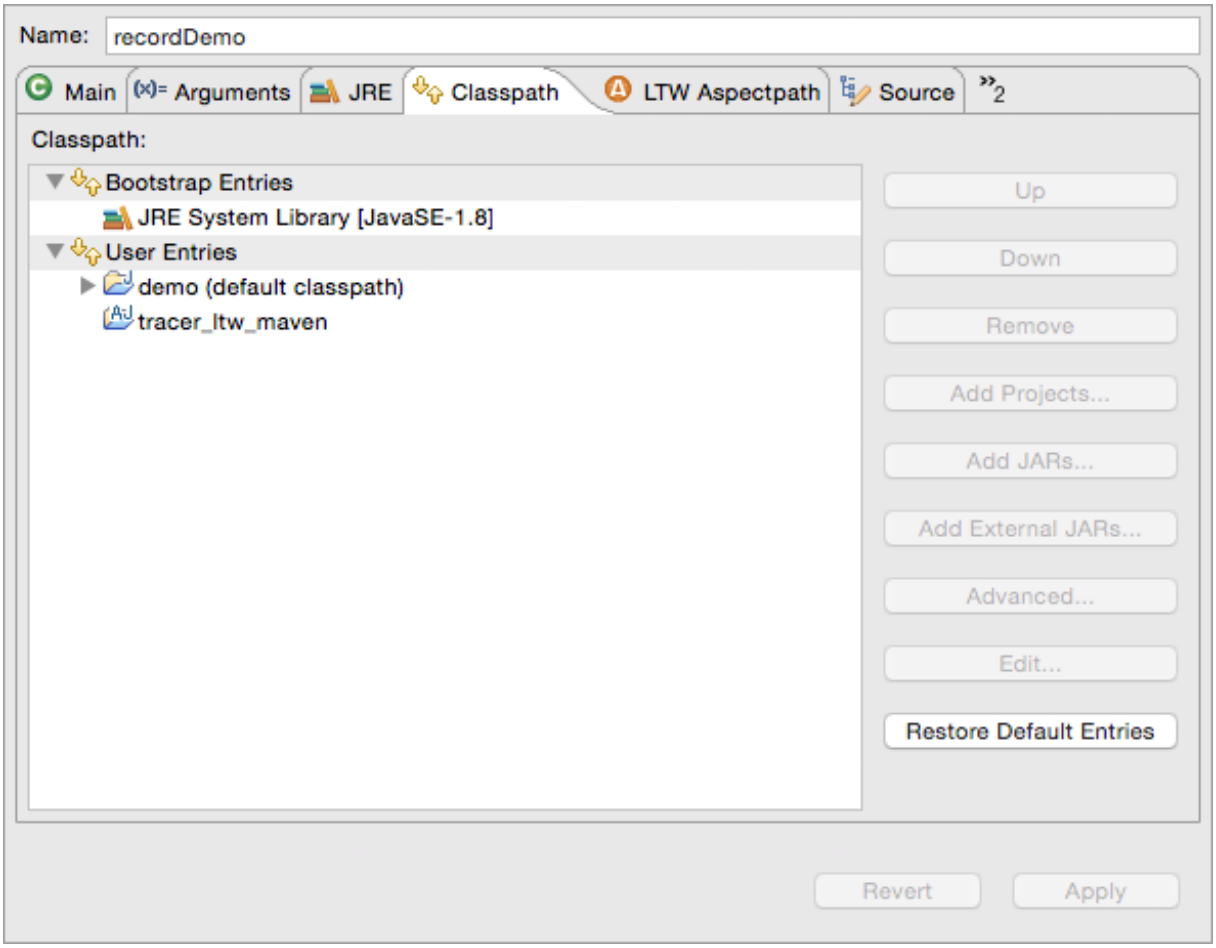
Im Run-Configurations-Menü muss eine neue AspectJ Load-time-Weaving Application erstellt werden.

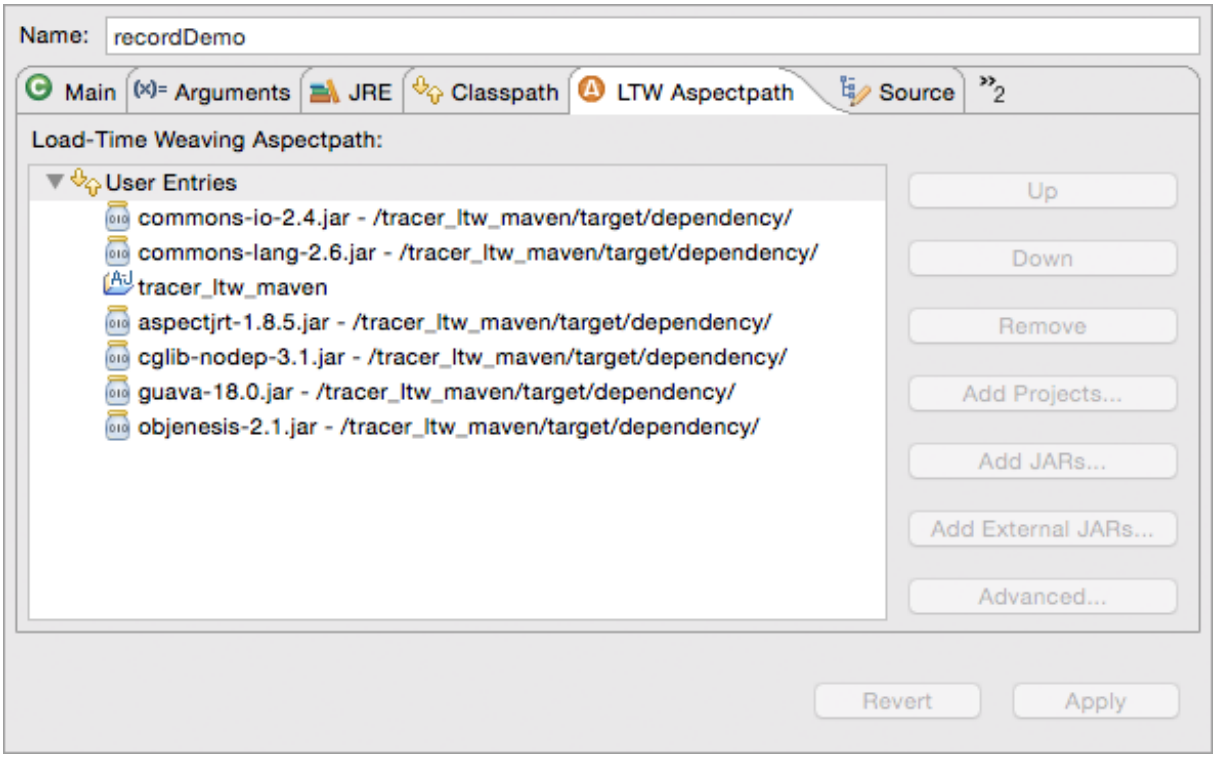


Dabei müssen die verschiedenen Reiter anhand der nachfolgenden Bilder konfiguriert werden. Reiter, welche nicht aufgeführt werden, können mit den Standardeinstellungen belassen werden.

Record

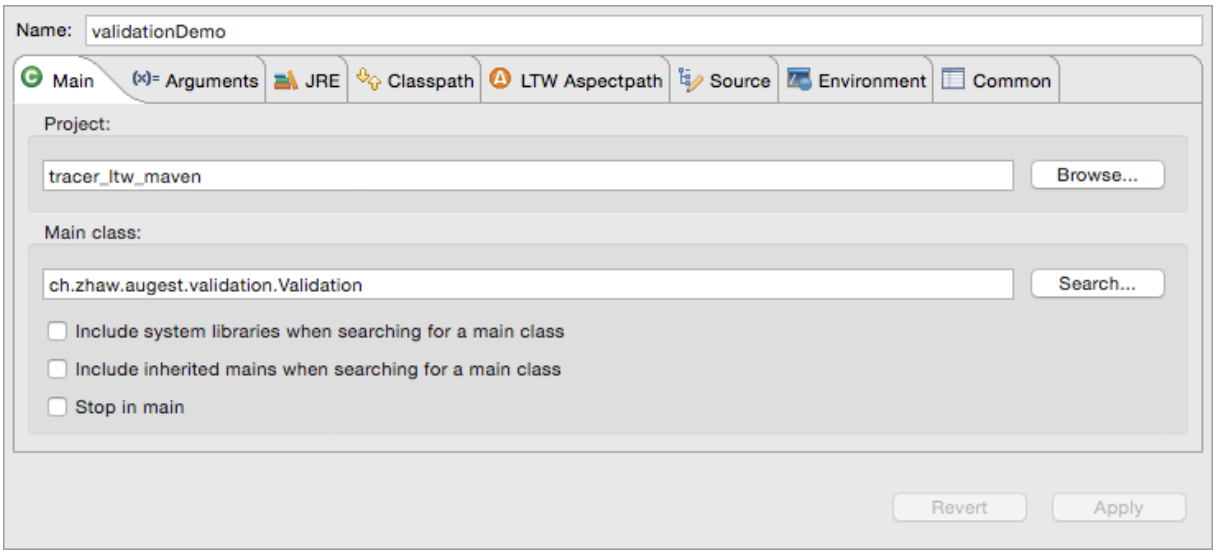


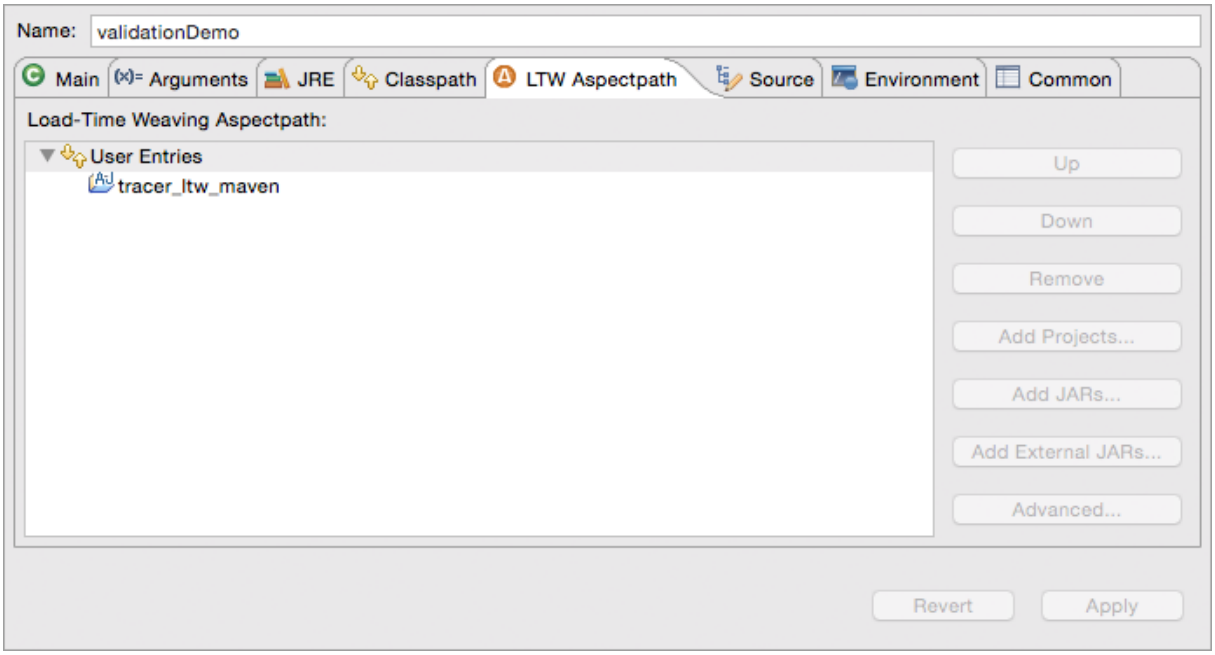




Damit diese jar-Files verfügbar sind, muss das Projekt im Vorfeld per Maven kompiliert worden sein.

Validation



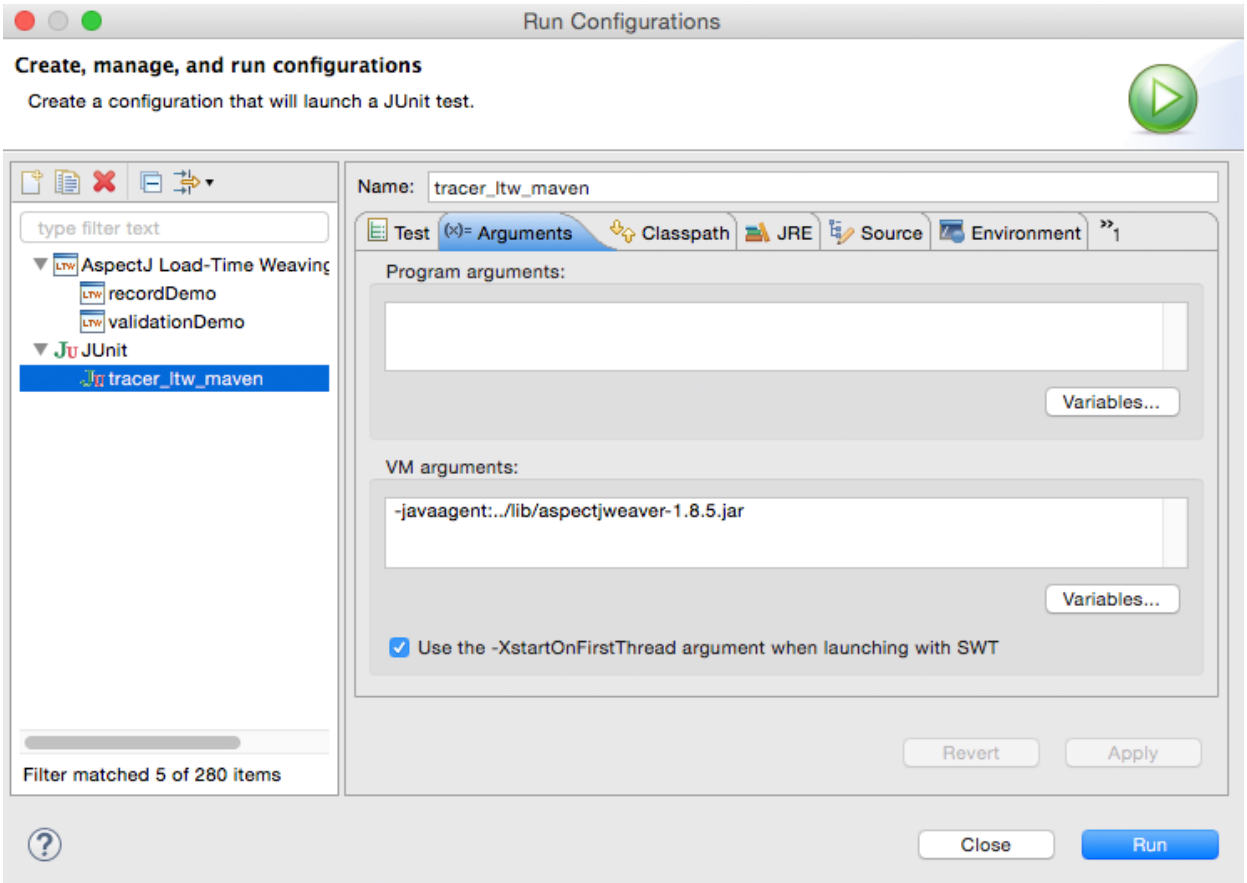


Damit das Projekt als AspectJ-Projekt gestartet werden kann, muss noch der Ordner `src/main/aspect/` als «Source Folder» zum Build-Path hinzugefügt werden (Rechtsklick → Build Path → Use as Source Folder). Ausserdem muss das Projekt «tracer_ltw_maven» als AspectJ-Projekt konfiguriert werden (Rechtsklick → Configure → Convert to AspectJ Project).

Wurden diese Schritte vorgenommen, kann über das Run-Configurations-Menü die Aufzeichnung respektive die Validierung gestartet werden. Bei der Aufzeichnung wird eine Datei `augest_execution_trace<timestamp>.ser` im `demo`-Verzeichnis erzeugt. Damit die Validierung durchgeführt werden kann muss diese Datei ins Verzeichnis `tracer_ltw.maven/` verschoben und zu `augest_execution_trace.ser` umbenannt werden.

JUnit-Tests

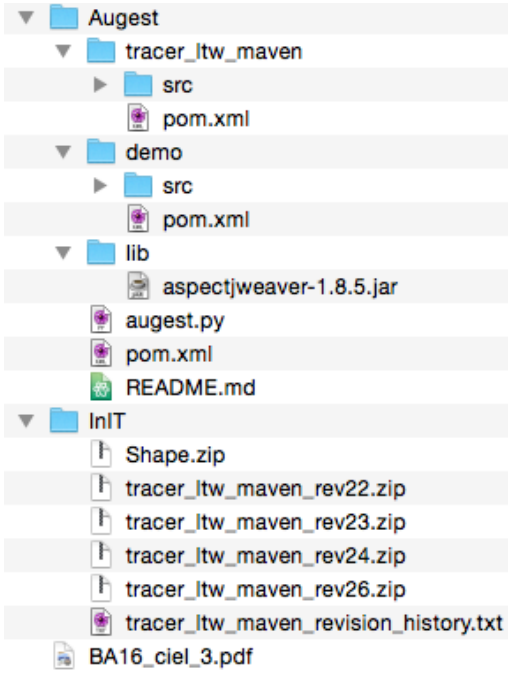
Damit die JUnit-Tests auch innerhalb von Eclipse ausgeführt werden können, muss dafür noch eine entsprechende Run-Configuration erstellt werden. Dafür kann das Projekt tracer_ltw_maven als JUnit-Test gestartet werden. Diese Tests werden fehlschlagen. Als Nebeneffekt wird ein Eintrag in den Run-Configurations namens «tracer_ltw_maven» erstellt. Bei dieser muss im Reiter «Arguments» ein Eintrag erstellt werden, damit die JUnit-Tests mit einem Java-Agent gestartet werden.



Wurden diese Schritte vorgenommen, können die JUnit-Tests in Eclipse mit der Run-Configuration «tracer_ltw_maven» gestartet werden.

6.8 CD

Auf der beiliegenden CD sind sämtliche Unterlagen dieser Arbeit zu finden.



`Augest/`: Das Projektverzeichnis der entwickelten Software.

`Augest/tracer_ltw_maven/`: Sourcecode des Aufzeichnungs-/Validierungstools

`Augest/demo/`: Sourcecode der Beispiel-Software

`Augest/lib/`: Java-Agent, welcher für das Load-Time-Weaving benötigt wird

`InIT/`: Codebasis, welche als Grundlage für diese Bachelorarbeit zur Verfügung stand. Als Ausgangslage wurde Revision 24 verwendet.

`BA16_ciel_3.pdf`: Der vorliegende Bericht der Bachelorarbeit