Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Abschlussarbeit Bachelor of Science
Erstbetreuer: Dr. Axel Schmolitzky
Zweitbetreuerin: Dr. Carola Eschenbach

Modul- und Systemtests am Fallbeispiel StuReSy

Vorgelegt von:
Jens Dallmann
Matrikelnummer: 6043783
8dallman@informatik.uni-hamburg.de

Software-System-Entwicklung
8. Fachsemester

Hamburg, den 22. Mai 2013

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit beschäftigt sich mit dem von Wolf Posdorfer im Rahmen seiner Bachelorarbeit entwickelten Audience-Response-System (ARS) [1] StuReSy. Ein ARS ist ein System mit dessen Hilfe abgestimmt werden kann. Das freie ARS StuReSy von Wolf Posdorfer ist als Prototyp für den Einsatz in der Lehre entwickelt worden. Als Prototyp weist das System softwaretechnische Mängel auf, die die Wartbarkeit und die Weiterentwicklung beeinträchtigen. Die Mängel müssen in Zukunft behoben werden. Bevor dies geschehen kann, muss zunächst die bestehende Funktionalität abgesichert werden, damit durch das Beheben keine Fehler eingearbeitet werden. Das Ziel dieser Arbeit ist das Absichern der Funktionalität mit Hilfe von Modul- und Systemtests.

Während dieser Arbeit wurde das System mit Modultests versehen. Da die Modultests nicht ausreichten, um die Funktionalität ausreichend abzusichern, traten zusätzlich Systemtests in den Fokus der Arbeit. Bei diesen fiel das Problem der aufwändigen Erzeugung und Implementierung auf, weswegen eine Lösung für das Problem gesucht wurde.

Diese Ausarbeitung beginnt mit einer fachlichen und technischen Einführung in das Software-System StuReSy. Darauf folgend werden Tests in Softwareprojekten betrachtet und anschließend die Testarten Modul- und Systemtests näher erläutert. Im Anschluss werden Modul- und manuelle Systemtests in StuReSy eingeführt. Anschließend wird eine Möglichkeit gezeigt Systemtests in HTML-Tabellen zu implementieren. Im letzten Kapitel werden die Ergebnisse in den Bereichen Modul- und Systemtests näher erläutert. Im Bereich der Modultests wird der Einsatz von Mock-Objekten und das Messen der Testabdeckung diskutiert und die Verwendung der Werkzeuge Mockito und Cobertura bewertet. Bei den Systemtests liegt der Fokus auf der systematischen Erstellung der Testfälle und der Implementation von automatisierten Testfällen in HTML-Tabellen durch die Werkzeuge Framework for integrated test (Fit) und FEST.

2. Das Audience Response System StuReSy

StuReSy ist ein ARS, das frei verfügbar und speziell für die Lehre entwickelt worden ist. Die Benutzerschnittstelle ist sprachlich an die Lehre angepasst, jedoch kann das System auch zu Abstimmungen in anderen Kontexten verwendet werden. Das System kann aus verschiedenen Sichten betrachtet werden. Zum einen aus der Sicht der Benutzer, zum anderen aus der Sicht der Entwickler. Die Benutzer sind an der Funktionalität interessiert und müssen nichts über die technische Realisierung wissen. Die Entwickler hingegen müssen die Funktionalität und die technische Realisierung kennen. Aus diesem Grund werden die beiden Sichten auf StuReSy getrennt betrachtet. Die Beschreibung des Systems bezieht sich auf die Version 0.5 vom 12. September 2012.

2.1. Funktionale Sicht - Die Anforderungen der Benutzer

Die funktionale Sicht auf StuReSy enthält zwei Perspektiven. Die Perspektive desjenigen, der eine Abstimmung durchführen will, und die Perspektive derjenigen, die an der Abstimmung teilnehmen wollen. Beide Perspektiven haben unterschiedliche Benutzungsszenarien des Systems und haben daher auch verschiedene Anforderungen an das System.

2.1.1. Anforderungen der Dozenten

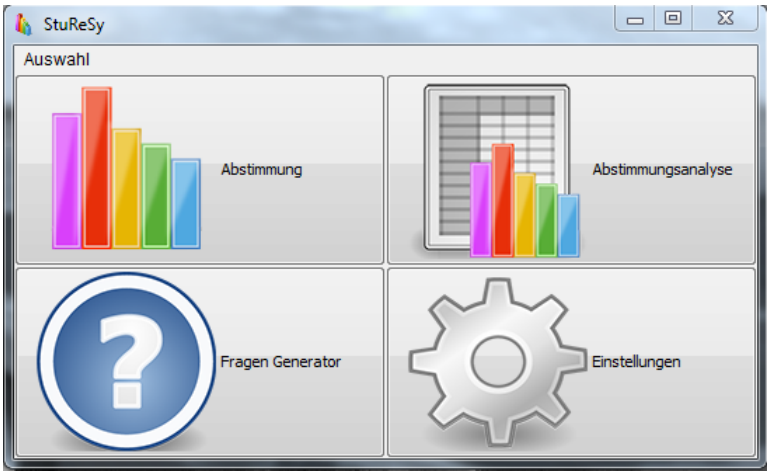


Abbildung 1: Startbildschirm von StuReSy

Für Dozenten ist ein Klient entwickelt worden, der es ermöglicht, Fragen zur Abstimmung zu stellen. In Abb. 1 ist der Klient nach dem Start zu sehen. Der Klient bietet nach dem Start direkten Zugriff auf die vier Grundfunktionen: Frageneditor, Abstimmung durchführen, Analyse durchgeführter Abstimmungen und Einstellungen. Im Folgenden werden die Anforderungen der einzelnen Grundfunktionen aufgelistet.

- Frageneditor
  - Es soll Vorlesungen geben.

  - Zu Vorlesungen sollen Fragesätze erzeugt werden können.
  - Zu jedem Fragesatz sollen beliebig viele Fragen zugeordnet werden können.
  - Eine Frage soll bestehen aus:
    * einem Fragetext
    * 2-10 Antworten
    * genau einer oder keiner richtigen Antwort.
    * der Dauer ihrer Abstimmung.

- Abstimmung durchführen
  - Es soll ein Fragesatz aus einer Vorlesung ausgewählt werden können.
  - Es soll jeweils genau eine Frage zur Abstimmung stehen.
  - Zu Beginn soll die erste Frage des Fragesatzes zur Abstimmung stehen.
  - Es soll in dem Fragesatz navigiert werden können, um die zur Abstimmung gestellte Frage auszuwählen.
  - Die Dauer soll angepasst werden können.
  - Die Reaktion auf die Frage soll auf verschiedene Arten angezeigt werden können:
    * In einem Balkendiagramm
    * Durch Einfärben der korrekten Antworten im Balkendiagramm oder in der textuellen Anzeige
  - Die Stimmen sollen zurückgesetzt und die Abstimmung erneut begonnen werden können.
  - Die Abstimmung soll beliebig gestoppt und fortgeführt werden können.

- Durchgeführte Abstimmungen analysieren
  - Es soll ein Fragesatz aus einer Vorlesung ausgewählt werden können.
  - Zu Beginn soll die erste Frage des Fragesatzes angezeigt werden.
  - Es soll die Möglichkeit geben, durch die Fragen des Fragensatzes zu navigieren
  - Die Ergebnisse sollen wie folgt Dargestellt werden:
    * durch ein Balkendiagramm mit prozentualen Stimmanteilen der Antworten.
    * durch ein Diagramm mit zeitlichem Verlauf: Vergangene Zeit einer Stimmabgabe nach Beginn der Abstimmung
    * durch eine Zusammenfassung der Diagramme in tabellarischer Form mit anonymisiertem Schlüssel, an dem erkennbar ist, über welche Benutzerschnittstelle die Stimme eingegangen ist.

  - die Zusammenfassung soll im Format Comma-separated values (CSV) exportiert werden können.

- Einstellungen
  - Es sollen weitere Hardwareschnittstellen als Plugins importiert werden können.
  - Die geladenen Plugins sollen Einstellungen bereitstellen können.

2.1.2. Die Anforderungen der Studenten

Die minimalen Anforderungen des Studenten bestehen lediglich daraus, dass er die Möglichkeit haben will, anonym eine Stimmabgabe durchzuführen. Dazu soll es zwei Kanäle geben, die dieselbe Funktionalität bereitstellen. Ein Kanal soll das Internet sein, wofür eine Webseite benötig wird. Dies soll ermöglichen, dass die Stimmabgabe unabhängig von der räumlichen Distanz erfolgen kann. Als zweiter Kanal sollen Hardwaresysteme der Firma H-ITT [2] benutzt werden können. Diese bestehen aus einem Empfänger und beliebig vielen Geräten zum Abstimmen, die nachfolgend als *Clicker* bezeichnet werden.

2.2. Technische Sicht - Die Perspektive der Entwickler

Die technische Sicht auf StuReSy wird in vier Unterkapiteln näher beleuchtet. Um einen Überblick über die zu testenden technischen Komponenten zu erhalten, werden diese und ihre Kommunikation untereinander vorgestellt. Danach wird betrachtet, wie das Einpflegen von Abhängigkeiten zu anderen Bibliotheken und das automatische Ausführen von wiederkehrenden Aufgaben in das System integriert ist. Abschließend werden die bisherigen Maßnahmen zur Qualitätssicherung und deren automatische Ausführung erläutert.

2.2.1. Technische Komponenten im Software-System StuReSy

In StuReSy gibt es bisher drei verschiedene Komponenten. Dies sind: der Dozentenklient, das Web-Plugin, sowie ein Plugin zum Benutzen von Clickern der Firma H-ITT.

Dozentenklient Der Dozentenklient ist mit ungefähr 8600 Zeilen Quelltext die größte Komponente im System. Er ist in der Programmiersprache Java nach dem Werkzeug- und Materialansatz (WAM-Ansatz) [3] entwickelt worden. Der Kern dieses Ansatzes ist es, Softwareprogramme nach Metaphern zu strukturieren. Die Abkürzung WAM steht für Werkzeug Automat Material und gibt die bedeutendsten Metaphern des Entwurfskonzepts wieder.
Werkzeuge bilden die Schnittstelle zur Ein- und Ausgabe zu dem Benutzer. Wie ein handwerkliches Werkzeug soll eine Oberfläche einer Software benutzt werden können, um ein Material zu bearbeiten oder zu betrachten. Ein Beispiel für ein handwerkliches Werkzeug ist ein Hammer, mit dessen Hilfe intuitiv Nägel in Holz

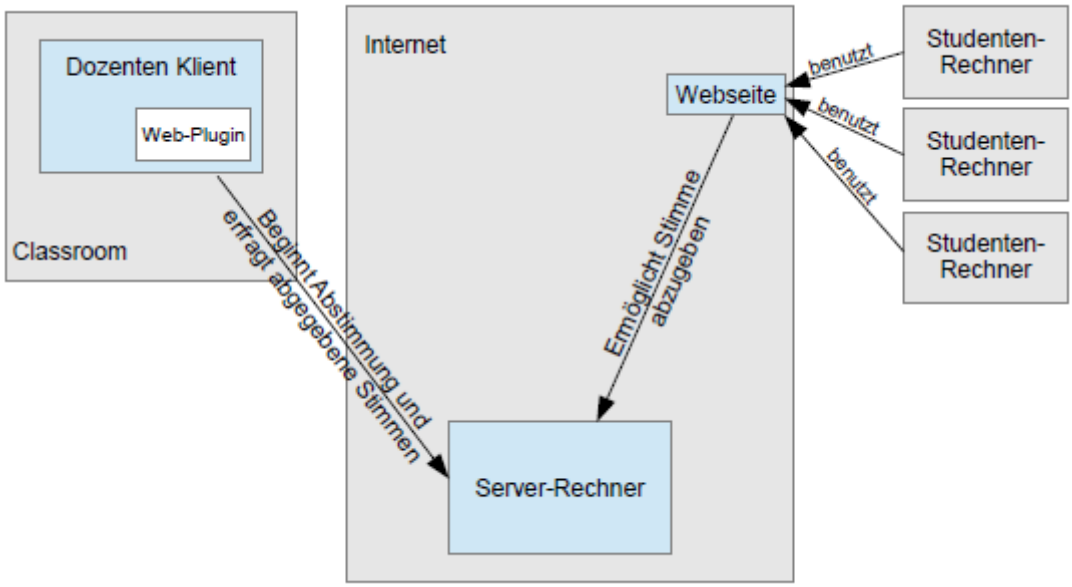


Abbildung 2: Verteilt benutzte Hardware-Komponenten des Web-Plugins

geschlagen werden können. Ein Beispiel für ein Werkzeug in der Softwareentwicklung ist eine Eingabemaske zum Erzeugen einer Rechnung. Die Eingabemaske bietet die Möglichkeit, die für die Rechnung relevanten Eingaben zu tätigen, und sollte dabei ebenso intuitiv nutzbar sein. Durch Bestätigung der Eingaben wird vom Werkzeug der Prozess zur Bearbeitung des Materials angestoßen. In diesem Fall ist das bearbeitete Material das Rechnungsdokument. Das Ergebnis der Datenverarbeitung kann beispielsweisse ein PDF-Dokument oder eine gesendete E-Mail sein.
Die fachlichen Grundlagen von StuReSy sind Fragesätze und Fragen. Diese sind in den Materialien *QuestionSet* und *QuestionModel* wiederzufinden. Fragesätze werden durch das Werkzeug *QuestionGenerator* erzeugt oder bearbeitet. In StuReSy bestehen Werkzeuge aus mindestens zwei Klassen. Eine Klasse enthält die grafische Benutzungsschnittstelle, die andere die Logik für die Schnittstelle. Die Klassen besitzen die gleichen Namen, mit dem Unterschied, dass die Klasse mit der grafischen Benutzungsschnittstelle auf UI (Userinterface) endet. Über weitere Werkzeuge können Einstellungen vorgenommen, Fragesätze zur Abstimmung gestellt oder vergangene Abstimmungen analysiert werden.
Damit die Fragesätze und ihre Ergebnisse zu einem späteren Zeitpunkt abrufbar sind, werden sie auf dem Datenträger gespeichert. Der Benutzer wählt einen Ordner in seinem Dateisystem aus, in dem StuReSy die Vorlesungen, Fragesätze und Ergebnisse in Extensible Markup Language (XML)-Dateien verwaltet.

Web-Plugin Das Web-Plugin wird standardmäßig mit StuReSy ausgeliefert und beinhaltet die Möglichkeit, an einem Server die Ergebnisse abzufragen, die über eine Webseite an den Server geschickt wurden. Daraus folgt, dass zu dem Web-Plugin drei trennbare Softwarekomponenten gehören (s.h. Abb. 2). Zunächst gibt es einen Server, der Ergebnisse von laufenden Abstimmungen bereit hält. Er ist in PHP programmiert und benutzt zum Speichern der Daten eine Datenbank. Für den

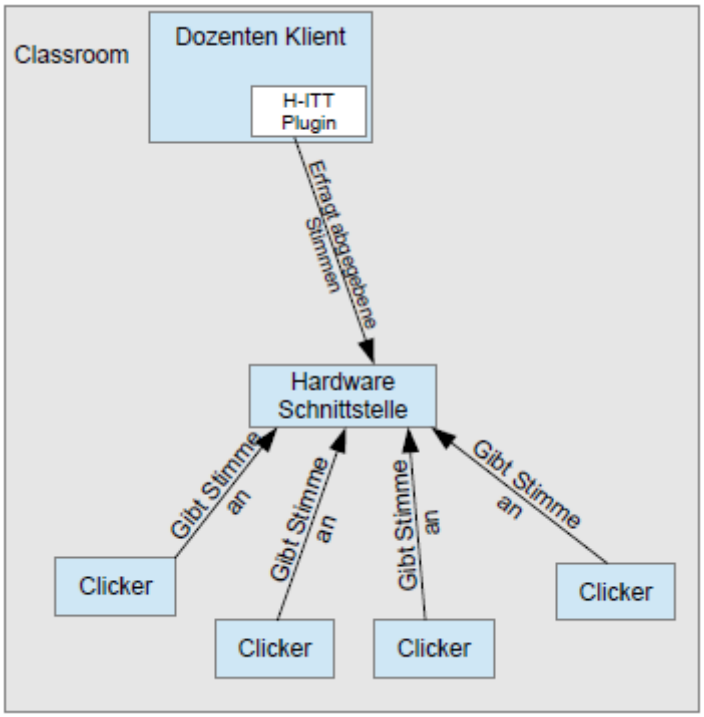


Abbildung 3: Verteilt benutzte Hardware-Komponenten des H-ITT-Plugins

Server gibt es eine Schnittstelle zum Durchführen der Abstimmung (siehe Anhang A) und eine Benutzerschnittstelle für administrative Einstellungen. Eine weitere Komponente ist die Webseite, über die das Abstimmen für Studenten möglich ist. Diese ist mit PHP, HTML und CSS entwickelt. Sie sendet eine Stimmabgabe zu einer laufenden Abstimmung an den Server, der sie dann verarbeitet (siehe Anhang B). Die letzte Komponente ist das im Dozentenklienten mitgelieferte und geladene Plugin. Das Plugin stellt in regelmäßigen Abständen Anfragen an den Server und lässt sich die abgegebenen Stimmen mitteilen. Der Server läuft in der Regel physisch getrennt von dem Dozentenklienten.

H-ITT-Clicker-Plugin Die Kommunikation mit H-ITT-Clickern funktioniert nicht wie das Web-Plugin über Internet als Übertragungsmedium, sondern über Radiofrequenzen. Mit Hilfe einer Hardware-Schnittstelle, die an den Rechner des Dozenten angeschlossen werden muss, können die Stimmen von Clickern (Hardwaregeräte zum Abstimmen) gesammelt werden. Beim Abstimmen über einen Clicker wird die Stimme an die Hardware-Schnittstelle gesendet und mit Hilfe einer nativen Bibliothek an das in Java realisierte Plugin weitergegeben. Dort werden sie vorbehalten, bis der Klient des Dozenten sie abfragt. Wie in Abb. 3 zu sehen, gibt es für diesen Kanal keinen physisch getrennten Server, über den die Abstimmung läuft.

2.2.2. Erstellungsprozess

Im Folgenden wird als *Erstellungsprozess* das Erzeugen einer ausführbaren Datei bezeichnet. In der Java-Dokumentation [4] wird gezeigt, wie ein manueller Erstel-

lungsprozess einer einfachen in Java geschriebenen Software aussieht. Zunächst muss der Quelltext kompiliert und in ein jar-Archiv gepackt werden. Zusätzlich muss eine *Manifest*-Datei [5] abgelegt werden, in der die Konfiguration für die Ausführung des Archives beschrieben ist. Wenn ein Projekt auf mehrere Komponenten aufgeteilt wird, müssen diese Schritte nacheinander für jede einzelne Komponente vollzogen werden. Je komplexer eine Software wird, desto komplexer wird der Prozess. Daher bietet es sich an, diesen Prozess zu automatisieren. Hierzu werden verschiedene Werkzeuge für Java angeboten. Das in StuReSy benutzte Werkzeug heißt ANT [6]. In ANT können Entwickler wiederkehrende Aufgaben automatisieren. Dies kann für unterschiedliche Ziele benutzt werden. Ein Ziel heißt in ANT *target*. Beispiele für *targets* sind das Kompilieren von Quelltext oder das Erzeugen einer ausführbaren Datei. Ziele können mit Hilfe von konfigurierbaren Werkzeugen erreicht werden. Diese Werkzeuge heißen *tasks*. Ein Beispiel ist das Werkzeug JarBundler [7], dass ausführbare Dateien für Max OS X erstellt. Welche Ziele von ANT übernommen werden sollen, ist in einer XML-Datei, der Datei *build.xml*, beschrieben.
Für jede Komponente in StuReSy ist eine Datei *build.xml* angelegt. Dies betrifft die Komponenten: Dozentenklient und H-ITT-Plugin. Das in der *build.xml* des H-ITT-Plugin beschriebene Ziel ist, ein Zip-Archiv mit den kompilierten Klassen zu erzeugen, damit es später im Dozentenklient importiert werden kann. Die Datei *build.xml* des Projektes für den Dozentenklienten beinhaltet neben dem Erzeugen eines plattformunabhängigen ausführbaren JAR-Archives zusätzlich das Erzeugen einer speziell für Mac OS X ausführbaren Datei.

2.2.3. Softwarequalität

Der Begriff der Softwarequalität umfasst nach ISO-Norm 9126 [8] die Faktoren Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit. Durch das Beschränken auf Modul- und Systemtests werden die Faktoren Funktionalität und Änderbarkeit in den Fokus der Arbeit gesetzt. Dazu werden zwei Testverfahren benutzt. Dies sind statische Tests und dynamische Tests. Während sich statische Tests auf den Quelltext der Software beziehen, beziehen sich dynamische Tests auf die Software zur Laufzeit [9]. Im Folgenden werden die zu Beginn dieser Arbeit benutzten statischen und dynamischen Qualitätssicherungsansätze analysiert.
In der Analyse wurden zwei bereits integrierte Ansätze aufgedeckt. Der erste Ansatz beschäftigt sich mit der Vereinheitlichung des Quelltextes. Hierzu wurden drei Maßnahmen getroffen. Es wurde eine Datei mit dem Lizenztext, der zu Beginn jeder Projektdatei stehen soll, abgelegt. Außerdem wurde eine Formatierungsrichtlinie erzeugt und die Datei für den Import in Eclipse bereitgestellt. Dadurch wird ermöglicht, einen einheitlichen Quelltextstil beizubehalten. Abschließend wurde statische Quelltextanalyse mit Checkstyle [10] eingeführt, um den einheitlichen Quelltext abzusichern. Checkstyle benutzt Regeln, die auf den Quelltext angewendet werden, welche vordefinierte Formatierungen und Quelltextrichtlinien prüfen. Hierzu wurde eine Konfigurationsdatei für Checkstyle erzeugt, die speziell auf das System StuReSy angepasst wurden. Die Überprüfung mit Checkstyle wurde vor Beginn der Arbeit bereits in den Erstellungsprozess integriert, damit ab einer bestimmten Anzahl von Regelverstößen das Projekt nicht gebaut werden kann. Dies soll verhindern, dass Regelverstöße ignoriert werden und erzwingt, dass spätestens kurz vor dem Bereit-

stellen als Download behoben werden.
Der zweite Ansatz, die Qualität zu sichern, sind Modultests. Modultests testen kleine Einheiten eines Programms. Vor Beginn dieser Arbeit sind insgesamt 7 Testklassen mit 11 Testfällen vorhanden. Diese sind für den Dozentenklienten. Die automatische Ausführung ist im ANT-Skript integriert. Für die anderen Teilnehmer am System sind keine Tests vorhanden. Um einen Einblick über die abgedeckten Quelltextstellen durch die bestehenden Tests zu bekommen, wurde im Rahmen dieser Arbeit ein Werkzeug benutzt, das visualisiert welche Klassen und Methoden durchlaufen werden. Abb. 4 zeigt den Bericht mit den vorhandenen Testklassen. Es wurden ins-

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	179	9%	382/3855	15%	146/924	2,018
sturesy	11	0%	0/246	0%	0/54	1,833
sturesy.export	9	74%	131/177	82%	61/74	4
sturesy.items	4	61%	116/188	49%	26/53	1,764
sturesy.plugin	4	0%	0/84	0%	0/36	2,812
sturesy.qgen	48	0%	0/872	0%	0/213	1,871
sturesy.qgen.gui	3	0%	0/191	0%	0/16	1,211
sturesy.settings	9	0%	0/147	0%	0/22	1,677
sturesy.settings.about	2	0%	0/39	0%	0/2	1,333
sturesy.settings.mainsettings	3	0%	0/36	0%	0/2	1,111
sturesy.settings.websettings	15	0%	0/260	0%	0/32	1,538
sturesy.util	12	18%	119/652	23%	53/224	3,87
sturesy.voting	34	3%	16/496	4%	6/124	1,857
sturesy.voting.gui	7	0%	0/144	0%	0/10	1,172
sturesy.voting.webvoting	4	0%	0/97	0%	0/14	1,714
sturesy.votinganalysis	11	0%	0/144	0%	0/42	1,84
sturesy.votinganalysis.gui	3	0%	0/82	0%	0/6	1,231

Abbildung 4: Testabdeckungsbericht des Quelltextes des Dozentenklienten in Version 0.5 unter Berücksichtigung aller Klassen

gesamt 9% der Zeilen im Quelltext und 15% der Verzweigungen durchlaufen. Diese Messung bezieht sich auf den gesamten Quelltext des Dozentenklienten.

2.3. Zukunftsaussichten von StuReSy

In der Beschreibung der fachlichen Sicht ist erkennbar, dass das bestehende System die Grundfunktionalität abdeckt. Im Laufe der Zeit kamen Ideen für weitere Funktionalität, die entweder im Rahmen von Bachelorarbeiten oder vom Hauptentwickler Wolf Posdorfer entwickelt werden. Parallel zu dieser Arbeit laufen zwei weitere Bachelorarbeiten. Eine der Arbeiten beschäftigt sich mit dem Entwickeln einer Anwendung für Smartphones mit dem Betriebssystem Android. Hierbei wird ein weiterer Klient zum Abstimmen für die Studenten entwickelt. Die andere Bachelorarbeit portiert den Dozentenklient auf iPads. Wolf Posdorfer beschäftigt sich damit das bestehende System zu warten und weiterzuentwickeln. Dabei geht es überwiegend darum, die Benutzbarkeit zu verbessern und weitere Funktionalität einzuführen.
Zweck dieser Arbeit ist es, das bestehende System durch das Einführen von Tests auf die Wartung und Weiterentwicklung vorzubereiten.

3. Tests in Softwareprojekten

Software ist ein Produkt und Produkte müssen bestimmte Anforderungen erfüllen, damit sie den Kunden oder Zielgruppen genügen. Um sicherzustellen, dass eine Software den Anforderungen genügt, können Softwaretests benutzt werden. Zu Softwaretests gehören sowohl dynamische, als auch statische Testverfahren [9]. In dieser Arbeit werden dynamische Testsverfahren behandelt. Zu den dynamischen Testverfahren gehören unter anderem Modultests, Integrationstests, Systemtests und Akzeptanztests. Gegenstand dieser Arbeit sind Modultests und Systemtests. Bevor diese beiden Testarten erläutert werden, werden zunächst Black-Box-Tests und White-Box-Tests erläutert.
Als Black-Box-Tests werden Tests bezeichnet, bei deren verfassen keine Kenntnisse über die Implementation des *Testobjekts* vorhanden sind [11]. Als Testobjekt wird ein System, Teilsystem oder eine Komponente bezeichnet [9]. Im Mittelpunkt des Black-Box-Tests stehen die funktionalen Anforderungen. Ein Beispiel hierfür sind Tests, die durchgeführt werden, um sicherzustellen, dass die implementierte Funktionalität der gewünschten Funktionalität entspricht. Auf Seiten der Entwicklung spielen die Systemtests eine große Rolle, da sie vor der Abnahme des Kunden beziehungsweise der Nutzung im Produktivbetrieb stehen.
Werden bei den Tests Kenntnisse über die technische Realisierung des Testobjekts benutzt, wird von White-Box-Tests gesprochen [11]. Durch diese Kenntnisse sind komplexe Stellen im Quelltext bekannt und können besonders intensiv getestet werden. Modultests sind Tests für einzelne Module, aus denen eine Applikation besteht. Da sie meistens von den Entwicklern der Module geschrieben werden, werden sie häufig als White-Box-Tests geschrieben.
Software birgt das Problem, dass Veränderungen die bestehende Funktionalität beeinflussen können. Deshalb sollten möglichst viele White- und Black-Box-Tests Regressionstests sein. Regressionstests sind Tests, die immer wieder ausgeführt werden, um die Auswirkungen von Veränderungen zu beobachten [9]. Die Ausführung von Regressionstests kann manuell oder automatisiert erfolgen. Je früher die Tests erneut ausgeführt werden, desto schneller können Fehler aufgedeckt und behoben werden. Die manuelle Ausführung kann sehr viel Zeit in Anspruch nehmen, weswegen eine dauerhafte manuelle Ausführung sehr teuer sein kann. Ein weiterer Nachteil ist die Gefahr, dass die Ausführung der Tests in zeitkritischen Phasen vernachlässigt wird. Aus diesem Grund lohnt es sich, Regressionstests zu automatisieren. Jeder Test muss einmal programmatisch umgesetzt und im weiteren Verlauf des Projekts bei Änderungen der Anforderungen überarbeitet werden.
Da das vollständige Testen von Testobjekten unmöglich ist [9], muss festgelegt werden, bis zu welchem Maß sich Tests lohnen und ab wann der Nutzen der Tests zu gering ist, um den Mehraufwand zu rechtfertigen. Dazu wird betrachtet, welcher Teil des Testobjekts von einem Test durchlaufen wurde. Bei Black-Box-Tests wird betrachtet, welche Anforderungen durchlaufen wurden [11], und bei White-Box-Tests kann betrachtet werden, welche Quelltextstellen getestet sind [12].

3.1. Modultests

Wie zuvor beschrieben, werden in Modultests einzelne Module einer Software getestet. Ein Modul ist eine abgeschlossene Einheit, die unabhängig vom restlichen Quelltext kompiliert werden kann. Bei objektorientierten Sprachen sind das häufig einzelne Klassen. Der Sinn der Tests liegt darin, die Funktionalität einzelner Module zu testen, bevor sie in den Gesamtkontext des Systems integriert werden. Dies hilft dabei, frühzeitig Fehler zu finden. Fehler, die erst bei der Integration der Komponenten auftauchen, sind schwerer zu finden, da das Fehlverhalten in allen beteiligten Komponenten auftreten kann. Modultests sind stark abhängig von der Architektur, da die Module losgelöst von anderen Modulen getestet werden sollen [13]. Da Modultests Entwicklerschnittstellen statt Benutzerschnittstellen benutzen, sind sie im Vergleich mit den angesprochenen Systemtests leicht automatisierbar. Es gibt eine Vielzahl von Werkzeugen, die dies übernehmen. Das bekannteste Werkzeug für Java ist JUnit [14]. Bei JUnit wird ein Test in einer Testklasse beschrieben. In dieser Klasse können mehrere Testfälle geschrieben werden. Zusätzlich gibt es Methoden, die vor und nach dem Ausführen jedes Testfalls ausgeführt werden und Methoden die vor und nach des gesamten Tests ausgeführt werden.
Ein weiterer Nutzen von Modultests im Vergleich zu den umfassenden Systemtests ist die Ausführungsdauer. Einzelne Modultests können in akzeptabler Zeit ausgeführt werden. Da Modultests genauso wie Systemtests zu den Regressionstests zählen, ist eine häufige Ausführung der Tests ein wichtiger Aspekt.

3.1.1. Modultests aus einem Klassendiagramm aufbauen

Eine objektorientierte Software besteht aus vielen Modulen, die getestet werden können. Um Module möglichst einfach zu testen, muss die Architektur des Programms das Testen von Modulen unterstützen. Ein mögliches Problem ist, dass Module nicht unabhängig voneinander getestet werden können. Dies entspricht nicht dem Zweck von Modultests und es müsste eine Restrukturierung des Quelltextes stattfinden. Zum Einführen von Modultests in ein bestehendes Programm wird von Peter Liggesmeyer [12] eine Strategie erläutert, die Klassendiagramme zu Hilfe nimmt.
Zur Vorbereitung der Modultests sollen zunächst Klassendiagramme erstellt werden. Diese Klassendiagramme benötigen nicht die genauen Attribute oder Methoden, sondern lediglich die Beziehungen zu anderen Klassen. Anhand der Beziehungen kann erkannt werden, welche Klassen von anderen benutzt werden. Wenn eine Klasse eine andere benutzt, sollte die benutzte Klasse bereits gemäß den Anforderungen funktionieren, da sonst Fehler in der benutzenden Klasse auftauchen, die nicht von ihr stammen. Anhand des Klassendiagramms kann eine Reihenfolge der zu testenden Klassen erstellt werden. Dabei kann das Klassendiagramm als ein Graph betrachtet werden. Die Klassen, die die Blätter des Graphen darstellen, sind von weiteren Klassen unabhängig und können unabhängig getestet werden. Um die Klassen, die durch die inneren Knoten des Graphen dargestellt sind, zu testen, sollten zunächst die Klassen der Knoten des Graphen getestet werden, zu der eine Kante besteht.
In Abb. 5 ist ein vereinfachtes Klassendiagramm dargestellt. Aufgrund der Implementation ist es nicht möglich die Module einzeln zu testen, daher soll im Folgenden nach dem oben erklärten Schema eine Testreihenfolge entwickelt werden.

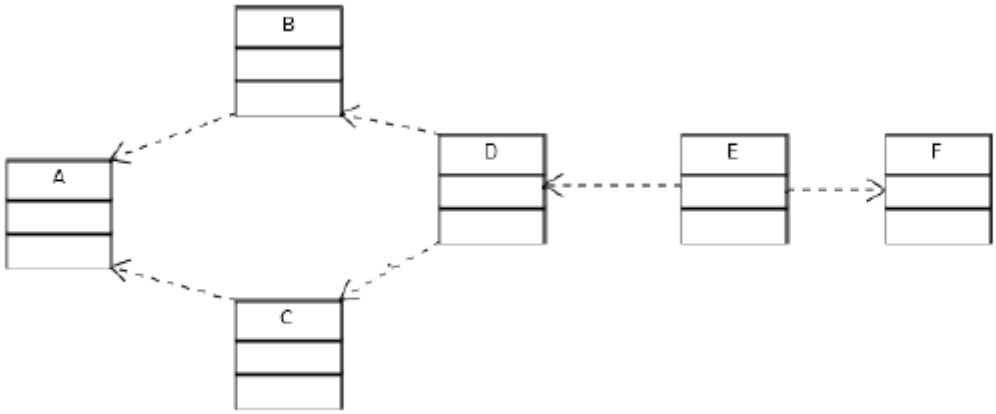


Abbildung 5: Klassendiagramm eines Teil eines Programms

In den Blättern des Graphen befinden sich die Klassen A und F. Diese können unabhängig getestet werden. Um die inneren Knoten der Klassen B und C zu testen, sollte die Klasse A bereits getestet sein. Der innere Knoten für die Klasse D besitzt Kanten zu den Knoten für die Klassen B und C, daher ist die Klasse D im Anschluss testbar. Für die Klasse E ergibt sich nach dem gleichen Schema, dass zunächst die Klassen D und F getestet sein sollten.

3.1.2. Testabdeckung

Für die Entwickler von Modultests kann es von Nutzen sein, einen Überblick darüber zu gewinnen, welche Zeilen des Quelltextes des Testobjekts durch die Modultests getestet sind. Es gibt verschiedene Maße, um die Testabdeckung zu messen [12]. Diese Maße werden *Überdeckungsmaße* genannt. Im Folgenden werden die Maße zur Zeilenüberdeckung und Zweigüberdeckung, die für den späteren konstruktiven Teil dieser Arbeit benötigt werden, erläutert. Die Zweigüberdeckung ist strenger als die Zeilenüberdeckung, dies bedeutet, dass die Zeilenüberdeckung durch die Zweigüberdeckung vollständig überdeckt ist. Damit die im Folgenden erklärten Metriken Sinn ergeben, muss das Testobjekt vorher festgelegt werden und welche Tests mit einbezogen werden. Werden beispielsweise die Anzahl der Zeilen für eine Metrik benötigt, ergibt die sich aus dem Testobjekt. Wird nur ein Modul getestet, dann werden nur die Zeilen des Moduls gezählt.

Zeilenüberdeckung Die Zeilenüberdeckung ist ein Maß dafür, wieviele Zeilen im Verhältnis zu der Anzahl der gesamten Zeilen im Laufe der Tests benutzt werden. Daher ergibt sich die einfache Formel:

$$\mathrm{C0} = \frac{\text{Anzahl der durchlaufenen Zeilen}}{\text{Anzahl der Zeilen}}$$

Dies wird benötigt, um festzustellen, ob ein Test alle Zeilen erreicht. Dadurch, dass Zeilen nicht durchlaufen worden sind, können diese Zeilen noch nicht auf die Existenz von Fehlern überprüft worden sein. Das Problem bei der Zeilenüberdeckung

ist allerdings, dass die Komplexität der aufgerufenen Zeile keine Rolle spielt. Zum Beispiel muss die Zeile
boolean c = a || b
nur einmal durchlaufen werden, damit die Zeilenüberdeckung sie als überdeckt zählt. Tatsächlich gibt es aber weitere Variablenbelegungen für a und b die zu Ergebnissen in c führen. Die Zeilenüberdeckung sagt in diesem Beispiel nichts über die tatsächliche Testabdeckung aus.
Ein weiteres Problem ist das Verhältnis der Anzahl der Zeilen pro Verzweigung einer Kontrollstruktur. Eine Verzweigung die viele Zeilen enthält, erzeugt für die Zeilenüberdeckung eine hohe Überdeckung, wobei die Testeffizienz nicht besser sein muss. In Listing 1 ist ein Beispiel, das dies verdeutlicht. Als Testobjekt wird die Methode istANull(**int** a)
betrachtet. Wenn die Bedingung im if-Zweig zutrifft, dann werden vier Zeilen durchlaufen, wenn a nicht 0 ist, dann wird lediglich eine Zeile durchlaufen. Mit der Variablenbelegung
a=0
kann daher eine hohe Zeilenüberdeckung erreicht werden. Es ist jedoch zu beachten, dass der else-Zweig vermutlich häufiger durchlaufen werden würde, da jede andere beliebige ganze Zahl im Integer-Wertebereich nicht 0 ist.

Listing 1: if-else Beispiel zur Erklärung der Unterschiede zwischen Anweisungs- und Zweigüberdeckung

```
1  public String istANull(int a) {
2    if(a == 0) {
3      String konkatenierterString = "a ";
4      konkatenierterString += "ist ";
5      konkatenierterString += "0";
6      return konkatenierterString;
7    }
8    else {
9      return "a ist nicht 0";
10   }
11 }
```

Zweigüberdeckung Besser geeignet für das in Listing 1 erwähnte Beispiel ist die Zweigüberdeckung. Zweigüberdeckung misst die Anzahl der durchlaufenen Verzweigungen. Im obigen Beispiel ist das sehr einfach. Es gibt insgesamt zwei Zweige. Beim Aufruf des Testobjektes mit der Variablenbelegung
a = 0
wird ein Zweig durchlaufen. Insgesamt werden 50% der Zweige durchlaufen. Das gleiche Ergebnis wird durch den Aufruf des Testobjektes mit der Variablenbelegung
a = 1
erwartet.
Dies zeigt die höhere Strenge der Zweigüberdeckung gegenüber der Anweisungsüberdeckung. Das Ergebnis für die Belegung
a = 0
ist für die Zweigüberdeckung schlechter wie für die Anweisungsüberdeckung.

3.2. Systemtests

Systemtests sind rein funktionale Tests und testen das Verhalten der Funktionalität im Kontext der gesamten Software [9]. Das System sollte zum Testen wie ein Produktivsystem aufgesetzt werden. Allerdings werden zu Systemtests lediglich Testdaten herangezogen. *Testdaten* sind Daten, die nicht durch ein bestehendes Produktivsystem erzeugt wurden. Sie sind fiktiv und müssen keinen reellen Bezug haben, sondern lediglich die korrekten Abhängigkeiten der Entitäten aufweisen.
Spezialfälle von Systemen stellen Systeme mit interaktiver Benutzungsmöglichkeit dar. Die Interaktion kann über eine Kommandozeile oder über eine grafische Benutzeroberfläche erfolgen. Bei solchen Systemen ist es nötig, die von dem System angebotene Interaktion mit dem Benutzer, zu testen.

3.2.1. Vorarbeit zu Systemtests

Zum Erstellen von Systemtests, benötigt der Tester die Anforderungen der Software. Die Anforderungen sollten schriftlich festgehalten sein, damit im Testfall darauf Bezug genommen werden kann. Zu jeder Anforderung wird ein Test mit Testfällen erstellt. Die Testfälle testen die Anforderung auf unterschiedliche Art und Weise. Je nach Anforderung können mehrere Testfälle benötigt werden. Ihre Anzahl ist abhängig von der Komplexität der Anforderung.
Ein Testfall wird in dieser Arbeit als vollständig bezeichnet, wenn folgendes dokumentiert ist:

- Version der Spezifikation auf den sich der Test bezieht.
- Vorbedingungen für den Testfall.
- Detaillierte Testschritte, die durchgeführt werden müssen.
- Erwartete Ergebnisse.

Tests beziehen sich auf eine Version der Spezifikation der Software, damit nachvollzogen werden kann, ob sich seit dem Erzeugen des Tests etwas an der Spezifikation geändert haben kann. Auftretende Fehler können auch als Ursache haben, dass Implementierung und Test zu verschiedenen Versionen der Spezifikation gehören. Für einen umfassenden Test muss jede Anforderung in mindestens einem Testfall vorkommen (Minimalforderung) [11]. Jede Anforderung sollte genau notiert sein, um herauszulesen, welche Testfälle es bedarf, um die Anforderung gut zu testen.
Ein Testfall beinhaltet die Vorbedingungen, Schritte die durchzuführen sind und erwartete Ergebnisse. Die Vorbedingungen umfassen den Zustand des Testsystems zu Beginn des Testfalls. Das kann zum Beispiel sein, dass eine Datenbank aufgesetzt sein muss, Testdaten eingespielt sein müssen oder eine Internetverbindung vorhanden sein muss. Schritte, die durchgeführt werden müssen, beinhalten eine genaue Anleitung wie das Programm bedient werden muss, um in die Ausgangssituation für den Test zu gelangen. Außerdem sollten Zwischenerwartungen dokumentiert sein, damit Fehler, die in Schritten vor dem eigentlichen Testfall auftauchen, identifiziert werden können. Die genauen Schritte, Eingaben und Erwartungen an das Verhalten der Software müssen dokumentiert sein. Sollten Ausgaben nicht in der Software

stattfinden, sollte notiert werden, wo die Ausgaben gefunden werden können.
Es müssen sinnvolle Eingaben ausgearbeitet werden, da nicht jede mögliche Kombination getestet werden kann. Die möglichen Kombinationen übersteigen bei den Wertebereichen der meisten technischen Datentypen die technischen Möglichkeiten. Sinnvolle Eingaben sind jene, bei deren Eingabe ein Fehler nicht unwahrscheinlich ist. Um sinnvolle Eingaben auszuarbeiten, müssen die *Äquivalenzklassen* betrachtet werden [11]. Eine Äquivalenzklasse besteht aus Eingaben, die auf die gleiche Art vom System behandelt werden. Wird also eine Eingabe korrekt verarbeitet, wird erwartet, dass die anderen Eingaben innerhalb der Äquivalenzklasse ebenso richtig verarbeitet werden. Als Beispiel wird die Division zweier Zahlen betrachtet, wobei Äquivalenzklassen für den Divisor gesucht werden. Betrachtet wird dabei nur der Zahlenraum der ganzen Zahlen. Äquivalenzklassen dafür sind:

positiveZahlen Alle Zahlen größer 0.

null Die Zahl 0, da durch sie nicht geteilt werden kann.

negativeZahlen Alle Zahlen kleiner 0, da sie einen Vorzeichenwechsel herbeiführen.

Ein Element einer Äquivalenzklasse auszutesten deckt bereits die Anforderung ab. Allerdings sind die Grenzfälle der Äquivalenzklassen fehleranfälliger, da sie häufig übersehen werden. Grenzfälle sind Fälle, an denen ein Übergang zwischen Äquivalenzklassen besteht oder aufgrund technischer Gegebenheiten Fehler verursacht werden können. Bezogen auf das obige Beispiel werden Grenzfälle der Äquivalenzklasse *negativeZahlen* betrachtet. Ein Grenzfall beim Übergang zwischen Äquivalenzklassen ist -1. Die nächste ganze Zahl ist 0 und eine eigene Äquivalenzklasse. Ein weiterer zunächst nicht offensichtlicher Grenzfall ist -2, da bei Division durch -1 nur das Vorzeichen geändert wird. Um die obere Grenze der negativen Zahlen abzutesten, sind die Zahlen -2 und -1 eine mögliche sinnvolle Wahl. Da es eine obere Grenze gibt, stellt sich die Frage nach der unteren Grenze. Die untere Grenze ist rein funktional nicht zu erkennen, da der Zahlenraum der ganzen Zahlen von $-\infty$ bis $+\infty$ reicht. Die Tests können daher nur von heutigen Darstellungen von Zahlen in Rechnern abgeleitet werden. Der Wertebereich einer Integer Zahl umfasst in der Sprache Java 2^{32}. Es ist demnach sinnvoll diese Grenze auszutesten. Ein weiterer Zahlentyp ist Long. Der Wertebereich von Long in Java ist 2^{64}. In diesem Fall muss in den Anforderungen genauer ersichtlich sein, welcher Wertebereich benutzt wird. Zusammengefasst sollten für sinnvolle Eingaben Äquivalenzklassen zu den Anforderungen gebildet und deren Grenzwerte betrachtet werden.
Da Tests regelmäßig ausgeführt werden, ist das Dokumentieren der Testfälle ein wichtiger Bestandteil der Erzeugung von Tests. Zur Dokumentation gehört neben der Eingabe das erwartete Ergebnis, da ansonsten das Ergebnis zunächst nachvollzogen werden muss. Außerdem gehören die Schritte, die vollzogen werden müssen, um das System in den erwarteten Zustand zu bringen, zur Dokumentation. Je detaillierter die Schritte dokumentiert sind, desto sicherer ist es, dass Tests immer gleich ausgeführt werden.

Datum	Software-version	Testfall erfolgreich	Fehler Nr.	Schritt 1	Schritt 2	...
22.3.2013	Beta 0.5	x	-	x	x	...
1.4.2013	Beta 0.6		10	x	-	...

Tabelle 1: Tabelle zur Dokumentation eines Testfalls

3.2.2. Ausführung von Testfällen

Nach dem Erzeugen der Testfälle können sie ausgeführt werden. Zunächst bietet sich die manuelle Ausführung an, um Fehler vor der Automatisierung zu finden. Das Ausführen eines Testfalls besteht aus dem Durchführen der einzelnen notierten Schritte und dem Festhalten der Ergebnisse [11]. Während des Ausführens sollten die vorgegebenen Schritte genauestens beachtet werden, da die Schritte beim Erzeugen mit Bedacht gewählt worden sind.
Die Ausführung des Testfalls sollte dokumentiert werden. Der Tester sollte protokollieren, wann die Ausführung des Testfalls stattfand, auf welcher Version getestet wurde und ob der Testfall erfolgreich durchgeführt werden konnte. Zusätzlich ist es hilfreich, die einzelnen Testschritte und deren Status zu vermerken. Die Dokumentation kann tabellarisch wie in Tab. 1 zu sehen erfolgen. In der ersten Spalte wird das Datum, an dem der Testfall ausgeführt wurde, vermerkt. In der zweiten Spalte folgt die zugehörige Programmversion und die nächsten Spalten dokumentieren das Ergebnis der Ausführung und geben aufgetretene Fehler an.
Da die Tests Regressionstests sind, werden sie öfter ausgeführt. Der Prozess der Ausführung und Dokumentation benötigt Zeit und ist aufwändig. Allerdings ist es möglich Testfälle automatisiert ausführen zu lassen. Dazu müssen die Testfälle programmatisch umgesetzt werden. Die umgesetzten Testfälle können entweder regelmäßig automatisch oder manuell gestartet werden. Das manuelle Starten kann auch vor bestimmten Ereignissen, wie der Fertigstellung einer Version oder einer Auslieferung erfolgen.

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	179	9%	382/3855	15%	146/924	2,018
sturesy	11	0%	0/246	0%	0/54	1,833
sturesy.export	9	74%	131/177	82%	61/74	4
sturesy.items	4	61%	116/188	49%	26/53	1,764
sturesy.plugin	4	0%	0/84	0%	0/36	2,812
sturesy.qgen	48	0%	0/872	0%	0/213	1,871
sturesy.qgen.gui	3	0%	0/191	0%	0/16	1,211
sturesy.settings	9	0%	0/147	0%	0/22	1,677
sturesy.settings.about	2	0%	0/39	0%	0/2	1,333
sturesy.settings.mainsettings	3	0%	0/36	0%	0/2	1,111
sturesy.settings.websettings	15	0%	0/260	0%	0/32	1,538
sturesy.util	12	18%	119/652	23%	53/224	3,87
sturesy.voting	34	3%	16/496	4%	6/124	1,857
sturesy.voting.gui	7	0%	0/144	0%	0/10	1,172
sturesy.voting.webvoting	4	0%	0/97	0%	0/14	1,714
sturesy.votinganalysis	11	0%	0/144	0%	0/42	1,84
sturesy.votinganalysis.gui	3	0%	0/82	0%	0/6	1,231

Abbildung 6: Testabdeckungsbericht des Quelltextes des Dozenten Klienten in Version 0.5 unter Berücksichtigung aller Klassen

4. Modultests in StuReSy

In diesem Abschnitt werden Modultests in dem Software-System StuReSy behandelt. Gegenstand dieser Arbeit ist zunächst das Arbeiten mit Testabdeckungsberichten und anschließend werden Methoden zum Erzeugen von Modultests evaluiert.
Im Ergebnis wird eine Bewertung des Arbeiten mit Testabdeckungsberichten mit dem Werzeug Cobertura [15] und die vorgestellten Methoden bewertet.

4.1. Testabdeckung messen mit Cobertura

Zur Messung der Testabdeckung wurde im Rahmen der Arbeit Cobertura [15] in StuReSy integriert. Cobertura ist eine freie Software und kann sowohl aus der Entwicklungsumgebung Eclipse mittels des Plugins eCobertura [16] heraus, als auch als ANT-Task ausgeführt werden. Die Vorteile beim Ausführen in Eclipse ist, dass die Zeilen im Quelltext farbig markiert werden. Dies gibt in der Entwicklungsumgebung die Möglichkeit, fehlende Stellen zu erkennen und entsprechende Testfälle nachzupflegen. Der Vorteil des Ant Tasks ist es, dass ein gesamter Bericht in HTML erzeugt wird. Zu Beginn dieser Arbeit wurde Cobertura in das ANT-Skript von StuReSy integriert. Alle in dieser Arbeit erzeugten Berichte wurden auf diese Art und Weise erzeugt. Zusätzlich zu den einzelnen Zeilen wird die Testabdeckung in einzelnen Paketen angegeben. Cobertura unterstützt die Überdeckungsmaße Zeilen- und Zweigüberdeckung.
In Abb. 6 ist ein Beispiel im Bezug auf StuReSy zu sehen. Es zeigt die Testabdeckung zu Beginn der Arbeit unter Berücksichtigung aller Klassen. Zu sehen ist in der ersten Zeile die Messung der gesamten Software. In den darauffolgenden Zeilen wird die Messung jedes einzelnen Paketes dargestellt. Daraus lässt sich ersehen, welche Pakete bereits ausreichend getestet sind und welche mehr Testbedarf aufweisen.

4.2. Sinnvolle Testabdeckungsberichte

Testabdeckungsberichte sollten mit Vorsicht gelesen werden, da die Interpretation von verschiedenen Informationen über die Messung abhängig ist. Zu einer sinnvollen Interpretation des in Abb. 6 dargestellten Berichts ist es nötig zu wissen, dass dort alle Klassen mit einbezogen wurden, die im Projekt vorhanden sind. Darin enthalten sind auch Klassen, die schwer testbar, nicht sinnvoll zu testen oder aber von dritten geschrieben und anschließend ohne Tests nach StuReSy kopiert wurden. Daher wurde im Vorhinein die Überlegung angestellt, welche Klassen nicht in die Messung mit einbezogen werden sollten. Daraus ergab sich folgende Liste:

Klassen die mit UI enden Klassen die mit UI enden enthalten keine Logik, sondern sind für das Erzeugen und Platzieren von grafischen Komponenten zuständig.

Die Klasse VerticalLayout VerticalLayout ist nicht von Wolf Posdorfer geschrieben und wird daher nicht getestet. Es gibt verschiedene freie Bibliotheken, die eine Klasse beinhalten, um Komponenten vertikal auszurichten. Diese beinhalten allerdings weitere Klassen, daher hat Wolf Posdorfer sich dafür entschieden die Klasse in das System zu übernehmen.

Die Klasse JFontChooser Wie VerticalLayout ist auch JFontChooser nicht von Wolf Posdorfer geschrieben. Zudem ist es eine grafische Komponente und damit schwer mit Modultests testbar. Anstelle, der in der System kopierten Klasse, wäre es sinnvoll eine freie Bibliothek zu benutzen. Ein Beispiel hierzu ist JFontChooser [17].

Daraus ergibt sich für die erste Messung der neue Bericht, der in Abb. 7 zu sehen ist. Die Differenz der einbezogenen Klassen liegt bei 19 Klassen. Die Abdeckung der Zeilen ist um 3% Punkte gestiegen und die Abdeckung der Verzweigungen um 2% Punkte.
Es existiert zwar nun ein aussagekräftigerer Bericht, allerdings gibt es weiterhin das Problem, dass Klassen nicht miteinbezogen werden sollten, die noch mit einbezogen werden. Das liegt an der Vermischung der Darstellung und der Logik der Oberfläche. Außerdem gibt es UI-Klassen, die nicht auf UI enden, und daher in die Messung mit einbezogen werden.

4.3. Testen nach Klassendiagrammen

Da nun die Möglichkeit besteht, den Stand des Testens zu messen, soll anhand eines Beispiels begonnen werden, die Strategie des Testens nach Klassendiagrammen in StuReSy zu demonstrieren. Dazu wurde als zu testende Klasse die Klasse `WebVotingHandler` aus dem Paket `sturesy.voting.webvoting` ausgewählt. In Abb. 8 ist ein vereinfachtes Klassendiagramm zu sehen. `WebVotingHandler` ist demnach von zwei Klassen und einem Interface abhängig. Daher müssen zunächst die beiden Klassen getestet werden. Positiv ist, dass diese keine weiteren Abhängigkeiten innerhalb von StuReSy aufweisen, und somit nur diese beiden getestet werden müssen. Nachdem diese getestet sind, bleibt das Interface. Während des Testens der

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	160	12%	382/3123	17%	146/819	2,155
sturesy	11	0%	0/246	0%	0/54	1,833
sturesy.export	9	74%	131/177	82%	61/74	4
sturesy.items	4	61%	116/188	49%	26/53	1,764
sturesy.plugin	4	0%	0/84	0%	0/36	2,812
sturesy.qgen	39	0%	0/528	0%	0/152	1,935
sturesy.settings	9	0%	0/147	0%	0/22	1,677
sturesy.settings.about	2	0%	0/39	0%	0/2	1,333
sturesy.settings.mainsettings	3	0%	0/36	0%	0/2	1,111
sturesy.settings.websettings	15	0%	0/260	0%	0/32	1,538
sturesy.util	12	18%	119/652	23%	53/224	3,87
sturesy.voting	33	3%	16/434	5%	6/102	1,836
sturesy.voting.gui	1	0%	0/9	0%	0/4	3
sturesy.voting.webvoting	4	0%	0/97	0%	0/14	1,714
sturesy.votinganalysis	11	0%	0/144	0%	0/42	1,84
sturesy.votinganalysis.gui	3	0%	0/82	0%	0/6	1,231

Abbildung 7: Testabdeckungsbericht des Quelltextes des Dozenten Klienten in Version 0.5 ohne die Klassen die auf UI enden und die Klassen VerticalLayout und JFontChooser

Klasse WebVotingHandler muss ein Objekt dieses Typs vorhanden sein, da ansonsten Fehler zur Laufzeit produziert werden. Daher ist zwingend ein Objekt nötig. Nun könnte eine konkrete Implementation des Interfaces benutzt werden. Es gibt zwei Klassen, die das Interface implementieren. Die eine ist die Klasse `VotingWindow` , die andere ist die Klasse `HittSettingsComponent`, welche nicht zum Quelltext des Klienten des Dozenten gehört. Das Problem bei der Klasse `VotingWindow` ist, dass sie sehr viele Abhängigkeiten besitzt, die auch zuvor getestet werden müssten (siehe Abb. 9). Das Klassendiagramm beinhaltet nicht die indirekten Abhängikeiten zu Klassen, die ebenfalls zuvor getestet werden müssten. Im nächsten Abschnitt wird eine Methode vorgestellt, mit der es möglich ist, ohne eine Implementation des Interfaces, die Klasse `WebVotingHandler` zu testen.

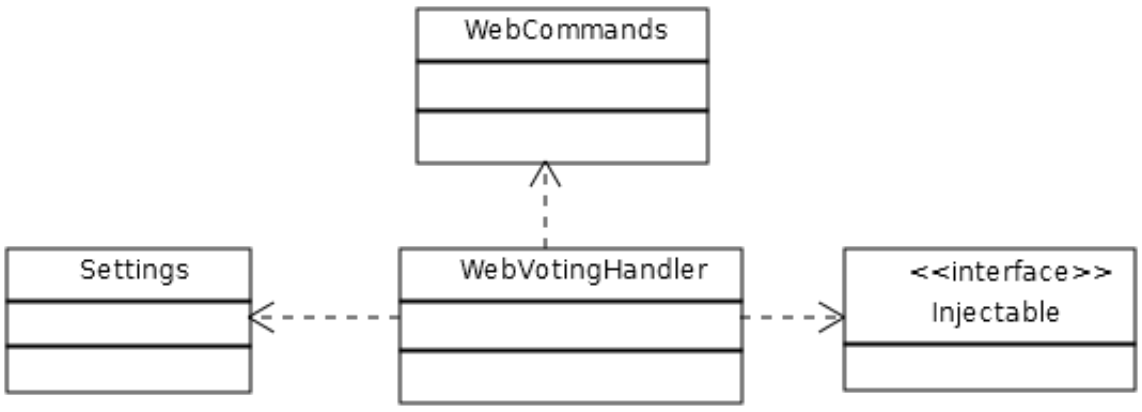


Abbildung 8: Beispiel: Modultests mit Hilfe von Klassendiagrammen in StuReSy

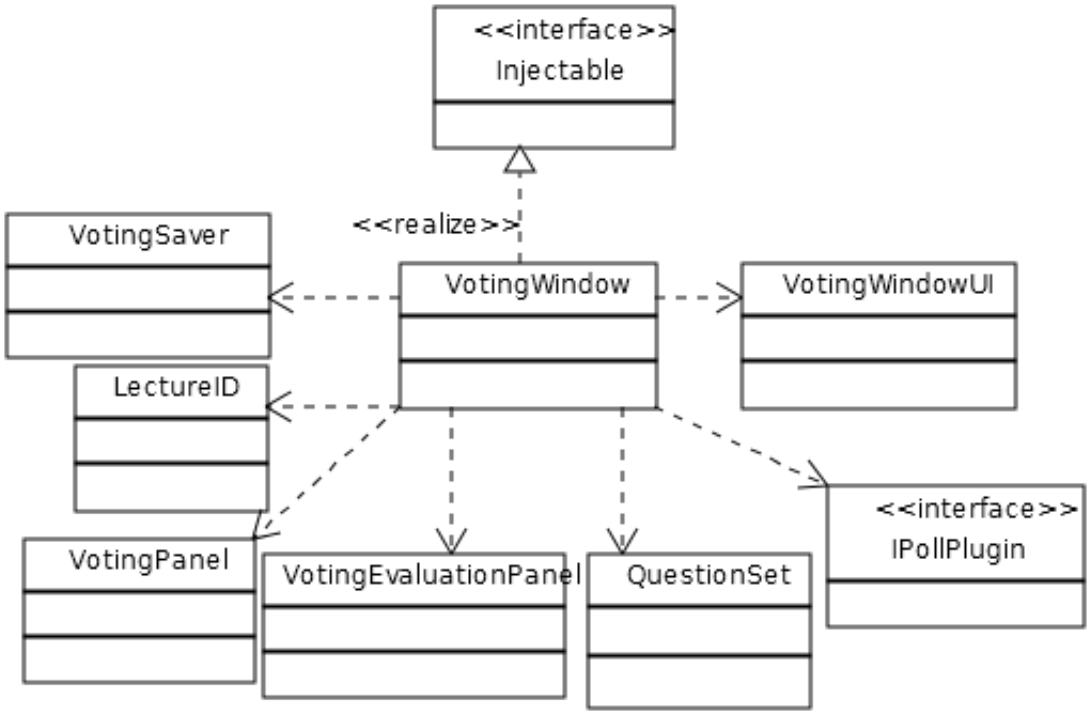


Abbildung 9: Implementation des Interfaces Injectable und seine direkten Abhängigkeiten.

4.4. Mock-Objekte

In der Softwareentwicklung werden Mock-Objekte als Platzhalter für reale Objekte benutzt. Im Folgenden werden sie lediglich als *Mocks* bezeichnet. Mocks implementieren die Schnittstelle einer Klasse oder eines Interfaces. Dabei werden die Operationen der Schnittstelle so implementiert, dass Testdaten zurückgegeben werden. Dies dient dem Zweck, dass das angenommene Verhalten eines realen Objektes nachgestellt wird. Mocks werden in Modultests eingesetzt, um ein Modul unabhängig von einem anderen zu testen.
Für das Interface `Injectable` ist eine Mock-Implementation in Listing 2 zu sehen. Dies ist ein sehr einfaches Beispiel.

Listing 2: Imlementation eines Mocks für die Klasse Injectable

```
1 public class InjectableMock implements Injectable {
2     @Override
3     public void injectVote(Vote vote) {
4
5     }
6 }
```

Um den Nutzen von Mocks zu verdeutlichen, wird daher ein weiteres Beispiel aus StuReSy herangezogen. In Abb. 10 ist ein Klassendiagramm für die Klasse `VotingSaver` mit Mock zu sehen. Die Klasse `VotingSaver` besitzt mehrere Methoden, doch in der Methode `VotingWriter.saveToFile(VotingSaver saver, String filename`

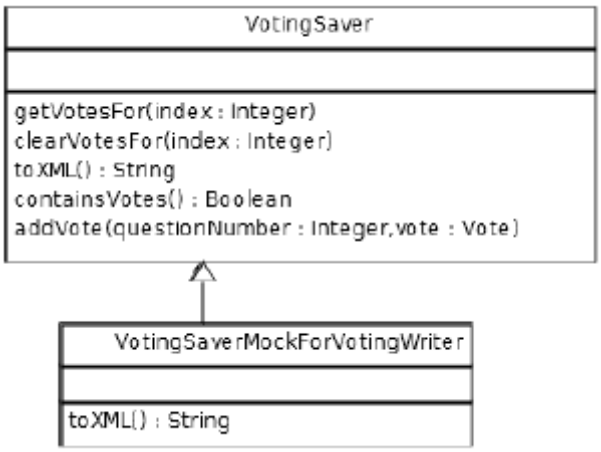


Abbildung 10: Beispiel eines Mocks in StuReSy

) wird nur die Methode `toXML()` benutzt. Daher überschreibt der Mock nur diese Methode. Für den Test wird von der Mock-Implementation eine feste Zeichenkette zurückgegeben, damit der Inhalt der geschriebenen Datei unabhängig von der korrekten Aufbereitung des Inhalts eines Objekts der Klasse `VotingSaver` ist.

Beim Erstellen von Modultests für StuReSy sind insbesondere drei Probleme aufgetreten.

1. Das Benutzen von Mocks ist stark abhängig von den Schnittstellen der zu testenden Module, da es möglich sein muss, Mocks in einem Objekt einzusetzen.

2. Das Erstellen von Mocks bedeutet, dass zusätzliche Klassen erzeugt werden. Dadurch entstehen Klassen, die für den Produktiveinsatz des Systems nicht relevant sind.

3. Es kommt vor, dass geprüft werden soll, ob das zu testende Modul Methoden eines Mocks aufruft. Das ist mit diesen Mocks nur dadurch möglich, dass JUnit-Anweisungen in den Mock-Implementationen verwendet wird, wodurch Mock und Testfall stark aneinander gekoppelt sind.

Das erste Problem kann am besten zu Beginn eines Projekts gelöst werden, indem beim Erzeugen von Schnittstellen der Einsatz von Mocks bedacht wird. Bei der Entwicklung von StuReSy lag der Fokus nicht auf der Testbarkeit der Module. Daher müssten zum Einsatz von Mocks die Schnittstellen der Klassen angepasst werden. Allerdings sollten die Tests vor den Veränderungen geschrieben werden, weswegen der Einsatz von Mocks beschränkt möglich ist.
Das zweite und dritte Problem sind generelle Probleme beim Einsatz von Mocks, daher gibt es zahlreiche Lösungen in Form von Frameworks. Eins dieser Frameworks wird im Folgenden vorgestellt.

4.5. Mock-Objekte mit Mockito

Mockito [18] ist ein Mock-Framework. Es bietet es eine einfache Handhabung von Mocks. Die grundlegenden Funktionen sind:

- Erzeugen von Mocks.
- Konfigurieren der Rückgabewerte von Methoden.
- Verifizieren von Aufrufen an Mocks unter Beachtung der Eingabeparameter.

Im Folgenden werden diese Funktionen anhand des bekannten Beispiel des `VotingSaver` erläutert.

4.5.1. Erzeugen von Mocks

Mockito bietet zwei einfache Varianten an, Mocks zu Klassen zu erzeugen.

- Annotation @Mock an Exemplarvariablen (s.h. Listing 3, Zeile 2).
- Die Klassenmethode `<T> T mock(java.lang.Class<T> classToMock)` in der Klasse Mockito (s.h. Listing 3, Zeile 7).

Listing 3: Erzeugen von Mocks.

```
1 public class VotingSaverMockBeispiel {
2   @Mock
3   private VotingSaver votingSaver1;
4
5   @Test
6   public void writeVotingToFile() {
7     VotingSaver votingSaver2 = Mockito.mock(VotingSaver.class);
8   }
9 }
```

Diese zwei Varianten lösen das Problem, dass extra Klassen erzeugt werden müssen, da das Erzeugen des Mocks in dem Framework gekapselt ist. Nach dem Erzeugen sind alle Rückgabewerte der Methoden des erzeugten Objekts auf einen Standardwert gesetzt. Diese richten sich nach dem Datentyp des Rückgabewerts der jeweiligen Methode (siehe Tab. 2).

4.5.2. Konfigurieren der Rückgabewerte

Damit die Methoden andere Werte zurückgeben, muss der Mock konfiguriert werden. Hierzu bietet Mockito verschiedene Möglichkeiten. Im Folgenden wird eine Übersicht über die konfigurierbaren Methoden gegeben.

- Die Methode besitzt keinen formalen Parameter.
- Die Methode besitzt einen formalen Parameter. Die Konfiguration soll nur benutzt werden, wenn der Parameter einen bestimmten Wert besitzt.

Datentyp	Standardrückgabewert
boolean	false
int	0
long	0
short	0
byte	0
double	0.0
float	0.0
char	Leerzeichen (ASCII 0)
Object	null

Tabelle 2: Standard Rückgabewerte von Mockito-Mocks

- Die Methode besitzt einen formalen Parameter. Die Konfiguration soll unabhängig vom Wert des Parameters immer benutzt werden.

Weiterhin ist es möglich, bei gleichen Aufrufen einer Methode verschiedene Werte zurückzugeben.
Es ist also möglich Testdaten nach Belieben zurückzugeben, um den Kontrollfluss des getesten Moduls zu bestimmen.

4.5.3. Verifikation der Methodenaufrufe eines Mocks

Damit sichergestellt werden kann, dass die richtigen Methoden von Mock-Objekten im Programmfluss aufgerufen werden, können Methodenaufrufe an Mocks erwartet werden. Wenn der erwartete Methodenaufruf nicht stattfindet, schlägt der Test fehl. Wie beim Konfigurieren von Rückgabewerten, bietet Mockito auch zum Verifizieren von Methodenaufrufen verschiedene Möglichkeiten.

- Es kann angegeben werden, wie oft der Aufruf der Methode erwartet wird.
- Es kann angegeben werden, wie oft der Aufruf mit bestimmten Parametern erwartet wird.
- Ein Methodenaufruf kann für bestimmte Parameterwerte oder beliebige Parameter verifiziert werden.

Somit löst Mockito auch das dritte der Probleme von Mock-Objekten.

4.5.4. Beispiel aus StuReSy

Im Abschnitt 5.4 wurde die Klasse `VotingSaver` in StuReSy betrachtet. In Listing 4 ist die Benutzung von Mockito zu sehen. In Zeile 3 wird ein Mock-Objekt der Klasse `VotingSaver` erzeugt und eine Referenz in der lokalen Variablen `votingSaver` gespeichert. Anschließend wird in Zeile 4 die Methode `toXML()` des Mock-Objekts konfiguriert, damit bei Aufruf der Methode die Zeichenkette „<xml></xml>“ zurückgegeben wird. Hierzu wird der Methode `when()` der zu konfigurierende Methodenaufruf und der Methode `thenReturn()` die zurückzugebende Zeichenkette übergeben. In Zeile 5 wird die Methode `saveToFile()` des Testobjekts VotingWriter

Listing 4: Test mit Mockito. Ein Voting wird in eine XML-Datei geschrieben.

```
1 @Test
2 public void writeVotingToFile() {
3   VotingSaver votingSaver = Mockito.mock(VotingSaver.class);
4   Mockito.when(votingSaver.toXML()).thenReturn("<xml></xml>");
5   VotingWriter.saveToFile(votingSaver, "anyFile.xml");
6   Mockito.verify(votingSaver, Mockito.times(1)).toXML();
7 }
```

aufgerufen und das Mock-Objekt als Parameter übergeben. Der zweite Parameter ist eine Datei in die das Objekt gespeichert werden soll. Abschließend wird in Zeile 6 überprüft, ob die Methode `toXML()` an dem Mock-Objekt genau einmal aufgerufen wurde.
Da Mockito die festgestellten Probleme löst und die Erzeugung, Konfiguration und Verifikation von Mock-Objekten und deren Methoden gut lesbar sind, wird Mockito im Rahmen der Arbeit zur Unterstützung beim Schreiben weiterer Modultestes in StuReSy benutzt.

4.6. Ergebnis

Die Anzahl der Tests wurde während dieser Arbeit von mir auf 289 erweitert.
Durch die Unterstützung von Mockito, mussten keine extra Klassen für Mock-Objekte eingeführt werden. Durch eine umfassende, von den Entwicklern bereitgestellte, Dokumentation, ist die benötigte Zeit zur Einarbeitung in Mockito gering. Dies, und die intuitive Schreibweise zur Erzeugung, Konfiguration und Verifikation von Mocks und ihren Methoden ermöglichte mir schnell einen sicheren Umgang mit dem Framework.
Die Testabdeckung wurde von mir in zwei Berichte getrennt. Es wurde ein Bericht unter Einbezug aller Klassen erstellt (s.h. Abb. 11) und ein weiterer Bericht, der die Klassen mit der Endung UI und die Klassen `VerticalLayout` und `JFontChooser` ignoriert (s.h. Abb. 12). Dies ermöglichte während dem Schreiben der Tests einen besseren Überblick zu bekommen, für welche Klassen weitere Tests geschrieben werden können. In Tab. 4 sind die zu Beginn und zum Ende erzeugten Testabdeckungsberichte zusammengefasst. Im Vergleich zu den ersten Messungen ist die Testabdeckung in beiden Berichten gestiegen. Auffällig ist die Abweichung der nicht einbezogenen Klassen in den Berichten mit den ignorierten Klassen. Die Anzahl der ignorierten Klassen ist um 31 Klassen gestiegen. Dies lässt sich darauf zurückführen, das beim Schreiben der Modultests die Trennung von Logik und Oberfläche von mir konsequent umgesetzt wurde, damit die Logik der Klassen getestet werden konnte.
Die Differenz der Zeilenüberdeckung zwischen den getrennten Berichten ist im Vergleich von Revision 130 zu Revision 299 um 17% Punkte gestiegen. Dies erklärt sich ebenfalls durch die konsquente Trennung von Logik und Oberfläche, da Oberflächenklassen sehr viele Zeilen zur Erzeugung und Positionierung der grafischen Komponenten benötigen.
Die Differenz der Zweigüberdeckung zwischen den getrennten Berichten ist um 11% Punkte gestiegen. Dies ist darauf zurückzuführen, dass Verzweigungen, die bisher in UI-Klassen enthalten waren, nun in Logik-Klassen zu finden sind.

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 3: Mögliche Belegungen der Variablen a und b für Listing 5

Die Unterstützung durch Cobertura ermöglichte es mir, während der Arbeit einen Überblick über die noch zu testenden Klassen zu bekommen. Für diese Arbeit war vorallem die Zweigüberdeckung sehr hilfreich, da sie eine zuverlässigere Aussage über noch zu testende Quelltextabschnitte gibt. Negativ aufgefallen ist, dass eine Testabdeckung von 100% theoretisch erreicht, praktisch aber von Cobertura nicht erkannt wurde. In Listing 5 ist eine Methode zu sehen, für die eine Abdeckung von 100% durch die Wahrheitstabelle in Tab. 3 theoretisch erreichbar sein sollte. Praktisch kann allerdings nur eine Testabdeckung von 75% erreicht werden. Dies liegt daran, dass Java, sobald die erste Variable `true` ist, die Bedingung zu `true` auswertet und die zweite Variable nicht mit einbezogen wird. Die ersten beiden Zeilen der Wahrheitstabelle decken in Cobertura nur eine der vier möglichen Belegungen ab, wodurch insgesamt nur drei von vier Belegungen als getestet angezeigt werden. Dieses Problem wird von Cobertura nicht behandelt und führt zu unübersichtlicheren Testabdeckungsberichten. Die Benutzung von Cobertura erwies sich, trotz dieses Mangels, als sehr hilfreich.

Listing 5: Einfaches Beispiel für das eine 100%ige Testabdeckung in Cobertura nicht erreicht werden kann, obwohl eine 100%ige Testabdeckung erreichbar sein sollte.

```
1 public void einfacheVerzweigung(boolean a, boolean b) {
2   if(a || b) {
3     System.out.println("true");
4   }
5   else {
6     System.out.println("false");
7   }
8 }
```

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	199	36%	1362/3760	47%	376/790	1,649
sturesy	16	36%	107/292	20%	16/80	1,917
sturesy.export	9	90%	156/172	93%	60/64	2,385
sturesy.export.jaxb	1	100%	17/17	100%	2/2	1,286
sturesy.filter.file	2	100%	8/8	100%	14/14	3
sturesy.filter.filechooser	3	100%	17/17	95%	19/20	1,857
sturesy.items	5	98%	185/187	100%	47/47	1,621
sturesy.plugin	5	20%	22/108	14%	6/42	2,684
sturesy.qgen	40	38%	304/796	46%	87/189	1,775
sturesy.qgen.gui	8	0%	0/309	0%	0/38	1,436
sturesy.services	9	93%	123/132	95%	38/40	1,595
sturesy.settings	9	14%	21/148	13%	3/22	1,677
sturesy.settings.about	2	0%	0/40	0%	0/2	1,333
sturesy.settings.mainsettings	4	0%	0/59	0%	0/4	1,25
sturesy.settings.websettings	15	0%	0/263	0%	0/32	1,538
sturesy.test.plugin.testmaterial	1	0%	0/1	N/A	N/A	0
sturesy.tools	3	100%	49/49	100%	10/10	0
sturesy.util	8	50%	68/136	61%	11/18	1,567
sturesy.voting	35	26%	129/496	30%	30/100	1,635
sturesy.voting.gui	7	0%	0/164	0%	0/10	1,122
sturesy.voting.webvoting	4	11%	11/100	0%	0/16	1,727
sturesy.votinganalysis	11	71%	145/204	82%	33/40	1,4
sturesy.votinganalysis.gui	2	0%	0/62	N/A	N/A	1

Abbildung 11: Testabdeckungsbericht des Quelltextes des Dozentenklienten in Revision 299 unter Einbeziehung aller Klassen

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	149	53%	1336/2489	58%	376/643	1,749
sturesy	14	32%	81/253	20%	16/80	2,073
sturesy.export	9	90%	156/172	93%	60/64	2,385
sturesy.export.jaxb	1	100%	17/17	100%	2/2	1,286
sturesy.filter.file	2	100%	8/8	100%	14/14	3
sturesy.filter.filechooser	3	100%	17/17	95%	19/20	1,857
sturesy.items	5	98%	185/187	100%	47/47	1,621
sturesy.plugin	5	20%	22/108	14%	6/42	2,684
sturesy.qgen	31	67%	304/452	67%	87/128	1,793
sturesy.services	9	93%	123/132	95%	38/40	1,595
sturesy.settings	3	28%	21/74	18%	3/16	1,947
sturesy.settings.about	2	0%	0/40	0%	0/2	1,333
sturesy.settings.mainsettings	1	0%	0/15	N/A	N/A	1
sturesy.settings.websettings	1	0%	0/19	N/A	N/A	1
sturesy.test.plugin.testmaterial	1	0%	0/1	N/A	N/A	0
sturesy.tools	3	100%	49/49	100%	10/10	0
sturesy.util	8	50%	68/136	61%	11/18	1,567
sturesy.voting	35	26%	129/496	30%	30/100	1,635
sturesy.voting.gui	1	0%	0/9	0%	0/4	3
sturesy.voting.webvoting	4	11%	11/100	0%	0/16	1,727
sturesy.votinganalysis	11	71%	145/204	82%	33/40	1,4

Abbildung 12: Testabdeckungsbericht des Quelltextes des Dozentenklienten in Revision 299 mit allen Klassen außer: UI-Klassen, VerticalLayout und JFontChooser

Revision	Einbezogene Klassen	Nicht einbezogene Klassen	Tests	Zeilenüberdeckung	Zweigüberdeckung
130	Alle	0	11	9%	15%
130	Alle außer: UI-Klassen, VerticalLayout, JFontChooser	19	11	12%	17%
299	Alle	0	289	36%	47%
299	Alle außer: UI-Klassen, VerticalLayout, JFontChooser	50	289	53%	58%

Tabelle 4: Ergebnisse der Messung der Testabdeckung in Revision 130 und Revision 299

5. Manuelle Systemtests für StuReSy

In diesem Abschnitt wird systematisch ein Testfall für StuReSy erzeugt, mit dem Ziel eine vollständige Testfallbeschreibung als Fallbeispiel für die Automatisierung von Systemtests mit grafischer Benutzeroberfläche zu erhalten. Als Fallbeispiel wurde die Liste und die zugehörigen Buttons zum Verwalten von Fragen innerhalb eines Fragensatzes im Frageneditor (s.h. Abb. 13) gewählt. Zunächst werden die Anforderungen aufgestellt. Auf Grundlage der Anforderungen wird dann ein Testfall der später automatisiert werden soll geschrieben und anschließend das Ergebnis einer manuellen Ausführung beschrieben.

5.1. Anforderungen

Zur systematischen Erzeugung von Testfällen, müssen die Anforderungen an das System bekannt sein. Da es für StuReSy keine formale Spezifikation gibt, wurden die Anforderungen an das Verwalten von Fragesätzen im Frageneditor aus der Software abgeleitet. Dies beinhaltet das Hinzufügen und Entfernen von Fragen mittels der Buttons „+“ und „−“ zu dem Fragesatz, sowie Änderung der Sortierung der Fragen mittels der Buttons „Hoch“ und „Runter“. Die Fragen werden in einer Liste durch einen dynamischen Bezeichner repräsentiert, der sich aus der Position der Frage im Fragensatz und dem Fragentext der Frage zusammensetzt.
Die Anforderungen, die sich durch Ausprobieren ergaben, wurden durch meine persönliche Erwartungen ergänzt. Dies betrifft die Zustände der folgenden Buttons: −-Button, „Hoch-Button“ und „Runter“-Button. Deren Benutzung ergeben in der Benutzung in bestimmten Zuständen der Liste keinen Sinn und liefern kein erkennbares Resultat.
Im Folgenden werden die aufgestellten Anforderungen für das Verhalten durch Benutzen der Buttons zur Veränderung der Liste aufgelistet.

Anforderungen an die Liste zum Bearbeiten des Fragesatzes:

A1: Hinzufügen von Fragen Es soll einen Button zum Hinzufügen einer neuen Frage geben. Dieser Button soll immer benutzbar sein. Die neue Frage soll nicht selektiert in der Liste erscheinen, in der die anderen Fragen stehen. Der dynamische Bezeichner der Frage soll beim Hinzufügen eine fortlaufende Nummerierung beginnend bei F1 sein. Bei Hinzufügen eines Fragetextes soll der Fragetext in Klammern angehängt werden.
Beispiel:
Fragetext: Test
Frage in der Liste: F1 (Test)

A2: Entfernen von Fragen Es soll einen Button zum Entfernen einer Frage geben. Dieser soll benutzbar sein, wenn eine Frage in der Liste selektiert ist. Bei Benutzung soll die selektierte Frage aus der Liste der Fragen entfernt werden. Nach dem Entfernen soll keine Frage mehr selektiert sein.

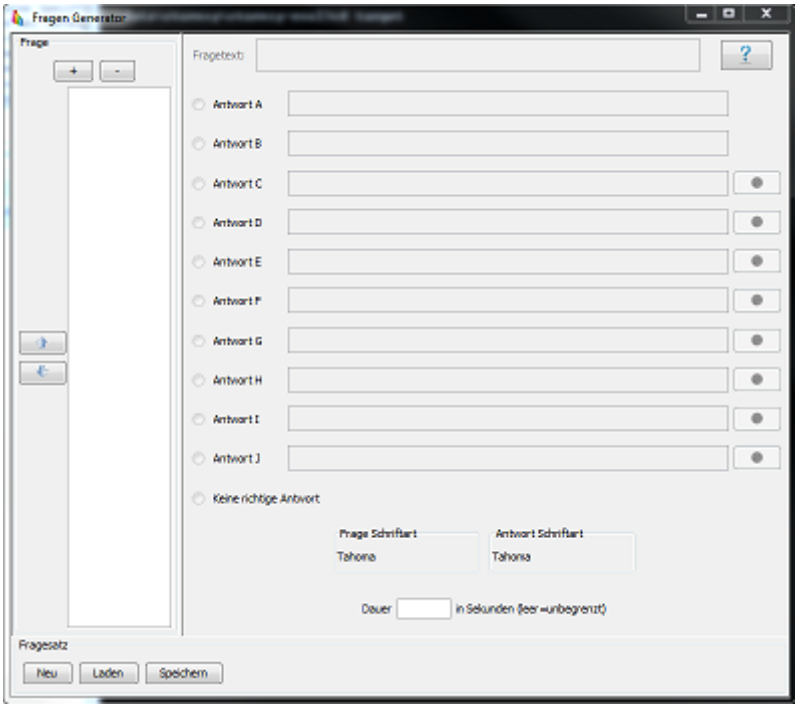


Abbildung 13: Der Frageneditor, zum Editieren eines Fragesatzes. Liste links: Hinzufügen von Fragen. Textfelder rechts: Editieren einer Frage. Schaltflächen unten: Werkzeuge zum Erstellen/Speichern/Laden eines Fragesatzes

Gegebenenfalls sollen die Nummerierungen der nachfolgenden Listeneinträge angepasst werden.

A3: Frage im Fragesatz nach oben verschieben Es soll einen Button geben, um eine Frage einen Platz nach oben in der Liste zu verschieben. Dieser Button soll nur dann aktiv sein, wenn eine Frage selektiert ist und sie nicht bereits ganz oben in der Liste steht. Der dynamische Bezeichner soll beim Verschieben angepasst werden und die verschobene Frage selektiert bleiben. Die Frage an oberster Stelle in der Liste soll nicht weiter nach oben verschoben werden können.

A4: Frage im Fragesatz nach unten verschieben Es soll einen Button geben, um eine Frage einen Platz nach unten in der Liste zu verschieben. Dieser Button soll nur dann aktiv sein, wenn eine Frage selektiert ist und sie nicht bereits ganz unten in der Liste steht. Der dynamische Bezeichner soll beim Verschieben angepasst werden und die verschobene Frage selektiert bleiben. Die Frage an unterster Stelle soll nicht weiter nach unten verschoben werden können.

5.2. Testfälle und ihre Testschritte

Nach der Beschreibung der Anforderungen kann ein Test für die Liste geschrieben werden. Ein Test wird in mehrere Testfälle unterteilt. Dazu werden die verschiedenen Anforderungen betrachtet. Zunächst stellt jede Anforderung einen Testfall dar. Daraufhin werden Testfälle zusammengefasst, die getrennt voneinander nicht gut

testbar sind.
Daraus ergeben sich für die Anforderungen A1 und A2 der Testfall 1 in Tab. 5 und für die Anforderungen A3 und A3 der Testfall 2 in Tab. 6.
Die beiden Tabellen geben jeweils einen vollständigen Testfall an. In der ersten Spalte der Tabellen werden die Testschritte durchnummeriert. Die Nummerierung ist eine Hilfe für die Orientierung bei der Ausführung und die Referenzierung bei auftretenden Fehlern. Die zweite Spalte der Tabellen gibt an, was in diesem Testschritt zu erledigen ist. Je detaillierter der Testschritt beschrieben wird, desto genauer kann der ausführende Tester die Geschehnisse nachvollziehen. In der dritten Spalte steht das erwartete Verhalten und in welchem Zustand sich die Software befindet.
Die Testfälle wurden mit Hilfe der folgenden Äquivalenzklassen erstellt.
Die Äquivalenzklassen für A1 und A2 sind folgende:

- Die Liste wird um eine Frage erweitert.
- Es wird aus einer bestehenden Liste eine Frage gelöscht.
- Es gibt keine Frage, die gelöscht werden kann.

Daraus ergibt sich der Grenzfall, dass wenn die letzte Frage gelöscht wurde in die dritte Äquivalenzklasse übergegangen wird.
Für A3 und A4 ergeben sich folgende Äquivalenzklassen:

- Es gibt keine Fragen, die verschoben werden können.
- Es gibt genau eine Frage.
- Es gibt Fragen, zwischen denen eine Frage verschoben werden kann.

Hieraus ergeben sich die Grenzfälle, wenn eine Frage das erste Element der Liste und wenn das Element das letzte Element der Liste ist.
Mit allen drei Testfällen ist eine volle Anforderungsabdeckung dieser Teilkomponenten gegeben, da jede Anforderung mindestens einmal durchlaufen wird.

5.3. Testfall Durchführung

Die Ausführung von Testfall 1 wurde zur Vorbereitung der Automatisierung durchgeführt und dokumentiert. Zur Dokumentation wird Tab. 7 benutzt. Allerdings wird statt der Fehlernummer der Fehler direkt in der Spalte dokumentiert, da die Fehler nicht in einem System erfasst werden. Der Testfall wurde auf der Revision 577 vom 3. April 2013 ausgeführt. Wie zu erkennen ist, weichen die aufgestellten Anforderungen von der Implementation ab. Dies wurde dem Entwickler Wolf Posdorfer mitgeteilt und er bestätigte die Einarbeitung zu einem späteren Zeitpunkt.
Der Testfall 1 wird im folgenden Abschnitt benutzt, um die Automatisierung mit den Frameworks Fit und FEST zu evaluieren.

Testschritt	**Durchzuführen**	**Erwarteter Zustand/Erwartetes Verhalten**
1	Öffne den Dozentenklienten und öffne den Frageeditor	Frageeditor ist offen, + Button ist benutzbar, − Button ist nicht benutzbar.
2	Drücke den +-Button	Eine neue Frage erscheint in der Liste und hat den Bezeichenr F1. Die Frage ist nicht selektiert. Die Buttons verbleiben in ihren Zuständen. + ist benutzbar, − ist nicht benutzbar.
3	Drücke den +-Button.	Eine neue Frage erscheint in der Liste und hat den Bezeichner F2. Keine Frage ist selektiert. Die Buttons verbleiben in ihren Zuständen. + ist benutzbar, − ist nicht benutzbar.
4	Selektiere F1.	Die Buttons + und − sind nun beide benutzbar.
5	Drücke −-Button.	Es existiert nur noch eine Frage mit dem Bezeichner F1 und keine Frage ist selektiert. − Button ist nicht benutzbar.
6	Drücke −-Button.	Es existiert keine Frage mehr in der Liste und der Button − ist nicht benutzbar.

Tabelle 5: Testfall 1: Deckt die Anforderung A1 und A2 ab.

Testschritt	Durchzuführen	Erwarteter Zustand/Erwartetes Verhalten
1	Öffne den Dozentenklienten und öffne den Frageeditor	Frageeditor ist offen. „Hoch“-Button ist nicht benutzbar. „Runter“-Button ist nicht benutzbar.
2	Füge eine Frage über den +-Button hinzu und selektiere die Frage.	„Hoch“-Button ist nicht benutzbar. „Runter“-Button ist nicht benutzbar.
3	Füge eine weitere Frage hinzu. Selektiere F1.	„Runter“-Button ist benutzbar, „hoch“-Button ist nicht benutzbar.
4	Selektiere F2.	„Hoch“-Button ist benutzbar, „runter“-Button ist nicht benutzbar.
5	Gebe Fragetext „Test“ ein und selektiere F1.	F2 heißt nun F2 (Test), „runter“-Button ist benutzbar, „hoch“-Button ist nicht benutzbar.
6	Benutze „runter“-Button.	An Position 1 in der Liste ist nun F1 (Test). An Position 2 ist F2. Selektiert ist F2.
7	Benutze „hoch“-Button	An Position 1 steht wieder F1 und an Position 2 steht F2 (Test). F1 ist selektiert.
8	Füge eine weitere Frage hinzu und selektiere F2 (Test).	Buttons „hoch und „runter sind beide benutzbar.

Tabelle 6: Testfall 2: Deckt die Anforderung A3 und A4 ab.

Datum	Version	Status	Fehler	1	2	3	4	5	6
04.04.2013	Beta 0.5.2 Revision 577	Fehler	−-Button ist benutzbar, obwohl keine Frage selektiert ist	F	F	F	T	F	F

Tabelle 7: Dokumentation der Ausführung von Testfall 1 auf zwei verschiedenen Versionen. T steht für den Wahrheitswert True und gibt ein positives Ergebnis an. F steht für den Wahrheitswert false und gibt ein negatives Ergebnis an.

6. Automatisierte Systemtests für StuReSy

Aufgrund der Problematik, dass Tests als Regressionstests oft ausgeführt und dennoch häufig aus Zeitmangel vernachlässigt werden, bietet es sich an, auch die Systemtests zu automatisieren. Das Automatisieren von Systemtests bietet die Möglichkeit, die manuell ausgeführten Testfälle immer wieder auszuführen, ohne darauf Zeit aufzuwänden. Daher soll im folgenden Abschnitt eine Möglichkeit evaluiert werden, mit Hilfe der Frameworks FEST und Fit Systemtests zu automatisieren. Beiden Frameworks benutzen *Fixture*-Klassen, allerdings in verschiedenen Kontexten. Um zwischen diesen beiden Frameworks besser zu differenzieren, werden die Fixtures von FEST als FEST-Fixtures bezeichnet, während mit Fixture immer Fit-Fixtures gemeint sind. Bei Klassennamen kann die Differenzierung nur anhand der Kontexte festgestellt werden. In der Regel beziehen sich FEST-Fixtures auf Oberflächenkomponenten.

6.1. FEST-Swing

FEST [19] besteht aus mehreren Komponenten, zum Testen von Software. Eine dieser Komponenten ist FEST-Swing, ein Werkzeug zum Testen von Oberflächen. Mit Hilfe des Frameworks lassen sich Benutzereingaben über die grafische Oberfläche simulieren. Um die Benutzeroberfläche zu bedienen, müssen die grafischen Komponenten gefunden und die Bedienung simuliert werden. Zum Finden der Komponenten wird ein *ComponentLookup* durchgeführt und die gefundene Komponente durch ein FEST-Fixture repräsentiert. FEST-Fixtures sind Klassen zum Bedienen einer Swing-Komponenten. Swing-Komponenten lassen sich in zwei Typen unterteilen. Dies sind *Komponenten* und *Container*. Container beinhalten Komponenten oder andere Container und platzieren diese. Komponenten stellen die Interaktion mit dem Benutzer bereit. In den entsprechenden FEST-Fixtures können in Container-Komponenten gesucht und auf Komponenten Benutzeraktionen ausgeführt werden.

6.1.1. Component Lookup

Eine wichtige Rolle in FEST spielt der ComponentLookup. Dieser ist dafür zuständig Komponenten in Containern zu finden. Gesucht wird in der Komponentenhierarchie eines `ContainerFixture`-Objekts. Die Komponentenhierarchie ist nach dem Kompositum-Entwurfsmuster [20] aufgebaut. Die Suchstrategie kann aus drei Suchstrategien ausgewählt werden.

Name Es wird nach dem —der Komponenten programmatisch zugeordnetem— Namen gesucht.

- Vorteile
  - Der Name ist im Quelltext festgelegt. Da er losgelöst von der Programmlogik ist, wird er durch Änderungen an der Logik der Software nicht verändert.
  - Nicht abhängig von fachlichen Bezeichnungen, daher ist es leicht eine Identität herzustellen.

- Nachteile
  - Es können doppelte Namen vorkommen. Das führt zu Fehlern beim ComponentLookup.
  - Die Namen sind bei der Entwicklung nicht verpflichtend zu setzen. Werden aber für den ComponentLookup benötigt.
  - Die Namen müssen dem Testentwickler bekannt sein.

Text Es wird nach dem Text auf der Komponenten gesucht.

- Vorteile
  - Der Testentwickler kann anhand der Oberfläche den Text ablesen.
- Nachteile
  - Es können doppelte Texte vorkommen. Das führt zu Fehlern beim ComponentLookup.
  - Texte sollten nicht abgeändert werden, damit sie für die Automatisierung einzigartig sind, da sie Teil der Kommunikation mit dem Benutzer sind.
  - Nicht alle Komponenten haben Texte

Typ Es wird eine Komponente eines bestimmten Typs gesucht. Zum Beispiel: `JTextField` oder `JButton`.

- Vorteile
  - Es müssen weder Namen noch Texte bekannt sein.
- Nachteile
  - Es kommen sehr häufig Komponenten in einer Komponentenhierarchie mehrfach vor. Das führt zu Fehlern beim ComponentLookup.

Da der Name der Komponenten als einziges flexibel gestaltbar ist, wird in dieser Arbeit der Name zum Suchen von Komponenten benutzt.
Ob eine Komponente gefunden werden kann, hängt zum einen von der Suchstrategie ab, aber auch von dem Sichtbarkeitsbereich, in dem gesucht werden soll. Der benutzte Sichtbarkeitsbereich kann an `ContainerFixture`-Klassen gesetzt werden. Zur Auswahl stehen drei Sichtbarkeitsbereiche.

VISIBLE Alle sichtbaren Komponenten können gefunden werden.

DEFAULT Alle sichtbaren Komponenten können gefunden werden und zusätzlich werden alle JMenuItem als sichtbar behandelt.

ALL Alle Komponenten in der Komponentenhierarchie können gefunden werden.

In dieser Arbeit wird von dem DEFAULT-Sichtbarkeitsbereich ausgegangen.

6.1.2. Beispiel in StuReSy

Die Umsetzung von Testfall 1 aus Kapitel 5 (s.h. Tab. 5) ist in Anhang C zu finden. In der Umsetzung gibt es einen Testfall. Zu Beginn des Testfalls wird in Zeile 3 der Dozentenklient gestartet und anschließend in Zeile 4, das Fenster über ein Objekt der Klasse `FrameFixture` bedienbar gemacht. In den folgenden Zeilen werden die Testschritte aufgerufen. Die Testschritte sind in ihrer Implementation sehr ähnlich. Die zu benutzenden Komponenten während des Testschrittes werden gesucht und anschließend über die jeweilige `Fixture`-Klasse benutzt.
Anhand des Beispiels ist zu sehen, dass sehr viel technisches Wissen nötig ist und häufig die gleichen Methoden benutzt werden. Desweiteren ist das Entwickeln des Tests zwar nicht komplex, aber aufwändig. Auch muss auf die Lesbarkeit geachtet werden. In dem Beispiel wurde bessere Lesbarkeit durch das Auslagern in Methoden zum Prüfen von Zuständen erreicht.
Da davon ausgegangen werden kann, dass der Test häufig ausgeführt wird, lohnt sich die Umsetzung in der Automatisierung trotz des Aufwandes der Automatisierung.

6.1.3. Beobachtete Probleme

Bei diesem Beispiel sind drei Probleme beobachtet worden:

- Wenn der Frageneditor als neuer Frame geöffnet wird, wird er nicht gefunden. Der entsprechende Frame müsste aus der Applikation per get-Methode geholt und es müsste ein neues FrameFixture erzeugt werden. Eine weitere Möglichkeit ist, statt einen neuen Frame einen Dialog zu öffnen. Beide Varianten greifen in den Quelltext des Programmes ein.

- Alle Komponenten müssen Namen besitzen, da sie sonst nicht gefunden werden können. Auch das greift in den Quelltext ein. Allerdings nicht invasiv, da an der Programmlogik nichts geändert wird.

- Beim Ausführen darf die Maus nicht bewegt werden, da es sonst zum Fehlschlagen des Tests kommen kann, obwohl er erfolgreich sein müsste.

6.2. Fit

Fit ist ein Opensource-Werkzeug für Integrationstests [21]. Fit wurde entwickelt, um eine einfache Darstellung für Integrationstests bereitzustellen, damit der Kunde an dem Testprozess teilnehmen kann, ohne dass er technisches Wissen benötigt. Dazu werden Tests in HTML-Tabellen geschrieben und mit Hilfe von Fit eingelesen, interpretiert und auf den zu testenden Quelltext angewendet [22].
In Tab. 8 ist ein Beispiel für einen Test einer Methode, die die Fibonaccizahlen berechnet.
Durch den Eintrag `example.fixtures.FibonacciFixture` in der ersten Zelle, erkennt Fit mit welcher Klasse die HTML-Tabelle geladen werden soll. `FibonacciFixture` ist in diesem Beispiel die Klasse, die von Fit benutzt wird, um den zu testenden Quelltext aufzurufen. Danach folgt eine weitere Zeile, die die Namen der Spalten enthält. Der Name der ersten Spalte ist der Name eines öffentlichen Attributs der

example.fixtures.FibonacciFixture	
n	berechneFibonaccizahl()
0	0
1	1
2	1
3	2
5	5
14	37

Tabelle 8: Fibonacci Zahlen in einer Tabelle, die von Fit als Test ausgeführt werden kann.

Listing 6: Fixture für Fibonaccizahlen

```
1 package example.fixtures;
2
3 public class FibonacciFixture extends ColumnFixture {
4   public int n;
5
6   public int berechneFibonaccizahl() {
7     return Fibonacci.berechneFibonaccizahl(n);
8   }
9 }
```

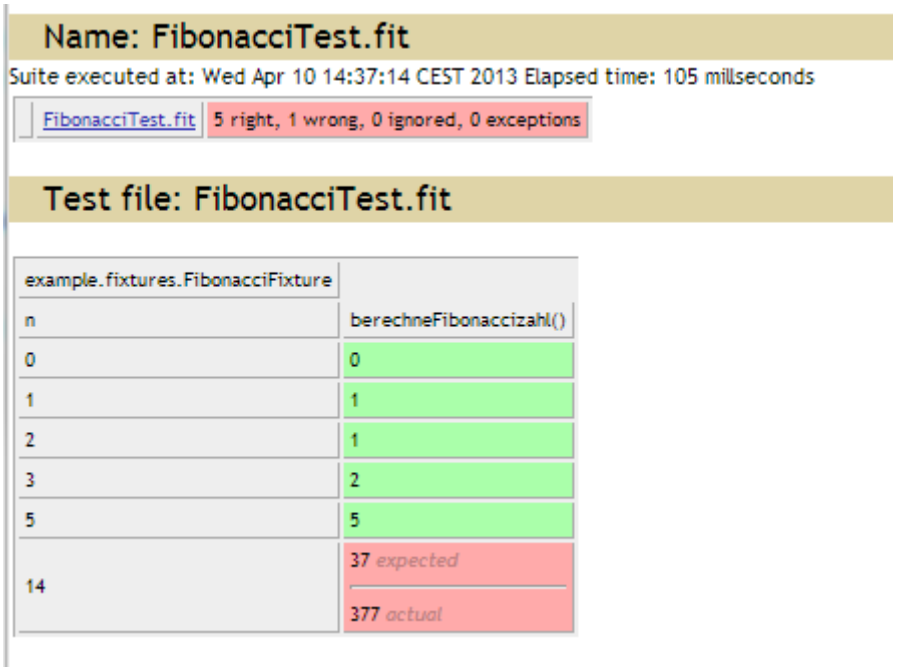
Klasse `FibonacciFixture`, während der Name der zweiten Spalte eine öffentliche Methode angibt. Jede folgende Zeile gibt in der ersten Spalte die Eingabe und in der zweiten Zeile das erwartete Ergebnis an der Methode an.
In Listing 6 ist das Fixture, das die Eingabe zum Aufruf einer Methode nutzt und das Ergebnis der Methode zurück gibt. In Abb. 14 ist das Ergebnis des Tests zu sehen. Die korrekten Ergebnisse werden grün markiert und die roten falsch. Außerdem werden bei falschen Ergebnissen der errechnete und der erwartete Wert angezeigt.

6.2.1. Fixture

Ein Fixture in Fit ist eine Klasse, die eine HTML-Tabelle interpretiert. Zur Interpretation der HTML-Tabelle werden in einem Fixture die im Klassendiagramm in Abb. 15 auf der linken Seite zu sehenden Methoden von oben nach unten aufgerufen. Mit jedem Methodenaufruf werden tiefere Tabellenelemente verarbeitet. Ein Beispiel ist in Abb. 15 auf der rechten Seite zu sehen. Durch Vererbung können diese Methoden überschrieben werden. Das bedeutet, dass der Entwickler den Ablauf steuern und zu verschiedenen Zeiten die Tabelleninterpretation verändern kann. Es ist also möglich ein eigenes Fixture zu schreiben, das eine eigene Interpretation einer Tabelle vornimmt.

6.2.2. Standard Fixtures

Fit enthält eine ganze Liste von fertigen `Fixture`-Klassen. Die drei Basis-Fixtures sind `ColumnFixture`, `RowFixture` und `ActionFixture`. Für die Automatisierung von Systemtests kommen `ColumnFixture` und `RowFixture` nicht in Frage, da sie



Name: FibonacciTest.fit

Suite executed at: Wed Apr 10 14:37:14 CEST 2013 Elapsed time: 105 milliseconds

FibonacciTest.fit	5 right, 1 wrong, 0 ignored, 0 exceptions

Test file: FibonacciTest.fit

example.fixtures.FibonacciFixture	
n	berechneFibonaccizahl()
0	0
1	1
2	1
3	2
5	5
14	37 *expected* 377 *actual*

Abbildung 14: Fit Test Ergebnisse für den Test in Tab. 8

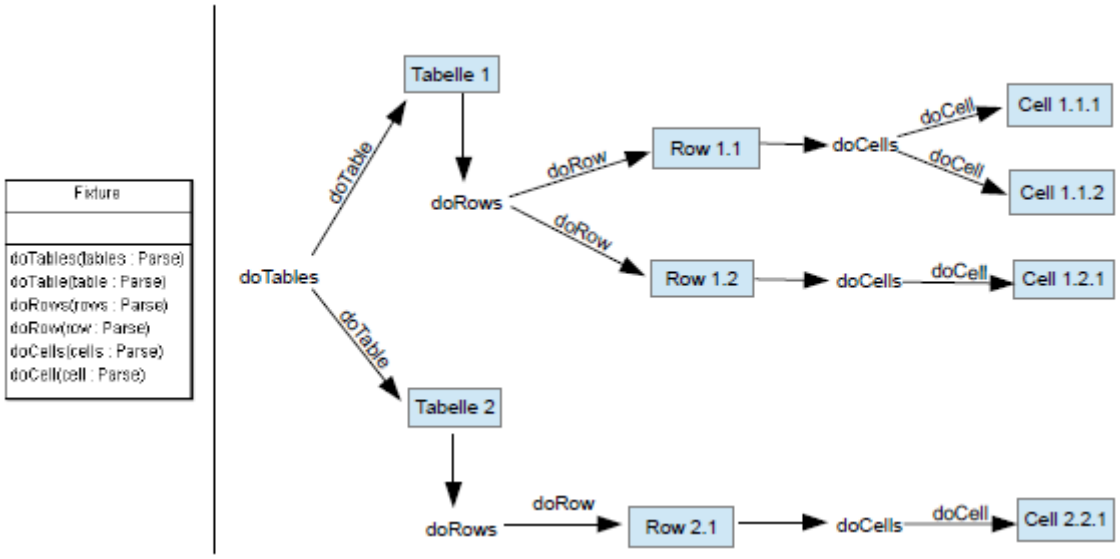


Abbildung 15: Klassendiagramm und aufrufe der Methoden

dafür konzipiert sind, Daten auf eine Methode anzuwenden, wie es im Fibonacci-Beispiel geschehen ist.
Die Idee der Klasse `ActionFixture` ist Befehle nacheinander auszuführen. Die Idee ist auf Systemtests übertragbar, denn durch sequentielles Ausführen von Befehlen kann die Oberfläche bedient werden. Der Grundgedanke von `ActionFixture` passt auf die Automatisierung von Systemtests, daher wird im Folgenden Abschnitt überprüft, inwiefern dieses Fixture zur Automatisierung von Systemtests benutzbar ist.

6.3. Symbiose von Fit und FEST-Swing

Im Folgenden wird eine Symbiose der beiden vorgestellten Frameworks angestrebt. Ziel ist es, die Vorteile der beiden Frameworks zu vereinen und Systemtests in HTML als sequentielle Abfolge von Befehlen zu schreiben. Die in HTML benutzten Befehle werden in dieser Arbeit als HTML-Befehle bezeichnet.
Die Anforderungen zum Steuern der Oberfläche mit Fit und FEST sind:

1. Es sollen Swing-Komponenten über HTML-Befehle bedienbar sein.
2. Es sollen mit Hilfe von HTML-Befehlen die Zustände von Swing-Komponenten geprüft werden können.
3. Die entwickelte Lösung soll generell auf Software, die mit Swing entwickelt wird, anwendbar sein.
4. Die Lesbarkeit eines Testfalls soll vereinfacht werden.

Folgende Probleme ergeben sich aus den Anforderungen:

1. Es muss ein passendes Fixture gefunden oder entwickelt werden, das das zeilenweise sequentielle Ausführen von Befehlen erlaubt.
2. Es müssen entsprechende Befehle implementiert werden, die aus den HTML-Dokumenten heraus benutzt werden können.

Ebenfalls aus den Anforderungen ist ein auf diese Art und Weise geschriebener Test vorstellbar. In Tab. 9 ist meine Vorstellung des Testfalls 1 (siehe Tab. 5) dargestellt. Die Tabelle ist zeilenweise zu lesen. In der ersten Spalte befindet sich der HTML-Befehl. In den darauf folgenden Spalten können, zur flexiblen Benutzung, Parameter an den HTMl-Befehl mitgegeben werden.

6.3.1. Ist das ActionFixture das richtige Fixture?

Im Abschnitt über Fit wurde das Fixture `ActionFixture` für Systemtests in Betracht gezogen.
Das `ActionFixture` führt zeilenweise Befehle aus und stellt bereits vier Befehle bereit.

start Wird benutzt um ein ActionFixture mit seinen Befehlen zu laden.

check Ruft eine Methode auf und überprüft ihren Rückgabewert mit einem erwarteten Wert.

HTML-Befehl	**Param. 1**	**Param. 2**	**Param. 3**	**...**
comment	Testschritt 1	Öffne den Dozenten Klienten und öffne den Frageeditor	Frageeditor ist offen, + Button ist benutzbar, − Button ist nicht benutzbar.	
starteSturesy				
pressButton	EditQuestion SetButton			
checkEnabled	AddQuestion			
checkDisabled	RemoveQuestion			
comment	Testschritt 2	Drücke den + Button	Eine neue Frage erscheint in der Liste und hat den Bezeichenr F1. Die Frage ist nicht selektiert. Die Buttons verbleiben in ihren Zuständen. + ist benutzbar, − ist nicht benutzbar.	
pressButton	AddQuestion			
listCount	QuestionList	=	1	
listContainsItem	QuestionList	F1		
listItemNotSelected	QuestionList	F1		
checkEnabled	AddQuestion			
checkDisabled	RemoveQuestion			

Tabelle 9: Vision eines Tests mit Hilfe von Fit und FEST

fit.ActionFixture		
start	fixtures.FrageneditorFixture	
press	editQuestionButton	
press	plusButton	
check	plusButtonEnabled	true
check	removeButtonEnabled	false

Tabelle 10: Implementation eines Teils des Testfall 1 mit dem ActionFixture

enter Ruft eine Methode mit einem Parameter auf.

press Ruft eine Methode ohne Parameter und Rückgabewert auf.

In Tab. 10 ist eine Benutzung des ActionFixture mit dem ersten Testschritt des bereits bekannten Testfall 1 zu sehen. Die Tabelle ist zeilenweise zu lesen. In der ersten Zelle der Tabelle wird Fit mitgeteilt, dass die Tabelle mit der Klasse `ActionFixture` zu interpretieren ist. In der zweiten Zeile wird eine neue Fixture-Klasse geladen, die Methoden bereitstellt, um Buttons zu drücken und den Zustand von Buttons zu prüfen.
Bei der Benutzung des Befehls check, ergibt sich das Problem, das für jede Komponente und jede ihrer Eigenschaften eine Methode implementiert werden muss. Diese Methoden werden in einer von `ActionFixture` erbenden Klasse implementiert. Es ist also ein hoher Programmieraufwand erforderlich, bevor die HTML-Tabellen verfasst werden können.
Dennoch erfüllt die Klasse `ActionFixture` die aufgestellten Anforderungen an das Bedienen und Prüfen von Oberflächen mittels Fit und FEST.

6.3.2. Probleme der Klasse ActionFixture

Auch wenn durch die Klasse `ActionFixture` die Anforderungen erfüllt scheinen, gibt es ein zu lösendes Problem der Klasse `ActionFixture`. In Tab. 9 werden bisher neun Befehle benutzt. Alle Befehle müssen in einer von ActionFixture erbenden Klasse implementiert werden. Das führt dazu, dass diese Klasse schon diese neun Methoden beinhalten wird. Bei Erweiterung der Klasse um weitere Befehle für Textfelder, Tabellen und andere Komponenten, wächst die Anzahl der Methoden weiter und die Klasse wird unübersichtlich. Die Erweiterbarkeit, die Wartbarkeit und die Wiederverwertbarkeit der Befehle ist dadurch stark eingeschränkt. Weiterhin ist die Testbarkeit der Befehle ein Problem, da die Methoden weder Parameter noch Rückgabewerte enthalten. Zusammenfassend ist das Problem zu lösen, dass nur Methoden des Objekts ausgeführt werden, das in der ersten Zelle der HTML-Tabelle angegeben wird und das sie parameterlos sein müssen.

6.3.3. Lösung des ActionFixture-Problems

Aus den erkannten Problemen wurden folgende Anforderungen an die Lösung herausgearbeitet:

1. Methoden, die als Befehle benutzt werden können, sollen in einer beliebigen Anzahl von Klassen implementiert sein können.

2. Die Methoden sollen eine beliebige Anzahl an Parametern und einen Rückgabewert besitzen.

Aus diesen Anforderungen wurde eine Lösung erarbeitet.
Es können beliebig viele Klassen mit HTML-Befehlen erzeugt werden. Diese Klassen implementieren ein Interface, damit erkannt werden kann, dass sie Methoden für HTML-Befehle implementieren. Diese Methoden sind mittels einer Annotation versehen. Diese Klassen können in einer neuen Fixture-Klasse registriert werden, damit die Fixture-Klasse die benutzbaren Befehle kennt.
Bei Ausführung einer Zeile einer HTML-Tabelle wird die erste Zelle als Befehl interpretiert und eine gleichnamige Methode unter den annotierten Methoden der registrierten Klassen gesucht. Der Aufruf der Methode erfolgt dann mittels der Java Reflection API. Das Ergebnis des Aufrufes wird in einem Objekt zusammengefasst und dieses von der neuen Fixture-Klasse interpretiert und verarbeitet.

6.3.4. Integration von FEST-Befehlen

Der letzte Schritt um Systemtests nach Tab. 9 umzusetzen, ist die Implementation von FEST-Befehlen. Dazu können nun für einzelne Komponenten Klassen implementiert werden und bei der neuen Fixture-Klasse registriert werden. Ein Beispiel für die Klasse `JButton` ist in Listing 7 zu sehen. Die Klasse implementiert das Interface `HasCommands` und ist somit als Klasse mit Methoden als Befehle markiert. Ein Objekt der Klasse benötigt eine Referenz auf ein Fenster, um einen JButton zu finden.
In der Klasse gibt es eine Methode, die als HTML Befehl benutzt werden kann. Dies ist die Methode `pressButton()` mit dem Buttonnamen als Parameter. In der Methode wird über die Methode `allocateButton()` (Zeile 17-26) versucht einen Button mit dem als Parameter übergebenen Namen zu suchen. Ist dies erfolglos, wird das Ergebnis entsprechend gesetzt. Ansonsten wird an dem gefundenen Button ein Mausklick simuliert und der Zustand des Ergebnisses gesetzt.

6.4. Exemplarische Automatisierung des Testfall 1 aus Kapitel 5

In Tab. 11 ist die Automatisierung des Testfall 1 aus Kapitel 5 zu sehen. Die Tabelle weist fast keine Unterschiede zu meiner Vision (s.h. Tab. 9) einer HTML-Tabelle als Oberflächentest auf. Der einzige Unterschied ist die erste Zelle der Tabelle, die das Fixture angibt, mit der die Tabelle zu interpretieren ist.
In der Tabelle werden Befehle für die beiden Komponenten `JButton` und `JList` benutzt. Die Befehle können unterschiedlich viele Parameter besitzen. In dem Beispiel sind HTML-Befehle mit ein bis drei Parametern enthalten.

fixtures. SturesyFestFixture Aggregator	**Param. 1**	**Param. 2**	**Param. 3**

comment	Öffne den Dozenten Klienten und öffne den Frageeditor	Frageeditor ist offen, + Button ist benutzbar, − Button ist nicht benutzbar.	
start			
pressButton	EditQuestionSetButton		
checkEnabled	AddQuestion		
checkDisabled	RemoveQuestion		
comment	Drücke den + Button	Eine neue Frage erscheint in der Liste und hat den Bezeichenr F1. Die Frage ist nicht selektiert. Die Buttons verbleiben in ihren Zuständen. + ist benutzbar, − ist nicht benutzbar.	
pressButton	AddQuestion		
checkListItemCount	QuestionList	=	1
checkListItemExist	QuestionList	F1	
checkNoListItem Selected	QuestionList		
checkEnabled	AddQuestion		
checkDisabled	RemoveQuestion		
comment	Drücke den + Button	Eine neue Frage erscheint in der Liste und hat den Bezeichenr F2. Die Frage ist nicht selektiert. Die Buttons verbleiben in ihren Zuständen. + ist benutzbar, − ist nicht benutzbar.	
pressButton	AddQuestion		
checkListItemCount	QuestionList	=	2
checkListItemExist	QuestionList	F1	
checkListItemExist	QuestionList	F2	
checkNoListItem Selected	QuestionList		
checkEnabled	AddQuestion		
checkDisabled	RemoveQuestion		
comment	Selektiere F1.	Die Buttons + und − sind nun beide benutzbar.	
selectListItem	QuestionList	F1	

checkEnabled	AddQuestion		
checkEnabled	RemoveQuestion		
comment	Drücke – Button.	Es existiert nur noch eine Frage mit dem Bezeichner F1 und keine Frage ist selektiert. – Button ist nicht benutzbar.	
pressButton	RemoveQuestion		
checkListItemExist	QuestionList	F1	
checkListItemCount	QuestionList	=	1
checkNoListItem Selected	QuestionList		
checkEnabled	AddQuestion		
checkDisabled	RemoveQuestion		
comment	Drücke – Button.	Es existiert keine Frage mehr in der Liste und der Button – ist nicht benutzbar.	
selectListItem	QuestionList	F1	
pressButton	RemoveQuestion		
checkListItemCount	QuestionList	=	0
checkDisabled	RemoveQuestion		
checkEnabled	AddQuestion		

Tabelle 11: Automatisierung mit Fit und FEST-Swing von Testfall 1 (siehe Tab. 5)

6.5. Ergebnis

Das Ziel war es Systemtests für Software mit grafischer Benutzerschnittstelle zu automatisieren. Um die Automatisierung zu vereinfachen, wurde im Rahmen der Arbeit eine Software entwickelt, die es ermöglicht Systemtests in HTML-Tabellen zu automatisieren.
Dazu wird eine HTML-Tabelle zeilenweise interpretiert. Jede Zeile der HTML-Tabelle stellt einen Befehl mit beliebig vielen Parametern dar. Die HTML-Befehle können durch Benutzung von Parametern wiederverwendet werden und müssen daher nur einmalig entwickelt werden. Die Software ist leicht um Befehle zu erweitern. Klassen müssen dazu lediglich ein Interface implementieren und die HTML-Befehle der Klasse müssen mit einer Annotation versehen werden.
Zur Entwicklung der Software wurden zur Entwicklung die zwei Werkzeuge Framework for integrated test und Fixtures for Easy Software Testing benutzt.
Durch die Einbindung von Fit war eine Grundstruktur zur Interpretation von HTML-Tabellen gegeben. Die Interpretation von HTML-Tabellen konnte leicht auf eigene Bedürfnisse angepasst werden. Das Einarbeiten des Frameworks wurde sehr gut

Listing 7: Implementation eines Fit Befehls zum Drücken eines Buttons

```
1  public class SwingButtonUIAdapter implements HasCommands {
2    private SwingFrameWrapper _frameWrapper;
3
4    public SwingButtonUIAdapter(SwingFrameWrapper frameWrapper) {
5      _frameWrapper = frameWrapper;
6    }
7    @FITCommand
8    public CommandResult pressButton(String buttonName) {
9      CommandResult result = new CommandResult();
10     JButtonFixture button = allocateButton(buttonName, result);
11     if (result.getResultState() != CommandResultState.WRONG) {
12       button.click();
13       result.setResultState(CommandResultState.RIGHT);
14     }
15     return result;
16   }
17   private JButtonFixture allocateButton(String buttonName,
18       CommandResult result) {
19     try {
20       return _frameWrapper.getFrameFixture().button(buttonName);
21     }
22     catch(ComponentLookupException e) {
23       FestResultBuilder.buildWrongResultComponentFailure(result,
24         buttonName);
25       return null;
26     }
27   }
28 }
```

durch die gute Dokumentation auf der Fit-Webseite [21] und das Buch „Fit for Developing Software: Framework for integrated Tests“ [22] unterstützt.
FEST wurde zur Steuerung der Oberfläche gewählt, da es das Testen von Swing-Oberflächen vereinfacht. Dazu stellt es die Möglichkeit bereit, Komponenten innerhalb eines Fensters zu finden und zu benutzen. FEST ist intuitiv handhabbar, sobald ein erster Einstieg gefunden wurde. Sollten allerdings Probleme auftauchen, bietet die Dokumentation wenig Hilfe, da sie zum einen schwer aufzufinden und zum anderen nicht sehr umfangreich ist.

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung der Ergebnisse

In dieser Arbeit wurden Modul- und Systemtests am Fallbeispiel StuReSy bearbeitet.
Zum Entwickeln weiterer Modultests wurden das Benutzen von Mock-Objekten mit dem Mock-Framework Mockito und das Messen der Testabdeckung mit dem Werkzeug Cobertura untersucht. Der Einsatz von Mock-Objekten ermöglichte da losgelöste Testen eines Testobjekts von dessen abhängigen Klassen. Durch das Benutzen des Mock-Frameworks Mockito wird das Benutzen von Mock-Objekten vereinfacht. Dies fällt besonders beim Erzeugen von Mock-Objekten auf, da dies das Framework vollständig für den Entwickler übernimmt. Mit Hilfe von Mockito wurden im Rahmen der Arbeit weitere Tests für das Software-System StuReSy geschrieben. Eine weitere Unterstützung beim Schreiben von Modultests erfolgte durch das Werkzeug Cobertura. Dies ermöglicht einzusehen, welche Quelltextstellen bereits getestet sind oder welche mehr Testbedarf besitzen. Cobertura wurde ohne Probleme als ein ANT-Ziel in den Entwicklungsprozess eingebunden. Dadurch können zu jeder Zeit HTML-Berichte über den Stand der Testabdeckung des Berichtes erzeugt werden. Die Testabdeckungsberichte waren sehr sinnvoll für die Neuentwicklung von Modultests. Die Aussagekraft der Testabdeckungsberichte wurde durch Ignorieren von nicht zu testenden Klassen verbessert.
Für die Entwicklung von Systemtest mit grafischer Benutzerschnittstelle wurde eine Software entwickelt, die es ermöglicht die Oberfläche über Befehle in HTML-Tabellen zu benutzen. Es wurde ein Grundstock an Befehlen implementiert. Durch die gute Erweiterbarkeit der Software können, die vorhandenen Befehle um weitere Befehle erweitert werden. Zur Implementation der Software wurden die Frameworks Fit unf FEST benutzt. Fit vereinfacht das Interpretieren von HTML-Tabellen. In den Zyklus der Interpretation ann einfach eingegriffen werden, um eine eigene Interpretation einer HTML-Tabelle zu schaffen. Dies wurde sich bei der Software zu nutze gemacht, um eine zeilenweise Interpretation zu schaffen, bei der in der ersten Spalte ein Befehl und in den folgenden Spalten seine Parameter abgelegt sind. Die HTML-Befehle, um die Oberfläche zu steuern, wurden mit Hilfe von FEST implementiert. FEST übernimmt die Suche von grafischen Komponenten in einem Fenster und ermöglicht die Simulierung der Benutzung von Komponenten. Die beiden Frameworks weisen keine Probleme bei gemeinsamer Benutzung auf.
Insgesamt ist durch die neue Software das Schreiben von Systemtests mit grafischer Oberfläche abstrahiert, und somit die Komplexität von Oberflächentests reduziert worden.

7.2. Ausblick

Insbesondere in der Automatisierung der Systemtests ist in Zukunft zusätzliche Arbeit erforderlich. Diese Arbeit bietet die Grundlage für die Automatisierung von Systemtests mit grafischer Oberfläche und führte exemplarische Tests in StuReSy ein. Es wurde allerdings nicht gemessen, wieviel Funktionalität durch die entwickelten Systemtests abgedeckt ist. Dies, und die Erweiterung um weitere Systemtests,

wäre eine Baustelle für zukünftige Arbeiten. Interessant daran ist die Fragestellung, wie die Abdeckung der Funktionalität durch Systemtests gemessen werden kann.
Ein weiteres Thema ist das Ausbauen der Neuentwicklung zu einem Testeditor, der Debugging-Mechanismen unterstützt. Dieser ist Teilweise außerhalb der Arbeit von mir bereits entwickelt worden, allerdings ist noch sehr viel Funktionalität zu implementieren. Ein solcher Testeditor muss HTML-Tabellen effizient editieren können und schrittweises Ausführen ermöglichen. Dadurch kann die Ausführung von Tests Schritt für Schritt erfolgen und das Ergebnis live beobachtet werden.
Ein ebenfalls im Rahmen der Neuentwicklung interessantes Thema ist die Parallelisierung von Systemtests. Da Systemtests eine höhere Ausführungsdauer als Modultests aufweisen, lohnt es sich mehrere parallel ausführen zu können. Systemtests benötigen allerdings spezielle Ressourcen wie zum Beispiel die grafische Oberfläche, wodurch das Parallelisieren erschwert wird.

Abkürzungsverzeichnis

StuReSy Student-Response-System

ARS Audience-Response-System

CSV Comma-separated values

WAM-Ansatz Werkzeug- und Materialansatz

WAM Werkzeug Automat Material

XML Extensible Markup Language

FEST Fixtures for Easy Software Testing

Fit Framework for integrated test

Abbildungsverzeichnis

Literatur

[1] Wolf Posdorfer. Prototyp einer freien Software für interaktive Vorlesungen mit Echtzeitabstimmungen, 2013. Bachelorarbeit im Studiengang Software-System-Entwicklung des Fachbereichs Informatik der Universtität Hamburg.

[2] Object contraint language specification. `http://www.h-itt.com/`, 2013. zuletzt abgerufen am 13.01.2013.

[3] Heinz Züllighoven. *Object-oriented construction handbook - developing application-oriented software with the tools and materials approach.* dpunkt.verlag, 2005. ISBN 978-3-89864-254-5.

[4] Wie lassen sich *.jar-dateien herstellen? `http://www.javabeginners.de/Grundlagen/Jar_erstellen.php`, 2013. zuletzt abgerufen am 03.03.2013.

[5] Jar file specification. `http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html`, 2013. zuletzt abgerufen am 03.03.2013.

[6] The apache ant project. `http://ant.apache.org/`, 2013. zuletzt abgerufen am 03.03.2013.

[7] Mac os x jarbundler ant task. `http://informagen.com/JarBundler/index.html`, 2008. zuletzt abgerufen am 03.03.2013.

[8] International Standards Organisation (ISO). International standard ISO/IEC 9126. information technology: Software product evaluation: Quality characteristics and guidelines for their use, 1991.

[9] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester: Foundation Level nach ASQF- und ISEB-Standard.* dpunkt.verlag, 2002.

[10] Oliver Burn et al. Checkstyle. `http://checkstyle.sourceforge.net/`, 2012. zuletzt abgerufen am 03.03.2012.

[11] Kurt Schneider. *Abenteuer Softwarequalität - Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement.* dpunkt.verlag, 2007.

[12] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software.* Spektrum Akademischer Verlag, 2002.

[13] S. Kleuker. *Qualitätssicherung Durch Softwaretests: Vorgehensweisen und Werkzeuge Zum Test Von Java-Programmen.* Vieweg Verlag, Friedr, & Sohn Verlagsgesellschaft mbH, 2012.

[14] Junit. `http://junit.org/`, 2013. zuletzt abgerufen am 24.03.2013.

[15] Cobertura. `http://cobertura.sourceforge.net/`, 2013. zuletzt abgerufen am 15.04.2013.

[16] ecobertura. http://ecobertura.johoop.de/, 2013. zuletzt abgerufen am 15.04.2013.

[17] Jfontchooser. http://jfontchooser.sourceforge.net/, 2004. zuletzt abgerufen am 18.05.2013.

[18] Mockito. http://code.google.com/p/mockito/, 2012. zuletzt abgerufen am 13.04.2013.

[19] Fixtures for easy software testing. http://fest.easytesting.org/, 2013. zuletzt abgerufen am 13.04.2013.

[20] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, München, 1996.

[21] Fit: Framework for integrated test. http://fit.c2.com/, 2007. zuletzt abgerufen am 04.04.2013.

[22] R. Mugridge and W. Cunningham. *Fit for Developing Software: Framework for Integrated Tests.* Robert C. Martin Series. Pearson Education, 2005.

Eidesstattliche Erklärung (für Bachelorarbeiten)

Ich erkläre, dass ich meine Bachelor-Arbeit „Modul- und Systemtests am Fallbeispiel StuReSy“ selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Ich versichere, dass die eingereichte schriftliche Fassung der auf dem beigefügten Medium gespeicherten Fassung entspricht.

Hamburg, den 22.05.2013

(Jens Dallmann)

A. Schnittstelle des Servers

relay.php Schnittstelle

Request:	relay.php?command=get&name=lectureid&pwd=password
Nutzen:	Fragt den Server nach Votingergebnissen
Antwort A:	VOTES:id,vote,timestamp;id,vote,timestamp;....
Antwort B:	VOTES:No Data
lectureid	Die Lecture-ID
password	Das dazugehörige Passwort
id	Die ID des Abstimmenden Gerätes (W123123, A123123, i123123, o.ä.)
vote	Die Stimme als Integer von 1 bis 10 für A bis J (nicht bei 0 anfangen)
timestamp	2013-01-08 16:30:12 (YYYY-MM-DD HH:mm:ss)

Request:	relay.php?command=info
Nutzen:	Fragt den Server nach Verfügbarkeit und Version
Antwort:	sturesy version
	Bisher wird keine version mitgesendet

Request:	relay.php?command=clean&name=lectureid&pwd=password
Nutzen:	Löscht alle Ergebnisse zur passenden lectureid mit password
Antwort:	*Keine Antwort*

Request:	relay.php?command=redeem&token=tokenstring
Nutzen:	Holt Lecture-ID und passendes Passwort vom Server, nur einmalig möglich, Token wird anschließend aus der DB gelöscht
Antwort A:	lectureid,password
Antwort B:	*Keine Antwort* bei falschem Token

Request:	command=update&name=lectureid&pwd=password &type=count&text=qtext&answers=ans
Nutzen:	Aktualisiert die Umfrage auf dem Server
Antwort:	*Keine Antwort*
count	Anzahl der Verfügbaren Antwortmöglichkeiten
qtext	Text der Frage
ans	JSON codierte Antworttexte: `{"0":"Hier Antwort A","1":"Hier Antwort B"}` Einzelne Strings sind HTML-Codiert (Leerzeichen werden „+“ etc..)

B. Schnittstelle der Webseite

index.php Schnittstelle

Request	index.php?lecture=lectureid
Nutzen:	Zeigt die Votingseite
Antwort A:	Die Votingseite wird gezeigt
Antwort B:	There is currently no Voting with this Lecture-ID

Request	index.php?lecture=lectureid&mobile=mobiletype
Nutzen:	Gibt JSON-Codierte Informationen über aktuelles Voting
Antwort A:	`[{"lecture":"test","question":"FRAGETEXT","date":"2013-01-08 13:42:59","answernumber":"0","answertext":"Antwort 0"}, {"lecture":"test","question":" FRAGETEXT","date":"2013-01-08 13:42:59","answernumber":"1","answertext":"Antwort 1"}]`
Antwort B:	`[]` (Ungültige LectureID oder kein Voting)
mobiletype	Bisher wird „android“ übergeben, da JSON auch von anderen Plattformen unterstützt wird. Also „android“ verwenden.

Request	index.php?lecture=lectureid&mobile=mobiletype&vote=voteencrypted
Nutzen:	Schickt eine Stimme an den Server
Antwort A:	`true` Stimme wurde aktzeptiert
Antwort B:	`false` Stimme wurde nicht aktzeptiert (falsch codiert, bereits abgestimmt, etc)
voteencrypted	Eine BASE64 codierte mit AES verschlüsselte kommaseparierte Nachricht bestehend aus: • Key – Der AES-Key selbst • LectureID – Die Lecture-ID zur Abstimmung • ID – Die ID des Telefons (A123123, i123123, o.ä.) • Vote – Die Antwort von 1 – 10 (nicht bei 0 beginnen) Bsp: `0011223344556677,testid,A123123,1`

C. Mit FEST-Swing automatisierter Testfall 1 aus Kapitel 5

Listing 8: Mit FEST–Swing automatisierter Testfall 1 aus Kapitel 5

```
@Test
public void fuegeFrageHinzu() {
  MainScreen mainscreen = new Startup(new String[]{}).getScreen();
  FrameFixture frame = new FrameFixture(mainscreen.getJFrame());
  testStep1(frame);
  testStep2(frame);
  testStep3(frame);
  testStep4(frame);
  testStep5(frame);
  testStep6(frame);
}

private void testStep1(FrameFixture frame) {
  frame.button("EditQuestionSetButton").click();
  pruefePlusMinusButtons(frame, true, false);
}

private void testStep2(FrameFixture frame) {
  frame.button("AddQuestion").click();
  pruefeListenInhalte(frame, new String[]{"F1"}, new String[0]);
  pruefePlusMinusButtons(frame, true, false);
}

private void testStep3(FrameFixture frame) {
  frame.button("AddQuestion").click();
  pruefeListenInhalte(frame,
      new String[]{"F1","F2"}, new String[0]);
  pruefePlusMinusButtons(frame, true, false);
}

private void testStep4(FrameFixture frame) {
  frame.list("QuestionList").selectItem("F1");
  pruefePlusMinusButtons(frame, true, true);
}
private void testStep5(FrameFixture frame) {
  frame.button("RemoveQuestion").click();
  pruefeListenInhalte(frame, new String[]{"F1"}, new String[0]);
  pruefePlusMinusButtons(frame, true, false);
}
private void testStep6(FrameFixture frame) {
  frame.list("QuestionList").selectItem("F1");
  frame.button("RemoveQuestion").click();
  pruefeListenInhalte(frame, new String[0], new String[0]);
  pruefePlusMinusButtons(frame, true, false);
}
private void pruefePlusMinusButtons(FrameFixture frame,
    boolean plusButtonEnabled, boolean minusButtonEnabled) {
  JButtonFixture plusButton = frame.button("AddQuestion");
  JButtonFixture minusButton = frame.button("RemoveQuestion");
  pruefeButton(plusButton, plusButtonEnabled);
  pruefeButton(minusButton, minusButtonEnabled);
}
private void pruefeButton(JButtonFixture button,
    boolean isEnabled) {
  if(isEnabled){
    button.requireEnabled();
  }
  else {
    button.requireDisabled();
  }
}
```

```
62| private void pruefeListenInhalte(FrameFixture frame,
63|     String[] erwarteteFragen, String[] erwarteteSelektierteFragen) {
64|   JListFixture fragenListe = frame.list("QuestionList");
65|   String[] listenInhalte = fragenListe.contents();
66|   String[] selektierteFragen = fragenListe.selection();
67|   assertEquals(erwarteteFragen.length, listenInhalte.length);
68|   assertArrayEquals(erwarteteFragen, listenInhalte);
69|   assertArrayEquals(erwarteteSelektierteFragen, selektierteFragen);
70|   assertEquals(erwarteteSelektierteFragen.length,
71|       selektierteFragen.length);
72| }
```