Hochschule für angewandte Wissenschaften München

Fakultät 07

Informatik und Mathematik

BACHELORARBEIT

Verwaltung von Patch-Files für Experimente mit Mutation Analysis

Patch-File management for experiments with mutation analysis

Autor: **Markus Maximilian Stromer**

Matrikelnr.: **04527011**

Prüfer: **Prof. Dr. Axel Böttcher**

Ort: **München**

Inhalt

1. Einführung

In dieser Arbeit werden Möglichkeiten zur automatisierten Bewertung von Programmieraufgaben erörtert und vorgestellt. Im Vordergrund stehen hierbei die Bewertung der abgegebenen Testklassen. Im Hintergrund zu dieser Arbeit wurde ein entsprechendes Eclipse-Plug-In erstellt, welches eine mögliche Lösung für diese komplexe Aufgabe stellt. Die technische Ausführung unterstützt ausschließlich die Programmiersprache Java bzw. JUnit Testklassen.
Das Entwickelte Plug-In ist unter https://github.com/StromerMarkus/PatchManagement zu finden.

1.1 Bedeutung von Tests in der Softwareentwicklung

In der Industrie werden die hergestellten Produkte nach der Fertigung üblicherweise einer Qualitätskontrolle unterzogen. Bei dieser Kontrolle wird geprüft, ob das Produkt die jeweils gestellten Anforderungen erfüllt. Vergleichbar mit der Industrie kann der Testbegriff auch in der Softwareentwicklung verwendet werden. Problematisch hierbei ist jedoch das bei der Softwareentwicklung in der Regel ein immaterielles Produkt entsteht. Dadurch erschweren sich die üblichen Testmethoden aus der Industrie, so ist z.B. eine optische Prüfung des Produktes, wie es bei einer gefertigten Maschine erfolgt nur eingeschränkt möglich. Softwaretests müssen grundsätzlich auf einem Rechner ausgeführt werden. Dabei wird festgestellt, ob das Verhalten des Softwareproduktes den Anforderungen entspricht. Sollte dies nicht der Fall sein ist dies, nach [DIN 66271] ein *Fehler*. Ein Softwaretest kann, ab einen gewissen Komplexitätsgrad, kein Fehlerfreies System garantieren. Meist werden bestimmte Ausnahmesituationen nicht bedacht oder können evtl. gar nicht überprüft werden. Aus diesem Grund sind Softwaretests auch gesondert zu behandeln und bedürfen einer ausführlichen Analyse, über ihre Qualität.
Wie aus der vorangegangen Erläuterung hervorgeht, werden für komplexe Softwareprojekte üblicherweise ein Entwicklerteam und ein Tester Team benötigt. Damit die Kommunikation bzw. die Synchronisation der beiden Gruppen abgestimmt wird müssen Vorgehensmodelle erstellt und durchgeführt werden. Hier wird ein mögliches Modell, welches in der Praxis häufig vorkommt vorgestellt:

V-Modell

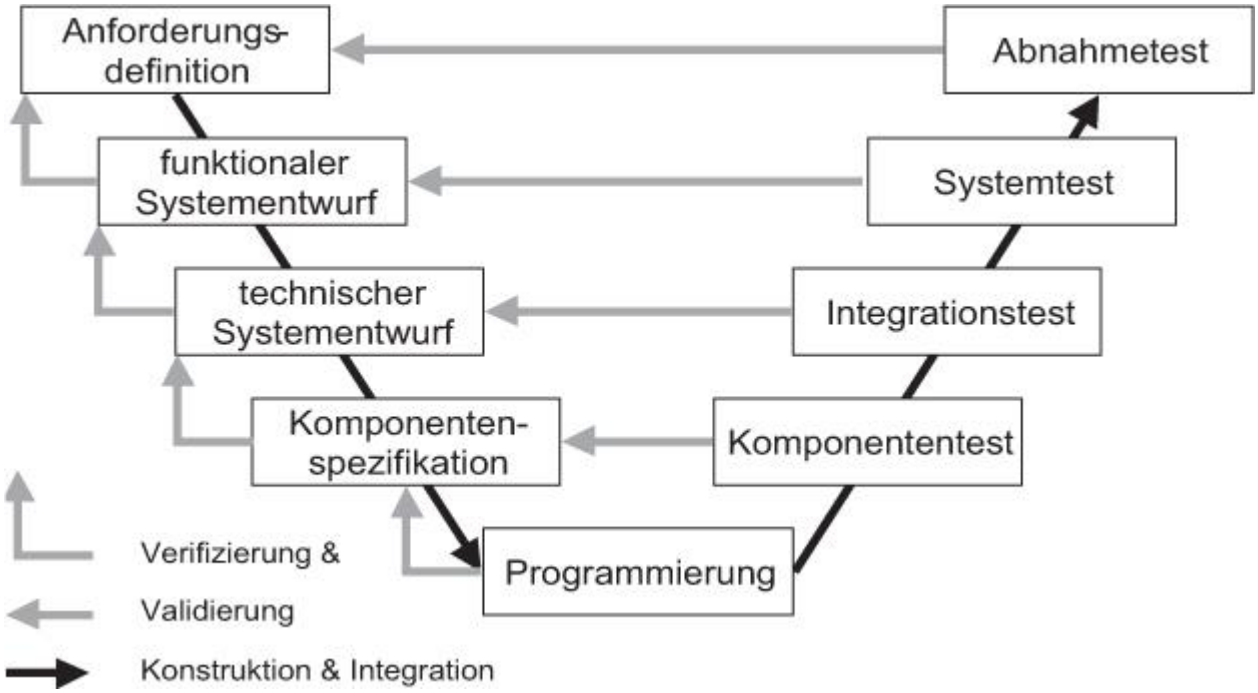


Abbildung 1.1: V-Modell [1]

Hierbei wird die Entwicklungsarbeit der Testarbeit gleichgestellt. Das V-Modell stellt für jede Gruppe jeweils einen Ast dar. Die Teststufen werden hierbei getrennt betrachtet und testen ihre jeweils zugehörige Entwicklerstufe.

Im Zuge der nachfolgenden Testbewertungen werden hier ausschließlich Komponententests betrachtet. Bei den Komponententests werden die zuvor definierten (möglichst kleinen) Softwarebausteine oder hier Komponenten einen systematischen Test unterzogen. Bei diesen Bausteinen kann es sich im Allgemeinen um jede mögliche Zusammenfassung handeln. In diesem Fall beschäftigen wir uns ausschließlich mit Klassentest bzw. Unit Tests. Diese Testklassen sollen eine bestimmte Klasse analysieren und bewerten. Die jeweiligen Komponenten müssen dabei völlig unabhängig gemäß der gegebenen Softwarearchitektur agieren können, also müssen auch die Testklassen voneinander isoliert werden um Testabhängigkeiten zu unterbinden.

1.2 Qualitätsansprüche von Tests

Testklassen oder Testsuiten sollen nach Möglichkeit folgende Kriterien erfüllen:[1]

- Test der Funktionalität
- Test auf Robustheit
- Test der Effizienz
- Test auf Wartbarkeit

In unserem Fall interessieren wir uns in erster Linie die Tests auf Funktionalität. Diese müssen sicherstellen, dass die jeweiligen Testobjekte der geforderten Spezifikation gemäß den Anforderungen vollständig erfüllt werden. Dazu gehört sowohl das interne Verhalten des Testobjekts, als auch die dazugehörigen Ein- und Ausgaben.
Typische Fehler die dabei entdeckt werden sind Berechnungsfehler, bei ungünstig gewählten Eingabeparametern oder falsch gewählte Programmpfade (vor allem bei Heterogenen Systemen). Um den entgegenzuwirken empfiehlt es sich eine hohe Testabdeckung (engl. coverage) zu erzielen. Dies kann man über folgende zwei Verfahren erreichen:

Blackbox-Verfahren:[2]

Beim diesem Verfahren wird der innere Aufbau des Testobjektes nicht betrachtet. Anhand der Spezifikation werden Testfälle erstellt. Um das Objekt vollständig zu testen, müssten Testfälle mit allen möglichen Kombinationen von Eingabedaten erstellt werden, was jedoch unrealistisch ist. Hierbei empfiehlt es sich Extrem- oder Grenzwerte als Eingabe zu verwenden, an denen vielleicht schon Probleme vermutet werden

[1] Vgl. [1] Seite 47 f.
[2] Vgl. [1] Seite 114 ff.

Whitebox-Verfahren:

Hierbei wird der Programmtext des Testobjektes betrachtet und analysiert. Die Testfälle dir hierbei erstellt werden sollen nach Möglichkeit alle Teile des Quellcodes mindestens einmal ausführen. Das Verhalten des Testobjektes muss nach wie vor den geforderten Spezifikationen entsprechen.

Beispiel:

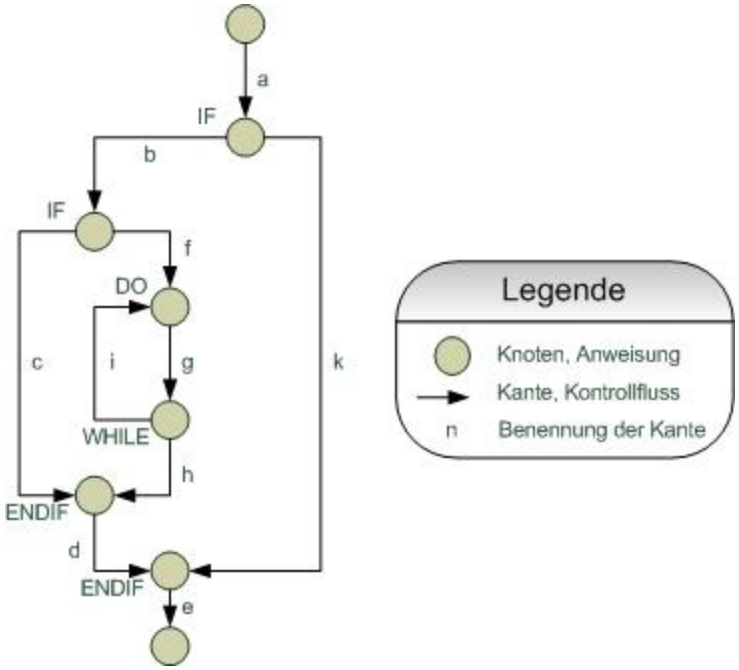


Abbildung 1.2: Beispiel Kontrollflussgraph [3]

In diesem Beispiel reicht ein Testfall zur Abdeckung aller Knoten aus. Hierbei müssen folgende Kanten durchlaufen werden:

a, b, f, g, h, d, e

Es gibt noch weitere Möglichkeiten um alle Knoten abzudecken, jedoch ist Ziel des Whitebox-Verfahren mit möglichst wenig Testfällen eine maximale Abdeckung zu erreichen.

[3] Vgl. [1] Seite 151

2. Methodik/Didaktik

2.2 Testentwicklung

Viele (unerfahrene) Studierende überspringen bei gestellten Aufgaben zunächst das Erstellen von Testklassen. Sofern die gestellte Aufgabe, wie es für die Einführung üblich ist, noch sehr simpel ist, ist dies auch noch nicht problematisch. Bei immer komplexeren Aufgaben/Projekten können sich jedoch die Anforderungen laufend ändern, wie es auch in der freien Wirtschaft durchaus üblich ist. Dies hätte in den Projekte zur Folge, dass die Studierenden ihre Designentscheidungen für die gegebene Aufgabenstellung laufend ändern bzw. anpassen müssen. Dadurch entstehen höhere Wartungsaufwände der erstellten Projekte, welches nicht das Lernziel der Aufgaben darstellt.
Um zu verhindern das die Codequalität mit wachsender Programmgröße abnimmt, wird hier eine Strategie vorgestellt, die helfen soll Studierenden das Testen näher zu bringen.

Test-Driven Development:
Zu Deutsch testgetriebene Entwicklung oder testgetriebenes Programmieren, ist ein iteratives Vorgehen, in denen die Testklassen vor der eigentlichen Implementierung geschrieben werden. Dies mag zu Anfang vielleicht etwas verwirrend erscheinen, da es zu diesem Zeitpunkt noch keine Testobjekte vorhanden sind, die man überprüfen könnte. Dieser Schritt ist jedoch entscheidend, da sich jeder Softwareentwickler (insbesondere die Studierenden) darüber im Klaren sein sollte was überhaupt erforderlich ist.

Der Ablauf lässt sich als Zusammenspiel von zwei Zuständen beschreiben:

- JUnit Failure: erwarteter Fehler
- JUnit OK: erwarteter Erfolg

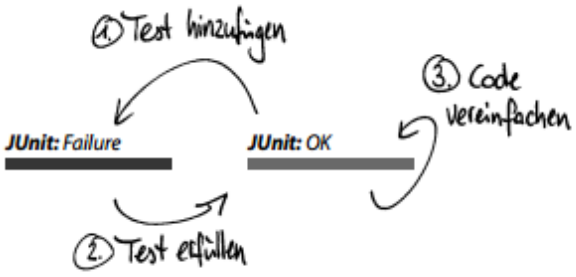


Abbildung 2.1: Vereinfachtes Zustandsdiagramm testgetriebene Entwicklung [4]

[4] Vgl. [2] Seite 52

Die Zustandsübergänge lassen sich über den JUnit Testbalken realisieren. Die drei Schritte der Testgetriebenen Entwicklung:[5]

1. Wir beginnen damit zu einer gegebenen Anforderung einen Test der Entwicklung voranstellen. Dies führt in erster Linie dazu das der Test fehlschlägt, also ein roter Balken bei der Testausführung erscheint. Der Test sollte nach Möglichkeit die entsprechende Anforderung komplett erfassen und abdecken.

2. Der Test wird schrittweise entsprechend der zu Punkt 1 gestellten Anforderungen erfüllt. Meist wird hierbei die entsprechende Methode die der Test prüfen soll implementiert. Hierbei werden vorerst hauptsächlich die funktionalen Aspekte berücksichtigt. Ziel ist ein Erfolg des JUnit Tests.

3. Nach erfolgreichen Test muss der Code auf eine möglichst einfache Form gebracht werden. Es können z.B. unnötige Schleifen oder komplizierte Ausdrücke zusammengefasst oder unter Umständen entfernt werden. Nach dieser Vereinfachung muss erneut geprüft werden ob der Test nach wie vor erfolgreich abgeschlossen wird.

Diese Möglichkeit erleichtert es den Studierenden komplexe Aufgaben durch Teilung von großen Problemen in lösbare kleine Probleme. Auf diese Weise wird zudem noch die Motivation der Studierenden über jeden kleinen Erfolg gefördert, woran sie sich immer weiter vorarbeiten können

[5] Vgl. [2] Seite 52

2.3 Bewertungsmechanismen

Praktikumsaufgaben für Studierende lassen sich meist in zwei Hauptaufgaben aufteilen.

- Implementierung eines bestimmten Interface oder einer allgemeinen Aufgabe
- Entsprechende Testklassen zu den Implementierungen entwickeln

Zur Bewertung der Implementierungen liefert der Dozent in der Regel fertige Testklassen, die die vorgegebenen Anforderungen an den abgegebenen Lösungen überprüfen. Dies kann automatisiert z.B. mithilfe eines Jenkins-Webservers erfolgen. Die abgegebenen Lösungen müssen dafür die Methoden bereitstellen, die die Testklassen erfordern. Hierfür sollten in der Aufgabenstellung Interfaces vorgegeben werden, die von den Studierenden verwenden werden müssen, um Kompilierfehler zu vermeiden.
Für die abgegebenen Testklassen ist dieser Mechanismus nicht möglich. Wie schon in Kapitel 1.1 erläutert kann man für einen Test nie garantieren, dass dieser absolut zuverlässig ist. Um eine möglichst genaue Aussage über die Qualität der abgegebenen Tests zu treffen, müssen diese auf einige entsprechende (kompilierbare) Implementierungen laufen. Nachfolgend werden zwei mögliche Ansätze vorgestellt.

2.3.1 Studententests gegen anderen Studentenlösungen

Dieser Ansatz bewertet zugleich die Implementierungen und Testklassen der Studierenden. Ähnlich wie zuvor erläutert müssen von allen Studierenden bestimmte Interfaces bei der Abgabe eingehalten werden. Das Hauptproblem hierbei ist, dass die Studierenden jeweils unterschiedliche Strukturen innerhalb ihrer jeweiligen Testklassen haben. Dies kann bei der Ausführung zu Compilerfehlern führen, da z.B. eigene Hilfsmethoden getestet werden, die in anderen Lösungen nicht vorkommen. Um den entgegenzuwirken wird hier eine Transformation der Java-Reflektion verwendet, damit die Testklassen jeweils kompilierbar bleiben, sofern diese bereits mit einer bestimmten Lösung (z.B. der jeweils eigenen) kompilierbar waren.

1. *Compiler JUnit tests against the solution provided alongside them (i.e., written by the same author) when they are first received (an action that existing automated grading systems already perform).*
2. *Transform the bytecode in the .class files from the compiled version of a test set so that it performs all manipulation of the solution's class(es) using reflection. In other words, use a tool to automatically convert the "plain" JUnit tests into purely reflective tests.*
3. *Run transformed versions of test sets against other solutions as needed, now that the test sets have no compile-time dependencies of the software to be tested.*[6]

Bei dieser Vorgehensweise können einzelne Testfälle drei verschiedene Zustände/Ergebnisse liefern:

- Der Test ist erfolgreich
- Der Test schlägt fehl, da bei der Transformation gewisse Abhängigkeiten innerhalb der Testklasse fehlerhaft sind
- Der Test schlägt aus unbestimmten Grund, also fehlerhafter Implementierung oder falsch entwickelten Testfall fehl

Um die letzten beiden Fehlschläge voneinander unterscheiden zu können, lässt man vorher die Transformierten Testklassen gegen die geprüfte Musterlösung des Dozenten laufen. Sind diese Tests erfolgreich, so können diese gegen andere Lösungen laufen. Bei Fehlschlag haben die Studierenden für den Testfall interne Abhängigkeiten verwendet welche für diesen Mechanismus ungeeignet sind.

Die Ergebnisse der Testläufe kann man sich anschließend mit einer Matrix darstellen lassen.

	Impl. Muster	Impl. 001	Impl. 002	Impl. 003	Impl. 004
Test Muster					
Test 001					
Test 002					
Test 003					
Test 004					

Tabelle 2.2: Beispielergebnis

[6] Vgl. [3] Seite 224

Der Einfachheit halber gehen wir davon aus das alle Testläufe zueinander kompilierbar waren. Die Spalten geben hier die Implementierungen und die Zeilen die jeweiligen Testklassen zu der Aufgabe. Bei dieser Darstellung werden alle Testfälle zusammengefasst.
Um hier jetzt zu erkennen ob die Implementierung oder der zugehörige Test fehlerhaft ist wird hier die Musterlösung des Dozenten in der jeweils ersten Zeile und Spalte als Referenz hergenommen. Hierbei ist zu erkennen, dass die Abgabe des Studierenden(oder Gruppe) 002 von den Dozenten als fehlerhaft markiert wird. Demzufolge sollten alle Studententests bei der Implementierung von 002 ebenfalls einen Fehler erkennen. In diesem Fall jedoch erkennen die Testklassen von 003 diesen Fehler nicht was vermutlich zu geringe Testabdeckung zu Grunde liegt. Auch liegt hier der Fall bei 004 vor der korrekte Lösungen als Fehler deklariert.
Zusammengefasst lässt sich das Ergebnis dieser Abgaben wie folgt ausdrücken:

- 001: Implementierung OK, Test OK
- 002: Implementierung FAIL, Test FAIL
- 003: Implementierung OK , Test OK
- 004: Implementierung OK, Test FAIL

Da wir bei üblichen Praktikumsgruppen eine weitaus höhere Abgabezahl haben wächst diese Matrix natürlich entsprechend stark an, was die manuelle Analyse sehr erschwert.

2.3.2 Mutation Analysis

Mutation Analysis stellt hierbei den Schwerpunkt dieser Arbeit dar worauf sich auch die spätere Implementierung bezieht, weshalb es hier einer detaillierten Erklärung benötigt:

Einführung Mutation Analysis:
"Mutation testing is a powerful, but computationally expensive, technique for unit testing software. This expense has prevented mutation from becoming widely used in practical situations, but recent engineering advances have given us techniques and algorithms for significantly reducing the cost of mutation testing."[7]

[7] Vgl. [4] Einführung

Bei der Mutation Analysis oder Mutation Testing wird die Qualität der Testklassen mithilfe von modifiziertem Code untersucht. Durch diese Modifikationen entstehen sogenannte Mutanten, welche von den jeweiligen Testfällen/Testklassen erkannt werden müssen.

Historische Entwicklung:

1971 wurde diese Technik erstmals von dem damaligen Studenten Richard Lipton vorgestellt. 1978 wurde diese Arbeit weiter ausgearbeitet und erstmals von DeMillo, Lipton und Sayward veröffentlicht. Im Rahmen seiner Doktorarbeit entwickelte Timothy Budd erstmals ein Tool für Mutation Testing an der Yale-Universität. Mit wachsender Computerleistung konnte auch diese Technik weiterentwickelt werden und etablierte sich mit der Zeit in mehreren objektorientierten Programmiersprachen.[8]

Vorgehensweise:

Bei der Analyse werden viele Fehler erzeugt und dabei mehrere Versionen, der zu testenden Software erstellt. Jede dieser verschiedenen Versionen nennt man *Mutanten (engl. mutants).* Jeder dieser Mutanten sollte *einen* Fehler enthalten, der erkannt werden muss. Wird dieser Mutant erkannt so kann dieser entfernt (engl. *killed*) werden.

Beispiel:

Mutation mit folgendem Java-Code:

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

In dieser Mutation wird jetzt die AND Verknüpfung (&&) mit einem OR (||) ausgetauscht:

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

[8] Vgl. [4] Kapitel 1

Um diese Mutation zu erkennen müssen folgende Bedingungen erfüllt werden:

- Der Test muss den Mutierten Ausdruck erfassen
- Der Test muss eine Abweichung von den Mutierten Ausdruck zu dem Original erzwingen. In diesem Fall ist dies möglich wenn z.B. a = true und b = false gesetzt werden
- Der inkorrekte Zustand oder die Ausgabe (hier das c) muss von dem Test überprüft und abgefangen werden

Weitere Bespiele für Mutationsoperationen:

- Ausdrücke/Variablen entfernen
- Boolean Ausdrücke umkehren
- Arithmetische Ausdrücke austauschen (z.B. + mit *)
- Relationen austauschen (z.B. < mit <=)

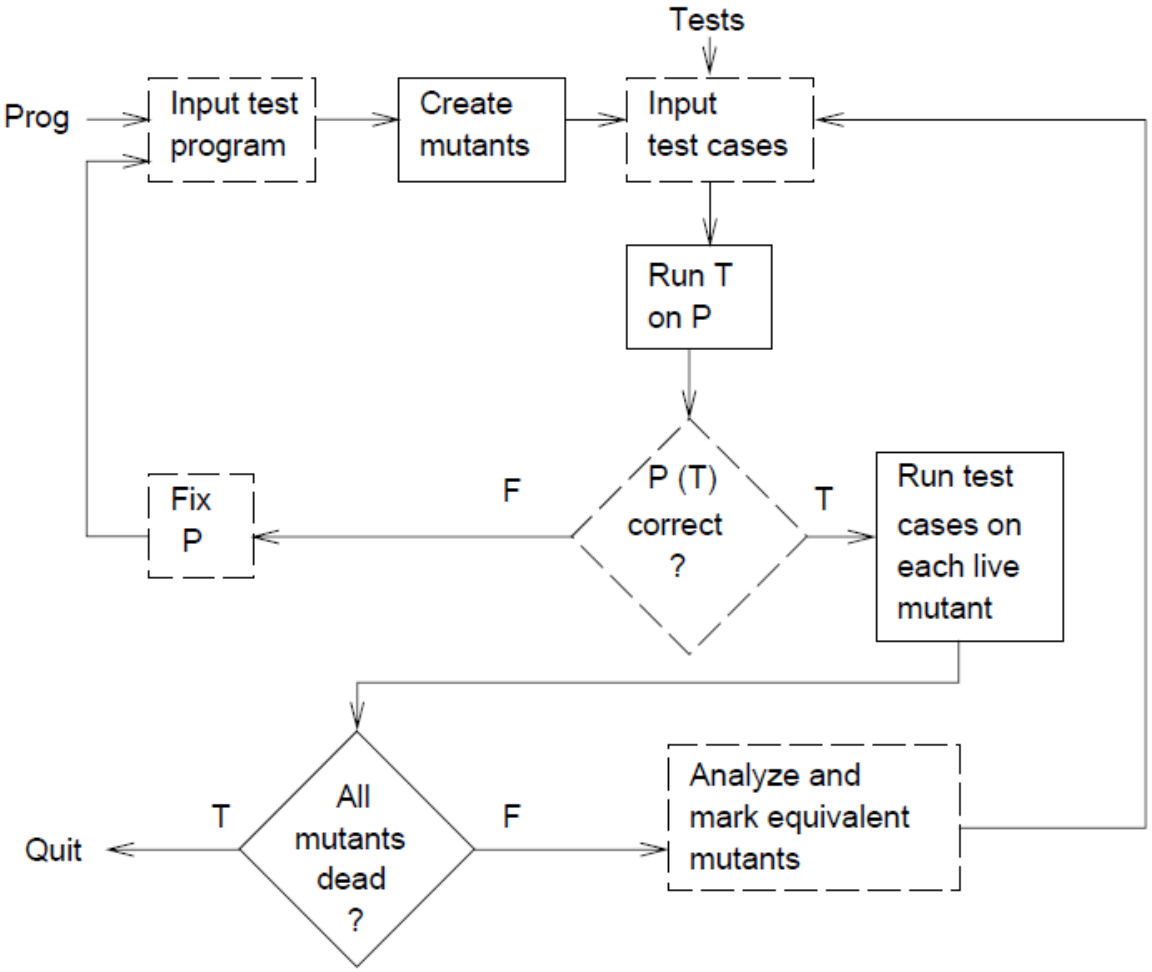


Abbildung 2.3: Mutation-Testing Prozess[9]

[9] Vgl. [4] Figure 1

2.4 Bewertungsablauf

Jetzt, da wir uns mit den Grundlagen befasst haben gehen wir konkret auf das Bewertungssystem bzw. dessen Ablauf ein.

Ziel des Systems:

Eine möglichst automatisierte Bewertung von abgegebenen Lösungen der Studierenden. Dabei sind sowohl die Implementierungen der Aufgabenstellung als auch die dazugehörigen Test inbegriffen. Dazu soll ein entsprechendes Feedbacksystem entstehen, welches den Studierenden ihren aktuellen Leistungsstand präsentiert.

Ablauf der Abgabe:

Um eine Bewertung zu ermöglichen ist es erforderlich eine Plattform zu schaffen, an denen Studierende ihre Lösungen abgeben können. Hierbei muss auch festgelegt werden welche Abgaben akzeptiert werden. Dies kann man z.B. über einen Jenkins-Webserver mit verschiedenen Policies regeln. Hier sind mögliche Abgabekriterien aufgelistet:

- Abgabe muss kompilierbar sein
- Die Unit-Tests des Dozierenden laufen fehlerfrei
- Checkstyle meldet keine Fehler
- PMD, Findbugs oder ähnliche Tools für Performancetest erfolgreich
- Geforderte Struktur (Packages oder bestimmte Abhängigkeiten) ist gegeben

WICHTIG: Abgabekriterien haben in erster Linie nichts mit den allgemeinen Bewertungskriterien zu tun. Sie dienen lediglich der Filterung von Abgaben.
Die Wahl der Abgabekriterien ist abhängig von den Motivationsgedanken für die Studierenden oder auch von der Art der Bewertung. Die Bewertungsplattform soll direkt nach Abgabe die festgelegten Abgabekriterien prüfen. Sind alle Bedingungen erfüllt so bekommt der Studierende direkt eine Rückmeldung, dass seine Lösung eingegangen ist und bewertet wird. Sollte eine Bedingung nicht erfüllt sein sollte eine möglichst detaillierte Fehlerbeschreibung erscheinen, sodass die Abgabe von den jeweiligen Studierenden überprüft bzw. überarbeitet werden können.

Um den Schwierigkeitsgrad bei der Abgabe optimal einstellen zu können sehen wir uns folgende schematische Darstellung an.

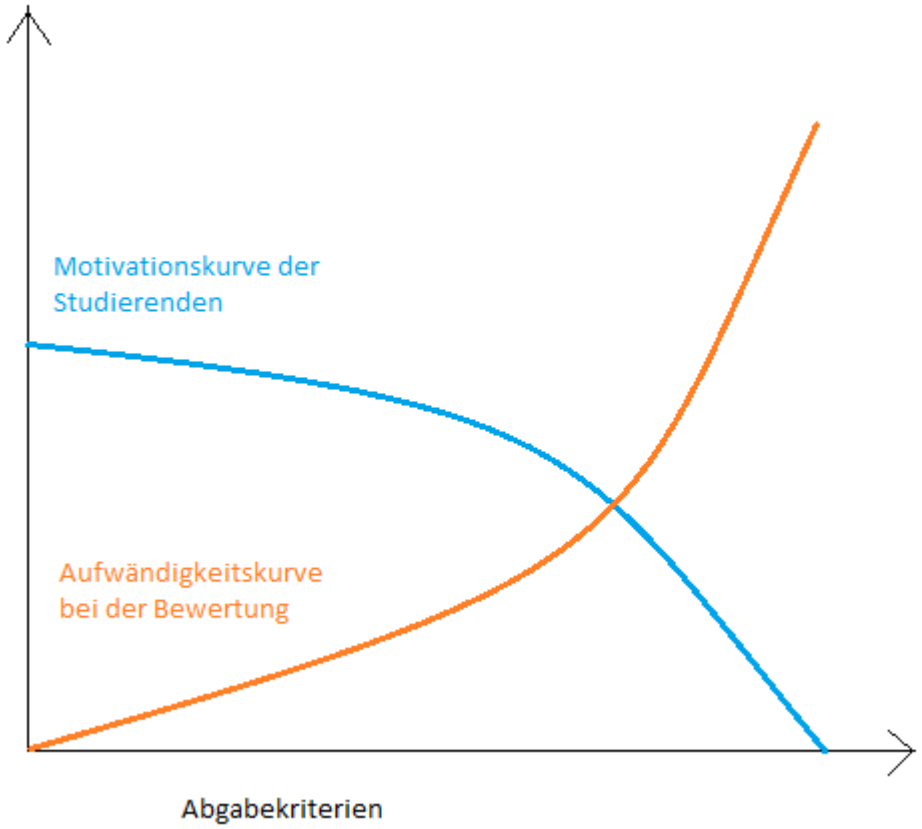


Abbildung 2.4: Zusammenhang Motivation und Bewertungsaufwand

Bei den kompletten Bewertungsvorgang müssen zwei Dinge berücksichtigt werden:

- Motivation der Studierenden
- Aufwand bei der evtl. manuellen Nachbearbeitung

Ein Student ist zunächst glücklich, wenn er seine Aufgabe abgeschlossen und abgegeben hat. Eine einfache Hürde für die Abgabe zu schaffen bedeutet folglich einen Motivationsschub für die Studierenden. Um jedoch eine Angemessene Beurteilung der Arbeit zu geben müssen verschiedene Aspekte dieser untersucht werden. Wenn in diesem Fall z.B. eine Lösung abgegeben wurde, die nicht kompilierbar ist, so ist eine automatisierte Bewertung nicht möglich (außer man bewertet die Abgabe direkt als durchgefallen bzw. Note 5.0). Um den vorzubeugen werden Hürden eingebaut um einen automatisierten Bewertungsmechanismus gewährleisten zu können. Hält man die Abgabekriterien allerdings zu strikt, kann dies dazu führen, dass Studierende den Kurs vorzeitig abbrechen, weil sie die Aufgabe evtl. für nicht lösbar halten.

Dazu ist es nötig ein ausgewogenes Verhältnis für Abgabekriterien zu schaffen, die die Motivationskurve der Studierenden möglichst hoch hält.

Erfahrungen aus dem Praktikum Softwareentwicklung 2 aus dem Sommersemester 2015 ergeben folgende Erkenntnisse:

- Abgaben sollten kompilierbar sein
- Eine vorher definierte Struktur sollte eingehalten werden (Package Hierarchien)
- Die Abgabe sollte gegen Grundtests des Dozierenden kompilierbar sein
- Sollten die Grundtests fehlschlagen jedoch kompilieren wird die Lösung akzeptiert jedoch ein Hinweis auf Fehler zurückgeben
- Die hinterlegten Testklassen sollten bei Fehlschlag eine aussagekräftige Fehlermeldung liefern
- Sollte es bei der Abgabeplattform zu technischen Schwierigkeiten kommen, muss es eine alternative Abgabemöglichkeit gegeben sein (z.B. e-mail)

Ablauf der Bewertung:

Die nun abgegebenen Lösungen müssen nun nach vorher definierten Kriterien bewertet werden. Mögliche Bewertungskriterien hierbei sind:

- Grundtests erfolgreich
- Checkstyle ohne Fehler
- Erweiterte Tests erfolgreich
- Dokumentation (Javadoc) vorhanden und aussagekräftig
- Abgegebene Testklassen erkennen zuverlässig Fehler

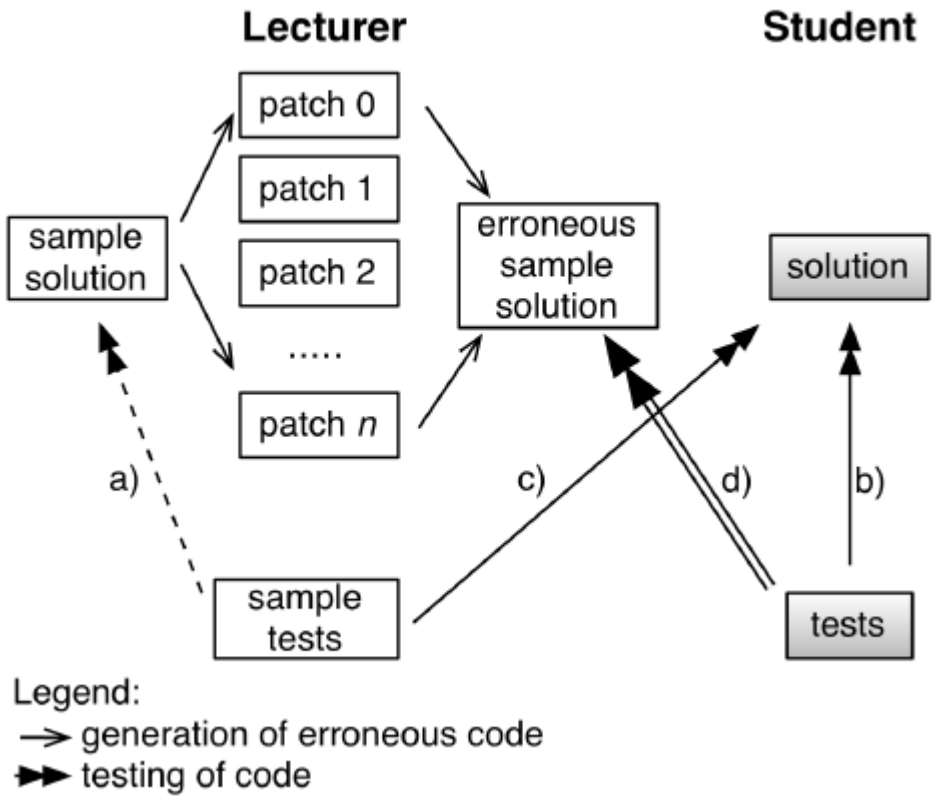


Abbildung 2.5: Prozess zur Bewertung von Tests und Produktivcode[10]

Der Ablauf zeigt das Zusammenwirken zwischen Testbewertung und Bewertung der Implementierungen. Hierbei werden auch die vier verschiedenen Testkombinationen aufgezeigt:

A) Der Dozent hält zu der Aufgabe eine eigene Musterlösung und zugehörige Testklassen bereit. Beide sind nur zulässig wenn alle Tests erfolgreich sind.

B) Studierende prüfen mit ihren Testklassen ihre eigene Lösung. Diese Tests sollten vor der Abgabe erfolgreich sein und entsprechend dokumentiert sein

C) Der Dozent prüft die abgegebenen Lösungen. Für diesen Vorgang gibt es bereits eine Vielzahl von Automatisierungstools, die diese Kontrolle ermöglichen

D) Hier werden die Testklassen der Studierenden bewertet. Dabei wird die Musterlösung des Dozierenden verwendet auf denen mit Hilfe von verschiedenen Patches Mutanten generiert werden. Die abgegebenen Testklassen prüfen diese mutierten Musterlösungen und sollten hierbei immer einen Fehler erkennen

[10] Vgl. [5] Figure 2

3. Patch-Management

Um nun bei der Bewertung der Testklassen zu unterstützen wurde im Rahmen dieser Arbeit ein Eclipse-Plug-In erstellt, welches eine einfache Erstellung von Patch-Files ermöglicht. Diese Patches werden benötigt um Mutanten der Musterlösung zu generieren welche die Testklassen überprüfen können. Das Plug-In wurde für die Java Entwicklung im Kurs Softwareentwicklung 1+2 der Hochschule München angepasst, kann aber bei Bedarf entsprechend erweitert werden.

3.1 Anforderungen

Bevor wir uns mit den Einzelheiten des Plug-Ins befassen muss zuerst geklärt werden was wir überhaupt benötigen. Dazu wurde ein Anwendungsfalldiagramm erstellt welches zunächst über den kompletten Bewertungsmechanismus läuft

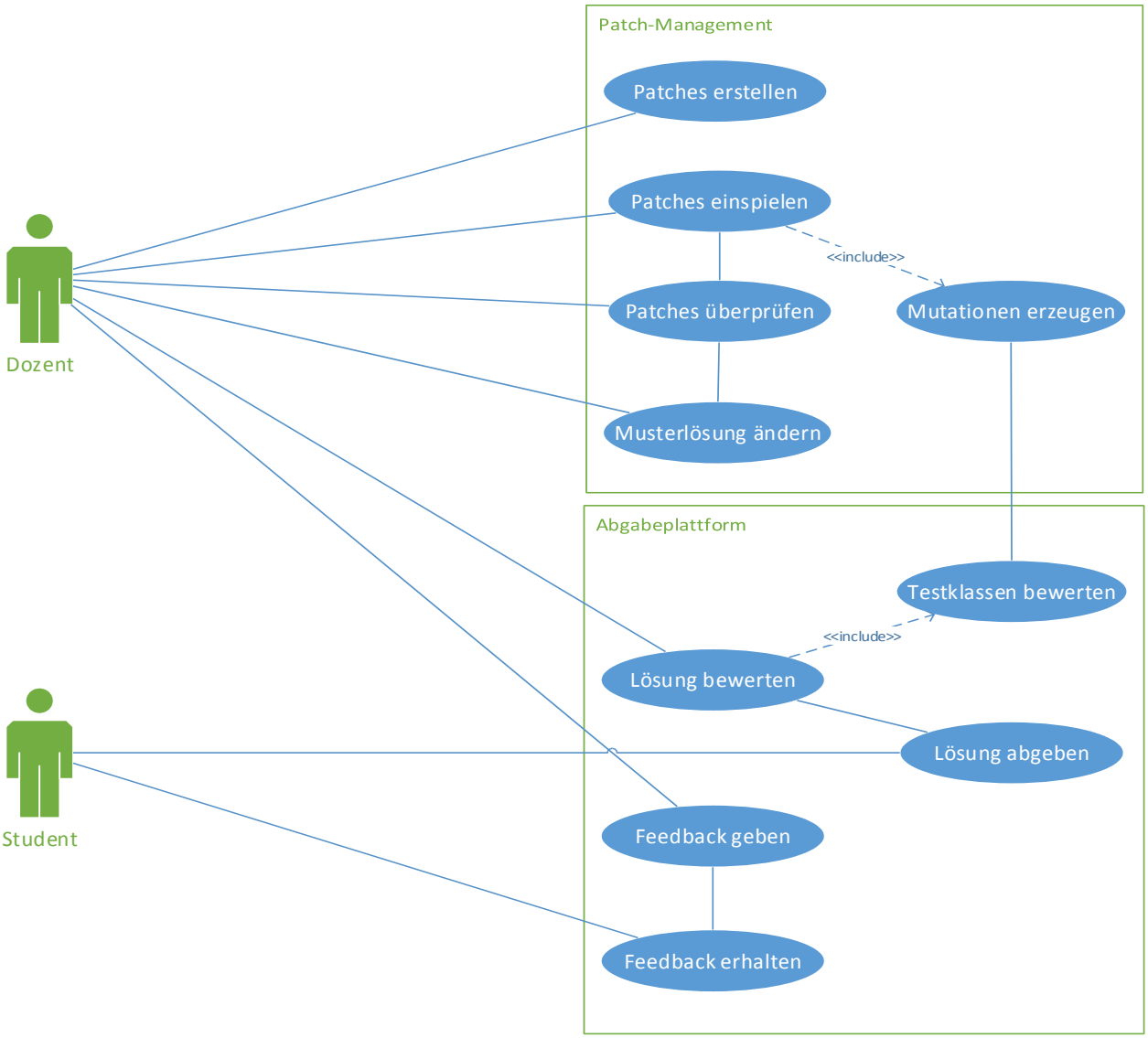


Abbildung 3.1: Anwendungsfalldiagramm

Zu beachten sind hier die Verschränkungen zwischen der Patch-Verwaltung und dem Bewertungssystem. Die Kapselung zwischen der Patch-Verwaltung und der Abgabeplattform ist von entscheidender Bedeutung, da die Studierenden auf die Verwaltung keinen Einfluss nehmen dürfen. Daraus ergeben sich für das Patch-Management-Tool folgende Anforderungen:

- Einfache Generierung von Patch-Files über das (manuelle) Erzeugen von Fehlern in der Musterlösung
- Erstellen von Mutanten zur Qualitätskontrolle der Testklassen
- Möglichkeit zur Prüfung ob vorhandene Patch-Files zur gegebenen Musterlösung passen
- Evtl. fehlerhafte Musterlösung korrigieren und anschließende Prüfung der Patch-Files

Im Optimalfall könnten die Patch-Files automatisch angepasst werden sofern sich die Musterlösung ändert. Jedoch wird hiervon aufgrund einer zu hohen Komplexität abgesehen.

Auswahl der technischen Mittel

Um eine entsprechende Benutzerfreundlichkeit zu schaffen, sind die gewählten technischen Mittel von entscheidender Bedeutung.

Die Wahl der verwendeten Programmiersprache fällt dabei naheliegen auf Java, da die verwendeten Sourcen und Testklassen auch alle in Java verfasst wurden.

An der Hochschule München wird hauptsächlich die Entwicklungsumgebung Eclipse gelehrt, daher arbeiten viele Studierende und Dozenten mit dieser Umgebung. Um die Benutzerfreundlichkeit so einfach wie möglich zu halten, wurde daher ein Eclipse-Plug-In für das Patch-Management festgelegt. Dies hat den Vorteil, dass direkt in der Eclipse-Umgebung damit gearbeitet werden kann. Die erstellten Ergebnisse oder Mutanten müssen entsprechend von einer anderen Toolchain weiterverarbeitet werden

3.2 Grundsätzlicher Ablauf

Wie schon in den oben genannten Anforderungen erläutert gibt es innerhalb des Plug-Ins unterschiedliche Abläufe, die unabhängig voneinander ausgeführt werden können:

- Erstellen von Patch-Files
- Sicherung der Musterlösung
- Änderung der Musterlösung
- Einspielen von Patch-Files

Erstellen der Patch-Files

Befassen wir uns zunächst mit der Erstellung der Patches, welche auch mit der Sicherung zusammenhängt. Zur Veranschaulichung hilft das folgende Zustandsdiagramm:

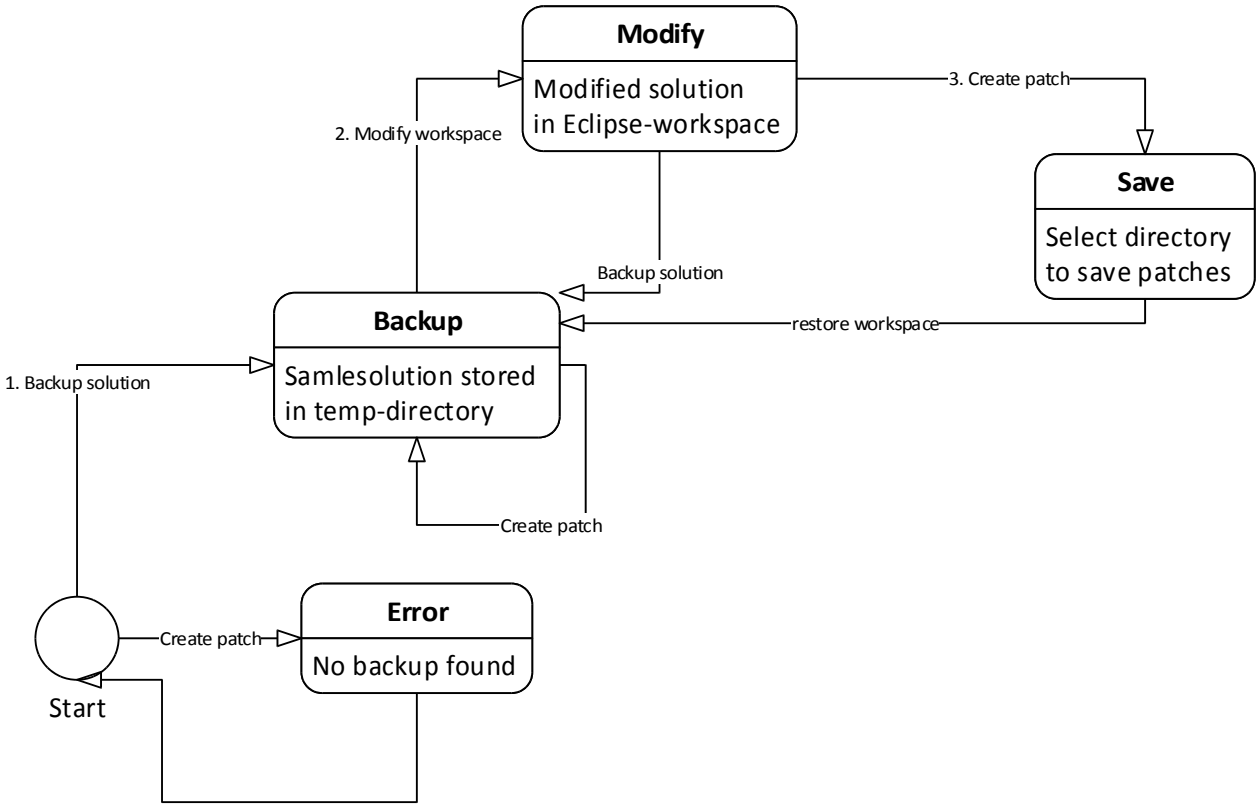


Abbildung 3.2: Zustandsdiagramm bei Patcherstellung

Die Patcherstellung lässt sich in drei Schritten realisieren:

1. Sicherung (backup) des Projekts. Hierbei wird das ausgewählte Projekt in den systemeigenen temporären Ordner (festgelegt durch die Umgebungsvariable TEMP) kopiert. Dadurch werden zwei vergleichbare Projekte geschaffen, ohne zusätzliche Erstellung eines Vergleich Projektes.
2. Änderung des Programmcodes. Oder anders ausgedrückt, hier wird der Fehler manuell eingebaut. Der Code kann an beliebig vielen Stellen und an beliebig vielen Klassen geändert werden.
3. Erstellen der Patches. Hier wird das Projekt aus dem Workspace mit dem gesicherten Projekt auf dem temporären Verzeichnis verglichen. Werden Unterschiede erkannt, so kann der Benutzer auswählen an welche Stelle er die erstellten Patches (.diff Files) gespeichert haben will. Der Ordnername sollte dabei möglichst aussagekräftig sein, damit später erkannt wird, um welche Art von Änderung/Fehler es sich dabei handelt

Nach der Erstellung und Sicherung der Patches wird der Workspace automatisch wieder auf den Stand des Backups zurückgesetzt.
Die übrig gebliebenen Zustandsübergänge dienen der Fehlerbehandlung und sind für den eigentlichen Vorgang nicht relevant

Erstellen der Mutanten

Nachdem wir nun unsere Patches erstellt haben, wollen wir nun unsere Mutanten erstellen. Darüber hinaus wird eine Möglichkeit zur Prüfung der Patch-Files benötigt, für den Fall, dass an der Musterlösung sich etwas geändert hat.

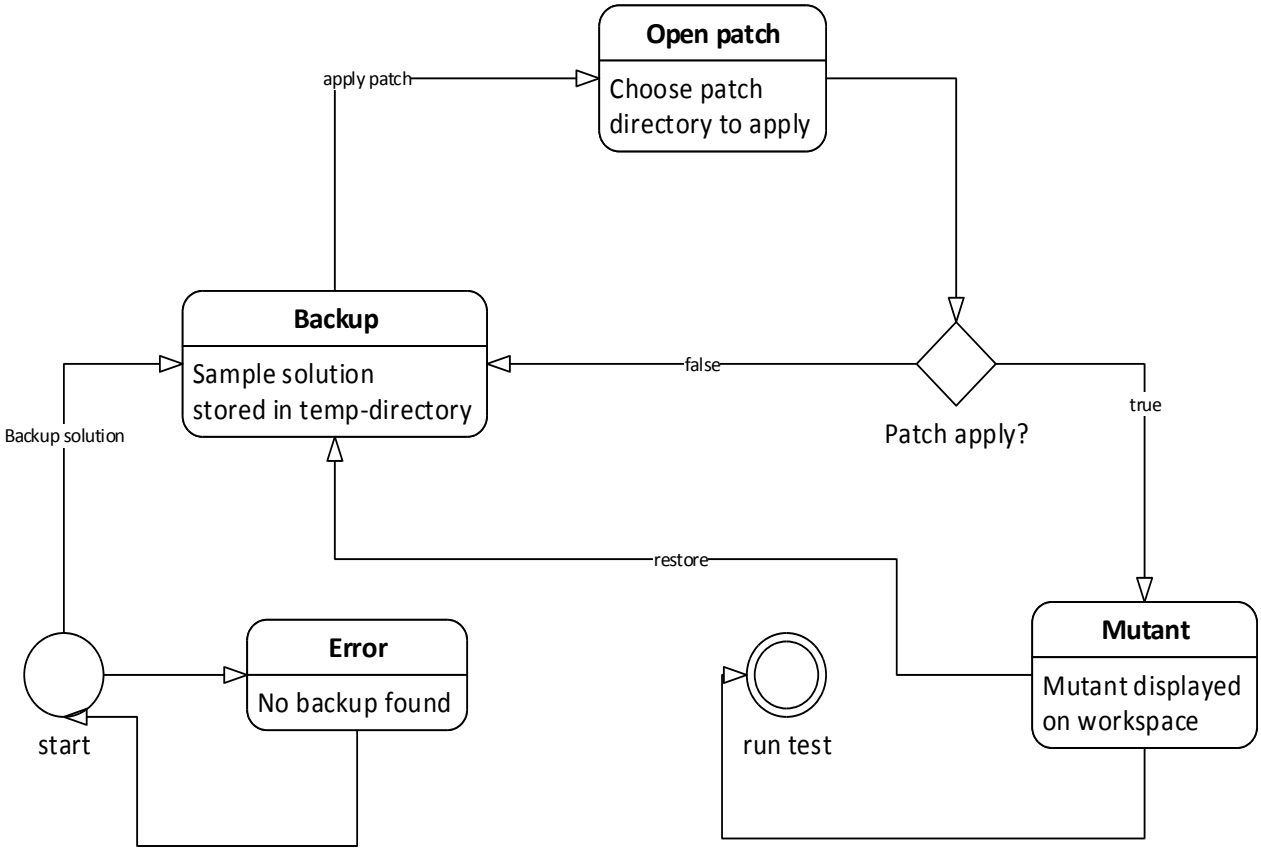


Abbildung 3.3: Zustandsdiagramm Erstellung von Mutanten

Um notfalls auf den gesicherten Stand zurückfallen zu können muss vorher wieder das Projekt gesichert werden. Die Patches die eingespielt werden sollen, müssen über ein Filedialog ausgewählt werden. Nach Auswahl der Patches wird zunächst geprüft ob alle vorhandenen Änderungen eingepflegt werden können. Sollte dabei ein Fehler auftreten wird der alte Stand wiederhergestellt. Bei erfolgreicher Überprüfung wird die alte Lösung im Workspace durch den Mutanten ersetzt. Weitere Schritte sind von der Verarbeitung des Benutzers abhängig.

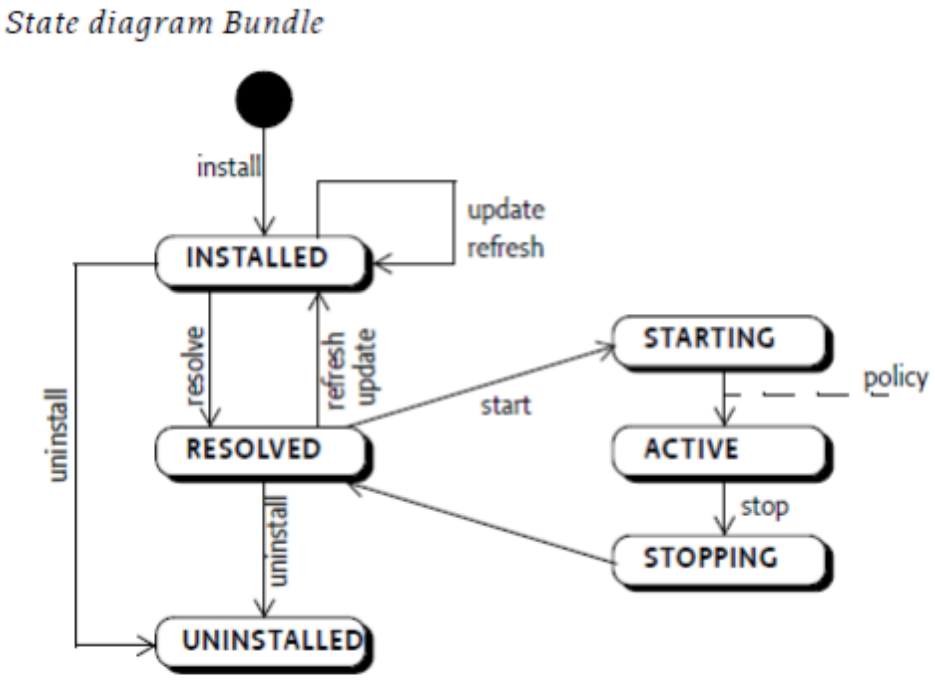
Beim Beenden der Eclipse-Umgebung wird automatisch der gesicherte Stand von dem temporären Verzeichnis zurück in den Workspace geschrieben

3.3 Architektur

Für den Architekturentwurf müssen wir uns hierbei der Plug-In API von Eclipse halten.

3.3.1 Plug-In-Architektur

In der Eclipse-Installation ist ein Plug-In-Ordner vorhanden, in welche die installierten Plug-Ins bereitgestellt werden. Jedes Plug-In erhält eine eigene Ordnerstruktur. Das Herzstück jedes Plug-Ins ist das XML Manifest File, welches plugin.xml heißt. Dieses Manifest hält Informationen für die Eclipse-Laufzeitumgebung bereit, damit diese weiß, wie dieses Plug-In aktiviert bzw. verwendet wird.
Ein Eclipse-Plugin/Bundle kann sechs Zustände annehmen



Quelle: OSGi Service PlatformCore Specification Release 4, Version 4.2

Abbildung 3.4: Zustandsdiagramm Eclipse-Bundle[11]

- Installed: Bundle wurde erfolgreich installiert
- Resolved: Bundle ist zum starten bereit
- Starting: Bundle wird gestartet
- Active: Bundle ist aktiv
- Stopping: Bundle wird gestoppt/beendet
- Uninstalled: Bundle wurde deinstalliert und kann nicht mehr verwendet werden

[11] Vgl. [8] Seite 9

Ein Plug-In sollte eine Aktivator Klasse besitzen welches das Interface osgi.framework.BundleActivator implementiert. Dabei werden die beiden Methoden start() und stop() implementiert, welche jeweils entscheiden, was beim Starten und stoppen passiert.

Um zusätzliche Aktionen während der Laufzeit des Plug-Ins zu ermöglichen können im Manifest sogenannte Extentions eingebettet werden. Diese Extention verweisen auf callback Objekte, welche die gewünschten Aktionen durchführen

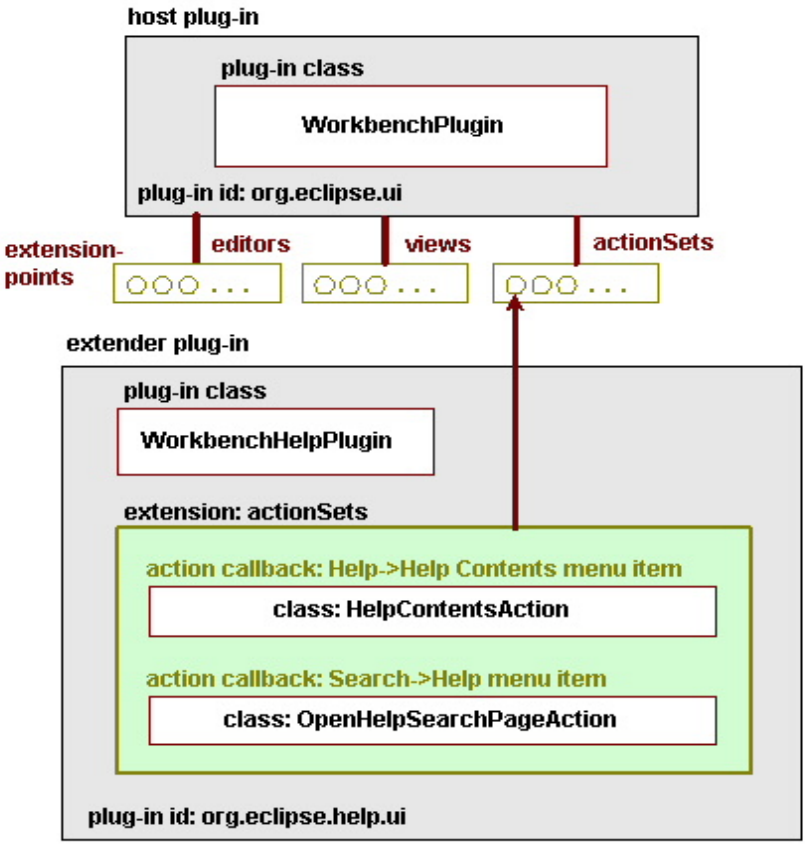


Abbildung 3.5: Beteiligte bei Eclipse-Extentions[12]

Vordefinierte Extentionpoints definieren hierbei den bestimmte Zugriffe auf die verwalteten Sourcen.

Extension Point	Description
org.eclipse.ui.command	Declarative description of the component
org.eclipse.ui.handlers	Defines the behavior, e.g., the Java class which should be called
org.eclipse.ui.menu	Where and how should the command be included in the user interface, e.g., menu, popup menu, toolbar, etc.

Tabelle 3.6: Beispiele für Extentionpoints[13]

[12] Vgl. [6]
[13] Vgl. [7]

3.3.2 Projekt-Architektur

Für die Usecases der gestellten Anforderungen wurden drei Handler Klassen erstellt, welche Zugriff auf einen gemeinsamen Aktivator haben:

- BackupHandler.java
- Patcher.java
- PatchGenerator.java

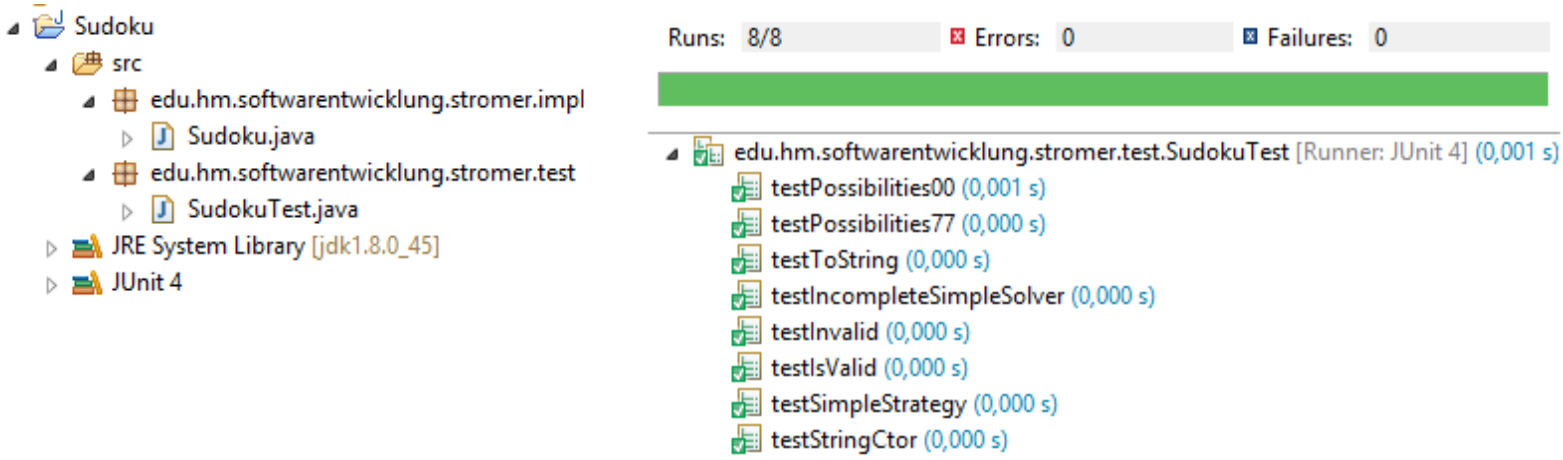
Da die Handler untereinander keine Möglichkeit der Kommunikation besitzen, verlaufen die Schnittstellen über die Connector-Klasse, welche von dem gemeinsamen Aktivator bereitgestellt wird. Der Connector stellt hierbei geteilte Daten wie z.B. die gesicherte Musterlösung und Hilfsmethoden für die Handler Klassen zur Verfügung.

Für die Erstellung der Patches und Mutanten wird die diff_match_patch library von Neil Fraser verwendet. Diese Library wird von dem Connector angesteuert.

Die einzelnen Handler werden jeweils über den Extentionpoint org.eclipse.menus angesteuert. Dieser definiert als Zugriffspunkt bei allen Handlern das Kontext/Popup-Menü des Package Explorers. Dieser ist durch die location URI popup:org.eclipse.jdt.ui.PackageExplorer adressiert.

3.3.3 Ausführung

Da wir uns jetzt mit allen relevanten Anforderungen befasst haben wird nun ein Anwendungsfall exemplarisch dargestellt. Als Beispiel wird hier die Praktikumsaufgabe Sudoku aus der Lehrveranstaltung Softwareentwicklung 2 des Sommersemesters 2015, welche von Prof. Dr. Axel Böttcher verfasst wurde, hergenommen.
Um den Ablauf möglichst genau nachzustellen werden hier mehrere Screenshots der Bedienung gezeigt



Das Projekt enthält zu der gestellten Aufgabe eine Implementierung Sudoku.java und eine zugehörige Testklasse SudokuTest.java. Bei Ausführung der Tests wird noch kein Fehler erkannt, also ist die Lösung korrekt.

Beginnen wir nun mit der Erstellung von einem Patch-File. Nach Abbildung 3.2, muss bei der Erstellung zunächst ein Backup der Lösung gemacht werden. Dies wird durch das Aufrufen des Backup Handlers über das Kontextmenü ausgelöst. Dabei wird aus dem markierten Projekt ein IProject gecasted, welches an ein neu erstellen Ordner im temporären Verzeichnis kopiert wird. Der Speicherort wird an der Statuszeile geschrieben

```
Backup successful:[redacted]AppData\Local\Temp\Sudoku1295107466751963750
```

Als nächstes müssen wir in der Lösung einen Fehler einbauen. Dazu nehmen wir als Beispiel die toString() Methode:

```
@Override
public String toString() {
    final StringBuffer sb = new StringBuffer();
    for (int row = 0; row < SUDOKU_LENGTH; row++) {
        for (int col = 0; col < SUDOKU_LENGTH; col++) {
            sb.append(grid[row][col] + " ");
        }
        sb.append('\n');
    }
    return sb.toString();
}
```

Diese Methode liefert gemäß Aufgabenstellung eine String-Repräsentation über den Inhalt des Sudokus. Dabei soll zwischen jeder Zahl ein Leerzeichen stehen. Wir bauen nun einen Fehler ein der genau dieses Leerzeichen entfernt. Die veränderte Methode sieht anschließend wie folgt aus:

```
@Override
public String toString() {
    final StringBuffer sb = new StringBuffer();
    for (int row = 0; row < SUDOKU_LENGTH; row++) {
        for (int col = 0; col < SUDOKU_LENGTH; col++) {
            sb.append(grid[row][col]);
        }
        sb.append('\n');
    }
    return sb.toString();
}
```

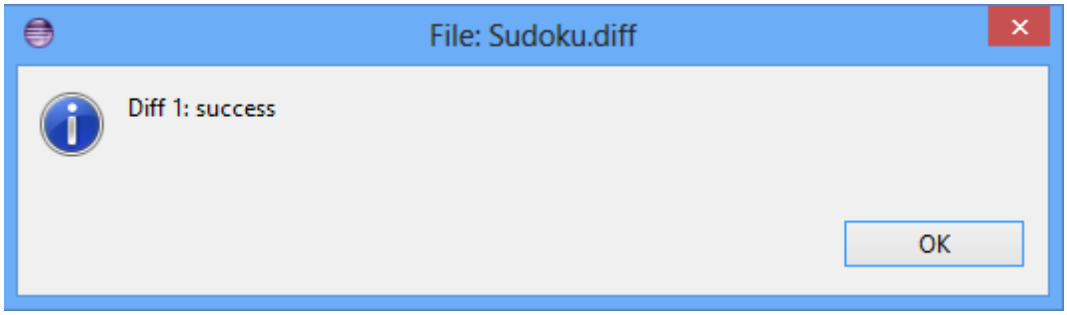
Zum Erstellen des Patch-Files, wird im Kontextmenü des Projektes nun der Punkt „Create Patches“ ausgewählt. Nach der Auswahl des Speicherortes wird das Patch-File Sudoku.diff generiert, welches wie folgt aussieht:

```
1 @@ -1682,14 +1682,8 @@
2  col%5D
3 - + %22 %22
4  );%0A
5
```

Dieses Patch-File beinhaltet wie gewünscht lediglich eine Änderung. Bei Bedarf können auch mehrere Änderungen in einen Patch-File geschrieben werden oder auch Änderungen über mehrere Klassen eingepflegt werden. Dementsprechend wird für jede Klasse, in der eine Änderung festgestellt wird, ein Patch-File generiert.

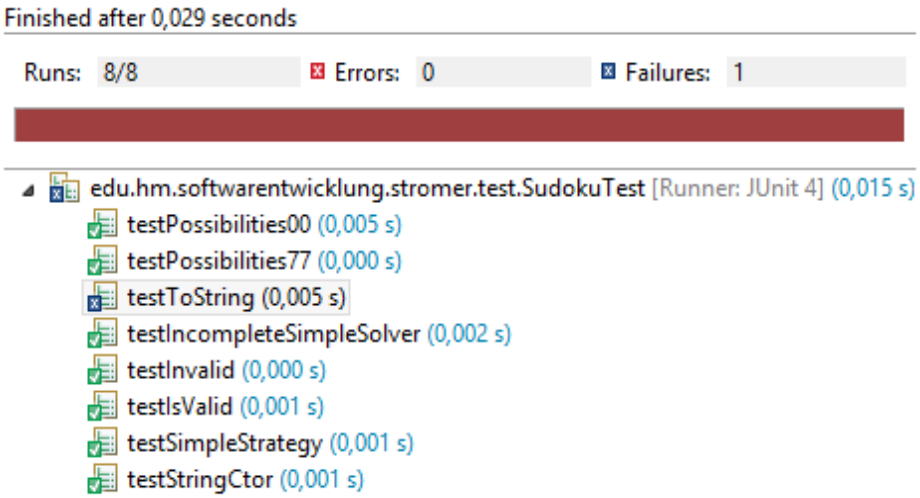
Nach erfolgreicher Generierung wird die zuvor eingegebene Änderung automatisch wieder rückgängig gemacht, sodass wieder eine korrekte Lösung vorliegt.

Um nun einen gewünschten Mutanten zu erzeugen, müssen wir unsere vorhandenen Patch-Files auf die Musterlösung einspielen. Dabei empfiehlt es sich vorher sicher zu gehen, dass die Musterlösung vorher gesichert wurde. Über den Menüeintrag „Apply Patches“ werden dann alle Patches aus dem ausgewählten Ordner auf die Musterlösung eingespielt. Dabei wird zu Kontrollzwecken zu jedem vorhandenen Patch-File ein Dialog erscheinen



Hierbei wird in jedem Dialog aufgelistet, ob die jeweilige Änderung akzeptiert wurde. Sollte eine Änderung nicht akzeptiert werden so werden keine übernommen. In diesem Fall hat sich vermutlich die Musterlösung seit der Patchgenerierung geändert und es müssen ggfs. neue Patches erstellt werden.

Nun haben wir unseren Mutanten erstellt und wollen diesen gegen den Test laufen lassen.



Wie zu erwarten schlägt der Test, der die toString()-Methode testet fehl, da nun die Leerzeichen zwischen den Zahlen im Sudoku fehlen

Die hierbei generierten Mutanten können für weiter Automatisierungszwecke extra gesichert werden, um die Tests effektiver durchzuführen. Der Testansatz geschieht hier noch manuell.

4. Ausblick

Dieser Ansatz zur Testbewertung hält gute Chancen für die Weiterentwicklung bereit, da sich mit wachsender Computerleistung auch größere Datenmengen verarbeiten lassen

Der momentane Stand dieser Lösung bietet gute Erweiterungsmöglichkeiten durch z.B. hinzufügen weiterer Action Handler. Aktuell ist dieses Tool gut für die Vorbereitung und Überprüfung der einzelnen Patch-Files geeignet, der eigentliche Testvorgang ist hier nur manuell möglich.

Eine weitere Möglichkeit besteht in der automatischen Generierung von Mutanten, mithilfe der in Kapitel 2.2.2 beschriebenen Operationen. Insbesondere kann man dadurch zusätzlich eine höhere Testabdeckung erzielen wodurch vermutlich noch mehr Fehler erkannt werden können.
Vermutlich wird sich die Mutationsanalyse bei der Bewertung in naher Zukunft am stärksten etablieren, dazu gibt es bereits eine Werkzeuge bzw. Forschungsprojekte wie z.B:

- MuJava[14]
- Mutator[15]
- Jester[16]

[14] Vgl. http://cs.gmu.edu/~offutt/mujava/
[15] Vgl. http://ortask.com/mutator/
[16] Vgl. http://jester.sourceforge.net/

Literaturverzeichnis

[1] Andreas Spillner; Tilo Linz: *Basiswissen Softwaretest*. 4. Auflage. Heidelberg, DE : dpunkt, 2010, ISBN 987–3–89864-642-0

[2] Frank Westphal:*Testgetriebene Entwicklung mit JUnit und FIT*. 1. Auflage. Heidelberg, DE: dpunkt, 2005, ISBN 3-89864-220-8

[3] Stephen H. Edwards, Zalia Shams, Michael Cogswell, Robert C. Senkbeil: *Running Students' Software Tests Against Each Others' Code: New Life for an Old "Gimmick".* Blacksburg, VA, USA

[4] A. Jefferson Offutt, Rolan H. Untch: *Mutation 2000: Uniting the Orthogonal, 45-55,* San Jose, CA, 2000: http://cs.gmu.edu/~offutt/rsrch/papers/mut00.pdf (25.07.2016)

[5] Veronika Thurner, Axel Böttcher: *An "Objects First, Tests Second" Approach for Software Engineering Education.* München, DE

[6] Azad Bolour: *Notes on the Eclipse Plug-in Architecture*: 2003, https://eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html (18.07.2016)

[7] Lars Vogel: *Eclipse-Commands*. 2014, http://www.vogella.com/tutorials/EclipseCommands/article.html (15.07.2016)

[8] Julia Dreier: *Plug-in-Programmierung mit Eclipse*: Hochschule Osnabrück, DE: 2010, http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/Eclipse/arbeitsanleitungPlugInProgrammierung.pdf (15.07.2016)