FACHHOCHSCHULE TRIER
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences

Software-Qualitätssicherung

Testing can show the presence but not the absense of defects

Edsger Wybe Dijkstra (1930-2002)

Hans-Jürgen Stemmer

Herbst 2003

Inhalt

Einleitung 3
Motivation 4
Abgrenzung 6
Der Test im Software-Entwicklungsprozeß 6
Klassifikation von Prüftechniken 7
Inhalt der Arbeit 7
Funktionsorientiertes Testen 8
Bildung von Äquivalenzklassen 9
- Beispiel: Tarifrechner einer Krankenversicherung 10
Zustandsbasiertes Testen 11
- Beispiel: Kartenautomat in einem Parkhaus 12
Sequenzdiagramme 13
Beispiel: Kunde login 13
Testen graphischer Benutzeroberflächen 14
Beschränkungen 15
Kontrollflussorientiertes, strukturorientiertes Testen 15
Aufzählung und Gewichtung der Verfahren 16
Anweisungsüberdeckungstest (C_0) 17
Zweigüberdeckung (C_1) 17
Einfacher Bedingungsüberdeckungstest (C_2) 18
Bedingungs-/Entscheidungsüberdeckungstest 18
Mehrfach-Bedingungsüberdeckungstest (C_3) 19
Minimaler Mehrfach-Bedingungsüberdeckungstest (Mischung C_2 und C_3) 19
Modifizierter Bedingungs-/Entscheidungsüberdeckungstest 20
Pfadüberdeckungstest (C_4) 21
Boundary-interior-Test 22
Strukturierter Pfadtest 22
Werkzeuge für das Testen von Software 23
JUnit 23
JProbe 24
Purify und DevPartner 25
SQS 26
Abschließende Bewertung 26

Einleitung

„... Also mal ehrlich: Was tun sie [ihre Mitarbeiter] den ganzen Tag?"

„Ich weiß nicht."

„ Stellen Sie sich vor, Sie würden jeden Tag um, sagen wir drei Uhr nachmittags, festhalten, was alle Mitarbeiter eines Projektes in diesem Augenblick tun. Dann ordnen Sie die Einzeldaten, die Sie für alle Ihre Mitarbeiter über die gesamte Laufzeit des Projektes gesammelt haben, in Kategorien. Was tun die meisten Leute die meiste Zeit?"

„Debuggen, vermute ich. Ein Großteil der Arbeit scheint in diese Kategorie zu fallen."

„Dann liegt hier unsere Chance. Wir müssen zusehen, dass wir auf einen Teil der Debugging-Zeit verzichten können."

„Wir müssen lernen, effizienter zu debuggen?"

„Nein", korrigierte Kenoros ihn. „Wir müssen lernen, effizienter zu entwerfen."

[Es schließt sich die Erklärung einer Last-Minute-Implementierungstechnik an, die z.B. bei einem auf ein Jahr angelegtem Projekt erst in den letzten beiden Monaten mit der Implementierung beginnt, dafür aber einen überaus detaillierten Entwurf vorsieht.]

„...Praktisch niemand erstellt einen Entwurf, der dem eigentlichen Code so nahe kommt, dass eine vernünftige Überprüfung möglich ist."

„Aber natürlich erstellen wir einen Entwurf. Jeder tut das."

„Natürlich. Aber nicht in der Entwurfsphase. In der Entwurfsphase stellt das Team ein Dokument zusammen. Es besteht aus ein paar Platitüden über die Programmphilosophie, ein oder zwei Datendefinitionen und einer Pro-forma-Überprüfung. Sie tun alles, was notwendig ist, um das Management loszuwerden und endlich mit dem Codieren beginnen zu können. Irgendwann sagt der Manager: ‚Okay, ihr könnt mit dem nächsten Teil anfangen.' Das Team triumphiert, und der sogenannte Entwurf verschwindet im Schrank und wird nie mehr wieder konsultiert. Der Entwurf ist Schrank-Ware.
Beim Codieren erstellen sie dann den eigentlichen Entwurf. Beim Codieren! Dann entscheiden sie, wie die Module und die Schnittstellen wirklich aussehen sollen. Dann fallen die Entscheidungen, die sich der Überprüfung entziehen."

Mr Tompkins legte seinen Stift aus der Hand. Das Debuggen konnte bis zu fünfzig Prozent der gesamten Projektressourcen aufzehren. Aber wenn er sich entschied die neue Technik anzuwenden, mußte er mit Meuterei in der Belegschaft rechnen. Programmierer sind süchtig nach Fehlersuche. Sie würden den radikalen Plan nicht eben begeistert aufnehmen.

Tom deMarco; Der Termin; Hanser 1998; S.153ff

Tom deMarco ist Autor und Berater mit Sinn für Humor und einem klaren Blick für die Realität. Gleichzeitig sind seine Bücher aber auch visionär. Es ist schwer, deMarco zu lesen, ohne sich (sehnsüchtig) vorzustellen, wie schön es wäre, wenn...
Seine Bücher richten sich an Entscheider. Für einfache Mitarbeiter sind sie mitunter gefährliche Kost.

Motivation

Kent Beck und Erich Gamma schreiben in der Einleitung zum Testframework JUnit:

„Jeder Programmierer weiß, dass er für seinen Code Tests schreiben sollte. Nur wenige tun es. Nach dem Grund gefragt, fehlt den meisten die Zeit dazu. Ein Kreislauf beginnt: Je weniger Zeit, desto weniger Tests werden geschrieben. Je weniger Tests geschrieben werden, desto schlechter wird der Code. Die Produktivität sinkt. Mit sinkender Produktivität erhöht sich aber der Druck auf den Entwickler.“[1]

Beck und Gamma favorisieren einen Ansatz, bei dem vor dem Codieren (oder zumindest parallel dazu) Tests geschrieben werden. Voraussetzung für die Akzeptanz einer solchen Methode ist ihrer Meinung nach, dass das Testen Spaß macht. Es muß einfach sein und darf vom Entwickler keinesfalls als zusätzliche Belastung wahrgenommen werden.

Mit JUnit (inzwischen ein Standardframework, dass in die meisten professionellen Werkzeuge integriert ist), stellen sie ein objektorientiertes Framework vor, dass sehr einfach zu bedienen ist und in dem Tests in der Software-Entwicklungsumgebung programmiert werden. Das heißt, vom Entwickler ist weder eine neue Sprache noch ein neues Werkzeug zu erlernen.

Tests werden stets so formuliert, dass das Ergebnis eindeutig richtig oder falsch ist. Alles was gewöhnlich über Tracing ausgegeben wird (um dann vom Entwickler kontrolliert zu werden) und alles, was im Debugger überprüft wird (Zustände, Bedingungen etc.) sollen als Teststatements formuliert werden. Einzelne Tests können in Gruppen zusammengefaßt werden.

Testklassen sind wiederverwendbar und insbesondere dann anwendbar, wenn sich Teile der Anwendung verändert haben. Der Programmierer hat jederzeit die Möglichkeit den „Gesundheitszustand“ seiner Anwendung zu prüfen, kann auf fehlgeschlagene Tests reagieren oder zufrieden ein positives Testergebnis zur Kenntnis nehmen. Tests werden nicht erst nach der Codierung durchgeführt, sondern permanent.[2] JUnit wird im Kapitel Werkzeuge noch weiter besprochen.

Nach dem im 19. Jahrhundert lebenden Professor für politische Ökonomie an der Universität von Lausanne Vilfredo Pareto benannt ist das 80/20-Pareto-Prinzip. Pareto untersuchte verschiedenen Zusammenhänge und stellte zum Beispiel fest, das 20% der Italiener 80% des Vermögens besaßen.

1 http://junit.sourceforge.net/doc/testinfected/testing.htm
2 http://www.xprogramming.com/xpmag/whatisxp.htm

Dieses Verhältnis wurde seitdem auch in anderen Zusammenhängen gefunden. So haben Untersuchungen ergeben, das 80% der Aufgaben in 20% der Zeit erledigt werden. Übertragen in die Softwaretechnik kann man etwa feststellen, dass 20% der Fehlerursachen 80% der Fehlerwirkungen erzeugen[3].

Die höchsten Kosten werden Erhebungen zufolge von Fehlern verursacht, die sich durch ungenaue oder fehlerhafte Anforderungen (Kunden- und Sofware-anforderungen) ergeben. Logisch erscheint in diesem Zusammenhang, dass Fehler, je später sie gefunden werden, immer höhere Kosten verursachen[4]. Beides stützt in hohem Maß auch den Gedanken der Einleitung, nach dem dem Entwurf die eigentliche Bedeutung im Software-Entwicklungsprozeß zukommt.

So geht schon aus den ersten Seiten deutlich hervor, dass Qualitätssicherung auch zu einer philosophischen oder grundsätzlichen Frage werden kann. Ist es richtig zehn Monate zu entwerfen, um dann zwei Monate zu codieren? Wie intensiv muß man testen? So wie es auf viele Fragen der Philosophie keine endgültige Antwort gibt, ist auch die Frage nach dem richtigen Test kaum zu beantworten. Da sich alle Projekte zumindest in einigen Parametern unterscheiden, hat sicher jeder Ansatz seine Berechtigung.

Ohne Tests wird (und will) man aber schwerlich auskommen. Sehr viele Firmen definieren Prozesse und Forderungen für die Qualitätssicherung. Der Umfang dieses Prozesses wird immer mit der „Bedeutung“ der Software korrelieren. Liggesmeyer[5] gibt in der Einleitung seines Buches zahlreiche Beispiele dafür.

Auch die vor einigen Jahren aufgekommene Zertifizierung entstammt wohl vor allem dem Wunsch des Kunden nach mehr Transparenz – eben auch im Bereich der Qualitätssicherung. Der Kunde möchte wissen, welche Anstrengungen ein Unternehmen unternimmt, um die Qualität sicherzustellen.

Intensive Tests kosten allerdings Geld. Schon rein rechnerisch sind vollständige Tests nicht möglich[6], so dass in jedem Fall ein Kompromiß eingegangen werden muß. Sicherheitskritische Anwendungen in Banken, Luftfahrt, Medizin, etc. dürfen schon mal etwas teurer werden. Kein Anwender aber wäre bereit für ein Office-Paket die Mehrkosten hoher Testaufwände zu bezahlen.

Daneben wird auf der Mikroebene jeder Entwickler im Bemühen um Qualität seine Software auch ohne ausdrückliche Aufforderung testen. Oft wird es hier sogar so sein, dass zu viel getestet wird und wiederum ein Kompromiß – diesmal zwischen Termin und Qualität – erforderlich ist. Eine Reihe der von Liggesmeyer vorgestellten Techniken sind in der Entwicklung (u.U. unbewußt) gängige Praxis. (Jeder Programmierer weiß, dass die Null eine kritische Zahl ist und wird seine Algorithmen gegen Null testen.)

Statt aber das Testen (bzw. die Aussagefähigkeit des Tests) dem Zufall zu überlassen, macht es Sinn die vorhandenen Möglichkeiten zu klassifizieren und zu

[3] Liggesmeyer, S.15f
[4] Liggesmeyer, S.29
[5] Peter Liggesmeyer; Software-Qualität – Testen, Analysieren und Verifizieren von Software, Spektrum 2002
[6] vgl. Liggesmeyer S.132f

bewerten. Im Bedarfsfall ist so eine schnelle und geeignete Wahl der Testmethode(n) möglich. Techniken können kombiniert werden um Schwächen einer Technik durch Stärken einer anderen Technik auszugleichen.

Abgrenzung

Der Test im Software-Entwicklungsprozeß

Beispielhaft wird hier der Software-Entwicklungsprozeß nach Bernd Oesterreich (OEP) skizziert.

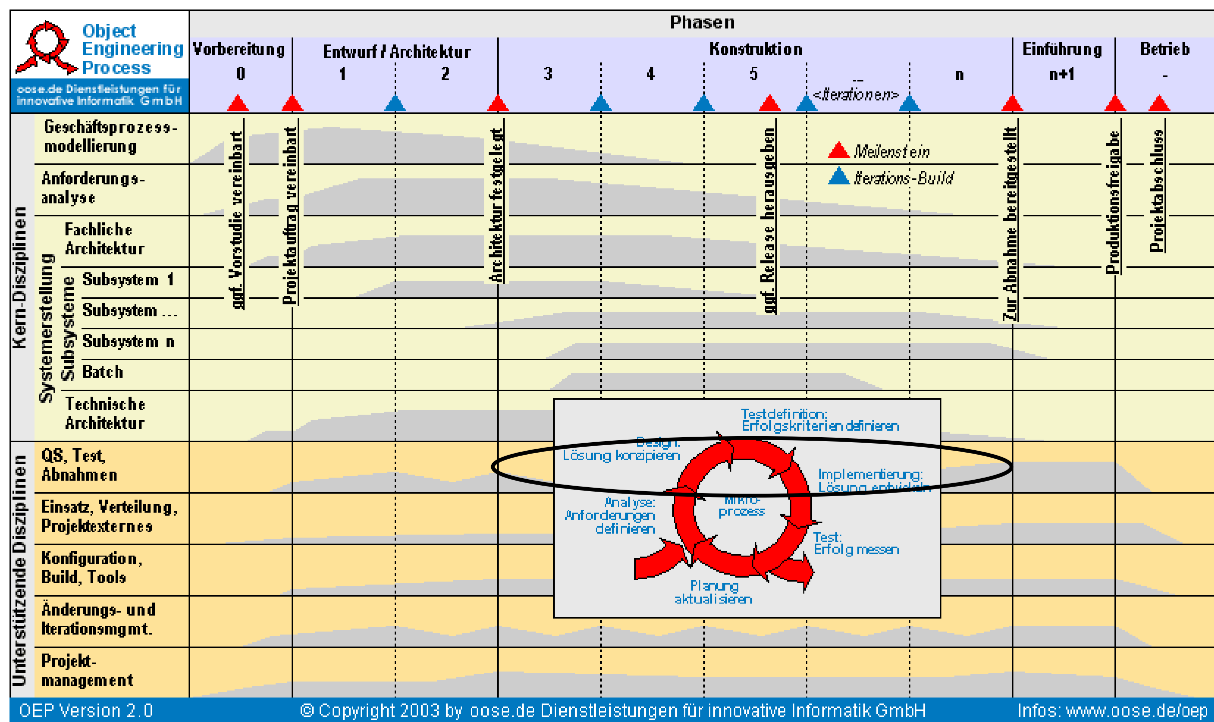

Abb. 1

Auf dem Markt werden inzwischen zahlreiche Prozesse angeboten. Viele sind Variationen bekannter Prozesse wie OEP, RUP (Rational Unified Process) oder des V-Modells, als Vorgehensmodell des Bundes.

Ein solcher Prozeß umfaßt alle Phasen der Entwicklung. Dazu gehören neben der Konzeption auch Konfigurations- und Projektmanagement sowie Einführung und Betrieb.

Interessant ist in diesem Zusammenhang vielleicht, das die Zertifizierung (ISO9001-9003) weniger die Qualität am Ende des Prozesses im Blick hat, als vielmehr den Prozeß selbst.

Grundlage und Ausgangspunkt des Vortrages (und der Lehrveranstaltung in derem Rahmen dieses Referat entstand) ist das Buch von Peter Liggesmeyer, dessen Inhalt das Testen, Analysieren und Verifizieren von Software ist. Im Bild oben befinden wir uns damit bei der unterstützenden Disziplin Test in den Phasen 3 bis n.

Klassifikation von Prüftechniken

Das von Liggesmeyer vorgestellte Klassifikationsschema[7] unterscheidet statische und dynamische Tests. Zu den statischen Tests gehören beispielsweise formale Methoden und auch Codereviews.
Die dynamischen Techniken unterteilt er in funktionsorientierte, strukturorientierte und diversifizierende Tests sowie Bereichstests. Die strukturorientierten Verfahren zerfallen noch einmal in kontrollflußorientierte und datenflussorientierte Verfahren.

Die in dieser Arbeit vorgestellten Verfahren zählen demnach zu den funktionsorientierten bzw. den strukturorientierten, kontrollflußorientierten Testtechniken.

Gängig ist daneben eine weitere Klassifikation. So kann Software mit und ohne Kenntnis der Struktur getestet werden. Während im ersten Fall Daten- und Kontrollflüsse getestet werden können (white box), ist man im zweiten auf die Spezifikation angewiesen (black box) um zu entscheiden, ob die Software tatsächlich das tut, was von ihr verlangt wird.

Allen diesen Verfahren gemeinsam ist, das sie immer eine Stichprobe mit konkreten Eingabewerten in einer realen Systemumgebung darstellen.

Inhalt der Arbeit

Im nächsten Kapitel behandelt diese Arbeit in einem kurzen Überblick funktionsorientierte Tests (black box) und geht auf die Bildung und Bedeutung von Äquivalenzklassen ein. Im Anschluß wird ein zustandsbasierter Test mit Zustandsübergangstabellen beschrieben, wie er vor allem für objektorientierte Systeme interessant ist. Am Ende wird transaktionsflussorientiertes Testen auf Basis von Sequenzdiagrammen besprochen. Sequenzdiagramme finden sich im Umfang von UML und sind relativ verbreitet.

In einem weiteren Kapitel werden kontrollflussorientierte, strukturorientierte Testtechniken (white box) vorgestellt. Es wird eine Übersicht über die gebräuchlichen Testverfahren gegeben. Dabei werden die Verfahren in eine Ordnung gebracht und bewertet. Auf die Probleme beim Testen von einfachen und zusammengesetzten Bedingungen wird dabei verstärkt eingegangen.

Im letzten Kapitel werden Werkzeuge (JUnit und JProbe) vorgestellt, die Teile dieser Techniken unterstützen. Beide Werkzeuge finden in der Java-Entwicklung Anwendung. Die Aufzählung ist nicht vollständig und wird den Funktionsumfang der Tools nur anreißen. Auffällig ist, dass die Theorie in keinem der Werkzeuge abgedeckt wird. Die sich aus der Theorie ergebenen Schwierigkeiten für die Praxis machen das Fehlen komplexer Automation vielleicht verständlich.

So wird man auch bei den Werkzeugen Stärken kombinieren müssen um einen Test zu erreichen, bei Testüberdeckung sowie Kosten und Nutzen in vernünftiger Relation stehen.

---

[7] Liggesmeyer S.34

Während die Verfahren selbst in unterschiedlichem Umfang relativ bekannt sind, sind Werkzeuge noch nicht sehr verbreitet.

Funktionsorientiertes Testen

Ohne Beachtung und/oder Kenntnis des inneren Aufbaus einer Klasse, eines Moduls oder einer Anwendung werden beim **black-box-Test** nur Eingabe und Ausgabe betrachtet.

Ausgangspunkt sind die funktionalen Anforderungen an das Programm (Pflichtenheft, Modulspezifikation). Eine solche Spezifikation wäre ohne Frage das Handbuch einer Anwendung. Hier ist (sogar für den Endanwender verständlich) genau beschrieben, was eine Software leistet (oder leisten sollte). In der Regel entsteht das Handbuch aber nicht vor (evtl. während) der Testphase.

Darüber hinaus wird ein Entwurf beschreiben, wie die einzelnen Module arbeiten. Jede Modulspezifikation muss mindestens die Schnittstelle (zu anderen Modulen oder dem Anwender) beschreiben.

Je detaillierter die Spezifikation bzw. die Anforderung, desto einfacher ist ein funktionsorientierter Test. Für jeden Testfall untersucht der Tester ein Modul oder eine Funktion der Anwendung. Für den Test hat er Testwerte ausgewählt und anhand der Spezifikation erfahren, welches Verhalten bzw. welchen Ausgaben zu erwarten sind. Stimmen die Ergebnisse nicht mit den erwarteten Ausgabedaten überein, liegt ein Fehler vor. Entspricht das Ergebnis den Erwartungen, wird der Test mit anderen Testdaten wiederholt. Am Ende eines erfolgreichen Tests steht die Freigabe des Moduls.

Teilweise ist der Modultest ohne weitere Komponenten möglich. In den meisten Fällen werden die Komponenten hierarchisch einander benötigen. Kennzeichen schlechter Architektur ist es, wenn das Produkt nur als Ganzes getestet werden kann.

Nach dem Test der Einzelmodule wird sich in jedem Fall ein Integrationstest anschließen. Je geringer die Abhängigkeiten desto einfacher können Teilkomponenten verändert oder auch ausgetauscht werden.

Ein Vorteil des funktionsorientierten Tests ist, dass die Testfälle auch von Nicht-Entwicklern spezifiziert und überprüft werden können. Mitunter ist gerade das sinnvoll. In vielen Fällen wird der Auftraggeber sein Produkt ohnehin selbst testen und über eine Freigabe entscheiden. Da er die fachlichen Anforderungen am besten kennt, ist er auch als funktionaler Tester am besten geeignet.

Zu beachten ist, dass häufig mehr als eine Systemumgebung betrachtet werden muss. War dies zu Zeiten von C und C++ selbstverständlich, entkommen auch jüngere Programmiersprachen wie Java dieser Problematik nicht. So wurde das Java-Markenzeichen „Write once, Run anywhere“ unter Entwicklern längst in „Write once, Test everywhere“ umgetauft.

Um insbesondere im Fehlerfall eine Aussage zur korrigierten Version treffen zu können, müssen die Testfälle vor dem Test kategorisiert und spezifiziert werden. Ein aussagekräftiger Test setzt Vorbereitung voraus. Insbesondere die Wahl der Testdaten ist von Bedeutung. Über den Test selbst muß genau Protokoll geführt werden.

Der black-box-Test betrachtet nicht den Aufbau eines Moduls oder einer Klasse. So kann der Test auch keine Aussage über die im white-box-Test untersuchte Pfadüberdeckung treffen. Ausnahmen, die vom Programmierer behandelt werden – und denen oft intensive Überlegungen vorausgehen – werden unter normalen Bedingungen gar nicht auftreten.

Bildung von Äquivalenzklassen

Aus dem Gesagten geht klar hervor, dass funktionsorientierte Tests nur so gut sein können, wie die Testdaten, mit denen sie testen. Die Menge möglicher Testdaten ist dabei fast immer unendlich groß. Durch die Bildung von Äquivalenzklassen versucht man die Menge möglicher Eingabewerte in Klassen zu unterteilen, für die sich das Programm (bzw. Modul) gleich verhalten sollte. Es wird angenommen, das das Programm bei der Verarbeitung eines Vertreters dieser Klasse genauso reagiert, wie bei allen anderen Werten der Klasse.

Die Äquivalenzklassen können noch einmal in gültige und ungültige Äquivalenzklassen unterteilt werden. Die Menge der gültigen Äquivalenzklassen enthält alle Mengen, die gültige Werte für das Modul enthalten. Äquivalenzklassen können sowohl für Eingaben als auch für Ausgaben gebildet werden. Die Regeln gelten entsprechend.

Die Auswahl von Vertretern der Eingabe-Äquivalenzklassen ist nicht festgelegt. Als sinnvoll hat sich insbesondere die Betrachtung der Grenzwerte erwiesen, jener Werte also, die, sofern sich die Äquivalenzklasse natürlich ordnen läßt, am unteren bzw. am oberen Rand der Klasse liegen (Grenzwertanalyse). Werte können aber auch zufällig aus den Äquivalenzklassen ausgewählt werden.

Darüber hinaus kommt der intuitiven Testfallermittlung Bedeutung zu. Diese schon zu Anfang erwähnte Technik testet das Programm gegen erfahrungsgemäß kritische Werte wie 0, -1, "", etc. Man spricht hier auch von *error guessing*.

Liggesmeyer gibt folgende Regeln zur Bildung von Äquivalenzklassen an[8]:

1. Falls die Eingabebedingung einen zusammenhängenden Wertebereich oder eine Anzahl von Werten spezifiziert, so werden eine gültige und zwei ungültige Äquivalenzklassen gebildet. (Z.B. natürliche Zahlen kleiner 100)
2. Falls eine Eingabebedingung eine Menge von Werten spezifiziert, die unterschiedlich behandelt werden, so ist für jeden Wert eine eigene gültige Äquivalenzklasse zu bilden. Für alle Werte mit Ausnahme der gültigen Werte ist eine ungültige Äquivalenzklasse zu bilden. (Z.B. männlich, weiblich)
3. Falls eine Eingabebedingung eine Situation festlegt, die zwingend erfüllt sein muß (z.B. Buchstabe als erstes Zeichen), so sind eine gültige und eine ungültige Äquivalenzklasse zu bilden.

[8] Liggesmeyer S.49, auch Balzert

4. Falls Grund zu der Annahme besteht, dass Element einer Äquivalenzklasse unterschiedlich behandelt werden, so ist die Klasse entsprechend aufzutrennen.

Bei der Auswahl von Testwerten ist darauf zu achten, beim Test mit einem Wert aus einer ungültigen Äquivalenzklasse, für alle übrigen Werte (soweit benötigt) nur Werte aus gültigen Äquivalenzklassen zu verwenden, da nur so eine Zuordnung der Fehlerbehandlung möglich ist.

- **Beispiel: Tarifrechner einer Krankenversicherung**

Als Beispiel (auch für die Vorführung der Werkzeuge) wird eine Tarifkombination aus der Krankenversicherung herangezogen.

Sei AJ ein Tarif für eine Auslandsreisekrankenversicherung. Der Tarif kann von Personen bis 69 Jahren zu einem Tarifbeitrag von 8€ abgeschlossen werden. Ältere Personen zahlen 25€. Der Tarif gilt für ein Jahr und für Auslandsreisen, die nicht länger als 56 Tage dauern.

Sei AR ein Tarif für eine Auslandsreisekrankenversicherung. Der Tarifbeitrag wird nach der Dauer der Auslandsreise berechnet. Personen bis 69 Jahre können Reisen bis zu 90 Tagen zu einem Beitrag von 30 Cent pro Tage versichern. Jeder Tag, der darüber hinaus versichert wird, kostet 1,50€. Ältere Personen können Reisen bis zu 90 Tagen für 1€ pro Tag versichern. Längere Reisen werden für diese Personengruppe nicht versichert. Der Mindestbeitrag für alle Personen beträgt 3€.

Wenn der Tarif AJ vor mehr als einem Jahr abgeschlossen wurde, können AJ und AR können kombiniert werden, so dass Reisen bis zu 56 Tagen durch den Tarif AJ versichert sind und für weitere Tage ein Tarif AR abgeschlossen werden kann.

Es ergeben sich folgende 22 Eingabeäquivalenzklassen:

Tarif	Alter	Reisedauer	Kombination	gültig/ungültig
AJ	>=70	-	-	gültig
	1-69	-	-	gültig
	<=0	-	-	ungültig
AR	>=70	<=0		ungültig
	>=70	>=1;<=?	sinnlos	gültig/ungültig
	>=70	>=?;<=56	sinnlos	gültig/ungültig
	>=70	>=57;<=90	möglich	gültig
	>=70	>=91		ungültig
	1-69	<=0		ungültig
	1-69	>=1;<=?	sinnlos	gültig/ungültig
	1-69	>=?;<=56	sinnlos	gültig/ungültig
	1-69	>=57;<=90	möglich	gültig
	1-69	>=91;<=365	möglich	gültig
	1-69	>=366		ungültig
	<=0			ungültig

Tab. 1

Durch die Kombination mit einem AJ ergeben sich immer zwei Klassen, die hier in einer Zeile zusammengefaßt sind.

Durch den Mindestbeitrag ergibt sich der Tarifbeitrag einer sehr kurzen Reise nicht durch das Produkt von Reisedauer und Beitrag für einen Tag (30ct bzw. 1€), so dass hier nach Regel 4 eine eigene Äquivalenzklasse zu bilden ist.

Zu den Eingabeäquivalenzklassen werden folgende 16 Testfälle gebildet:

Tarif	Alter	Reisedauer	Komb. AJ	Tarifbeitrag
AJ	Alt (>=**70**)	-	-	25€
AJ	Jung (<=**69**)	-	-	8€
AR	Alt	Kurz (<=**56**)	Nein	56€
AR	Alt	Kurz (<=**56**)	Ja	sinnlos
AR	Alt	Kurz (>=**57**;<=90)	Nein	57€
AR	Alt	Kurz (>=**57**;<=90)	Ja	28€ (Mindestb.)
AR	Alt	Mittel (>=**91**)	-	ungültig
AR	Jung	<=**0**	-	ungültig
AR	Jung	Kurz (**1**)	Nein	3€ (Mindestb.)
AR	Jung	Kurz (<=**56**)	Nein	16,80€
AR	Jung	Kurz (<=**56**)	Ja	sinnlos
AR	Jung	Kurz (>=**57**;<=90)	Nein	17,10€
AR	Jung	Kurz (>=**57**;<=90)	Ja	11€ (Mindestb.)
AR	Jung	Mittel (>=**91**;<=365)	Nein	28,50€
AR	Jung	Mittel (>=**91**;<=365)	Ja	19,70€
AR	Jung	>=**366**	-	ungültig

Tab. 2

Als Alter werden immer die Randwerte getestet. Das Programm unterscheidet nur zwischen <70 und >= 70. Darum werden keine Testfälle für den Fall <=0 aufgenommen. Die Fälle Reisedauer <=0 (Fehler) und Reisedauer sehr kurz (Mindestbeitrag) werden nur für eine Altersgruppe getestet.

Zustandsbasiertes Testen

Mitunter ist es erforderlich, die Zustände, in denen sich ein System befinden kann, in die Vorbereitung eines Tests einzubeziehen. Eine, auch in der UML vorhandene und verbreitete Darstellung für zustandsbasierte Systeme ist der **Zustandsautomat**.

Zustandsautomaten sind ein hervorragendes Mittel, um Systeme, die auf Ereignisse reagieren, zu beschreiben. Darunter fallen, zumindest in Teilen, eigentlich alle objektorientierten Programme. Nicht die gesamte Funktionalität läßt sich geeignet zustandsbasiert beschreiben. Als Ergänzung bietet sich häufig ein Sequenzdiagramm an, aus welchem man wiederum Tests ableiten kann.

Zustandsautomaten bestehen aus Zuständen und Zustandsübergängen und veranschaulichen Zusammenhänge, die sich nur schwer in einem Text beschreiben lassen. Außerdem hilft die graphische Darstellung Unvollständigkeiten oder Fehler in der Spezifikation zu erkennen. Im Beispiel wird man vielleicht erwarten, dass der Zustand „wartet auf Geld“ nicht nur durch den Einwurf ausreichender Münzen beendet werden kann.

- **Beispiel: Kartenautomat in einem Parkhaus**[9]

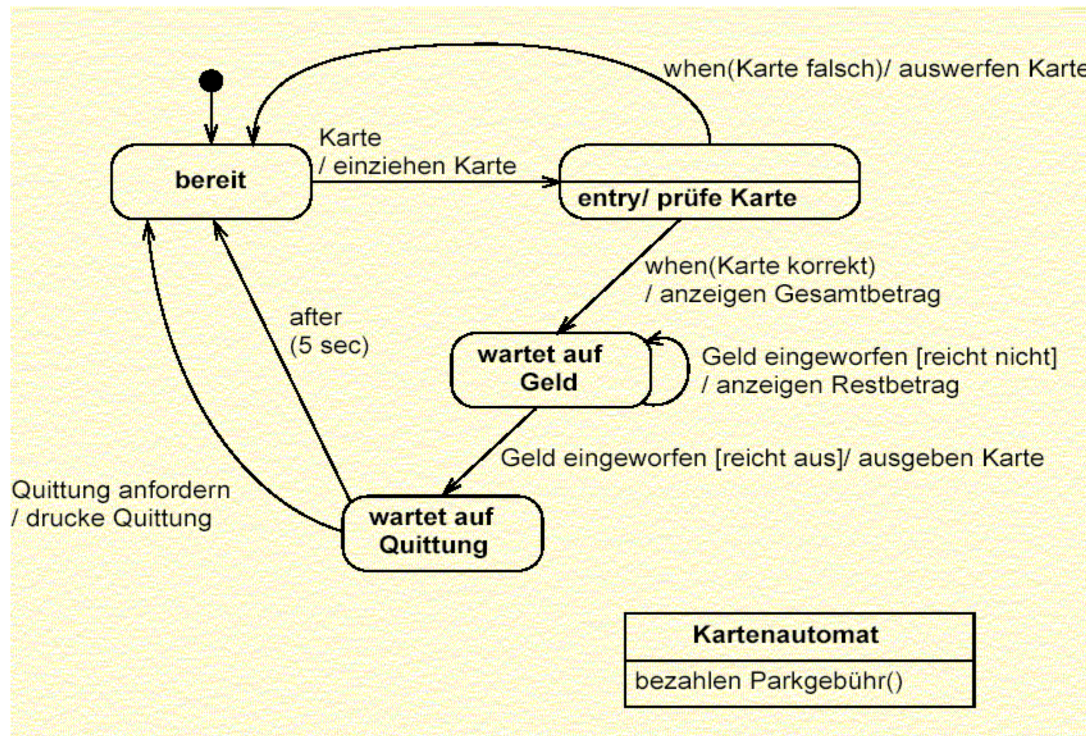

Abb. 2

Zu erkennen ist, dass es nicht nur einen Übergang von „wartet auf Quittung“ zu „bereit“ gibt. Es gibt auch nicht nur einen Zustand, von dem aus das System in den Zustand „bereit“ gelangen kann.

Daraus kann man unmittelbar ableiten, dass ein Test, der jeden Zustand einmal durchläuft, nicht vollständig ist, da nicht gewährleistet ist, dass alle Übergänge getestet wurden. Die Forderung der Zustandsüberdeckung ist also in diesem Fall nicht ausreichend. Ähnliche Probleme werden im nächsten Abschnitt zu kontrollflussorientierten, strukturorientierten Tests besprochen. Grundsätzlich gilt, dass beim Durchlaufen aller Zustandsübergänge auch alle Zustände durchlaufen werden.

Die Übersetzung eines Zustandsautomaten in eine Zustandsübergangstabelle geschieht, indem die Zustände und Ereignisse in einer Matrix eingetragen werden.

Zustand/ Ereignis	bereit	entry	wartet auf Geld	wartet auf Quittung
Karte	entry			
Karte falsch		bereit		
Karte korrekt		wartet auf Geld		
Geld [reicht]			wartet auf Qu.	
G. [reicht nicht]			wartet auf Geld	
Quittung				bereit
„Timeout“				bereit

Tab. 3

9 geklaut aus http://lls.informatik.uni-oldenburg.de/lehre/vorlesungen/ss2002_wiwisowi2/vl6.pdf

Sofort fällt auf, dass die meisten Felder nicht gefüllt sind. Die bisher aus der Spezifikation abgeleiteten Testfälle sind Normalfälle (erste Gruppe). Die Spezifikation ordnet hier Folgezustände zu, deren Eintreten getestet werden muss. In allen anderen Fällen (zweite Gruppe) aber wird das System das Ereignis ignorieren und den aktuellen Zustand beibehalten (wollen). Auch das kann und sollte getestet werden! Als dritte Gruppe gibt Liggesmeyer Ereignisse an, die, falls sie in einem Zustand auftreten, einen Fehler darstellen und eine gesonderte Behandlung erfordern. So könnte beispielsweise diskutiert werden, ob und wie das System auf die Ereignisse Karte falsch oder Geld in anderen als den bisher zugelassenen Zuständen reagieren soll.

Anmerkung: Vorstellbar wäre, das dem Zustandsübergang „wartet auf Geld“ zugeordnete Ereignis „Geld eingeworfen [reicht nicht]“ in verschiedene Ereignisse aufzuspalten, die z.B. die akzeptierten Münzen unterscheiden. Ein vollständiger Test würde dann neben der Überdeckung der Zustandsübergänge auch die Überdeckung der Ereignisse (die zu den Zustandsübergängen führen) berücksichtigen müssen.

Anmerkungen: Korrekt wäre es, hier dem neuen Zustand auch die zum Übergang gehörige Aktion zuzuordnen. Für das Verständnis der Testfallermittlung ist das hier aber nicht notwendig.

Sequenzdiagramme

- **Beispiel: Kunde login**

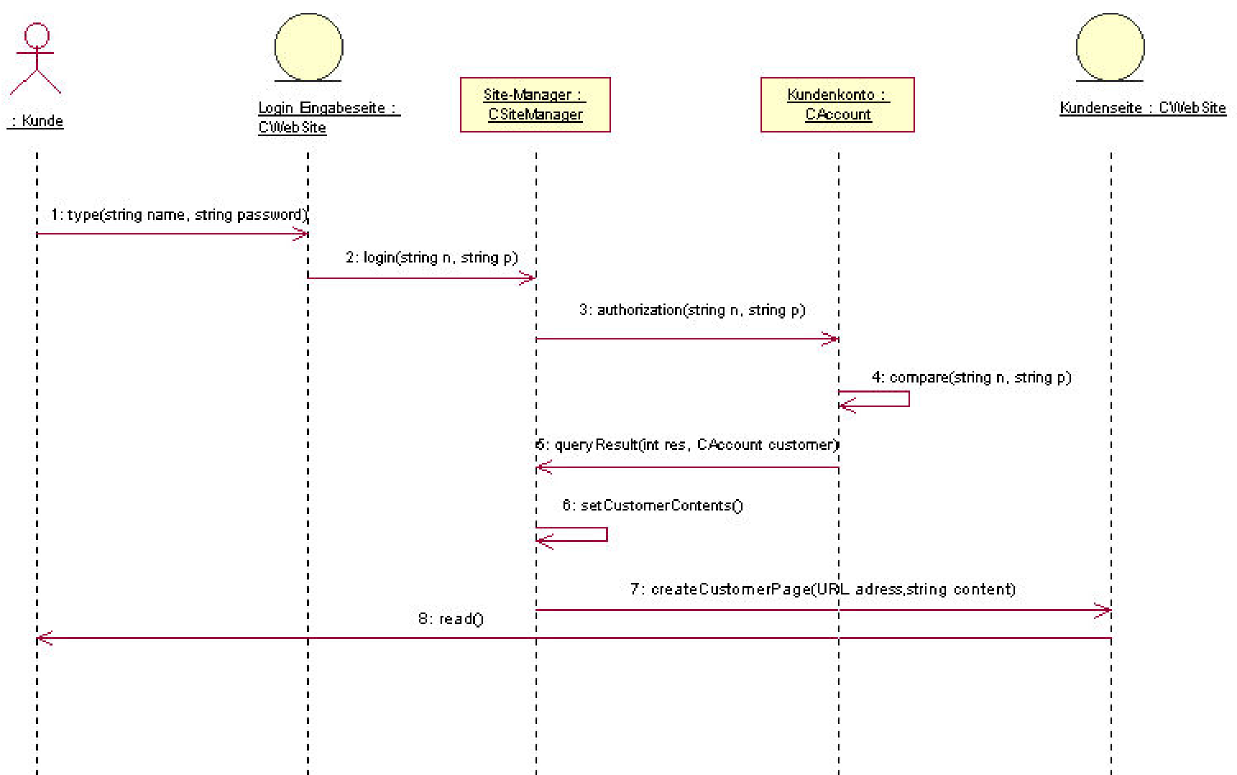

Abb. 3

Üblicherweise mit Use Cases assoziiert sind Sequenzdiagramme, die die Interaktion von Objekte im Zeitfluss (von oben nach unten) angeben. [10]

Sequenzdiagramme wird man daher nicht für alle möglichen Pfade erstellen, sondern vorrangig für „Normalfälle“ (happy day-Szenarien) und wichtige Abläufe. Sie sind daher für einen vollständigen Test nicht geeignet, lassen sich dafür aber fast unmittelbar in Testfälle „übersetzten“.

Testen graphischer Benutzeroberflächen

Eine besondere Form des black-box-Tests ist das Testen graphischer Benutzeroberflächen. Die meisten Anwendungen werden inzwischen über eine solche Oberfläche bedient.

Da viele Testfälle bestimmte Zustände voraussetzen, die unter Umständen nicht trivial herzustellen sind, kann ein solcher Test ohne geeignete Werkzeuge zu einem äußerst mühseligen Unterfangen werden. Darum wird hier in der Regel eine hohe Automatisierung angestrebt. So wird zum einen eine (schnelle) Reproduzierbarkeit gewährleistet und eine hohe Abdeckung im white-box-Test erst möglich. Sog. Capture-Replay-Tools unterstützen/leisten hier die Automation.

Sie ermöglichen das Aufzeichnen der Benutzerevents wie Mausaktionen (Klick, Move, Drag&Drop) und Eingaben über die Tastatur. Sie ermöglichen auch den Vergleich von Ausgaben (Texte, Meldungen, etc.) mit Erwartungswerten.

Dabei verwenden die Werkzeuge unterschiedliche Techniken. Jedes Tool ist in der Lage einen Arbeitsgang mitzuschneiden (Capture). Komfortablere Werkzeuge können diese Mitschnitte in einem Protokoll auch textuell darstellen. Dabei werden Controls der GUI im Idealfall erkannt und unterschieden. Der Vorteil besteht darin, dass spätere kleine Veränderungen bzgl. Größe und Anordnung oder Platzierung von Fenstern, keine Rolle spielen, da das Werkzeug benannte Element wiederfindet. Reine Aufnahmeverfahren versagen an solchen Stellen.

Allen Werkzeugen gemeinsam ist die Notwendigkeit stets mit exakt dem gleichen Startzustand zu beginnen. So ist zum Beispiel beim Erstellen einer Datei keine Rückfrage an den Anwender üblich. In einem zweiten Durchlauf wäre diese Datei aber schon vorhanden, so dass eine Meldung (Datei überschreiben?) erscheint, mit der das Replay nicht rechnet und folglich mit einem Abbruch und/oder einer Fehlermeldung reagiert. Startzustand bedeutet in diesem Fall, dass die Datei nicht vorhanden sein darf.

Einige Tools (textuelle Darstellung der Mitschnitte vorausgesetzt) stellen eine eigene Scriptsprache zur Verfügung, mit der die Mitschnitte bearbeitet, verfeinert oder auch parametrisiert werden können.

Die wichtigsten Tools sind Rational (nun IBM) Robot, Compuware QA Run und Mercury WinRunner (der Mercedes unter den Werkzeugen).

[10] Terry Quatrani; Visual Modeling with Rational Rose und UML; Addison Wesley, 1998

Beschränkungen

Neben den Vorteilen hat der black-box-Test auch Nachteile. Die Bildung von Äquivalenzklassen bzw. die Auswahl der Testdaten bleibt dem Tester überlassen, da das System selbst nicht offen liegt. Eine Automatisierung ist nicht möglich.

Gleichzeitig bleiben Zusammenhänge innerhalb des Programmes verborgen. Wechselwirkungen und Seiteneffekte können, wenn sie nicht ausdrücklich in der Spezifikation benannt sind, nicht gezielt getestet werden.

Ein Maß für die Testüberdeckung kann nicht angegeben werden.

Kontrollflussorientiertes, strukturorientiertes Testen

Steht dem Tester mehr als „nur“ die Spezifikation und das „fertige“ Programm zur Verfügung, kann er zum Test die Struktur des Programmes heranziehen. Da Tests dieser Kategorie das Programm nicht „von außen“ betrachten, sondern sein Inneres beleuchten, bezeichnet man sie als **white-box-Test** (oder auch glass-box-Test).

Einerseits setzt der white-box-Test bei den Schwachstellen des black-box-Tests an, da er
- den Objektzustand der Klasse mit berücksichtigt
- einen Test so konstruieren kann, dass *alle* Anweisungen getestet werden und
- einen Test (theoretisch) so konstruieren kann, dass *alle* möglichen Pfade durchlaufen werden (Testdeckung).

Andererseits setzt er detaillierte Kenntnisse über
- den Aufbau der Objekte
- die Abläufe im Programm (z. B. event-handling, etc.) und
- die Struktur des Programmes (Schnittstellen, Module)

voraus.

Zu einem solchen Test sind Fachabteilung und Auftraggeber kaum in der Lage. Größere Häuser bilden eigene Abteilungen, die sich mit dieser Art des Testens befassen. Es erscheint logisch, dass ein Tester in der Regel die Entstehung eines Programmes von seiner Spezifikation bis zur Auslieferung begleitet. Neben der Kenntnis über Technik und Werkzeuge benötigt der Tester Wissen über die Entwicklungsumgebung und natürlich das Programm selbst.

Firmen definieren Qualitätsstufen, die ein Programm erreichen muss. Liggesmeyer gibt in der Einleitung seines Buches Beispiele für solche Qualitätsstufen.

Allerdings kommt auch oft dem Entwickler selbst die Aufgabe zu, sein Programm auf „Herz und Nieren“ zu testen. Kenntnisse über Programm und SEU sind beim Programmierer ohne Frage vorhanden. Kenntnisse über Testmethoden und Werkzeuge sind in solchen Fällen hilfreich.

Aufzählung und Gewichtung der Verfahren

Die Umfänge möglicher Tests sind sehr unterschiedlich. Es gibt einfache Tests, die schnell und ohne große Vorbereitung durchgeführt werden können (ad hoc Test), dafür aber nicht sehr fehlersensitiv sind. Andere Tests sind aufgrund der Vielzahl der zu prüfenden Testfälle garantiert sehr viel effektiver aber in der Praxis meist nicht möglich.

Um a+b für zwei 32 Bit Integerzahlen zu testen sind $2^{32}*2^{32}=10^{20}$ Tests notwendig. Bei 10 Milliarden Tests pro Sekunde dauert dies 325 Jahre.[11]

Zu n Schleifendurchläufen gehören 2^n Pfade. Neben der zu hohen Pfadanzahl, sind nicht alle konstruierbaren Pfade auch ausführbar, was nicht als Fehler gewertet werden darf.[12]

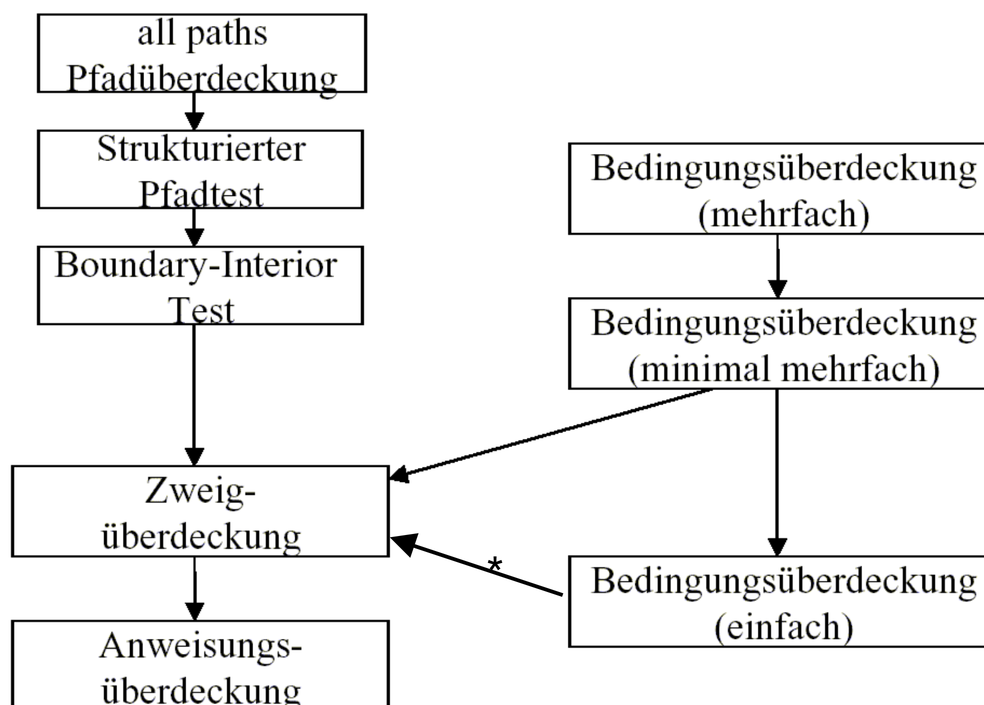

Abb. 4 [13]

Die Übersicht kann auch als Mengendiagramm gelesen werde. Der jeweils untere Test ist eine Teilmenge des übergeordneten Tests. So ist der Anweisungsüberdeckungstest z. B. im Zweigüberdeckungstest vollständig enthalten.

Mit C_x wird durch ein Verhältnis ein Testmaß definiert, dass 1 erreichen soll. x läuft dabei von 0 bis 4 und gibt die steigende Güte der Testmethode wieder.

Die Testfälle müssen jeweils so gewählt werden, dass die Testbedingung $C_x>=1$ erreicht wird.

---

[11] Vorlesung Formale Methoden H. Schmitz, FH-Trier, WS 2003/2004
[12] Liggesmeyer, S.133
[13] geklaut aus http://www-is.informatik.uni-oldenburg.de/~sauer/lehre/swp_01_02/vl_s5.pdf (sehr gut!)

Anmerkung *: Für den einfachen Bedingungsüberdeckungstest gilt der gezeigte Zusammenhang nur bei vollständiger Auswertung aller Teilentscheidungen. Andernfalls schließt auch dieser Test die Zweigüberdeckung ein.

Anweisungsüberdeckungstest (C_0)

Als Testmaß wird der Anweisungsüberdeckungsgrad definiert. Er ist das Verhältnis

$$C_0 = \frac{\text{Anzahl der ausgeführten Anweisungen}}{\text{Gesamtzahl der Anweisungen}}$$

Eine abweisende Schleife wie *do something() while (a<1)* wird in jedem Fall einmal durchlaufen. Die Anweisungsüberdeckung ist erreicht. Ob die Bedingung a<1 korrekt ist, bleibt unbeantwortet. Lt. Unterlagen werden durch den Test 18% der Fehler entdeckt[14].

Bewertung:
C_0 ist die einfachste kontrollflussorientierte Testtechnik, die noch keine brauchbare Aussage über die Qualität der Software machen kann.

Zweigüberdeckung (C_1)

Als Testmaß wird der Zweigüberdeckungsgrad definiert. Er ist das Verhältnis

$$C_1 = \frac{\text{Anzahl besuchter Zweige}}{\text{Gesamtzahl der Zweige}}$$

In einer Fallunterscheidung *if (a>1) doThis() else doThat()* werden bei Testfällen a=0 und a=5 beide Methoden ausgeführt. Die Anweisungsüberdeckung ist erreicht. Hätte die richtige Bedingung lauten müssen a>=1, wird der Fehler nicht erkannt. Mit Ausnahme des Wertes 1 arbeitet das Programm immer korrekt.

Zu Beginn eines Tests wird jeder weitere Testfall die Zahl der überdeckten Zweige stark ansteigen lassen. Um eine vollständige Zweigüberdeckung zu erreichen, ist eine erheblich größere Anzahl Testfälle notwenig. Dieses Verhältnis ist bei weitem <u>nicht</u> linear. Einige Zweige (z. B. Speicherplatzprobleme, Systemfehler, etc.) sind unter Umständen nur durch erheblichen (und evtl. nicht gerechtfertigten) Aufwand zu erreichen. Eine Aussage „Ein Test mit einer Zweigüberdeckung von 80% ergab...“ macht daher keine wirkliche Qualitätsaussage.

Dieser rechnerischen Problematik kann man begegnen, indem man nur solche Zweige als Zweige zählt, deren Ausführung sich nicht zwangsläufig an die Ausführung eines anderen Zweiges anschließt. Es ergibt sich wieder ein lineares Verhältnis zwischen Überdeckung und Testfallanzahl.[15]

Bewertung:
C_1 gilt als minimales Testkriterium und ist Bestandteil aller höheren Techniken (vgl. Übersicht). Der Test konzentriert sich auf die Kanten des Kontrollflussgraphen. Bedingungen und Schleifen werden nicht speziell getestet. Die Anweisungsüberdeckung ist vollständig enthalten.

---

[14] Liggesmeyer S. 84 oben
[15] Liggesmeyer S. 87

Einfacher Bedingungsüberdeckungstest (C_2)

Als Testmaß wird der Deckungsgrad der atomaren Prädikate gewählt:

$$C_2 = \frac{\text{Anzahl atomare Prädikate}_{\text{TRUE}} + \text{Anzahl atomarer Prädikate}_{\text{FALSE}}}{2 * \text{Gesamtzahl aller atomaren Prädikate}}$$

Das bedeutet, das die Testfälle so zu wählen sind, dass alle atomaren Prädikate einmal zu TRUE und einmal zu FALSE ausgewertet werden.

Der Bedingungsüberdeckungstest betrachtet auch die logische Struktur des Programmes und der zusammengesetzten Entscheidungen.

Problematik:
Eine einfach ODER-Verknüpfung A || B ergibt

A	B	A|B	
1	1	1	
1	0	1	x
0	1	1	x
0	0	0	

Tab. 4

A	B	A||B	
1	-	1	x
0	1	1	x
0	0	0	x

Tab. 5

In Tabelle 4 werden immer beide Terme ausgewertet. Die zweite und dritte Zeile (x) genügen den Anforderungen der Bedingungsüberdeckung, ohne dass als resultierender Wahrheitswert FALSE ermittelt wird. Der ELSE-Zweig des Programmes (so es einen gibt) wird nicht durchlaufen. Eine Anweisungs- oder Zweigüberdeckung wurde also nicht erreicht.

Die meisten Compiler optimieren heute den Code so, dass Bedingungsketten nur so lange ausgewertet werden, bis das Ergebnis feststeht. Der Wahrheitswert von B hat im ersten Fall (A=TRUE) keine Auswirkung auf den resultierenden Wahrheitswert, der in jedem Fall TRUE ist.

Bei der partieller Evaluation der Terme in Tabelle 5 fallen die ersten beiden Zeilen der vierten Tabelle zusammen. Um eine Bedingungsüberdeckung zu erreichen, sind alle drei Fälle erforderlich. Dann wird auch eine Anweisungsüberdeckung erreicht, da der resultierende Wahrheitswert sowohl zu TRUE als auch zu FALSE ausgewertet wird. Dieser Zusammenhang gilt grundsätzlich. Bei teilweiser Auswertung der Entscheidungen subsumiert der einfache Bedingungsüberdeckungstest den Zweigüberdeckungstest.

Bewertung:
Bei vollständiger Auswertung nicht ausreichend. Ansonsten sehr geeignet, da die Zahl der Testfälle linear zur Zahl der Bedingungen steigt.

Bedingungs-/Entscheidungsüberdeckungstest

Dieser Spezialfall verlangt im Falle vollständiger Auswertung der Entscheidungen von den Testfällen zusätzlich eine vollständige Zweigüberdeckung.

Mehrfach-Bedingungsüberdeckungstest (C_3)

Als Testmaß wird der Deckungsgrad aller Variationen atomarer Prädikate gewählt:

$$C_3 = \frac{\text{Anzahl möglicher Wertekombinationen}_{TRUE} + \text{Anzahl mögl. Kombinationen}_{FALSE}}{2 * \text{Gesamtzahl aller möglichen Wertekombinationen}}$$

Alle möglichen Wertekombinationen müssen einmal TRUE und einmal FALSE sein.

Unabhängig von der Verknüpfungslogik werden auch alle Pfade erreicht, so dass eine Zweigüberdeckung enthalten ist.

Für eine Bedingung, die aus n atomaren Bedingungen besteht, führt dies zu 2^n Variationsmöglichkeiten. Davon sind in der Regel einige Fälle gar nicht erreichbar (vgl. Kopplung beim modifizierten Bedingungs-/Entscheidungsüberdeckungstest). Diese Fälle müssen dennoch identifiziert werden, da sie nicht als Fehler gewertet werden dürfen. Werden zusammengesetzte Entscheidungen nur teilweise evaluiert, ist es schon rein technisch nicht möglich, alle Kombinationen zu bilden.

In diesen Fällen ist es unmöglich, ein Testmaß zu definieren, da der geforderte Deckungsgrad 1 nicht erreicht werden kann.

Bewertung:
Obwohl der Mehrfach-Bedingungsüberdeckungstest alle Bedingungen vollständig testet, ist er in der Praxis nicht relevant. Das liegt einerseits an der großen Menge möglicher Testfälle (2^n) und andererseits an der beschriebenen Schwierigkeit einen hohen Deckungsgrad zu erreichen.

Minimaler Mehrfach-Bedingungsüberdeckungstest (Mischung C_2 und C_3)

Als Testmaß wird die Überdeckung der Prädikate gewählt. Gemessen wird:

$$C_{MinMehrfach} = \frac{\text{Anzahl Prädikate}_{TRUE} + \text{Anzahl Prädikate}_{FALSE}}{2 * \text{Anzahl aller Prädikate}}$$

Bei diesem Test wird gefordert, dass neben allen atomaren Prädikaten auch alle zusammengesetzten Prädikate einmal zu TRUE und einmal zu FALSE ausgewertet werden. Ist dies nicht möglich, ist die Bedingung überflüssig und kann anders (einfacher) formuliert werden.

	A	B	A&&B	C	(A&&B)||C	1.	2a	2b
1	1	1	1	(1)	1	X	X	X
2	1	1	1	(0)	1			
3	1	0	0	1	1		X	
4	1	0	0	0	0			X
5	0	(1)	0	1	1			X
6	0	(1)	0	0	0			
7	0	(0)	0	1	1			
8	0	(0)	0	0	0	X	X	

Tab. 6

Auch in diesem Test ist es von wesentlicher Bedeutung, ob die Wahrheitswerte vollständig ausgewertet werden, oder ob das Compiler Optimierungen durchführt. Die in Klammern gesetzten Wahrheitswerte in der Tabelle werden bei der Bedingung (A&&B)||C im Falle einer unvollständigen (optimierenden) Auswertung nicht evaluiert.

Das hat Auswirkungen auf die Auswahl der Testfälle. Im ersten Fall (1.) reichen die markierten Fälle 1 und 8 aus, um das Testmaß zu erreichen. Alle einfachen und zusammengesetzten Bedingungen werden einmal zu TRUE und einmal zu FALSE ausgewertet.

Bei unvollständiger Evaluation werden unter 1. C bzw. B nicht ausgewertet. Somit fehlen Fälle, in denen C zu TRUE bzw. B zu FALSE ausgewertet wird. Das Testmaß wird nicht erreicht. Es müssen weitere Fälle betrachtet werden. Die Wahl der Testfälle obliegt dem Tester (oder der Test-Software). Hier sind unter 2a und 2b zwei Möglichkeiten (mit je drei Testfällen) gezeigt. Es gibt weitere. Im Allgemeinen sind bei unvollständiger Auswertung von zusammengesetzten Bedingungen mehr Testfälle erforderlich, als bei vollständiger Auswertung aller Bedingungen.

Bewertung:
Der minimale Mehrfach-Bedingungsüberdeckungstest gewährleistet eine Zweigüberdeckung und stellt insbesondere ein erfüllbares Testmaß zur Verfügung. Allerdings ist auch dieser Test kaum in der Lage logische Fehler in einer Verknüpfung von Bedingungen zu erkennen (z.B. A&&B statt A||B).

Modifizierter Bedingungs-/Entscheidungsüberdeckungstest

Dieser Test berücksichtigt stärker als alle anderen Tests die Logik der Verknüpfungen und bleibt im Verhältnis atomarer Teilentscheidungen zur Anzahl erforderlicher Testfälle linear. Bei n atomaren Bedingungen sind mindestens n+1 Testfälle erforderlich.

Durch die Bildung von Testpaaren, wird untersucht, wie die <u>Veränderung einer</u> Teilentscheidung die Gesamtentscheidung <u>unabhängig von den übrigen</u> Teilentscheidungen beeinflußt.

Dazu werden Testpaare so ausgewählt, dass sich nur genau ein Wahrheitswert eines atomaren Prädikates ändert. Diese Veränderung muss dann ebenfalls zur Änderung des Wahrheitswertes der Gesamtentscheidung führen.

	A	B	C	(A&&B)||C	1.	2.	3.
1	1	1	(1)	1			
2	1	1	(0)	1	X	X	
3	1	0	1	1			
4	1	0	0	0		X	
5	0	(1)	1	1			X
6	0	(1)	0	0	X		X
7	0	(0)	1	1			
8	0	(0)	0	0			

Tab. 7

Bei unvollständiger Auswertung der Entscheidung, wird statt unveränderter Wahrheitswerte der verbleibenden Prädikate auch zugelassen, dass ein Prädikat nicht evaluiert wird.

In beiden Fällen wird beim Test der Entscheidung (A&&B)||C (n=3) mit vier Testfällen die Forderung des modifizifierten Bedingungs-/Entscheidungsüberdeckungstest erfüllt.

Für Werkzeuge zur Ermittlung von Testfällen können bei dieser Methode Schwierigkeiten auftreten, wenn Bedingungen voneinander abhängen. So sind bei einer Entscheidung key==‘a‘ || key==‘b‘ || key==‘c‘ nicht die acht kombinatorischen Möglichkeiten, sondern nur vier Kombinationen darstellbar, da alle Bedingungen von der selben Variablen abhängen. Man spricht in diesem Fall von schwach gekoppelten atomaren Teilentscheidungen.

Abhängigkeiten entstehen auch durch eine starke Kopplung. Wenn zum Beispiel a und !a in einer Entscheidung vorkommen, kann immer nur genau eine der beiden Teilentscheidungen wahr oder falsch sein.

Bewertung:
Der modifizierte Bedingungs-/Entscheidungsüberdeckungstest enthält einen minimalen Mehrfach-Bedingungsüberdeckungstest und berücksichtigt stärker die Verknüpfungslogik bei gleichzeitigem linearen Komplexitätsanstieg (n+1). Als Testtechnik ist er damit (neben dem minimalen Mehrfach-Bedingungsüberdeckungstest) am besten geeignet Bedingungen zu testen.

Pfadüberdeckungstest (C_4)

Als Testmaß wird der Überdeckung der Programmpfade gewählt:

$$C_4 = \frac{\text{Anzahl besuchter Pfade}}{\text{Gesamtzahl der Pfade}}$$

Zwei Pfade sind dann identisch, wenn die Sequenz ihrer Knoten in einem Kontrollflußgraphen identisch ist. Insbesondere ein Schleifendurchlauf fügt dieser Sequenz stets mindestens einen Knoten hinzu, so dass bei Schleifen ohne feste Wiederholungszahl die Anzahl der Pfade meist beliebig groß werden kann.

Dadurch verliert dieses umfassendste[16] kontrollflussorientierte Testverfahren aber auch seine praktische Relevanz.

Bewertung:
In der Regel ist es nicht möglich, alle Pfade zu durchlaufen, da diese unendlich oder zumindest sehr groß sind. So kann kein sinnvolles Testmaß angegeben werden, da 1 nicht erreichbar ist und eine hohe Zahl besuchter Pfade gegen Unendlich immer noch nahe 0 liegt.

Darüber hinaus leistet der Pfadüberdeckungstest keine Bedingungsüberdeckung, so dass er alleine noch kein ausreichendes Testmaß ist.

[16] umfassender ist nur der erschöpfende Test mit allen möglichen Eingabewerten. Liggesmeyer S.132

Eine Annährung an den Pfadüberdeckungstest ist der

Boundary-interior-Test

Der Boundary-interior-Test führ einen eingeschränkten Pfadtest durch. Durch Gruppenbildung soll die Anzahl der Testfälle (und Pfade) gering gehalten werden. Liggesmeyer bezeichnet mit k die Anzahl der Durchläufe des Schleifenrumpfes, die betrachtet werden.

Außerhalb der Schleife werden alle Pfade betrachtet. Innerhalb der Schleife werden drei Gruppen unterschieden: Die Schleife wird nicht durchlaufen. Die Schleife wird genau einmal durchlaufen (Grenztest oder boundary test). Die Schleife wird zwei (oder mehrmals) durchlaufen (interior test).

Zu beachten ist, dass innerhalb der Schleife (sowohl im boundary als auch im interior test) unterschiedliche Pfade (Fallunterscheidungen und geschachtelte Schleifen) durchlaufen werden können. Auch können Schleifen an unterschiedlichen Stellen verlassen (oder betreten) werden. In diesen Fällen sind in den betroffenen Gruppen jeweils Untergruppen mit eigenem Testfall zu bilden.

Bei nicht-abweisenden (fußgeprüften) Schleifen (immer mindestens ein Schleifendurchlauf) oder in Kombination mit Bedingungen (Fallunterscheidungen oder Schleifenbedingungen) kann es vorkommen, dass nicht für jede Gruppe ein Testfall möglich ist oder Pfade in den ersten k Ausführungen des Schleifenrumpfes nicht durchlaufen werden. Das bedeutet, dass eine Zweigüberdeckung als minimales Testkriterium nicht immer gegeben ist.

Liggesmeyer fügt (in einem modifizierten boundary-interior-Test) die Forderung der Zweigüberdeckung explizit hinzu.

Er schlägt darüber hinaus vor, auch die maximale Anzahl von Schleifendurchläufen als Testfall aufzunehmen sowie den Versuch zu unternehmen, diese zu überschreiten.

Bewertung:
Wegen der möglichen Abhängigkeiten zwischen Variablen vor, in und hinter der Schleife ist diese Gruppenbildung besonders sinnvoll. Der Test umfaßt mit der vorgeschlagenen Ergänzung den Zweigüberdeckungstest und hat eine um 100% bessere Effektivität als dieser[17].

Strukturierter Pfadtest

Der eben beschriebene Boundary-interior-Test ist ein Spezialfall des strukturierten Pfadtest mit k=2 (Durchläufe 0,1 und 2 mal). Allgemein wird beim strukturierten Pfadtest ein k festgelegt, wodurch die Anzahl der zu betrachtenden Gruppen pro Scheife angegeben wird.

[17] http://www-is.informatik.uni-oldenburg.de/~sauer/lehre/swp_01_02/vl_s5.pdf, Seite 37

Der von Liggesmeyer vorgestellt **LCSAJ-Test** (Linear Code Sequenz And Jump) wird hier nicht besprochen, da er vor allem für solche Programme gewählt wird, die reichlich Gebrauch von Sprüngen machen. Da GOTOs aber in vielen Programmiersprachen nicht mehr vorkommen oder zumindest von „guten“ Programmierern nicht verwendet werden, verliert der Test zunehmend an Bedeutung.

Werkzeuge für das Testen von Software

JUnit

Schon erwähnt wurde JUnit, ein Open-Source-Testframework, das inzwischen sehr weit verbreitet ist. JUnit liegt ein einfaches Design zugrunde. Tests werden aus Testklassen zusammengesetzt. Ein solcher Testcase erbt von Test. Testsuite ist ebenfalls ein Test, dient aber dazu, unterschiedliche Tests zusammenzufassen. Damit können auch Testsuites in einer Suite zusammengefaßt werden.

Üblich ist es, pro Testklasse eine statische Factorymethode suite() zur Verfügung zu stellen, die entweder eine Suite von sich selbst oder die Instanz einer Suite bestehend aus anderen Tests zurückgibt. Beim Erzeugen, werden über Reflection alle Methoden eingesammelt, die mit *test* beginnen. Diese werden, eingefaßt von setUp und tearDown beim Test ausgeführt. Neben einer TextUI stellt JUnit verschiedene GUIs zur Verfügung.

Die Testmethoden selbst machen Gebrauch von der Klasse Assert, über die eine Vielzahl von Überprüfungen möglich sind. So können Gleichheiten, Unterschiede und Wahrheitswerte gefordert werden. Daneben ist es möglich, auch unmittelbar eine Fehlermeldung abzusetzen. Alle Forderungen werden so formuliert, dass sie im Normalfall TRUE liefern.

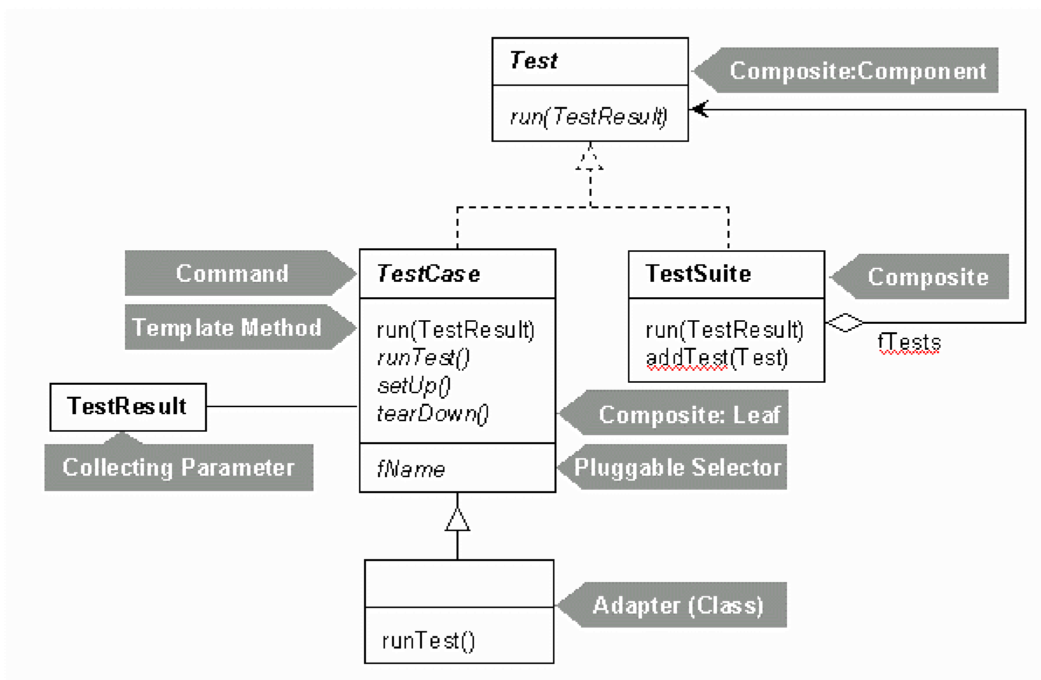

Abb. 5 [18]

[18] http://junit.sourceforge.net/doc/cookstour/cookstour.htm

Die Vorgehensweise beim Schreiben von Testklassen ist so, dass nach dem Festlegen einer Schnittstelle idealerweise zuerst die Testklassen geschrieben werden. Dabei entsteht pro Klasse eine Testklasse. In den Testmethoden wird gegen die noch leeren Methoden der Schnittstelle programmiert. Ein Test wird also zuerst nur Fehler liefern. Dann werden die Methoden so ausprogrammiert, dass die Tests „grün liefern“.

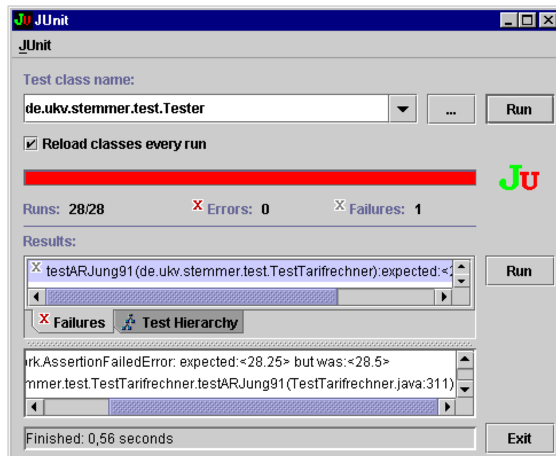

Abb. 6

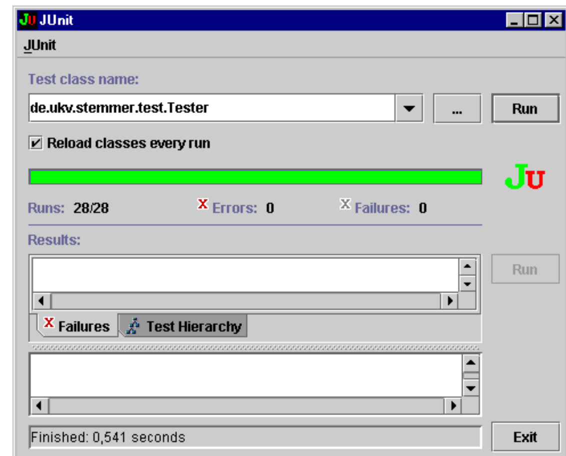

Abb. 7

Disziplin ist erforderlich, um wirklich viele Pfade in einem Test zu testen. Ein Test, der immer grün ist, weil er sämtliche Problemfälle vermiedet, ist sinnlos. Auch müssen die Tests mit dem Programm laufend weiterentwickelt werden. Sonderfälle, die zu einem späteren Zeitpunkt in einem Programm aufgenommen werden, brauchen auch eine neue Testmethode.

Da mit JUnit durch Asserts vor allen Ausgaben des Programmes ausgewertet werden, zählt JUnit zu den Black-Box-Testtools, auch wenn der Test selbst programmiert wird und dem Entwickler bekannt ist, an welchen Stellen er seinen Code durch Testfälle untersuchen sollte.

JProbe

Sitraka (heute Quest) JProbe ist ebenfalls eine Werkzeug, dass weit verbreitet ist und eine gutes Preis-Leistungs-Verhältnis hat.

Anmerkung: Vorgestellt wird hier nur die Version 3.0[19]. Aktutuell ist JProbe 5.0 mit einer ganzen Reihe neuer Features. JProbe besteht aus vier Komponenten, von denen JProbe Coverage vorgestellt wird.

Coverage läßt sich sehr einfach bedienen und liefert nach der Programmausführung eine Aufstellung über die Anweisungsüberdeckung auf Methodenebene.

---

[19] Um die gezeigten Beispiele mit JProbe 3.0 nachstellen zu können, muss excluded.properties in junit.jar um die Einträge com.klg.* und com.sitraka.* ergänzt werden.

Coverage Browser: snapshot_2

File Edit Tools Display Window Help

Show Only Methods With Coverage: <= 100%

Name	Calls	Hit Methods	Total Methods	Hit Lines	Total Lines
de.ukv.stemmer.test.Tester	2.137	45 (58,4%)	77	186 (68,9%)	270
de.ukv.stemmer.test	2.137	45 (58,4%)	77	186 (68,9%)	270
TarifrechnerException	16	1 (50,0%)	2	1 (50,0%)	2
TarifPropertyException	22	3 (75,0%)	4	11 (78,6%)	14
TarifCrosscheckException	14	3 (75,0%)	4	9 (75,0%)	12
TarifFactory	250	4 (40,0%)	10	29 (50,9%)	57
Tarif	1.081	9 (64,3%)	14	24 (82,8%)	29
AJ	98	5 (83,3%)	6	20 (90,9%)	22
AR	218	5 (83,3%)	6	43 (93,5%)	46
Person	382	7 (41,2%)	17	29 (56,9%)	51
Person$Sex	3	2 (100,0%)	2	4 (100,0%)	4
Property	5	2 (100,0%)	2	7 (100,0%)	7
Tarifrechner	48	4 (40,0%)	10	9 (34,6%)	26

Abb. 8

Die Auswertung ist mit dem Sourcecode der Anwendung verknüpft, so dass sofort visualisiert werden kann, welche Codezeilen nicht durchlaufen worden sind.

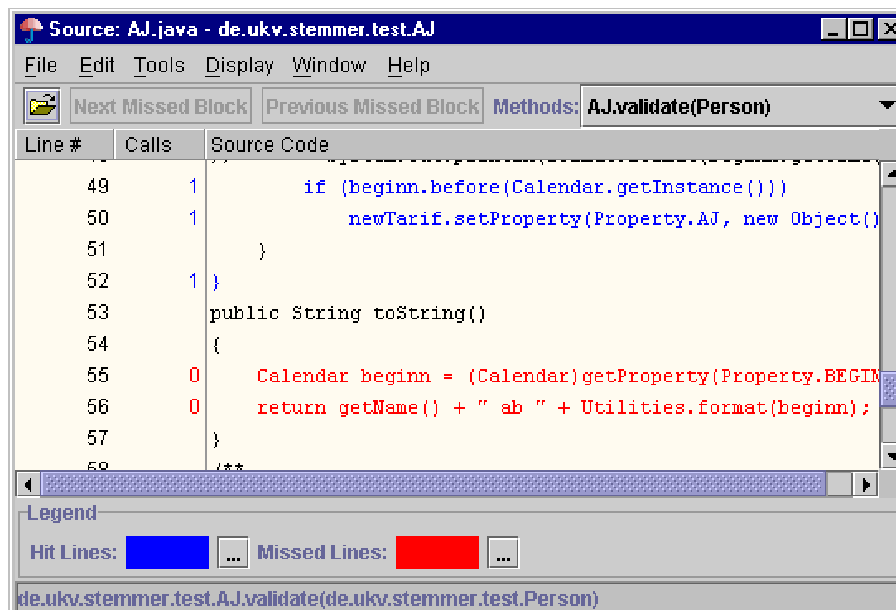

Abb. 9

Durch die Auswertung auf Sourcecodeebene wird JProbe zu einen klaren Vertreter der White-Box-Testtools. Ausgaben des Programmes spielen keine Rolle.

Zusammen mit einer Äquivalenzklassenbildung und JUnit erhält man schnell einen Aussage über Aussagekräftigkeit des Tests.

Purify und DevPartner

Neben JProbe sind IBM (früher Rational) Purify und Compuware (früher NuMega) DevPartner in der Lage Codeüberdeckung (Coverage) auszuwerten. Der Leistungsumfang ist recht ähnlich, so dass hier keine weitere Vorstellung stattfindet.

SQS

Ebenfalls nicht besprochen wird eines der größten Programm am Markt, dass sich mit der Ermittlung, Generierung und Verwaltung von Testdaten befaßt, SQS. Dazu wäre ein eigenes Seminar erforderlich. Der Leistungsumfang von SQS ist erheblich.

Abschließende Bewertung

Auffällig ist, dass keines der „großen“ Tools in der Lage ist, automatisierte white-box-Tests zu erstellen. Die Coverageanalye selbst ist ein C_0-Test und kann damit keinesfalls als hinreichender Test bewertet werden. Dazu wäre zumindest ein C_1-Test zu fordern.

Ansätze zur Erstellung und Auswertung von Kontrollflußgraphen hat zum Beispiel Panorama[20]. Diese reichen für einen produktiven Einsatz aber nicht aus.

Abb. 10 (Panorama)

Bei funktionsorientierten Tests von komplexeren Anwendungen wird man immer der Schwierigkeit gegenüberstehen, dass sinnvolle Äquivalenzklassen nicht manuell gebildet werden können. In der Anwendung sind trotz Klassenbildung zahllose Eingaben möglich. Diese multiplizieren sich schnell in Größenordnungen, die nicht mehr handhabbar sind. Lösungen zur Testplanung und Verwaltung von Testdaten wie SQS, können hier zwar helfen, das eigentliche Problem aber nicht lösen. Für

[20] http://www.softwareautomation.com/java

eine Qualitätsaussage müssen die Ausgabedaten überprüft werden. Wenn keine manuelle Überprüfung stattfinden soll (kann), müssen Ausgabenäquivalenzklassen gebildet werden. Eine Zuordnung zu Eingabeäquivalenzklassen muss wieder manuell geschehen.

Die Programmierung von Testcases wird mit Frameworks wie JUnit zwar einfacher und der psychologische Aspekt eines grünen Balken sollte auf keinen Fall unterschätzt werden. JUnit zielt aber vor allem auf den Komponententest und nicht auf die Integration in ein Gesamtsystem. Alleine die immer neue Herstellung eines Startzustandes würde den Rahmen des Möglichen wahrscheinlich sprengen.

Daraus kann man allerdings auch ableiten, dass dem Test der Komponenten und damit auch der Spezifikation die wichtigste Bedeutung zukommt. Auf dieser Ebene sind die Komplexität und damit verbunden Ein- und Ausgabedaten beherrschbar und die Spezifkation ist außerdem Voraussetzung für eine gute Integration.

Schwierig wird es auch, wenn eine GUI, eine Weboberfläche oder Engines (wie Servlet/SOAP oder EJB) ins Spiel kommen. Hier wird die Schnittstelle der Komponente von einem vorgelagerten System bedient, was die Herstellung eines Startzustandes erheblich komplizierter macht.

Oberflächentests mit Capture-Replay-Tools setzten (wie der Name schon sagt) eine manuelles Capture voraus. Dieses kann zwar dann parametrisiert werden. Der Pfad durch die Anwendung bleibt aber dabei immer gleich. Für einen umfassenden Test sind sehr viele Pfade zu betrachten, wodurch der Aufwand wieder unverhältnismäßig hoch wird.

Darüber hinaus wird (oder kann) nicht jede Infrastruktur die Voraussetzung für „gute“ Tests liefern. Ein gutes Capture-Replay-Tool wie WinRunner kostet 10.000€ und mehr. Die Einführung von SQS ist noch teurer. Hinzu kommen Auswände für Schulung und Einführung eines Werkzeugs.

Nicht immer wird die Zeit für einen Test in den Projektplan (richtig) eingerechnet, oder sie dient schnell als Puffer für Verzögerungen in der Codierung (aufgrund von Designfehlern etc.). Das allerdings ist eine Frage des Projektmanagements und damit verbunden des Entwicklungsprozesses und nicht Inhalt des Vortrages.