Studienarbeit

Testen interaktiver Werkzeuge

Dezember 2003

Sebastian Sanitz
Wohlwillstr. 1
20359 Hamburg
Email: sebastian@sanitz.net
Matrikelnummer: 4724018

Betreuer: Prof. Dr.-Ing. Heinz Züllighoven

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Vogt-Kölln-Str. 30
22527 Hamburg

Inhaltsverzeichnis

Kapitel 1

Einleitung

In dieser Arbeit wird untersucht, wie interaktive Software mit Akzeptanztests getestet werden kann. Zu Beginn möchte ich darstellen, wie ich auf das Thema der Studienarbeit aufmerksam geworden bin, einen Überblick über den Inhalt geben, welches Ziel ich mit dieser Arbeit verfolge und wo sich die Arbeit in die aktuelle Diskussion um das Thema Testen von Software einordnet. Danach werde ich zur Orientierung einen kurzen Überblick über den Aufbau der Arbeit geben.

1.1 Motivation und Ziele der Arbeit

Als ich mich mit der Konstruktion objektorientierter Anwendungen mit Rahmenwerken befasst habe, wurde ich von Stefan Roock auf das Thema Testen von Software aufmerksam gemacht.

Testen von Software ist das Ausführen von Programmcode unter verschiedenen Kombinationen von Eingaben und Zuständen, um Fehler zu entdecken [Bin99]. Das Kapitel 2 wird genauer auf die Konzepte, Begriffe und deren Definition im Zusammenhang mit dem Testen von Software eingehen.

Bei der bisherigen Entwicklung des Rahmenwerkes JWAM[1] [JWA02] wurden mit der Hilfe des Testframeworks *JUnit* [GB98] Komponententests geschrieben. Diese Komponententests testen jeweils eine kleine Einheit, im konkreten Fall meist eine Klasse. Sie sind vorrangig für den Entwickler gedacht und sollen die Funktionalität der jeweiligen Klasse prüfen. Die Tests werden als Javacode geschrieben, genau wie Anwendungscode kompiliert und lassen sich mit einem Werkzeug wie JUnit automatisch ausführen. Beim *eXtreme Programming* [Bec99] werden neben diesen Komponententests auch noch Akzeptanztests, früher funktionale Tests genannt,

[1] Abkürzung für Java-Rahmenwerk und Werkzeug, Automat und Material, siehe Kapitel 3

beschrieben. Diese Akzeptanztests prüfen, ob die gesamte Anwendung die geforderte Funktionalität aus der Sicht der Auftraggeber bzw. der Benutzer erfüllt. Jeder Akzeptanztest beschreibt dabei auch nur einen kleinen Teil der gesamten Anforderungen. Während der gesamten Entwicklung müssen Komponententests erfolgreich, also fehlerfrei durchlaufen. Dagegen müssen Akzeptanztests erst bei einem Release bzw. einer Auslieferung an den Kunden diesem Anspruch genügen. Zudem gibt die durchschnittliche Anzahl der erfolgreich durchlaufenden Akzeptanztest ein grobes Maß für den Projektfortschritt an. Für Akzeptanztests soll im JWAM-Rahmenwerk, genau wie für Komponententests, die Möglichkeit geschaffen werden möglichst einfach Tests schreiben und automatisch ausführen zu können.

In der Literatur zum Thema Testen von Software klassifiziert hauptsächlich der Gegenstand der getesteten Anwendung den Test, sowie die Herangehensweise und den Testprozess. Robert V. Binder [Bin99] unterscheidet zwischen Komponententests, Integrationstests und Systemtests. Hier wird die Größe und Zusammengehörigkeit der zu testenden Softwarekomponenten als Unterscheidungsmerkmal gewählt. Bei einem Komponententest wird eine kleine Einheit getestet, bei dem Integrationstest soll das Zusammenspiel mehrerer zusammengehörender Einheiten geprüft werden und bei einem Systemtest soll das fertige System als Ganzes getestet werden.

In dieser Studienarbeit soll der Blickpunkt und die Herangehensweise beim Testen auch auf den Umgang mit dem Gegenstand des Tests berücksichtigt werden. Beim Testen soll nicht nur der technische Kontext der Software in Betracht gezogen werden. Der Umgang des Benutzers mit der Software ist zu beachten.

Das in Java geschriebene Framework JWAM zur Unterstützung der Entwicklung interaktiver Anwendungen soll eine Grundlage dieser Arbeit sein. Bisher gab es im Kontext von JWAM nur die Unterstützung von Tests für kleine Einheiten mit dem Testwerkzeug *JUnit* [GB98]. Das Testen von ganzen Systemen oder von interaktiven Anwendungen ist nicht unterstützt worden. Es ist aufgefallen, das die in JWAM vorhandenen Konstruktionsmuster zur Anfertigung von interaktiven Anwendungen und insbesondere die Trennung von Interaktionsformen und Präsentationsformen eine Grundlage bieten für die Erstellung von Tests für interaktive Anwendungen. Zudem sind die aus den Entwicklungsdokumenten im WAM-Kontext [Zül98] stammenden Handhabungsvisionen eine gute Vorlage für die Erstellung von Akzeptanztests. Es soll bei den Tests nicht direkt die konkrete Oberfläche, also die einzelnen GUI-Elemente[2] eines Werkzeuges getestet werden wie mit einem GUI-Playback-Capture-Tool, das auf Ebene des Betriebssystems Benutzereingaben simuliert. Vielmehr soll die Inter-

[2]GUI - Graphical User Interface

aktion eines Benutzers mit dem Werkzeug simuliert werden, ohne das auf ein konkrete Oberfläche zugegriffen wird. Für eine eingehende Betrachtung dieses Themas möchte ich auf die Studienarbeit von Malte Kröger [Krö00] verweisen. Eine Unterstützung für Akzeptanztests durch das Rahmenwerk JWAM zu realisieren ist das Ziel dieser Studienarbeit. Außerdem soll das dort bis dato benutzte Testwerkzeug JUnit mit einem Werkzeug ergänzt werden, das die verschiedenen Tests automatisch ausführt, zwischen Komponententests und Akzeptanztests unterscheiden kann und nach der WAM-Metapher [Zül98] gebaut ist.

1.2 Aufbau der Studienarbeit

Zuerst werde ich mich mit den Grundlagen und Zielen des Testens von Software beschäftigen. Es soll ein allgemeiner Überblick geschaffen werden, was Testen ist, was damit erreicht werden kann und wie sich Testen zu anderen Methoden der Qualitätssicherung abgrenzt. Ich werde dabei auf vorhandene Konzepte und Sichtweisen in diesem Zusammenhang eingehen und deren konzeptionellen Unterschiede und Begriffe erläutern. Es werden die Besonderheiten und Probleme erklärt, die sich beim Testen von objektorientierten Systemen ergeben. Ich möchte aufzeigen wo die Probleme liegen, z.B. bei der bisherigen Klassifikation von Tests und einen Lösungsansatz bieten. Danach werde ich beschreiben wie das Testen in JWAM vor meiner Studienarbeit unterstützt wurde. Es werden Akzeptanztests und die dazugehörigen Konzepte genau erörtert.

Die in JWAM vorhandenen und für die Realisierung der Arbeit nützlichen Muster zur Trennung von Interaktion und Funktion, sowie der Trennung von Interaktions- und Präsentationsformen werde ich kurz zum besseren Verständnis erläutern. Auf diesen aufbauend wird die Konzeption dieser Arbeit für Akzeptanztests beschrieben. Ausführlich werde ich auf das Design der Realisierung von interaktiven Tests und des neuen Testwerkzeuges für JWAM eingehen. Abgeschlossen wird die Arbeit mit einer kritischen Zusammenfassung und einem Ausblick auf weitere Fragestellungen und Erweiterungsmöglichkeiten.

1.3 Danksagung

Ich möchte Stefan Roock danken für seine geduldigen Reviews und die vielen guten Ratschläge. Heinz Züllighoven gilt mein Dank für seine Unterstützung und die wertvollen Hinweise, die diese Arbeit erst ermöglichten.

Auch gilt mein Dank allen Mitarbeitern der Firma it-wps (ehemals Apcon WPS) die mir sowohl bei der Implementation, als auch in zahlreichen Diskussionen sehr geholfen haben. Hier möchte ich Robert F. Beeger erwähnen, der bei dem praktischen Teil der Arbeit immer zur Diskussion bereit und sehr hilfreich war. Vielen Dank auch an Martin Lippert, der die Arbeit nochmal mit kritischem Auge durchgelesen hat.

Kapitel 2

Testen von Software

Um sich mit dem Testen von Software eingehender zu befassen, muss geklärt werden, was Testen von Software genau ist. Dabei stellt sich die Frage, welche Ziele mit dem Testen verfolgt werden. Den Definitionen von Binder [Bin99] folgend wird auf Testmodelle und Teststrategien, sowie die Merkmale eines Fehler und seiner unterschiedlichen Klassifikation eingegangen. Zudem werden die Besonderheiten erläutert, die durch die objektorientierte Programmierung für das Testen entstehen.

2.1 Ziele beim Testen von Software

Das Testen von Software ist neben Analyse, Entwurf, Implementation, sowie Einsatz und Wartung, Teil jeder Vorgehensweise bei der Entwicklung von Softwaresystemen. Zudem ist es ein wichtiger Bestandteil der Qualtitätssicherung.

Unter der Qualitätssicherung versteht man sämtliche Aktivitäten, die die Qualität des zu erstellenden Produktes gewährleisten sollen. Qualität ist nach DIN 55350, Teil 11 *„. . . die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht.“* Die Qualitität eines Produktes ist somit die Summe aller seiner Eigenschaften bezogen auf die gestellten Anforderungen.

Eine Implementation muss geprüft werden, um die Qualität zu sichern. Dabei soll festgestellt werden, inwieweit sie den Eigenschaften ihrer Spezifikation entspricht. Die Software muss ausgeführt werden, um zwischen der Spezifikation und den wirklichen Eigenschaften dieser Software vergleichen zu können. Testen bedeutet also das Ausführen eines Programmes, bzw. von Programmteilen, unter verschiedenen Kombinationen von Eingaben und Zuständen, wobei erwartete und reale Ausgaben verglichen werden.

Testen hat also die Aufgabe Fehler in einem Programm zu entdecken.

Es ist insbesondere bei interaktiven Programmen praktisch unmöglich alle erdenkbaren Kombinationen von Eingaben zu simulieren und mit den erwarteten Ausgaben zu vergleichen. In [Dij72] wird deshalb geschrieben: „Man kann nur die Anwesenheit von Fehlern, nicht deren Abwesenheit beweisen“. Testen kann also nicht formal die Korrektheit von Software beweisen, noch Evaluation oder konstruktive Kritik bei der Qualitätssicherung ersetzen.

2.2 Testmodelle und Teststrategien

Für die meisten Programme ist die Anzahl der möglichen Eingaben kombiniert mit den Zuständen unendlich, siehe [Leh80]. Welche Eingaben in Kombination mit welchen Zuständen sollen dann getestet werden? Ziellos herumprobieren, was an dem Programm fehlerhaft sein könnte, macht das Testen ineffektiv. Eine Menge von Tests die an zufällig ausgewählten Teilen der Software prüfen, vermitteln den Entwicklern ein falsches Gefühl von Vertrauen. Es muss dort getestet werden, wo Fehler am wahrscheinlichsten sind.

Da die Anzahl der möglichen Tests für ein Programm praktisch unendlich ist, muss sich das Testen auf ein *Testmodell* beschränken. Ein Testmodell ist eine prüfbare Repräsentation der Zusammenhänge von Teilen des Systems. Ein Testmodell muss sich an einem *Fehlermodell* orientieren. Ein Fehlermodell identifiziert Beziehungen oder Teile des Systems in denen mit großer Wahrscheinlichkeit Fehler gefunden werden können.

Eine *Teststrategie* legt anhand eines Testmodelles fest, wie geprüft wird. Sie legt fest, welche *Testfälle* zu erstellen sind. Dabei definiert ein Testfall die Umgebung und den Zustand des Systems vor dem Test, sowie eine Menge von Eingaben, bedingten Ausführungsfolgen und erwarteten Ausgaben. Die Ausführung des Testfalles stellt also den konkreten Test dar.

Teststrategien lassen sich je nach Blickwinkel in verschieden Kategorien einteilen. Unterscheidet man die Teststrategien daran, ob Kenntnis über den Aufbau und die Implementierung der zu testenden Software besteht, so wird unterteilt in *Whitebox-Testing* und *Blackbox-Testing*. Bei dem Blackbox-Testing wird nur anhand der Spezifikation und der Schnittstelle getestet, unabhängig davon, wie die Implementation aussieht. Bei dem Whitebox-Testing wird vorrausgestzt, dass die Implementierung des zu prüfenden Objektes bekannt ist. Sie basiert also auf der Analyse des Sourcecodes. Dabei wird versucht mögliche Schwachstellen zu erkennen und mit Testfällen abzudecken.

Teststrategien werden als *fault-directed* oder *conformance-directed* bezeichnet. Sie werden an der Absicht unterscheiden, ob die Tests (teils bekannte) Fehler suchen sollen oder ob getestet wird, dass das zu prüfende Programm genau seiner Spezifikation entspricht. Je kleiner der Gegen-

stand der Tests ist, desto kleiner ist auch die Anzahl der Kombinationen von Eingaben und Zuständen. Somit ist es bei Tests für kleine Einheiten wahrscheinlicher, zu einem frühen Zeitpunkt in der Entwicklung Fehler zu finden.

2.2.1 Komponenten-, Integrations- und Systemtests

Für die einzelnen Softwaretests gibt es nach klassischer Definition [Mye78, Bin99] die Klassifikationen in *Komponententests*, *Integrationstests* und *Systemtests*. Dabei wird die Größe und Zusammengehörigkeit der jeweils zu testenden Softwarekomponenten als Unterscheidungsmerkmal gewählt.

Bei einem Komponententest wird eine kleine Einheit getestet. Bei objektorientierten Systemen handelt es sich hier um eine Klasse oder mehrere eng zusammenhängende Klassen. Fehler können lange unentdeckt blieben, wenn keine Komponententests geschrieben werden. Tests auf dieser Ebene prüfen eine wesentlich größere Menge von möglichen Eingaben und Zuständen, als das ein Integrationstest oder Systemtest könnte. Die Zeit einen Fehler zu finden wird deutlich reduziert, wenn die Entwickler der zu prüfenden Klassen diese auch ausgiebig testen. Kapitel 2.3 befasst sich genauer mit diesem Aspekt.

Das Zusammenspiel mehrerer zusammengehörender Einheiten soll bei dem Integrationstest geprüft werden. Bei der Entwicklung eines objektorientierter Systems lässt sich zwischen Komponententests und Integrationstests schwer trennen, besonders wenn eine Komponente von mehreren anderen Komponenten abhängt.

Der Systemtest soll das fertige System als Ganzes prüfen. Hierbei wird nicht nur die Funktionalität des Systems anhand der Ein- und Ausgaben und das Zusammenspiel der verschiedenen Komponenten geprüft. Zusätzlich wird überprüft, ob die nichtfunktionalen Anforderungen, wie z.B. die Performanz des Systems, erfüllt werden.

2.2.2 Entscheidungstabellen und Transitionsdiagramme

In diesem Kapitel werden zwei Techniken beschrieben, mit dessen Hilfe sich ein Testmodell definieren lässt. Das sind *Entscheidungstabellen* und *Transitionsdiagramme*.

Wenn ein Programm nur anhand der Eingaben, also zustandsunabhängig, die Ausgabe berechnet, dann können Entscheidungstabellen bei der Erstellung eines Testmodells hilfreich sein. Eine Entscheidungstabelle besteht aus zwei Spalten, zum einen den Eingaben und zum anderen den Ausgaben. Jede mögliche Kombination von Eingaben wird in der Entscheidungstabelle in einer Zeile mit den entsprechenden Ausgaben dargestellt.

Jahresumsatz	Kunde seit	Rabatt
unter 1.000 €	unter 5 Jahren	0,0%
	über 5 Jahren	1,5%
1.000 bis 10.000 €	unter 5 Jahren	2,3%
	über 5 Jahren	3,4%
über 10.000 €	unter 5 Jahren	5,0%
	über 5 Jahren	6,6%

Abbildung 2.1: Beispiel für eine Entscheidungstabelle

Die Tabelle dient als Basis des Testmodells für die Erstellung von Testfällen. Ein fiktives Beispiel ist die Rabatttabelle in Abbildung 2.1. Anhand des Jahresumsatzes und der Dauer der Kundenbeziehung wird entschieden, wieviel Rabatt dem Kunden gewährt werden darf. Ein Testmodel könnte z.B. festlegen, das für jede Rabattstufe mindestens ein Test geschrieben wird.

Entscheidungstabellen sind für die Beschreibung und als Grundlage für Testmodelle bei zustandsabhängigen Systemen nicht mehr ausreichend. Bei zustandsabhängigen Systemen wird anhand des Zustandes und der Eingabe die Ausgabe entschieden, und in welchen Zustand das System übergeht. Eine Entscheidungstabelle für ein zustandsabhängiges Systeme müsste zusätzlich zu den möglichen Eingaben die möglichen Zustände des Systems berücksichtigen. Die Anzahl der Kombinationen von Eingaben und Zuständen wird dadurch um so größer.

Bei zustandsabhängigen Systemen eignet sich ein Transitionsdiagramm als Grundlage für ein Testmodell besser. Ein Transitionsdiagramm kann mit Hilfe eines *endlichen Automaten*[1] erstellt werden. Ein endlicher Automat ist ein mathematisches Modell eines Systems. Dieses Modell besteht aus einer endlichen Menge von Zuständen, einem endlichen Eingabealphabet, einem Anfangszustand, einer Menge von Endzuständen und einer Übergangsfunktion. Die Übergangsfunktion bildet von den Zuständen und den Eingaben auf die Zustände ab. Das bedeutet, dass für jeden Zustand, kombiniert mit jedem Eingabesymbol genau ein Zustandsübergang existiert, auch wenn er wieder in den gleichen Zustand überführt. Ein endlicher Automat lässt sich mit Hilfe eines Transitionsdiagrammes visualisieren, wobei Kreise die Zustände darstellen und mit den Eingaben beschriftete Pfeile die Zuständsübergänge kennzeichnen.

Ein Beispiel für einen endlichen Automaten ist in Abbildung 2.2 dargestellt. Der mit *Start* markierte Pfeil zeigt auf den Anfangszustand q_0. Der Endzustand ist ebenfalls q_0 und durch einen doppelten Kreis markiert. Dieser endliche Automat akzeptiert alle Zeichenketten, die aus Nullen und

[1] siehe [HU79]

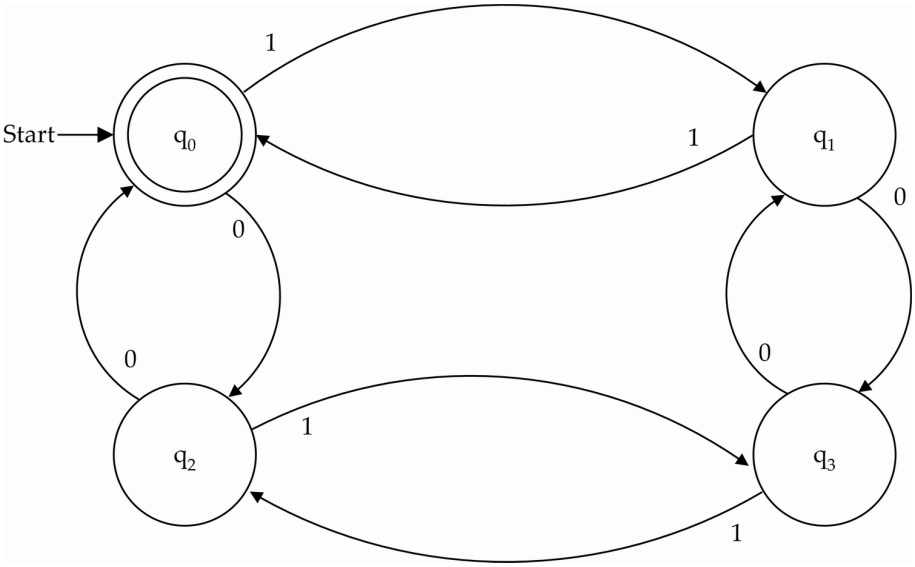

Abbildung 2.2: Beispiel für einen endlichen Automaten

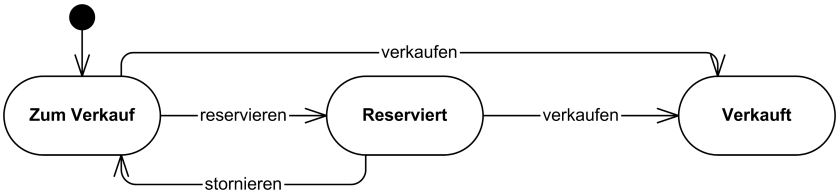

Abbildung 2.3: Beispiel für ein UML Zustandsdiagramm

Einsen bestehen, wobei die Anzahl der Nullen als auch die Anzahl der Einsen gerade ist. Das Beispiel ist [HU79] entnommen.

Zur einfachen Vergegenständlichung der Zustandsmöglichkeiten und Zustandsübergänge in einem zu testenden System eignen sich auch die von Harel [Har87] definierten *state charts*, die *FREE-Charts* von Robert V. Binder [Bin99], oder die *Zustandsdiagramme* aus der Beschreibungssprache Unified Modelling Language (UML), siehe auch [Fow97]. Ein Beispiel für ein Zustandsdiagramm ist in Abbildung 2.3 zu sehen. Hier werden Zustände und zugehörige Zustandsübergänge für beliebige Waren gezeigt, die reserviert und verkauft werden. Ein Testmodel könnte in diesem Beispiel fordern, das in den Tests jeder Zustand eingenommen wird und jeder Zustandsübergang von dem zu testendem System durchlaufen wird.

Es ist allen Darstellungen der Zustandsmöglichkeiten eines Systems gemeinsam, dass sie durch Visualisierung verdeutlichen, aus welchem Zustand durch welche Eingabe ein Zustandsübergang erfolgen soll. Dieser

Umstand führt zu einem leichteren Verständniss komplexer Sachverhalte. Ein Transitionsdiagramm bzw. Zustandsdiagramm ist daher eine Grundlage für ein Testmodell. So lässt sich an ihm leicht ablesen, welcher Zustand nach welcher Eingabe eintreten soll.

2.2.3 Testorakel

Zu einem Testfall gehören eine Menge von Eingaben und entsprechende erwartete Ausgaben. Ein Programm welches diese Eingaben mit den zugehörigen zu erwartenden Ausgaben erstellt, nennt man ein Testorakel. Ein Testorakel kann zumeist nur Teile der Eingabemenge abdecken. Ein Testorakel wird ein *perfektes Testorakel* genannt, wenn es sämtliche Bereiche der Eingabemenge abdeckt und zu jeder Eingabe ein richtiges Ergebniss liefert. Es wäre äquivalent zu der idealen Lösung der zu testenden Software. Somit ist es genauso schwer, ein perfektes Testorakel zu erstellen, wie die eigentliche Software fehlerfrei zu implementieren. Im Rahmen dieser Arbeit wird nicht weiter auf das Thema eingegangen, in [Bin99] werden verschiedene Ansätze zur Entwicklung von Testorakeln präsentiert.

2.3 Testen bei der Entwicklung

Wie in Kapitel 2.1 beschrieben ist Testen Bestandteil jeder Vorgehensweise. In [FZ99] wird Testen unter dem Begriff Validation geführt und gehört zu den Maßnahmen zur konstruktiven Qualitatssicherung. Zum Beispiel ist das Testen beim Wasserfallmodel eine eigene Phase, die nach der Implementation und vor dem Einsatz durchlaufen werden soll. In zyklischen Modellen werden die Tests in jedem Zyklus nach der Erstellung bzw. Weiterentwicklung der Software ausgeführt.

Testen ist nicht nur Teil der Qualitätssicherung und der Vorgehensweise bei der Softwareentwicklung, es ist auch eine besonderere Hilfe für den Entwickler, wie beschrieben beim *Refactoring* [FBB+99] und beim *eXtreme Programming* [Bec99]. In beiden Zusammenhängen gilt das Schreiben von Komponententests als unabdingbare Vorraussetzung für die Entwicklung und Weiterentwicklung von Software, und sollte für alle Erweiterungen, Änderungen und zur Fehlersuche am Programm genutzt werden.

Bei der Erweiterung eines Systems um neue Funktionalität werden beim *eXtreme Programming* schon vor der Implementierung der einzelnen Eigenschaften entsprechende Komponententests geschrieben. In diesen Komponententests werden alle erwarteten Eigenschaften geprüft. Bei der Erstellung werden immer wieder sämtliche Tests durchgeführt. Die Funktionalität gilt als fertig implementiert wenn die Tests fehlerfrei durchlaufen. Die Sammlung von Tests dient somit als Indikator für den Erfolg bei der Entwicklung.

Um allgemein Änderungen an einem bestehenden System vorzunehmen, ist es von unschätzbarem Vorteil, wenn das System mit ausreichend Tests ausgestattet ist. Nach kleinen Änderungen können diese immer wieder durchlaufen werden, um zu sehen, das nicht unbeabsichtigt Fehler eingebaut wurden. Diese Tests werden *Regressionstests* genannt.

Wenn ein System einen Fehler aufweist, der berichtigt werden soll, so werden zugehörige Komponententests geschrieben. Diese Tests müssen genau die Eigenschaften prüfen, die den Fehler ausmachen. Jetzt dient auch hier die Sammlung von Komponententests als Erfolgsindikator. Wenn nach der Korrektur alle Tests erfolgreich durchlaufen, gilt der Fehler als behoben.

Eine wichtige Rolle bei der Entwicklung spielen die *Akzeptanztests*. Im klassischen Sinne ist es der Test des Kunden, der das Softwaresystem in Auftrag gegeben hat. Hier wird an einem laufenden System geprüft, ob es den im Auftrag beschriebenen Anforderungen entspricht. Diese Tests werden üblicherweise schon während der Entwicklung geschrieben, um zu sehen wie weit die Entwicklung fortgeschritten ist und welche Eigenschaften noch implementiert werden müssen. Die Akzeptanztests sind sowohl System-, als auch Integrationstests. Bei den Akzeptanztests wird nicht wie bei Komponententests nur ein kleiner Teil des Systems geprüft, sondern mindestens eine Anwendung bzw. das ganze System selber. Bei interaktiven Anwendungen müssen die Anwendungen „von Hand“ bedient und die Ausgaben auf Korrektheit geprüft werden. Es werden dabei häufig Szenarien aus der Anwendungswelt durchgespielt und geprüft ob das System der Spezifikation konform reagiert. Diese Tests lassen sich nur schwer automatisch ausführen. Hierzu werden in [Krö00] Werkzeuge zum *GUI-Capture-Playback* vorgestellt. Diese protokollieren die Benutzereingaben einer interaktiven Anwendungssitzung und können diese automatisch wieder abspielen. Dabei werden die Ausgaben des Programmes mit erwarteten Ergebnissen vergleichen. In Kapitel 4 wird eine Lösung aufgezeigt, wie für interaktive Anwendungen Akzeptanztests gestaltet und auf einfache Weise erstellt und automatisch ausgeführt werden können. Eine auf dem JWAM-Rahmenwerk basierende Umsetzung dieses Konzeptes wird in Kapitel 5 vorgestellt.

2.4 Testmetriken

Auf die Frage, ob ein bestehendes Softwaresystem ausreichend getestet wird, versuchen Testmetriken über verschiedene *Abdeckungsgrade* eine Antwort zu liefern.

Wie in Kapitel 2.2 beschrieben, legt eine Teststrategie fest, welche Testfälle zu erstellen sind. Allgemein besagt der Abdeckungsgrad welcher Prozensatz der zu erstellenden Testfälle geschrieben ist. Wird der Begriff

Abdeckung ohne Prozentangabe gemacht, so wird eine 100%ige Abdeckung unterstellt. Die Abdeckung gibt es als konkrete Maße u.a. in der *Zeilenabdeckung*, der *Verzweigungsabdeckung* und der *Pfadabdeckung*[2].

Dabei gibt die Zeilenabdeckung an, wieviele der Zeilen im Programmcode von den Tests aufgerufen werden. Sie beschreibt das Verhältnis der aufgerufen Programmzeilen zu allen vorhanden Zeilen.

Ein genaueres Maß ist die Verzweigungsabdeckung. Sie beschreibt die Abdeckung der möglichen Verzweigungen im Kontrollfluss des Programmes. Sie gibt das Verhältnis der durchlaufenen Verzweigungen zu den möglichen Verzweigungen an.

Die Pfadabdeckung orientiert sich an den möglichen Pfaden von Anweisungen, die in dem zu testendem Programm durchlaufen werden können. Durch Schleifen und bedingte Verzweigungen ist die Anzahl der möglichen Pfade meist zu groß um alle Kombinationen zu testen.

Für weitere Abdeckungsgrade sei auf [Bin99] verwiesen. Die verschiedenen Abdeckungsgrade ersetzen auf keinen Fall ein Testmodell. Sie sind nur ein Mittel um zu erkennen ob den Ansprüchen im Testmodell entsprechend getestet wird. Es besteht die Gefahr, das Tests die einen Abdeckunsgrad erfüllen, fälschlicherweise gleichzeitig die Abwesenheit von Fehlern suggerieren. Eine 100%ige Zeilen-, Verzweigungs- oder Pfadabdeckung bedeutet noch keine Fehlerfreiheit. Es kann z.B. trotz einer kompletten Verzweigungsabdeckung immer noch eine Kombination von Eingaben und Systemzuständen geben, die nicht getestet werden und zu einem Fehler führen.

2.5 Klassifikation von Fehlern

Fehler und Fehlerursachen bei Softwaresystemen werden anhand ihrer Merkmale in unterschiedliche Kategorien eingeteilt. Binder [Bin99] definiert wie folgt:

Failure Der Oberbegriff Failure beschreibt, wenn ein System oder eine Komponente eine in der Spezifikation geforderte Funktionalität nicht erfüllt. Dies kann sich durch Ausgabe falscher Daten, einem unerwarteten Programmabbruch oder durch das Überschreiten von Ressourcen oder zeitlichen Einschränkungen äußern.

Fault Ein Fault ist fehlender oder fachlich falscher Quellcode. Wird dieser Quellcode als Programm ausgeführt, kann das zu einem Failure führen.

[2] In der englischsprachigen Literatur, z.B [Bin99], als *line coverage*, *branch coverage* und *path coverage* bezeichnet.

Error Eine Interaktion oder Eingabe eines Benutzers, die zu einem Fault führt wird als Error bezeichnet.

Abend Unter einem Abend versteht man den unerwartet Abruch eines Programmes. Der Name Abend ist entstanden aus *abnormal end*.

Omission Unter Omission versteht man geforderte, aber nicht implementierte Funktionalität.

Surprise Wenn ein Programm oder ein System eine bestimmte Anforderung nicht unterstützt, obwohl sie gefordert und implementiert ist.

Bug Ein Bug ist eine andere Bezeichnung für einen Error oder Fault. Das Wort entstand, als 1945 von Grace Hopper eine Motte in einem defekten Computerrelay fand.[3]

2.6 Besonderheiten beim Testen von OO-Software

Das Testen objektorientierter Software unterscheidet sich in vielen Aspekten vom Testen prozeduraler Anwendungen. Auf diese Unterschiede und die daraus resultierenden Schwierigkeiten für das Testen soll in diesem Kapitel eingegangen werden.

Die Kapselung von Daten, die Polymorphie und das dynamische Binden sind u.a. Eigenschaften objektorientierter Sprachen wie Java. Diese Eigenschaften werden in Entwurfsentscheidungen genutzt. Paradoxerweise können diese Eigenschaften das Testen behindern. Hier soll angedeutet werden welche zusätzlichen Fehlermöglichkeiten beim Schreiben von objektorientierten Programmen bestehen können.

2.6.1 Kapselung von Daten

Die Kapselung der internen Repräsentation eines Objektes verringert das Risiko Fehler zu implementieren. Diese Kapselung erschwert aber das Testen. Der Zugriff auf die interne Repräsentation eines Objektes ist durch das *Geheimnisprinzip* auf die sondierenden Methoden eingeschränkt. Diese greifen auf die Attribute des Objektes zu. Tests können dann nur die Rückgabewerte einer Operation prüfen. Es bestehen dann keine Möglichkeiten die konkreten Attribute eines Objektes zu observieren. In diesem Fall ist man auf das *Blackbox-Testing* beschränkt.

[3] Dijkstra argumentiert, dass die Bezeichnung Bug eine Verherrlichung eines Fehlers ist und deshalb Error oder Failure genant werden sollte. Im deutschen Sprachgebrauch lässt sich das mit der passiven Formulierung „...da hat sich ein Fehler eingeschlichen...“ vergleichen. Auch hier wird angedeutet, dass nicht der Entwickler für diesen Fehler verantwortlich ist.

2.6.2 Vererbung

Vererbung ist eines der wesentlichen Merkmale objektorientierter Programmiersprachen. Vererbung kann es erschweren den sequentielle Ablauf eines Programmes zu verstehen, insbesondere wenn eine Klasse in einer großen Vererbungshierarchie steht.

Eine Fehlerursache kann die falsche Initialiserung eines Objektes sein, z.B. wenn eine Oberklasse im Konstruktor eine Methode zur Initialisierung benutzt. Eine Unterklasse kann diese Methode oder den Konstruktor der Oberklasse überschreiben, ohne die initialisierende Methode der Oberklasse aufzurufen. Damit verursacht die Unterklasse eventuell einen Fehler, denn Attribute der Oberklasse können falsch oder nicht initialisiert sein.

Einschubmethoden sind ein anderes Beispiel für mögliche Fehlerursachen im Zusammenhang mit Vererbung. Es kann zu Fehlern kommen, wenn in einer Unterklasse vergessen wird objektspezifische Methoden zu überschreiben. Wird z.B. eine Methode die die Gleichheit zweier Objekte überprüft, nicht an Spezialisierungen der Unterklasse angepasst, wird die Gleichheit der Objekte dann nur anhand der Merkmale der Oberklasse geprüft.

2.6.3 Dynamisches Binden

Durch Polymorphismus kann eine Referenz zur Laufzeit an unterschiedliche Klassen gebunden werden. Durch dynamisches Binden von Methoden an Objekte, ist erst zur Laufzeit erkenntlich, welche konkrete Methode aus welcher Klasse gerufen wird. Dadurch lässt sich der sequentielle Ablauf eines Programmes schwer nachvollziehen. Dies führt zu einer für das Testen unangenehmen Situtation. Zur Laufzeit ist nicht immer überschaubar, welche Methoden aus welcher Klasse beim Tests indirekt aufgerufen werden.

Ein weitere Auswirkung durch das dynamische Binden wird in [Bin99] als *YoYo-Effekt* erläutert. Hierbei handelt es sich um das Problem den Programmablauf eines Methodenaufrufes zu verstehen. Der YoYo-Effekt kann durch tiefe Vererbungshierarchien und häufige interne Benutzung entstehen. Um zu erkennen was eine Methode in einer Unterklasse genau macht, wenn sie Methoden der Oberklasse aufruft, muss diese angeschaut werden, also „hochgeschaut“ werden. Ruft diese wiederum Methoden am eigenen Objekt („self“) auf, so muss wieder „runtergeschaut“ werden in die Unterklasse usw. Hier kommt es durch einen einzelnen Methodenaufruf zu langen Ketten von Operationen am eigenen Objekt. Diese Ketten von Operationen in verschiedenen Klassen sind nur noch schwer nachzuvollziehen. Daher sind sie anfälliger für Fehler und schwerer zu testen.

2.6.4 Abdeckung

Wie in Kapitel 2.4 erläutert sind Testmetriken Maße für die Erfüllung der gewählten Teststrategie. Die dort eingeführten Abdeckungsmaße müssen unter der speziellen Sichtweise objektorientierter Programmierung gesehen werden.

Die schon am Anfang dieses Kapitels angegebenen Eigenschaften objektorientierter Programmiersprachen wie Vererbung, Polymorphie und Kapselung der Daten beeinflussen und erschweren die Erfüllung der verschiedenen Abdeckungsgrade. Für den Entwickler ist der Programmfluß schwerer nachzuvollziehen. Durch genannte Merkmale ist schon die Feststellung von komplexeren Abdeckungsgrade ein Problem. Im Java-Umfeld existieren zur Zeit Werkzeuge die Zeilenabdeckung und Verzweigungsabdeckung feststellen können.

Der Aufwand ein System nach einem bestimmten Abdeckungsgrad zu testen, ist im Vergleich zu dem Nutzen meist zu hoch. Auch werden häufig die Zusammenhänge zwischen Testmodell, Teststrategie und Abdeckung falsch verstanden. Das Testmodell identifiziert die Stellen an denen mit hoher Wahrscheinlichkeit Fehler auftreten können. An diesen Stellen orientiert sich die Teststrategie. Die Ziele der Teststragie bestimmen die Abdeckung und nicht umgekehrt.

In jeder Klasse können Fehler auftreten, also sollte auch jede Klasse getestet werden. Um eine Klasse komplett zu testen, schlägt [Bin99] folgende Anhaltspunkte vor:

- Jede Methode wird mindestens einmal ausgeführt.
- Alle Methodenparameter und exportierte Attribute werden mit Beispielen aus entsprechenden Äquivalenzklassen[4] und Grenzwerten getestet.
- Jede ausgehende Ausnahme wird provoziert, sowie jede eingehende Ausnahme wird behandelt.
- Jedes änderbare Attribute wird mit einem neuen Wert aktualisiert.
- Jeder Zustand wird erreicht. Dafür muss definiert sein, in welchen Zuständen sich das zu testende Objekt befinden kann.
- Jeder Zustandsübergang wird ausgeführt um Zusicherungen zu testen.

[4]Eine Zerlegung in Äquivalenzklassen liefert eine Aufteilung der Eingabewerte in endlich viele Teile. Alle Elemente einer Äquivalenzklasse besitzen eine gemeinsame Eigenschaft, sie führen zum gleichen beobachteten Ergebnis oder liefern Ergebniswerte in der gleichen Teilmenge des Ergebnisraumes.

- Angemessene Performanzkriterien, z.B. Antwortzeiten die eingehalten werden müssen, werden formuliert und getestet.

2.7 Zusammenfassung

Das Testen von Software ist ein wichtiger und nicht zu ersetzender Bestandteil der professionellen Softwareentwicklung. Das Ziel beim Testen von Software ist das Auffinden von Fehlern. Tests können nur die Anwesenheit von Fehlern entdecken, aber nie die Fehlerfreiheit eines Programmes beweisen.

Für ein zielorientiertes Testen ist das Verständnis der Begriffe Fehlermodell, Testmodell und Teststrategie wichtig. Ein Fehlermodell identifiziert Beziehungen oder Teile in einem System, wo mit hoher Wahrscheinlichkeit Fehler vermutet werden. Die Teststrategie legt fest, wie geprüft wird und welche Testfälle zu erstellen sind. Der Einsatz von Testmetriken, wie die verschiedenen Abdeckunsgrade, sollte sich an dieser Strategie orientieren und nicht umgekehrt. Sie können nur einen anhaltspunkt geben, ob genügend getestet wird.

Das Testen von Software ist immer im Kontext mit dem Vorgehensmodell und Entwicklungsprozesses zu betrachten. Jedes Vorgehensmodell beinhaltet Testen, häufig zu unterschiedlichen Zeitpunkten. Ein frühes Auffinden von Fehlern – und somit auch häufiges Testen – kann nur als Vorteil gelten. Der Entwicklungsprozess eXtreme Programming (siehe auch Kapitel 3.2) fordert sogar das Schreiben von Tests vor der eigentlichen Implementierung.

Bei der Entwicklung von Programmen sind die Besonderheiten der eingesetzten Programmiersprache zu beachten. Die Merkmale objektorientierter Programmiersprachen, wie Vererbung, Kapselung von Daten und das dynamische Binden können zusätzliche Fehlerursachen sein. Zudem erschweren sie die Messung von Abdeckunsgraden.

Kapitel 3

Testen in JWAM

Dieses Kapitel soll einen Überblick über den Kontext und die Ausgangslage für die Studienarbeit geben. Am Anfang wird ein kurzer Überblick über das Rahmenwerk JWAM gegeben, für das die Unterstützung interaktiver Tests entwickelt werden sollte. Dabei wird auch auf das zugrunde liegende Leitbild WAM[1] und den Entwicklungsprozess eXtreme Programming – abgekürzt XP – eingegangen. Die derzeit unterstützten Komponententests in JWAM und das dabei benutzte Testframework JUnit werden vorgestellt.

3.1 Überblick WAM und JWAM

Das Rahmenwerk JWAM (Abkürzung für Java-Rahmenwerk und WAM) wurde am Arbeitsbereich Softwaretechnik der Universität Hamburg von Studenten und wissenschaftlichen Mitarbeiteren entwickelt. Die bei der Entwicklung zugrunde liegenden Leitbilder enstammen dem WAM-Ansatz, beschrieben in [Zül98]. Das JWAM Framework wird durch die dem Fachbereich Informatik nahestehende Firma it-wps (ehemals Apcon Workplace Solutions) in Projekten in der Industrie konkret eingesetzt und weiterenwickelt.

3.1.1 Der WAM-Ansatz

Der WAM-Ansatz definiert verschiedene Leitbilder die den Entwicklern und den späteren Anwendern ein gemeinsames Verständins der Anwendungssoftware ermöglichen. In [GLL+99] wird das grundlegende Leitbild wie folgt beschreiben:

> Unser vorherrschendes Leitbild ist der gut ausgestattete Arbeitsplatz für eigenverantwortliche, kooperative Tätigkeiten.

[1] Abkürzung für Werkzeug-Automat-Material

> An diesem Arbeitsplatz sollen Werkzeuge, Automaten und Materialien so bereitgestellt werden, daß ein qualifizierter Anwender seine Arbeitsaufgaben situationsabhängig erledigen kann.

Es kann in den meisten Arbeitssituationen an einem Arbeitsplatz unterschieden werden zwischen den Gegenständen die bearbeitet werden (Materialien) und den Arbeitsmitteln (Werkzeuge). Ein Werkzeug unterstützt wiederkehrende Arbeitshandlungen und kann sinvoll für verschieden Zwecke und Aufgaben verwendet werden, dabei wird die Handhabung eines Werkzeuges immer durch den Benutzer veranlasst. Des weiteren gibt es bei WAM die Metaphern Automat und Arbeitsumgebung, auf die hier nicht weiter eingegangen werden soll.

Neben den Leitbildern und Metaphern beschreibt der WAM-Ansatz im Rahmen eines Vorgehensmodelles anwendungsorientierte Dokumenttypen. Der Ansatz fordert neben dem Grundprinzip des Autor-Kritiker-Zyklus eine dokumentbasierte Entwicklung, eine permanente Rückkopplung und einen Soll-Ist-Abgleich. Durch Interviews mit den späteren Anwendern soll deren Fachsprache erlernt und die Arbeitsgegenstände identifiziert werden. In den anwendundsorientierten Dokumenten werden die Ergebnisse festgehalten. Sie sind Grundlage für die Entwicklung und weitere Gespräche.

An dieser Stelle soll auf die Handhabungsvisionen weiter eingegangen werden. [Zül98] definiert: „Handhabungsvisionen beschreiben das Aussehen und die Verwendung des künftigen Anwendungssystems“. Handhabungsvisionen werden aus den Szenarios entwickelt. Szenarios beschreiben die aktuelle Arbeitssituation, insbesondere die Aufgaben in dem Anwendungsbereich. Eine Handhabungsvision beschreibt prosaisch die einzelnen Schritte des Anwenders im Umgang mit dem zunkünftigen System bei der Bearbeitung einer bestimmten Aufgabe. Diese Handhabungsvisionen können Grundlage für Akzeptanztests sein.

3.1.2 Das Rahmenwerk JWAM

Das in Java geschriebene Framework JWAM unterstützt die Entwicklung interaktiver Anwendungen nach dem WAM-Ansatz. Das Framwork setzt eine Schichtenarchitektur um. Die Schichtenarchitektur von JWAM ist vereinfacht in Abbildung 3.1 dargestellt. In der Technologieschicht werden anwendungsnahe Modelle von verwendeter Technik implementiert, sie abstrahiert von den konkreten eingesetzten Techniken z.B. zur Verteilung von Materialien und Nachrichten, der GUI-Anbindung oder Persistenzmedien. Die Handhabungs- und Präsentationsssschicht unterstützt den Bau von Werkzeugen, Materialen und deren Darstellung. In dieser Schicht befinden sich u.a. die Werkzeugkonstruktion, Materialkonstruktion und die Umgebung.

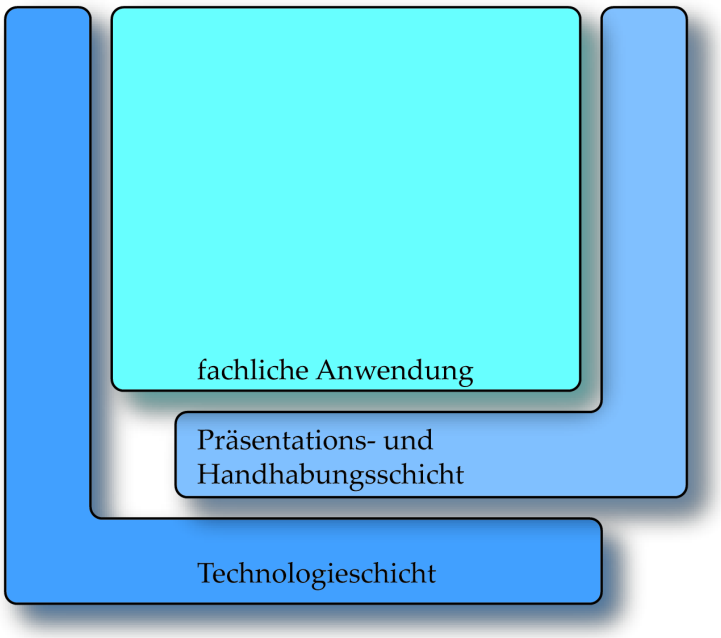

Abbildung 3.1: Vereinfachte Schichtenarchitektur von JWAM

3.2 Entwicklungsprozess mit eXtreme Programming

Bei der Entwicklung des Frameworks JWAM wurden verschiedene Prinzipien und Techniken des eXtreme Programmings adaptiert. XP ist ein leichtgewichtiger Softwareentwicklunsprozess, der zum ersten mal komplett vorgestellt wurde von Kent Beck in [Bec99]. Er ermöglicht es, Software zu konstruieren, wobei flexibel auf schnell ändernde Anforderungen reagiert werden kann. Die verschiedenen XP-Techniken *Kunde vor Ort*, *Planungsspiel*, *Metapher*, *kurze Releasezyklen*, *Testen*, *einfaches Design*, *Refactoring*, *Programmieren in Paaren*, *gemeinsame Verantwortlichkeit*, *fortlaufende Integration*, *Programmierstandards* und *40-Stundenwoche* basieren auf den XP-Prinzipien Einfachheit, Kommunikation, Feedback und Mut.

Auf die für diese Arbeit relevanten Techniken soll kurz eingegangen werden.

XP fordert bei der Entwicklung *kurze Releasezyklen*. Durch häufige Auslieferungen der zu erstellenden Software soll eine möglichst schnelle Rückkopplung über implementierte Eigenschaften erlangt werden. Die einzelnen Releases müssen den Anforderungen des Kunden genügen, einsetzbar und möglichst fehlerfrei sein. Es werden an dem System nur schrittweise kleine Änderungen durch die Entwickler vorgenommen. Das System muss schon zu jeder Integration den Ansprüchen genügen, möglichst fehlerfrei,

einsetzbar und funktionsfähig sein.

Testen ist ein Bestandteil von XP. Für jede neue Funktionalität im System soll ein Test geschrieben werden, am besten vor der eigentlichen Implementation. Diese Tests werden von den Entwicklern bei der Arbeit immer wieder ausgeführt. Sie müssen entsprechend den sich änderenden Anforderungen angepasst werden. Um die Tests zu struktieren, bietet es sich an, für jede Methode einer zu testenden Klasse einen Test zu schreiben. Diese Tests sollten in einem Testfall gebündelt werden, so dass für jede Klasse ein Testfall vorhanden ist. Die Tests müssen spätestens bei der Integration vollständig ausführbar sein. Hierbei handelt es sich um Komponententests die sich automatisch ausführen lassen sollten. Mit dieser Vorgehensweise soll verhindert werden, das durch Änderungen schon implementierte Funktionalität beeinträchtigt wird. Komponententest sind auch auch ein Hilfsmittel bei der Behebung von Fehlern. Für einen gefunden Fehler wird der Testfall um Tests erweitert die genau diesen Fehler entdecken. Ist der Fehler behoben, müssen die Tests durchlaufen. Es wird damit auch vermieden, das die gleichen Fehler erneut implementiert werden. Die Tests sollten häufig ausgeführt werden, idealerweise alle paar Minuten. Durch das Schreiben von Tests vor der Implementation wird die Entwicklung als *testgetrieben* beschrieben.

Zudem sind die Tests eine erste Benutzung der zu erstellenden Schnittstelle. Der Entwickler bekommt dadurch eine schnelle Rückkopplung, wie sich die neu erstellte Klasse verhält. Die Klasse wird dabei aus der Sicht einer nutzenden Klasse betrachtet. Mit dem Ansatz vor der Implementierung Tests zu schreiben, sollten Klassen leicht zu testen sein. Da es einfacher ist, eine gut strukurierte Klassen zu testen, werden komplexe und fehleranfällige Entwürfe vermieden.

Die Komponententests sind auch Basis bei der Weiterentwicklungen und für *Refactoring*[2]. Wird von den Entwicklern erkannt, das Teile der Anwendung umständlich oder fehleranfällig implementiert oder die Architekur für eine Weiterentwicklung zu unflexibel ist, wird das Problem mit einer Umstrukturierung, also einem *Refactoring* behoben.

Die Kombination von fortlaufenden Integrationen, die jeweils erfolgreich getestet sein müssen, und eine hinreichenden Testabdeckung, erleichtert die Integrationstests innerhalb eines zusammenhängenden Systems.

Bei der Entwicklung von JWAM wird das in Kapitel 3.3 beschriebene JUnit in einer für JWAM angepassten Version für die Erstellung von Komponententests benutzt. Das während dieser Arbeit entstandene Buch [LRW02] beschreibt detailiert XP-Techniken die u.a. bei der Entwicklung JWAM eingesetzt wurden.

[2] kontinuierliche Software-Restrukturierung

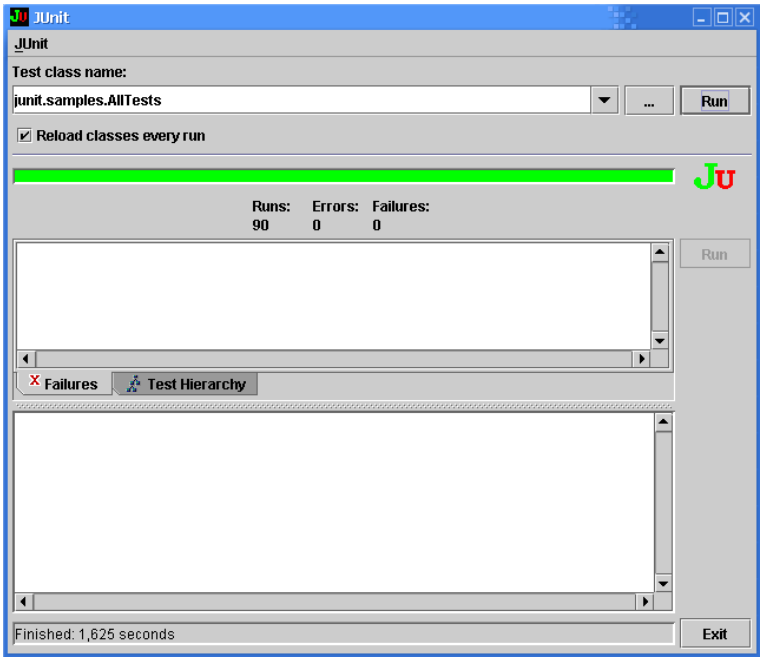

Abbildung 3.2: Der grafische TestRunner von JUnit

3.3 JUnit

Das in Java geschrieben Testframework JUnit wurde zuerst vorgestellt von Kent Beck und Erich Gamma in [GB98]. Mit dessen[3] Hilfe können mit der Programmiersprache Java Komponententests geschrieben und automatisch ausgeführt werden. Somit können direkt mit der selben Entwicklungsumgebung wie sie für die Anwendungsentwicklung benutzt wird, Tests geschrieben werden. Diese Tests können mit dem sogenannten `TestRunner` von JUnit automatisch ausgeführt werden. Der `TestRunner` von JUnit ist in Abbildung 3.2 dargestellt. Der grüne Balken stellt den Fortschritt beim erfolgreichen Testen dar. Wenn er sich rot färbt ist mindestens ein Test fehlgeschlagen. In dem unteren Feld wird angezeigt in welchem Test und in welcher Codezeile der Fehler aufgetreten ist.

Jeder Testfall wird duch eine eigene Klasse abgebildet. Diese Klasse muss von der JUnit-Klasse `TestCase` erben. Jeder Test wird in dem Testfall durch eine eigene Methode geschrieben, die anhand ihrer Signatur erkannt wird. Methoden, die mit dem Präfix `test` beginnen, den Rückgabewert `void` und eine leere Parameterliste haben, werden als Tests ausgeführt. Die Klasse `TestCase` stellt mit der Methode

```
public void assert (boolean cond)
```

[3] Korrektur 'einer' nicht verstanden...

```
public void testKaffeeMaschineAnmachen ()
{
  _kaffeMaschine.anschalten();
  assert(_kaffeMaschine.istAngeschaltet());
}
```

Listing 3.1: Beispiel für eine Testmethode

den Aufruf für den eigentlichen Test zur Verfügung. Innerhalb eines Tests wird diese Methode mit der Prüfung eines erwarteten Zustandes oder eines Rückgabewertes aufgerufen. In Abhängigkeit vom Zustand des boolschen Parameters `cond` gilt der Tests als bestanden oder als nicht erfolgreich. Als simples Beispiel soll hier ein Ausschnitt aus einem Test für eine Klasse dienen, die eine Kaffeemaschine abbildet. Nachdem man diese anschaltet, wird erwartet, das sie sich im Zustand „angeschaltet“ befindet.

Wird der Test ausgeführt und die Maschine befindet sich nicht in diesem Zustand, wird eine Exception erzeugt. Diese wird durch den TestRunner abgefangen. Der TestRunner signalisiert den fehlgeschlagenen Test und der Fortschrittsbalken wird rot dargestellt. Mit den Stacktraces der geworfenen Exception wird der fehlgeschlagene Test im unteren Detailfenster des Testrunners dargestellt.

Für die einfachere Handhabung bietet die Klasse `TestCase` zusätzlich folgende Methoden zum Testen, die sich alle auf die Methode `assert` zurück führen lassen:

- `assertEquals(Object expected, Object actual)`

  Diese Methode prüft die zwei (als Referenz) übergebenen Objekte, `expected` und `actual`, auf Gleichheit. Dabei wird die Methode `equals` benutzt, die an jedem Objekt in Java definiert ist. Sind die beiden Objekte nicht gleich, ist der Test nicht bestanden. Die Methode ist eine Kurzform von `assert(expected.equals(actual))`. Zusätzlich wird vorher geprüft ob eines der beiden Refenzen auf `null` verweist und im Fehlerfall werden das erwartete und das aktuelle Objekt mittels der Methode `toString` zur einfacheren Fehlerdiagnose der Fehlermeldung hinzugefügt.

- `assertSame(Object expected, Object actual)`

  Hier wird auf die Gleicheit der übergebenen Referenzen getestet. Es wird also erwartet, das die beiden Objekte die selben sind. Ein andere Art dieses auszudrücken wäre `assert(expected == actual)`.

```
try {
  obj.teile(24, 0);
  fail("Keine IllegalArgumentException geworfen!");
} catch (IllegalArgumentException ignore) {
  // Diese Exception wurde erwartet.
}
```

Listing 3.2: Beispiel der Methode `fail`

- `assertNotNull(Object expected)`

  Es wird bei diesem Test erwartet, das die übergebene Referenz auf ein Objekt zeigt und nicht `null` ist.

- `fail (String message)`

  Diese Methode kann genutzt werden, wenn im Programmfluss des Tests ein Punkt erreicht wird, der nur im Fehlerfall erreicht werden kann. Die Methode löst sofort eine Exception aus, mit der übergebenen Fehlermeldung. Ein Ausschnitt aus einem Test soll hier als Beispiel dienen. Mit einem ungültigen Parameter wird eine Exception provoziert. Wird die Exception bei der Ausführung des Tests nicht geworfen, kommt der Programmfluss zu der Methode `fail` und signalisiert einen Fehler. Ein Beispiel hierfür wird in Listing 3.2 gezeigt.

Die oben angegeben Methoden gibt es jeweils noch mit einem `String`-Parameter. Es kann dann ein zur Fehlerdiagnose nützlicher Text, z.B. eine knappe Beschreibung des Tests, mit angegeben werden.

Das Testwerkzeug von JUnit ist in JWAM erweitert. Es wird dort die Bündelung von mehreren Testfällen erheblich erleichtert. Bei JUnit müssen hierfür die Testfälle explizit im Programmcode einer `TestSuite` hinzugefügt werden. In JWAM lassen sich alle Tests, deren Klassennamen mit „_`Test`“ enden automatisch einsammeln und sofort ausführen.

3.4 Komponententests

Bei der Entwicklung von JWAM werden Komponententests erstellt mit der Unterstützung des in Kapitel 3.3 vorgestellten Testframeworks JUnit. Die Entwickler von JWAM halten sich an den Leitsatz von XP, für jede Funktionalität einen Testfall zu erstellen. Jeder Testfall hat als Gegenstand seiner Tests ein Exemplar der zu testenden Klasse, bzw. ein Objekte das das zu testende Interface implementiert. Jede Methode dieser Klasse sollte mit mindestens einem Test aufgerufen und geprüft werden. Die Tests einer Klasse zusammengefasst ergeben den Testfall dieser Klasse.

Es sollte vermieden werden, in diesem Testfall auch Anforderungen einer anderen Klasse zu überprüfen. Benutzt die zu testende Klasse Exemplare einer anderen Klasse, so werden von dieser zwar implizit auch Methoden aufgerufen, diese sollten aber durch eigene Testfälle ausreichend abgedeckt werden.

Wenn Klassen von anderen Klassen aus dem Framework erben, so müssen sie auch den Testfall der Oberklasse durchlaufen. Gerade bei einem Framework impliziert diese Forderung, das die Testfälle für Klassen, von denen geerbt werden soll, auch mit Exemplaren der Unterklassen ausgeführt werden können. Das Framework JWAM wird von Entwicklern genutzt, die mit dessen Hilfe Anwendungen programmieren. Sie können mit diesen Testfällen Exemplare eigener Klassen, die vom Framework erben, testen.

Vor jeder Integration müssen sämtliche Tests durchlaufen werden. Nur wenn diese erfolgreich sind, kann integriert werden. Bei der Entwicklung von JWAM wird eine Erweiterung der Sourcecodeverwaltung CVS, der *IntegrationGuard* eingesetzt, siehe [Int03]. Der IntegrationGuard führt bei einer Integration automatisch alle Komponententests aus. Nur wenn diese erfolgreich durchlaufen, wird die Integration akzeptiert. Bei einem Fehler wird die Integration abgewiesen. Durch diese Vorgehensweise soll verhindert werden, das durch Änderungen und Erweiterungen am Framework bestehende Funktionalität beeinträchtigt wird. Da Komponententests gesammelt und immer wieder ausgeführt werden, haben sie somit auch die Funktion von Regressionstests.

3.5 Zusammenfassung

In diesem Kapitel wurde in kurzer Form der Kontext der Studienarbeit vorgestellt, die Konstruktionsprinzipien nach den WAM-Leitbildern und der testgetriebene Entwicklungsprozess eXtreme Programming. Diese werden bei der Entwicklung von JWAM und Projekten der Firma it-wps eingesetzt. Das Testframework JUnit und Erweiterungen für JUnit in JWAM sind beschrieben worden. In JWAM gibt es bisher keine Unterstützung für automatische Akzeptanztets.

Kapitel 4

Konzeption

In diesem Kapitel soll die Konzeption einer Akzeptanztestunterstützung erläutert werden. Die Grundlagen des Testens von Software wurden vorgestellt. Der Kontext dieser Studienarbeit, die WAM-Leitbilder, das JWAM-Framework und das Testframework JUnit sind beschrieben worden.

In der vorhanden Form lassen sich mit der angepassten Version von JUnit Komponententests (siehe Kapitel 2.2.1 und 3.4) für Java-Klassen erstellen und automatisch ausführen. Für Akzeptanztests gibt es in JUnit und der Anpassung in JWAM keine direkte Unterstützung.

Um das Konzept hinter der praktischen Lösung für diese Studienarbeit zu verstehen, wird auf die fachlich motivierte Modellarchitektur in JWAM und deren Konstruktion eingegangen. Es wird erläutert, welche Besonderheiten beim Testen interaktiver Werkzeuge zu beachten sind.

Es wird auf die technischen Erweiterungen eingegangen, die für Akzeptanztests notwendig sind. Die Akzeptanztests sollen automatisch ausgeführt werden können, es werden aber nur bestimmte Interaktionen des Benutzers durch den vorgestellten Ansatz automatisiert werden können. Nicht-funktionale Anforderungen müssen weiterhin händisch oder mit anderen Werkzeugen getestet werden. Funktionale Tests, wie Perforamceprüfungen, werden in dieser Arbeit nicht weiter betrachtet. Des weiteren wird der Umgang mit dem neu erstellten Testwerkzeug aufgezeigt.

4.1 Testen interaktiver Werkzeuge

Bei der Konstruktion interaktiver Arbeitsplatzsysteme geht die Interaktion vom Benutzer aus. Interaktive Werkzeuge reagieren auf Benutzeraktionen und sind somit reaktive Systeme. Um ein interaktives Werkzeug zu testen, muss es zuerst in den gewünschten Zustand gebracht und dann mit Eingaben versehen werden. Dabei muss geprüft werden, ob das Werkzeug sich wie erwartet verhält. In der Praxis sieht ein solcher Test häufig so aus, das das Werkzeug „von Hand“ gestartet wird, initiiert durch den

Entwickler oder einen Tester. Für diesen Zweck können Drehbücher geschrieben werden, in denen die Aktionen des Benutzers und das erwartetete Verhalten des Werkzeuges beschrieben sind. Diese Dokumente können abgeleitet werden von den Szenarios und Handhabungsvisionen des Entwicklungsprozesses. Die Nachteile dieser Herangehensweise werden deutlich: Die Dokumente, die die Akzeptanztests beschreiben, müssen parallel zum Sourcecode verwaltet werden und die Tests müssen immer wieder händisch ausgeführt werden. Es bietet sich in diesem Bereich eine Automatisierung an.

4.2 Werkzeuge nach dem WAM-Ansatz mit JWAM

Im WAM-Ansatz wird unterschieden in Materialen und Werkzeuge. Werkzeuge sind Arbeitsmittel die andere Arbeitsgegenstände, Materialen, bearbeiten können. Ein interaktives Werkzeug besteht aus einem interaktiven und einem funktionalen Teil. Dieses Entwurfsmuster wird in [Zül98] vorgestellt und *Trennung von Funktion und Interaktion* genannt. Die Funktionskomponente bietet die Möglichkeit, das Material unabhängig von der Darstellung zu bearbeiten. Die Interaktionskomponente des Werkzeuges hat Referenzen auf die Funktionskomponente und realisiert Präsentation und Handhabung des Werkzeuges. Benutzereingaben werden durch die Interaktionskomponente interpretiert.

Ist die Benutzeraktion anwendungsfachlich motiviert und verändert sie nicht nur die Darstellung des Werkzeuges, dann wird die Benutzeraktion mit Methodenaufrufen an die Funktionskomponente umgesetzt. Durch lose Kopplung[1] reagiert es auch auf Veränderungen am Zustand der Funktionskomponente und passt entsprechend die Darstellung an. Das gesamte Werkzeug wird durch ein eigenes Objekt repräsentiert, das Werkzeugobjekt. Bei der Aktivierung eines Werkzeugobjektes wird zuerst ein Exemplar der Funktionskomponente erzeugt. Dieses wird dann der Interaktionskomponente bei deren Erzeugung im Konstruktor übergeben.

Mit JWAM gibt es auch die Möglichkeit monolithische Werkzeuge zu bauen, die sogenannten Monotools. Hier gibt es nur ein Objekt, das die Aufgaben der funktionalen und interaktiven Komponente, und des Werkzeugobjektes übernimmt. Diese Monotools sind nützlich bei der Erstellung von Prototypen in einer frühen Phase eines Projektes. Monotools werden in dieser Arbeit nicht weiter betrachtet.

Für die interaktve und die funktionale Komponente, sowie das Werkzeugobjekt eines Werkzeuges gibt es in JWAM ensprechende Oberklassen. Dies ist beispielhaft anhand eines Kassierwerkzeuges in Abbildung 4.1 dargestellt. Die Beziehungen zwischen Werkzeugobjekt, Funktions- und Interakstionskomponente sind zu erkennen. Grundlegende Entwurfsmuster

[1] z.B. mit dem *Ereignismuster*, siehe auch[GHJV95]

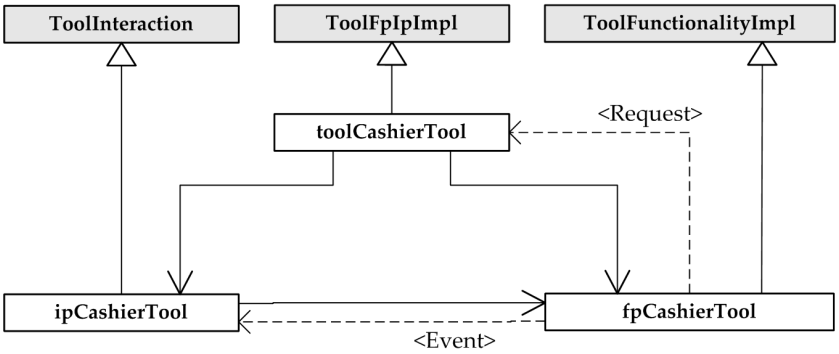

Abbildung 4.1: Die Konstruktion eines Werkzeuges mit JWAM

der Werkzeugkonstruktion mit JWAM sind in [Zül98] zu finden und detailliertere Beschreibungen werden in der Dokumentation zu dem Framework JWAM [JWA02] und dem einführenden Artikel [BGL+99] gegeben.

4.3 Interaktions- und Präsentationsformen

Für den Lösungsansatz dieser Arbeit ist das Konzept der Trennung von Interaktion- und Präsentationsform von grundlegender Bedeutung. Ein Ziel dieses Konzeptes ist dabei laut [BGL+99] „Fenstersystembibliotheken anbinden bei größtmöglicher Unabhängigkeit vom konkreten Fenstersystem“. Das Ziel wird erreicht über Abstraktionen des konkreten GUI-Toolkits mit *Interaktionsformen*[2], die als IAF abgekürzt werden. In der Abbildung 4.2 sind beispielhaft Verknüpfungen der Interaktions- und Präsentationsform und der gekapselten Oberflächenelemente dargestellt. So ist ein `pfJButton` ein Aktivator, er implementiert das Interface `ifActivator`. Der `pfJButton` ist an der Oberfläche ein Button, dafür erbt er von `JButton` aus Swing.

Ein Beispiel für eine Interaktionsform ist der *Aktivator*. Jedes Oberflächenelement das aktiviert werden kann, z.B. ein Button oder ein Element aus einer Liste, kann ein Aktivator sein. Dieser Aktivator hat nur einen fachliche Interaktion, er kann aktiviert werden. Der Button kann z.B. gedrückt oder das Listenelement doppelt angeklickt werden. Die Interaktionsform wird mittels einem oder einer Kombination aus GUI-Elementen an der Werkzeugoberfläche präsentiert. Die konkrete Darstellung an der Oberfläche durch GUI-Elemente wird Präsentationsform der Interaktionsform genannt. In JWAM werden Interaktionsformen durch Interfaces realisiert. Die Präsentationsformen implementieren dieses Interfaces. Bei der

[2] In [Zül98] auch *Interaktionstypen* genannt

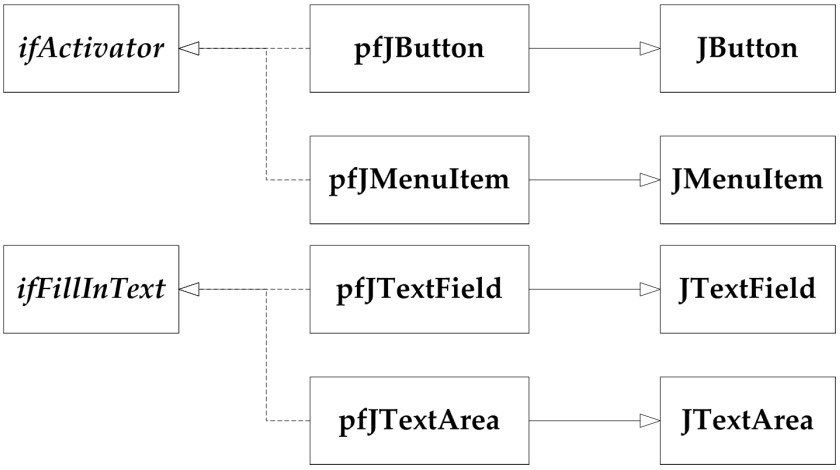

Abbildung 4.2: Interaktionsformen, zugehörige Präsentationsformen und gekapselte GUI-Elemente

Programmierung der Interaktionskomponente eines Werkzeuges wird nur über die Schnittstelle der Interaktionsform auf die Oberfläche des Werkzeuges zugegriffen.

Die Interaktionskomponente kann die Interaktionsformen direkt von GUI-Implementierung, der konrketen Präsentation, beziehen. Um eine bessere Trennung zwischen der Interaktion und der benutzten Präsentation zu bekommen, sollte hierfür ein Interaktionskontext benutzt werden. Ein Interaktionskontext abstrahiert von der benutzten GUI-Oberfläche und liefert der Interaktionskomponente Interaktionsformen anhand von Namen. Somit besitzt die Interaktionskomponente keinerlei Kenntnis über die Realisierung der Darstellung. Der Interaktionskontext wird in Kapitel 4.5 näher erläutert. Die Beziehungen zwischen Interaktionskontext, Präsentations- und Interaktionsformen werden in Abbildung 4.3 generell verdeutlicht. Es hat sich als Konvention herausgestellt, dass die Namen der Schnittstellen von Interaktionsfomen mit dem Präfix '`if`', die Namen der Interaktionskomponenten mit dem Präfix '`ip`' und die Namen der Präsentationsformen mit dem Präfix '`pf`' anfangen. Ein Interaktionskontext ist in JWAM durch die Klasse `IAFContextImpl` realisiert.

Eine Aktion des Benutzers am Werkzeug muss von der Interaktionsform an die Interaktionskomponente gemeldet werden. Die Interaktionsform kennt die Interaktionskomponente nicht, daher werden *Commands* benutzt. Commands kapseln einen Befehl. Ein Command-Objekt kann bei der Interaktionsform für ein Ereignis angemeldet werden. So kann z.B. bei einem Activator ein Command angemeldet werden, das bei Aktiverung ausgeführt wird. Für diesen Zweck definiert die Klasse `cmdObject` aus

```
ifActivator exit = (ifActivator)
  iafsContext.interactionForm(ifActivator.class, "exit");
exit.attachActivateCommand(new cmdObject()
{
  public void doExecute ()
  {
    exit();
  }
});
```

Listing 4.1: Anmelden eines Commands

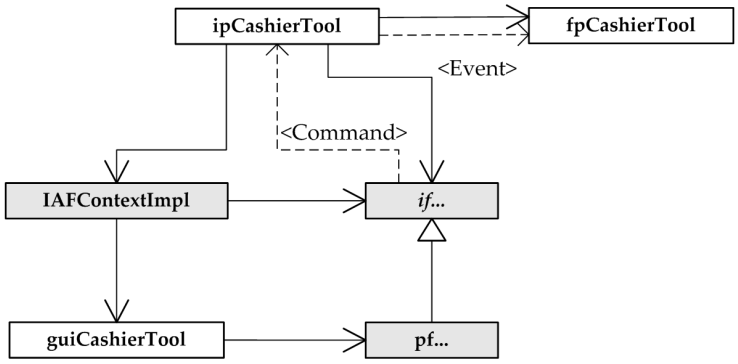

Abbildung 4.3: Frameworkunterstützung für Interaktionsformen

JWAM die folgende Methode:

```
public final void execute (Object commander) { ...
```

Die Interaktionskomponente meldet bei der Interaktionsform Commands an, um auf eine Aktion des Benutzers zu reagieren. Die Interaktionsform kennt nur das Command, nicht direkt die Interaktionskomponente. Diese lose Kopplung wird in Abbildung 4.3 durch die gestrichelte Linie zwischen Interaktionsform und Interaktionskomponente verdeutlicht.

Es gibt die Methoden zum An- und Abmelden von Commands. In Listing 4.1 wird gezeigt, wie an den Activator mit dem Namen „exit“ ein Command angemeldet wird.

Die Aktivierung selber wird vom Benutzer durch Anklicken oder Auswählen des GUI-Widgets ausgelöst. Das bei `attachActivateCommand` übergebene Command wird mit der Methode `execute` bei der Aktivierung des Aktivators durch die Präsentationsform aufgerufen. Von der Methode `execute` aus der abstrakten Klasse `cmdObject` wird die abstrakte Einschubmethode `doExecute` ausgeführt.

Des weiteren gibt es Interaktionsformen für die Auswahl eines oder mehrerer Elemente aus einer Menge, die Eingabe oder Darstellung eines

```
/**
 * Returns the name of the interaction form with the
 * given type.
 * @require ifType != null
 * @require ifType.isAssignableFrom(this.getClass())
 * @ensure result != null
 */
public String ifName (Class ifType);
```

Listing 4.2: Signatur der Methode `ifName`

Fachwertes und zur Unterstützung von Drag&Drop.

Eine Präsentationsform kann mehrere Interaktionsformen implementieren. So kann eine Liste zum einen die Interaktion „Auswahl eines Elementes“ aus einer Menge von Elementen anbieten, zum anderen kann sie bei einem Doppelklick auf eines der Elemente aktiviert werden.

Für die verschiedenenen Interaktionsformen gibt es Interfaces, die alle von dem Interface `ifInterface` erben. Eine wichtige Methode dieses Interfaces ist `ifName`, gezeigt in Listing 4.2.

Diese Methode ist notwendig zur Unterscheidung, wenn eine Präsentationsform mehrere Interaktionsformen anbietet. Zum Beispiel kann eine Liste die Interaktion *aktivieren* anbieten, wenn man sie anklickt, und gleichzeitig *selektieren*, wenn man ein Element auswählt. Unterschiedliche Interaktionsformen haben an der gemeinsamen Präsentationsform unterschiedliche Namen. Anhand des verlangten Typen `ifType` der Interaktionsform, wird der entsprechende Name zurück gegeben. Es ist eine Vorbedingung der Methode `ifName`, das der angefragte Typ Interaktionsform auch von der Klasse implementiert wird.

Interaktionsformen beinhalten Informationen über Art und Weise der Interaktion mit dem Programm. Diese lassen sich mit Methodenaufrufen direkt abfragen.

4.4 Dummy-Objekte für Widgets

Die in einem vorhanden System benutzten Präsentationsformen können aus mehreren Gründen nicht für Tests interaktiver Systeme genutzt werden.

Die Interaktionsformen beinhalten nur die Methoden die von der interaktiven Werkzeugkomponente zur Sondierung der Benutzerinteraktion benötigt werden. So gibt es Methoden, die eine Eingabe oder den Zustand einer Interaktion abfragen und zusätzlich Methoden zum Anmelden von Commands. Es fehlen Methoden die Aktionen des Benutzers simulieren. Für Akzeptanztests müssen diese Aktionen aus dem Programmfluss

des Tests gerufen werden können. Ausserdem sollten Akzeptanztests auch schon lauffähig sein, wenn noch gar keine GUI vorhanden ist.

Zudem bietet nicht jedes GUI-Toolkit die Möglichkeit Benutzerinteraktion zu simulieren.

Um für die Akzeptanztests Interaktionsformen mit den geforderten Eigenschaften bereit zu stellen, wurde in dieser Arbeit das Konzept der *Dummy-Objekte* aufgegriffen. Dummy-Objekte werden vorgestellt in [MFC00]. Sie deuten eine Funktionalität nur an, ohne sie vollends zu implementieren.

Für diese Arbeit sollen Präsentationsformen gebaut werden, die die Schnittstelle der Interaktionsformen implementieren. Zusätzlich werden für diese speziellen Präsentationsformen Methoden erstellt, die Benutzereingaben simulieren. Da diese speziellen Interaktionsformen keine GUI benutzen und nicht sichtbar sind, werden sie *faceless widgets* genannt. Mit dieser Lösung besteht eine Unabhängikeit von dem Vorhandensein einer GUI oder deren konkreten Implementation. Entsprechend den Namenskonventionen in JWAM beginnen die Klassennamen der *faceless widgets* mit dem Präfix '`fl`'. Das *faceless widget* für die Interaktionsform `ifActivator` wird also `flActivator` genannt.

4.5 Interaktionskontext für Akzeptanztests

In JWAM benutzen Interaktionskomponenten für den Zugriff auf die Oberfläche Interaktionsformen. Diese werden mittels Präsentationsformen realisiert. In dem Werkzeugobjekt wird bei der Initialisierung die GUI erzeugt und einem *Interaktionskontext* im Konstruktor übergeben. Die Interaktionskomponente eines Werkzeuges wird dann mit mit diesem Interaktionskontext initialisiert. Um auf Interaktionsformen zuzugreifen, benutzt der Interaktionskomponente die Methode `interactionForm` aus dem Interface `IAFContext`.

Bei dieser Methode sucht der Interaktionskontext – abstrahiert in dem Interface `IAFContext` – nach der Präsentationsform mit dem übergebenen Namen und dem übergegeben Typ. Als Vorbedingung muss diese Interaktionsform dem Interaktionskontext bekannt sein. Das wird mit der Methode `hasInteractionForm` abgefragt und über das Vertragsmodell zugesichert. In einer konkreten Implementierung dieser Schnittstelle – z.B. in einem Interaktionskontext für Oberflächen, die mit Swing[3] gebaut sind – wird die GUI rekursiv nach konkreten Interaktionsformen mit den gewünschten Eigenschaften durchsucht.

Ein typischer Aufruf in der Initialisierung der interaktiven Komponente, um sich die Interaktionsform des Typs `ifActivator` mit dem Namen

[3]Standard GUI-Toolkit im Java Development Kit

```
/**
 * Return the interaction form with the given type and name
 *
 * @param name The name of the required interaction form
 * @param ifType The type of the required interaction form
 * @return the interaction form with the given name
 *
 * @require name != null
 * @require ifType != null
 * @require hasInteractionForm(ifType, name)
 * @ensure result not null
 */
public ifInterface interactionForm (Class ifType, String name);
```

Listing 4.3: Signatur der Methode `interactionForm`

„exit“ von dem Interaktionskontext zu besorgen und mit einem Befehlsobjekt auszustatten ist in Kapitel 4.3 dargestellt.

Um in dieser Arbeit Akzeptanztests für die Interaktion mit einem Werkzeug zu realisieren, werden *faceless widgets* benötigt. Das Vorhandensein einer konkreten GUI für die Akzeptanztests soll nicht unbedingt notwendig sein. Um dieses Problem zu lösen, wird in dieser Arbeit ein Interaktionskontext zum Testen interaktiver Anwendungen gebaut, der abhängig von dem Typ der geforderten Interaktionsform ein entsprechendes *faceless widget* liefert, das diese Interaktionsform implementiert.

4.6 Umgang mit dem Testwerkzeug

Das bisher genutzte Testwerkzeug von JUnit wurde schon in Kapitel 3.3 beschrieben. Bei dem `TestRunner` von JUnit wird mit einem Fortschrittsbalken die Anzahl der gelaufenen Tests dargestellt. Im Anfangszustand ist der Balken grün gefärbt, schlägt ein Test fehl, signalisiert das der Balken mit einer roten Färbung. Dies ist in Abbildung 4.4 dargestellt. Es werden dabei keine Unterscheidungen zwischen verschiedenen Typen von Tests gemacht. Die Anzahl der ausgeführten Tests und die Anzahl der Fehler wird dargestellt. Das untere Fenster zeigt diese Fehler in einer Liste an. Mit einem Doppelklick auf eines der Elemente der Fehlerliste wird ein modales Fenster geöffnet, in dem der Stacktrace der Exception angezeigt die den Fehler verursacht hat.

Akzeptanztests unterscheiden sich von Komponententests. Sie haben nicht die strenge Bedingung zu jeder Zeit fehlerfrei durchlaufen zu müssen. Sie sollten mindestens bei einem Release erfolgreich durchlaufen. Um dem Benutzer eine Unterscheidung dieser beiden unterschiedlichen Testtypen zu signaliseren, muss das Testframework JUnit erweitert werden. Das

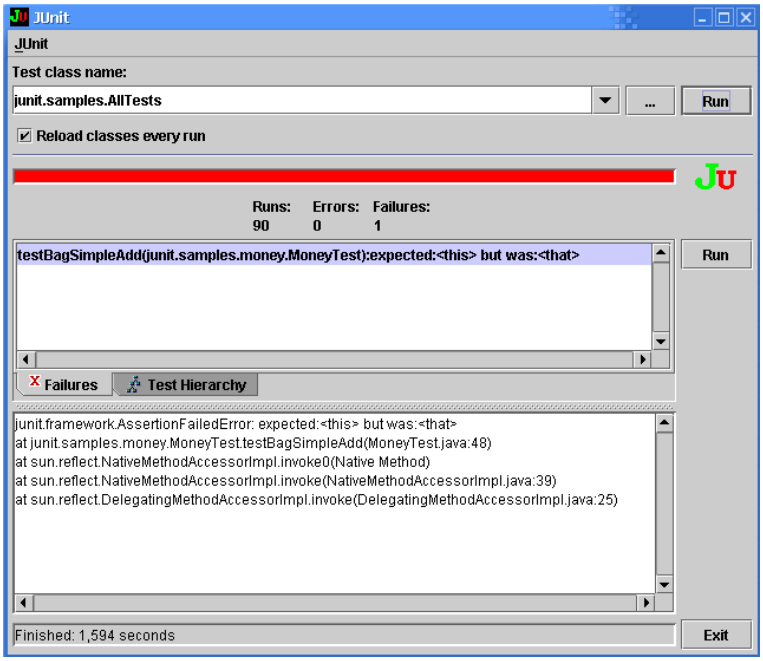

Abbildung 4.4: JUnit bei einem fehlerhaften Test

von dem Benutzer genutzte Testwerkzeug soll die Sammlungen der beiden Testtypen getrennt voneinander in einem einzelnen Werkzeug starten können. Angelehnt an das Testwerkzeug von JUnit soll sich die Bedienung des neuen Werkzeuges in der Handhabung nicht verändern. Es soll dabei einen Button zum Starten der Tests und einen je nach Erfolg gefärbten Fortschrittsbalken haben, der den Zustand des Testens signalisiert. Ausserdem wird ein Fenster benötigt, dass Details über die aufgetretenen Fehler anzeigt. Das bekannte Testwerkzeug soll in dem neuen Testwerkzeug zweimal vorhanden sein, einmal für Komponenten- und einmal für Akzeptanztests. Dabei soll der Benutzer auch sehen können, ob er Akzeptanztests oder Komponententests ausführt. Im Falle eines Fehlers bei den Akzeptanztests soll der Fortschrittsbalken sich nicht von grün in rot, sondern von grün in gelb verfärben. In Abbildung 4.5 ist die Oberfläche des neuen Testwerkzeuges skizziert.

Für die Realisierung dieses Testwerkzeuges von JUnit muss die Oberklasse für Testfälle in JUnit `TestCase` angepasst werden, um eine Unterscheidung zwischend den beiden unterschiedlichen Testtypen machen zu können. Wenn die Tests durch die Erweiterung von JWAM automatisch eingesammelt werden, müssen verschiedenen Typen von Tests erkannt werden. Die Komponententests und Akzeptanztests müssen automatisch in entsprechende Sammlung von Tests hinzugefügt werden können.

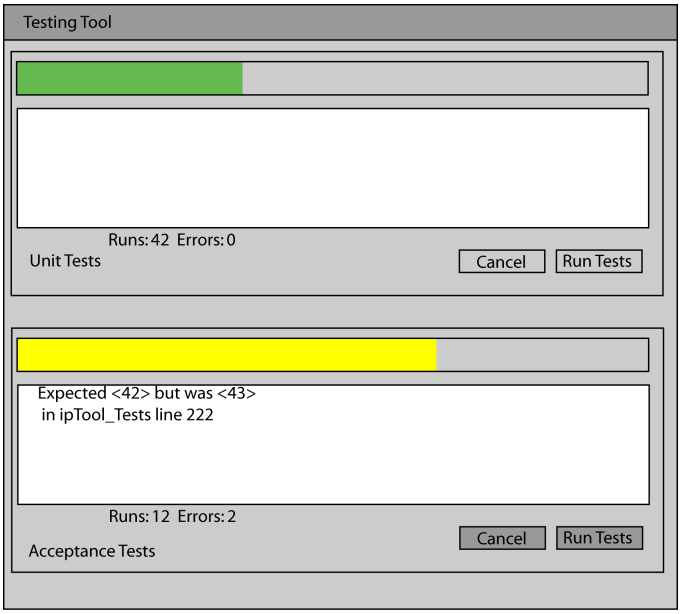

Abbildung 4.5: Skizze der GUI für das neue Testwerkzeug

4.7 Zusammenfassung

In diesem Kapitel wurden die Vorrausetzung für die praktische Realisierung der Unterstützung für Akzeptanztests in JWAM genauer betrachtet. Die Probleme beim Testen interaktiver Werkzeuge wurden dargestellt. Es wurde die Konstruktion von Werkzeugen in JWAM beschrieben. Anhand der durch die WAM-Metapher beschriebenen und in JWAM eingesetzten Muster zum Bau interaktiver Werkzeuge ist der Lösungsansatz dieser Arbeit vorgestellt worden. Um Akzeptanztests für interaktive Anwendungen zu realisieren, sind *faceless widgets*, ein spezieller Interaktionskontext für Akzeptanztests notwendig. Abschliessend wurde die Handhabung des neu zu erstellenden Testwerkzeuges beschrieben.

Kapitel 5

Realisierung der Akzeptanztests

Dieses Kapitel beschreibt die Umsetzung der Konzepte aus dem vorigen Kapitel. Der praktische Teil dieser Studienarbeit gliederte sich in drei Teile. Zuerst werden die *faceless widgets* programmiert und getestet. Um Akzeptanztests für JWAM-Werkzeuge zu bauen wird noch ein spezieller Interaktionskontext benötigt, der Testkontext. Abschliessend wird Junit erweitert und das Testwerkzeug realisiert, mit dem sich u.a. Komponententests von Akzeptanztests unterscheiden lassen.

5.1 Faceless Widgets

Faceless widgets sind nicht nur leere und funktionslose Implementationen der Schnittstelle der jeweiligen Interaktionsform. Bei den Interaktionen lassen sich Commands anmelden, wie schon in 4.3 mit dem Interface `cmdObject` beschrieben. Die *faceless widgets* bieten zusätzlich Methoden an, die nicht in der Schnittstelle der zugehörigen Interaktionsform sind, um Benutzeraktionen zu simulieren.

Ist an einem *faceless widget* ein Command angemeldet, und dieses Ereignis wird ausglöst, dann muss das *faceless widget* auch an diesem die Methode `execute` aufrufen. Dabei ist zu beachten, das nur durch den Programmfluss, im Regelfall also den Test, durch einen Methodenaufruf an dem *faceless widget* das Ereignis ausgelöst werden kann.

Beispielhaft aus dem schon besprochenen `flActivator` ist die Methode `fireActivateCommand` in Listing 5.1 aufgeführt. Hier wird der Aktivator ausgelöst und somit das Command – hier `_activateCmdObject` – ausgeführt.

Die Vererbungshierarchie der Interaktionsformen findet sich parallel wieder in der Vererbungshierarchie der *faceless widgets*. Anhand der Beispiele aus Abbildung 4.2 ist in Abbildung 5.1 die Vererbungshierarchie

```
/**
 * Executes the activate command
 *
 * @require isEnabled()
 */
public void fireActivateCommand ()
{
  Contract.require(isEnabled(), "isEnabled()");
  if (_activateCmdObject != null)
  {
    _activateCmdObject.execute(this);
  }
}
```

Listing 5.1: Die Methode `fireActivateCommand` in `flActivator`

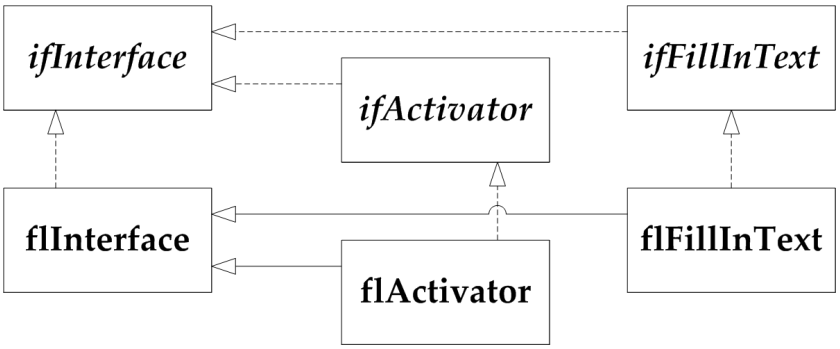

Abbildung 5.1: Vererbungshierarchie ausgewählter *faceless widgets*

der *faceless widgets* `flActivator` und `flFillInText` gezeigt. Beide *faceless widgets* implementieren die zugehörige Schnittstelle der Interaktionsformen `ifActivator` und `ifFillInText`. In ihrer gemeinsamen Oberklasse `flInterface` werden die Methoden aus der Schnittstelle `ifInterface` bereitgestellt. Das Interface `ifInterface` wird von jeder Interaktionsform implementiert. Daher erbt auch jedes *faceless widget* von der Klasse `flInterface`.

Im Rahmen dieser Arbeit wurden sämtliche Interaktionsformen die mit JWAM bereitgestellt werden, als *faceless widgets* implementiert.

5.2 Testkontext

Die *faceless widgets* alleine ermöglichen noch keine Akzeptanztests für die Interaktionskomponente einer JWAM-Anwendung. Bei der Konstruktion der Interaktionskomponente eines Werkzeuges muss für Akzeptanztests

ein spezieller Interaktionskontext, der *Testkontext*, mitgeliefert werden. Der Testkontext liefert Interaktionsformen als *faceless widgets*.

Der Testkontext kann erst zur Laufzeit wissen, welcher Typ von Interaktionsformen benötigt wird. Das Werkzeug bzw. der interaktive Teil benutzt die Methode `interactionForm` aus dem Interface `IAFContext`, siehe Listing 4.3 auf Seite 32 um auf Interaktionsformen zuzugreifen.

Das Werkzeug übergibt dabei den gewünschten Typ und den Namen der Interaktionsform. Der Testkontext kann erst in diesem Moment herausfinden, ob für die gewünschte Interaktionsform ein entsprechendes *faceless widget* vorhanden ist.

Der Testkontext, `IAFTestContext`, benutzt die Methode `makeTestClassName` um aus dem Namen der Interaktionsform den Namen der Klasse vom *faceless widget* zu erstellen. Gibt es parallel zu dem Package der Interaktionsform ein Package mit dem Namen `flinteraction`, so schaut der Testkontext hier nach dem *faceless widget*. Bei der Bennung der Klasse wird aus dem Namen der Interaktionsform, z.B. `ifActivator` das Präfix `if` mit dem Präfix `fl` – für *faceless widget*– ersetzt. Beispielsweise wird bei der Suche nach dem Interaktionstypen `de.abcd.interaction.ifBeispiel` die Klasse `de.abcd.flinteraction.flBeispiel` zur Laufzeit gesucht und instanziert.

Die Methode `getTestIAF` speichert schon alle im Laufe des Tests gefundenen *faceless widget* Klassen in einer Tabelle ab, um eine erneute Suche nach dem gleichen *faceless widget* zu beschleunigen. Die Methode ist in Listing 5.2 abgebildet. Sie wird intern von der Methode `interactionForm` aus der Schnittstelle `IAFContext` genutzt und liefert ein neues Exemplar des *faceless widgets*.

Es ist mit diesem Testkontext nun möglich, ein Werkzeug mit JWAM zu bauen, dem vorgetäuscht wird, es hätte eine Oberfläche. Anstatt der Interaktionskomponente Präsentationsformen zu liefern, die mit GUI-Elementen implementiert sind, werden von dem Testkontext *faceless widgets* zurück gegeben. Mit denen kann der Benutzer des Testkontextes Akzeptanztests schreiben. In der Testklasse für den Akzeptanztest muss die Funktionskomponente und ein Testkontext erzeugt werden. Diese beiden initialisieren im Konstruktor die Interaktionskomponente. Von dem Testkontext kann man sich nun alle für die Tests benötigten *faceless widgets* liefern lassen, um an diesen Benutzeraktionen zu simulieren.

Dies ist in den Listings 5.3 und 5.4 aus einer Testklasse für die Interaktionskomponente eines Kassierwerkzeuges gezeigt. In der initialisierenden Methode `setUp` aus Listing 5.3 wird der Interaktionskontext und die Interaktionskomponente erzeugt. Der Ablauf der Tests aus Listing 5.4 ist in dem Sequenzdiagramm Abbildung 5.3 verdeutlicht. Die Oberfläche des Kassierwerkzeuges ist in Abbildung 5.2 dargestellt. Es wird simuliert, dass der Benutzer in das Betragsfeld der Anwendung die Zahl 42 einträgt und auf

```
protected InteractionForm getTestIAF (Class ifType,
                                      String name)
{
  Contract.require(ifType != null, this,
                   "ifType not null");
  Contract.require(existsTestClass(ifType),
                   "exitsts test class for " + ifType);
  String testClassName = makeTestClassName(ifType.getName());
  InteractionForm result = null;
  try
  {
    Class cl = null;
    if (_testClasses.containsKey(testClassName))
    {
      cl = (Class) _testClasses.get(testClassName);
    }
    else
    {
      cl = Class.forName(testClassName);
      _testClasses.put(testClassName, cl);
    }
    result = (InteractionForm) cl.newInstance();
  }
  catch (InstantiationException e)
  {
    Contract.check(false, e);
  }
  catch (ClassNotFoundException e)
  {
    Contract.check(false, e);
  }
  catch (IllegalAccessException e)
  {
    Contract.check(false, e);
  }
  Contract.ensure(result != null, this,
                  "result not null");
  return result;
}
```

Listing 5.2: Die Methode `getTestIAF` in `IAFTestContext`

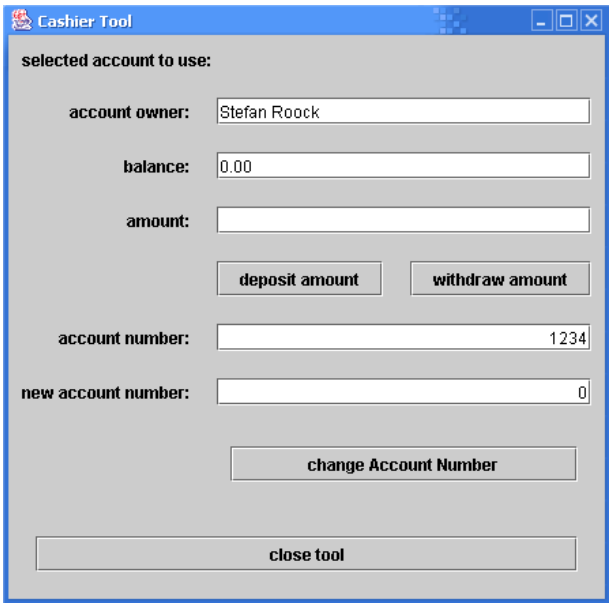

Abbildung 5.2: Oberfläche des Kassierwerkzeuges

```
// create the context
IAFContext tstContext =
       new IAFTestContext(new IAFContextImpl(
            new guiAccountToolSwing.class()));

// create the ip
ipAccountTool ip = new ipAccountTool(tstContext,
                   new fpAccountTool(Environment.instance()));
```

Listing 5.3: Die Methode `setUp` aus dem Testbeispiel

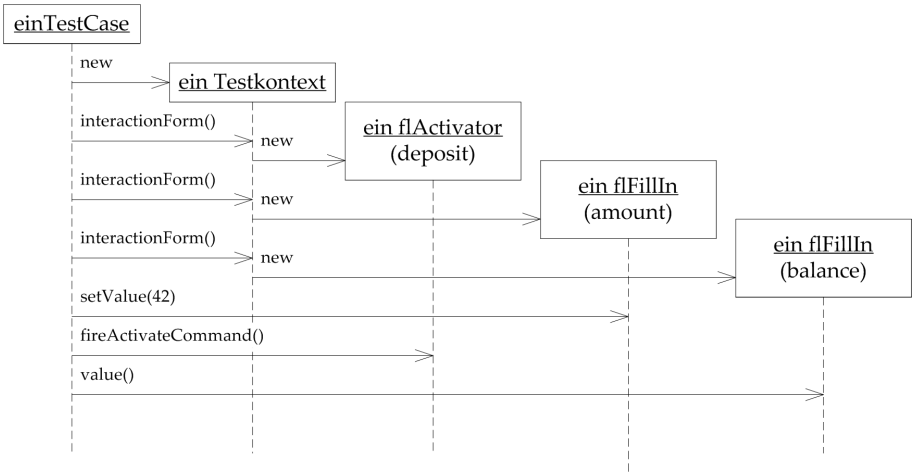

Abbildung 5.3: Diagramm für das Testbeispiel

den Button mit der Beschriftung „deposit“ (einzahlen) klickt. Der angezeigte Betrag aus dem Feld „amount“ sollte sich jetzt um 42 erhöhen. Anschliessend wird analog der Button „withdraw“ (auszahlen) getestet. Hierfür erzeugt der Test mit der GUI einen Testkontext. Mit diesem Testkontext als Interaktionskontext und der Funktionskomponente wird die Interaktionskomponente erzeugt. Von dem Testkontext werden die *faceless widgets* für die Benutzersimulation geholt. Mit diesen wird dann das beschriebene Szenario durchgespielt. Der Test erwartet, das das Werkzeug die richtigen Werte an den *faceless widgets* darstellt.

Wie in dem oben gezeigten Beispiel für das Kassierwerkzeug lassen sich mit diesem Testkontext alle Interaktionen eines JWAM-Werkzeuges testen, die mit der Trennung von Funktion und Interaktion, und Trennung von Interaktionsform und Präsentationsform arbeiten.

5.3 Testwerkzeug

Es hat sich in der praktischen Phase recht schnell herausgestellt, dass sich das JUnit-Framework nur schwer die geforderten Eigenschaften erweitern lässt. Des weiteren war es eine Anforderung an diese Arbeit, das Testwerkzeug nach den Konstruktionsprinzipien von WAM zu bauen, in JWAM zu integrieren und JWAM unabhängig von JUnit zu machen.

Es wurde eine eigene Oberklasse `TestCase` für Testfälle geschrieben, die von den Signaturen der Methoden kompatibel zu der Klasse `TestCase` aus JUnit ist. Es sind alle Methoden des Typs `assert` vorhanden, wie schon in Kapitel 3.3 auf Seite 22 beschrieben. Die Methoden `assert...` prüfen vom Prinzip her alle eine Bedingung. Wenn diese Bedingung nicht

```
// get the deposit activator
assert("deposit is registered as ifActivator",
   tstContext.hasInteractionForm(ifActivator.class,
                                 "deposit"));
ifInterface deposit = tstContext.interactionForm(ifActivator.class,
                                                 "deposit");
flActivator tstDeposit = (flActivator) deposit;
// get the withdraw activator
assert("withdraw is registered as ifActivator",
   tstContext.hasInteractionForm(ifActivator.class,
                                 "withdraw"));
flActivator tstWithdraw = (flActivator)
          tstContext.interactionForm(ifActivator.class,
                                     "withdraw");
// get the amount fill in
assert("amount ist als ifFillInFloatNumber registriert",
   tstContext.hasInteractionForm(ifFillInFloatNumber.class,
                                 "amount"));
flFillInFloatNumber tstAmount = (flFillInFloatNumber)
          tstContext.interactionForm(ifFillInFloatNumber.class,
                                     "amount");
// get balance fill in
assert("balance ist als ifFillInFloatNumber registriert",
   tstContext.hasInteractionForm(ifFillInFloatNumber.class,
                                  "balance"));
flFillInFloatNumber tstBalance = (flFillInFloatNumber)
          tstContext.interactionForm(ifFillInFloatNumber.class, "balance");
// deposit test
double oldBalance = tstAmount.value();
tstAmount.setValue(42.0);
tstDeposit.fireActivateCommand();
assertEquals(oldBalance+42.0, tstBalance.value(), 0.0);
// withdraw test
tstAmount.setValue(42.0);
tstWithdraw.fireActivateCommand();
assertEquals(oldBalance,  tstBalance.value(), 0.0);
```

Listing 5.4: Testbeispiel für das Kassierwerkzeug

```
/**
 * Asserts that the given condition is true.
 */
protected void assert(String message, boolean cond)
{
  if (!cond)
  {
    throw new AssertionFailedError(message);
  }
}
```

Listing 5.5: Die Methode `assert`

zutrifft, wird ein Exception des Typs `AssertionFailedError` geworfen. Bei diesen Exceptions des Typs `Error` wird von einer schweren Ausnahmeverletzung ausgegangen und nicht verlangt Methodenaufrufe in try-catch-Klauseln zu setzen. Das folgende Beispiel zeigt die Methode `assert`, die in ihrer Signatur keine Exception deklariert.

Einem `TestCase` kann zusätzlich über die Methode

```
public void setAcceptance (boolean value) { ...
```

gesagt werden, ob dieser Testfall zu einem Akzeptanztest gehört. Analog dazu kann ein `TestCase` mit der Methode

```
public boolean isAcceptance () { ...
```

geprüft werden, ob er ein Akzeptanztest ist. Damit lassen sich dann Komponententests und Akzeptanztests unterscheiden.

In JUnit werden Tests von dem Ergebnisobjekt `TestResult` ausgeführt. Die Tests in JUnit implementieren das Interface `Test`. Dazu gehört auch die Sammlung von Tests, die `TestSuite`. Das Ergebnisobjekt wird von dem `TestRunner` erzeugt. Es besteht bei der `TestSuite` keine Möglichkeit zwischen unterschiedlichen Testtypen zu unterscheiden.

Das Konzept aus der Klasse `TestSuite`, einer Sammlung von Tests, wurde nicht übernommen. Das Hinzufügen von Tests oder Testsammlungen sollte einfacher zu implementieren und eine Unterscheidung der Testfälle möglich sein. In dieser Arbeit können einem Testfall, bzw. der Klasse `TestCase` weitere Tests übergeben werden. Der Testfall kann seine Tests ausführen, dabei werden auch die an ihn übergebenen Tests ausgeführt. Der Testfall geht dabei schrittweise vor, wie ein Iterator. Mit den zwei Methoden `void runNext()` und `boolean hasNext()` kann ein Testfall schrittweise ausgeführt werden. Es muss beachtet werden, das `hasNext` eine notwendige Vorbedingung ist für `runNext`. Es gibt auch die Möglichkeit alle Tests eines Testfalls auf einmal auszuführen mit der Methode `void runAll()`.

Das Ergebnis des Tests kann beim Testfall abgefragt werden. Der Testfall liefert eine Liste der aufgetretenen Fehler. Diese sind unterteilt in zwei Kategorien. Zum einen gibt es Fehler die durch die Methoden `assert` und `fail` erzeugt werden können, wenn im Test eine falsche Annahme gemacht wurde. Zum anderen gibt es Fehler, wenn das getestete Programm eine unerwartete Exception auslöst, z.B. eine `NullPointerException`.

Die Funktionalität der Klasse `TestResult` aus JUnit wurde aufgeteilt. Das `TestResult` ist dort verantwortlich die einzelnen Testfälle auszuführen und gleichzeitig das Ergebnis des Testlaufs zu sammeln. Das Ergebnis des Testlaufs ist in der Lösung dieser Arbeit dem Testfall bekannt. Für die Ausführung des Tests wurde ein eigenes Werkzeug geschrieben.

Eine weitere Hilfsklasse in dem erstellten Testframework ist der `TestCollector`. Dem `TestCollector` wird der Name eines Verzeichnisses übergeben. Innerhalb dieses Verzeichnisses und aller Unterverzeichnisse werden alle Class-Dateien gesucht, die einen Testfall repräsentieren, instanziiert und in zwei Testfällen gesammelt. Eine `TestCase` wird mit Akzeptanztests befüllt, der andere mit Komponententests. Die Testfälle werden mit der Methode `isAcceptance` unterschieden.

Damit sind alle Vorrausetzungen geschaffen unabhängig von JUnit Testfälle zu schreiben. Für das Testwerkzeug wurde streng nach den WAM-Entwurfsmustern vorgegangen. Es wurde zuerst ein einfaches Testwerkzeug konstruiert, aufgeteilt in ein Werkzeugobjekt, eine interaktive und eine funktionale Komponente. Das einfache Testwerkzeug kann nur einen Testfall ausführen. Es zeigt den Erfolg des Testlaufs und gegebenenfalls die aufgetretenen Fehler. Dabei werden auch die Anzahl der in diesem Testfall vereinigten Testfälle angezeigt. Dieses einfache Testwerkzeug wird als Subwerkzeug zweimal in dem neuen Testwerkzeug, dem *TestingTool* benutzt. Zuerst werden alle Testklassen mit dem `TestCollector` eingesammelt. Die Akzeptanztests werden in dem einen, die Komponententests in dem anderen Testwerkzeug ausgeführt. Ein Bild der Oberfläches dieses Werkzeuges ist in Abbildung 5.4 zu sehen.

Bei der Erstellung des Testwerkzeuges sind mehrere Probleme aufgetreten. Die Konstruktion von komplexen Werkzeugen die aus mehreren Subwerkzeugen zusammengesetzt sind, wird durch JWAM unterstützt. Doch lassen sich nicht so ohne weiteres mehrere Subwerkzeuge gleichen Typs in ein Werkzeug einbinden. Daher mussten von dem einfachen Testwerkzeug und seinen drei Bestandteilen – Werkzeugobjekt, interaktive und funktionale Komponente – jeweils zwei Subklassen konstruiert werden, ohne neue Funktionalität. Damit konnten zwei der einfachen Testwerkzeuge unterschiedlichen Typs als Subwerkzeug in das neue Testwerkzeug, dem *TestingTool*, eingebunden werden. Dieses Problem scheint inwischen durch die Arbeit von [Bee01] gelöst worden zu sein.

Das Ausführen von mehreren Tests kann je nach Größe der Tests sehr lange dauern, meist mehrere Minuten. Daher wird der Fortschritt des Test-

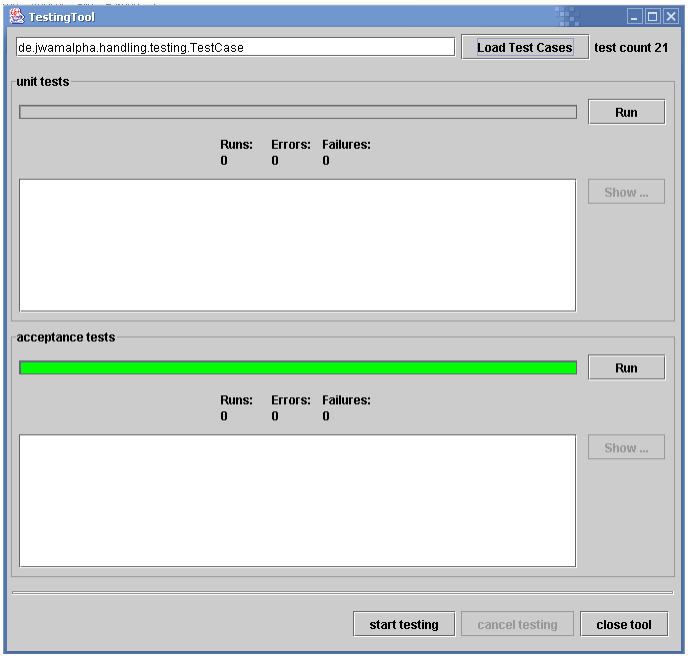

Abbildung 5.4: Das *TestingTool*

laufes durch einen Fortschrittsbalken dargestellt. Die Oberfläche für das Testwerkzeug ist mit Swing, der Standardbibliothek für GUIs aus dem JDK[1] erstellt worden. Es ergab sich das Problem, das die Oberfläche, und somit auch der Fortschrittsbalken erst nach Beendigung des Testlaufes aktualisiert wurde. Dadurch konnte der Fortschritt nicht angezeigt werden. Das Problem wurde gelöst, indem die funktionale Komponente des einfachen Testwerkzeuges einen `BackgroundTester` für die Ausführung der Tests benutzt. Dieser startet die Tests asynchron in einem neuen Thread. Mit dieser Lösung bleibt die Oberfläche des Werkzeuges reaktiv und die Anzeige des Fortschrittes über den Testverlauf aktuell.

5.4 Zusammenfassung

Das Kapitel hat den praktischen Teil dieser Arbeit beschrieben. Es wurde gezeigt wie die in dieser Studienarbeit vorgestellten Konzepte über das Testen interaktiver Anwendungen, die mit JWAM konstruiert sind, realisert wurden. Dabei sind die wichtigsten Details und Probleme der Implementation der *faceless widgets*, dem Testkontext und dem neuen Testwerkzeug, dem *TestingTool* beschrieben worden.

[1] Java Development Kit, siehe `http://java.sun.com/`

Kapitel 6

Abschluss

In diesem Kapitel wird das Ergebnis der Stueinarbeit zusammengefasst und kritisch betrachtet. Es wird ein Ausblick auf weitergehende Fragestellungen im Zusammenhang dieser Arbeit gegeben.

6.1 Zusammenfassung

In dieser Arbeit wurde das Testen von Software vorgestellt. Hierbei sind die wichtigsten Begriffe aus dem Kontext Testen erläutert worden. Es wurde beschrieben welche Rolle Testen bei der Entwicklung spielt, insbesondere beim Entwickungsprozess eXtreme Programming.

Für das Framework JWAM wurde eine Untertützung für Akzeptanztests interaktiver Anwendungen geschrieben. Bisher konnten nur Komponententests automatisch mit einer Erweiterung für JUnit ausgeführt werden. Akzeptanztests mussten händisch durchgeführt werden.

Anhand von Handhabungsvisionen eines Werkzeuges lassen sich Akzeptanztests definieren. Das WAM-Muster der Trennung von Interaktion und Präsentation bot die Möglichkeit Akzeptanztests zu automatisieren. Die speziellen Interaktionsformen *faceless widgets* wurden eingeführt, um Benutzereingaben zu simulieren. Es wurde ein neuer Interaktionskontext konstruiert, der Testkontext, um diese *faceless widgets* zu nutzen. Ein neues Testwerkzeug, das *TestingTool* wurde konstruiert. Akzeptanztests besitzen einen anderen Stellenwert, z.B. müssen sie erst bei der Auslieferung an den Kunden vollständig durchlaufen. Das wird vom erstellten Testwerkzeug berücksichtigt und Komponententests werden getrennt von Akzeptanztests ausgeführt und dargestellt.

Diese in dieser Arbeit erstellten Erweiterungen wurden in das Framework JWAM integriert. Mit der Hilfe von Handhabungsvisionen können nun Akzeptanztests für JWAM-Werkzeuge geschrieben und automatisch ausgeführt werden. Diese Akzeptanztests sind beschränkt auf interaktive Werkzeuge, die in ihrer Interaktionskomponente Interaktionsformen für

den Zugriff auf die Werkzeugoberfläche benutzen. Werkzeuge die direkt GUI-Elemente der Oberfläche ohne Interaktionsformen benutzen, lassen sich nicht mit der Lösung dieser Arbeit testen.

6.2 Ausblick

Die Werkzeugkonstruktion in JWAM wurde um sogenannte *Actions* durch [Bee01] erweitert. Diese Actions werden im Werkzeugobjekt definiert und direkt von der Werkzeugoberfläche in Menüs und Toolbars dargestellt. Die Interaktionskomponente des Werkzeug wird dabei nicht berücksichtigt. Die Lösung dieser Arbeit bietet somit keine Möglichkeit Benutzeraktionen an den *Actions* zu simulieren. Es muss noch eine Lösung gefunden werden, um *Actions* in Akzeptanztests zu berücksichtigen.

Eine weitere Frage ist, ob mit einer Erweiterung der Präsentationsformen der Ablauf von Interaktionen an einem Werkzeug protokolliert werden kann. Das Protokoll wäre eine Folge von Interaktionen. Das Protokoll müsste mit Tests erweitert zu einem Testfall werden. Dieser Testfall sollte sich dann automatisch immer wieder ausführen lassen können. Diese Lösung wäre zu vergleichen mit den vorhandenen GUI-Playback-Capture-Tool, siehe [**?**].

Das Testframework JUnit hat sich als der defacto Standard für automatische Tests in Java herausgestellt. JUnit wird in unzähligen Projekten erfolgreich benutzt. Viele Entwicklungsumgebungen haben eine direkte JUnit-Unterstützung. Die Akzeptanz eines zu JUnit inkompatiblen Testframeworks, wie das dieser Arbeit, ist gering. Aufgrund der breiten Akzeptanz von JUnit, sollte über eine Integration nachgedacht werden. Es sollte nach einer Lösung von Akzeptanztests mit dem Standard JUnit gesucht werden, die die verschieden Typen von Tests unterscheiden und getrennt voneinander ausführen kann. Damit könnte das *TestingTool* durch JUnit ersetzt werden.

Jemmy[1] ist eine Java-Bibliothek, mit der sich Benutzeraktionen bei Swing-Anwendungen simulieren lassen. Im Gegensatz zu dieser Arbeit, wird Swing direkt angesprochen und es werden keine Interaktionsformen benutzt. Mit Jemmy könnten auch Akzeptanztests unterstützt werden.

Es wäre auch eine interessante Frage, mit welchen Mitteln sich eine zufriedenstellende Testabdeckung bei der Erstellung von Softwaresystemen definieren und ermitteln lässt. Zudem könnten automatisch ausgeführte Akzeptanztests und händisch durchgeführte Akzeptanztests verglichen werden. Dabei sollte beachtet werden, welche Lösung schneller realisiert werden und welche Lösung schneller auf sich ändernde Anforderung reagieren kann.

[1] siehe [Jem03]

In dieser Arbeit wurden Akzeptanztests nicht betrachtet, die nichtfunktionale Anforderungen prüfen. Es könnte erforscht werden, mit welchen Mitteln die Performanz einer Anwendung getestet werden kann und inwieweit diese Tests automatisch ablaufen können.

Literaturverzeichnis

[Bec99] BECK, Kent: *Extreme Programming Explained: Embrace Change*. Reading, MA : Addison-Wesley, 1999

[Bee01] BEEGER, Robert F.: *Einbettung von Oberflächen bei der Werkzeugkomposition*, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Studienarbeit, 2001

[BG99] BECK, Kent ; GAMMA, Erich: JUnit: A cook's tour. In: *Java Report* 4 (1999), Nr. 5, S. 27–38

[BGL+99] BLEEK, Wolf-Gideon ; GRYCZAN, Guido ; LILIENTHAL, Carola ; LIPPERT, Martin ; ROOCK, Stefan ; WOLF, Henning ; ZÜLLIGHOVEN, Heinz: Frameworkbasierte Anwendungsentwicklung (Teil 2). In: *OBJEKTspektrum* 2 (1999), S. 78–83

[Bin99] BINDER, Robert V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA : Addison-Wesley, 1999

[Dij72] DIJKSTRA, Edsger W.: Notes on Structured Programming. In: DAHL, O. J. (Hrsg.) ; DIJKSTRA, E. W. (Hrsg.) ; HOARE, C. A. R. (Hrsg.): *Structured Programming*. London : Academic Press, 1972

[FBB+99] FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS, Don: *Refactoring: Improving the Design of Existing Code*. Reading, MA : Addison-Wesley, 1999

[Fow97] FOWLER, Martin: *UML distilled, Applying the standard object modelling language*. Reading, MA : Addison-Wesley, 1997

[FZ99] FLOYD, Christiane ; ZÜLLIGHOVEN, Heinz: Softwaretechnik. In: RECHENBERG, Peter (Hrsg.) ; POMBERGER, Gustav (Hrsg.): *Informatik-Handbuch*. 2. aktualisierte und erweiterte Auflage. München : Hanser Verlag, 1999, S. 763–790

[GB98] GAMMA, Erich ; BECK, Kent: Test Infected: Programmers Love Writing Tests. In: *Java Report* 3 (1998), Nr. 7, S. 37–50

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA : Addison Wesley, 1995

[GLL+99] GRYCZAN, Guido ; LILIENTHAL, Carola ; LIPPERT, Martin ; ROOCK, Stefan ; WOLF, Henning ; ZÜLLIGHOVEN, Heinz: Frameworkbasierte Anwendungsentwicklung (Teil 1). In: *OBJEKTspektrum* 1 (1999), S. 90–99

[Har87] HAREL, David: State Charts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987)

[HU79] HOPCROFT, John E. ; ULLMAN, Jeffrey. D.: *Introduction to Automata Theory, Languages and Computation*. London : Addison-Wesley, 1979

[Int03] *CVS IntegrationGuard*. 2003. – `http://iguard.sf.net/`

[Jem03] *Jemmy*. 2003. – `http://jemmy.netbeans.org/`

[JWA02] *JWAM Framework*. 2002. – `http://www.jwam.de/`

[Krö00] KRÖGER, Malte: *Erstellen und durchführen von Anwendungstests mit GUI Capture/Playback Testtools*, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Studienarbeit, 2000

[Leh80] LEHMAN, Meir M.: Programs, Life Cycles and Laws of Software Evolution. In: *Proceedings of the IEEE* 68 (1980), Nr. 9, S. 1060–1076

[LRW02] LIPPERT, Martin ; ROOCK, Stefan ; WOLF, Henning: *Software entwickeln mit eXtreme Programming: Erfahrungen aus der Praxis*. Heidelberg : dpunkt Verlag, 2002

[Mey97] MEYER, Bertrand: *Object-Oriented Software Construction*. 2nd edition. Englewood Cliffs (NJ), USA : Prentice-Hall, 1997

[MFC00] MACKINNON, Tim ; FREEMAN, Steve ; CRAIG, Philip: EndoTesting: Unit Testing with Mock Objects. In: *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, 2000

[Mye78] MYERS, G. J.: *The Art of Software Testing*. John Wiley & Sons, 1978

[Zül98] ZÜLLIGHOVEN, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. Heidelberg : dpunkt Verlag, 1998