Harry Sneed, Mario Winter

Testen objektorientierter Software

Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme

Harry Sneed, Mario Winter

Testen objektorientierter Software

Das Praxishandbuch für den Test objektorientierter Client/Server-Systeme

HANSER

Die Autoren

Harry M. Sneed, Arget, ist freiberuflicher Programmierer, Projektleiter, Berater und Fachreferent. Er ist international bekannt auf den Gebieten Software-Reengineering, -Management und -Test.

Mario Winter, Wuppertal**,** lehrt und forscht in den Bereichen Software-Engineering und Qualitätssicherung an der FernUniversität Hagen und begleitet entsprechende industrielle Projekte. Dr. Winter ist Sprecher des Arbeitskreises „Test objektorientierter Programme“ der GI-FG 2.1.7.

Die Informationen in diesem Buch Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext Blindtext

Die Deutsche Bibliothek – CIP Einheitsaufnahme
Ein Titelsatz für diese Publikation ist bei der Deutschen Bibliothek erhältlich

Internet: http:\\www.hanser.de
Gesamtlektorat:
Copy-editing:
Herstellung:
Umschlaggestaltung:
Druck und Bindung:
Printed in
ISBN:

Inhalt

Vorwort

In der Welt der Informatik wird dem Thema Testen viel Aufmerksamkeit gewidmet. Führende Fachzeitschriften wie die IEEE Transactions on Software Engineering, das Magazin IEEE Software, die Communications of the ACM und auch das Hausblatt der GI – Informatik Spektrum – sind voll mit Beiträgen über Testen. In der Tat ist kein anderes Teilgebiet der Informatik so faszinierend und so schwierig wie das Thema Test. In der ACM, der IEEE und der GI existieren schon seit langem Fachgruppen von Wissenschaftlern, die sich mit der Problematik des Testens auseinandersetzen. An theoretischen Lösungsansätzen fehlt es nicht. Dennoch, nirgendwo in der Informatik klaffen Theorie und Praxis weiter auseinander. Der Stand des Testens in der Praxis ist erbärmlich, besonders in Deutschland. Gerade in einem Lande, das sich für seine Spitzenqualität rühmt, wird in puncto Software-Test nur wenig getan. Man versucht, das Problem zu verdrängen, aber es taucht immer wieder auf. Schlimmer noch, je komplexer die Technologien, desto größer die Testproblematik.

Client/Server-Systeme, relationale Datenbanken, Intranet-Verbindungen und nicht zuletzt die Objekttechnologie haben dazu beigetragen, die Testproblematik zu verschärfen. Allen Beteuerungen der Methodenadvokaten zum Trotz steigen die Testkosten relativ zu den gesamten Projektkosten. Es gibt deshalb keinen Ausweg. Man muss sich diesem Problem stellen und eine Lösung suchen. Zum Glück gibt es in letzter Zeit auch erfreuliche Entwicklungen, wie das zunehmende Interesse an Testkonferenzen wie euroSTAR und Quality Week Europe und die Entstehung neuer Testberufe.

Die Lösung, die in diesem Buch beschrieben ist, dürfte vielen Lesern als hochgestochen erscheinen. Der Versuch liegt nahe, sie als „zu theoretisch“ abzuqualifizieren. Dies ist die Standardausrede aller Praktiker, wenn sie keine Lust haben, etwas Neues zu probieren. Als Autoren fühlen wir uns jedoch dazu verpflichtet, ein modellhaftes Rezept zu verschreiben. Was wäre das für eine Ethik, mit solchen Geboten anzukommen wie

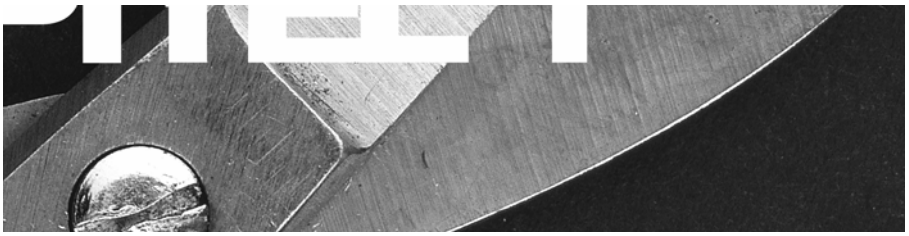
- klaue dem Nachbarn sein Geld,
- nimm dem Freund seine Frau,
- missachte des Vaters Ratschläge und
- schlage dem Bruder den Schädel ein,

bloß weil es in der Praxis wirklich so zugeht. Der Handlungsimperativ darf nicht der Praxis angepasst werden. Im Gegenteil, wir müssen alle versuchen, die Praxis dem Soll anzupassen, denn nur so bekommen wir eine bessere Welt. Diese Maxime gilt ebenso für die Welt des Software-Testens. Jeder ist verpflichtet, seinen Teil dazu beizutragen, den Stand der Praxis zu verbessern, auch wenn es manchmal mühselig erscheint.

Unser Beitrag als Autoren ist es, ein zwar praktikables, aber dennoch anspruchsvolles Verfahren zu schildern und anhand eines Beispiels zu demonstrieren. Jeder kann damit sehen, wie es sein sollte. Wie viel man in die eigene Praxis übernimmt, ist jedem Einzelnen überlassen – so wie mit den zehn Geboten.

1

Objektorientiertes Testen – Einführung



Das Testdilemma – warum wir testen müssen
Konventionelle Testansätze
Client/Server-Testproblematik
Besonderheiten objektorientierter Systeme
Objektorientierter Test – eine Herausforderung

Inhaltsübersicht Kapitel 1

1 Objektorientiertes Testen – Einführung

1.1 Das Testdilemma – warum wir testen müssen

Irren ist menschlich. Solange der Mensch Software schreibt, wird die Software mit Fehlern behaftet sein, unabhängig davon, in welcher Sprache bzw. auf welcher semantischen Ebene er seine Gedanken zum Ausdruck bringt. Denn schon bei der Übertragung dieser Gedanken entstehen Fehler. Eine falsche Taste drücken, eine 1 für I schreiben, und die Wiedergabe der Gedanken ist fehlerhaft. Somit entstehen schon während der ersten Kommunikation zwischen dem gedachten und dem geschriebenen Wort die ersten Kommunikationsfehler. Und hier beginnt bereits der erste Test: ob das, was man geschrieben oder getippt hat, auch das ist, was man gedacht hat. Autoren jeder Art sind mit dieser Problematik wohlvertraut.

Die Quelle mancher Fehler liegt jedoch noch vor dieser ersten Übertragung. Schon bei der Formulierung des Gedankens entstehen zwischen dem menschlichen Gedächtnis bzw. der Intelligenz und der natürlichen Sprache die ersten Übertragungsschwierigkeiten [Che66].

Diejenigen, die mehrere natürliche Sprachen beherrschen, wissen wohl, wie schwierig es oft ist, einen aus der einen Sprachwelt stammenden Gedanken in die andere Sprachwelt zu übertragen. Jeder Mensch hat sogar bewusst oder unbewusst Probleme, Gedanken in seiner Muttersprache auszudrücken. Ein komplexer Informationsinhalt wird selbst von der eigenen Sprache verfälscht.

Es ist deshalb töricht zu meinen, man könne das Testen durch eine höhere Sprache, eine bessere Methodik oder sogar durch Telepathie ersetzen. Solange das menschliche Gehirn die Quelle unserer Software ist, wird diese Software fehlerhaft sein. Dies trifft übrigens auch für die vielgerühmten Expertensysteme zu, die nicht zuverlässiger sind als die Experten selbst.

Wäre Software nur ein unverbindliches Kommunikationsmittel zwischen Menschen, könnte man sich mit ihrer Unvollkommenheit abfinden. Aber Software wächst in ihrer Bedeutung, sie steuert unsere Fertigungsprozesse, bezahlt unsere Gehälter und liefert uns Informationen für unsere Entscheidungen. Banale Fehler in der Software können verheerende Konsequenzen haben. Die SIGSOFT der ACM

macht in ihren Software Engineering Notes regelmäßig auf die Folgen solcher Softwarefehler aufmerksam [acm99]. In einigen Fällen sind Menschen zu Schaden gekommen, in anderen sind größere Unkosten entstanden. In manchen Fällen stand sogar der Weltfrieden auf dem Spiel.

Durch Testen hofft man, Softwarefehler zu finden, ehe sie zu solchen Folgen führen wie den obigen. Der Tester ist wie der Knecht, der hungrige Pferde mit frischem Heu füttern soll, in dem er jede Menge Stecknadeln vermutet (Abbildung 1.1). Natürlich könnte der Knecht, um Zeit zu sparen, einen Tierarzt holen, der mit einer Zange daneben steht, falls das arme Tier eine Nadel in den Hals bekommt. Aber die Anwesenheit des Tierarztes kostet Geld, und es ist nicht sicher, ob er das Tier in einem Notfall wirklich retten kann. Falls das Pferd nicht viel wert ist, könnte der Knecht es auch so fressen lassen – eben auf gut Glück. Wenn es stirbt, könnte der Bauer aber doch böse werden. Wenn der Knecht wirklich auf sein eigenes und auf das Wohl des Tieres bedacht ist, wird er sich bemühen, die Stecknadeln zu entfernen. Ein fauler Knecht würde wahrscheinlich nur im Heu herumwühlen, um sein Gewissen zu beruhigen. Nachher könnte er dem Bauern erzählen, er habe sein Bestes getan. Wenn er aber ernsthaft das Pferd schützen will, wird er das Heu sieben. Mit viel Zeit kann er mehrfach und fein sieben. Dennoch kann er nie sicher sein, dass er alle Stecknadeln gefunden hat. Auch wenn er 99 von 100 entfernt, könnte das Pferd just an der einen ersticken, die er nicht ausgesiebt hat. Und Sieben kostet Zeit. Warum denn dieser Aufwand? Und wenn schon Sieben: wie oft und wie fein?

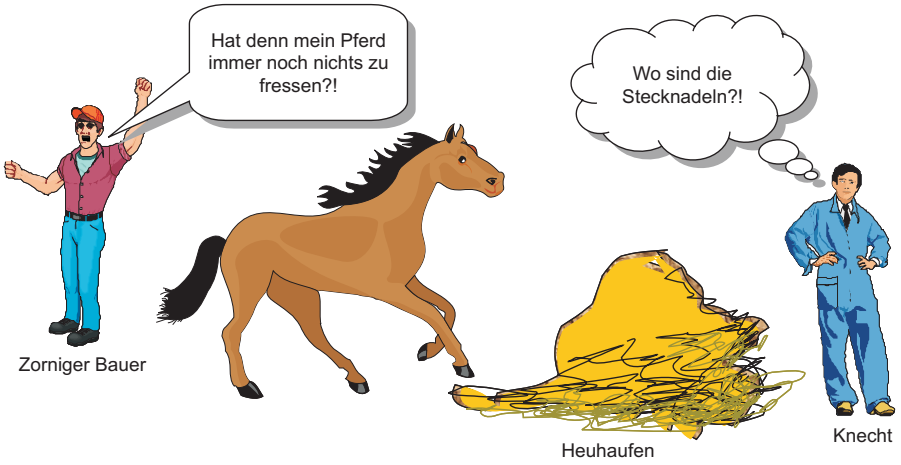


Abbildung 1.1 Die Stecknadel im Heuhaufen

Genau dies ist das Dilemma des Software-Testers. Er steht vor einem fast unlösbaren Problem. Und wie im Falle des Knechts gibt es auch bei seiner Arbeit wirtschaftliche Einschränkungen. Wenn nämlich das Sieben zu lange dauert, könnte das Pferd vor Hunger davonlaufen oder, schlimmer noch, sterben. Das Testen ist also

mit schwer abschätzbaren Risiken behaftet, die den Tester seinen Job kosten könnten. Verständlicherweise bleibt das Testen das unliebsamste Teilgebiet der Informatik, sogar noch unliebsamer als die Wartung.

1.2 Konventionelle Testansätze

Die Geschichte der Testtechnologie geht zurück auf eine Konferenz an der Universität North Carolina im Jahre 1972. Vorher hat es einzelne Beiträge zur Testproblematik gegeben, aber dies war das erste Mal, dass der Test im Mittelpunkt der wissenschaftlichen Betrachtung stand. Nach dieser besagten Konferenz, deren Inhalt in einem Buch von W. Hetzel im Jahre 1973 unter dem Titel *Program Test Methods* veröffentlicht wurde, erlebte die Testtechnologie einen regelrechten Boom [Het73]. 1975 erschien der grundlegende Beitrag von Goodenough und Gerhardt zur Theorie der Testdatenauswahl in der Communications of the ACM [GoGe75]. Zwei Jahre später wurde das Testsystem RXVP für das amerikanische Ballistic Missile Defense System fertiggestellt. Diese „Mutter aller Testsysteme“ enthielt bereits die Instrumentierung von FORTRAN- und PASCAL-Programmen, eine dynamische Programmanalyse, einen Testtreiber und die Zusicherungstechnik mit Vor- und Nachbedingungen. Später wurde das System in SQLAB umgetauft. Aber es blieb das Vorbild für alle später entwickelten Testwerkzeuge, einschließlich des Siemens-Prüfstand-Systems von einem der Autoren, das im Jahre 1978 im Budapester Testlabor zum Einsatz kam. Das besagte Testlabor in Ungarn war der erste Versuch, Test als kommerzielle Dienstleistung anzubieten. Die Ergebnisse wurden auf dem Florida Testworkshop im Jahre 1979 vorgestellt [BMS79]. Es genügt zu sagen, dass hier die angewandte Testtechnologie einen vorläufigen Höhepunkt erreichte, gestützt auf die Forschungsarbeit von Dr. Ed Miller in Amerika [Mil77] und Prof. Mike Hennel in England [WHH80]. Eine der besten Zusammenfassungen der Testtechnologie der 70er Jahre ist der *Infotech State of the Art Report on Software Testing* aus dem Jahre 1979, in dem über die Testsysteme SQLAB und PRÜFSTAND ausführlich berichtet wurde [Mil79].

Nach 1980 haben sich datenbezogene und funktionsbezogene Testansätze immer mehr durchgesetzt, vor allem dank der Arbeit von Howden [How80], Clarke [CPR+89] und Ntafos [Nta84]. Später hat sich eine weitere Variante des Testens unter der Bezeichnung *diversified Testing* oder *Back-to-Back-Testing* hervorgehoben [Zei86]. Voges und Gmeiner haben diesen Ansatz am Kernforschungszentrum Karlsruhe erprobt und darüber berichtet [GmVo80]. Auch die Gesellschaft für Reaktorsicherheit in Garching hat sich damit beschäftigt [SaEh86]. Dieser Ansatz ist mit der *Mutation Testing*-Methode von DeMillo und Budd verwandt, wonach verschiedene Mutationen eines Programms gegen dieselben Daten getestet werden. Die Mutationen werden durch geringfügige Änderungen im Code verursacht, und der Test soll beweisen, dass nur die geänderten Funktionen veränderte Ergebnisse er-

zeugen [BDL+78]. Bei dem *diversified testing*-Ansatz werden hingegen völlig verschiedene Versionen desselben Programms gegeneinander getestet [EcLe85].
Oft werden Programme mit Kästen (*boxes)* verglichen. So spricht man von der White-Box-Methode, der Black-Box-Methode und der Grey-Box-Methode. Diese Bezeichnungen sind jedoch sehr oberflächlich und lassen sich unterschiedlich auslegen. Genauer ist es, vom *ablaufbezogenen*, *datenbezogenen und funktionsbezogenen* Testen zu sprechen. Ablaufbezogen ist ein Test, wenn der prozedurale Ablauf die Basis für die Ermittlung der Testfälle bildet. Datenbezogen ist ein Test, wenn die Datenbeschreibungen diese Basis bilden. Funktionsbezogen ist der Test schließlich, wenn die funktionale Beschreibung bzw. die Spezifikation als Basis der Testfallermittlung dient [Sne83].
Um das Thema einigermaßen einzugrenzen, wird hier nur noch auf die folgenden Hauptansätze eingegangen:

- statische Programmanalyse,
- ablaufbezogenes Testen,
- datenbezogenes Testen,
- funktionsbezogenes Testen und
- Back-to-Back-Testen.

1.2.1 Statische Programmanalyse

Zu den Methoden der statischen Analyse gehören jene Techniken, die den Quellentext an und für sich analysieren und interpretieren (Abbildung 1.2). Der Text kann als formale Spezifikation, in einer Entwurfssprache oder in einer Programmiersprache abgefasst sein. Aufgrund einer syntaktischen Analyse wird der Quelltext in Elemente zerlegt und die Struktur und Verwendung der Elemente untersucht. Hat man den Text in seine syntaktischen Elemente zerlegt, so bestehen mehrere Möglichkeiten, die Semantik des Textes zu prüfen: innerhalb des Textes selbst, gegen von außen vorgegebene Konventionen oder gegen einen zweiten Text. Außerdem kann der Text interpretiert bzw. symbolisch ausgeführt werden.
Die erste Möglichkeit einer statischen Analyse ist, den Quelltext auf Vollständigkeit und Konsistenz zu prüfen. Bei Verwendung von Spezifikations- oder Entwurfssprachen kann diese Art der Prüfung besonders fruchtbar sein. So dürfen logische Regeln sich nicht widersprechen, es dürfen nur jene Daten benutzt werden, die schon bereitgestellt wurden, und für jede Programm/Objekt-Beziehung müsste es eine entsprechende Objekt/Programm-Beziehung geben. Dazu lässt sich prüfen, ob alle erforderlichen Beziehungen und Attribute der spezifizierten Elemente vorhanden sind. In einem Programmtext lassen sich interne Eigenschaften prüfen, z.B. ob Variablen gesetzt werden, bevor sie benutzt werden, ob Schleifen immer terminieren und ob die Datentypdeklaration nach Typ und Format mit der Datenverwen-

dung übereinstimmt. Nach der Untersuchung von Howden könnten 7 von 25 Fehlern auf diese Weise entdeckt werden [How78]. In einer anderen Studie von Marilyn Fujii wird über ein Drittel der Fehler in FORTRAN-Programmen von dem statischen Analysator RXVP aufgedeckt – vor allem inkonsistente Datendeklarationen, Parameterfehler, nicht erreichbarer Code und Datentypverletzungen [Fuj77].

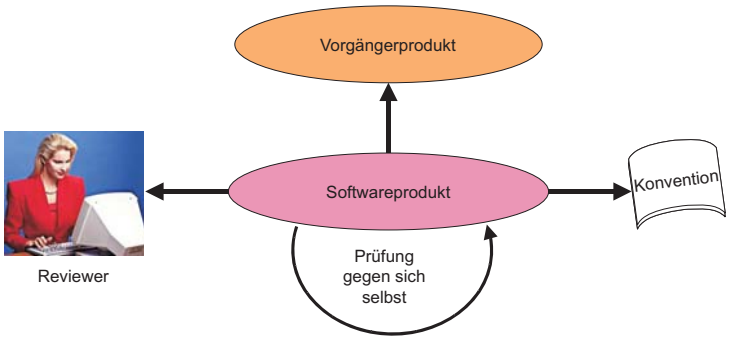


Abbildung 1.2 Statische Programmanalyse

Die zweite Möglichkeit einer statischen Analyse besteht darin, den Quelltext gegen eine Reihe vorgegebener Regeln zu testen. Diese Art der Prüfung, auch *Code-Auditing* genannt, kann manuell oder maschinell durchgeführt werden. Michael Fagan setzte diese Technik unter der Bezeichnung *Design and Code Inspection* innerhalb der IBM durch. Sein Artikel zu diesem Thema bildet einen Meilenstein auf dem Wege zu einer bewussten Software-Qualitätssicherung und wird vielerorts zitiert [Fag76]. Fagan sah eine Checkliste vor, mit deren Hilfe Entwurfs- und Programmtexte manuell geprüft wurden. Abweichungen von den Regeln wurden angezeigt und mit den Verfassern der Texte diskutiert. Nicht selten offenbarten sich zahlreiche weitere Fehler in der Diskussion selbst. Andere versuchten, die Prüftechniken zu automatisieren, indem sie die Regeln in Tabellen abspeicherten und die Programmanweisungen maschinell gegen sie prüften. Daraus folgten die ersten Code-Auditoren-Werkzeuge, welche die Einhaltung der Programmierregeln automatisch kontrollieren. Geprüft werden z.B. die Modulgröße, die Verschachtelungstiefe der Ablauflogik, die Kommentierung, die Verwendung verbotener Konstrukte wie `ALTER` oder `GO TO` und die unerwünschte Nutzung globaler Daten. Nach einer Untersuchung von Deutsch werden 18% der Programmfehler durch strengere Code-Konventionen unterbunden [Deu82].

Die dritte Möglichkeit einer statischen Analyse besteht darin, den Quelltext gegen einen anderen Text über den gleichen Sachverhalt zu testen, z.B. den Programmtext gegen die Spezifikation oder einen Programmtext gegen einen anderen. Diese statische Verifikation soll nachweisen, dass ein Text, der aus einem anderen Text abgeleitet wurde, immer noch mit dem Originaltext inhaltlich übereinstimmt. So dürfte eine lauffähige Prozedur in Pascal oder FORTRAN zwar zusätzliche programm-

technische Operationen beinhalten, aber keine problemspezifischen Operationen, die vorher nicht spezifiziert wurden. Die Bedingungen, Berechnungen und Datenbeschreibungen im Programm müssen sich mit denen in der Spezifikation decken. Darüber hinaus darf es keine funktionalen Festlegungen in der Spezifikation geben, die nicht durch das Programm abgedeckt sind. Bei dieser Art der statischen Analyse kommt es also darauf an, Übersetzungsfehler oder sonstige Kommunikationsstörungen bei der Übertragung von einem Text in einen anderen zu enthüllen. Durch den Abgleich zweier syntaktisch und semantisch unterschiedlicher Beschreibungen des gleichen Sachverhalts auf unterschiedlichen Abstraktionsebenen kommen auch Mängel der Problemlösung zu Vorschein, deren Durchleuchtung oft fehlerhafte Konstrukte enthüllt. Dieser statische Testansatz kommt dem formalen Programmbeweisen am nächsten, liefert aber keinen lückenlosen Beweis für die Korrektheit des Quelltextes. Voraussetzung für diesen Ansatz ist jedoch eine zweite formale Beschreibung, deren Genauigkeitsgrad mit dem des Quelltextes als Prüfobjekt vergleichbar ist. Als Miller sein IEEE Test-Tutorial [Mil81] verfasste, steckte dieser Ansatz wegen des Mangels an brauchbaren Spezifikationssprachen noch in den Kinderschuhen. Seitdem sind signifikante Fortschritte auf dem Gebiet der Spezifikationssprachen gemacht worden, sodass Testen und Spezifizieren immer mehr zusammenwachsen. Ein von Hausen herausgegebener Bericht über ein GMD-Symposium zum Thema *Software Validation* behandelt dieses Zusammenspiel von Inspektion, Testen, Verifikation und Spezifikation [HaMü83].

1.2.2 Ablaufbezogenes Testen

Ablaufbezogenes Testen (*Structured Testing*) ist die Realisierung der White-Box-Testmethode; das Programm wird gegen sich selbst getestet. Das Objekt des Tests ist der Ablaufgraph des jeweiligen Programms oder Moduls. Das Ziel ist es, die Überdeckung der möglichen bzw. relevanten Pfade eines Programms (z.B. durch einen gerichteten Graphen) in einem Test oder einer Testserie festzustellen.

Propagiert wurde diese Methode durch Edward Miller, der das erste Werkzeug – RXVP – zur Messung der Ablaufzweigdeckung entwickelte [Mil78], und Tom McCabe, der die McCabe-Metrik der Programm-Komplexität prägte [McC83].

Miller hat 8 Überdeckungsmaßstäbe vorgeschlagen ([Mil81], vgl. Tabelle 4-1):

C_0 = Ausführung aller Anweisungen
C_1 = Ausführung aller Ablaufzweige
C_2 = Erfüllung aller Bedingungen
C_3 = Wiederholung aller Schleifen
C_4 = Wiederholung aller unabhängigen Pfade
C_5 = Ausführung aller unabhängigen Pfade
C_6 = Ausführung aller Vorwärtspfade

C_7 = Ausführung sämtlicher Pfade

Die Messung von C_2, C_3 und C_4 hat sich in der Praxis als zu aufwändig erwiesen. C_2 lässt sich nur im Objektcode instrumentieren. C_3 und C_4 sind durch Trace-Werkzeuge zu ermitteln. C_7 hat sich als unerreichbar erwiesen. Geblieben sind die Maßstäbe C_0, C_1, C_5, C_6. Beizer hat die Ablaufüberdeckungsmaßstäbe auf die drei Überdeckungsgrade weiter reduziert [Bei83]:

- Anweisungsüberdeckung,
- Zweigüberdeckung und
- Vorwärtspfadüberdeckung.

Die Anweisungen, Zweige und Pfade lassen sich durch eine statische Analyse aus dem Programm ableiten. Bei der dynamischen Analyse wird anschließend die Anzahl Ausführungen registriert und protokolliert. Abbildung 1.3 illustriert den Unterschied zwischen diesen drei Überdeckungsgraden.

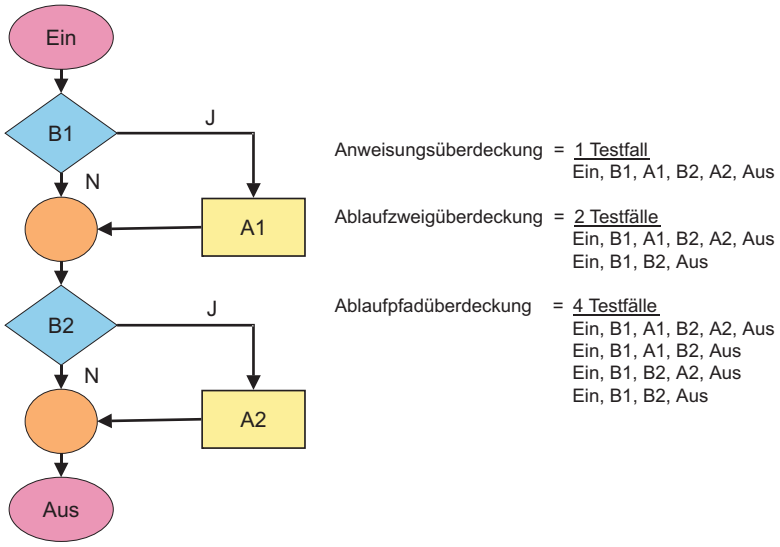


Abbildung 1.3 Ablaufbezogener Test

Um alle Anweisungen zu erreichen, ist lediglich ein Testfall erforderlich. Um alle Zweige zu erreichen, werden zwei Testfälle benötigt. Um alle Pfade auszuführen, werden allerdings vier Testfälle gebraucht. Anhand dieses Beispiels ist zu erkennen, dass die Anzahl Testfälle in einem bestimmten Verhältnis zum Überdeckungsgrad wächst.

Anfang der 80er Jahre hat das strukturierte Testen mit seinen mechanistischen Regeln großes Interesse gefunden. Vor allem die graphentheoretischen Grundsätze und die Objektivität der Methoden hat viele Forscher fasziniert. Hinzu kam die

relative Leichtigkeit, mit der man die Ergebnisse messen konnte. An der Effektivität dieser Methode gab es jedoch von Anfang an Zweifel, vor allem von Howden und Clarke. In einem Experiment von Howden wurden nur 12 von 25 Fehlern durch die Pfadüberdeckung aufgedeckt [How76]. In der Praxis des oben genannten Testlabors hier in Europa wurde diese Erfahrung bestätigt. Trotz hoher ablaufbezogener Überdeckung wurden weniger als 50% der Fehler angezeigt [Sne83].

Die Erklärung hierfür liegt auf der Hand. Nur eine bestimmte Klasse von Fehlern lässt sich durch den Strukturtest entlarven – nämlich grobe Abbruchsfehler, unerreichbare Zweige, Irrpfade, endlose Schleifen und unvollständige bzw. inkonsistente Bedingungen. Vergessene Funktionen, unberücksichtigte Daten, inkonsistente Schnittstellen, Tippfehler, IO-Fehler und Abweichungen von der Spezifikation können auch nach der Ausführung aller Pfade, d.h. der vollständigen Überdeckung des Ablaufgraphs, unerkannt bleiben. Denn der strukturierte Test prüft nur das, was das Programm tut und wie es dies tut, aber nicht, was es tun sollte und wie es dies tun sollte. Die eigentlichen Daten und Funktionen werden außer Acht gelassen. Deshalb ist der strukturierte Test für sich allein kein zuverlässiger Test.

Sein Hauptnutzen liegt darin, dass er den Tester zwingt, sich intensiv mit dem Testobjekt auseinander zu setzen. In dem Bestreben, Testfälle zu definieren, die alle Anweisungen, Zweige oder Pfade ausführen, stößt er unwillkürlich auf logische Ungereimtheiten und Lücken im Code. Martha Branstadt vom National Bureau of Standards hat diesen Aufsatz treffend als *nose rubbing technique* bezeichnet [ABC82]. Ein Garant für Korrektheit ist er wie alle Testansätze nicht.

1.2.3 Datenbezogenes Testen

Datenbezogenes Testen hat seine Wurzel im Random-Datentest. Danach werden wahllose Eingabe-Datenkombinationen generiert und in großer Menge dem zu testenden Programm zugeführt. Die generierten Testdaten sind reine Zufallswerte. Das Testen mit Random-Daten erzeugt aber auch nur Random-Ergebnisse. Die Auslösung der Funktionen im Programm geschieht nur zufällig und hängt von der statistischen Streuung der Daten ab.

In seinem Buch „The Art of Software-Testing“ hat G. Myers diesen Ansatz etwas verfeinert [Mye79]. Für die Auswahl der Testdaten teilt er die Eingaben in Mengen diskreter Werte, zusammenhängende Wertbereiche und Daten mit expliziten Beziehungen zu anderen Daten. Die Untersuchungen der Beziehungen zwischen Eingaben und Ausgaben einer Funktion nennt Myers Ursache-Wirkungsanalyse (*Cause and Effect Analysis)*. Die Mengen diskreter Werte können darüber hinaus in Äquivalenzklassen aufgeteilt werden, z.B. alle gültigen Schlüssel oder alle negativen Zahlen. Alle Werte in einer Äquivalenzklasse erzeugen die gleiche Wirkung im Programm. White und Cohen haben sie als Eingabebereich (*Input Domain)* gekennzeichnet [WhCo80]. Für jede Äquivalenzklasse bzw. jeden Eingabebereich genügt

es, einige repräsentative Werte zu testen. So wäre der Wert -1 stellvertretend für alle negativen ganzen Zahlen. Das Testen mit repräsentativen Werten einer Äquivalenzklasse nennt Myers *repräsentative Wertanalyse*.

Zusammenhängende Wertebereiche oder *Ranges* werden durch die Grenzwerterzeugung getestet. Nach Myers genügt es, den unteren Grenzwert sowie den nächstkleineren Wert, den oberen Grenzwert sowie den nächstgrößeren Wert und einen Mittelwert zu generieren, um einen geschlossenen Wertebereich zu testen. Diese Werte sind stellvertretend für alle weiteren innerhalb und außerhalb des jeweiligen Wertebereichs. Das Testen mit Grenzwerten nennt Myers *Grenzwertanalyse*.

Datenfelder mit definierten Relationen zu anderen Datenfeldern erhalten Relationswerte. So kann die Vereinigung zweier Eingabedaten die Erzeugung eines Ausgabewerts verursachen, oder der Wert einer Variablen kann immer kleiner, gleich oder größer als der einer zweiten Variablen sein. Falls diese Beziehung immer besteht, nennt man sie invariant. Beizer behandelt diese Thematik auch unter der Bezeichnung *State transition testing* [Bei84]. Die Festlegung der Beziehungen zwischen Daten unabhängig von der Funktion gehört zum Bereich der Datenanalyse und des Relationenmodells. Zur Normierung der Datenfelder kommt hier die Bestimmung der Feldinhalte und deren Beziehung zueinander hinzu (Abbildung 1.4). Die Festlegung der Datenbeziehungen vollzieht sich wie die Regeldefinition in der Prädikatenlogik.

Die von Myers geschilderte Betrachtung der Daten ist eine statische. Sie sieht Daten als invariante Mengen mit determinierten Beziehungen zu den zu testenden Funktionen. Im Gegensatz dazu beschäftigt sich die Datenflussanalyse mit der Veränderung der Daten während der Programmausführung. Hiernach wird die Programmausführung in Zeitintervalle zerlegt und der Zustand der Daten nach jedem Intervall untersucht bzw. neu gesetzt. Die Datenflussanalyse befasst sich vor allem mit dem temporären Zustand der Daten in Schnittstellen zwischen Modulen und externen Datenträgern. Manche Daten existieren nur vorübergehend in dieser Form. Der Zweck der Datenflussanalyse ist es, diesen temporären Zustand festzuhalten und möglicherweise zu verändern. Dazu sind spezielle Mechanismen zur Ablaufunterbrechung erforderlich. Bei Echtzeit-Programmen ist dies besonders schwierig [LaKo83].

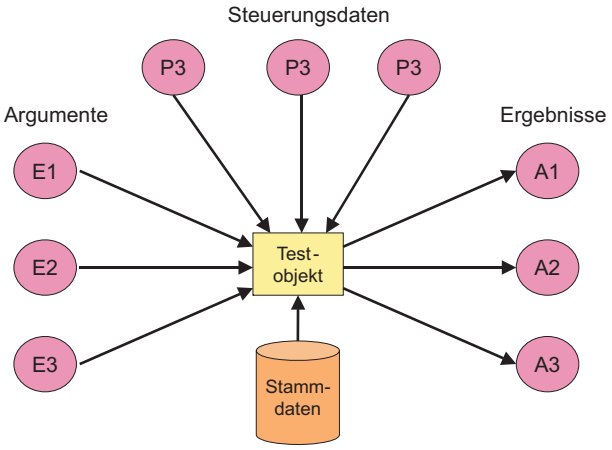


Abbildung 1.4 Datenbezogener Test

Der letzte Stand des datenbezogenen Testens ist, von einem Data Dictionary aus die Testdaten zu erzeugen und die Testergebnisse gegen die Sollergebnisse zu kontrollieren. Zur Beschreibung der Daten gehört außer dem Typ, der Länge, der Bedeutung usw. auch der Wertebereich des Datenelements bzw. seine Relation zu anderen Datenwerten. Aus diesen Wertbestimmungen heraus können Dateien und Datenbanken generiert und validiert werden [Sne86].

1.2.4 Funktionsbezogenes Testen

Funktionsbezogenes Testen oder *functional testing* wurde schon 1982 von Howden als Alternative zu dem bis dahin herrschenden Ansatz des strukturierten (ablaufbezogenen) Testens vorgeschlagen. Schon vorher hatte Howden davor gewarnt, das Programm ausschließlich gegen sich selbst zu testen. Dieser Ansatz sei unzulänglich. Man müsse stattdessen die Funktion des Programms testen [How81].
In einem späteren Artikel in den IEEE Transactions on Software Engineering vom Oktober 1986 schreibt Howden:

> *„Funktionstesten und Funktionsanalyse ... hängt von einem Orakel ab. Die Grundarten von Orakeln sind Eingabe/Ausgabe-Orakel, Ablauf-Orakel und Schnittstellen-Orakel...“ [How81]*

Für jede Funktion müssen nicht nur die Eingabe- und Ausgabedaten, sondern auch die Eingabe- und Ausgabewerte spezifiziert werden. Hiermit greift Howden auf die Ursache/Wirkung-Analyse von Myers zurück. Wichtig ist die eindeutige Zuordnung der Funktionen zu den Code-Abschnitten und das gezielte Testen der Abschnitte. In Howdens Worten:

> *„... Die Code-Abschnitte, die den einzelnen Funktionen entsprechen, müssen identifiziert, und ihr Eingabe-/Ausgabe-Verhalten muss über mehrere ausgewählte Testfälle beobachtet werden.“ [How81]*

Das Ablauf-Orakel sagt, wie das Programm ablaufen soll. Es steckt aber mehr dahinter als beim ablaufbezogenen Test. Howden schreibt:

> *„Funktionstesten ist anspruchsvoller als die bisherigen Überdeckungsmaßstäbe wie Zweig- und interne Pfaddeckung, da es verlangt, dass alle Kombinationen von Programmanweisungen, die eine Programmfunktion befriedigen, getestet werden müssen.“ [How81]*

Andererseits werden nur jene Pfade getestet, die einer spezifischen Funktion entsprechen, d.h. nicht spezifizierte Zweige und Pfade werden außer Acht gelassen.

Das Schnittstellen-Orakel bestimmt schließlich, welche Daten zwischen welchen Modulen ausgetauscht werden. Es geht also hier um eine exakte Spezifikation der Datenflüsse. Zu jeder Schnittstelle gehören bestimmte Variablen, die nur in bestimmten Konstellationen auftreten. Sie müssen sowohl statisch als auch dynamisch analysiert werden.

Howden definiert eine Schnittstelle als eine Reihe von Variablen `a`, `b`, `c`, ... bzw. als eine Parameterliste, in der bei n Testfällen jede Variable n (nicht notwendigerweise verschiedene) Werte mit dem Index $1 \leq i \leq n$ haben kann. Für die (Eingabe-) Variable `a` bezeichnet das Tupel (a_1, a_2,..., a_n) die Reihe von Werten, die einer entsprechenden Reihe von Ergebnissen (b_1, b_2 ... b_n) für die (Ausgabe-)Variable `b` gegenübersteht. Jede Argumentreihe bzw. Dateneingabe stellt einen Vorzustand dar, jede Ergebnisreihe bzw. Datenausgabe einen Nachzustand (Abbildung 1.5). Es gilt, Schnittstellen von beiden Seiten zu betrachten, sowohl aus der Sicht des Senders als auch aus der Sicht des Empfängers. Wenn beide Sichten übereinstimmen, ist die Schnittstelle korrekt.

Verifikation ist der Beweis, dass ein Programm im Sinne der Spezifikation korrekt ist. Validierung dagegen ist der Beweis, dass ein Programm in einer bestimmten Zielumgebung lauffähig ist und insbesondere das tut, was der Benutzer braucht. Ein verifiziertes Programm muss danach nicht unbedingt lauffähig sein, und umgekehrt muss ein validiertes Programm nicht unbedingt korrekt im Sinne der Spezifikation sein.

Allein der funktionale Ansatz nähert sich den Anforderungen der Programmverifikation, wonach das Verhalten des Programms in einer separaten deskriptiven Sprache exakt spezifiziert und das Programm gegen die Spezifikation dynamisch geprüft wird. Das Programm muss im Sinne des lateinischen *Veritas* wahr sein. Die Demonstration der Lauffähigkeit verlangt andere Testansätze, z.B. Structured Testing oder Stress Testing [BBF+82].

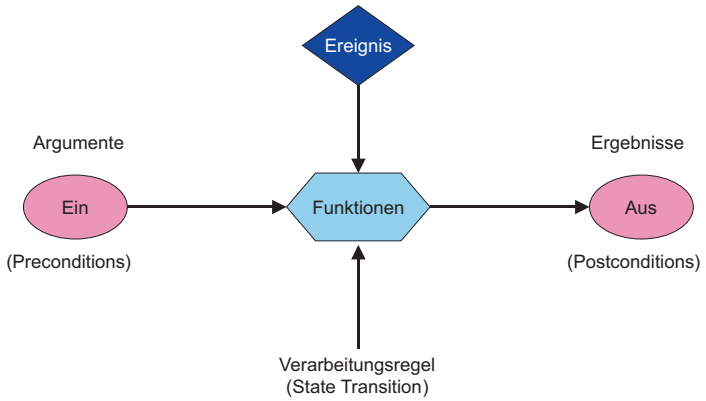


Abbildung 1.5 Funktionsbezogener Test

1.2.5 Regressionstesten

Der Begriff Regressionstest wurde zum ersten Mal in einem Beitrag von Scherr auf der ersten Software-Test-Konferenz an der Universität North Carolina im Juni 1972 geprägt [Sch73]. In dem IEEE „Standard Glossary of Software Engineering Terminology“ wird der Begriff wie folgt erklärt:

> *„Regression testing: selective testing to verify that modifications have not caused unintended adverse side effects or to verify that a modified system still meets it´s requirements” [IEEE610].*

Die Eigenschaft *modifiziert* könnte hier auch durch die Eigenschaften *saniert* und *konvertiert* erweitert werden.

Seit Anfang der 70er Jahre wurde viel geforscht, um vor allem den Aufwand für den Regressionstest zu reduzieren. Wurde ein Programm geändert, galt es ursprünglich als notwendig, alle bisherigen Testfälle plus die neuen Testfälle für die letzte Änderung auszuführen. Das Ziel war, das ganze Programm neu zu überdecken. Dadurch wird der Regressionstest genauso aufwändig oder sogar noch aufwändiger als der ursprüngliche Entwicklungstest. Regressionstestforschung zielt darauf, diesen Aufwand zu reduzieren, indem nur die Änderungen getestet werden.

In der Forschung werden grundsätzlich zwei Zielrichtungen verfolgt. Die eine stammt von einem Beitrag von Fischer auf der COMPSAC-Konferenz im Jahr 1977. Fischer hat damals eine Methode der Pfadanalogie entwickelt, um jene Pfade zu erkennen, die sich von den bisherigen Pfaden unterscheiden. Er schlug vor, den Regressionstest auf den Test aller neuen oder geänderten Ablaufpfade zu reduzieren [Fis77]. Yau und Kishimoto erweiterten das Konzept von Fischer, um den Eingabebereich einzubeziehen. Ihnen galt es, die geänderten und neuen Eingabevariablen zu berücksichtigen. Hierzu haben sie Ursache/Wirkungsgraphen, symbolische Ausführungsbäume und Testdatentabellen verwendet, um die neuen Testfälle zu unter-

scheiden [YaKi87]. Leung und White führten das Konzept der *impacted slices* ein, um Ablaufpfade aufzuzeigen, die durch die Änderung betroffen sind [LeWi89]. Hartmann und Robson haben die These von Fischer auf Modulgruppen erweitert, indem sie die Ablaufpfade über Modulgrenzen hinweg verfolgen und alle betroffenen Scheiben aus allen betroffenen Modulen identifizieren [HaRo90]. Benedusi, Cimitile und DeCarlina haben die Forschung in dieser Richtung mit einer ausführlichen Arbeit über Pfadänderungsanalyse abgerundet. Diese Arbeit führt zur automatischen Generierung der Regressionstestfälle aufgrund der Pfadanalyse [BCD89].

Die zweite Zielrichtung wurde erst Anfang der 90er Jahre durch eine Arbeit von Leung und White über Datenflussanalyse ausgelöst [LeWi90]. Sie setzten auf der Testforschung von Weiser, Ntafos, Weyuker, Korel und Laski auf, bei welcher der Datenfluss im Mittelpunkt steht. Datenflussbezogenes Testen geht entweder rückwärts von den Datenergebnissen oder vorwärts von den Dateneingaben aus, um den Datenfluss durch ein oder mehrere Module zu verfolgen. Jeder Datenflusspfad beginnt mit einer oder mehreren Variablen im Eingabebereich und führt über eine Kette von Zwischenergebnissen zu einem Endergebnis im Ausgabebereich. Durch die Programmänderung entstehen neue Datenflusspfade, oder es verändern sich bestehende Datenflusspfade. Ein data slicing Tool ist in der Lage, diese veränderten Datenströme zu erkennen und aufzuzeigen. Daraus ergeben sich datenflussbezogene Testfälle [Wey93].

Für den Regressionstest reengineerter bzw. konvertierter Programme sind diese herkömmlichen Testansätze zwar hilfreich, aber nicht ausreichend. Sie müssen durch weitere Testansätze ergänzt werden, die auf den Nachweis der funktionalen Äquivalenz zielen. Einer der Autoren hat schon 1992 die diversen Forschungsrichtungen auf diesem Gebiet im Hinblick auf ihre Nutzung bei Reengineering-Projekten untersucht und daraus fünf verschiedene Ansätze speziell für den Regressionstest in reengineerten Programmen abgeleitet [Sne92]. Sie sind alle Abgleichmethoden, welche die Eigenschaften der neuen Programme gegen die der alten Programme abgleichen (Abbildung 1.6):

- der Abgleich der Eingabe/Ausgabe-Bereiche,
- der Abgleich der Datenverwendungen,
- der Abgleich der Geschäftsregeln,
- der Abgleich der Ein/Ausgabe-Pfade und
- der Abgleich der Datenausgaben [Sne94].

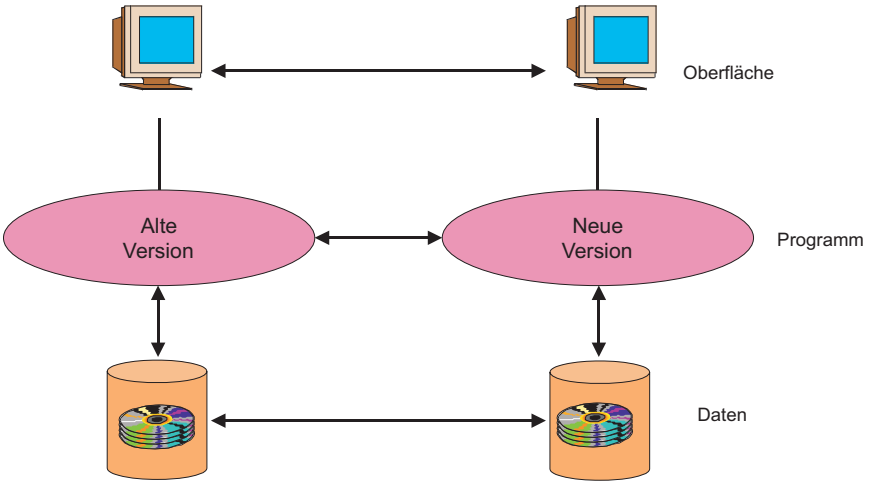


Abbildung 1.6 Regressionstest

1.2.6 Software Testnormen

Keine Einführung in die Testtechnologie wäre komplett ohne einen Verweis auf die ANSI/IEEE Standards zum Softwaretest. Diese Standards sind zwar in der Fachwelt umstritten, aber dass es sie überhaupt gibt, zeigt, dass dieses Teilgebiet der Informatik im Begriff ist, sich zu formieren.

Der IEEE Std. 829 mit dem Titel „Standard for Software Test Documentation" befasst sich mit der Dokumentation des Systemtests. Neben der Vereinbarung der wichtigsten Testbegriffe, z.B. *test case*, *test procedure*, *test plan* und *test item*, wird ein Gliederungsschema für einen Testplan, einen Testentwurf, eine Testfallspezifikation, eine Testprozedur und ein Testprotokoll vorgegeben. Erläutert werden die Dokumente anhand eines Personalabrechnungssystems. Diese Norm ist sehr praxisnah und lässt sich gut als Grundlage für eine betriebliche Testkonvention nutzen. Inzwischen ist sie auch in die ISO-Normen eingegangen [IEEE829].

Der *IEEE Standard for Software Unit Testing* [IEEE1008] ist eine Richtlinie für den Modultest. Er definiert, was Module (software units) sind und wie man sie testen sollte. Leider fehlt dieser Norm noch die Reife der ersten. Sie hat mehr den Charakter einer Schulungsunterlage mit ungefähr der gleichen Verbindlichkeit. Der Bezug zur Praxis fehlt. Trotzdem enthält sie einige nützliche Informationen, vor allem bezüglich der Testfallspezifikation und -validation. Dass sie nicht ganz gelungen ist, liegt an der Materie. Eine allgemeingültige, erprobte und praxisgerechte Methodik für den Modultest hat sich bisher noch nicht durchgesetzt [IEEE1008].

In Deutschland beschäftigten sich auch VDI, VDE und die Gütegemeinschaft Software mit Testnormen. Eine Untersuchung der Universität Köln zum Stand des Testens in der Bundesrepublik – eine der ersten dieser Art – führte zu einer Veröffent-

lichung durch die Träger der Untersuchung mit dem Titel „Software-Qualitätssicherung – Testen im Software-Lebenszyklus“ [SBM82].
In Zusammenarbeit mit dem VDE erarbeitete eine Arbeitsgruppe der deutschen Gesellschaft für Qualität eine Reihe von Vorschlägen, die ebenfalls unter dem Titel „Software-Qualitätssicherung“ erschien [DGQ12-51].
Schließlich hat die Gütegemeinschaft Software eine Testnorm für die Prüfung von Standard-Software durch die technischen Überwachungsvereine herausgebracht [DIN66285]. Der Normierungsprozess hat sich allerdings als sehr langwierig und politisch brisant erwiesen.

1.3 Client/Server-Testproblematik

Seit Anfang der 90er Jahre werden die bisherigen monolithischen Host-Systeme immer mehr durch verteilte Client/Server-Systeme abgelöst. Es ist nicht Sache dieses Buches, diese Trendwende zu beurteilen. Client/Server-Systeme haben sicherlich ihre Vorteile für den Anwender. Sie sind flexibel, modern und benutzerfreundlich. Außerdem erlauben sie es dem Anwender, unterschiedliche Plattformen und Software-Komponenten miteinander zu kombinieren. Man ist nicht mehr von einem einzigen großen Lieferanten abhängig. Andererseits haben Client/Server-Systeme auch ihre Nachteile. Sie sind schwieriger zu entwickeln und noch schwieriger zu testen.
Die Testproblematik bei Client/Server-Systemen darf nicht unterschätzt werden. Sie ist riesig. Nicht umsonst ziehen es die meisten Anwender vor, ihre Client/Server-Anwendungen als Fertigsoftware zu kaufen. Sie haben erfahren, was eine Eigenentwicklung kostet. Diese hohen Entwicklungskosten werden in erster Linie von dem Testaufwand verursacht. Die Komplexität solcher Systeme führt zu einer kombinatorischen Explosion der Anwendungsmöglichkeiten und damit zu den Testnotwendigkeiten. Die Anzahl der zu testenden Fälle wächst exponentiell im Verhältnis zur Systemgröße. Hinzu kommen einige gravierende Unterschiede zu den monolithischen Hostsystemen, Unterschiede, die zwar mehr Komfort und Flexibilität, aber auch mehr Testaufwand mit sich bringen. Die wichtigsten Unterschiede aus der Sicht des Tester sind folgende:

- die graphische Benutzungsoberfläche,
- die ereignisgesteuerte Programmlogik,
- die verteilten Programme,
- die verteilten Datenbanken und
- die heterogene Produktionsumgebung.

1.3.1 Graphische Benutzungsoberflächen

Graphische Benutzungsoberflächen bieten dem Anwender ein breites Spektrum an Kommunikationsmittel. Bei den alten Bildschirmmasken gab es nur die Felder der Masken und die Funktionstasten. Bei den neuen Oberflächen gibt es neben den klassischen Maskenfeldern und Funktionstasten auch Kommandoknöpfe, Ikonen, aufziehbare Menüfenster, Rollbalken, Listboxen, Druckknöpfe und viele andere visuelle Ein/Ausgabemedien. Endanwender können Tage damit verbringen, ihre Oberfläche zu arrangieren und die Farben, Fontgrößen, Schriftarten usw. einzustellen. Dies dürfte für den Benutzer ein Spass sein, für den Tester ist es ein Albtraum. Jedes Objekt bzw. Widget auf dem Schirm muss getestet werden und zwar potenziell im Zusammenhang mit jedem anderen Objekt. Der Test wird noch aufwändiger dadurch, dass Objekte gegenseitig voneinander abhängig sind. Es genügt nicht, einzelne Objekte in sämtlichen Erscheinungsformen zu testen, sondern es gilt, alle möglichen Kombinationen von Objekten zu testen. Dies führt schnell vom Hundertsten ins Tausendste. Um sicher zu gehen, müssen alle praktisch relevanten Kombinationen spezialisiert, dokumentiert und getestet werden. Der Oberflächentest wird so zu einem ungeheuerlichen Unterfangen, wenn man ihn systematisch angeht und nicht einfach willkürlich Aktionen ausprobiert. Ohne Automation läuft hier nichts. Es genügt, darauf hinzuweisen, dass die graphische Oberfläche eine Büchse der Pandora ist, mit einer fast unendlichen Anzahl möglicher Zustände. Ob sie auftreten werden oder nicht, hängt vom Verhalten des Endbenutzers ab, und das ist bekanntlich unvorhersehbar. Also muss man gegen alle Eventualitäten testen.

1.3.2 Ereignisgesteuerte Programmlogik

In konventionellen prozeduralen Programmen werden die Anweisungen prinzipiell deterministisch in der Reihenfolge abgearbeitet, in der sie im Quellcode stehen. Es gibt Entscheidungsknoten, bei denen der Steuerfluss in die eine oder andere Richtung gelenkt wird, und auch Schleifen, in denen der Steuerfluss zirkuliert, aber in jedem Fall lässt sich der Programmablauf verfolgen. Das dynamische Verhalten der Programme ist statisch anhand des Quelltextes nachvollziehbar.

Bei Client/Server-Programmen ist dies nicht der Fall. Sie sind ereignisgesteuert. Einzelne Programmbausteine reagieren auf externe oder interne Ereignisse (events). Ein solches Ereignis könnte ein Mausklick oder ein Tastendruck sein, es könnte auch eine Zeitüberschreitung oder der Eingang einer Nachricht aus dem Netz sein. Im Prinzip kann jede Zustandsänderung eine Programmfunktion auslösen. Die Ausführung einer Funktion führt zu einem neuen Zustand und dieser kann wiederum eine weitere Funktion auslösen. Auf diese Weise entsteht eine Kettenwirkung.

Die Reihenfolge der Funktionsausführungen ist nur dynamisch zu verfolgen. Statisch sind die Funktionen im Quellcode nach anderen Kriterien geordnet. In objekt-

orientierten Programmen sind die Funktionen nach dem Objekt ihrer Verarbeitung – der Klasse – gruppiert. In nicht objektorientierten Client/Server-Programmen sind sie oft nach logischen Kriterien geordnet. Der Zusammenhang zwischen dem statischen Programmaufbau und der dynamischen Programmausführung ist in beiden Fällen verloren gegangen.
Die Ereignisorientierung trägt jedenfalls zur Erhöhung des Testaufwands bei, denn dort, wo früher ein externes Ereignis wie z.B. die Freigabe einer Maske zu einem langen Pfad durch mehrere Funktionen führte, sind im Client/Server-System die Funktionspfade eher kurz und dafür viel zahlreicher. Sie haben obendrein viel mehr Auslöser. Insofern, als jede potenzielle Funktionsfolge zu testen sei, erhöht sich hier der Testaufwand relativ zum Zuwachs an Funktionen, die von außen auslösbar sind.

1.3.3 Verteilte Programme

In monolithischen Host-Systemen laufen alle Teile eines Programms in einem einzigen Adressraum ab. Auch wenn sie getrennt kompiliert werden, werden sie entweder statisch oder dynamisch zu einer Laufeinheit zusammengebunden. Bei Client/Server-Systemen haben wir es aber mit getrennten Adressräumen zu tun. Ein Teil des Programms – das Frontend – ist auf dem Clientrechner, ein anderer Teil – das Backend – ist auf dem Serverrechner. Im Falle einer Drei-Schichten-Architektur (3 tier) ist das Programm sogar auf drei Adressräume verteilt: ein Frontend, eine Mitte und ein Backend. Verbunden werden die verteilten Programmteile über entfernte Prozeduraufrufe (remote procedure calls, RPC). Hier wird eine Funktion in einem fremden Adressraum namentlich aufgerufen und statt Adressen werden Werte als Parameter übergeben (call by value).
Natürlich können hier mehr Probleme auftreten als in einem einzigen Adressraum. Die Ausnahmebehandlung (exception handling) wird zum zentralen Thema, da man nie sicher sein kann, ob die Nachricht von einem Programmteil zum anderen tatsächlich ankommt. Ein Großteil des Client/Server-Codes ist für den Notfall vorgesehen, dass die Verbindung zwischen den verteilten Programmteilen abreißt, und dieser Teil muss getestet werden. Bei konventionellen Programmen gab es auch Ausnahmefälle, z.B. beim Zugriff auf gesperrte Datenbanksätze, aber sie waren gering im Verhältnis zu den zahlreichen Ausnahmefällen, die in einem Client/Server-Programm vorkommen. Um sicher zu sein, dass sie alle korrekt behandelt werden, müssen sie alle erprobt werden.

1.3.4 Verteilte Datenbanken

Nicht nur die Programme, sondern auch die Datenbanken können in einem Client/Server-System verteilt sein. In dem Fall ist es durchaus möglich, dass ein

Programm seine Datensicht auf verschiedene Quellen beziehen kann, z.B. eine Auftragsbearbeitungssicht, die sich aus Kundendaten, Auftragsdaten und Artikeldaten zusammensetzt. Jede Datenart kann sich auf einem anderen Datenbankserver befinden. Mit der Sicht werden die Daten zur Laufzeit erst vereinigt und nach deren Nutzung wieder verteilt. Es versteht sich, dass dies den Testaufwand erhöht, vor allem was die Testdatengenerierung anbetrifft. Man muss die diversen Datenbestände so erstellen, dass sie zusammenpassen. Solange, wie alle Daten in einem Behälter lagen, war es einfach, bestimmte Datensichten zu generieren. Mit der Aufteilung einer Sicht auf mehrere Behälter wird es schwer, gerade jene Wertkombinationen zu produzieren, die ein Programm braucht. Hinzu kommen die zusätzlichen Fehlerquellen in den Gateways. Solche Middleware-Produkte wie ODBC und JDBC helfen, die Zugriffe für die Anwendungsprogramme zu vereinfachen, aber sie bieten komplexe Schnittstellen an, die getestet werden müssen.

1.3.5 Heterogene Produktionsumgebungen

Was des einen Glück ist, ist des anderen Misere. Manager freuen sich darüber, dass sie nicht mehr auf einen Hersteller angewiesen sind. Sie können ihre Lieferanten sogar gegeneinander ausspielen, um die Preise zu drücken. Es ist ferner möglich, dass jede Dienststelle sich für einen anderen Lieferanten entscheidet. Hinzu kommt, dass viele Client/Server-Systeme alte Host-Komponenten erben. Sie sind hybride Systeme. In einem Client/Server-Umfeld lässt sich im Prinzip alles mit allem verbinden (Abbildung 1.7).

Für den Tester kann diese Heterogenität leicht zum Albtraum werden. Denn ob die verteilten Komponenten wirklich zusammenpassen, stellt sich erst beim Test heraus. Die Lieferanten können viel versprechen. Manche täuschen, andere wissen es gar nicht. Auch die vielgerühmten Request Broker, die verschiedenartige Programme auf unterschiedlichen Rechnern koppeln können, funktionieren nicht immer wie erwartet. Mit selbst „gebastelter“ Middleware sieht es noch schlimmer aus. Hier ergeben sich unendlich viele Fehlerquellen. Mit jedem zusätzlichen Produkt erhöht sich der Integrationsaufwand multiplikativ, denn es müssen die Beziehungen zu allen anderen Produkten getestet werden.

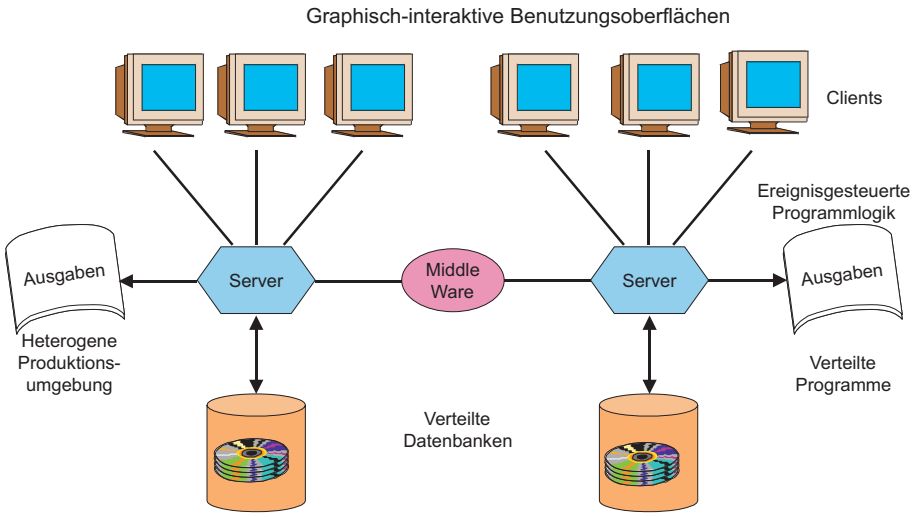


Abbildung 1.7 Client/Server Test

Man muss sich nur vorstellen, wie es wird, wenn eine Transaktion auf drei Rechnerstufen verteilt ist. Die Präsentationslogik befindet sich auf dem PC-Arbeitsplatz, die Geschäftslogik befindet sich auf einem Unix-Server und die Zugriffslogik ist auf einem zentralen Hostrechner. Der Test einer solchen Transaktion setzt voraus, dass alle Teiltransaktionen für sich in ihrer Umgebung getestet worden sind, dass die Middleware getestet wurde und dass alle Ressourcen gleichzeitig zur Verfügung stehen. Allein um den Test zu organisieren, ist ein großer Aufwand erforderlich, geschweige denn, der Aufwand für die Fehlerfindung, wenn etwas schief läuft.

Wenn so viele Produkte von so vielen Lieferanten beteiligt sind, ist es verständlich, wenn jeder die Schuld für Fehlverhalten auf den anderen schiebt. Ein Lieferant wird erst dann die Schuld auf sich nehmen, wenn klare Beweise vorliegen. Die Beweislast liegt beim Tester bzw. beim Anwender. Nicht umsonst werden so viele Client/Server-Anwendungen im Auftrag nach außen vergeben – Outsourcing. Kluge Anwender wissen, was auf sie zukommt.

1.4 Besonderheiten objektorientierter Systeme

1.4.1 Das Besondere an objektorientierten Programmen

Während zu Beginn der 80er Jahre die ersten industriell verwendeten Implementierungen objektorientierter Programmiersprachen verfügbar waren und sich ab 1985 methodische Ansätze zur objektorientierten Software-Entwicklung durchzusetzen begannen, blieb der Test objektorientierter Software lange Zeit über unbeachtet.

Viele populäre „Methodengurus“ verlieren in ihren Veröffentlichungen kein Wort über den Test. Grady Booch schreibt lediglich

> *„... the use of object-oriented design doesn’t change any basic testing principles; what does change is the granularity of the units tested.“ [Boo94]*

James Rumbaugh behauptet sogar

> *„Both testing and maintenance are simplified by an object-oriented approach...“ [RBP+91]*

Glaubte man also anfangs, dass objektorientierte Software einen erheblich reduzierten Aufwand für die Prüfung erfordern würde und dass bekannte Prüfverfahren unmodifiziert verwendet werden können, so weiß man heute, dass diese Hoffnungen sich nicht erfüllt haben. Erste Anzeichen für einen erhöhten Qualitätssicherungsbedarf für objektorientierte Software finden Perry und Kaiser, die bei der Untersuchung des Wiederverwendungspotenzials objektorientierter Software schon 1990 feststellen:

> *„... we have uncovered a flaw in the general wisdom about object-oriented languages — that "proven" (that is well-understood, well-tested, and well-used) classes can be reused as superclasses without retesting the inherited code.“[PeKa90]*

Im Allgemeinen gelten für objektorientierte Programme die gleichen Anforderungen wie für konventionell funktionsorientierte Programme. Es müssen repräsentative Test-Eingabedaten generiert, Zwischenergebnisse geprüft, Ablaufpfade verfolgt und Testergebnisse validiert werden. Ein Testfall wird in einem bestimmten Vorzustand ausgelöst und führt zu einem gegebenen Nachzustand. Das Programm, spezifiziert als Transformationsregel, gilt als korrekt, wenn der Ist-Nachzustand dem spezifizierten Soll-Nachzustand entspricht [LiRü96]. In dieser Hinsicht sind objektorientierte Programme nicht anders zu betrachten als die bisherigen. Dennoch ist diese Sicht nur für den Systemtest gültig. Was den Modultest und den Integrationstest betrifft, gibt es einige signifikante Unterschiede.

Erstens führt die stärkere Modularisierung zu mehr intermodularen Abhängigkeiten. Eine Methode einer bestimmten Klasse ist oft auf Methoden anderer Klassen angewiesen, um ihre Funktion ausführen zu können. Diese anderen Klassen unterliegen nicht selten der Zuständigkeit anderer Entwickler. So sind auch die Entwickler mehr voneinander abhängig [Dlu94]. Es ist nicht mehr so, dass ein Entwickler für die Abwicklung einer Transaktion vom Eintritt ins System bis zum Abschluss verantwortlich ist. Die Arbeitsteilung ist nicht mehr funktional ausgerichtet, sondern nach den Objekten der Verarbeitung. Hinzu kommen die vielen potenziellen Anwendungen. Nicht nur, dass öffentliche Methoden einer Klasse von Methoden jeder beliebigen anderen Klasse aufgerufen werden können, eine Klasse kann auch fremde Attribute und Methoden von jeder übergeordneten Klasse übernehmen. Das Erben fremder Eigenschaften aus anderen Klassen schafft weitere Abhängigkeiten.

In objektorientierten Systemen wird Redundanz auf Kosten der gegenseitigen Abhängigkeit eliminiert.

Zweitens kann eine Klasse eine Vielzahl einzelner Methoden beinhalten, die zwar prozedural voneinander unabhängig sind, jedoch über den Zustand der gemeinsamen Objektattribute miteinander verquickt sind. So kann die eine Methode einen Objektzustand hinterlassen, der das Verhalten der Nachfolgemethode beeinflusst, denn zum Argumentenbereich einer Methode gehören nicht nur ihre Eingangsparameter, sondern auch der Zustand ihres Objekts. Dadurch sind die Methoden einer Klasse doch über gemeinsame Daten gekoppelt. Falls Methoden derselben Klasse sich gegenseitig nutzen, was bei komplexen Klassen häufig vorkommt, entsteht auch noch eine prozedurale Kopplung [HaWi94].

Drittens ist bei der Entwicklung einer Klasse oft nicht bekannt, zu welchem Zweck die Methoden der Klassen verwendet werden. Sie müssten so programmiert sein, dass sie jeden potenziellen Zweck erfüllen können. Solche Offenheit führt zu einer Vielzahl möglicher Zustände, die nicht alle getestet werden können, ohne den finanziellen Rahmen eines Projekts zu sprengen. Ein Test aller relevanten Zustände erfordert eine Prognose über die potenzielle Nutzung eines Objekts. Falls diese Prognose daneben liegt, bleiben einige wichtige Zustände ungetestet. Hier offenbart sich eine Diskrepanz zwischen funktionsbezogener Software, die auf einen bestimmten Zweck zugeschnitten ist und die nur in Bezug auf diesen Zweck getestet werden muss, und objektorientierter Software, die vielen potenziellen Zwecken dienen sollte und deshalb in Bezug auf das potenzielle Nutzungsprofil getestet werden sollte [SmRo90].

Viertens können die von den Klassen beschriebenen Objekte viele mögliche Zustände annehmen. Je komplexer die Objekte sind, je mehr Attribute und Methoden sie haben, umso mehr Zustände können sie annehmen. Wer eine Klasse zu einhundert Prozent testen will, müsste alle möglichen Zustände des darin enthaltenen Objekts erzeugen und sämtliche Zustandsübergänge testen. Dies führt zu einer exponentiellen Steigerung der Anzahl von Testfällen für den Test der Klasse. Deshalb ist das 100-prozentige Austesten einer Klasse (wegen der Überzahl der Testfälle und Eingabedaten) in der Regel nicht praktikabel. Also muss man sich (wie schon beim herkömmlichen Test) auf geeignet ausgewählte Stichproben beschränken, Testen ist und bleibt also ein stichprobenartiges Verfahren. Bei der bisherigen prozeduralen transaktionsorientierten Programmierung hing die Anzahl der Datenzustände von der Logik der jeweiligen Transaktion ab, d.h. die Transaktionslogik grenzte den Problembereich ein. In offenen wiederverwendbaren OO-Systemen sind die Grenzen des Problembereiches eher fließend [TuRo93].

Schließlich reicht die klassische Instrumentierungstechnik nicht aus, um die Testüberdeckung des Codes einer Klasse festzustellen. Die prozedurale Testüberdeckungsmessung ging davon aus, dass alle Anweisungen bzw. Zweige und Ablaufpfade eines Programms der Erfüllung der Programmfunktion dienten und deshalb

getestet werden mussten. Das Programm war letzten Endes eine 1:1-Abbildung der Funktion. In objektorientierten Programmen gibt es diese einfache 1:1-Beziehung zwischen Programm und Funktion nicht mehr. Eine Klasse enthält Methoden, die alle Operationen auf einem bestimmten Objekt ausführen, unabhängig davon, zu welcher Anwendung sie gehören. So wird von einer bestimmten Anwendung nur ein Teil der Methoden benutzt. Es hat deshalb wenig Sinn, den Code in seiner Gesamtheit zu instrumentieren, denn große Teile davon bleiben von den zu testenden Transaktionen unberührt. Es gilt, nur jene Methoden zu testen, die eine gegebene Transaktion beansprucht. Darum muss nach einem Nutzungsprofil instrumentiert werden, um die funktionale Testüberdeckung zu ermitteln. Eine derart selektive Instrumentierung ist natürlich technisch anspruchsvoller als eine blinde Instrumentierung des gesamten Codes [Sne95].

Objektorientierte Software ist nicht nur schwieriger zu testen, sie kann auch mehr Fehler verursachen, vor allem wenn die Entwickler mit der Technologie nicht so vertraut sind. Polymorphie und dynamische Bindung führen zu einer Vermehrung der potenziellen Ablaufpfade und damit zu einer Vermehrung der potenziellen Fehler. Vererbung schafft viele subtile, unsichtbare Abhängigkeiten und Kapselung beschränkt die Sicht auf die Objektzustände und erschwert dadurch die Fehlererkennung. Die zahlreichen Kollaborationen zwischen Objekten schaffen auch viele Schnittstellen, die wiederum viele Abstimmungen erfordern. Abstimmungen führen wiederum zu Missverständnissen und Missverständnisse zu Fehlern [Bin99].

In bisherigen Untersuchungen über Fehlerraten in objektorientierten Systemen hat es sich gezeigt, dass die Fehlerdichte eher höher ist als in klassischen Systemen. Nach den Daten der Bournemouth Universität in England sind Klassen in Vererbungshierarchien dreimal so fehleranfällig wie Klassen ohne Vererbung [ShCa97].

Bereits 1996 hat Capers Jones 600 objektorientierte Projekte in 150 Anwenderbetrieben untersucht und ist zu den folgenden Schlüssen gekommen:

- Die Anzahl Fehler aus mangelnder Erfahrung mit der Technologie ist auffallend hoch.
- Fehler in der Analyse und im Entwurf haben eine viel größere Auswirkung als Fehler in den bisherigen Analyse- und Entwurfsmethoden.
- Es ist schwieriger, Fehlerursachen aufzudecken.
- Der Code ist weniger umfangreich, was zur Folge hat, dass die Fehlerdichte höher ist.
- Wiederverwendete Klassen weisen in der Regel weniger Fehler auf [Jon97].

Daraus lässt sich schließen, dass objektorientierte Systeme im Prinzip zuverlässiger sein könnten, vorausgesetzt, alles stimmt und die Entwickler beherrschen die Technologie, was aber in der Tat selten so ist. In der Praxis haben objektorientierte Systeme eine höhere Fehlerrate als die bisherigen. Hinzu kommt, dass der Test objektorientierter Programme sich um einiges anspruchsvoller als der Test klassischer

Programme gestaltet. Boris Beizer schreibt in einem Beitrag zum American Programmer:

> *„...it costs a lot more to test OO-software than to test ordinary software – perhaps four or five times as much ...Inheritance, dynamic binding, and polymorphism create testing problems that might exact a testing cost so high that it obviates the advantages [Bei94].“*

Ob diese Aussage stimmt oder nicht, bleibt dahingestellt. Fest steht, dass objektorientierte Programme anders zu testen sind. Einige Probleme sind geringer geworden, z.B. tief verschachtelte Ablaufstrukturen, dafür treten neue Probleme auf, z.B. dynamische Bindung mit einem nicht statisch determinierbaren „Zielobjekt“ einer Botschaft. Die Komplexität der Programmlogik verlagert sich von der intramodularen zur intermodularen Komplexität. Offenheit und Wiederverwendung fordern ihren Preis. Daher müssen neue Testansätze erprobt werden, Ansätze, die den Besonderheiten objektorientierter Programme gerecht werden.

1.4.2 Testgegenstände in einem OO-System

Der Test bezieht sich immer auf bestimmte Gegenstände. Ein Gegenstand wird aus dem Gesamtzusammenhang herausgeholt und für sich in einer kontrollierten Testumgebung getestet. Es werden Vorbedingungen erfüllt, Ausgangszustände gesetzt, Eingaben zugeführt und Ausgaben abgefangen und untersucht. Es gilt festzustellen, ob der Gegenstand sich korrekt bzw. laut Spezifikation verhält. Der Umfang des Tests hängt von der Größe des Gegenstands ab.

In einem objektorientierten System ist im Prinzip die kleinste testbare Einheit eine Methode (Abbildung 1.8). Eine Methode ist zwar eingebettet in einer großen Quellcode-Einheit, aber sie hat einen eigenen Eingang, einen Ausgang und eigene Parameter. Es ist daher ohne weiteres möglich, einzelne Methoden aufzurufen und ihr Verhalten zu prüfen. Methoden entsprechen in etwa den Kontrolleinheiten in einer graphisch-interaktiven Benutzungsoberfläche und sind dementsprechend einzeln testbar.

Die nächste größere Einheit ist die Klasse. In einer Klasse sind alle Methoden, die einen bestimmten Objekttyp verarbeiten, zusammengefasst bzw. gekapselt. In der Regel realisiert eine Klasse genau einen abstrakten Datentyp. Von bzw. anhand der Klasse werden Objekte (Instanzen) generiert, verarbeitet und am Ende entweder gelöscht oder aufbewahrt. Im zweiten Fall spricht man auch von persistenten Objekten. Getestet wird eine Klasse über ihre Schnittstellen bzw. über ihre Methoden, angefangen mit dem Konstruktor zur Erzeugung eines Objekts bis hin zum Destruktor zu seiner Zerstörung. Klassen können jedoch sehr klein sein. Oft sind mehrere Klassen in einem Source-Modul zusammengefasst bzw. einer Klassenmenge – einem Paket – zugeordnet.

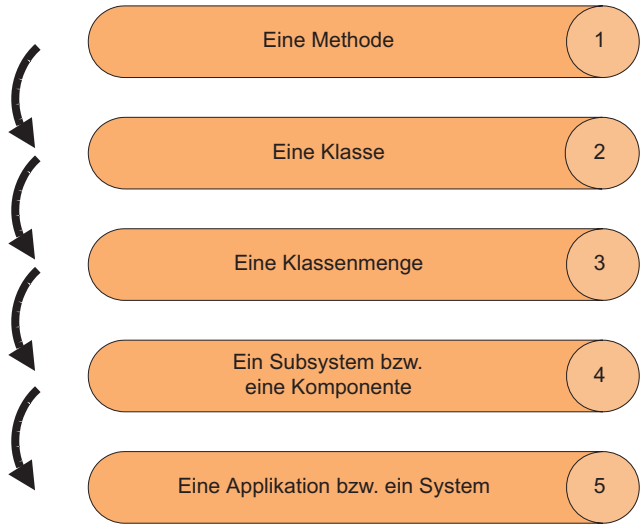


Abbildung 1.8 Testgegenstände in OO-Systemen

Objektorientierte Systeme sollten von Anfang an auf Testbarkeit ([Jun99]) und Wartbarkeit ausgerichtet, d.h. in Komponenten geteilt werden. Komponenten sind mehr oder weniger abgeschlossene Klassenmengen, in denen nur sehr wenige Abhängigkeiten zu Klassen in anderen Komponenten bestehen. Es soll nur definierte Schnittstellen bzw. APIs zwischen getrennt gebildeten Komponenten geben. Komponenten sind also gute Testgegenstände. Je mehr Komponenten ein System hat, desto leichter ist es zu testen [Bin94a].

Schließlich ist das Anwendungssystem selbst der endgültige Testgegenstand. Natürlich gibt es hier auch kleine und große Anwendungen. Aus der Sicht des Tests, aber nicht nur aus dieser Sicht, ist es immer besser, kleine überschaubare Anwendungen mit einer begrenzten Anzahl Ein- und Ausgaben zu haben. Je kleiner die Anwendung, desto geringer der Testaufwand. Gerade in einer komplexen Client/Server-Welt empfiehlt es sich, die Anwendungen möglichst klein zu halten. Die Argumentation, dass alles fachlich zusammengehört, ist nicht stichhaltig. Im Prinzip ist alles ein großes System. Dazu zählt das ganze Unternehmen. Demnach muss das noch lange nicht heißen, dass das Ganze als ein einziges allumfassendes Anwendungssystem implementiert werden muss. Die Kunst der Systemplanung ist, möglichst kleine unabhängige Anwendungssysteme zu schaffen, die in einer angemessenen Zeit implementierbar und auch testbar sind.

1.4.3 Folgen der Kapselung

Die Kapselung der Daten und Funktionen in einzelnen abgeschlossenen Objekten mit fest definierten Schnittstellen nach außen bringt für den Test bzw. für die Feh-

lerfindung auf den ersten Blick nur Vorteile. Es müsste leichter sein, gekapselte Objekte unabhängig von den anderen Objekten zu testen und es müsste vor allem leichter sein, Fehlerursachen zu lokalisieren. Andererseits können Objekte eine Vielzahl möglicher Zustände annehmen, und diese Zustände müssen alle ausprobiert werden. Hinzu kommt, dass die Schnittstellen zu den Objekten recht komplex werden können. Auch sie sind in allen Varianten zu testen. Methoden können nicht einfach einzeln aufgerufen werden. Da ihre Ausführung vom jeweiligen Objektzustand abhängt und dieser Zustand durch die vorher ausgeführten Methoden geprägt wird, müssen alle Kombinationen der Methodenausführungsfolge erprobt werden. Nur dadurch ist sicher zu stellen, dass die Methoden aufeinander abgestimmt sind (Abbildung 1.9). Dennoch wirkt sich die Kapselung für die Testbarkeit eher positiv aus, auch wenn die Kontrolle der Objektzustände dadurch erschwert wird. Die Kapselung sperrt nämlich die Sicht auf die Zwischenergebnisse. Tester sind gezwungen, über Umwege wie z.B. Friend-Funktionen in C++ auf die Objekte zuzugreifen. Insofern hat Kapselung auch eine (geringe) negative Wirkung auf die Testbarkeit ([Str92] [Jun99]).

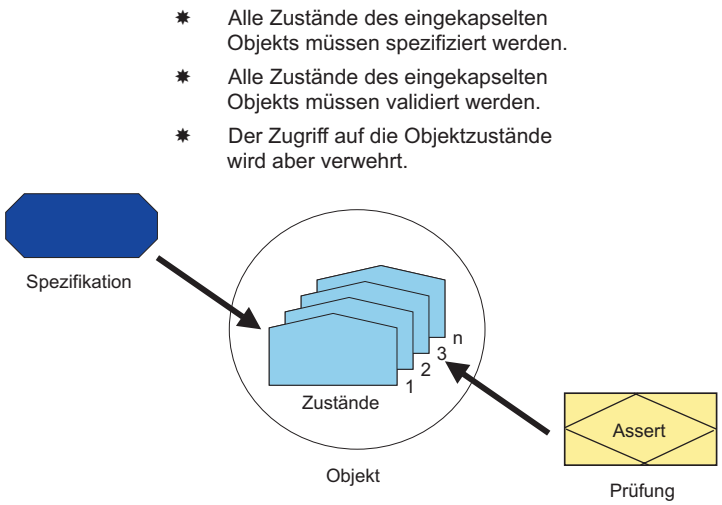


Abbildung 1.9 Folgen der Kapselung

1.4.4 Folgen der Vererbung

Bei der Vererbung sieht dies anders aus. Vererbung ist ein zweischneidiges Schwert. Man kann den Gegner – die Komplexität – damit bekämpfen, aber man kann sich auch selbst damit verletzen. Auf der einen Seite ist es positiv anzusehen, dass dadurch weniger Code entsteht. Die Unterklassen verweisen auf die Attribute und Funktionen der Oberklassen. Wenn diese stimmen, ist die Welt in Ordnung,

aber wehe wenn sie nicht stimmen. Probleme in den Basisklassen werden auf alle abgeleiteten (Unter-)Klassen übertragen, d.h. auch die Fehler werden vererbt. Basisklassen dürfen auch nicht verändert werden, denn jede Änderung kann zu neuen Fehlern in den abhängigen Klassen führen (Abbildung 1.10). Daraus folgt, dass die Basis- und übergeordneten Klassen im voraus sehr gut getestet werden müssen und dass sie nachher stabil bleiben müssen. Dies stellt hohe Anforderungen an die Entwickler dieser Klassen. Anforderungen, denen die meisten Anwendungsentwickler nicht gewachsen sind.

Es kann sein, dass Klassen in einer Generalisierungshierarchie so sehr voneinander abhängig sind, das es kaum möglich ist, einzelne Klassen allein für sich zu testen. In diesem Falle ist nicht die Klasse, sondern die Klassenmenge bzw. die Teilhierarchie der Testgegenstand. Hier wird nicht die einzelne Klasse, sondern die Klassenhierarchie als Modul behandelt und entsprechend getestet. Objektorientierte Systeme können sehr große Klassenhierarchien beinhalten, in denen alles von allem abhängt. Die untersten Klassen erben von bzw. verweisen auf die weiter oben in der Hierarchie stehenden Klassen. Dem Tester bleibt nichts anderes übrig, als alles zusammen zu testen. Dies ist oft der Fall bei Smalltalk-Systemen. Für den Test und auch für die Wartung ist dies eine denkbar ungünstige Situation, die, wenn möglich, zu vermeiden ist.

Vererbung ist in der Tat die `GOTO`-Anweisung der Objektorientierung. Sie gewährt abgeleiteten Klassen (Unterklassen) den direkten Zugriff auf Elemente der ihnen übergeordneten Klassen (Oberklassen) und durchbricht damit die Kapselungsmauern. Durch das Überschreiben geerbter Funktionen (*overloading)* wird die Komplexität der Abläufe erhöht, durch das Ändern von Parametertypen (*Overriding)* werden die Schnittstellen verschleiert. Vererbung bewirkt eine automatische Wiederverwendung, auch dort, wo sie nicht beabsichtigt ist. Deshalb muss sie mit großer Vorsicht angewendet werden, da sie sonst nicht nur den Test erschwert, sondern auch zu schwer erkennbaren Fehlern führt [Rya97].

Die beiden Forscher Perry und Kaiser fassen zusammen:

> *„Inheritance is one of the primary strengths of object-oriented programming. However, it is precisely because of inheritance that we have found so many problems arising with respect to testing....Encapsulation together with inheritance, which intuitively ought to bring a reduction in testing problems, compounds them instead."* ***[PeKa90]***

Vererbung hat, was die Testbarkeit betrifft, einen weiteren entscheidenden Nachteil. Es ist damit nicht mehr möglich, die untergeordneten Klassen unabhängig von den übergeordneten Klassen zu testen. Eine im objektorientierten Sinne gut entwickelte Klasse besteht fast nur aus Verweisen auf eine oder mehrere Ahnenklassen. Klasse `A` erbt von Klasse `B` und Klasse `B` erbt von Klasse `C` usw. Wer Klasse `A` testen will, muss die Klassen `B` und `C` usw. mit testen. Zum Schluss wird die Hälfte des Systems in den Klassentest einbezogen. Wo bleibt dann der Modultest? Was ist dann ein Modul – am Ende eine ganze Klassenhierarchie? Dies führt dazu, dass in komple-

xen Systemen mit einer hohen Vererbungstiefe die untergeordneten Klassen kaum testbar sind, zumindest nicht im Sinne eines Modultests. Auf jeden Fall stellt dies hohe Anforderungen an den Klassentestrahmen.

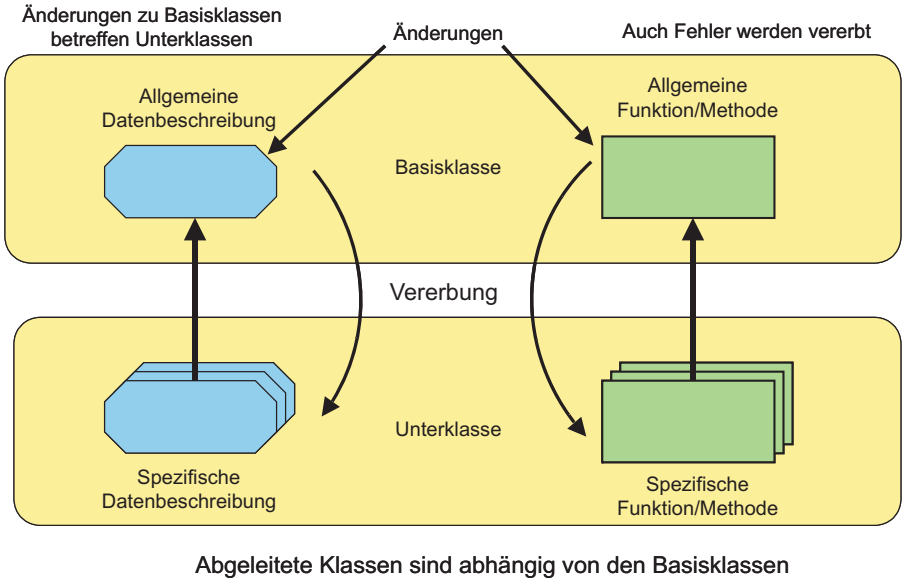


Abbildung 1.10 Folgen der Vererbung

1.4.5 Folgen der Polymorphie

Die Polymorphie stellt nicht weniger Probleme für den Tester dar. Die Entscheidung zur Laufzeit, welche Funktion in welchem Objekt einen Auftrag zu erledigen hat, macht aus dem Programmablauf einen nicht unmittelbar aus dem Quellcode herleitbaren Vorgang (Abbildung 1.11). Da die Ablauffolge nicht statisch voraussagbar ist, müssen alle möglichen dynamischen Folgen erprobt werden. Wenn die Polymorphie mehrfach wiederholt wird, explodiert die mögliche Anzahl der Ablaufpfade. Es wird kaum möglich sein, alle potenziellen Ablaufpfade zu testen. Man wird sich auf repräsentative Funktionsfolgen beschränken müssen, d.h. es bleibt die Unsicherheit, ob alle potenziellen Pfade durch das System wirklich funktionieren.

Jede mögliche dynamische Bindung einer Nachricht stellt einen weiteren Ablaufpfad dar. Die Tatsache, dass ein Pfad funktioniert, ist noch lange keine Garantie, dass die anderen Pfade auch korrekt sind. Jede einzelne Bindung muss für sich getestet werden. Außerdem bringt die Polymorphie eine Vielzahl neuer Fehlerquellen mit sich, die in der Literatur hinlänglich dokumentiert sind [PoBu94].

Es wäre verkehrt, den Polymorphismus als solchen pauschal zu verdammen, aber mit ihm muss ebenso wie mit der Vererbung äußerst diszipliniert und vorsichtig umgegangen werden. Nur so lassen sich potenzielle Fehler vermeiden und der Testaufwand in Grenzen halten.

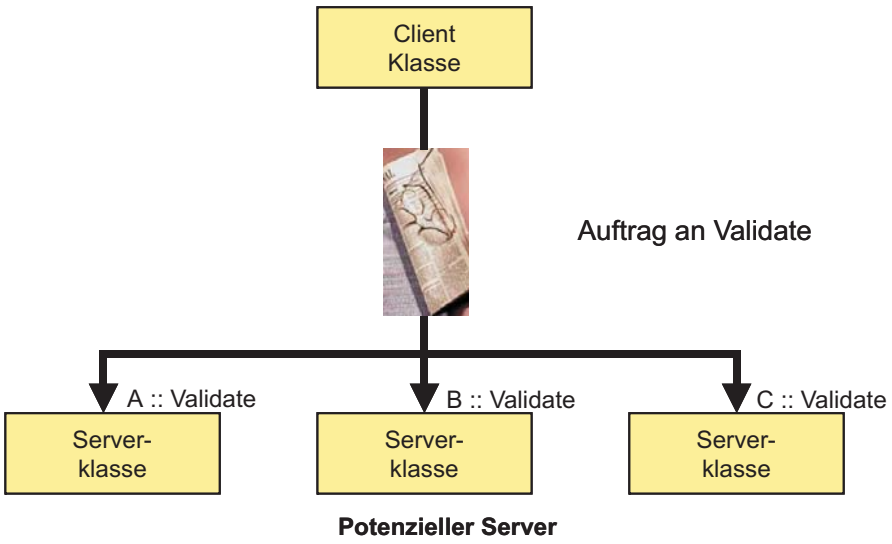


Abbildung 1.11 Folgen der Polymorphie

1.5 Objektorientierter Test – eine Herausforderung

Summa summarum ist der Test die Kehrseite der Objektorientierung [Rin96]. Viele der damit verbundenen Vorteile bringen für den Test nur Nachteile. Im Zusammenhang mit der Testproblematik von Client/Server-Systemen im Allgemeinen stellen objektorientierte Client/Server-Systeme eine ungeheure Herausforderung für den Test dar. Wenn er nicht systematisch und mit genug Aufwand betrieben wird, haben die Entwickler wenig Chancen, ihr System jemals stabil zu bekommen. Es darf niemanden wundern, wenn die Fehlerrate bei OO-Systemen höher liegt als bei vergleichbaren konventionellen Systemen. Dies wurde durch die u.A. von Les Hatten und Watts Humphries durchgeführten Zuverlässigkeitsstudien mehrfach nachgewiesen [Hat98]. Wer auf hohe Zuverlässigkeit zielt, sollte auf die Objektorientierung lieber verzichten, und wer damit unbedingt arbeiten will, sollte bereit sein, einen hohen Testaufwand zu betreiben – einen Aufwand, der den Entwicklungsaufwand bei weitem überschreitet.

Dieses Buch richtet sich an all diejenigen, die solche objektorientierten, verteilten Systeme testen müssen. Ihre Aufgabe ist nicht einfach und nur durch systematisches Vorgehen, gekoppelt mit automatisierten Techniken, überhaupt zu bewältigen. Beides soll in den folgenden Kapiteln ausführlich behandelt werden.

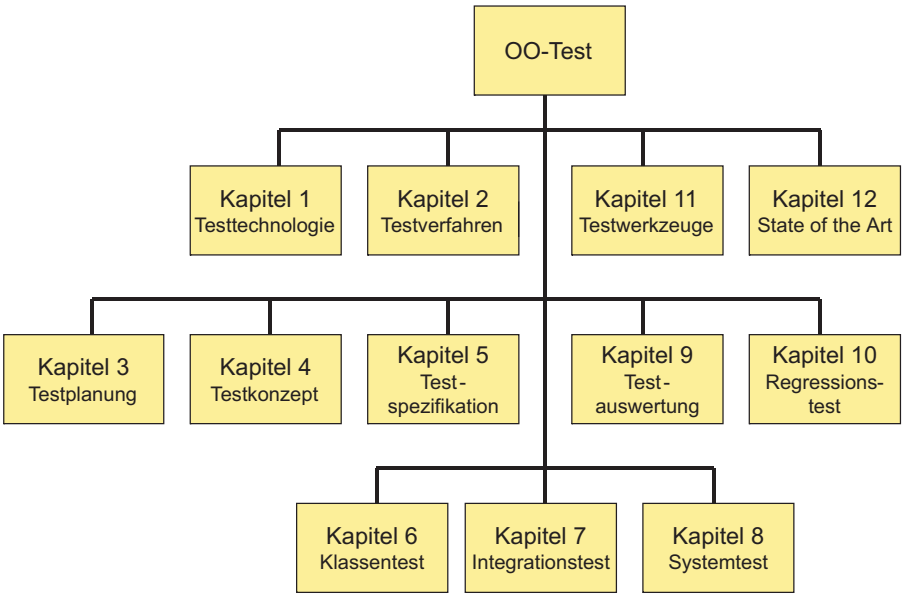


Abbildung 1.12 Gliederung der Thematik

Den Aufbau des Buchs skizziert Abbildung 1.12. Im nächsten Kapitel wird das iterative Testverfahren mit den Hauptstufen

- Klassentest = White-Box Test,
- Integrationstest = Grey-Box Test und
- Systemtest = Black-Box Test

geschildert.

Im dritten Kapitel wird auf die Testplanung eingegangen. Dort wird beschrieben, wie man den Test eines verteilten, objektorientierten Systems nach der ANSI-Norm 829 [IEEE829] plant und wie man den Testaufwand schätzen kann. Im vierten Kapitel wird im Rahmen des Testentwurfs ein Konzept für den Test objektorientierter Anwendungssysteme vorgestellt. Damit soll ein Überblick über die möglichen Testtechniken vermittelt werden, unabhängig von den jeweiligen Teststufen. Im fünften Kapitel wird beschrieben, wie objektorientierte Testfälle zu spezifizieren sind, und zwar für alle Teststufen, vom Klassentest bis zum Abnahmetest.

Nach der Einführung in den Testprozess werden die einzelnen Stufen des Tests objektorientierter Software beleuchtet. Im sechsten Kapitel wird der Klassentest

behandelt und an Hand einer in C++ implementierten Klasse demonstriert. Im siebten Kapitel wird ein Ansatz für die zweite Teststufe – den Integrationstest – vorgestellt und am Beispiel einer CORBA-Schnittstelle erläutert. Im achten Kapitel wird ein Ansatz für die dritte Teststufe – der Systemtest – beschrieben und mit einer Fallstudie aus der Praxis konkretisiert.

Im neunten Kapitel wird das wichtige Thema der Testauswertung bzw. die Testfortschrittsmessung behandelt. Hier geht es um die Testendekriterien, was heißt „genug getestet“ und wie man es feststellt. Dazu werden die wichtigsten Testmetriken herangezogen.

Im zehnten Kapitel wird die Bedeutung des Regressionstests bei der Wartung und Weiterentwicklung objektorientierter Software unterstrichen. Für solche Systeme ist der Regressionstest wegen der vielen Entwicklungszyklen besonders wichtig.

Im elften Kapitel kommen die Testwerkzeuge dran. Es werden die wesentlichen Toolarten geschildert und anhand einiger namhafter Produkte demonstriert. Hier wird auch eine Marktübersicht angeboten.

Im zwölften und letzten Kapitel wird auf den Stand der Technik zum Thema Testen objektorientierter Software eingegangen. Hier werden nicht nur Forschungsberichte, sondern insbesondere auch handfeste industrielle Erfahrungen beim Test objektorientierter Client/Server-Applikationen ausgewertet. Zum Schluss werden die Perspektiven für eine Lösung des Testproblems aus heutiger Sicht beurteilt.

2

Objektorientiertes Testverfahren

Testverfahren nach ANSI/IEEE-829
Testverfahren für Client/Server-Systeme
Testverfahren für objektorientierte Systeme
Phasen des objektorientierten Tests
Ergebnisse des objektorientierten Tests
Verantwortlichkeit für den objektorientierten Test
Werkzeuge für den objektorientierten Test
Der iterative Testprozess

Inhaltsübersicht Kapitel 2

2 Objektorientiertes Testverfahren

Objektorientierte Systeme sind in der Regel Client/Server-Systeme, und Client/Server-Systeme sind Software-Systeme. All das, was für Software-Systeme im Allgemeinen gilt, gilt auch für Client/Server-Systeme, und das, was für Client/Server-Systeme im Allgemeinen gilt, gilt auch für objektorientierte Systeme. Andersherum gesehen: Objektorientierte Systeme erben die Testverfahren von Client/Server-Systemen und Client/Server-Systeme erben die Testverfahren von Software-Systemen. Deshalb ist es notwendig, mit dem Testverfahren für Software-Systeme im Allgemeinen anzufangen, um das auf objektorientierte Systeme im Spezifischen zu projektieren.

2.1 Testverfahren nach ANSI/IEEE-829

Das Testverfahren für Software-Systeme ist in dem ANSI/IEEE-Standard 829 festgelegt. Nach dieser Norm ist der Test in sieben Phasen durchzuführen ([ISO12207], s. Abbildung 2.1):

- Testplanung
- Testentwurf
- Testfallspezifikation
- Testprozedurerstellung
- Testaufbau
- Testausführung
- Testauswertung

2.1.1 Testphasen nach der ANSI-Norm

Die *Testplanung* befasst sich mit der Planung der Testaufgaben, der Terminsetzung, der Zielsetzung, der Definition, der Abnahmekriterien und der Zuteilung der Betriebsmittel. Das Ergebnis ist ein normierter Testplan mit 16 Kapiteln.
Der *Testentwurf* ist die Konzipierung der Testszenarien. Hier wird bestimmt, welche Komponenten und welche Funktionen in welcher Reihenfolge wie getestet

werden. Das Ergebnis ist ein Testkonzept mit einem Kapitel für jede Testphase und je einem Abschnitt für jedes Teilsystem.

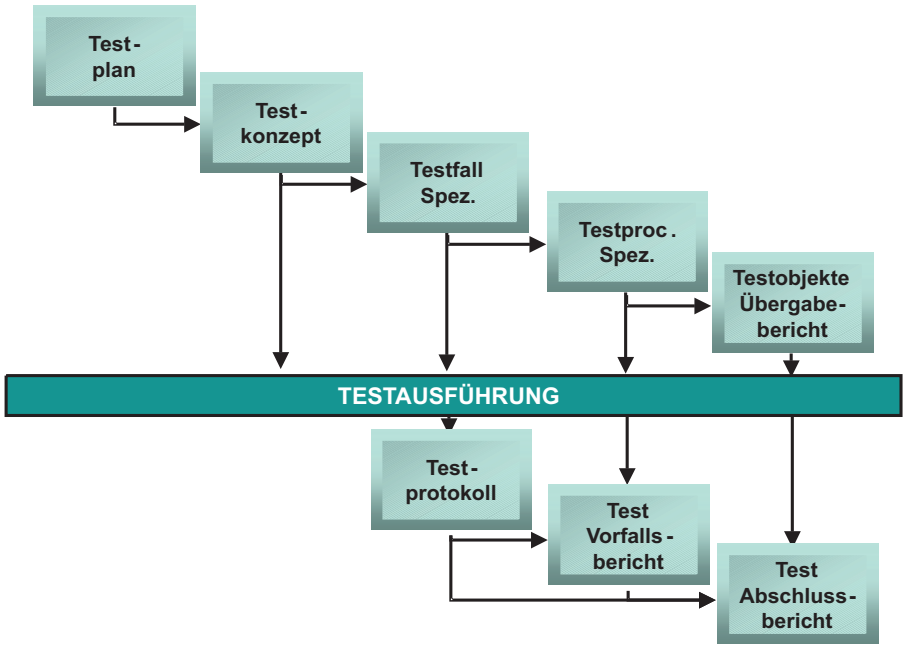


Abbildung 2.1 Testverfahren nach ANSI/IEEE-829

Die *Testfallspezifikation* ist eine detaillierte Beschreibung jedes einzelnen Testfalls anhand des Fachkonzepts und des Systementwurfs. Jeder Testfall zielt auf die Erprobung einer oder mehrerer Funktionen auf einer bestimmten Stufe der Software-Architektur. Der Testfall hat Vorbedingungen und Nachbedingungen, die hier zunächst semiformal beschrieben werden. Das Ergebnis dieser Phase ist ein Verzeichnis mit mehreren Hunderten bzw. Tausenden Testfällen je nach Systemumfang.

Die *Testprozedurerstellung* ist die Umsetzung der semiformalen Testfälle in formalen Testprozeduren bzw. in Testskripten. Hier wird also für jedes Testszenario eine Testprozedur mit n Testfällen generiert bzw. kodiert. Das Ergebnis ist eine Menge Source-Dateien mit ausführbaren Testskripten.

Der *Testaufbau* zielt auf den Aufbau der Testumgebung, die Installation der Testwerkzeuge, die Generierung der Testdaten, die Vorbereitung der Testprozeduren und die Übernahme der Testgegenstände. Das Ergebnis ist eine fertige Testumgebung samt Hardware und Software.

Die *Testausführung* ist der Test selbst. Es werden eine oder mehrere Testszenarien mit jeweils *n* Testprozeduren mal *m* Testfällen durchgeführt. Dabei werden diverse Testprotokolle produziert, darunter ein Ablaufprotokoll, ein Ergebnisprotokoll und

ein Überdeckungsprotokoll. Das Ergebnis ist ein Testlog und ein Testvorfallsbericht, in dem alle Probleme bzw. Abweichungen vom Soll notiert sind.
Die *Testauswertung* ist schließlich eine Bewertung der Testergebnisse im Hinblick auf die Testziele. Hier gilt es zu entscheiden, wann der Test zu Ende ist bzw. wann der Test abzubrechen ist. Das Ergebnis ist ein Testabschlussbericht.

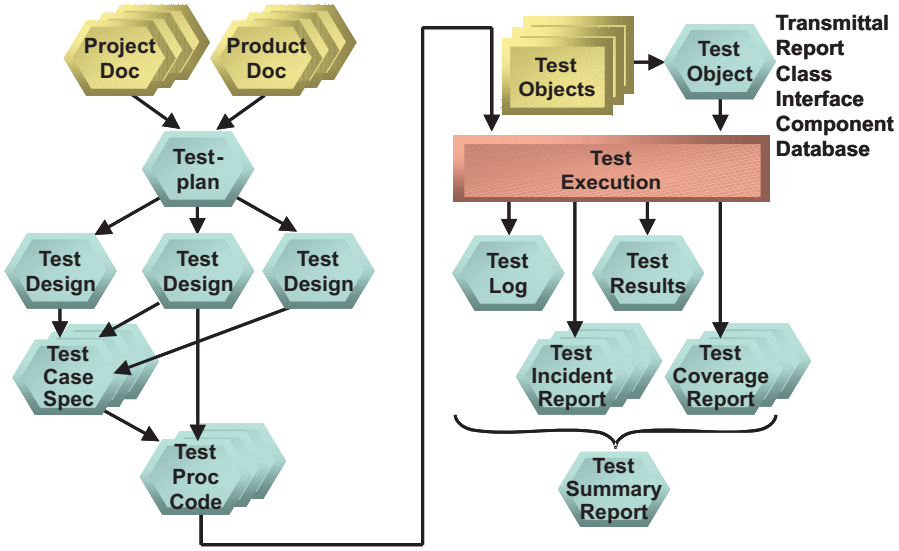


Abbildung 2.2 Testergebnisse nach ANSI/IEEE-829

2.1.2 Testergebnisse nach der ANSI-Norm

Nach der ANSI/IEEE-Norm ist der Test im Prinzip ein zweites Parallel-Projekt neben dem eigentlichen Entwicklungsprojekt mit eigenen Phasen, eigenen Betriebsmitteln und eigenen Ergebnissen, die mit denen der Entwicklung abgestimmt sind. Dies entspricht übrigens auch dem V-Modell der Software-Entwicklung, das in Europa propagiert wird ([BrDr93], [VM97]). Im Verlauf des Projekts entstehen also folgende Ergebnisse (Abbildung 2.2):

- ein Testplan,
- ein Testkonzept,
- eine Testfallspezifikation,
- eine Menge Testprozeduren,
- eine Testumgebung,
- eine Reihe Testprotokolle und
- ein Testabschlussbericht.

Dieses normierte Testverfahren gilt als Ausgangsbasis für den Test aller Software-Systeme, einschließlich Client/Server- und objektorientierter Systeme. Natürlich kommt bei jeder Systemart einiges dazu. Die Aufgaben werden anders durchgeführt und die Ergebnisse anders gestaltet, aber prinzipiell sind alle weiteren Testverfahren aus diesem Grundverfahren abzuleiten [KoPo99].

2.2 Testverfahren für Client/Server-Systeme

2.2.1 Problematik des verteilten Tests

Client/Server-Systeme sind verteilte Systeme, und verteilte Systeme bedingen einen verteilten Test. In einem verteilten Testverfahren gibt es zwar die gleichen Phasen wie in einem zentralen Testverfahren, aber die Phasen laufen etwas anders ab, mehr parallel. Sie haben eine größere Überlappung, und dem Integrationstest kommt eine stärkere Bedeutung zu [MiKo98].

Beim konventionellen Test auf dem Mainframe hat der Integrationstest oft völlig gefehlt. Die Module wurden vielleicht getestet, dann alle zusammengebunden, und was folgte, war ein Funktionstest auf Systemebene. Diese Vorgehensweise war dadurch begünstigt, dass alles miteinander eng zusammenhing – der TP-Monitor, das Datenbanksystem und das Anwenderprogramm. Die einzige klare Trennung gab es zwischen dem Batch- und dem Onlinebetrieb, was dazu führte, dass diese beiden Teilsysteme getrennt getestet wurden.

In einem Client/Server-Umfeld ist die Software verteilt (Abbildung 2.3). Auf den Clientrechnern befinden sich ein oder mehrere Teilsysteme für die Präsentationslogik. Beim Serverrechner kann es mehrere Teilsysteme für die Geschäftslogik geben. Schließlich kann es auch mehrere Datenhaltungssysteme in der Zugriffsschicht geben, d.h. wir haben es hier mit wesentlich mehr Teilsystemen zu tun. Demzufolge wächst der Integrationsaufwand und damit die Bedeutung des Integrationstests [Spi90].

Die Tatsache, dass mehrere Rechnerarten und eventuell mehrere Betriebssysteme in einem Client/Server-System beteiligt sind, bleibt nicht ohne Einfluss auf das Testverfahren. Es wird mehr Testpersonal benötigt, und dieses Personal muss gleichzeitig dasselbe Ziel, aber von verschiedenen Richtungen aus anstreben. Ihre Koordinierung erfordert eine straffere Planung und eine rechtzeitige Abstimmung der Testaktivitäten. Ein detaillierter Entwurf der Testszenarien wird unentbehrlich. Wegen der unterschiedlichen Hardware und Software wird auch der Testaufwand viel größer. Es muss eine Testumgebung eingerichtet werden, die der endgültigen Produktionsumgebung gleichkommt.

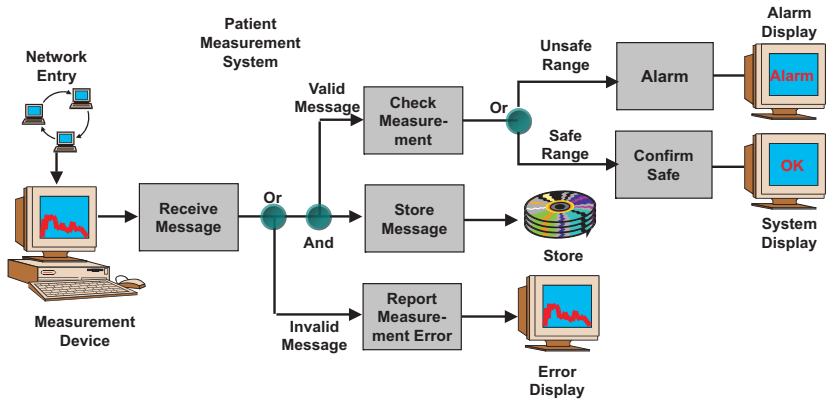


Abbildung 2.3 Test verteilter Systeme (RNETS)

2.2.2 Ansätze zum Test verteilter Systeme

Der Test verteilter Client/Server-Systeme wird in der Fachliteratur als eine neue Problematik behandelt, die auch neue Lösungsansätze bedingt. Kelly Bourne beschreibt in ihrem Buch „Testing Client/Server Systems“, warum Client/Server-Systeme anders getestet werden müssen [Bou97] und schlägt dafür fünf Teststufen vor:

- Unit-Test, wobei eine Unit ein Windows-Control, eine Geschäftsfunktion, ein Bericht oder eine gespeicherte Prozedur sein kann;
- Integrationstest, wobei es darum geht, die Interaktion zwischen verteilten Komponenten zu bestätigen;
- Systemtest, um das System in seiner Gesamtheit zu testen;
- Installationstest, wobei es darauf ankommt, das System auf allen vorgesehenen Plattformen zu testen;
- Performanztest, um schließlich die Belastbarkeit und Robustheit des Systems unter einer schweren Transaktionslast mit einer großen Datenmenge zu prüfen.

Bourne empfiehlt weiter, Client/Server-Systeme inkrementell zu entwickeln und zu testen, d.h. in regelmäßigen Abständen Komponenten neu zusammenzustellen (Builds) und neu zu testen. Natürlich müsste es eine eigene Testgruppe geben, die für alle Testaktivitäten nach dem Unit-Test verantwortlich ist. Bis zum Integrationstest ist der Test im privaten Zuständigkeitsbereich des Entwicklers. Ab dem Integrationstest ist der Test im öffentlichen Zuständigkeitsbereich des Testers. Hier werden alle Fehler offiziell gemeldet und verfolgt. Die Verteilung der Software verlangt

nach einer Verteilung der Testaktivitäten und einer Verteilung des Fehlermeldesystems.
In einem anderen Buch mit dem Titel „Testing Client/Server Applications“ [Gog93] beschreibt Patricia Goglia vier Phasen für den Test:

- die Planungsphase,
- die Entwurfsphase,
- die Ausführungsphase und
- die Wartungsphase.

In der *Planungsphase* werden Testaufgaben und Testergebnisse identifiziert, die Testressourcen zugeteilt, die Testwerkzeuge bereitgestellt, die Testziele gesetzt und die Verantwortlichkeiten festgelegt.
In der *Entwurfsphase* werden die Testzyklen bestimmt, die Testfälle spezifiziert, die Testprozeduren implementiert und die Testdaten generiert. Für den Test der Geschäftslogik empfiehlt Goglia die Entscheidungstabellentechnik zur Ermittlung der Testfälle. Für die Implementierung der Testprozeduren werden Testfall/Modul-Matrizen empfohlen, und für die Generierung der Datenbanken schlägt die Autorin eine Testfall/Datenbankmatrix vor. Schließlich wird im Hinblick auf die Verteilung eine Matrix der Netzknoten und Testprozesse erarbeitet. Auf diese Testentwurfstechniken wird später bei den Methoden eingegangen. Hier soll nur darauf hingewiesen werden, dass der Testentwurf für Client/Server-Systeme viel systematischer und detaillierter sein muss als der Testentwurf monolithischer Systeme.
In der *Ausführungsphase* werden die Testumgebung aufgebaut, die Testprozesse gestartet, die Testabläufe verfolgt, die Testzustände registriert, die Testergebnisse validiert und Fehlverhalten protokolliert. Die Autorin legt besonderen Wert auf den Paralleltest multipler Pfade, um die Synchronisierung der Pfade zu prüfen. Hier komme es darauf an, die vielen Ausnahmefälle einer Client/Server-Architektur auszulösen, denn wenn auf die Ausnahmefälle gezielt wird, ergeben sich die Normalfälle von selbst.
In der *Wartungsphase* geht es hauptsächlich um den Regressionstest. Da Goglia für eine inkrementelle Entwicklung plädiert, beginnt die Wartung bereits in der Entwicklung. Immer wenn neue Inkremente geliefert werden, müssen die alten wieder bestätigt werden. Um nicht alles völlig neu testen zu müssen, empfiehlt die Autorin eine Impaktanalyse. Diese soll zeigen, welche bestehenden Daten und Funktionen von den neu hinzugekommenen Funktionen betroffen sind. Hier sei von den Modul- und Komponentenschnittstellen auszugehen. Die inkrementelle Entwicklung hat zwar ihre Vorteile, aber für den Test stellt sie eine neue Herausforderung dar.
Client/Server-Systeme haben also ähnliche Testphasen wie monolithische Systeme, man kann sogar die ANSI-Norm als groben Rahmen übernehmen (Abbildung 2.4). Dennoch, wie die beiden zitierten Bücher zum Ausdruck bringen, gibt es innerhalb

der Phasen viele spezielle Aktivitäten, die Client/Server-spezifisch sind. Vor allem ist es die Komplexität verteilter Systeme, die es erforderlich macht, Testverfahren vorher detailliert auszuarbeiten und nachher streng einzuhalten. Den zusätzlichen Aufwand muss man in Kauf nehmen. Dies ist der Preis für die Vorteile solcher Systeme [Eli94].

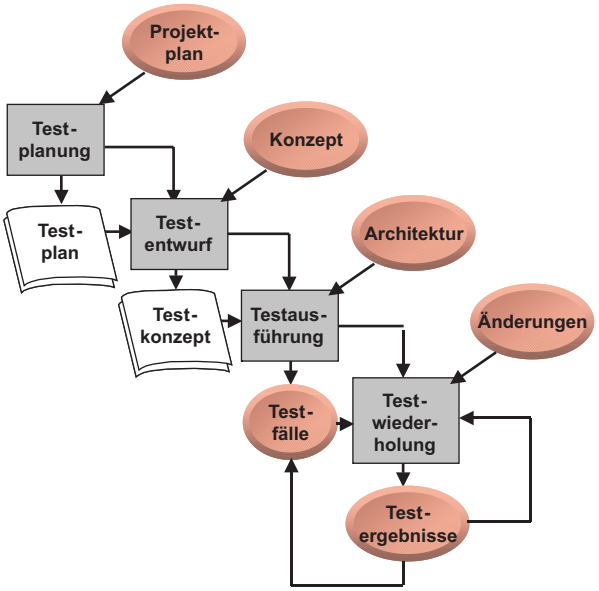


Abbildung 2.4 Testverfahren für verteilte Systeme

2.3 Testverfahren für objektorientierte Systeme

Objektorientierte Systeme sind in der Regel verteilte Systeme. Daher erben sie die Eigenschaften und Problematik eines Client/Server-Systems. Sie sind auch Software-Systeme im Allgemeinen, sodass sie auch von dem allgemeingültigen Software-Testverfahren erben. Daraus folgt, dass ein Testverfahren für objektorientierte Systeme ein spezieller Fall eines Testverfahrens für verteilte Systeme ist, und dass das Testverfahren für verteilte Systeme ein spezieller Fall des allgemeingültigen Software-Testverfahrens ist.

2.3.1 Vererbung der Testphasen

Von der ANSI-Norm für Software-Testverfahren im Allgemeinen erbt das objektorientierte Testverfahren die sechs Phasen (Abbildung 2.5)

- Testplanung,
- Testentwurf,
- Testfallspezifikation,
- Testdurchführung,
- Testauswertung und
- Testwiederholung.

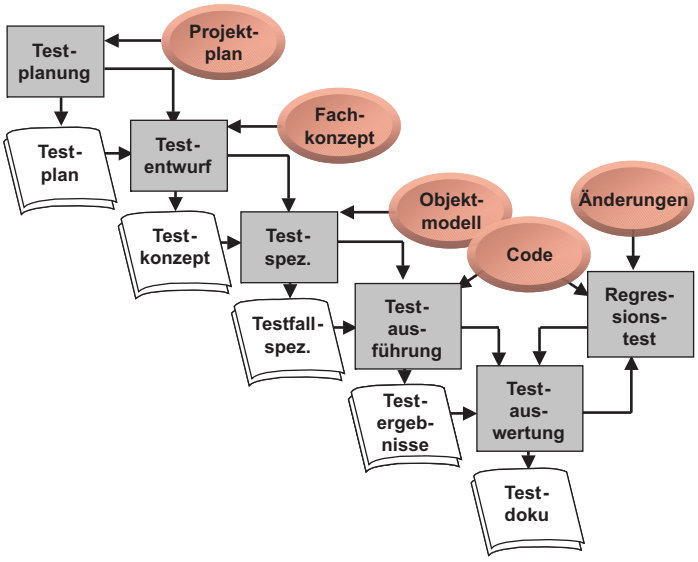


Abbildung 2.5 Testverfahren für verteilte OO-Systeme

2.3.2 Vererbung der Teststufen

Von dem Client/Server-Test werden die vier Teststufen

- Klassentest (Modul- bzw. Unit-Test),
- Integrationstest,
- Systemtest und
- Abnahmetest

geerbt [Sie96].
Der Unit-Test teilt sich bei objektorientierten Systemen in einen Klassentest und einen Modultest, wobei ein Modul z.B. eine Klassenhierarchie oder eine kleine

Menge sehr eng zusammengehöriger Klassen (cluster) ist. Der Integrationstest testet die Interaktion zwischen Modulen innerhalb einer Komponenten-Modulintegration sowie die Interaktion zwischen Komponenten – die Komponentenintegration. Systemtest und Abnahmetest werden unverändert übernommen. Abbildung 2.6 zeigt die entsprechenden Testobjekte.

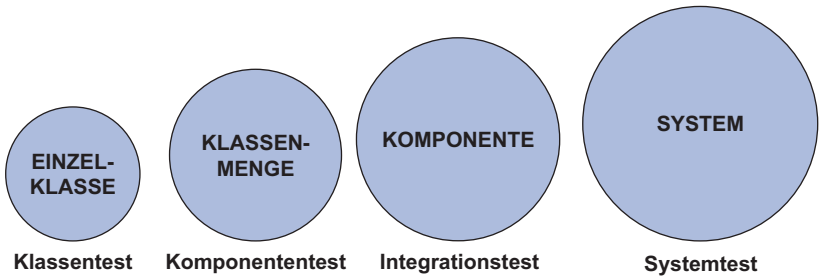


Abbildung 2.6 Testobjektarten

2.3.3 Bestimmung der Testaufgaben

Aus dem Kreuzprodukt der vier Teststufen und der sechs Testphasen in einer Testaktivitätenmatrix (Abbildung 2.7) gehen 4 x 6 bzw. 24 einzelne Testaufgaben hervor, denn jede Teststufe muss geplant, konzipiert, spezifiziert, implementiert, durchgeführt, ausgewertet und wiederholt werden.

Natürlich kann man Teststufen und auch Testphasen weglassen, um Zeit und Aufwand zu sparen, vor allem bei kleineren Projekten. Dennoch soll dies bewusst geschehen – d.h. mit der klaren Aussage: „Wir verzichten auf einen Klassentest!“ Oder: „Wir verzichten auf ein Testkonzept, weil ...“ – und nicht einfach aus Ignoranz vergessen werden. Aufgrund der Testaktivitätenmatrix soll es jedem klar werden, welcher Aufwand für einen systematischen objektorientiertem Test betrieben werden muss [ArFu94].

Testphasen / Testarten	Planung	Entwurf	Spezifikation	Ausführung	Auswertung
Klassentest	1	5	9	13	17
Integrationstest	2	6	10	14	18
Systemtest	3	7	11	15	19
Abnahmetest	4	8	12	16	20

Abbildung 2.7 Testaktivitätenmatrix

2.4 Phasen des objektorientierten Tests

2.4.1 Testplanung

Der Test eines objektorientierten Systems beginnt wie der Test aller Systeme mit einer Testplanung. Hier werden die Testziele gesetzt, die Testaktivitäten geplant, die Testergebnisse genannt und die Testressourcen zugewiesen (Abbildung 2.8). Die Planung unterscheidet sich nur um Nuancen von der Planung konventioneller Tests. Auch die Ziele sind weitgehend identisch. Es sind hauptsächlich die Testgegenstände, die sich unterscheiden.

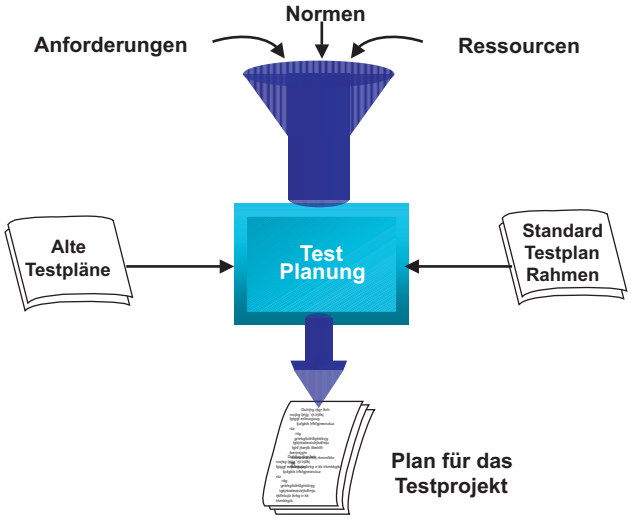


Abbildung 2.8 Testplanungsphase

2.4.2 Testentwurf

In der zweiten Phase, dem Entwurf des objektorientierten Tests, hat die Verteilung der Software einen größeren Einfluss auf die Gestaltung der Testprozesse. Nicht selten werden in einem System mehrere Plattformtypen miteinander verbunden, z.B. Java-Clientprogramme mit C++-Serverprogrammen und Object-COBOL-Hostprogrammen (Abbildung 2.9). Für jede Plattform werden andere Testprozesse konzipiert. Dem Integrationstest kommt eine besondere Bedeutung zu. Daher muss das Konzept für den Test der Schnittstellen besonders sorgfältig ausgearbeitet werden. Für den Systemtest müssen nicht nur die Ein- und Ausgaben des zu testenden Systems, sondern auch alle Daten-Importe von und -Exporte zu fremden Systemen bedacht werden.

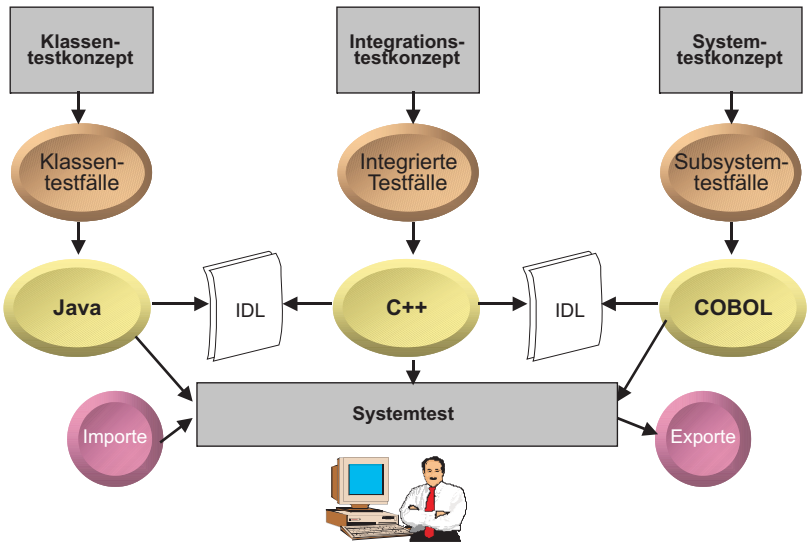


Abbildung 2.9 Testentwurfsphase

2.4.3 Testfallspezifikation

Die Testfallspezifikation als dritte Phase ist auch beim Test objektorientierter Systeme die aufwändigste. Hier kommt es darauf an, aus allen möglichen Sichten auf das System Behauptungen bzw. Zusicherungen über das Verhalten des Systems zu formulieren. Es stellt sich immer die Frage „Was macht die Software, wenn Dies oder Das geschieht?“, d.h. es sind sämtliche Ereignisse zu identifizieren und zu jedem Ereignis die erwarteten Ergebnisse vorauszusagen.

Dazu muss der Testfallspezifizierer auf die Anwendungsfälle, die Geschäftsregeln, die Schnittstellendefinitionen und die Zustandsdiagramme der einzelnen Klassen

bzw. Objekte zurückgreifen (Abbildung 2.10). Daraus ergeben sich für komplexe verteilte Systeme mehrere hundert bzw. mehrere tausend Testfälle. Das Besondere an verteilten objektorientierten Systemen ist gerade dies, dass die Anzahl der Testfälle sich verdreifacht, einmal wegen der Verteilung und nochmals wegen der Objektorientierung.

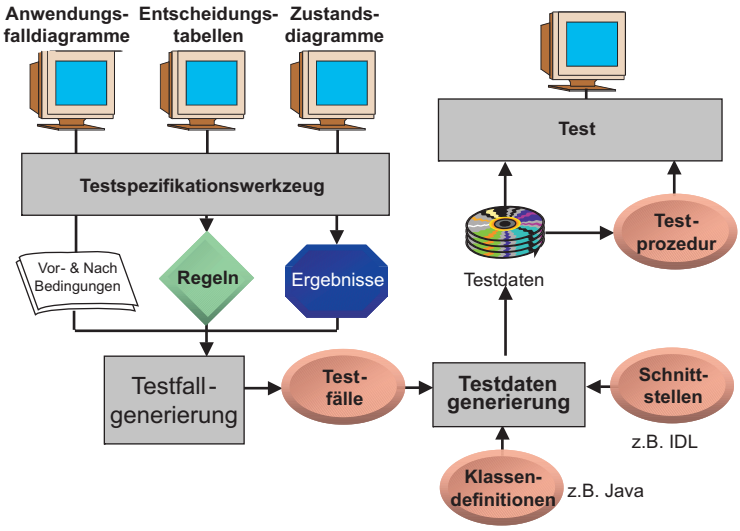


Abbildung 2.10 Testspezifikationsphase

2.4.4 Testdurchführung

Die vierte Phase, die Testdurchführung, beinhaltet drei Unterphasen:

- Testprozedurerstellung
- Testumgebungsaufbau
- die Testausführung selbst

Die Testprozedurerstellung hängt weitgehend von den Testwerkzeugen ab, die zur Verfügung stehen (Abbildung 2.11). Mit den richtigen Werkzeugen ist es möglich, Testprozeduren bzw. Testskripte automatisch aus den Testfällen zu generieren, vorausgesetzt, die Testfälle sind formal beschrieben. In diesem Fall ist der Aufwand für die Erstellung der Testprozeduren gering. Mit anderen Werkzeugen wird es nur möglich sein, die Testskripte semiautomatisch zu produzieren, d.h. Tester oder Entwickler müssen die generierten Skripte manuell ergänzen. Ohne Werkzeuge ist diese Phase äquivalent zu der manuellen Codierung der Software. Für jede Funktion in der Software muss eine entsprechende Testprozedur geschrieben werden. Dies

führt zu einer Verdoppelung des Codieraufwandes. Deshalb ist eine automatische Generierung der Testprozeduren eine unbedingte Voraussetzung für den Test komplexer objektorientierter Software.

Beim Testaufbau ist die Systemverteilung der Hauptkostentreiber. Da es bei objektorientierter Software oft um verteilte Systeme geht, muss für jede betroffene Plattformart eine eigene Testumgebung mit Testtreiber (driver), Teststellvertreter (stubs), Testdatenbanken usw. eingerichtet werden. Darüber hinaus muss es eine Testumgebung für das ganze System geben, in der alle Rechner und Betriebssysteme miteinander gekoppelt sind bzw. in der die Middleware wie Java Beans, Object Request Broker, DCOM oder Component Broker installiert sind.

Die eigentliche Testausführung beinhaltet den Test der Klassen, Module, Komponenten und Teilsysteme. Der Test der Komponenten und Teilsysteme unterscheidet sich kaum von dem Test verteilter Systeme im Allgemeinen. Die Problematik ist die gleiche. Beim Test der Klassen und Module hat jedoch die Objektorientierung eine eigene Problematik, die den Test erschwert. Vererbung und Polymorphie machen es schwer, einzelne Klassen oder Zweige einer Klassenhierarchie unabhängig zu testen. Die Verweise einer Klasse auf andere übergeordnete Klassen zwingen den Tester, diese übergeordneten Klassen mitzutesten. Die dynamisch gebundenen Operationsaufrufe zwingen den Tester, alle potenziellen Aufrufziele zu simulieren. Somit ist auch der Unit-Test objektorientierter Software um einiges komplizierter als der Unit-Test prozeduraler Software.

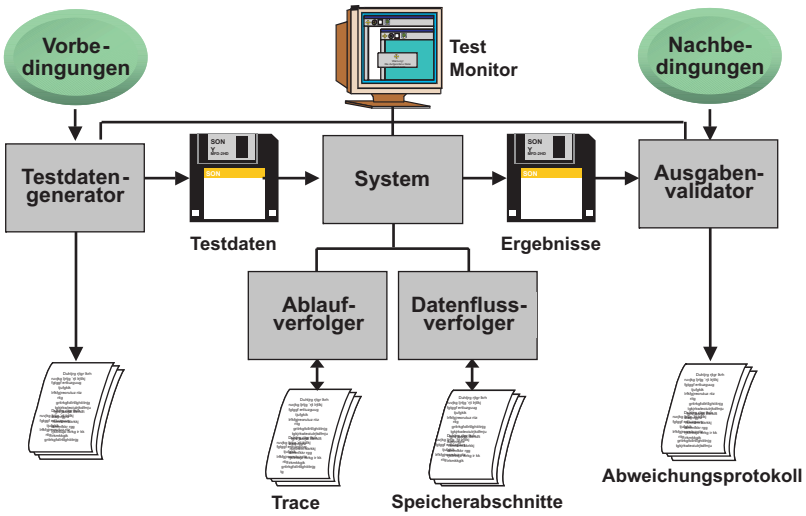


Abbildung 2.11 Werkzeuge in der Testausführung

2.4.5 Testauswertung

Auch die fünfte Phase, die Testauswertung, stellt im Falle objektorientierter Software eine größere Herausforderung dar als bei konventionellen Systemen (Abbildung 2.12). Die Dynamik der Erzeugung und Zerstörung von Objekten erschwert das Festhalten und Kontrollieren der Zwischenzustände. Das Vorhandensein vieler unbenutzter Funktionen erschwert die Testüberdeckungsmessung, und die hohe Anzahl der Interaktionen zwischen Objekten bzw. die Kollaborationen erschweren die Verfolgung des Ablaufs. Hinzu kommt die Komplexität der Schnittstellen. Die Protokollierung der Abläufe und Zwischenergebnisse zwecks der Testauswertung gestaltet sich also bei objektorientierter Software um einige Schwierigkeitsgrade höher als bei vergleichbarer prozeduraler Software. Ohne adäquate Werkzeuge ist sie kaum zu bewältigen [MgKo94].

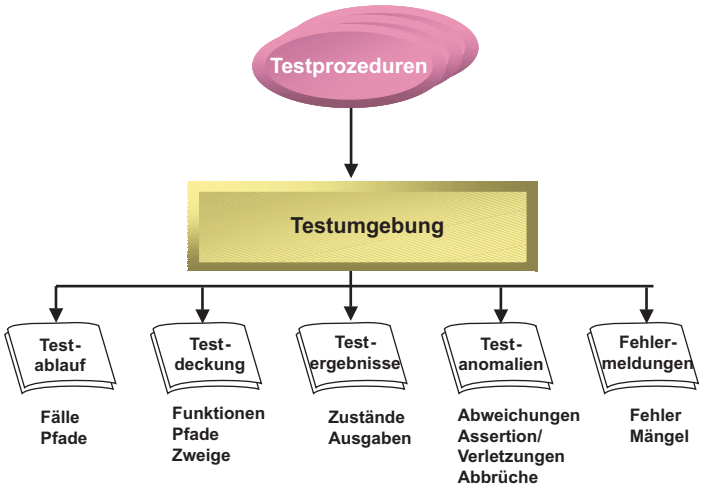


Abbildung 2.12 Testauswertungsphase

2.4.6 Testwiederholung

Die Testwiederholung als sechste Phase ist mehr eine Aktivität, die sich immer wiederholt. Sie ist für den Softwaretest das, was die Wartung für die Softwareentwicklung ist: eine endlose Nachbesserung. Parallel zur Nachbesserung des Konzepts und des Codes wird hier auch der Test nachgezogen. Jede Erweiterung oder Änderung der bestehenden Funktionalität, jede Korrektur und jede Optimierung bzw. Sanierung zieht eine entsprechende Fortschreibung des Tests mit sich. Falls die Änderung sich auf eine einzige Klasse bezieht, wird der entsprechende Klassentest mit verändert. Wenn die Funktionalität oder die Schnittstelle einer Komponente sich ändert, wird der Komponententest angepasst. Insofern, als die Benutzungsober-

fläche, die Systemschnittstellen oder die Datenbankstrukturen verändert werden, wird der Systemtest fortgeschrieben. Testwiederholung ist somit eine Folge der Software-Weiterentwicklung und dauert so lange, wie die Software noch im Wandel begriffen ist. Dafür sind besondere Regressionstesttechniken und vor allem spezielle Regressionstestwerkzeuge erforderlich.

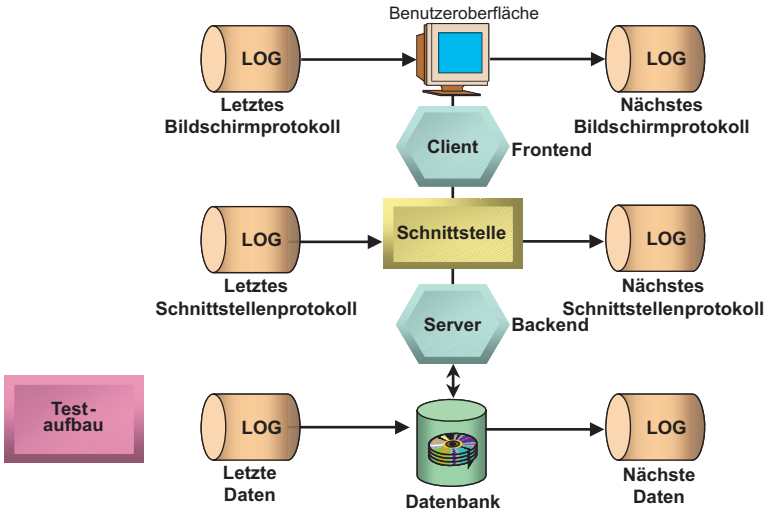


Abbildung 2.13 Testwiederholungsphase

2.5 Ergebnisse des objektorientierten Tests

2.5.1 Der Testplan

Aus den oben genannten Phasen mit ihren Einzelaktivitäten gehen mindestens 8 Ergebnistypen hervor (Abbildung 2.14). Der *Testplan* wird auf die Besonderheiten der Verteilung und der Objektorientierung eingehen, d.h. auf die Phasen, Aufgaben, Objekte, Ergebnisse und Abnahmekriterien des verteilten objektorientierten Tests.

2.5.2 Das Testkonzept

Das *Testkonzept* muss auf die Problematik des Klassentests mit Vererbung und Polymorphie, sowie auf die des Integrationstests mit verteilten Komponenten eingehen. Die geplanten Testprozesse müssen also sowohl der Verteilung als auch der Objektorientierung gerecht werden. Daraus folgt, dass es mehrere verschiedenartige Testprozesse mit unterschiedlichen Szenarien geben wird.

2.5.3 Die Testfallspezifikation

Die *Testfallspezifikation* muss neben den üblichen Systemtestfällen, die sich aus der Systemoberfläche und externen Schnittstellen ergeben, auch Integrationstestfälle auf der Basis der internen Schnittstellen und Modul- und Klassentestfälle beinhalten, die auf einzelne Funktionen bzw. Methoden zielen. Das Ergebnis wird eine Hierarchie von Testfällen sein, die wie Objekte voneinander erben. Die Basistestfälle bilden die Klassentestfälle. Aus ihnen werden Modultestfälle abgeleitet. Die Integrationstestfälle erben die Modultestfälle und ergänzen sie um die Komponententestfälle für den Test der Komponentenschnittstellen. Die Systemtestfälle umfassen alle bisherigen Testfälle, plus die Testfälle, die sich aus dem Nutzungsprofil des Systems ergeben, plus jene Testfälle, die für den Nachweis der Robustheit, Sicherheit und Performance des Systems notwendig sind.

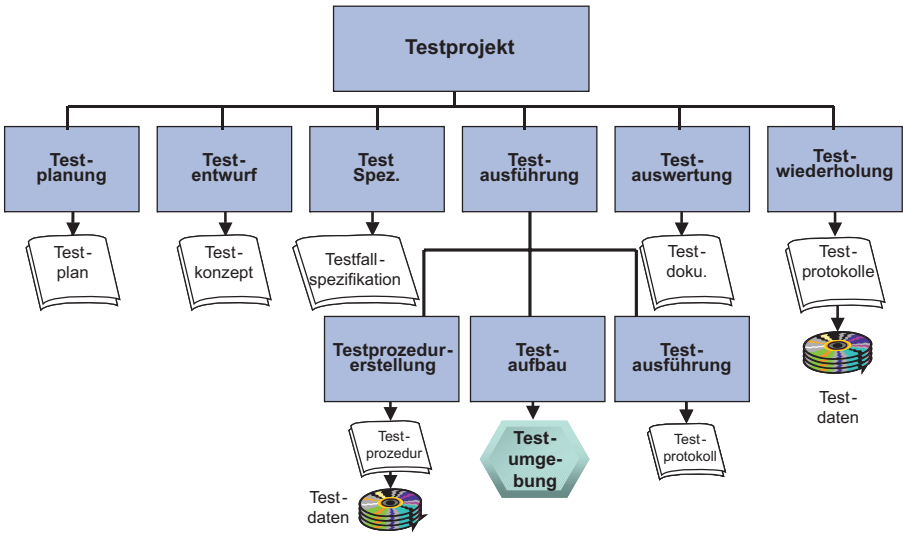


Abbildung 2.14 Ergebnisse des objektorientierten Tests

2.5.4 Die Testprozeduren

Die *Testprozeduren* sind compilierbare oder interpretierbare Source-Code-Skripte mit einer formalen Syntax. Sie müssen bei einem objektorientierten System einzelne Funktionen bzw. Methoden in einer Klasse sowie Funktionsketten über mehrere Objekte hinaus auslösen. Es wird auch extra Testprozeduren für den Schnittstellentest über die Middleware geben.

2.5.5 Die Testumgebung

Die Umgebung für den Test verteilter objektorientierter Systeme ist eine Mischung aus Hardware, Netzen und Software. Zur Hardware zählen mehrere PC-Arbeitsplätze, ein oder mehrere Serverrechner und eventuell auch ein Mainframe-Rechner. Zu den Netzen gehört die Verkabelung der Testrechner mit den entsprechenden Netztreibern und Protokollen. Zur Software gehören die Netzwerksoftware, die Betriebssysteme, die Datenbanksysteme und die Middleware. Hinzu kommen die Kompiler, Editoren und Debugger, sowie andere Entwicklungshilfen. Schließlich gehören auch die Testdatenbanken und Testwerkzeuge zur Testumgebung.
Dies alles bereitzustellen, zu installieren und auszuprobieren, ist an sich eine aufwändige Angelegenheit, die einige Personenmonate in Anspruch nehmen kann. Es muss aber geschehen, und zwar vor dem ersten Test der Anwendungssoftware. Zu empfehlen ist daher, mit dieser Arbeit gleich nach der Testplanung anzufangen.

2.5.6 Die Testprotokolle

Die Ergebnisse einer Testausführung sind die Testprotokolle, in erster Linie die Testablaufprotokolle, Testzustandsprotokolle und Testüberdeckungsprotokolle. Die Ablaufprotokolle dokumentieren, welche Funktionen in welchen Objekten in welcher Reihenfolge ausgeführt werden. Die Zustandsprotokolle dokumentieren den Zustand der erzeugten Objekte bzw. der Schnittstellen zu verschiedenen Zeitpunkten, z.B. gleich nach der Konstruktion, nach jeder Veränderung und unmittelbar vor der Destruktion. Die Überdeckungsprotokolle dokumentieren, welche Zweige, Methoden, Operationsaufrufe und Objekte wie oft ausgeführt wurden.

2.5.7 Die Testberichte

Die *Testberichte* für die Auswertung eines objektorientierten Systems sind wiederum ähnlich gestaltet wie die Berichte über einen konventionellen Test. Zum Ersten gibt es die Fehlerberichte an die Adresse der Entwickler, welche die Fehler verursacht haben. Zum Zweiten gibt es Berichte über die Testmetriken mit Fehlerrate, Überdeckungsrate und Vollständigkeitsgrad. Zum Dritten folgt der Testabschlussbericht mit einer Zusammenfassung aller Testereignisse und -ergebnisse, einer Fehlerstatistik, einer Beurteilung der Testüberdeckung und einer Aussage über die erreichte Qualität relativ zur geplanten Qualität [Sne96]. Die Objektorientierung hat keinen Einfluss auf die Berichterstattung, es sei denn, dass die Berichte deshalb umfangreicher werden.

2.6 Verantwortlichkeit für den objektorientierten Test

In der alten monolithischen Datenverarbeitungswelt war die Verantwortung für den Test zwischen dem Entwickler und dem Endanwender bzw. Auftraggeber geteilt. Der Entwickler, der in der Rolle eines Organisationsprogrammierers auftrat, machte alles von der Analyse bis zur Übergabe – er analysierte das Problem, entwarf eine fachliche Lösung, die er dann in eine technische Lösung umsetzte, und implementierte die Programme. Danach hat er sie mehr oder weniger getestet, ehe er sie an den Auftraggeber übergab, der sie nochmals aus seiner Sicht testete. Wenn das System zur Zufriedenheit der Benutzer lief, wurde es übernommen.

Somit hat es im Prinzip immer zwei Tests gegeben, einen Entwicklertest und einen Benutzertest. Diese Arbeitsteilung setzte jedoch ein hohes Engagement und eine ebenso große Geduld seitens der Benutzer voraus. Diese waren deshalb vorhanden, weil es in der Regel ein 1:1-Verhältnis zwischen der Fachabteilung und dem Produkt gab. Die Systeme wurden speziell für einen Kunden gefertigt und dieser Kunde war bereit, sich am Test zu beteiligen.

In der modernen verteilten Datenverarbeitungswelt werden DV-Systeme aus Kostengründen für mehrere Fachabteilungen eines Konzerns entwickelt. Diese Anwender haben oftmals ganz unterschiedliche Interessen. In multinationalen Firmen sind sie über mehrere Länder verstreut und können sich kaum verständigen, geschweige denn sich über etwas so Kompliziertes wie Softwaresysteme einigen. Den einen Auftraggeber, der alles bezahlt, gibt es leider nicht mehr. Dieser Zustand wird durch den Begriff „verteilt“ auch impliziert. Außerdem haben die heutigen Anwender weder die Zeit noch die Geduld noch die technische Kompetenz, um die Einsatzfähigkeit komplexer Client/Server- oder Intranet-Anwendungen zu testen. Folge ist, dass der Benutzertest entfällt. An die Stelle der alten Anwender tritt jetzt eine betriebliche Testgruppe, die stellvertretend für die poteniellen Anwender testet. Der Benutzertest ist jetzt zum Systemtest geworden und wird von der Testgruppe ausgeführt.

Diese Testgruppe setzt sich zusammen aus Technikern und Fachgebietsexperten. Die einen testen die vielen technischen Funktionen, bzw. die Form, die anderen testen die fachliche Funktionalität bzw. den Inhalt. Zusammen bilden sie ein Team, das die Kompetenz besitzen muss zu entscheiden, ob ein Softwareprodukt freigabereif ist. Die Mitglieder solcher Teams werden inzwischen weltweit als Testingenieure bezeichnet. Ihre Arbeit besteht darin, Testkonzepte zu verfassen, Testszenarien zu planen, Testfälle zu spezifizieren, Testskripte zu kodieren, Testumgebungen aufzubauen, Tests durchzuführen, Testergebnisse auszuwerten und Fehler zu melden. Ihre Produktivität wird an der Anzahl ihrer Testfälle, an der Anzahl der gemeldeten Fehler und der erreichten Testüberdeckung gemessen. Die Testaktivitäten dieser Gruppe finden neben den Entwicklungsaktivitäten statt. Sie bilden sozusagen ein Parallelprojekt.

Dies soll jedoch nicht bedeuten, dass die Entwickler nicht mehr testen müssen. Im Gegenteil, die Entwickler sind nach wie vor für den Unit-Test und zumindest Teile des Integrationstests zuständig. Sie sollen nur gut getestete Komponenten an die Testgruppe übergeben. Wie weit die Klassen und Komponenten von den Entwicklern vorher zu testen sind, ist eine Frage betrieblichen Vereinbarungen. Auf jeden Fall sind die Teststufen folgendermaßen aufzuteilen (Abbildung 2.15):

- Klassentest → Entwickler
- Integrationstest → Entwickler/Testgruppe
- Systemtest → Testgruppe
- Abnahmetest → Pilotanwender/Testgruppe

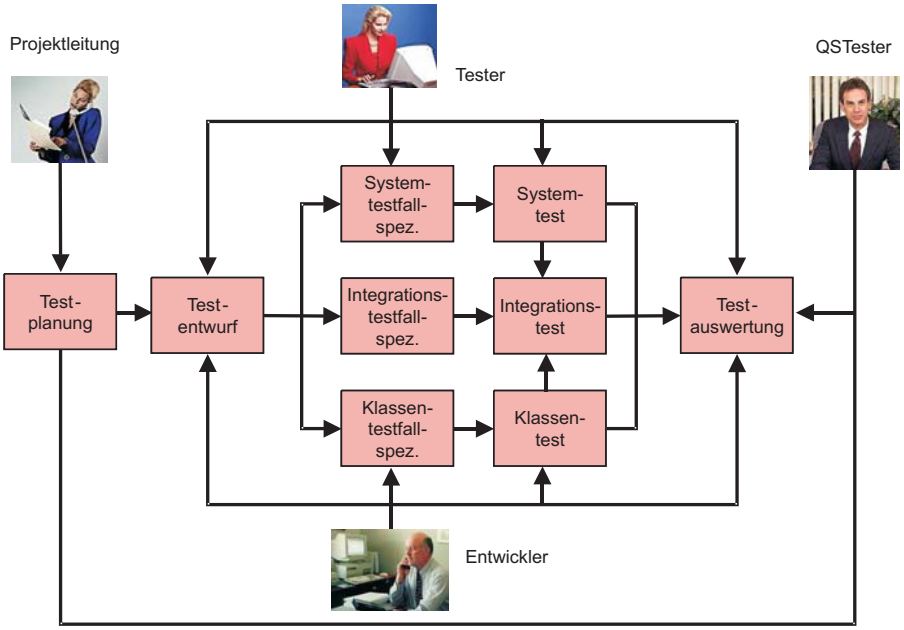


Abbildung 2.15 Testverantwortlichkeiten

2.7 Werkzeuge für den objektorientierten Test

Zu den bisherigen Testwerkzeugen für den konventionellen Test wie Datengeneratoren, Datenabgleichsvalidatoren, statischen und dynamischen Analysatoren kommen zusätzlich Tools für den Test der graphischen Oberflächen und den Test der

Interaktion zwischen verteilten Komponenten. Capture/Replay-Werkzeuge spielen eine wichtige Rolle beim Test der Systemoberfläche. Es muss möglich sein, den Benutzerdialog jederzeit zu wiederholen. Schnittstellensimulatoren sind erforderlich, um die Interaktionen zwischen verteilten Komponenten zu testen. Es muss möglich sein, Eingangsnachrichten im Netz abzusetzen und Ausgangsnachrichten im Netz abzufangen. Schließlich benötigt man einen Klassentestrahmen für den Klassen- und Modultest. In diesen Rahmen gehören ein Klassentesttreiber, der gezielt Methoden aufruft, und die Klassenteststubs, die aufgerufene Methoden in fremden Objekten simulieren. Hier beim Unittest ist der Bedarf an speziellen sprachbezogenen Testwerkzeugen am größten [KGH+95].

Auf die Problematik der Testwerkzeuge wird im Kapitel 11 tiefer eingegangen. Es genügt hier, zu erwähnen, dass der Test verteilter objektorientierter Software-Systeme einen viel höheren Grad an Automatisierung erfordert, als es bisher der Fall war. Das bedeutet, dass die Produzenten objektorientierter Systeme viel mehr in ihre Infrastruktur investieren müssen.

2.8 Der iterative Testprozess

Objektorientierte Systeme werden fast immer inkrementell entwickelt. Am Anfang werden die allgemeingültigen Basisklassen entwickelt oder bereitgestellt. Anschließend werden applikationsbezogene Basisklassen entwickelt. Danach erst werden projektspezifische Klassen implementiert, und zwar komponentenweise in der Reihenfolge ihrer Dringlichkeit. Es kommt darauf an, in Zyklen von maximal 6 Monaten neue Releases herauszugeben, um die Endanwender bzw. die Kunden bei der Stange zu halten und sie an der Entstehung ihres Produkts zu beteiligen ([Bec99], vgl. Kapitel 12).

Die Gründe für eine inkrementelle Entwicklung dürften inzwischen für alle klar sein. Die einschlägige Literatur zur Objektorientierung hat sie mehrfach angeführt. Es fragt sich nur, welche Folgen diese Vorgehensweise für den Test hat. Eine Folge ist die Bedeutung des Regressionstests. Ein Regressionstest ist ein wiederholter Test mit zusätzlichen Testfällen. Nach der ersten Freigabe der ersten Komponenten eines Software-Systems sind alle Testprozesse auch Regressionstestprozesse. Man testet nicht nur die neuen Komponenten, sondern auch die alten Komponenten mit. Dies könnte bedeuten, dass alle bisherigen Testfälle zu wiederholen sind, oder es bedeutet, eine Untermenge der bisherigen Testfälle nochmals durchzuführen. Im letzteren Fall muss die Testgruppe ziemlich genau wissen, welche Funktionen und Daten in den bereits getesteten Komponenten, Klassen und Datenbanken von den neuen Funktionen betroffen sind.

Dieses genaue Wissen setzt eine ausführliche und aktuelle technische Dokumentation voraus. In der Dokumentation muss erkennbar sein, welche Klassen von wel-

chen anderen Klassen erben, welche Funktionen welche anderen Funktionen mit welchen Parametern aufrufen, welche Funktionen von welchen anderen Funktionen aufgerufen werden, welche Daten eine Funktion ein Modul oder eine Komponente benutzt und welche Daten von welchen Funktionen, Modulen und Komponenten benutzt werden. Die Beziehungen zwischen fachlichen Vorgängen und technischen Methoden müssen auch erkennbar sein, um eine Auswahl der bisherigen Testfälle zu treffen [Pos94].

Eine Aufbewahrung und Fortschreibung aller bisherigen Testprozeduren ist für das iterative Testen selbstverständlich, ebenso die Sicherung aller bisherigen Testdatenbanken. Iteratives Testen setzt also eine geregelte Testdaten- und Testprozedurverwaltung voraus. Wer dies nicht beachtet, wird sich beim iterativen Test schwer tun.

Das iterative Testen ist ein zyklisches Verfahren, bei dem der Test in Zyklen wiederholt und ergänzt wird. In anderen Worten: Der Test ist quasi als „Parallelprojekt“ mit dem Entwicklungsprozess zu verzahnen und fortzuschreiben (Abbildung 2.16). Nach jeder Iteration wird der Test umfangreicher. Deshalb muss er auf den bisherigen Testdaten und Testprozeduren aufsetzen. Die Anzahl der Testfälle wird auch von Zyklus zu Zyklus zunehmen. Dieses Wachstum muss deshalb von vornherein geplant und gesteuert werden. Denn nur so wird es gelingen, den Test verteilter objektorientierter Systeme in den Griff zu bekommen. Der Eindruck, dass mit der Objektorientierung alles einfacher und billiger wird, ist eine Täuschung. Im Gegenteil, es bedeutet letzten Endes nur eine Verschiebung der Probleme von der Entwicklung auf den Test. Die Probleme werden dadurch nicht weniger, sie tauchen eben woanders auf [Bin94].

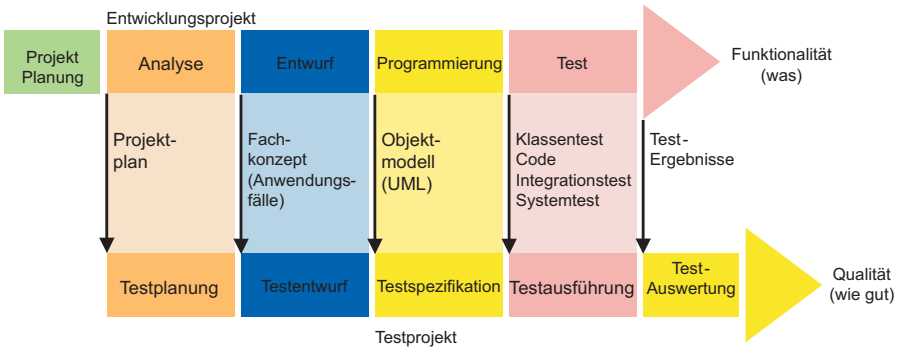


Abbildung 2.16 Test als Parallelprojekt

3

Objektorientierte Testplanung

Standort der Planung im Testprojekt
Organisation des Testprojekts
Inhalt des Testplans
Auswirkung der Objektorientierung auf die Testplanung
Auswirkung der Verteilung auf die Testplanung
Ein Testplan für den verteilten Kalender

Inhaltsübersicht Kapitel 3

3 Objektorientierte Testplanung

3.1 Standort der Planung im Testprojekt

Der Test als eigenständiges Projekt neben dem Entwicklungsprojekt bedarf ebenfalls einer eigenen Planung [KoPo99]. Dies ist auch der Gedanke hinter dem ANSI-Standard 829 und dem ISO Standard 12207. Die Testplanung soll direkt mit der Anforderungsermittlung beginnen und bis zum Ende der Systementwurfsphase fertig sein, d.h. der Testplan soll stehen, ehe der erste Code geschrieben ist (Abbildung 3.1).

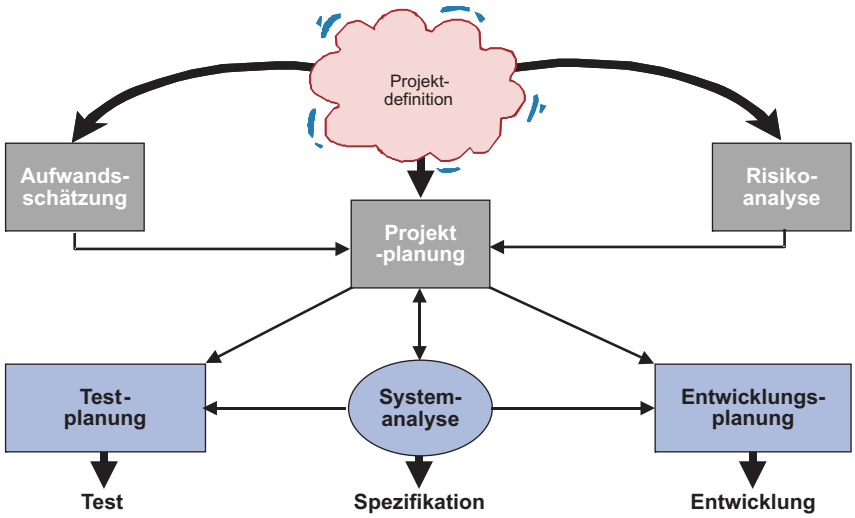


Abbildung 3.1 Standort der Testplanung

Das soll nicht heißen, dass danach nichts mehr geändert wird. In einer modernen Software-Entwicklung ist alles im Fluss, auch der Testplan. Der Plan wird sicherlich noch während der Testdurchführung geändert, wenn man neue Erkenntnisse gewinnt und unerwarteten Problemen begegnet. Deshalb wird es erforderlich sein,

den Testplan wie alle anderen Dokumente fortzuschreiben. Er sollte jedoch bis zur Kodierung der ersten Klassen im vollen Umfang bereit stehen.

3.2 Organisation des Testprojekts

Da der Test als eigenständiges Projekt im Projekt zu behandeln ist, wird das Testprojekt auch einen Projektleiter bzw. eine Projektleiterin haben. Diese Person müsste bereits zu Beginn des Projekts ernannt werden. Denn auch wenn noch nicht alle Mitglieder des Testteams verfügbar sind, wird der Teamleiter schon seine Tätigkeit aufnehmen können, und die erste Tätigkeit besteht darin, den Testplan zu erstellen. Anschließend wird der Teamleiter damit beschäftigt sein, das Testteam zusammenzustellen.

Bei der Zusammensetzung des Testteams ist auf die richtige Mischung zu achten. Neben OO- und Client/Server-Experten braucht man einige Fachexperten, die in der Lage sind, fachliche Testfälle zu spezifizieren. Ein Teammitglied wird auch für die Testwerkzeuge verantwortlich sein müssen. Seine Aufgabe ist es, die Werkzeuge zu installieren, anzupassen und die Tester bei deren Anwendung zu betreuen. Der Testteamleiter und der Testplanmanager sind die Schlüsselfiguren in dem Testteam. Sie tragen die Hauptverantwortung für den Erfolg oder Misserfolg des Testprojekts.

Der Beruf *Testingenieur* hat sich in Amerika schon längst etabliert und breitet sich inzwischen auch in Europa aus. Microsoft hat z.B. in kritischen Projekten einen Tester pro Entwickler, d.h. der Tester ist schon für den Unit-Test verantwortlich [Bor99]. Dies dürfte aber eine Ausnahme sein. In den meisten Software-Entwicklungsbetrieben ist der Entwickler selbst für den Unittest verantwortlich. Das Testteam übernimmt die Verantwortung erst ab dem Integrationstest. Das soll aber nicht heißen, dass das Testteam keinen Einfluss auf die Entwicklung hat. Je früher sich das Testteam einschaltet, desto besser. Die Tester sollten dafür sorgen, dass die Systemarchitektur im Hinblick auf Testbarkeit und Wartbarkeit gestaltet wird [Jun99].

Falls Produkte bereits im Einsatz sind und weiterentwickelt werden, bezieht sich das Testprojekt auf ein Release. Für jedes neue Release wird ein neues Testprojekt angesetzt. Das Testteam bleibt jedoch konstant, d.h. ein Testteam begleitet ein Produkt durch seinen Lebenszyklus von Anfang bis Ende. Wichtig ist, dass die Kontinuität der Testdienstleistung gewährleistet wird.

3.3 Inhalt des Testplans

Der Inhalt des Testplans ist im ANSI-Standard 829 festgelegt [IEEE829]. Er soll insgesamt 16 Abschnitte beinhalten. Der Grad, zu dem die Abschnitte ausgefüllt werden, hängt von der Art und Größe des Projekts ab. Bei größeren Projekten soll-

ten alle Abschnitte sorgfältig verfasst werden. Bei kleinen Projekten werden einige Abschnitte nur ein einziger Satz sein. Die 16 Abschnitte sind:

- Testidentifikation,
- Testprojektbeschreibung,
- Testgegenstände,
- Testziele,
- Testeinschränkungen,
- Teststrategie,
- Testendekriterien,
- Regressionstestkriterien,
- Testergebnisse,
- Testaufgaben,
- Testumgebungsanforderungen,
- Testverantwortlichkeiten,
- Testaufgabenteilung,
- Testzeitplan,
- Testrisiken und Notpläne und
- Genehmigung.

3.3.1 Testprojektidentifikation

In diesem ersten Abschnitt wird der Testplan eindeutig gekennzeichnet. Hier wird ein Name und ein Kennzeichen für das Testprojekt vergeben, welches dieses Projekt von den anderen unterscheidet.

3.3.2 Testprojektbeschreibung

Im zweiten Abschnitt wird das Testprojekt im Bezug zum Gesamtprojekt beschrieben. Nach einer kurzen Beschreibung des Gesamtprojekts wird die Rolle des Testens im Gesamtprojekt definiert. Dazu gehört der Zweck des Testens, der Testhintergund, der Testumfang und Referenzen auf die einschlägige Testliteratur. Dieser Abschnitt dient als Einführung in den folgenden Plan. Ein Beispiel hierzu zeigt Abbildung 3.2.

1. Testplan Titel: Auftragsbearbeitung Entwicklungstest

2. Einführung

2.1. Testziele:

Die Ziele dieses Tests sind:
-bestätigen dass die Auftragsbearbeitung so funktioniert wie vom Anwender erwartet wird.
-nachzuweisen dass die Applikation die ausreichende Robustheit hat, bzw. alle fehlerhaften Eingaben werden abgefangen und alle Bewegungsarten ohne Abbruch durchlaufen.
-prüfen ob die persistenten Objekte in ihrem letzten Zustand gesichert werden.
-verifizieren dass die erzeugten Ergebnisse - Versandaufträge, Rückstellposten, Auftragsprotokolle, Rechnungen, Lieferposten, Lieferaufträge - mit den spezifizierten Sollergebnissen übereinstimmen.
-validieren dass die Applikation problemlos in der Client / Server Zielumgebung läuft.
-sichern dass die qualitativen Anforderungen abgedeckt sind:

Korrektheit	0,995
Zuverlässigkeit	0,985
Sicherheit	0,800
Integrität	0,900
Benutzerfreundlichkeit	0,750

2.2. Testhintergrund

Die Auftragsbearbeitung ist auf Wunsch der Anwender in einer Client / Server Umgebung neu zu entwickeln. Dabei werden die alten Datenbestände und einige Bausteine der alten Hostprogramme wiederverwendet. Desshalb bleibt die Datenbank in der Sprache SQL. Die alten COBOL-Programme werden jedoch zerlegt und als Java Packages neu implementiert.Der Test soll mit der Entwicklung einhergehen. Zunächst werden die technischen Basisklassen, dann die Grundklassen und zum Schluss die Steuerungsklassen entwickelt werden. Nach jeder Stufe sind die fertigen Klassen zu testen.

2.3. Testumfang

Es sind 4 technische Basisklassen, 2 fachliche Basisklassen, 8 Grundklassen und 2 Steuerungsklassen zu testen.

Es sind circa 30 Schnittstellen zu testen

Abbildung 3.2 Testplaneinführung

3.3.3 Testgegenstände

Im dritten Abschnitt des Testplans werden alle zu testenden Objekte identifiziert. In einer objektorientierten Anwendung können dies

- Klassen,
- Schnittstellen,
- Module,
- Komponenten und
- Systeme

sein.
Aus der Systemsicht kommen

- Datenbanken,
- Dateien,
- Oberflächen,
- Anwendungsfälle und
- Geschäftsprozesse

hinzu (Abbildung 3.3).
Es versteht sich, dass zu Beginn eines Projekts nicht alle Testgegenstände erkennbar sind. Sie ergeben sich im Laufe der Entwicklung. Es ist aber möglich, die Testobjektarten zu identifizieren und einige grobe Ausprägungen aufzulisten. Dieser Teil des Testplans wird später fortgeschrieben, wenn immer mehr Testgegenstände im Projekt auftauchen.

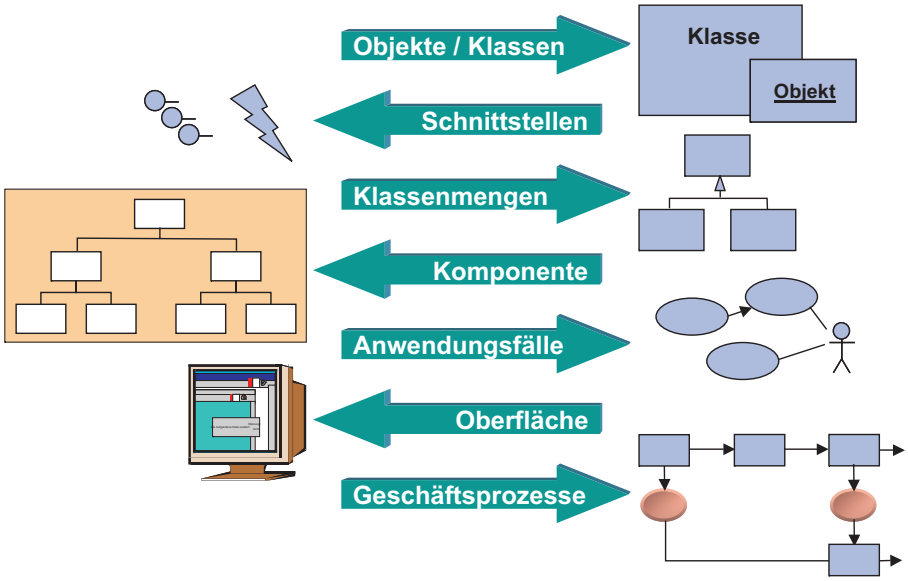


Abbildung 3.3 Identifikation der Testgegenstände

3.3.4 Testziele

Testziele sind unabhängig von der Art der Implementierung. Ob prozedural oder objektorientiert, monolithisch oder verteilt, das System muss gewisse Erwartungen erfüllen. Dazu gehört auf jeden Fall die Funktionalität der Anwendung. Der Test soll nachweisen, dass ein möglichst hoher Prozentsatz der funktionalen Anwendungen erfüllt wird. Dazu müssen alle funktionalen Anforderungen z.B. mit Anwendungsfällen dokumentiert sein.

Abbildung 3.4 Bestimmung der Testziele

Außerdem gilt es zu beweisen, dass auch die nicht-funktionalen Anforderungen im nötigen Grade erfüllt werden. Hierzu gehört die Zuverlässigkeit und Robustheit, die Sicherheit und Integrität sowie nicht zuletzt die Performanz. Für jedes dieser Qualitätsziele gilt es, einen Soll-Erfüllungsgrad zu definieren, und zwar relativ zu einem maximal möglichen und einem minimal akzeptablen Erfüllungsgrad (Abbildung 3.4). Es ist sehr wichtig, hier die Qualitätsziele quantitativ im Sinne von Software-Qualitätsmetriken z.B. nach ISO-Standard 9126 für Software-Produktbewertung zu erfassen [ISO9126].

3.3.5 Testeinschränkungen

Testeinschränkungen umreißen die Grenzen des Tests. Hier wird darauf verwiesen, welche Funktionen nicht zu testen sind, welche Mittel nicht zur Verfügung stehen und welche Qualitätsziele nicht erreicht werden können. Einschränkungen ergeben sich aus größtenteils wirtschaftlichen Zwängen. Die größte Einschränkung ist das Geld. Das Testbudget sollte hier zu Beginn des Testprojekts fixiert sein. Es dürfte zwischen 30% und 60% des Gesamtprojektbudgets betragen, je nachdem, wie viel Software wiederverwendet wird. Falls die Wiederverwendungsrate bzw. der Anteil vorgefertigter Klassen und Komponenten hoch ist, wird der Testanteil eher bei 60% eines insgesamt kleineren Budgets liegen. Falls die Wiederverwendungsrate niedrig ist, d.h. alles von Grund auf neu kodiert werden muss, wird der Testanteil eher bei 30% eines insgesamt größeren Budgets liegen [Jon97].
In diesem Zusammenhang ist nochmals zu betonen, dass Testkosten weniger von der Qualität als von der Quantität und Komplexität eines Software-Systems abhängen. Der Testaufwand kann anhand der Anzahl der Testgegenstände geschätzt werden. Für jede Testobjektart wird ein mittlerer Testaufwand in Personentagen ermittelt. Dieser wird mit der Anzahl der Testgegenstände dieser Art multipliziert, um zum geschätzten Aufwand zu kommen (Abbildung 3.5).

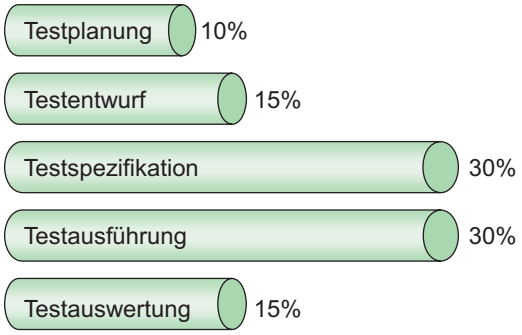


Abbildung 3.5 Aufteilung der Testaufwände

3.3.6 Teststrategie

Mit der *Teststrategie* ist die Vorgehensweise und Reihenfolge bei der Ausführung der Tests gemeint. Eine Teststrategie kann z.B. eine Top-Down- oder Bottom-Up-Integration vorsehen. Sie kann auch ein Big Bang – alles auf einmal – oder einen iterativen, stückweisen Test vorsehen. Die Strategie bestimmt also, in welcher Reihenfolge die Klassen, Komponenten und Teilsysteme zu testen sind. Darüber hinaus bestimmt sie den allgemeinen Testansatz. Es sind hier die Teststufen wie z.B. Klassentest, Integrationstest, Systemtest und Abnahmetest (Abbildung 3.6), sowie die Testmethodik wie z.B. regelbasierter, nachrichtenbasierter oder nutzungsbasierter Test, vorzugeben. Dieser Abschnitt wird gewöhnlich einer der längsten im Testplan sein.

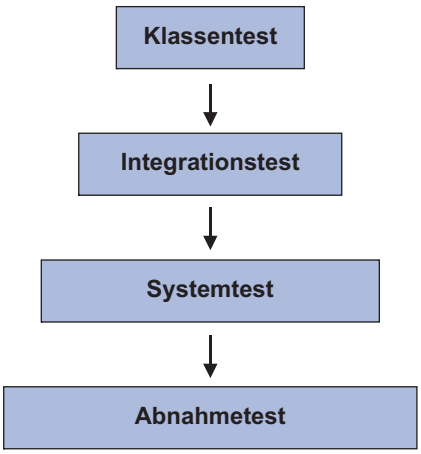


Abbildung 3.6 Festlegung der Teststufen

3.3.7 Testendekriterien

Testendekriterien sind die Erfolgskriterien für jede einzelne Teststufe und haben mit Testmetriken zu tun. Ein endloser Test ist kein Test! Um ein Ende zu erreichen, muss ein Test messbar sein, d.h. man muss das Erreichen eines Ziels quantifizieren. Dazu werden Testüberdeckungsmaße herangezogen. Ein Klassentest könnte ein Endekriterium wie z.B. 100% Funktions- und 90% Zweigüberdeckung haben. Der Integrationstest könnte ein Endekriterium wie 100% aller Nachrichten und 80% aller Ausnahmebehandlungen haben. Beim Systemtest wird das Endekriterium etwas wie 99% aller Anwendungsfälle und 90% aller Fehlerfälle sein.

Ein weiteres Testendekriterium ist die Fehlerrate. Eine fehlerlose Software wird es nie geben, aber eine Fehlerrate von 1 Fehler pro 1000 Transaktionen wäre erreichbar. Man könnte auch die Fehleranzahl schätzen oder hochrechnen und so lange testen, bis man diese Anzahl aufgedeckt hat. Messbare quantitative Testendekriterien sind auf jeden Fall eine Voraussetzung für einen erfolgreichen Test [BaSe87].

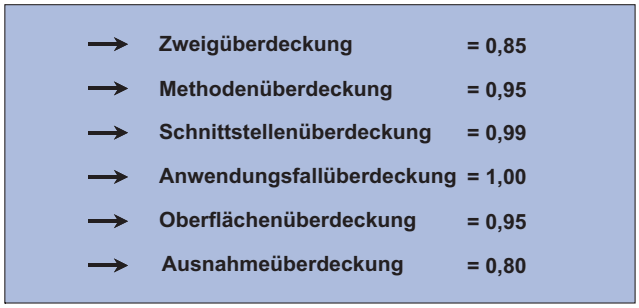


Abbildung 3.7 Testendekriterien

3.3.8 Regressionstestkriterien

Bei einer zyklischen iterativen Software-Entwicklung, so wie sie für objektorientierte Projekte vorgesehen ist, kommt dem Regressionstest eine besondere Rolle zu. Der Test jeder zusätzlichen Schicht von Klassen erbt den Test der darüberliegenden Schichten, die bereits getestet worden sind. D.h. bis auf den ersten Test der Basisklassen an der Spitze der Klassenhierarchie sind alle weiteren Testzyklen auch ein Regressionstest. Neben den neuen Testfällen für die neuen Klassen werden die alten Testfälle für die alten Klassen wiederholt. In diesem Zusammenhang sind die Erkenntnisse von Perry und Kaiser einzubeziehen, nämlich dass alle geerbten Methoden nochmals im neuen Kontext zu testen sind [PeKa90].
Umso wichtiger ist es, Kriterien für den Regressionstest vorzugeben, Kriterien wie der Grad, zu dem bereits getestete Funktionen nochmals getestet werden, wie die Testprozeduren aufzubewahren sind und wie die Regressionstestüberdeckung zu messen ist. Messziele für einen Regressionstest sind z.B.:

- der prozentuale Anteil der alten Testfälle,
- der prozentuale Anteil der neuen Testfälle,
- die Testüberdeckung der alten Klassen und
- die Testüberdeckung der neuen Klassen.

3.3.9 Testergebnisse

Die Testergebnisse sind die Dokumente bzw. Berichte und Programme, die vom Testprojekt erzeugt werden. Laut ANSI-Norm 829 sind es mindestens die acht in Abbildung 3.8 dargestellten Ergebnisse [IEEE829]:

- der Testplan,
- der Testentwurf,
- die Testfallspezifikation,
- die Testprozeduren,
- das Testobjektverzeichnis,
- der Testlog,
- der Testvorfallsbericht und
- der Testabschlussbericht.

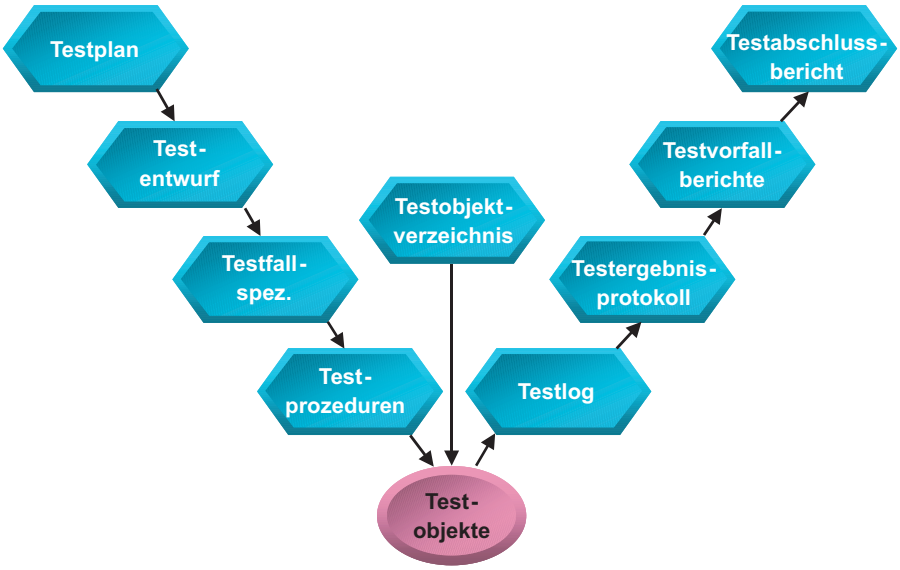


Abbildung 3.8 Testergebnisse

3.3.9.1 Testplan

Der *Testplan* ist das Dokument, das hier in diesem Kapitel beschrieben wird.

3.3.9.2 Testentwurf

Der *Testentwurf* ist das Ergebnis der Entwurfsphase. Er umfasst die Testarchitektur, die Teststufen und die Szenarien für jeden Test. Er wird im vierten Kapitel beschrieben.

3.3.9.3 Testfallspezifikation

Die *Testfallspezifikation* enthält eine Definition jedes einzelnen Testfalls mit den Vorbedingungen und Nachbedingungen sowie die zu testenden Funktionen. Die Testfälle können strukturiert, halbformal oder streng formal beschrieben werden. Testfälle gibt es für jede Teststufe, wobei eine Stufe Testfälle von anderen Stufen erben kann. Die Testfallspezifikation wird im fünften Kapitel beschrieben.

3.3.9.4 Testprozeduren

Die *Testprozeduren* sind im Grunde genommen Programme für die Ausführung des Tests. Sie werden in einer Testskriptsprache verfasst, oder sie werden von einem Tool generiert. Sie können nachher kompiliert oder interpretiert werden. Durch die Testprozeduren werden die Testfälle ausgeführt, d.h. die Vorbedingungen gesetzt, und die Nachbedingungen geprüft. Es gibt viele Arten von Testskripten in Abhängigkeit vom eingesetzten Testwerkzeug, denn leider gibt es keine normierte Testskriptsprache. Testprozeduren werden in den Kapiteln 6 bis 8 behandelt.

3.3.9.5 Testobjektverzeichnis

Das *Testobjektverzeichnis* ist ein Test aller aktuellen Testgegenstände. Dazu gehören die zu testenden Klassen, Module, Komponenten, Schnittstellen und Teilsysteme.

3.3.9.6 Testlog

Der *Testlog* ist eine Reihe automatisch generierter Testprotokolle. Protokolliert wird der Ablauf des Testgegenstands sowie die Zwischen- und Endergebnisse. So werden z.B. die erzeugten Objekte, die ausgeführten Methoden, die abgesendeten Nachrichten und die betätigten Oberflächenfunktionen registriert. Außerdem werden Objektzustände, Zusicherungsverletzungen, Soll/Istabweichungen und Datenbankinhalte festgehalten. Schließlich soll ein Testprotokoll die Testüberdeckung

berichten. Der Testlog dient der Fehlerfindung und der Testauswertung. Er wird im 9. Kapitel behandelt.

3.3.9.7 Testüberdeckungsbericht

Der *Testüberdeckungsbericht* dokumentiert, zu welchem Grade die Testgegenstände durch den Test überdeckt und die Testfunktionen erfüllt wurden. Diese Auskunft ist notwendig, um die Qualität und den Fortschritt des Tests beurteilen zu können. Auch die Testüberdeckungsmessung steht im Mittelpunkt von Kapitel 9.

3.3.9.8 Testvorfallsbericht

Der *Testvorfallsbericht* ist eine Zusammenfassung sämtlicher Fehlerberichte für einen Testzyklus. Jede Meldung bezieht sich auf eine Fehlererscheinung mit einer Beschreibung des Fehlers und der Umstände, unter denen der Fehler auftritt. Jeder Fehler ist auch nach Schwere und Risiko zu bewerten. In dem Vorfallsbericht wird eine Tabelle aller Fehler mit einer Fehlerstatistik aufgebaut. In der Fehlerstatistik geht es darum, das Verhältnis Fehler pro Kilo-Codezeilen, pro Testfall und pro Komponente aufzuzeigen. Die Fehlerberichtserstattung wird ebenfalls im 9. Kapitel behandelt.

3.3.9.9 Testabschlussbericht

Der *Testabschlussbericht* ist das letzte Dokument des Testprojekts. Er wird zum Abschluss des Tests geschrieben und fasst alle bisherigen Testergebnisse zusammen. Zum Abschlussbericht gehört die zuletzt erreichte Testüberdeckung, die endgültige Fehlerrate und die Restfehlerwahrscheinlichkeit. Auf die Gestaltung dieses Berichts wird ebenfalls in Kapitel 9 eingegangen.

3.3.10 Testaufgaben

Die Testaufgaben sind die Arbeiten, die in jeder Testphase zu erledigen sind. Typische Aufgaben sind der Entwurf der Testszenarien, die Spezifikation der funktionalen Testfälle, die Spezifikation der Integrationstestfälle, die Generierung der Testprozeduren und die Durchführung eines Komponententests. Im Allgemeinen lassen sich Testaufgaben gliedern in

- Testvorbereitungsaufgaben,
- Testausführungsaufgaben und

- Testauswertungsaufgaben.

Testvorbereitungsaufgaben sind Aktivitäten, die vor dem eigentlichen Test stattfinden. Dazu gehören die Planung, der Entwurf, die Testfallspezifikation, die Testprozedurerstellung und der Aufbau der Testdatenbanken.

Testausführungsaufgaben sind Aktivitäten, die während des Tests anfallen, z. B. Auslösung, Verfolgung und Aufzeichnung der Testläufe.

Testauswertungsaufgaben sind Aktivitäten, die nach dem Test folgen. Dazu gehört die Testauswertung, die Testdokumentation und die Fehlerberichtserstattung. Hier erscheint also eine Liste sämtlicher Aufgaben mit einer kurzen Beschreibung und einer aufgabenbezogenen Aufwandsschätzung. Dies entspricht einem z.B. mit der Netzplantechnik strukturierten Projektplan.

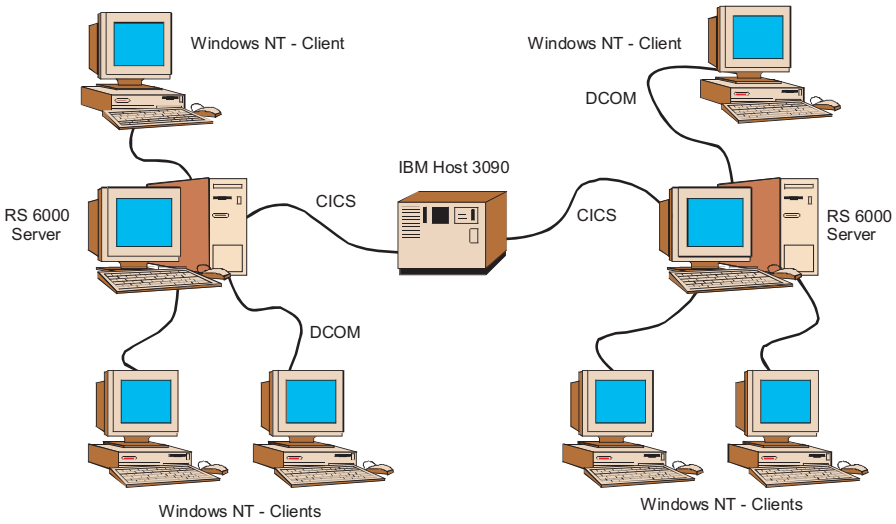


Abbildung 3.9 Aufbau der Testumgebung

3.3.11 Testumgebungsanforderungen

Die Anforderungen an die Testumgebung (Abbildung 3.9) sind an dieser Stelle aufzulisten. Hier geht es um die Zuteilung der Betriebsmittel. Für den Test werden Hardware, Software und Menschen benötigt (Abbildung 3.10). Unter Hardware sind die Rechner, Netze, Arbeitsplätze und sonstige Geräte zu verstehen, welche die Testkonfiguration ausmachen. Hierzu gehört auch die Speicherkapazität für die Testdaten. Unter Software sind die Basissoftwaresysteme wie Datenbanksysteme, Betriebssysteme, Middleware-Produkte und Dienstprogramme sowie die Testwerkzeuge zu verstehen. Unter Menschen ist das Testpersonal zu verstehen. In der Regel wird eine Mischung aus technischen und fachlichen Mitarbeitern benötigt. Es

kommt hier darauf an, den Personalbedarf sowie den Hardware- und Software-Bedarf zu dokumentieren.

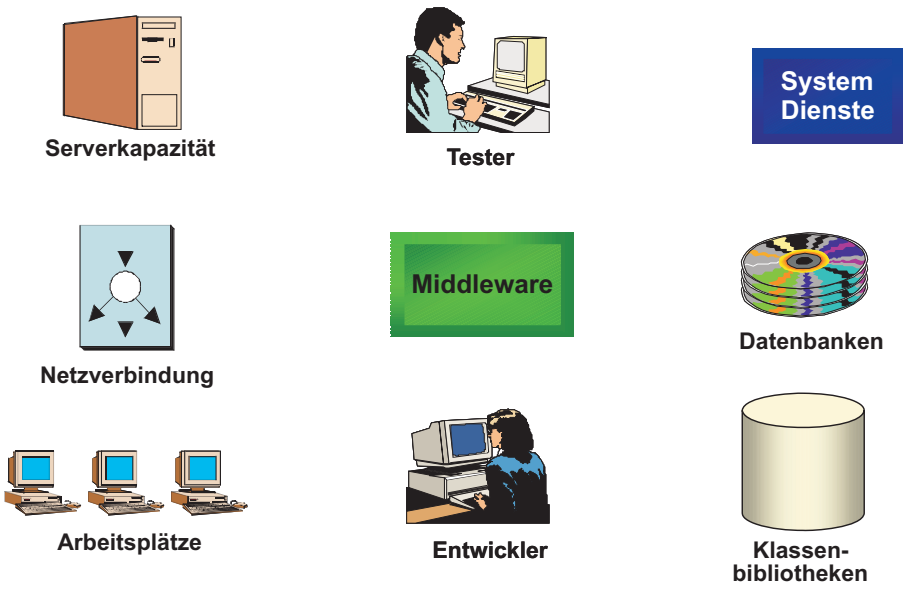


Abbildung 3.10 Zuteilung der Testbetriebsmittel

3.3.12 Testverantwortlichkeiten

In diesem Abschnitt wird geregelt, wer für welche Teststufen verantwortlich ist. Normalerweise sind die Entwickler für den Klassen- und Modultest zuständig, aber die Testgruppe ist dafür verantwortlich, eine Klassentestumgebung bereitzustellen. Für den Integrationstest liegt die Verantwortung bei der Testgruppe, aber die Entwickler sind beteiligt. Der Systemtest und alle weiteren Teststufen wie Belastungstest und Abnahmetest liegen im Verantwortlichkeitsbereich der Testgruppe. Wichtig ist die Fixierung der Verantwortung vor dem Start des Projekts.

3.3.13 Testaufgabenteilung

Die Verteilung der Testaufgaben auf das Testpersonal soll hier dokumentiert werden. Es ist eindeutig zu klären, wer welche Aufgaben wahrzunehmen hat. Natürlich wird dies zu Beginn des Projekts nicht immer vollständig möglich sein, sodass dieser Abschnitt meistens später eingefügt wird, wenn das Personal zur Verfügung steht und alle Testaufgaben bekannt sind. Eine persönliche Verantwortung für Testaufgaben ist eine wichtige Voraussetzung für den Erfolg des Testprojekts.

3.3.14 Testzeitplan

Der Testzeitplan ist ein Balkendiagramm für die Zuordnung der Testaktivitäten auf der Zeitachse bzw. auf dem Kalender. Hier kommt es darauf an, für jede Testaufgabe die Start- und Endzeit zu fixieren. Es versteht sich, dass das Testprojekt als eigenständiges Projekt auch einen eigenen Projektplan haben muss. Dieser Plan lässt sich mit den üblichen Projektplanungswerkzeugen erstellen.

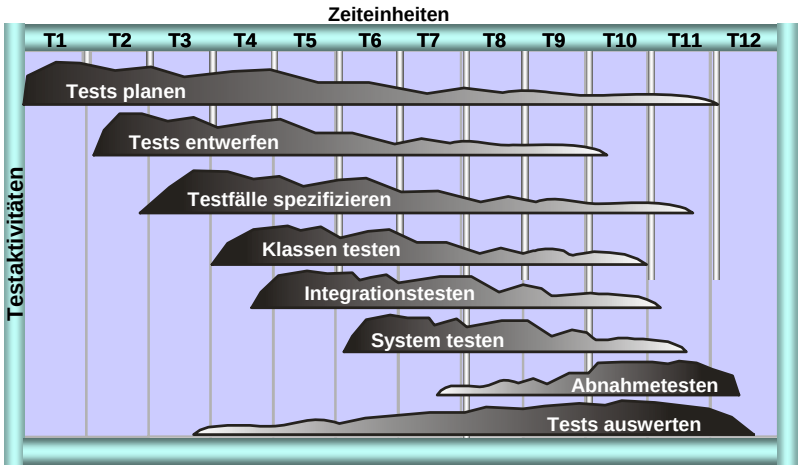


Abbildung 3.11 Testzeitplan

3.3.15 Testrisiken und Notpläne

Zum Schluss des Testplans darf dieses unangenehme Thema nicht fehlen. Es ist als Teil der Projektrisikoanalyse anzusehen. Nur werden hier ausschließlich Testrisiken behandelt. Damit sind solche Risiken wie die Unreife der Testwerkzeuge, der Verlust von Testpersonal und der Ausfall gewisser Testgeräte gemeint. Notpläne sind Maßnahmen, die vorgesehen sind, wenn Probleme auftreten. Was tun, wenn Werkzeuge nicht so funktionieren wie versprochen? Was tun, wenn Schlüsselpersonen ausfallen oder wenn geplante Betriebsmittel nicht rechtzeitig bereit stehen? Alle möglichen negativen Vorkommnisse sind hier mindestens anzudenken und zu dokumentieren. Anschließend sind Gegenmaßnahmen für deren Vermeidung bzw. für deren Behandlung vorzuschlagen. Es gibt inzwischen formale Verfahren für die Erfassung und Bewertung von Projektrisiken wie z.B. die vom Texas Department für Informationssysteme herausgegebenen Richtlinien zum Risikomanagement [Sta96].

3.3.16 Genehmigungen

Der letzte Abschnitt des Testplans ist vorgesehen für die Genehmigung des Plans. Hier erfolgen die Unterschriften des Testprojektleiters, des Gesamtprojektleiters, des Anwendervertreters und des Topmanagers. Die Führungskräfte bestätigen hiermit, dass sie mit dem Testplan einverstanden sind und ihn mit tragen. Wenn später Kostenüberschreitungen, Terminverschiebungen oder gar unüberwindliche Probleme auftreten, sind sie mitverantwortlich.

3.4 Auswirkung der Objektorientierung auf die Testplanung

Die objektorientierte Struktur eines Software-Systems hat natürlich Einfluss darauf, wie das System getestet wird, und dies wiederum beeinflusst die Testplanung. Der Einfluss ist umso größer, je tiefer man sich in der Software-Architektur befindet. Von der Oberfläche aus betrachtet ist es gleich, ob das dahinterliegende System objektorientiert ist oder nicht. Es kann sein, dass das System sich schneller verändern und fortschreiben lässt, aber das hat mit der momentanen Funktionalität nichts zu tun. Sie ist gegeben. Das Gleiche gilt für die Datenbasis, sofern sie nicht eine objektorientierte Datenbank ist. So gesehen ist die Objektorientierung – sieht man einmal von objektorientierten Anforderungsspezifikationen mit Anwendungsfällen und Domänen-Klassenmodellen ab – im Wesentlichen eine Frage der inneren Zusammensetzung.

Der Test der Software-Architektur bzw. der Integrationstest wird eher von der Objektorientierung der Software beeinflusst. Zum einen, wenn es sich um eine Hierarchie von Klassen und zum anderen, wenn es sich um eine Menge kollaborierender Klassen handelt. Im ersten Fall muss man Top-Down testen, d.h. angefangen mit den Objekten an der Spitze der Objekthierarchie. Zunächst werden die Basisklassen aus anderen Klassenbibliotheken getestet, dann die eigenen Basisklassen, dann die abgeleiteten Klassen usw., bis man an der untersten Stufe der Hierarchie angelangt ist. Siegel nennt dies in Anlehnung an eine Arbeit von Harrold et Al. die hierarchisch-inkrementelle Teststrategie (hierarchical incremental test strategy, HIT) [HMF92].

Im zweiten Fall wird man Bottom-Up testen, d.h. ausgehend von den Klassen, die aufgerufen werden, selbst aber keine weitere aufrufen. Solche Klassen befinden sich an der untersten Stufe der Aufrufhierarchie und sind als reine „Dienstleister“ die Senken der Aufruffolge. Von dort aus wird man sich aufwärts – bottom up – arbeiten zu den Steuerungsklassen oder reinen „Dienstnutzern“, deren Methoden selbst z.B. nur vom Anwender über die Benutzungsoberfläche aufgerufen werden und dann andere aufrufen. Sie stehen an der Spitze der Aufrufhierarchie. Diese

Integrationsstrategien spielen eine wichtige Rolle bei der Planung des Integrationstests.

Der Unit-Test wird am stärksten von der Objektorientierung geprägt. Es handelt sich hier um den Test von Klassen, und Klassen sind anders als konventionelle Module. Konventionelle Module konnten unabhängig voneinander getestet werden, denn die gemeinsamen Datenstrukturen und Codeabschnitte wurden zur Compilierzeit in sie eingebunden. Sie wurden also mehrfach dupliziert. Die „Call"-Aufrufe fremder Module waren auch wesentlich weniger als die Methodenaufrufe bzw. Nachrichten in C++ oder Java. Also war es möglich, Module in einen Testrahmen einzubinden.

Mit Klassen ist dies nicht so einfach. Klassen erben Funktionen von übergeordneten Klassen und rufen zahlreiche Funktionen in anderen Klassen auf. Die aufgerufenen Funktionen können eventuell durch Stellvertreter simuliert werden, nicht aber die übergeordneten Klassen. Sie müssen dazugebunden werden. Dies verändert die Ausgangsbasis für einen Klassentest und erzwingt eine andere Vorgehensweise für den Unit-Test. In dieser Hinsicht hat die Objektorientierung einen erheblichen Einfluss auf die Testplanung. Sie macht nämlich alles viel schwieriger, vor allem bei komplexen Klassenhierarchien [Hum95].

3.5 Auswirkung der Verteilung auf die Testplanung

Die Objektorientierung ist nicht der einzige Faktor, der Einfluss auf die Testplanung hat. Wie bereits gezeigt wurde, sind objektorientierte Anwendungssysteme in der Regel auch verteilte Systeme. Durch die Verteilung der Anwendung auf Client- und Serverrechner bzw. auf Client-, Server- und Hostrechner stellen sich viele zusätzliche Anforderungen an den Test; Anforderungen, die bei der Testplanung zu berücksichtigen sind [BaGo99].

Erstens muss eine vernetzte Rechnerarchitektur für den Test aufgebaut werden. Dies beeinflusst die Betriebsmittelplanung. Zum Zweiten muss die Middleware für die Verbindung der verteilten Komponenten getestet werden. Dieser Architekturtest mit Stellvertreter-Komponenten muss geplant werden. Zum Dritten muss der Planer an die vielen Ausnahmesituationen, die bei der Interaktion verteilter Komponenten auftreten können, denken. Für sie müssen spezielle Testszenarien vorgesehen werden. Schließlich müssen die diversen Testprozesse auf dem Clientarbeitsplatz, dem Serverrechner und dem Host abgestimmt werden. Dies alles stellt ziemlich hohe Ansprüche an den Testplaner und unterstreicht die Notwendigkeit eines wohldurchdachten Testplans.

Mit konventionellen Batch- und Dialog-Anwendungen auf einem Hostrechner konnte man sich immer noch ohne Plan durchwursteln. Nicht aber mit verteilten, objektorientierten Systemen. Der Komplexitätsgrad ist um Faktor 3 bis 4 größer.

Wer mit dem herkömmlichen Testansatz planlos vorgeht, ist zum Scheitern verurteilt. Die Messlatte der erforderlichen Professionalität ist um einige Intervalle höher gesetzt [MgKo94].

3.6 Ein Testplan für den verteilten Kalender

Als Beispiel für die Testplanung dient ein Projekt zur Entwicklung eines verteilten Kalenders. Jeder Mitarbeiter soll einen persönlichen Kalender auf seinem Rechner-Arbeitsplatz haben, um seine Tagesaktivitäten einzutragen. Jede Tagesaktivität soll eine Startzeit, eine Endzeit, einen Typ und eine Beschreibung haben. Die Anzahl der Aktivitäten sind auf 12 pro Tag begrenzt. Aktivitäten, die projektbezogen sind, werden gegen die entsprechenden Projekte gebucht. Dazu ist ein gemeinsames Projektkennzeichen erforderlich. Allerdings gehören die Projekte zu einem anderen System und werden auf einem anderen Rechner verwaltet. Es kommt darauf an, die Anzahl der geleisteten Stunden bei den Projekten sowie bei den Mitarbeitern zu akkumulieren.

Für den Mitarbeiter sind die Aktivitäten Tagen und die Tage Wochen zugeordnet. Ein Jahreskalender hat bis zu 52 Wochen, eine Woche hat bis zu 7 Tage und ein Tag bis zu 12 Aktivitäten. Tage unterscheiden sich nach Arbeitstagen und Feiertagen. Für die Arbeitstage gilt der Normaltarif, für Feiertage gilt ein Sondertarif.

Die Wochenkalender werden zwar an dem Arbeitsplatzrechner bearbeitet, jedoch an einem zentralen Server gehalten (Abbildung 3.12). Dort sollten sie in einer relationalen Datenbank gespeichert sein. Hier hat auch das Management den Zugriff auf sie, um zu verfolgen, was die Mitarbeiter so machen.

Es ist entschieden worden, das System mit UML zu konzipieren, die Programme mit C++ zu realisieren, die Kalenderdaten mit ORACLE-8 zu speichern und die Aktivitäten mit den Projekten über eine Visibroker CORBA-Schnittstelle zu verbinden.

Der Testmanager wird beauftragt, einen Testplan auszuarbeiten. Das Ergebnis seiner Arbeit könnte so aussehen, wie es in den folgenden Abschnitten gezeigt ist.

Kalender-Testprojektidentifikation

Das Kalendertestprojekt trägt den Titel *Kalendertest* und wird im Verzeichnis

```
X: Kalender\Test
```

geführt.

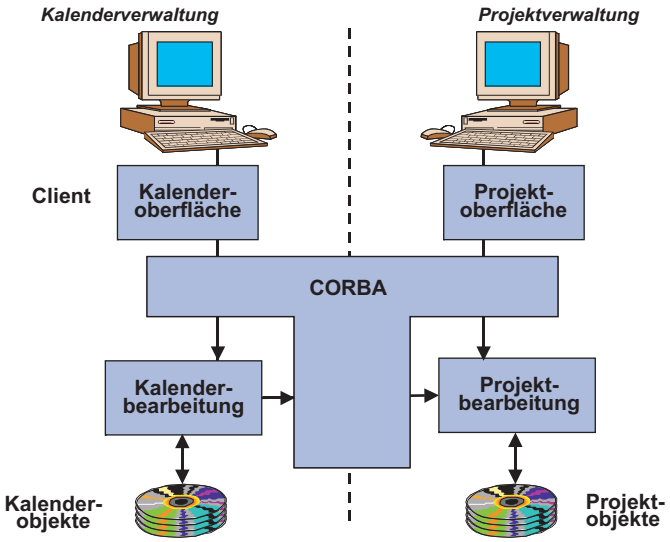


Abbildung 3.12 Kalendersystemarchitektur

Kalender-Testprojektbeschreibung

Das Kalendertestprojekt soll den Beweis liefern, dass

- Mitarbeiterkalender in einer verteilten Umgebung fehlerfrei erstellt und fortgeschrieben werden können;
- Mitarbeiterstunden korrekt akkumuliert werden, sowohl bei den Mitarbeitern selbst als auch bei den Projekten, gegen welche die Stunden gebucht werden, und
- Kalender bei Systemabbrüchen und Netzausfällen gerettet werden.

Das Testprojekt läuft unter der Leitung des Testmanagers, der für den Integrationstest allein verantwortlich ist. Für die Dauer des Systemtests wird ein stellvertretender Endanwender zum Testprojekt abgestellt. Für den Modultest sind die Entwickler zuständig. Ihre Ergebnisse müssen jedoch vom Testmanager abgenommen werden.

Das Testprojekt soll in 7 Phasen parallel zum Entwicklungsprojekt ablaufen:

- Testkonzepterstellung,
- Testfallspezifikation,
- Testumgebungsaufbau,
- Klassentest,
- Integrationstest,
- Systemtest und

- Systemabnahme.

Kalender-Testgegenstände

Es sind folgende Objekte zu testen
Klassen:

- Kalender
- Zeitraum
- Wochen
- Tag
- Aktivität
- Mitarbeiter
- Projekt

Schnittstellen:

- Mitarbeiter zu Kalender
- Aktivität zu Projekt
- Mitarbeiter zu Projekt

Systeme:

- Kalenderführung
- Projektsteuerung

Anwendungsfälle:

- Mitarbeiter anlegen
- Kalender anlegen
- Wochen anlegen
- Tage anlegen
- Aktivitäten anlegen
- Aktivitäten ändern
- Aktivitäten löschen
- Kalender löschen

Kalender-Testziele

Funktionale Testziele für den verteilten Kalender sind:

- Verifikation aller Klassenfunktionen
- Erzeugung aller Objektarten
- Erprobung aller Schnittstellen
- Auslösung aller Plausibilitätsprüfungen

- Bestätigung aller Anwendungsfälle mit mindestens zwei Varianten

Nichtfunktionale Testziele sind:

- Messung der Zeit für die Übertragung der Kalenderobjekte auf den Clientrechner
- Messung der Zeit für die Abspeicherung der Kalenderobjekte auf dem Serverrechner
- Validation der Systemwiederanlaufsfähigkeit
- Validation der Datenbanksicherheit

Kalender-Testeinschränkungen

Es stehen maximal zwei Client-Arbeitsplätze zur Verfügung.

Der Projektserver steht für Testzwecke nur abends ab 18.00 Uhr zur Verfügung.

Das Datenbanksystem wird erst zum Integrationstest installiert.

Nur ein Anwender kann für höchstens eine Woche am Systemtest beteiligt sein.

Die Testgruppe darf nicht länger als einen Monat mit dem Test des Kalendersystems belastet sein.

Es dürfen nur die folgenden Testwerkzeuge eingesetzt werden:

- CPPANAL,
- CPPTEST,
- CPPINST,
- IDLTEST und
- WinRunner

Kalender-Teststrategie

Die Teststrategien sind:

- Klassentest: Built-In-Strategie
- Integrationstest: Top-Down-Strategie
- Systemtest: Anwendungsfallstrategie

Diese Strategien werden im Testkonzept näher erläutert.

Kalender-Testendekriterien

Die Mindesttestendekriterien sind:

- Klassentest: einmalige Erzeugung aller Objektarten und die Ausführung aller Operationen
- Integrationstest: Austausch zweier Nachrichten – einer korrekten und einer unkorrekten – zwischen den Kalendersystemen und dem Mitarbeiter sowie zwischen dem Kalendersystem und dem Projektsystem
- Systemtest: Test jedes spezifizierten Anwendungsfalls und jeder spezifizierten Fehlermeldung

Die endgültigen Testendekriterien werden im Testkonzept vereinbart.

Regressionstestkriterien für den verteilten Kalender

Da der Built-In-Test der Klassen im Klassensource bleiben soll, wird er immer wiederholbar sein. Der Entwickler muss allerdings die Zusicherungen den Codeänderungen anpassen.

Der Integrationstest wird über die CORBA-Schnittstelle wiederholbar sein, weil die IDL-Definition vom IDLTEST-Schnittstellentreiber interpretiert wird. Dennoch muss auch hier der Tester die Vor- und Nachbedingungen den Schnittstellenänderungen anpassen.

Der Systemtest wird das erste Mal an der Benutzungsoberfläche von WinRunner aufgezeichnet und alle weiteren Male von WinRunner zurückgespielt. Bei Änderungen in der Oberfläche muss das Testskript angepasst werden.

Kalender-Testergebnisse

Das Kalendertestprojekt hat folgende Ergebnisse zu liefern:

- Testkonzept
- Klassentestfälle
- Integrationstestfälle
- Systemtestfälle
- Testobjektverzeichnis
- Klassentestprotokoll
- Schnittstellenprotokoll
- Testüberdeckungsprotokoll
- Datenbankprotokoll
- Oberflächenprotokoll
- Mängelberichte

- Testvorfallsbericht
- Testabschlussbericht

Das Format der Ergebnisse ist in der betrieblichen Testrichtlinie vorgegeben.

Kalender-Testaufgaben

Vor dem Test des verteilten Kalenders sind folgende Aufgaben wahrzunehmen:

- Testanforderungen ermitteln
- Testansatz festlegen
- Testszenarien ausarbeiten
- Testendekriterien setzen
- Testkonzept verfassen
- Testfälle spezifizieren
- Testwerkzeuge bereitstellen
- Testumgebung aufbauen
- Testdatenbank installieren

Im Test des verteilten Kalenders sind folgende Aufgaben wahrzunehmen:

- Objektmodell verifizieren
- Klassen statisch analysieren
- Klassen dynamisch analysieren
- Klassen abnehmen
- Komponenten testen
- Komponentenschnittstellen testen
- Komponenten abnehmen
- Oberfläche testen
- Datenbank testen
- Gesamtsystem testen
- Gesamtsystem abnehmen

Nach jedem Test des verteilten Kalenders sind folgende Aufgaben wahrzunehmen:

- Testprotokolle auswerten
- Testüberdeckung messen
- Testergebnisse validieren
- Mängelberichte erfassen
- Testbericht schreiben

Kalender-Testumgebungsanforderungen

Um das verteilte Kalendersystem zu testen, ist als Hardware erforderlich

- mindestens zwei PC-Arbeitsplätze
- ein Server für die Kalender
- ein zweiter Server für die Projekte
- ein LAN-Netz

Folgende Software ist erforderlich:

- ein CORBA-Konformer ORB
- eine relationale Datenbank
- eine Netzsoftware
- ein C++-Compiler
- ein C++-statischer-Analysator
- ein C++-dynamischer-Analysator
- ein CORBA-Schnittstellengenerator
- ein Capture/Replay-System
- ein Datenbankvalidator

Kalender-Testverantwortlichkeiten

Es werden folgende Verantwortlichkeiten festgelegt:

- Testkonzept → Testmanager
- Einrichtung der Testumgebung → Netzwerkgruppe
- Installation der Datenbank → Datenbankgruppe
- Bereitstellung der Testwerkzeuge → Toolgruppe
- Test der Klassen → Entwickler
- Integrationstest → Testmanager
- Systemtest → Testgruppe
- Systemabnahme → Personalabteilung

Kalender-Testaufgabenverteilung

Die Einzelaufgaben werden folgenden Personen zugeordnet:

- Testanforderungen ermitteln → Testmanager
- Testansatz festlegen → Testmanager
- Testszenarien ausarbeiten → Testmanager
- Testendekriterien setzen → Testmanager
- Testkonzept verfassen → Testmanager

- Testfälle spezifizieren → Testmanager, Entwickler
- Testwerkzeuge bereitstellen → Toolgruppe
- Testumgebung aufbauen → Netzwerkgruppe
- Testdatenbank installieren → Datenbankgruppe
- Objektmodell verifizieren → Benutzervertreter
- Klassen statisch analysieren → Entwickler
- Klassen dynamisch analysieren → Entwickler
- Klassen abnehmen → Testmanager
- Komponenten testen → Testgruppe
- Schnittstellen testen → Testmanager, Testgruppe
- Komponenten abnehmen → Testmanager
- Oberfläche testen → Testgruppe, Benutzervertreter
- Datenbank testen → Testgruppe
- Gesamtsystem testen → Testgruppe
- Gesamtsystem abnehmen → Benutzervertreter
- Testprotokolle auswerten → Testmanager
- Testüberdeckung messen → Testmanager
- Testergebnisse validieren → Testmanager
- Mängelberichte erfassen → Entwickler, Testgruppe, Benutzervertreter, Testmanager
- Testbericht schreiben → Testmanager

Kalender-Testzeitplan

Das Testkonzept ist zeitgleich mit dem Klassenmodell abzuschließen.

Die Spezifikation der Testfälle und die Vorbereitung des Tests sollten bis zur Fertigstellung der ersten Klassen abgeschlossen sein.

Der Klassentest soll unmittelbar auf die Kodierung der Klassen folgen und maximal zwei Wochen dauern.

Der Integrationstest folgt auf den Klassentest und dauert ebenfalls maximal zwei Wochen.

Der Systemtest folgt dem Integrationstest und dauert bis zu einem Monat.

Nach Abschluss der Klassenkodierung sollte die Zeitdauer bis zur Systemabnahme nicht länger als die Zeitdauer bis zu diesem Zeitpunkt sein, d.h. der Beginn des Klassentests markiert die Mitte des Projekts, was Kalenderzeit und Personalaufwand anbetrifft.

Kalender-Testrisiken und Notpläne

Es könnte sein, dass die Zeit für die Spezifikation aller notwendigen Testfälle nicht ausreicht. In diesem Falle sind extreme Testfälle und Ausnahmefälle wegzulassen.

Es könnte auch sein, dass die ORB-Verbindung zwischen den Kalendern und den Projekten nicht rechtzeitig realisiert wird. In diesem Fall wird eine Batchschnittstellendatei mit den projektbezogenen Aktivitäten erzeugt.

Es könnte schließlich vorkommen, dass das Testwerkzeug IDLTEST für den Test der Interaktionen zwischen verteilten Komponenten nicht so funktioniert, wie erwartet. In diesem Falle ist der Integrationstest zu überspringen und direkt vom Klassentest in den Systemtest zu gehen.

Kalendertestgenehmigung

Dieser Testplan wird vom Projektleiter, dem Leiter der Anwendungsentwicklung und dem Leiter der zuständigen Fachabteilung genehmigt. Mit ihrer Unterschrift übernehmen sie die Verantwortung für das Testprojekt zusammen mit dem Testmanager, der den Testplan verfasst hat.

4

Objektorientierter Testentwurf

Überblick und Ergebnisse
Testanforderungen an objektorientierte Systeme
Testansätze für objektorientierte Systeme
Testszenarien für objektorientierte Systeme
Testendekriterien für objektorientierte Systeme
Ein Testkonzept für den verteilten Kalender

Inhaltsübersicht Kapitel 4

4 Objektorientierter Testentwurf

4.1 Überblick und Ergebnisse

Der objektorientierte Testentwurf (*Test Design*) erfolgt als zweite Phase im Testprojekt nach der Testplanung. Ergebnis des Testentwurfs ist das Testkonzept, in dem die Testanforderungen festgehalten und eine Strategie für deren Erfüllung ausgearbeitet werden. Im ANSI-Teststandard heißt es

> *„...the purpose of the test design is to specify the test requirements and how they are to be met“* ***[IEEE610].***

Das Testkonzept sollte nach der Norm 829 folgenden Inhalt haben [IEEE829]:

- Testkonzeptkennung,
- Testanforderungen,
- Testansätze,
- Testszenarien und
- Testendekriterien.

4.1.1 Testkonzeptkennung

Die *Testkonzeptkennung* ist eine Bezeichnung für die Testkonzeptbibliothek und die darin gespeicherten Konzeptdokumente. Die Art der Kennung hängt vom Dateiverwaltungssystem ab, z.B.

```
Test\Konzept\Anford.doc
       Ansatz.doc
       Szenario.doc
       Kriteria.doc
```

4.1.2 Testanforderungen

Die *Testanforderungen* sind die Funktionen und Eigenschaften der Software, die durch den Test zu bestätigen sind. Voraussetzung für die Spezifikation der Testanforderungen ist die Spezifikation der Systemanforderungen. Dazu gehören funktionale und nicht-funktionale Anforderungen. Funktionale Anforderungen machen die Funktionalität der Software aus. Hier geht es um die Anwendungsfälle, d.h. um die verschiedenen Bewegungsarten und Transaktionen, die von der Anwenderseite aus angestoßen werden können. Funktionale Testanforderungen sind die Maßnahmen, die erforderlich sind, um zu beweisen, dass die Systemfunktionen korrekt implementiert sind. Nicht-funktionale Anforderungen sind hingegen Systemeigenschaften wie Robustheit, Sicherheit, Performanz und Benutzerfreundlichkeit, deren Erfüllungsgrad vorgegeben ist. Qualitative Testanforderungen sind die Maßnahmen, die erforderlich sind, um den Erfüllungsgrad der nicht-funktionalen Anforderungen zu messen [ISO9126].

4.1.3 Testansätze

Die *Testansätze* sind, laut der Norm, die spezifischen Techniken, die anwendbar sind, um die Testanforderungen zu erfüllen. Als Beispiele werden in der Norm der Abgleich von Dateien und Aufzeichnung der Benutzereingaben angegeben. Für den Oberflächentest ist die Aufzeichnung und Rückspielung der Interaktionen ein bewährter Ansatz. Ein weiterer Ansatz ist der Abgleich alter und neuer Datenbankinhalte durch ein Vergleichswerkzeug (Comparator Tool). Für den Unittest wäre die Nutzung eines Testrahmens für die Simulation der Modulumgebung ein möglicher Ansatz. Im zweiten Kapitel wurde der regelbasierte Ansatz für den Klassentest, der nachrichtenbasierte Ansatz für den Integrationstest und der anwendungsfallbasierte Ansatz für den Systemtest zitiert. Im Testkonzept kommt es darauf an, solche Ansätze im Bezug auf das jeweilige Projekt zu beschreiben.

4.1.4 Testszenarien

Die *Testszenarien* sind Pläne für die Durchführung des Tests. Ein Testszenario ist im Grunde genommen eine Folge aufeinander bauender Testläufe. Beim Unit-Test spielt diese Sichtweise keine große Rolle, doch ab dem Integrationstest wird sie immer wichtiger. Jeder Test baut auf dem vorhergehenden Test auf. Deshalb muss die Reihenfolge der Testläufe genau überlegt und festgeschrieben werden. Z.B. müssen die Datenbanken im Dialogbetrieb aufgebaut werden, ehe die Berichtgenerierung läuft. In Client/Server-Netzen wird es besonders kompliziert, vor allem wenn es darum geht, nebenläufige Prozesse in mehreren Threads zu testen. Die

zeitlichen Abhängigkeiten der parallel laufenden Prozesse müssen in Testszenarien abgebildet werden. So gesehen entsprechen Testszenarien dem Entwurf der geplanten Testprozesse.

4.1.5 Testendekriterien

Die *Testendekriterien* sind nach der Norm

> *„Criteria to be used to determine whether the test passed or failed“* ***[IEEE610].***

Solche Kriterien richten sich nach dem Gegenstand des Tests. Ist der Gegenstand eine Klasse, sind die Endekriterien in der Regel Überdeckungsmaße wie z.B. der Prozentsatz der Methoden und Zweige, die pro Objektinstanz ohne Fehler ausgeführt werden. Ist der Gegenstand eine Komponente, sind die Endekriterien Maße für die fehlerfreie Aktivierung aller Interaktionen zwischen Klassen innerhalb der Komponente sowie die fehlerfreie Befriedigung aller externen Schnittstellen der Komponente. Im Falle eines Systems oder Teilsystems beziehen sich die Endekriterien auf die Summe der Anwendungsfälle, die durch das System fehlerfrei auszuführen sind, sowie auf die Erfüllung nichtfunktionaler Anforderungen, wie Antwortzeit, Durchlaufzeit, Fehlerabfangkapazität und Belastungskapazität. Die Dokumentation der Endekriterien ist eine unausbleibliche Voraussetzung für einen systematischen Test, denn ein Test ohne Endekriterien ist ein endloser Test, und ein endloser Test ist kein Test.
Abbildung 4.1 zeigt die Elemente des Testkonzepts im Zusammenhang.

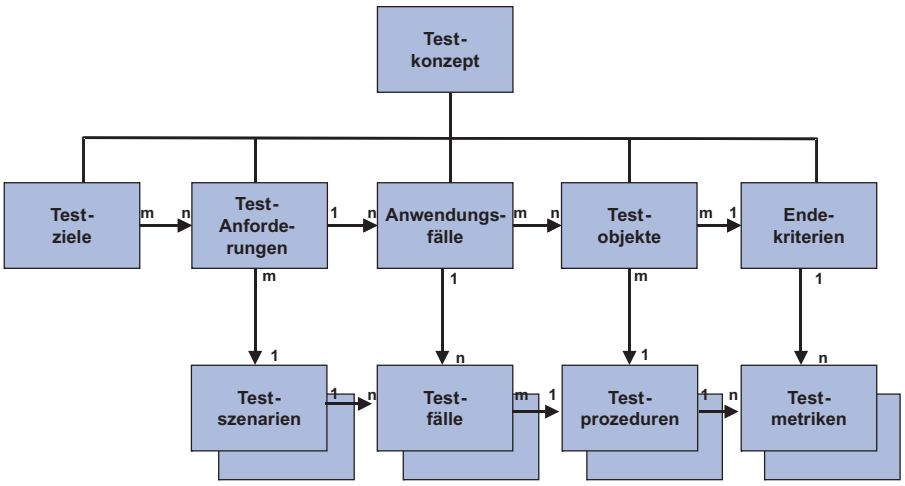


Abbildung 4.1 Struktur des Testkonzeptes

4.2 Testanforderungen an objektorientierte Systeme

Moderne objektorientierte Systeme sind in der Regel auch Client/Server-Systeme mit graphischen Oberflächen und Objekt/Relationalen-Datenbanken. Es kommen deshalb mehrere Anforderungen zusammen. Darin liegt der Hauptgrund für den erhöhten Testaufwand, der laut der Literatur mindestens 40% der Gesamtkosten beträgt. Eigentlich sind die Testanforderungen aufgrund der Objektorientierung gering im Verhältnis zu den Anforderungen, die sich aus der Systemverteilung und der Komplexität der Oberflächen ergeben, d.h. die Hauptkostentreiber beim Test sind Verteilung, Vernetzung und Vervielfältigung.

4.2.1 Client/Server-Testanforderungen

Client/Server-Systeme haben verteilte Programme und möglicherweise auch verteilte Daten. Eine einzige Geschäftstransaktion wird von mehreren Teilprogrammen an mehreren Rechnern ausgeführt und verarbeitet Daten, die ebenfalls in verschiedenen Datenbanken gelagert sind. Die erfolgreiche Ausführung ist daher von vielen Faktoren abhängig. Die Datenübertragung muss funktionieren, die Programme bzw. Objekte müssen richtig geladen bzw. erzeugt werden, die Schnittstellen müssen übereinstimmen, die Daten müssen zusammenpassen, die Zugriffe auf die Datenbanken müssen synchronisiert sein, parallel laufende Transaktionen dürfen sich nicht gegenseitig stören. Speicher muss wieder freigegeben werden, Datenbanken müssen geschlossen werden, Fehler müssen abgefangen und Transaktionen ordnungsgemäß abgeschlossen werden. In Fehlerfällen müssen die alten Zustände wiederhergestellt werden. *Handshaking, Two Phase Commit* und *Rollback* sind wohlklingende Begriffe, hinter denen ein enormer Testaufwand steckt, denn sie müssen leider in allen Variationen ausprobiert werden [Wey98].
Typische Probleme, die bei verteilten Systemen auftreten können, sind u.a.:

- Einschränkungen in den vorhandenen Netzwerkressourcen, wie z.B. Verbindungen, Geschwindigkeit und Bandbreite;
- Netzwerkausfälle, die zum Abbruch der laufenden Transaktionen führen;
- Fehler in der Synchronisation parallel arbeitender Komponenten;
- Deadlocks beim Zugriff auf gemeinsam benutzte Ressourcen;
- Performanz-Engpässe im Netz, vor allem beim Zugriff auf gemeinsame Serverobjekte;
- Entwurfsfehler in Kommunikationsprotokollen aufgrund unvorhergesehener Situatione;

- Time-out-Fehler, die einen Abbruch laufender Transaktionen verursachen [Bou97].

Dies sind nur einige der vielen Fehlermöglichkeiten eines verteilten Systems. Der Kontrollfluss in einer verteilten Anwendung ist wesentlich komplexer als in einer nicht verteilten Anwendung. Insofern gibt es auch viel mehr potenzielle Probleme.
Wer also die Testanforderungen für ein Client/Server-System formuliert, muss mit den technischen Eigenschaften solcher Systeme wohl vertraut sein. Es gilt vor allem, solche Ausnahmesituationen wie Deadlock und Netzausfall zu erproben. Deshalb wird es bei Client/Server-Systemen viele Testanforderungen geben, die mit der eigentlichen Anwendung nichts zu tun haben. Sie beziehen sich auf die Systemarchitektur (Abbildung 4.2).

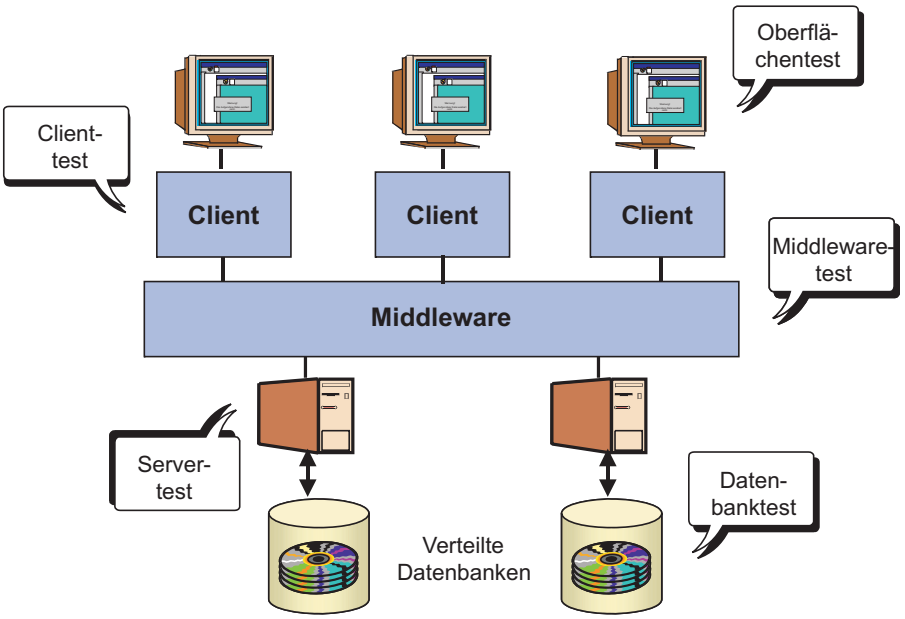


Abbildung 4.2 Test der C/S Architektur

4.2.2 GUI-Testanforderungen

Neue objektorientierte Anwendungen haben meistens eine *graphische Benutzungsoberfläche* (graphical user interface, GUI) Die heutigen WIMP-Oberflächen (windows, icons, menus, pointing device) sind sehr komplex und bieten viele Möglichkeiten der Bedienung an. Oft kann jeder Endbenutzer seine Oberfläche individuell anpassen, zumindest was Farben, Schriftgrößen und Layout anbelangt. Er kann auch wahlweise mit der Maus oder mit der Tastatur arbeiten, um mit der Maschine

zu kommunizieren. Daraus ergeben sich hohe Anforderungen an den Test, denn die vielen Bedienungsmöglichkeiten müssen getestet werden (Abbildung 4.3). Jede Oberfläche hat eine Reihe von Ein- und Ausgabemöglichkeiten. Beim alten 3270-Terminal mit seinen festformatierten Masken waren diese Möglichkeiten beschränkt. Ein Großteil der Felder war geschützt. Der andere Teil ließ nur bestimmte Eingaben zu. Das einzige Eingabemedium war die Tastatur. Der höchste Komfort war die Nutzung von Funktionstasten. Moderne graphische Oberflächen sind mindestens dreimal so komplex. Neben der Tastatur und den Funktionstasten kommt die Maus als Eingabemedium und Ikonen bzw. Bilder als Ausgabemedium hinzu. Es können zur gleichen Zeit mehrere Masken aktiv sein, sodass der Benutzer gleichzeitig mehrere Transaktionen bearbeiten kann. Er kann mit dem Tabulator oder mit der Maus positionieren, und er kann die Reihenfolge der Eingabesignale beliebig kombinieren [Tro95].

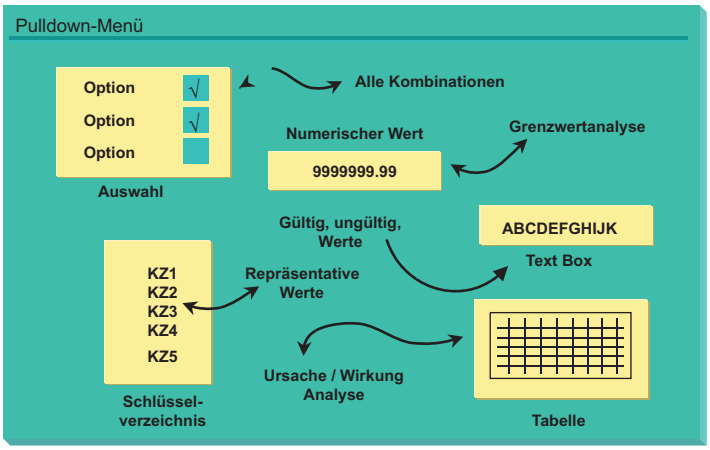


Abbildung 4.3 Test der Benutzungsoberfläche

All dies hat seinen Preis im Test. Die zugelassenen wie die nicht zugelassenen Bedienungsmöglichkeiten müssen alle erprobt und bestätigt werden. Die ersten, um zu beweisen, dass sie funktionieren, und die letzten, um zu beweisen, dass sie abgefangen werden. Deshalb hat der Test von GUIs zu einer anspruchsvollen eigenen Disziplin mit zahlreichen Hilfsmitteln geführt. Die automatische Aufzeichnung von Eingabesignalen und deren Rückspulung sowie die flächendeckende Bombardierung der Oberfläche mit Mausklicken und die Simulation möglicher Tastenkombinationen sind alle nur Versuche, den Anforderungen eines GUI-Tests gerecht zu werden.

Die Mindestleistung ist die Dokumentation der GUI-Testanforderungen in Form von Checklisten, die darauf hinweisen, was alles zu testen wäre. Auch wenn diese

Anforderungen aus Zeit- und Kapazitätsgründen nicht alle erfüllbar sind, dienen sie trotzdem als Richtlinie für den Oberflächentest [Sch95].

4.2.3 Datenbanktestanforderungen

In der Praxis benutzen die wenigsten objektorientierten Anwendungen auch *objektorientierte Datenbanken*. Wenn dies so wäre, könnte man die Datenhaltung bzw. die Objektaufbewahrung als verlängerte Form der persistenten Objektverwaltung testen, d.h. der Datenbanktest wäre Bestandteil des Objekttests. Zur Zeit sind die meisten objektorientierten Anwendungen in der Regel mit relationalen Datenbanken kombiniert. Die Objekte werden erst zur Laufzeit nach Bedarf dynamisch aus den Relationen bzw. aus den Sichten auf die Relationen gebildet und nachher wieder in Relationen aufgelöst. Diese dynamische Objektbildung ist nicht unproblematisch. Hinzu kommt, dass relationale Datenbanken eigene Testanforderungen stellen, vor allem was die Datenkonsistenz, die Datenkorrektheit, die Datensicherheit und die Performanz anbetrifft [Koe95].

Relationale Datenbanken sind komplexe Strukturen mit vielen verschleierten Abhängigkeiten. Eine besonders mächtige, aber auch gefährliche Abhängigkeit ist die referenzielle Integrität, wobei die Existenz einer Relation von der Existenz anderer Relationen in anderen Tabellen abhängt. Falls die Basisrelation gelöscht wird, werden die anderen abhängigen Relationen automatisch mit gelöscht. Auch Änderungen können die Abhängigkeitskette durchbrechen und zu Datenbankabbrüchen führen. Insofern kann eine einzige SQL-Operation eine größere Datenmenge zerstören. `Delete`- und `Modify`-Operationen müssen daher in allen Ausprägungen sorgfältig getestet werden.

Datenbankabfragen können zu unterschiedlichen Ergebnissen führen, je nachdem, welche Datenkonstellation zur Zeit der Abfrage vorherrscht. Besonders fehleranfällig ist die Vereinigung verteilter Tabellen durch eine `Join`-Operation. Die geringste Abweichung in einer der betroffenen Tabellen kann das Ergebnis beeinflussen, erst recht, wenn die Joins verschachtelt sind. Daher ist es für den Test der Datenbankzugriffe erforderlich, Datenbankzustände einzufrieren und zu archivieren, um denselben Test wiederholen zu können. Dies erfordert wiederum eine große Speicherkapazität und eine systematische Testdatenverwaltung. Die Testdatenbanken müssen ebenso versioniert und verwaltet werden wie die Programme.

Moderne relationale Datenbanken haben zahlreiche mächtige Eigenschaften, um den Umgang mit ihnen zu erleichtern, aber diese Eigenschaften müssen alle getestet werden (Abbildung 4.4). Dazu gehören u.a.

- Stored Procedures,
- Triggers,
- Rules und

- Constraints [Hal94].

Stored Procedures müssen wie andere Programme Schritt für Schritt getestet werden, um sicher zu sein, ob alle logischen Ausgänge korrekte Ergebnisse liefern. Stored Procedures sind nicht nur mächtige Automaten, die wichtige Funktionen auf den Daten ausführen, sie sind auch mächtige Fehlerquellen. Am besten ist es, wenn sie instrumentiert werden, um ihr Verhalten zu analysieren.

Triggers sind eine besondere Art von Stored Procedures, die eigenmächtig in Aktion treten, wenn ein bestimmter Zustand auftritt, z.B. wenn eine vorgegebene Zeit erreicht ist, oder wenn eine bestimmte Datenrelation geändert wird. Um sie zu testen, muss ihr Auslöser manipuliert werden, d.h. die Systemuhr setzen, oder den auslösenden Zustand künstlich hervorrufen. Dafür braucht man besondere Testfälle.

Rules sollten verhindern, dass die Datenbank in einen logisch inkonsistenten Zustand gerät. Zu diesem Zweck prüfen sie die Eingabedaten, ehe sie in die Tabellen einfließen, und kontrollieren die Tabelleninhalte in regelmäßigen Intervallen. Wenn Regeln verletzt werden, wird eine Ausnahmebedingung erhoben. Z.B. dürfen Kontobestände die spezifizierte Kreditgrenze nicht unterschreiten. Wenn sie dies tun, wird das Konto automatisch gesperrt. Um solche Regeln zu testen, müssen Testfälle aus der Regelspezifikation abgeleitet werden.

Constraints sind Regeln, die bestimmte unerwünschte Datenkombinationen verhindern. Fremdschlüssel-Constraints verhindern z.B., dass nur Werte in einer als Fremdschlüssel bestimmten Spalte vorkommen, die auch in der anderen Tabelle als Primärschlüssel vorkommen. Der Test von Constraints verlangt Testfälle, die diese Einschränkungsregel verletzen. Der Tester muss versuchen, Relationen einzufügen, die nicht zulässig sind, oder Relationen zu löschen, die nicht änderbar sind. Aus den Einschränkungen ergeben sich reichlich Testfälle.

All dies zeigt, wie umfangreich der Test relationaler Datenbanken werden kann. Hinzu kommen die ganzen Ausnahmesituationen wie Deadlocks, Speicherfehler, Timeouts und Überläufe. Es wird kaum möglich sein, alle potenziellen Probleme im Test hervorzurufen. Dennoch ist es nützlich, eine Checkliste zu führen, zumindest als Erinnerung an all das, was man noch nicht getestet hat. Dies gilt übrigens auch für die anderen Testanforderungen.

Zum Schluss gibt es besondere Testanforderungen für Systeme, die eine gekapselte Datenbank als zentrale Objektrepository benutzen. Die Daten für die Bildung der Objekte auf dem Server werden aus der zentralen Datenbank, z.B. IMS/DB oder DB/2, genommen und per Datenfernübertragung auf den Server übertragen. Danach werden die betroffenen Segmente oder Tabellen wieder freigegeben. Nachdem die Objekte auf dem Server verarbeitet wurden, kommen die Daten wieder zurück zum Host. Bis dahin könnte es gut sein, dass die Segmente oder Tabellen, aus denen die Daten gewonnen wurden, durch andere Hosttransaktionen verändert worden sind. Dies ist ein typischer Fall, der unbedingt getestet werden muss, auch wenn der

Testaufwand erheblich ist. Daran kann man sehen, welche Probleme verteilte Daten mit sich bringen [PBR90].

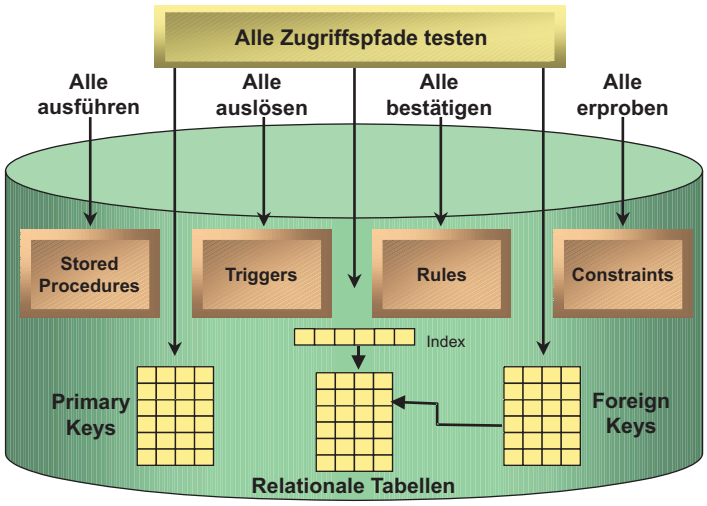


Abbildung 4.4 Test der Datenbanken

4.2.4 Objekttestanforderungen

Zu den Anforderungen für den Test der verteilten Verarbeitung der graphischen Oberfläche und der relationalen Datenbanken kommen die Anforderungen für den Test verteilter Objekte noch hinzu (Abbildung 4.5). In gewissem Sinne sind verteilte Objekte eine Variante der verteilten Verarbeitung. Es geht um die Abarbeitung von Transaktionen über mehrere Netzknoten hinweg. Sowohl die Daten als auch die Funktionalität sind auf verschiedenen Rechnern verteilt. In einer nicht-objektorientierten Applikation sind jedoch die Teilprogramme statisch gebunden. Der wiederanlauffähige Code (reentrant) für die Verarbeitung einer Transaktionsart bleibt unverändert. Nur die Daten ändern sich. Beim Test solcher Systeme kommt es darauf an, alle relevanten Pfade durch den Code zu durchlaufen. Damit ist automatisch der Test aller relevanten Datenkonstellationen verknüpft.

Im Falle der verteilten Objekte wird der Code vervielfältigt. Für eine Transaktionsart gibt es nicht eine einzige Codeeinheit mit mehreren Datenbereichen, sondern mehrere Codeeinheiten, jede mit eigenem Datenbereich. Sie werden dynamisch erzeugt und sollten nach ihrer Nutzung wieder freigegeben werden. Die Pfade durch die Objekte sind auch nicht deterministisch wie bei den Prozeduren. Ein- und dieselbe Transaktion kann einmal so und einmal anders ablaufen, je nachdem, welche Objekte geladen und welche Ressourcen vorhanden sind. Insofern ist eine genaue

Voraussage eines Ablaufpfades nicht möglich, ebensowenig der Abgleich von Soll- und Istpfad.
Getestet werden hier nicht nur die Schnittstellen und der Datenaustausch zwischen Teilen eines einzelnen Programms, sondern auch die Schnittstellen und der Datenaustausch zwischen allen Ausprägungen bzw. Instanzen eines Programms. Statt auf Pfade durch den Code zielen Testfälle auf Ausprägungen des Codes und auf Pfade durch die Ausprägungen. Die Anzahl der Testfälle ist nicht gleich der Anzahl der Pfade, sondern gleich der Anzahl der Instanzen mal der Anzahl der Pfade. Damit steigt die Testfallanzahl multiplikativ im Verhältnis zur Anzahl möglicher Instanzen.

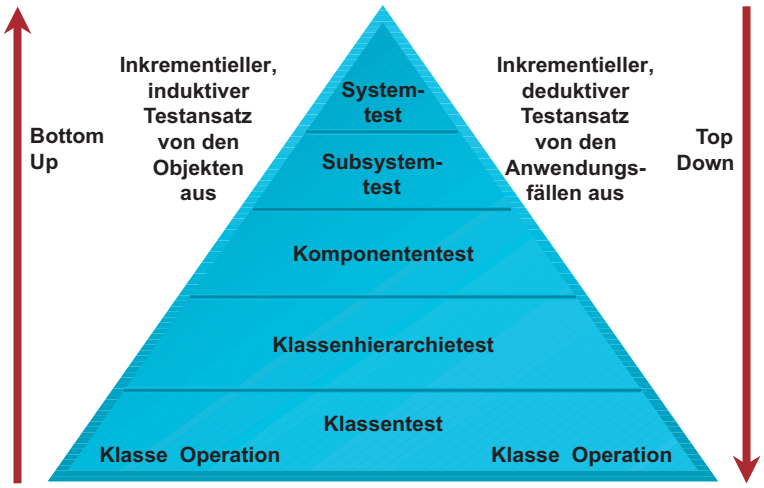


Abbildung 4.5 Testobjekthierarchie

Die Herausforderung an den Test verteilter Objekte ist somit hauptsächlich ein Mengenproblem – wie testet man alle Kombinationen aller Objekte mit allen potenziellen Interaktionen. Diese Testanforderung ist im Falle komplexer Systeme nur über die Definition einer praxisrelevanten Untermenge zu erfüllen, d.h. über ein operationales Profil des Systems unter Test. Es kommt deswegen hier darauf an, folgende Objekttestanforderungen zu spezifizieren:

- welche Transaktionsarten sind zu testen;
- welche Klassen sind durch diese Transaktionen betroffen;
- welche stellvertretenden Objekte werden aus diesen Klassen erzeugt;
- welche Funktionen führen die stellvertretenden Objekte aus;
- welche relevanten Zustände können die stellvertretenden Objekte annehmen und

- welche repräsentativen Interaktionen zwischen stellvertretenden Objekten sind zu testen [Fir93].

4.3 Testansätze für objektorientierte Systeme

Der Ansatz, im Englischen *the approach*, ist die Art und Weise, wie man bei der Problemlösung vorgeht bzw. das Problem anpackt. Im Zusammenhang mit dem Test eines Software-Systems bedeutet der Begriff so viel wie Vorgehensweise. Ein *objektorientierter Testansatz* ist somit eine Vorgehensweise beim Test objektorientierter Software. Im Testkonzept wird an dieser Stelle die geplante Vorgehensweise festgehalten.
Die Literatur zum objektorientierten Test ist voller Testansätze für jede Phase des Tests. Für alle Phasen gemeinsam gilt die in Abbildung 4.6 skizzierte Unterscheidung zwischen

- White-Box Test
- Grey-Box Test und
- Black-Box Test.

Von einem *White-Box Test* spricht man, wenn der Tester sich mit der Implementierung des Testgegenstands auseinandersetzt und seinen Test auf der Konstruktion des Objekts aufbaut. Im *Grey-Box Test* beschränkt der Tester sich auf die Interaktionen zwischen Teilen des Testgegenstands und baut seinen Test darauf auf. Im *Black-Box Test* betrachtet der Tester den Testgegenstand nur von außen und testet die extern sichtbare Funktionalität. Man spricht auch von implementierungsbasierten, schnittstellenbasierten und spezifikationsbasierten Testansätzen [Bei90].

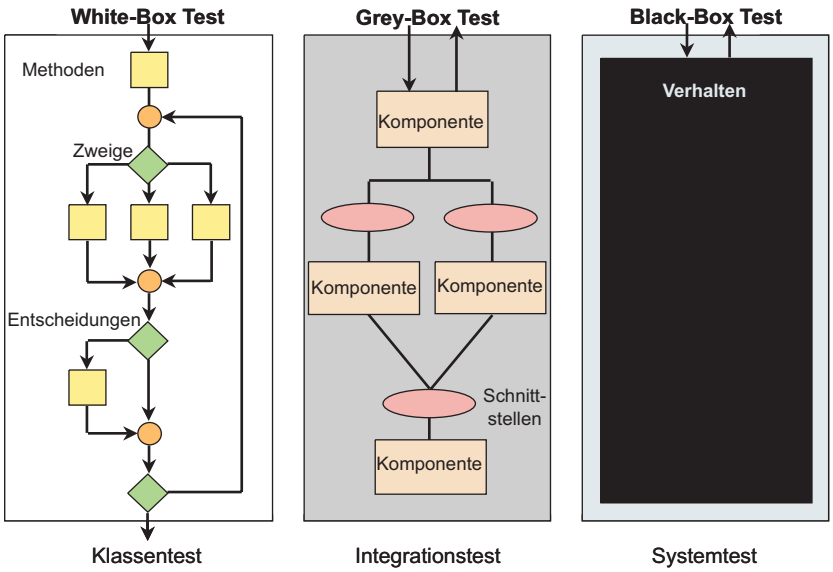


Abbildung 4.6 Testansätze

4.3.1 Ansätze für den Klassentest

Für den *Klassentest* gibt es drei Hauptansätze (Abbildung 4.7):

- Test über einen Testtreiber
- Built-In Test
- Hierarchisch inkrementeller Testansatz.

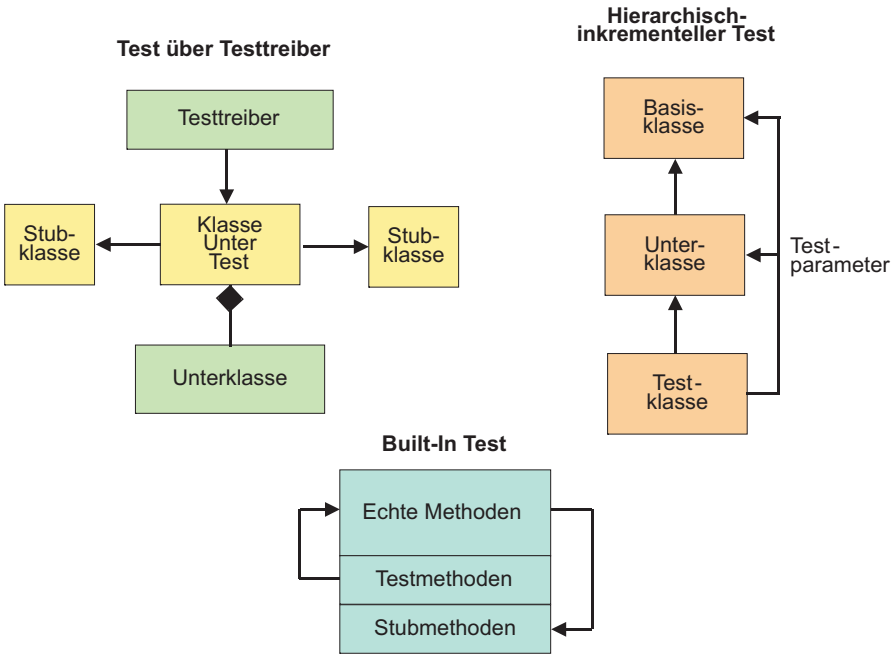


Abbildung 4.7 Klassentestansätze

Der *Test über einen Testtreiber* setzt voraus, dass die Entwickler entweder einen allgemeingültigen Testtreiber erhalten oder eigene Testtreiber anfertigen. Erstes bedeutet die Beschaffung eines geeigneten Werkzeugs, Letzteres bedeutet mehr Arbeit [HoSt97].

Beim *Built-In Test* werden die Testfunktionen in die zu testenden Klassen eingepflanzt, um diese „von innen heraus“ zu testen. Entweder werden sie durch ein Tool generiert oder von den Entwicklern programmiert. Das Erste setzt wiederum ein Werkzeug voraus, das Zweite bedeutet etwas mehr Arbeit, aber längst nicht so viel wie beim Testtreiber [Maq93].

Der *hierarchisch-inkrementelle Testansatz* bedeutet, die Basisklassen vor den abgeleiteten Klassen zu testen. Man benutzt dabei die abgeleiteten Klassen, um die oberen Klassen zu testen. Dazu werden künstliche Parameter für die Operationsaufrufe verwendet. Dieser Testansatz verlangt etwas mehr Zusammenarbeit zwischen den Entwicklern [Sie96].

Ein letzter Testansatz für den Klassentest wäre, ganz darauf zu verzichten, d.h. überhaupt keinen Klassentest durchzuführen. In Anbetracht des mit dem Klassentest verbundenen Aufwands wird dieser Ansatz allzu oft vorgezogen – um in späteren Teststufen umso mehr Aufwand betreiben zu müssen.

4.3.2 Ansätze für den Integrationstest

Für den Integrationstest werden u.a. folgende Ansätze vorgeschlagen (Abbildung 4.8):

- der Top-Down Ansatz
- der Bottom-Up Ansatz und
- der nachrichtenbasierte Ansatz.

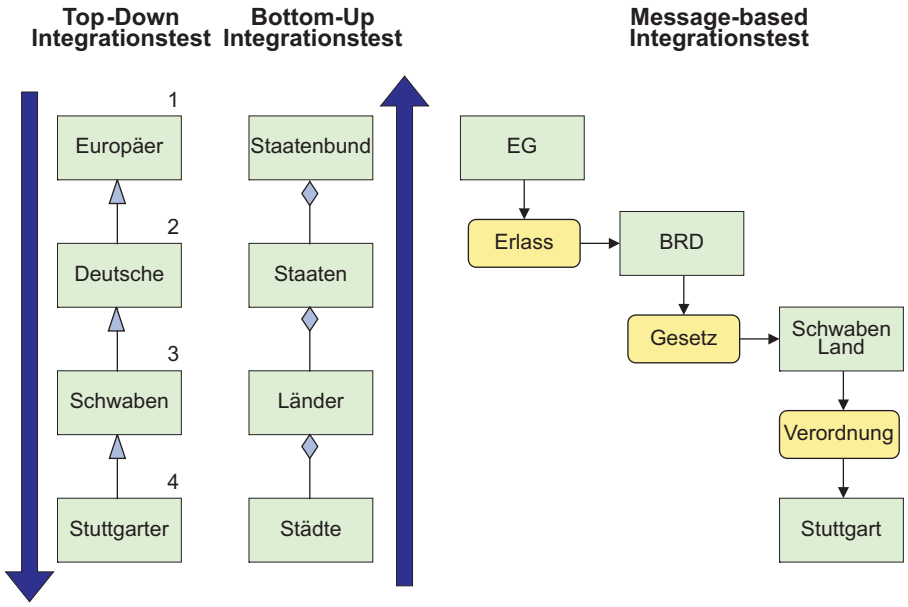


Abbildung 4.8 Integrationstestansätze

Der *Top-Down Integrationstestansatz* sieht vor, dass die obersten Basisklassen, d.h. die Klassen an der Spitze der Generalisierungshierarchie, zuerst getestet werden, dann die nächste Stufe in der Generalisierungshierarchie usw., bis man auf die Klassen stößt, die keine Unterklassen haben und somit nichts weitervererben. Auf diese Weise werden einzelne ungetestete (Unter-)Klassen gegen die bereits getesteten (Ober-)Klassen getestet. Dieser Ansatz ist aufwändig, weil die Funktionen in den Basisklassen über künstliche Ereignisse angestoßen werden müssen [HMF92].

Der *Bottom-Up Integrationstestansatz* sieht vor, dass solche Klassen zuerst getestet werden, deren Operationen keine Operationen anderer Klassen aufrufen. Sie sind die Senke der Aufrufhierarchie. Meistens befinden sich diese Funktionen in den abgeleiteten Klassen, auch deshalb die Bezeichnung *Bottom-Up*. Dieser Ansatz ist einfacher, weil die Testfälle über Operationsaufrufe realisiert werden können [JoEr94] [Ove94].

Der *nachrichtenbasierte Integrationstestansatz* sieht vor, dass interne Nachrichten generiert werden, um Funktionspfade durch die Objekte ab einer bestimmten Schnittstelle zu testen. Diese bestimmten Schnittstellen werden in der Regel die Nahtstellen zwischen verteilten Komponenten sein. Dieser Ansatz setzt einen Schnittstellentreiber vor, der die Eingangsnachrichten generiert und die Ausgangsnachrichten validiert [Sne98] [Win00].

4.3.3 Ansätze für den Systemtest

Für den Systemtest bieten sich mehrere Testansätze an (Abbildung 4.9), darunter

- der Zufallstest,
- der Anwendungsfallbasierte Test,
- der simulierte Produktionstest und
- der Wiederholungstest.

Der *Zufallstest* bedeutet so viel wie die Datenbanken mit Zufallsdaten füllen und die Benutzungsoberfläche mit Zufallseingaben bombardieren. Man könnte sich vorstellen, dass eine Horde Affen an den Bildschirmen hockt und mit der Maus und der Tastatur herumfummelt. Man könnte sich aber auch vorstellen, dass ein Automat zufällige Mausklicke und Tastaturanschläge simuliert. Der Effekt käme auf dasselbe, einen Zufallstest heraus [Bei95].
Der *Anwendungsfallbasierte Test* ist ein gezielter Funktionstest gegen die Anforderungsspezifikation [Win99]. Ein intelligentes Wesen sitzt am Bildschirm und startet die spezifizierten Anwendungsfälle. Ein Anwendungsfall nach dem anderen wird gestartet und das Ergebnis kontrolliert. Vorher wird die Datenbank mit korrekten Daten gefüllt, die zu den spezifizierten Anwendungsfällen passen. Somit wird nur das getestet, was das System eigentlich leisten sollte. Das Problem: Es könnte auch Mal ein Affe am Bildschirm sitzen, der sich nicht an das hält, was das System leisten sollte.

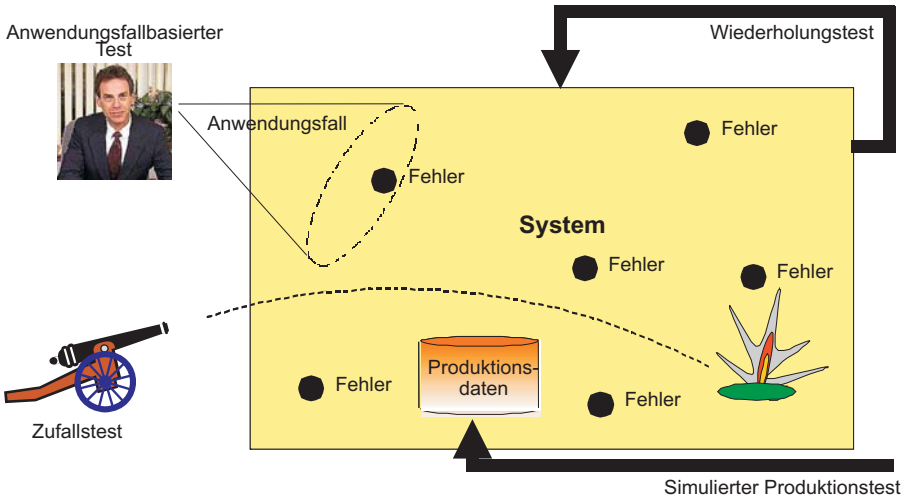


Abbildung 4.9 Systemtestansätze

Der *simulierte Produktionstest* geht davon aus, dass der produktive Einsatz des Systems in der Zielumgebung zu Testzwecken möglich ist. Dabei wird mit dem System so umgegangen, als ob es bereits in der Produktion wäre und die Operatoreingaben sowie die Systemeingaben alle aufgezeichnet würden. Die Datenbanken sind mit echten Daten aus der Produktion gefüllt. Dieser Ansatz wird auch als *live test* bezeichnet [Mun88].

Der *Wiederholungstest* setzt voraus, dass einer der bereits erwähnten Tests bereits gelaufen ist und alle Interaktionen zwischen dem Bediener und dem System aufgezeichnet sind. Er setzt weiterhin voraus, dass die Datenbankinhalte vor und nach dem Test aufbewahrt werden. Somit ist es möglich, alle Oberflächeneingaben zurückzuspielen und die Oberflächenausgaben zu vergleichen. Abweichungen sind sofort als Fehler erkennbar. Dies ist zweifelsohne der billigste und auch effektivste Testansatz, doch ist er nur in Verbindung mit einem der anderen Testansätze möglich [RoHa94].

4.4 Testszenarien für objektorientierte Systeme

Ein *Testszenario* schreibt vor, welche Tests in welcher Reihenfolge auszuführen sind. Für ein objektorientiertes System gäbe es für jede Phase und jedes Testobjekt ein eigenes Szenario. Im Klassentest wird das Testszenario vom Entwickler selbst verfasst. Er muss vorgeben, welche Methoden in welcher Reihenfolge anzustoßen

sind bzw. er überlässt das Szenario einem Automat, welcher die Testfälle für den Klassentest generiert.

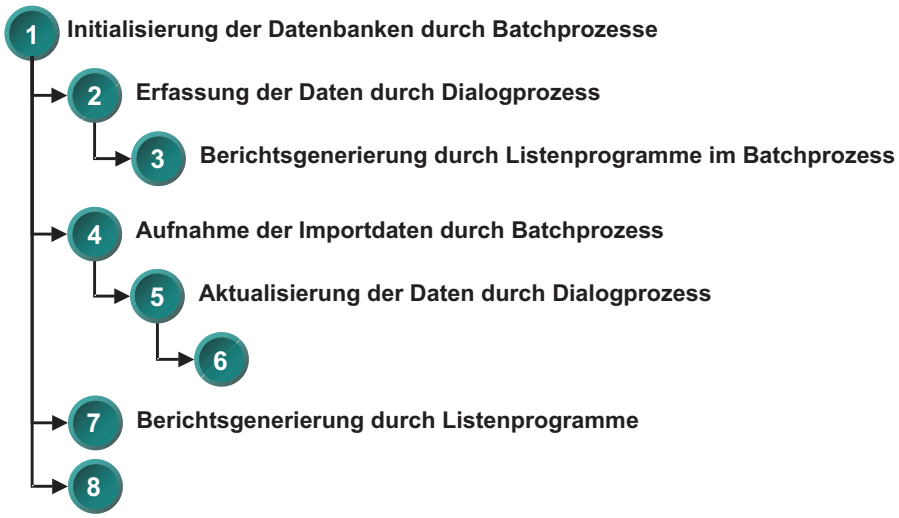


Abbildung 4.10 Festlegung eines Testszenarios

Das Szenario für den Integrationstest wird vom Testmanager der Komponenten verfasst. Er gibt an, welche Testläufe er in welcher Reihenfolge ausführen möchte und wie er sie auszuführen gedenkt. Dieses „Wie?“ ist keineswegs trivial, vor allem bei verteilten Systemen. Es ist nicht möglich, einzelne Komponenten, z.B. die Server- oder Host-Komponenten in einer Drei-Schichten Architektur, von der Oberfläche aus zu testen. Entweder muss man einen Testtreiber einsetzen oder ein Testdienstprogramm schreiben, um Aufträge für die Zielkomponenten zu generieren und Ergebnisse zu validieren. Um die Back-End Software zu testen, müssten auch die Datenbanken gefüllt sein, entweder mit künstlichen Testdaten oder mit Daten aus der Produktion. So gesehen verlangt der Test einzelner Komponenten eines Gesamtsystems den Aufbau eines Testgerüsts. Die Komponentenschnittstellen müssen simuliert werden, und dies erfordert Werkzeuge.

Das Szenario für den Systemtest müsste vom Leiter des Testteams verfasst werden. Es bestimmt, welche Transaktionen mit welchen Ausnahmebedingungen und welchen fehlerhaften Angaben in welcher Folge zu starten sind (Abbildung 4.10). Es könnte sein, dass die fehlerhaften Fälle zunächst getestet werden, dann die extremen und zuletzt die normalen Fälle. Es könnte ebenso gut umgekehrt sein. Auf jeden Fall wird es erforderlich sein, die Testdatenbanken vorher aufzubauen und nachher zu kontrollieren.

Im Systemtestszenario sind folgende Aktivitäten zu berücksichtigen:

- die Generierung der Testdatenbanken,

- die Auslösung fehlerhafter Transaktionen,
- die Auslösung extremer Transaktionen,
- die Auslösung parallel laufender Transaktionen,
- die Inszenierung von Transaktionsabbrüchen,
- die Inszenierung von Datenbankausfällen,
- die Aufzeichnung der Testtransaktionen,
- die Validierung der Testdatenbanken,
- die Verfolgung der Testabläufe und
- die Registrierung der Abweichungen [Gra96].

4.5 Testendekriterien für objektorientierte Systeme

Die *Testendekriterien* dienen dazu, ein Ziel für die Tester zu setzen. Sie unterscheiden sich je nach Testobjekt. Ist der Testgegenstand eine Klasse, beziehen sich die Endekriterien auf die Eigenschaften der Klasse, z.B. Methoden, Zweige, Parameter, Objektinstanzen und Objektzustände. Das Ziel wird sein, diese Eigenschaften bis zu einem gewissen Grade zu bestätigen, z.B. durch die Ausführung der Methoden, die Überdeckung der Zweige, die Zuweisung gewisser Parameter, die Erzeugung bestimmter Objektinstanzen und die Herbeiführung bestimmter Objektzustände. Entweder wird genau angegeben, welche Eigenschaften zu testen sind, oder es wird ein Prozentsatz gesetzt, den der Tester zu erreichen hat, wie z.B. 90% aller Methoden oder 80% aller Zweige. Wurde für die Klasse ein Zustandsdiagramm angegeben, so können Endekriterien wie z.B. 100% aller Zustände oder 90% aller Transitionen festgesetzt werden. Auf Anweisungsebene kann für besonders kritische Operationen gefordert werden, 90% aller aufgrund der Generalisierungshierarchie möglichen dynamischen Bindungen für die enthaltenen Operationsaufrufe zu erzielen.
Ist der Testgegenstand eine Komponente, sind die Endekriterien Maße für die Überdeckung der Komponentenfunktionen, z.B. die Ausführung aller Datenbankzugriffe, die Auslösung aller Ausnahmebedingungen, die Bedienung aller externen Schnittstellen und die Erzeugung aller stellvertretenden Nachrichten. Wenn nicht alle erreichbar sind, muss ein Mindestmaß spezifiziert werden, z.B. 75% aller Datenbankzugriffe und/oder 90% aller Funktionen und/oder 80% aller Nachrichtenarten. Solche Testmetriken setzen wiederum voraus, dass Messwerkzeuge vorhanden sind, um sie zu ermitteln [FrWe93].
Die folgende Tabelle 4-1 listet die im Test prozeduraler Anwendungen häufig anzutreffenden Überdeckungsmaße auf. Alle Maße basieren auf der Struktur des Pro-

grammcodes und legen eine implementierungsbasierte Testfallermittlung nahe. Für die Ermittlung der Werte dieser Überdeckungsmaße ist Werkzeugunterstützung zwingend notwendig.

Überdeckungsmaß	Kurzbeschreibung
C_0-Überdeckung bzw. Anweisungsüberdeckung	$= \frac{\text{Anzahl der durchlaufenen Anweisungen}}{\text{Anzahl aller Anweisungen des Testlings}}$
C_1-Überdeckung bzw. Zweigüberdeckung	$= \frac{\text{Anzahl der durchlaufenen Zweige}}{\text{Anzahl aller Zweige des Testlings}}$ Verschiedene Zweige entstehen durch bedingte Anweisungen wie binäre Fallunterscheidungen, CASE-Anweisungen oder Schleifenkonstrukte.
C_2-Überdeckung bzw. Bedingungsüberdeckung	$= \frac{\text{Anzahl der zu } \texttt{true} \text{ und } \texttt{false} \text{ ausgewerteten Bedingungen}}{\text{Anzahl aller Bedingungen des Testlings}}$ Es werden die elementaren Glieder in den Bedingungsausdrücken der bedingten Anweisungen gezählt.
C_3-Überdeckung bzw. Überdeckung der Bedingungskombinationen	Wie Bedingungsüberdeckung, aber es werden die verschiedenen Kombinationen der elementaren Glieder in den Bedingungsausdrücken der bedingten Anweisungen gezählt.
C_4-Überdeckung bzw. Pfadüberdeckung	$= \frac{\text{Anzahl der durchlaufenen Pfade}}{\text{Anzahl aller Pfade des Testlings}}$ Die Anzahl der Pfade eines Programms wächst exponentiell mit der Anzahl der bedingten Anweisungen. Schleifen mit einer beliebigen Anzahl von Schleifendurchläufen (z. B. `while`-Schleifen) führen zu unendlich vielen Pfaden.

Tabelle 4-1: Überdeckungsmaße für prozedurale Software

Wegen der andersartigen Struktur des Programmcodes in objektorientierten Sprachen spielen die Überdeckungsmaße C_0 bis C_4 keine herausragende Rolle bei der Definition von Testzielen für objektorientierte Anwendungsentwicklungen. Trotzdem sollten die Programme entsprechend instrumentiert und diese Überdeckungsmaße – sozusagen als „Kontrollgrößen“ für die Güte der spezifikationsbasierten Tests – gemessen werden.

In der folgenden Tabelle 4-2 sind einige mögliche Überdeckungsmaße für objektorientierte Programme zusammengestellt. Diese basieren nicht alle nur auf dem ablauffähigen Programmcode, sondern teilweise auch auf den Spezifikationen des Anwendungssystems oder einzelner Klassen (vgl. Abschnitt 5.7).

Überdeckungsmaße für die Spezifikation sind sehr gut geeignet zur Definition von Testzielen in Verbindung mit spezifikationsbasierten Tests. Werkzeuge sind allerdings oft nicht in der Lage, die automatische Ermittlung solcher Kenngrößen zu unterstützen. Andererseits ist die manuelle Ermittlung spezifikationsbezogener Testziele vergleichsweise unproblematisch, da sie gemeinsam mit der Testfallermittlung, die selbst auch nur zu einem geringen Teil automatisierbar ist, erledigt werden kann: Die Definition von spezifikationsbezogenen Black-box-Testfällen

sollte genau auf die Erfüllung der spezifikationsbezogenen Testziele hin ausgerichtet sein.

Überdeckungsmaß	Kurzbeschreibung
Überdeckung der angebotenen Operationen einer Klasse	$= \frac{\text{Anzahl der aufgerufenen Operationen der Klasse}}{\text{Anzahl aller angebotenen Operationen einer Klasse}}$
Überdeckung der durch eine Klasse aufgerufenen Operationen	$= \frac{\text{Anzahl der aufgerufenen Operationen anderer Klassen}}{\text{Anzahl aller aufrufbaren Operationen anderer Klassen}}$ Die Aufrufe der klassenfremden Operationen werden gezählt. Beim Auftreten von dynamisch gebundenen Operationsaufrufen können als „Zählmethoden" angewandt werden: • Reiner Aufruf der klassenfremden Operationen. • Aufrufe der klassenfremden Operationen für jede mögliche Klassenbindung im Anwendungskontext
Überdeckung von Operationsparametern und exportierten Attributen einer Klasse	$= \frac{\text{Anzahl der überprüften Parameter und Attribute}}{\text{Anzahl aller Operationsparameter und Attribute der Klasse}}$ Die Überdeckung von Operationsparametern und exportierten Attributen mit Hilfe von Äquivalenzklassenbildungen oder Grenzwerttests wird gezählt.
Überdeckung der ausgelösten (behandelten) Ausnahmen	$= \frac{\text{Anzahl der ausgelösten (behandelten) Ausnahmen}}{\text{Anzahl aller auslösbaren (behandelbaren) Ausnahmen}}$
Überdeckung der Attributänderungen	$= \frac{\text{Anzahl modifizierter Attribute}}{\text{Anzahl aller Attribute der Klasse}}$
Zustandsüberdeckung	$= \frac{\text{Anzahl der erreichten Zustände}}{\text{Anzahl aller Zustände im Zustandsdiagramm}}$ Nur für Spezifikationen mit Zustandsdiagramm.
Zustandssensitive Überdeckung der Operationen einer Klasse	$= \frac{\text{Anzahl der aufgerufenen Operationen der Klasse}}{\text{Anzahl aller Operationen * Zustände der Klasse}}$ Die Aufrufe der von einer Klasse angebotenen Operationen werden für jeden Zustand des Zustandsdiagramms der Klasse gezählt. Dabei werden die nicht erlaubten Kombinationen von Operationsaufruf und Zustand mitgezählt (für diese sollte das Testergebnis eine angemessene Fehlerreaktion/ Ausnahmebehandlung nachweisen). Nur für Klassenspezifikationen mit Zustandsdiagramm.
Transitionsüberdeckung	$= \frac{\text{Anzahl der geschalteten Transitionen}}{\text{Anzahl aller Transitionen im Zustandsdiagramm}}$ Nur für Klassenspezifikationen mit Zustandsdiagramm.
Anwendungsfall-Überdeckung	$= \frac{\text{Anzahl der geprüften Anwendungsfälle}}{\text{Anzahl aller Anwendungsfälle}}$ 100% Anwendungsfall-Überdeckung sollte unbedingt als Testziel für den Systemtest definiert werden.
Szenario-Überdeckung	$= \frac{\text{Anzahl der geprüften Szenarien}}{\text{Anzahl aller Szenarien}}$ Für komplexere Anwendungsfälle sind oft sehr viele Szenarien möglich, ähnlich der Pfadüberdeckung.

Tabelle 4-2: Überdeckungsmaße für objektorientierte Software

Die Auswahl bzw. zusätzliche Definition von Testüberdeckungsmaßen hängen von der jeweiligen Teststufe, der dabei eingesetzten Testtechnik und von der Verfügbarkeit von Werkzeugen zur Ermittlung des Wertes ab. Das Gleiche gilt für das System in seiner Gesamtheit. Nur im Falle eines Systems ist die Feststellung, ob die Kriterien erfüllt werden oder nicht, viel einfacher. Da hier von außen getestet wird, ist das, was passiert, leicht zu beobachten. Typische Ziele für den Systemtest sind eine repräsentative Menge aller möglichen Anwendungsfälle, eine stellvertretende Auswahl aller möglichen Datenbankrelationen, ein bestimmter Prozentsatz aller möglichen fehlerhaften Angaben und eine repräsentative Auswahl der möglichen Ausnahmebedingungen [FHL+98].

Zu all diesen objektbezogenen Maßstäben kommt noch die Fehleranzahl dazu. Gezählt werden Mängel, leichte Fehler und schwere Fehler. Der Test kann lediglich zeigen, dass noch Fehler vorhanden sind; er kann jedoch nicht beweisen, dass keine Fehler mehr vorhanden sind. Wenn man also wissen will, wie viele der vorhandenen Fehler gefunden wurden, kann man z.B. künstliche Fehler einbauen und sehen, wie viele davon vom Test aufgedeckt werden, oder man rechnet die verbleibende Fehleranzahl aufgrund der bisher gefundenen Fehlermenge und der bisherigen Testüberdeckung hoch. Daraus ergibt sich die Restfehlerwahrscheinlichkeit. Beide Methoden sind letzen Endes Schätzungen. Gewissheit geben sie nicht.

Zu glauben, man könne testen, bis alle Fehler gefunden werden, ist illusorisch. Fehlerfreie Software wird es nie geben, denn die Realität, welche die Software abbildet, ist selbst nicht fehlerfrei. Also kann das Teilsystem Software nicht besser sein als das System, zu dem es gehört. Software kann allerdings gut genug sein, um einen nützlichen Zweck zu erfüllen. Es obliegt den Testern, in Absprache mit den Auftraggebern – seien es Endanwender oder die eigene Vertriebsabteilung – zu vereinbaren, was dieses „Gut genug“ bedeutet und wie es zu messen ist.

Auf dieses schwierige Thema kommen wir im neunten Kapitel – Testauswertung – nochmals zurück. Es genügt hier anzumerken, dass das Testendekriterium ein wichtiger Bestandteil des Testkonzepts ist.

4.6 Ein Testkonzept für den verteilten Kalender

Wir demonstrieren die Ausarbeitung eines Testkonzepts wieder am Beispiel der verteilten Kalenderverwaltung. Zur Erinnerung: Jeder Mitarbeiter der Firma pflegt seinen eigenen Jahreskalender, der nach Wochen gegliedert ist. Für jede Woche im Jahr gibt es die sieben Wochentage und für jeden Wochentag bis zu 12 Aktivitäten. Bei den Tagen wird unterschieden zwischen Arbeitstagen und Feiertagen. Arbeitsstunden an Feiertagen werden extra vergütet. Bei den Mitarbeitern wird unterschieden zwischen Angestellten und freien Mitarbeitern. Bei den Aktivitäten wird unter-

schieden zwischen projektbezogenen und nicht projektbezogenen Aktivitäten. Die Stunden der projektbezogenen Aktivitäten werden gegen Projekte gebucht. Abbildung 4.11 zeigt eine Skizze des entsprechenden Klassendiagramms.

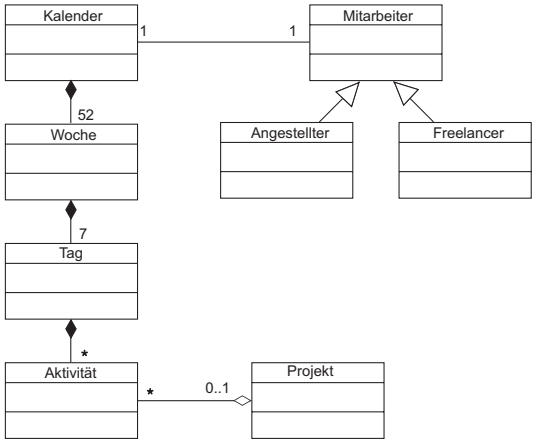


Abbildung 4.11 Kalenderklassen

4.6.1 Kalender-Testkonzeptkennung

Für die Aufnahme des Testkonzepts wird eine Bibliothek mit vier Unterbibliotheken eröffnet.
Die Hauptbibliothek heißt

```
X:Kalender\Test\Konzept
```

Die Unterbibliotheken heißen

```
X:Kalender\Test\Konzept\Anford
X:Kalender\Test\Konzept\Ansatz
X:Kalender\Test\Konzept\Szenario
X:Kalender\Test\Konzept\Kriteria
```

4.6.2 Testanforderungen für den verteilten Kalender

Die Testanforderungen der verteilten Kalenderverwaltung gliedern sich in die folgenden vier Bereiche:

- Verteilung,
- Benutzungsoberfläche,
- Datenbank und
- Klassen/Objekte.

4.6.2.1 Test der Kalenderverteilung

Die Mitarbeiterkalender sind auf einem zentralen Server in einer relationalen Datenbank gespeichert. Für die Verarbeitung der Kalender werden Objekte auf dem Arbeitsplatzrechner der Mitarbeiter erzeugt: pro Mitarbeiter ein Kalender mit bis zu 52 Wochen mit jeweils bis zu 7 Wochentagen mit jeweils bis zu 12 Aktivitäten pro Tag. Die Projekte werden auf einem anderen Serverrechner gehalten.
Somit haben wir es hier mit verschiedenen Interaktionen zwischen unterschiedlichen Netzknoten zu tun. Wenn der Mitarbeiter seinen Kalender fortschreiben möchte, müssen die dazugehörigen Objekte auf dem Server erzeugt und auf seinen Arbeitsplatzrechner übertragen werden. Wenn der Mitarbeiter mit seinem Kalender fertig ist, müssen die Objekte an den Server zurückfließen und dort in der Datenbank wieder abgelegt werden. Außerdem müssen projektbezogene Aktivitäten bei den Projekten auf dem anderen Server eingetragen werden. Daraus ergeben sich 3 Rechnerinteraktionen:

- Server zu Client Download
- Client zu Server Upload und
- Client zu Fremdserver Update.

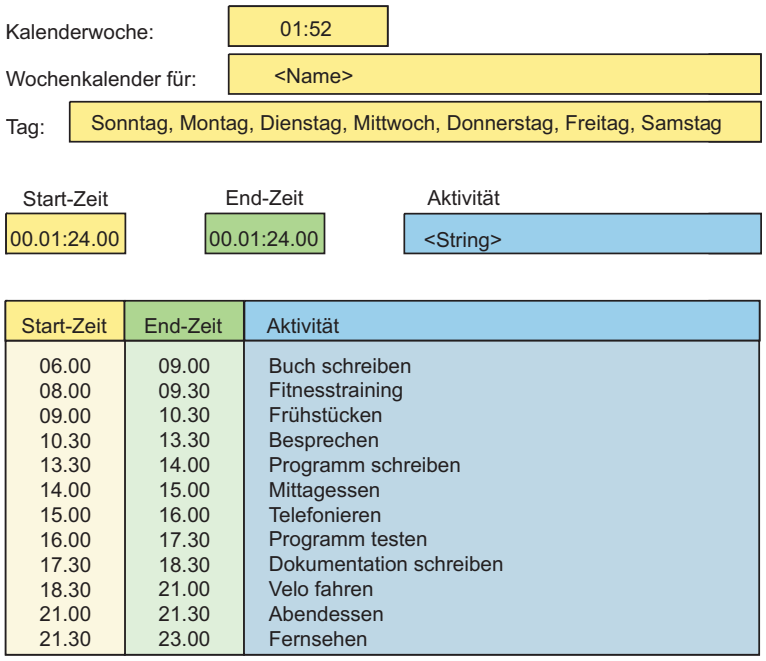
Aus diesen drei Client/Server-Interaktionen könnten folgende Client/Server-Testanforderungen abgeleitet werden:

1. Testen, ob alle Objekte auf dem Clientrechner ankommen.
2. Testen, was passiert, wenn die Verbindung zwischen Client und Server während der Download-Operation abbricht.
3. Testen, ob alle Objekte zum Server zurückkommen.
4. Testen, was passiert, wenn die Verbindung zwischen Client und Server während der Upload-Operation abbricht.
5. Testen, ob die Aktivitäten auf dem Projektserver ankommen.
6. Testen, was passiert, wenn die Verbindung zwischen Client und Projektserver abbricht.
7. Testen, was passiert, wenn der Serverrechner ausfällt, während Kalender in Arbeit sind.
8. Testen, was passiert, wenn der Projektserver während der Kalenderverarbeitung ausfällt.

Diese beispielhaften Client/Server-Testanforderungen sind nicht alle, die abzuleiten sind. Sie reichen jedoch aus, um zu zeigen, worum es hier eigentlich geht.

4.6.2.2 Test der Kalenderoberfläche

Die Benutzungsoberfläche zum verteilten Kalender ist ein einziges großes Fenster mit mehreren Unterfenstern (Abbildung 4.12). In der oberen linken Ecke gibt es zwei Textboxen, in denen der Besitzer des Kalenders seinen Namen und seine Personalnummer eingeben muss. Falls sie leer bleiben, ist der Kalender nicht identifizierbar. Wie bei allen Texten kann der Benutzer sie entweder mit einem Mausklick oder mit dem Tabulator positionieren.
Nach dem Namen und der Personalnummer kommt eine Box für die Wochenzahl. Der Benutzer muss hier eine Nummer von 1 bis 52 eintragen. Eine weitere Box ist für den Wochentag gedacht. Der Benutzer gibt eine Nummer von 1 bis 7 ein und das System wandelt sie in einen Wochentagnamen um und zeigt den Namen.



Start-Zeit	End-Zeit	Aktivität
06.00	09.00	Buch schreiben
08.00	09.30	Fitnesstraining
09.00	10.30	Frühstücken
10.30	13.30	Besprechen
13.30	14.00	Programm schreiben
14.00	15.00	Mittagessen
15.00	16.00	Telefonieren
16.00	17.30	Programm testen
17.30	18.30	Dokumentation schreiben
18.30	21.00	Velo fahren
21.00	21.30	Abendessen
21.30	23.00	Fernsehen

Abbildung 4.12 Kalenderoberfläche

Unter diesen Kopffeldern erscheinen fünf Kopffelder für die Eingabe einer Aktivität. Der Benutzer gibt in das erste Feld eine gültige Startzeit und in das zweite Feld eine gültige Endzeit ein. Das dritte Feld ist ein Pop-up-Menü mit vorgegebenen Aktivitätstypen, das vierte Feld ein Pop-up-Menü mit laufenden Projekten. Im ersten Fall muss der Benutzer einen Typ auswählen, im zweiten Fall nur dann ein Projekt auswählen, wenn die Aktivität projektbezogen ist. In das fünfte Feld ist ein

Text zur Beschreibung der Aktivität einzugeben. Es kann per Mausklick oder per Tabulator positioniert werden.

Im unteren Teil des Fensters gibt es ein schreibgeschütztes (read-only) Fenster für die Anzeige der bisher erfassten Aktivitäten. Auf der gleichen Zeile neben jeder Aktivität ist eine Box für das Aktionskennzeichen. Dieses enthält ein Pop-up-Menü mit zwei möglichen Aktionen – *ändern* oder *löschen*. Wenn der Benutzer die Aktion *löschen* wählt, wird diese Aktivitätenzeile entfernt. Wenn der Benutzer die Aktion *ändern* wählt, wird die Aktivität oben in den Editierfeldern angezeigt, damit der Benutzer sie ändern kann.

In der oberen rechten Ecke des Gesamtfensters ist ein Knopf, den der Benutzer anzuklicken hat, wenn er diesen Tag abschließen möchte. Danach wird das Fenster bis auf seinen Namen, seine Nummer und die Wochenzahl aufgefrischt. Ein zweiter Knopf erlaubt ihm, auch die Woche abzuschließen und eine neue Wochenzahl einzugeben. Der Kalender selbst wird beendet, wenn er das Gesamtfenster schließt.

Für den Test dieser Oberfläche stellen sich folgende Testanforderungen:

1. Eröffnung eines Kalenders,

9. Eingabe einer ungültigen Personalnummer,
10. Kalender ohne Personennamen,
11. Positionieren des Namens per Mausklick,
12. Positionieren des Namens per Tabulator,
13. Eingabe einer ungültigen Wochenzahl,
14. Eingabe einer ungültigen Tageszahl,
15. Eingabe einer falschen Startzeit,
16. Eingabe einer falschen Endezeit,
17. Eingabe einer Endezeit kleiner als die Startzeit,
18. Aktivität ohne Aktivitätentyp,
19. Aktivität ohne Aktivitätenbeschreibung,
20. Positionieren der Beschreibung per Mausklick,
21. Positionieren der Beschreibung per Tabulator,
22. Eingabe mehrerer Aktivitäten,
23. Löschen einer Aktivität,
24. Änderung einer Aktivität,
25. Schließung eines Tages,
26. Schließung einer Woche und

27. Schließung des Kalenders.

Daran lässt sich erkennen, welcher Aufwand damit verbunden ist, eine einfache graphische Oberfläche zu testen. Anspruchsvollere Oberflächen können leicht drei- bis viermal mehr Testanforderungen verursachen.

4.6.2.3 Test der Kalenderdatenbank

Für die Sicherung der Objekte des verteilten Kalenders wird eine relationale Datenbank benutzt. Wenn ein Objekt freigegeben wird, wie z.B. eine Aktivität, wird es in einzelne Attribute aufgelöst und als Zeile in die Tabelle eingefügt. Wenn der Anwender seinen Kalender wieder eröffnet, müssen alle Objekte auf dem Server – der Kalender, die Wochen, die Tage und die Aktivitäten – gebildet und auf den Clientrechner übertragen werden. Jeder Objekttyp wird in einer eigenen Tabelle gesichert. Somit gibt es vier SQL-Tabellen:

- Tabelle `Kalender` mit `Mitarbeiternummer` und `Mitarbeitername`
- Tabelle `Wochen` mit `Mitarbeiternummer` und `Wochenzahl`
- Tabelle `Tage` mit `Mitarbeiternummer`, `Wochenzahl`, `Tageszahl` und `Wochentag`
- Tabelle `Aktivitäten` mit `Mitarbeiternummer`, `Wochenzahl`, `Tageszahl`, `Startzeit`, `Endezeit`, `Aktivitätentyp` und `Aktivitätenbeschreibung`.

Um die Konsistenz und Sicherheit der Objekte zu gewährleisten, sind einige Rules und Constraints definiert. Die *Rules*:

- Es darf keine Aktivität ohne Tage geben.
- Es darf keinen Tag ohne Woche geben.
- Es darf keine Woche ohne Kalender geben.
- Es darf keinen Kalender ohne Mitarbeiter geben.

Falls ein Mitarbeiter gelöscht wird, wird auch sein Kalender samt Wochen, Tagen und Aktivitäten gelöscht. Dies wird von einer *stored Procedure* bewerkstelligt.
Neben den Regeln sind folgende *Constraints* vereinbart:

- Wochenzahlen nur von 1 bis 52.
- Tageszahlen nur von 1 bis 7.
- Maximal 12 Aktivitäten pro Tag.
- Zu jeder projektbezogenen Aktivität muss es ein entsprechendes Projekt geben.
- Ein Kalender darf nur von einem Anwender auf einmal bearbeitet werden.

Für den Test der Datenbank werden folgende Testanforderungen gestellt:

- Eintrag eines Kalenders mit mindestens einer Woche, einem Tag und einer Aktivität.
- Eintrag eines Kalenders mit mindestens einer Woche und einem Tag, aber ohne Aktivitäten.
- Eintrag eines Kalenders mit einer Woche ohne Tage.
- Eintrag eines Kalenders ohne Woche.
- Eintrag eines Kalenders ohne Mitarbeiter.
- Eintrag eines Kalenders mit einer ungültigen Woche.
- Eintrag einer Woche mit mehr als 7 Tagen.
- Eintrag eines Tages mit mehr als 12 Aktivitäten.
- Eintrag einer projektbezogenen Aktivität, für die es kein Projekt gibt.
- Löschung einer projektbezogenen Aktivität.
- Löschung einer nicht projektbezogenen Aktivität.
- Löschung einer Woche.
- Löschung eines Kalenders.
- Änderung einer projektbezogenen Aktivität.
- Änderung einer nicht projektbezogenen Aktivität.
- Lesezugriff auf einen Kalender, der bereits in Arbeit ist.
- Schreibzugriff auf einen Kalender, der bereits in Arbeit ist.

4.6.2.4 Test der Kalenderobjekte

Aus Sicht des Anwenders gibt es im verteilten Kalendersystem lediglich vier Anwendungsfälle:

- „Wochenkalender anlegen“,
- „Tag anlegen“,
- „Aktivität eingeben“ und
- „Aktivität löschen“.

Vom Anwendungsfall „Wochenkalender anlegen“ sind zwei Klassen betroffen:

- `Kalender` und
- `Woche`.

Vom Anwendungsfall „Tag anlegen“ sind drei Klassen betroffen:

- `Kalender`,
- `Woche` und
- `Tag`.

Vom Anwendungsfall „Aktivität eingeben“ sind fünf Klassen betroffen:

- `Kalender`,
- `Woche`,
- `Tag`,
- `Aktivität` und
- `Projekt`.

Vom Anwendungsfall „Aktivität löschen“sind die gleichen fünf Klassen wie vom Anwendungsfall „Aktivität eingeben“ betroffen. Zur Erzeugung stellvertretender Testobjekte für die betroffenen Klassen benutzt man z.B. die Äquivalenzklassenbildung oder die Grenzwertanalyse bzgl. der Attribute bzw. Instanzvariablen (vgl. [Bei95], [Win99]).

Für die Klasse `Kalender` gibt es drei stellvertretende Objekte:

- ein Angestellter,
- ein freier Mitarbeiter und
- eine illegale Person.

Für die Klasse `Woche` gibt es vier stellvertretende Objekte:

- eine Woche von 1 bis 52 ohne Feiertage,
- eine Woche von 1 bis 52 mit Feiertagen,
- eine Woche vor der 1. Woche und
- eine Woche nach der 52. Woche.

Für die Klasse `Tag` gibt es vier stellvertretende Objekte:

- ein Arbeitstag,
- ein Feiertag,
- ein Sonntag und
- ein ungültiger Tag.

Für die Klasse `Aktivität` gibt es vier stellvertretende Objekte:

- eine Aktivität mit ungültiger Startzeit,
- eine Aktivität mit ungültiger Endezeit,
- eine Aktivität ohne Bezeichnung und

- eine gültige Aktivität.

Für die Klasse `Projekt` gibt es drei stellvertretende Objekte:

- ein ungültiges Projekt,
- ein gültiges Projekt ohne Stunden, d.h. in der Planung und
- ein gültiges Projekt mit Stunden, d.h. in Arbeit.

Die zu testenden Operationen sind für die Klasse `Kalender`:

- den Kalenderinhaber prüfen,
- illegale Personen abweisen,
- einen Kalender anlegen,
- die akkumulierten Arbeitsstunden anzeigen,
- einen Kalender speichern und freigeben.

Für `Woche`:

- die Wochenzahl prüfen,
- ungültige Wochen abweisen,
- gültige Wochen anlegen,
- Wochenstunden summieren,
- Wochen speichern und freigeben.

Für `Tag`:

- den Wochentag prüfen,
- den Feiertag prüfen,
- ungültige Wochentage abweisen,
- gültige Wochentage anlegen
- Tagegsstunden summieren,
- Tage speichern und freigeben.

Für `Aktivität`:

- die Startzeit prüfen,
- die Endezeit prüfen,
- die Bezeichnung prüfen,
- die Anzahl prüfen,
- ungültige Aktivitäten abweisen,
- gültige Aktivitäten anlegen,

- das Vorhandensein eines Projekts prüfen,
- Stunden gegen ein Projekt buchen,
- vorhandene Aktivitäten löschen,
- Aktivitäten speichern und freigeben.

Die stellvertretenden Zustände der Objekte sind folgende.

Für `Kalender`:

- nicht vorhanden,
- vorhanden.

Für `Woche`:

- keine vorhanden,
- eine vorhanden,
- mehrere vorhanden,
- alle 52 vorhanden.

Für `Tag`:

- keiner vorhanden,
- einer vorhanden,
- mehrere vorhanden,
- Feiertag vorhanden,
- alle 7 vorhanden.

Für `Aktivität`:

- keine vorhanden,
- eine vorhanden,
- mehrere vorhanden,
- eine projektbezogene vorhanden,
- eine nicht projektbezogene vorhanden,
- maximale Anzahl vorhanden,
- mehr als maximale Anzahl vorhanden.

Repräsentative Interaktionen zwischen den Objekten sind:

- Erzeugung eines Kalenders durch den Benutzer,
- Erzeugung einer Woche durch den Kalender,
- Erzeugung eines Tages durch die Woche,

- Erzeugung einer Aktivität durch den Tag,
- Prüfung des Vorhandenseins eines Projekts durch die Aktivität,
- Buchung der Arbeitsstunden gegen das Projekt durch die Aktivität,
- Summierung der Arbeitsstunden aller Aktivitäten eines Tages durch den Tag,
- Speicherung und Freigabe der Aktivitäten durch den Tag,
- Summierung der Arbeitsstunden aller Tage durch die Woche,
- Speicherung und Freigabe der Tage durch die Woche,
- Summierung der Arbeitsstunden aller Wochen durch den Kalender,
- Speicherung und Freigabe der Wochen durch den Kalender.

4.6.3 Testansätze für den verteilten Kalender

Für den Test des verteilten Kalenders wird ein dreifacher Testansatz verfolgt:

- erst der Klassentest nach der Built-In Testtechnik,
- dann der Integrationstest nach der Bottom-Up Strategie,
- zum Schluss der Systemtest mit den Anwendungsfällen.

Für den Klassentest nach der Built-In Testtechnik wird in allen einzelnen Klassen bis auf die Dialogklasse eine Testoperation eingebaut, die alle anderen Operationen aufruft und mit Parametern versorgt. Fremde Operationen werden durch Stubfunktionen simuliert. Das Ziel des Klassentests sollte sein, die Verarbeitungslogik der Funktionen und die Zustandsübergänge der Objekte zu verifizieren.

Beim Integrationstest werden die Objekte bzgl. der Benutzungen von unten nach oben durch einen Testtreiber angelegt und ihre Funktionen ausgeführt. Für den Test der Verbindung zwischen Aktivitäten und Projekten wird eine CORBA-Schnittstelle generiert, über die stellvertretende Nachrichten gesendet werden. Es werden alle Objekttypen vom Kalender bis hinunter zur Aktivität erzeugt, um die Interaktionen zwischen Objekten zu testen. Dabei wird auch die Aufbewahrung der persistenten Objekte und deren Wiedergewinnung erprobt. Das Ziel des Integrationstests ist sowohl der Test der Objektverteilung im Netz als auch der Test der Datenbankzugriffe.

Beim Systemtest wird an der Benutzungsoberfläche mit sämtlichen Anwendungsfällen getestet. Außerdem werden alle fehlerhaften Angaben und alle Ausnahmebedingungen erprobt. Der Systemtest soll beweisen, dass alle spezifizierten Funktionen erfüllt sind und alle definierten Ausnahmeereignisse abgefangen werden. Es

wird nach der Methode der Grenzwertanalyse und der Äquivalenzklassen vorgegangen.

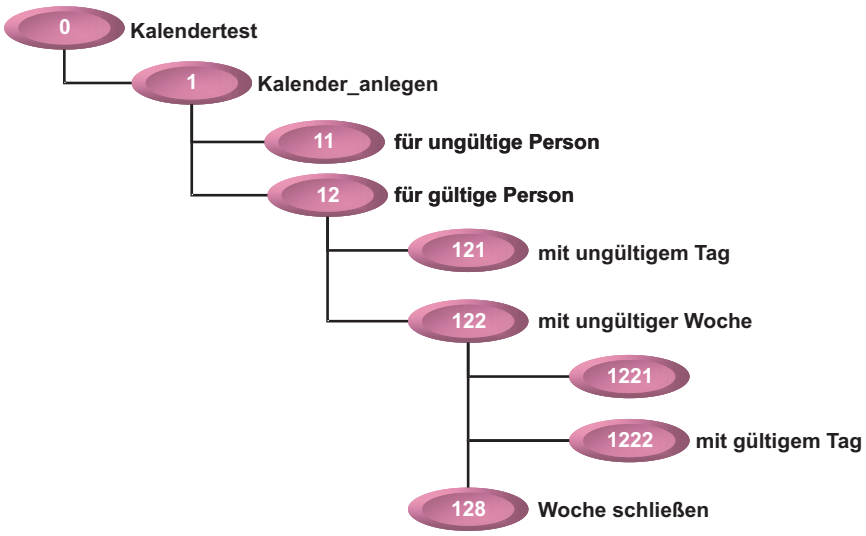


Abbildung 4.13 Testszenario für den Kalender

4.6.4 Testszenarien für den verteilten Kalender

Das Testszenario für den Klassentest sieht eine statische Analyse durch das Tool CPPANAL und eine dynamische Analyse durch das Tool CPPTEST vor. Jeder Entwickler ist verpflichtet, seine Source-Module durch CPPANAL prüfen und messen zu lassen. Die daraus folgenden Berichte muss er über das betriebsinterne E-Mail-System an die Qualitätssicherung senden.
Außerdem ist jeder Entwickler verpflichtet, seine Klassen durch CPPTEST automatisch testen zu lassen. Er soll durch Zusicherungen in den Testfunktionen die Ergebnisse seiner Funktionen kontrollieren. Es ist zu sichern, dass jede Operation aufgerufen und jede Bedingung erfüllt wird. Das Protokoll des Klassentests mit der Testüberdeckung ist an die Qualitätssicherung zu leiten. Die Module bzw. Klassen dürfen erst zu Integrationen freigegeben werden, wenn die Qualitätssicherung sie abgenommen hat. Das Testszenario für den Integrationstest baut auf dem Werkzeug IDLTEST auf. Mit IDLTEST werden folgende Fälle getestet:

- zwei Kalender erzeugen, speichern und wiedergewinnen,
- für jeden Kalender drei Wochen erzeugen, speichern und wiedergewinnen,
- für jede Woche einen Arbeitstag und einen Feiertag erzeugen, speichern und wiedergewinnen,

- für jeden Tag bis zu 12 Aktivitäten erzeugen, speichern und wiedergewinnen,
- für einen Tag 13 Aktivitäten erzeugen,
- mehrere Aktivitäten gegen ein Projekt buchen,
- eine Aktivität gegen ein nicht existierendes Projekt buchen,
- eine Aktivität ohne Tag erzeugen,
- einen Tag ohne Woche erzeugen,
- eine Woche ohne Kalender erzeugen,
- einen Kalender ohne Mitarbeiter erzeugen,
- eine Aktivität löschen,
- einen Kalender löschen,
- einen Mitarbeiter mit Kalender löschen,
- ein Projekt löschen,
- den Projektserver vom Kalenderserver abkoppeln und
- einen Kalender aktualisieren, der bereits in Arbeit ist.

Beispiele hierzu zeigen Abbildung 4.13 und Abbildung 4.14, eine Ein/Ausgabe-Matrix ist in Abbildung 4.15 dargestellt.

Vorher werden alle Klassen mit CPPINST instrumentiert, um die Testüberdeckung bis auf Zweigebene zu registrieren. Der Integrationstest soll von dem IDLTEST-Testtreiber auf einem Test-Clientarbeitsplatz gesteuert werden. Dieser Testtreiber wird die entsprechenden Operationsaufrufe absetzen.

Für den Systemtest werden Tester eingesetzt, um die Benutzungsoberfläche einmalig zu bedienen. Bei diesem einen Mal werden alle Bedienereingaben mit dem Tool Winrunner aufgezeichnet und nachher wieder eingespielt. Die Bedienungsanleitung lautet:

- anmelden,
- ungültige Mitarbeiter immer angeben,
- Mitarbeiter ohne Namen angeben,
- Nummer und Name angeben,
- ungültige Wochenzahl angeben,
- gültige Wochenzahl angeben,
- ungültige Tageszahl angeben,
- gültige Tageszahl angeben,

- ungültige Startzeit angeben,
- ungültige Endezeit angeben,
- ungültigen Aktivitätentyp angeben,
- Aktivität ohne Bezeichnung angeben,
- gültige Aktivität eingeben
- mehrere Aktivitäten eingeben,
- Aktivität ändern,
- Aktivität löschen,
- neuen Tag eingeben,
- mehr als 12 Aktivitäten erfassen,
- Kalender löschen und
- abmelden.

Dieses Szenario ist sicherlich nicht vollständig, aber es vermittelt den Eindruck, welcher Aufwand damit verbunden ist, eine Oberfläche mit allen Anwendungsfällen und allen fehlerhaften Angaben zu testen.

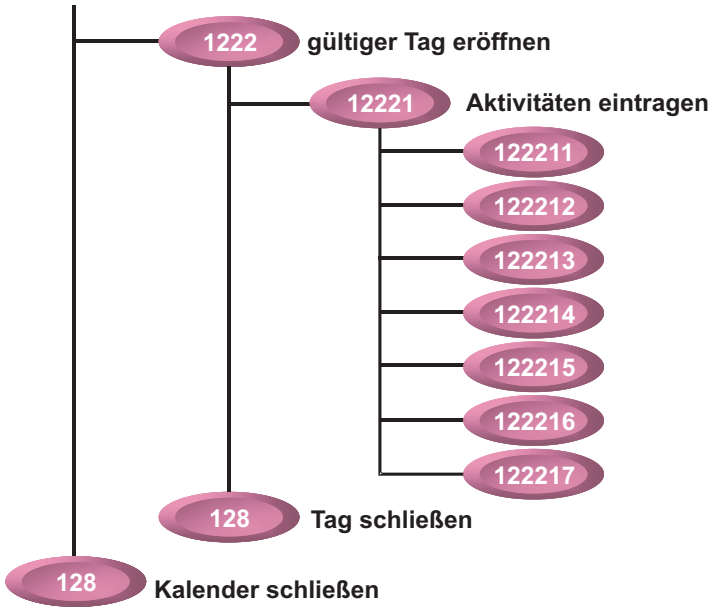


Abbildung 4.14 Weiteres Testszenario für den Kalender

		Testfälle											
Eingaben		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**Ausgaben**
Mitarbeiter	Ungültig	X											Fehlermeldung
	Gültig		X	X	X	X	X	X	X	X	X	X	Kalender anlegen
Woche	Ungültig		X										Fehlermeldung
	Gültig			X	X	X	X	X	X	X	X	X	Woche erzeugen
Tag	Ungültig			X									Fehlermeldung
	Gültig				X	X	X	X	X	X	X	X	Tag erzeugen
Aktivität	Falsch Startzeit					X							Fehlermeldung
	Gültige Starzeit						X	X	X	X	X	X	Weiter
	Falsche Endzeit						X						Fehlermeldung
	Gültige Endzeit					X		X	X	X	X	X	Weiter
	Falsche Aufgabe							X					Fehlermeldung
	Gültige Aufgabe					X	X		X	X	X	X	Weiter
	Ohne Projekt								X				Eintragen
	mit Projekt									X			Gegen Projekt buchen
	Zu viele										X		Fehlermeldung
	Löschen											X	Aktivität löschen

Abbildung 4.15 Eingabe/Ausgabe-Matrix

4.6.5 Testendekriterien für den verteilten Kalender

Für jede Phase des Kalendertests wird es eigene Endekriterien geben.
Der Klassentest ist beendet, wenn die Klasse

- eine allgemeine Komplexität < 0,6,
- eine allgemeine Qualität > 0,6,
- eine Konformität > 0,8,
- eine Zweigüberdeckung > 0,9,
- eine Zustandsüberdeckung > 0,85 und
- eine Bedingungsüberdeckung = 1,00 hat.

Der Integrationstest ist beendet, wenn die Komponente

- eine Objekttyperzeugungsüberdeckung > 0,90,
- eine Operationsaufrufüberdeckung > 0,80,
- eine Vererbungsüberdeckung > 0,90,
- eine Assoziationsüberdeckung > 0,90 und
- eine Schnittstellenüberdeckung > 0,99 hat.

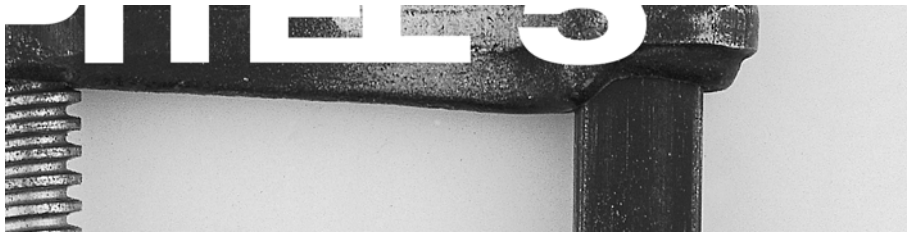
Der Systemtest ist beendet, wenn das System

- eine Funktionsüberdeckung > 0,99,
- eine Anwendungsfall Überdeckung > 0,99,
- eine Ausnahmeüberdeckung > 0,99,
- eine Testfallüberdeckung > 0,99 und
- eine Fehlerrate < 0,002 hat.

Darüber, wie diese Messwerte zu ermitteln sind, wird im neunten Kapitel eingegangen. Es genügt hier zu bemerken, dass wir quantitativ messbare Kriterien brauchen, um unseren Test beenden zu können.

5

Spezifikation objektorientierter Testfälle



Testfälle als regelbasierte Programme
Spezifikation der Testfälle
Ermittlung der Testfälle
Quellen der Testfälle
Konventionelle Testfallspezifikationsansätze
Das Besondere an der objektorientierten Testfallspezifikation
Einfluss der UML auf die Testfallspezifikation
Testfallspezifikation für den verteilten Kalender

Inhaltsübersicht Kapitel 5

5 Spezifikation objektorientierter Testfälle

5.1 Testfälle als regelbasierte Programme

In der ANSI/IEEE-Norm 829 [IEEE829] wird ein Testfall als eine einmalige Zusammenstellung von Datenwerten und deklarativen Anweisungen definiert, die darauf zielt, eine bestimmte Funktion oder einen bestimmten Pfad durch ein Programm auszuführen bzw. eine spezifische Anforderung zu bestätigen. Demnach ist ein Testfall so etwas wie ein Miniprogramm, allerdings ein deklaratives, denn ein prozeduraler Testfall wäre ein Widerspruch in sich. So wie z.B. die Sprachen PROLOG und SQL Ergebnisse aus verschachtelten Regeln oder Relationen zwischen vorhandenen Argumenten und Zwischenergebnissen ableiten, definieren Testfälle Ergebnisse im Bezug zu einzelnen Argumenten und Prädikaten. Somit ist Testfallspezifikation eine typische Anwendung der regelbasierten Programmierung [How80].
In der einfachsten Form reduziert sich ein Testfall auf die Funktionsvereinbarung bzw. den „Zustandsübergang“ (Abbildung 5.1)

$$x = f(y)$$

wobei x das Ergebnis, y das Argument und f: Y → X die Funktion ist. X ist hier stellvertretend für den gültigen Ausgabewertebereich und Y für den erwarteten Eingabewertebereich [Zel90]. Es gilt, einen Eingabewert $y \in$ Y zu generieren und den Ausgabewert $x \in$ X zu validieren. Die Funktion f ist der Softwarebaustein unter Test, d.h. der Testgegenstand. Der Testgegenstand könnte eine Methode, eine Klasse, eine Klassengruppe, eine Komponente, ein Teilsystem oder ein ganzes System sein, je nachdem, auf welcher Granularitätsstufe getestet wird.

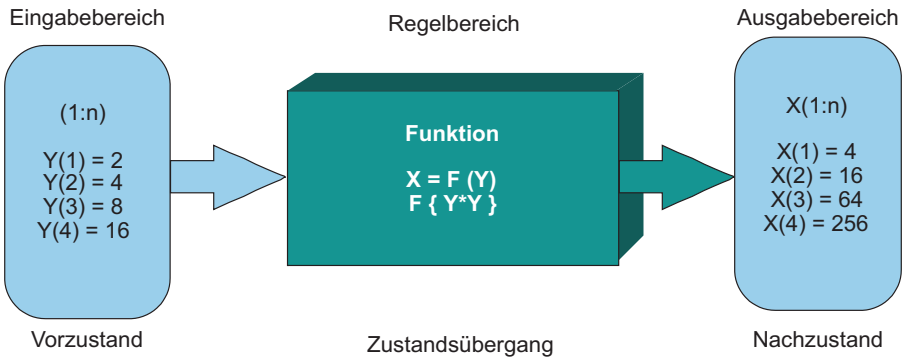


Abbildung 5.1 Testfälle als Zustandsübergänge

Am einfachsten ist der Testfall auf der kleinsten Granularitätsstufe zu verstehen und zu formulieren. Hier ist *y* in der Regel eine Reihe von Parameterwerten und *x* eine Reihe von Rückgabewerten so wie im Beispiel

```
x = Mittelwert(10, 20, 30);
```

Die Funktion `Mittelwert()` müsste das Ergebnis 20 zurückgeben. Wenn dies so ist, ist der Testfall eine Behauptung über die Funktion in Bezug auf die Vor- und Nachzustände [Hay86].

```
Mittelwert:
          Assert post x = 20;
          if pre y (1:3) = 10, 20, 30;
```

Dabei ist es unwichtig zu wissen, wie die Mittelwertfunktion implementiert ist. Wichtig ist nur zu wissen, was rein und raus geht.

Testfälle auf einer höheren Granularitätsstufe sind prinzipiell ähnlich zu formulieren, nur besitzen sie wesentlich mehr Argumente und Ergebnisse. Um zu testen, ob eine bestimmte Transaktion für eine Banküberweisung korrekt funktioniert, müsste man die erwartete Überweisung mit allen Attributen beschreiben, so z.B.:

```
testcase ueberweisung_01:
   Assert post
         Empfängerkonto.Saldo = Saldo@pre + 600,00,
         if pre Sendername = „Sneed, Harry“,
             Senderkonto = „19404712“,
             Überweisungsbetrag = 600,00,
             Empfängername = „Sneed, Stephan“,
             Empfängerkonto = „19774711“,
             Empfängerbank = „KSK_Miesbach“,
             Empfänger_blz = „71152570“,;
```

Daraus wird ersichtlich, dass Testfälle grundsätzlich Relationen zwischen Eingangs- und Ausgangsdaten darstellen und so eine bestimmte Funktionalität be-

schreiben. Je nachdem, auf welcher Granularitätsstufe getestet wird, könnte diese abstrakt beschriebene Funktionalität

- eine Operation,
- eine Reihe von Operationen,
- eine Schnittstelle,
- eine Komponente oder
- eine Systemtransaktion bzw. ein Anwendungsfall sein [Yam98].

Demzufolge lassen sich Testfälle nach der Art des Testobjekts wie in Abbildung 5.2 dargestellt klassifizieren in

- Klassentestfälle,
- Integrationstestfälle und
- Systemtestfälle [Sie96].

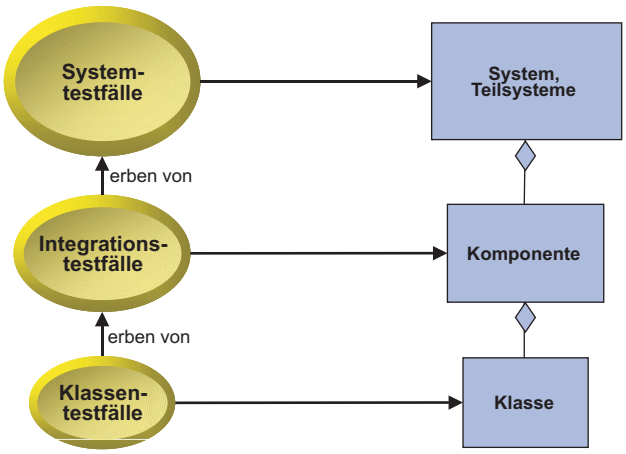


Abbildung 5.2 Testfallhierarchie

5.1.1 Klassentestfälle

Auf der Klassenebene ist ein Testfall ein Methoden- bzw. Operationsaufruf, und Testdaten stellen konkrete Werte für die Ein-, Aus- und Zustandsvariablen dar, die für die betrachtete Operation relevant sind. Möglicherweise verursacht dieser Operationsaufruf den Aufruf weiterer Operationen, d.h. es wird eine Operationskette ausgelöst. Wichtig für den Klassentest ist, dass die aufgerufenen Operationen in der Klasse unter Test definiert sind. Entweder gehören sie zur Klasse selbst – in der Klasse neu definierte Operationen (member functions) – oder sie gehören zu einer

Oberklasse – geerbte Operationen (inherited functions). Operationen in fremden Klassen dürfen nicht betroffen werden. Wie das zu verhindern ist, ist Sache des Klassentestrahmens.

Die Argumente eines Klassentestfalls sind die Eingangsparameter der aufgerufenen Operationen, der Attribute des Zielobjekts und die Zustände der geerbten Attribute. Die Ergebnisse sind die veränderten Attribute des Zielobjekts und die Rückgabewerte bzw. Ausgangsparameter. Insofern, als die Klasse unter Test nicht auf eine Datenbank zugreift, sind alle Argumente im Hauptspeicher. Sie müssen dort auch versorgt werden. Das Gleiche trifft für die Ergebnisse zu. Sie bleiben unsichtbar im Hauptspeicher und müssen dort kontrolliert werden, entweder durch eine visuelle Anzeige, eine Protokollierung oder einen internen Abgleich.

Vor dem Methodenaufruf werden die Attribute initialisiert und die Eingangsparameter gesetzt. Während der Methodenausführung werden die Ergebnisse fremder Methoden künstlich erzeugt, um den Test auf die eine Klasse zu beschränken. Nach dem Methodenaufruf werden die veränderten Objektzustände abgeglichen und die Ausgangsparameter auf ihre Übereinstimmung mit den erwarteten Ergebnissen kontrolliert. Also muss der Klassentestfall

- die Methode identifizieren, die anzustoßen ist;
- die betroffenen Objektzustände initialisieren;
- die Eingangsparameter setzen;
- die Ergebnisse fremder Methoden vorgeben;
- die veränderten Objektzustände voraussagen und
- die Ausgangsparameter bestimmen [HaWi93].

Normalerweise werden die Klassentestfälle in einer Klassentestprozedur bzw. einem Testskript zusammengefasst, denn eine Klasse hat mehrere Methoden und jede Methode wird mehrere Testfälle haben. So betrachtet ist die Klassentestprozedur ein Baum der Klassentestfälle mit der Struktur:

$$\texttt{Klasse}\ \{\texttt{Objekt}_{1\text{-}n}\ [\texttt{Methode}_{1\text{-}n}(\texttt{Testfall}_{1\text{-}n})]\}$$

Die Beschreibung der Klassentestfälle sollte möglichst an die Zielsprache, z.B. C++, Java oder Objekt-COBOL angelehnt sein, damit beide Sprachen – Programmiersprache und Testsprache – möglichst kompatibel und die Entwickler nicht gezwungen sind, mit zwei völlig unterschiedlichen Sprachen umzugehen. Die Anzahl der Klassentestfälle kann leicht mehrere hundert pro Klasse betragen, wenn es darum geht, alle Logikzweige in allen Methoden für alle Objektausprägungen auszuprobieren [MTW94]. Die Spezifikation der Klassentestfälle kommt einem zweiten Programm gleich. Dieses zweite Gegenprogramm lässt sich leichter formulieren, wenn es in Syntax und Semantik dem ersten Programm ähnelt.

5.1.2 Integrationstestfälle

Auf der Stufe des Integrationstests sind Testfälle in der Regel Nachrichten von einer Klasse bzw. Komponente an eine andere. Auch hier werden bestimmte Methoden angesteuert, aber die angesteuerten Methoden gehören nicht nur einer Klasse an. Sie sind verteilt auf mehrere Klassen einer Komponente. Somit ist der Wirkungsbereich eines Integrationstestfalls größer als der eines Klassentests.

Der Integrationstestfall baut auf einer Schnittstelle auf und zielt auf eine Klassenmenge bzw. eine Komponente. Die Argumente sind die Parameter in der Schnittstelle, die globalen Daten bzw. die persistenten Objekte und nicht zuletzt die Inhalte der Datenbanken, auf welche die Komponente zugreift. Simuliert wird nur die Schnittstelle zu der Zielkomponente [NaSa92].

Integrationstestfälle müssen aufeinander aufbauen, um die Initialisierung der internen Datenzustände einer Komponente bei jedem Testfall zu vermeiden. Denn hier handelt es sich bereits um mehrere hundert Objektattribute zuzüglich der Parameter-Werte als Argumente. Das Gleiche trifft für die Ergebnisse zu. Hunderte von Objekten können durch einen einzigen Testfall verändert werden. Es wird kaum möglich sein, alle veränderten Zustände zu kontrollieren. Deshalb beschränkt sich der Integrationstestfall auf die Prüfung der Rückgabewerte.

Im Gegensatz zu den Klassentestfällen, die oft in beliebiger Reihenfolge ausgeführt werden können, müssen die Integrationstestfälle meistens in einer bestimmten Reihenfolge erfolgen. Z.B. muss ein Konto zuerst eröffnet werden, ehe der erste Betrag eingezahlt wird, und der erste Betrag muss eingezahlt werden, ehe ausgezahlt wird. Auf diese Weise baut die Komponente ihre eigenen internen Zustände auf. Falls jedoch die Komponente auf bestehende Datenbanken zugreift, sind die Inhalte jener Datenbanken Argumente für die Komponente und müssen vorher generiert werden. Es kann also vorkommen, dass der Integrationstestfall auch Datenbankattribute betrifft, so wie im folgenden Beispiel:

```
testcase ueberweisung_04:
           Assert post
                  Überweisung.Empfängername = „Sneed, Stephan“,
                  Überweisung.Empfängerkonto = „10774711“,
                  Überweisung.Empfängerbank  = „KSK_Miesbach“,
                  Überweisung.Datum = heutiges Datum,
                  Überweisung.Betrag = 620,00
           if pre Schnittstelle.Sendername = „Sneed, Harry“,
                  Schnittstelle.Senderkonto = 19404712“,
                  Schnittstelle.Senderblz = „71152570“,
                  Schnittstelle.Betrag = 620,00,
                  Datenbank.Senderbank = KSK_Miesbach,
                  Datenbank.Kontobetrag > 620,00;
                  Datenbank.Kontotyp = Giro;
```

Daran ist zu erkennen, dass sich ein Teil der Argumente in der Schnittstelle und ein anderer Teil in der Datenbank befindet. Beide Teile müssen vorbelegt werden, um die erwartete Überweisung zu produzieren.

5.1.3 Systemtestfälle

Klassen- und Integrationstestfälle sind interne Testfälle, die sich auf die Software-Konstruktion beziehen. Daher setzen sie gewisse Kenntnisse der Implementierung, im Fall der Klassentestfälle Kenntnisse der Klassenkonstruktion, im Fall der Integrationstestfälle Kenntnisse der Komponentenschnittstellen voraus. Systemtestfälle sind hingegen externe Testfälle, bei denen das System von außen über die Benutzungsschnittstelle betrachtet wird. Die Argumente sind im Falle von Dialog-Transaktionen die Kontrolltasten, Mausklicks und Textfelder, die der Benutzer eingibt. Die Ergebnisse sind die Textfelder, die angezeigt werden, die Ereignisse, die ausgelöst werden, und die Veränderungen zu den Datenbanken. Bei Batch-Transaktionen sind die Argumente die Bewegungsdaten, die über eine Systemeingabeschnittstelle einlaufen und eventuell die Parameter, die von irgendeinem Gerät gelesen werden. Die Ergebnisse sind die Exportdateien, die erzeugten Berichte und die veränderten Datenbanken [Bei95].

Systemtestfälle sind auf der einen Seite sehr komplex, weil sie viele Daten in vielen verschiedenen Bereichen betreffen – die Benutzungsoberfläche, die internen Schnittstellen, die persistenten Objekte, die Datenbanken, die Import/Export-Dateien und nicht zuletzt die Berichte. Auf der anderen Seite sind sie einfacher zu formulieren, weil ihre Daten sichtbar sind. Im Gegensatz zum Klassentest kann der Tester sehen, was in das System hineingeht und was herauskommt.

Systemtestfälle beschreiben, welche Ausgaben zu erwarten sind, wenn gewisse Eingaben getätigt werden. Sie stellen eine Beziehung her zwischen dem, was der Benutzer des Systems eingibt, was in der Datenbank vorgespeichert ist und was das System ausgibt bzw. macht. Sie werden in einem Testskript erfasst, das zur Laufzeit entweder von einem menschlichen Tester oder von einem Testautomaten interpretiert wird. Von einem solchen Systemtestfall können zahlreiche Einzelfunktionen und etliche hundert Einzeldaten betroffen werden. Daher kommt es darauf an, nur jene Ergebnisse zu kontrollieren, die Aufschluss über die Korrektheit der Transaktion geben, z.B. die Rückmeldungen am Bildschirm, oder die Transaktionsprotokolle.

5.2 Spezifikation der Testfälle

Die ANSI/IEEE-Norm 829 schreibt folgende Struktur einer Testfallspezifikation vor [IEEE829]:

- Testfallkennzeichen,
- Testfallobjekte,
- Testfallargumente,
- Testfallergebnisse,
- Testfallumgebung,
- Testfallbedingungen und
- Testfallabhängigkeiten.

Das *Testfallkennzeichen* ist eine Bezeichnung, die diesen Testfall von allen anderen unterscheidet, z.B.:

```
Konto_07
```

Die *Testfallobjekte* sind die Klassen, Schnittstellen, Komponenten und Teilsysteme, auf die der Testfall sich bezieht. Ein Klassentestfall bezieht sich auf die Klasse unter Test sowie auf die betroffenen Objektinstanzen, z.B.:

```
Konto
   19404711
   19404712
   19404715
```

Ein Integrationstestfall bezieht sich auf die Komponenten und deren Schnittstellen, z.B.:

```
Kontoführung
   Kontoeröffnung
   Einzahlung
   Auszahlung
   Überweisung
```

Ein Systemtestfall bezieht sich schließlich auf ein ganzes oder ein Teilsystem z.B.:

```
Zahlungsverkehr
```

Die *Testfallargumente* sind die Eingabedaten für den jeweiligen Fall. Sie werden entweder als Konstante bzw. mit einem festen Wert oder als Variable bzw. mit einer Referenz auf ein anderes Attribut angegeben, z.B.:

```
Assert pre Kontostand = - 1 012,44,
           Kontonummer = „19404711“,
           Kontoinhaber = „Sneed, Stephan“,
           Datum = Heutiges_Datum;
```

In Worten: Der Kontostand wird mit einem festen Wert besetzt und das Datum bekommt den Wert einer internen Variablen zugewiesen. Die Summe der spezifizierten Argumente ergibt den Vorzustand bzw. die Vorbedingung des Testfalls.

Die *Testfallergebnisse* sind die Ausgabedaten für den jeweiligen Testfall. Auch sie werden als Konstanten oder Variablen definiert, wobei hier auch Wertebereiche oder Wertebeziehungen spezifiziert werden können, z.B.:

```
Assert post Kontostand > Kreditlimit,
            Datum = Heutiges_Datum,
            Zinsrate = 5,5 : 9,5;
```

Kontostand wird hier mittels einer Beziehung spezifiziert, Datum ist eine variable Zuweisung und Zinsrate ist ein Wertebereich. Die Summe der spezifizierten Ergebnisse ergibt den Nachzustand bzw. die Nachbedingung des Testfalls.

Die *Testfallumgebung* ist eine Beschreibung der Hardware- und Software-Bedingungen, unter denen dieser Testfall gilt, z.B.:

- eine 640 BPSI-Bildschirmauflösung,
- ein HP Laserdrucker und
- ein Visibroker Request Broker.

Sie können als Bedingung dem Testfall angehängt werden:

```
Where {$Auflösung = „640“;
            $ORB = „Visibroker“;};
```

Die *Testfallbedingungen* sind Verweise auf andere Datenzustände, die herrschen müssen, damit der Testfall gültig wird: z.B.

```
Assert post Meldung = „Auszahlung nicht möglich“;
            if (Kontostand < Kreditgrenze);
```

Hier ist die Ergebnismeldung abhängig von der Bedingung, dass der Kontostand die Kreditgrenze unterschritten hat.

Die *Testfallabhängigkeiten* sind letztlich Verweise auf andere Testfälle, die vor, nach oder neben diesem Testfall ausgeführt werden müssen, z.B.:

```
Konto_07:
            if (Konto_06);
```

Das heißt, der Testfall `Konto_07` darf erst nach dem Testfall `Konto_06` gestartet werden. Es besteht eine prozedurale Abhängigkeit zwischen den beiden Testfällen.

Eine umfassende Testfallspezifikation, in der alle Strukturelemente der ANSI/IEEE-Norm vorkommen, könnte wie folgt aussehen:

```
testcase: Konto_07                  //Testfallkennzeichen
            if (Konto_06)                   //Testfallabhängigkeit
            where ($ORB = „Visibroker“)     //Testfallumgebung
            for (class = Konto,             //Testfallobjekte
            object = 19404711)
        {Assert post
        //Testfallergebnis
```

```
    Meldung = „Auszahlung nicht möglich“
//Testfallbedingung
    if (Kontostand < Kreditgrenze);
 Assert pre
    Kontostand = - 1 900,00;
    //Testfallargumente
  Assert pre Kreditgrenze = 2 000,00;
  Assert pre Auszahlungsbetrag = 150,00;
};
```

5.3 Ermittlung der Testfälle

Klassische Darstellungsmittel für die Ermittlung und Spezifikation der Testfälle sind

- Ursache-Wirkungs-Graphen,
- Entscheidungsbäume,
- Entscheidungstabellen und
- Testmatrizen.

Ursache-Wirkungs-Graphen (cause & effect graphs) wurden schon Anfang der 70er Jahre bei IBM für Testzwecke eingesetzt [Mye79]. Sie beschreiben logische Beziehungen zwischen Eingaben und Ergebnisse mit den Booleschen Operatoren – und, oder, nicht. Annotierte Pfeile verbinden die Eingabeknoten über Zwischenergebnisknoten mit den Endergebnisknoten. Insofern als die Anzahl der Ein- und Ausgaben nicht allzu hoch wird, sind sie ein übersichtliches und einprägsames Mittel der Testfallermittlung, wie das folgende Beispiel zeigt (Abbildung 5.3).

In dem Beispiel geht es darum, einen Fahrpreis auszurechnen, und zwar abhängig von der Entfernung, der Fahrklasse, der Zugart, dem Fahrgastalter, dem Wochentag und der Uhrzeit. Diese sind die Eingaben oder Vorbedingungen. Die Ausgaben sind die alternativen Preise je nach Tag und Uhrzeit. Demnach gebe es drei alternative Preise:

- ein Preis für die Fahrt am Werktag zur Hauptverkehrszeit,
- ein Preis für die Fahrt am Werktag außerhalb der Hauptverkehrszeit und
- ein Preis für die Fahrt an Sonn- und Feiertagen.

Mit Berücksichtigung der möglichen fehlerhaften Eingaben kommen noch vier Fehlermeldungen als Ergebnis hinzu:

- ungültige Fahrklasse,
- unbekannte Zugart,

- unplausibles Alter und
- ungültiger Wochentag.

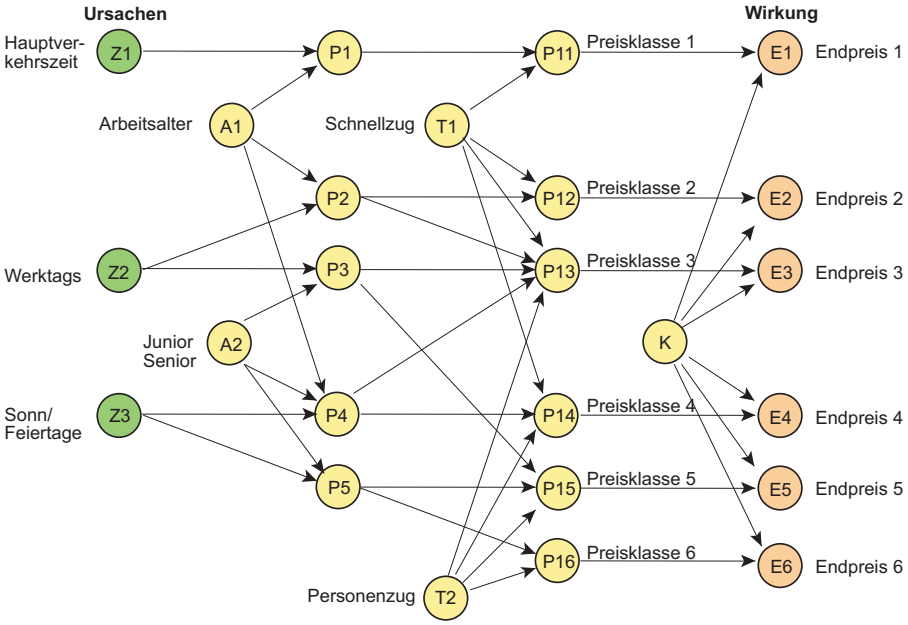


Abbildung 5.3 Ursache-Wirkungs-Graph

Demnach gäbe es 7 Testfälle, einen für jedes definierte Ergebnis. Zu bemerken ist, dass Ursache-Wirkungs-Graphen von den Ergebnissen ausgehen, also sozusagen „Ausgabeorientiert“ (output driven) sind.

Entscheidungsbäume waren bereits ein Mittel der strukturierten Analyse vom Yourdon und DeMarco [You89]. Darin sind die Knoten des Baumes die Entscheidungen und die Kanten die Entscheidungsausgänge. Dies entspricht der klassischen prozeduralen Sicht der Entscheidungslogik. Die Testfälle sind die Wurzel des Baumes bzw. die terminalen Entscheidungsausgänge. Wie bei den Ursache-Wirkungs-Graphen dienen Entscheidungsbäume hauptsächlich dazu Testfälle zu ermitteln. Sie entsprechen in etwa dem Aktivitätsdiagramm der UML. Abbildung 5.4 zeigt einen Entscheidungsbaum für das Fahrpreis-Beispiel.

Entscheidungstabellen wurden ebenfalls sehr früh für Testzwecke eingesetzt. [SeCh80] Sie leisten das Gleiche wie Entscheidungsbäume, sind aber kompakter und vor allem vollständiger (Abbildung 5.5). Mit einer solchen Tabelle ist es weniger wahrscheinlich, Testfälle auszulassen. Hinzu kommt, dass sie auch maschinell verarbeitbar sind. Falls die Namen der Bedingungsoperanden mit den Namen der Daten im Programm übereinstimmen, können aus den Entscheidungstabellen Test-

daten oder sogar komplette Testtreiber abgeleitet werden. Es gibt bereits Testwerkzeuge, die aus Entscheidungstabellen Testprozeduren generieren.

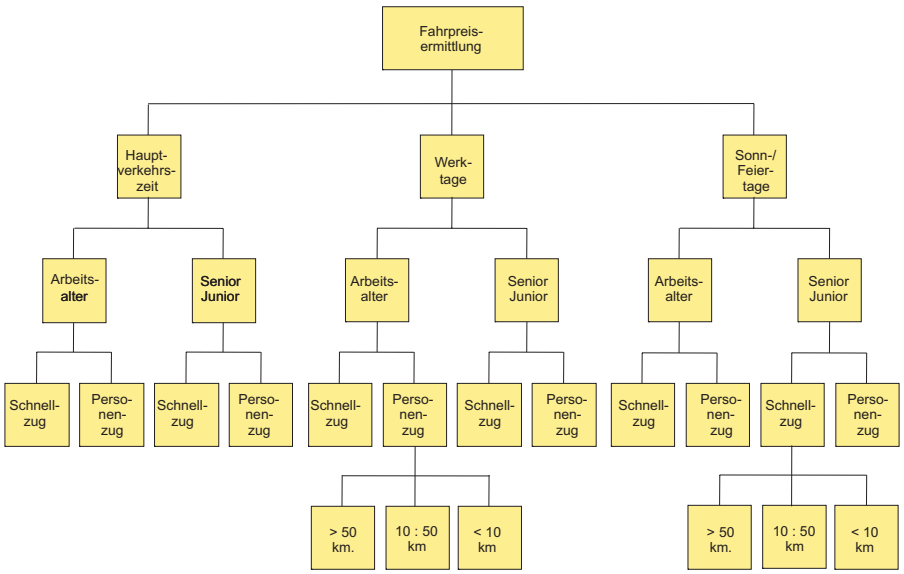


Abbildung 5.4 Entscheidungsbaum

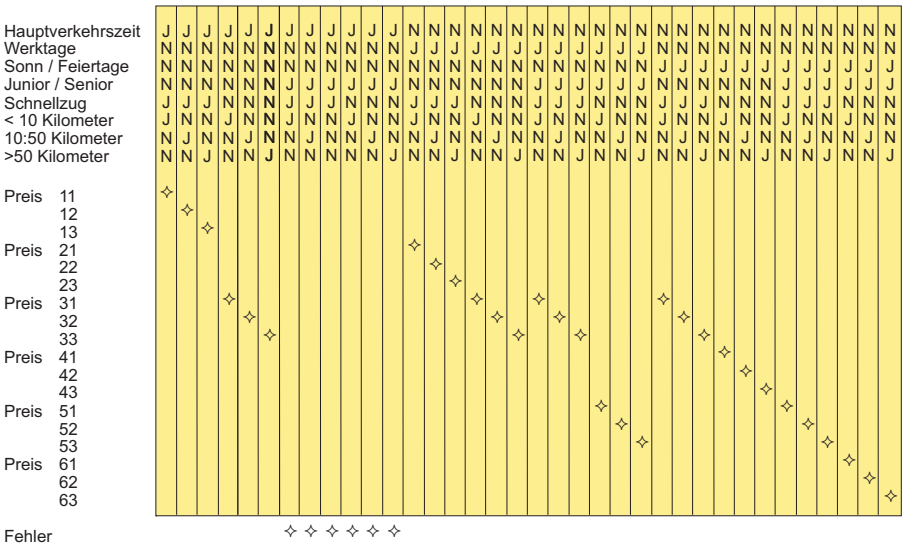


Abbildung 5.5 Entscheidungstabelle

Testmatrizen sind ein neuerlicher Beitrag vom Robert Binder [Bin93]. Sie sind zweidimensionale Tabellen, in der die eine Dimension die Datenpartitionen bzw. Äquivalenzklassen enthält und die andere die Testfälle mit stellvertretenden Werten darstellt (Abbildung 5.6). Derartige Matritzen sind vor allem dann zu empfehlen wenn es darum geht, Datenbanken und Masken für den Systemtest zu generieren. Wichtig ist die Verbindung der Maskenfelder mit den Attributen der Datenbank. Sie müssen aufeinander abgestimmt sein, z.B. dass die Zugart, die am Bildschirm gewählt wird, ein Element der Menge aller Zugarten in der Datenbank sein muss. Diese Abstimmung des Tests der Benutzungsoberfläche mit dem Datenbankinhalt ist eine der größten Herausforderungen des Systemtests.

			Testfälle						Testfall	Preisklasse	Zuordnung
		Kilometer	0	1	9	10	50	80	TF PK	TF PK	TF PK
Hauptverkehrszeit	Arbeitsalter	Schnellzug	1	2	3	4	5	6	2/3=11	4/5 = 12	6=13
		Personenzug	7	8	9	10	11	12	8/9=31	10/11=32	12=33
	Senior/Junior	Schnellzug	13	14	15	16	17	18	Fehler	Fehler	Fehler
		Personenzug	19	20	21	22	23	24	Fehler	Fehler	Fehler
Werktage	Arbeitsalter	Schnellzug	25	26	27	28	29	30	26/27=21	27/28=22	30=23
		Personenzug	31	32	33	34	35	36	32/33=31	34/35=32	36=33
	Junior/Senior	Schnellzug	37	38	39	40	41	42	38/39=31	40/41=32	42=33
		Personenzug	43	44	45	46	47	48	44/45=51	46/47=52	48=53
Sonn- und Feiertage	Arbeitsalter	Schnellzug	49	50	51	52	53	54	50/51=31	52/53=32	54=33
		Personenzug	55	56	57	58	59	60	56/57=41	58/59=42	60=43
	Senior/Junior	Schnellzug	61	62	63	64	65	66	62/63=51	64/65=52	66=53
		Personenzug	67	68	69	70	71	72	68/69=61	70/71=62	72=63

Abbildung 5.6 Testfallmatrix

5.4 Quellen der Testfälle

Testfälle lassen sich aus unterschiedlichen Quellen ableiten, z.B. aus dem Fachkonzept, aus den Anwendungsfällen, aus dem Objektmodell, aus dem Benutzerhandbuch oder aus dem Quellcode selbst. Ausschlaggebend für die Wahl der Quelle ist die semantische Ebene, auf der getestet werden soll. Es wäre genauso absurd, Systemtestfälle aus dem Quellcode ableiten zu wollen, wie Klassentestfälle aus dem Benutzerhandbuch. Klassentestfälle lassen sich nur aus der Klassenspezifikation ableiten, z.B. aus den Klassendiagrammen oder aus der Methodenbeschreibung. Integrationstestfälle sind aus der Schnittstellenspezifikation, bzw. aus den Sequenzdiagrammen und Kollaborationsdiagrammen abzuleiten. Systemtestfälle beziehen sich wiederum auf das Fachkonzept, die Anwendungsfälle, die Bedienungsanleitung oder gar auf das operationale Profil. So etwas wie einen allgemeingültigen generischen Testfall gäbe es nur, wenn die Klasse gleich der Komponente und die Komponente gleich dem System ist.

Testfälle werden im Allgemeinen aus den Spezifikationen der Testgegenstände abgeleitet, denn ein Test ist immer ein Test gegen „Etwas“ (Abbildung 5.7).

- Der Klassentest ist ein Test der Klasse gegen die Klassenspezifikation. Ergo werden die Klassentestfälle aus der Klassenspezifikation abgeleitet.
- Der Integrationstest ist ein Test der Systemarchitektur – Komponente und deren Schnittstellen – gegen die Spezifikation der Systemarchitektur. Ergo werden die Integrationstestfälle aus dem Architekturmodell abgeleitet.
- Der Systemtest schließlich ist ein Test des Systems gegen die Systemanforderungen. Systemtestfälle werden daher aus der funktionalen Spezifikation des Systems bzw. aus dem Fachkonzept oder den Anwendungsfällen abgeleitet.

Natürlich können Testfälle der unteren semantischen Ebene als Testfälle in die höhere semantische Ebene übernommen werden. Aus einigen Klassentestfällen werden Integrationstestfälle und aus einigen Integrationstestfällen werden Systemtestfälle, aber – wie in der objektorientierten Programmierung – es werden nicht alle Testfälle einer Ebene weiter vererbt und auf jeder Ebene kommen zusätzliche Testfälle hinzu. Daraus ergibt sich eine gewisse Vererbungshierarchie der Testfälle.

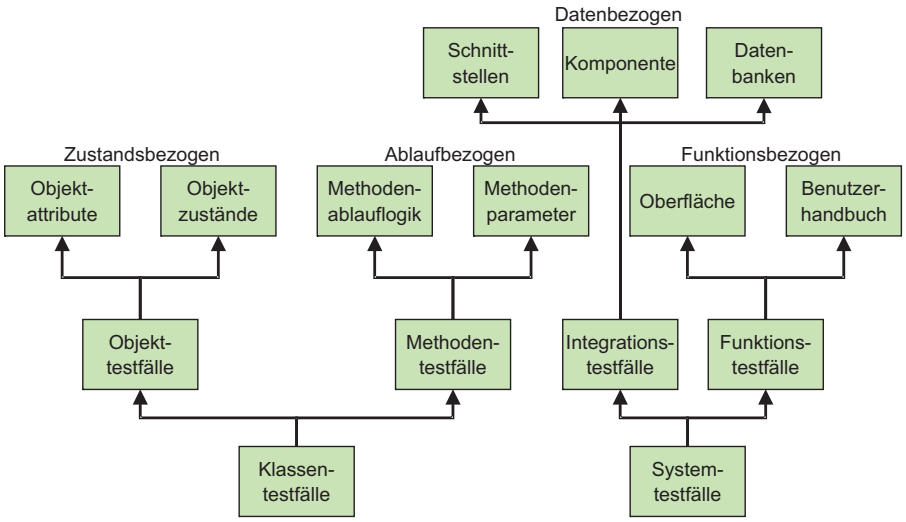


Abbildung 5.7 Ableitung der Testfälle

5.4.1 Spezifikation der Klassentestfälle

Klassentestfälle werden, wie oben erwähnt, aus der Klassenbeschreibung abgeleitet. Diese besteht wiederum aus Attribut- und Methodenbeschreibungen. Eine Attribut-

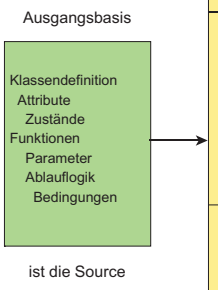
beschreibung bezieht sich auf die erlaubten Wertebelegungen bzw. Zustände eines Attributs. Die Methodenbeschreibung bezieht sich auf die Methodenparameter und die Ablauflogik. Klassentestfälle testen also die Relation zwischen Objektzuständen, Eingaben, Ausgaben und den Regeln der Methoden.

Zur Spezifikation der Klassentestfälle sind Kenntnisse der Klassenimplementierung erforderlich. Darum spricht man hier von einem White-Box Test. Eine Klasse hat Attribute, die zu belegen, und Methoden, die auszuführen sind. Ausgelöst werden sie über eine Parameterschnittstelle. Um die Testfälle für den Klassentest zu spezifizieren, muss der Tester zumindest diese interne Schnittstelle kennen. Darüber hinaus ist es gut zu wissen, wie die Methoden der Klassen zusammenhängen, d.h. welche Methode welche anderen Methoden aufruft. Diese Information ist nötig, um die Reihenfolge der Testfälle zu planen. Schließlich soll der Tester wissen, welche Rückgabewerte bzw. Ausgangsparameter zu erwarten sind und welche Objektzustände erzeugt werden, damit er sie kontrollieren kann.

Der Klassentest ist zwar wegen der vielen externen Abhängigkeiten schwer durchzuführen, aber relativ leicht zu spezifizieren. Im Prinzip muss man nur alle Pfade vom Eingang bis zum Ausgang jeder Methode verfolgen und die Parameter und Objektattribute so setzen, dass alle Pfadprädikate erfüllt werden. Ist die Ablauflogik nicht allzu tief, wird dies zu bewältigen sein. Schwieriger wird es im nächsten Schritt, wenn man versucht, eine Methodenfolge zu testen. Hier kommt es nämlich darauf an, alle Aufrufe von einer Methode zur anderen innerhalb der gleichen Klasse, aber mit unterschiedlichen Objekten zu verfolgen und für jede Methodenkette einen Testfall zu spezifizieren. Dies kann recht aufwändig werden, vor allem bei Klassen, in denen sich Methoden gegenseitig benutzen. Die Testfälle der einzelnen Methoden werden dadurch nochmals potenziert [LiRü96].

Bei der Spezifikation der Klassentestfälle muss der Tester also die Methodenschnittstellen, die Objektzustände und die implizite Reihenfolge der Methodenaufrufe berücksichtigen. Außerdem muss er auf den Zustand der geerbten Attribute achten, da diese auch Bestandteil der abgeleiteten Objektzustände sind (Abbildung 5.8). Will man noch alle Ablaufzweige in den Methoden erreichen, muss man sich schließlich mit der Ablauflogik der Methoden auseinandersetzen, um über die Testfälle alle Bedingungen zu erfüllen. Daraus wird klar, dass zur Spezifikation der Klassentestfälle detaillierte Kenntnisse der Klassenkonstruktion notwendig sind.

Prinzipiell ist es möglich, Klassentestfälle zu generieren: entweder aus der Klassenspezifikation, falls diese detailliert genug ist, oder aus dem Source-Code selbst. Im letzteren Fall wird allerdings nur die Klasse gegen sich selbst getestet. In beiden Fällen ist die Reihenfolge der Testfälle nicht determinierbar, wenn diese nicht explizit beschrieben sind. Mehr zu diesem Thema im nächsten Kapitel.



Function	Parameter	Bedingungen	Testwerte
Einzahlung	Währungsart	= DEM = Euro = Dollar = Yen	DEM Euro Dollar Yen
	Betrag	< 0 = 0 > 0	-1 0 100
	Retcode	sonst < = 0 > 0	Falsche Währung Ungültiger Betrag OK
Auszahlung	Währungsart	= DEM = Euro = Dollar sonst	DEM Euro Dollar
	Betrag	< 0 = 0 < = Kontostand > Kontostand	-1 0 100 9999
	Retcode	sonst < = 0 < = Kontostand > Kontostand	Falsche Währung Ungültiger Betrag OK überzogen

Abbildung 5.8 Klassentestfälle

5.4.2 Spezifikation der Integrationstestfälle

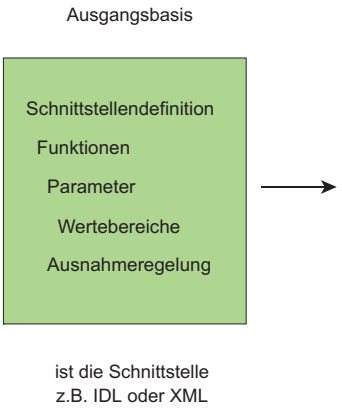
Zur Spezifikation der Integrationstestfälle sind Kenntnisse der Systemarchitektur und insbesondere der internen Schnittstellen zwischen Komponenten erforderlich. Deshalb wird hier auch von einem Grey-Box Test gesprochen. Der Tester braucht nicht zu wissen, wie die Klassen im Einzelnen implementiert sind, muss aber sehr wohl wissen, wie die Komponenten konstruiert sind und wie sie miteinander kommunizieren.

Komponenten sind abgeschlossene Gruppen von Klassen und Klassenhierarchien mit Schnittstellen nach außen, über welche die öffentlich sichtbaren Operationen von fremden Komponenten aufgerufen werden können. Für einen geordneten Integrationstest wird vorausgesetzt, dass diese Schnittstellen spezifiziert sind, denn sie sind der Bezugspunkt für die Spezifikation der Integrationstestfälle.

In einer Komponentenschnittstelle sind in der Regel folgende Elemente spezifiziert:

- die öffentlich sichtbaren Operationen,
- die Parameter jener Operationen und
- die Ausnahmebehandlung.

Zusätzlich sollten zumindest im Kommentar die Annahmen bezüglich der Objektzustände beschrieben sein. Ist z.B eine Klasse Konto Bestandteil einer Komponente „Kontoführung“, wird erwartet, dass das Konto bereits vorhanden ist, wenn ein Auszahlungsauftrag eintrifft. Diese Schnittstellenspezifikation, ob formal mit einer Sprache wie IDL oder informal in Prosa, ist die bevorzugte Praxis für die Spezifikation der Integrationstestfälle (Abbildung 5.9).



Funktion	Parameter	Testwerte
Einzahlung	Währungsart	DEM Euro Dollar
	Betrag	0 100 9999
	Retcode	ok
Auszahlung	Währungsart	DEM Euro Dollar
	Betrag	0 100 9999
	Retcode	ok Kein Geld überschritten

Abbildung 5.9 Intergrationstestfälle

Bei der Spezifikation der Integrationstestfälle muss der Tester noch mehr als beim Klassentest auf die Reihenfolge der Testfälle achten. Deshalb ist es ratsam, auch die Annahmen vorausgegangener Operationsaufrufe in der Komponentenschnittstelle festzuhalten. Zusätzlich zur Sequenz der Testfälle ist auch der Zustand der Objekte zu berücksichtigen, und hier wird es schwierig. Beim Klassentest wird nur auf die Objektzustände der Klasse unter Test geachtet. Beim Integrationstest gehören alle Zustände aller aktuellen Objekte aller Klassen einer Komponente zu den Vorbedingungen eines Testfalls. Hier zählen die persistenten Objekte bzw. die Datenbanken dazu. D.h. als Argumente und Ergebnisse eines Integrationstestfalls kommen außer den Parametern und internen Objektattributen noch Datenbankattribute dazu. Der Wirkungsbereich eines Integrationstestfalls ist daher weitaus umfangreicher als der eines Klassentestfalls. Daraus folgt, dass die Spezifikation der Integrationstestfälle aufwändiger ist. Der Tester muss nicht nur den Zusammenhang zwischen Komponenten und Klassen kennen, sondern auch mit der Struktur und dem Inhalt der Datenbanken vertraut sein.

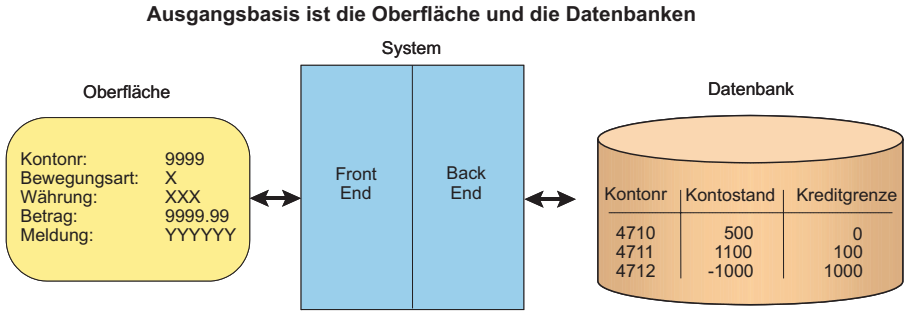
Integrationstestfälle lassen sich wie Klassentestfälle prinzipiell generieren, und zwar aus der Schnittstellenbeschreibung und der Datenbankbeschreibung. Voraussetzung dafür ist, dass beide formal beschrieben sind, z.B. in IDL und in SQL, und dass die Vor- und Nachbedingungen jeder Operation in der Schnittstelle festgelegt sind. Dies läuft darauf hinaus, die Integrationstestfälle als Bestandteil der internen Schnittstellenspezifikation zu betrachten. Dieser Ansatz wird im Kapitel 7 „Integrationstest“ näher behandelt.

5.4.3 Spezifikation der Systemtestfälle

Zur Spezifikation der Systemtestfälle sind Kenntnisse der Anwendung erforderlich. Der Tester handelt hier als Stellvertreter des Endanwenders. Er braucht daher weder Kenntnisse der Klassenkonstruktion noch der Systemarchitektur. Er soll nur wissen, wie das System insgesamt funktioniert und wie es zu benutzen ist. Im Grunde genommen testet er blind bezüglich des Systeminneren. Deshalb wird dieser Test als Black-Box Test bezeichnet.

Die Bezugspunkte für die Systemtestfallspezifikation sind in erster Linie

- die Benutzungsoberfläche,
- die externen Systemschnittstellen und
- die Datenbanken (Abbildung 5.10).



Oberfläche					Datenbank		
Kontonr	Bewegungsart	Währung	Betrag	Meldung	Kontonr	Kontostand	Kreditgrenze
4710	Ein	Euro	100	OK	4710	600	0
4711	Aus	DEM	1000	OK	4711	100	100
4712	Aus	Euro	10	überzogen	4712	-1000	1000
4713	Ein	DEM	200	NOK	4714	500	500

Abbildung 5.10 Systemtestfälle

Die Benutzungsoberfläche und ihre Handhabung wird im Bedienungshandbuch beschrieben sein. Der Tester muss nur in der Lage sein, diese zu verstehen und in Testfälle umzusetzen. Im Prinzip testet er die Benutzungsoberfläche gegen die Bedienungsanleitung, aus der er alle gültigen und ungültigen Nutzungsfälle ableitet. Wenn z.B. die Benutzungsoberfläche ein Geldautomat ist, kann er alle Varianten der Geldausgabe ausprobieren:

- mit falscher PIN-Nummer,
- mit ungültiger Karte,
- mit Kontoauskunft,

- ohne Kontoauskunft, usw.

Außerdem kann er so lange Geld ausgeben lassen, bis das Konto überzogen ist und er eine Meldung bekommt, dass die Auszahlung verweigert wird.

Im Falle einer graphischen Benutzungsoberfläche sollte der Tester mit den Möglichkeiten der Bedienung vertraut sein, d.h. welche Kontrolltasten und welche Mausklicks sind erlaubt und welche nicht. Dazu kommen gültige und ungültige Eingabewerte – Texte, Zahlen und Menü-Auswahlen. Es obliegt dem Tester, alle gültigen Fälle und so viele ungültige Fälle wie möglich auszuprobieren. Die Betonung liegt hier auf *so viel wie möglich*, denn eine Erprobung sämtlicher potenzieller Nutzungsfälle wird in den seltensten Fällen möglich sein.

Bei den Systemschnittstellen wird eine Beschreibung der Schnittstellenformate vorausgesetzt. Es wird sich in der Regel entweder um einen Datenstrom oder um fest formatierte Sätze handeln. Falls das Sendersystem bereits existiert, können die Dateneingaben vom System selbst produziert werden. Dies ist der Test mit echten Daten. Falls jedoch das Sendersystem noch nicht existiert, müssen die Eingabedaten von einem Testprogramm generiert werden. Dies ist der Test mit simulierten Daten.

Die Berichte sind nur insofern für die Testfallspezifikation wichtig, als sie Ergebnisse enthalten, die über Testfälle auszulösen sind. Der Tester muss darauf achten, dass alle repräsentativen Inhalte einmal vorkommen. Dazu muss er die entsprechenden Listenparameter versorgen.

Eine automatische Generierung der Systemtestfälle kommt kaum in Frage. Zu individuell sind die Anwendungen, zu komplex die Oberflächen und zu informal die Systembeschreibungen. Was in Frage kommt, ist die Aufzeichnung der Systemeingaben für spätere Wiederholungen und den Abgleich aktueller Ergebnisse mit früheren. Auf diese Capture/Replay- und Comparison-Techniken wird in den Kapiteln 8 „Systemtest“ und 11, „Testwerkzeuge“ näher eingegangen.

5.5 Konventionelle Testfallspezifikationsansätze

In der klassischen Testliteratur wird zwischen drei Ansätzen zur Testfallspezifikation unterschieden:

- ablaufbezogene Testfälle,
- datenbezogene Testfälle und
- funktionsbezogene Testfälle.

Die Unterscheidung basiert auf der Betrachtung des Testobjekts, nämlich darauf, ob die Ablaufstruktur, die Datenbereiche oder die Funktionalität als Anhaltspunkt für die Spezifikation der Testfälle dient.

5.5.1 Ablaufbezogene Testfälle

In seiner Ablaufstruktur besteht ein Programm aus einzelnen ausführbaren Anweisungen und deren dynamischen Beziehungen zueinander. Dargestellt wird die Struktur durch einen gerichteten Graphen. Darin bilden die Befehle die Knoten und die Beziehungen die Kanten. Dort, wo Befehle unbedingt aufeinander folgen, bilden sie eine Sequenz, d.h. wird ein Befehl ausgeführt, werden alle ausgeführt. Im Ablaufgraph wird dies als eine Kante dargestellt. Dort, wo die Ablaufflusslinie sich in zwei oder mehr vorwärtsgerichtete Linien trennt, wird eine Auswahl getroffen, welcher Weg einzuschlagen ist. Im Ablaufgraphen bilden jene Entscheidungen die Knoten. Dort, wo eine Ablaufflusslinie rückwärts verzweigt bzw. eine Kante zum Ausgangspunkt zurückkehrt, handelt es sich um eine Wiederholung bzw. um eine Schleife.
Die Sequenz-, Auswahl- und Wiederholungsstrukturen sind die Grundablaufstrukturen strukturierter Programme. Die Ausnahme ist die `GOTO`-Verzweigung, die von einem Knoten im Ablaufgraph zu einem beliebigen anderen Knoten hinführt. Sie zerstört die sonst saubere Ablaufstruktur des Programms.
Die Elemente der Programmablaufstruktur sind nach Beizer [Bei83]:

- Anweisungen,
- Zweige und
- Pfade.

Anweisungen sind die Befehle der jeweiligen Programmiersprache wie z.B. `WHILE DO`, `IF THEN`, `ADD`, `READ`, `PUT`, usw. In Sprachen wie C sind sie logische Ausdrücke. Zweige sind die Kanten im Ablaufgraphen. Zweige enthalten in der Regel mindestens eine Anweisung, aber nicht immer. Z.B. aus einem `IF` ohne `ELSE` gehen zwei Zweige hervor, einer mit den auszuführenden Anweisungen und ein Leer-Zweig, der implizite Sonst-Zweig. Deshalb ist der Test aller Zweige stärker als der Test aller Anweisungen, denn dadurch werden beim Zweigtest alle Anweisungen mit getestet, nicht aber umgekehrt.
Pfade sind dynamische Folgen von Ablaufzweigen bzw. eine einmalige Kombination von Zweigen. Ein Pfad beginnt beim Eingang in den Testgegenstand und endet bei einem Ausgang, d.h. ein Pfad ist ein Weg durch den Ablaufgraphen vom ersten bis zu einem letzten Zweig, z.B. zu einem einer `RETURN`-Anweisung entsprechenden Zweig. Je mehr Verzweigungen in der Ablauflogik, desto mehr Ablaufpfade sind zu testen. Die Anzahl der Ablaufpfade wächst exponentiell im Verhältnis zur Anzahl der Zweige. Um einen bestimmten Pfad zu testen, müssen alle Bedingungen an allen Knoten des Pfads erfüllt werden.
Wer ablaufbezogene Testfälle spezifizieren will, orientiert sich an der Ablaufstruktur und versucht, Fälle zu konstruieren, die in jeden Zweig hineinlaufen. Bei einer Fallanweisung muss z.B. jeder Fall einmal vorkommen. Bei `IF`-Anweisungen muss

die Bedingung einmal erfüllt und einmal nicht erfüllt sein. Bei Schleifen muss die Schleifenbedingung mindestens einmal erfüllt und einmal nicht erfüllt werden. Schleifen sind meistens schwierig zu testen, weil sie jede Menge andere Bedingungen beinhalten können.

Die Testfälle für einen ablaufbezogenen Test sind im Grunde genommen ein Spiegelbild der Bedingungen im Testobjekt und lassen sich aus demselben automatisch generieren, wie das Beispiel in Abbildung 5.11 zeigt.

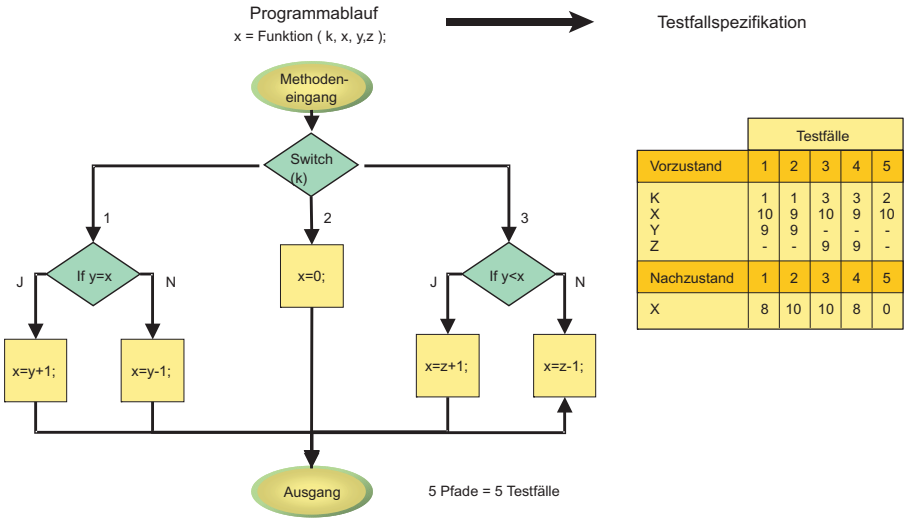


Abbildung 5.11 Ablaufbezogene Testfälle

Daran wird erkennbar, dass der ablaufbezogene Test ein Test des Programms gegen sich selbst ist, d.h. die Ablauflogik wird nur bestätigt. Fehlende Bedingungen oder Operationen werden nicht berücksichtigt, ebenso wenig wie falsche Datenstrukturen. Deshalb kann dieser Test nur dazu dienen, den Entwickler zu zwingen, sich mit seinem Programm auseinander zu setzen. Bei dieser Auseinandersetzung wird er vielleicht auf seine logischen Fehler stoßen. Dies entspricht der bereits erwähnten „Nose Rubbing Technik“ in Anspielung darauf, wie man junge Hunde trainiert, Gassi zu gehen [ABC82].

5.5.2 Datenbezogene Testfälle

Ein datenbezogener Testfall soll demonstrieren, wie der Testgegenstand auf seine Eingaben reagiert (Abbildung 5.12). Im Mittelpunkt steht also die Struktur der Eingabedaten. Datenbezogene Testfälle werden anhand der Eingabedaten und deren Wertebereichen formuliert. Es gilt, alle relevanten Datenwerte zu produzieren, um

auf diese Weise festzustellen, ob der Testgegenstand alle möglichen Eingaben korrekt behandelt.

In der Tat ist die Ermittlung der Eingabestruktur gar nicht so einfach. Die Eingabedaten können aus recht unterschiedlichen, verstreuten Quellen stammen. Beim Klassentest können sie Argumente in der Parameterliste, Attribute eines geerbten Objekts oder Ergebnisse fremder Operationen sein. Beim Integrationstest können sie Argumente in der Schnittstelle, Attribute in der Datenbank oder Zustände persistenter Objekte sein. Beim Systemtest sind sie Mausklicks, Kontrolltasten, Textfelder, Importdateien, persistente Objekte, Nachrichten, Datenbankinhalte usw. Es ist wirklich nicht leicht, als Voraussetzung für den datenbezogenen Test den Eingabedatenbereich einzugrenzen [Wey90].

Gelingt es, die Eingabedaten zu identifizieren, können datenbezogene Testfälle spezifiziert werden. Hierzu bieten sich mehrere Möglichkeiten an. Zum einen können Zufallswerte zugewiesen werden. Solche Testfälle setzen keine Anwendungskenntnisse voraus. Es werden in Abhängigkeit vom jeweiligen Datentyp nur Zahlen oder Zeichenfolgen generiert. Dies ist zwar verlockend, aber unergiebig, denn die Wahrscheinlichkeit, sinnvolle Datenkombinationen zu produzieren, ist sehr gering. Außerdem erzeugen Zufallsdaten auch Zufallsergebnisse, die nur schwer zu kontrollieren sind. Deshalb scheidet dieser Ansatz für kommerzielle Anwendungssysteme aus.

Ein zweiter Ansatz ist der Test mit repräsentativen Werten, die Äquivalenzklassen bilden. Statt alle möglichen Werte zu produzieren, weist man nur Werte zu, die stellvertretend sind für andere Werte derselben Klasse. Z.B. das Kennzeichen „D“ könnte für alle europäischen Kennzeichen stellvertretend sein, wenn sicher ist, dass alle europäischen Kennzeichen gleich behandelt werden, Die Zahl 101 könnte auch für alle Zahlen > 100 sein, d.h. alle Zahlen > 100 bilden eine Äquivalenzklasse, die durch 101 repräsentiert ist. Eine andere Äquivalenzklasse derselben Eingabevariablen wären alle Zahlen von 0 bis 100. Eine weitere Äquivalenzklasse wären die Zahlen < 0. Für diese Variablen gäbe es demzufolge drei Testwerte -1, 1 und 101, die stellvertretend sind für alle anderen Werte im Wertebereich dieser Variablen [RiCl85].

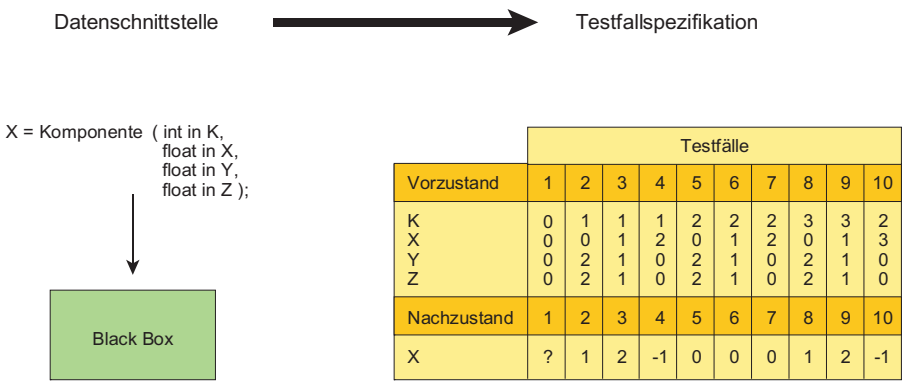
Ein dritter Ansatz ist die Grenzwertanalyse. Dies gilt allerdings nur für numerische Eingaben oder solche mit einer Rangordnung. Hier nimmt man den niedrigsten gültigen Wert als Untergrenze und den höchsten gültigen Wert als Obergrenze. Die zugrunde liegende Annahme: Wenn der Berechnungsalgorithmus für den höchsten und den niedrigsten Wert funktioniert, wird er auch für alle dazwischen liegenden Werte funktionieren. Um die Ausnahmebehandlung zu testen, wird außerdem ein Wert unter die Untergrenze und ein Wert über die Obergrenze zugewiesen. Daraus ergeben sich vier Testfälle für jede numerische Variable [Zei89].

Die bisherigen Ansätze beziehen sich immer nur auf einzelne Eingabedaten. In der Regel stehen aber die Eingaben in einer Beziehung zueinander oder zu internen

Werten. Es heißt, wenn der Status Rentner ist, müsste das Alter > 64 sein, d.h. es gibt eine Beziehung zwischen den Eingabevariablen „Status“ und „Alter“. Um solchen impliziten Beziehungen gerecht zu werden, muss man den relationalen Ansatz zur Testfallspezifikation verfolgen. Nach diesem Ansatz wird die Wertzuweisung einer Eingabevariable vom Inhalt einer anderen Variable bestimmt. Diese andere Variable könnte auch ein Objektattribut oder das Attribut einer Datenbank sein, so im Beispiel

```
Assert in Auto = „BMW“
       if Kennzeichen = „D“ &
          Besitzer = „Bayer“;
```

Relationale Testfälle setzen voraus, dass der Tester den Zusammenhang der Daten kennt und in der Lage ist, logische Beziehungen zu spezifizieren. Möglicherweise werden diese Beziehungen in einer Datenbankbeziehungsgraphik dargestellt. Daran wird man erkennen, wie komplex die Datenstrukturen sind [ReWe85].



	Testfälle									
Vorzustand	1	2	3	4	5	6	7	8	9	10
K	0	1	1	1	2	2	2	3	3	2
X	0	0	1	2	0	1	2	0	1	3
Y	0	2	1	0	2	1	0	2	1	0
Z	0	2	1	0	2	1	0	2	1	0
Nachzustand	1	2	3	4	5	6	7	8	9	10
X	?	1	2	-1	0	0	0	1	2	-1

Abbildung 5.12 Datenbezogene Testfälle

5.5.3 Funktionsbezogene Testfälle

Der funktionsbezogene Test - Functional Testing genannt – ist ein Test auf der Basis der Funktionsspezifikation. Eine Funktion ist eine Regel für die Erzeugung eines Ergebnisses oder einer Ergebnismenge aus einer Reihe von Argumenten.

$$x = f(y, z);$$

Bei einem funktionsbezogenen Testfall wird vom Ergebnis ausgegangen, d.h. man erwartet ein bestimmtes Resultat wie die Nummer *42* aus der Geschichte „Per Anhalter durch das Universum“. Die Frage ist, welche Daten muss man eingeben, um zu diesem Ergebnis zu gelangen.

Um das herauszubekommen, bieten sich drei Möglichkeiten an:

- erstens, es ist im Benutzerhandbuch beschrieben;
- zweitens, es ist in der Programmvorgabe spezifiziert;
- drittens, es steckt im Programm selbst [How86].

Wenn es weder dokumentiert noch spezifiziert ist und wenn keiner das Programm lesen kann, bleibt nur die vierte Möglichkeit übrig: Jemand hat es in seinem Kopf. Ein Domänenexperte, der das Anwendungsgebiet aus dem Effeff kennt, wird immer wissen, wie bestimmte Ergebnisse zustande kommen sollten Ob sie wirklich so zustande kommen, ist eine andere Frage. Er wird z.B. wissen, wie der Zinsbetrag oder wie eine Projektion errechnet wird. Entweder schreibt er sein Wissen auf in einer informalen Dokumentation oder in einer formalen Spezifikation, oder er schreibt die Programme selbst. In den prototypbasierten Entwicklungen werden die Experten gezwungen, ihr Wissen in einer Programmiersprache wie z.B. Visual-Basic festzuhalten und zu bestätigen. Anschließend werden die endgültigen Programme anhand der Prototypprogramme gefertigt. Danach ist es möglich, die Testfälle aus dem Prototyp abzuleiten, denn dort sind die Funktionen angeblich richtig beschrieben. Dies ist eine bewährte Methode für die Ermittlung funktionaler Testfälle.

Da es keinen Sinn macht, die Funktionalität des Testobjekts aus dem Objekt selbst abzuleiten, bleiben vier Alternativen der funktionalen Testfallspezifikation übrig:

- ein Test gegen das Expertenwissen,
- ein Test gegen das Benutzerhandbuch,
- ein Test gegen die Programmspezifikation und
- ein Test gegen das Operationsprofil [How85].

Bei der ersten Alternative formuliert der Experte die Testfälle aus seinem Kopf. Geistig stellt er Beziehungen zwischen einem erwarteten Ergebnis und den dafür erforderlichen Eingaben her und schreibt diese Beziehungen als Testfall auf z.B.

```
Assert out Stundensatz = 40,00
       if (Job = „Programmierer“ &
              Alter < 30 &
              Stundenzahl < = 40);
```

Bei der zweiten Alternative sind keine Experten erforderlich. Hier genügt es, wenn der Testfallspezifizierer das Benutzerhandbuch lesen kann. Darin sollten alle gültigen Ergebnisse erkennbar sein und wie sie zustande kommen, d.h. was man eingeben muss, um sie zu bewirken. Wer also die Fehlermeldung „falsche Postleitzahl“ testen will, muss eine ungültige Zahl eingeben, z.B.

```
Assert out Meldung = „falsche Postleitzahl“
           if (PLZ < 1 | PLZ > 99999);
```

Bei der dritten Alternative sind die Funktionen einer Software zumindest halbformal spezifiziert. Oft werden Entscheidungstabellen oder Entscheidungsbäume verwendet. In UML werden Aktivitäts- und Zustandsübergangsdiagramme verwendet. Ein häufiges Beispiel ist die Flugbuchung. Aus der Sicht des Testers ist Flugbuchung eine Funktion mit zwei möglichen Ergebnissen „bestätigt“ oder „abgelehnt“. Das Ergebnis „bestätigt“ erfolgt so lange, als noch ein Platz im gewünschten Flug frei ist, vorausgesetzt natürlich, dass der Fluggast mit jedem Platz zufrieden ist. Das Ergebnis „abgelehnt“ kommt, wenn alle Plätze belegt sind. Die beiden Testfälle können sogar aus dem Zustandsübergangsdiagramm generiert werden, z.B.

```
Assert in Flugnummer = „LH 820“;
   Assert in Passagier = „Foshag“;
   Assert out Meldung = „Bestätigt“,
            if (LH 820.Freieplätze > 0);
   Assert out Meldung = „Abgelehnt“,
            if (LH 820.Freieplätze < 1);
```

Die spezifikationsbezogenen Testfälle stellen natürlich die eleganteste Form des Testens dar, weil sie eine formale Basis haben (Abbildung 5.13). Man darf aber nicht vergessen, dass auch die formalste Spezifikation falsch sein kann. Sie ist nur so zuverlässig wie die Menschen, die sie verfasst haben. Das Gleiche trifft für die Benutzerdokumentation zu. Nur mit der Benutzerdokumentation ist es wahrscheinlicher, dass ein Domänenexperte sie versteht, während dies bei den UML-Dokumenten weniger der Fall ist.

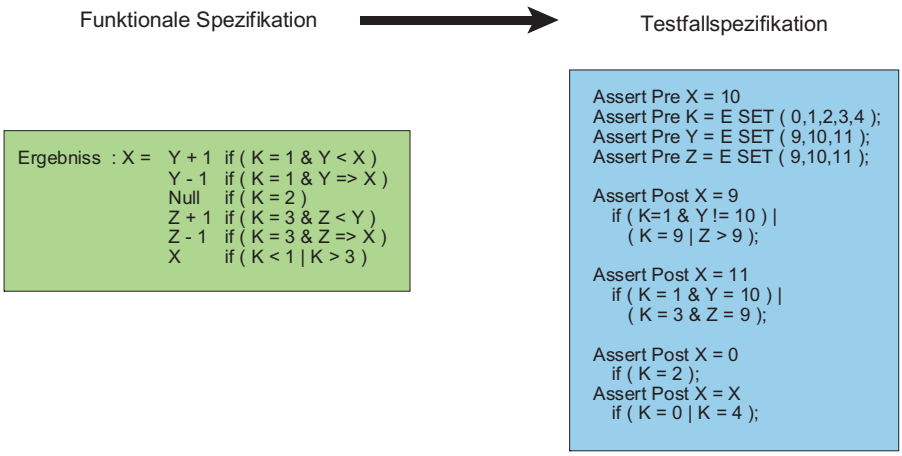


Abbildung 5.13 Funktionsbezogene Testfälle

Eine zuverlässigkeitsorientierte Form der funktionalen Testfallspezifikation ist letzten Endes der Test gegen ein bereits bestätigtes *Operationsprofil* . Entweder gibt es einen funktionsfähigen Prototyp oder es gibt eine bereits produktive Version. Der

Grundansatz ist in beiden Fällen gleich: Man kopiert die Datenbankzustände vor dem Test und zeichnet alle Transaktionen samt Eingaben und Ausgaben auf. Danach lädt man die Testdatenbanken und wiederholt die Testfälle mit dem neuen System. Auf diese Weise ist es nicht nötig, die Testfälle als solche zu spezifizieren. Sie sind im Operationsprofil enthalten. Die Ergebniskontrolle ist auch einfacher, weil man die neuen Ergebnisse mit den alten automatisch abgleichen kann.
Der Preis für diesen relativ bequemen Ansatz ist die Konstruktion und Bestätigung eines lauffähigen Prototyps bzw. einer Vorabversion. Es gilt abzuwägen, was effektiver und billiger ist. Einen Prototyp zu bauen und zu testen, die Testfälle aus einer formalen vollständigen UML-Spezifikation abzuleiten oder die Testfälle manuell zu spezifizieren.

5.6 Das Besondere an der objektorientierten Testfallspezifikation

Testfälle für den Test objektorientierter Systeme erben natürlich die Eigenschaften der konventionellen Testfälle. Sie zielen auf die Ausführung bestimmter Teile der Software und beschreiben dafür die Vor- und Nachzustände. Sie können auch ablauf-, daten- oder funktionsbezogen sein, aber hier beginnt die Unterscheidung.

5.6.1 Unterschiede beim ablaufbezogenen Test

Prozedurale Programme enthalten mehrfach verschachtelte Ablaufstrukturen, sodass es sich lohnt, sich mit der Ablauflogik zu beschäftigen. Schon allein der Versuch, Testfälle zu formulieren, um alle Zweige zu erreichen, kann zur Aufdeckung einiger Fehler führen. Objektorientierte Programme sind hingegen Netze verschachtelter Operationen, die im Quellcode verstreut liegen. Der dynamische Zusammenhang ist anhand der statischen Zusammensetzung nicht offensichtlich. In den Methoden selbst ist die Entscheidungslogik meistens nicht kompliziert, sodass es zwar einfach ist, alle Zweige zu überdecken, aber in den Zweigen werden weitere Operationen aufgerufen, und diese rufen noch andere auf. Um die Testfälle für eine Methode zu spezifizieren, muss der Tester alle aufgerufenen Methoden berücksichtigen. Dies kann auf zwei Arten geschehen: entweder, indem er sie alle inline in den Code der zu testenden Klasse einzieht und die entsprechenden Aufrufe bzw. Instanzvariablen an die Instanz dieser Klasse (in Java: `this`) umdirigiert, oder indem er sie mit einem Klassenbrowser aufsucht. Falls die Methoden allzu tief verschachtelt sind, führt dies zu sehr komplexen und aufwändigen Testfällen mit einem großen Datenbereich, verteilt durch mehrere Objekte.
Die Alternative dazu wäre, die verschachtelten Methoden zu vernachlässigen und die Testfälle nur auf die Mitgliedsmethoden einer Klasse zu beschränken. Damit

wird die Testfallspezifikation viel einfacher. Sie basiert nur auf der Entscheidungslogik des sichtbaren Codes. Aber der Test selbst wird unergiebiger, d.h. der einfache Klassentest aufgrund der Ablaufstruktur ist mit dem ablaufbezogenen Modultest prozeduraler Programme nicht vergleichbar. Er ist, was die Fehlerauffindung anbetrifft, wesentlich weniger effektiv. Nur ein komplizierter Klassentest mit der Einbeziehung aller verschachtelten Klassen kann vergleichbare Ergebnisse erzielen. Dies grenzt jedoch an einen Integrationstest.

5.6.2 Unterschiede beim datenbezogenen Test

Konventionelle Programme erhalten ihre Eingabedaten aus einer von vier Quellen:

- direkt aus der Benutzungsoberfläche,
- aus der Datenbank,
- aus einer Eingabedatei oder
- über eine Parameterschnittstelle.

Die Parameterschnittstelle spielt dabei die geringste Rolle. Die Hauptquellen sind die Oberflächen und Dateien. Wer also konventionelle Programme datenbezogen testet, muss sich mit der Struktur und dem Inhalt der Masken, Dateien und Datenbanken auseinandersetzen. Objektorientierte Systeme sind verteilte Komponenten, jede mit eigenen Klassenhierarchien. Einige Komponenten beziehen ihre Eingaben von der Benutzungsoberfläche, andere lesen Importdateien, aber viele Komponenten empfangen ihre Eingaben von einander über die Parameterschnittstelle. Der Datenaustausch zwischen Komponenten spielt eine wesentlich größere Rolle als bei konventionellen Programmen. Dieser Austausch lässt sich über verschiedene Mechanismen bewerkstelligen, z.B. über einen Socket, eine RPI, eine RMI oder eine ORB. Dabei wird eine Nachricht mit den Eingabedaten des Zielobjekts in die Zielkomponente übermittelt. Wer also Objekte über ihre Eingabedaten testet, muss sich mit der Struktur und dem Inhalt der internen Schnittstellen auseinander setzen. Dies erfordert wiederum mehr programmtechnisches Wissen als die Beschäftigung mit externen Schnittstellen wie Masken und Dateien. Es ist auch schwieriger, interne Nachrichten zu generieren als Masken oder Dateien. Notfalls muss die Senderkomponente simuliert werden, nur um den gewünschten Eingabedatenstrom zu produzieren. Darum ist der datenbezogene Test zwar effektiver, aber schwer umzusetzen, besonders in einer verteilten, objektorientierten Umgebung.

5.6.3 Unterschiede beim funktionsbezogenen Test

Beim funktionsbezogenen Test sind die Unterschiede zwischen konventionellen und objektorientierten Systemen am geringsten – beide liefern die gleichen Ergebnisse ab. An der Benutzungsoberfläche in den Listen und in den Exportdaten unterscheiden sich die Systemarten nicht voneinander. Ein Unterschied besteht nur darin, wie interne Ergebnisse erzeugt werden. Auf der Methodenebene werden Rückgabewerte erzeugt, auf der Klassenebene werden Objektzustände erstellt und auf der Komponentenebene werden Ausgangsnachrichten gesendet. Hier wird wohl eine ausführliche UML-Objektspezifikation ergänzt durch eine IDL-Schnittstellenspezifikation erforderlich sein, um die Ergebnisse zu erkennen und auf ihren Ursprung zurückzuführen. Je tiefer die Vererbungshierarchien und je länger die Operationsaufrufketten, umso schwieriger ist es, die Entstehung eines bestimmten Ergebnisses zurückzuverfolgen. Der Hauptunterschied liegt also hier in der Komplexität der Architektur, die es schwer macht, Funktionalität zu isolieren und getrennt für sich zu bestätigen.

5.7 Einfluss der UML auf die Testfallspezifikation

Natürlich besteht eine Verbindung zwischen der Spezifikation der Programme und der Spezifikation der Testfälle. Es sind die zwei Seiten derselben Medaille – auf der einen Seite die Hypothese, es sollte so funktionieren, auf der anderen Seite die Beweisführung, dass es wirklich so funktioniert. Die Sprache, in der die Hypothese formuliert wird, hat zwangsläufig Einfluss auf die Formulierung der Beweisführung. Die eine Semantik bedingt die andere.

Objektorientierte Software wird zunehmend mit Hilfe der „Unified Modelling Language“ – UML – spezifiziert. Diese Entwurfssprache ist eine graphische Beschreibung der eigentlichen Software [OMG99]. Sie setzt sich zusammen aus einer Reihe Diagrammtypen, die verschiedene Sichten auf das geplante oder bereits implementierte System darstellen (Abbildung 5.14). Dazu gehören (vgl. z.B. [Fow98]):

- das Anwendungsfalldiagramm
- das Klassendiagramm
- das Sequenzdiagramm
- das Kollaborationsdiagramm
- das Aktivitätsdiagramm
- das Zustandsdiagramm
- das Komponentendiagramm
- das Verteilungsdiagramm.

Hinzu kommt die Object Constraint Language – OCL –, mit der Zusicherungen bzw. Verarbeitungsregeln ausgedrückt werden können [WaKl99].

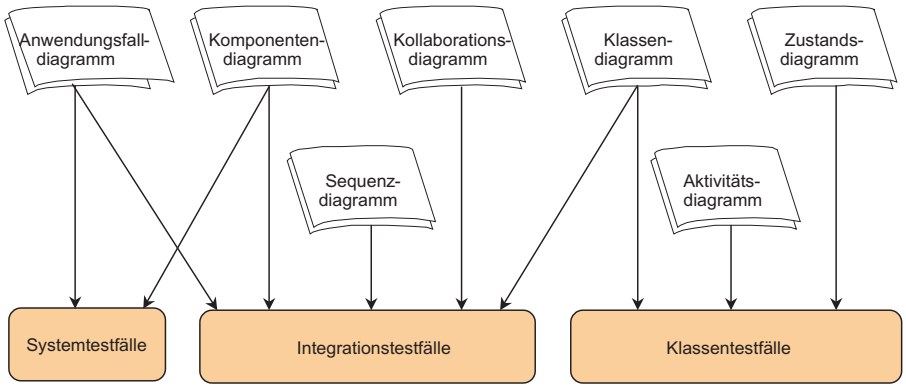


Abbildung 5.14 Ableitung der Testfälle aus UML

5.7.1 Anwendungsfalldiagramm

Das *Anwendungsfalldiagramm* (use case diagram) beschreibt die Interaktion zwischen den Benutzern des Systems und dem System selbst. Es zeigt, welche Ereignisse von wem ausgelöst werden und wie sie miteinander zusammenhängen. Diese Diagramme eignen sich als Basis für die Spezifikation funktionaler Testfälle im Systemtest, aber nur wenn die Vor- und Nachbedingungen der einzelnen Fälle spezifiziert sind. Ist dies der Fall, sind sie eine nützliche Basis für die Ermittlung der Systemtestfälle. Außerdem lassen sich die Systemtestszenarien aus ihnen ableiten.

5.7.2 Klassendiagramm

Das *Klassendiagramm* (class diagram) beschreibt die Generalisierungshierarchien und Assoziation der Klassen sowie die Attribute und Methoden der einzelnen Klassen. Es kann als wichtige Unterlage sowohl für den Klassentest als auch für den Klassenintegrationstest dienen. Daraus lässt sich die Information für einen Klassentestrahmen ableiten, insbesondere die anzustoßenden, in der Klasse selbst definierten Operationen und die zu simulierenden, in anderen Klassen definierten Operationen. Für den Integrationstest deuten sie auf Assoziationen zwischen Klassen sowie auf Generalisierungen.

Assoziationen können als Aggregation und Komposition markiert werden. Bei einer Aggregation können Instanzen der „Teileklasse“ auch ohne eine Instanz der „Ganzesklasse“ existieren, bei einer Komposition erzwingt das Zerstören der Ganzes-Instanz auch das der Teile-Instanzen. Diese Integritätsregeln lassen sich durch Test-

fälle bestätigen. Die Enden der Assoziationen sind mit Multiplizitäten markiert, die angeben, wie viele Instanzen der dem jeweiligen Assoziationsende zugeordneten Klasse gemäß der Assoziation miteinander verbunden können. Multiplizitäten werden in der Form `0:1`, `1:N` und `M:N` angegeben, wobei die Multiplizität `0..N` bei nicht näher spezifiziertem `N` auch zu `*` abgekürzt notiert werden kann. Die Enden der Assoziationen bestimmen somit die Kardinalität der Menge gemäß der Assoziation verbindbarer Instanzen. Diese Kardinalität kann bedingt oder unbedingt sein. Daraus lassen sich etliche Integritätsregeln ableiten, wie z.B. kein Konto ohne Kontoinhaber und kein Kontoinhaber mit nicht mindestens einem Konto. Diese aus den Klassendiagrammen abgeleiteten Integritätsregeln können ohne weiteres in Testfälle umgesetzt werden, um die Objektintegrität zu prüfen. Es fehlen jedoch die Verarbeitungsregeln, die man braucht, um die Methoden zu testen.

5.7.3 Sequenzdiagramm

Das *Sequenzdiagramm* (sequence diagram) beschreibt die Ausführungsfolge der Operationen über Objektgrenzen hinweg. Damit werden die funktionalen Pfade durch die Objekte aufgezeichnet. Diese funktionalen Pfade entsprechen den Testfällen des Integrationstests. Ein Testfall, der mehrere Objekte „durchkreuzt“, müsste 1:1 mit einem Pfad durch das Sequenzdiagramm übereinstimmen. D.h. es sollte möglich sein, die statischen Sequenzdiagramme durch Tabellen, die nach entsprechender Instrumentierung während des Klassenintegrationstests generiert werden (test traces), nachträglich dynamisch abzubilden. Demnach braucht der Tester nur die beiden Operationsfolgen miteinander abzugleichen. Was fehlt, sind die Bedingungen, unter denen die Kontrollübergaben stattfinden. Der Tester muss noch für jeden Pfad durch das Sequenzdiagramm sämtliche Pfadprädikate definieren, die zu erfüllen sind, um diesen Pfad anzusteuern. Aus dem Grund kann das Sequenzdiagramm nur als Verifikationsmittel verwendet werden, d.h. zu kontrollieren, ob alle darin angezeigten Objektfolgen tatsächlich ausgeführt wurden.

5.7.4 Kollaborationsdiagramm

Das *Kollaborationsdiagramm* (collaboration diagram) beschreibt die Interaktion zwischen Objekten bzw. die fremden Operationsaufrufe. Diese Darstellung ist für die Planung und Spezifikation der Integrationstestfälle nützlich, denn beim Integrationstest kommt es darauf an, sämtliche interne Schnittstellen zwischen verteilten Objekten zu testen. Hier ist jede Schnittstelle zu jeder fremden Operation festgehalten. Hinter jeder solchen Schnittstelle stecken eine Menge von Testfällen, und zwar eine für jede Kombination von Parametern bzw. für jede relevante Kombination. Das Kollaborationsdiagramm ist somit nützlicher als das Sequenzdiagramm, weil es

nicht nur auf eine Kontrollfluss-Übergabe zwischen Klassen verweist, sondern jeden einzelnen Operationsaufruf mit Parameterliste sowie die Art der Verbindung, über die der Operationsaufruf „geleitet“ wird, beschreibt. Trotzdem bietet es nur einen Anhaltspunkt für den Tester. Die Parameterwerte und Rückgabewerte selbst muss der Tester noch ausarbeiten.

5.7.5 Aktivitätsdiagramm

Das *Aktivitätsdiagramm* (activity diagram) kann benutzt werden, um drei unterschiedliche Flussarten zu beschreiben:

- Steuerfluss
- Datenfluss und
- Prozessfluss.

Welche Flussart gemeint ist, geht meistens aus der Darstellung hervor. Klar abgegrenzte Bereiche der Aktivitätsdiagramme werden in der Literatur als Schwimmbahnen bezeichnet. Zu jeder Schwimmbahn gehört mindestens ein Testfall. Da jedoch Schwimmbahnen bekanntlich sehr breit sein können, werden in der Regel mehrere gebraucht, um alle möglichen Verzweigungen abzudecken. Die Mehrdeutigkeit dieser Diagrammart macht das Leben der Tester schwer. Man könnte alles und nichts darunter verstehen. Auf der positiven Seite können Aktivitätsdiagramme dazu dienen, den Zusammenhang zwischen nebenläufigen Pfaden durch die Objekte und Komponenten zu spezifizieren. In dieser Hinsicht erfüllen sie den gleichen Zweck wie Petrinetze. Daraus lassen sich technische Testfälle für den Test der Transaktionen im Netz ableiten. Hier lässt sich feststellen, ob und wo „Deadlocks“ und andere Synchronisierungsprobleme auftreten können.

5.7.6 Zustandsdiagramm

Das *Zustandsdiagramm* (state diagram, state chart) beschreibt die möglichen Objektzustände und deren Übergänge. Es ist das detaillierteste aller UML-Diagramme und am ehesten für die Spezifikation der Klassentestfälle geeignet, denn in ihm kommt die eigentliche Verarbeitungslogik zum Tragen. Sind Zustandsdiagramme formal, vollständig und konsistent, können aus ihnen automatisch Testfälle generiert werden (vgl. Abschnitt 6.5.4). Genau dies hat Robert Poston in einem auf entsprechend angereicherten Diagrammen basierten Testfallgenerierungswerkzeug gemacht [Pos94].

Für Testtheoretiker erscheint dies als ideale Lösung. Es ist auch viel darüber geschrieben wurden. Leider werden in der Projektpraxis die Zustandsdiagramme sel-

ten vollständig ausgearbeitet, wenn überhaupt. Es bleibt nur der Code als Beschreibung der Verarbeitungslogik.

5.7.7 Komponentendiagramm

Das *Komponentendiagramm* (component diagram) beschreibt die Zuordnung der Klassen bzw. Objekte zu den Komponenten. Außerdem definieren sie die Schnittstellen zwischen Komponenten. In dieser Hinsicht sind sie eine unentbehrliche Grundlage für den Komponentenintegrationstest, wo es darum geht, alle Schnittstellen zwischen Komponenten zu validieren. Es fehlt allerdings eine detaillierte Spezifikation der Schnittstelleninhalte. Dazu muss man die IDL-Sprache heranziehen. Nur auf der Basis der IDL ist es möglich, Schnittstellentestfälle zu spezifizieren.

5.7.8 Verteilungsdiagramm

Das *Verteilungsdiagramm* (deployment diagram) beschreibt die Verteilung der Komponenten auf dem Rechnernetz, d.h. welche Komponente sich auf welchen Netzknoten befindet. Damit lassen sich zwar keine Testfälle erstellen, aber für den Systemintegrationstest verteilter Komponenten ist die darin enthaltene Information sehr hilfreich. Ansonsten bieten die Verteilungsdiagramme keine Hilfe.

5.7.9 Object Constraint Language

Schließlich gibt es im Zusammenhang mit der UML auch die *OCL* – Object Constraint Language. Jene Sprache ist die Antwort der OMG auf die Notwendigkeit einer Regelformulierungssyntax. Mit der OCL lassen sich Zusicherungen über Objektzustände sowie über Vor- und Nachbedingungen verfassen. Somit ist OCL die Sprache der Testfallspezifikation, so wie sie hier in diesem Kapitel präsentiert wird. Sie kann verwendet werden, um gültige und ungültige Zustände sowie Beziehungen zwischen den Attributzuständen festzulegen, z.B.

```
Programmierer.gehalt < Analytiker.gehalt; Objektptr != null;
```

Die OCL wird benutzt, um die Zustände der Objekte zuzusichern, die Eingangsparameter zu spezifizieren und die Ausgangsparameter zu validieren (Abbildung 5.15). Richtig angewandt, kann sie auch als Testskriptsprache dienen. Durch die Extrahierung und Allokierung der OCL-Anweisungen lassen sich Testprozeduren bilden, die für den Test der Klassen gegen die Klassenspezifikation genutzt werden können. So gesehen ist die UML ergänzt durch OCL ein wichtiger Schritt in Richtung selbst-verifizierender Systeme. [WaKl99]

```
Bank.Überzogene_Konten ➔ Select ( Bank.Konto.Kontostand
                                  < ( 0 - Bank.Konto.Kreditgrenze ) );

Bank.Schlechte_Kunden  ➔ Select ( Bank.Überzogene_Konten ) &&
                         Select ( Bank.Kunde.Bonität = “niedrig” );
```

```
Assert Post Bank.Schlechte_Kunden
    if ( Bank.Kunde.Bonität = “niedrig” )
    & ( Bank.Konto.Kontostand < ( 0 - Bank.Konto.Kreditgrenze ) )
    & ( Bank.Kunde.Kontonr = Bank.Konto.Kontonr )
```

Abbildung 5.15 Objekt Constraint Language

Die Kernfrage ist, ob die UML wirklich dazu dienen kann den Test zu erleichtern bzw. zu verbessern. Die Antwort lautet „jein“. Theoretisch liefern die Dokumente, sofern sie mit dem Code übereinstimmen, eine nützliche Quelle für Testfälle. Mit UML wäre es möglich, den Code gegen das Objektmodell zu testen. In der Praxis sind die UML-Diagramme jedoch selten dazu brauchbar, weil sie zu oberflächlich gehalten werden. Testen hat mit Zuständen und deren Veränderung zu tun und die UML ist schwach, was die Beschreibung dieser Zustände betrifft. Die Zustandsdiagramme gehen auf die Zustände ein, werden aber in der Praxis nur selten verwendet.

Der OO-Testexperte Robert Binder nimmt folgende Stellung zum Beitrag der UML zum Test ein. Er schreibt:

> *“Some OOA/OOD approaches, including those of Embley-Kurtz, Booch and OMT obliterate the distinction between event and action on state transition, static state, and that between event on transition and action on state...They allow both forms in a hybrid state diagram. UML carries this mutation forward. The resulting hybrid has all of the disadvantages of the Moore model, causes additional testing headaches and does not in any way improve the model’s ability to represent or analyze object behavior.... In short, the UML hybrid solution is error prone and effort intensive.”* ***[Bin99]***

Lediglich die OCL [WaKl99] – ein „add on“ zur UML – verspricht wahre Hilfe für die Spezifikation adäquater Testfälle.

5.8 Testfallspezifikation für den verteilten Kalender

5.8.1 Klassentestfälle für den verteilten Kalender

Um Testfälle für den verteilten Kalender zu spezifizieren, muss man von bereits festgelegten Testszenarien ausgehen und dieses in konkrete Testfälle umsetzen. Da der Klassentest den Code der Klassen voraussetzt, werden die Klassentestfälle im Zusammenhang mit der Klassenentwicklung erstellt. Dies ist ohnehin eine Aufgabe

der Entwickler. Deshalb werden die Klassentestfälle erst im nächsten Kapitel „Klassentest“ behandelt. In diesem Kapitel werden nur jene Testfälle gezeigt, die vom Tester bzw. vom Testmanager spezifiziert werden können:

- Integrationstestfälle und
- Systemtestfälle.

5.8.2 Integrationstestfälle für den verteilten Kalender

Die Umsetzung des Testszenarios für den Integrationstest erfordert eine Testprozedur mit einer Zusicherungs-basierten Testskriptsprache im Rahmen einer CORBA IDL-Schnittstelle, definiert für jede interne Schnittstelle. Hier wird nur zwecks der Methodendarstellung die Schnittstelle zur Anlegung eines Wochenkalenders gezeigt. Das volle Testbeispiel für den Integrationstest ist im Kapitel 7 zu finden.

```
Module Kalender
       {Interface Wochenkalender : Kalender
          {attribute string Mitarbeiter;
            attribute int Wochenzahl;
            attribute int Retcode;
            attribute enum Retcodes (ok,
              unbekannter_Mitarbeiter,
              ungueltige_Wochenzahl);
            int Wochenkalender (in string Mitarbeiter,
                            in int Wochenzahl);
            raises („keine Verbindung zum Kalender“);
            //$IDLTEST (int $Testfall)
            //$TestCase 1: Assert in Mitarbeiter = ‘ ‘;
                         Assert out Retcode = 1;
            //$TestCase 2:
                  Assert in Mitarbeiter = ‘Meyer’,
                                Wochenzahl = 0;
             Assert out Retcode = 2;
            //$TestCase 3:
                  Assert in Mitarbeiter = ‘Meyer’,
                                Wochenzahl = 53;
             Assert out Retcode = 2;
               //$TestCase 4:
                  Assert in Mitarbeiter = ‘Meyer’,
                                Wochenzahl = 12;
             Assert out Retcode = 0;
              Retcode = Wochenkalender(Mitarbeiter,
                      Wochenzahl);
```

```
    } // Ende der Testprozedur Wochenkalender
} // Ende der Schnittstelle Wochenkalender
```

5.8.3 Systemtestfälle für den verteilten Kalender

Die Umsetzung des Testszenarios für den Systemtest erfordert zwei Testfalltabellen:

- eine für die Eingaben, die der Tester zu betätigen hat, und
- eine für die Ausgaben, die der Tester zu bestätigen hat.

Die Eingabetabelle hat eine Zeile pro Taste und Textfeld, sowie eine Zeile für die Mausposition und eine Spalte für jeden spezifizierten Testfall mit den einzugebenden Werten.
Die Ausgabetabelle hat eine Zeile pro Textfeld sowie eine Zeile für jedes Signal und eine Spalte für jeden spezifizierten Testfall mit den zu erwartenden Werten.
Die folgenden beiden Teiltabellen dienen nur dazu, die Methodik im Ansatz zu erläutern.

Kalendereingabetabelle

Eingabe	TF_1	TF_2	TF_3	TF_4
Mausposition	Start	Start	Start	Start
Mitarbeitername	Space	Meyer	Meyer	Meyer
Wochennummer	0	0	53	1
Startzeit				xx:yy
Endezeit				xx:yy

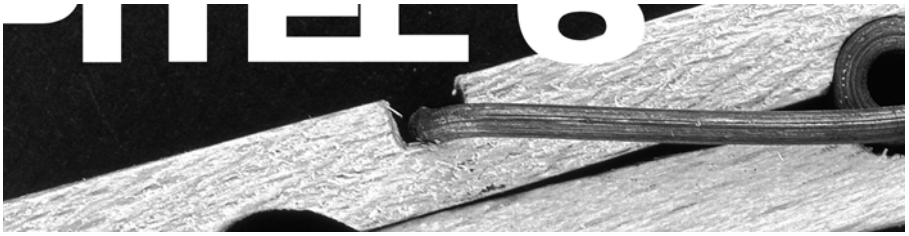
Kalenderausgabetabelle

Eingabe	TF_1	TF_2	TF_3	TF_4
Signal	beeb	beeb	Beeb	beeb
Meldung	Mitarbeiter fehlt	Mitarbeiter unbekannt	Wochenzahl ungültig	Startzeit ungültig
Kalender	Null	null	Null	null
Projekt	unverändert	unverändert	Unverändert	unverändert

usw.

6

Klassentest



Unterschiede zwischen Klassentest und Modultest
Zweck des Klassentests
Einschränkungen zum Klassentest
Theoretische Ansätze zum Klassentest
Praktische Ansätze zum Klassentest
Beispiel einer Build-In Testtechnik
Beispiel eines Klassentestrahmens
Klassentestarten
Test der Tagesklasse im Kalendersystem

Inhaltsübersicht Kapitel 6

6 Klassentest

6.1 Unterschiede zwischen Klassentest und Modultest

Klassen sind das objektorientierte Äquivalent zu den prozeduralen Modulen, allerdings mit einigen signifikanten Unterschieden. Prozedurale Module sind im strengeren Sinne Compilierungseinheiten. Der Quellcode ist eine Sourcedatei und das Compilierungsergebnis eine Objektdatei. Prozedurale Module haben in der Regel nur einen Eingang mit einer Parameterliste. Von diesem einen Eingang aus wird in das Modul verzweigt. Der Datenbereich eines prozeduralen Moduls kann sehr groß werden. Bis auf Sprachen wie PL/I und C, die lokale Daten für einzelne Prozeduren zulassen, sind die Moduldaten alle global, d.h. sie stehen allen einzelnen Prozeduren zur Verfügung. Die Prozeduren kommunizieren über diesen globalen Datenbereich miteinander bzw. tauschen Daten aus. Die Ausgaben einer Prozedur sind die Eingaben für ihre Nachfolgerprozeduren. Dies hat für den Test den Vorteil, dass der Datenzustand jederzeit kontrollierbar ist.

Da prozedurale Module sehr groß geraten können, kann auch ihre Ablauflogik entsprechend komplex werden. Die ausführbaren Anweisungen sind in Auswahl- und Wiederholungsstrukturen eingebettet und die Prozedurblöcke ineinander verschachtelt. Es ergeben sich daraus zahlreiche Verzweigungen und Verästelungen im Code. Die Anzahl der Pfade vom Eingang des Moduls durch die Ablaufverzweigungen bis hin zu einem Ausgang wächst exponentiell im Verhältnis zur Anzahl Verzweigungen. Prozedurale Module sind deshalb durch sehr lange Pfade gekennzeichnet.

In der kommerziellen Datenverarbeitungspraxis haben prozedurale Module meistens eigene IO-Operationen. Sie kommunizieren mit ihrer Umgebung über Datenschnittstellen oder direkte Benutzungsschnittstellen. Die Anzahl der aufgerufenen Untermodule hält sich gewöhnlich im Rahmen. Dafür sind die Parameterschnittstellen zu den anderen Modulen umso umfangreicher.

Beim konventionellen Modultest kommt es darauf an, die Schnittstellen des Moduls unter Test zu simulieren und alle Pfade des Moduls anzusteuern mit dem Ziel, alle logischen Verzweigungen zu überdecken [IEEE1008]. Um dies zu erreichen, sind bei komplexen Modulen mehrere hundert Testfälle erforderlich. Bei jedem Testfall

werden die Eingabedaten variiert, die einmal über eine Parameterschnittstelle, einmal über eine Dateischnittstelle oder einmal über eine Benutzungsschnittstelle in das Modul einfließen. Je mehr Schnittstellen ein Modul hat und je mehr Daten die Schnittstellen haben, umso schwieriger und aufwändiger wird der Modultest. Hinzu kommt die Komplexität der Ablauflogik. Je mehr Verzweigungen und je tiefer die Verschachtelungen, desto mehr Testfälle werden benötigt, um alle logischen Verzweigungen zu überdecken.

Der Aufwand für einen konventionellen Modultest wird also durch folgende Elemente bestimmt [Lig90]:

- die Anzahl Modulschnittstellen,
- die Anzahl Daten in den Schnittstellen,
- die Anzahl Verzweigungen im Modul und
- die Anzahl der Verschachtelungsstufen.

In der Praxis sind prozedurale Module oft zu groß und ihre Schnittstellen zu komplex, um einen ordentlichen Modultest auszuführen. Der Aufwand, die Schnittstellen zu simulieren, ist auch mit einem Testrahmen zu hoch. Das kommt daher, dass die wenigsten Entwickler ihre Systeme im Hinblick auf die Testbarkeit der Module konzipieren. Als Ergebnis sind die Module weitgehend untestbar. Es ist nur dann sinnvoll, Module zu testen, wenn sie aus kleinem, überschaubarem Code mit beschränkten, wohldefinierten Schnittstellen bestehen.

Klassen in objektorientierten Sprachen sind mit prozeduralen Modulen nicht gleichzusetzen. COBOL oder PL/I Module entsprechen jeweils einer Quellcode-Datei (source member). In einigen OO-Sprachen wie z.B. C++ und Smalltalk-80 kann eine Quellcode-Datei mehrere Klassen beinhalten oder nur Teil einer Klasse sein. Um eine Klasse zu testen, ist es erforderlich, mehrere Source-Members heranzuziehen bzw. einen Ausschnitt aus einer Source-Member herauszuschneiden. In solchen Fällen ist das alte 1:1-Verhältnis zwischen dem Quellcode und dem Testobjekt nicht gegeben.

Klassen können wie prozedurale Module einen „globalen“ Datenbereich (Klassenvariablen) haben, der allen Objekten gemeinsam zur Verfügung steht. Zusätzlich gibt es meist Datenbereiche, die für jede Instanz angelegt werden (Instanzvariablen) und für alle Methoden der Klasse zugreifbar sind. Das sind die eigentlichen Objektdaten. Außerdem kann jede Methode eigene lokale Variablen haben. Jede Methode bzw. Operation hat einen eigenen Eingang mit einer eigenen Schnittstelle. Somit unterscheiden sich Klassen grundsätzlich von prozeduralen Modulen. Sie sind ergebnisgesteuert. Nicht die Klasse als Ganzes wird aufgerufen, sondern nur Einzelmethoden. Ergo wird eine Klasse über den Aufruf ihrer Methoden getestet. Jede öffentliche Methode ist also im Gegensatz zu den Prozeduren in konventionellen Modulen von außen erreichbar (Abbildung 6.1). Private und geschützte Methoden

sind zwar von außen nicht ohne weiteres aufrufbar, können aber für Testzwecke wie öffentliche Methoden behandelt werden [LaNa97].

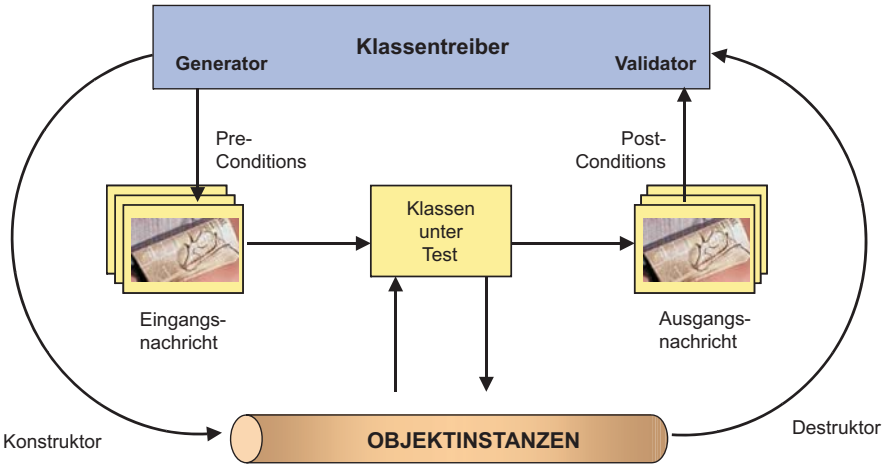


Abbildung 6.1 Klassentestumgebung

Objekte können, ebenso wie die Globaldatenbereiche der prozeduralen Welt, sehr groß werden, dennoch bleiben die Schnittstellen zu den Methoden in der Regel viel einfacher als die Modulschnittstellen. Die Methoden selbst sind in der Regel kleiner und von der Ablauflogik her weniger komplex. Dafür gibt es umso mehr Aufrufe fremder Methoden. Nicht selten bestehen Methoden fast ausschließlich aus Verweisen auf bzw. Aufrufen von Methoden z.T. fremder Klassen. Daraus folgt eine große Abhängigkeit zu fremden Klassen.

Durch die Vererbung entstehen weitere Abhängigkeiten. Eine Klasse, die von einer (oder sogar mehreren) anderen erbt, hat einen direkten Zugang zu allen Methoden und Attributen der anderen Klassen. Insofern gehören sie zusammen in einen Adressraum. Wer die abgeleitete Klasse testet, testet die Basisklassen mit. Im Falle einer tiefen Vererbungshierarchie weitet sich der Klassentest dadurch aus zu einem Test der Klassenhierarchie. Dies widerspricht jedoch dem Sinn eines Modultests, der darauf zielt, ein isoliertes Testobjekt zu bestätigen. Um eine Klasse wie ein Modul zu testen, ist es notwendig, die Klasse sowohl von den fremden Klassen als auch von ihren Ahnenklassen zu isolieren. Wie dies zu bewerkstelligen ist, ist ein Thema für sich. Es genügt vorerst, auf diese Problematik und den Unterschied zum klassischen Modultest hinzuweisen [Bin94a].

Der Hauptunterschied zwischen prozeduralen Modulen und Klassen liegt letztendlich darin, dass die Module größere interne Komplexität, die Klassen hingegen eine größere externe Abhängigkeit haben. Die Methoden einer Klasse sind zwar leichter

zu erreichen und ihre Schnittstellen leichter zu bedienen, aber es ist umso schwieriger, die Klasse getrennt von ihrer Umgebung zu testen. Die Instanzierung der Objektdaten mit unterschiedlichen Zuständen kommt entscheidend dazu. Es ist daher keineswegs leichter, Klassen zu testen. Beide Testgegenstände – Klassen und prozedurale Module – haben ihre eigene Problematik. Es ist schwer zu sagen welche testbarer ist.

Robert Binder unterscheidet Klassen anhand von vier Arten der Zustandsabhängigkeit ([Bin96a], s. Abbildung 6.2) in

- nonmodale,
- unimodale,
- quasimodale und
- modale Klassen.

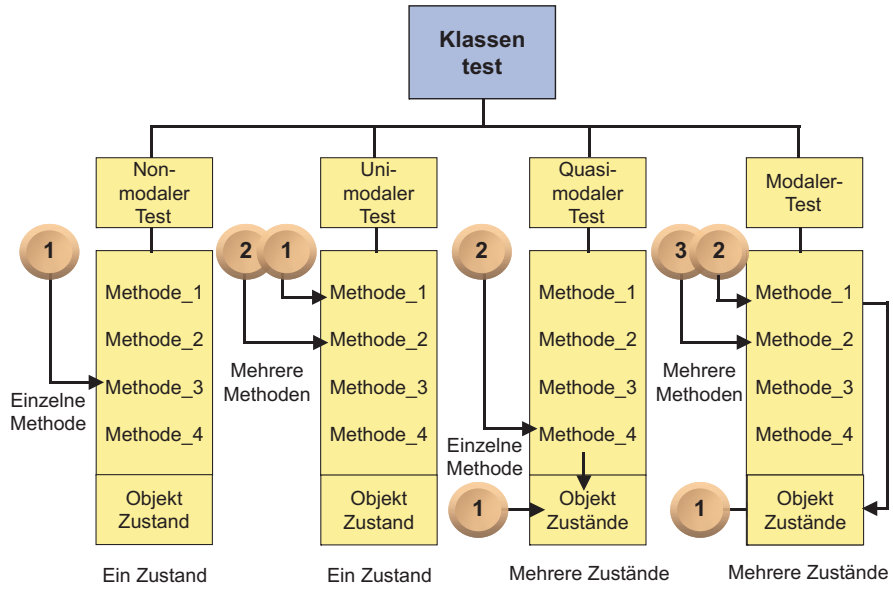


Abbildung 6.2 Zustandsbasierte Klassentestarten

Nonmodale Klassen haben keine Abhängigkeiten zwischen den Methoden. Die Methoden können in jeder beliebigen Reihenfolge aufgerufen werden, ohne einander gegenseitig zu beeinflussen. Der Zustand der Objekte hat keinen Einfluss auf das Verhalten der Methoden. Solche Klassen sind am einfachsten zu testen.

Unimodale Klassen setzen voraus, dass ihre Methoden in einer gewissen Reihenfolge ausgeführt werden. Der Zustand, der von der letzten Methodenausführung hinterlassen wird, ist immer eine Voraussetzung für die Ausführung der nächsten Methode. Der Tester muss auf diese Ausführungsfolge achten, z.B. muss das Konto

im ersten Schritt eröffnet werden, und erst im zweiten Schritt darf z.B. eingezahlt werden. Sonst müsste eine Fehlermeldung erfolgen.

Quasimodale Klassen sind vom Zustand der Objekte abhängig, egal welche Methode vorher ausgeführt wurde. Das Ergebnis der Methodenausführung wird anders ausfallen, je nachdem in welchem Zustand sich das Zielobjekt befindet. Z.B. wenn das Konto gesperrt ist, werden alle Bewegungen abgelehnt, egal in welcher Reihenfolge sie kommen.

Modale Klassen sind sowohl von der Reihenfolge der Methodenaufrufe als auch vom Zustand der Objekte abhängig. Um einen gewissen Fall zu testen, müssen andere Testfälle vorausgegangen sein, und das Objekt muss einen bestimmten Zustand haben. Ein Beispiel dieser Art Klasse wäre, wenn eine Überweisung nur auf eine Einzahlung folgen kann und der Kontostand größer als der Überweisungsbetrag sein muss. Derartige Klassen sind am schwierigsten zu testen. Der Tester muss nicht nur auf die Reihenfolge der Methodenausführung, sondern auch auf den Zustand der Objekte und auf die Werte bestimmter Member-Variablen achten.

6.2 Zweck des Klassentests

Es ist das Hauptziel der Software-Qualitätssicherung, Fehler in der Entwicklung dort zu entdecken, wo sie entstehen. Die meisten Fehler entstehen bereits im Fachkonzept. Fachkonzepte sind in der Regel unvollständig, inkonsistent und inkorrekt, aber keiner merkt es, weil sie informell verfasst sind. Aus dem Fachkonzept wird ein technisches Konzept, vielleicht in Form eines Objektmodells, abgeleitet. Obwohl wesentlich formaler, ist das technische Konzept, sprich Objektmodell, ebenfalls unvollständig, inkonsistent und fehlerhaft, aber keiner merkt es, weil das Modell nicht testbar ist. Durch die Vereinheitlichung der technischen Konzeptdokumentation über die Unified Modelling Language (UML, [OMG99]) ist das Problem der Fehlererkennung nur geringfügig besser geworden, da UML-Dokumente nicht wesentlich formaler sind als die herkömmlichen [Bur97].

Erst wenn aus den Klassenentwürfen echte Klassen in einer formalen, maschinenlesbaren Syntax kodiert oder generiert werden, kann der erste dynamische Test beginnen. Alles, was bis dahin in punkto Qualitätssicherung geschehen ist, gehört eher zur Rubrik statische *Prüfung*. Die fertig implementierten Klassen sind die ersten dynamisch testbaren Software-Einheiten. Also ist der Klassentest der erste Schritt in der Testdurchführung. Außerdem wird eine Klasse in der Regel von einem Entwickler geschrieben. Nachher, im Integrationstest, sind mehrere Klassen betroffen, sodass auch mehrere Entwickler involviert sind.

Der Klassentest gibt dem Entwickler somit die Gelegenheit, die eigenen Fehler zu finden, ehe andere dadurch betroffen sind. Der Klassentest gibt ihm auch eine so-

fortige Rückkopplung zur Lauffähigkeit und Korrektheit seines Codes. Cheatham und Mellinger stellten schon 1990 fest:

> *"Testing of classes is similar to unit testing of modules in conventional languages. Unit testing may begin as soon as a class is implemented." [ChMe90]*

Dennoch muss man sich darüber im klaren sein, dass sogar der beste Klassentest keine Fehlerfreiheit garantieren kann. Es werden allenfalls ein Drittel der potenziellen Fehler im Klassentest auftauchen, nämlich die Codier- und Logikfehler. Andere Fehler wie z.B. Spezifikations- und Entwurfsfehler können dort überhaupt nicht erkannt werden, weil die Klasse nur ein Teil des Ganzen ist.

Bekanntlich sind bis zu 50% aller Software-Fehler Versäumnisfehler, d.h. „Errors of Omission" [Gla81]. In anderen Worten: Was nicht vorgegeben ist, wird auch nicht kodiert! Es fehlen einfach Operationen, Fehlerbehandlungen, Bedingungen, Bedingungsklauseln oder Attribute. Sie fehlen im Fachkonzept sowie im technischen Konzept. Ergo werden sie auch im Code fehlen. Kein sonst so ausgeklügelter Klassentest wird sie jemals aufdecken können, es sei denn, der Klassenentwickler ist ein Domänexperte, dem auffällt, dass irgendetwas fehlt.

Darum darf man sich vom Klassentest nicht allzu viel versprechen. Er wird weder die Spezifikations- noch die Entwicklungsfehler zum Vorschein bringen. Schnittstellenfehler werden auch erst an die Oberfläche kommen, wenn mehrere Klassen miteinander kollaborieren. Wozu dann ein Klassentest?

Der Klassentest ist wichtig, weil er dem Entwickler die Gewissheit bringt, dass seine Klasse so funktioniert, wie er sich das vorgestellt hat, d.h., der Code entspricht seinem Verständnis der Vorgabe. Alle vorgesehenen Argumente bzw. Eingangsparameter werden behandelt, alle spezifizierten Bedingungen werden erfüllt und alle erwarteten Ergebnisse werden erzeugt. Schließlich wird von allen potenziell verarbeitbaren Objektausprägungen ein repräsentativer Querschnitt produziert und verarbeitet. Mehr kann man vom Klassentest nicht erwarten; dennoch: darauf zu verzichten wäre töricht! Damit würden Probleme nicht gelöst, sondern nur verschoben.

6.3 Einschränkungen zum Klassentest

Es mag paradox klingen, aber die Hindernisse beim Klassentest sind im Wesentlichen identisch mit den Vorzügen der objektorientierten Programmierung. Es sind:

- die Vererbung,
- die Polymorphie,
- das Überladen der Parameter und
- die Wiederverwendung fremder Klassen [CTC+98].

Es könnte leicht der Verdacht aufkommen, dass die Väter der objektorientierten Sprachen mit dem Unittest nichts im Sinne hatten. Auf jeden Fall haben sie sich – bis auf Bertrand Meyer, den Vater der objektorientierten Programmiersprache Eiffel ([Mey88]) – wenig Gedanken darüber gemacht. Mit seinen `require` und `ensure` Zusicherungen hat Meyer einen testgetriebenen Entwicklungsansatz gewählt, aber Eiffel hat sich leider nicht durchgesetzt. Für den gewöhnlichen Entwickler ist sie zu anspruchsvoll. Ansonsten ist der Klassentest nur unter Berücksichtigung der folgenden Einschränkungen möglich.

6.3.1 Klassentest und Vererbung

Die Vererbung bestimmt die Reihenfolge des Klassentests. Da abgeleitete Klassen – ähnlich wie verschachtelte Prozeduren – die Daten und Operationen ihrer Oberklassen mit benutzen, ist es nicht möglich, abgeleitete Klassen für sich allein zu testen. Sie müssen zusammen mit ihren Oberklassen getestet werden. Wenn diese Klassen auch abgeleitet sind, werden ihre Oberklassen mitgetestet (Abbildung 6.3). Es ist wie Spaghetti essen: wer einen Strang herausholen will, bekommt die ganze Schüssel mit. Daraus ergibt sich eine strenge Reihenfolge, nach der getestet werden muss. Am Anfang sind die Klassen an der Spitze der Klassenhierarchie zu testen, d.h. jene, die keine Ahnen mehr haben. Sie sind die Wurzel des Klassenbaums. Von dort aus wird die jeweils nächste Schicht von abgeleiteten Klassen bis hin zu den Blättern am anderen Ende des Baums getestet.

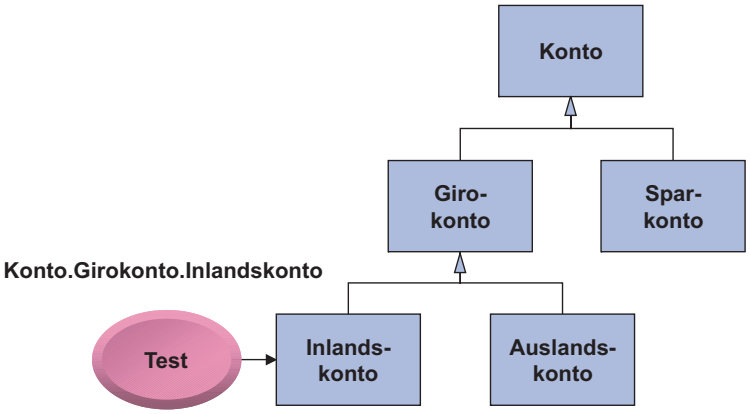


Abbildung 6.3 Klassentest mit Vererbung

Ursprünglich haben Verfechter der objektorientierten Programmierung behauptet, man brauche in den abgeleiteten Klassen nur die neu hinzugekommenen Methoden zu testen. Dies hat sich jedoch als Fehlannahme erwiesen. Da die Umgebungsbedingungen sich ändern, ist es schon erforderlich, auch die geerbten Methoden mit

zu testen. Sie werden als Bestandteile der abgeleiteten Klasse behandelt. Die dafür erforderliche Technik heißt „class flattening“ [Bin99].
Dies führt dazu, dass der Testgegenstand eines Klassentests sehr groß werden kann. Er umfasst nicht nur die Zielklasse, sondern alle darüber liegenden Ahnenklassen. Dies spricht gegen die Mehrfachvererbung und gegen tiefe Vererbungshierarchien. Beide erschweren den Klassentest.

6.3.2 Klassentest und Polymorphie

Die dynamische Bindung erschwert die Kontrolle des Klassentests, denn wenn es aus dem Text der Klasse nicht ersichtlich ist, welche Variante einer Operation ausgeführt wird, ist es auch nicht determinierbar, welches Ergebnis zurückkommt. Z.B. wird eine fremde Methode namens „Validation“ aufgerufen. Diese Methode gibt es in Dutzenden Klassen. Welche Ausprägung davon wirklich aufgerufen wird, wird erst zur Laufzeit entschieden (Abbildung 6.4). Thuy meint, man müsse beim Klassentest alle möglichen Ausprägungen testen:

> *“... class coverage is complete only when all the redefinitions of the called method have been exercised.” [Thu93]*

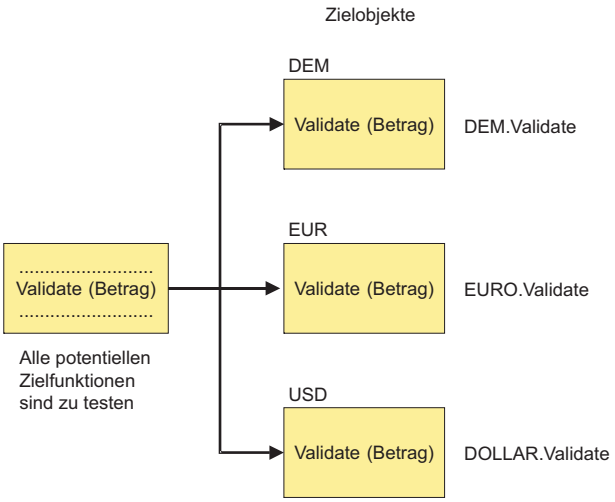


Abbildung 6.4 Klassentest mit Polymorphie

Dies würde bedeuten, dass der Tester einen Teststellvertreter (stub) für jede Ausprägung der Zielmethode schreibt, z.B. `Betrag.validation()`, `Name.validation()`, `Datum.validation()`, usw. Um nicht ins Uferlose zu geraten, muss der Tester hier eine Entscheidung treffen, welche Validations-Operationen für

die Klasse unter Test relevant sind, und ihre Ergebnisse simulieren. Dies bedeutet allerdings eine Einschränkung in der Testüberdeckung.
Andererseits, falls die polymorphe Methode zur Klasse unter Test gehört, muss der Tester im Testtreiber alle möglichen Aufrufe mit allen möglichen Parametervariationen einbauen. D.h., wenn die Klasse Datum getestet wird, muss die dazugehörige Methode „Validation“ für alle möglichen Datumsarten wie z.B. Julianisch, Gregorianisch usw. getestet werden. Dadurch führt die Polymorphie zu einer Explosion von Testfällen.

6.3.3 Klassentest und Überladen von Parametern

Die dynamische Veränderung der Parameter ist eine beliebte Technik der Objektorientierung zur Erzielung unterschiedlicher Ergebnisse mit ein- und demselben Operationsaufruf im Code. Dies erhöht zwar die Flexibilität der Anwendung, erhöht aber gleichzeitig den Testaufwand, denn es müssen alle potenziellen Parameterkombinationen getestet werden bzw. mindestens jene, die am wahrscheinlichsten vorkommen. Diese Technik erschwert auch die Generierung eines Testtreibers, weil der generierte Testtreiber nur die eine Standard-Parameterliste kennt. Die verschiedenen Variationen müssen manuell im Testtreiber angepasst werden. Auch dies erhöht den Aufwand für den Klassentest [SmRo92].

6.3.4 Klassentest und Wiederverwendung

Wenn der Tester wirklich davon ausgehen könnte, dass fremde Klassen in jedem beliebigen Zusammenhang wiederverwendet werden könnten, wäre es kein Problem, sie in den Klassentest einzubinden. Leider ist dies jedoch selten der Fall. Fremde Klassen sind ebenso eine Fehlerquelle wie die eigenen. Das hat zur Folge, dass sie im Klassentest entweder durch Stubs zu simulieren oder direkt mit zu testen sind. Dieses „Mittesten“ bedeutet, ihre Rückgabewerte genau zu kontrollieren, z.B. durch Zusicherungen, um auszuschließen, dass sie die eigenen Klassen nicht mit ungültigen Ergebnissen korrumpieren [Ros95].
Zusammenfassend ist festzustellen, dass der Klassentest um einiges komplizierter und aufwändiger ist als der konventionelle Modultest. Er setzt anspruchsvollere Testwerkzeuge und eine weitaus höhere Ausbildung der Tester voraus. Insofern hat Beizer Recht, wenn er von einer Kostensteigerung von 300% spricht[Bei99]. Dies kann durchaus zutreffen.

6.4 Theoretische Ansätze zum Klassentest

Robert Binder unterscheidet zwei Arten des Klassentests [Bin96b]:

- implementierungsbezogen und
- spezifikationsbezogen.

6.4.1 Implementierungsbezogener Klassentest

Der implementierungsbezogene Klassentest geht vom Quellcode der Klasse aus. Dort werden die Methoden, Verzweigungen, Operationsaufrufe und Parameter identifiziert und der Test so konstruiert, dass die Methoden mit allen Parameterkombinationen aufgerufen, alle Verzweigungen durchlaufen und alle Operationsaufrufe betätigt werden. Binder betont, dass dies eigentlich für jede Instanzierung der Klasse bzw. für jede Objektausprägung erforderlich wäre. Da der Klassentest irgendwie abzugrenzen ist, müssen die Aufrufe fremder Operationen irgendwo und irgendwie abgefangen werden. Binder gibt dafür jedoch kein Rezept. Der Klassentest wird von einem Klassentesttreiber aus getrieben, der die Methoden der Klasse unter Test auslöst [Bin94b].

Stellvertretend für den implementierungsbezogenen Ansatz stellen wir im Folgenden die Methoden von Smith und Robson, McGregor und Sykes, Thuy, Hoffman und Strooper, Turner und Robson, Kung, Parrish, D'Souza und Leblanc, Overbeck, McCabe sowie Harrold und Rothermel vor.

Smith und Robson haben die FOOT-Methode vorgeschlagen – „framework for object-oriented testing". Danach werden Testfälle automatisch aus dem Source-Code generiert. Vererbung wird simuliert, indem alle geerbten Operationen und Daten in den Quellcode der Zielklasse eingebaut werden – class flattening. Fremde Operationen werden durch Stubs simuliert. Getestet wird, ob der Code in sich konsistent ist und ob alle Objektzustände dem Sollzustand entsprechen. Schnittstellen nach außen werden nicht getestet, da die Klasse unter Test nach außen völlig abgeschottet ist. Bestätigt wird eigentlich nur die Ausführbarkeit und logische Korrektheit der einzelnen Methoden [SmRo92].

McGregor und Sykes setzen voraus, dass der Entwickler Vor- und Nachbedingungen in den Klassencode einbaut. Die Vorbedingungen prüfen am Eingang jeder Methode, ob die Eingangsparameter und der Vorzustand des Zielobjekts stimmen. Die Nachbedingungen prüfen am Ausgang jeder Methode, ob die Rückgabewerte stimmen und ob der Nachzustand des Zielobjekts immer noch stimmt. Der Klassentest baut auf einer Interpretation jener Pre- und Post-Konditionen auf, um die entsprechenden Testfälle zu generieren. Somit wird die Klasse unter Test schrittweise interpretiert und ihre Zustände mit den zugesicherten Zuständen abgeglichen [MgSy92].

Thuy schlägt vor, den Klassentest auf die zusätzlichen und erweiterten Methoden zu beschränken. Die geerbten, unveränderten Methoden werden ausgeklammert. Für die Methoden, die getestet werden, gilt es, alle Logikzweige zu durchlaufen und alle Parameter-Kombinationen zu erproben. Zu diesem Zweck sind repräsentative Objekte zu erzeugen, sogar künstliche für die abstrakten Klassen [Thu92].

Hoffman und Strooper haben einen Klassentest auf der Basis eines Testgraphen vorgeschlagen. Ihre Methode heißt CUT für Class under Test. Demnach werden für jede zu testende Klasse drei Testklassen erstellt – eine Testgraphenklasse, eine Treiberklasse und eine Orakelklasse. Die Testgraphenklasse enthält die Testsequenzen bzw. die Zustandsübergänge der Methoden und deren Abhängigkeiten voneinander, einschließlich der Ausnahmebehandlung. Sie wird vom Entwickler verfasst bzw. aus den Zustandsübergangsdiagrammen generiert. Die Treiberklasse holt die Testfolgen aus dem Testgraphen und ruft die Methoden der Klasse unter Test auf. Sie wird aus der Testgraphenklasse abgeleitet. Die Orakelklasse prüft den aktuellen Zustand der Objekte gegen die in der Testgraphenklasse spezifizierten Zustandsübergänge und bestätigt ihre Korrektheit. Auch sie wird also aus der Testgraphenklasse abgeleitet. Somit bildet der Testgraph bzw. die Zustandsübergangsdiagramme den Ausgangspunkt für den Klassentest. Sie sollten deshalb auf der gleichen semantischen Stufe wie die Klasse selbst sein [HoSt93].

Turner und Robson haben ein Testszenario für C++-Objekte entwickelt. Sie weisen darauf hin, dass als Erstes die Konstruktor-Operation aufgerufen werden muss, um eine Objektinstanz anzulegen und als Letztes die Destruktor-Operation aufgerufen werden muss, um die Objektinstanz zu löschen. Dazwischen wird der Zustand des Objekts durch die restlichen Operationen verändert. Prinzipiell ist die Reihenfolge der restlichen Operationsausführungen beliebig. Um die Zwischenzustände der Objekte zu kontrollieren, ist es notwendig, Testszenarien mittels Zustandsübergangsdiagrammen zu bestimmen. Jeder Zustandsübergang versetzt das Zielobjekt in einen anderen Zustand, der mit Nachbedingungen vorausgesagt wird. Wenn der vorausgesagte Zustand nicht eintritt, ist dies ein Zeichen dafür, dass die betreffende Operation unkorrekt ist. Die Basis dieser Methode ist die Voraussage einer Reihe Objektzustände, die aufeinander folgen müssen, und die Bestätigung dieser Voraussage mittels Zusicherungen [TuRo93].

Kung hat einen Ansatz geprägt, der vom Code ausgeht. Zunächst werden aus den Klassen automatisch Zustandsdiagramme abgeleitet. Diese werden anschließend in einem endlichen Automatengraph zusammengeführt. Daraus geht hervor, welche Zustände überhaupt vorkommen können und welche Bedingungen erfüllt werden müssen, um zu jedem Zustand zu gelangen. Wenn also all jene Bedingungen erfüllt werden, werden auch alle Zustände hervorgerufen. Testfälle werden generiert, um alle Bedingungen zu erzwingen. Dieser Ansatz sichert zwar eine 100%ige Testüberdeckung, testet aber nur die Konsistenz der Bedingungen und Daten [KGH93].

Parrish verfolgt einen datenflussbezogenen Ansatz zum Klassentest. Der Datenfluss beginnt dort, wo eine Operation aufgerufen wird, und endet mit einem Return aus der Klasse. Dazwischen fließen Werte durch die Objektattribute und interne Variablen. Es ist das Ziel, alle Datenveränderungen aufgrund von SET- und USE-Operationen zu definieren und zu Testfällen zusammenzufassen. Ein Testfall entspricht einem Datenflusspfad durch die Methoden. Als Ausgangspunkt empfiehlt Parrish die Returnwerte. Sie sollten durch den Zufluss anderer Werte zurückverfolgt werden [PBC93].

D'Souza und Leblanc haben sich auf das Problem der Objektreferenzen konzentriert. Sie behaupten, die Hauptfehlerquelle in den Klassen liege in der falschen Referenzierung der Objekte – man meint z.B., man würde die dritte Objektinstanz verarbeiten, wenn man in Wirklichkeit die zweite verarbeitet. Zu verhindern wäre dies über die Suche nach Aliaszeiger in einem Trace. Alle Objektreferenzen und deren Zuweisungen werden registriert und in einem Protokoll festgehalten, welches dem Tester dann zwecks einer visuellen Kontrolle präsentiert wird [DsLe94].

Overbeck hat sich mit dem Test generischer Klassen beschäftigt. Generische Klassen sind von zweierlei Art: parametrisierte Klassen, die neue Klassen schaffen (in C++-Templates) und abstrakte Oberklassen bzw. Operationen, die nur eine Schnittstelle ohne die dazugehörige Implementierung spezifizieren (in C++ als `virtual` gekennzeichnet, in Java als `abstract` gekennzeichnete Klassen und Operationen sowie Interfaces). Overbeck behauptet, auch diese Klassen müssen instanziert werden, zumindest beim Test. Zu diesem Zweck verwandelt er sie in einfache Klassen mit Konstruktor und Destruktor und ruft sie von einem Testtreiber auf, um ihre Methoden mit den künstlich erzeugten Objekten zu testen. Nur so lässt sich verhindern, dass unkorrekte generische Klassen ihre Fehler wie Viren verbreiten [Ove94].

Harrold und Rothermel haben eine Methode für die Modellierung der Interaktionen zwischen Operationen einer Klasse erfunden. Sie behaupten, vielleicht zu Recht, der Schlüssel zum Klassentest liege nicht im Test einzelner Methoden, sondern im Test der Abhängigkeiten zwischen Methoden. Ihre Methode basiert auf drei Arten von Datenfluss: dem Datenfluss innerhalb einer Methode, dem Datenfluss zwischen zwei Methoden und dem Datenfluss über alle Methoden einer Klasse hinweg. Durch eine statische Analyse des Klassen-Source-Codes werden diese Datenflüsse identifiziert und in einer Treiberklasse als gerichteter Graph abgebildet. Die Treiberklasse ist sozusagen eine Straßenkarte der Klasse unter Test, wobei die Straßen die Datenflüsse sind. Dieser Testrahmen benutzt DEFINE/USE-Paare, um Testfälle nach dem PLR-Algorithmus zu generieren, um jeden Datenflusspfad bzw. jede Straße durch die Klasse unter Test – CUT – zu „befahren". Problematisch wird es, wenn verschiedene Variablen das gleiche Objekt refernzieren (aliasing) [HaRo94].

6.4.2 Spezifikationsbezogener Klassentest

Brad Cox, einer der Väter der objektorientierten Programmierung, hat auch einen Beitrag zum Klassentest gemacht. Er schlug vor, alle Klassen zweimal zu schreiben, einmal in einer formalen Spezifikationssprache und einmal in einer Implementierungssprache. Die Implementierung der Klasse ist praktisch eine Spezialisierung der Klassenspezifikation. Damit wird es möglich, die beiden Repräsentationen miteinander abzugleichen, und zwar sowohl statisch als auch dynamisch [Cox88]. Statisch ist es möglich, die Signaturen, Bedingungen und Ablauflogikmuster zu vergleichen. Dynamisch ist es möglich, die Spezifikation zu analysieren, um modellhafte Ablaufpfade zu gewinnen, die gegen die tatsächlichen Ablaufpfade der Klasse geprüft werden können. Aus dieser Forschung ging das ASSERT-Makro in Objective-C hervor [CoNo91]. Es ist das Muster für andere Spezifikations/Test-Konstrukte geworden – auch für die `ensure`- und `require`-Anweisungen in der Programmiersprache Eiffel [Mey92].

Doong und Frankl haben schließlich schon 1994 ein allumfassendes Klassentestsystem namens ASTOOT entwickelt. Wie Cox vor ihnen gehen auch sie von einer übergeordneten Klassenspezifikation aus. Ihre Sprache LOBAS benutzt das Prädikatenkalkül erster Ordnung, um die Relation zwischen Objektvor- und nachzuständen sowie zwischen Parametereingangswerten und Rückgabewerten zu beschreiben. Diese abstrakte Datentypspezifikation wird von einem Treibergenerator gelesen, der daraus einen Testtreiber in der Zielsprache generiert. Der Treiber versorgt die Vorbedingungen, ruft die Klassenmethoden auf und bestätigt die Nachbedingungen. Somit dient das ASTOOT-System als Modell für künftige Klassentestwerkzeuge. Das Wesentliche daran, nämlich die algebraische Spezifikationssprache LOBAS, erwies sich jedoch als zu schwierig bzw. zu aufwändig für den durchschnittlichen Klassenentwickler. Dies ist ein wiederkehrendes Thema in fast allen Versuchen, das Klassentestproblem zu lösen. Um die Korrektheit einer Klasse zu verifizieren, muss die Klassenimplementation gegen eine semantisch äquivalente Klassenspezifikation getestet werden. Und dies setzt wiederum voraus, dass jemand den gleichen Aufwand in die Klassenspezifikation investiert wie in die Klassenimplementierung [DoFr94].

6.5 Praktische Ansätze zum Klassentest

Zwischen theoretischen und praktischen Ansätzen zur Lösung eines Problems, wie dem Testen von Klassen, läuft eine äußerst fragwürdige Grenze – eine Grenze, die durch die Einstellung der Anwender geprägt wird. In diesem Falle sind die Anwender die Entwickler, und Entwickler sind grundsätzlich gegen alles, was sie von ihrer Lieblingsbeschäftigung abhält, der Schaffung immer neuer, immer ausgeklügelterer Codekonstruktionen. Der Test ihrer Lösungen wird als eine unangenehme Pflicht

angesehen, die so schnell wie möglich zu entledigen ist. Darum sind sie grundsätzlich gegen alles, was ihnen allzu viel abverlangt. Wenn eine Testmethode schon sein muss, dann eine, die möglichst einfach zu lernen und anzuwenden ist. Entscheidend also für den Übergang eines theoretischen Testansatzes in die praktische Anwendung ist die Einfachheit und leichte Erlernbarkeit der Methode, des Grads der Toolunterstützung und die Einsicht, dass er dazu führen kann, Fehler zu finden, die der Entwickler sonst nicht finden würde.

6.5.1 Klassentesttreiber

Schon in den 70er Jahren gab es Testtreiber für den Test der Module. RXVP, eines der ersten allumfassenden Testsysteme für den Test der Programme im amerikanischen Antiballistic Missile Verteidigungssystem, generierte Testtreiber für FORTRAN-Subroutinen aus dem Source-Code heraus [Hay86]. Parallel dazu wurden Stubs generiert, um die Aufrufe der Subroutinen unter Test abzufangen und die erforderlichen Ergebnisse zu simulieren. Einer der Autoren dieses Buches hatte selbst ein ähnliches Testsystem namens PRÜFSTAND für den Test von Assembler- und SPL-Modulen bei der Siemens AG in München entwickelt. PRÜFSTAND hatte alle Eigenschaften, die von einem Modultestsystem zu erwarten sind – einen Testtreiber, der das Modul unter Test aufrief, Teststubs, die fremde Unterprogramme simulierten, einen dynamischen Analysator, um die Testüberdeckung zu messen, einen Ablaufverfolger, um die Pfade durch den Testgegenstand zu protokollieren, und einen Validator, um die Zwischenergebnisse mit Sollergebnissen abzugleichen [Sne83a]. Ähnliche Testwerkzeuge wurden später weltweit entwickelt, Werkzeuge wie Testbed in England, Verilog in Frankreich und die Test-Werkzeuge von McCabe und Miller in den USA. In Deutschland hat einer der Autoren seine bei Siemens angefangene Arbeit in die IBM-Welt übertragen und dort eine Modultestumgebung – SOFTEST – für die Verifikation von COBOL- und PL/I-Modulen gegen eine Testspezifikationssprache entwickelt und in der Praxis eingesetzt [SnMa83].

Als die objektorientierte Programmierung aufkam, gab es bereits in der konventionellen Programmierwelt mehrere ausgereifte Modultestansätze. Es lag also nahe, diese Ansätze auf die Objekttechnologie zu übertragen. Die ersten neuen OO-Testwerkzeuge waren in der Tat die letzten alten prozeduralen – VERILOG, McCabe, Testbed und Cantata wurden alle auf den Klassentest umgestellt.

Ein Klassentesttreiber kennt die Operationsschnittstellen der Zielklasse und kann die Operationen der Reihe nach aufrufen (Abbildung 6.5). Das Problem liegt darin, die richtigen Parameterwerte zu besorgen. Entweder werden sie vom Tester über eine Benutzungsoberfläche eingegeben oder vom Tester vor der Testausführung in einem Testskript festgelegt. Das Gleiche gilt für die Ergebnisse. Um sie zu validieren, müssen sie entweder angezeigt und manuell geprüft oder gegen vorbestimmte,

im Testskript enthaltene Sollwerte verglichen werden. Darüber hinaus stellen sich die anderen klassenspezifischen Anforderungen, die der Testtreiber irgendwie zu erfüllen hat: Vererbung, Polymorphie, mehrfache Instanzierung, Überladen von Parametern usw. Es hat sich also bald herausgestellt, dass konventionelle Testtreiber schnell an ihre Grenzen stoßen. Der Aufwand, einen guten Klassentesttreiber zu generieren, ist unvergleichbar hoch im Verhältnis zum konventionellen Modultesttreiber [Fie89].

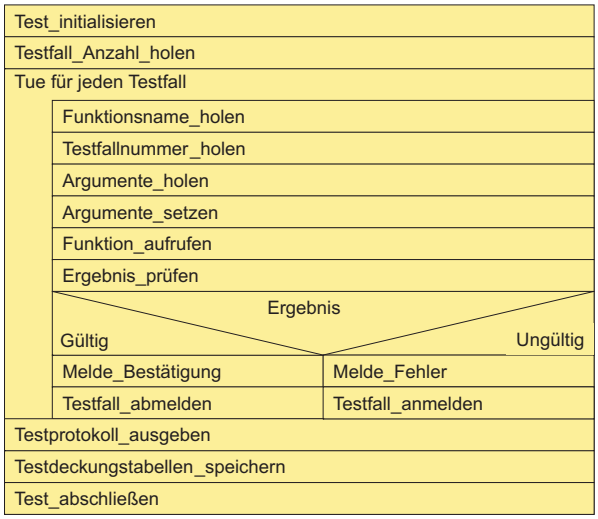


Abbildung 6.5 Klassentesttreiber

6.5.2 Build-In Tests

Diese Erkenntnis brachte viele Klassentestpraktiker dazu, einen anderen Weg einzuschlagen. Statt die Klasse von außen über eine Treiberklasse zu testen, haben sie den Test in der Klasse eingebaut. Nach der Build-In Testtechnik gibt es eine oder mehrere Testoperationen in der Klasse, von wo aus die anderen Methoden aufgerufen werden (Abbildung 6.6). Dies hat den Vorteil, dass die eingebauten Testoperationen Zugang zu den Objektzuständen haben und sie manipulieren können. Vor dem Operationsaufruf wird der Vorzustand des Objekts von der Testoperation festgelegt. Nach dem Operationsaufruf wird der Nachzustand derselben abgefragt [Fir96].

Ein weiterer Vorteil der Build-In Testtechnik ist das Fehlen zusätzlicher Quellcode-Dateien. Mit einem Testtreiber hat der Entwickler eine weitere Quellcode -Datei, die er parallel zum Klassensource fortschreiben muss. Mit dem Build-In Test hat er alles in einer Quellcode-Datei und kann beides – die Implementierungs- und die Testoperationen – gemeinsam fortschreiben. Auch die Stub-Operationen, die frem-

de Operationsaufrufe simulieren, sind als Member-Operationen derselben Klassen enthalten.

Der Nachteil dieser Technik ist, dass sie den Source-Code aufbläht – er wird ca. 50 bis 100% größer – und dass sie über Compiler-Optionen ein- und ausgeschaltet werden muss, d.h. die Testoperationen und Operations-Stellvertreter sind in `IFDEF`-Makros eingebettet.

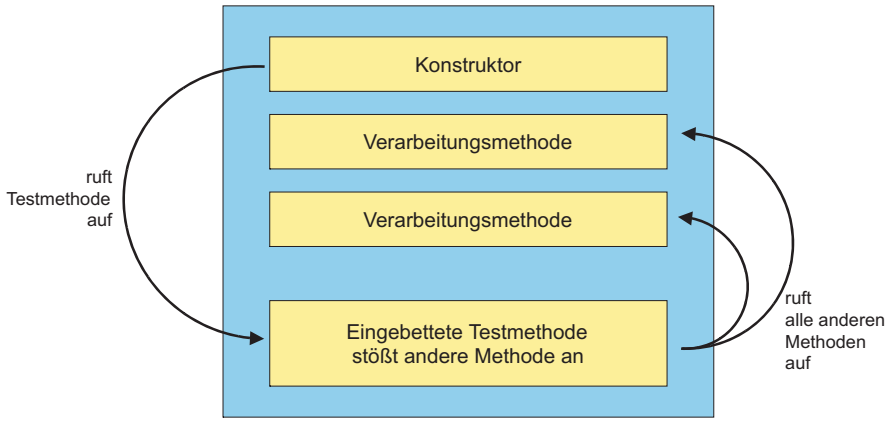


Abbildung 6.6 Build-in Klassentest

6.5.3 Zusicherungstest

Der Build-In Test stützt sich weitgehend auf die Verwendung der Zusicherungstechnik. Die Zusicherungs- oder Assertions-Technik wurde schon von Brad Cox in Objective C eingeführt. In der Sprache Eiffel spielt sie eine entscheidende Rolle. Bertrand Meyer, der Vater von Eiffel, sieht vor, dass Zusicherungen benutzt werden, um Eiffel-Klassen zu spezifizieren [Mey92].

Operationen werden in der Schnittstelle durch Angabe ihrer Signatur (Parameter und ihre Typen bzw. Klassen) und ihrer Semantik beschrieben. Die Semantik wird dabei durch Vor- und Nachbedingungen sowie durch eine Klasseninvariante präzisiert. Die *Vorbedingung* definiert, welche Voraussetzungen vor dem Aufruf einer Operation gelten müssen, damit die (durch die Nachbedingung) definierte Leistung erbracht werden kann. Diese Voraussetzungen beziehen sich gewöhnlich auf die Zustände der aktuellen Parameter und den Zustand des dienstleistenden Objekts. Die *Nachbedingung* definiert das durch das dienstleistende Objekt abgelieferte Ergebnis der Operation unter der Annahme, dass die Vorbedingung beim Aufruf erfüllt war. Die Nachbedingung bezieht sich gewöhnlich auf den Zustand des dienstleistenden Objekts und – falls vorhanden – des Rückgabeparameters sowie ggf. auf die Zustände assoziierter Objekte nach der Operationsausführung. Die

(*Klassen-)Invariante* erlaubt es, gemeinsame Teile aus allen Vor- und Nachbedingungen der Operationen der Klasse herauszuziehen und nur einmal zu notieren. Die Klasseninvariante gilt vor und nach jeder Ausführung jeder Operation der Klasse.

Wenn ein Entwickler Objekte konzipiert, unterscheidet er gleich zwischen varianten- und invarianten Zuständen. Invariante Zustände entsprechen den Integritätsregeln in Datenbanken. Sie müssen immer erfüllt sein, sonst sind die Daten fehlerhaft. Z.B. gilt als Regel, dass eine Firma keine Angestellten jünger als 18 Jahre oder älter als 60 Jahre hat. Die Invariante dazu würde heißen:

```
invariant
   Alter => 18 und Alter <= 60
```

Falls ein Angestelltenobjekt mit einem Alter außerhalb dieses Bereiches auftaucht, wird eine Ausnahme geworfen.

Wenn der Entwickler Methoden konzipiert, spezifiziert er gleich Vor- und Nachbedingungen zu jeder Methode. Die Vorbedingungen werden mit einer `require` Zusicherung, die Nachbedingungen mit einer `ensure` Zusicherung definiert. `require` stellt fest, ob das Objekt im erforderlichen Vorzustand ist, z.B.

```
Alter               = 30
Beruf              = Programmierer
Dienstjahre     = 5
Monatsgehalt = 5000
```

Nach Ausführung der Methode `Gehaltserhöhung()` prüft `ensure`, ob das Objekt den gewünschten Nachzustand hat, z.B.

```
Monatsgehalt = 6000
```

Insgesamt gesehen könnte die Klasse `Angestellter` in Eiffel so spezifiziert werden:

```
class ANGESTELLTER [P] export
       Alter, Beruf, Dienstjahre, Gehalt, Gehaltserhöhung
       feature
              Alter: INTEGER;
              Beruf: STRING;
              Dienstjahre: INTEGER;
              Gehalt : FLOAT;
       Gehaltserhöhung (Prozentsteigerung : FLOAT) : is
       require
              Gehalt >1000
              Alter   > 29
              Dienstjahre > 5
              Prozentsteigerung > 0
       ensure
              Gehalt = old.Gehalt +
```

```
            old.Gehalt x Prozentsteigerung
    end;
    invariant
           Alter => 18 and Alter <= 60
           Gehalt => 1000 and Gehalt <= 9000
    end
```

Im Weiteren stellen wir mit dem *Zusicherungstest* eine auf Zusicherungen aufbauende Testtechnik vor. Das Ziel des Zusicherungstests (auch „Vertragstest" genannt, [Win01]) ist die Prüfung der Schnittstelle einer Klasse, also der öffentlich sichtbaren Operationen (und Instanzvariablen).

Der Nutzen des Zusicherungstests besteht darin:

- die Konformanz der Operationen einer Klasse zu ihrer Spezifikation zu prüfen,
- das Zusammenspiel mehrerer Operationen zu prüfen und
- die für die Operationen möglichen Ausnahmen (exceptions) zu erzeugen.

Die Voraussetzung für den Zusicherungstest ist das Vorliegen von Zusicherungen (constraints, assertions) für jede Operation der zu testenden Klasse (Klasse Unter Test, KUT) sowie der Klasseninvariante der KUT. Das Ergebnis des Zusicherungstests ist eine Klasse, deren öffentlich sichtbare Operationen gegen die spezifizierenden Constraints (Verträge) getestet sind.

Man beginnt mit der Testfallerzeugung aus den Vorbedingungen. Da die Konformanztestfälle die Verträge einhalten müssen, wendet man z.B. die minimale Mehrfach-Bedingungsüberdeckung eingeschränkt auf insgesamt erfüllte Vorbedingungen an. Das bedeutet, für jede zusammengesetzte Vorbedingung Testfälle auszuwählen, sodass jeder Term mindestens einmal ausschlaggebend für den Wahrheitswert der gesamten Vorbedingung ist.

Invarianten schränken den Zustandsraum der Objekte einer Klasse ein und müssen vor und nach jeder Operationsausführung erfüllt sein, werden also im Falle zusammengesetzter Ausdrücke ebenso wie Vorbedingungen belegt. Das bedeutet, dass man jede Operation für jede mögliche Art erfüllter Invarianten testen sollte. Bei den Nachbedingungen will man auch mögliche Verletzungen der Nachbedingung erreichen – man befindet sich ja auf der Jagd nach Fehlern. Laut der minimalen Mehrfachbedingungsüberdeckung versucht man, Testfälle auch für nicht-erfüllte Nachbedingungen zu finden.

Bei der Festlegung der Testfälle geht man inkrementell vor, indem beginnend mit dem Konstruktor der Aufruf von Operationen der KUT simuliert wird, bis die geforderte Überdeckung der Terme erreicht ist. Jeder Testfall entspricht dem Aufruf einer oder mehrerer Operationen der KUT bei der entsprechend erfüllten Vorbedingung. Hierbei notiert man die erwartete Auswirkung der Operationsausführung auf die Vorbedingungen weiterer Operationen, wobei man bereits vollständig überdeckte Terme nicht mehr beachtet. Testfälle für diese Operationen können dementspre-

chend auf bereits ermittelte Testfälle aufbauen. Auf diese Weise fährt man fort, bis letztendlich alle Prädikate überdeckt sind.

Bei objektorientierten Client/Server-Anwendungssystemen müssen bestimmte Teilsysteme sehr robust, zuverlässig und deswegen defensiv realisiert werden. Darunter fallen insbesondere solche Teilsysteme, die nicht-lokale Dienste anbieten für Klienten, deren Verhalten nicht vorhersehbar ist. Ähnliches kann über die Bereitstellung von Bibliotheksklassen zur Wiederverwendung gesagt werden. Außerdem weisen einige objektorientierte Programmiersprachen wie beispielsweise Java und C++ Sprachmittel zur Ausnahmebehandlung auf. Zusätzlich sollte man also mit gezielten Verletzungen der Vorbedingungen einer Operation oder durch Manipulation der Testumgebung die spezifizierten Ausnahmen erzwingen.

Es wird im Wesentlichen die *Ausnahmeaktivierung* angewendet, bei welcher Ausnahmen durch bestimmte Kombinationen des Zustands der Komponente und der Parameterwerte eines Operationsaufrufs erzwungen werden. Hierzu betrachtet man die oben erstellten Testfälle. Sei `z()` eine Operation, für die eine Ausnahme spezifiziert ist. Ist nach einem Testfall *Y* die Vorbedingung von `z()` verletzt (`false`), so ergibt sich ein Robustheits-Testfall, indem man zunächst den Testfall *Y* „ausführt“ und dann `z()` aufruft. Erwartetes Ergebnis ist, dass die entsprechende Ausnahme geworfen wird. Auf diese Weise versucht man, alle Ausnahmen zu überdecken.

Voraussetzung für die zuletzt genannte Art der Testfallermittlung ist eine Programmiersprache, die eigene Konstrukte zur Behandlung oder zum Auslösen von Ausnahmesituationen aufweist. Die Menge der Testfälle ist dann so zu wählen, dass jede behandelte Ausnahmesituation mindestens einmal durchlaufen bzw. jede mögliche Ausnahmebehandlung mindestens einmal ausgelöst wird. Bei hochgradig defensiver Programmierung und einer kontrollierten Behandlung aller möglichen Ausnahmesituationen werden viele Ausnahmesituationen berücksichtigt, die nur höchst unwahrscheinlich oder unter regulären Bedingungen überhaupt nicht auftreten können. In einem solchen Fall kann es schwierig sein, den Testling in einen Zustand zu versetzen, in dem die Ausnahmebehandlungen getestet werden können.

Endekriterien für den Zusicherungstest sind:

- Jede Operation wurde mindestens einmal ausgeführt.
- Jede Zusicherung ist entsprechend der minimalen Mehrfach-Bedingungsüberdeckung geprüft.
- Jede Ausnahme wurde mindestens einmal geworfen.

Durch die Betonung der Zusicherungstechnik mit `require`, `ensure` und `invariant` verfolgt Meyer in Eiffel einen testgetriebenen Ansatz zur Klassenentwicklung. Der Entwickler beschreibt zuerst die Vor- und Nachbedingungen, ehe er daran geht, die Umsetzungsregel zu kodieren (Abbildung 6.7). Somit wird erst an den Test gedacht, dann an die Implementierung und nicht wie sonst umgekehrt. Die Eiffel-Klassen entstehen als Testrahmen, die anschließend mit Operationen und

Attributen ausgefüllt werden. Dies ist überhaupt die beste Methode, Programmieren zu lernen. Es ist nur schade, dass Informatiker wie Bertrand Meyer so wenig Einfluss auf die industrielle Software-Fertigung haben.

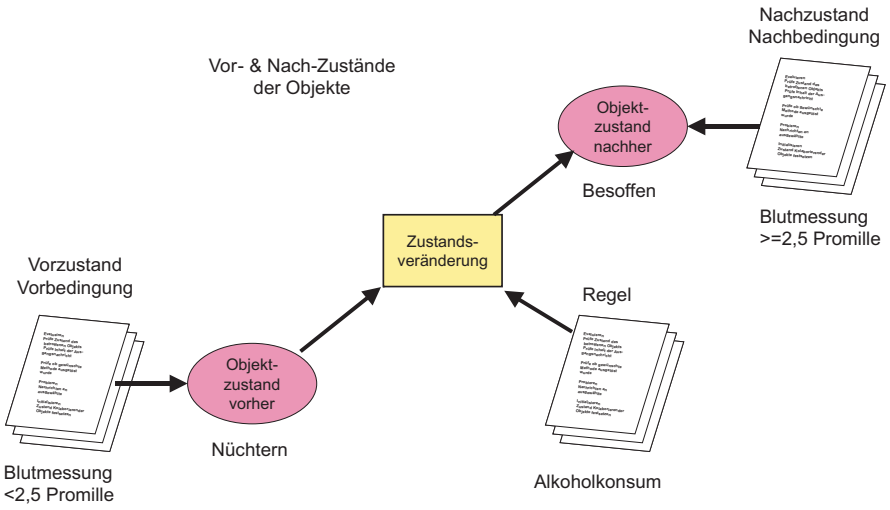


Abbildung 6.7 Zustandszusicherung

Immerhin, ganz ohne Wirkung ist die Meyers „Lehre der Zusicherungen“ nicht geblieben. Auch C++ hat ein `Assert`-Makro, und dieses wird sogar in einem Buch von Microsoft hoch gelobt: Steve Maguire stellt die Assert-Anweisung in den Mittelpunkt seines Buches „Writing Solid Code“ [Maq93]. Er empfiehlt seinen Lesern, ein `Assert` am Anfang jeder Operation zu plazieren, um die Operationsargumente zu prüfen und vor jedem `return`, um die eigenen Ergebnisse zu validieren. Außerdem soll ein `Assert` auf jeden fremden Operationsaufruf folgen, sowie nach jeder Schleife kommen. Damit kontrolliert man die Rückgabewerte der fremden Operationen und den Ausgang aus den eigenen Schleifen. Die `Assert`-Makros können über eine Compiler-Option beliebig ein- und ausgeschaltet werden, sodass sie keine Auswirkung auf die Performanz haben. Wenn sie verletzt werden, wird der Tester automatisch benachrichtigt, z.B. durch:

```
Assertion Violation
       at Line 64 in Function UnsToStr"
       Ist:          u = 65537
       Funk:  Void UnsToStr (unsigned u, char *str)
       Soll:  ASSERT (u >= 0 && u <= 65535)
```

Maguire ist ebenso wie Meyer überzeugt von der Rolle der Zusicherungstechnik beim Test einzelner Klassen. Sie ist besonders wichtig im Zusammenhang mit Testtreiber und Teststubs, um die Annahmen des Entwicklers zu bestätigen und unan-

genehme Überraschungen auszuschließen. Ihr gebührt eine zentrale Rolle in jedem praktischen Ansatz zum Klassentest. Kent Beck, Erfinder des „eXtreme Programming“ ([Bec99a]) und einer der „Gurus“ der objektorientierten Programmierung, behauptet, dass ein Entwickler mindestens genauso viel Testcode wie Produktionscode schreiben sollte [Bec94]. Es ist abzusehen, dass auch in Java entsprechende Konstrukte Einzug finden werden.

6.5.4 Zustandstest

Klassen oder ganze Teilsysteme werden oft so entworfen, dass sie eine Sequenz von Operationsaufrufen (Nachrichten) dann akzeptieren und behandeln, wenn die Nachrichten in einer gewissen Reihenfolge mit bestimmten Eingabewerten und unter entsprechenden Bedingungen eintreffen. Andernfalls sollten die Nachrichten zurückgewiesen und die dadurch angeforderten Dienste verweigert werden. In diesem Fall darf sich der (interne) Zustand des Anbieters nicht verändern.

Das geschilderte Verhaltensmuster kann häufig im Bereich von Client-/Server-Anwendungssystemen beobachtet werden bei Klassen oder Subsystemen, welche die Schnittstelle des Servers für seine Klienten realisieren. Für solche Dienstleister ist es besonders wichtig, dass sie robust und defensiv konzipiert und realisiert sind, was durch einen angemessenen Test nachgewiesen werden sollte.

Andererseits ist der Test von Client-/Server-Systemen nur dann praktikabel durchführbar, wenn die Serverschnittstelle robust und defensiv ausgelegt ist. Im anderen Fall müssten nämlich alle Klienten das Protokoll der durch den Anbieter akzeptierten Nachrichten genauestens kennen und strikt verfolgen. Vor dem Hintergrund, dass

- sich die Realisierung des Anbieters während seiner Lebenszeit ändern kann – mit möglichen Auswirkungen auf alle seine Klienten;
- nicht alle Klienten, die während der Lebenszeit des Gesamtsystems auf den Server zugreifen dürfen, von Anfang an bekannt sein müssen und
- die Einhaltung des Protokolls ggf. voraussetzt zu wissen, welche Dienste alle (anderen) Klienten bisher in Anspruch genommen haben,

ist die robuste und defensive Umsetzung des Dienstanbieters eine notwendige Anforderung an die Konstruktion der gesamten Anwendung, um deren Testbarkeit zu gewährleisten. Die angemessene Realisierung des Dienstanbieters nachzuweisen, ist ein Ziel des zustandsorientierten Tests, die Testfallermittlung wird dabei in zwei Schwerpunkte aufgeteilt:

- Testfälle für gültige Nachrichtensequenzen prüfen den Fall, dass das Protokoll eingehalten wird und der Anbieter ordnungsgemäß arbeitet.

- Testfälle für ungültige Nachrichtensequenzen prüfen die kontrollierte Reaktion des Anbieters auf nicht erlaubte Operationsaufrufe.

Das Ziel der Testtechnik *Zustandstest* ist der Nachweis, dass die zu testende Klasse (Klasse unter Test, KUT) konform zu dem das zustandsabhängige Verhalten der KUT spezifizierenden Zustandsdiagramm ist (Zustands-Konformanztest). Zusätzlich wird noch das Verhalten der KUT unter nicht-konformanten Benutzungen getestet (Zustands-Robustheitstest) [Win01]. Als Testmodell kommt der aus dem Zustandsdiagramm abgeleitete „Übergangsbaum" zum Einsatz. Hierdurch werden insbesondere solche Zustandsdiagramme, die Schleifen enthalten, mit endlich vielen Testfällen systematisch geprüft.

Zunächst fokussiert man das Zustandsdiagramm, welches das zustandsabhängige Verhalten der KUT spezifiziert. Wichtig ist, dass die an den Übergängen notierten Aktionen durch entsprechende Operationen der KUT implementiert werden. Darüber hinaus sollte die KUT Operationen bereitstellen, mit denen man prüfen kann, in welchem Zustand sich eine Instanz der KUT befindet. Ggf. sind hierzu die Werte von Instanzvariablen auf Zustände abzubilden. Schon in diesem Schritt achtet man darauf, ob das Zustandsdiagramm Zyklen enthält.

Als Nächstes leitet man den so genannten *Übergangsbaum* für den Zustands-Konformanztest ab, der bestimmte Folgen von Zustandswechseln repräsentiert. Ziel dieses Schritts ist es, aus den (bei zyklischen Zustandsdiagrammen potenziell unendlich vielen) möglichen Folgen von Übergängen in einem Zustandsdiagramm eine repräsentative Menge auszuwählen. Insbesondere sollen alle Zustände mindestens einmal eingenommen und alle Zustandsübergänge mindestens einmal ausgeführt worden sein.

Bei der Ableitung des Übergangsbaums werden gezielt Pfade durch das Zustandsdiagramm ermittelt, sodass jeder Zustand und jeder Zustandsübergang mindestens einmal in einem Pfad vorkommt. Der Übergangsbaum zu einem Zustandsdiagramm wird folgendermaßen erzeugt.

1. Der Anfangszustand wird die Wurzel des Baumes.
2. Für jeden möglichen Übergang von der Wurzel aus erhält der Baum eine Verzweigung zu einem neuen Knoten, der den Nachfolgezustand des Übergangs repräsentiert.
3. Der letzte Schritt wird für jedes Blatt des Baumes so lange wiederholt, bis eine der beiden Endbedingungen eingetreten ist:
   - Der dem Blatt entsprechende Zustand ist auf dem Weg von der Wurzel zum Blatt bereits einmal im Baum enthalten. Diese Endbedingung entspricht einem vollen Durchlauf von einem Zyklus in einem Zustandsdiagramm.
   - Der dem Blatt entsprechende Zustand ist ein Endzustand des Verhaltens und hat keine weiteren Übergänge.

Als Nächstes wird der Übergangsbaum so erweitert, das auch Testfälle bzgl. der Robustheit der KUT unter spezifikationsverletzenden Benutzungen abgeleitet werden können. Für jeden Knoten und alle Botschaften, für die aus dem betrachteten Knoten kein Übergang spezifiziert ist, wird hierzu der Übergangsbaum um einen Zustandsübergang in einen hinzuzufügenden „Fehler"- oder „Ausnahme"-Zustand erweitert.

Zur Testfallableitung werden nun die Pfade von der Wurzel zu den Blättern in dem erweiterten Übergangsbaum so als Botschaftssequenzen aufgefasst, dass durch die Stimulierung der KUT mit den entsprechenden Botschaften alle Zustände und Zustandsübergänge im Zustandsdiagramm abgedeckt werden.

Zur Erstellung der Botschaftssequenzen für den Zustands-Konformanztest durchläuft man den Übergangsbaum in einem Breitendurchlauf von der Wurzel zu solchen Blättern, die regulären Zuständen im Zustandsdiagramm entsprechen. Dabei notiert man die an den durchlaufenen Übergängen stehenden Botschaften bzw. Ereignisse und erhält so eine Menge von Botschaftssequenzen. Dasselbe erfolgt für den Robustheitstest unter Berücksichtigung der bei der Verfeinerung des Zustandsbaums zugefügten „Fehler"-Knoten.

Für die so ermittelten Botschaften werden unter Beachtung etwaiger Einschränkungen (Wächterbedingungen der Übergänge im Zustandsdiagramm) aus den Operationssignaturen passende Parameterwerte abgeleitet. Hierbei werden natürlich gleiche Teilsequenzen nur einmal parametrisiert. Sind im Zustandsdiagramm an den Übergängen die bei der Operationsausführung auszulösenden Ereignisse (Ausnahmen, weitere Operationsaufrufe) notiert, werden auch diese als Teil des erwarteten Ergebnisses mit in die Botschaftssequenzen aufgenommen.

Wenn einzelne Zustandsübergänge in der Spezifikation des Verhaltens an Bedingungen geknüpft sind, so müssen diese natürlich auch in erlaubte und nicht erlaubte Übergänge unterschieden werden. Erlaubt ist ein Übergang und dementsprechend die Annahme und Behandlung des daran geknüpften Operationsaufrufs genau dann, wenn die Bedingung im aktuellen Zustand erfüllt ist. Anhand dieser Einteilung müssen entsprechende Testfälle genauso wie erläutert definiert werden. Für den Spezialfall von zeitlichen Bedingungen ist die Verwendung einer Grenzwertanalyse für das betroffene Zeitintervall ratsam.

Die so erstellten Testfälle bzw. Botschaftsfolgen werden in Testprozeduren verkapselt und unter Benutzung eines Testtreibers ausgeführt. Dabei werden die erreichten Zustände über die zustandserhaltenden Operationen der KUT ermittelt und protokolliert.

Endekriterien für den Zustandstest sind:

- Jeder Zustand wurde mindestens einmal eingenommen.
- Jeder Zustandsübergang wurde mindestens einmal ausgeführt.
- Alle nicht spezifizierten Zustandsübergänge wurden angeregt.

Gemeinsam mit dem Test auf Einhaltung der Verhaltensspezifikation kann als Testziel für Komponenten, die in dieser Beziehung als kritisch einzustufen sind, die vollständige Überdeckung aller Zustände und aller Operationen gefordert und durch die entsprechende Überdeckungskenngröße (vgl. Abschnitt 11.2.5.1) gemessen werden. Bei der Testdurchführung sollte insbesondere auf die Unabhängigkeit der einzelnen Testfälle voneinander geachtet werden, das heißt für jeden Testfall, dass der Testgegenstand erneut vom definierten Anfangszustand in den zu prüfenden Zustand gebracht werden sollte.

Als Sonderformen des Zustandstests gelten weiterhin

- der partielle Zustandstest,
- der Test aller Zustandsübergangskombinationen und
- der Test aller Zustandsübergänge in jeder beliebigen Reihenfolge mit allen möglichen Zuständen mehrfach hintereinander.

Der *partielle Zustandstest* ist ein Test ausgewählter Zustandsübergänge, die stellvertretend für die anderen sind, z.B. die Auszahlung bei einem überzogenen und bei einem nicht überzogenen Girokonto als stellvertretend für alle Kontobewegungen. Der Test aller Zustandsübergänge umfasst sämtliche Ereignisse und Aktionen. Dabei spielt die Reihenfolge der Ausführung noch keine Rolle. Übergänge werden auch nicht wiederholt. Im Falle des Girokontos bedeutet dies den Test jeder Bewegungsart, z. B. Einzahlung, Auszahlung, Überweisung, usw., für jeden Kontozustand – überzogen, nicht überzogen, gesperrt usw.

Der *Test aller Zustandsübergangskombinationen* impliziert die Wiederholung aller Zustandsübergänge in jeder beliebigen Reihenfolge, z.B. erst Einzahlung, Auszahlung und Überweisung, dann Überweisung, Auszahlung und Einzahlung usw. für jeden möglichen Objektzustand – überzogen, nicht überzogen und gesperrt.

Der Test aller wiederholbaren Kombinationen der Zustandsübergänge ist eine weitere Steigerung. Hier werden alle Zustandsübergänge in jeder beliebigen Folge mindestens zwei Mal hintereinander ausgeführt. So müssen alle Bewegungen, die zu einer Sperrung des Girokontos führen, doppelt ausgeführt werden, um zu sehen, ob der erste Durchgang Folgen für den zweiten Durchgang hat bzw. ob er zum gleichen Ergebnis führt. Hiermit werden fehlende Beziehungen zwischen Ereignissen, Aktionen und Zuständen aufgespürt.

Die höchste Stufe des Zustandstests testet alle *Zustandsübergänge in jeder beliebigen Reihenfolge mit allen möglichen Zuständen mehrfach hintereinander*. Es ist klar, dass ein derartiger Test sehr viele Testfälle benötigt und daher sehr aufwändig wird. Mit nur einem Konto mit drei Zuständen und drei Bewegungsarten kommen 9 Übergangspfade mit 81 Testfälle zustande. Deshalb ist diese Art des Tests nur möglich, wenn sie automatisiert ist.

6.6 Beispiel einer Build-In Testtechnik

Ein Ansatz zur Realisierung der Build-In Testtechnik ist das CPPTEST-Tool von einem der Autoren. Danach wird eine instrumentierte Kopie der ursprünglichen Klasse generiert. In die Kopie wird eine Test-Operation als Member-Operation eingebaut. Von dieser Operation aus werden die anderen Methoden aufgerufen. Vorher wird der Zustand des zu testenden Objekts mittels Precondition-Zusicherungen in den gewünschten Vorzustand versetzt. Nach jeder Methodenausführung wird der Nachzustand des Objekts mittels Postconditions bestätigt. Die Argumente für die Operationen werden ebenfalls von der Test-Operation generiert und die Rückgabewerte validiert. Eine Zusicherungsverletzung führt zur Unterbrechung des Tests und zur Ausgabe einer Fehlermeldung.

Für die Behandlung der fremden Methodenaufrufe gibt es zwei Möglichkeiten: Entweder werden die geerbten Operationen aufgerufen und nur die externen Operationen durch Teststellvertreter-Operationen simuliert, oder sowohl die geerbten als auch die externen Methoden als Member-Operationen in die Zielklasse eingebaut. In jenen eingebauten Stub-Operationen werden erwartete Rückgabewerte zugewiesen, um den Datenfluss der Klasse unter Test zu befriedigen. CPPTEST ist ein gutes Beispiel für die Anwendung der „class flattening"-Technik (Abbildung 6.8).

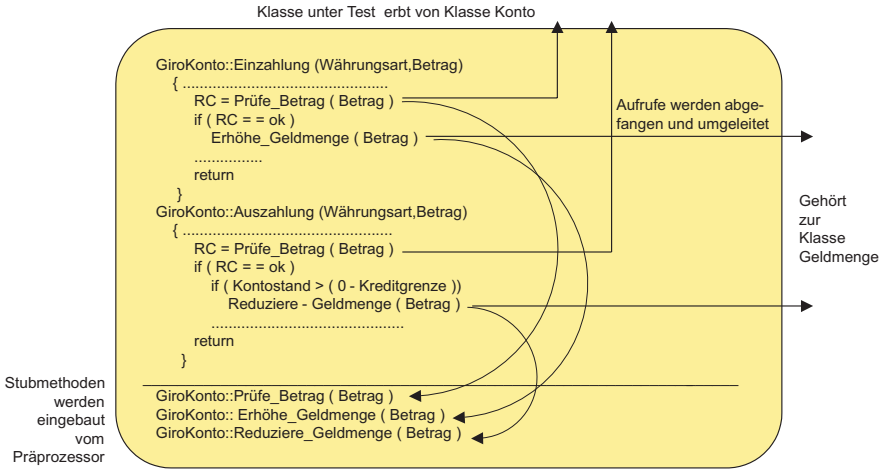


Abbildung 6.8 Class Flattening

Durch die eingebauten Test-Operationen im Zusammenhang mit der Zusicherungstechnik hat der Klassentest die volle Kontrolle über die Objektzustände und die Methodenausführung. Er kann seine Annahmen erproben und die Ergebnisse ohne die Einwirkung fremder Klassen kontrollieren. Die Klasse unter Test wird an und für sich ohne externe Abhängigkeiten getestet, die hier alle gekappt werden.

Die Instrumentierung des Codes sorgt dafür, dass die Ablaufüberdeckung gemessen wird. Es werden Durchlaufzähler in alle Methoden sowie in alle Ablaufzweige eingefügt, um die Anzahl der Durchläufe zu zählen. Die Überdeckungstabelle wird anschließend am Ende des Klassentests ausgegeben, damit der Tester die Wirkung des Tests verfolgen kann [Sne99].

6.7 Beispiel eines Klassentestrahmens

Stellvertretend für den Stand der Technologie beim Klassentestrahmen ist der Testrahmen JUnit von Kent Beck und Erich Gamma [Gam99]. Beck und Gamma gehen davon aus, dass der Test mit der Programmierung einhergeht. Der Entwickler kodiert ein Stück, z.B. eine Methode mit 12 Anweisungen, und testet sofort, um seine Annahmen zu bestätigen. Dazu braucht er eine sehr komfortable Umgebung, in der auch halbfertige Klassen testbar sind.

In JUnit gibt es eine übergeordnete Basisklasse namens `Test`. Die Testklassen für alle Klassen unter Test erben von dieser Klasse (Abbildung 6.9). Insofern entstehen alle neuen Klassen in einem Testrahmen, sodass ihre Methoden alle jederzeit testbar sind. Die Wurzelmethode `testsimple()` erzeugt Testobjekte, stößt ihre Operationen an und verifiziert ihre Resultate. Die Ergebnisse werden durch die geerbte Methode `assert()` bestätigt. Wenn sie nicht wahr sind, wird eine Fehlermeldung ausgelöst. Es wird also ein Testfall nach dem anderen für jede neue Methode entwickelt, gestartet und geprüft. Ist das Ergebnis gut, bekommt der Tester ein grünes Signal, ist es schlecht, ein rotes.

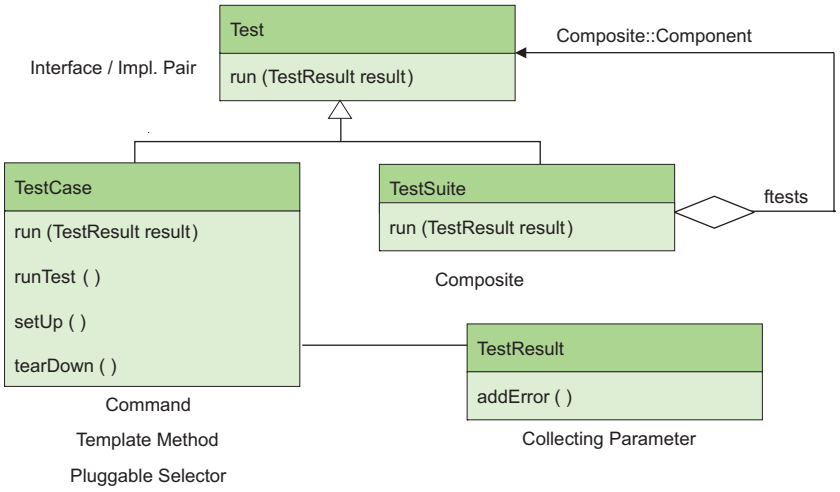


Abbildung 6.9 JUnit Testrahmen

Alle Testfälle werden in der Testtreiberklasse nachgehalten, sodass der Tester jede Methode jederzeit wieder testen kann. Mit JUnit erreicht man dies, indem man die Fixture-Objekte in Instanzvariablen der Klasse `TestCase` speichert und diese initialisiert, indem man die `setup()`-Methode überschreibt. `setup()` konstruiert den jeweiligen Testrahmen. Das Pendant zu `setup()` ist die `teardown()`-Methode, die man überschreiben kann, um den Testrahmen am Ende eines Testlaufs wieder aufzuräumen. Jeder Test läuft in seinem eigenen Rahmen ab und JUnit aktiviert die Operationen `setup()` und `teardown()`.

JUnit unterstützt zwei Testtreiberarten: statische und dynamische. Die erste Art ist statisch. Hier wird die von TestCase geerbte Methode `test()` überschrieben und der gewünschte Testfall ausgelöst. Jeder Testfall bekommt eine eigene Bezeichnung, um ihn im Fehlerfall identifizieren zu können. Eine Template-Methode in der Oberklasse stellt sicher, dass der richtige Testfall in der richtigen Folge ausgeführt wird.

Die dynamische Art, einen Testfall zu erzeugen, ist die Benutzung von Reflektion zur Implementierung der Operation `runTest()`. Dabei wird davon ausgegangen, dass der Name des Testfalls mit dem Namen der aufzurufenden Testfallmethode übereinstimmt. Die Test-Operation wird dynamisch ermittelt und aufgerufen. Die dynamische Art erlaubt kompakten Code, kann aber nicht bereits zur Compilierzeit geprüft werden. Ein Fehler in der Benennung bzw. Typisierung der Testfälle wird daher erst zur Laufzeit auffallen.

Für die Ausführung mehrerer Testfälle hintereinander sieht JUnit eine Methode namens `suite()` vor. `suite()` entspricht einer auf das Ablaufen von Tests spezialisierten Main-Methode. In der Suite werden alle Testfälle, die zu einem Test zusammengehören, zu einem `TestSuite`-Objekt hinzugefügt. Eine TestSuite kann somit eine Reihe Testfälle ablaufen lassen. Sowohl `TestSuite` als auch `TestCase` realisieren eine Schnittstelle namens `Test`, welche die Methoden für die Testausführung festlegt. Dies ermöglicht die Generierung von Testläufen durch die Zusammenstellung geeigneter Instanzen von `TestCase` und `TestSuite`.

Für die eigentliche Testausführung bietet JUnit eine graphische Benutzungsoberfläche an. Der Tester gibt den Namen der gewünschten Testklasse an und drückt den Startknopf. Während der Ausführung des Tests zeigt JUnit den Fortschritt in einem Balkenfeld an. Der Balken ist grün, so lange alles stimmt, färbt sich jedoch rot, sobald ein Test fehlschlägt. Demnach muss der Tester das falsche Ergebnis im Testprotokollfenster suchen.

Für diejenigen, die nicht im Dialogmodus testen wollen, bietet JUnit auch eine Batch-Oberfläche an. Damit kann man mehrere TestSuites auf einmal anstoßen. Diese Art Batchtest ist für den Regressionstest gedacht. Auch wenn Fehler auftreten, fängt JUnit sie hier ab und läuft weiter mit den restlichen Testfällen.

Hier, wie in anderen vergleichbaren Testansätzen, ist es wichtig, dass der Tester die Testfälle fortschreibt. Sobald er eine Methode geändert hat, passt er auch die ent-

sprechenden Testfälle an und wiederholt den Test. Falls er eine neue Methode einfügt, kreiert er parallel dazu neue Testfälle. D.h., die Fortschreibung des Testrahmens ist mit der Fortschreibung der Zielklassen fest integriert.
JUnit ist ein klassisches Beispiel für die Anwendung der Objekttechnologie einschließlich Frameworks, Templates, Vererbung und Polymorphie, um Java-Objekte zu testen. D.h. man benutzt die Eigenschaften der Objektorientierten Programmierung, um objektorientierte Programme zu testen. In dieser Hinsicht kann JUnit als Muster für den Klassentest gelten.

6.8 Klassentestarten

Verteilte objektorientierte Systeme haben in der Regel eine Drei-Schichten-Architektur mit folgenden Schichten (Abbildung 6.10):

- der Oberflächenschicht,
- der Anwendungsschicht und
- der Zugriffsschicht.

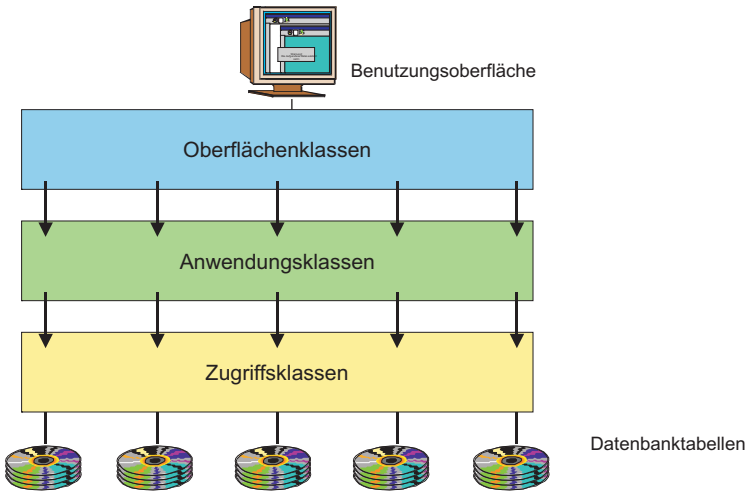


Abbildung 6.10 Klassenschichten

Die Oberflächenschicht enthält jene Klassen, welche die Benutzungsoberfläche realisieren. Sie sind ereignisgesteuert und eng mit der Oberflächentechnologie und der verwendeten Plattform verbunden. Sie verwenden auch viele Oberflächen Operationen, die sie von den Standardklassen der jeweiligen Programmierumgebung erben.

In der Zugriffsschicht sind jene Klassen, die auf die Datenbank zugreifen, Daten selektieren, Objekte bilden, alte Daten aktualisieren und neue Daten einfügen. Sie sind transaktionsgesteuert und eng mit der Datenbank verbunden. Zum größten Teil bestehen sie aus eingebetteten SQL-Operationen bzw. aus ODBC- oder JDBC-Operationen.
In der Anwendungsschicht sind jene Klassen, welche die eigentliche Applikationslogik beinhalten. Sie erhalten ihre Parameter von den übergeordneten Oberflächenklassen und die Objekte von den untergeordneten Zugriffsklassen. Mit den ihnen zugewiesenen Parametern verändern sie bestehende Objekte, schaffen neue Objekte und lassen alte Objekte wieder abspeichern. Es versteht sich, dass jede Klassenart andere Anforderungen an den Klassentest stellt und anders zu testen ist.

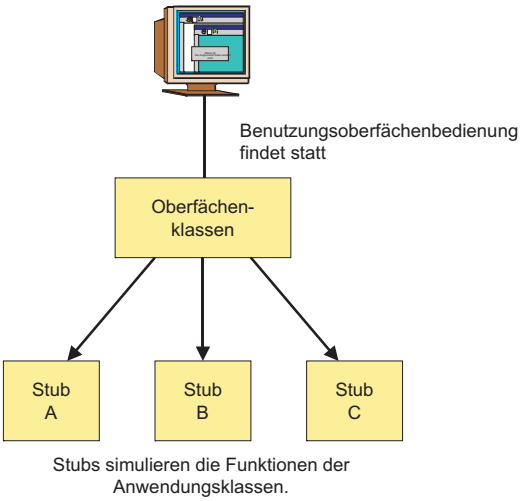


Abbildung 6.11 Test der Oberflächenklassen

6.8.1 Test der Oberflächenklassen

Die Oberflächenklassen lassen sich am besten mit der Oberfläche selbst testen. Die Oberflächen-Operationen werden ausgeführt und liefern Kontrollwerte, die per ASSERT gesichert werden. Lediglich die Aufrufe der Anwendungsoperationen sind abzufangen und zu simulieren. D.h., in den Oberflächenklassen müssen die Anwendungsaufrufe erkannt und als Stub-Operationen in die Klasse eingefügt werden (Abbildung 6.11). In den Stub-Operationen müssen die möglichen Rückgabewerte zugewiesen und an die aufrufenden Operationen zurückgereicht werden. Dadurch benutzen die Oberflächenklassen die normale Oberfläche als Testtreiber. Ein extra Testtreiber oder eingebaute Test-Operationen sind nicht nötig. Lediglich die Stubs

sind erforderlich, um die Aufrufe der Anwendungsoperationen zu befriedigen, ohne die Anwendungsklassen selbst in den Test einzubinden.

6.8.2 Test der Zugriffsklassen

Die Zugriffsklassen lassen sich am besten mit den Datenbanken selbst testen, d.h. es sollten Testdatenbanken bereitgestellt werden. Die Zugriffsfunktionen – `SELECT`, `UPDATE`, `INSERT` und `DELETE` – werden so ausgeführt, als ob die Klasse in der Produktionsumgebung wäre. Andererseits müssen hier die Aufträge von den übergeordneten Applikationsklassen simuliert werden. Entweder gibt es dafür einen externen Testtreiber oder eine interne Test-Operation, von wo aus die anderen Zugriffsmethoden angestoßen werden. Dort werden die Eingangsparameter erzeugt.

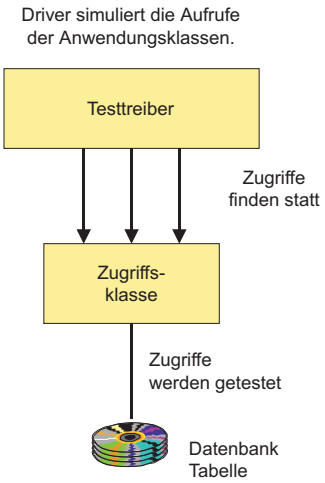


Abbildung 6.12 Test der Zugriffsklassen

Das Hauptproblem hier ist die Konsistenz der Parameter mit den Datenbankinhalten (Abbildung 6.12). Die Suchbegriffe, die vom Treiber zugewiesen werden, müssen mit dem Schlüssel der Datenbank übereinstimmen, sonst passiert gar nichts. Da oft Surrogate, also künstlich erzeugte Daten, verwendet werden, ist der Suchbegriff nicht selten eine Zufallsnummer. Deshalb muss der Tester unbedingt Zugriff zu den Datenbanktabellen haben, um Schlüssel und Indexfelder in die Parameterlisten zu übertragen. Nur so ist die Konsistenz zwischen Argumenten und Objektinhalt zu gewährleisten.

Ein typischer Testrahmen für Zugriffsklassen wird die Testdatentabelle in einem Fenster und die Parameter in einem anderen anzeigen. Dann wird der Tester Werte aus der Datenbank in die Parameterliste kopieren und auf diese Weise eine gezielte,

semantisch korrekte Datenbanktransaktion aufbauen, die anschließend gestartet und getestet wird. Hier ist es unbedingt erforderlich, die Datenbankzugriffe mit zu testen, denn sie bilden den Kern solcher Klassen und lösen auch die Ausnahmebedingungen aus.

6.8.3 Test der Anwendungsklassen

Die Anwendungsklassen liegen zwischen den übergeordneten Oberflächenklassen, aus denen sie die Parameterwerte beziehen und den untergeordneten Zugriffsklassen, aus denen die Objekte stammen. Insofern brauchen sie sowohl einen Treiber, um die Oberflächenklasse zu simulieren, als auch Stubs, um die Zugriffsklassen zu simulieren (Abbildung 6.13). Für sie eignet sich die oben geschilderte Build-In Testtechnik am besten. Demnach wird eine Test-Operation in sie hinein gepflanzt, um die Aufrufe der Oberfläche zu generieren. Jede Anwendungs-Operation wird für jede Parametervariation angestoßen und die Rückgabewerte über Assert-Anweisungen validiert.

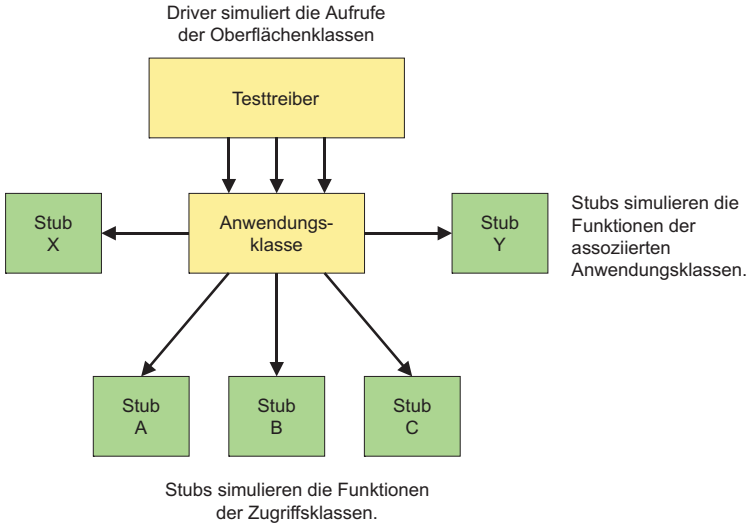


Abbildung 6.13 Test der Anwendungsklassen

Für die Aufrufe der Zugriffsoperationen werden Stubs als Member-Operationen der Klasse eingefügt. Diese Stubs müssen Speicher für die Objekte und die Attributwerte zuweisen und dem Aufrufer eine Objektreferenz bereitstellen. Da dies alles nicht so einfach ist, wird es unvermeidlich sein, die Stub-Operationen aufgrund einer Analyse der Objektstruktur zu generieren. Dafür ist ein geeignetes Testwerkzeug unerlässlich.

6.9 Test der Tagesklasse im Kalendersystem

Im verteilten Kalender gibt es eine Klasse `Task`, die von der Klasse `Tag` die Tagesdaten, von der Klasse `Woche` die Wochendaten und von der Klasse `Kalender` die Mitarbeiterdaten bearbeitet. `Task` hat einen Konstruktor, der eine Aktivität als Objekt anlegt, und einen Destruktor, der die Aktivität wieder freigibt. Zusätzlich hat die Klasse `Task` drei öffentlich sichtbare Methoden

`get_Aktivitätenliste()`,
`add_Aktivität()` und
`buche_Stunden()`.

Die Operation `get_Aktivitätenliste()` bekommt als Eingangswert die Anzahl Aktivitäten und gibt das Task-Objekt zurück. `add_Aktivität()` erhält als Eingangsparameter die Start-Zeit, Ende-Zeit und Task-Beschreibung und gibt einen Return-Code zurück. `buche_Stunden()` rechnet die Arbeitsstunden pro Aktivität aus und gibt sie weiter an den Projektclient.
Die Konstruktor- und Destruktor-Operationen haben keine interne Logik. Sie müssen daher nur einmal pro Objekttag aufgerufen werden.
Die Operation `get_Aktivitätenliste()` verweist mit einem Zeiger auf das Objekt Aktivitätenliste und gibt die Anzahl der Aktivitäten zurück. Sie müsste mindestens drei Mal pro Objekttag aufgerufen werden:

- einmal für eine leere Aktivitätenliste,
- einmal für eine volle Aktivitätenliste und
- einmal für eine halbvolle Aktivitätenliste.

Der Test dieser Operation geht also von der Grenzwertanalyse aus.
Die Operation `add_Aktivität()` hat eine Bedingung. Sie prüft nämlich, ob die Aktivitätentabelle voll ist. Wenn ja, gibt sie den Return-Code 0 zurück. Wenn nicht, fügt sie die neue Aktivität in die Tabelle ein und gibt den Return-Code 1 zurück. Für den Test dieser Operation werden nur zwei Testfälle benötigt:

- eine leere Aktivitätentabelle und
- eine volle Aktivitätentabelle.

Die Operation `cancel_Aktivität()` hat zwei `while`-Schleifen. In der ersten Schleife sucht sie die zu löschende Aktivität. In der zweiten Schleife rückt sie die darauf folgenden Aktivitäten um eine Position nach vorne und überschreibt damit die zu löschende Aktivität. Für den Test dieser Operationen gibt es zwei Varianten:

- die zu löschende Aktivität wird gefunden und entfernt, oder
- die zu löschende Aktivität wird nicht gefunden.

Im ersten Fall ist der Return-Code 1, im zweiten Null.

Nach dem Build-In Testansatz wird eine Test-Operation `test()` eingefügt, in welcher der Konstruktor aufgerufen wird, um eine leere Aktivitätentabelle anzulegen. Danach könnte die folgende Testfallsequenz ausgeführt werden:

```
get_Aktivitätenliste (die Liste ist leer)
can_Aktivität (die Liste ist leer)
add_Aktivität (1:12)
add_Aktivität (die Liste ist voll)
can_Aktivität (die Liste ist voll)
```

Zum Schluss soll der Destruktor aufgerufen werden.

Die Operation `can_Aktivität()` ruft drei Operationen der geerbten Klasse `Task` auf:

- `get_Start_Zeit()`,
- `get_End_Zeit()` und
- `get_Aktivität()`.

Um die Klasse `Task` nicht mittesten zu müssen, werden diese drei Operationen als Stub-Operation in die Klasse `Tag` eingefügt. Dort werden den Rückgabewerten mit ASSERT Anweisungen zugewiesen. Dadurch ist es möglich, die Klasse `Tag` allein für sich zu testen, denn die Datentypen der geerbten Klasse werden per `include` aufgenommen. Um die Korrektheit der Logik zu bestätigen, schreibt der Tester am Anfang der Operation ein ASSERT zur Prüfung der Eingangsparameter und am Ende der Operation ein ASSERT zur Bestätigung der Ergebnisse, z.B. in der Operation `add_Aktivität()`:

Am Anfang:

```
ASSERT (sz > 0 && sz < 2400 && sz < ez);
ASSERT (ez > 0 && ez < 2400 && ez > sz);
```

und am Ende:

```
ASSERT (nr_task <= 12);
```

Der Test der Klasse `Tag` weist nach, dass alle Operationen der Klasse funktionieren, dass alle Zweige erreichbar sind und dass die Istergebnisse mit den Sollergebnissen übereinstimmen (Abbildung 6.14). Mehr kann man von einem Klassentest nicht erwarten. Die Korrektheit der Objektinteraktionen zwischen Instanzen der Klassen `Tag` und `Woche` sowie `Tag` und `Task` ist Sache des Integrationstests.

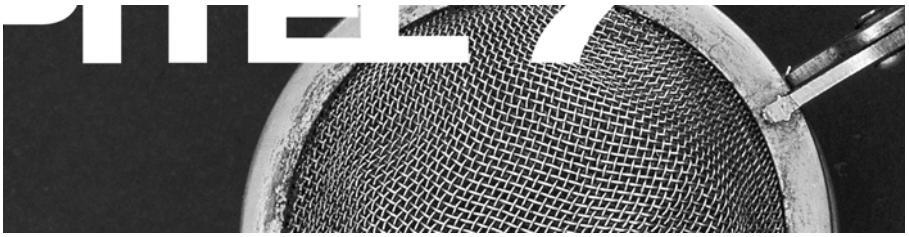
```
//*************************************************************/
// Test Driver to invoke member functions                     */
 main()
 {
   int    TestCaseMax = 20;
   int    TestCase    =  0;
   int    ResultNr    =  0;
   int    DefectNr    =  0;
   float  DefectRate  =  0.0;
   cout << "\nEingabe Anzahl der Testfaelle: ";
   cin >> TestCaseMax;
   for (TestCase = 0; TestCase < TestCaseMax; TestCase++)
    {
//tag: METHODS;
//+-------------------------------------------------------------------------------
//Tag::Tag (int xtag)
  cout << "\nEigene Methode: Tag::Tag (int xtag)";
  int Tag_xtag;
  cout << "\nEingabe Wert int xtag: ";
  cin  >> Tag_xtag;
  Tag oTag(Tag_xtag);
//+-------------------------------------------------------------------------------
//Task ** Tag::get_Aktivitaetenliste (int *pj)
  cout << "\nEigene Methode: Task ** get_Aktivitaetenliste (int *pj)\n";
  Task **get_Aktivitaetenliste_Task;
  int  *get_Aktivitaetenliste_pj = new int;
  cout << "\nEingabe Wert int *pj: ";
  cin  >> *get_Aktivitaetenliste_pj;
  get_Aktivitaetenliste_Task = oTag.get_Aktivitaetenliste(get_Aktivitaetenliste_pj);
//+-------------------------------------------------------------------------------
//int Tag::add_Aktivitaet (float sz, float ez, char *task)
  cout << "\nEigene Methode: int add_Aktivitaet (float sz, float ez, char *task)\n";
  int add_Aktivitaet_int;
  float add_Aktivitaet_sz;
  float add_Aktivitaet_ez;
  char *add_Aktivitaet_task;
  int  add_Aktivitaet_itask;
  cout << "\nEingabe Wert float sz: ";
  cin  >> add_Aktivitaet_sz;
  cout << "\nEingabe Wert float ez: ";
  cin  >> add_Aktivitaet_ez;
  cout << "\nEingabe der Groesse des Wertes char *task: ";
  cin  >> add_Aktivitaet_itask;
  cout << "\nEingabe Wert char *task: ";
  cin >> add_Aktivitaet_task;
  add_Aktivitaet_int = oTag.add_Aktivitaet (add_Aktivitaet_sz, add_Aktivitaet_ez, add_Aktivitaet_task);
//+-------------------------------------------------------------------------------
//Tag::~Tag ()
  Tag.~Tag();
//+-------------------------------------------------------------------------------
//END METHODS;
   } // End of Method Test
  cout << "\nEnde der Testfaelle\n";
  int quitt;
  cin  >> quitt;
 } // End of Class Test Loop
```

```
// Test Stubs to simulate foreign function calls
// tag: STUBS;
//+-------------------------------------------------------------------------------
//Task
//tasks[max_task] = new Task (sz, ez, task);*/
Task::Task (float xsz, float xez, char *xtask)
{
 cout << "\nFremde Methode: float Task::Task(float xsz, float xez, char *xtask)\n";
}
//+-------------------------------------------------------------------------------
//Task
//tasks[max_task] = new Task (sz, ez, task);*/
Task::~Task ()
{
 cout << "\nFremde Methode: float Task::~Task()\n";
}
//+-------------------------------------------------------------------------------
//get_Start_Zeit
//sztask =       tasks[i]->get_Start_Zeit ();
float Task::get_Start_Zeit ()
{
 float v;
 cout << "\nFremde Methode: float Task::get_Start_Zeit ()\n";
 cout << "\nEingabe Rueckgabewert: ";
 cin >> v;
 return v;
}
//+-------------------------------------------------------------------------------
//get_End_Zeit
//eztask =       tasks[i]->get_End_Zeit   ();
float Task::get_End_Zeit ()
{
 float v;
 cout << "\nFremde Methode: float Task::get_End_Zeit ()\n";
 cout << "\nEingabe Rueckgabewert: ";
 cin >> v;
 return v;
}
//+-------------------------------------------------------------------------------
//get_Aktivitaet
//ttask  =       tasks[i]->get_Aktivitaet ();
char * Task::get_Aktivitaet ()
{
 char *v;
 int  iv;
 cout << "\nFremde Methode: char * Task::get_Aktivitaet ()\n";
 cout << "\nEingabe der Groesse des Rueckgabewertes: ";
 cin >> iv;
 try {
     v = new char[iv];
     }
 catch (xalloc)
    {
     cout << "Fehler Speicherreservierung, Abbruch!";
     exit(-1);
    }
 cout << "\nEingabe Rueckgabewert: ";
 cin >> v;
 return v;
}
```

Abbildung 6.14 Tagesklassentest

7

Integrationstest



Stufen der Integration
Integrationsteststrategien
Integrationstestansätze
Klassenintegrationstest
Komponentenintegrationstest
Integrationstest verteilter Objekte
Integrationstest des verteilten Kalenders

Inhaltsübersicht Kapitel 7

7 Integrationstest

Wie Barry McGibbon in seinem Buch “Managing your Move to Object Technology” so treffend bemerkt, sind Einzelklassen zwar interessant, aber für sich allein nutzlos. Das erste nützliche Stück Software ist

“…a cluster of classes that provides distinct behavior or functionality.” [McG95].

Ein solcher Cluster soll möglichst in einem Adressraum sein und eine einzige Schnittstelle nach außen haben. Damit sind wir in der Nähe des Begriffs *Komponente* gelandet. Eine Komponente ist eine Menge gegenseitig abhängiger Klassen, die zu einer binären, ausführbaren Codeeinheit zusammengefasst sind. In der Microsoft-Welt können sie DLL´s sein, in der Unix-Welt sind sie Libraries. In der Regel handelt es sich um kompilierten Objektcode, aber in Sprachen wie Smalltalk-80 und Java kann es sich um Byte-Code handeln. Komponenten sind die eigentlichen Bausteine zur Ausführungszeit.
Boris Beizer beschreibt den Integrationstest als den Test der Schnittstellen zwischen bereits getesteten Modulen bzw. Klassen. Er beginnt in dem Augenblick, wenn zwei Module bzw. zwei Klassen zusammen getestet werden, setzt sich fort, wenn eine Klasse nach der anderen hinzugefügt wird, und endet, wenn alle Klassen in den Test eingebunden sind. Zu vermeiden ist es, alle Klassen auf einmal zusammenzubinden und als Ganzes zu testen. Im Mittelpunkt des Integrationstests steht die Interaktion zwischen getrennt entwickelten Bausteinen. Die Fehler, um die es hier geht, sind in erster Linie Inkompatibilitäten in den Schnittstellen zwischen den Bausteinen und zwischen dem System und seiner Umgebung. Dazu gehören unverträgliche Datentypen, unterschiedlich gereihte Parameterlisten, Speicherverlust, falsche Adressvermittlung, fehlende Operationen, fehlende Ausnahmebehandlung und unkorrekte Datenbankzugriffe. Laut Myers ist der Integrationstest eng mit einem inkrementellen *Build*-Prozess verbunden [Mye76].

7.1 Stufen der Integration

Bei größeren objektorientierten Systemen gibt es in der Regel drei Stufen der Integration [Thu93]:

- Klassenintegration,
- Komponentenintegration und
- Schichtenintegration.

7.1.1 Klassenintegration

Auf der ersten Stufe werden alle Klassen, die zu einer Komponente gehören, eine nach der anderen hinzugebunden und im Hinblick auf ihr Zusammenwirken getestet. Diese Stufe der Integration gehört eher zum Zuständigkeitsbereich der Entwickler. Nur testet hier nicht ein einzelner Entwickler eine einzelne Klasse, sondern es testen mehrere Entwickler zusammen einen von ihnen als Team entwickelten Klassencluster. Bei kleineren Applikationen, wie z.B. einem isolierten Erfassungsprogramm auf einem Arbeitsplatz, ist damit der Integrationstest beendet. Bei komplexeren Systemen, zu denen die meisten Informationssysteme zählen, ist dies nur die erste Integrationsstufe.

7.1.2 Komponentenintegration

Auf der zweiten Stufe werden alle Komponenten, die zu einer Anwendungsschicht gehören, eine nach der anderen in den Test einbezogen. Im Falle verteilter Systeme wird es sich hier um verschiedene Adressräume bzw. verschiedene Netzknoten handeln. Es sind also nicht nur mehrere Entwickler, sondern auch mehrere Entwicklerteams betroffen. Deshalb gehört diese Stufe der Integration eher zum Zuständigkeitsbereich des Testteams. Denn nur das Testteam kann die Verantwortung für das Ganze tragen, und das Ganze ist mehr als die Summe aller einzelnen Komponenten. Dazu gehören sämtliche Beziehungen zwischen den Komponenten sowie sämtliche Effekte, die durch das Zusammenwirken der Komponenten entstehen. Die Integration von Komponenten, insbesondere verteilter Komponenten, ist um einiges schwieriger als die Integration von Klassen einer einzelnen Komponente.

7.1.3 Schichtenintegration

Auf der dritten Stufe werden alle Schichten eines Anwendungssystems miteinander verbunden und getestet. Bei Systemen mit einer zweischichtigen Architektur gibt es ein „Frontend“ und ein „Backend“, bzw.

- die Client-Schicht und
- die Server-Schicht

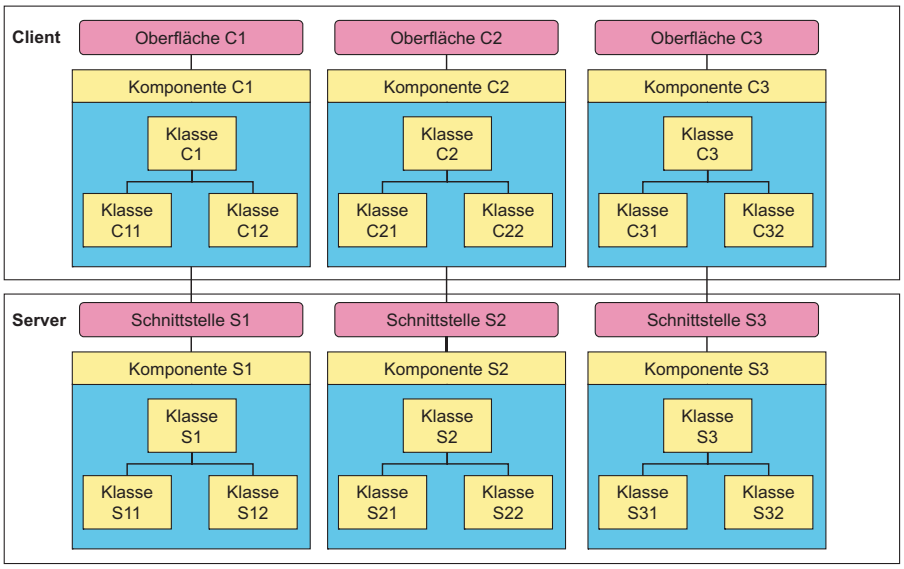


Abbildung 7.1 Integration der Komponenten

Bei Systemen mit einer dreischichtigen Architektur gibt es

- die Präsentationsschicht,
- die Verarbeitungsschicht und
- die Zugriffsschicht.

Die Verarbeitungsschicht könnte auf dem Clientrechner, auf dem Serverrechner oder auf einem Zwischenrechner sein. Hier kommt es vor allem darauf an, die Beziehungen zwischen den Schichten, d.h. zwischen Client und Server, bzw. Client, Server und Vermittlungsrechner zu testen. Da die Nachrichtenvermittlung zwischen verteilten Komponenten hauptsächlich eine Aufgabe der Middleware ist, zielt der Schichtentest auf die Bestätigung der Interaktion zwischen der Anwendungssoftware und der jeweiligen Middleware.

7.2 Integrationsteststrategien

Laut Siegel ist die beste Strategie zum Integrationstest, wenn immer ein Baustein, z.B. einer Klasse, nach der anderen nach jedem Test hinzugefügt wird [Sie94]. Diese Integration des Testobjekts kann sich von oben nach unten – Top-Down – oder von unten nach oben – Bottom-Up – vollziehen. Bei der Integration der Klassen einer Generalisierungshierarchie wird die Integrationsrichtung eher Top-Down sein. Man beginnt mit den Basisklassen und fügt die abgeleiteten Klassen eine nach der anderen hinzu. Die Klassenhierarchie wird also von der Spitze aus integriert.

Bei der Komponentenintegration ist die Integrationsrichtung eher Bottom-Up. Man beginnt mit den Backend-Komponenten der Datenzugriffsschicht und fügt zunächst die Anwendungskomponente und später die Frontend- bzw. die Oberflächenkomponente hinzu. Ausschlaggebend für den Integrationstest ist hier, dass immer nur ein zusätzlicher Baustein mit seinen Schnittstellen auf einmal getestet wird. Bei der Schichtenintegration gibt es neben den Top-Down- und Bottom-Up-Strategien auch noch die Outside-In- und Inside-Out-Strategien, wobei diese beiden Strategien mindestens eine dreischichtige Architektur voraussetzen (Abbildung 7.2).

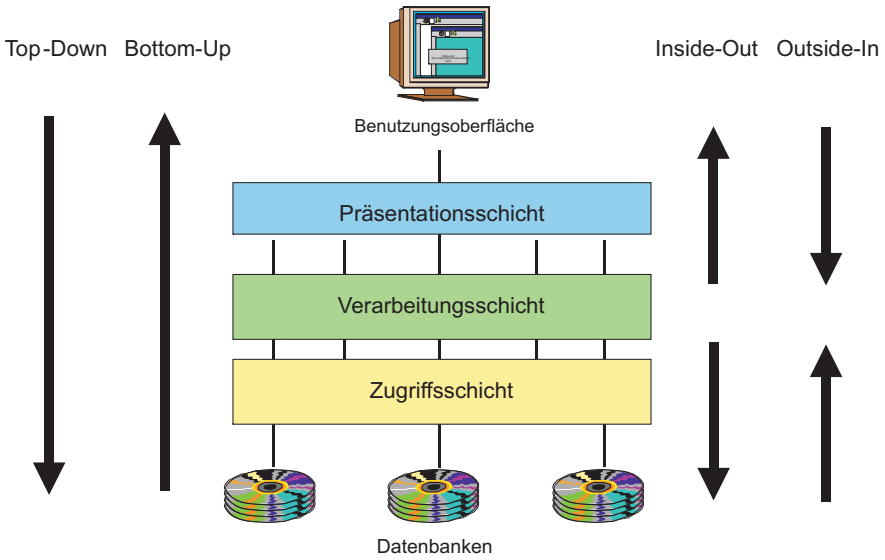


Abbildung 7.2 Integrationsteststrategien

Die schlimmste Art des Integrationstests ist laut Beizer, wenn alles auf einmal zusammengebunden wird – der Big-Bang-Ansatz. Big-Bang ist in Wirklichkeit kein Integrationstest mehr, sondern ein Systemtest. In der Praxis kommt dieser Ansatz leider allzu häufig vor, weil die schrittweise Integration von Klassen bzw. Komponenten äußerst mühselig und zeitraubend ist. Die Verbindung zu den fehlenden benutzten Klassen bzw. Komponenten müssen durch Stubs gekappt bzw. abgefangen werden, während die fehlenden benutzenden Klassen durch Testtreiber simuliert werden. Um den Integrationstest zu erleichtern, müssten die Entwickler die Integrationsfähigkeit schon beim Systementwurf berücksichtigen und die Stubs als Schnittstellen oder Proxyklassen vorsehen [Bei99].

Siegel unterscheidet zwischen vertikaler und horizontaler Integration (Abbildung 7.3). Vertikale Integration ist ein Test bezüglich der Generalisierungshierarchie. Er soll nachweisen, dass alle geerbten Methoden auch für die Objekte der erbenden Klasse funktionieren, d.h. es gibt keine vererbte Methode, die nicht für jede Objekt-

ausprägung der untergeordneten Klasse zutrifft. Horizontale Integration ist ein Test der Assoziationen. Assoziationen zwischen Komponenten entstehen, wenn Methoden einer Klasse Methoden einer fremden Klasse in einer anderen Klassenhierarchie aufrufen. Hier soll der Nachweis erbracht werden, dass die Schnittstelle zwischen den assoziierten Methoden – die aufrufende und die aufgerufene – aus beiderlei Sichten stimmt und dass die aufgerufene Methode in allen Fällen ein korrektes Ergebnis zurückgibt. Integriert werden nicht nur die Klassen, sondern auch ihre Instanzen – die Objektausprägungen [Sie96].

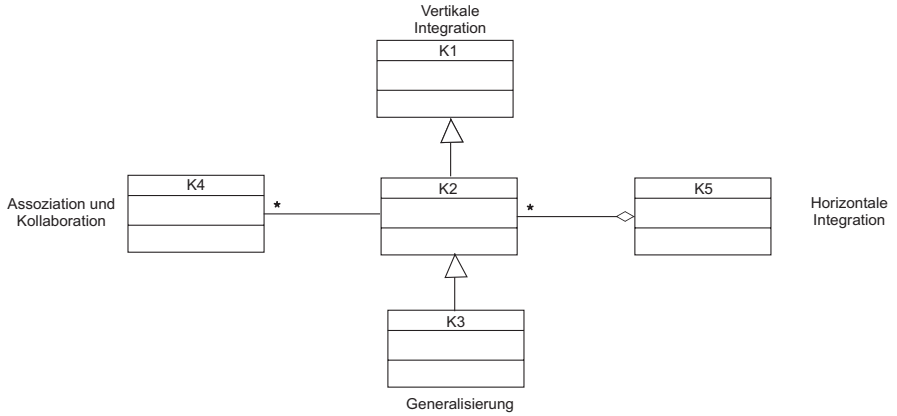


Abbildung 7.3 Horizontale und Vertikale Integration

7.2.1 Vertikale Integration

Bei der vertikalen Integration geht man davon aus, dass die übergeordneten Klassen bereits getestet sind. Jetzt wird geprüft, ob die Attribute und Operationen, die von ihnen übernommen werden, auch in jenem Zusammenhang noch gültig sind (Abbildung 7.4). Z.B. muss geprüft werden, ob die Eigenschaften eines Auftrags auch für jeden seiner Auftragsposten gelten oder ob die Bedingungen eines Mitarbeiters auch für die freien Mitarbeiter gelten. Hier wird also die Gültigkeit einer Klassenhierarchie bestätigt: ist das was als allgemeingültig deklariert wurde, in der Tat allgemeingültig?

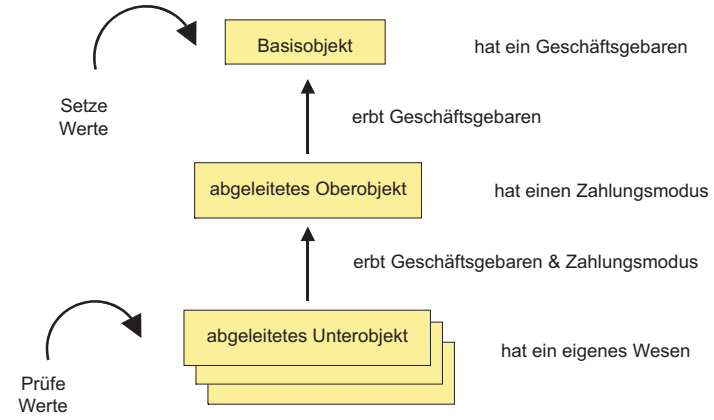


Abbildung 7.4 Vertikale Integration

7.2.2 Horizontale Integration

Bei der horizontalen Integration geht man davon aus, dass die assoziierten Klassen bereits (Klassen-)getestet sind. Jetzt geht man daran, ihre Kollaboration zu prüfen bzw. zu testen, ob sie wirklich kompatibel zueinander sind (Abbildung 7.5). Dazu wird die aufrufende Klasse bzw. Methode ausgeführt und jeder Aufruf zwecks der Kontrolle aufgezeichnet. Kontrolliert wird, ob der Aufrufer die erwarteten Eingangsparameter liefert und ob der Aufgerufene die richtigen Ergebnisse zurückgibt. Wenn ja, sind sie verträglich und die Ist-Interaktion entspricht der Soll-Interaktion.

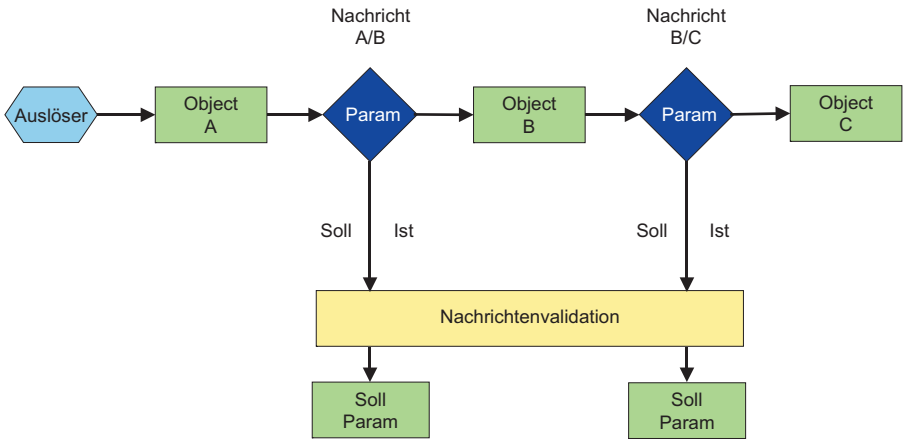


Abbildung 7.5 Horizontale Integration

7.3 Integrationstestansätze

Siegel gliedert den Integrationstest in die drei Teile ([Sie96], s. Abbildung 7.6):

- Architekturtest,
- Klasseninteraktionstest und
- Oberflächentest.

Im *Architekturtest* wird die Klassenhierarchie vertikal von oben nach unten – Top-Down – oder von unten nach oben – Bottom-Up – getestet. Als Basis für den Test können die Klassendiagramme dienen.
Im *Klassenintegrationstest* wird die Klassenassoziation horizontal gestestet, und zwar über die Schnittstellen zwischen gleichrangigen Klassen. Als Basis für diesen Test können die Sequenz- und Kollaborationsdiagramme dienen.
Im *Oberflächentest* wird die Interaktion zwischen dem System und dem Bediener getestet. Als Ausgangsbasis dienen die Anwendungsfalldiagramme und die Oberflächenbeschreibungen bzw. die Bedienungsanleitung.

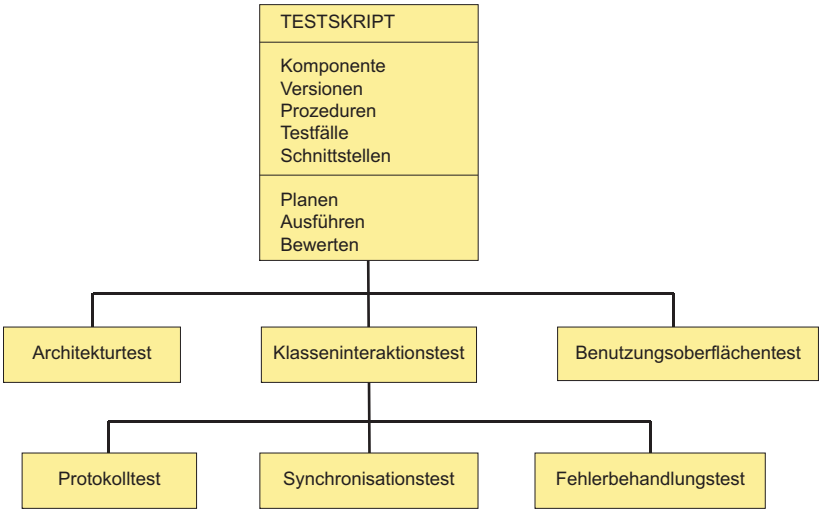


Abbildung 7.6 Integrationstesthierarchie

Robert Binder hat in der einschlägigen Literatur zum objektorientierten Testen nicht weniger als 14 verschiedene Integrationstestansätze ausfindig gemacht [Bin96]. Diese sind:

- der dreistufige Integrationsansatz von Jüttner,
- der Regressionsansatz von Rothermel und Harrold,
- der Use-Case basierte Integrationsansatz von Jacobson,

- der hierarchisch-inkrementelle Integrationsansatz von Thuy,
- der Client/Server-orientierte Integrationsansatz von Overbeck,
- der Propagationsmusteransatz von Lieberherr und Xiao,
- der Reverse-Engineering-Ansatz von Kung,
- der Modusautomaten-Ansatz von Binder selbst,
- der zunehmende Umfangsansatz von Siegel,
- der vierschrittige Integrationsansatz von Lee,
- der Assemblierungsansatz von Desfray,
- der Integrationstest nach Komposition von Jorgenson und Erickson,
- der Flutwellenansatz von McGregor und Korson und
- der Objektkommunikationsansatz von Jüttner.

Die Ansätze werden im Folgenden beschrieben.

7.3.1 Dreistufiger Integrationstest

Jüttner sieht drei unterschiedliche Stufen des Integrationstests:

- Klassenintegrationstest,
- Komponentenintegrationstest und
- Applikationsintegrationstest.

Die *Klassenintegration* bezieht sich auf den Test der Vererbungsbeziehungen, Call/Use-Beziehungen und Datenaggregationen innerhalb einer Klassenhierarchie. Im Mittelpunkt der *Komponentenintegration* steht die Methodeninteraktion und Objektkommunikation. Die *Applikationsintegration* zielt auf den Test der Komponentenschnittstellen. Jüttner weist darauf hin, dass es sehr schwierig ist, nur eine Stufe nach der anderen zu testen, weil man die fehlende Funktionalität durch zahlreiche Stubs ersetzen muss. Darum empfiehlt er eine Mischstrategie, nach der die Stufen parallel zueinander getestet werden [JKZ94].

7.3.2 Regressionstest

Mary Jean Harrold sieht im Klassenintegrationstest eigentlich ein Regressionstestverfahren. Ab dem Test der ersten zugrunde liegenden Basisklassen werden die gleichen Testläufe immer wiederholt und nur geringfügig ergänzt, nämlich um jene Testfälle, die neue Methoden in neuen Klassen tangieren. Der Testgegenstand selbst

– die Klassenhierarchie – wird Methode für Methode und Klasse für Klasse ausgebaut, wobei man die bereits getesteten Klassen als Testrahmen für den Test der neu hinzugebundenen Klassen verwendet [HMF92].

7.3.3 Anwendungsfallbasierter Integrationstest

Der anwendungsfallbezogene Integrationstest gehört zur Anwendungsfallspezifikation von Ivar Jacobson, der empfiehlt, einen Anwendungsfall nach dem anderen zu testen. Somit wird die Programmlogik gegen die Anforderungen getestet. Gleichzeitig wird die Interaktion der Objekte getestet und die Schnittstellen zwischen Komponenten bestätigt. Die Steuerung des Integrationstests über die Anwendungsfälle wird auch von Firesmith [Fir96], Graham [GDT93] und Winter [Win98] [Win99] empfohlen. Es ist jedoch zu bemerken, dass dieser Ansatz eher für die Komponentenintegration und weniger für die Klassenintegration geeignet ist. Er setzt auch voraus, dass das Nutzungsprofil vollständig spezifiziert ist.

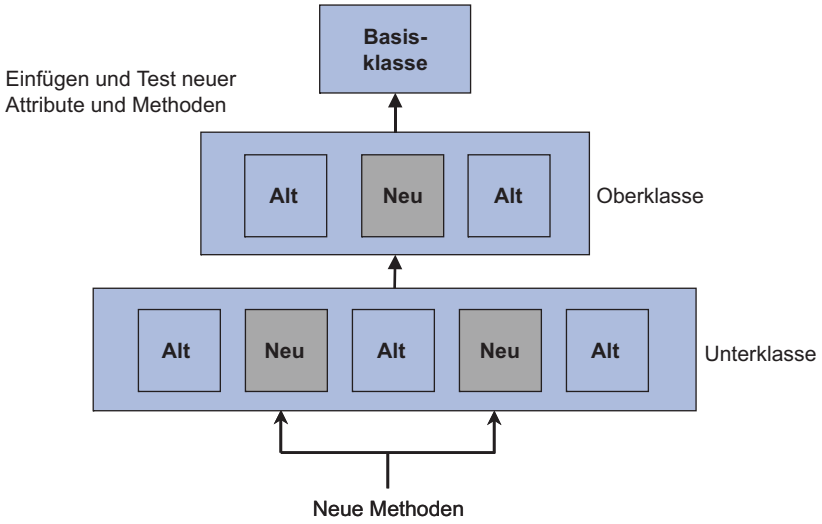


Abbildung 7.7 Inkrementeller Integrationstest

7.3.4 Hierarchisch-inkrementeller Integrationstest

In Anlehnung an den stufenweisen Integrationsansatz von Jüttner plädiert Thuy für einen Architekturstufentest (Abbildung 7.7). Er geht davon aus, dass das Anwendungssystem sich aus einer Hierarchie von Klassenschichten zusammensetzt. Nach seiner Methode werden alle Objekte auf der gleichen Hierarchiestufe zunächst integriert und die darunterliegende Objektschicht simuliert. Wenn in ihrer Interaktion

keine Fehler mehr auftauchen, wird die nächste Schicht herangezogen. Am besten ist es, wenn man mit der untersten Schicht beginnt. Thuy behauptet, es sei schwierig und aufwändig, eine Komponente unabhängig von den Komponenten, mit denen sie kollaboriert, zu testen. Dies bedeute den Ersatz der fehlenden Komponenten durch *Stubs*, und das wiederum könne eine weitere Fehlerquelle sein. Deshalb sei es vorzuziehen, neue Komponenten im Rahmen der bereits getesteten Komponenten zu testen [Thu93].

7.3.5 Client/Server-orientierter Integrationstest

Der Client/Server-orientierte Integrationstest stammt aus der Dissertation von Jan Overbeck. Der Autor schlägt vor, die Objekte in Client- und Server-Objekte zu teilen. Ausgehend von den untersten Serverobjekten werden die Aufträge an diese Objekte durch Testtreiber simuliert, bis alle ihre Operationen bestätigt sind. Dann werden die Testtreiber gegen die echten Client-Objekte ausgetauscht. Dieser Test wird so lange für eine Objektschicht nach der anderen wiederholt, bis man an die Benutzungsoberfläche kommt, d.h. an die endgültigen Ausgangsobjekte.

Erst wenn jede Klasse zunächst für sich und dann für jede Client/Server-Beziehung zwischen mehrfachen Clientobjekten eines Serverobjekts getestet wird, könnten wir mit Gewissheit von einem korrekten System sprechen. Dieser Ansatz setzt auf das Prinzip des *„Design by Contract“* von Bertrand Meyer (vgl. Abschnitt 6.5.3) und bestätigt die Erfüllung der mit Zusicherungen vereinbarten Verträge zwischen den einzelnen Klassenentwicklern [Mey92].

Overbeck bietet ein Integrationstestrezept in zwei Stufen an:

1. Finde eine Klasse, deren Basisklassen und Serverklassen bereits getestet sind. Teste diese Klasse mit Hilfe der bereits getesteten Klassen.
   Teste alle Aufträge an die Serverklassen.
   Wiederhole diesen Vorgang für jede Klasse.
2. Falls Klassen noch übrig bleiben, ist dies ein Zeichen für eine zyklische Nutzungsbeziehung. Im nächsten Schritt werden alle Klassen in einem solchen Zyklus behandelt. Jede Client-Klasse in einem Zyklus wird gegen sämtliche Serverklassen getestet. Wenn weiterhin Zyklen auftreten, werden jene Klassen ausfindig gemacht, deren Ersetzung durch einen Stub den Zyklus durchbrechen könnten. Diese Klassen werden mit Hilfe von Stubs getestet. Nachher werden die Stubs durch die Originalklassen ersetzt und die Aufträge an die Server getestet.

Zum Schluss sollten alle Klassen nicht nur gegen sich selbst, sondern auch gegen all ihre Serverklassen getestet sein. Dann sind sämtliche Beziehungen zwischen Klassen getestet worden [Ove94].

7.3.6 Propagierungsmustertest

Der Test von Propagierungsmustern stammt von Lieberherr und Xiao [LiXi93]. Er setzt einen so genannten *Class Dictionary Graphen* voraus, in welchem die Klassenabhängigkeiten – Vererbung, Aggregation und Assoziation – dokumentiert sind. Ein Propagierungsmuster definiert die Verantwortlichkeiten der einzelnen Objekte. Es wird hier also auf die Dienste abgestellt. Jeder Dienst wird der Reihe nach zusammen mit allen Zulieferdiensten getestet.

Lieberherr und Xiao schreiben, sie wollen Komponenten schrittweise integrieren, indem immer nur ein Teil der Propagierungsmuster nach dem anderen getestet wird. Dies wird wiederholt, bis alle Propagierungsmuster erprobt worden sind.

7.3.7 Reverse-Engineering-Test

Der Reverse-Engineering-Testansatz ist auf eine Forschungsarbeit von Kung et Al. zurückzuführen [KGH93]. Er stellte die Frage, welche anderen Klassen betroffen sind, wenn eine bestimmte Klasse getestet wird. Um diese Frage zu beantworten, werden einem Objektbeziehungsdiagramm die Klassenbeziehungen entnommen. Dieses Diagramm wird wiederum aus dem Klassenquellcode gewonnen. Die Knoten im Diagramm sind die Klassen. Die Kanten sind die Klassenbeziehungen wie Vererbung, Aggregation und Assoziation. Für jede Beziehung werden aus der Schnittstelle eine Reihe Testfälle generiert, die stellvertretend für die Schnittstellenausprägungen sind, um diese Beziehungen zu testen. Insofern werden die Klassen gegen die Schnittstellen getestet. Dies entspricht dem Schnittstellentest von einem der Autoren, der weiter unten eingehender behandelt wird.

7.3.8 Zustandsübergangstest

Den Zustandsübergangstest hat Binder entwickelt [Bin94c]. Er geht von den Zustandsdiagrammen in der OMT- bzw. UML-Spezifikation aus. Für jede Objektzustandsänderung wird ein Test definiert, um von einem fremden Objekt aus jene Zustandsänderung zu bewirken. Ein Testpfad wird definiert als eine Folge bestimmter Zustandsübergänge, die zu einem erwarteten Nachzustand aller betroffenen Objekte führt. Ein Integrationstestfall entspricht danach einem Pfad durch die Zustandsübergangsdiagramme. Binder bezeichnet diese als „Event/Response Threads“. Ein ähnlicher Testansatz wurde auch von Poston vorgeschlagen, um Testfälle aus einer OMT-Objektzustandsspezifikation abzuleiten [Pos94]. Der Ansatz von Poston wird im Zusammenhang mit dem Systemtest im nächsten Kapitel behandelt.

7.3.9 Integration durch zunehmenden Testumfang

Siegel schlägt in seinem Buch „Object-Oriented Testing“ einen Integrationstest mit drei Phasen und neun Stufen vor [Sie96]. Die erste Phase ist die Klasse-zu-Klasse-Integration, die zweite die Architekturintegration und die dritte die Integration der Benutzungsoberfläche.

In der Klasse-zu-Klasse-Integration sind folgende fünf Stufen enthalten:

- *Klassifikationstest*: Abgleich mit dem Entwurf
- *Protokolltest*: Bestätigung der Klasseninteraktionen
- *Schnittstellentest*: Kontrolle der Parameterübergabe
- *Synchronisierungstest*: Prüfung der Nebenläufigkeit
- *Aggregationstest*: Test der Sammelobjekte

In der Architekturintegration befinden sich die drei Stufen:

- *Horizontaltest*: Erprobung der Schnittstellen zwischen gleichrangigen Komponenten auf der gleichen Architekturstufe
- *Vertikaltest*: Bestätigung der Schnittstellen zwischen über- und untergeordneten Komponenten auf verschiedenen Architekturstufen, z.B. zwischen Frontend- und Backend-Komponente
- *Clustertest*: Test des Zusammenspiels zwischen Komponenten in einem Teilsystem

In der letzten Stufe wird die Benutzungsoberfläche mit getestet.

7.3.10 C++-Integrationstest

Einen speziellen vierstufigen Integrationstest für C++-Systeme empfiehlt Lee [LFC94]. In der ersten Stufe werden alle Beziehungen zwischen Systemelementen – Vererbungen, Fremdbeziehungen, Nachrichtenübergabe, Operationsaufrufe und Main-Programm – ermittelt und erprobt. In der zweiten Stufe werden zunächst die Wurzelobjekte getestet, d.h. jene Objekte, die nichts weiter vererben und keine weiteren Objekte benutzen, die Objekte am Ende der Klassenhierarchie. Danach wird die nächstübergeordnete Objektschicht getestet und so weiter hinauf, bis zur Spitze der Klassenhierarchie bzw. die endgültigen Basisklassen. In der dritten Stufe werden die Klassen mit rekursiven Beziehungen untereinander bzw. Zyklen getestet. Hier werden Stubs benutzt, um noch nicht einbezogene Objekte zu simulieren und die Zyklen zu unterbrechen. In der vierten Stufe wird das Hauptprogramm mit der globalen Steuerung getestet.

7.3.11 Assemblierungsansatz

Desfray schlägt vor, Zyklen in den Assoziationen zu entfernen, indem man eine oder mehrere Klassen durch so genannte Schnittstellenklassen ersetzt [Des94]. Anwenderobjekte können also nur über Schnittstellenobjekte miteinander kommunizieren. Die Schnittstellen werden benutzt, um Testargumente zu erzeugen und Testergebnisse zu validieren. Dieser Ansatz, der einen Build-In-Integrationstest beinhaltet, setzt natürlich voraus, dass der ganze Systementwurf im Hinblick auf den Integrationstest ausgelegt ist.
Nachdem die einzelnen Klassen eine nach der anderen miteinander über die Schnittstellentestklassen integriert sind, werden die Schnittstellenobjekte entfernt und die Anwendungsobjekte direkt miteinander verbunden. Untergeordnete Objekte, die Eigenschaften erben, werden hier nur dann getestet, wenn alle übergeordneten Objekte bzw. Basisobjekte vorher getestet wurden. Die Integration vollzieht sich also im Gegensatz zum Lee-Ansatz – Top-Down – von oben nach unten.

7.3.12 Integrationstest nach Komposition

Jorgenson und Erickson stellten fest, dass viele objektorientierte Systeme sich nur schwer nach konventionellen Testansätzen integrieren lassen, weil sie im Prinzip keine klare Hierarchie, sondern ein Netzwerk von gegenseitig abhängigen Objekten darstellen. Es gibt daher fast unendlich viele Pfade durch das Beziehungsnetz. Worauf es ankommt, ist, die anwendungsrelevanten Pfade zu identifizieren und zu testen. Ein Pfad beginnt mit einem externen Ereignis und endet mit einem veränderten Zustand der betroffenen Objekte und einer Mitteilung an die Systemumgebung. Dazwischen werden Objekte erzeugt und Nachrichten an sie verteilt. Deshalb liegt es nahe, pfadweise zu testen. Man nimmt einen Pfad nach dem anderen und testet das Zusammenspiel aller beteiligten Objekte auf dem Pfad. Dies wird als *atomic system function*-Testansatz bezeichnet [JoEr94].

7.3.13 Flutwellenansatz

Wave Front Integration ist Teil der SASY-Testmethodik von McGregor und Korson [MgKo94]. Nach diesem Ansatz sind die Klassen in der Reihenfolge zu entwickeln, wie sie getestet werden können. SASY ist somit ein testgetriebener Entwicklungsansatz. Nicht nur die Klassen, sondern auch die einzelnen Methoden werden eine nach der anderen erstellt, integriert und getestet, angefangen mit den Wurzelmethoden, die keine weiteren Methoden benutzen. Dadurch wird das System von unten nach oben, Methode für Methode und Klasse für Klasse integriert.

7.3.14 Objektkommunikationsansatz

Der Objektkommunikationsansatz zum Integrationstest stammt von Jüttner [JKN+94]. Objekte werden nach Jüttner über ihre Verwendung integriert: Operationsaufruf, Nachrichtenübergabe, Variablenverwendung. Daher müssen sämtliche Verwendungen zunächst einmal über eine statische Analyse ermittelt und nachdokumentiert werden. Danach werden die Objektverwendungen in einfache, mehrfache und zyklische klassifiziert (single, multiple und cyclic). Einfache Verwendung liegt vor, wenn eine Methode nur von einem Objekt aufgerufen wird. Mehrfache Verwendung bedeutet, dass eine Methode von mehreren Objekten aufgerufen wird. Die zyklische Verwendung ist gegeben, wenn die Aufrufe rekursiv sind, d.h. das verwendete Objekt verwendet auch den Verwender. Die einfache Verwendung ist am einfachsten, die zyklische Verwendung am schwierigsten zu testen.

7.4 Klassenintegrationstest

In einem objektorientierten System findet die Integration prinzipiell auf zwei verschiedenen Ebenen statt:

- der Klassenebene und
- der Komponentenebene.

Ein Grund für die Vielfalt der Integrationstestansätze ist, dass diese beiden Ebenen oft durcheinander gebracht werden. Ausschlaggebend für den Testansatz ist daher der Gegenstand der Integration, ob Klasse oder Komponente.

Im Falle der Klassenintegration geht es darum, eine Menge gegenseitig abhängiger Klassen zu testen – eine Klassenhierarchie. Natürlich setzt dies voraus, dass die einzelnen Klassenhierarchien voneinander unabhängig sind, d.h. zwischen ihnen dürfen keine Vererbungen, keine Aggregationen und keine direkten Assoziationen existieren. Notfalls muss man sich hier von den Grundsätzen der Objektorientierung entfernen, indem man Basisklassen dupliziert und Schnittstellenklassen hinzufügt, denn nur so ist es möglich, eigenständige, unabhängige Klassenhierarchien zu schaffen. In anderen Worten: man muss die Testbarkeit beim Entwurf der Systeme berücksichtigen.

Der Klassenintegrationstest wird zum einen über die Klassenhierarchie und zum anderen über die Klassenkollaborationen getrieben. Die Klassenkollaborationen werden wiederum über die Folge der Operationsaufrufe gesteuert. Insofern ist der Test der Klassenkollaborationen in der Terminologie vom Beizer ein Steuerflusstest, wonach die Ablaufsteuerung über Klassengrenzen hinweg verfolgt wird (Abbildung 7.8). Ein interaktionsbasiertes Modell hierzu ist von einem der Autoren entwickelt worden [Win00].

Eine testbare Komponente wird nicht mehr als 40 Klassen beinhalten, und zwar einschließlich Basisklassen, aber ohne abstrakte Klassen. Die Klassenhierarchie sollte nicht mehr als drei Vererbungsebenen haben – dies wird überall in der Literatur empfohlen. Das würde bedeuten, dass eine Hierarchieebene nicht mehr als 20 Klassen hat. Diese quantitativen Einschränkungen dürften etwas künstlich erscheinen, aber nur auf diese Weise ist die Testbarkeit zu gewährleisten [Bin94a].

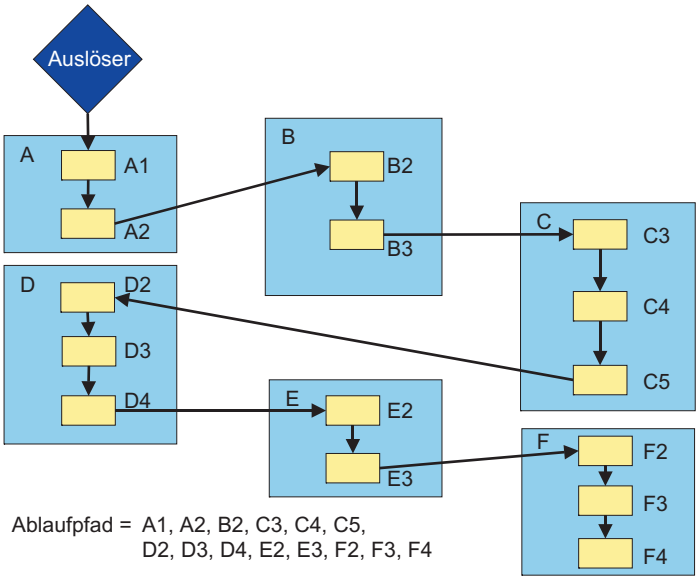


Abbildung 7.8 Steuerfluss-basierter Integrationstest

Wenn also von der Klassenintegration die Rede ist, dann geht es darum, bis zu 40 Klassen auf maximal drei Stufen zu testen. Getestet wird von den Basisklassen aus – ob oben oder unten – ist eine Frage der Integration. Insofern die Ahnenklassen oben und die erbenden Klassen unten sind, fängt man mit der obersten Hierarchiestufe an und testet die Ahnen zuerst. Die obersten Basisklassen dürften nur Beziehungen untereinander bzw. zu anderen Klassen auf der gleichen Stufe haben. Die erste Stufe der Integration beschränkt sich auf den Test der Beziehungen.

Nachdem die obersten (Ober-)Klassen getestet worden sind, wird die nächsttiefere Stufe der (Unter-)Klassen getestet. Hier werden nicht nur die horizontalen Beziehungen bzw. die Assoziationen zwischen den Klassen auf der gleichen Stufe, sondern auch die vertikalen Beziehungen bzw. die Vererbungen getestet. Bei den Assoziationen können in der Tat Zyklen auftreten, und diese müssen sehr sorgfältig getestet werden, am besten mit Eingriffen in den Code.

Dieser Test der horizontalen und vertikalen Beziehungen wird auch für die dritte, unterste Hierarchiestufe wiederholt. Die Ahnenklassen sollten bis dahin alle schon

getestet sein, sodass hier nur Fehler in den erbenden Klassen auftreten dürfen. Wichtig ist vor allem der Test redefinierter bzw. überschriebener Operationen.

Alle Beziehungen zu Objekten außerhalb der Komponenten unter Test sind zu Instanzen von Schnittstellenklassen bzw. Stubklassen anzulegen. Sie werden nicht geprüft. Zugriffe auf persistente Objekte in der Datenbank sind zuzulassen, die Testdatenbanken müssen aber vorher aufgebaut werden. Falls die Zugriffe über eine zusätzliche Datenbankschale erfolgen, ist zu empfehlen, auch solche Zugriffe mit den Schnittstellenklassen abzufangen und zu simulieren.

Insofern, als die Komponente unter Test eine Benutzungsoberfläche bedient, kann man von der Benutzungsoberfläche aus testen. Es werden Ereignisse generiert, und diese Ereignisse lösen eine Sequenz von Operationsaufrufen aus. Wenn aber die Komponente unter Test keine Benutzungsoberfläche, sondern nur eine interne Schnittstelle – API – hat, dann braucht man einen Testtreiber, um diese Schnittstelle zu bedienen und mit Parameterwerten zu versorgen.

Am besten ist es, wenn der Integrationstester einen Ablaufverfolger einschalten kann, um die Aufrufsequenzen sowie die Objekterzeugung zu verfolgen. Ersteres beinhaltet den Aufruf einer Durchlaufprotokollierungsfunktion zu Beginn jeder Membermethode, Letzteres bedeutet den Aufruf einer Operation zur Protokollierung des Objektzustands in jeder Konstruktor- und Destruktormethode. Damit wird es möglich, den Verlauf des Integrationstests in Form einer dynamischen Pfadanalyse zu verfolgen und auch die Anfangs- und Endezustände aller Objekte zu validieren [Bei95].

Mit dem *Assoziationstest*, dem *Interaktionstest* und dem *Test dynamisch gebundener Operationsaufrufe* werden im weiteren drei Techniken für den Klassenintegrationstest vorgestellt.

7.4.1 Assoziationstest

Das Ziel des Assoziationstests ist es, Fehler in den Verbindungen innerhalb einer Menge eng zusammenarbeitender Objekte bzw. Komponenten zu finden [Win01]. Hierbei spielt die Implementation der Assoziationen des Klassendiagramms eine Rolle. Die Fehler, die im Assoziationstest aufgefunden werden sollen, sind in erster Linie falsche Zielobjekte in einer Verbindung und fehlende Verbindungen.

Der Nutzen des Assoziationstests besteht darin:

- gezielt die im Klassendiagramm festgehaltene Strukturinformation zu untersuchen und
- die Einhaltung der Multiplizitäts-Beschränkungen für Assoziationen zu prüfen.

Die Voraussetzung für den Assoziationstest ist das Vorliegen von Multiplizitäten für alle Enden der Assoziationen im Klassendiagramm und die Annotation der Assoziationen als Aggregation oder Komposition.

Zunächst konzentriert man sich auf die Assoziationen im Klassenmodell. Eine Assoziation zwischen zwei Klassen bedeutet, dass Instanzen dieser Klassen miteinander verbunden sein können, d.h. über einen längeren Zeitraum als z.B. den Ablauf einer einzigen Operation Kenntnis voneinander haben.

Aus den Assoziationen im Klassendiagramm werden Testfälle abgeleitet, welche Fehler der folgenden Kategorien aufdecken sollen, die durch die vielfältigen, z.T. durchaus komplexen Implementationsmöglichkeiten von Assoziationen verursacht werden [Bin99]:

- Einfügeanomalien,
- Änderungsanomalien,
- Löschanomalien und
- inkonsistente Verbindungen im Falle von beidseitig navigierbaren Assoziationen.

Ergebnis dieses Arbeitsschritts ist eine Testfallmatrix, in der für jede Assoziation/Aggregation/Komposition im Klassenmodell die zu testenden Anzahlen jeweils über die entsprechende Objektbeziehung miteinander verbundener Instanzen angegeben werden. Zur Ermittlung der Testfälle nutzt man die Tatsache, dass die Multiplizitäten Wertebereiche für die entsprechende Implementation darstellen. Zur Testfallermittlung können somit die Grenzwertanalyse und die Äquivalenzklassenbildung bezüglich der Multiplizitäten verwendet werden. Es sind die untere Grenze (Min.), die untere Grenze–1, ein typischer Wert, die obere Grenze (Max.) sowie die obere Grenze+1 zu prüfen. Für jede Assoziation ergeben sich somit maximal 5*5=25 Testfälle, die tabellarisch notiert werden.

Jeder Testfall beschreibt eine Objektkonstellation, die eine entsprechende Anzahl von miteinander verbundenen Instanzen enthält. Zusätzlich werden durch eine entsprechende Verkettung von Testfällen auch Einfüge-, Änderungs- und Löschanomalien geprüft. Bei mehrwertigen Assoziationen ist nach jedem Testfall auch die Konsistenz der Objektbeziehungen zu prüfen.

Die resultierenden Testfälle werden in – ggf. verschachtelte – Nachrichtensequenzen zur Erzeugung bzw. Manipulation der entsprechenden Objektkonstellationen umgesetzt. Ggf. sind auch Generalisierungsbeziehungen zu berücksichtigen, da ja anstelle von Instanzen einer Oberklasse auch Instanzen der Unterklassen auftreten können (vgl. Abschnitt 7.4.3, „Test dynamisch gebundener Operationsaufrufe“).

Hat man die Testfälle abgeleitet und Testskripte erstellt, so sind für jeden Testfall wiederum Testdaten abzuleiten bzw. zu generieren. Hierzu muss der Tester die Eingaben durch die Wertebereiche bzw. die Vorzustände der Eingabe ergänzen (vgl. Abschnitt 6.5.3, „Zusicherungstest“).

Endekriterien für den Assoziationstest sind:

- Für jede Assoziation sind Testfälle für die Multiplizitäten der Assoziationsenden nach der Äquivalenzklassenbildung und Grenzwertanalyse festgelegt.
- Einfüge-, Änderungs- und Löschanomalien sind berücksichtigt.
- Generalisierungsbeziehungen („geerbte“ Assoziationen) wurden berücksichtigt.

7.4.2 Interaktionstest

Das Ziel des Interaktionstests ist es, Fehler im Zusammenspiel einer Menge eng zusammenarbeitender Klassen oder Komponenten aufzudecken [Win01]. Hierbei spielen die möglichen Reihenfolgen von Operationsaufrufen bzw. Nachrichten eine Rolle. Die Fehler, die durch Interaktionstests aufgefunden werden sollen, sind in erster Linie Inkompatibilitäten in den Schnittstellen zwischen den Klassen und bzw. der Komponente und der Umgebung. Dazu gehören unverträgliche Typen, unterschiedlich gereihte Parameterlisten, Speicherverlust, falsches Zielobjekt eines Aufrufs, fehlende Funktionen und fehlende Ausnahmebehandlung.

Der Nutzen des Interaktionstests besteht darin, die Erfüllung der mit Interaktionsdiagrammen vorgegebenen Spezifikationen durch die Klassen eines Pakets zu prüfen. Die Voraussetzung für den Interaktionstest ist das Vorliegen von detaillierten Sequenzdiagrammen für die Abläufe von Operationen.

Znuächst konzentriert man sich auf alle Interaktionsdiagramme für eine bestimmte Operation. Hierzu gehören in erster Linie solche Interaktionsdiagramme, deren Ablauf mit der Operation startet, sowie andere Interaktionsdiagramme, in denen die Operation innerhalb des Ablaufs aufgerufen wird. Jedes dieser Interaktionsdiagramme stellt einen konkreten (ggf. partiellen) Ablauf der Operation dar.

Interaktionsdiagramme werden in der UML entweder als Sequenz- oder als Kollaborationsdiagramm eingesetzt. Die in den Kollaborationsdiagrammen enthaltene strukturelle Information (Objektbeziehungen) wird schon durch die Technik „Assoziationstest“ berücksichtigt, sodass sich diese Technik auf Sequenzdiagramme bezieht.

Zwei Arten von Kontrollinformation, welche in Interaktionsdiagrammen dargestellt werden können, sind im Rahmen des Interaktionstests besonders erwähnenswert:

- Bedingungen zeigen in Form von Booleschen Ausdrücken an, wann eine Operation aufgerufen wird (z.B. `[NOT allePostenErzeugt]` und `[NOT MengeOK]`). Zur Formulierung der Bedingungen verwendet man die OCL [WaKl99] und bezieht sich z.B. auf Attribute des dienstnutzenden Objekts oder „Variablen“, denen Rückgabewerte von zuvor aufgerufenen Operationen zugewiesen worden sind.
- Das Iterationssymbol `*` zeigt an, dass eine Operation mehrmals (ggf. auf verschiedene Zielobjekte) angewendet werden kann. Das Iterationssymbol wird

verwendet, wenn z.B. innerhalb einer Operationsausführung über Objekte iteriert wird, die mit dem ausführenden Objekt aufgrund einer mehrwertigen Assoziation zwischen den entsprechenden Klassen verbunden sind.

Darüber hinaus können in Sequenzdiagrammen noch die Erzeugung und die Zerstörung von Objekten kenntlich gemacht werden. Im ersten Fall zeigt der Pfeil auf das Objektsymbol, im zweiten Fall wird die Lebenslinie mit einem Kreuz abgeschlossen. Objekte können sich selbst zerstören, wenn sie ihre Aufgabe erfüllt haben, oder durch andere Objekte zerstört werden.

Ein Sequenzdiagramm ohne Kontrollinformationen stellt einen einzigen Ablauf dar und führt somit auch nur zu einem Testfall. Ein solcher Testfall wird durch die Angabe von Testdaten, d.h. konkreter (Unter-)Klassen für die beteiligten Objekte bei dynamisch gebundenen Aufrufen sowie konkreter Werte für die Parameter der „von außen“ eingehenden Nachrichten (Stimuli), zu einem ausführbaren Test (vgl. auch Technik „Test dynamisch gebundener Operationsaufrufe“). Erwartetes Ergebnis dieses Tests sind der im Sequenzdiagramm vorgegebene Ablauf sowie die Ergebnisse des bzw. der Stimuli.

Enthält ein Sequenzdiagramm Kontrollinformation oder sind unterschiedliche Abläufe einer Operation mit mehreren Sequenzdiagrammen modelliert, so können mehrere Abläufe aus dem Diagramm abgeleitet werden, die jeweils einen Testfall ergeben. Hierzu wandelt man das Sequenzdiagramm bzw. die Menge zugehörender Sequenzdiagramme in einen äquivalenten Kontrollflussgraphen um [Bin99]. Mit dem so erzeugten Kontrollflussgraphen können dann ablauforientierte („white box“) Testfälle abgeleitet werden.

Hat man die Testfälle abgeleitet und Testskripte erstellt, sind für jeden Testfall wiederum Testdaten abzuleiten bzw. zu generieren. Hierzu muss der Tester die Eingaben durch die Wertebereiche bzw. die Vorzustände der Eingabe ergänzen (vgl. auch Abschnitte 6.5.3, „Zusicherungstest“ und 6.5.4 „Zustandstest“).

Zur Ermittlung entsprechender Testdaten werden die Bedingungen und Iterationen im Sequenzdiagramm herangezogen. Die Objektkonstellation zu Beginn eines Tests ergibt sich aus den Objekten am oberen Ende des Sequenzdiagramms (die also nicht innerhalb des bzw. der Szenarien erzeugt werden). Als Testendekriterien kommen die Überdeckungsmaße für Kontrollflussgraphen in Frage (siehe Tabelle 4-1).

Die resultierenden Testfälle werden in – ggf. verschachtelte – Nachrichtensequenzen zur Erzeugung bzw. Manipulation der entsprechenden Objektkonstellationen umgesetzt. Ggf. sind auch Generalisierungsbeziehungen zu berücksichtigen, da ja anstelle von Instanzen einer Oberklasse auch Instanzen der Unterklassen auftreten können.

Endekriterien für den Interaktionstest sind:

- Alle Konstruktoren und Destruktoren wurden mindestens einmal ausgeführt, d.h. von jeder Klasse der Komponente wurde mindestens eine Instanz erzeugt und wieder zerstört.

- Testfälle für alle Sequenzdiagramme sind erstellt.
- Für jedes Sequenzdiagramm mit Kontrollflussinformation sind Testfälle erstellt, die jede Kante und jeden Knoten des zugehörigen Kontrollflussgraphen überdecken.
- Generalisierungsbeziehungen („geerbte“ Interaktionen) wurden berücksichtigt.

7.4.3 Test dynamisch gebundener Operationsaufrufe

Das Ziel des Tests dynamisch gebundener Operationsaufrufe ist die Prüfung eines Operationsaufrufs unter Berücksichtigung dynamischer Bindungen (Polymorphismus) und möglicher Zustände von beteiligten Objekten [MgSy00] [Win01]. Hierbei werden primär die aufgrund der Generalisierungsbeziehungen bzw. der implementierten Schnittstellen (Interfaces) möglichen Klassen des aufrufenden Objekts, der Parameterobjekte und des die Operation ausführenden Objekts betrachtet. Zusätzlich kann berücksichtigt werden, dass sich die beteiligten Objekte in verschiedenen Zuständen befinden können.

Der Nutzen des Tests dynamisch gebundener Operationsaufrufe besteht darin, die kombinatorisch schnell explodierende Anzahl von Testfällen, mit denen Fehler bei der Verwendung überschriebener oder in einem neuen Kontext ausgeführter geerbter Operationen gefunden werden können, systematisch einzugrenzen, ohne die Fehler-Aufdeckungswahrscheinlichkeit zu sehr zu verringern. Zusätzlich können auch aufgrund nicht korrekt berücksichtigter Zustände verursachte Fehler gefunden werden (vgl. auch Abschnitt 6.5.4, „Zustandstest“).

Die Voraussetzung für den Test dynamisch gebundener Operationsaufrufe ist das Vorliegen eines Klassenmodells, der vollständigen Signatur und Spezifikation der aufgerufenen Operation und des Quellcodes der aufrufenden Operation (Operation unter Test, OpUT). Sollen Zustände betrachtet werden, so werden auch Zustandsdiagramme der beteiligten Klassen benötigt.

Zunächst fokussiert man einen bestimmten Operationsaufruf innerhalb der OpUT. Dabei sind alle beteiligten Objekte bzw. deren Typen (Klassen/Interfaces) anhand der Generalisierungsbeziehungen und der realisierten Schnittstellen (Interfaces) zu ermitteln. Für jede Klasse werden die relevanten Zustände der betrachteten Instanzen innerhalb der Operationsausführung aus den Zustandsdiagrammen ermittelt. Ergebnis ist eine Menge von Klassen und ihren Unterklassen, die in einem Operationsaufruf involviert sein können, sowie der für diesen Aufruf relevanten Zustände von Instanzen dieser Klassen.

Danach erfolgt eine Eingrenzung der zu testenden Kombinationen von Klassen und Zuständen unter Zuhilfenahme der Grundidee, nicht alle möglichen Kombinationen, sondern lediglich alle *paarweise* verschiedenen Kombinationen für je zwei Objekte

bzw. Zustände von Objekten zu testen. In anderen Worten: Man wählt die Testfälle so aus, dass nur alle möglichen Kombinationen von je zwei Objekten/Zuständen geprüft werden. Nach dieser Eingrenzung sind für die auszuführenden Testfälle Testskripte zu erstellen, zu parametrisieren und auszuführen.
Endekriterien für den Test dynamisch gebundener Operationsaufrufe sind:

- Ist der Aufruf an einen Parameter gerichtet, erfolgte der Aufruf aus Instanzen der in der Signatur angegebenen Klasse und aller Unterklassen in allen möglichen Zuständen.
- Ist der Aufruf an eine Instanzvariable gerichtet, erfolgte der Aufruf aus Instanzen der in der Variablendeklaration angegebenen Klasse und all ihrer Unterklassen in allen möglichen Zuständen.
- Die aufgerufene Operation ist von Instanzen der definierenden Klasse und all ihrer Unterklassen in allen möglichen Zuständen ausgeführt worden.
- Die Parameter des Aufrufs waren Instanzen der in der Signatur angegebenen Klasse und all ihrer Unterklassen in allen möglichen Zuständen.

7.5 Komponentenintegrationstest

Beim Komponentenintegrationstest werden mehrere Komponenten bzw. bereits gebündelte und getestete Klassenmengen in ihrer Interaktion miteinander erprobt. Eine Komponente ist ein Stück ausführbarer binärer Code. Sie hat viele Bezeichnungen, je nachdem, in welcher Umgebung sie sich befindet: Dynamic Link Load, Load Module, Enterprise Java Bean (ejb), COM+ Component, Corba Component oder einfach Programm. Sie hat nach außen eine Schnittstelle, die sich intern oder extern bedienen lässt. Eine externe Schnittstelle ist die Benutzungsoberfläche. Sie kann aber auch eine Batchschnittstelle sein. Sie wird von einem Menschen oder von einem Gerät außerhalb des eigenen Systems bedient.

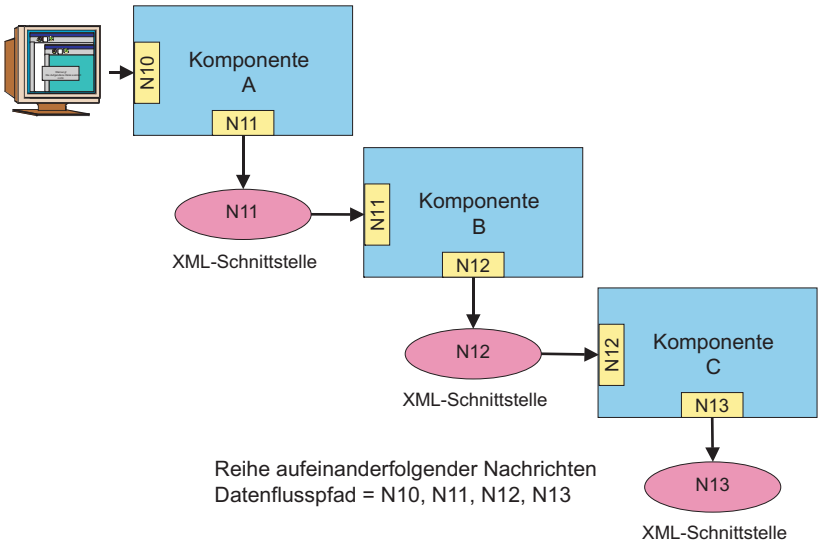


Abbildung 7.9 Datenfluss-basierter Integrationstest

Eine interne Schnittstelle ist innerhalb des Speichers – hier werden Parameter abgelegt und die Komponente angestoßen von einer anderen Komponente bzw. einem anderen Programm. Es handelt sich also hier um eine Programm-zu-Programm-Kommunikation. Auch hierfür gibt es zahlreiche Techniken mit unterschiedlichen Bezeichnungen, u.a. Application Program Interfaces, Remote Procedure Calls, Sockets and CORBA Connections. Ausschlaggebend ist die Tatsache, dass zwei aktiv laufende Programme Daten miteinander austauschen (Abbildung 7.9).

Der Komponentenintegrationstest ist ein Test des Datenflusses zwischen getrennten Komponenten. Die eine Komponente sendet eine Nachricht, die andere empfängt sie. In dieser Hinsicht entspricht der Komponententest einem Datenflusstest nach der Klassifikation vom Beizer [Bei95].

Im Gegensatz zur Klassenintegration, die sich in einem Programm und somit in einem Adressraum abspielt, geht es hier um die Verbindung verschiedener Programme in unterschiedlichen Adressräumen. Die eine Komponente befindet sich auf dem PC-Arbeitsplatz, die zweite auf einem Unix-Server und die dritte auf dem Host. Die PC-Komponente benutzt die Unix-Komponente, die wiederum benutzt die Hostkomponente. Dies wäre ein Beispiel der vertikalen Integration. Andererseits können aber auch Komponenten auf der gleichen Rechnerstufe miteinander kommunizieren, z.B. eine PC-Komponente benutzt eine Komponente auf einem anderen PC oder eine Server-Komponente tauscht Daten mit einer anderen Server-Komponente aus. Dies wäre ein Beispiel der horizontalen Integration.

Egal ob vertikal oder horizontal integriert wird, das Ziel ist dasselbe: Die Interaktion zwischen getrennt zusammengestellten Programmen bzw. Komponenten zu testen. Um dies zu bewerkstelligen, werden die Datenbanken, auf welche die Kom-

ponenten zugreifen, benötigt, ebenso die Kommunikationseinrichtung, z.B. DCOM oder CORBA und die Benutzungsoberfläche. Datenbanken werden mit systemweit gültigen Testdaten gefüllt. Die Testfälle werden von der Klassenintegration bzw. vom Test der Einzelprogramme übernommen und durch programmübergreifende Testfälle ergänzt.

Im Klassenintegrationstest wurden die Verbindungen zu fremden Komponenten gekappt bzw. durch Stellvertreterklassen abgefangen und simuliert. Jetzt werden diese Stubs entfernt, sodass die Interprogrammkommunikation stattfinden kann. Allerdings werden sie nicht alle auf einmal, sondern nur schrittweise entfernt. Darin liegt der Kernpunkt der Komponentenintegration: es wird stets nur eine Interaktion nach der anderen bzw. die nächste RPC, RMI oder CORBA-Verbindung getestet. Die anderen Verbindungen bleiben weiterhin gekappt.

Das Integrationsverfahren sieht also vor, dass eine Komponente nach der anderen integriert und für jede Komponente eine Interaktion nach der anderen getestet wird. Diese schrittweise Integration kann sich sowohl vertikal über verschiedene Architekturstufen als auch horizontal innerhalb der gleichen Architekturstufe entfalten. Die programminterne Ablaufüberwachung und Zustandskontrolle, die zum Zweck der Klassenintegration gedient hat, wird ausgeschaltet. Stattdessen werden nur die Kontrollübergaben zwischen Komponenten verfolgt und die Inhalte der Programmschnittstellen protokolliert, z.B. die Nachrichten an fremde Komponenten. Der Tester muss sich versichern, dass die Programm-zu-Programm-Verbindung funktioniert, die korrekten Argumente übergeben werden und die korrekten Ergebnisse zurückkommen. Ferner muss er versuchen, alle Ausnahmebedingungen, die mit der Kommunikation verbunden sind, auszulösen und die korrekte Fehlerbehandlung zu bestätigen.

Der Komponentenintegrationstest ist letztendlich ein Test der Kommunikation zwischen getrennt entwickelten, eigenständigen Bausteinen, die im gleichen Adressraum oder auf verschiedene Adressräume verteilt sind. Je mehr Adressräume bzw. je mehr Plattformen und je mehr Bausteine es gibt, umso aufwändiger der Test. Hinzu kommt die Anzahl der Interaktionen, die es zu testen gilt. Als Konsequenz kann der Komponentenintegrationstest recht lange dauern.

7.6 Integrationstest verteilter Objekte

Verteilte Objekte sind in mehreren Adressräumen auf verschiedenen Rechnern verteilt. Zum Beispiel ist die Präsentationslogik in einem Java-Applet auf dem Arbeitsplatzrechner, die Applikationslogik in einer Menge C++-Klassen auf dem Serverrechner und die Zugriffslogik in einem alten COBOL-Programm auf dem Hostrechner realisiert. Alle drei Rechner sind Knoten in einem betrieblichen Netzwerk. Die Aufgabe der Klassenintegration wäre es, die Java-Klassen auf dem Ar-

beitsplatz und die C++-Klassen auf dem Server untereinander zu integrieren. Die Aufgabe der Komponentenintegration ist es, die Verbindung der drei entfernten Komponenten zu testen.

Verbindungen zwischen entfernten Komponenten werden über Remote Procedure Calls, Sockets, Application Program Interfaces, Remote Method Invocations oder über einen Request Broker hergestellt [MoZa95]. Da eine Komponente mehrere solcher Verbindungen haben kann, wird es erforderlich sein, jede Verbindung bzw. Steuerungsübergabe mit jeder relevanten Kombination von Parametern zu testen. Jede Verbindung findet in Form eines Auftrags statt. Der Sender als Auftraggeber sendet einen Auftrag an einen Empfänger, den Auftragnehmer. In dem Auftrag werden die auszuführende Operation bzw. Operationen identifiziert und deren Parameter angegeben. Als Antwort gibt der Auftragnehmer ein oder mehrere Ergebnisse zurück. Ausserdem wird vom Auftraggeber bestimmt, was zu tun ist, wenn etwas Außergewöhnliches passiert, z.B. wenn der Auftragnehmer nicht auffindbar ist. Dies ist die so genannte Ausnahmeregelung.

Beim Test kommt es darauf an, möglichst viele Auftragsarten mit möglichst vielen unterschiedlichen Parametern zu bestätigen. Wenn nur die Senderkomponente getestet wird, lassen sich ihre Aufträge mit Stubs oder Schnittstellenobjekten abfangen und prüfen, ob die Zieloperationen und deren Argumente plausibel sind. D.h. die Verbindung muss gar nicht stattfinden, um die Korrektheit der Nachricht zu bestätigen (Abbildung 7.10). Wenn andererseits nur die Empfängerkomponente getestet wird, lassen sich die eingehenden Aufträge durch einen Testtreiber oder Schnittstellensimulator erzeugen. Sobald die Antwort erfolgt, kann der Treiber die Ergebnisse protokollieren oder sogar automatisch validieren, etwa durch einen Soll-Ist-Abgleich.

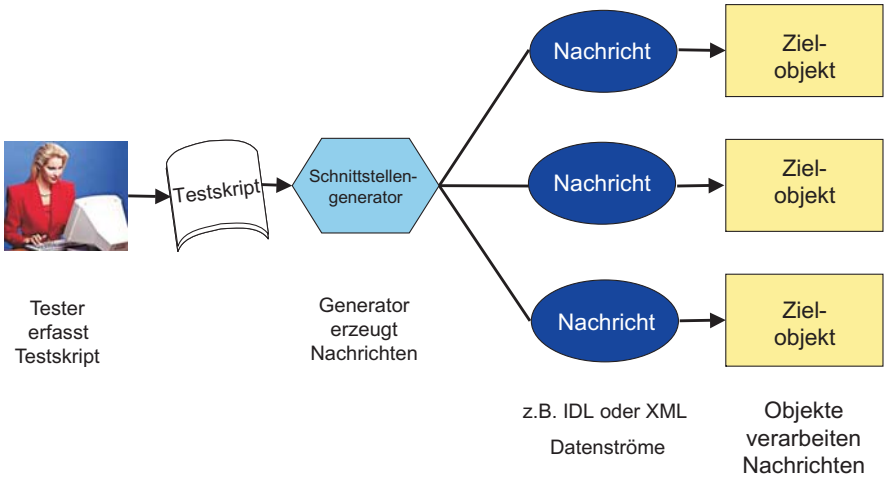


Abbildung 7.10 Schnittstellentest

Falls eine Komponente sowohl Auftraggeber als auch Auftragnehmer ist, müsste diese aus beiden Richtungen getestet werden, d.h. mit Treibern und Stubs.
Nachdem die Schnittstellen aller beteiligten Komponenten an und für sich getestet worden sind, werden einzelne Verbindungen nicht mehr abgefangen, sondern durchgelassen. Dennoch ist zu empfehlen, auch die durchgelassenen Nachrichten zwecks Kontrolle zu protokollieren. Das Gleiche gilt für die Parameter. So werden die Verbindungen eine nach der anderen ausprobiert, bis man die Gewissheit hat, dass alle Interaktionen stimmen. Dabei darf man nicht vergessen, auch die Ausnahmeregelungen zu testen.

7.6.1 Test einer CORBA-Schnittstelle

Mit einem Object Request Broker ist es möglich und ratsam, den Integrationstest gleich in die Schnittstellenbeschreibung einzubauen. Dies ist deshalb möglich, weil CORBA die Schnittstellen als eigenständige Source-Members in IDL vorsieht.

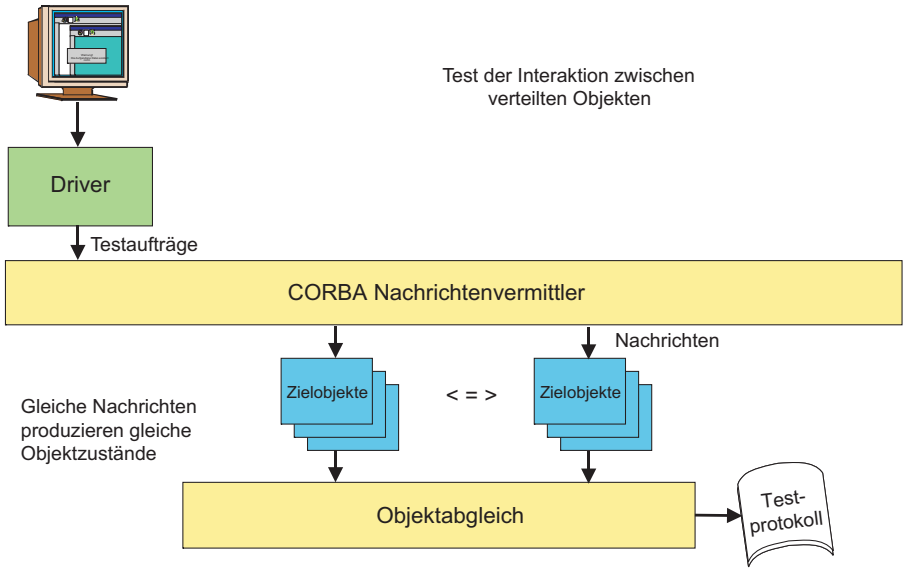


Abbildung 7.11 Test mit CORBA-Nachrichten

Das Beispiel hier ist aus dem Buch von Robert Orfali und Dan Harkey „Client/Server Programming with Java and CORBA“ entnommen [OrHa97]. In einem CORBA-Netz sind drei Objektarten zu finden: das Clientobjekt, das Serverobjekt und das Schnittstellenobjekt dazwischen. Im folgenden Beispiel fordert das Clientobjekt das Serverobjekt auf, eine Integerzahl hochzuzählen. Die IDL-Schnittstelle sieht wie folgt aus:

```
module counter
{
   interface count
   {
      attribute
         inout long sum
         long increment ();
   };
};
```

Die Serverklasse, wo die Zählung stattfindet, ist wie folgt definiert:

```
class count Impl:public_SK_counter::SK_Count
{
   private:
      long_sum;
   public;
      CORBA::long sum ();
      void sum (CORBA::Long val);
      CORBA::long increment ();
}
// Operation für die Hochzählung
CORBA::long Count_Impl::increment ();
{
   this ->_sum++;
   return this ->_sum;
}
```

Um diese Verbindung zu testen, kann die IDL-Schnittstellenbeschreibung leicht ergänzt werden, und zwar durch Zusicherungen, die bestimmte Eingangswerte generieren und Ausgangswerte validieren (Abbildung 7.11). Eine SET-Zusicherung gibt eine Reihe stellvertretender Werte vor, z.B.

```
Assert pre sum = SET(10, 20, 30, 40);
```

Eine RANGE-Zusicherung gibt die untere und obere Grenze eines Wertebereichs vor:

```
Assert pre sum = RANGE(1:99);
```

Eine absolute Zusicherung weist einen einzelnen Wert zu, so z.B.

```
Assert pre sum = 50;
```

Die Nachbedingung prüft, ob ein Ergebniswert zu den angegebenen Sollwerten gehört. `Assert post sum = SET(11, 21, 31, 41);` bedeutet, dass der erste Rückgabewert 11, der zweite 21, der dritte 31 und der vierte 41 sein muss.

Die Nachbedingung `Assert post sum = RANGE(1:99);` verlangt lediglich, dass der Rückgabewert irgendwo zwischen 1 und 99 liegt.

`Assert post sum = 50;` würde bedeuten, dass der Rückgabewert genau 50 sein müsste.
Mit der Einfügung der Zusicherungen wird die Beispielschnittstelle wie folgt verändert:

```
module counter
{
   interface count
   { attribute inout long sum;
      //Assert Nr_Testcase = 3;
     //Assert pre sum = SET(0, 49, 99);
        long increment();
     //Assert post sum = SET(1, 50, 100);
     }
}
```

Über einen Präprozessor werden die Assert-Anweisungen in Zuweisungen eines generierten Client-Objekts umgewandelt, d.h. es wird ein künstlicher Client erzeugt, der die entfernte Operation *increment* dreimal aufruft mit den Argumenten 0, 49 und 99. Nach jedem Aufruf wird der Rückgabewert sum geprüft, ob er nach dem ersten Aufruf 1, nach dem zweiten 50 und nach dem dritten 100 ist. Wenn nicht, wird ein Fehler gemeldet.

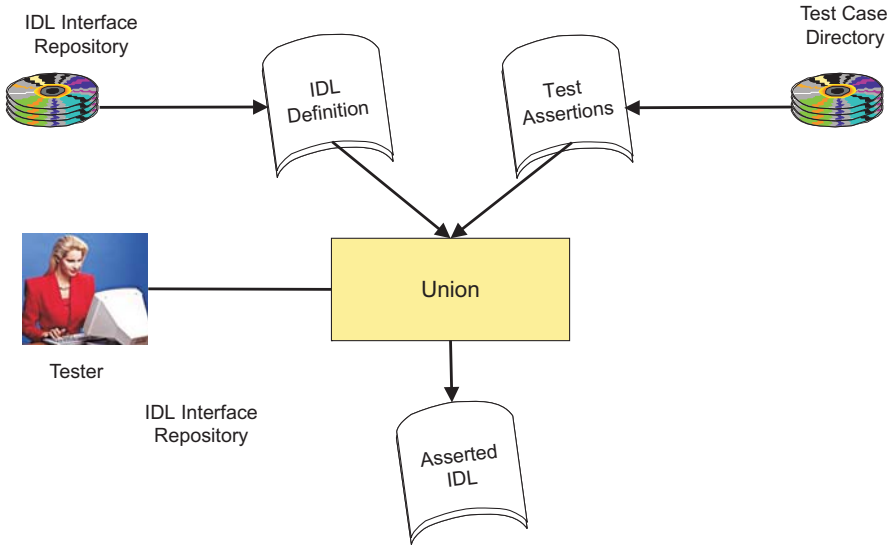


Abbildung 7.12 Test einer IDL-Schnittstelle

Das Beispiel ist zwar sehr einfach, doch es genügt, um zu zeigen, worum es hier geht – nämlich darum, entfernte Methoden mit ausgewählten Argumenten anzu-

steuern. Dadurch wird nicht nur der Verbindungsmechanismus, sondern auch die Korrektheit des Serverobjekts und die Korrektheit der Schnittstellendefinition bestätigt. Voraussetzung dafür ist nur ein Werkzeug, um aus einer IDL-Schnittstellendefinition eine stellvertretende Client-Klasse zu generieren. Das Werkzeug IDLTEST leistet genau das. Die einzige Aufgabe des Testers besteht darin, Asserts in die IDL-Schnittstellen einzubauen. Darüber hinaus generiert IDLTEST auch nach Wahl ein künstliches Serverobjekt, das alle Argumente und Ergebnisse, die über die CORBA-Schnittstelle fließen, protokolliert. Dies wird dadurch realisiert, dass zwei weitere Operationsaufrufe in die Schnittstelle eingefügt werden: eine vor dem eigentlichen Ziel-Operationsaufruf und eine nach dem eigentlichen Ziel-Operationsaufruf.

Dies sieht in einem etwas abgewandelten Beispiel so aus:

```
module counter
{
   interface count
   { attribute in last_sum;
      attribute out new_sum;
      void $IDL_Protocol(last_sum);// Eingabe_Protokoll
      long increment();
      void $IDL_Protocol(new_sum); //Ausgabe_Protokoll
   }
}
```

Die Operation `$IDL_Protocol` registriert vor dem Aufruf der Operation `increment()` den Stand der Eingangsparameter und nach dem Aufruf den Stand der Ausgangsparameter.

Durch die Verbindung der Zusicherungen mit den Protokollfunktionen können die Argumente vor dem Aufruf sowohl erzeugt als auch protokolliert und die Ergebnisse nach dem Aufruf sowohl geprüft als auch protokolliert werden (Abbildung 7.13). Es zeigt sich eben, dass die IDL-Schnittstellen viele Möglichkeiten für die Steuerung des Integrationstests anbieten [Sne98].

```
Module Account
{interface Withdrawal: Transaction
    {   attribute string Bank;
        attribute double Account;
         attribute String Name;
        attribute float Balance;
        // Assert pre Bank = „National“;
        // Assert pre Account = SET (1000, 3000, 5000, 9999);
        // Assert pre Name = SET („Mary“, „Jane“, „Nally“, „Nancy“);
        // Assert pre Balance = SET (100, 400, 500,999);
        // Assert pre Pin = SET (1024, 2056, 3000, 4048);
        // Assert pre Account = SET (1000, 3000, 5000, 9999);
        // Assert pre payment = SET (100, 300, 600, 999);
        // Assert start $ CASE = RANGE (1:4);
         void withdrawal   (in short PIN,
                            in double account,
                            in float payment,
                            out string message_text;
        raises („withdrawal_not_possible“);
        // If ( $CASE == 1  )
        // Assert post Balance = 0;
        // If ( $CASE == 2 )
        // Assert post Balance = 100;
        // If ( $CASE == 3 )
        // Assert post message _text=„Withdrawal_not_possible“;
     .....
    }
}
```

Abbildung 7.13 IDL Testprozedur

7.6.2 Test einer XML-Schnittstelle

In letzter Zeit wird die W3C Sprache XML – eXtensible Markup Language – zunehmend verwendet, um Daten zwischen Komponenten auszutauschen [WHB01]. Der Hauptvorteil von XML als Schnittstellensprache liegt vor allem darin, dass sie eine Datenschnittstelle verkörpert, während IDL eine Funktionsschnittstelle ist. Über IDL werden bestimmte entfernte Operationen mit Parameter angesprochen. Dies setzt voraus, dass der Client weiß, welche Operationen der Server hat. Im Falle von XML ist das nicht mehr notwendig. Der Client braucht über den Server überhaupt nichts zu wissen. Er stellt ihm nur ein Dokument zur Verfügung und erhält ein anderes Dokument zurück. Wie das zweite Dokument zustande kommt, ist Sache der Serverkomponente. Aus der Sicht des Clients ist der Server wahrlich eine Black-Box.

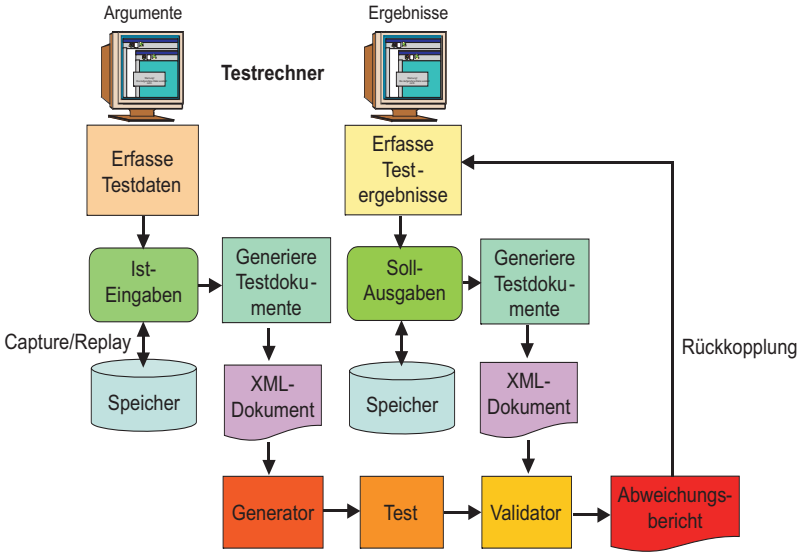


Abbildung 7.14 Test mit XML-Dokumenten

Dies erleichtert natürlich den Integrationstest, denn der Tester muss auch nichts über die Serverkomponente wissen. Er muss nur wissen welche Struktur und welchen Inhalt die XML-Dokumente haben sollte. Die Struktur geht entweder aus der DTD – Document Type Definition – oder aus dem Schema hervor. Jene Strukturbeschreibungen sind Teil der Schnittstellenspezifikation. Im Falle des DTD's ist sie als Vorspann zu dem Inhalt vorgesehen. Im Falle des Schemas handelt es sich um ein getrenntes Dokument [Mic99].

Bezüglich des Inhalts muss der Tester die Wertbereiche der Datenelemente kennen(Abbildung 7.14). Dazu gehört entweder eine detaillierte Spezifikation der Vorzustände oder genaues Wissen über den erwarteten Dateninhalt. Mit einem einfachen Werkzeug können passende Daten generiert und in die Dokumentenstruktur eingebaut werden. Durch diese Vereinigung von Struktur und Inhalt werden Testdokumente generiert, die an die Komponente unter Test übergeben werden können. So wird die Schnittstelle zwischen Client- und Serverkomponenten simuliert.

Die Testprozeduren bzw. Testskripte der XML-Schnittstellen können wie bei IDL durch Zusicherungsanweisungen gebildet werden (Abbildung 7.15). Für jedes Datenelement gebe es eine Zuweisung von einer Menge diskreter Werte, einem numerischen Wertebereich oder einer Relation zu einem anderen Datenelement im gleichen Dokument. Daraus werden verschiedene stellvertretende Dokumente erzeugt, die als Eingangsnachrichten für die Komponente unter Test dienen können. Die Ausgangsnachrichten der Komponente werden ihrerseits abgefangen und ihre Inhalte gegen die Nachbedingungen der Testskripte geprüft. Auf diese Weise werden falsche Ergebnisse erkannt und ausgewiesen (Abbildung 7.16).

```
<?xml version = ´1.0´?>
<!DOCTYPE Auftrag [
<!--* Kundenauftrag zur Bestellung von Damenwäsche
    *-->
<!ELEMENT Auftrag                (Kundennr,
                                  Auftragsnr,
                                  Datum,
                                  Bestellanzahl,
                                  Bestellung*)>
<!ELEMENT Kundennr               (#PCDATA)>
<!ELEMENT Auftragsnr             (#PCDATA)>
<!ELEMENT Datum                  (Tag, Monat, Jahr)>
<!ELEMENT Tag                    (#PCDATA)>
<!ELEMENT Monat                  (#PCDATA)>
<!ELEMENT Jahr                   (#PCDATA)>
<!ELEMENT Bestellanzahl          (#PCDATA)>
<!ELEMENT Bestellung             (Bestellnr, Artnr, Bestellmenge)>
<!ELEMENT Bestellnr              (#PCDATA)>
<!ELEMENT Artnr                  (#PCDATA)>
<!ELEMENT Bestellmenge           (#PCDATA)>
]>
```

Abbildung 7.15 Stylesheet zum Test einer XML-Schnittstelle

Sowohl die generierten Eingangsdokumente als auch die abgefangenen Ausgangsdokumente lassen sich aufzeichnen, abspeichern und nach Bedarf zurückspielen. Mit XML ist also ein Regressionstest jederzeit einfach wiederholbar. Das größte Problem besteht allerdings in der Verwaltung so vieler verschiedener Dokumente, was ein Versionierungssystem und ein schnelles Wiedergewinnungsverfahren verlangt. Hierfür wird ein Document Management System bzw. eine XML Datenbank benötigt.

7.7 Integrationstest des verteilten Kalenders

Der Integrationstest des verteilten Kalenders findet in zwei Phasen statt. In der ersten Phase werden die Klassen in der Kalenderklassenhierarchie integriert. In der zweiten Phase werden die beiden Komponenten

- Kalender und
- Projekt

integriert.

```
XML.Objekt:: Auftrag
{ assert pre Kundennr                = range (00000001:99999999);
  assert pre Auftragsnr              = 00000000;
  assert pre Datum                   = Tag, Monat, Jahr;
  assert pre Tag                     = range (01:31);
  assert pre Monat                   = range (01:12);
  assert pre Jahr                    = range (00:99);
  assert pre Bestellanzahl           = set (1, 5, 9);
  assert pre Bestellung [1:9]        = Bestellnr, Artnr, Artname, Menge);
  assert pre Bestellnr               = range (1+1);
  assert pre Artnr                   = set (0, 400, 650, 900, 942, 969, 988, 999);
  assert pre Menge                   = range (0+100);
}
```

Abbildung 7.16 XML-Testprozedur

Es ist vom Testplan her vorgesehen, dass die Klassen Top-Down integriert werden (Abbildung 7.17). Also werden zunächst die Klassen `Kalender` und `Woche` getestet. Der Kalender ist eine Aggregation von bis zu 52 Wochen. Es ist hier nicht erforderlich, alle 52 Wochen-Instanzen zu testen, aber zumindest die erste, die letzte und eine dazwischen. Dies wäre ein Beispiel für die Grenzwertanalyse. Als Nächstes wird die Klasse `Tag` hinzugefügt. Jetzt werden für jede stellvertretende Woche die Tage getestet, d.h. pro Woche werden bis zu 7 Objekte des Typs `Tag` generiert. Auch hier wäre es nicht erforderlich, alle 7 Wochentage für jede Woche zu erproben. Es würde genügen, einen Arbeitstag und einen Feiertag zu kreieren. Damit hätten wir zwei repräsentative Instanzen.

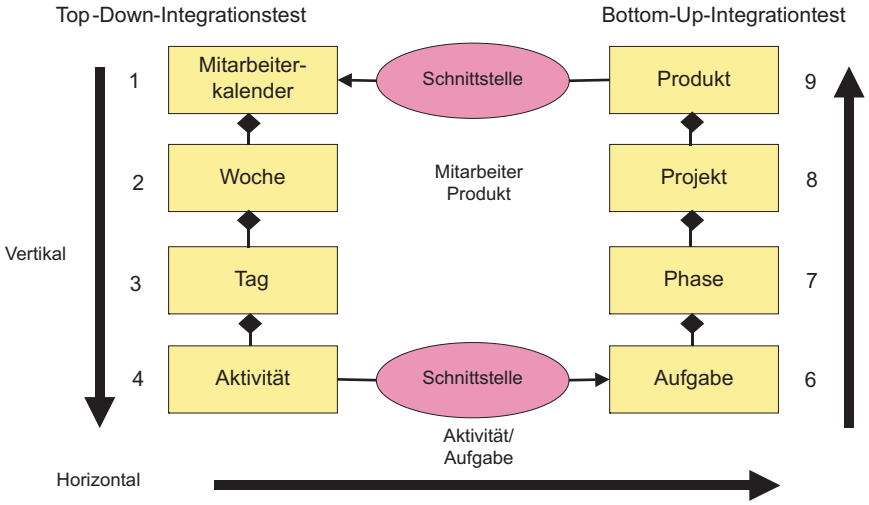


Abbildung 7.17 Teststrategie für den verteilten Kalender

Als letztes wird die Klasse `Aktivität` angehängt. Für jeden Tag werden bis zu der maximalen Grenze von 12 Aktivitäten getestet, und zwar mit Aktivitäten, die gegen Projekte gebucht werden, und Aktivitäten, die nicht gegen Projekte gebucht werden. Da die projektbezogenen Aktivitäten mit der fremden Komponente `Projekte` verbunden sind, muss diese Verbindung über die Schnittstelle `Mitarbeiter_Projekt` abgefangen und simuliert werden, d.h. die Schnittstelle muss die Übergabenachricht prüfen und zumindest einen gültigen Rückgabewert erzeugen. Es wird auch nötig sein, mindestens an einem Tag mehr als die maximal zulässige Anzahl von Aktivitäten zu testen, um die Fehlerbehandlung in diesem Fall auszulösen.

Die IDL-Schnittstelle für die Simulation der Verbindung zur Projektverwaltung könnte wie folgt aussehen. Zu bemerken ist, dass der Operationsaufruf für `aufwand_buchen()` deaktiviert ist.

```
Module.Projekt
{
   interface Aufwand
   {
      attribute string Projekt_KZ;
      attribute long Mitarbeiter_Nr;
      attribute short Stunden;
      attribute int Retcode;
      Assert in Stunden = Range(1:8);
      Assert in Projekt_KZ = Set(´Projekt_1´,
       ´Projekt_2´,´Projekt_3´);
       //Retcode aufwand_buchen(in Projekt_KZ,
       //in Mitarbeiter_Nr,
       //in Stunden),
       //raises („Fehlermeldung“);
     Assert Retcode = Set(0, 1, 2);
  };
};
```

In der zweiten Integrationsphase wird der Operationsaufruf `aufwand_buchen()` in der Schnittstelle `Aufwand` reaktiviert, um die Verbindung zur Komponente `Projekt` zu testen. Dort werden, von den angelegten Aktivitäten aus, die Aufwände tatsächlich gebucht. Dazu muss die Komponente `Projekt` vorher auf dem Server aktiviert werden. Die Nachrichten, die an die Komponente gesendet werden, werden an der CORBA-Schnittstelle zwecks der Verfolgung protokolliert. Es werden auch die veränderten Projektzustände protokolliert. Damit lässt sich kontrollieren, ob die Aufwände tatsächlich gebucht wurden

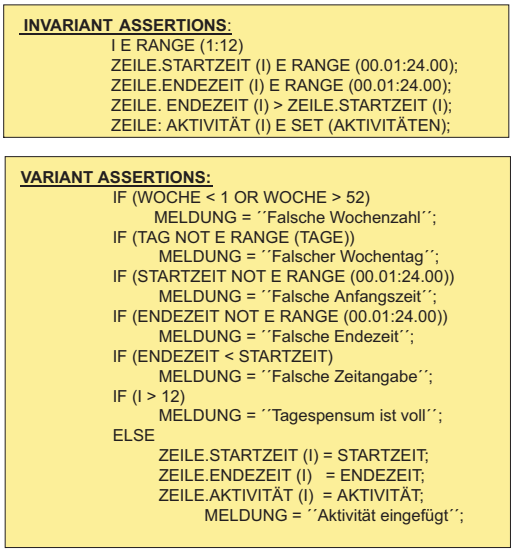


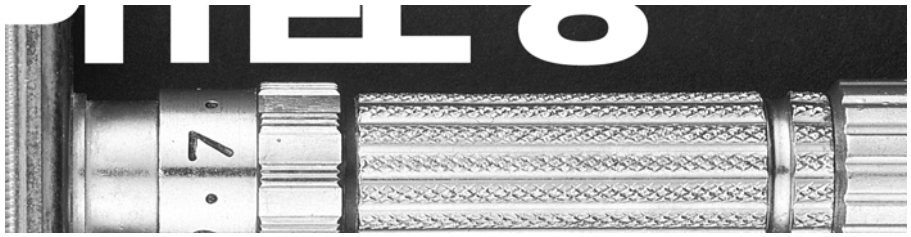
Abbildung 7.18 Zusicherungen für den Test der Kalenderoberfläche

Beim Komponentenintegrationstest wird also gesichert, dass die Interaktion zwischen dem Client `Kalender` und dem Server `Projekt` reibungslos funktioniert. Validiert wird die Interaktion an Hand der protokollierten Aufwandsnachrichten und Projektaufwandszustände (Abbildung 7.18). Wenn der Aufwand für das entsprechende Projekt um die übergebene Stundenzahl erhöht wird und die letzten Projektaufwände zum Schluss gesichert werden, hat die Integration für den Standardfall funktioniert. Der Integrationstest ist aber noch lange nicht beendet.

Es müssen jetzt alle Ausnahmefälle getestet werden, z.B. alle vier Return-Codes vom Projekt sowie die Ausnahmebedingungen vom CORBA ORB. Außerdem muss die Testüberdeckung der Objekte und Operationen in `Kalender` und `Projekt` kontrolliert werden. In der Klasse `Kalender` müssen fast alle Zweige in allen Operationen ausgeführt werden. In `Projekt` müssen nur die Zweige der Operation `aufwand_buchen()` getestet werden. Anhand der Testüberdeckungsprotokolle wird ersichtlich, ob ausreichend getestet wurde. Der Integrationstest ist beendet, wenn alle relevanten Knoten durchlaufen worden sind und die Objektzustände der Wochen, Tage, Aktivitäten, Aufwände und Fehlermeldungen mit den Sollzuständen übereinstimmen.

8

Systemtest



Umgebungstest
Funktionstest
Performanz- und Belastungstest
Testorakel
Systemtest des verteilten Kalenders

Inhaltsübersicht Kapitel 8

8 Systemtest

Ein Software-System ist bekanntlich mehr als nur die Summe aller Einzelteile. Es ist die Summe aller Einzelteile plus des Produkts aller Beziehungen zwischen den Teilen und dem Produkt der Effekte, die durch das Zusammenwirken der Einzelteile auftreten [Sne88].

Dies trifft im besonderen Maße für objektorientierte Systeme zu, weil sie zum einen aus vielen kleinen Einzelteilen bestehen, den Objekten, und zum anderen, weil sie viele Beziehungen zwischen den Einzelteilen haben, nämlich die Interaktionen zwischen Objekten. Wenn das System auch noch verteilt ist, kommen noch mehr Teile und noch mehr Beziehungen dazu. So gesehen ist ein objektorientiertes System relativ zu einem konventionellen System ein sehr komplexes Gebilde mit vielen Abhängigkeiten [Bei94].

Obwohl objektorientierte Systeme um einiges komplexer als konventionelle Systeme sind, ist die Systemtestmethodik ähnlich. Beide werden im Systemtest als schwarze Kästen betrachtet, fertig geschnürte Pakete, die von außen getestet werden. Insofern ist es, was die Testmethodik anbetrifft, gleich, ob es sich um ein prozedurales oder ein objektorientiertes System handelt. Sie werden beide über die externen Schnittstellen getestet, d.h. von der Benutzungsoberfläche bzw. von den Importschnittstellen aus. Der Unterschied liegt in der Menge der Testfälle, die zu bewältigen ist. Diese Menge wird zum Teil von der Objektorientierung geprägt. Darüber hinaus tragen auch die Verteilung der Komponenten und die Gestaltung der Oberflächen dazu bei [Gog93].

Der Systemtest beinhaltet mindestens drei Testarten:

- den Umgebungstest,
- den Funktionstest und
- den Performanz- und Belastungstest [Bei84].

Mit dem Umgebungstest wird erprobt, ob das Anwendungssystem zu der Zielumgebung passt. Mit dem Funktionstest wird nachgewiesen, dass das Anwendungssystem alle spezifizierten Funktionen erfüllt. Im Performanz- und Belastungstest wird geprüft, ob die Anwendung die qualitativen, nicht funktionalen Kriterien ausreichend befriedigt.

8.1 Umgebungstest

Software-Systeme sind in zweierlei Umgebungen eingebettet:

- einer technischen Umgebung und
- einer organisatorischen Umgebung.

Beim Systemtest kommt es darauf an, die Kompatibilität und Interoperabilität der Anwendungssoftware mit den beiden anderen umfassenden Systemen, der Systemumgebung und der Organisationsumgebung zu bestätigen (Abbildung 8.1). Validation bedeutet letztendlich, die Funktionsfähigkeit eines Produkts in einer gegebenen Umgebung unter vordefinierten Umständen nachzuweisen. Es wird also nachgewiesen, dass das Anwendungssystem X in der technischen Umgebung Y für die fachlichen Anforderungen der Organisation Z adäquat funktioniert.

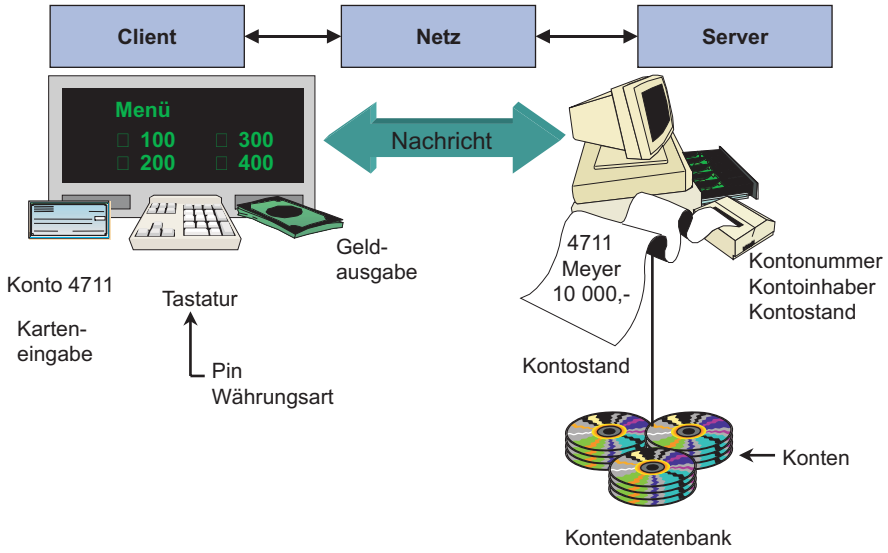


Abbildung 8.1 Umgebungstest

8.1.1 Test der Systemumgebung

Die Systemumgebung umfasst sowohl die Hardware-Konfiguration als auch die Basissoftware und Middleware. Schon was die Hardware anbetrifft, unterscheidet sich der Systemtest moderner, objektorientierter Systeme vom Systemtest konventioneller Hostsysteme. Letzterer beschränkt sich auf einen Rechner mit einem Adressraum. Man muss eine Testdatenbank aufbauen und ein paar Bildschirmgeräte besetzen. Die Hardware-Konfiguration eines verteilten Systems beinhaltet jedoch außer dem Serverrechner mehrere Client-Arbeitsplätze und die dazugehörigen

Netzverbindungen. Im Falle einer Drei-Schichten-Architektur kämen noch ein Hostrechner und mindestens ein Server dazu. Auf alle Fälle gibt es mehrere Adressräume und vielleicht noch mehrere Datenbanken.

Es liegt daher auf der Hand, dass der Test eines verteilten Systems mehr Kapazität bednötigt als der Test eines monolithischen Systems, schon allein wegen der Hardware-Ausstattung. Die Basissoftware und Middleware erhöhen die Komplexität des Systemtests noch weiter. Möglicherweise hat man es mit mehreren Betriebssystemen, Datenbanken und Runtimesystemen zu tun. Die Clientumgebung könnte eine andere Software-Ausstattung haben als die Serverumgebung, und diese hat bestimmt andere Basissoftware als der Host. Die Middleware, sei sie COM, CORBA oder Java, stellt die Verbindung zwischen den Software-Welten her und muss selbstverständlich mit getestet werden [Sne98].

Es geht also beim Test der Systemumgebung in erster Linie darum, die Kompatibilität der Anwendersoftware mit dem Betriebssystem, Datenbanksystem und Kommunikationssystem zu erproben und zu demonstrieren, dass das System als Ganzes mit der jeweiligen Konstellation von Systemsoftware ausreichend abläuft. Dabei spielen Performanz und Sicherheitsüberlegungen eine entscheidende Rolle, d.h. man muss sich erst darüber einig sein, was „ausreichend“ bedeutet.

8.1.2 Test der Organisationsumgebung

Jede betriebliche Organisation hat ihre Eigenart. Es wird von DV-Systemen erwartet, dass sie nahtlos in den laufenden oder geplanten Geschäftsprozess hineinpassen. Betriebe haben ihre Vorschriften und betriebliche Mitarbeiter ihre Arbeitsgewohnheiten. Jedes neue System muss diesen Arbeitsvorschriften und Benutzererwartungen zumindest nahe kommen, um akzeptiert zu werden.

Die Anforderungen der Organisationsumgebung reichen vom Datenschutz und der Datensicherung bis hin zur Schriftauflösung in den Fenstern bzw. Bildschirmmasken. Wenn diese Anforderungen nicht erfüllt sind, wird das Anwendungssystem nicht zum Einsatz kommen, egal welche Funktionalität und technische Qualität es anzubieten hat. Deshalb liegt die Betonung hier auf der Kompatibilität der Anwendungssoftware nicht nur mit der technischen Umgebung, sondern auch mit der betrieblichen Umgebung. Es ist bei der Anforderungsermittlung darauf zu achten und gleich zu Beginn des Systemtests nachzuweisen, dass die Anwendung alle relevanten betrieblichen Vorschriften und Benutzerkriterien erfüllt [ISO12119].

In diesem Zusammenhang sind folgende Kriterien beispielhaft:

- Die Daten sind hinlänglich gegen Verlust oder Verzerrung gesichert.
- Die Daten sind gegen unberechtigte Zugriffe geschützt.
- Die Datenkommunikation ist gesichert.
- Die Benutzungsoberflächen entsprechen den geltenden ergonomischen Normen.

- Die DV-technischen Prozesse passen zu den betrieblichen Geschäftsprozessen.
- Die vorgesehenen Endbenutzer können mit dem System umgehen.

8.2 Funktionstest

Der Funktionstest soll demonstrieren, dass das Anwendungssystem alle vereinbarten fachlichen Funktionen mit einer ausreichenden Qualität ausführt (Abbildung 8.2). Hier beginnt schon das Problem. Man muss davon ausgehen, dass alle vereinbarten Funktionen explizit festgeschrieben sind: entweder im Fachkonzept, bzw. der Systemspezifikation, oder im Benutzerhandbuch. Der Tester braucht sich nur auf das entsprechende Dokument zu beziehen und die dort beschriebenen Funktionen zu testen. Das würde bedeuten, die Gesamtmenge aller Funktionen sei irgendwo definiert.

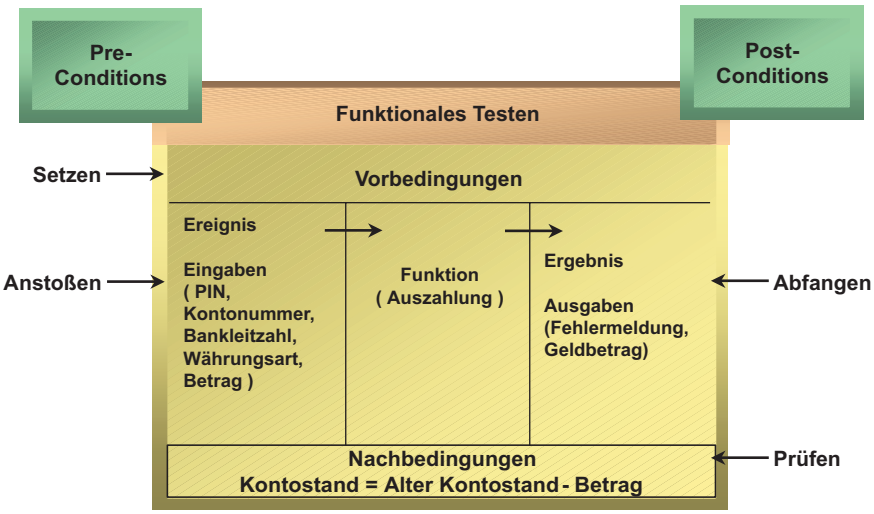


Abbildung 8.2 Funktionstest

Funktionales Testen ist nach Howden ein Black-Box-Test [How86]. Der Funktionstest eines gesamten Systems ist ein Test, bei dem das System in seiner Gesamtheit als geschlossener Kasten betrachtet wird. Es wird von außen mit Eingabedaten bzw. Ereignissen bombardiert und die Ausgaben bzw. Reaktionen werden registriert, um sie gegen die Erwartungen des *Orakels* zu bestätigen. Es klingt einfach: Man brauche nur am Arbeitsplatz zu sitzen und alles einzugeben oder anzuklicken, was einem gerade einfällt. Dies würde auf einen Zufallstest (siehe unten) hinauslaufen, und ein Zufallstest ergibt nur zufällige Ergebnisse. Um effektiv zu sein, muss auch

beim Funktionstest systematisch getestet werden, und dies bedeutet: eine Methode haben. Beizer identifiziert fünf methodische Ansätze zum Funktionstest [Bei95]:

- Datenflusstest,
- Funktionsflusstest,
- Bereichstest,
- Syntaxtest und
- Zustandstest.

Zu diesen konstruktiven Ansätzen kommt noch ein destruktiver hinzu, nämlich der

- Zufallstest.

8.2.1 Datenflusstest

Der Datenflusstest geht von den Systemeingaben und -ausgaben aus (Abbildung 8.3). Die Systemeingaben sind hier die Eingabefelder in der Benutzungsoberfläche, die vom Benutzer gesetzt werden. Die Systemausgaben sind die Ausgabefelder sowie die Berichte. Für jede stellvertretende Ausgabemaske wird eine Kombination von Eingabewerten eingegeben, die zu diesem bestimmten Ergebnis führt. Im Falle von periodischen oder Ad-hoc-Berichten muss jenes Ergebnis produziert werden, das zur Generierung des Berichts führt. Dazu müssen auch stellvertretende Parameterwerte eingegeben werden. Im Prinzip geht dieser Test von der Oberfläche aus und benutzt die Bedienungsanleitung als Grundlage.

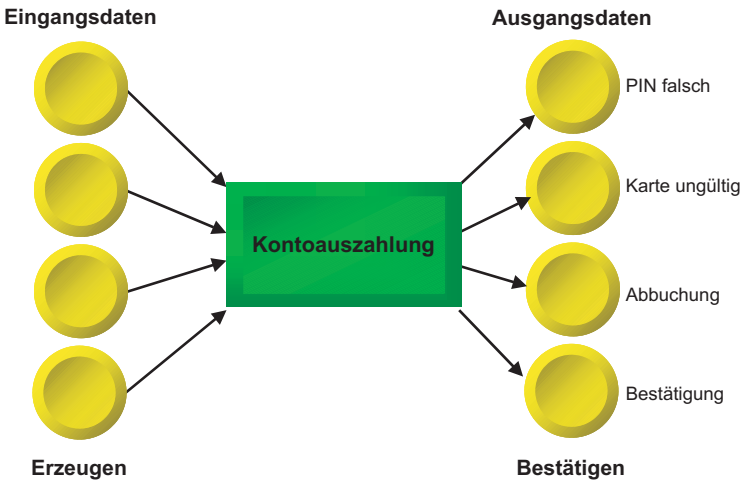


Abbildung 8.3 Datenflusstest

8.2.2 Funktionsflusstest

Der Funktionsflusstest geht von den Anwendungsfällen aus. Für jeden spezifizierten Anwendungsfall wird die Oberflächenausprägung erzeugt, die zur Ausführung des Anwendungsfalls führt. Auf der Ausgabenseite wird die von der Funktion hergestellte Ausgabenmaske abgefangen und bestätigt (Abbildung 8.4). Nach diesem Ansatz gilt es, jeden Anwendungsfall in jeder praktisch relevanten Variation zu testen (vgl. auch Abschnitt 8.2.7 und [Win99]).

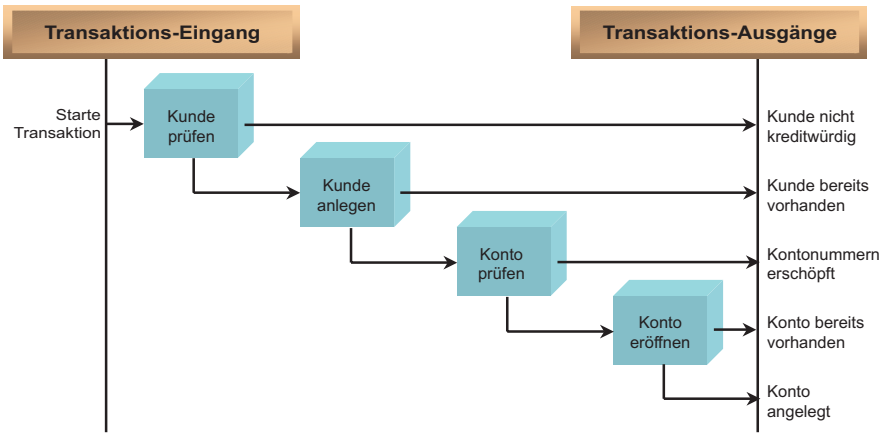


Abbildung 8.4 Funktionsflusstest

8.2.3 Bereichstest

Beim Bereichstest werden an der Eingabeoberfläche Grenzwerte und falsche Werte eingegeben, um die Robustheit und Fehlertoleranz des Systems zu prüfen (Abbildung 8.5). Im Falle numerischer Werte werden die unteren und oberen Grenzwerte erprobt. Im Falle von Texten werden falsche und leere Zeichenfolgen eingegeben. Außerdem werden sowohl zulässige als auch unzulässige Tastenkombinationen und Mausklicks erprobt. Hier wird also bestätigt, ob das System auf alle stellvertretenden Impulse richtig reagiert.

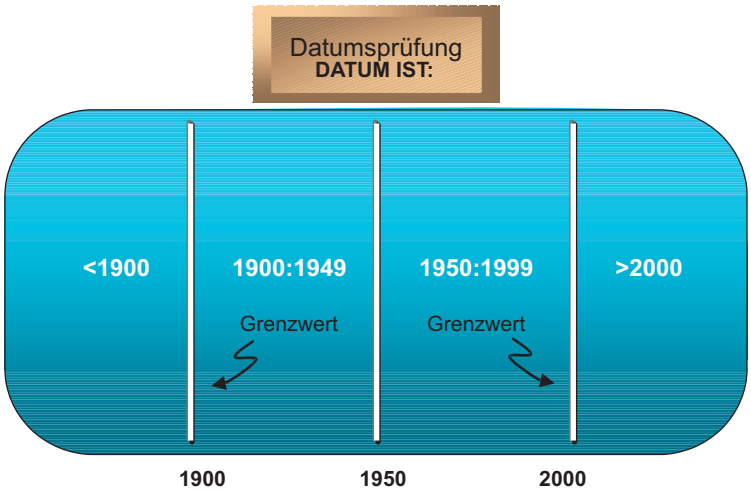


Abbildung 8.5 Bereichstest

8.2.4 Syntaxtest

Im Syntaxtest werden alle Arten von Kommandozeilen erprobt, wobei immer andere Syntaxfehler produziert werden: einmal ein falsches Trennungszeichen, einmal ein fehlendes Wort, einmal eine falsche Wortfolge. Es wird hier also kontrolliert, ob die Syntax richtig interpretiert wird und ob bei Syntaxfehlern die korrekte Fehlermeldung erscheint (Abbildung 8.6).

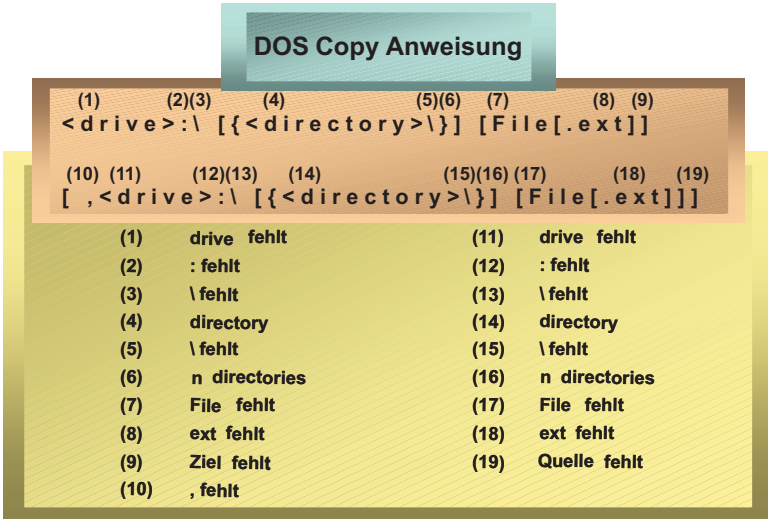


Abbildung 8.6 Syntaxtest

8.2.5 Zustandstest

Der Zustandstest zielt letztlich auf die Erprobung aller spezifizierten Systemzustände. Dazu werden die UML-Zustandsdiagramme verwendet, um die Testfälle an der Oberfläche zu definieren und den Endzustand an der Ausgabeoberfläche zu validieren. Es empfiehlt sich, eine Tabelle sämtlicher anwendungsrelevanter Zustände zu bilden und sie als Basis für die Verifikation der Objektzustände zu benutzen (Abbildung 8.7).

Kontozustandstabelle

ZUSTAND	nicht existent	normal	überzogen	gesperrt	gelöscht
nicht existent		X			
normal			X		X
überzogen		X		X	
gesperrt		X			X
gelöscht	X				

Abbildung 8.7 Zustandstest

8.2.6 Zufallstest

Zu den oben genannten konstruktiven Testansätzen kommt noch ein weiterer, destruktiver dazu. Es soll z.B. versucht werden, das Netz auszuschalten, die Speicherkapazität auszulasten und die Leitung zu überlasten. Diese Tests dienen dazu, die Robustheit des Systems auf die Probe zu stellen (Abbildung 8.8). Das Ausmaß dieses Tests hängt von den Sicherheitsanforderungen ab. Ein hoher Grad an Sicherheit erfordert einen langwierigen Test aller Abbruchmöglichkeiten.

8.2.7 Funktionstest mit Anwendungsfällen

Die oben genannten sechs funktionalen Testansätze gelten gleichermaßen für alle Software-Systeme, unabhängig davon, ob sie objektorientiert, prozedural oder funktionsorientiert sind. Der Funktionstest für objektorientierte Anwendungen sollte zusätzlich dazu ein Test der Anwendungsfälle sein. Schon 1995 hat Ivar Jacobson vorgeschlagen, die „Use Cases“ als Basis für den Test objektorientierter Systeme heranzuziehen [Jac95]. Er wies darauf hin, dass Anwendungsfälle ein idealer Ausgangspunkt für die funktionalen Testfälle sind. Man müsse nur jeden Anwendungs-

fall in seine verschiedenen Ausprägungsarten zerlegen und für jede Ausprägung einen Testfall definieren. Einer der Autoren hat den ursprünglichen Ansatz von Jacobson ergänzt und zu einer systematischen Testmethodik ausgebaut [Win99].

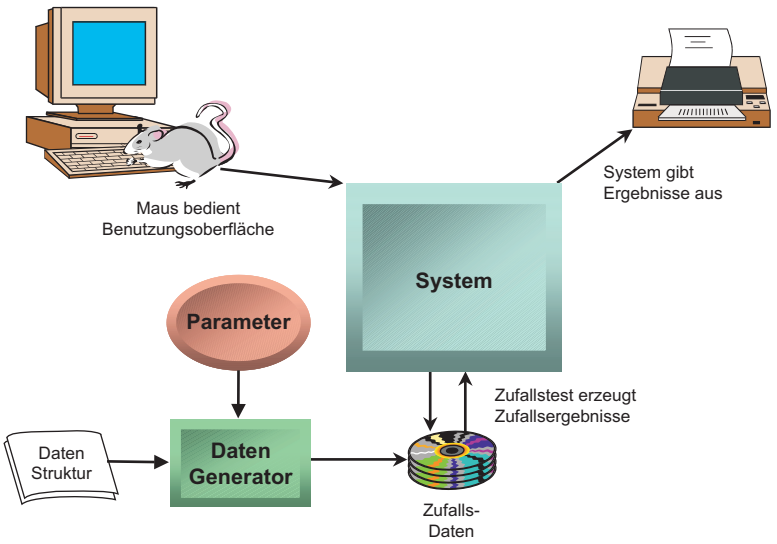


Abbildung 8.8 Zufallstest

Das Ziel des anwendungsfallbasierten Systemtests ist es, bei der Ausführung möglicher Ereignisflüsse durch die ausgewählten Anwendungsfälle Fehler des Anwendungssystems zu finden. Hierzu gehören die Angabe der Szenarien und der konkreten Eingabedaten sowie erwarteten Ausgaben, mit denen das Anwendungssystem gegen das Anwendungsfallmodell getestet wird.

Der Nutzen des anwendungsfallbasierten Systemtests besteht darin, zu zeigen, dass das Anwendungssystem bereit bzw. geeignet für den Einsatz in der Produktion ist ([JBR99] [Win99] [Win01]). Er wird in der Regel als Abnahmetest unter der Regie des Auftraggebers unter Einbeziehung der Benutzer durchgeführt und soll diese davon überzeugen, dass die Anwendung der Anforderungsspezifikation bzw. dem „Pflichtenheft“ entspricht. Hierzu wird das System systematisch gegen alle wesentlichen Ereignisflüsse der Anwendungsfälle mit sinnvollen Anfangszuständen und Eingaben getestet, wobei folgende Fehlertypen gefunden werden können:

- Unvollständig implementierte Anwendungsfälle.
- Fehlerhaft implementierte Geschäftslogik.
- Nicht vom Anwendungssystem beachtete Abhängigkeiten zwischen Anwendungsfällen.

Die Voraussetzungen für den anwendungsfallbasierten Systemtest sind ein vollständiges Anwendungsfallmodell, welches für jeden Anwendungsfall die Spezifika-

tion seiner Funktionalität mit Vor- und Nachbedingungen und Szenarien, die Beschreibung seiner elementaren Aktionen und der diese ausführenden Akteure sowie die Beschreibung seiner Dynamik mit Hilfe eines die möglichen Reihenfolgen dieser Aktionen beschreibenden Aktivitätsdiagramms beinhaltet. Das Ergebnis des anwendungsfallbasierten Systemtests ist das gegen die Anwendungsfälle getestete System.

Im ersten Schritt des anwendungsfallbasierten Systemtests wählt man einen bestimmten Anwendungsfall aus, für den Testfälle abgeleitet werden sollen, sowie das zugehörige Aktivitätsdiagramm und ggf. die Aktivitätsdiagramme von Anwendungsfällen, zu denen eine `include`- oder `extend`-Beziehung ([OMG99]) besteht (Abbildung 8.9).

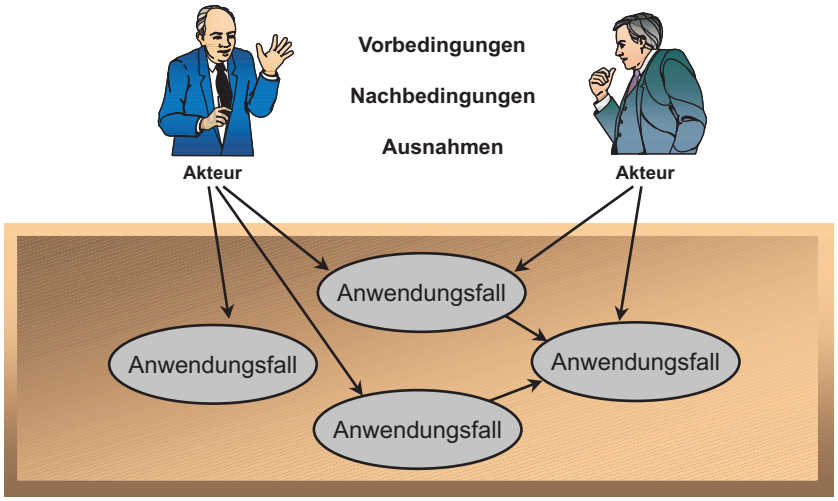


Abbildung 8.9 Anwendungsfallbasierter Systemtest

Die möglichen Ereignisflüsse des Anwendungsfalls (Testszenarien), zusammengefasst im Aktivitätsdiagramm, werden zu den Testfällen für den anwendungsfallbasierten Systemtest. Jedes Testszenario wird zu einem Testskript. Insbesondere die extern beobachtbaren Abläufe und Interaktionen der Anwender mit dem Anwendungssystem werden durch die Vor- und Nachbedingungen der Aktionen und die Übergangsbedingungen der Kanten im Aktivitätsdiagramm betrachtet. Testskripte können (anhand des hierarchischen Aufbaus der Aktivitätsdiagramme) hierarchisch strukturiert werden.

Bei den anwendungsfallbasierten, ablauforientierten Systemtests wählt man die Testszenarien so aus, dass die geforderte Überdeckung des Aktivitätsdiagramms erzielt wird. Hierbei versucht man, die Anzahl der zu testenden Szenarien bezüglich der Überdeckung aller Knoten (Aktionen) und Kanten (Übergänge) im Aktivitätsdiagramm zu minimieren. Wie im strukturellen (White Box) Programmtest kann

man die Überdeckungsmetriken auch zur Generierung weiterer Testszenarien verwenden.

Die Grundausprägungen der Anwendungsfälle sind die Testfälle für den Funktionstest. Jetzt müsste der Tester die Eingaben durch die Wertebereiche bzw. die Vorzustände der Eingabe variabel ergänzen. Es hieße z.B. nicht nur „Artikelnummer", sondern „Artikelnummer mit folgenden repräsentativen Werten: 4711, 4712, 4713" usw. Auf der Ausgabeseite hat der Tester die zu erwartenden Ergebnisse zu spezifizieren, z.B.:

```
If letzte Artikelmenge = 100
      & Bestellmenge = 5
      & Return Code = ok
   then neue Artikelmenge = 95;
```

So werden für alle relevanten Eingaben oder Ereignisse die entsprechenden Sollergebnisse festgehalten.

In einem Testskript stellt man Testfälle zu einem Szenario anhand der Vor- und Nachbedingungen der Aktionen und der Flussbedingungen der Kanten im Aktivitätsdiagramm zusammen. Man unterscheidet Konformitätstests und Robustheitstests:

- Konformitätstests halten die Bedingungen explizit ein.
- Gezielte Verletzungen der Bedingungen der Aktionen und Kanten im Aktivitätsdiagramm und der Interaktionsparameter führen zu Robustheitstests, mit denen man die Robustheit des Anwendungssystems prüft.

Endekriterien für den anwendungsfallbasierten Systemtest sind:

- Die Eingabedaten zu jeder Interaktion sind festgelegt.
- Die Werte der Vorbedingung des Anwendungsfalls, d.h. des Zustands des Anwendungssystems zu Beginn des Tests, sind spezifiziert.
- Die erwarteten Werte der Nachbedingungen, d.h. des Zustands des Anwendungssystems nach dem Test, sind spezifiziert.
- Alle Aktionen in den Aktivitätsdiagrammen sind durch Tests überdeckt.
- Alle Kanten in den Aktivitätsdiagrammen sind durch Tests überdeckt.

Insofern als die Spezifikation der Anwendungsfälle formalisiert ist, ließe sie sich gleich automatisch per Tool in ein Testskript umsetzen. Sonst kann sie nur als Vorlage für einen Tester dienen, der sie entweder manuell in ein Testskript übersetzt oder direkt am Bildschirm in eine Testtransaktion umsetzt. Natürlich ist ein Testskript zu empfehlen, weil es die Wiederholbarkeit und Wiederverwendbarkeit des Tests fördert.

8.2.8 Modellbasierter Funktionstest

Eine besondere Art des Funktionstests ist der modellbasierte Test bzw. „specification based Testing“. Für diesen Test wird das Objektmodell herangezogen. In einem Artikel aus dem Jahr 1994 mit dem Titel „Automated Testing from Object Models“ beschreibt Robert Poston den automatisierten Test gegen ein OMT-Modell [Pos94]. Ausgangspunkt ist das Klassendiagramm zusammen mit dem Sequenzdiagramm und dem Zustandsübergangsdiagramm. Aus den Klassendiagrammen werden alle Operationen des Systems abgeleitet. Aus den Sequenzdiagrammen werden die Ausführungsfolgen der Operationen gewonnen. Eine Ausführungsfolge ist äquivalent zu einem Testszenario. Von den Zustandsdiagrammen folgen die Vor- und Nachbedingungen. Daraus ergeben sich die alternativen Operationsergebnisse. Jeder Nachzustand ist das Resultat eines Testfalls. Mit Hilfe eines Werkzeugs hat Poston die Testfälle aus der Kombination der oben genannten OMT-Diagramme generiert (Abbildung 8.10, vgl. auch Abschnitt 11.3.3.3).

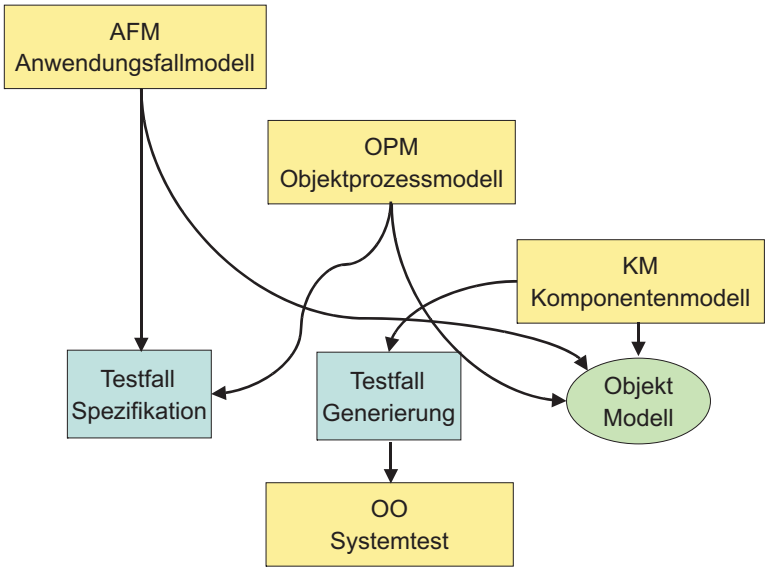


Abbildung 8.10 Modellbasierter Systemtest

Jeder generierte Testfall identifiziert die von ihm betroffenen Objekte und beschreibt den Zustand ihrer Instanzvariablen vor der Testfallausführung. Dies ist der Vorzustand oder *Precondition* der Objekte. Dann werden die Eingabeparameter spezifiziert, um diesen Testfall auszulösen. Anschließend wird die Operationsfolge festgehalten. Dies ist das vorausgesagte Funktionsprofil, aus dem hervorgeht, welche Funktion in welcher Reihenfolge ausgeführt wird. Abgeschlossen wird die Testfallspezifikation mit einer Beschreibung der Objektendzustände, d.h. die neuen

Werte der Mitgliedsvariablen und die erwarteten Ausgangsparameter. Die letzteren sind die Ergebnisse des Testfalls.

Das Besondere an dem Ansatz von Poston ist, dass die Testfälle alle automatisch aus den OMT-Diagrammen generiert werden. Sie sind deshalb nur so vollständig und genau, wie das Klassenmodell selbst. Wenn das Klassenmodell Lücken hat, oder wenn es sehr ungenau formuliert ist, werden auch die daraus abgeleiteten Testfälle Lücken und Ungenauigkeiten aufweisen. Um zu brauchbaren Testfällen zu kommen, müsste man zurückgehen und das Klassenmodell vertiefen. Im Endeffekt müsste das Klassenmodell den gleichen Vertiefungsgrad wie die Programme selbst aufweisen. So verlangen es auch Bashir und Goel in ihrem Buch "Testing object-oriented Software" [BaGo99].

8.3 Performanz- und Belastungstest

Der Performanz- und Belastungstest zielt darauf ab, das Verhalten des Systems bei einer großen Anzahl von Transaktionen bzw. bei großen Datenmengen zu messen. Dieser Test gewinnt durch den eBusiness-Hype zunehmend im Bereich von Client/Server-Systemen an Bedeutung. Der Belastungstest ist allerdings auch für Host-Anwendungen wichtig, um deren Leistungsfähigkeit unter Belastung und bei Lastspitzen in Stressbereichen zu überprüfen.

Beim Performanztest werden viele Clients simuliert, die gleichzeitig das System frequentieren. Hierbei soll festgestellt werden, ob die Software und die technische Ausstattung der Hardware den Belastungen der simulierten Clients gewachsen sind. Beim Belastungstest für Client/Server-Anwendungen wird überwiegend geprüft, welches Datenvolumen pro Zeiteinheit ein Server ohne Leistungseinbußen verkraften kann.

Voraussetzung zur Durchführung des Performanz- und Belastungstests ist, dass die Teststufen Klassen-, und Integrationstest sowie der Funktionstest erfolgreich abgeschlossen sind. Allerdings kann die Durchführung von Belastungstests – z.B. mit entsprechenden Prototypen – auch schon zu früheren Zeitpunkten sinnvoll sein, wenn z.B. die Machbarkeit von Anforderungen untersucht werden soll.

In einer objektorientierten Anwendung kann z.B. die Menge der gleichzeitig im Hauptspeicher aktiven Objekte sehr groß werden. Man muss nur an einen Stücklistenprozessor mit Tausenden Einzelteilen denken, um das Ausmaß dieses Problems zu begreifen. Tausende oder gar Hunderttausende Objekte werden zu einem Zeitpunkt aktiv im Rechner sein. Diese Objekte werden auch miteinander millionenfache Nachrichten austauschen. Sogar moderne Rechner werden an die Grenze ihrer Leistungsfähigkeit gebracht. Es kann sehr leicht zu einem Speicherüberlauf und zum Verlust von Objekten kommen, so genannte Speicherlöcher (memory leaks) [Win93].

Es kann ebenso leicht zu Engpässen in der Datenkommunikation kommen, vor allem, wenn die Objekte verteilt sind. Deshalb ist gerade bei objektorientierten Systemen ein wohldurchdachter Massentest mit einer großen Objektmenge und einer hohen Transaktionsrate unverzichtbar. Für den Belastungstest muss die Systemeingabe beliebig reproduzierbar sein, sodass eine große Menge an Transaktionen hergestellt werden kann. Dazu braucht man entsprechende Werkzeuge, ohne die ein derartiger Massentest nicht möglich ist [Bou97].

Testfälle für den Belastungstest prüfen nicht die fachlich korrekte Verarbeitung, sondern das Verhalten des Testobjekts in Situationen außergewöhnlich hoher Belastungen. Diese Situationen müssen in den nicht-funktionalen Anforderungen exakt beschrieben werden, z.B. folgendermaßen:

> „*n* gleichzeitig mit dem Server arbeitende Client-Rechner aktivieren *m* Transaktionen vom Typ *T* in einem bestimmten Zeitraum t_1-t_2“, wobei die mittlere Antwortzeit t_A einzuhalten ist und die maximale Antwortzeit t_M nicht überschritten werden darf“.

Bei der Testfallermittlung für den Performanz- und Belastungstest sind folgende Punkte zu betrachten:

- Beim Performanztest muss neben der maximalen Anzahl von Client-Rechnern auch eine bestimmte Anzahl unterschiedlicher Anwendungen ausgeführt werden, da diese das Antwortzeitverhalten unterschiedlich beeinflussen können.
- Das Antwortzeitverhalten bei der höchsten zu erwartenden Anzahl gleichzeitig das System frequentierender Client-Rechner ist zu beobachten.
- Das Verhalten der Server bei stufenweise ansteigender sowie der höchsten anzunehmenden Systembelastung ist zu beobachten.

Die Testdaten im Belastungstest setzen sich zusammen aus:

- Benutzereingaben und
- persistenten Daten.

Persistente Daten lassen sich wiederum unterteilen in

- Systemspezifische Daten und
- Systemübergreifende Daten.

Sowohl systemspezifische Daten (zum Beispiel die auszuwertende Umsatzdatei) als auch systemübergreifende Daten (zum Beispiel Personen- und Artikeldaten) sind in der Regel in einem neutralen Format verfügbar und bei Bedarf ladbar und updatefähig zu halten. In Abhängigkeit von den für den Belastungstest vorgesehenen Testfällen können u.U. sehr große Datenmengen erforderlich werden, sodass ggf. Werkzeuge zu ihrer Generierung notwendig werden.

Die Testprozeduren für den Belastungstest können auf den Testprozeduren des Integrations- und Funktionstests aufbauen, wenn es sich um die Testgegenstände handelt, die im Integrationstest auf ihr Zusammenspiel, im Funktionstest auf ihre

fachliche Korrektheit und im Belastungstest auf ihre Belastbarkeit geprüft werden. Testprozeduren im Belastungstest enthalten zusätzliche Ablaufanweisungen, die die geforderte Belastung erzeugen. Bei Systemen mit graphisch-interaktiver Benutzungsoberfläche können dazu z.B. mit Capture-Replay-Werkzeugen Mitschnitte aufgezeichnet und entsprechend der Testfallspezifikation mehrfach aktiviert werden.

Die Testumgebung für den Belastungstest enthält zum einen die Testdaten z.B. in Form von Dateien und Datenbanken. Bei der Testausführung werden die Testdaten immer wieder aus der Testvorbereitungsumgebung in die Ausführungsumgebung geladen. Alle Testgegenstände können so in beliebiger Reihenfolge immer in der gleichen Umgebung getestet werden („paralleles“ Testen).

Zudem gehören projektspezifische Elemente zur Testumgebung. Umgebungselemente wie z.B. zentrale Ressourcen und benachbarte Systeme können von mehreren Projekten in der jeweiligen Testumgebung genutzt werden. Durch die zentrale Bereitstellung stehen sie allen projektspezifischen Testumgebungen zentral zur Verfügung. Alle haben gleichzeitig Zugriff auf die Standardressourcen. Darüber hinaus sind technische Vorkehrungen in der Testumgebung zu treffen, die es erlauben, die im Test zu prüfenden Belastungssituationen zu simulieren.

Zu den Testendekriterien des Performanz- und Belastungstests können zählen, dass die vorher bestimmten Belastungsgrößen simuliert wurden, bei der Testausführung die Performanzanforderungen eingehalten wurden und das Anwendungssystem unter Belastung keine unerwarteten Reaktionen (Stillstand, Absturz, ...) aufzeigte.

8.4 Testorakel

Ein reales System wird immer gegen ein Idealbild des gleichen Systems getestet. Entweder existiert dieses Idealbild in den Köpfen der Tester oder es ist irgendwo dokumentiert. Bill Howden spricht von einem Systemorakel [How85]. Funktionales Testen setzt immer ein Orakel voraus, denn in einem funktionalen Test muss für jede Funktion die erwartete Ausgabe für jede mögliche Eingabe vorausgesagt sein. In Howdens Worten:

> *“Functional Testing assumes the existence of an input/output oracle, which for any input x for f and output y = f(x), can determine if y = f*(x). If a function contains a fault, it will produce the wrong output for some input.”* ***[How85]***

Daraus folgt, dass es nicht genügt, die Funktionen zu identifizieren, es müsste auch für jede mögliche Eingabe zu einer Funktion die zu erwartende Ausgabe festgelegt sein.

Robert Poston geht einen Schritt weiter und beschreibt den Test gegen ein Orakel in drei Stufen:

- die Ableitung der Testfälle aus dem Orakel,

- die Ausführung der Testfälle und
- die Auswertung der Testfälle gegen das Orakel [Pos96a].

Die Kernfrage ist: Woraus werden die Testfälle abgeleitet? Was ist das Orakel für einen Systemtest? Es kommen vier Kandidaten in Frage (Abbildung 8.11):

- die Benutzerdokumentation
- das Fachkonzept
- das Objektmodell
- das Nutzungsprofil

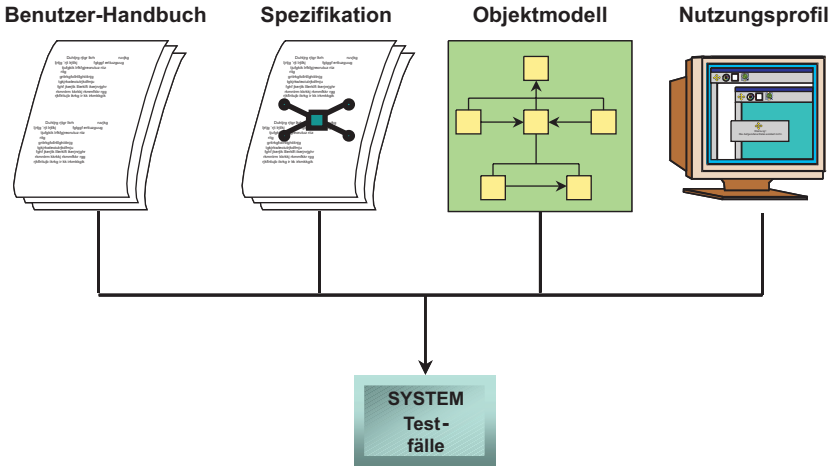


Abbildung 8.11 Orakel für den Systemtest

8.4.1 Test gegen die Benutzerdokumentation

Ein mögliches Orakel ist die Benutzerdokumentation. Darin wird beschrieben, wie das System zu bedienen ist und was es zu leisten hat. Wenn diese Dokumentation einigermaßen vollständig und gut strukturiert ist, so z.B. nach ANSI/IEEE-Norm 1063 [IEEE1063], können daraus Testfälle abgeleitet werden, und zwar nicht nur von professionellen Software-Testern, sondern von den Endbenutzern oder ihren Vertretern.

In der Benutzerdokumentation sind die Bildschirmoberflächen beschrieben. Normalerweise werden die Oberfläche abgebildet und darunter die Nutzungsmöglichkeiten aufgelistet. Alle diversen Eingabemöglichkeiten werden erwähnt, z.B. die Texteingabe, die Menüauswahl und die Kontrolltasten. Wer eine solche Beschreibung vor sich hat, muss in der Lage sein, alles auszuprobieren (Abbildung 8.12). Außerdem müssten alle möglichen Fehlermeldungen angegeben werden und wie sie zustande kommen. Ergo obliegt es dem Systemtester zu versuchen, all jene Zustände zu

produzieren, die diese Meldungen auslösen. Er geht einfach durch die Oberfläche und probiert die Anwendungsfälle der Reihe nach aus. Besser wäre es jedoch, durch die Benutzerdokumentation zu blättern und die Testfälle festzuschreiben, z.B. in Testfalltabellen.

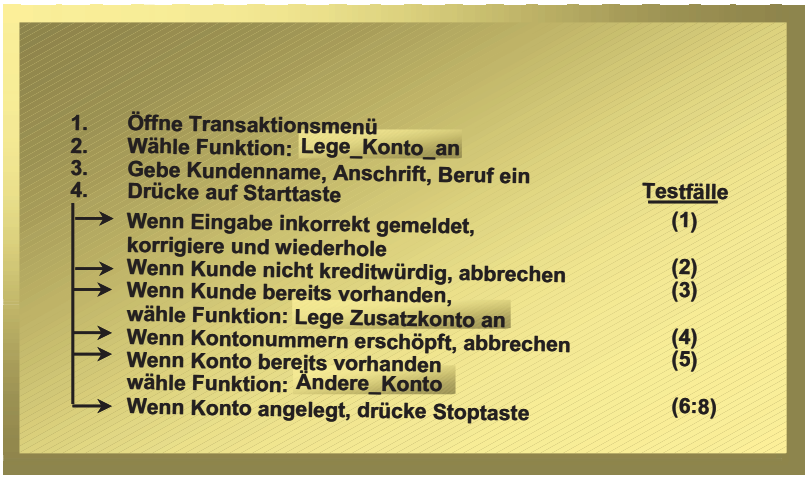


Abbildung 8.12 Test gegen die Benutzerdokumentation

Bei gut dokumentierten Dialogschnittstellen sind die Testfälle relativ einfach zu erkennen. Anders sieht es bei den Batchschnittstellen aus. Sie enthalten in der Regel viel mehr Daten und sind nicht so genau beschrieben. Es empfiehlt sich hier, Testdatentabellen zu verwenden, in der die Spalten die Felder und die Zeilen die Sätze sind. Jedes Feld in jedem Satz hat einen Wert, den der Tester zuweist und variiert, je nach Auswirkung. Dabei kommt es auf die Zuweisung der bereits vorgestellten Testdatentypen an, z.B. Grenzwerte, Falschwerte und repräsentative Werte.

Zur Benutzerdokumentation gehört das Datenmodell mit der Beschreibung der Datenbanken und ihre Inhalte. Auch hierfür ist die Testdatentabelle das geeignete Mittel, um die Testdaten zu erfassen. Die Attribute der Tabellen werden mit repräsentativen Werten versorgt. Die Primärschlüssel werden mit den Fremdschlüsseln und die indizierten Attribute mit den Suchbegriffen abgestimmt, damit stellvertretende Zugriffspfade vorkommen. Besser ist es hier, wenn man auf bereits bestehende Datenbanken zurückgreifen kann. In diesem Fall müssen sie nur umformatiert und angepasst werden.

Der Test von der Oberfläche aus kann auf zweierlei Weise vor sich gehen. Erstens kann der Tester jedesmal die Testfälle manuell in Mausklicks oder Tasteneingaben umsetzen. Dies ist zwar beim ersten Mal bequemer und schneller, aber bei jeder Wiederholung des Tests aufwändiger und fehleranfälliger. Immer wieder werden die gleichen Vorgänge wiederholt. Die visuelle Ergebniskontrolle lässt nach, und

Fehler bleiben unbemerkt. Der Tester wird auch versuchen, Testschritte wegzulassen, umso schneller durchzukommen. Demzufolge ist die Automatisierung als zweite Möglichkeit besser. Sie ist zwar am Anfang etwas aufwändiger und unbequemer, aber je länger der Test dauert, umso effektiver und billiger. Statt das System direkt zu testen, wird der Test an einer Prototypoberfläche für den späteren echten Systemtest aufgezeichnet. Der Tester geht durch alle Anwendungsfälle und betätigt die Eingaben. Gleichzeitig registriert er die dazu passenden Ergebnisse. Auch sie werden aufgezeichnet. Nach dem Prototyptest können die Oberflächendaten und deren Reihenfolge immer noch angepasst werden. Wenn der Prototyptest vollständig ist, kann er anschließend für den echten Test wiederholt eingesetzt werden.
Maßgeblich in diesem Zusammenhang ist, dass die Testfälle – sowohl die Eingaben als auch die erwarteten Ausgaben – aus der Benutzerdokumentation abgeleitet werden, denn nur so lässt sich das System gegen das Benutzerhandbuch testen [PePa98].

8.4.2 Test gegen das Fachkonzept

Ein zweites potenzielles Testorakel ist das Fachkonzept. Fachkonzepte sind funktionale Beschreibungen einer Anwendung aus der Sicht der Anwender. Laut ANSI/IEEE-Standard 830 für *Requirement Specification* [IEEE830] umfasst ein Fachkonzept

- die Funktionen,
- die Eingaben,
- die Ausgaben,
- die Schnittstellen,
- die Einschränkungen und
- die Ausnahmebehandlungen.

In der Praxis beinhalten Fachkonzepte oft weniger und manchmal mehr. Um als Testorakel dienen zu können, muss das Fachkonzept eine ausreichende Tiefe haben. So müssen alle wesentlichen Funktionen zusammen mit ihren Argumenten und Ergebnissen erfasst sein. Die Funktionen müssen auch mit einer externen Schnittstelle, z.B. mit einer Benutzungsoberfläche, einer Importdatei oder einer Nachricht von außen verbunden sein, denn ohne diese Verbindung ist eine Testspezifikation nicht möglich, da die Testfälle den Schnittstellen zugeordnet sind.
Für den Test gegen das Fachkonzept muss der Tester von den spezifizierten, elementaren Funktionen ausgehen. Eine elementare Funktion empfängt Eingaben und erzeugt Ausgaben. Der Tester muss diese in eine Testskriptsprache umsetzen. Jede Funktion wird dadurch zu einer elementaren Testprozedur. Die Menge aller elemen-

taren Testprozeduren für eine Schnittstelle bilden eine Haupttestprozedur. Die Logik der Haupttestprozedur bestimmt die Reihenfolgen, in denen die Funktionen ausgeführt werden können. Die Logik der elementaren Testprozeduren bestimmt die Testdaten und die erwarteten Ergebnisse. Die Ausnahmebedingungen werden wie elementare Funktionen behandelt. Die Regeln werden berücksichtigt, um die erwarteten Ergebnisse vorauszusagen. Somit fließen fast alle Beschreibungselemente der Funktionsspezifikation in die Testspezifikation ein (Abbildung 8.13). Was aber in der Spezifikation fehlt, wird nicht getestet. Insofern ist hier der Funktionstest nur so gut wie die Funktionsbeschreibung selbst.

Entscheidungstabelle

Bedingungen	**Regeln**								
Name, Anschrift oder Beruf ungültig	J	N	N	N	N	N	N	N	N
Kunde vorbestraft		J	N	N	N	N	N	N	N
Kunde vorhanden			J	N	N	N	N	N	N
Kontonummer erschöpft				J	N	N	N	N	N
Konto vorhanden					J	N	N	N	N
Aktionen	**Testfälle**								
Melde Eingabefeld	*								
Melde nicht kreditwürdig		*							
Melde Kunde vorhanden			*						
Melde keine Kundennummer				*					
Melde Konto vorhanden					*				
Eröffne Konto						*	*	*	*
Update Kontostatistik						*	*	*	*
Bestätige Anlegung						*	*	*	*

Abbildung 8.13 Test gegen das Fachkonzept

Ohne Werkzeugunterstützung ist der Test gegen das Fachkonzept bei kleinen, überschaubaren Anwendungen mit einer begrenzten Anzahl Funktionen noch möglich. Schon ab ca. 100 Funktionen erfordert diese Testart jedoch eine automatische Umsetzung der Funktionsspezifikation in Testskriptprozeduren. Dies bedeutet wiederum, dass die Funktionen formal spezifiziert werden. Ohne in einer normierten Grammatik wie Z, VDL oder OCL formuliert zu sein, sind die Funktionsspezifikationen für ein Tool nicht interpretierbar [DeOf91]. Voraussetzung für den Test gegen das Fachkonzept ist also zumindest eine halbformale Funktionsspezifikation. Falls diese Voraussetzung nicht erfüllbar ist, scheidet diese Alternative für die Praxis aus.

8.4.3 Test gegen die objektorientierte Spezifikation

Das dritte potenzielle Testorakel ist die objektorientierte Spezifikation als Beschreibung der technischen Lösung. Die UML-Diagramme, aus denen sich die objektorientierte Spezifikation zusammensetzt, sind Anforderungs- und Entwurfsdokumente (Abbildung 8.14). Für den Systemtest sind folgende Diagrammtypen von besonderem Interesse [Pos94]:

- Anwendungsfalldiagramm,
- Aktivitätsdiagramm
- Sequenzdiagramm und
- Zustandsdiagramm.

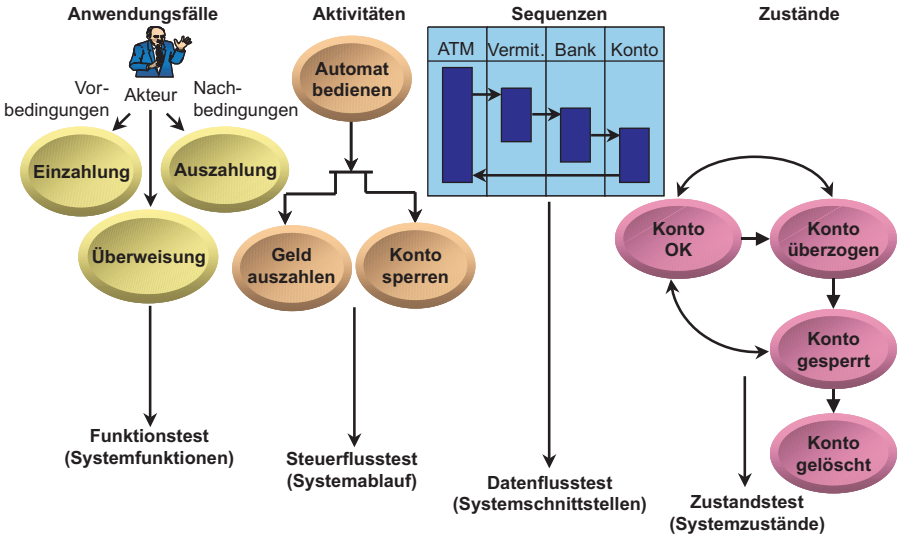


Abbildung 8.14 Test gegen das Ojektmodell

Bis auf das Anwendungsfalldiagramm sind diese Diagramme alle Verhaltensdiagramme, die das dynamische Verhalten der Anwendung beschreiben. Die Anwendungsfalldiagramme beschreiben die Interaktion zwischen dem Endbenutzer und dem System. Sie enthalten zu wenig Information, um direkt in Testfälle umgesetzt zu werden, aber sie können sehr wohl als Grundlage für die manuelle Erstellung von Testfällen dienen. Der Tester muss die einzelnen Fälle um die Eingangs- und Ausgangswerte ergänzen und die Beziehungen zwischen diesen Werten in Form der Object Constraint Language formulieren, d.h. Anwendungsfälle im Zusammenhang mit OCL. Diese Kombination, richtig angewandt, kann zu automatisch interpretierbaren Systemtestprozeduren führen.

Aktivitätsdiagramme beschreiben die Abläufe bei der Nutzung eines Systems und deren gegenseitige Abhängigkeiten. Jeder Pfad durch ein Aktivitätsdiagramm entspricht einem Systemtestfall. Es ist daher durchaus möglich, Testfälle aus den Aktivitätsdiagrammen automatisch abzuleiten, allerdings müssen sie mit Daten ergänzt werden ([Win99] [Win01]).

Das Gleiche gilt für die Sequenzdiagramme. Ein Sequenzdiagramm beschreibt die Interaktion zwischen Objekten bei der Abarbeitung eines Vorgangs. Das Ziel hier ist, alle skizzierten Interaktionen zu durchlaufen. Die Testdaten müssen wiederum aus den Klassendiagrammen entnommen werden. Jeder Pfad durch ein Sequenzdiagramm wird durch eine bestimmte Datenkonstellation ausgelöst. Im Testfall muss diese Konstellation exakt spezifiziert werden. Hierfür ist die Objekt Constraint Sprache der OMG bestens geeignet.

Schließlich ist das Zustandsdiagramm nützlich, um die Vor- und Nachzustände der Objekte für den Test zu spezifizieren. Ein Systemtestfall soll den Anfangs- und den Endzustand aller betroffenen Objekte spezifizieren, damit die Testprozedur den Anfangszustand herstellen und den Endzustand prüfen kann. Diese Information liefert zumindest teilweise das Zustandsdiagramm [Bin96a].

Es wird daher nicht möglich sein, gegen einen Diagrammtyp allein zu testen. Egal, welches UML-Diagramm verwendet wird, es fehlen immer Informationen für den Test. Vollständige automatisch verarbeitbare Testfälle können nur aus der Zusammensetzung mehrerer UML-Diagrammtypen abgeleitet werden, z.B. aus den Anwendungsfalldiagrammen, den Sequenzdiagrammen und den Zustandsdiagrammen in Kombination mit der Object Constraint Language – OCL [WaKl99].

8.4.4 Test gegen das Nutzungsprofil

Das vierte und letzte potenzielle Testorakel für den Systemtest ist die Nutzung des Systems selbst. Dieser Ansatz bietet sich an, wenn weder ein Benutzerhandbuch, ein Fachkonzept noch ein Objektmodell vorliegt. In diesem Falle bleibt nichts anderes übrig, als das System einfach zu benutzen und ein Nutzungsprofil zu erstellen. Der Ansatz kann aber auch in Kombination mit den anderen Ansätzen verwendet werden, insbesondere für den Regressionstest.

Der Begriff des Nutzungsprofils (*operational profile)* wurde von Musa geprägt. Er stammt aus der Software-Zuverlässigkeitstheorie [MIO87]. Er wird dort verwendet, um das Verhalten von Systemen zu analysieren. Im Grunde genommen heißt es, man setzt ein System einer oder mehreren Nutzungsumgebungen aus und sammelt Daten über die Art und Weise, wie es benutzt wird. Im Falle eines Lagerhaltungssystems würde man das System in einer Lagerhalle einsetzen. Die Lagerarbeiter würden damit arbeiten und man registriert alle Transaktionen mit einem Testmonitor. Auch die Lagerstammdaten werden auf diese Weise aufgebaut. Im Falle eines Buchungssystems würde man das System Pilotbenutzern aussetzen und protokollie-

ren, was sie mit dem System buchen und wie sie buchen. Dabei werden sämtliche Vorgänge elektronisch aufgezeichnet (Abbildung 8.15).

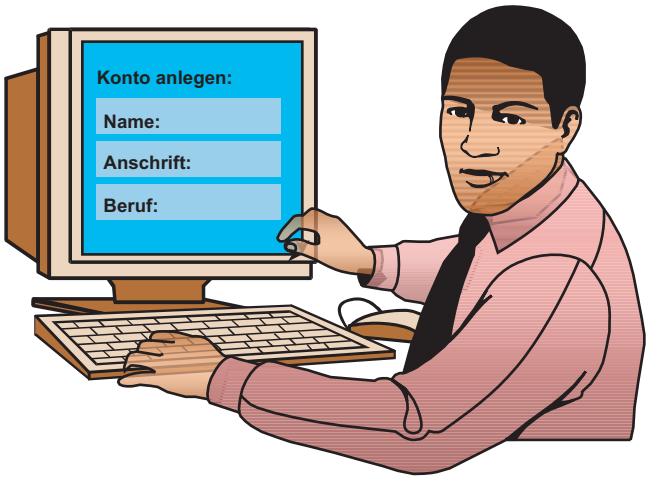


Abbildung 8.15 Test gegen das Nutzungsprofil

Für die Protokollierung der Systemnutzung wird natürlich ein Aufzeichnungswerkzeug gebraucht, das alle Ereignisse und alle Eingabedaten aufzeichnet und abspeichert. Für den Probeeinsatz werden auch tolerante und kooperative Benutzer gebraucht, die einerseits bereit sind, Fehler und Systemabstürze in Kauf zu nehmen und andererseits neugierig genug sind, alle Nutzungsmöglichkeiten auszuschöpfen. Schließlich braucht man ein Datenarchivierungssystem, um in festen Intervallen den Produktionsdatenbestand bzw. die Datenbankinhalte zu sichern.

Nach dem Probeansatz bleiben mehrere Testdatenbestände und zahlreiche aufgezeichnete Testtransaktionen übrig. Diese lassen sich notfalls durch weitere gezielte Testdaten ergänzen. Danach werden die Testdatenbestände geladen und die aufgezeichneten Testtransaktionen mit jeder neuen Version des Systems gestartet. Die neuen Versionen werden damit gegen die Produktionserfahrung der letzten Version getestet. In der Tat wird dieser Testansatz schon vielfach verwendet. Es fehlt nur an Systematik. Im Prinzip könnte man eine Affenherde an die Bildschirme setzen und zuschauen was passiert. Aber wir haben bereits gesehen: ein Zufallstest erzeugt auch nur zufällige Ergebnisse. Deshalb empfiehlt es sich, systematischer vorzugehen und das System so zu benutzen, wie es nachher in der Produktion wirklich benutzt werden sollte. Es kommt darauf an, ein möglichst repräsentatives Nutzungsprofil zu gewinnen [Mun98].

8.5 Systemtest des verteilten Kalenders

Wir illustrieren den Systemtest mit Bezug auf das Fachkonzept wieder am Beispiel des verteilten Kalenders. Der Systemtest vollzieht sich in drei Phasen: In der ersten Phase wird die Benutzungsoberfläche getestet. In der zweiten Phase wird die Funktionalität des Systems getestet. In der dritten Phase wird die Belastbarkeit des Systems getestet.

8.5.1 Oberflächentest

Die Oberfläche des verteilten Kalenders hat ein nummerisches Feld für die Eingabe der Wochenzahl, ein Textfeld für die Eingaben des Kalenderbesitzers und ein Auswahlfeld für die Auswahl des Wochentages (Abbildung 8.16). Diese Angaben bilden den Kopf der Oberfläche. Den Rumpf der Oberfläche bildet die Tabelle der Tagesaktivitäten mit bis zu 12 Zeilen. Jede Zeile hat je ein nummerisches Feld für die Anfangs- und Endezeit, ein Schlüsselfeld für das Projektkennzeichen mit Pop-up-Menü, das alle Projektschlüssel zur Auswahl stellt, und eine Testbox für eine bis zu 80 Zeichen lange Testbeschreibung. Der Fuß der Oberfläche enthält nur eine Textanzeige für die Rückmeldung. Dies kann eine Fehlermeldung oder eine Bestätigung sein.

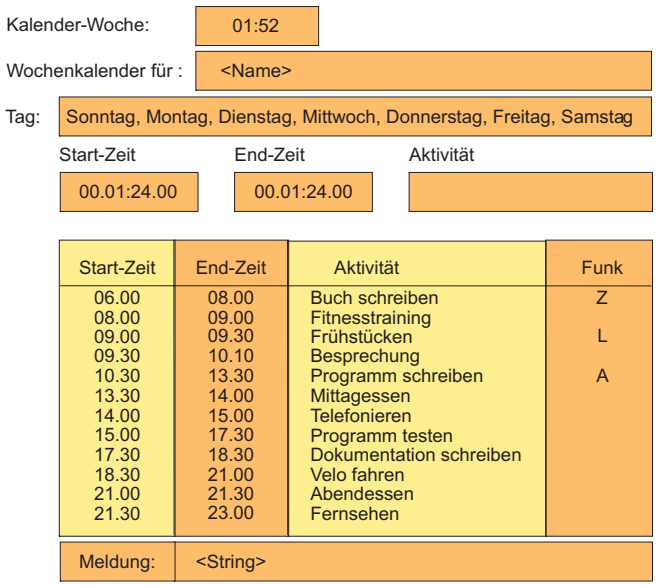


Abbildung 8.16 Kalenderoberflächentest

Wenn es darum geht, diese Oberfläche zu testen, fallen spontan viele Testfälle ein, so z.B.

- eine falsche Wochenzahl,
- ein ungültiger Name,
- keine Wochenauswahl,
- falsche Startzeit,
- falsche Endezeit usw.

Wer systematisch an die Sache herangeht, würde jeden Testfall vorher notieren. Für die Wochenzahl gilt die Grenzwertanalyse. Dort müsste es eine Zahl unter der Untergrenze, die Untergrenze, die Obergrenze und eine Zahl über der Obergrenze, z.B. 0, 1, 52 und 53, sein.
Für den Kalenderbesitzer werden Äquivalenzklassen gebildet. Darin gibt es:

- keinen Namen,
- einen ungültigen Namen und
- einen gültigen Namen, also Leerzeichen (spaces), Mr. X und einen gültigen Mitarbeiternamen.

Für den Wochentag hat man nur die Möglichkeit, einen Tag aus der Menüauswahl zu wählen oder nicht. Dennoch müsste man wissen, dass Feiertage anders behandelt werden als Arbeitstage. Daher die drei folgenden Testfälle:

- keine Auswahl,
- Auswahl eines Wochentages und
- Auswahl eines Feiertages.

Da falsche Kopfdaten zu einem Abbruch der Transaktion führen, gibt es fünf Fälle, die zum Abbruch führen:

- eine Wochenzahl 0,
- eine Wochenzahl 53,
- kein Name,
- ein ungültiger Name und
- keine Wochentagauswahl.

Vier Fälle führen zu einer Fortsetzung der Transaktion:

- 1. Woche & Arbeitstag,
- 1. Woche & Feiertag,
- 52. Woche & Arbeitstag und
- 52. Woche & Feiertag

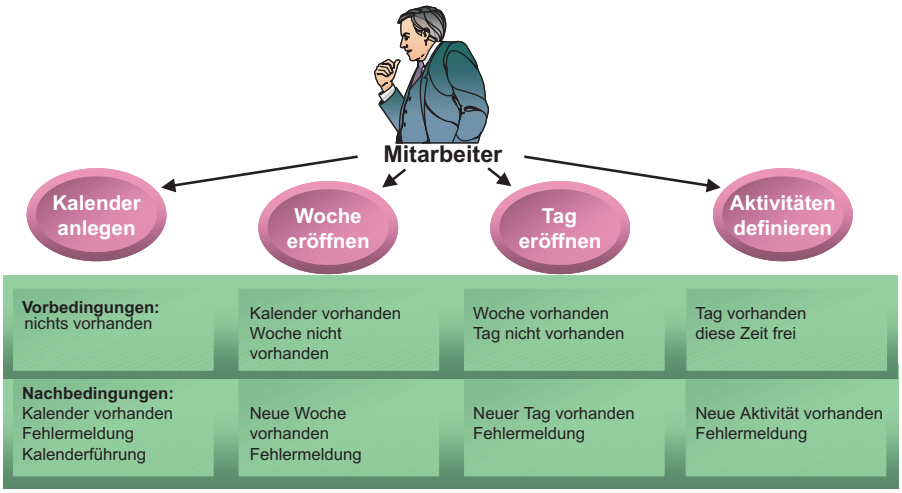


Abbildung 8.17 Kalender: Anwendungsfallbasierter Test

Jetzt kommen wir zum Rumpfteil. Hier gelten die repräsentativen Werte sowie die Grenzwerte. Die Grenzwerte der Anfangs- und Endezeiten sind 00:00 und 24:00. Es müsste also für beide Angaben einen negativen Wert und einen Wert größer als 24:00 Uhr geben, z.B. –01:00 und 24:30. Das ergibt schon vier Testfälle. Außerdem muss die Startzeit kleiner als die Endezeit sein, also fünf fehlerhafte Testfälle. Für den Projektschlüssel gibt es nur zwei Möglichkeiten, da eine Auswahl aus gültigen Schlüsseln erfolgt – also einen gültigen Projektschlüssel und gar keinen. Schließlich gibt es für die Aktivitätenbeschreibung auch nur zwei Möglichkeiten: mindestens ein Buchstabe und gar nichts. Aus der Kombination der Eingabemöglichkeiten ergeben sich acht Testfälle, die zu einer Fehlermeldung führen sollten:

- Startzeit = –01:00
- Startzeit = 24:30
- Endezeit = –01:00
- Startzeit = 25:00
- Startzeit = 09:00 & Endezeit = 09:00
- Startzeit = 09:00 & Endezeit = 08:30
- kein Projektschlüssel
- kein Aktivitätenschlüssel.

Die korrekten Tagesaktivitäten sollten die gültigen Zeiten und Projektschlüssel repräsentieren. Es ist auch an die Aktivitätenanzahl zu denken. Hier kommt die Grenzwertanalyse wieder zum Tragen. Folgende Fälle wären zu testen:

- keine Aktivitäten,
- eine Aktivität = Untergrenze,

- sechs Aktivitäten = Mittelwert,
- zwölf Aktivitäten = Obergrenze und
- dreizehn Aktivitäten.

In der Summe ergeben sich insgesamt 13 Testfälle für den Rumpftest bei der Ersterfassung der Tagesaktivitäten. Hinzu kommen die Fälle für die Fortschreibung bestehender Tagesaktivitäten (Abbildung 8.18). Dazu gehören:

- die Löschung einzelner Aktivitäten,
- die Löschung aller Aktivitäten,
- die Änderung einer Startzeit,
- die Änderung einer Endezeit,
- die Änderung eines Projektschlüssels,
- die Änderung einer Aktivitätenbeschreibung und
- die Änderung aller Attribute einer Aktivität.

Das ergibt sieben weitere Testfälle.

TESTFALL	WOCHE	TAG	STARTZEIT	ENDEZEIT	AKTIVITÄT	MELDUNG
1	0	-	-	-	-	Falsche Wochenzahl
2	53	-	-	-	-	Falsche Wochenzahl
3	1	Sonntag	-	-	-	Falscher Wochentag
4	2	Montag	00.00	-	-	Falsche Anfangszeit
5	3	Dienstag	24.01	-	-	Falsche Anfangszeit
6	4	Mittwoch	00.01	24.01	-	Falsche Endezeit
7	5	Donnerstag	22.30	00.00	-	Falsche Endezeit
8	6	Freitag	23.00	00.30	-	Falsche Zeitangabe
9	52	Samstag	05.00	06.00	Sport treiben	Aktivitäten eingefügt
10	52	Samstag	06.00	07.00	Frühstücken	Aktivitäten eingefügt
11	52	Samstag	07.00	08.00	zur Arbeit fahren	Aktivitäten eingefügt
12	52	Samstag	08.00	09.00	E-Mail lesen	Aktivitäten eingefügt
13	52	Samstag	09.00	10.00	Besprechung	Aktivitäten eingefügt
14	52	Samstag	10.00	11.00	Dokumentieren	Aktivitäten eingefügt
15	52	Samstag	11.00	12.00	Programmieren	Aktivitäten eingefügt
16	52	Samstag	12.00	13.00	Mittagessen	Aktivitäten eingefügt
17	52	Samstag	13.00	14.30	Programmieren	Aktivitäten eingefügt
18	52	Samstag	14.30	17.00	Testen	Aktivitäten eingefügt
19	52	Samstag	17.00	18.00	nach Hause fahren	Aktivitäten eingefügt
20	52	Samstag	18.00	20.00	Abendessen	Aktivitäten eingefügt
21	52	Samstag	20.00	20.15	Sex	Tagespensum ist voll

Abbildung 8.18 Kalenderfunktionstest

Insgesamt sind somit mindestens die dargestellten Testfälle für den Oberflächentest erforderlich. Natürlich gibt es außerdem Zufallsmöglichkeiten, wie Drücken falscher Kontrolltasten oder Betätigen falscher Mausklicks, sowie Alternativen der Dateneingabe, wie Positionieren durch Mausklick oder Positionieren durch Tabulatortasten. Solche Bedingungsvarianten führen zu einer Verdoppelung der Testfallanzahl. Daran lässt sich sehen, wie aufwändig der Test einer graphischen Oberfläche werden kann.

8.5.2 Funktionalitätstest

Die Funktionalität des Kalendersystems wird an Hand der Systemtestfälle demonstriert. Als Grundlage dienen alle Testfälle aus dem 5. Kapitel. Dazu zählen alle gültigen Testfälle aus dem Oberflächentest plus die Integrationstestfälle und die Sonderfälle.

Um die Funktionalität des Kalendersystems zu validieren, müssen der Projektserver und der Kalenderserver aktiviert werden. Eine Datenbank gültiger Projekte muss am Projektserver vorhanden sein, ebenso die Programme zur Fortschreibung dieser Datenbank. Die Datenbank für die Speicherung der Kalender muss auf dem Kalenderserver eingerichtet sein. Schließlich muss die Middleware bzw. der Request Broker in Betrieb sein. Es steht also die gesamte Konfiguration zur Verfügung:

- Benutzerarbeitsplatz,
- Projektserver,
- Projektdatenbank,
- Projektkomponente,
- Kalenderserver,
- Kalenderdatenbank,
- Kalenderkomponente und
- Request Broker.

Jetzt kann es losgehen. Ein Tester sitzt am Benutzerarbeitsplatz und testet eine Funktion nach der anderen:

- Kalender anlegen,
- Aktivitäten definieren,
- Kalender abspeichern,
- Kalender wiedergewinnen,
- Kalender aktualisieren,
- Kalender löschen.

All diese Funktionen auf dem Kalenderserver lösen weitere Funktionen auf dem Projektserver aus, so z.B. die Buchung von Stunden gegen ein Projekt und die Abbuchung von Stunden, falls die Aktivität gelöscht wird. Somit wird gleichzeitig die Interaktion zwischen Mitarbeiterkalender und Projektaufwänden getestet.

Um den Testaufwand zu reduzieren, kann der Tester natürlich durch einen Automaten ersetzt werden. Der Automat zeichnet die Testfälle vom Oberflächentest auf und spielt sie in den Funktionstest zurück. Dazu müssen die Oberflächentestfälle durch die Funktionstestfälle ergänzt werden, d.h. zu den ursprünglichen 25 Oberflächentestfällen kommen weitere 20 bis 30 Funktionstestfälle hinzu. Nach Abschluss des Funktiontests hat sich die Zahl der Systemtestfälle auf mindestens 50 erhöht.

8.5.3 Performanz- und Belastungstest

Die Belastbarkeit des Kalendersystems muss unter echten Produktionsbedingungen getestet werden. Es genügt nicht, von einem Benutzerarbeitsplatz aus zu testen, sondern es müssen mehrere Benutzerarbeitsplätze gleichzeitig bedient werden. Am besten wäre es, den Funktionstest aufzuzeichnen und zu duplizieren. Im duplizierten Testskript wäre es dann erforderlich, die Mitarbeiternamen, die Wochenzahl, den Wochentag und die Projektschlüssel zu ändern, um zu einer größeren Anzahl von Objekten zu kommen. Dies wäre die Aufgabe einer Testprozedur, durch das duplizierte Testskript zu gehen und die bestehenden Objektidentifizierungsmerkmale mit weiteren Ausprägungen zu ersetzen. Auf diese Weise werden die funktionalen Testfälle wiederverwendet und vervielfältigt.

Das Ziel ist, aus den wenigen funktionalen Testfällen eine größere Menge Testfälle zu produzieren, nur indem die Objektschlüssel – Kalendername, Kalenderwoche, Wochentag und Projektziffer – variiert werden. Somit werden aus der relativ kleinen Menge von 50 Testfällen 5.000 Testfälle erzeugt. Denn dadurch unterscheidet sich der Belastbarkeitstest vom Funktionstest. Im Funktionstest wurden pro Klasse immer nur einzelne repräsentative Objekte erzeugt. Im Belastungstest werden pro Klasse eine Vielzahl von Objekten erzeugt, z.B. so viele Kalender, wie es Mitarbeiter geben kann. Auf diese Weise entstehen 100 Kalender, 5.200 Wochen, 36.400 Tage, 436.800 Aktivitäten und 10 Projekte.

Das sind insgesamt 478.510 Objekte, die getestet werden. Wenn man bedenkt, dass jedes dieser Objekte erzeugt, verarbeitet und gesichert werden muss, kann man sich leicht vorstellen, wie komplex der Test realer objektorientierter Systeme werden kann.

9

Testauswertung

Testendekriterien
Testmetriken
Testmessung
Testberichtswesen
Testfortschritt
Testauswertung des verteilten Kalenders

Inhaltsübersicht Kapitel 9

9 Testauswertung

9.1 Testendekriterien

Die ewige Gretchenfrage des Testens lautet: Wann ist man fertig? Wann darf man aufhören? Die Antwort ist ambivalent. Im Prinzip kann man jederzeit aufhören. Eigentlich muss man gar nicht erst damit anfangen, denn das (theoretische) Ziel des Testens, die Fehler zu entfernen und die Korrektheit nachzuweisen, ist ohnehin unerreichbar, und wenn es erreichbar wäre, wäre es nicht erkennbar, d.h. es wird hier ein unerkennbares und unerreichbares Ziel angestrebt.

Unerkennbar ist das Ziel, weil nicht feststellbar ist, wie viele Fehler in der Software stecken. Ein Test kann nur nachweisen, dass noch Fehler existieren, nicht jedoch, dass keiner mehr vorhanden ist. Unerreichbar ist das Ziel, weil Fehler an sich eine Frage der Auslegung sind. Was für den einen ein Feature ist, ist für den anderen ein Fehler. Man kann ja schließlich jeden Fehler als Feature umdefinieren [Goe85].

Ein typisches Beispiel ist die Bearbeitung von Dateien in Windows. Früher, unter DOS, durften Datei- und Verzeichnisnamen nur 8 Zeichen lang sein. Verzeichnisnamen durften keine Erweiterung haben wie `Sources.001`. Jetzt, unter Windows-NT und Windows-XP, ist dies möglich. Dateinamen können bis zu 32 Zeichen lang sein und dürfen auch Erweiterungen haben. Ein Programm von einem der Autoren aus der DOS-Zeit weigert sich, solche Dateien zu öffnen. Der Anwender würde sagen, dies sei ein Fehler. Der Programmersteller könnte argumentieren, dies sei ein Feature. Das Programm verarbeitet nur Dateien in Verzeichnisnamen ohne Erweiterung.

Ein anderes Beispiel ist die Rundung arithmetischer Ergebnisse. Ein von einem der Autoren konvertiertes Programm schneidet von sechs Dezimalstellen die letzten beiden ab. Der Programmersteller behauptet, die Präzision sei mit vier Dezimalstellen ausreichend. Der Kunde will aber wieder seine sechs Kommastellen haben.

Beide „Fehler“ sind eine Frage der Vereinbarung zwischen Auftraggeber und Auftragnehmer bzw. zwischen Kunde und Lieferant. Im ersten Fall müsste die Vereinbarung lauten: Alle Dateien in jeder von MS-Windows bekannten Schreibform sind zu verarbeiten. Dann ist die Ablehnung der Verzeichnisnamen mit Erweiterung

eindeutig eine Verletzung dieser Vereinbarung. Im zweiten Fall müsste die Vereinbarung lauten: Die Ergebnisse der konvertierten Programme müssen haargenau mit den Ergebnissen des ursprünglichen Programms übereinstimmen. Dann sind vier Dezimalstellen in der Tat zu wenig.

So und nicht anders sind Fehler zu interpretieren. Ohne vorherige Vereinbarung ist eine eindeutige Definition von Fehlern nicht möglich, aber wer hat so viel Zeit, alle Eigenschaften eines Software-Systems in dieser Genauigkeit festzulegen. Wenn es so wäre, könnte der Tester die Spezifikation nehmen und sie Punkt für Punkt abarbeiten, indem er jeden Fall in allen Variationen testet. Der Test wäre beendet, wenn alle Anforderungen erfüllt sind – oder nicht? Was wäre, wenn die Software mehr macht als die spezifizierten Anforderungen, und der Benutzer aufgrund eines Bedienungsfehlers einen unspezifizierten (und somit auch nicht geprüften) Weg einschlägt, auf dem das System abbricht? Ist dies ein Fehler? Mehr noch, was wäre, wenn der Programmierer einen Zweig einbaut, der abgerundete Pfennigbeträge auf sein persönliches Konto überweist? Das sind alles Fälle für die Juristen. Hier sollen sie nur dazu dienen, zu zeigen, wie schwer es ist, Fehler zu definieren [NaKu91].

Fehler sind nicht nur Abweichungen von einer vereinbarten Norm. Sie sind auch Verstöße gegen Gesetze und allgemein akzeptierte Verhaltensregeln für Software wie z.B. *„Du sollst nicht abstürzen, egal was der Benutzer macht!“*.

Wenn jedoch all dies berücksichtigt wird, wird das Testen unbezahlbar. Bis das System *ausgetestet* ist, wäre es schon längst veraltet. Also muss man mit gewissen Fehlern leben. Die Frage ist nur, mit welchen?

Hier tritt das zweite Problem auf, das Problem der Unerkennbarkeit. Auch dann, wenn alle Fehlermöglichkeiten im vorhinein definiert wären, wäre es nicht möglich, zu wissen, wann alle gefunden wären, denn verbleibende Fehler lassen sich nicht ohne weiteres erkennen. Man kann nur vermuten, dass es welche gibt. Der Fehleraufdeckungsgrad ist nicht messbar. Also muss man etwas anderes messen, etwas, das stellvertretend ist für das, was wir eigentlich messen wollen, aber nicht messen können. Damit stellt sich die Frage der Testmetriken.

9.2 Testmetriken

Die Testauswertung soll dazu dienen, herauszufinden, ob und in welchem Grad die Testendekriterien erfüllt sind (Abbildung 9.1). Ohne Testendekriterien kann es keine Testauswertung geben. Auch wenn sie nicht schriftlich festgehalten sind, müssen sie mindestens in den Köpfen der Tester existieren.

Testendekriterien sind Ersatzerfolgskriterien für das wahre, aber nicht messbare Erfolgskriterium, die Fehlerlosigkeit. Um eine Aussage über die Menge der verbleibenden Fehler zu treffen, muss man den Umfang des bisherigen Tests in Zahlen ausdrücken und mit der Zahl der bisher gefundenen Fehler verbinden. Mit

diesen beiden Zahlen lässt sich zumindest die Restfehlerwahrscheinlichkeit prognostizieren. Beide Zahlen sind Testmetriken. Testmetriken sind also Maßzahlen zur Messung der Testendekriterien [Dro95].

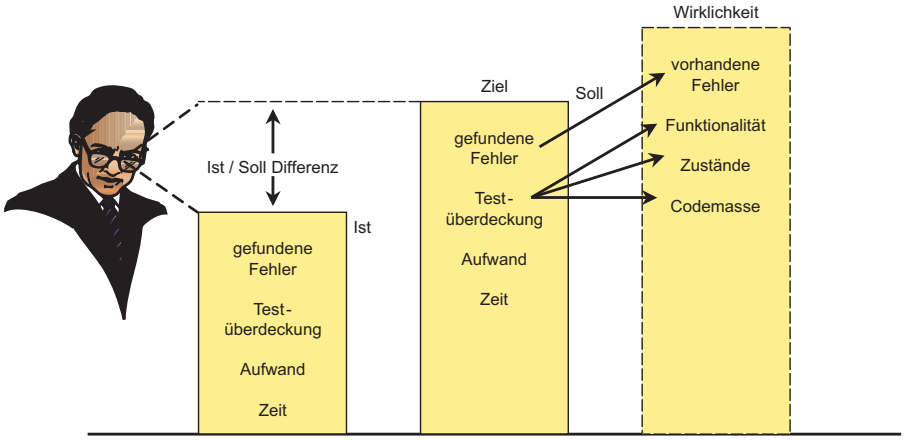


Abbildung 9.1 Testauswertung als Soll/Ist-Vergleich

Testmetriken dienen gleichzeitig einem anderen Zweck, nämlich der Messung der Quantität und Qualität des Tests selbst. Einerseits lässt sich Testquantität in Laufzeit oder Aufwand ausdrücken, so z.B., dass ein System 48 Stunden lang getestet wurde, oder dass drei Tester jeweils zwei Wochen getestet haben. Andererseits lässt sich Testquantität über Testüberdeckungsmaße ermitteln, so z.B., 750 von 1000 Zweigen oder 80 von 100 Funktionen, die getestet wurden. Die letzteren Metriken sind um einiges aussagekräftiger als die ersten, weil Aufwand nicht gleich Leistung bedeutet. Gerade beim Testen können Personen sehr viel Zeit verschwenden, ohne etwas zu erreichen. So kann ein Tester im Prinzip tagelang testen, ohne den Testüberdeckungsgrad im geringsten zu erhöhen. Deshalb ist die Testüberdeckung das bevorzugte Maß für die Testquantität.

Testqualität lässt sich leider nur an der Zahl der gefundenen Fehler messen. Dies könnte eine absolute oder eine relative Zahl sein. Eine absolute Zahl ist sie, wenn die Fehleranzahl allein für sich steht. In diesem Fall ist sie schwer zu bewerten, denn sie hängt von der Quantität und Qualität der Software ab. Je größer und schlechter das Programm, desto mehr Fehler werden gefunden. Dies kann doch nicht der Sinn der Messung sein. Daher die Notwendigkeit einer relativen Fehlerzählung. Demnach werden die gefundenen Fehler mit der Größe der Software in Zusammenhang gebracht. Diese Größe könnten Codezeilen, Anweisungen, Function-Points, Object-Points oder irgendein anderes Größenmaß sein. Jetzt heißt es nicht einfach: 150 gefundene Fehler, sondern 2 Fehler pro 1000 Codezeilen oder 0,2 Fehler pro 100 Function-Points. Dies wird als Fehlerdichte bezeichnet [Bol95].

Dennoch ist auch die Fehlerdichte allein für sich ungenügend. Sie müsste durch eine weitere Metrik, z.B. den Testüberdeckungsgrad, relativiert werden:

$$\textit{Testqualität} = 1 - (\textit{Fehlerdichte} \cdot \textit{Testüberdeckung})$$

So gesehen ist die Testqualität eine Funktion der Fehlerdichte und Testüberdeckung. Doch auch dies ist unbefriedigend. Besser wäre es natürlich, wenn die gefundenen Fehler mit den tatsächlich vorhandenen Fehlern verglichen werden könnten. Dann würde es heißen, von 500 Fehlern im System sind bisher 400 gefunden worden. Da aber niemand wissen kann, wie viele Fehler die Software tatsächlich enthält, lässt sich die Zahl nur schätzen. Diese Fehlerschätzung kann entweder von der Erfahrung mit ähnlichen Projekten ausgehen oder von wissenschaftlichen Studien, in denen die Fehlerdichte für Systeme dieser Art untersucht wurden. Sie könnte auch durch die *Bebugging* zustande kommen. *Bebugging* ist die gezielte Streuung von Fehlern in die Software, mit dem Ziel, zu testen, ob der Tester sie findet [CoMi90].

Die wichtigsten Testmetriken sind jedenfalls

- der Testaufwand in Personentagen,
- die Testzeit in Stunden,
- die Testüberdeckung,
- die Fehlerdichte,
- die gefundenen Fehler und
- die geschätzten bzw. eingepflanzten Fehler.

Testendekriterien stützen sich auf diese Maßzahlen und werden durch sie bestätigt. Ein Test ist fertig, wenn die Testendekriterien erfüllt sind. Die Testendekriterien sind erfüllt, wenn die Istmaße die Sollmaße erreicht haben. Anders lässt sich der Testfortschritt kaum messen. Im restlichen Teil dieses Kapitels werden die oben genannten Testmetriken näher erläutert und erklärt, wie sie bei objektorientierter Software angewandt werden können. Dabei kommen die besonderen Schwierigkeiten der Objekttechnologie bezüglich Testbarkeit zum Vorschein. Zum Schluss wird das Fallbeispiel *verteilter Kalender* mit Bezug auf jene Metriken ausgewertet.

9.2.1 Testprozessmetriken

Der Testprozess hat eine Dauer und einen Aufwand. Die Dauer wird in Kalendertagen und Teststunden gemessen. Der Aufwand wird in Personentagen für den Test gemessen. Diese Tage teilen sich wiederum in Analytikertage, Entwicklertage und Testertage. Mit diesen beiden Maßzahlen misst man die Kosten des Tests. Auf der Nutzenseite gibt es die Anzahl der Testfälle und der Anzahl der gemeldeten Fehler. Diese beiden Maßzahlen sind Kennzahlen für die Testproduktivität, denn im Gegensatz zur Entwicklung, wo Codezeilen, Anweisungen, Function-Points und Ob-

ject-Points produziert werden, werden beim Test Testfälle und Fehlermeldungen produziert (Abbildung 9.2).

Kalendertage messen die Dauer eines Tests vom Beginn bis zum Ende. Da das Testprojekt parallel zum Entwicklungsprojekt läuft und die Testaktivitäten mit den Entwicklungsaktivitäten vermischt sind, empfiehlt es sich, den Testanfang als die erste Übergabe der ersten Version an die Testgruppe zu deklarieren. Das Ende ist, wenn das Software-System freigegeben bzw. in der Produktion eingesetzt wird. Somit bleiben die ganzen vorbereitenden Maßnahmen wie Testplanung, Testkonzipierung und Testfallspezifikation außen vor, weil hier nur die Dauer der funktionalen Testausführungsaktivitäten gemessen werden. Auch die Testaktivitäten der Entwickler bleiben unberücksichtigt. Dies ist aber die einzig korrekte Antwort auf die Frage „Wie lange dauert der Test?“. Gemeint ist: Wie lange wird das System eigentlich getestet?

Teststunden bedeuten die Gesamtzahl der Stunden, in denen das System von den Testern ausgeführt wird. Sie haben mit der Anzahl Kalendertage nichts zu tun, zum einen, weil nicht den ganzen Tag getestet wird und zum anderen, weil man oft parallel testet. Wenn das System an zwei verschiedenen Rechnern bzw. in zwei getrennten Adressräumen zur gleichen Zeit ausgeführt wird, gilt dies als zwei verschiedene Tests. Also werden die Teststunden doppelt gezählt. Teststunden sind im Prinzip die Systemzeit bzw. die Summe aller Stunden für die Ausführung der Testprozesse. Dieses Maß ist nicht nur als Maß für die Testintensivität wichtig, sondern auch als Maß für die Ermittlung der mittleren Zeitintervalle zwischen zwei Systemabbrüchen (mean time to failure, MTTF) [MIO87].

$$\text{MTTF} = \frac{\text{Teststunden}}{\text{Systemabbrüche}}$$

Der *Testaufwand* wird in Personentagen gemessen. Im Gegensatz zur Testdauer werden hier sämtliche Testaktivitäten mit berücksichtigt, und zwar vom Anfang bis zum Ende des Projekts. Dazu zählen die Testplanung, der Testentwurf, die Testfallspezifikation und die Testvorbereitung ebenso wie die Testdurchführung und die Testauswertung. Außerdem werden hier alle Entwicklerstunden für testbezogene Aufgaben mitgezählt. Voraussetzung dafür ist, dass die Projektmitarbeiter ihre Stunden bei der Zeiterfassung differenziert angeben. Sie müssen die Stunden, die sie für den Test aufwenden, von den Stunden für die Entwicklung unterscheiden. Dies mag wie Haarspalterei erscheinen, aber da ein Großteil des Testaufwands vom Entwickler selbst erbracht wird, ist dies die einzige Möglichkeit, den echten Testaufwand zu erfassen.

Der Testaufwand lässt sich außerdem nach der Mitarbeiterrolle differenzieren. Von den Analytikern erfasste Teststunden gelten als Analytikertestaufwand, von den Entwicklern erfasste Teststunden gelten als Entwicklertestaufwand, und alle von

den Testern erfassten Stunden gelten als reiner Testaufwand. In der Summe müssten diese Stunden mehr als die Hälfte aller Projektstunden ausmachen [HKK+94].

Testfälle sind die Anzahl aller dokumentierten Testfälle, die von den Analytikern, Entwicklern und Testern erstellt werden, und zwar für alle Teststufen: Klassentest, Integrationstest und Systemtest. Jeder Testfall testet eine bestimmte Eigenschaft der Software, wird von einem bestimmten Ereignis ausgelöst und hat eine einmalige Kombination von Eingaben, Ausgaben und internen Zuständen. Die Anzahl Testfälle ist ein Hauptmaß für den Fleiß bzw. die Produktivität der Tester und Entwickler.

Durchgeführte Testläufe sind ein zweites Maß für die Testproduktivität. Jedesmal, wenn eine Dialogsession ausgeführt oder ein Batchprozess ausgelöst wird, gilt dies als Testlauf. Je mehr Testläufe pro Zeiteinheit wie Tag oder Woche gestartet werden, umso höher die Testproduktivität.

Fehlermeldungen sind das dritte Maß für die Testproduktivität. Das Hauptziel des Testens ist, Fehler aufzudecken. Je mehr Fehler aufgedeckt werden, umso produktiver der Test. Es kann aber sein, dass nur wenig Fehler vorhanden sind. In diesem Falle können nur wenig Fehler gefunden werden, auch dann, wenn intensiv getestet wird. Deshalb brauchen wir andere Maße wie Testfälle und Testläufe, um der Testproduktivität wirklich gerecht zu werden. Zusammenfassend ist Testproduktivität ein Produkt der Faktoren Testfälle, Testläufe und Fehlermeldungen relativ zur Testzeit. Es kommt darauf an, möglichst viele Testfälle in möglichst vielen Testläufen mit möglichst vielen Fehlermeldungen in möglichst kurzer Zeit durchzuführen [PoBr87].

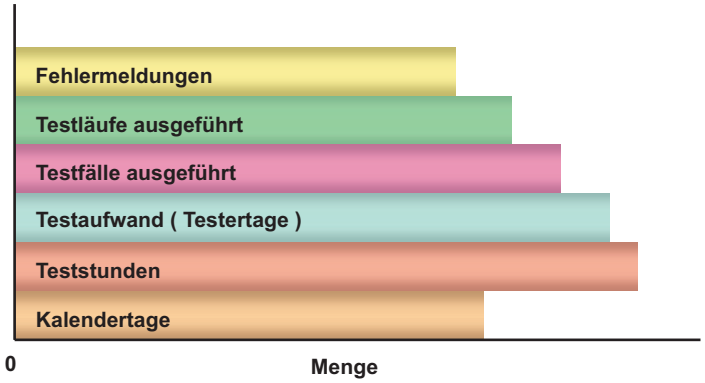


Abbildung 9.2 Testprozessmetriken

9.2.2 Testobjektmetriken

Der Testgegenstand in einer objektorientierten Anwendung ist eine Methode, eine Klasse, eine Komponente, ein Teilsystem oder das ganze System. In allen Fällen ist das Ziel dasselbe: einen möglichst großen Anteil des jeweiligen Testgegenstands in den Test einzubeziehen. In den folgenden Abschnitten werden die in Abbildung 9.3 aufgeführten Metriken für die einzelnen Testobjektarten näher erläutert.

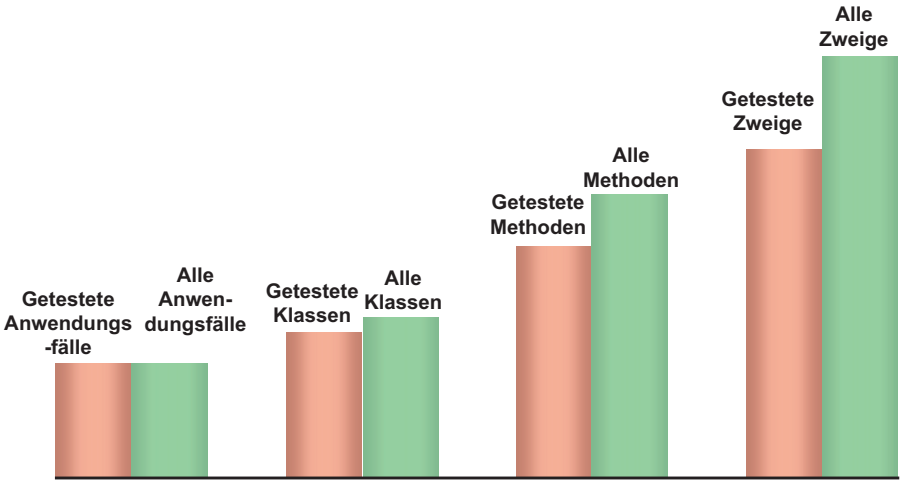


Abbildung 9.3 Testobjektmetriken

9.2.3 Objektabdeckung

Objektinstanzen sind einmalige Ausprägungen einer Objektklasse wie z.B. das Konto `4711` für den Kunden `Meyer`. Sie haben zu einem Zeitpunkt immer einen Zustand wie z.B. `Kontostand = 0`. Beim Test der Objektinstanzen kommt es darauf an, alle stellvertretenden Zustände zu testen. Die stellvertretenden Zustände eines Kontos sind z.B. positiv, negativ und gesperrt. Jeder dieser Zustände bildet eine Äquivalenzklasse, d.h. ein konkreter Zustand ist äquivalent zu allen anderen Zuständen dieser Klasse. Das Konto `4711` ist getestet, wenn alle stellvertretenden Zustände bzw. alle Äquivalenzklassen mindestens einmal vorgekommen sind. Demnach sollte der Test ein Konto `4711` mit einem positiven Kontostand, ein Konto `4711` mit einem negativen Kontostand und ein Konto `4711` mit einem überzogenen Kontostand erzeugen. Damit wären alle repräsentativen Zustände abgedeckt. Die erreichbaren Überdeckungsmaße der Objektinstanzen sind also:

- Instanzüberdeckung = die Instanz wird mindestens einmal erzeugt.
- Zustandsüberdeckung = jeder repräsentative Zustand einer Instanz wird erzeugt.

Ein Objekt befindet sich immer in einem bestimmten Zustand. Ein Konto ist z.B. in einem der Zustände „Offen“, „Überzogen“ oder „Gesperrt“. Ein Mensch kann ledig, verheiratet, verwitwet oder geschieden sein. Zustandsüberdeckung bedeutet, für jeden Objekttyp jeden möglichen Zustand zu testen. Dies ist zwar leicht gesagt, aber nur schwer zu erreichen. Um die Zustände zu ermitteln, wäre es erforderlich, Zusicherungen in sämtlichen Konstruktor- und Änderungsmethoden einzubauen. Typische Testziele sind:

- Sämtliche Objektausprägungen,
- alle stellvertretenden Ausprägungen oder
- mindestens eine Ausprägung pro Objekttyp

zu erzeugen und zu testen (Abbildung 9.4). Das sind die drei möglichen Objektüberdeckungsmaße [Sie96].

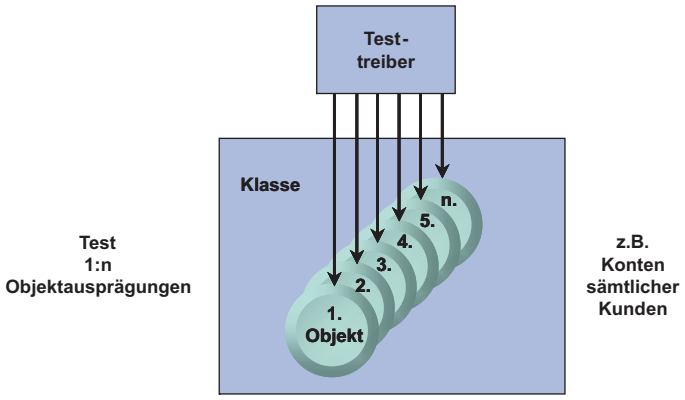


Abbildung 9.4 Objektabdeckung

9.2.3.1 Methodenabdeckung

Methoden als Grundbausteine der Objektorientierung implementieren die Operationen auf den Objekten einer Klasse, z.B. die Eröffnung eines Kontos, die Einzahlung auf das Konto, die Auszahlung von dem Konto oder die Schließung des Kontos. Methoden erhalten Eingangsparameter und liefern Ergebnisse. Es gibt Methoden, die den Zustand eines Objekts verändern, und andere, die den Zustand eines Objekts bereitstellen. Methoden können je nach Sprache 1 bis *n* Anweisungen beinhalten, wobei *n* nicht größer als circa 50 sein sollte.

Methoden sind in den Klassendiagrammen aufgelistet und z.B. mit Sequenz- oder Aktivitätsdiagrammen oder textuell in der OCL beschrieben. Beim Test kommt es darauf an, jede Methode mindestens zwei Mal hintereinander auszuführen (Abbildung 9.5). Um dies feststellen zu können, wird eine Traceanweisung zu Be-

ginn jeder Methode eingefügt. Methodenüberdeckung ist ein Mindestmaß für den objektorientierten Test [WGM95].

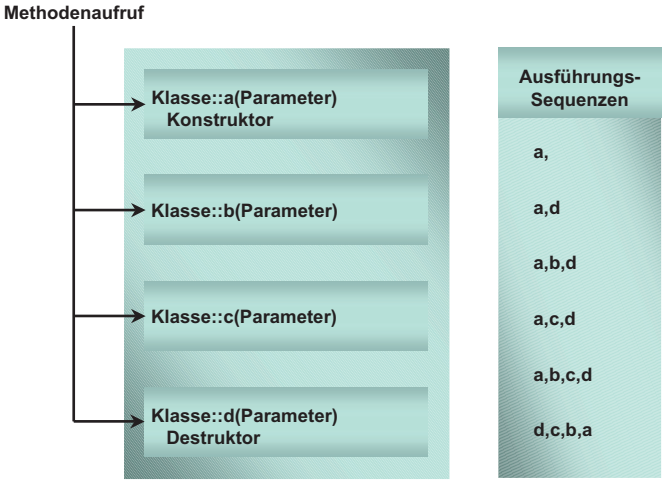


Abbildung 9.5 Methodenabdeckung

9.2.3.2 Zweigabdeckung

Methoden beinhalten Anweisungen, die wiederum in Ablaufzweigen verschachtelt sind. Ablaufzweige entsprechen Entscheidungsausgängen: Überall, wo eine Entscheidung gefällt wird, gehen zwei oder mehr Zweige aus (Abbildung 9.6). So erzeugt eine `IF`-Abfrage zwei Zweige, den `THEN`-Zweig und den `ELSE`-Zweig. Auch Schleifen haben zwei Zweige, nämlich die Fortsetzung der Schleife und den Ausgang aus der Schleife. `CASE`- oder `SWITCH`-Anweisungen haben mehrere Zweige: einen für jeden expliziten Fall und einen für den Sonst-Fall. Operationsaufrufe gelten normalerweise als ein Zweig, aber polymorphe Operationsaufrufe, bei denen verschiedene Funktionen in Frage kommen, haben einen Zweig pro potenziell erreichbarer Funktion.

Wie wir bereits gezeigt haben, können Methoden als gerichtete Graphen mit Knoten und Kanten dargestellt werden. Hierbei werden Entscheidungen durch Knoten und Entscheidungsausgänge durch Kanten repräsentiert. Beim Test der Methoden kommt es darauf an, jede Kante bzw. jeden Zweig mindestens einmal zu durchlaufen. Ein noch anspruchsvolleres Testziel ist es, jede Kombination von Ablaufzweigen bzw. jeden Pfad durch die Methode zu testen. Die Anzahl der Pfade wächst quadratisch mit der Anzahl der Zweige, sodass die Pfadüberdeckung wesentlich schwieriger zu erreichen ist. Andererseits kann man sich damit abfinden, lediglich jede Anweisung auszuführen, also nur die Knoten zu überdecken. Das Mindestmaß

wäre, eine Methode mindestens einmal anzustoßen und irgendwie zu durchlaufen. Daraus ergeben sich die drei Testüberdeckungsmaße für Methoden:

- Methodenüberdeckung,
- Anweisungsüberdeckung und
- Ablaufzweigüberdeckung (vgl. Tabelle 4-1 und Tabelle 4-2).

Die verschiedenen Codeüberdeckungsmaße sind allgemein bekannt, werden aber nur wenig angewendet [PaZw95]. Es hängt von der Kritikalität und Komplexität der Methoden ab, welches dieser Maße angestrebt wird. Dabei darf man nicht übersehen, dass diese Maße nur ein Maß für den Test und nicht ein Maß für die Fehlerfreiheit ist. Auch wenn alle Zweige durchlaufen werden, kann die Methode immer noch fehlerhaft sein. Die ANSI/IEEE-Norm für den Unit-Test verlangt Anweisungdeckung als Mindestmaß [IEEE1008]. Der Hauptnutzen solcher Maße ist, dass sie den Tester zwingen, sich mit dem Testobjekt auseinander zu setzen. Um alle Zweige zu erreichen, muss er sich mit der Entscheidungslogik der Methode sowie der Auswirkung der Parameter auf den Ablauf auseinander setzen. Dadurch wird er zwangsläufig auf gewisse Fehler stoßen. Dennoch wird auf diese Weise nur ein Teil aller Fehler gefunden, laut aller Erfahrungen nicht mehr als 50% [Chu97].

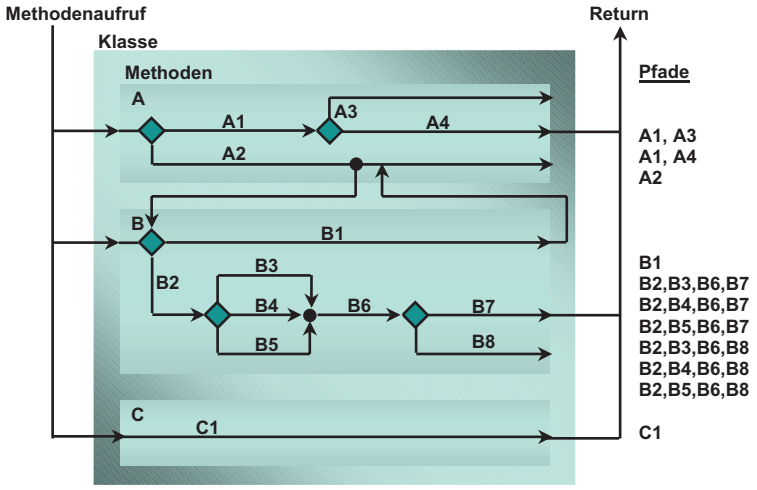


Abbildung 9.6 Zweigabdeckung

9.2.3.3 Komponentenabdeckung

Komponenten bestehen aus einer in sich geschlossenen Menge von Klassen. Die Methoden der Klassen können zwar Methoden in fremden Klassen aufrufen, aber nur über vorbestimmte Schnittstellen. Außerdem, was noch wichtiger ist, keine

Klasse erbt von einer Klasse außerhalb der Komponente. Wenn zur Compilerzeit die Komponente kompiliert und gebunden wird, kann dies geschehen ohne Verweis auf Elemente in anderen Komponenten. Alles, was der Compiler braucht, um ein Lademodul zu erstellen, gehört zur Komponente selbst.

Es gibt also innerhalb der vom Compiler zusammengebundenen Komponenten zweierlei Verknüpfungen,

- interne und
- externe.

Die internen Verknüpfungen sind Vererbungen und Operationsaufrufe innerhalb der Komponente. Sie werden beide zur Compiler- bzw. Linkzeit gelöst. Die Vererbungsverweise werden befriedigt, ebenso die Verweise auf Methoden innerhalb der Komponente. Offen bleiben hingegen die Verweise auf Methoden außerhalb der Komponente. Sie werden nur im Bezug zur Schnittstelle bzw. zur Prototypdefinition der fremden Methode geprüft.

Im Sinne des Komponententests sind vor allem interne Verknüpfungen von Interesse. Externe Verknüpfungen können nicht getestet werden, da fremde Komponenten nicht Bestandteil des Tests sind. Sie müssen abgefangen und simuliert werden. Anders die internen Verknüpfungen: die Vererbungsbeziehungen werden zur Kompilierzeit aufgelöst, die internen Operationsaufrufe zur Laufzeit gebunden. Diese beiden Verknüpfungen sind die Maße für die Vollständigkeit eines Komponententests.

Zum einen gilt es zu messen, ob alle geerbten Operationen durchlaufen werden. Dazu gehört eine spezielle Instrumentierung aller Aufrufe geerbter Operationen. Sie sind an der Qualifizierung erkennbar, z.B.

```
Konto.einzahlung()    = Basis-Operation
Girokonto : Konto     = Girokonto erbt von Konto
Girokonto.kontostand = Girokonto.einzahlung(float betrag);
```

Die Qualifizierung der Operation `einzahlung()` durch die Klasse `Girokonto` zeigt, dass es sich hier um eine geerbte Operation handelt. Also muss diese Anweisung als interne Verknüpfung registriert werden.

Zum anderen gilt es zu messen, ob alle assoziierten Operationen durchlaufen werden. Assoziierte Operationen sind Methoden anderer Klassen derselben Komponente, die nicht geerbt werden. Somit hat die aufrufende Methode keinen Zugang zu ihren Daten, sie sind ihr gegenüber gekapselt. Dazu gehört z.B. folgender Aufruf in der Klasse `Konto`

```
Konto:: auszahlung( );
Bonitaet = Kunde.pruefe_Bonitaet (int kundennr);
```

wobei `pruefe_Bonitaet()` zur Klasse `Kunde` gehört, zu der eine Assoziation besteht. Also muss diese Anweisung als Referenz auf eine fremde Methode registriert werden.

Das Ziel des Komponententests ist es, möglichst alle Aufrufe vererbter Operationen sowie möglichst alle Aufrufe assoziierter Operationen auszuführen. Schließlich sollten alle Aufrufe fremder Komponenten getestet werden, auch dann, wenn sie nur durch Proxy-Schnittstellen simuliert sind.

Die drei Testüberdeckungsmaße der Komponenten sind demzufolge (Abbildung 9.7):

- Die Vererbungsdeckung,
- die Assoziationsdeckung und
- die Schnittstellendeckung.

Ein Blick auf die Entwurfsdokumentation bezeugt, dass der Komponententest eigentlich ein Test gegen das Klassendiagramm ist, denn alle hier zu testenden Beziehungen sind dort abgebildet [JoEr94].

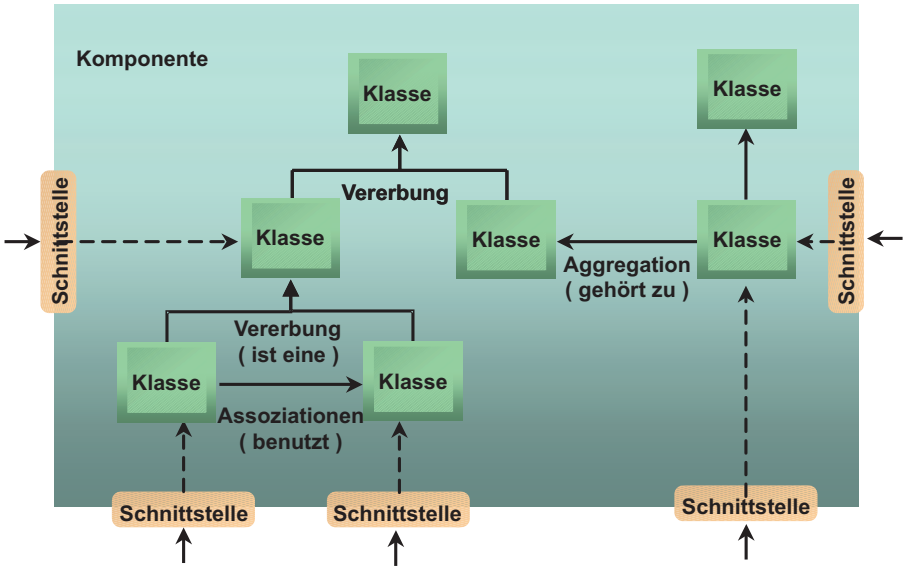


Abbildung 9.7 Schnittstellenabdeckung

9.2.3.4 Teilsystemabdeckung

Im Teilsystem bleiben folgende Beziehungen zu testen (Abbildung 9.8):

- Beziehungen zwischen Komponenten und
- Beziehungen zwischen dem System und der Außenwelt.

Operationen in Komponenten rufen Operationen in fremden Komponenten auf, und zwar in der Regel über spezielle Programmschnittstellen (APIs). Solche Interaktio-

nen zwischen Komponenten sind in der Schnittstellendefinition erkennbar und zählbar. Außerdem lässt sich zählen, wie oft sie ausgeführt werden.
Etwas schwieriger gestaltet sich die Messung der Interaktionen zwischen einem System und seiner Umgebung. Hierzu zählen alle Nachrichten an die Benutzungsoberfläche, alle Zugriffe auf die Datenbank sowie alle Datenübergaben an fremde Systeme. Was hier gezählt wird, sind letztendlich alle Datenströme von und an die Außenwelt. Die meisten können im Source erkannt und markiert werden, aber nicht alle. Wenigstens von denen, die sich erkennen lassen, gilt es, die Häufigkeit zu messen.

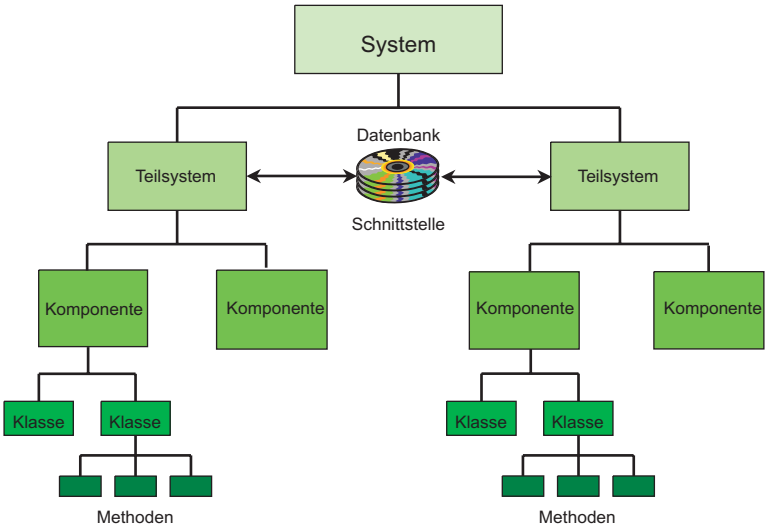


Abbildung 9.8 Teilsystemabdeckung

9.2.3.5 Zusicherungsabdeckung

Wie wir bereits dargestellt haben, wird zum Zwecke der Software-Korrektheit empfohlen, Zusicherungen bzw. Assert-Anweisungen an bestimmten Stellen in den Code einzubauen (vgl. Abschnitt 6.5.3). Zumindest nach jedem Methodeneingang soll eine Assert-Anweisung platziert werden, um die Eingangsparameter zu prüfen, und vor jedem Methodenausgang bzw. `RETURN` sollte eine `Assert`-Anweisung stehen, um den Ausgangswert zu bestätigen. Darüber hinaus können Zusicherungen in Schleifen sowie in Fallanweisungen zur Kontrolle der Zwischenzustände eingebaut werden (Abbildung 9.9). Falls die Zusicherung nicht erfüllt ist, wird eine Ausnahme gemeldet bzw. eine Ausnahmebehandlung (exception handling) ausgelöst.
Das Ziel der Zusicherungsabdeckung ist es, jede einzelne Zusicherung mindestens einmal zu verletzen, sodass die Fehlermeldung erfolgt. Dies soll bestätigen, dass die

Zusicherungen funktionieren und dass sie zwischen korrekten und nicht korrekten Zuständen unterscheiden können. Dazu muss der Entwickler die Testdaten entsprechend manipulieren. Zum einen muss er Methoden mit falschen Eingangsparametern aufrufen, zum anderen muss er ggf. die Methoden selbst verfälschen, damit sie falsche Ausgangswerte erzeugen. Bei Schleifen muss er z.B. die Grenzen überschreiten und bei Fallanweisungen unabgedeckte Fälle konstruieren. All dies soll ihn zwingen, sich mit der Logik seiner Methoden auseinanderzusetzen, umso hinter dem künstlich konstruierten Fehler andere echte Fehler aufzudecken [Ros95].

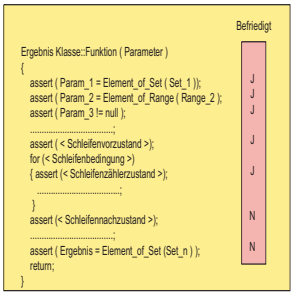


Abbildung 9.9 Zusicherungsabdeckung

Die Zusicherungsabdeckung ist mit der Bebuggingtechnik verwandt, nach der künstliche Fehler in den Code verplanzt werden. Dies mag für viele als eine akademische Übung erscheinen, aber sie ist die einzige Möglichkeit, die Qualität des Tests zu testen. Wenn der Test nicht einmal die offensichtlichen Fehler aufdeckt, wie soll der dann die subtilen aufdecken? Die Korrektheit der Software kann letztendlich nur über die Erprobung aller Varianten nachgewiesen werden.

9.2.4 Funktionstestmetriken

Neben den bisher genannten Architekturmaßen hat jedes System auch eine Funktionalität. Es dient dazu, gewisse Anwenderanforderungen zu erfüllen. In prozeduralen Systemen sind diese Funktionen oft 1:1 im Code abgebildet, d.h. eine Funktion wird von einer Code-Einheit implementiert. Wenn die Codeüberdeckung gemessen wird, wird auch die Funktionsüberdeckung gemessen. Nicht so in objektorientierten Systemen. Dort sind Architektur und Funktionalität voneinander entkoppelt. Eine Überdeckung der inneren Architektur ist noch lange keine Überdeckung der äußeren Funktionalität und umgekehrt. Deshalb muss der Test der äußerlichen Funktionalität an und für sich gemessen werden [How87] [Win99].

9.2.4.1 Funktionsüberdeckung

Die Funktionalität eines Software-Systems sollte im Fachkonzept festgehalten werden. Dort sind die Funktionen z.B. in Form eines Funktionsbams hierarchisch aufgelistet und jede elementare Funktion im Bezug auf ihre Vor- und Nachzustände sowie auf ihre Prädikate, Argumente und Ergebnisse und ihre Auslöser beschrieben. Dieses Funktionsverzeichnis bietet die statische Betrachtung der Anwendung. Die dynamische Betrachtung liefert die Beschreibung der Anwendungsfälle. Die Anwendungsfälle steigen über die Ereignisse bzw. Auslöser in das System ein, d.h. wenn der Anwender einen bestimmten Impuls gibt, folgt eine oder mehrere Reaktionen. Diese Reaktionen sind oft Funktionsketten, d.h. es werden Funktionen aus dem Funktionsverzeichnis in einer bestimmten Reihenfolge ausgeführt, z.B. wenn ein Artikel bestellt wird.

Zwischen den elementaren Funktionen des Fachkonzepts und den Methoden der Klassen gibt es ein ambivalentes Verhältnis. Manchmal sind sie identisch, z.B. könnte die fachliche Funktion Geld_abbuchen der Operation `auszahlung()` in der Klasse `Konto` 1:1 entsprechen. Manchmal sind sie hierarchisch, nämlich wenn zu einer fachlichen Funktion mehrere Operationen unterschiedlicher Klassen gehören. Dies ist der Fall bei einer Artikelbestellung. Hier werden mehrere Klassenoperationen betätigt, z.B.:

- `prüfe_Bonität()` in der Klasse `Kunde`,
- `prüfe_Artikel()` in der Klasse `Artikel` und
- `sende_Paket()` in der Klasse `Versand`.

Hier handelt es sich um eine 1:*n*-Beziehung.

Nicht selten ist jedoch die Beziehung zwischen fachlichen und technischen Funktionen eine *m*:*n*-Beziehung. Um eine Menge zusammenhängender fachlicher Funktionen zu erfüllen, ist eine andere Menge technischer Funktionen in der Software erforderlich. Welche Menge dies ist, wird bei objektorientierten Systemen oft dynamisch zur Laufzeit bestimmt. An diesem Punkt reißt der Zusammenhang zwischen fachlicher Funktionalität und technischer Implementierung ab. Ein klassisches Beispiel ist ein Suchvorgang, der über mehrere Wege versucht, an ein bestimmtes Objekt heranzukommen. Wenn ein Weg blockiert ist, wird ein anderer Weg versucht. So kann es sein, dass zur Erfüllung dieser fachlichen Funktion jedes Mal eine andere Menge technischer Funktionen ausgeführt werden.

Demzufolge darf der Funktionstest einer objektorientierten Anwendung sich niemals an der implementierten Architektur orientieren. Im Gegenteil, er muss nach dem Funktionsmodell richten, um die Überdeckung der Funktionalität zu messen [Jal89].

Test	Fach_Funktion	Source_Funktion	Test	
4	Kundenerfassung	get_Input	4	
		check_Input	4	
		pass_Input	4	
		Customer	4	
		store_Customer	4	
		verify_Complete	4	
4	Artikelbestellung	get_Order	4	
4	Prüfe_Bestellung	check_Order	4	
4	Prüfe_Bonität	Customer	4	
		check_Credit	4	
		reject_Order	-	Nicht getestet
4	Prüfe_Menge	check_Stock	4	
		reject_Order	-	Nicht getestet
4	Sende_Paket	send_Package	4	

Abbildung 9.10 Funktionsabdeckung

9.2.4.2 Schnittstellenüberdeckung

Zum einen gibt es das statische Operationsverzeichnis. Hier ist das Ziel, jede einzelne elementare Operation der Schnittstelle mindestens drei Mal auszuführen:

- Einmal mit dem normalen Ausgang bei durchschnittlichen Eingangswerten,
- einmal mit dem normalen Ausgang mit Extremeingangswerten (Extreme im Sinne von zu hoch oder zu niedrig, zu kurz oder zu lang, zu viel oder zu wenig) und
- einmal mit einem abnormalen Ausgang mit falschen Eingangswerten (Exception Handling).

Die unterste Funktionsüberdeckung ist die der einmaligen Ausführung mit Normalwerten, die nächst höhere wäre die mehrmalige Ausführung mit Extremwerten, die nächst höhere wäre die mehrmalige Ausführung mit Falschwerten und die höchste Überdeckung wäre die mehrmalige Ausführung jeder Elementarfunktion mit normalen, extremen und falschen Eingangswerten.

9.2.4.3 Anwendungsfallüberdeckung

Zum anderen hat der Tester die Anwendungsfälle. Hierbei geht es um die dynamische Funktionsüberdeckung. Das Ziel ist nicht nur, jeden Anwendungsfall einmal auszuführen, sondern jeden Anwendungsfall mehrfach mit verschiedenen Ausgangszuständen und in verschiedenen Zusammenhängen zu testen.

Die niedrigste Überdeckung wäre im Falle der Anwendungsfälle, dass jeder Anwendungsfall mindestens einmal getestet wird. Die höchste Überdeckung wäre dann gegeben, wenn jeder Anwendungsfall mit jedem möglichen Ausgangszustand und in jedem möglichen Zusammenhang getestet wird. Da dieses Ziel ziemlich hoch gesteckt ist, kommen z.B. folgende, leichter erreichbare Ziele hinzu:

- Jeder Anwendungsfall mit mindestens zwei stellvertretenden Ausgangszuständen in einem Zusammenhang und
- jeder Anwendungsfall mit mindestens zwei stellvertretenden Ausgangszuständen in mehreren Zusammenhängen.

Folgende funktionale Testziele müssen berücksichtigt werden:

- Statisches Ziel und
- dynamisches Ziel.

Der Grad, zu dem diese Ziele erfüllt werden, hängt von der Kritikalität der Anwendung, der Testzeit und der Testkapazität ab. Sogar die niedrigste funktionale Testüberdeckung ist für viele Projekte aufgrund der knappen Zeit und der fehlenden Kapazität nicht erreichbar. Trotzdem soll der Grad der Erfüllung gemessen und registriert werden, um wenigstens zu messen, um wie viel man das Ziel verfehlt hat [Hof99].

9.3 Testmessung

Bei der Ermittlung der Testmetriken sind verschiedene Quellen zu benutzen. Die Prozessmetriken entstammen dem Projektberichtswesen. Die Objektmetriken fallen automatisch als Abfallprodukt des Architekturtests an und die Funktionsmetriken müssen manuell anhand des Fachkonzepts erfasst werden.

9.3.1 Ermittlung der Testprozessmetriken

Die Testprozessmetriken

- Kalendertage und
- Testaufwand

sind den Projektberichten zu entnehmen. Kalendertage sind die bisherige Projektlaufzeit. Testaufwand sind die Stunden, die Mitarbeiter gegen Testaufgaben buchen. Zu diesem Zweck müssen Testaufgaben von Entwicklungs-, Dokumentations- und anderen Aufgaben sauber getrennt werden.

Die Anzahl der Fehlermeldungen geht aus der Fehlerstatistik hervor. Die Tester geben ihre Fehlermeldungen in ein Fehlermeldungssystem ein, wo sie in einer Fehlerdatenbank gesammelt werden. In regelmäßigen Abständen, z.B. alle drei Monate,

kann man Statistikberichte generieren lassen, in dem die Fehler nach Art, Schwere und Komponenten geordnet sind.
Die restlichen Prozessmetriken, dazu gehören

- Teststunden,
- Testläufe und
- Testfälle,

sind der Testdokumentation zu entnehmen. Jeder Tester meldet, wie oft er welche Testläufe startet und wie lange sie dauern. Dies gehört zum Testberichtswesen. Insofern als die Testfälle alle dokumentiert sind, können sie leicht gezählt werden. Ansonsten muss der Tester die Anzahl melden.
Die Erfassung der Prozessmetriken setzt also ein funktionierendes Berichtswesen voraus. Ist dies gegeben, sind sie nur ein Abfallprodukt. Ist dies nicht gegeben, müssen Tester und Entwickler sie extra aufschreiben. Sonst können sie nur geschätzt werden.

9.3.2 Ermittlung der Testobjektmetriken

Die Testobjektmetriken werden über die Instrumentierung des Sources gewonnen. Dazu gehört ein Werkzeug, welches den Source vor dem Test verändert und nach dem Test auswertet. Für die Messung der Objektabdeckung wird in jede Konstruktorfunktion eine Messoperation eingebaut, die registriert, wie oft Objekte der jeweiligen Klasse erzeugt werden. Für die Messung der Methodenabdeckung wird in jede Member-Operation ein Durchlaufzähler nach dem Eingang und vor jedem Ausgang, sowie wahlweise in jeder Schleife, in jeder Fallunterscheidung und in jeder If-Verzweigung eingebaut. Für die Messung der Komponentenabdeckung reicht es, wenn jede Konstruktor-Operation und jeder Aufruf einer fremden Operation instrumentiert werden. Die instrumentierten Konstruktor-Operationen registrieren die Vererbung, die instrumentierten fremden Operationsaufrufe zählen die Assoziationen. Für die Messung der Teilsystemabdeckung werden nur die Schnittstellenoperationen instrumentiert, d.h. jene Operationen, die Operationen in einer fremden Komponente aufrufen. Schließlich muss für die Messung der Zusicherungsabdeckung jede Ausnahmebehandlungsoperation, die durch eine Zusicherungsverletzung ausgelöst wird, instrumentiert werden. Der eingebaute Zähler registriert, ob und wie oft die jeweilige Zusicherung verletzt wird.
Sämtliche Instrumentierungen führen zur Testzeit zum Aufbau einer Tabelle mit einem Eintrag für jeden Messpunkt. Darin wird festgehalten, wie oft die Messpunkte durchlaufen werden. Nach dem Test wird die Tabelle von einem Postprozessor ausgewertet und ein Überdeckungsbericht generiert. So gesehen ist die Messung der Testobjektüberdeckung im ersten Range eine Frage der Source-Instrumentierung durch automatische Testwerkzeuge.

9.3.3 Ermittlung der Funktionstestmetriken

Anders sieht es aus mit der Messung der Funktionsüberdeckung, denn Anwenderfunktionen existieren selten als eindeutig identifizierbare Codeelemente, sondern allenfalls als Knoten eines Funktionsbaums oder als Anwendungsfälle. Es ist daher notwendig, die Ausführung der Anwenderfunktionen in der Funktionsdokumentation bzw. im Konzept festzuhalten.

In der statischen Funktionsbeschreibung bzw. dem Funktionsverzeichnis gehören zu den Attributen einer Funktion die Anzahl der Ausführungen sowie der Korrektheitsstatus der jeweiligen Funktion, z.B. *korrekt*, *falsch* oder *unbekannt*. Es obliegt dem Tester, diese Attribute nach jedem Test fortzuschreiben. Ein einfaches Auswertungsprogramm kann die Attribute anschließend für alle Funktionen im Funktionsbaum auflisten und zusammenfassen.

Das Gleiche gilt für die Anwendungsfälle. Auch für diese sollte dokumentiert sein, wie oft und mit welchem Ergebnis sie getestet worden sind. Auch hier obliegt es dem Tester, diese Statistik nach jedem Test fortzuschreiben und periodisch auszuwerten. Insofern ist die funktionale Testüberdeckung im Wesentlichen eine Frage der Konzeptabdeckung und dies ist wiederum eine Frage der Dokumentation.

9.4 Testberichtswesen

Voraussetzung für die Bewertung der Testeffektivität und Testvollständigkeit ist ein Testberichtswesen. Die ANSI-Norm 829 schreibt vor, welche Berichte zu erstellen sind. Im Wesentlichen handelt es sich um die in Abbildung 9.11 skizzierten fünf Berichtsarten [IEEE829]:

- Testlog,
- Testdeckungsbericht,
- Testvorfallsbericht,
- Testergebnisbericht und
- Testabschlussbericht.

9.4.1 Testlog

Der *Testlog* ist ein Protokoll der jeweiligen Testausführung, aus dem hervorgeht, welche Testfälle in welcher Reihenfolge getestet wurden. Er wird vom Tester selbst oder von einem Testautomat geführt. Wenn ein menschlicher Tester die Testfälle anstößt, ist es auch seine Aufgabe zu registrieren, welche er angestoßen hat. Entweder notiert er sie auf einem Schmierzettel oder er erfasst sie gleich mit einem Test-

falleditor. Wenn ein Automat die Testfälle auslöst, ist es Aufgabe des Automaten, sie in einer Logdatei zu registrieren.

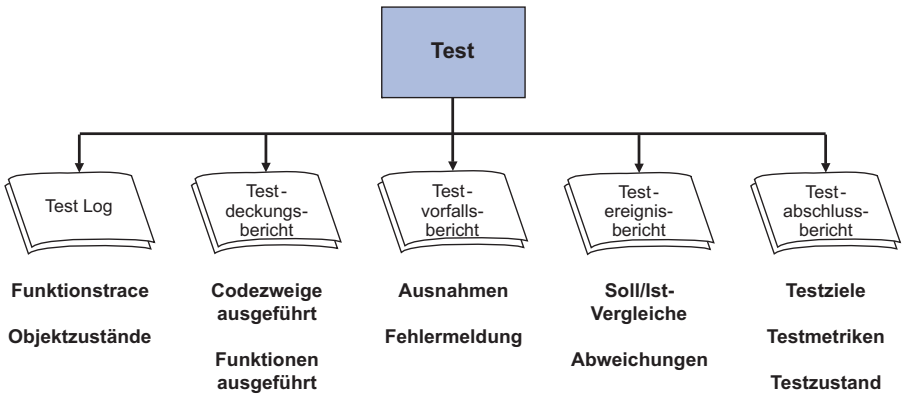


Abbildung 9.11 Testdokumentation

Auf jeden Fall müssen im Endbericht die ausgeführten Testfälle der Reihe nach mit dem Testfallkennzeichen aufgelistet sein. Außerdem ist der Ausgang des Testfalls anzugeben, ob bestanden oder nicht bestanden. Diese einfache Checkliste wäre der Mindeststand. Zusätzlich könnten für jeden Testfall auch die Argumente und Ergebnisse bzw. die Eingaben und Ausgaben angegeben werden.
Die funktionale Testüberdeckung wird anhand des Testlogs ermittelt, und zwar durch den Abgleich der erfolgreich durchgeführten Testfälle mit der Menge sämtlicher Testfälle, die erforderlich sind, um alle statischen und dynamischen Funktionen zu testen.

9.4.2 Testüberdeckungsbericht

Der *Testüberdeckungsbericht* kann nur von einem Automaten erzeugt werden. Durch die Instrumentierung des Codes werden zur Testzeit Durchlauftabellen aufgebaut, in denen festgehalten wird, welche Testknoten im Code wie oft durchlaufen werden. Nach dem Test werden diese Tabellen von einem Postprozessor ausgewertet und die entsprechenden Testdeckungsberichte generiert.
Der Testdeckungsbericht dient der Messung der Testobjektabdeckung und ist demzufolge eher für den Entwickler interessant. Aus den schon erwähnten Gründen ist er nur bedingt für die Testrevision objektorientierter Systeme geeignet, da der Code in der Regel viel umfangreicher als die Applikation ist. Je höher der wiederverwendbare Anteil, umso niedriger wird die Codeüberdeckung ausfallen, weil auch viele Codestrukturen übernommen werden, die für eine bestimmte Applikation irrelevant sind.

9.4.3 Testvorfallsbericht

Der *Testvorfallsbericht* hält alle Sollabweichungen fest. Ein Vorfall, im Englischen incident, ist irgend etwas Unerwartetes. Es muss nicht ein Systemfehler sein, es könnte auch ein ungewöhnliches Ereignis, ein Bedienungsfehler oder ein Zufallsgeschehnis sein. Alles wird hier notiert, was aus der Reihe fällt bzw. als unerwartet auffällt. Später werden diese Vorfälle näher untersucht und ihre Ursachen erforscht. Erst danach werden sie als Fehler, Mängel, Zufälle oder Features klassifiziert. Die Fehler und Mängel werden durch das Fehlermeldungswesen erfasst. Die anderen Vorfälle werden nur zur weiteren Beobachtung notiert.
Es ist äußerst schwierig, einen Automat zu schaffen, der in der Lage wäre, alle Testvorfälle zu erkennen. Er erkennt entweder zu wenig oder zu viel und dann nicht die richtigen. Sogar dazu müssen sämtliche Sollzustände vorgegeben werden. Deshalb ist eine menschliche Kontrolle kaum entbehrlich. Allenfalls ist es möglich, Testzustände aufzuzeichnen und in einem Folgetest zurückzuspielen. So lassen sich die neuen Zustände gegen die alten abgleichen, um Abweichungen zu erkennen und zu protokollieren. Damit stößt die Testautomation an ihre Grenzen. Eine weitergehende Vorfallsberichterstattung obliegt den Testern. Dies ist auch ein Grund für den hohen Personalaufwand beim Test. Nur der Mensch ist heute in der Lage, richtiges vom falschen Verhalten zu unterscheiden.

9.4.4 Testergebnisbericht

Der *Testergebnisbericht* beschreibt den Zustand der Datenbanken nach der Testausführung. Er wird von einem Automaten erstellt, der die Datenbanken durchliest und deren Inhalte protokolliert. Eine einfache Variante listet die Attributwerte zusammen mit den Attributnamen und -typen auf. In dem Fall hat der Tester zu prüfen, ob die Werte richtig sind oder nicht.
Eine anspruchsvollere Variante vergleicht automatisch die Attribute vor dem Test mit denen danach und protokolliert nur gegen Werte, die sich verändert haben. Sie können falsch oder richtig sein. Es bleibt dem Tester überlassen, sie zu kontrollieren, aber die Menge der zu kontrollierenden Daten ist durch die Vorauswahl viel geringer.

9.4.5 Testabschlussbericht

Der *Testabschlussbericht* ist laut ANSI-Norm eine zusammenfassende Darstellung des ganzen Testprojekts, sozusagen eine Art „Abrechnung“. Darin werden die Testaktivitäten beschrieben, die Testergebnisse zusammengefasst und die Testmetriken berichtet. Dazu gehört auch eine Nachkalkulation der Testkosten und eine Gegen-

überstellung mit den Sollkosten. Vor allem wird aber hierin berichtet, zu welchem Grade die Testziele erfüllt wurden. Bei Nichterfüllung ist eine Erklärung nötig, um Schlüsse für die Zukunft ziehen zu können.
Dieser Bericht ist der einzige, der an das Produktmanagement weitergeht. Er soll als Entscheidungsgrundlage dienen, ob das Produkt freigabereif ist oder nicht. Die Kriterien sind gegeben. Es ist dennoch anzunehmen, dass sie nur selten alle erfüllt wurden. In diesem Falle muss das Management entscheiden, ob der Test ausreichend ist. Wenn nicht, wird der Test so lange fortgesetzt, bis die festgelegten Testendekriterien erreicht sind.

9.5 Testfortschritt

Der Testfortschritt lässt sich nur schwerlich messen. Es erübrigt sich daher die naive Frage des Projektmanagements nach dem Stand des Tests. Die Frage „Wie weit ist der Test gediehen?“ ruft unweigerlich die Antwort hervor: „Relativ zu was?“. D.h. zur Beantwortung dieser Frage braucht man ein messbares Ziel. Wenn das Ziel eine Zeitvorgabe wie drei Monate ist, lässt sich die Frage leicht beantworten. Nach zwei Monaten Test ist man zu 66,6...% fertig. Das Gleiche trifft für den Aufwand zu. Wenn es heißt, man hat neun Personenmonate in den Test zu investieren, ist nach dem Einsatz von neun Personenmonaten der Test beendet, auch wenn kein einziger Fehler gefunden wurde.

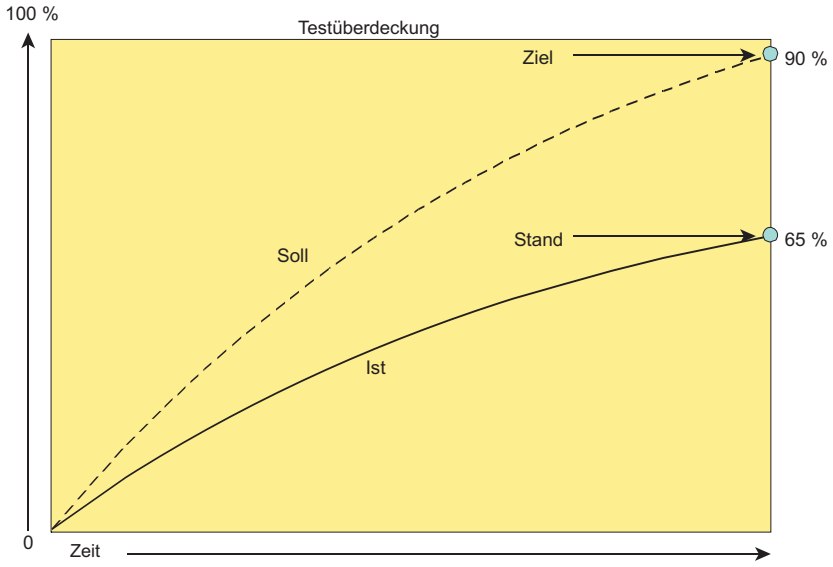


Abbildung 9.12 Testfortschritt

Anders sieht es aus, wenn nach einem bestimmten Testüberdeckungsmaß gearbeitet wird. Hier bestimmt der Test selbst, wie weit er gediehen ist. Ist z.B. im Testprotokoll festgehalten, dass erst 65% der Zweige getestet wurden, und ist das Ziel 90% Zweigüberdeckung, so folgt daraus, dass der Istzustand erst 72,2...% des Sollzustands erreicht hat (Abbildung 9.12). Dasselbe trifft für die Funktionsüberdeckung zu. Wenn 500 Funktionen spezifiziert sind und erst 300 getestet wurden, ist der Test erst zu 60% fertig. Das Projektmanagement kann entscheiden, ob das Produkt trotzdem freizugeben ist. Fertig ist der Test jedoch nicht.

Ohne messbare Testziele ist es also unmöglich, Aussagen über den Stand des Tests zu machen. Man kann allenfalls sagen, es wurde etwas getestet. Was sich hinter dieser Aussage verbirgt, kann jeder raten.

Das eigentliche Ziel des Testens ist die Fehleraufdeckung. Jedes Softwaresystem hat eine bestimmte Anzahl Fehler, die berühmten „Stecknadeln im Heuhaufen“. Demnach ist man erst am Ziel, wenn man den letzten Fehler gefunden hat. Leider kann keiner wissen, wie viele Fehler tatsächlich vorhanden sind. In objektorientierten Systeme und besonders in der Sprache C++ gibt es mehr als genügend Fehlermöglichkeiten. Ob die Entwickler sie alle ausschöpfen oder sie geschickt vermeiden, lässt sich nur durch empirische Untersuchungen feststellen. Testen ist sozusagen der empirischer Beweis, dass Fehler existieren. Eine Aussage bezüglich der Fehlerfreiheit eines Systems ist ausschließlich im Bezug auf eine bekannte oder geschätzte Anzahl Fehler gültig (Abbildung 9.13).

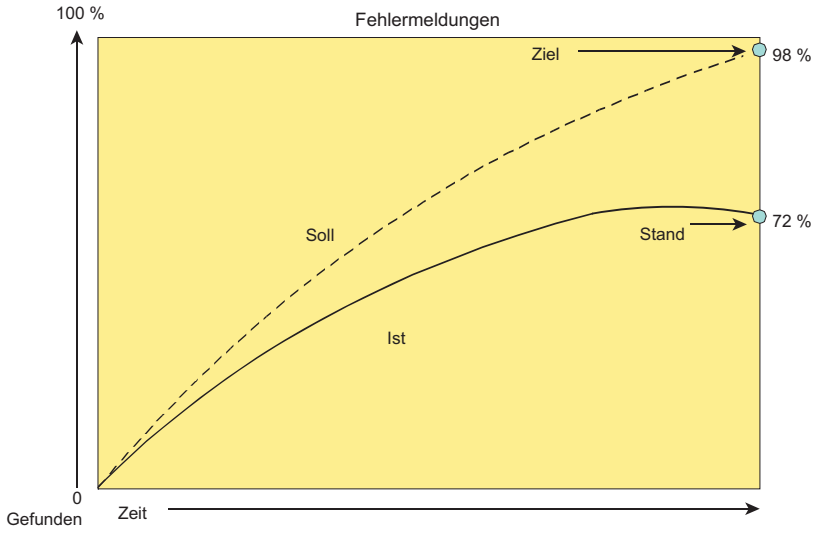


Abbildung 9.13 Fehlerfindungsfortschritt

Leider ist die Anzahl der Fehler keineswegs statisch. Oft glaubt man, mit der Fehlerbehebung gut voranzukommen und fast alle Fehler gefunden zu haben, wenn

plötzlich wieder neue auftreten. Es sind jene Fehler, die durch die Korrektur der bisherigen Fehler entstanden sind, so genannte *Fehler zweiter Ordnung* (2nd level errors). Es kann sogar so weit kommen, dass mit jeder Fehlerkorrektur ein neuer Fehler entsteht. Es ist daher ratsam, die letzten Fehler ruhen zu lassen. Ihre Korrektur könnte nur noch zusätzliche Fehler verursachen. Die Kunst hierbei ist es zu merken, wenn man an dieser kritischen Grenze ankommt.

Hilfreich in dieser Hinsicht ist die Fehleranalyse [MuAc89]. Ein Testbetrieb sollte die durchschnittliche Fehlerdichte der Systeme in Erfahrung bringen. Diese lässt sich an Hand der Relation gefundener Fehler zur Code- bzw. Funktionsgröße messen (Abbildung 9.14). In der Literatur werden 3 Fehler pro 1000 Codezeilen oft als kritische Grenze erwähnt. Es könnten je nach Anwendung auch zwei oder sogar nur einer sein. Wichtig ist, dass die Fehlerdichte überhaupt ermittelt wird. Wenn die letzte Version mit 2 Fehlern pro 1000 Codezeilen freigegeben wurde, wird die nächste Version kaum besser sein. Fehlerdichte ist ein Maß wie die Körpertemperatur. Alle Softwaresysteme haben sie, ohne sie gäbe es keine Software und sie kann von Körper zu Körper und von Zeit zu Zeit schwanken. Sie darf aber nicht zu hoch werden, sonst ist der Körper bzw. das Softwaresystem in Gefahr.

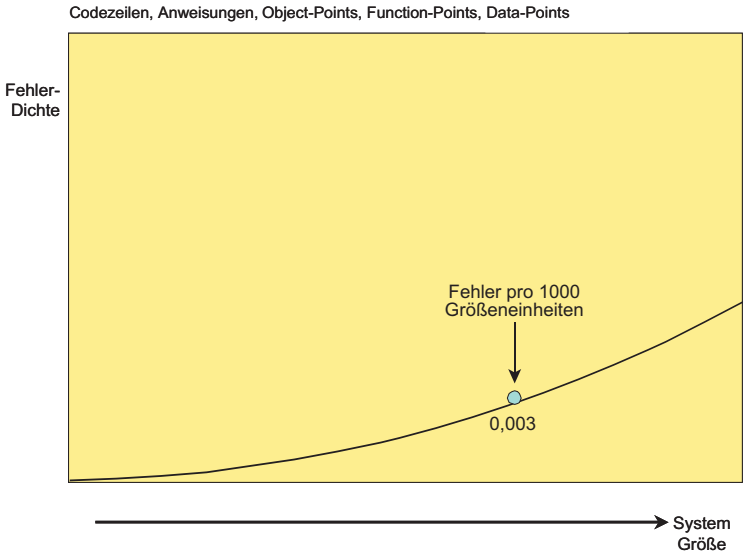


Abbildung 9.14 Fehlerdichte

9.6 Testauswertung des verteilten Kalenders

Der Testplan für den Kalendertest sah unterschiedliche Testendekriterien für den Klassentest, den Integrationstest und den Systemtest vor. Beim Klassentest sollten

alle Objektarten erzeugt und alle Member-Operationen ausgeführt werden. Beim Integrationstest sollten alle Nachrichtenarten getestet werden, einmal mit gültigem und einmal mit ungültigem Inhalt. Beim Systemtest sollten alle Anwendungsfälle erprobt und alle Fehlermeldungen des Systems ausgelöst werden. Dies waren zunächst relativ grobe Testziele.
Später im Testentwurf wurden diese Testziele verfeinert. Für den Klassentest (White Box) werden als Testziele angestrebt:

- Komplexität < 0,60
- Qualität > 0,60
- Konformität > 0,80
- Zweigüberdeckung > 0,90
- Zusicherungsüberdeckung > 0,99

Für den Integrationstest (Grey Box) sind es folgende Ziele:

- Objekttyperzeugnisrate > 0,90
- Methodenüberdeckung > 0,99
- Vererbungsüberdeckung > 0,99
- Assoziationsüberdeckung > 0,99
- Schnittstellenüberdeckung > 0,99

Für den Systemtest (Black Box) gilt es, folgende Testendekriterien zu erfüllen:

- Schnittstellenüberdeckung > 0,99,
- Anwendungsfall-Überdeckung > 0,99,
- Ausnahmeüberdeckung = 0,99 und
- Fehlerrate < 0,002

Diese Testendekriterien sind alle präzise quantifiziert und daher auch messbar. Allerdings braucht man dafür entsprechende Werkzeuge.

9.6.1 Klassentestauswertung

Zur Bewertung des Klassentests wird zum einen ein statischer Analysator gebraucht, der anhand des Source Codes Komplexität, Qualität und Konformität misst. Für die statische Analyse des Kalendersystems wird das Werkzeug CPPANAL benutzt, das die gewünschten Messwerte für jede Klasse sowie für die Summe aller Klassen in Form eines Metrikberichts liefert (Abbildung 9.15). Daran ist zu erkennen, ob die angestrebten Komplexitäts-, Qualitäts- und Konformitätsmaße erreicht wurden.

Code Quantity Metrics

Number of Source Members analyzed	11
Number of Lines in all	784
Number of Genuine Code Lines	358
Number of Comment Lines	188
Number of Major Rule Violations	65
Number of Medium Rule Violation	48
Number of Minor Rule Violations	230

Structural Quantity Metrics

Number of Modules	2
Number of Inludes	22
Number of Classes declared	6
Number of Classes inherited	3
Number of Methods declared	29
Number of Methods inherited	3
Number of Procedures declared	0
Number of Interfaces declared	29
Number of Object-Points	214

Data Quantity Metrics

Number of Panels processed	7
Number of Reports produced	1
Number of Defined Definitions	48
Number of Data Variables declared	55
Number of Data Variables inherited	5
Number of Data Constants / Enums declared	9
Number of Redefinitions (Unions)	0
Number of Arrays (Vectors)	16
Number of external Data Elements	0
Number of different Data Types used	32
Number of Data References	239
Number of Arguments / Input Variables	87
Number of Results / Output Variables	83
Number of Predicates / Conditional Daten	36
Number of Parameter / Function Arguments	33
Number of Data-Points	134

Procedural Quantity Metrics

Number of Statements	242
Number of Input Operations	6
Number of Output Operations	5
Number of Function References	50
Number of Foreign Functions referenced	36
Number of If-Statements	16
Number of Switch-Statements	0
Number of Case Statements	0
Number of Loop Statements	7
Number of GOTO Branches	0
Number of Return Statements	21
Number of Control Flow Branches	67
Number of all Control Statements	32
Number of Literals in Statements	53
Number of Nesting Levels (Maximum)	4
Number of Test Cases (Minimum)	64
Number of different Statements Type	131
Number of Assertions made	0
Number of Function - Points	118

Abbildung 9.15 Kalender-Metrikbericht

Zur Messung der Zweig- und Zusicherungsüberdeckung wird ein dynamischer Analysator benötigt, der den Source Code vor dem Klassentest instrumentiert und die Durchläufe registriert. Für die dynamische Analyse der Kalenderklassen wird das Werkzeug CPPINST eingesetzt. CPPINST misst sowohl die Zweigüberdeckung als auch die Zusicherungsüberdeckung und produziert einen Klassentestdeckungsbericht, aus dem hervorgeht, zu welchem Grade alle Zweige und Zusicherungen ausgeführt wurden (Abbildung 9.16). Daran ist erkennbar, ob die angestrebten Überdeckungsziele im Klassentest erreicht wurden.

9.6.2 Integrationstestauswertung

Zur Bewertung des Integrationstests wird der gleiche Instrumentor CPPINST benutzt, aber mit anderen Parametern. Anstelle der Ablaufzweige und Zusicherungen werden folgende Elemente instrumentiert:

- Methodeneingänge,
- vererbte Operationsaufrufe,
- fremde Operationsaufrufe und
- Schnittstellen

Test Coverage Report Date: 02.09.1998

System-Name: Calender
User-Name: Bahr
Module: Tag

(∅ = Missed)

Node-Nr.	Line-Nr.	Node-Name.		
000	00022	Konstruktor_Def/Tag	3	
001	00029	Destruktur_Def/Tag	0	∅
002	00033	Methode_Def/Tag/get_AKtivitätenliste	8	
003	00035	Logic/return	8	
004	00039	Methode_Def/Tag/add_Aktivität	6	
005	00040	Logic/if	0	∅
006	00040	Logic/return	0	∅
007	00042	Konstruktor_New/tasks/Task	0	∅
008	00044	Logic/return	0	∅
009	00048	Methode_Def/Tag/can_Aktivität	0	∅
010	00060	Logic/while	0	∅
011	00061	Methode_Inv/tasks/get_Start_Zeit	0	∅
012	00062	Metode_Inv/task/get_Ende_Zeit	0	∅
013	00063	Methode_Inv/task/get_Aktivität	0	∅
014	00070	Logic/if	0	∅
015	00075	Logic/else	0	∅
016	00075	Logic/if	0	∅
017	00075	Logic/return	0	∅
018	00080	Logic/while	0	∅
019	00085	Logic/return	0	∅

Nodes-Total: 20
Nodes-Execution: 4
Coverage Ration 20,00%

Abbildung 9.16 Überdeckung der Klasse `Tag`

Hinter jedem Methodeneingang, vor jedem Aufruf einer geerbten Operation, vor jedem Aufruf einer fremden Operation und als Parameter in jeder Schnittstelle wird ein Durchlaufzähler eingebaut. Außerdem werden in die Konstruktormethoden Zustandszähler eingebaut, um die Art der erzeugten Objekte zu registrieren.

9.6.3 Systemtestauswertung

Bei der Bewertung des Systemtests hat man es nicht mehr mit den Konstruktionselementen des Kalendersystems, sondern mit seiner Funktionalität zu tun. Die Funktionalität drückt sich in den Funktionsbäumen, in den Anwendungsfall-Tabellen und in den Testfallverzeichnissen aus. Hier hilft kein Automat, welcher den Source oder gar den Objektcode instrumentiert, denn die Funktionalität des Kalendersystems ist dort nicht messbar. Hier hilft nur ein Automat, der misst, welche externen Ereignisse ausgelöst werden, welche externen Datenflüsse eintreffen und welche Attribute sich in den Datenbanken verändern. Leider stößt die Testautomation bei letzterer Messung an ihre Grenzen. Es ist zwar möglich, die Zustände der Datenbankattribute zu registrieren, nicht jedoch die externen Ereignisse und Datenflüsse. Es sind Versuche im Gange, Capture/Replay-Werkzeuge in diese Richtung auszubauen, aber diese Versuche stecken noch in den Kinderschuhen. Also bleibt nichts anderes übrig, als mit Hilfe von Checklisten zu notieren, welche

Funktionen, Anwendungsfälle, Fehlerbehandlungen und Testfälle man getestet hat (Abbildung 9.17). Für jede Testmetrik wird eine separate Checkliste geführt, aus der hervorgeht, was bisher getestet wurde.

Anwendungsfall	Ereignis	Test	Ergebnis	Fehler
Kalender_anlegen	Mit ungültiger Person	X	Fehlermeldung	0
	mit gültiger Person	X	Weiter	0
	mit ungültiger Woche	X	Fehlermeldung	0
	mit gültiger Woche	X	Weiter	0
	mit ungültigem Wochentag	X	Fehlermeldung	0
	mit gültigem Wochentag	X	Weiter	0
	mit Abschluss	X	Kalender gespeichert	1
Aktivität_anlegen	mit falscher Anfangszeit	X	Fehlermeldung	0
	mit falscher Endzeit	X	Fehlermeldung	0
	mit falscher Bezeichnung	X	Fehlermeldung	1
	mit ungültigem Projekt	X	Fehlermeldung	1
	mit gültigem Projekt	X	Stunden gebucht	2
	mit 13 Aktivitäten	-	-	-
Aktivität_löschen	mit einer Aktivität	-	-	-
		13		5

Abbildung 9.17 Kalendertest Checkliste

Das Gleiche gilt für die Fehlerrate. Hier werden die im Systemtest gemeldeten Fehler erfasst und gezählt. Da man aufgrund der statischen Analyse weiß, wie groß das Kalendersystem ist, kann die Fehlerdichte leicht errechnet werden. Es ergibt sich, dass noch immer zu viele Fehler pro Object-Point vorhanden sind (Abbildung 9.18). Also müsste der ganze Test wiederholt werden, um hoffentlich dieses Mal weniger Fehler aufzudecken.

Es dürfte klar geworden sein, wie aufwändig es ist, objektorientierte, verteilte Software zu testen, vor allem wenn man den Anspruch erhebt, systematisch zu testen. Systematisch heißt gegen Endekriterien testen, und Endekriterien sind Testmetriken. Um den Aufwand in Grenzen zu halten, braucht man unbedingt ein werkzeuggestütztes Regressionstestverfahren, um den gleichen Test mit geringfügigen Änderungen und Erweiterungen wiederholen zu können. Dies ist das Thema des nächsten Kapitels.

Fehlerzählung

Klasse	Fehler	Schwere Fehler	Mittlere Fehler	Geringe Fehler
Mitarbeiter	3	2	1	
Kalender	3		1	2
Woche	1	1		
Tag	1		1	
Aktivität	2	1	1	
Projekt	1	1		

Fehlerdichte

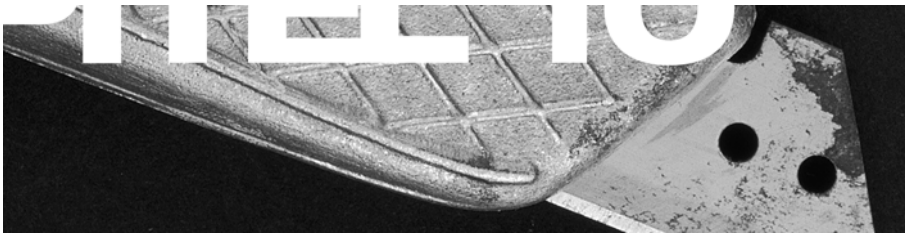
Klasse	Lines	Stmts	Obj.-Points	Funct.-Points
Mitarbeiter	0,009	0,018	0,027	0,180
Kalender	0,012	0,024	0,036	0,240
Woche	0,010	0,020	0,030	0,200
Tag	0,008	0,016	0,024	0,160
Aktivität	0,011	0,022	0,033	0,220
Projekt	0,015	0,030	0,045	0,300

$$\frac{\text{Fehler}}{\text{Zeilen}} \quad \frac{\text{Fehler}}{\text{Anweisungen}} \quad \frac{\text{Fehler}}{\text{Obj.-Points}} \quad \frac{\text{Fehler}}{\text{Funct.-Points}}$$

Abbildung 9.18 Kalendersystem Fehleranalyse

10

Regressionstest



Iterative, inkrementelle Softwareentwicklung
Bedeutung des Regressionstests
Forschung zum Thema Regressionstest
Konventionelle Regressionstesttechniken
Regressionstest der Klassen und Komponenten
Capture/Replay Technik
Regressionstest des verteilten Kalenders

Inhaltsübersicht Kapitel 10

10 Regressionstest

10.1 Iterative, inkrementelle Softwareentwicklung

In Anbetracht der iterativen Natur der meisten objektorientierten Projekte hat der Regressionstest an Bedeutung zugenommen. Es ist kaum möglich, eine komplexe objektorientierte Anwendung auf Anhieb zu erstellen. In der Regel werden zwei bis vier Entwicklungszyklen benötigt, bis das Anwendungssystem einigermaßen freigabereif ist [Lor93]. Das hat zur Folge, dass auch der Test mehrfach wiederholt werden muss (Abbildung 10.1). Jeder Durchgang durch den Entwicklungszyklus ist ein eigenständiges Projekt mit einem neuen Release als Ergebnis und zu jedem Entwicklungsprojekt gehört der Test im Sinne des „Eisenbahnmodells“ [KoPo99]oder „W-Modells“ [Spi00] als Parallel- oder Schattenprojekt.

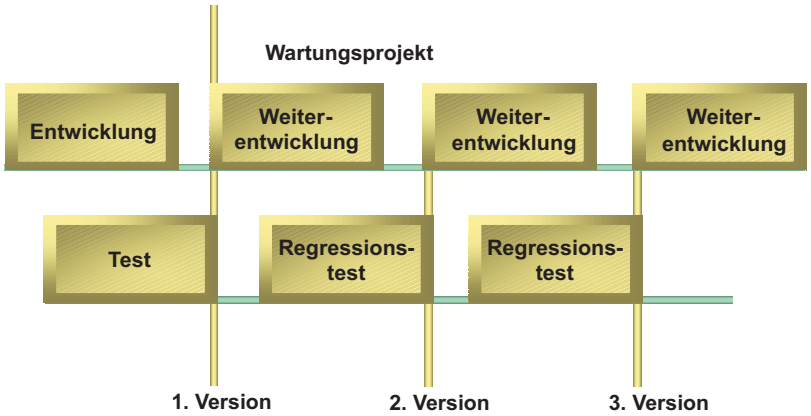


Abbildung 10.1 Bedeutung des Regressionstests im Entwicklungszyklus

Die Wartung und Weiterentwicklung von Software tritt sofort nach der ersten Freigabe ein. Ab diesem Zeitpunkt geht der Lebenszyklus eines Softwaresystems in einen anderen Modus über. Alles wird vom bereits Vorhandenen geprägt, auch der

Test. Das oberste Prinzip besteht darin, nicht zu viel auf einmal zu ändern, sondern die Software nur in kleinen Inkrementen fortzuschreiben.

Im allerersten Projekt wird eine Testumgebung für die Zielapplikation aufgebaut. Dies erfordert natürlich eine nicht unerhebliche Vorinvestition, die sich aber später mehrfach amortisiert. In den Folgeprojekten wird im Produktlebenszyklus diese Testumgebung ständig ausgebaut und verbessert. Der Testplan wird angepasst, das Testkonzept überarbeitet, die Testfälle werden erweitert und die Testprozeduren fortgeschrieben. Ebenso wie die Programme wird also auch der Test gewartet und weiterentwickelt. Insofern ist ein Applikationstest wie die Applikationsentwicklung selbst ein endloser Prozess. Beide evolvieren so lange, bis sie an ihre Grenzen stoßen, d.h. beide unterliegen den Gesetzen der Software-Evolution von Belady und Lehman [LeBe85].

Daraus folgt, dass mindestens 2/3, wenn nicht viel mehr, der gesamten Lebenszykluskosten nach dem ersten Projekt entstehen. So betrachtet ist das erste Release fast immer nur ein Prototyp, auch wenn es als solcher nicht gedacht ist. Dies trifft vor allem für Informationssysteme zu. Moderne Betriebe sind in einem ständigen Wandlungsprozess begriffen. Externe Einflüsse wie Globalisierung, Marktbewegungen und neue Technologien zwingen sie, sich ständig anzupassen. Man spricht im Englischen von *Emergent Organizations*. In dem Maße, wie die Betriebsstrukturen sich verändern, müssen sich auch die IT-Strukturen ändern, um Schritt zu halten. Da Softwaresysteme nur eine Abbildung der betrieblichen Wirklichkeit sind, werden sie ebenfalls nie fertig. Sie haben immer nur einen vorläufigen Stand [Sne90].

Dies erfordert neue Ansätze zum Projektmanagement. Statt langen Projekten mit einer Laufzeit von 2–4 Jahren, dürfen nur kurze Teilprojekte bis zu einem Jahr geplant werden. Jedes Teilprojekt liefert ein lauffähiges Teilprodukt, d.h. ein Stück des Ganzen. Die ersten Teilprodukte werden eventuell gar nicht zum Einsatz kommen. Sie werden nur erprobt und dann zurückgestellt. Irgendwann werden die Teilprodukte einen Umfang und eine Reife erlangen, die es erlaubt, sie produktiv einzusetzen. Dennoch – auch dann sind sie noch lange nicht fertig. Nachdem sie in Produktion sind, werden sie längere Zeit erweitert, geändert, korrigiert und optimiert. Hier spricht man von

- korrektiver,
- adaptiver,
- enhansiver und
- perfektiver Wartung [LiSw80].

Korrektive Wartung beinhaltet die Korrektur nach gemessenen Fehlern bzw. Fehlerbehebung. *Adaptive Wartung* umfasst die Anpassung vorhandener Daten und Funktionen bzw. Änderungen. *Enhansive Wartung* bedeutet die Ergänzung der Software durch zusätzliche Daten und Funktionen bzw. die Erweiterung. *Perfektive*

Wartung ist die Summe aller Maßnahmen zur technischen Verbesserung der Software wie Optimierung, Sanierung, Refakturierung usw.
In allen Fällen beginnt das Wartungsprojekt mit einer Impaktanalyse und endet mit einem Regressionstest. Die Impaktanalyse soll die Auswirkung der geplanten Änderung ermitteln. Der Regressionstest soll die Auswirkung der durchgeführten Änderung bestätigen.
All das wirkt sich auf den Test aus. Der Test darf nicht so geplant und konzipiert werden, als ob er nur einmal stattfindet. Im Gegenteil, der Test muss vom Anfang an so ausgelegt sein, dass er sich mehrfach wiederholen lässt. Die Investitionen, die man beim ersten Projekt für den Test tätigt, werden sich bei den Folgeprojekten mehrfach auszahlen. Es kommt daher darauf an, eine flexible und ausbaufähige Testumgebung aufzubauen mit Testdokumenten und Testfällen, die sich leicht fortschreiben lassen. Dadurch wird der Aufwand für den Test im Laufe der Produktentwicklung immer weniger. Dies lässt sich anhand der sinkenden Kosten pro Testfall messen. Bei jedem sukzessiven Projekt müsste die Fehlerrate sinken und die Testproduktivität steigen. Wenn nicht, ist der Testprozess nicht ausreichend definiert. Regressionsprozesse sollten mindestens sowohl definiert als auch wiederholbar sein ([Eva84][KoPo99][KPS00]).

10.2 Bedeutung des Regressionstests

Jeder Test nach dem ersten ursprünglichen Test im ersten Lebenszyklus ist im Prinzip ein Regressionstest. Wenn im ersten Projekt die Testaufwände mindestens 40% der gesamten Aufwände ausmachen, sind sie in den Folgeprojekten eher 60%. Da die Weiterentwicklungskosten inzwischen mehr als 75 % der gesamten Lebenszykluskosten ausmachen, bedeutet dies, dass die Regressionstestkosten allein mehr als 45% der Lebenszykluskosten ausmachen. Dies ist mehr als irgendeine andere Projekttätigkeit [LiHe95].
Eine Einsparung von lediglich 20% der Regressionstestkosten ist eine Ersparnis von fast 10% der Lebenszykluskosten. Wenn überhaupt eine Aktivität automatisiert werden sollte, ist es der Regressionstest. Damit ist die größte Kostenreduktion im gesamten Lebenszyklus zu erreichen. Dies erklärt das wachsende Interesse am Regressionstest nicht nur in der Praxis, sondern auch in der Informatikforschung [RoHa96].

10.3 Forschung zum Thema Regressionstest

Die Forschung zum Thema *Regressionstest* begann bereits im Jahre 1977. Damals auf der COMPSAC-Konferenz in Dallas präsentierte H. Fischer ein Schema für Pfadanalyse auf der Basis von *Program Slicing*, nach dem jene Pfade durch die

Programme identifiziert werden, die von einer bestimmten Änderung betroffen sind. Laut der These Fischers könnte der Wiederholungstest sich ausschließlich auf diese Pfade konzentrieren. Bis dahin galt es als erforderlich, bei Änderungen das ganze Programm nochmals voll auszutesten [Fis77].

Einige Jahre später wurde die Originalthese von Fischer durch die Informatikforscher Yau und Kishimoto aufgegriffen und durch weitere Pfadanalysetechniken wie Eingabepartitionierung, Ursache/Wirkungsgraphen, symbolische Ausführungsbäume und Zustandstabellen erweitert, um die Testdaten für ausgewählte Pfade zu bestimmen [YaKi87]. Leung und White haben ebenfalls die These von Fischer angenommen und den Begriff *Impact Domain* zur Bezeichnung des Auswirkungsbereichs einer Änderung eingeführt. *Impact-Analyse* ist die Untersuchung des Source-Codes mit dem Ziel, den Auswirkungsbereich abzustecken [LeWi89].

Anfang der 80er Jahre brachte M. Weiser einen Artikel über seine Doktorarbeit zum Thema *Program Slicing* heraus. Die *Slicing*-Technik sollte ein wichtiger Beitrag sowohl zur Wartungs- als auch zur Testtechnologie sein. Demnach wird ein Programmablaufgraph in Teilgraphen zerlegt, die getrennt behandelt werden können. Da Änderungen in der Regel nur einen Teilgraph betreffen, muss nur dieser getestet werden [Wei84].

Im Rahmen des ESPRIT TRUST-Projekts hat einer der Autoren zusammen mit Dr. Peter Puhr vom Kernforschungszentrum Garching die *Slicing*-Technik angewandt, um Code zu strippen. *Code Stripping* ist eine Regressionstesttechnik, nach der alle Source-Code-Anweisungen, die nicht auf dem Zielpfad liegen, in Kommentarzeichen gesetzt werden, sodass nur Anweisungen an dem markierten Pfad kompiliert werden. Der Rest ist ausgeblendet. Als Folge wird nur ein Teilprogramm ausgeführt (Abbildung 10.2). Der Test beschränkt sich zwangsläufig auf diesen Teilgraphen. Die Markierung der durch die Änderung betroffenen Programmpfade erfolgt automatisch anhand der Rückverfolgung aller Kanten, die zu den geänderten Knoten führen – *backwards slicing* – und die Verfolgung aller Kanten, die von den geänderten Knoten ausgehen – *fowards slicing*. Dadurch konnten eingebettete Echtzeitprogramme pfadweise getestet werden, ohne ihre Performanz zu beeinträchtigen [SnRi93].

Ende der 80er Jahre wurde Regressionstesten ein zentrales Forschungsthema. In England haben Hartmann und Robson den Pfadanalysealgorithmus von Fischer ergänzt, um Pfade über Modulgrenzen hinaus zu verfolgen. Dies erlaubte eine selektive Revalidation ganzer Systeme [HaRo90]. In Italien haben Benedusi, Cimitile und DeCarlina einen Algorithmus erfunden, um zwischen gelöschten, geänderten und hinzugefügten Ablaufpfaden zu unterscheiden. Dieser Unterscheidungsalgorithmus bildete die Basis für eine Pfadmutationsanalyse, die dazu führte, Regressionstestfälle aus dem Source-Code abzuleiten [BCD89].

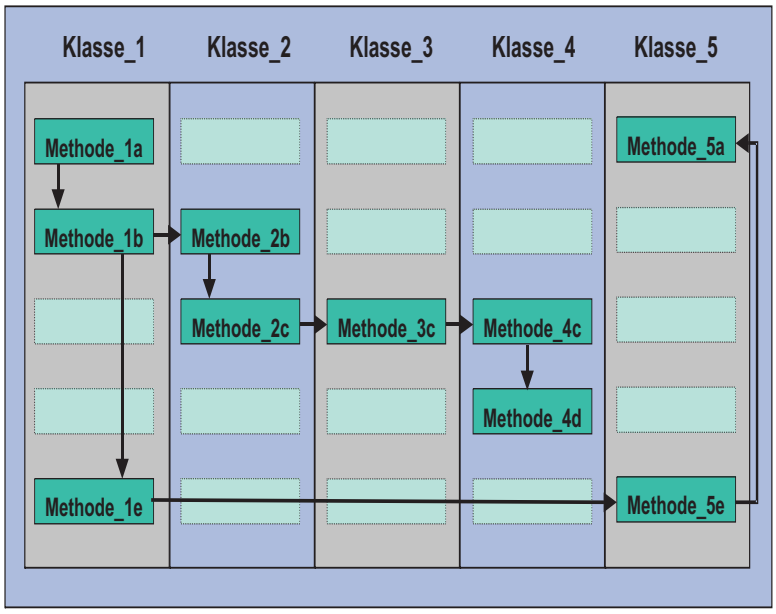


Abbildung 10.2 Regressionstestpfadauswahl

Inzwischen entstand ein weiterer Ansatz zum Regressionstest, und zwar über die Datenflussanalyse. Leung und White waren die Ersten, die diesen Weg einschlugen. Danach wird nicht die Auswirkung der Änderungen auf den Steuerfluss, sondern ihre Auswirkung auf den Datenfluss analysiert. Man erkennt, welche Daten betroffen sind [LeWi90]. Die Spezifikation der Regressionstestfälle auf der Grundlage der Datenflussanalyse wurde durch die Forschungsarbeiten von Ntafos [Nta84], Weyucker [RaWe89], Korel [LaKo83] und Clark [PoCl90] vorangetrieben und ist heute weitgehend automatisiert. Danach werden Programmergebnisse auf die ursprünglichen Eingaben zurückverfolgt und die anderen betroffenen Variablen einbezogen. Datenfluss- und Steuerflussanalyse wurden inzwischen zusammengeführt, sodass sie sich bei der Absteckung des Wirkungsbereichs einer Änderung gegenseitig ergänzen.

10.4 Konventionelle Regressionstesttechniken

Im Jahre 1992 hat einer der Autoren den damaligen Stand der Regressionstesttechnologie für prozedurale Softwaresysteme in einem Bericht auf der amerikanischen Testkonferenz STAR zusammengefasst und fünf Axiome zum Regressionstest im Allgemeinen formuliert:

- Die Eingabe/Ausgabe-Bereiche bleiben bis auf jene Variablen bzw. Felder, die gestrichen oder hinzugefügt werden, invariant.

- Die Datenverwendung bzw. der Datenfluss bleibt bis auf jene Datenreferenzen, die gelöscht, geändert oder hinzugefügt wurden, unverändert.
- Die Programmablaufpfade bleiben, bis auf die geänderten Bedingungen, identisch.
- Die Eingabe/Ausgabesequenzen bleiben bis auf zusätzliche oder gelöschte IO-Operationen unverändert.
- Die Datenbankzustände müssen bis auf die geänderten, gelöschten oder hinzugefügten Werte die gleichen sein [Sne92].

Diese fünf Axiome lassen sich durch fünf verschiedene Validierungstechniken zum Regressionstest bestätigen. Im Prinzip geht es darum, funktionale Äquivalenz nachzuweisen. Die neue Programmversion soll bis auf die wenigen geänderten Elemente funktional äquivalent zur alten Version sein. Alle fünf Techniken gleichen Eigenschaften neuer Programmversionen gegen die der alten Versionen ab. Es handelt sich dabei um folgende Abgleiche (Abbildung 10.3):

- Abgleich der Datenstrukturen,
- Abgleich der Datenverwendung,
- Abgleich der Ablaufpfade,
- Abgleich der IO-Sequenzen und
- Abgleich der Datenbanken.

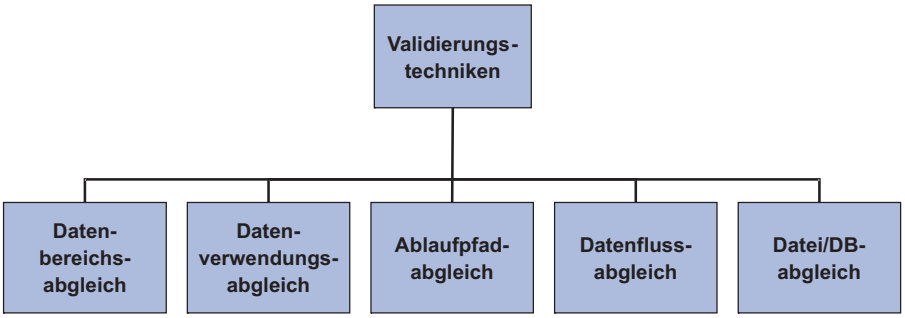


Abbildung 10.3 Validierungstechniken

10.4.1 Abgleich der Datenstrukturen

Der Abgleich der Datenstrukturen geschieht mittels der statischen Analyse. Beide Programmversionen – die ursprüngliche und die geänderte – werden analysiert, um die Eingabedaten und die Ausgabedaten abzuleiten (Abbildung 10.4). Beide Listen werden miteinander verglichen. Die Eingaben und Ausgaben der alten Programmversion müssen mit denen der neuen Programmversion bis auf die gelöschten oder

hinzugefügten Variablen übereinstimmen. Wenn die beiden Datenstrukturen gleich sind, sind die beiden Programmversionen funktional äquivalent.

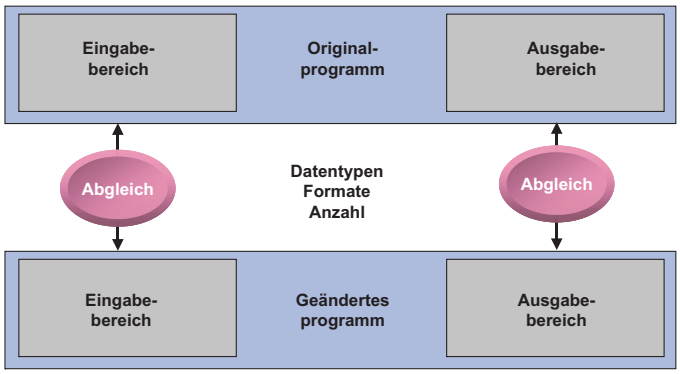


Abbildung 10.4 Datenbereichsabgleich

10.4.2 Abgleich der Datenverwendung

Der Abgleich der Datenverwendung ist auch eine Folge der statischen Analyse. Dem Source-Code wird entnommen, welche Variablen von welchen Anweisungen in welchen Prozeduren wie verwendet werden (Abbildung 10.5). Eine Variable kann benutzt, abgefragt, gesetzt oder weitergereicht werden. Wenn alle Variablen in der gleichen Reihenfolge in der gleichen Art und Weise verwendet werden, ist der Code äquivalent. Den Nachweis bringt der Vergleich der alten mit den neuen Datenreferenzen. Nur dort, wo das Programm geändert wurde, dürfen die Referenzen voneinander abweichen.

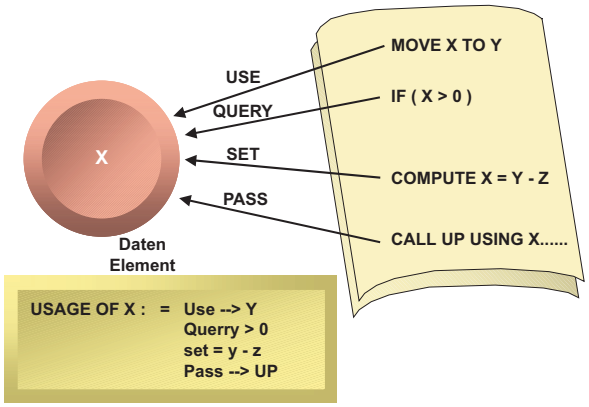


Abbildung 10.5 Datenverwendungsabgleich

10.4.3 Abgleich der Ablaufpfade

Der Abgleich der Ablaufpfade setzt eine dynamische Analyse der beiden Programmversionen voraus. Beide Programmversionen werden mit den gleichen Testdaten ausgeführt und ihre Durchlaufpfade aufgezeichnet (Abbildung 10.6). Anschließend werden die Ablaufpfade verglichen. Dort, wo die Pfade voneinander abweichen, sind die Programmversionen ungleich. Entweder ist dies eine Folge der Änderung oder eines Fehlers, denn zwei funktional äquivalente Programme müssen die gleiche dynamische Ablauffolge haben, auch dann, wenn sie statisch unterschiedlich zusammengesetzt sind.

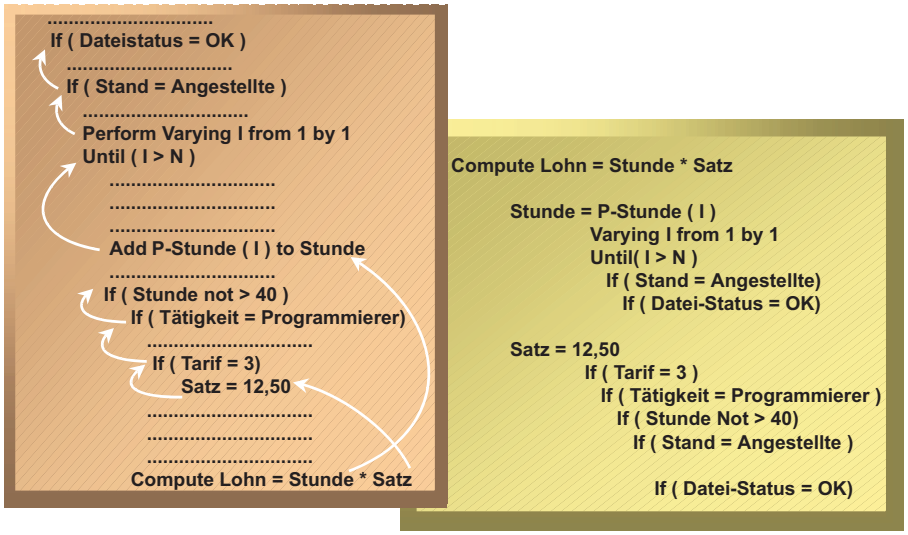


Abbildung 10.6 Ablaufpfadabgleich

10.4.4 Abgleich der IO-Sequenzen

Der Abgleich der IO-Sequenzen ist ebenfalls ein Ergebnis der dynamischen Programmanalyse. Statt aber die ganzen Pfade über alle Verzweigungen hinweg zu verfolgen, wird nur die Folge der IO-Operationen bzw. der Datenbankzugriffe zum Vergleich aufgezeichnet (Abbildung 10.7). Diese *lightweight* Pfadanalyse weist viel weniger Vergleichspunkte auf und ist deshalb leichter zu implementieren. Um als funktional äquivalent zu gelten, müssen lediglich die gleichen Ein/Ausgabe-Operationen in der gleichen Sequenz folgen. Abweichungen deuten hier auf grobe Veränderungen zum Programm.

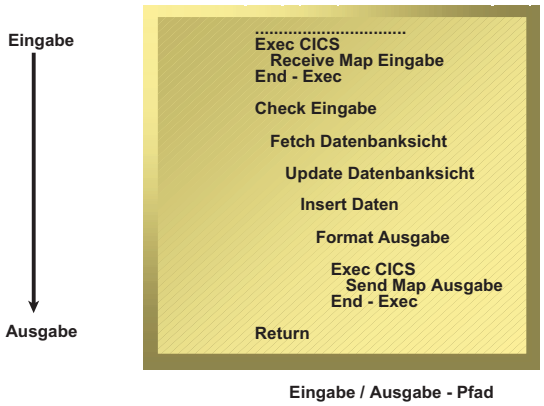


Abbildung 10.7 Datenflussabgleich

10.4.5 Abgleich der Datenbanken

Der Abgleich der Datenbanken bzw. der Dateien ist die konventionelle Art, funktionale Äquivalenz nachzuweisen. Zuerst wird die alte Programmversion ausgeführt, um einen Referenzzustand zu erzeugen. Danach wird die neue Programmversion mit den gleichen Eingaben, doch auf einer anderen Datenbank ausgeführt. Anschließend werden die beiden Datenbanken bzw. Dateien miteinander satzweise oder zeilenweise verglichen (Abbildung 10.8). Falls Sätze oder Zeilen nicht übereinstimmen, deutet dies auf eine Änderung oder einen Fehler. Es obliegt dem Tester nachzuweisen, welcher dieser beiden Fälle zutrifft.

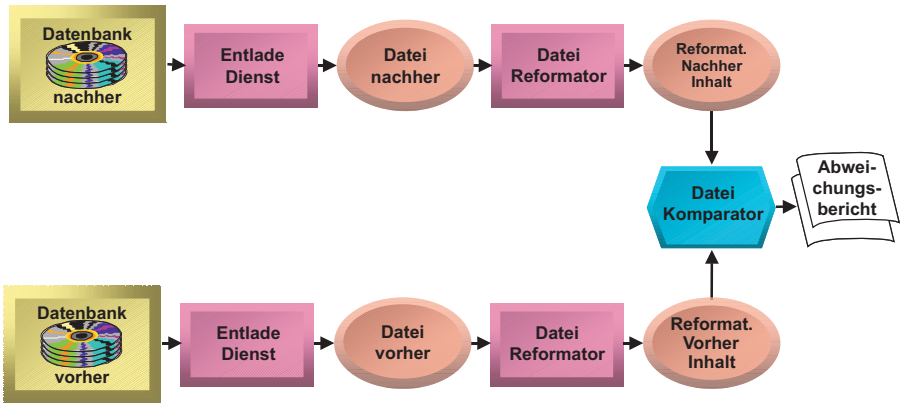


Abbildung 10.8 Datei/DB Abgleich

10.4.6 Abgleich prozeduraler Programme

Für prozedurale Programme hat einer der Autoren alle fünf Abgleichsmöglichkeiten in einem Regressionstestwerkzeug namens SofRetest realisiert. Mit SofRetest werden die Programme erstens statisch analysiert, zweitens dynamisch analysiert und drittens ihre Datenergebnisse analysiert. Bei der statischen Analyse werden sowohl die Datenstrukturen als auch die Datenverwendung der alten mit denen der neuen Programmversion abgeglichen und Abweichungen ausgewertet (Abbildung 10.9). Bei der dynamischen Analyse werden die Ablaufpfade und IO-Sequenzen aufgezeichnet und verglichen. Auch hier werden Differenzen dokumentiert. Schließlich werden die Dateien und Datenbanken der beiden Programmversionen abgeglichen und Felder mit unterschiedlichen Werten protokolliert. Damit bekommt der Tester einen umfassenden Überblick über die Unterschiede zwischen Programmversionen – Unterschiede, die entweder gewollt oder ungewollt sind. Im Falle einer Programmtransformation beweist Softretest die funktionale Äquvalenz der neuen Version. Der Haupzweck ist der Regressionstest in Reengineering-Projekten.

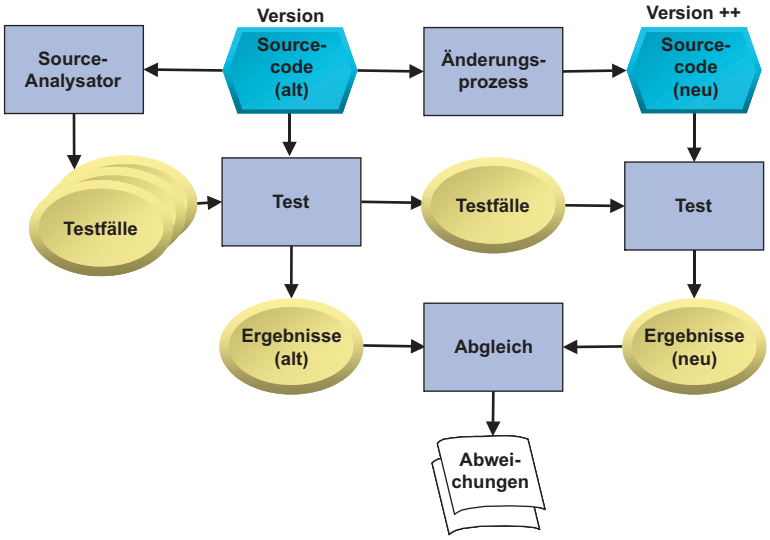


Abbildung 10.9 Abgleich zweier Programmversionen

10.4.7 Objektorientierte Regressionstesttechniken

Für den Regressionstest objektorientierter Systeme sind die oben erwähnten Techniken nur bedingt einsetzbar. Hsia und Kung stellen fest,

> *"..software revalidation is a complex and expensive activity. The introduction of the object-oriented paradigm makes this issue even more complicated and difficult to deal*

with. The problem is due to the use of numerous OO features such as inheritance, messaging, polymorphism and dynamic binding..". [HLK+97]

Der Nutzen der statischen Analyse ist wegen der Polymorphie und der Umbenennung der Parameter ohnehin stark eingeschränkt. Hinzu kommt die Verwendung von Zeigern in C++, die eine exakte Identifizierung der Parameter weiter erschwert. Es bleibt also nur die dynamische Analyse als Basis für den Abgleich alter und neuer Programmversionen [HLK+97].

Die Pfadanalyse ist nach wie vor gültig, aber nur insofern, als die Polymorphie vom Tester determinierbar ist. Die Auswahl der aufzurufenden Operation muss immer nachvollziehbar und wiederholbar sein. Sie darf nicht von der Speicherallokierung oder der Zeit abhängig sein. In diesem Fall ist es sehr wohl möglich, Pfade als Sequenzen von Operationsaufrufen festzuhalten. Darüber hinaus lässt sich durch die geeignete Instrumentierung die Reihenfolge der Ablaufzweige innerhalb der Operationen verfolgen. Somit hat der Tester die Möglichkeit, Pfade durch die Objekte der letzten Version mit Pfaden durch die Objekte der nächsten Version abzugleichen. Die Sequenz der IO-Operationen ist ebenfalls in verschiedenen Versionen der gleichen Komponente vergleichbar.

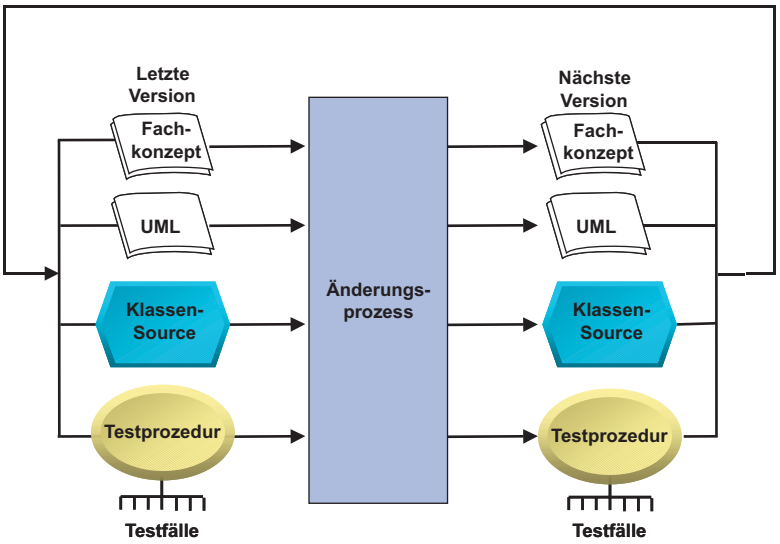


Abbildung 10.10 Weiterentwicklungzyklus

Der Abgleich der Datenbanken und Dateien ist unabhängig davon, ob die Programme prozedural oder objektorientiert sind. Er findet außerhalb der eigentlichen Software statt (Abbildung 10.10). Das Gleiche trifft für die Benutzungsoberfläche zu. Auch Oberflächen lassen sich aufzeichnen und abgleichen. Hier werden die meisten Regressionstestwerkzeuge eingesetzt. Sie zeichnen den Inhalt der Oberflächen der Vorgängerversion auf und gleichen sie nachher mit den Oberflächen der

Nachfolgeversion ab. Jede Abweichung in den Oberflächen, die nicht zeit- oder datumsbedingt ist, deutet auf eine Änderung oder einen Fehler hin. Wenn die Änderungen zu umfangreich sind, stimmt natürlich kaum noch etwas überein, sodass der Wert der Vergleiche abnimmt. Deshalb ist es zu empfehlen, immer nur kleine Änderungen auf einmal durchzuführen, da sonst die Vergleichsbasis verloren geht.

10.4.8 Objektorientierte Regressionstesttechniken in der Forschung

In der Forschung haben sich die amerikanischen Professoren Mary Jane Harrold, Phyllis Frankl und Annaliese von Mayrhauser bei der Entwicklung von Regressionstesttechniken für objektorientierte Systeme verdient gemacht. Harrold hat mehrere Ansätze erprobt, um die Anzahl irrelevanter Testfälle zu reduzieren. Die bekannteste davon ist die *Test Suite Minimization Method*, um jede Methodensequenz nur einmal zu durchlaufen. Alle weiteren Wiederholungen derselben Methodensequenz werden eliminiert, da sie als redundant gelten. Es hat sich gezeigt, dass ein Test durch die Sequenz in der Regel für alle anderen stellvertretend ist. Bestätigt wurde diese These durch die Messung der Testüberdeckung und die Aufdeckung eingepflanzter Fehler. Es werden zwar nicht alle Fehler aufgedeckt, doch genug relativ zum Testaufwand. In dem Experiment wurden 73% der eingepflanzten Fehler gefunden [RoHa94].

Harrold benutzt Ablaufgraphen, um die Unterschiede zwischen alten und neuen Codeversionen aufzuweisen. Anhand der Ablaufgraphen wird erkannt, wo die Operationssequenzen und die Entscheidungslogik innerhalb der Operationen divergieren. Phyllis Frankl schlägt vor, die Source-Texte abzugleichen. Ihre Methode heißt *Textual Differencing*. Durch den Abgleich der beiden Quellcodetexte wird jede Anweisungsänderung erkannt, auch die Nutzung anderer Variablennamen. Mit Hilfe eines Test-Werkzeugs namens *Pythica* wird für jede solche Abweichung ein Testfall generiert, der diese Stelle im Code erreicht. Das Experiment mit C++-Programmen hat gezeigt, dass die Fehlerfindungsrate annähernd 100% ist. Nur einer von 33 bekannten Fehlern blieb unentdeckt. Dafür wurde die ursprüngliche Anzahl Testfälle, d.h. die Anzahl, die erforderlich wäre, um den ganzen Code zu überdecken, um mehr als 80% reduziert. Das heißt, es ist durch Textabgleiche möglich, mit nur 20% der Testfälle 97% der Fehler zu finden [FHL+98].

Annaliese von Mayrhauser und ihre Kollegen an der Colorado State University haben ein Systemregressionstestwerkzeug namens *Sleuth* entwickelt. *Sleuth* wird eingesetzt, um die Chip-Logik von Hewlett-Packard-Platten zu testen. Dafür wird die gesamte Entscheidungslogik von über 100.000 C++-Codezeilen in Bitmap-Dateien abgebildet. Immer wenn der Code sich verändert, werden die neuen mit den alten Bitmaps abgeglichen, um Differenzen festzustellen. Dann wird durch rekursive „data slicing“-Algorithmen der Weg zurück zu den Eingabedaten verfolgt. Die

Änderungen im Eingabebereich ergeben die Untermenge der Testfälle, die für den Regressionstest der Software zu verwenden sind. So konnte der Regressionstest für eine neue Version auf einen Bruchteil des ursprünglichen Gesamttests reduziert werden, ohne die Fehlerfindungsrate negativ zu beeinflussen [MaZh99].

In einem Beitrag für die 20. Internationale Software-Engineering-Konferenz in Kyoto mit dem Titel "An Empirical Study of Regression Test Selection Techniques" hat Mary Jean Harrold den Stand der Regressionstesttechniken zusammengefasst. Dabei handelt es sich um fünf Technikklassen:

- Minimization-Techniken,
- Safe-Techniken,
- Dataflow-Coverage-Based-Techniken,
- Ad hoc/Random-Techniken und
- Retest-All-Techniken.

In einem Experiment an der Ohio State University wurden diese Techniken miteinander verglichen. Daraus folgte, dass keiner der Ansätze alle Ziele erreicht. Die Minimization-Techniken sind zwar billig, aber unsicher. Die Safe-Techniken sind sicher, aber teuer. Die Dataflow-Techniken sind teuer und relativ unsicher. Die Random-Techniken sind billig, aber absolut unzuverlässig. Die Retest-All-Techniken sind am sichersten, aber auch am teuersten. Fazit ist, dass der Anwender entscheiden muss, was für ihn wichtiger ist: Zuverlässigkeit oder Wirtschaftlichkeit [RoHa98].

Einer der Autoren hat eine Methode zum Regressionstest entwickelt, die auf dem selben interaktionsbasierten Modell des Programms basiert, mit dem auch eine Integrationsstrategie berechnet werden kann [Win00]. Hierbei werden sowohl die Aufruf- als auch die Dateninteraktionen betrachtet. Die Verwendung ein- und desselben Modells sowohl für die Ermittlung der Integrationsstrategie als auch der Regressionstests reduziert dabei den Testaufwand. Dies kommt dem inkrementellen, iterativen Entwicklungsprozess entgegen, in welchem ja in jeder Iteration sowohl die aus den Änderungen und Erweiterungen resultierenden Klassen und Komponenten integriert als auch Regressionstests für die bereits vorhandenen Funktionalitäten durchgeführt werden müssen.

10.5 Regressionstest der Klassen und Komponenten

Für den Entwickler kommt es darauf an, korrigierte oder geänderte Programmbausteine wieder zu bestätigen. Er muss sicher sein, dass die Korrekturen, Änderungen und Erweiterungen keine unerwünschten Nebeneffekte haben. Da er nur wenig Zeit hat, kommt es darauf an, die bisherigen Testläufe möglichst kosteneffektiv durchzu-

führen. D.h. man braucht auch Kriterien, um die Effektivität der Regressionstesttechniken zu messen [RoWe97].

Wenn eine Klasse geändert wird, müssen nicht nur diese Klasse, sondern auch alle erbenden Klassen und alle aufrufenden Klassen nochmals getestet werden. Zunächst wird die eine Klasse allein für sich im Rahmen des Klassentests bestätigt. Falls nur einzelne Methoden geändert werden, braucht man nur jene Methoden anzusteuern, die sich verändert haben oder die z.B. auf geänderte Instanzvariablen zugreifen. Falls Klassenvariablen geändert wurden, müssen alle Methoden, die diese Variablen referenzieren, getestet werden. Mit einem Klassentesttreiber ist es relativ einfach, einzelne Methoden anzusteuern. Mit einem Build-in-Test muss der Entwickler gezielt jene Methoden aktivieren, die betroffen sind. Vorher sollte die alte Klasse getestet, die alten Abläufe durch die Methoden registriert und die erzeugten Objektzustände aufbewahrt werden. Anschließend wird der gleiche Test mit der geänderten Klasse wiederholt. Danach ist es möglich, die Sequenzen der Methodenabläufe und die Zustände der erzeugten Objektinstanzen abzugleichen. Der Bezugspunkt ist also das Verhalten der alten Klasse. Die neue Klasse darf nur dort davon abweichen, wo die Änderung dies auch verursachen soll. Ein Werkzeug für den automatischen Abgleich der Methodenpfade und Objektzustände kann natürlich von großem Nutzen sein und viel Zeit einsparen.

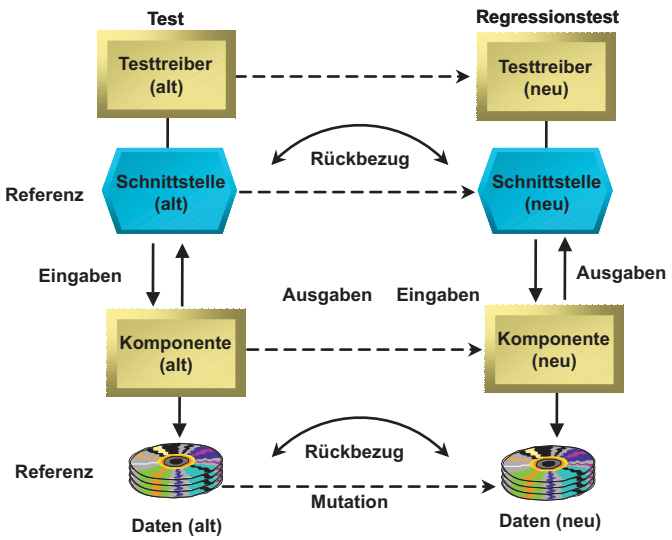


Abbildung 10.11 Rückbezug auf den letzten Test

Nachdem die geänderte Klasse bestätigt ist, geht der Entwickler weiter und testet alle Klassen, die von dieser Klasse erben. Dabei kommt es darauf an, die Abhängigkeiten von der geänderten Klasse zu prüfen. Falls die Klasse in mehreren Komponenten verwendet wird, gilt es, sämtliche betroffenen Komponenten zu testen.

Der Regressionstest muss dafür sorgen, dass alle Methoden in Klassen, die von der geänderten Klasse erben, durchlaufen werden. Darüber hinaus müssen alle assoziierten Methoden, die Methoden der geänderten Klasse aufrufen, ebenfalls erreicht werden. Am besten ist es, diese Methoden über einen Testtreiber direkt anzusteuern. Ansonsten muss man versuchen, sie über die Komponentenschnittstelle zu erreichen. Hier ist das Ziel, das Verhalten der geänderten Komponente mit dem der alten Komponente zu vergleichen (Abbildung 10.11). Je kleiner die Komponenten sind, desto leichter wird es, dieses Ziel zu erreichen.

Das Verhalten der Komponente wird zum Ersten durch die Ablaufpfade, zum Zweiten durch die persistenten Objektzustände und zum Dritten durch die Ausgangsparameter festgehalten. Zur Protokollierung der Ablaufpfade wird ein Trace-Werkzeug bzw. ein dynamischer Analysator benötigt. Zum Abgleich der persistenten Objektzustände kann ein Datenbankkomparator eingesetzt werden. Die Festhaltung der Ausgangsparameter ist die Aufgabe des Testtreibers, der nach jeder Transaktion die Parameterwerte in eine Arbeitsdatei schreibt. Anschließend lassen sich die Daten vor und nach der Modifikation abgleichen.

Durch das sorgfältige Abgleichen der Abläufe, Zustände und Ergebnisse der geänderten Klassen und Komponenten mit denen der ursprünglichen Klassen und Komponenten kann der Entwickler verhindern, dass weitere Fehler durch die Änderungen entstehen. Die Möglichkeit dazu muss er sich durch die geeigneten Testwerkzeuge schaffen. Nirgendwo ist die Bedeutung von Werkzeugen wichtiger als bei der Unterstützung des Regressionstests.

10.6 Capture/Replay-Technik

Ein probates Mittel zum Nachweisen der funktionalen Äqivalenz zweier Versionen desselben Systems ist die Capture/Replay-Technik. Begonnen hat diese Technik mit der Benutzungsoberfläche. Dialoge zwischen Mensch und Computer wurden mit sämtlichen Tastendrücken und Mausklicks aufgezeichnet. Auch die Ausgaben am Bildschirm, ob Signale, Anzeigen oder Meldungen, werden festgehalten. Die Kommunikation am Bildschirm wird in Form von Testskripten protokolliert, die sich später noch editieren lassen. Bei der nächsten Version werden die Testskripte ausgeführt, um die gleiche Dialogfolge wieder zurückzuspielen. Insofern als nur kleine Änderungen vorgenommen wurden, bekommt man ein Protokoll der Differenzen zwischen dem alten und dem neuen Dialog. Dies hilft, unbeabsichtigte Abweichungen zu erkennen. Sind die Änderungen zu groß, weichen die beiden Dialogfolgen so weit voneinander ab, dass eine Fehlererkennung nicht mehr möglich ist. [FeGr99]

Inzwischen ist diese Technik auch auf den Abgleich interner Schnittstellen übertragen wurden. In verteilten Systemen ist es möglich, Nachrichten zwischen verteilten

Programme abzufangen und abzuspeichern, z.B. die Aufträge vom Clientprogramm oder Applet an das Serverprogramm oder Servlet. Diese Nachrichten in Form von Datenströmen oder HTML- bzw. XML-Formaten werden in eine Logdatei geschrieben. Später werden die Eingangsnachrichten gelesen und dem Serverprogramm zugeführt, unabhängig vom Client. Andererseits werden die Ausgangsnachrichten, bzw. die Antworten des Servers gelesen und dem Clientprogramm zugeführt, unabhängig vom Server. Insofern ist es möglich, geänderte Netzknoten unabhängig von einander einzeln zu testen (Abbildung 10.12). Zu diesem Zweck hat sich die Capture/Replay-Technik vor allem bei CORBA-Anwendungen bereits bewährt.

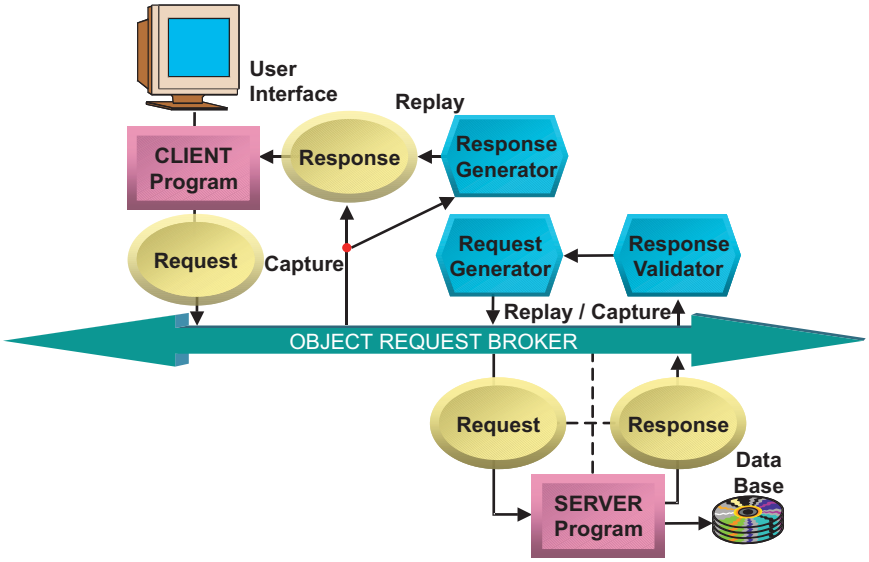


Abbildung 10.12 Capture/Replay in einer verteilten Umgebung

10.7 Regressionstest des verteilten Kalenders

Eine Fehlerkorrektur zum System Wochenkalender wird durch die Anzeige einer verkehrten Fehlermeldung, wenn der Mitarbeiter mehr als 12 Aktivitäten pro Tag eingibt, verursacht.

Adaptive Wartung bzw. eine Änderung zum Wochenkalender wäre die Ausklammerung vom Sonntag als potenziellem Arbeitstag. In Zukunft darf der Mitarbeiter keine Stunden gegen einen Sonntag buchen. Wenn er Sonntag als Wochentag eingibt, soll eine Fehlermeldung wie „Arbeiten am Sonntag ist verboten“ erscheinen.

Enhansive Wartung bedeutet, dass der Funktionsumfang des Kalenders sich erweitert. Eine Erweiterung zum Wochenkalender wäre eine Klassifizierung der Mitar-

beiter in Analytiker, Entwickler, Tester und Manager. Neben der Textbox für den Namen soll eine Menübox mit den Tätigkeitsprofilen erscheinen. Der Mitarbeiter muss das entsprechende Tätigkeitsprofil bzw. seine Rolle im Projekt anklicken.

Jeder Eingriff in den bestehenden Code hat eine Auswirkung auf den Test. Im Falle der Fehlerkorrektur erscheint eine andere Fehlermeldung am Bildschirm. Um diese Korrektur zu bestätigen, braucht man nur einen Testfall zu wiederholen, nämlich den Fall mit 13 Aktivitäten pro Tag. Die neue Meldung wird durch das Capture/Replay-Werkzeug mit der Soll-Meldung verglichen. Wenn sie übereinstimmt, ist der Fall erledigt. Bei solch trivialen Eingriffen wird es kaum nötig sein, den Klassentest und Integrationstest zu wiederholen. Der Systemtest reicht aus, um die Korrektur zu bestätigen.

Die adaptive Änderung hat eine größere Auswirkung auf die Logik des Programms. Hier wird die Menge der gültigen Wochentage in der Klasse `Woche` reduziert. Es darf kein Tages-Objekt mit der Eigenschaft `Sonntag` erzeugt werden. Zu verhindern wäre dies über eine Änderung der ENUM-Deklaration, gekoppelt mit einer zusätzlichen Operation, die den Sonntag prüft und abweist. Hier wäre es angebracht, den Klassentest für die Klasse `Woche` zu wiederholen und gezielt einen Sonntag als Parameter für die Konstruktor-Operation einzugeben. Anschließend wird es erforderlich sein, die Klasse `Woche` zu integrieren und den Integrationstest für die Kalenderkomponente zu wiederholen, wobei hier der Sonntag am Bildschirm auszuprobieren wäre. Der Systemtest wäre in diesem Falle überflüssig.

Die Erweiterung eines Tätigkeitsprofils führt zu einer Änderung der Benutzungsoberfläche. Jetzt erscheint eine weitere Menübox, ein zusätzliches Attribut in der Klasse `Kalender` und ein weiteres Merkmal in der Klasse `Projekt`, nämlich `aktivitätenart`. Die neue Aktivitätenart umfasst die möglichen Werte

- `Managen`,
- `Analysieren`,
- `Entwickeln` und
- `Testen`

entsprechend dem Tätigkeitsprofil des Mitarbeiters. Darüber hinaus muss auch die Schnittstelle zwischen der Komponente `Kalender` und der Komponente `Projekt` um einen Parameter vom Typ `Mitarbeiter` ergänzt werden. Insofern betrifft die Erweiterung sowohl die Client- als auch die Serverkomponente.

Daraus ergeben sich folgende Konsequenzen für den Regressionstest:

- Erstens muss der Klassentest für die Klasse `Kalender` wiederholt werden, und zwar mit dem neuen Konstruktorparameter `Mitarbeiterprofil`.
- Zweitens, da die Klasse `Aufgabe` das Attribut `mitarbeiterprofil` vom Kalender erbt, muss sie auch noch getestet werden, insbesondere die Operation, die die Stunden an den Projektserver sendet.

- Drittens ist die Klasse `Projekt` nochmals zu testen, weil sie einen neuen Parameter erhält und damit ein neues Attribut – `aktivitätenart` – schafft.
- Viertens ist die Interaktion zwischen den Kalender- und Projektkomponenten nochmals durch den Integrationstest zu validieren, weil die IDL-Schnittstelle sich geändert hat. Sowohl das Senden der neuen ergänzten Nachricht durch Kalender als auch das Empfangen derselben durch Projekte muss erprobt werden.
- Schließlich ist der Systemtest zu wiederholen, und zwar mit wenigstens fünf neuen Testfällen:

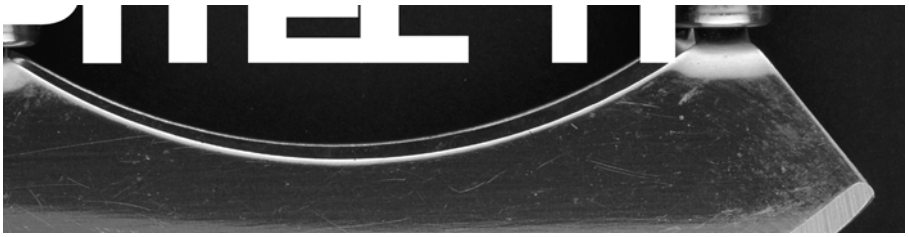
  einem Mitarbeiter vom Typ Manager,
  einem Mitarbeiter vom Typ Analytiker,
  einem Mitarbeiter vom Typ Entwickler,
  einem Mitarbeiter vom Typ Tester und
  einem Mitarbeiter vom Typ unbekannt.

Diese fünf neuen Testfälle werden zusammen mit allen alten Systemtestfällen durchgeführt, um zu sichern, dass die erweiternde Änderung keine unerwünschten Folgen hat. Es werden sowohl der Oberflächentest als auch der Funktionstest wiederholt. Auf den Belastungstest kann verzichtet werden.

Diese Wartungsbeispiele zeigen, wie aufwändig ein Regressionstest werden kann, insbesondere dann, wenn der Test nicht automatisiert ist. Mit einem automatisierten Regressionstest muss man nur die Pre- und Postbedingungen ändern und das Testskript anpassen. Ansonsten müsste man jeden Testfall manuell nochmals aufbauen, anstoßen und visuell kontrollieren. Hier lohnt sich die einmalige Investition in eine werkzeuggestützte Testumgebung, und zwar für alle Teststufen: Klassentest, Integrationstest und Systemtest.

11

Testwerkzeuge



Vorgehensweise
Testorganisation und Testdatenhaltung
Werkzeugkategorien
Empfehlungen zum Werkzeugkauf

Inhaltsübersicht Kapitel 11

11 Testwerkzeuge

Wir haben gezeigt, das die effektive Qualitätssicherung gerade für objektorientierte Software aufgrund des iterativen, inkrementellen Vorgehens sehr komplex werden kann. Infolgedessen benötigt man einerseits einen definierten Testprozess, in dem die erforderlichen Maßnahmen zur Qualitätssicherung entwicklungsbegleitend koordiniert werden. Andererseits kommen – technisch gesehen – nicht zuletzt aufgrund der Wiederverwendung von Spezifikationen und Code durch die Vererbung und den Polymorphismus dem Integrations- und Regressionstest erhöhte Bedeutung zu. Dies erfordert, dass Tests

- wiederholt, z.T. mehrmals täglich ausgeführt,
- oft editiert bzw. variiert und
- über die gesamte Lebensdauer der Software benötigt werden.

Sind Testwerkzeuge also schon für die Prüfung herkömmlicher Software von großer Bedeutung, so wird ihr Einsatz aufgrund der oben genannten Punkte beim Test objektorientierter Software generell noch wichtiger.

Ein Testwerkzeug unterstützt bzw. automatisiert eine oder mehrere der in den vorigen Kapiteln vorgestellten Testaktivitäten und -techniken. Hierbei ist zu beachten, dass der Einsatz eines Werkzeugs im Sinne des Wortes „unterstützt“ gewinnbringend sein muss – Werkzeuge dürfen nicht zum Selbstzweck werden. Der Gewinn kann jedoch durchaus unterschiedliche Fassetten haben: von einer Zeitersparnis über größere Testabdeckung, größere Flexibilität und Änderbarkeit der Tests bis hin zu verbesserter Transparenz und Steuerbarkeit des Testprozesses.

In diesem Kapitel stellen wir zunächst die unterschiedlichen Funktionalitäten von Testwerkzeugen vor und geben eine „generische“ Vorgehensweise beim werkzeuggestützten Test an. Danach beleuchten wir, inwieweit die einzelnen Testphasen von Testwerkzeugen unterstützt werden, wobei natürlich die objektorientierten Merkmale besondere Beachtung finden.

11.1 Funktionalität und Vorgehensweise

Im Prinzip lassen sich Testwerkzeuge anhand der zehn in Abbildung 11.1 gezeigten möglichen Funktionalitäten unterscheiden.

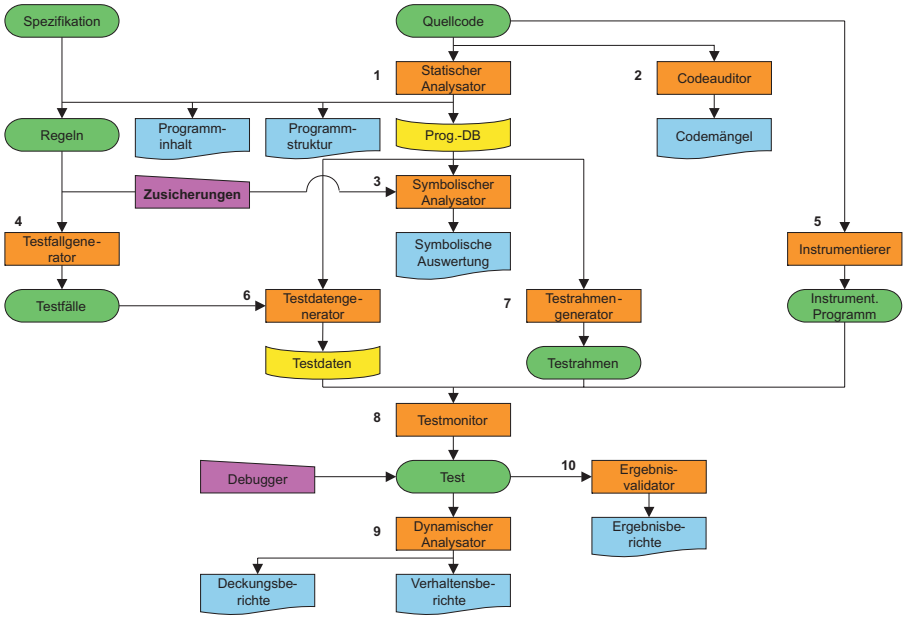


Abbildung 11.1 Werkzeug-Funktionalitäten

Man erkennt folgende Werkzeugtypen:

- Statischer Analysator (1): Analysiert den Quellcode und zerlegt ihn in Strukturen, Inhalte und Zusammenhänge.
- Code-Auditor (2): Prüft den Quellcode gegen die Programmierrichtlinie und meldet Mängel.
- Symbolischer Analysator (3): Führt den Quellcode symbolisch aus und bestätigt die Programmierlogik.
- Testfallgenerator (4): Generiert Testfälle entweder aus der Programmspezifikation oder aus dem Quellcode.
- Instrumentierer (5): Instrumentiert den Testling zwecks Ablaufanalyse. Die Instrumentierung kann dabei im Quellcode, im Objekt- bzw. Bytecode oder im ausgeführten Programm erfolgen.
- Testdatengenerator (6): Generiert Eingangsparameter, Dateien und Datenbanken, die als Programmeingabe dienen.

- Testrahmengenerator (7): Generiert Testtreiber und Teststubs für jede Komponente.
- Testmonitor (8): Stößt die Testprozeduren an und steuert den Testablauf.
- Dynamischer Analysator (9): Protokolliert die Ablaufpfade und die Ablaufüberdeckung.
- Ergebnisvalidator (10): Vergleicht die Ist-Ergebnisse mit den Soll-Ergebnissen und weist die Abweichungen aus.

Neben den hier anhand ihrer Funktionalität dargestellten Werkzeugtypen existieren noch weitere Spezial- und Mischformen, die zum Teil bereits in den vorherigen Kapiteln angerissen wurden, wie z.B. die in Kapitel 10 bereits skizzierten Werkzeuge zum Regressionstest. Andere bieten zwar interessante Details, sind aber bzgl. ihrer Funktionalität jeweils leicht einem der hier beschriebenen Werkzeugtypen zuzuordnen. Betrachtet man die zahlreichen Produkte kommerzieller Anbieter, die allerdings zur Zeit hauptsächlich herkömmliche Testverfahren unterstützen, so steht man vor einer Vielzahl von Werkzeugen, die leider viele Überschneidungen in ihrer Funktionalität aufweisen. Hier bieten unabhängige kommerzielle Werkzeugevaluationen sowie verschiedene, z.T. frei im Internet erhältliche Werkzeuglisten eine Hilfe.

Die angekündigte „generische" Vorgehensweise für den werkzeuggestützten (dynamischen) Test objektorientierter Programme ist im unten stehenden Algorithmus `Test` skizziert. Hierbei ist zu beachten, dass sich die in den einzelnen Schritten zu verwendenden Techniken natürlich hinsichtlich der Teststufe und der jeweils dem Testobjekt zugrunde liegenden Spezifikation *S*, gegen die getestet wird, unterscheiden. So wird z.B. in der Teststufe *Systemtest* das Anwendungssystem gegen die Anwendungsfälle getestet. Die in Schritt 1 zu ermittelnde Testreihenfolge *S.tre* ergibt sich in diesem Fall z.B. aus der topologischen Sortierung der als azyklischen Graph interpretierten (invertierten) «include» und «extend»-Assoziationen zwischen den Anwendungsfällen.

```
Algorithmus Test
Eingabe Spezifikation S und Klassen K;
Ausgabe Überdeckungs- und Ergebnisprotokolle;
BEGIN
1.    Test-Reihenfolge S.tre und Test-Ressourcen S.trs planen;
2.    Funktionale Testfälle BB_TF aus der Spezifikation ermitteln;
3.    S.tre und S.trs auf Testfälle übertragen;
4.    Für jeden Testfall tf Testdaten tf.BB_TD aus S ermitteln;
5.    K instrumentieren;
6.    FORALL tf ∈ BB_TF ORDERED BY S.tre DO
7.    BEGIN
8.       FORALL td ∈ tf.BB_TD DO
9.       REPEAT
10.         Testumgebung aufbauen;
11.         Testprozedur tf mit Testdaten td und Klassen K ausführen;
12.         Überdeckungs- und Ergebnisprotokoll aktualisieren;
13.      UNTIL Test-Endekriterium erfüllt OR tf.trs ausgeschöpft;
```

```
14.      IF NOT tf.trs ausgeschöpft THEN
15.         Strukturelle Testfälle generieren und ausführen;
16.         Überdeckungs- und Ergebnisprotokoll ergänzen;
17.      END
18.   END
END Test.
```

Wesentlich ist, dass die im Folgenden benannten Schritte dieses Vorgehens für jede Iteration, d.h. jeden Build, zu wiederholen sind:

- Falls zwischen zwei Builds nur korrektive Maßnahmen erfolgt sind, müssen mindestens die Schritte 5 bis 18 erneut ausgeführt werden,
- sonst zusätzlich noch die Schritte 1 bis 4 für die additive Funktionalität des zu testenden Builds.

In diesen Bereichen liegt somit der hauptsächliche Rationalisierungseffekt bei der Werkzeugunterstützung.

11.2 Testorganisation und Testdatenhaltung

In der Einleitung zu diesem Kapitel wurde dargelegt, dass Regressionstests beim Test objektorientierter Software unabdingbar sind. Dazu ist es erforderlich, die einzelnen Testprozeduren und Testdaten persistent zu halten. Man kann hierfür im Prinzip vier mögliche Architekturen unterscheiden, die sich durch die möglichen Kombinationen der Speicherung von Testprozeduren und den eigentlichen Testdaten ergeben [Sie96]:

- Die *Testprozeduren*, also die Logik der Testfälle wie z.B. Testoperationen und Testbotschaften (Skripts, ...), können entweder in eigenen Test- (Unter-)Klassen (parallel) oder aber in den zu testenden Klassen selber (eingebettet) verwaltet werden.
- Die *Testdaten*, also z.B. Werte für Botschaftsparameter und Instanzvariablen, können entweder in den Testprozeduren direkt „fest verdrahtet“ oder aber jeweils zur Testausführung aus der Datenbank geladen werden.

Darüber hinaus kann die *Ablaufsteuerung* der Testfälle entweder in den Testprozeduren oder in den Testdaten realisiert werden. Im ersten Fall enthalten die Testprozeduren „Kontrollflussanweisungen“ wie z.B. Verzweigungen oder Schleifen. Im zweiten Fall werden die möglichen Abläufe z.B. durch Schlüsselworte in den Testdaten kodiert, die dann während der Testausführung von einer Treiberprozedur gelesen und interpretiert werden . Man erreicht so eine größere Unabhängigkeit vom jeweils verwendeten Testwerkzeug auf Kosten des Interpretationsaufwands zur Laufzeit.

Es ergeben sich die vier in Abbildung 11.2 skizzierten Architekturen, die weiter unten im Einzelnen vorgestellt und bewertet werden.

In der Realität trifft man diese Architekturen nicht in Reinform an, vielmehr stellen (kommerzielle) Werkzeuge immer Abwandlungen dar, indem sie z.B. proprietäre Skriptsprachen zur Formulierung und Speicherung der Testfälle und Testdaten verwenden, was als Abwandlung der parallel-codebasierten Architektur eingeordnet werden kann. Im Einzelfall sind ein intensives Studium der Handbücher und gezielte Nachfragen beim Werkzeughersteller unumgänglich, wenn man die Architektur des Werkzeugs und damit seine Skalierbarkeit und den Aufwand zur Erstellung, Pflege und Dokumentation der Tests abschätzen möchte.

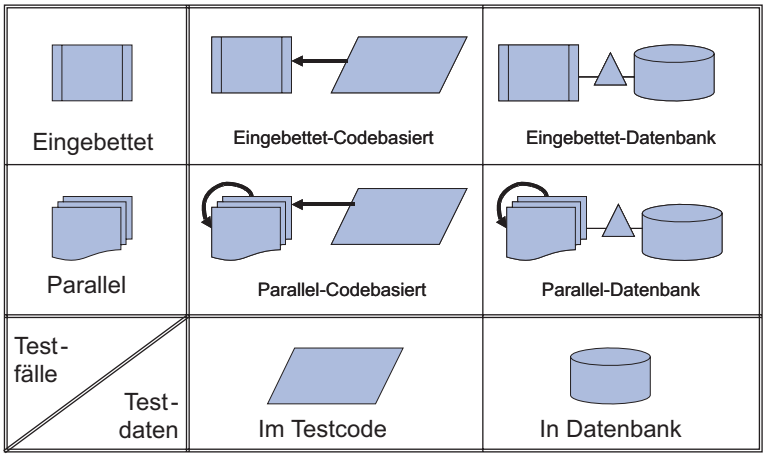


Abbildung 11.2 Architekturen für Testprozeduren und Testdaten

Natürlich darf in diesem Zusammenhang die Konfigurations- und Versionsverwaltung (z.B. mit RCS, CVS) nicht unerwähnt bleiben – *testware is software* [Pos96a]. Bei parallelen oder datenbankgestützten Architekturen werden auch Testfälle und –daten in das Konfigurationsmanagement übernommen. Nur so ist für alle Entwickler bzw. Tester der Zugriff auf die jeweils passenden Testfälle und die entsprechenden Testdaten für eine bestimmte Version sichergestellt.

11.2.1 Parallel-Code-basierte Architektur

Die einfachste Möglichkeit, zu persistenten Testprozeduren und -daten zu kommen, ist ihre Implementierung in speziellen Methoden von zusätzlichen Testklassen. Dadurch, dass Testprozeduren und -daten in Methoden der Testklassen abgespeichert sind, kann man außerdem durch Unterklassenbildung leicht Tests der Oberklasse wiederverwenden. Testprozeduren und Testdaten sind somit in eigenen Klassen untergebracht, deren Generalisierungshierarchie quasi als „Schatten“ die Hierarchie der zu testenden Klassen widerspiegelt. Hauptvorteile dieser Architektur

sind die einfache Implementierung und die automatische Vererbung von Testprozeduren und Testdaten für den Test abgeleiteter Klassen sowie die einheitliche Implementierung und Ausführung von Tests und Prüflingen. Hier trifft die oben schon zitierte Aussage "*testware is software*" zu, da die Testprozeduren eigenständige Programme darstellen, welche als Dienstnutzer der zu testenden (Produktions-) Klassen fungieren.
Nachteile sind einerseits die Einschränkung auf funktionale Tests, da man sich auf die Schnittstellen der zu testenden Klassen beschränken muss, sowie der hohe Wartungsaufwand für die Testfälle, da jede Änderung der Vererbungshierarchie der zu testenden Klassen immer Änderungen der „Testware" bedingt. Daneben muss immer die ganze Testhierarchie wiederverwendet werden, wobei ganze Testfälle (in Form von Testoperationen) zu ersetzen sind, da die Testdaten einer Testklasse von außen nicht zugreifbar sind. Ein weiteres Problem bei diesem Vorgehen liegt in der mangelnden Flexibilität bzgl. Parameterobjekten bei der parallelen Architektur: Parameterobjekte sind oft komplex und dementsprechend kompliziert zu erstellen. So bietet dieser Ansatz wenig Spielraum, um Testfälle z.B. mit unterschiedlichen Parameterobjekten zu variieren. Und schließlich ist der Ansatz auch vom Aspekt der Modellierung fragwürdig, da hier ein „Ding" (ein Testfall) wie eine Operation behandelt wird.

11.2.2 Eingebettet-Code-basierte Architektur

In dieser Architektur sind Testprozeduren und Testdaten direkt in den zu testenden Klassen untergebracht. Hauptvorteile sind dieselben der parallelen, codebasierten Architektur, wobei zusätzlich der Wartungsaufwand geringer ist. Nachteile sind auch hier, dass die Testdaten außerhalb der Klasse nicht zugreifbar bzw. wiederverwendbar sind sowie eine eventuell geringere Performanz und erhöhter Speicherplatzbedarf. Zusätzlich sind Überlegungen erforderlich, ob und wie die Testprozeduren und Testdaten bei der Auslieferung aus den Produktionsklassen entfernt werden sollen (*stripping*).

11.2.3 Parallel-Datenbank-basierte Architektur

Die Testprozeduren sind wie bei der parallel-codebasierten Architektur in Testklassen untergebracht, deren Generalisierungshierarchie eine „Schattenhierarchie" zu jener der zu testenden Klassen darstellt. Die Testdaten befinden sich in dieser Variante nicht mehr direkt in den Testprozeduren, sondern in einer Datenbank. Diese Architektur vereinbart die Vorteile der parallelen, code-basierten Architektur mit der Wiederverwendbarkeit von Testdaten in anderen Testprozeduren. Nachteil ist der erhöhte Aufwand zur Pflege und Dokumentation der Testdaten, da diese ja nun

vollständig getrennt von den Testprozeduren gehalten werden. Auch der Wartungsaufwand für die Testprozeduren bzw. Testklassen bei Änderungen an der Vererbungshierarchie der zu testenden Klassen bleibt. Trotzdem empfiehlt sich diese Architekturvariante für große (Test-)Projekte.

11.2.4 Eingebettet-Datenbank-basierte Architektur

In dieser Variante hält man die Testprozeduren wieder in den zu testenden Klassen selber und lädt die Testdaten aus einer Datenbank. Vorausgesetzt, die „Testschnittstelle“ ist einheitlich gestaltet, kann hier der größte Automatisierungseffekt erzielt werden, da ein geringer Wartungsaufwand für den Testcode mit der erhöhten Wiederverwendbarkeit der Testdaten verbunden ist. Dies gilt allerdings nur, so lange Produktionscode und Testcode von einer Person gewartet werden. In allen anderen Fällen ist die parallele datenbank-basierte Architektur aufgrund der Trennung von Produktions- und Testcode und ihrer besseren Skalierbarkeit vorzuziehen.

11.2.4.1 Fallbeispiel: Persistenz als Grundlage für den Regressionstest

Sollen Testdaten echte persistente Objekte sein, dann müssen als wichtigste Voraussetzung auch alle Parameterobjekte persistent sein. Insbesondere bei objektorientierten Programmiersprachen ohne inhärente Persistenzeigenschaften müssen deshalb Mechanismen zur Unterstützung persistenter Objekte implementiert werden. Hunt zeigt, wie Hilfsmittel zum Regressionstest von C++-Klassen implementiert werden können [Hun95a]. Zunächst werden zwei Klassen implementiert, die Basisfunktionalitäten bereitstellen, um Instanzvariablen in einem ASCII-Format abzuspeichern und einzulesen (`Load`, `Save`). Mit Hilfe der Klasse `Save` wird dann in jeder Anwendungsklasse eine Methode zum Abspeichern der eigenen Repräsentation implementiert (`saveMe(Save *saveObject)`).

Um eine so gespeicherte Repräsentation wieder einlesen zu können, wird innerhalb der Klasse `Load` zunächst ein Objekt der Klasse der ASCII-Repräsentation erzeugt, welches danach selbst für die Wiederherstellung seiner Instanzvariablen zuständig ist (`loadMe(Load *loadObj)`). Parameterobjekte und Testobjekte können so gespeichert und geladen werden. Die Namen der zu überprüfenden Methoden werden dagegen in einer Treiber-Operation festgehalten.

Die Testauswertung geschieht zunächst manuell, indem die ASCII-Repräsentation des Testobjekts nach der Testdurchührung mit einem üblichen Werkzeug (z.B. in einem Texteditor oder einem Dateikomparator) vom Tester überprüft wird. Für Regressionstests kann die geprüfte ASCII-Repräsentation dann als Testreferenz dienen. Diese Technik einer bedingten Automatisierung der Testauswertung – oft

als Referenztest (reference testing) bezeichnet – ist sehr einfach umsetzbar, allerdings müssen bei Änderungen der Repräsentation der zu testenden Klasse in der Regel alle Testreferenzen erneut manuell überprüft werden. Bei Änderungen der Repräsentation von Parameterobjekten und Testobjekten können die bereits gespeicherten Objekte dieser Klasse weiterhin für den Regressionstest benutzt werden, indem passende Default-Werte für veränderte Instanzvariable zugewiesen werden.
Trotz der Einfachheit dieses Ansatzes darf der Arbeitsaufwand für die Erzeugung komplexer Objekte und für deren Evaluierung nicht unterschätzt werden. Testfälle werden bei Hunts Technik nicht durchgängig als Objekte repräsentiert, denn die Namen der Nachrichten sind fest in einer Treiberprozedur verankert (innerhalb der `main()`-Hauptfunktion von C- und C++-Programmen). Dadurch ergeben sich Probleme beim Test von Unterklassen, weil Vererbungsmechanismen nicht auf die Konstruktion einer parallelen Testtreiber-Hierarchie anwendbar sind.

11.3 Werkzeugkategorien

Nach diesen einführenden Worten skizzieren wir in den nächsten Abschnitten Werkzeuge zu den Testphasen:

- Testplanung und Testmanagement,
- Testentwurf,
- Testfallspezifikation,
- Testprozedurerstellung,
- Testaufbau,
- Testdurchführung und
- Testauswertung.

An Fallbeispielen – vornehmlich bezüglich wohldokumentierter experimenteller Prototypen – werden jeweils die besonderen Aspekte der Objektorientierung bezüglich der Werkzeugunterstützung hervorgehoben.
Um die Werkzeuge vergleichend zu den in den vorigen Kapiteln behandelten objektorientierten Teststufen- und -Phasen betrachten zu können, lehnen wir uns an das in den Arbeiten von Grimm, Grochtmann und Wegener [Gri94][GrWe93] veröffentlichte Schema zur Darstellung der Testaktivitäten an. In Abbildung 11.3 sind die einzelnen Testaktivitäten und ihr Zusammenwirken dargestellt. Diese Aktivitäten werden auch beim Test objektorientierter Software ausgeführt.

11.3.1 Testplanung und Testmanagement

Im Bereich des Testmanagements findet man hauptsächlich Werkzeuge zur Testplanung und Testfortschrittskontrolle. Da beim objektorientierten Test Integrationsstrategien ebenso wichtig wie kompliziert zu erstellen sind (siehe Kapitel 7, Integrationstest), empfiehlt sich der Einsatz von Werkzeugen, welche die Erstellung von Integrationsplänen unterstützen. Hierzu können statische Analysatoren dienen, mit denen die Abhängigkeiten der Komponenten (Klassen, Pakete, ...) ermittelt und geeignet dargestellt bzw. abgespeichert werden. Oft lassen sich diese Abhängigkeiten auch mit CASE-Werkzeugen modellieren bzw. ermitteln.

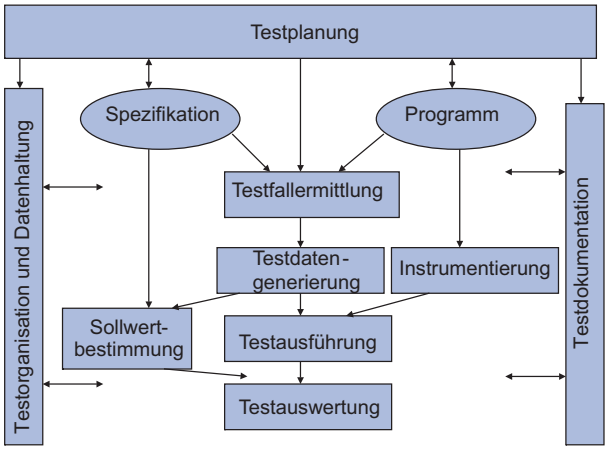


Abbildung 11.3 Werkzeugbezogene Testphasen (Angelehnt an [Gri94])

Bzgl. der Planung der Testumgebung ist darauf zu achten, dass Teststümpfe i.d.R. einen höheren Realisierungsaufwand abverlangen, da zumindest minimale, die Spezifikation erfüllende Implementierungen vorliegen müssen.

Werkzeuge zur Testfortschrittskontrolle, also z.B. Festlegung des Testkriteriums und des Testziels, erlauben es, aufbauend auf den Testberichten (Überdeckungsprotokolle und Fehlermeldungen im Zeitverlauf oder akkumuliert) Aussagen zum Stand der Testaktivitäten zu machen. Hierbei muss darauf geachtet werden, dass traditionelle Testkriterien (z.B. Zweigüberdeckung) und die damit verbundenen Ziele (z.B. Zweigüberdeckung min. 90%) angepasst werden müssen (siehe Tabelle 4-1 und Tabelle 4-2).

11.3.1.1 Zeit- und Ressourcenplanung

Zur Zeit- und Ressourcenplanung können herkömmliche Werkzeuge wie z.B. Spreadsheet- oder Netzplan-Editoren eingesetzt werden. Spezielle Projektmanagementwerkzeuge bieten unterschiedliche Sichten auf das Projekt.
Sehr hilfreich ist es, wenn Daten und Modellelemente wie z.B. Anwendungsfälle oder Klassen direkt aus dem CASE-Werkzeug in das Projektmanagement-Werkzeug übernommen werden können. Durch eine Anreicherung dieser Elemente um Planungsinformation lässt sich nämlich die Planung sehr eng an den Spezifikationen gekoppelt durchführen.
Hierzu bestimmen die Analytiker und Tester ein Profil für jeden Anwendungsfall z.B. nach folgenden Kriterien (vgl. [Win99] [MgSy00]):

- Die *Kritikalität C* repräsentiert die (schlimmst-)möglichen Auswirkungen bei Ausfall oder fehlerhafter Bearbeitung des Anwendungsfalls. Sie wird durch Folgeabschätzungen wie z.B. die Fehler- oder Klassifikationsbaumanalyse ([Gri94] [GrWe93]) in Zusammenarbeit mit dem Auftraggeber und ggf. den Benutzern festgelegt.
- Das *technische Risiko* R_T (wie kompliziert wird die Realisierung des Anwendungsfalls) wird aus der Komplexität der Beschreibung (textuelle Beschreibung, ggf. Aktivitätsdiagramm und Sequenzdiagramme) abgeleitet.
- Das *geschäftliche Risiko* R_B (inwiefern ist der Absatz/die Akzeptanz des Anwendungssystems bei Nichterfüllung der Anforderung gefährdet) wird z.B. durch Befragungen von (prospektiven) Benutzern ermittelt.
- Das *Projektrisiko* R_P (inwiefern gefährdet die Nichterfüllung des Anwendungsfalls zu einem bestimmten Zeitpunkt den weiteren Projektfortschritt) ergibt sich aus der Anzahl der über `«include»` und `«extend»`-Assoziationen vom betrachteten Anwendungsfall abhängigen Anwendungsfälle.

Die Zuteilung der Profile zu den ersten Skizzen der Anwendungsfälle erfolgt aus der Erfahrung früherer Projekte und Risikoabschätzungen des Projektmanagements. Drei Werte *low*, *medium* und *high* reichen erfahrungsgemäß zur Quantifizierung der einzelnen Faktoren des Profils aus, zudem diese auch nicht unabhängig voneinander sind (s. z.B. [Lyu96]). Indem man diese drei Werte auf die natürlichen Zahlen 1, 2 und 3 abbildet, kann z.B. mit der folgenden Formel jedem Anwendungsfall *A* ein bestimmtes Gewicht W_A zugewiesen werden:

$$W_A = \frac{R_T + R_B + R_P}{3} + F \cdot R_B + C \cdot R_T$$

Für die Planung der Entwicklung überträgt man die Gewichte der Anwendungsfälle über die in den Sequenzdiagrammen vorkommenden Klassen zunächst auf Elemente des Domänen-Klassenmodells und kann dann erste, grobe Aufwandsschätzungen vornehmen. Hierbei berechnet man das Gewicht einer Klasse z.B. als Mittelwert

oder – konservativer – als Maximum der Gewichte aller Anwendungsfälle, in deren Beschreibung die Klasse vorkommt.
Über die Verfolgungsrelation (s. unten) können die Gewichte dann später auf entsprechende Elemente des Entwurfs und der Implementation abgebildet werden. Die am Anfang der Entwicklung grob abgeschätzten Profile der Anwendungsfälle werden dann im Laufe der Entwicklung verfeinert, wobei auch die Komplexität der entsprechenden (Entwurfs- und Implementations-) Klassen mit einfließt.

11.3.1.2 Anforderungsmanagement und Verfolgbarkeit

Eines der ausschlaggebenden Kriterien für den Projekterfolg ist die Zufriedenheit der Kunden bzw. Benutzer des Anwendungssystems. Diese kann nur erreicht werden, wenn sich alle Anforderungen gemäß ihrer Gewichtung im Produkt wiederfinden. Da sich die Anforderungen erfahrungsgemäß im Laufe der Zeit ändern, müssen möglichst effizient die Teile des Anwendungssystems und der Testware identifiziert werden können, die von einer solchen Änderung betroffen sind.
Das Identifizieren von aufeinander aufbauenden bzw. (inhaltlich) zusammenhängenden Elementen wird als Verfolgbarkeit (traceability) bezeichnet (vgl. [GoFi94]). Was aber bedeutet Verfolgbarkeit genau, und wie wird sie ermöglicht? Eine Definition in Bezug auf die Anforderungsspezifikation gibt die IEEE Norm 830 (Guide to Software Requirements Specifications), in der man liest:

> *"A SRS (software requirements specification) is traceable, if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation." [IEEE830]*

Man kann bei der Verfolgbarkeit zunächst die horizontale und vertikale Verfolgbarkeit unterscheiden ([Dav90]):

- Die *horizontale Verfolgbarkeit* setzt einerseits Elemente der gleichen Entwicklungstätigkeit in einen (zeitlichen) Bezug – man spricht in diesem Zusammenhang auch von Versionen eines Artefakts. Andererseits beschreibt die horizontale Verfolgbarkeit, wie bestimmte Modellierungselemente innerhalb einer Entwicklungstätigkeit durch andere (gleichartige Elemente) verfeinert bzw. präzisiert werden.
- Die *vertikale Verfolgbarkeit* betrachtet die Relationen zwischen Elementen verschiedener Entwicklungstätigkeiten und gibt Antwort auf die Frage, wie ein bestimmtes Element (z.B. eine Klasse im Quellcode) aus einem anderen (z.B. einer Klasse im Entwurf) hervorgegangen ist.

Relativ zur Anforderungsspezifikation lassen sich die folgenden vier Richtungen der vertikalen Verfolgbarkeit ableiten (nach [Dav90]):

- Verfolgbarkeit „zurück von den Anforderungen“ impliziert das Wissen, warum ein Element der Anforderungsspezifikation existiert, also eine Referenz zur Quelle der Anforderung (Gesprächsprotokolle, Skizzen, ...).
- Verfolgbarkeit „vorwärts zu den Anforderungen“ bedeutet, dass jedes der Anforderungsspezifikation vorausgehende Dokument die aus ihm hervorgegangenen Elemente der Anforderungsspezifikation referenziert.
- Verfolgbarkeit „vorwärts von den Anforderungen“ impliziert das Wissen, welche Elemente der Anwendung eine bestimmte Anforderung befriedigen, also z.B. welche Entwurfselemente ihren Ursprung in einer bestimmten Anforderung haben.
- Verfolgbarkeit „zurück zu den Anforderungen“ ermöglicht, zu jedem Element der Anwendung festzustellen, welche Anforderung es zu erfüllen hilft.

Bei den beiden ersten Richtungen wird auch von Vor-Verfolgbarkeit (pre-traceability) gesprochen, bei den letzten beiden dementsprechend von Nach-Verfolgbarkeit (post-traceability) (vgl. [GoFi94]).

Im Sinne des Buches interessiert die Verfolgbarkeit der Anforderungen im Wesentlichen bezüglich des System- und Abnahmetests des objektorientierten Anwendungssystems gegen die Anforderungsspezifikation. Somit steht die Nach-Verfolgbarkeit der Anforderungsspezifikation über den Entwurf in die Implementation im Mittelpunkt. Daher betrachtet man die horizontale Verfolgbarkeit meistens nicht im Sinne des zeitlichen Bezugs, sondern sieht diese rein als Gegenstand des Konfigurations-Managements bzw. der Versionsverwaltung an. Auch die Vor-Verfolgbarkeit wird nicht in diesem Zusammenhang, sondern nur im Rahmen der Anforderungsermittlung an sich betrachtet. Im Bereich des Tests objektorientierter Software spricht man somit, wenn keine Verwechslungsgefahr besteht, anstelle von vertikaler Nach-Verfolgbarkeit nur noch von Verfolgbarkeit.

In Anlehnung an die UML verwendet man oft Abhängigkeitsbeziehungen zur Modellierung der Verfolgbarkeitsrelationen. Hierbei genügen oft die beiden Relationen *Verfolgung* (trace) und *Verfeinerung* (refinement) als Unterarten der allgemeinen Relation *Abhängigkeit* (dependency).

Die `Trace` und `Refinement`-Beziehungen als Unterarten der allgemeineren `Dependency`-Beziehung fassen jeweils eine Menge von Modellierungselementen zusammen. Zur Verfolgbarkeit dienen Instanzen der Klasse `Trace`, die Modellierungselemente aus verschiedenen Modellen verknüpfen. Kann über eine einfache Zuordnung hinaus angegeben werden, wie die Modellelemente einander entsprechen, wird diese „Verfeinerung“ genannte Abhängigkeit im Attribut `mapping` einer Instanz der Klasse `Refinement` festgehalten. Während Instanzen der Klasse `Trace` also lediglich syntaktische Abhängigkeiten angeben, erfassen Instanzen der Klasse `Refinement` zusätzlich semantische Abhängigkeiten, die z.B. zur Steuerung von Verfolgbarkeits-Anfragen etc. verwendet werden können.

11.3.1.3 Konfigurations- und Versionsmanagement

In einem Softwareprojekt entsteht eine Vielzahl von Dokumenten bzw. Entwicklungsprodukten, angefangen von der Anforderungsspezifikation über Entwurfsdokumente, Programmcode und Testfälle bis hin zu Testberichten, ausgelieferten Softwareversionen und Anwenderhandbüchern. Gerade in den inkrementellen, iterativen Entwicklungsprozessen für objektorientierte Software werden in kurzen Zeitabständen immer wieder Softwareversionen ausgeliefert, zu denen Versionen der entsprechenden Entwicklungsprodukte existieren.

Wichtig ist es hierbei, dass die Werkzeugumgebung die Aufnahme aller dieser Entwicklungsprodukte unterstützt. Insbesondere die Testprodukte sind also mit in das Konfigurations- und Versionsmanagement aufzunehmen (Abbildung 11.4). Nur so ist gewährleistet, dass bei kurzfristigen „Reparaturen“ (Patches) einer ausgelieferten Version die erforderlichen Regressionstests in der Kürze der zur Verfügung stehenden Zeit aufgefunden und durchgeführt werden können.

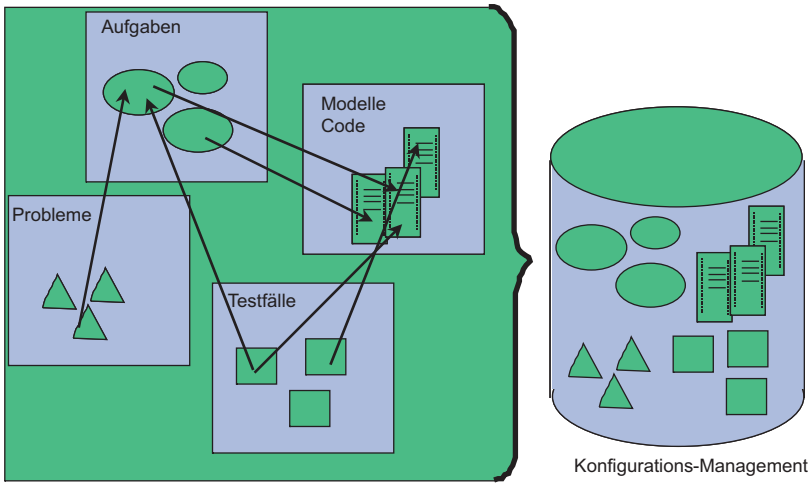


Abbildung 11.4 Produkte im Versions- und Konfigurationsmanagement

11.3.1.4 Fehlerdatenbanken

Wesentlich für den Erfolg qualitätssichernder Maßnahmen ist die Sicherstellung der Berichtigung gefundener Fehler (Debugging und Correction). Insbesondere in der Produktion aufgetretenes Fehlverhalten muss geeignet protokolliert und in fehlerlokalisierenden und korrigierenden Tätigkeiten behandelt werden. Da die Auslieferung von Patches und neuen Versionen mit hohen Kosten verbunden ist, sind diese Maßnahmen geeignet zu koordinieren, was durch Fehlerdatenbanken in Verbindung mit dem Konfigurationsmanagement erfolgt. In einer Fehlerdatenbank werden –

ausgehend vom Fehlerbericht bis hin zum Release eines Patches oder einer neuen Version – der gesamte Lebenszyklus eines Fehlers nachgehalten und ggf. die erforderlichen Maßnahmen geplant und angestoßen (Abbildung 11.5).

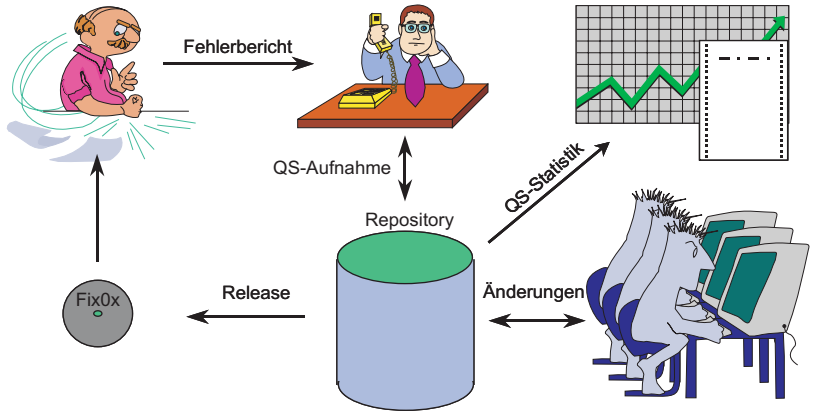


Abbildung 11.5 Fehlerverfolgung mit Fehler-Repository

Nicht zu vergessen ist die statistische Aufbereitung der anfallenden Daten, mit der Aussagen zur Zuverlässigkeit, MTTF, MTTR etc. möglich werden.
Für jeden Fehler sind zumindest die in Abbildung 11.6 aufgelisteten Informationen zu speichern.

- Fehlernummer
  - ProduktID
  - Betroffene Releases
  - GemeldetAm
  - GemeldetVon
  - AufgenommenVon
  - Status (gemeldet, offen, zugewiesen, inBearbeitung, ...)
  - Priorität (hoch, mittel, niedrig)
  - FixRelease
  - BearbeitetVon

Abbildung 11.6 Einträge im Fehler-Repository

Jeder gemeldete Fehler durchläuft im Rahmen der oben in Abbildung 11.5 dargestellten Fehlerverfolgung einen Lebenszyklus gemäß Abbildung 11.7. Jeder gemeldete Fehler wird zunächst im Zustand `gemeldet` in das Fehler-Repository eingetragen. Zur weiteren Bearbeitung muss die Fehlermeldung in den Zustand `offen` versetzt werden. Offene Fehler werden weiter untersucht bzw. nachvollzogen und je nach Untersuchungsergebnis entweder abgewiesen oder aber einem Entwickler

bzw. einer Projektgruppe zur Behebung zugewiesen. Abgewiesene Fehler werden nach einer bestimmten Frist entweder geschlossen oder aber bei erneutem Auftreten bzw. erneuter Meldung (z.B. bei mehrfach gemeldeten Benutzerfehlern, die tatsächlich auf Benutzbarkeitsfehler hinweisen) in den Zustand „wiedereröffnet“ versetzt und weiter bearbeitet.

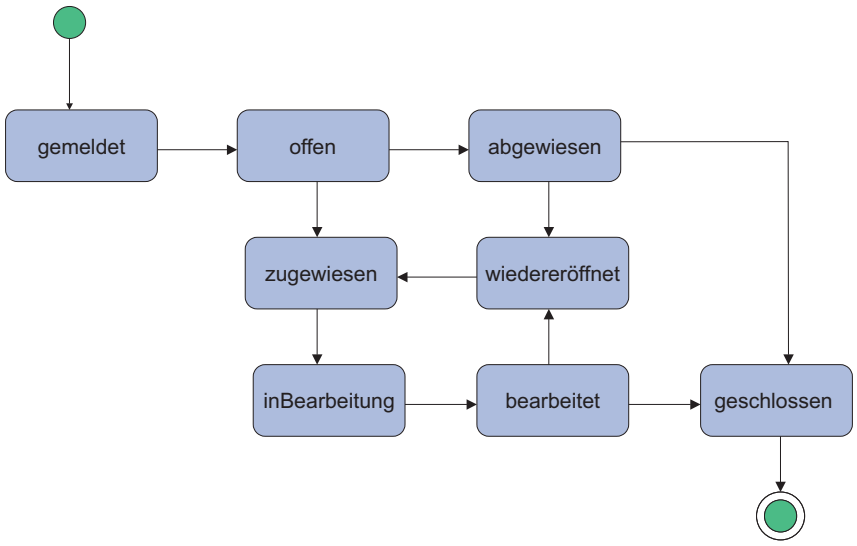


Abbildung 11.7 Lebenszyklus eines Fehlers

11.3.1.5 Dokumentenverwaltung

In der Testdokumentation müssen alle zu testenden Klassen und die jeweils benötigten Testumgebungen (Treiber und Stümpfe) festgehalten werden. Außerdem sollten für jede zu testende Klasse eine Reihe von Testfallspezikationen erstellt werden, anhand derer später konkrete Testfälle erstellt oder erzeugt werden. In Kapitel 5 haben wir eine ausführliche Beschreibung der objektorientierten Testfallspezifikation gegeben. Bzgl. des Werkzeugeinsatzes ist dieser Bereich genauso zu behandeln wie beim traditionellen Test imperativer Programme: Es müssen geeignete Werkzeuge zur Dokumentation eingesetzt werden.

11.3.1.6 Fallbeispiel: Integrated Object-Oriented Testing and Maintenance (OOTM)

Kung et Al. haben eine integrierte Test- und Wartungsumgebung für objektorientierte Programme entwickelt, mit welcher die folgenden Bereiche abgedeckt werden (vgl. [KGH+94]):

1. Das Reverse-Engineering, mit dem Entwurfsinformationen aus C++ Programmen zurückgewonnen werden.
2. Ein datenflussorientiertes Testverfahren für objektorientierte Programme.
3. Ein Algorithmus zur Berechnung von Testplänen bei Änderung einer Klasse.
4. Eine strukturelle, zustandsbasierte Testtechnik.
5. Ein szenario-basiertes Verfahren zum Akzeptanztest, das auf denselben Konzepten wie die zustandsbasierte Testtechnik aufsetzt.

Für die Berechnung von Testplänen bei Klassenmodifikationen und die Testfallgenerierung für die zustandsbasierte Testtechnik werden durch das implementierte Verfahren zum Reverse-Engineering folgende Diagramme aus dem Quellcode erzeugt:

- Das *Object Relation Diagram* (ORD) zeigt die Vererbungs- und Benutzungsrelationen zwischen den Anwendungsklassen an und wird bei der Testplanberechnung verwendet.
- Das *Block Branch Diagram* (BBD) repräsentiert den Kontroll- und Datenfluss einer Methode inklusive ihrer Schnittstellen zu anderen Methoden und wird zur Anwendung struktureller Testkriterien benötigt.
- Das *Object State Diagram* (OSD) modelliert das zustandsabhängige Verhalten von Instanzen einer Klasse und wird zur Testfallgenerierung eingesetzt.

Wir haben bereits gezeigt, dass Änderungen an Klassen einer objektorientierten Anwendung die Durchführung von Regressionstests erfordern. Diese Regressionstests umfassen nicht nur die Überprüfung der geänderten Klassen, sondern vielmehr auch den Test aller von den geänderten Klassen abhängiger Klassen. Die Menge der von einer Klassenmodifikation betroffenen Klassen nennen Kung et Al. *Class Firewall*. Die in einer Class Firewall enthaltenen Klassen werden mit Hilfe des ORD berechnet. Vereinfacht ausgedrückt, werden alle Klassen, die direkt oder indirekt eine modifizierte Klasse benutzen, sowie alle mit der modifizierten Klasse in einer Vererbungsbeziehung stehenden Klassen der Class-Firewall hinzugefügt.
Anhand des ORD wird ein Testplan berechnet, der die Verwendung von Teststümpfen minimiert. Der Algorithmus zur Testplanberechnung entspricht dabei weitgehend dem von Overbeck (vgl. [Ove94]), ohne jedoch dynamisches Binden und Polymorphismus zu berücksichtigen.
Auch die Testfall-Generierung beginnt mit einem Reverse-Engineering-Schritt, bei dem durch symbolische Programmausführung das Zustandsverhalten – bzw. das

OSD – einer zu testenden Klasse zurückgewonnen wird. Anhand des OSD werden anschließend mit bekannten Verfahren zum Zustandstest Testfälle generiert, die alle Zustände und alle Zustandsübergänge des OSD abdecken.

```
Algorithmus Testreihenfolge
Eingabe Object Relation Diagramm;
Ausgabe Testreihenfolge;
BEGIN
   Transformiere das ORD in einen azyklischen gerichteten
   Graphen ORG'
                                  // Hauptreihenfolge
   Sortiere die Knoten des ORG' topologisch
   FORALL starke Zusammenhangskomponenten des ORG' DO
   BEGIN
      FOREACH Zyklus im Cluster DO
      BEGIN
         Wähle und entferne eine Kante, um den Zyklus
         aufzubrechen
                                   // Nebenreihenfolge
         Erstelle eine topologische Sortierung der
         verbleibenden Knoten
      END
   END
END Testreihenfolge.
```

Das von Kung et Al. vorgeschlagene Testverfahren kann folgendermaßen charakterisiert werden:

- Durch die Testplangenerierung können einerseits die Entwickler feststellen, wie hoch der Regressionstestaufwand für eine bestimmte Änderung anzusetzen ist, und andererseits können die Tester ihn als Hilfsmittel zur Durchführung der Regressionstests verwenden.
- Alle Mechanismen zur Testplangenerierung und Testfallgenerierung benötigen lediglich den Programmcode. Die einzige Prämisse zur Einsetzbarkeit des Werkzeugs ist demnach die Verwendung von C++ als Implementierungssprache. Dadurch ist es in weit mehr Entwicklungsprojekten einsetzbar als Werkzeuge, die bestimmte Spezifikationssprachen voraussetzen.
- Damit verbunden ist aber gleichzeitig der Nachteil, dass eine automatische Bestimmung der erwarteten Ergebnisse nicht möglich ist. Hier wäre eine Integrationsmöglichkeit von formalen Testverfahren und Werkzeugen wünschenswert, wie sie z.B. in ASTOOT verwendet wird (siehe unten).

Die verwendeten Diagrammarten (ORD,OSD,BBD) spiegeln prinzipiell dieselben Informationen wider, wie sie z.B. in den UML-Modellen enthalten sind (Klassendiagramm, Interaktionsdiagramme, Zustandsdiagramm). Dadurch ergeben sich Möglichkeiten der statischen Analyse zur Überprüfung der Konsistenz zwischen Entwurf und Implementierung. Zusätzlich könnten Verfahren zur Testfallgenerierung und Testdatengenerierung anhand von Entwurfsdokumenten integriert werden,

weil die Testfallgenerierung allein aufgrund des Quellcodes, wegen Problemen mit der symbolischen Programmausführung und wegen der Komplexität der extrahierten Automaten zu einer sehr großen Menge von Testfällen führt.

11.3.2 Testentwurf

Im Testentwurf wird im Wesentlichen das Testkonzept erarbeitet, in dem die Testanforderungen festgehalten und eine Strategie für deren Erfüllung ausgearbeitet werden. Die Erstellung des Testkonzepts ist ein kreativer Prozess, der nur eingeschränkt durch Werkzeuge unterstützt werden kann. Hauptsächlich können Werkzeuge dabei helfen, auf der Grundlage bereits vorhandener Entwicklungsprodukte Metriken zu erstellen, mit denen die Auswahl von Testtechniken und die Definition von Testendekriterien vereinfacht wird. Forschungsprototypen ermöglichen hierbei bereits eine – allerdings größtenteils produktzentrierte – heuristische Auswahl von Techniken und Methoden [Lig94]. In der Praxis werden im Rahmen der Testkonzepterstellung statische Analysatoren für die unterschiedlichen Entwicklungsprodukte verwendet, um Aussagen über die Einsetzbarkeit und zu erwartende Effektivität der unterschiedlichen Testtechniken zu ermöglichen.

11.3.2.1 Statische Analysatoren

Statische Analysatoren prüfen ein Entwicklungsprodukt (in den meisten Fällen den Programmcode) entsprechend bestimmter „defekt-trächtiger“ Eigenschaften. Einfache Analysatoren arbeiten rein syntaxbasiert auf dem Quellcode und prüfen gegen eine Reihe fest voreingestellter Merkmale, wie z.B. C-Programmierstil nach Kernighan und Ritchie o.ä..

Firmeneigene Programmierrichtlinien können in konfigurierbaren Analysatoren berücksichtigt werden, wenn sie sich einfach (syntaktisch) z.B. durch Parametrisierung vorhandener Prüfungen ausdrücken lassen. Als Beispiel sei die Anzahl der Kommentarzeilen pro Methode genannt.

Kompliziertere Prüfungen erfordern regelbasierte Eingabemöglichkeiten, mit denen mehrere Bedingungen verknüpft und im Zusammenhang ausgewertet werden. So lässt sich z.B. die Programmierstilrichtlinie „Jeder Parameter muss einzeln kommentiert werden“ nicht einfach durch eine kontextfreie Grammatik oder eine Parametrisierung formulieren, sondern erfordert eine Regel der folgenden Art:

```
FOREACH Methode
   FOREACH Parameter
      EXISTS Comment WITH
             Comment.firstWord() = Parameter.identifier
```

11.3.3 Testfallspezifikation

Bei der Testfallspezifikation werden die Testfälle für eine Klasse, ein Teilsystem oder das komplette objektorientierte Anwendungssystem spezifiziert. Bzgl. der Form dieser Testfallspezifikationen sei auf Kapitel 5 verwiesen. Zum Erstellen dieser Spezifikationen werden die in den Kapiteln 6 bis 8 angesprochenen Testtechniken verwendet. Deshalb können hier sehr unterschiedliche Werkzeuge eingesetzt werden, die je nach verwendeter Testtechnik unterschiedliche Mechanismen zur Unterstützung der Erstellung von Testfallspezifikationen anbieten: Bei Verwendung von zustandsbasierten Testverfahren z.B. können Werkzeuge Nachrichtensequenzen vorschlagen, die alle Zustände eines Testgegenstands wenigstens einmal durchlaufen. Wichtig ist hierbei zumindest eine Datenanbindung des Testwerkzeugs an das CASE-Werkzeug, damit die Spezifikationen (Diagramme, Bereichsangaben, Regeln, ...) direkt für die Testfall-generierung ausgewertet werden können.

In der Sollwertbestimmung legt man für jeden Testfall anhand der Spezifikation das erwartete Resultat, die Testreferenz fest: Je nach Testebene umfasst die Testreferenz nur den Nachzustand des Testobjekts (Klassentest) oder auch den Zustand von Parameterobjekten und Rückgabeobjekt (Integrationstest). Die Werkzeugunterstützung für diese Testaktivität wird von der verwendeten Spezifikation und der Testtechnik bestimmt. So lässt sich eine Testreferenz nur dann vollautomatisch berechnen, wenn die Spezifikation operational, also automatisch interpretierbar bzw. „ausführbar“ ist.

Bei der Zuordnung von Werkzeugen zu den Aktivitäten Testfallermittlung und Testdatengenerierung werden Werkzeuge, die Nachrichtensequenzen erzeugen, ohne die benötigten Parameterobjekte bereitzustellen, der Testfallermittlung zugeordnet, während Werkzeuge, die Parameter(-objekte) für Nachrichtensequenzen bereitstellen, der Testdatengenerierung zugeordnet werden.

11.3.3.1 Fallbeispiel: Zustandsbasierter Test

Hoffman und Strooper haben eine Technik für den zustandsbasierten Test von C++-Klassen entwickelt und ein entsprechendes Testwerkzeug implementiert [HoSt94][Hof95]. Zunächst wird ein so genannter „Testgraph“ als vereinfachter Zustandsautomat der zu testenden Klasse entworfen. Der Testgraph ist insofern vereinfacht, als er nur die beim Test zu berücksichtigenden Zustände und Zustandsübergänge beinhaltet. Anhand des Testgraphen werden dann für jede zu testende Klasse zwei Hilfsklassen implementiert:

- Eine Orakelklasse bzw. ihre (einzige) Instanz überprüft, ob der Zustand des Testgraphen mit dem Zustand einer Instanz der zu testenden Klasse übereinstimmt.

- Die Treiberklasse bzw. ihre (einzige) Instanz durchläuft alle Kanten (Zustandsübergänge) des Testgraphen und vergleicht nach jedem Übergang mit Hilfe der Instanz der Orakelklasse die Zustände des Testgraphen und des Testgegenstands. Der Testgraph wird also durch das Zusammenspiel der Instanzen beider Hilfsklassen implementiert, was zu einer automatischen Testdurchführung und Testauswertung führt.

Hoffman und Strooper berücksichtigen in ihrem Testwerkzeug den Test von Unterklassen, durch eine parallele Generalisierungshierarchie von Treiber- und Orakelklassen entsprechend der Hierarchie der Anwendungsklassen. Wie schon im Abschnitt 11.2 erläutert, ist der Aufwand zur Implementierung der Hilfsklassen hierbei nicht zu vernachlässigen, da die Testumgebung teilweise einen doppelt so großen Umfang wie die zu testende Klasse hat. Da dieser hohe Aufwand aber vor allem durch die (manuelle) Implementierung der Treiberklasse verursacht wird, empfehlen Hoffman und Strooper zur Erstellung der Testgraphen die Implementierung einer entsprechenden unterstützenden grafisch-interaktiven Komponente.

11.3.3.2 Fallbeispiel: Test abstrakter Datentypen

Parrish, Cordes und Borie stellen ein Testwerkzeug zum Test abstrakter Datentypen (ADT) vor, das vollautomatisch Testfälle anhand von Ada-Quellcode erzeugt [PBC93]. Als einschränkende Grundvoraussetzung zum Einsatz der Methode bzw. des Testwerkzeugs müssen die zu testenden Ada-Module allerdings so programmiert sein, dass es keine illegalen Folgen von Operationsaufrufen gibt bzw. alle Reihenfolgen von Operationen des zu testenden ADT legale Testfälle darstellen. Um diese Testfälle – also alle Reihenfolgen von Operationen des zu testenden ADT – mit Parametern zu versorgen, wird je nach Parametertyp folgendes Verfahren angewandt:

- Handelt es sich um einen Parameter vom Typ des zu testenden ADT, wird er vom vorhergehenden Aufruf – also der vorherigen Nachricht – übernommen, oder es wird ein neues ADT-Exemplar erzeugt.
- Bei Parameter anderer Typen werden wiederum zwei Fälle unterschieden:
  - Im Falle von Basistypen wird der Benutzer zu dessen interaktiver Eingabe aufgefordert.
  - Bei benutzerdefinierten Typen wird ein neues ADT-Exemplar mit der Default-Initialisierung erzeugt.

Die Testdatengenerierung wird also zumindest ansatzweise durch das Werkzeug unterstützt.

Die Testauswertung wird im Testwerkzeug von Parrish, Cordes und Borie unterstützt, indem für jeden ADT die Implementierung einer `DISPLAY`-Operation gefordert wird. Diese `DISPLAY`-Operation wird jeweils nach Abarbeitung einer Operation

der Testsequenz aufgerufen, wonach die tatsächliche Auswertung dann manuell durch den Benutzer anhand des ausgegebenen Textes erfolgen muss.
Rüppel weist darauf hin, dass bei diesem Testwerkzeug gegen ein wichtiges Grundprinzip der Testautomatisierung verstoßen wird [Rüp96]: Nachgelagerte Testaktivitäten sollten immer vor vorgelagerten Testaktivitäten automatisiert werden. Da hier jedoch eine automatische Testfallgenerierung betrieben wird, ohne die Testauswertung ausreichend zu unterstützen, müssen evtl. große Mengen von generierten Testfällen mühsam manuell ausgewertet werden.
Weiterhin ist das starre Verfahren zur Unterstützung der Testdatengenerierung nicht geeignet, um z.B. unterschiedliche Parameterobjekte – z.B. bei fremden ADTs – bereitzustellen, wodurch mit Sicherheit die Qualität der Testfälle leidet. Schließlich ist die Grundvoraussetzung von strikt defensiv programmierten Modulen für reale Softwareprojekte kaum haltbar.

11.3.3.3 Fallbeispiel: Modellbasierter Test mit StP/T

StP/T ist ein in die Softwareentwicklungsumgebung StP integriertes[1] Testwerkzeug [Pos94]. Poston beschreibt in seinem Artikel, wie in einem Entwicklungsprojekt der Systemtest durch Extraktion von Informationen aus Analyse- und Entwurfsdokumenten (teil-)automatisiert werden konnte. Zunächst wurden mit den Modellierungswerkzeugen von StP entsprechend der UML Analyse- und Entwurfsdokumente erstellt. Aus diesen wurden von StP/T Testfälle anhand von sechs unterschiedlichen Testverfahren erzeugt:

- Funktionaler Test: Für jede Operation im Klassenmodell wird wenigstens ein Testfall erzeugt. Zur Wahl von Parameterobjekten werden die folgenden Testverfahren angewendet.
- Aquivalenzklassentest: Für alle Eingabebereiche werden Testfälle erzeugt.
- Grenzwertanalyse: Es werden Testfälle erzeugt, die Grenzwerte für alle Eingabebereiche erfordern.
- Logische Bedingungen: Testfälle werden erzeugt, die alle logischen Bedingungen mindestens einmal wahr und einmal falsch werden lassen.
- Ereignis-basierter Test: Für jedes von außen kommende Ereignis wird ein Testfall generiert.
- Zustands-basierter Test: Anhand der Zustandsdiagramme werden Testfälle generiert.

Leider bleibt Poston in seinen Ausführungen zur Testfallgenerierung sehr oberflächlich, sodass kaum geklärt werden kann, was genau generiert wird, geschweige

[1] In der aktuellen Version von StP wird StP/T leider nicht mehr angeboten.

denn, wie der Generierungsprozess arbeitet. Auch bleibt unklar, wie die herkömmlichen Techniken wie Grenzwertanalyse und Äquivalenzklassentest ohne Anpassungen auf die objektorientierte Welt übertragen wurden. Schließlich müssen die Testdaten vom Tester selbst eingegeben werden, was bedeutet, dass die Testfallermittlung automatisiert wird, ohne die Testdatengenerierung zu automatisieren.

11.3.3.4 Fallbeispiel: Testfallermittlung aus algebraischen Spezifikationen

Frankl und Doong beschreiben ein Verfahren und ein entsprechendes Testwerkzeug, welches Testfälle anhand einer algebraischen Spezifikation generiert [DoFr91]. ASTOOT ist kein einzelnes, monolithisches Werkzeug, sondern besteht aus zwei ineinandergreifenden Teilsystemen:

- dem Testfallgenerator und
- dem Treibergenerator.

Wir betrachten die beiden Komponenten getrennt. Der Testfallgenerator besteht seinerseits aus einem Übersetzer für die algebraische Spezifikation, welches die Termersetzung erleichtert, die von der zweiten Komponente – dem Vereinfacher – angewandt wird, um anhand einer vom Benutzer vorgegebenen Nachrichtensequenz verkürzte Sequenzen zu erzeugen. Bei der Verkürzung einer Originalsequenz werden für jede Bedingung innerhalb eines Axioms zwei neue Sequenzen mit unterschiedlichen Pfadausdrücken erzeugt. Der Verkürzungsprozess wird so lange fortgesetzt, bis keine Axiome mehr auf die resultierenden Sequenzen anwendbar sind – also keine linke Seite eines Axioms mit einer Teilsequenz übereinstimmt. Entsprechend der Pfadausdrücke der erzeugten Nachrichtensequenzen müssen die resultierenden Testobjekte entweder gleich oder ungleich zum Testobjekt der Originalsequenz sein. Ein Testfall ist also ein Paar von Nachrichtensequenzen mit einer zusätzlichen Kennzeichnung, ob diese nach ihrer Ausführung gleiche oder ungleiche Testobjekte hinterlassen. Der Benutzer muss allerdings selbst dafür sorgen, dass die von ihm bereitgestellten Parameterobjekte den Pfadbedingungen entsprechen. Außerdem muss der Benutzer eine Methode zur Bestimmung der Gleichheit zweier Testobjekte implementieren.

Der Treibergenerator erzeugt anhand des Codes der zu testenden Klasse Code für einen Testtreiber, der anschließend übersetzt werden kann. Die zuvor generierten oder manuell erstellten Testfälle können mit dem übersetzten Treiber ausgeführt und evaluiert werden.

ASTOOT bietet umfangreiche Unterstützung bei der Testfallerstellung, während die Testdatengenerierung nur wenig erleichtert wird. Dadurch können leicht große Mengen von Testfällen erzeugt werden, deren Versorgung mit Parametern sehr mühsam geraten kann. Deshalb kann der Benutzer steuernd in den Prozess der Test-

fallgenerierung eingreifen, um die Anzahl der erzeugten Testfälle zu verringern. Außerdem wird vorgeschlagen, die Testfallgenerierung durch Methoden der künstlichen Intelligenz wie z.B. Resolutionssysteme weiter zu vereinfachen.

Ein positiver Aspekt von ASTOOT ist auf jeden Fall darin zu sehen, da der Treibergenerator auch unabhängig vom Testfallgenerator verwendet werden kann: Die Testausführung und -auswertung kann auch anhand manuell erstellter Testfälle erfolgen. Das besondere Format der in ASTOOT verwendeten Testfälle macht die Testauswertung auch recht robust gegenüber Änderungen der zu testenden Klassen, da die Testauswertung keine Informationen über die interne Repräsentation der Testobjekte benötigt.

11.3.3.5 Fallbeispiel: Ein Testwerkzeug für komplexe Objekte

Siepmann und Newton stellen das Testwerkzeug TOBAC zur Unterstützung insbesondere des Regressionstests im Falle komplexer Objekte vor [SiNe94]. TOBAC erleichtert neben der Testdurchführung vor allem die Testdatengenerierung und die Testauswertung komplexer Smalltalk-Objektstrukturen. Für beide Aktivitäten wird dem Tester eine umfangreiche graphische Benutzungsschnittstelle zur Verfügung gestellt – im Gegensatz zu den meisten anderen vorgestellten Testwerkzeugen, die sehr viel stärker an herkömmlichen Übersetzern orientiert sind.

Zur Erzeugung komplexer Objekte diskutieren Siepmann und Newton drei Möglichkeiten:

- Ausschließlich unter Benutzung existierender Methoden.
- Unter Benutzung spezieller Methoden wie z.B. Test-Konstruktoren etc.
- Durch rekursiven Aufbau über die Teilobjekte (Instanzvariablen).

Die erste Möglichkeit wird unter Hinweis auf Probleme beim Testen von Hardware, die nicht speziell für Testbarkeit entworfen ist, verworfen: Es kann Objekte innerhalb eines Objekts geben, auf die kein direkter Zugegriff möglich ist. Die korrekte Botschaftssequenz, die das Objekt in den gewünschten Zustand vor dem Test bringt, kann beliebig komplex werden und führt zur inhärenten Möglichkeit der Fehlermaskierung. Darüber hinaus basiert dieser Ansatz auf der Korrektheit der angewandten Methoden, was zu einem Abhängigkeits- und Reihenfolgeproblem beim Testen führen kann. In Siepmann und Newtons Worten:

> *"[...] this approach relies on the correctness of the methods that are used, which leads to a nasty dependency problem." [SiNe94].*

Die Benutzung spezieller, nur zum Test implementierter Methoden zur Erzeugung komplexer Objekte wird aufgrund von erhöhtem Testaufwand sowie Implementierungs- und wiederum Testproblemen bei komplexen Objekten verworfen. Es könnten jedoch auch Aspekte der Testdaten-Adäquatheit angeführt werden, da man ja die tatsächliche Implementierung und keine Modifikation derselben testen sollte.

Als einzig realistischer Ansatz zur Erzeugung komplexer Objekte, so Siepmann und Newton, wird der rekursive Aufbau über die Teilobjekte ausgewählt: Diese Methode passt sich automatisch an die Komplexität des zu erzeugenden Objekts an, indem das Erzeugungswerkzeug rekursiv auf die Teilobjekte angewandt wird, wobei die Rekursion bei einfachen Objekten (integer, real, char, string etc.) endet, welche mit den beiden ersten Ansätzen (ggf. interaktiv) erzeugt werden können. TOBAC verwendet hierzu spezielle Zugriffsmethoden, die vom Smalltalk-Laufzeitsystem bereitgestellt und z.B. auch vom Smalltalk Debugger verwendet werden. Unter Verwendung der Reflektion ist die verwendete Technik auch in Java möglich.
Ein mit Hilfe von TOBAC erstelltes Objekt wird nicht direkt innerhalb eines Testfalls abgespeichert, sondern es wird eine zusätzliche Interpretationsebene eingeführt, indem zunächst eine Objektbeschreibung – der Smalltalk Code zur Erzeugung eines solchen Objekts – abgespeichert wird. Die Erzeugung konkreter Objekte kann dadurch bis zur Ausführung eines Testfalls aufgeschoben werden. Diese Möglichkeit der Aufschiebung der Objekterzeugung soll die Wiederverwendbarkeit von Testfällen von Oberklassen verbessern. Die Autoren gehen davon aus, dass nicht nur das Testobjekt eines wiederverwendeten Testfalls durch ein Objekt der Unterklasse ersetzt werden muss – sie räumen diese Art von Ersetzung durch Unterklassenobjekte auch für alle Parameterobjekte ein. Mit Hilfe der in TOBAC verwendeten Objektbeschreibungen können diese Spezialisierungen beliebiger Parameterobjekte festgehalten werden, wodurch Testfälle von Oberklassen sich für ihre Anwendung auf Objekte der Unterklasse anpassen lassen.
Die Objektbeschreibung eines Objekts wird rekursiv für alle aggregierten Objekte interaktiv vom Benutzer erstellt. Für eine rein objektorientierte Sprache bzw. den entsprechenden Teil einer hybriden Sprache schlagen die Autoren folgende Informationen zur Beschreibung eines Testfalls vor:

- ein Orakel zur Prüfung der globalen Zustandsvariablen (Klassenvariablen etc.) vor und nach der Testausführung,
- die rekursive Erzeugungsvorschrift für die zu testende Klasse,
- die zu testende Botschaft bzw. Botschaftssequenz,
- die rekursive Erzeugungsvorschrift für die benötigten Parameterobjekte,
- ein Orakel zur Prüfung der Zustandsvariablen des zu testenden Objekts vor und nach der Testausführung,
- ein Orakel zur Prüfung des Ergebnissobjekts nach der Testausführung,
- ein Orakel zur Prüfung der Argumentobjekte nach der Testausführung und
- ggf. das erwartete Ausnahmebehandlungsobjekt inklusive Parameterobjekten bzw. -werten.

Der Objekterzeuger `ObjectCreator` erlaubt die interaktive Erzeugung von Objekten und Definition von Testdaten für die erzeugten Objekte. Er wird immer dann aufgerufen, wenn ein Objekt z.B. als Parameter einer Botschaft oder als Teilobjekt

benötigt wird. Hierbei hat der Tester sieben Möglichkeiten der Objekterzeugung bzw. Auswahl:

- Selektion einer Klasse aus einer Liste aller Klassen im Smalltalk-System.
- Eingabe eines Klassennamens.
- Selektion einer Klasse aus einer Liste von im aktuellen Testfall bereits verwendeten Klassen.
- Selektion eines Objekts aus einer Liste von im aktuellen Testfall bereits verwendeten Objekten.
- Selektion eines Objekts aus einer Liste von innerhalb der aktuellen Testmenge verwendeten Objekten.
- Eingabe eines Smalltalk-Ausdrucks.
- Auswahl des Smalltalk-Objekts `nil` (undefinierte Objektreferenz).

Ist ein Objekt erzeugt worden, wird seine Struktur vom Objekterzeuger analysiert. Für jede Zustandsvariable, welche nicht durch die Erzeugungsmethode an ein Teilobjekt gebunden worden ist, wird rekursiv eine neue Instanz des Objekterzeugers zur Definition eines Teilobjekts aufgerufen. Dieser Vorgang terminiert, wenn der Benutzer ein Objekt ohne oder mit ausschließlich elementaren, direkt einzugebenden Zustandsvariablen erzeugt hat. Der Erzeugungsvorgang für ein so definiertes Objekt kann in der TOBAC Object Description Language (TODL) zur späteren automatischen Erzeugung des Objekts gespeichert werden. Interessant bei den Erzeugungsarten sind vor allem die Punkte 4 und 5, da dort keine neuen Objekte erzeugt, sondern lediglich bereits vorhandene Objekte eines Testfalls bzw. einer Menge von Testfällen mehrfach verwendet werden. Die mehrfache Verwendung von Objekten eines Testfalls ist aber von grundlegender Bedeutung, um eine Überprüfung der Identität eines Objekts durchführen zu können.

TOBAC bietet im Werkzeug `TestOracle` sieben unterschiedliche Arten der Überprüfung an:

- Gleichheit zweier Objekte.
- Identität zweier Objektreferenzen.
- Objektveränderung aufgetreten oder nicht.
- Bestimmtes Teilobjekt verändert oder nicht.
- Ist das Objekt das Smalltalk-Objekt `true`.
- Ist das Objekt das Smalltalk-Objekt `false`.
- Ist das Objekt das Smalltalk-Objekt `nil`.

Die Konstruktion des Testorakels verläuft analog zur Objekterzeugung wieder rekursiv über alle Teilobjekte. Dadurch kann z.B. interaktiv bestimmt werden, da ein bestimmtes Teilobjekt eines Testobjekts `t.s1` identisch mit einem Parameterobjekt `p1` sein soll, während die übrigen Teilobjekte des Testobjekts `t.s2, . . . , t.sn` nach Abarbeitung einer bestimmten Nachricht `t.m1(p1)` unverändert bleiben

sollen: Das geschilderte Verhalten ist typisch für Methoden zum Setzen von Instanzvariablen und lässt sich ohne die Unterscheidung zwischen Objektgleichheit und Objektidentität nicht überprüfen.

In der speziellen Klasse `TestCase` wird die inkrementelle Definition von Testdaten ermöglicht. Objekte dieser Klasse können sich in den Zuständen Objektdefinition, Botschaftsdefinition und Orakeldefinition befinden. Zum Regressionstest können vorhandene Testdaten anhand der Klassenhierarchie automatisch weiter verfeinert und modifiziert werden. Ähnliche Testdaten können zusammen mit den globalen Objekten zu Test-Suiten zusammengefasst und mit einem Testdaten-Browser durchsucht werden. Diese Testdaten können dann für Objekte bzw. abgeleitete Objekte aufgerufen werden.

TOBAC stellt somit eine umfangreiche Sammlung von Möglichkeiten zur Testauswertung bereit. Auch die Möglichkeit der Anpassung von Parameterobjekten bei der Wiederverwendung von Testfällen von Oberklassen ist vorteilhaft. Allerdings ist die rekursive, interaktive Konstruktion komplexer Objekte sicher ein aufwändiges Unterfangen – zumal bei der Konstruktion des Testorakels die komplexe Objektstruktur erneut traversiert werden muss. Wie die Autoren schreiben, eignet sich der gewählte Ansatz nur für rein objektorientierte Sprachen. Die Implementation von TOBAC basiert sehr stark auf den mitgelieferten Systemklassen des verwendeten Smalltalk-Systems. Eine Erweiterung auf hybride Sprachen wie z.B. C++ ist in diesem Rahmen nicht möglich. Außerdem muss angemerkt werden, dass TOBAC keinerlei Mechanismen zur Unterstützung der Testfallermittlung und der Sollwertbestimmung bereitstellt.

11.3.4 Testprozedur-Erstellung

Testprozeduren – in vielen Werkzeugen auch Skripte genannt – stellen einen Großteil der Testware dar. Viele kommerzielle Testausführungs- und Aufzeichnungswerkzeuge (capture and replay) bieten komplexe Programmierumgebungen für ihre Skriptsprachen an, wobei die Sprache selbst oft ein Dialekt einer Programmiersprache wie z.B. Basic oder Java ist. Abbildung 11.8 zeigt die Programmierumgebung für das Testwerkzeug Rational Robot®.

Als Werkzeuge zur Testprozedurerstellung können einerseits Standard-Software-Entwicklungsumgebungen (SEU) zum Einsatz kommen, wenn die Testprozeduren in einer Standard-Programmiersprache ggf. unter Verwendung spezieller Test-Rahmenwerke wie z.B. JUnit entwickelt werden. Auf der anderen Seite bieten die meisten kommerziell vertriebenen Testwerkzeuge spezielle Editoren, Kompiler und Debugger für ihre proprietäre Skriptsprache an.

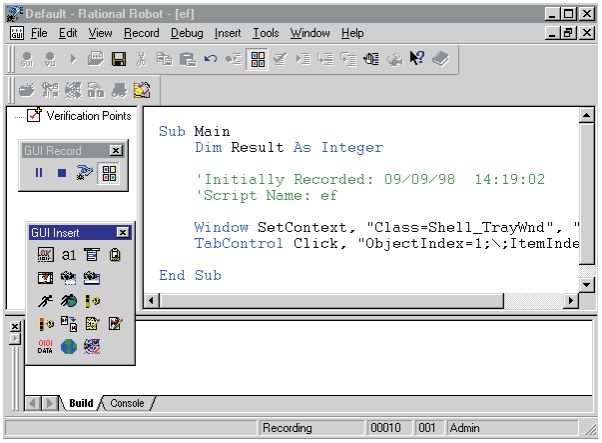


Abbildung 11.8 Programmierumgebung für die Test-Skriptsprache

11.3.4.1 Testdatengeneratoren

Als Testdaten werden aus den Testfallspezifikationen konkrete Daten bestimmt, sodass man ausführbare Tests als Nachrichtensequenzen und Parameterobjekte auf der Ebene der verwendeten Programmiersprache erhält. Je nach Grad der Formalisierung der Testfallspezikationen können hier Werkzeuge eingesetzt werden, die Testdaten automatisch generieren.

11.3.4.2 Fallbeispiel: Make Test Cases (MKTC)

Turner und Robson verwenden Testskripts, anhand derer Testfälle generiert werden [TuRo92]. Die Skripts werden von dem Werkzeug MKTC (MaKe Test Cases) in eine Treiberklasse umgeformt. Die Skript-Sprache macht Gebrauch von der Implementierungssprache der zu testenden Klasse. Allerdings sind die Skripts feiner gegliedert, um neben der Testdurchführung auch die Testdatengenerierung und die Testauswertung zu erleichtern. MKTC unterstützt außerdem die zustandsbasierte Testfallerstellung. Ein Testskript besteht aus sieben Abschnitten:

- Der *Kopf* enthält den Namen der zu testenden Klasse, die benötigten `include`-Dateien und Dokumentationsinformationen wie den Namen des Testers oder das Erstellungsdatum.
- Für die *Zustandsdeklarationen* wird jeder Zustand durch eine eindeutige Nummer, die Anzahl der möglichen Werte und zusätzliche Kommentaren spezifiziert.
- In *Data Scenarios* werden die Werte der konkreten Instanzvariablen auf die Zustände abgebildet.

- Durch die Definition einer *Invariante* können illegale Testfälle bei der Treibergenerierung ausgeschlossen werden. Außerdem verringert sie den Arbeitsaufwand für die Sollwertbestimmung.
- *Generatoren* dienen zur Erzeugung von Objekten in bestimmten Zuständen. Generatoren können zu Ketten verknüpft werden und erleichtern so die Testdatengenerierung.
- *Extra-Operationen* dienen zur Erfassung von zusätzlichen Hilfsmethoden zum Test.
- *Testfälle* haben letztendlich die Form „S_1, `Änderung`, S_2, `Nachbereinigung`". Hierbei bezeichnen S_1 und S_2 Zustände, `Änderung` den Code für die Zustandsänderung und `Nachbereinigung` den Code für abschließende Aktionen wie z.B. das Schließen von zuvor geöffneten Dateien. Bzgl. des Eintrags `Änderung` ist zu beachten, dass neben der zustandsändernden Nachricht auch für die Bereitstellung von Parameterobjekten gesorgt werden muss.

Insbesondere das Konzept der Generatoren erleichtert die Testdatengenerierung. Bei der Testfallerstellung kann sich der Tester auf die wesentlichen Konzepte des Zustandstests konzentrieren: Zustände und Zustandsübergänge. Diese Vorteile sind mit der festen Bindung des Werkzeugs an die zustandsbasierte Testmethodik verbunden. Als Nachteil ergibt sich also eine mangelnde Flexibilität für die Unterstützung anderer Testverfahren.

11.3.5 Testaufbau

Der Testaufbau zielt auf den Aufbau der Testumgebung, die Installation der Testwerkzeuge und die Bereitstellung der Testdaten, die Vorbereitung der Testprozeduren und die Übernahme der Testgegenstände. Das Ergebnis ist eine fertige Testumgebung samt Hard- und Software.

Im Rahmen des Testaufbaus im Entwicklertest können Werkzeuge insbesondere bei der Erstellung von Test-Treibern und -Stellvertretern helfen, da zumindest die Generierung von syntaktisch verwendbaren „Skeletten" der Treiber- und Stellvertreterklassen aus dem Quellcode der zu testenden Klasse eine rein mechanische Angelegenheit darstellt.

Unter Ausnutzung der in weiteren Entwicklungsdokumenten wie z.B. Klassendiagrammen und insbesondere Sequenzdiagrammen, welche die Abläufe von Operationsausführungen darstellen, können jedoch z.T. auch mit Implementationen der Operationen versehene, „semantisch" einsetzbare Treiber und Stellvertreter erzeugt werden.

11.3.5.1 Fallbeispiel: Testaufbau aus Sequenzdiagrammen

Fraikin, Henapl und Leonhardt von der TU Darmstadt sehen den Grund dafür, dass trotz der vorhandenen theoretischen Grundlagen auch heute noch in vielen Bereichen der objektorientierten Anwendungsentwicklung unzureichend oder ineffizient getestet wird, vor allem in der mangelnden Integration des Testens in den Entwicklungsprozess. Dies ist ihrer Ansicht nach jedoch weniger ein theoretisches Problem als vielmehr ein praktisches. Zwar sehen viele Vorgehensmodelle in der Softwareentwicklung (Rational Unified Process, Extreme Programming etc.) Testen bereits zu frühen Zeitpunkten vor, es fehlen jedoch die notwendigen Werkzeuge, um das Testen effizient in einem frühen Stadium in den Softwareentwicklungsprozess zu integrieren.

Aus diesen Überlegungen heraus wurde an der Technischen Universität Darmstadt SeDiTeC (Sequence Diagram Test Center) entwickelt, ein Werkzeug zum entwicklungsbegleitenden Testen von Java-Programmen. Die Eingabe für die Ausführung von Tests durch SeDiTeC besteht aus UML Sequenzdiagrammen, die um Daten für Argumente und Rückgabewerte der Methodenaufrufe erweitert werden und somit ausführbar sind.

Der Vorteil dieser Art der Testspezifikation liegt darin begründet, dass Sequenzdiagramme in (auf der UML beruhenden) Entwicklungsprozessen häufig bereits während der Analyse- und der Designphase erstellt werden und nur um entsprechende Testdaten zu erweitern sind (Abbildung 11.9). Aus diesem Grund kann die Testspezifikation in vielen Bereichen bereits vorgenommen werden, bevor die Implementierung überhaupt begonnen hat. Es müssen hierfür allerdings erheblich mehr Sequenzdiagramme erzeugt werden, als das bei momentanen Vorgehensweisen üblicherweise der Fall ist. Da es sich bei Sequenzdiagrammen um ein leicht verständliches und weitverbreitetes Mittel zur Kommunikation handelt, fällt dies jedoch vergleichsweise leicht. Damit ist ein entwicklungsbegleitendes Testen ohne großen zusätzlichen Aufwand von Beginn der Implementierungsphase an möglich.

SeDiTeC benötigt drei Arten von Informationen für die Durchführung eines Tests:

- Informationen über zu testende Sequenzdiagramme
- Informationen über Klassen und zugehörige Methoden, die in den zu testenden Sequenzdiagrammen vorkommen
- Argumente und Rückgabewerte der in den zu testenden Sequenzdiagrammen vorkommenden Methoden (Testfälle)

Die beiden erstgenannten Informationsarten sind in jedem beliebigen UML CASE-Tool vorhanden und können meist mit relativ geringem Aufwand extrahiert werden (derzeit besteht hierfür eine Schnittstelle zu Together®). Die relevanten Testfalldaten (Argumente und Rückgabewerte) werden über eine grafische Benutzungsoberfläche eingegeben und verwaltet.

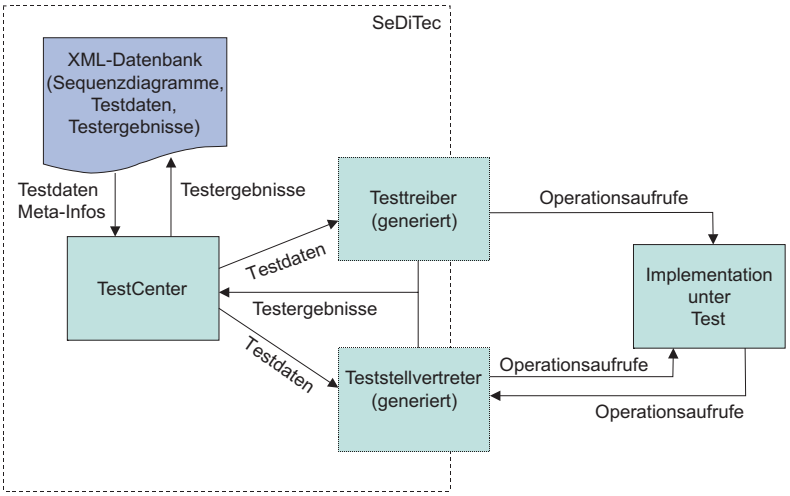


Abbildung 11.9 Komponenten von SeDiTeC

Bei einem Testlauf ruft der Testtreiber dann eine oder mehrere Methoden auf dem Testobjekt auf. Problematisch ist hierbei, dass das Testobjekt in vielen Fällen seinerseits weitere Objekte oder Komponenten verwendet, die u.U. noch nicht zur Verfügung stehen. Daher werden für spezifizierte, aber noch nicht implementierte Klassen automatisch Teststümpfe generiert (s. Abbildung 2). Diese beziehen ihr spezifiziertes Verhalten vollständig dynamisch zur Laufzeit, sodass sie – einmal generiert – für beliebige Sequenzdiagramme lauffähig sind. Ein selektives Entwickeln und Testen (Unit Test) einzelner Klassen und Komponenten eines großen Systems ist somit möglich. Der Aufwand für Integrationstests geht deutlich zurück.

Als Hauptvorteile von SeDiTeC nennen die Entwickler

- die Verkürzung der Entwicklungszeit, da auf Integrationstests weitgehend verzichtet werden kann;
- geringere Entwicklungskosten, da Fehler früher erkannt und beseitigt werden können;
- die Möglichkeit, sogar eine einzelne Klasse unabhängig vom Rest des Systems zu entwickeln und zu testen;
- mehr Vertrauen in das Gesamtsystem zu erhalten, da es jederzeit vollständig gegen die Spezifikation getestet werden kann;
- eine leichtere Integration des Testens in den Softwareentwicklungsprozess.

Da in heutigen Entwicklungsprozessen das Erstellen von Sequenzdiagrammen zwar häufig vorgesehen ist, aber nur in seltenen Fällen in ausreichendem Maße betrieben wird, um ein umfassendes Testen der betroffenen Klassen zu ermöglichen, sehen Fraikin, Henapl und Leonhardt als Perspektiven die Entwicklung von Methoden, die das Erstellen von Sequenzdiagrammen erleichtern bzw. teilweise automatisie-

ren. Denkbar sind hier Capture/Replay-Werkzeuge, die eine entsprechende Abbildung auf Sequenzdiagramme vornehmen oder auch Statische-Analyse-Werkzeuge, die bereits bestehenden Code analysieren können.
Insbesondere ließe sich mit Hilfe solcher Werkzeuge das Problem entschärfen, dass bei Änderungen im Design bzw. der Implementierung die zugehörigen Sequenzdiagramme ebenfalls geändert werden müssen, da dieser Vorgang zumindest teilweise automatisiert erfolgen kann. Beide Ansätze werden derzeit in weiteren Entwicklungen an der TU Darmstadt verfolgt.

11.3.6 Testausführung

Die Testfälle sind in einer festgelegten Reihenfolge unter Verwendung der ermittelten Testdaten auszuführen und die Testergebnisse zu protokollieren. Angefangen von einfachen Capture/Replay-Werkzeugen, welche die manuelle Bedienung eines Anwendungssystems aufzeichnen und dann immer wieder automatisch nachstellen können, bis hin zu komfortablen Werkzeugen mit Skriptsprachen und Möglichkeiten zur Testparametrisierung und Verschachtelung fallen die meisten Testwerkzeuge in diesen Bereich.
Aufgrund der Abhängigkeit vom Layout und den Elementen der Bedienoberfläche sind die Anpassungen der Testfälle für Regressionstests nach entsprechenden Änderungen manuell äußerst mühsam durchzuführen, wenn nicht direkt von Anfang an Wert auf Modularität und Parametrisierbarkeit der Testskripte gelegt wurde.

11.3.6.1 Fallbeispiel: Automated Class Exerciser (ACE)

Murphy, Townsend und Wong beschreiben in [MTW94] ein Werkzeug namens „Automated Class Exerciser (ACE)“ zur Unterstützung des Klassentests. ACE ist eine Erweiterung des Werkzeugs PGMEM zum Test von C Modulen. Das Werkzeug soll vor allem den Regressionstest – daher die Testdurchführung – erleichtern. ACE verwendet keine weiteren Informationen außer dem Quellcode (unterstützt werden C++ und Eiffel) und einer Testskriptsprache. Jedes Testskript zum Test einer einzelnen Klasse besteht aus drei Abschnitten:

- Umgebungsabschnitt: enthält Informationen zum Übersetzen und Binden.
- Globaler Abschnitt: erlaubt die Definition globaler Variablen und Funktionen in der Implementierungssprache der zu testenden Klasse.
- Testfall-Sektion: Die einzelnen Testfälle werden hier unter eindeutigen Namen aufgelistet. Dabei wird wiederum die Implementierungssprache der zu testenden Klasse verwendet, um sowohl die Nachrichtensequenz als auch das erwartete Ergebnis zu beschreiben.

Anhand des Testskripts wird ein Testtreiber erzeugt. Bei diesem Generierungsprozess werden auch die Testskripts der Oberklassen einbezogen, um Testfälle von Oberklassen wiederverwenden zu können. Nach Ausführung des Testtreibers wird eine Teststatistik ausgegeben, die Informationen zur Anzahl der ausgeführten Testfälle, zur Anzahl der fehleroffenbarenden Testfälle und Hinweise auf fehlerverursachende Methoden enthält.
Zum Test von abstrakten Klassen wird durch das Werkzeug eine minimale Unterklasse gebildet, die abstrakte Methoden so definiert, dass bei ihrem Aufruf jeweils eine Ausgabe erzeugt wird, die besagt, dass die entsprechende Methode gerade ausgeführt wurde.
Zum Test von generischen Klassen (Templates) wird jeweils eine minimale Instanziierung automatisch durch das Werkzeug vorgenommen. Dabei wird die Wurzelklasse aller weiteren Klassen im Eiffel-System – die Klasse `ANY` bzw. `Object` – verwendet, falls der Klassenparameter der generischen Klasse keine Einschränkungen zur Instanzierung einschließt. Falls Einschränkungen für die Instanziierung vorliegen, wird die Instanziierung mit der Klasse des Klassenparameters vorgenommen oder mit ihrer minimalen Unterklasse, falls der Klassenparameter eine abstrakte Klasse bezeichnet.
Grundsätzlich muss für die Erstellung der Skripts ein sehr hoher Aufwand vorgesehen werden, da alle Testaktivitäten außer der Testausführung manuell durchgeführt bzw. vorbereitet werden müssen. Außerdem dürften die Mechanismen zur Unterstützung von abstrakten und generischen Klassen oft nicht ausreichen, da ACE beide Konzepte auf einer rein syntaktischen Ebene behandelt, ohne deren Semantik zu berücksichtigen.

11.3.6.2 Fallbeispiel: FOOST

Rüppel stellt in seiner Dissertation die Architektur eines Rahmenwerks vor, mit dem sich Werkzeuge zur Unterstützung verschiedenster objektorientierter Testverfahren konfigurieren lassen [Rüp96]. Innerhalb dieses FOOST (Framework for Object-Oriented Software Testing) genannten Testrahmenwerks werden die invarianten Eigenschaften objektorientierter Testverfahren (Testfälle als Nachrichtensequenzen, Regressionsfähigkeit, ...) in Kernklassen realisiert (Abbildung 11.10). Kernklassen repräsentieren also ein allgemeines Modell von Treibern, Testfällen und Testfallsammlungen für den objektorientierten Test. Unterschiedliche Merkmale der Testverfahren wurden demgegenüber in separate Klassenhierarchien ausgelagert: Evaluierungsklassen repräsentieren unterschiedliche Arten der Testauswertung. Generatorklassen repräsentieren verschiedene Mechanismen der Testfallerzeugung, Testdatengenerierung und Sollwertbestimmung. Vermittlerklassen repräsentieren Techniken zur Protokollierung bei der Testdurchführung.

- Evaluierungsklassen entkoppeln Kernklassen und Generatorklassen von unterschiedlichen Varianten des Abgleichs von Ist-Ergebnissen mit Sollwerten. Objekte dieser Klassen können untereinander verknüpft werden, um komplexe Auswertungstechniken zu ermöglichen und um Anpassungen der Auswertung beim Unterklassentest zu ermöglichen.
- Objekte von Generatorklassen traversieren und modifizieren Objekte von Kernklassen. Elementare Generierungsmechanismen (wie z.B. Generatoren für Nachrichtensequenzen) können in Form von Generatorobjekten mit weiteren Generierungsmechanismen (z.B. Testdatengeneratoren) kombiniert werden. So können komplexe Testwerkzeuge aus Generatorobjekten zusammengesetzt werden. Dabei ist zu beachten, dass der Austausch eines Bestandteils – also eines elementaren oder bereits zusammengesetzten Generatorobjekts – ein neues Testwerkzeug hervorbringt. Hierdurch kann der Aufwand zur Erstellung neuer Testwerkzeuge erheblich reduziert werden.
- Durch Objekte von Vermittlerklassen können unterschiedliche Protokollierungsmechanismen an Objekte der zu testenden Klassen gebunden werden. Auch Objekte von Vermittlerklassen können verkettet werden, sodass unterschiedliche Protokollierungsmechanismen gleichzeitig in einem konkreten Testwerkzeug angewendet werden können.

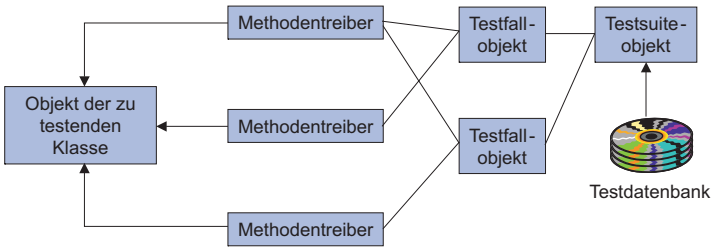


Abbildung 11.10 Struktur der Kernklassen des Rahmenwerks FOOST [Rüp96]

Objekte der drei Klassenhierarchien können frei kombiniert werden, wenn ein neues Testwerkzeug entwickelt werden soll. Dadurch wird es möglich, einzelne Bestandteile von Testwerkzeugen bei der Konstruktion anderer Werkzeuge einzusetzen: Generatoren für eine bestimmte Testtechnik können also mit Auswertungs- und Protokollierungsmechanismen anderer Testverfahren zusammen eingesetzt werden. Gegenüber den herkömmlichen Ansätzen zur Konstruktion von Testwerkzeugen eröffnet der vorgestellte Rahmenwerk-basierte Ansatz sehr viele Möglichkeiten, um Bausteine von Testwerkzeugen auf einfache Art und Weise wieder zu verwenden.

11.3.6.3 Dynamische Analysatoren

Gemäß dem gewählten White-Box-Testkriterium müssen gegebenenfalls zusätzliche Protokollierungsmechanismen im Programmcode verankert werden. Beim traditionellen Test fügt man z.B. zusätzliche Variablen ein, anhand derer man feststellen kann, wie viel Prozent der ausführbaren Zweige beim Test durchlaufen wurden.

Beim Test objektorientierter Software ist es im Fall von Testgegenständen, deren Verhalten große Zustandsabhängigkeit zeigt, möglich, endliche Automaten im Laufzeitsystem zu verankern, sodass nach der Testdurchführung festgestellt werden kann, wie hoch die Überdeckung von Zuständen und Zustandsübergängen ist. Weitere Ideen zur Instrumentierung finden sich in [HoSt94][Hof95][KGH+94]. Außerdem existieren Ansätze, traditionelle Überdeckungsmaße – und damit auch die zugehörigen Arten der Instrumentierung – für den objektorientierten Test einzusetzen [Bin95][PBC93]. Diese Verfahren benötigen dazu allerdings jeweils bestimmte benutzerdefinierte Operationen zur Abbildung des konkreten Zustands der Implementierung in den abstrakten Zustandsraum der Spezifikation, d.h. auf das Zustandsdiagramm.

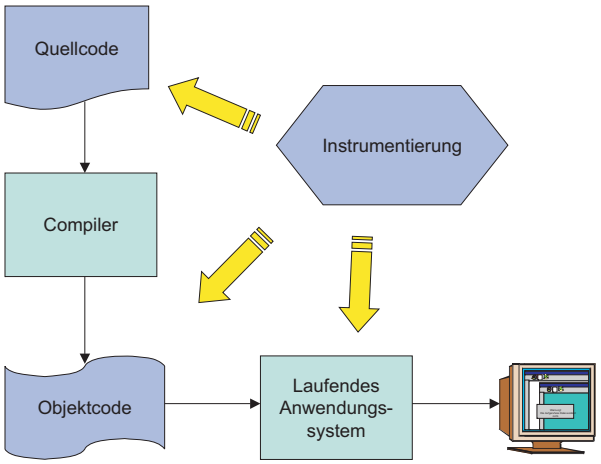


Abbildung 11.11 Instrumentierungstechniken

Bezüglich des Ziels der Instrumentierung unterscheidet man die in Abbildung 11.11 skizzierten drei Möglichkeiten:

- Die Instrumentierung des Quellcodes ist für fast alle Sprachen und Umgebungen möglich und wird von der überwiegenden Mehrheit der Werkzeuge durchgeführt. Diese Technik kann aber zu unvorhersehbaren Performanzverfälschungen und – bei Sprachen ohne automatische Speicherbereinigung – zur Maskierung von Speicherfehlern führen.

- Die Instrumentierung des vom Kompiler erzeugten Objektcodes bedingt eine starke Kopplung von Testwerkzeug und Entwicklungsumgebung und ist daher nur in Einzelfällen zu finden. Vorteile sind die vorhersehbare Beeinflussung der Performanz und die feingranulareren Auflösungsmöglichkeiten.
- In Ausnahmefällen kann die Instrumentierung im Laufzeitsystem erfolgen, wenn die Programmiersprache ausreichend starke reflektive Fähigkeiten hat. Vorteile sind der wegfallende Instrumentations- und Kompilationszyklus, Nachteile die geringe Auflösung (meistens nur bis zur Ebene ganzer Methoden möglich).

11.3.6.4 Fallbeispiel: Instrumentierung mit Makros

Hunt gibt Techniken zum Test des Laufzeitverhaltens und zur Instrumentierung von C++-Programmen an [Hun95b][Hun95c]. Bzgl. der Instrumentierung zeigt Hunt eine auf Makros basierende Implementierungstechnik, bei der Programmierer zunächst mit Bezeichnern versehene Bedingungen in den Programmcode einstreuen sollen. Zusätzlich werden einfache Datenverwaltungstechniken benutzt, um abzuspeichern, welche dieser Bedingungen bei bisher durchgeführten Tests jeweils welche Wahrheitswerte angenommen haben. Dadurch wird es möglich zu prüfen, wie gut die bisher durchgeführten Tests die eingestreuten Bedingungen erfasst haben. Diese Art der Instrumentierung ist insofern interessant, weil durch sie nicht die Abdeckung von Code-Strukturen, sondern eher die Abdeckung von Grenzfällen gemessen wird. Nachteilig ist aber, dass die Bedingungen vom Programmierer selbst eingefügt werden müssen, sodass er indirekt Einfluss auf die Güte der Tests nimmt.

11.3.7 Testauswertung

Die protokollierten Testergebnisse sind mit den Sollwerten, der Testreferenz zu vergleichen. Dabei ist zu entscheiden, ob der Test einen Fehler offenbart hat oder nicht. Außerdem muss überprüft werden, ob der gewünschte Überdeckungsgrad – insbesondere bei White-Box-Verfahren – erreicht wurde.
Es existieren sehr unterschiedliche Ansätze, um den Abgleich der Testergebnisse durchzuführen:

- Oft wird der Abgleich anhand druckbarer Repräsentationen von Testobjekt und Parameterobjekten durchgeführt. Dies zunächst nur sehr wenig vorbereitende Maßnahmen, da zum Vergleich von druckbaren Repräsentationen, also Texten, fasst jede moderne Sprach- oder Betriebssystemumgebung Werkzeuge bereit stellt. Bei Änderungen der Repräsentation einer Klasse – z.B. durch Hinzufügen

einer neuen Instanzvariablen – müssen aber i.d.R. alle Testergebnisse zumindest einmal manuell ausgewertet werden, was eine erhebliche Arbeitsbelastung ausmachen kann.

- Voas, Miller, Sneed und Marick schlagen die Verwendung von Zusicherungen für die Testauswertung vor [Mar95][Sne95][VoMi95]. Da Zusicherungen auch in der konstruktiven Qualitätssicherung eine große Rolle spielen, ist ihr Einsatz in der Testauswertung günstigstenfalls ohne zusätzlichen Aufwand möglich.
- Doong und Frankl schlagen vor, spezielle Auswertungs- bzw. Vergleichsmethoden in den zu testenden Klassen zu implementieren, anhand derer die Gleichheit zweier Testobjekte festgestellt werden kann [DoFr91]. Dies wird insbesondere dann problematisch, wenn die Testobjekte Verbindungen zu anderen Objekten haben, über welche der Gleichheitstest (rekursiv) fortgesetzt werden muss (deep equality). Hierbei ist der Abbruch der Rekursion bei zyklischen Objektstrukturen sicherzustellen. In der Praxis wird die Gleichheit daher meistens nur über eine Verbindungsstufe hinweg geprüft.
- Mehrfach wurde (und wird) vorgeschlagen, die Testauswertung mit Hilfe von ausführbaren Spezifikationen vorzunehmen. Allerdings müssen die Entwickler oder Tester hierzu umfangreiches Vorwissen mitbringen, was in der Praxis leider oft nicht gegeben ist. Offen bleibt auch die Frage, wer bzw. wie ausführbare Spezifikationen selbst geprüft werden.
- Die Testauswertung bei GUI-Testwerkzeugen erfolgt im einfachsten Fall durch den Vergleich der resultierenden Bildschirmdarstellung auf Pixel-Ebene. Dies ist aufgrund der Plattform- und Konfigurationsabhängigkeit abzulehnen, sodass die Vergleiche mindestens auf der Ebene der vom grafischen Fenstersystem dargestellten Ein- und Ausgabefelder (Widgets) möglich sein müssen. Im besten Fall erkennt und verwendet das Werkzeug die Abhängigkeit von Elementen der Benutzungsoberfläche und den jeweils dargestellten Objekten des Anwendungssystems (Abbildung 11.12).

Unabhängig davon, wie die Testergebnisse überprüft werden, wird es immer einige aktuelle Ausgaben geben, die von der Testreferenz abweichen. Man denke z.B. an einen Report, in dessen Kopfzeile immer das aktuelle Datum angegeben ist. Um solche gewünschten bzw. erwarteten Abweichungen der aktuellen Ausgabe von der Referenz bei der Prüfung der Testergebnisse zu berücksichtigen, müssen die Testausgaben vor dem eigentlichen Vergleich mit der Referenz „bereinigt“ werden.

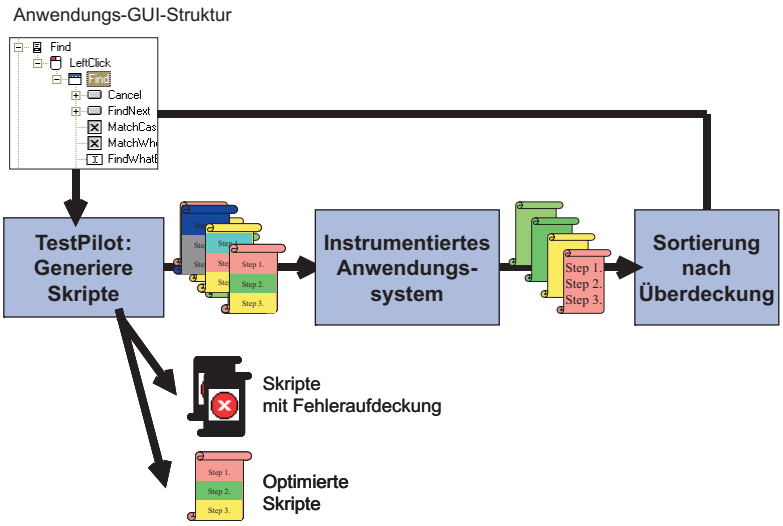


Abbildung 11.12 Abgleich der Testergebnisse und Optimierung der Skripte

Hierzu empfehlen Fewster und Graham, eigene Filterprogramme zu erstellen, die vor dem Vergleich sowohl die Referenz als auch die aktuelle Testaufgabe gezielt im Bereich solcher Felder angleichen ([FeGr99], vgl. Abbildung 11.13).

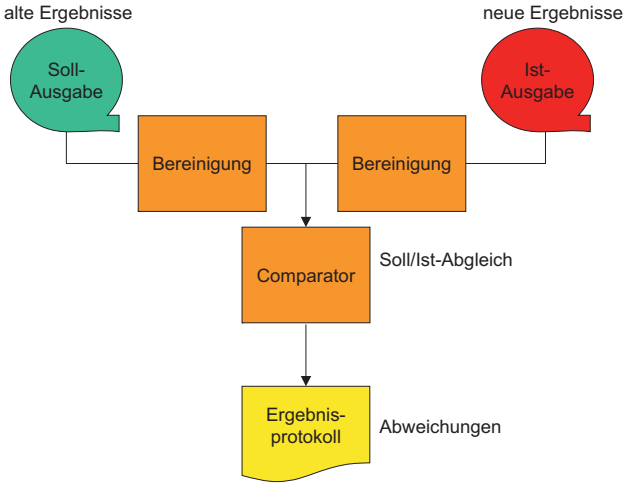


Abbildung 11.13 Vergleich der Ausgaben nach der Bereinigung

11.4 Empfehlungen zum Werkzeugkauf

Bei der Einführung von Testwerkzeugen sind grundsätzlich zunächst die zugrunde liegenden Testverfahren im Entwicklungs- bzw. Testprozess zu verankern. Hierbei empfiehlt es sich, sozusagen bottom-up vorzugehen und die „späteren" Testtätigkeiten zuerst zu automatisieren ([Rüp96]). Man beginnt also z.B. mit Werkzeugen zur Automatisierung des Testablaufs und der Testauswertung, für das man Testfälle manuell erstellt. Gleichzeitig kann man einen dynamischen Analysator zur Messung der Testüberdeckung des Programmcodes einsetzen. Haben sich diese Werkzeuge bewährt, kann ein Testdatengenerator hinzugenommen werden, welcher für die – weiterhin manuell erstellten – Testfälle (beliebig viele) Testdaten generiert und an das bereits vorhandene Werkzeug für den Testablauf und die Testauswertung übergeben. Werden formale Spezifikationen eingesetzt, so kann als Nächstes die Testfallspezifikation und ggf. zum Schluss die Sollwertbestimmung automatisiert werden.

In einem wichtigen Punkt bzgl. der Werkzeugunterstützung unterscheidet sich die Automatisierung von Tests für objektorientierte Software also keineswegs von der für herkömmliche Software: die Automatisierung von Aktivitäten sollte mit den nachgelagerten Phasen begonnen werden bzw. alle Phasen umfassen. In anderen Worten: Es macht keinen Sinn, vorgelagerte Testaktivitäten zu automatisieren, wenn nicht auch die nachfolgenden Testaktivitäten weitgehend automatisiert werden können [Rüp96]. Da z.B. durch die Automatisierung der Aktivitäten Testfallermittlung oder Testdatengenerierung meist eine sehr große Anzahl von Testfällen ermittelt wird, sollten in diesem Fall auch die nachfolgenden Aktivitäten – Testausführung und Testauswertung – automatisiert sein. Automatisierung der Testdurchführung und Auswertung ist aber beim Test objektorientierter Systeme besonders wichtig, da Regressionstests nicht auf die Wartung beschränkt sind, sondern bereits beim Klassentest, also bei der Implementation, obligatorisch sind.

Mit fast allen Testwerkzeugen können daher Testfälle wiederholt ausgeführt und ausgewertet werden Außerdem verschmilzt die Testfallermittlung und Testdatengenerierung oft zu einer einzigen Tätigkeit bei Anwendung einiger dieser Werkzeuge.

Anhand der vorgestellten Werkzeuge erkennt man außerdem ein grundlegendes Dilemma bzgl. der Konstruktion aller Testwerkzeuge: Einfache Hilfsmittel und Werkzeuge zur wiederholten Testausführung z.B. für Regressionstests wie z.B. jUnit oder die von Hunt vorgestellte Technik zur Speicherung von Tests sind zwar leicht anwendbar, da sie weder eine bestimmte Spezifikationssprache voraussetzen, noch eine bestimmte Testmethode implizieren. Allerdings belassen solche Werkzeuge auch einen großen Teil der Arbeit des Testers ohne Unterstützung, da weder Testdaten, noch Testfälle oder Sollergebnisse generiert werden.

Werkzeuge, die bestimmte strukturelle Testtechniken unterstützen, ermöglichen zwar die Testfallgenerierung, ohne jedoch die Generierung von Testdaten oder eine

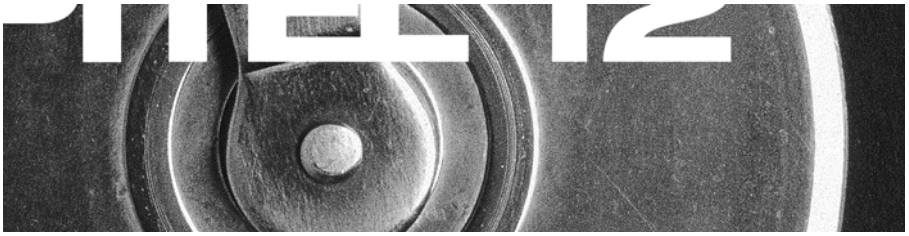
automatische Bestimmung von Sollwerten zu gestatten. Diese Testaktivitäten lassen sich nur von Werkzeugen mechanisieren, die mathematisch fundierte Spezifikationssprachen verwenden. Umgekehrt ist der Einsatzrahmen solcher Werkzeuge am meisten begrenzt, da sie sowohl die Verwendung einer bestimmten formalen Spezifikationssprache voraussetzen, als auch eine bestimmte Testmethode implizieren.

Dabei besitzen sowohl die dargestellten Testverfahren als auch die in diesem Kapitel vorgestellten Werkzeuge für ihre Unterstützung neben Unterschieden auch viele Ähnlichkeiten und Überschneidungspunkte. Zusammenfassend lässt sich das Verhältnis von Testwerkzeugen zu Testverfahren sehr gut durch das folgende Zitat von Boris Beizer beschreiben:

> *"Typically, a [test] design automation tool is based on a specific testing technique - so we have functional and structural tools and almost as many tools as there are techniques." [Bei91]*

12

Objektorientiertes Testen in der Praxis



Testprozesse und Vorgehensmodelle
Stand kommerzieller OO-Testwerkzeuge
Zum Abschluss: Die Grundsätze des Testens

Inhaltsübersicht Kapitel 12

12 Objektorientiertes Testen in der Praxis

In diesem Kapitel wenden wir uns dem “state of the art“ des Testens objektorientierter Software in der Praxis zu. Zunächst beleuchten wir vor dem Hintergrund zweier Erhebungen der industriellen Praxis in Deutschland mehrere Vorgehensmodelle bezüglich der in ihnen beschriebenen Testprozesse. Dann wenden wir uns den in der industriellen Praxis verwendeten Testmethoden und -techniken zu und betrachten zum Abschluss des Buches noch kurz den Stand der kommerziell angebotenen Testwerkzeuge.

12.1 Testprozesse und Vorgehensmodelle

Zur momentanen Situation der Software-Qualitätssicherung in der Industrie bemerken Cusumano und Selby:

> *„Weder wir noch Microsoft glauben, dass mittlerweile alle Probleme gelöst seien und kein Produkt mehr verspätet oder fehlerfrei ausgeliefert würde. Dies kann ohnehin kein Software-Hersteller von sich behaupten.“ [CuSe96]*

Müller und Wiegmann von der Universität Köln untersuchten 1998 in Deutschland den „Stand der Praxis der Prüf- und Testprozesse in der Softwareentwicklung“ ([MüWi98]). Immerhin 47% der Befragten verwendeten C++, 19% Java als Programmiersprache. Bei der Auswertung der Fragebögen trafen Müller und Wiegmann auf eine Reihe alarmierender Fakten:

- Eine spezialisierte Testgruppe fehlt in den meisten Firmen.
- In 70% aller Fälle werden Prüfungen und Tests von Entwicklern durchgeführt.
- Hauptsächlich wird in den „späten“ Phasen der Softwareentwicklung geprüft; die wichtigen „frühen“ Phasen wie Anforderungsermittlung und Entwurf werden in Bezug auf die Qualitätssicherung vernachlässigt.
- 12 von 74 Softwarehäusern testen auch in der Realisierungsphase nicht oder nur teilweise.
- Nur bei wenigen Firmen erfolgt eine detaillierte Zeit- und Ressourcenplanung. Daher kommt es zu Engpässen, die bewirken, dass nur ein Teil der vorgesehenen Prüfungen durchgeführt werden können.

- Nur ein Drittel der Firmen unterstützen Schulungen im Bereich der Qualitätssicherung oder stellen Mitarbeiter der Fachabteilungen für die Tests frei.
- Bei über zwei Dritteln der Firmen wird der Soll-Ist-Vergleich bei den Tests nicht regelmäßig durchgeführt.
- Bezüglich formaler Kriterien zur Messung des Testfortschritts wird die „Funktionsabdeckung“ am häufigsten verwendet.
- Bei den Testwerkzeugen ist der Debugger das meistverwendete Werkzeug.

Als Gründe für die vernachlässigte Qualitätssicherung werden u.a. angeführt, dass

- dem hohen Aufwand beim manuellen Testen ein hoher Erstaufwand bei der Testautomatisierung entgegensteht;
- der Werkzeugeinsatz aufgrund mangelnder Schulung und fehlender Funktionalität Probleme bereitet;
- die Wiederherstellung von Testdaten (insbesondere bei datenbankgestützten Anwendungssystemen) schwierig ist.

Darüber hinaus hat die Organisationsabteilung oft zu wenig Zeit für den Test. Auch wird die Prüf- und Testgruppe bisweilen als stille Reserve für anderweitige Personalengpässe missbraucht. Insgesamt stellt man fest, dass die Schere zwischen Forschung und Praxis bei der Qualitätssicherung so weit wie nur in wenigen anderen Bereichen der Softwareentwicklung auseinander klafft.

Die im Jahr 2000 für das Bundesministerium für Bildung und Forschung (BMBF) durchgeführte Studie „Analyse und Evaluation der Softwareentwicklung in Deutschland“ stellt bei der Untersuchung von 671 mit der Softwareentwicklung befassten deutschen Unternehmen fest, dass die Bedeutung der Softwarequalität und deren Einfluss auf die Qualität des gesamten Produktes und Geschäftserfolgs mittlerweile von der Mehrzahl der Unternehmen erkannt wurde [BMBF00]. In der Studie wird diese Feststellung insbesondere durch die Beobachtung argumentiert, dass für Qualitätssicherung ein klares Rollen- und Methodenverständnis vorhanden sei. Der Akzent der Unternehmen in Bezug auf Qualitätssicherung liege jedoch in aller Regel auf den späten Phasen der Softwareentwicklung, d.h. im Bereich der Testdurchführung.

Nach Binder führen nur ca. 20% aller Organisationen zumindest einen systematischen, anwendungsfallbasierten Systemtest durch [Bin99a]. Charakteristisch hierbei ist, dass die Klassen stillschweigend als korrekt angesehen werden, was zu einer wachsenden Frustration aufgrund der hohen Fehlerrate und damit verbundenen Terminproblemen führt. Hier bietet sich zur Verbesserung des Testprozesses an, die Anwendungsfälle „testbar“ zu gestalten, d.h., z.B. mit Aktivitätsdiagrammen und der Angabe von Wertebereichen für die Interaktionsparameter zu spezifizieren [Win99]. Darauf basierend kann die Testausführung und später die Testfallgenerierung automatisiert werden.

Als Konsequenz der größtenteils erst in den späten Phasen der Entwicklung betonten Qualitätssicherung werden in vielen Projekten die hauptsächlichen Risiken viel zu spät erkannt. Abbildung 12.1 verdeutlicht, dass diese Risiken erst mit der wachsenden Anzahl aufgedeckter Fehler – von Berechnungsfehlern bis hin zu Performanzproblemen – zutage treten und somit frühestens mit Beginn der qualitätssichernden Maßnahmen wirksam eingedämmt werden können.

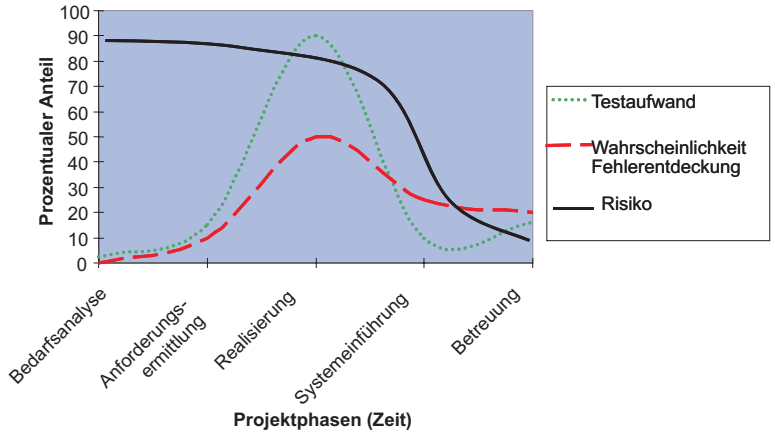


Abbildung 12.1 Späte Risikoeindämmung wegen zu später Qualitätssicherung

Die organisatorischen Rahmenbedingungen der Qualitätssicherung stellen sich in den verschiedenen Unternehmen unterschiedlich dar [BMBF00]:

- Ein großer Teil der befragten Unternehmen mit einer größeren Zahl an Softwareentwicklern hat eine eigene von der Softwareentwicklungs-Abteilung unabhängige Qualitätssicherungsabteilung, die einen Teil der Tests zum Zwecke der Qualitätssicherung selbst durchführt.
- Zunehmend sind Qualitätssicherungsmaßnahmen auch direkt in die Softwareentwicklung integriert, womit Projektleiter oder Entwickler selbst für die Qualität der Software verantwortlich werden. Als Begründung für die Wahl dieser Organisationsform gaben die Unternehmen an, dies schaffe für die Projektbeteiligten einen größeren Anreiz, qualitativ hochwertige Software zu entwickeln, erlaube, die Fehler schneller zu beheben und aus den Fehlern zu lernen.
- Nur ein geringer Teil der Unternehmen bezeichnete seine derzeitige Qualitätssicherung als unzureichend/rudimentär. Dabei handelt es sich überwiegend um sehr junge kleine Unternehmen (Start-ups). Auch hier war die Bedeutung der Qualitätssicherung im Bewusstsein der Verantwortlichen vorhanden. Einige dieser Unternehmen befinden sich gerade dabei, die Position des Qualitätsmanagers/Qualitätssicherers zu etablieren und das bestehende Vorgehen zu systematisieren.

Zusammenfassend stellt man also fest, dass ebenso wie im „konventionellen“ Test-Bereich auch – oder gerade – der Test objektorientierter Software in der Praxis weit hinter den Forschungsergebnissen zurück ist.

12.1.1 Generisches Prozessmodell

Die objektorientierte Anwendungsentwicklung erfolgt größtenteils in einem iterativen, inkrementellen Entwicklungsprozess und erzeugt mehrfach neue Versionen des im Entstehen begriffenen Anwendungssystems (Inkremente). Der Test objektorientierter Software ist somit letztlich nur dann beherrschbar, wenn er entwicklungsbegleitend, also auch iterativ und inkrementell erfolgt. Nur so, durch eine entwicklungsbegleitende Qualitätssicherung, wie in den vorigen Kapiteln dargelegt, können die Risiken schon früh im Projekt aufgedeckt und bekämpft werden (Abbildung 12.2).

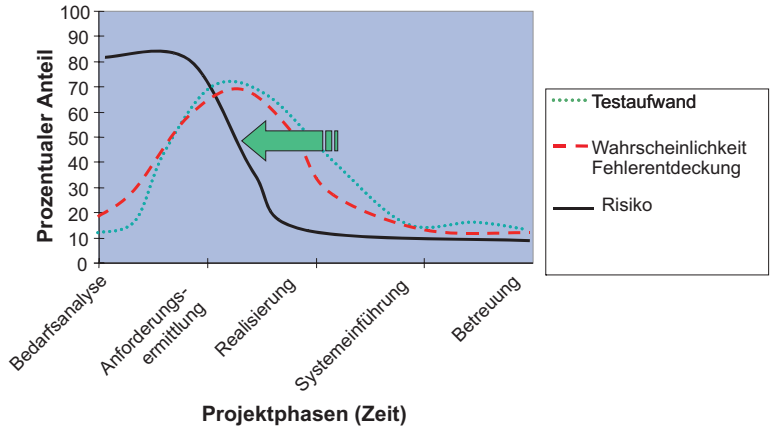


Abbildung 12.2 Frühe Risikoeindämmung durch frühzeitige Qualitätssicherung

Die Qualitätssicherung bzw. das „Testen“ ist ein „Schattenprozess“, der zeitlich parallel zum Entwicklungsprozess verläuft und über die Testobjekte eng mit diesem verzahnt ist [KPS00][Spi00]. Um die Abhängigkeiten explizit darstellen zu können, wird der objektorientierte Testprozess – ebenso wie die (konstruktiven) Anteile des Entwicklungsprozesses – in Elemente wie Rollen, Stufen, Phasen, Aktivitäten und Techniken unterteilt. Diese Unterteilung spiegelt verschiedene sachliche Aspekte wider:

- *Rollen* als organisatorischer Aspekt zeigen, wer etwas ausführt bzw. ausführen kann (und darf);
- *Testaktivitäten* als funktionaler Aspekt spezifizieren, was ausgeführt wird;

- *Teststufen* und *Testphasen* beschreiben den ablauf- oder verhaltensbezogenen Aspekt, also wann etwas ausgeführt wird;
- *Testtechniken* liefern den operationalen Aspekt und geben an, wie etwas ausgeführt wird;
- *Testaufgaben* als kausaler Aspekt geben an, warum etwas ausgeführt wird;
- *Testgegenstände* und *Artefakte* wie z.B. Spezifikationen, Pläne, Programmcode, Testfälle und Testberichte beschreiben letztendlich als informationsbezogener Aspekt die Daten, also: womit etwas ausgeführt wird.

In jeder Teststufe wie z.B. Klassentest, Integrationstest und Systemtest lassen sich somit verschiedene Testphasen wie z.B. Testplanung, Testentwurf und Testausführung identifizieren, innerhalb derer die einzelnen Testaufgaben angesiedelt sind. Zur methodischen Durchführung der Testaufgaben werden Testtechniken verwendet, die zum Teil nur für bestimmte Teststufen und deren Aktivitäten anwendbar sind, zum Teil aber auch stufenübergreifend. Aktivitäten werden von Mitarbeitern durchgeführt, deren Verantwortlichkeiten und jeweils geforderten Qualifikationen durch Rollen charakterisiert werden. Diese Zusammenhänge sind in Abbildung 12.3 skizziert.

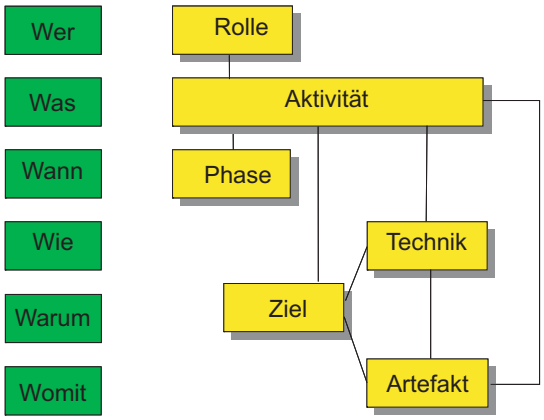


Abbildung 12.3 Elemente eines Testprozesses

Als Beispiele für definierte Testprozesse betrachten wir die Qualitätssicherungsanteile in den „schwergewichtigen“ Vorgehensmodellen:

- V-Modell 97 und
- Unified Software Development Process.

Mit dem leichtgewichtigen Vorgehensmodell

- Xtreme Programming

stellen wir darüber hinaus noch einen alternativen Ansatz für kleine Projekte bzw. gut abgrenzbare Teilprojekte vor, der in den letzten zwei Jahren zunehmend mit Erfolg eingesetzt wird.
Wir verwenden in den folgenden Kapiteln die bisher eingeführte Terminologie; zum Abgleich der im V-Modell benutzten Begriffe dient die im Anhang angegebene Vergleichstabelle.

12.1.2 V-Modell 97

Testen und Qualitätssicherung erhalten im V-Modell 97 alleine dadurch eine hervorragende Bedeutung, dass sie neben den Teilmodellen Systemerstellung (SE), Projektmanagement (PM) und Konfigurationsmanagement (KM) als eigenständiges Teilmodell „Qualitätssicherung“ (QS) etabliert sind.
Die im Teilmodell QS beschriebenen Maßnahmen dienen folgenden Zwecken:

- dem Nachweis der Erfüllung dieser vorgegebenen Anforderungen,
- der präventiven Vermeidung von Mängeln und
- der Sicherstellung einer Prozessqualität.

Zum einen wird Softwarequalität durch den Einsatz konstruktiver Maßnahmen erreicht, zum anderen werden konstruktive Maßnahmen durch analytische Maßnahmen ergänzt. Darüber hinaus ist bei Projekten mit Auftragsvergabe an externe (industrielle) Auftragnehmer durch geeignete Vertragsvereinbarungen dafür zu sorgen, dass der Auftraggeber in angemessener Form an den Qualitätssicherungsmaßnahmen beteiligt ist.
Der Grundsatz, dass Qualität nicht im Nachhinein in ein Produkt hineingetestet werden kann und es daher unerlässlich ist, die Erzeugung von Qualität durch konstruktive und analytische Maßnahmen während des gesamten Entwicklungsprozesses zu fördern, ist auch einer der Eckpfeiler der Qualitätssicherung im V-Modell. Zielsetzung ist außerdem, präventiv qualitätsrelevante Risiken zu vermeiden, Qualitätsmängeln entgegenzuwirken und die den analytischen Maßnahmen zu unterziehenden Prüfgegenstände überhaupt erst prüfbar zu gestalten.
Unter den konstruktiven/präventiven Maßnahmen nennt das V-Modell beispielsweise:

- die Gliederung des Entwicklungsprozesses durch Anwendung eines SW-Entwicklungsstandards und
- die Unterstützung des Entwicklungsprozesses durch Methoden und Werkzeuge.

Konstruktive/präventive Maßnahmen werden im Submodell QS festgelegt, die Anwendung der konstruktiven Maßnahmen erfolgt im Submodell SE.

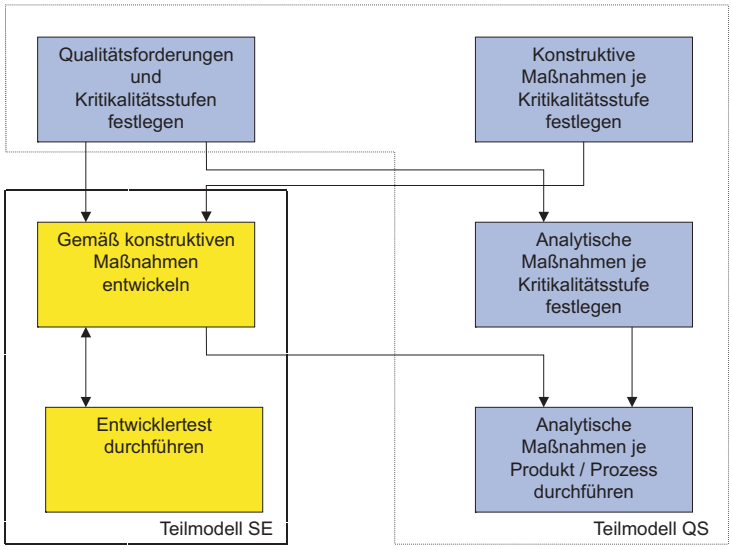


Abbildung 12.4 Konstruktive und analytische Maßnahmen im V-Modell 97

Analytische Maßnahmen haben die Prüfung, Bewertung und den (externen) Nachweis der Qualität der Testgegenstände zum Ziel und werden im Submodell QS sowohl festgelegt als auch durchgeführt. Analytische Maßnahmen betreffen die Produkte und die Aktivitäten aller Submodelle.

Das V-Modell unterscheidet eher informelle Entwicklertests und formelle Tests durch die unabhängigen Tester. Vom Entwickler sind Tests der von ihm realisierten Klassen und Komponenten und ggf. Datenbanken durchzuführen. Bei den Entwicklertests sind zunächst Testvoraussetzungen wie Installation des Testobjekts in der Testumgebung und Generierung der benötigten Testfälle herzustellen. Nach der Durchführung der Tests erfolgt die Auswertung der Testergebnisse, die den weiteren Entwicklungsverlauf bestimmt. Sofern der Test aus Sicht von SE erfolgreich war, erfolgt die Weitergabe des Testobjekts an QS zum Zweck des formellen Produkttests, sonst sind Iterationen von SE-Aktivitäten erforderlich, die mit einer erneuten Selbstprüfung des Produktes abzuschließen sind.

Die Aktivitäten und Produkte des Teilmodells QS werden im V-Modell 97 folgendermaßen aufgegliedert [VM97]:

QS 1	QS-Initialisierung			
	QS 1.1	Testplan erstellen	→	Testplan
	QS 1.2	Prüfplan erstellen		
QS 2	Testvorbereitung			
	QS 2.1	Testmethoden und -kriterien festlegen		
	QS 2.2	Testumgebung definieren	→	Testkonzept

	QS 2.3 Testfälle festlegen	→	Testfallspezifikationen
	QS 2.4 Testprozedur erstellen	→	Testprozeduren
QS 3	Prozessprüfung von Aktivitäten	→	Prüfprotokolle
QS 4	Produkttest		
	QS 4.1 Testbarkeit feststellen		
	QS 4.2 Produkt inhaltlich testen	→	Testergebnisse
QS 5	QS-Berichtswesen	→	Testberichte

Das V-Modell als die Basis für die technische Abwicklung eines Projekts zeichnet sich durch Allgemeingültigkeit aus und ist unabhängig vom Einsatzbereich. Diese Allgemeingültigkeit bedingt aber, dass nicht alle enthaltenen Regelungen für jedes beliebige Projekt relevant sind. Um für ein konkretes Projekt das V-Modell anwendbar zu machen, muss man deshalb zunächst entscheiden,

- welche Aktivitäten für die Durchführung des Projekts erforderlich sind und
- welche Produkte im Rahmen der Projektabwicklung erzeugt werden müssen.

Die damit verbundene Streichung *nicht* relevanter Aktivitäten und Produkte wird als Tailoring bezeichnet. Hauptanliegen des Tailoring ist es, für jedes Projekt zu gewährleisten, dass der eingesetzte Aufwand den Projektzielen dienlich ist. Hierdurch sollen Probleme wie z.B.

- übermäßige Papierflut,
- sinnlose Dokumente, aber auch
- das Fehlen wichtiger Dokumente

vermieden werden. Dies wird durch die Reduzierung der allgemeingültigen (generischen) Regelungen des V-Modells auf die aus sachlichen Gründen erforderlichen Regelungen erreicht. Die entstehende Teilmenge des V-Modells als „projektspezifisches V-Modell“ ist neben der Beschreibung des Projekts, seiner Organisation und seiner Ziele Hauptbestandteil des Projekthandbuchs (PHb) [VM97]. Das V-Modell erläutert weiter:

> *„Das V-Modell kann als Checkliste angesehen werden, die eine große Menge erprobter und sinnvoller Regelungen enthält. Für jedes Einzelprojekt wird aus diesem Angebot eine Teilmenge ausgewählt. Das V-Modell kann daher auch als ein ‚Baukasten‘ angesehen werden, aus dem im Rahmen des Tailoringvorgangs die für ein Projekt geeigneten Bausteine (Aktivitäten/Produkte) herausgenommen werden.“ [VM97]*

12.1.3 Unified Software Development Process

Der aus der Verschmelzung von Ivar Jacobsons „Objectory”-Prozess und dem bei der Firma Rational Software Corporation eingesetzten Firmeninternen Vorgehens-

modell entstandene „Unified Software Development Process“ (kurz: Unified Process[2]) baut auf sechs *Grundprinzipien* auf [JBR99]:

- iterative, inkrementelle Entwicklung,
- Architekturzentrierung,
- Modellierung,
- Qualitätsmanagement,
- Anforderungsmanagement und
- Konfigurationsmanagement.

Diese Prinzipien werden durch entsprechende Aktivitäten in einer phasenorientierten Vorgehensweise operationalisiert. Hierbei werden zeitlich gesehen vier so genannte *Phasen* unterschieden:

- Projektfindung und Konzeption (inception),
- Projektdefinition (elaboration),
- Konstruktion (construction) und
- Übergang in den Betrieb (transition).

Abbildung 12.5 zeigt, dass in jeder Phase die *Aktivitäten* in jeweils unterschiedlicher Gewichtung ausgeführt werden.

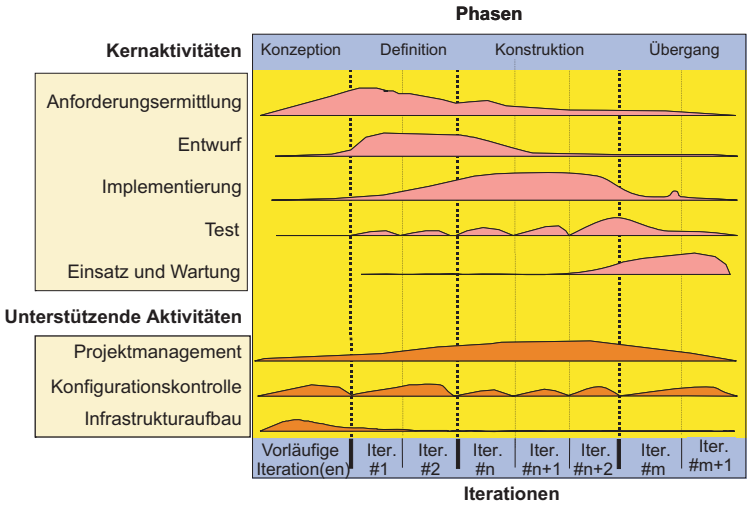


Abbildung 12.5 Aktivitäten und Phasen im Unified Software Development Process (nach [JBR99])

[2] Der Unified Software Development Process wird – in erweiterter und konkretisierter Form – von der Firma Rational Software unter der Bezeichnung “Rational Unified Process“ (RUP) als Kombination von HTML-Seiten, Produktmustern, Werkzeuganleitungen etc. vertrieben.

Die Software wird im Unified Process iterativ entwickelt, wobei in jeder Phase mehrere Iterationen (ca. 3-9) ausgeführt werden, deren jede in einem ausführbaren Produkt mündet, welches sich in bestimmten Aspekten von dem der Vorgänger-Iteration unterscheidet. Die Gesamtheit aller Unterschiede zwischen einer bestimmten Version und ihrer Vorgängerversion wird als Inkrement bezeichnet, woher sich die Bezeichnung iterative, inkrementelle Entwicklung ableitet. In Abbildung 12.6 ist der zeitliche Verlauf einer beispielhaften Iteration dargestellt.

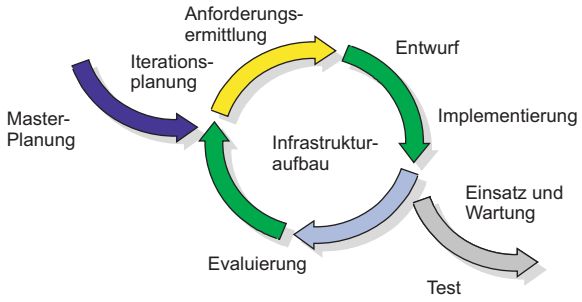


Abbildung 12.6 Typische Iteration im Unified Software Development Process

Ähnlich wie im V-Modell wird auch im Unified Process die Qualitätssicherung (test) im Prinzip über die gesamte Laufzeit des Projekts hinweg betrieben, wobei der Fokus der qualitätssichernden Aktivitäten auf der Konstruktionsphase liegt.
Zu den Hauptzielen des Tests im Unified Process zählen:

- die Interaktionen zwischen Objekten,
- die ordentliche Integration aller Software-Komponenten,
- die fehlerfreie Implementation aller Anforderungen und
- das frühe Auffinden von Fehlern *vor* der Auslieferung der Software und damit insbesondere
- die Verschiebung der Hauptrisiken eines Projekts in die frühen Phasen (vgl. auch Abbildung 12.2).

Dazu erläutert der Unified Process:

> *"Well-performed tests, initiated early in the software lifecycle, will significantly lower the cost of completing and maintaining the software. It will also greatly reduce the risks or liabilities associated with deploying poor quality software, such as poor user productivity, data entry and calculation errors, and unacceptable functional behavior. Nowadays, many MIS system are 'mission-critical', that is, companies cannot fulfill their functions and experience massive losses when failures occur. For example: banks, or transportation companies. Mission-critical systems must be tested using the same rigorous approaches used for safety-critical systems." [JBR99]*

Der Unified Process unterscheidet innerhalb der Aktivität „Testen“ die in der untenstehenden Abbildung 12.7 dargestellten Teilaktivitäten

- Test planen,
- Test entwerfen,
- Test implementieren,
- Test ausführen und
- Test auswerten.

Diese Inhalte und Ergebnisse dieser Aktivitäten decken sich im Wesentlichen mit den im vorliegenden Buch vorgestellten Testphasen.

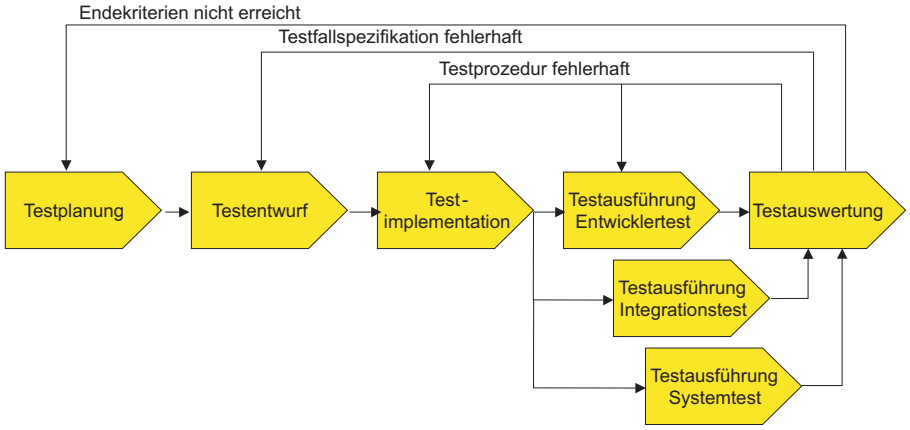


Abbildung 12.7 Test-Aktivitäten im Unified Process (nach [JBR99])

Auch die Teststufen Klassentest (unit test), Integrationstest (integration test) und Systemtest (system test) des Unified Process entsprechen denen im vorliegenden Buch, wobei der Unified Process selbstverständlich auch die Teststufe Abnahmetest (acceptance test) beinhaltet, die im vorliegenden Buch wegen des mangelnden Einflusses der Objektorientierung nicht behandelt wurde.

Zur Testauswertung definiert der Unified Process die folgenden Metriken:

- Überdeckungsmetriken (Anforderungs- und spezifikationsbasierte Kriterien, Codebasierte Kriterien);
- Qualitätsmaße (Defekt Reporte, Defektdichte, Defekttrends);
- Performanzmaße (Dynamisches Profiling, Antwortzeitverhalten/Durchsatz, prozentuale Anteile einzelner Teile, vergleichende Reports, Traces und Profile).

Zur Konkretisierung des Vorgehensmodells beinhaltet der Unified Process detaillierte Handlungsanweisungen (guidelines), in denen für ausgewählte Aktivitäten auf der Grundlage der zu erstellenden und der in vorgelagerten Aktivitäten erstellten

(Eingangs-)Dokumente und Ergebnisse (artifacts) beispielhaft das Vorgehen dargestellt wird.
Es ist festzuhalten, dass der Unified Process gerade in Bezug auf die UML und die Objektorientierung wesentlich konkreter als das V-Modell 97 ist. Dies ist jedoch nicht weiter verwunderlich, da Letzteres eine allgemeine Vorgehensweise (relativ) unabhängig von konkreten Techniken und Notationen definiert, wohingegen der Unified Process speziell für die objektorientierte Systementwicklung mit der in der UML standardisierten Notation angelegt ist. Für die Praxis empfehlen wir, das (umfassendere) V-Modell mit konkreten Techniken und Produktbeschreibungen des Unified Process anzureichern und – zusammen mit Vorlagen für die organisationsintern definierten Produkte – allen Beteiligten in einer Art „Wissens-Datenbank“ im Intranet verfügbar zu machen.

12.1.4 Xtreme Programming

Kent Beck führte bereits mehrere erfolgreiche Projekte mit seiner Entwicklungsmethodik *Xtreme Programming* durch [Bec99a]. Hierbei arbeiten kleinere Teams von ca. 10–20 Entwicklern in einem relativ schlanken Entwicklungsprozess. Jeweils zwei Entwickler sitzen gemeinsam an einem Arbeitsplatz (pair programming), wobei einer der beiden den Rechner bedient und die Eingaben vornimmt. Der andere verfolgt und vollzieht die Aktionen des ersten nach und prüft sie damit direkt bei ihrer Durchführung. Die drei Säulen der Methode sind Einfachheit, Kommunikation und direktes Feedback, was in eine gewisse Aggressivität bei der Softwareentwicklung mündet:

- Mit dem Grundsatz der *Einfachheit* ist nicht nur gemeint, dass jeweils das einfachste Konstrukt zur Lösung einer Entwicklungsaufgabe verwendet wird, sondern auch der einfachste Prozess mit dem wenigsten administrativen Overhead.
- Die effektive *Kommunikation* ist der Schlüssel sowohl für die produktive Entwicklung als auch für die Entwicklung eines Produkts, welches den Kunden zufrieden stellt. Hierfür sitzen alle Entwickler in einem Raum und repräsentative Benutzer sowie Fachexperten sind eng in die Entwicklung eingebunden. Statt mit formalen Dokumenten erfolgt die Kommunikation soweit wie möglich im direkten Gespräch von Person zu Person.
- Für das sofortige *Feedback* bei der Entwicklung dienen sowohl die paarweise Programmierung als auch die immer wieder durchzuführenden und zu pflegenden Tests.

Im Mittelpunkt der Methode steht die Spezifikation der Funktionalität mittels vor der Implementierung zu erstellender Testfälle, die sofort in einem Testorganisations- und Ausführungswerkzeug formuliert werden (vorzugsweise unter Benutzung

der verwendeten Programmiersprache als Skriptsprache). Als allgemeine Vorgehensweise hierbei schlägt Jeffries vor [JAH00]:

1. Erzeuge einen Rahmen der neuen Klasse, der die Köpfe aller Operationen inkl. der Parameter und Rückgabewerte sowie die öffentlich sichtbaren Instanzvariablen enthält.
2. Erzeuge eine Testklasse für die neue Klasse und bezeichne sie mit dem Namen der zu testenden Klasse mit vorangestelltem Test. Heißt die neue Klasse also z.B. Konto, so nenne die Testklasse TestKonto.
3. Implementiere die Konstruktor-Operation der zu testenden Klasse sowie entsprechende Tests des Konstruktors in der Testklasse.
4. Implementiere und teste alle weiteren Operationen der zu testenden Klasse.

Während und nach der Realisierung der Klassen und Komponenten werden die Testfälle immer wieder ausgeführt und so die Konformität der Implementation zur Spezifikation geprüft. Eine Komponente darf nur dann freigegeben werden, wenn die Testfälle ohne Fehleraufdeckung durchlaufen. Es gilt die Maxime, dass neu erstellter oder geänderter Code erst dann in das projektweite Repository eingestellt wird, wenn alle Tests erfolgreich bestanden werden. Eine typische Iteration in Xtreme Programming ist in Abbildung 12.8 skizziert. Man beachte die vor der Implementierung durchgeführte Testfallermittlung (und Testprozedurerstellung, d.H. Testfallcodierung im Test-Rahmenwerk).

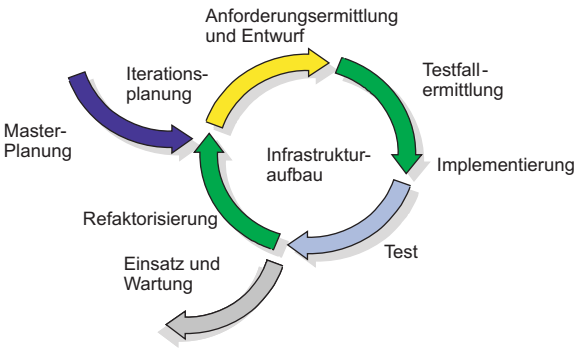


Abbildung 12.8 Typische Iteration in Xtreme Programming

Parrish et Al. konkretisieren die obige Vorgehensweise für den Test und die Implementierung in XP [PJD01]:

1. Entwerfe die neuen Testfälle.
2. Für jeden neu entworfenen Testfall
3. Implementiere den Testfall (im Testwerkzeug);

4. Führe den Testfall aus und prüfe, ob er fehlschlägt (die vom Testfall aufgerufenen Operationen sind ja noch nicht implementiert – läuft der Testfall jetzt schon fehlerfrei, so führt er entweder lediglich schon vorhandenen Code aus, oder aber die Ergebnisprüfung ist fehlerhaft);
5. Implementiere die fehlenden Operationen;
6. Führe den neuen Testfall aus und debugge die fehlerhaften Operationen, bis der Testfall keinen Fehler mehr aufdeckt;
7. Führe alle Testfälle aus und debugge die fehlerhaften Operationen, bis die Testfälle keinen Fehler mehr aufdecken.

Hierbei ist es vorteilhaft, die neuen Testfälle in einer solchen Reihenfolge abzuarbeiten, dass pro Testfall nur möglichst wenig neue Operationen implementiert werden müssen. In diesem Fall ist der Aufwand minimal, die Ursache für einen vom Test aufgedeckten Fehler im Code zu suchen und zu korrigieren.

Da die Bestimmung der optimalen Reihenfolge NP-hart ist, verwenden Parrish et Al. den folgenden einfachen Algorithmus. Die Testfälle werden zunächst im Prinzip proportional zur Anzahl der in ihnen verwendeten Operationen gewichtet und dann in aufsteigender Reihenfolge sortiert [PJD01]. Danach werden die Operationen der Klasse in einer solchen Reihenfolge implementiert und getestet, dass mit jeder neu implementierten Operation möglichst viele Testfälle „ausführbar“ werden.

Wesentlich für die Praktikabilität des Xtreme Programming ist, dass die Ausführung aller Testfälle nicht zu viel Zeit beansprucht. Hierzu weist Jeffries in [JAH00] darauf hin, dass insbesondere solche Testfälle, die den Zugriff auf Datenbanken erfordern, die Performanz der gesamten Testsuite unter die erforderliche Grenze senken können. Als Abhilfe in solchen Fällen empfiehlt Jeffries, möglichst viele Objekte lokal z.B. serialisiert in Dateien auszulagern und bei Bedarf von dort einzulesen oder sogar während des gesamten Testlaufs im Speicher zu halten. Nur einige wenige Testfälle sollten tatsächlich auf die Datenbank zugreifen und die entsprechenden Funktionalitäten absichern.

12.1.5 Fallstudie: Pilotprojekt CEE bei Ericsson-Kanada

Jean Boisvert berichtet über den Einsatz objektorientierter Testverfahren bei der Entwicklung eines Organisations- und Dokumentationssystems für Basisstationen im zellulären Mobilfunk (CEE)[Boi97]. In diesem Projekt waren ca. 15 Mitarbeiter beschäftigt, zu denen neun Entwickler und zwei Testspezialisten zählten. Bei anfänglichen Versuchen, den bei Ericsson installierten Standard-Testprozess auszuführen, wurde schnell klar, dass die strikte Trennung von strukturellen Tests (white-box) und funtionalen Tests (black-box) aufgrund der Testbarkeitseigenschaften objektorientierter Software zu erheblichen Problemen führte. Als Ausweg wurde ein gemischtes Testverfahren entwickelt (grey-box test), bei dem die Entwickler

und Tester jeweils einen Cluster funktional testen und dabei mittels Instrumentierung die Codeüberdeckung messen (Abbildung 12.9). Hierbei wurden Cluster bottom-up integriert und gleichzeitig – die Entwicklung erfolgte in C++ – das Speicherverhalten gemessen (Memory leak detection).

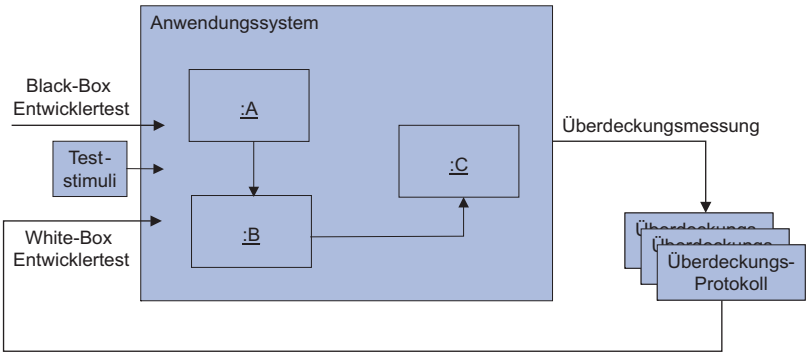


Abbildung 12.9 Grey-Box Test

Durch die inkrementelle Vorgehensweise konnten getestete Klassen zum Testaufbau für neu zu integrierende Klassen bzw. als Testorakel für den Regressionstest nach Änderungen herangezogen werden. Zu den wichtigsten Erfahrungen aus diesem Projekt zählt Boisvert:

- Die frühe Testplanung und Integrationsstrategie müssen nach der Konsolidierung der Anwendungsfälle überarbeitet werden.
- Der grey-box Test führt zusammen mit Reviews zu einer hohen Qualität des Programmcodes.
- Die automatische Testausführung ergab kürzere Regressionszyklen, wobei das Hauptproblem die Sollwert-Ermittlung darstellt.
- Der Einsatz von Testwerkzeugen ist nur mit geschulten Werkzeugbenutzern erfolgreich.
- Die Qualitätssicherung muss während der Entwicklung erfolgen, nicht nachher.

Mit nur 7 größeren Fehlern in der Produktion bei einem Umfang von 75 KLOC bzw. 1364 Function Points – also lediglich 0.0051 Fehlern/FP – stellte sich das Pilotprojekt als äußerst erfolgreich heraus.

12.2 Stand kommerzieller OO-Testwerkzeuge

Den hinter dem Stand der Forschung herhinkenden Stand der industriellen Praxis des Testens objektorientierter Software erkennt man auch sehr gut daran, inwieweit es kommerzielle Werkzeugunterstützung in diesem Bereich gibt. Der Arbeitskreis

„Testen objektorientierter Programme“ (TOOP) der GI-Fachgruppe 2.1.7 „Test, Analyse und Verifikation von Software“ (TAV) hat sich zum Ziel gesetzt, Anwendern Unterstützung bei der Auswahl bzw. Vorselektion von geeigneten existierenden OO-Testwerkzeugen anzubieten und gleichzeitig Hersteller zu einer stärkeren Unterstützung des objektorientierten Paradigmas durch ihre Testwerkzeuge anzuregen.

Ende des letzten Jahrtausends führte der Arbeitskreis TOOP hierzu eine Umfrage bezüglich des Stands der Testwerkzeugunterstützung für objektorientierte Programme durch [JuWi99]. Basierend auf den für den Test relevanten Eigenschaften objektorientierter Programme wurden zunächst Anforderungen an Testwerkzeuge identifiziert. Diese Anforderungen wurden dann bzgl. der Testphasen kategorisiert und in Form eines (html-) Fragebogens abgefasst. Der Fragebogen wurde im WWW veröffentlicht und an Werkzeughersteller gesendet, die mit ihren Antworten darstellen konnten, welche der ausgeschriebenen Anforderungen ihre Produkte erfüllen. Die wesentlichen Kriterien des Fragebogens sind im Anhang zusammengestellt und können als Checkliste für die Werkzeugauswahl dienen.

Bis Ende 1998 hatten lediglich fünf Hersteller den Fragebogen beantwortet und so die Erfüllung objektorientierter Anforderungen durch ihre Testwerkzeuge selbst bewertet. Aufgrund von Zeitschriftenartikeln und kommerziellen Marktübersichten wurde geschätzt, dass zur damaligen Zeit etwa 15 bis 20 Hersteller Testwerkzeuge mit speziellen Funktionen für den Test objektorientierter Programme anbieten, sodass diese Rückläufer – vorsichtig geschätzt – knapp 30% der verfügbaren Werkzeuge darstellen.

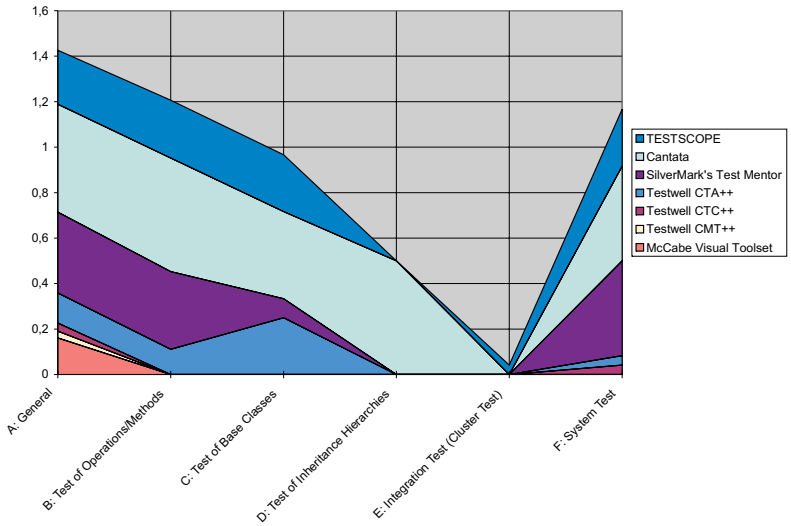


Abbildung 12.10 Unterstützung der Teststufen durch kommerzielle Werkzeuge

Zur Auswertung der Antworten wurden die Anforderungen nach der Unterstützung der unterschiedlichen Testphasen sowie nach den objektorientierten Eigenschaften der Testgegenstände gegliedert. Als grobe Näherung wurde z.B. für jede Teststufe der Quotient aus der Anzahl erfüllter Anforderungen und der Gesamtanzahl der Anforderungen gebildet. Eine diesbezügliche zusammenfassende Darstellung für die fünf Testwerkzeuge zeigt Abbildung 12.10.

Obwohl diese Auswertung vorläufig und sicherlich abhängig von der Anzahl der Anforderungen pro Testphase und ihrer Art war, drängte sich folgende Bewertung auf: Die Werkzeugunterstützung für das Testen objektorientierter Programme nimmt vom Unittest hin zum Integrationstest tendenziell stetig ab, um dann zum Systemtest hin wieder steil anzusteigen. Dies erklärt sich dadurch, dass der Unittest (als Test einzelner Operationen) noch die größte Ähnlichkeit zum Test in konventionellen Systemen besitzt (wie auch der Systemtest, der gleichfalls besser unterstützt ist).

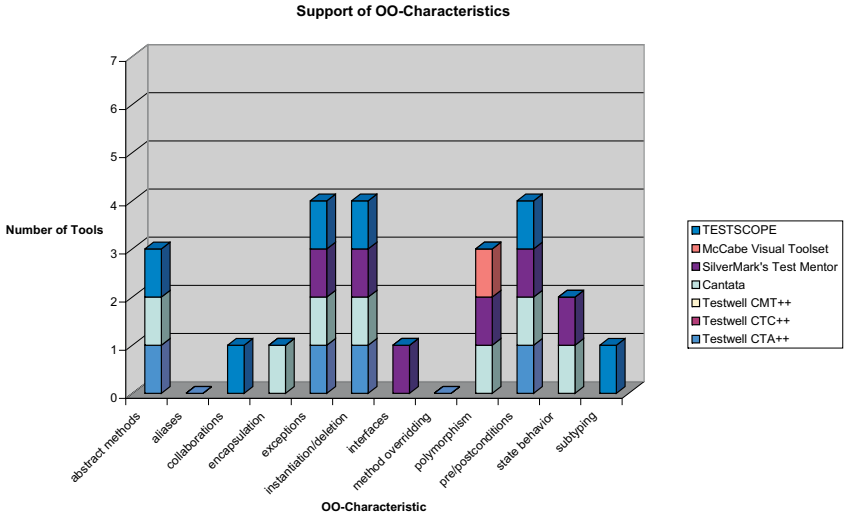


Abbildung 12.11 Unterstützung der Werkzeug- Anforderungen

Abbildung 12.11 zeigt, ob das Testwerkzeug zumindest eine der mit einer bestimmten objektorientierten Eigenschaft verknüpften Anforderungen erfüllt. Diese zweite Auswertung ist nun objektiver, da keine Abhängigkeit zur Anzahl der Anforderungen bzw. Fragen besteht und auch die Art der Frage keinen wesentlichen Einfluss ausübt. Auffällig sind die schwache Unterstützung von Alias-Betrachtungen, Kollaborationen, Methodenredefinition/Subtypbildung und Zustandsverhalten. Zusammenfassend halten wir fest, dass zur damaligen Zeit nur der Klassentest und der Black-Box-Systemtest ausreichende Werkzeugunterstützung erfahren haben und

somit in der industriellen Praxis durchführbar gewesen sein sollten. In Kung et Al.'s Worten:

> *"Strictly speaking, OO testing and maintenance tools have not been seen in the commercial market: most CASE tool vendors are advocating the use of conventional testing tools to cope with OO testing problems." [KGH+95].*

Entsprechend diesem Hintergrund der Werkzeugproblematik stellt sich der „state of the art" leider – mit wenigen Ausnahmen – immer noch dar [Bin99]. Kommerzielle Testwerkzeuge unterstützen leider auch heute nur wenige der vorgestellten Methoden und Techniken. So verwundert es nicht, dass in fast 70% aller Organisationen vornehmlich die Entwickler testen (Klassentest), und das meistens manuell und relativ unsystematisch. Es gibt keine vorgeschriebenen Testeingangs- und Testendekriterien, und niedrige Qualität wird i.Allg. zugunsten hoher Produktivität (in LOC) toleriert. Zur Verbesserung sollten die Entwickler wenigstens eine Basisschulung in der Testfallerstellung erhalten und eine minimale Werkzeugunterstützung (dynamischer Analysator, Testtreiber-Generator) installiert werden.

Die weiter oben bereits genannte Studie des BMBF stellt fest, dass sowohl die Entwicklung als auch die Qualitätssicherung derzeit durch eine Vielzahl von Werkzeugen unterstützt werden [BMBF00]. Bei dieser Flut von Werkzeugen sei es allerdings auffällig, dass sich die Softwareentwicklungs-Umgebungen aus verschiedenen Einzellösungen unterschiedlicher Anbieter zusammensetzen. Die einzige erkennbare Gemeinsamkeit der verschiedenen in der Studie untersuchten Unternehmen ist die steigende Bedeutung von Sprachen und Werkzeugen zur Unterstützung objektorientierter Methoden.

Folgende Probleme wurden in Bezug auf die Unterstützung durch Werkzeuge häufig genannt [BMBF00]:

- Die fehlende Durchgängigkeit oder schlechte Integration der verschiedenen Werkzeuge führt zu einem erheblichen Zusatzaufwand (d.h. Transformation von Dokumentationen) durch manuelle Schritte oder ergänzende Eigenentwicklungen.
- Die Werkzeuge erfordern einen hohen Trainingsaufwand, bis sie effizient eingesetzt werden können. Wird ein Softwaresystem über lange Zeit gewartet, muss die Fähigkeit zur Bedienung der entsprechenden Entwicklungswerkzeuge lange im Unternehmen aufrecht erhalten werden. Als Beispiel nennt die Studie die Programmierfähigkeiten in Cobol, deren Verlust vielen Unternehmen bei den Umstellungen im Zusammenhang mit dem Jahr-2000-Problem unangenehm bewusst geworden ist.
- Einige wenige große Unternehmen beklagen die mangelnde Werkzeugunterstützung für große Entwicklungsteams.
- Es gibt derzeit kaum Werkzeugunterstützung für verteilte Entwicklungsstandorte.

- Die Werkzeugunterstützung zur Verwaltung von Anforderungsänderungen wird von einigen Unternehmen als mangelhaft bezeichnet.
- Das Fehlen von guten Werkzeugen zur Testautomatisierung wird beklagt. Insbesondere werden Testwerkzeuge und Simulationsumgebungen gefordert, welche die Integration der Software im Gesamtsystem möglichst gut abbilden. Diese Forderung ist für Client/Server-Systeme wichtig, die ja meistens in heterogenen Umgebungen eingesetzt werden. Insbesondere die Qualitätssicherung von Softwareprodukten in heterogenen Umgebungen gewinnt durch die steigende Nutzung des Internets zunehmend an Bedeutung.

12.3 Zum Abschluss: Die Grundsätze des Testens

Um den Fokus wieder von den Prozess- und Werkzeugbetrachtungen zurück auf die methodischen Grundsätze des Testens zu lenken, bildet die folgende Liste der "Top Ten"-Grundsätze des Testens den Abschluss des Buches. Diese Liste ist sicher unvollständig, und die Befolgung aller Punkte garantiert noch kein erfolgreiches (Test-)Projekt. Die Erfahrung zeigt jedoch, dass die Nicht-Beachtung dieser Grundsätze schon viele Projekte gefährdet bzw. zum Scheitern verurteilt hat.

1. Testprozess installieren und unabhängige Testgruppe einsetzen.
2. Testmetriken und Testendekriterien für jede Teststufe definieren und einsetzen.
3. Testbare Anforderungen spezifizieren.
4. OO-Entwurf mit Verträgen (Zusicherungen) einsetzen.
5. Programmierrichtlinien einsetzen und kontrollieren.
6. Testgetriebene Programmierung verankern.
7. Eingebaute Debugging-Hilfen aller Werkzeuge frühzeitig verwenden.
8. Testfall-Repository mit fachlichen und technischen Testfällen bilden.
9. Testware wie jede andere Software behandeln (testware is software).
10. Konfigurations- und Versionsmanagement für Software und Testware installieren und verwenden.

Anhang

Checklisten für die Auswahl von Testtechniken

Begriffsvergleich: V-Modell vs. IEEE 829

Checkliste für die Werkzeugauswahl

Inhaltsübersicht des Anhangs

13 Anhang

13.1 Checklisten für die Auswahl von Testtechniken

Die nachfolgenden Checklisten (Tabellen 1 bis 3) geben erste Hilfestellungen zur Auswahl von Testtechniken für objektorientierte Software. Die Tabellen 1 bis 3 sind alle nach demselben Muster aufgebaut; Tabelle 4 fasst die Teilergebnisse der Tabellen 1 bis 3 zusammen.

Zur Auswahl geeigneter Testverfahren tragen Sie in diesen Tabellen in der Spalte Häufigkeit die statistisch ermittelte oder geschätzte Häufigkeit

1= selten

2 = manchmal

3 = oft

der Fehler ein. Graue Felder bedeuten dabei, dass im Feld kein Eintrag erlaubt ist, da die entsprechende Kategorie nicht zutrifft. Multiplizieren Sie diesen Faktor mit dem (ggf. für Ihr Projekt angepassten) Kritikalitätsfaktoren in der Spalte Kritikalität:

1= niedrig

2 = mittel

3 = hoch

Den sich ergebenden Wert, das Zeilengewicht, tragen Sie nun in alle freien Felder der Reihe ein. Addieren Sie danach die Werte in jeder Spalte und übertragen die Teilergebnisse der Tabellen 1 bis 3 in Tabelle 4. Summieren Sie abschließend in Tabelle 4 noch einmal die Werte in jeder Spalte auf – und Sie haben das Eignungsprofil für den Test Ihres Projekts ermittelt.

Abschließend noch ein kleiner Hinweis: Haben Sie in der Summe der Zeilengewichtungen einen Wert größer als 50 ermittelt, sollten Sie einen erhöhten Aufwand für die Qualitätssicherung Ihrer Anwendung planen.

13.1.1 Technikauswahl: Operationenfehler

Tabelle 13-1: Operationenfehler

Testtechnik vs. Fehlerhypothese	Häufigkeit	Kritikalität	Zeilengewichtung	Zusicherungen	Ausnahmen	Funktionaler Test	Struktureller Test	Zustandstest	Integrationstest	Systemtest	Akzeptanztest	Reviews	Statische Analyse	Memory Checker
Interner Logikfehler		3			■				■	■	■		■	■
Zusicherungsfehler		3			■		■			■	■		■	■
Performanzfehler		1		■	■		■	■	■				■	
Unerreichbarer Code		1		■	■			■	■	■	■			■
Summe Operationen-fehler	■	■	■											

13.1.2 Technikauswahl: Objekt/Klassenfehler

Tabelle 13-2: Objekt/Klassenfehler

Testtechnik vs. Fehlerhypothese	Häufigkeit	Kritikalität	Zeilengewichtung	Zusicherungen	Ausnahmen	Funktionaler Test	Struktureller Test	Zustandstest	Integrationstest	Systemtest	Akzeptanztest	Reviews	Statische Analyse	Memory Checker
Spezifikation verletzt		3			■		■				■		■	■
Dokumentationsfehler		2		■	■	■	■	■	■	■	■			■
Persistenzfehler		2		■	■		■	■	■		■		■	■
Zustandsfehler		3					■			■	■		■	■
Vererbungsfehler		3			■					■	■			■
Speicherfehler		2		■		■	■	■			■			
Summe Objekt/Klassenfe	■	■	■											

13.1.3 Technikauswahl: Nutzungsbasierte Fehler

Tabelle 13.3: Nutzungsbasierte Fehler

Testtechnik vs. Fehler-hypothese	Häufigkeit	Kritikalität	Zeilengewichtung	Zusicherungen	Ausnahmen	Funktionaler Test	Struktureller Test	Zustandstest	Integrationstest	Systemtest	Akzeptanztest	Reviews	Statische Analyse	Memory Checker
Nachrich-tenziel		3												
Nachricht		3												
Rückgabe-objekt		3												
Ausnahme-behandlungs-		2												
Ausnahme		2												
Nebenläu-figkeits-fehler		2												
Summe														

13.1.4 Auswertung

Tabelle 13-4: Auswertung der Technikauswahl

Zwischenresultate	Zusicherungen	Ausnahmen	Funktionaler Test	Struktureller Test	Zustandstest	Integrationstest	Systemtest	Akzeptanztest	Reviews	Statische Analyse	Memory Checker
Operationen-fehler											
Objekt/Klassen-fehler											
Nutzungsbasierte Fehler											
Gesamt Summe											

13.2 Begriffsvergleich: IEEE 829 vs. V-Modell 97

In diesem Abschnitt stellen wir die in diesem Buch nach dem IEEE-Standard 829 verwendeten Begriffe den im V-Modell 97 verwendeten gegenüber. Dies erleichtert Anwendern des V-Modells den Zugang zu den relevanten Kapiteln des vorliegenden Buches und die Übertragung des in diesem Buch vorgestellten Testprozesses auf das V-Modell.

Buch / (IEEE 829)	**V-Modell 97**	**Erläuterung im V-Modell 97**
Entwicklertest (developer test)	Selbstprüfung	Prüfung des Arbeitsergebnisses durch den Ausführenden selbst gemäß festgelegter Regeln.
Test (test)	Prüfung	Eine Tätigkeit wie Messen, Untersuchen, Ausmessen von einem oder mehreren Merkmalen einer Einheit sowie Vergleichen mit festgelegten Forderungen, um festzustellen, ob Konformität für jedes Merkmal erzielt ist.
Testabschlussbericht (test summary report)	Projektabschluss-bericht	Die Qualitätssicherung (QS) im Projekt ist häufig eingebettet in ein QM-System. Um dem Anspruch der stetigen Verbesserung (DIN ISO 9000 ff.) gerecht zu werden, müssen Erfahrungen über Projektgrenzen hinweg bewahrt und ausgewertet werden. Insbesondere die Erfahrungen mit QS-Maßnahmen sind zusammenzufassen. Dazu eignet sich der Projektabschlussbericht Der Projektabschlussbericht ist eine besondere Ausprägung des Sachstandsberichtes, der eine Gesamtschau über den Projektverlauf bietet und gemeinsam mit dem entwickelten Produkt übergeben wird. Es sollte außerdem ein Hinweis enthalten sein, wie mit den Projektergebnissen und -produkten (im Hinblick auf nachfolgende SWPÄ-Projekte) weiter verfahren wird.
Testendekriterium (test criteria, pass/fail criteria)	Prüfkriterium Endekriterium	Unter diesem Gliederungspunkt der →Prüfspezifikation werden die Kriterien jeder Prüfung genannt. Sie sind derart festzulegen, dass die Prüfung hinsichtlich ihrer erfolgreichen Durchführung bewertbar ist. Endekriterien benennen Bedingungen, unter denen die Prüfung als erfolgreich abgeschlossen betrachtet werden kann. In diesem Gliederungspunkt der →Prüfspezifikation werden sowohl Endekriterien einer erfolgreichen Prüfung (z. B. die geforderte Genauigkeit ist mit einer maximalen Abweichung von +/- 0.0005 erfüllt) als auch einer nicht bestandenen Prüfung (z. B. Meldung "Überlauf", "Division durch Null", "Speichermedium voll") genannt.
Testfall (test case)	Prüffall	Prüffälle der einzelnen →Prüfgegenstände sind auf der Basis der in der →Prüfspezifikation definierten Methoden und der Prüfanforderun-

		gen derart auszuwählen, dass durch sie die Erfüllung der Qualitätsforderungen ausreichend nachgewiesen werden kann. (Jeder Prüffall beschreibt, was (Funktion, Genauigkeit, usw.) zu prüfen ist, welche Ausgangssituation hierfür erforderlich ist, welche Eingaben (Daten und Signale mit allen für die Prüfung ausschlaggebenden Eigenschaften wie Zeitbedingungen) notwendig sind und welche Ergebnisse (Ausgabedaten und Reaktionen/Effekte) zu erwarten sind. Mit den Prüffällen müssen die →Endekriterien ausreichend erfüllbar und entscheidbar sein.
Testkonzept (test design)	Prüfplan Prüfspezifikation	Der Prüfplan definiert die Prüfgegenstände und die Aufgaben und Verantwortlichkeiten bei den Prüfungen, die zeitliche Planung sowie die für die Durchführung erforderlichen Ressourcen. Im Prüfplan ist festgelegt, welche Produkte und Aktivitäten in welchem Zustand wann, von wem und womit zu prüfen sind. Die Prüfspezifikation enthält die Beschreibung von Prüfanforderungen und -zielen, Prüfmethoden, von den Anforderungen abgeleitete Prüfkriterien und die Prüffälle. Die Abdeckung der Anforderungen durch die Prüffälle wird durch eine Abdeckungsmatrix dokumentiert. Mit Hilfe der Prüfspezifikation muss entschieden werden können, ob die Prüfung erfolgreich war oder nicht.
Testlog (test log) Testdeckungsbericht (test coverage report) Testvorfallsbericht (test incidence report) Testergebnisbericht (test report)	Prüfprotokoll	Das Prüfprotokoll enthält die vom Prüfer verfassten Aufzeichnungen über den Verlauf der →Prüfung, vor allem die Gegenüberstellung von erwartetem und erzieltem Ergebnis. Diese Aussagen werden im Rahmen der Auswertung um Analysen über mögliche Ursachen des Fehlverhaltens und um vorgeschlagene Maßnahmen ergänzt. Das Prüfprotokoll existiert je Prüfgegenstand und je Prüfung.
Testobjekt test item	Prüfgegenstand	Zu den Prüfgegenständen zählen: Anwenderforderungen, technische Anforderungen, Systemarchitektur, SW-Architektur, HW-Architektur, Schnittstellenbeschreibung, Integrationsplan, Betriebsinformationen, System, SW-Entwurf, Datenkatalog, Implementierungsdokumente, SW-Komponente, SW-Modul, Datenbank, Realisierungsdokumente. Des Weiteren kann es aufgrund strenger Qualifikationserfordernisse erforderlich sein, auch Prüfungen für folgende Prüfgegenstände festzulegen: SWPÄ-Konzept, KM-Plan, Projektplan.
Testplan (test plan)	QS-Plan	Der QS-Plan enthält die für ein Projekt gültigen generellen Festlegungen bezüglich der Prävention und der Nachweisführung (Arbeitsweisen, Hilfsmittel, Abläufe). Geplant werden konstruktive Maßnahmen zur Erreichung der Qualitätsziele, präventive Maßnahmen zur Vermeidung von Qualitätsrisiken und analytische Maßnahmen zum Nachweis der Erfüllung von Qualitäts-

		forderungen.
Testprozedur (test procedure)	Prüfprozedur	Die Prüfprozedur ist eine Arbeitsanleitung, die exakte Anweisungen für jede einzelne Prüfung enthält. Hier sind die einzelnen Schritte der Prüfung definiert. Ebenfalls festgelegt sind die erwarteten Prüfergebnisse sowie Vorschriften zur Prüfungsvor- und -nachbereitung.
Testtechnik (test technique)	Prüfmethode	Methoden zur Durchführung der Prüfung sind z. B. statische Analyse, Test, Simulation, Korrektheitsbeweis, symbolische Programmausführung, Review, Inspektion. Die Methoden der Prüfungsdurchführung werden anhand der kritikalitäts- und IT-sicherheitsbezogenen Einstufung des Prüflings, der den jeweiligen Stufen zugeordneten Maßnahmen und weiteren an ihn gestellten Qualitätsforderungen ermittelt. Festzuschreiben sind die Art und Weise der Ergebnissicherung und -auswertung, insbesondere im Hinblick auf Wiederholung von Prüfungen. Es wird geklärt, welche Daten während und nach der Prüfung wie festzuhalten sind.Die Methoden und Vorgehensweisen werden hier festgelegt und beschrieben, z. B. die Nutzung von automatisierten Vergleichsroutinen, die persönliche Begutachtung, das Führen eines chronologischen Logbuchs.
	Prüfanforderung	Prüfanforderungen sind Anforderungen allgemeiner Art an eine Prüfung, z.B. Prüfungen sind mit Normal-, Grenz- und fehlerhaften Werten durchzuführen, Prüfungen sind unter Normal- und Ausnahmebedingungen (Höchstleistungen, Komponentenausfall usw.) durchzuführen, Prüfungen sind mit Echtdaten durchzuführen, möglichst alle Ausführungsoptionen und fehlerhafte Nutzereingaben sind abzuprüfen.
Testumgebung (test environment)	Prüfumgebung	Die zur →Prüfung notwendige Prüfumgebung ist mit ihren HW-Bausteinen (Geräte, Analysatoren, Monitore, Testrigs, usw.) und SW-Bausteinen (Systemsoftware, Firmware, Simulatoren, Testdatengeneratoren, Testtreiber, Stubs, usw.) eindeutig zu identifizieren; andernfalls sind die Anforderungen an diese Umgebung zu definieren.
	Ressourcen	Die für die einzelnen Prüfungen außerdem benötigten Ressourcen (IT-Betriebsmittel, betriebliche Infrastruktur, Arbeitsmittel) sind zu nennen mit Detailangaben, wann und in welchem Umfang sie verfügbar sein müssen.

13.3 Checkliste für die Werkzeugauswahl

In Hinblick auf die objektorientierten Eigenschaften wurde vom Arbeitskreis „Testen objektorientierter Programme“ (TOOP) der GI-Fachgruppe 2.1.7 „Test, Analyse und Verifikation von Software“ (TAV) folgender Katalog von Anforderungen in Form einer Checkliste bzw. eines Fragebogens zusammengestellt und soll bei der Werkzeugauswahl sicherstellen, dass keine relevanten Punkte bezüglich der Objektorientierung übersehen werden [JuWi99].
Die Checkliste gliedert sich in die Teile

- Allgemeines,
- Test von Operationen/Methoden,
- Test von Basisklassen,
- Test von Vererbungshierarchien,
- Integrationstest (Cluster-Test) und
- Systemtest.

13.3.1 Allgemeines

Seit wann gibt es das Werkzeug auf dem Markt?
Wie viele Lizenzen wurden bisher verkauft?
Referenzkunde?
Plattformen?

- Windows95
- WindowsNT
- Unix (Solaris/HP-UX/Ultrix/...)
- MVS

13.3.2 Integration des Werkzeugs in die Entwicklungsumgebung

Kann das Werkzeug mit anderen Werkzeugen zusammenarbeiten?

- CASE-Werkzeuge
- Repositories
- Konfigurations-Management
- andere CAST Werkzeuge

Können Modellierungsinformationen zur Testfallermittlung übernommen werden?

- Multiplizitäten von Beziehungen
- Navigationsrichtung von Beziehungen
- mögliche Sortierung (ordered) von Beziehungen

- Sequenzdiagramme
- Kollaborationsdiagramme
- Zustandsautomaten
- Wie übernommen?

Werden Modellierungsinformationen zur Generierung von Testtreibern verwendet?

13.3.3 Sprachen und Umgebungen

Welche Programmiersprache(n) unterstützt das Werkzeug?

- Java
- C++
- C#
- Smalltalk
- Eiffel
- Mehrsprachige Programme (Smalltalk/C bzw. C++ Binding)

Unterstützte Sprachbesonderheiten?

13.3.4 Statische Analysen

Werden OO-Metriken ermittelt?

- Chid
- Chidamber/Kemerer
- Andere:

Ist eine Gruppierung der Methoden möglich?

- nach Variablenverwendung
- nach Konstruktor-, Modifier- und Observer-Methoden?

Welche sprachabhängigen Checks werden ausgeführt?

- Lint-like
- Eigene Programmierrichtlinien ([Baldwin92])

Welche semantischen Analysen werden ausgeführt?

- Prüfung der Subtyp-Eigenschaft von abgeleiteten Klassen
- Alias-Analysen
- Dynamische Analysen

Auf welcher Granularität werden Überdeckungsanalysatoren angeboten?

- Methode
- Botschaft/Aufruf
- Dto., mit Polymorphismus

Ist überprüfbar, welche Objekte wirklich erzeugt und zerstört werden?

4. Wird die Auslösung und Behandlung von Ausnahmen über Klassen hinweg geprüft?

13.3.5 Aufzeichnung von Testfällen und Testdaten

Ist es möglich, Testfälle mit einer Skriptsprache aufzuzeichnen? Wenn ja, ist diese Skriptsprache

- Editierbar,
- Werkzeugspezifisch,
- erweiterung einer Programmiersprache oder
- objekt-orientiert?

Auf welcher Granularität können Skripts strukturiert/aufgezeichnet werden?

- auf Methodenebene
- auf Klassenebene
- auf Subsystem/Paket-Ebene (Integrationsebene)
- auf Systemebene (z.B. GUI-Skript)

Ist es möglich, Testskripte zu parametrisieren?

Ist es möglich, Testdaten zu speichern bzw. zu lesen?

- aus externen sequenziellen Dateien
- aus internen Datenbanken / Datenbank-Tabellen
- aus externen Datenbanken / Datenbank-Tabellen (z.B. MS-ACCESS)

Gibt es eine initiale Datenbank mit Testdaten für Grunddatentypen?

13.3.6 Testumgebung

Wird eine gesonderte Testablaufumgebung (Testbed) angeboten?

- Generierung von Stubs
- Generierung von Treibern
- auf der Methodenebene
- auf der Klassenebene
- auf Subsystem/Paket-Ebene
- auf Systemebene (z.B. GUI-Skript)

Wie werden Prozeduren und Testfallketten gestartet?

- Sequenziell
- Asynchron
- synchron

13.3.7 Regressionsfähigkeit

Die selektive Testfallauswahl für Regressionstests basiert auf:

- manueller, risikogesteuerter Auswahl

- automatischer Auswahl nach Implementierungsstatus von Methoden bzgl. der Vererbungshierarchie (unverändert, redefiniert, neu)
- automatischer Auswahl auf Basis von Zustandsverfeinerungen

Wird dabei das Konfigurations-Management berücksichtigt?

Wird dabei das ModellierungsWerkzeug berücksichtigt? Wenn ja, in welchem Umfang werden Modellmodifikationen berücksichtigt?

- semantische Änderungen (Vor- u./oder Nachbedingungen)
- Änderungen der Implementierung
- Änderungen der Schnittstelle

13.3.8 Besonderheiten

Werden Entwurfsmuster (design pattern, Bsp. Fassadenpattern) berücksichtigt? Wenn ja, wie?

Werden Test-Entwurfsmuster mitgeliefert?

Können gezielt Klassen aus Bibliotheken bzw. Frameworks in den Test übernommen oder vom Test ausgeschlossen werden?

Welche technisch interessanten Features hat das Werkzeug zusätzlich zu den objektorientierten?

13.3.9 Test von Operationen/Methoden

Werden Black-box-Techniken/Testfallgeneratoren angeboten? Wenn ja, welche?

- Formale Spezifikation
- VDM
- Object-Z
- Andere

Erfolgt die Definition und Kontrolle von Vor- und Nachbedingungen sowie Zwischenzusicherungen für Methoden [Sne95]?

- Im Code
- Im Testfall/Werkzeug

Welche White-box-Techniken/Testdatengeneratoren werden angeboten?

- Kontrollflussbezogene
- Dto., mit Botschaften
- Dto., mit Botschaften und Polymorphismus
- Daten/Objektflussbezogene
- Dto., mit Polymorphismus

13.3.9.1 Testausführung und Auswertung

Welche Prüfungen/Manipulationen sind zur Laufzeit möglich?

- Abfragen von Parameter- und Rückgabeobjekten

- Gezieltes Austesten von Pfaden
- Prüfen von Ausnahmen (Exceptions)

13.3.10 Test von Basisklassen

13.3.10.1 Kapselung

Durchbricht das Werkzeug die Kapselung? Wenn ja, wie?

- durch generierte Selektor- bzw. Beobachter-Operationen (Operationen zum Lesen von Instanzvariablen im Rahmen des Tests)
- durch Generierung von „Friendklassen“ (C++)
- durch Introspektion / Reflexion (Java, Smalltalk-80)

13.3.10.2 Zustand

Werden Zustandsautomaten benutzt?

- Aus Code generiert
- Aus Modellierungs/Casewerkzeug übernommen
- Im Werkzeug definiert

Werden Vor/Nachbedingungen/Klasseninvarianten benutzt?

- Im Code
- Im Testfall/Werkzeug

13.3.10.3 Testfallgenerierung

Unterstützt das Werkzeug den Test von Basisklassen mit virtuellen Methoden?
Unterstützt das Werkzeug

- Vor/Nachbedingungen/Klasseninvarianten
- Zustände
- Zustandsübergänge (durch Aufruf der Konstruktoren und einer beliebigen Mischung der Methoden, basierend auf Spezifikationen, z.B. Pre und Postconditions)?

13.3.10.4 Testausführung und Auswertung

Welche Prüfungen/Manipulationen sind zur Laufzeit möglich?

- Direkte Manipulation von Instanzvariablen
- Indirekte Manipulation von Instanzvariablen über public Methoden (setup-Botschaftssequenzen)
- Direkte Manipulation von Klassenvariablen (static fields)
- Indirekte Manipulation von Klassenvariablen über `public`-Methoden (setup-Botschaftssequenzen)
- Instrumentierte Kopie, alles public

13.3.11 Test von Vererbungs-Hierarchien

13.3.11.1 Vererbung

Liefert das Werkzeug Informationen, inwieweit abgeleitete Klassen im Rahmen des Basisklassentests bereits getestet wurden? Wenn ja,

- ist die sich daraus ergebende Differenz zur Erstellung neuer Testfälle geeignet?

Berücksichtigt das Werkzeug die Modifikationen in Unterklassen zur Testfallauswahl?

13.3.11.2 Polymorphismus

Können polymorphe Aufrufe gezielt ausgeführt werden? Wenn ja, durch welchen Mechanismus?

13.3.12 Integrationstest (Cluster-Test)

13.3.12.1 Aggregation und Assoziation

Welche Informationen werden zur Testfallermittlung verwendet?

- Multiplizitätsbeschränkungen
- Navigationsrichtungen
- Sortierungsbedingungen
- Sequenzdiagramme
- Kollaborationsdiagramme

Wie sind diese Informationen einzugeben?

- Manuell
- Modellsprache (Werkzeugspezifisch / STL / XML ...)

Wird das korrekte Löschen innerhalb einer Aggregation geprüft?
Werden Invarianten oder Constraints innerhalb von Aggregationen/Assoziationen geprüft? (z.B. `Rechnung.betrag=Summe(Position.betrag*Position.anzahl)`
Unterscheidet das Werkzeug Aggregation und Assoziation? Wenn ja, wie?

13.3.13 Systemtest

13.3.13.1 Generierung von Testfällen

Ist es möglich, aus einem Benutzungsmodell Testfälle abzuleiten? Wenn ja, aus welchem Modell?

- Anwendungsfälle
- Aktivitätsdiagramme
- Zustandsautomaten

13.3.13.2 Testausführung

Ist es möglich, Datenbank-Schnittstellen bzw. Anbindungen zu testen?

- SQL
- ODBC
- JDBC

Ist es möglich, den Zugriff auf andere Systemkomponenten oder Ressourcen, welche über Internetprotokolle erfolgen, zu testen? Wenn ja, welche:
Welche Performanz- und Speicherplatz-Informationen werden gesammelt?

- Execution time profiling
- Memory Leaks
- Konstruktor/Destruktor-Anomalien

14 Literatur

[ABC82] Adrian, W., Branstadt, M., Cherniavsky, J.: “Validation, Verification and Testing of Computer Software”, in ACM Computing Survey, Vol. 14, Nr. 2, 1982, S. 159, 220

[acm99] ACM/SIGSOFT: “Risks to the Public by Computer Systems”, Software Engineering Notes, Vol. 24, Nr. 4, Juli 1999

[ArFu94] Arnold, T.; Fuson, W.: “Testing in a Perfect World”, Comm. of ACM, Vol.37, Nr. 9, Sept. 1994, S. 78

[BaGo99] Bashir,I., Goel,A: Testing Object-Oriented Software – Life Cycle Solutions, Springer Verlag, Ney York, 1999, S. 13, 128–144

[BaSe87] Basili, V.; Selby, R.: “Comparing the Effectiveness of Software Testing Strategies”, IEEE Trans. on S.E., Vol. 13, Nr. 12, Dez. 1987, S. 1278

[BBF+82] Berg, H., Boebert, W., Franta, W., Moher, T.: Formal Methods of Program Verification and Specification, Prentice-Hall, Englewood Cliffs, 1982, S. 38

[BCD89] Benedusi, P., Cimitile, A., DeCarlina, U.: “Postmaintenance Testing based on Path Change Analysis”, in Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, Phoenix, 1989, S. 352

[BDL+78] Budd, T., DeMillo, R., Lipton, R., Sayward, F.: “Design of a Prototype Mutation system for Program Testing”, in Proc. of AFIPS, Vol. 47, 1978, S. 623

[Bec94] Beck, K.: “Simple SmallTalk Testing”, Smalltalk Report, Vol. 4, Nr. 2, Oct. 1994, S. 16

[Bec99] Beck, K.: “Embracing Change with extreme Programming”, IEEE Computer, Oct. 1999, p. 70

[Bec99a] Beck, Kent: Extreme Programming Explained – Embrace Change, Addison Wesley, 1999

[Bei83] Beizer, B.: Software Testing Techniques, van Nostrand Reinhold, New York, 1983, S. 89

[Bei84] Beizer, B.: Software System Testing and Quality assurance, van Nostrand Reinhold, New York, 1984, S. 79, 165

[Bei90] Beizer, B.: Software Testing Techniques, van nostrand Reinhold, New York, 1990, S. 33

[Bei91] Beizer, Boris: “On Becoming the Mainstream”, American Programmer, April 1991, S. 11–21

[Bei94] Beizer, B.: “Testing Technology – The Growing Gap”, American Programmer, Vol. 7, Nr. 4, April 1994, S. 3, 9

[Bei95] Beizer, B.: Black-Box Testing, John Wiley & Sons, New York, 1995, S. 38, 86, 146

[Bei99] Beizer, B.: “Best and Worse Testing Practices”, Cutter IT Journal, Vol. 12, Nr. 2, Feb. 1999, S. 32

[Bin93] Binder, R.: “Test Case Design for object-oriented Programs: the FREE Approach”, RBSC Corporation Technical Report, Chicago, June. 1993

[Bin94] Binder,R.: “Testing object-oriented systems – A Status Report”, American Programmer, Vol. 7, Nr. 4, April 1994, S. 23

[Bin94a] Binder, R.: “Design for Testability in object-oriented Systems”, Comm. of ACM, Vol. 37, Nr. 9, Sept. 1994, S. 28

[Bin94b] Binder, R.: “Testing object-oriented Programs”, Tech. report 94-003, Binder systems Consulting, Chicago, 1994

[Bin94c] Binder, R.: “The Free Approach to testing object-oriented Systems”, Tech. report 94-004, Binder systems Consulting, Chicago, 1994

[Bin95] Binder, Robert V.: “The FREE- Fow Graph: Implementation-based Testing of Objects Using State-determinining Flows” in Proc, 8th International Quality Week, 1995

[Bin96] Binder, R.: “Testing object-oriented Software – A Survey”, Software Testing, Verification & Reliability, Vol. 6, Nr. 3/4, Sept. 1996, S. 195

[Bin96a] Binder, R.: “Modal Testing Strategies for object-oriented Software”, IEEE Computer, Vol. 29, Nr. 11, Nov. 1996, S. 97

[Bin96b] Binder, R.: “Testing object-oriented Software”, Software Testing, Verification & Reliability, Vol. 6, Nr. 3/4, Sept. 1996, S. 180

[Bin99] Binder, R.: Testing object-oriented Systems, Addison-Wesley, Reading, Mass., 1999, S. 69, 212, 269, 499, 627

[Bin99a] Binder, Robert V.: “Testing Object-Oriented Systems: Lessons Learned“, In Proc. 2nd Quality Week Europe, Bruxelles, Belgium, 1999

[BMBF00] „Analyse und Evaluation der Softwareentwicklung in Deutschland“, Studie für das Bundesministerium für Bildung und Forschung, Projektgemeinschaft: GfK Marktforschung GmbH, Fraunhofer-Institut für Experimentelles Software Engineering IESE, Fraunhofer-Institut für Systemtechnik und Innovationsforschung ISI, Dez. 2000

[BMS79] Budd, T., Majoros,M., Sneed,H.: “Experiences with a Software Test Factory” in Proc. First COMPCON Workshop on Software Testing, IEEE Computer Society Press, Fort Lauderdale, Dez. 1979, S. 319

[Boi97] Boisvert, Jean: “OO Testing in the Ericsson Pilot Project”, Object Magazine, Juli 1997

[Bol95] Boloix, G.: “A Software System Evaluation Framework”, IEEE Computer, Dez. 1995, S. 17

[Boo94] Booch, Grady: Object-Oriented Analysis and Design, Addison-Wesley, Reading, Mass., 1994

[Bor99] Borelli, N.: “Seizing Control of the Development Life cycle”, in Proc. of European Quality Week-99, Software Research, Brüssels, Nov. 1999, S. 310

[Bou97] Bourne, K.: Testing Client/Server Systems, McGraw-Hil, New York, 1997, S. 4, 353

[BrDr93] Bröhl, A., Dröschel, W.: Das V-Modell – Der europäische Standard für die Softwareentwicklung, Oldenbourg Verlag, München, 1993

[Bur97] Burkhardt, R.: UML – Unified Modeling Language, Addison-Wesley Verlag, Bonn, 1997

[Che66] Cherry, C.: On Human Communication, M.I.T. Press, Cambridge, Mass., 1966, S. 32

[ChMe90] Cheatham, T.; Mellinger, L.: “Testing object-oriented Software Systems”, in Proc. of 18th Computer Science Conference, ACM Press, New York, Feb. 1990

[Chu97] Chuso, T.: “Test Data Selection and Quality Estimation based on the Concept of Essential Branches for Path Testing”, IEEE Trans. on SE, Vol. 13, Nr. 5, Mai 1987, S. 509

[CoMi90] Cobb, R.; Mills, H.: “Engineering Software under Statistical Quality Control”, IEEE Software, Nov. 1990, S. 44

[CoNo91] Cox, B.; Novobilski, A.: Object-Oriented Programming – An Evolutionary Approach, Addison-Wesley, Reading, Mass., 1991, S.32

[Cox88] Cox, B.: “The need for specification and testing Languages”, Journal of OO-Programming, Vol. 1, Nr. 2, 1988, S.44

[CPR+89] Clarke, L., Podgurski, A., Richardson, D., Zeil, S.: “A Formal Evaluation of Data Flow Path Selection Criteria”, in: IEEE Trans on SE, Vol. 15, Nr. 11, Nov. 1989, S. 1318

[CTC+98] Chen, H.Y.; Tse, T.; Chan, F.; Chen, T.: “An integrated Approach to class-level Testing of object-oriented Programs”, ACM Trans. on S.E. and Methodology, Vol. 7, Nr. 3, July 1998, S. 250

[CuSe96] Cusumano, M.A.; Selby, R.W.: Die Microsoft Methode, Heyne, München, 1996

[Dav90] Davis, A.M.: Software Requirements – Analysis and Specification, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[DeOf91] DeMillo, R.; Offutt, A.: “Constraint based Automatic Test Data Generation”, IEEE Trans. on SE, Vol. 17, Nr. 9, Sept. 1991, S. 900

[Des94] Desfray, P.: Object Engineering – the Fourth Dimension, Addison-Wesley, reading, Mass., 1994

[Deu82] Deutsch, M.: Software Verification and Validation, Prentice-Hall Computer Series, Englewood Cliffs, 1982, S. 161

[DGQ12-51] DGQ-NTG: Software-Qualitätssicherung, Deutsche Gesellschaft für Qualität, Schrift 12-51, Frankfurt, 1986

[DIN66285] DIN: Anwendungssoftware – Prüfgrundsätze, Deutsches Normen Institut, Norm Nr. 66285, Beuth Verlag, Frankfurt, 1985

[Dlu94] Dlugosz,J.: “The Dark Side of OOP”, American Programmer, Vol. 7, Nr. 10, Okt. 1994, S. 12

[DoFr91] Doong, R.-K.; Frankl, P. G.: “Case Studies on Testing Object-Oriented Programs”, in Proc. 4th Symposium on Testing, Analysis, and Verification (TAV4), 1991, S. 165-177

[DoFr94] Doong, R.; Frankl, P.: “The ASTOOT Approach to testing object-oriented Programs”, ACM Trans. on SE and Methodology, Vol. 3, Nr. 4, April 1994, S. 101

[Dro95] Dromey, R.: “A Model for Software Product Quality”, IEEE Trans. on SE, Vol. 21, Nr. 2, Feb. 1995, S. 146

[DsLe94] D´Souza, R.; LeBlanc, R.: “Class Testing by examining Pointers”, Journal of OO-Programming, Vol. 7, Nr. 4, 1994, S. 33

[EcLe85] Eckhardt, J., Lee, R.: “A theoretical Basis for the Analysis of multiversion Software subject to coincident Errors”, IEEE Trans. on SE, Vol. 11, Nr. 12, 1985, S. 812

[Eli94] Eliot,L.: “Critical Success Factors for implementing Client/Server Applications” American Programmer, Vol. 7, Nr. 11, Nov. 1994, S. 10

[Eva84] Evans, M.: Productive Software Test Management, John Wiley & Sons, New York, 1984, S. 27

[Fag76] Fagan, M.: “Design and Code Inspections to reduce Errors in Program Development“, IBM Systems Journal, Vol. 15, Nr. 3, 1976, S. 31

[FeGr99] Fewster, M.; Graham, D.: Software Test Automation, Addison-Wesley, Harlow, G.B., 1999, S. 26

[FHL+98] Frankl, P., Hamlet, R., Littlewood, B., Strigini, L.: “Evaluating Testing Methods by delivered Reliability”, IEEE Trans. on SE, Vol. 24, Nr. 8, Aug. 1998, S. 586

[Fie89] Fiedler, S.: “Object-oriented Unit Testing”, Hewlett-Packard Journal, Vol. 40, Nr. 2, 1989, S. 69

[Fir93] Firesmith, D.: Object-oriented Requirements Analysis and Logical Design, John Wiley & Sons, New York, 1993, S. 248

[Fir96] Firesmith, D. “Pattern Language for Testing object-oriented Software”, Object Magazine, SIG Pub., London, Vol. 5, Nr. 8, Jan. 1996, S. 32

[Fis77] Fischer, K.: “A Test Case Selection Method for the Validation of Software Maintenance Modifications”, in Proc. of COMPSAC, IEEE Computer Society Press, Chicago, 1977, S. 421

[Fow98] Fowler, M.: UML Konzentriert, Addison-Wesley Verlag, Bonn, 1998

[FrWe93] Frankl, P.G.; Weyuker, E.: “A formal Analysis of the Fault-detecting Ability of Testing Methods”, IEEE Trans. on SE, Vol. 19, Nr. 3, March 1993, S. 202

[Fuj77] Fujii, M.: “Independent Verification of highly reliable Programs”, in Proc. of COMPSAC, IEEE Computer Society Press, Chicago, 1977, S. 38

[Gam99] Gamma, E.: “JUnit – A Testing Framework for Java”, in Proc. of OOP-99, SIG Conferences, Munich, Jan. 1999

[GDT93] Graham, J. A.; Drakeford, A. C. T.; Turner, C. D: “The Verification, Validation, and Testing of Object-Oriented Systems”, BT Technology Journal, vol. 11, p. 10, Juli 1993

[Gla81] Glass, R.: “Persistent Software Errors”, IEEE Trans. on S.E., Vol. 7, Nr. 2, März 1981, S. 162

[GmVo80] Gmeiner, L., Voges, U.: “SADAT – An Automated Test Tool”, IEEE Trans on SE, Vol. 6, Nr. 3, 1980, S. 286

[Goe85] Goel, A.: “Software Reliability Models – Assumptions, Limitations and Applicability”, IEEE Trans. on SE, Vol. 11, Nr. 12, Dez. 1985, S. 1411

[GoFi94] Gotel, O.C.Z., Finkelstein, A.C.W.: ”An Analysis of the Requirement Traceability Problem”, Proc. RE’94, Colorado Springs, IEEE Press, S. 94-101

[Gog93] Goglia,P.: Testing Client/Server Applications, QED Publishing Group, Wellesley, Mass., 1993, S. 15

[GoGe75] Goodenough, J., Gerhardt, S.: “Towards a Theory of Test Data Selection”, IEEE Trans. on SE, Vol. 1, Nr. 1, 1975, S. 156

[Gra96] Graham, D.: Testing object-oriented Systems, Ovum Ltd, London, Feb. 1996, S. 9

[Gri94] Grimm, Klaus: Systematisches Testen von Software — Eine neue Methode und eine effektive Teststrategie. Dissertation, TU Berlin, FB 13, Softwaretechnik, 1994

[GrWe93] Grochtmann, M., Wegner, J.: “Werkzeugunterstützende Testfallermittlung für den funktionalen Test mit dem Klassifikationsbaumeditor CTE”, Softwaretechnik-Trends, 13(3):95, Aug. 1993, S. 103

[Hal94] Hall, C.: Technical Foundations of Client/Server Systems, John Wiley & Sons, New York, 1994, S. 57

[HaMü83] Hausen, H-L., Müllerburg, M.: “An Introduction to Quality Assurance and Control of Software”, in Proc. of GMD Symposium on Software Validation, Darmstadt, North-Holland, 1983, S. 3

[HaRo90] Hartmann, J., Robson, D.: “Techniques for Selective Revalidation”, IEEE Software, Jan. 1990, S. 31

[HaRo94] Harrold, M.J., Rothermel, G.: “Performing Data Flow Testing on Classes”, Proc. of 2nd ACM SigSoft Symposium, ACM Press, New York, 1994, S. 154

[Hat98] Hatton, L.: “Does OO really sync with how we think”, IEEE Software, Mai 1998, S. 46

[HaWi93] Halladay, S., Wiebel, M.: Object-oriented Software Engineering, R&D Publications, Lawrence, KA., 1993, S. 189

[HaWi94] Halladay,S., Wiebel,M.: Object-Oriented Software Engineering – Object Verification, R&D Publications, Lawrence, KA., 1994, S. 189

[Hay86] Hayes, I.: “Specification directed Module Testing”, IEEE Trans. on SE, Vol. 12, Nr. 1, Jan. 1986, S. 124

[Het73] Hetzel, W.: Program Test Methods, Prentice-Hall, Englewood Cliffs, N.J., 1973

[HKK+94] Haase, V., Koch, G., Kugler, H., Decrinis, P.: “Bootstrap – Fine Tuning Process Assessment”, IEEE Software, July 1994, S. 25

[HLK+97] Hsia, P.; Li, X.; Kung, D.; Hsu, C-T.: “Techniques for Selective Revalidation of object-oriented Software”, Journal of Software Maint., Vol. 9, Nr. 4, Juli 1997, S. 217

[HMF92] Harrold, M.J.; McGregor, J.; Fitzpatrik, K.: “Incremental Testing of object-oriented Class Structures” in Proc. of 14th Int. Conf. on S.E., IEEE Computer Society Press, Melbourne, May 1992, S. 68

[Hof95] Hoffman, D.: “Classbench: A Framework for Class Testing“, In Proc. 8th International Quality Week, 1995

[Hof99] Hoffman, D.: “Heuristic Test Oracles”, Software Testing & Quality Engineering, Vol. 1, Nr. 2, März 1999, S. 28

[HoSt93] Hoffman, D.; Strooper, P.: “Graph-based Class Testing”, in Proc. of 7th Australian Software Eng. Conf,, Sidney, Oct. 1993, S. 85

[HoSt94] Hoffman, D.; Strooper, P.: “Graph-based Class Testing”“, The Australian Computer Journal, Nov. 1994

[HoSt97] Hoffmann, D.; Strooper, P.: “A Framework for automated Class Testing”, Software Practice & Experience, May 1997, S. 573

[How76] Howden, W.: “Reliability of the Path Analysis Testing Strategy”, IEEE Trans on SE, Vol. 2, Nr. 3, 1976, S. 208

[How78] Howden, W.: “Theoretical and Empirical Studies of Program Testing”, IEEE Trans. on SE, Vol. 4, Nr. 4, 1978, S. 293

[How80] Howden, W.: “Functional Program Testing”, IEEE Trans on SE, Vol. 6, Nr. 2, 1980, S. 162, 293

[How81] Howden,W.: “Completeness Criteria for testing elementary Program Functions”, in: Proc. of 5th Int. Conf. on SE, IEEE Computer Society Press, Los Angelos, 1981, S. 235

[How85] Howden,W.: “The Theory and Practice of Functional testing”, IEEE Software, Vol. 2, Nr. 5, Sept. 1985, S. 6

[How86] Howden, W.: “A functional Approach to Program testing and Analysis”, IEEE Trans. on S.E., Vol. 12, Nr. 10, Oct. 1986, S. 997

[How87] Howden, W.: Functional Program testing, McGraw-Hill, New York, 1987, S. 129

[Hum95] Humphrey, W. A Discipline of Software Engineering, Addison-Weslöey-Longman, reading, Mass., 1995, S. 48

[Hun95a] Hunt, N.: “Unit testing”, Journal of Object-Oriented Programming, Feb. 1995, S. 18-23

[Hun95b] Hunt, N.: “C++ boundary conditions and edge cases”, Journal of Object-Oriented Programming, May 1995

[Hun95c] Hunt, N.: “Automatically tracking test case execution”, Journal of Object-Oriented Programming, Nov. 1995, S. 22–27

[IEEE1008] IEEE: Standard for Software Unit Testing, ANSI/IEEE Standard 1008-1987, IEEE Computer Society Press, New York, 1987

[IEEE1063] IEEE: ANSI/IEEE Standard 1063 – Guide for Software User Documentation, IEEE Press, New York, 1993

[IEEE610] IEEE: ANSI/IEEE Standard 610 – Glossary of Software Engineering Terminology, IEEE Press, New York, 1990

[IEEE729] IEEE: Glossary of Software Engineering Terminology, ANSI/IEEE Standard 729-1983, IEEE Computer Society, New York, 1983

[IEEE829] IEEE: Standard for Software Test Documentation, ANSI/IEEE Standard 829-1998, IEEE Computer Society Press, New York, 1998

[IEEE830] IEEE: ANSI/IEEE Standard 830 – Guide for Software Requirement Specification, IEEE Press, New York, 1998

[ISO12119] ISO: ISO/IEC-Standard 12119 – Software Package Quality Requirements and Testing, Int. Standard Org., Genf, 1998

[ISO12207] ISO: ISO/IEC Standard 12207 – Software Life Cycle Processes, Int. Standards Organization, Genf, 1998

[ISO9126] ISO: ISO/IEC-9126 Software Product Evaluation – Quality Characteristics and Guidelines for their use, International Standards Organization, Genf, 1994

[Jac95] Jacobson, I.; Christersson, M., Jonsson, P., Övergaard, G.: “The Use-Case Construct in object-oriented Software Engineering” in Scenario-based Design, Ed. J. Carroll, John Wiley & Sons, New York, 1995

[JAH00] Jeffries, R.; Anderson, A.; Hendrickson, C.: Extreme Programming Installed, Addison Wesley, Reading, 2000

[Jal89] Jalote, P.: “Testing the Completeness of Specifications”, IEEE Trans. on SE, Vol. 15, Nr. 5, Mai 1989, S. 526

[JBR99] Jacobson, I.; Booch, G. ; Rumbaugh, J. : The Unified Software Development Process, Addison Wesley, Reading, 1999

[JKN+94] Jüttner, P.; Kolb, S.; Naumann, U.; Zimmer, P.: “Experiences in Testing object-oriented Software”, in Proc. of Int. Conf. on Testing Computer Software, USPDI, Washington, D.C., June 1994

[JKZ94] Jüttner, P.; Kolb, S.; Zimmerer, P.: “Integrating and Testing of object-.oriented Software” in Proc. of EuroStar-94, SQE, Brüssels, Okt. 1994, S. 13

[JoEr94] Jorgensen, P.; Erickson, C.: “Object-oriented Integration Testing”, Comm. of ACM, Vol. 37, Nr. 9, 1994, S. 30

[Jon97] Jones, C.: “The Economics of object-oriented Software”, Software Productivity Research, Burlington, Mass., April 1997

[JuWi99] Jungmayr, S.; Winter, M.: „OO-Testwerkzeuge: Wunsch und Wirklichkeit“, in Proc. GI Working Group Test, Analysis and Verification of Software, München, 4.-5. Feb. 1999. In Softwaretechnik-Trends, Vol. 19, Nr. 1, 1999

[Jun99] Jungmayr, S.: “Reviewing Software Artifacts for Testability”, in: Proc. Euro-STAR’99, Barcelona, Spanien, 8.-12. Nov, 1999

[KGH+93] Kung, D.; Gao, J.; Hsia, P.; Lin, J.: “Design Recovery for Software Testing of object-oriented Programs” in Proc. of 1st Working Conf. on Reverse Eng., IEEE Computer Society Press, Baltimore, May, 1993, S. 201

[KGH+94] Kung D. C.; Gao J.; Hsia P.; Suchak N.; Toyoshima, Y; Chen, C: “On Object State Testing”, in Proc. 18th Annual International Computer Software & Applications Conference (COMPSAC '94), 1994, S. 222-227

[KGH+95] Kung, D. C., Gao, J., Hsia, P., Chen, C., Kim, Y.-S., Toyoshima, Y.: “Developing an object-oriented Software Testing and Maintenance Environment”, Comm. of ACM, Vol. 38, Nr. 10, Okt. 1995, S. 75

[KGH93] Kung, D., Gao, J., Hsia, P.: “Design Recovery for Software Testing of object-oriented Programs”, in Proc. of 1st Working Conf. on Reverse Eng., IEEE Press, Baltimore, May 1993, S. 202

[Koe95] Koelmel, R.: Implementing Application Solutions in a Client/Server Environment, John Wiley & Sons, New York, 1995, S. 77

[KoPo99] Koemen, T., Pol, M.: Test Process Improvement, Addison-Wesley, Harlow, G.B., 1999, S. 14

[KPS00] Koemen, T., Pol, M., Spillner, A: Management und Optimierung des Testprozesses, dpunkt.verlag, Heidelberg, 2000

[KSW01] Kösters, Six, H.-W., Winter, M.: “Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications”, Requirements Engineering, Vol. 6, Nr. 1, Springer, London, 2001

[LaKo83] Laski, J., Korel, B.: A Data Flow oriented Program Testing Strategy”, IEEE Trans on SE, Vol. 9, Nr. 3, 1983, S. 347

[LaNa97] Lange, D.; Nakamura, Y.: “Object-oriented Program Tracing and Visualization”, IEEE Software, Mai 1997, S. 63

[LeBe85] Lehman, M.; Belady, L.: Program Evolution – Process of Software Change, Academic Press, London, 1985, S. 21

[LeWi89] Leung, H., White, L.: “Insights into Regression Testing”, in Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, Miami, 1989, S. 60

[LeWi90] Leung,H., White,L.: “Insights into Testing and Regression Testing Global Variables”, Journal of Software Maint., Vol. 2, Nr. 4, Dec 1990, S. 209

[LFC94] Lee, J., Feng, M., Chung, C.: “A structural Testing Method for C++ Programs”, in: Proc. of COMPSAC Conference, IEEE Computer Society Press, Chicago, Nov. 1994, S. 234

[Lig90] Liggesmeyer, P.: Modultest und Modulverifikation, B.I. Wissenschaftsverlag, Mannheim,1990, S. 49

[Lig94] Liggesmeyer, P.: „Eine Methode zur Konstruktion von Prüfstrategien für Software“, Informatik Forschung und Entwicklung, Sept. 1994

[LiHe95] Li, W., Henry, S.: ”An Empirical Study of Maintenance Activities in two object-oriented Systems”, Journal of Software Maintenance, Vol. 7, Nr. 2, March 1995, S. 131

[LiRü96] Liggesmeyer,P., Rüppel,P.: „Die Prüfung von objektorientierter Systeme“, Objektspektrum, Nr. 6, 1996, S. 68

[LiSw80] Lientz, B., Swanson, B.: Software Maintenance Management, Addison-Wesley, Reading, Mass.,1980, S. 67

[LiXi93] Lieberherr, K., Xiao, C.: “Object-Oriented Software”, IEEE Trans. on Software Engineering, Vol. 19, Nr. 4, 1993, S. 313

[Lor93] Lorenz, M.: Object-oriented Software Development, Prentice-Hall, Englewood Cliffs, N.J., 1993, S. 18

[Lyu96] Lyu, M. R.: Software Reliability Engineering, IEEE Computer Society Press/McGraw Hill, New York, 1996

[Maq93] Maquire, S.: Writing Solid Code, Microsoft Press, 1993, S. 41

[Mar95] Marick, B.: The Craft of Software Testing (Subsystem Testing), Prentice Hall, 1995

[MaZh99] Mayrhauser, A.., Zhang, N.: “Automated Regression Testing using DBT and Sleuth”, Journal of Software Maint., Vol. 11, Nr. 2, März 1999, S. 93

[McC83] McCabe, T.: Structured Testing Tutorial, IEEE Computer Society Press, New York, 1983

[McG95] McGibbon, B.: Managing your Move to Object Technology, SIG Books, New York, 1995, S. 76

[Mey88] Meyer, B.: Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, N.J., 1988, S. 78

[Mey92] Meyer, B.: “Applying Design by Contract”, IEEE Computer, Vol. 25, Nr. 10, Oct. 1992, S. 40

[MgKo94] McGregor, J., Korson, T.: “Integrating object-oriented Testing and Development Processes”, Comm. of ACM, Vol. 37, Nr. 9, Sept. 1994, S. 59

[MgSy00] McGregor, J., Sykes, D.A.: A Practical Guide to Testing Object-Oriented Software, Addison Wesley, Reading, 2000

[MgSy92] McGregor, J., Sykes, D.: Object-Oriented Software Development – Engineering Software for Reuse, Int. Thomson Computer Press, London, 1992, S. 60

[Mic99] Michel, T.: XML Kompakt, Hanser Verlag, München, 1999, p. 52–90

[MiKo98] Miller, J., Kotopoulis, A.: „Überwachung und Diagnose verteilter CORBA-Anwendungen“, Objektspektrum, Nr. 6, 1998, S. 32

[Mil77] Miller, E.: “Program Testing – Art meets Theory”, IEEE Computer Magazine, Juli, 1977, S. 42

[Mil78] Miller, E.: “Program Testing Tools and their use”, in: Infotech State of the Art Report on Software Reliability, Maidenhead, G.B., 1978, p.183

[Mil79] Miller, E.: “State of the Art of Software Testing”, Infotech Ltd, Berkshire,G.B., 1979

[Mil81] Miller, E.: “Structural Techniques of Program Validation”, in: IEEE Tutorial on Software Testing & Validation Techniques, IEEE Computer Society Press, New York, 1981, S. 304

[MIO87] Musa, J.; Iannino, A.; Okumoto, K.: Software Reliability, McGraw-Hill, New York, 1987, S. 20, 72

[MoZa95] Mowbray, T.; Zahavi, R.: The essential CORBA, John Wiley & Sons, New York, 1995, S. 231

[MTW94] Murphy, G.; Townsend, P.; Wong, P.: “Experiences with Cluster and Class testing”, Comm. of ACM, Vol. 37, Nr. 9, Sept. 1994, S. 39

[MuAc89] Musa, J.; Ackerman, A.: “Quantifying Software Validation – When to stop Testing”, IEEE Software, Mai 1989, S. 19

[Mun88] Munos, C.: “An Approach to Software Product Testing”, IEEE Trans. on S.E., Vol. 14, Nr. 11, Nov. 1988, S. 1589

[Mun98] Munoz, C.: “An Approach to Software Product Testing”, IEEE Trans. on SE, Vol. 14, Nr. 11, Nov. 1998, S. 1589

[MüWi98] Müller, U.; Wiegmann, T.: „‘State of the practice’ der Prüf- und Testprozesse in der Softwareentwicklung – Ergebnisse einer empirischen Untersuchung bei Softwareunternehmen in Deutschland”, Technischer Bericht, Universität Köln, März 1998

[Mye76] Myers, G.: Software Reliability – Principles and Practices, John Wiley & Sons, New York, 1976, S. 229

[Mye79] Myers, G.: The Art of Software Testing, John Wiley & Sons, New York, 1979, S. 44, 56

[NaKu91] Nakajo, T.; Kume, H.: “A Case History of Software Error Cause-Effect Relationships”, IEEE Trans. on SE, Vol. 17, Nr. 8, Aug. 1991, S. 830

[NaSa92] Naik, K.; Sarikaya, B.: “Testing Communication Protocols”, IEEE Software, Jan. 1992, S. 27

[Nta84] Ntafos, S.: “On Required Element Testing”, IEEE Trans on SE, Vol. 10, Nr. 6, 1984, S. 795

[OMG99] Unified Modeling Language (UML) Specification, V. 1.3, Object Management Group, Juni 1999

[OrHa97] Orfali, R.; Harkey, D.: Client/Server Programming with Java and CORBA, John Wiley & Sons, New York, 1997, S. 73

[Ove94] Overbeck, J.: Integration testing for object-oriented Software, Dissertation, Tech Univ. Wien, 1994

[PaZw95] Parrish, A.; Zweben, S.: “On the Relationships among the all-uses, all-du-paths and all edges Testing Criteria”, IEEE Trans. on S.E., Vol. 21, Nr. 12, Dez. 1995, S. 1006

[PBC93] Parrish, A.; Borie, R.; Cordes, D.: “Automated Flow Graph-based Testing of object-oriented Software Modules”, Journal of Systems and Software, Vol. 23, Nr. 2, 1993, S. 95

[PBR90] Premerlani, W.; Blaha, M.; Rumbaugh,J.: “An object-oriented Relational Database”, Comm. of ACM, Vol. 33, Nr. 11, Nov. 1990, S. 99

[PeKa90] Perry, D., Kaiser,G.: “Adequate Testing and Object-Oriented Programing”, Journal of OO-Programming, Vol. 2, Nr. 5, Jan. 1990, S. 13

[PePa98] Peters, D.; Parnas, D.: “Using Test Oracles generated from Program Documentation”, IEEE Trans. on SE, Vol. 24, Nr. 3, März 1998, S. 161

[PJD01] Parrish, A.; Jones, J.; Dixon, B.: ”Extreme Unit Testing: Ordering Test Cases to Maximize Early Testing”, Proc. XPUniverse’01, Raleigh, NC, 2001

[PoBr87] Poston, R.; Bruen, M.: “Counting down to Zero Software Failures”, IEEE Software, Sept. 1987, S. 54

[PoBu94] Ponder, C., Bush,W.: “Polymorphism considered harmful”, ACM Software Eng. Notes, Vol. 18, Nr. 1, Jan. 1994, S. 35

[PoCl90] Podgurski, A.., Clarke, L.: “A formal Method of Program Dependencies and its implications for Software Testing, Debugging and Maintenance”, IEEE Trans. on SE, Vol. 16, Nr. 9, Sept. 1990, S. 965

[Pos94] Poston, R.: “Automated Testing from Object Models”, Comm. of ACM, Vol. 37, Nr. 9, Sept. 1994, S. 48

[Pos96] Poston, R.: Automating specification-based Software Testing, IEEE Computer Society Press, Los Almitos, Cal., 1996, S. 197

[Pos96a] Poston, R.: “Action or Function driven Test Case Design” in IEEE Tutorial – Automating specification-based Software Testing“, IEEE Computer Society Press, Los Alamitos, Cal., 1996, S. 47

[RaWe89] Rapps, S., Weyuker, E.: “Selecting Software Test Data using DataFlow Information”, IEEE Trans. on SE, Vol. 11, Nr. 4, April 1989, S. 367

[RBP+91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991

[ReWe85] Repps, S.; Weyuker, E.: “Selecting Software Test Data using Dataflow Information”, IEEE trans. on S.E., Vol. 11, Nr. 4, April 1985, S. 367

[RiCl85] Richardson, D.; Clarke, L.: “Partition Analysis – A Method combining Testing and Verification”, IEEE trans. on S.E., Vol. 14, Nr. 12, Dec. 1985, S. 1477

[Rin96] Rine, D.: “Structural Defects in object-oriented Programming”, ACM Software Eng. Notes, Vol. 21, Nr. 2, März 1996, S. 86

[RoHa94] Rothermel, G.; Harrold, M.J.: “Selecting Regression Tests for object-oriented Software”, in Proc. of Int. Conf. on Software Maint.-94, IEEE Press, Montreal, Okt. 1994, S. 14

[RoHa96] Rothermel, G.; Harrold, M.J.: “Analyzing Regression Test Selection Techniques”, IEEE Trans. on SE, Vol. 22, Nr. 8, Aug. 1996, S. 529

[RoHa98] Rothermel, G.; Harrold, M.J.: “Empirical Studies of a Safe Regression Test Selection Technique”, IEEE Trans. on S.E., Vol. 24, Nr. 6, June 1998, S. 401

[Ros95] Rosenblum, D.: “A practical Approach to Programming with Assertions”, IEEE Trans. on S.E., Vol. 21, Nr. 1, Jan. 1995, S. 19

[RoWe97] Rosenblum, D.; Weyuker, E.: “Using Coverage Information to predict the Cost Effectiveness of Regression Testing Strategies”, IEEE Trans. on SE, Vol. 23, Nr. 3, März 1997, S. 146

[Rüp96] Rüppel, Peter: Ein generisches Werkzeug für den objektorientierten Softwaretest, Dissertation, TU Berlin, 1996

[Rya97] Ryant,I.: “Why Inheritance means extra Trouble”, Comm. of ACM, Vol. 40, Nr. 10, Okt., 1997, S. 118

[SaEh86] Saglietti, F., Ehrenberger, W.: “Considerations on Software Diversity on the Basis of experimental and theoretical Work”, ESPRIT REQUEST Project Report Nr. 5, GRS, Garching, 1986

[SBM82] Schmitz,P., Bons,H., van Megen,R.: Software-Qualitätssicherung – Testen im Software Lebenszyklus, Vieweg Verlag, Braunschweig-Wiesbaden, 1982

[Sch73] Scher, A.: “Developing and Testing a Large Programming System”, in Hetzel, W. (ed), “Program Test Methods”, Prentice-Hall, Englewood Cliffs, 1973, S. 165

[Sch95] Schurr, E.: “Testing GUIs – The RAD/QA Approach for automated Testing of GUI Client/Server Applications”, American Programmer, Vol. 8, Nr. 4, April 1995, S. 21

[SeCh80] Sethi, L.; Chatterjee, B.: “Conversion of Decision tables to efficient sequential Testing Procedures”, Comm. of ACM, Vol. 23, Nr. 5, 1980, S. 279

[ShCa97] Sheppard,M., Cartwright,M.: “An empirical Study of object-oriented Metrics”, Tech. Report Nr. TR 97/01, Dept of Computing, Bournemouth Univ., U:K., 1997

[Sie92] Siegel, S.: “Strategies for Testing object-oriented Software”, CompuServe Case Forum, L.A., Sept. 1992

[Sie94] Siegel, S.: “OO Integration Testing Specification”, in Proc. of 7th Int. Quality Week, Software Research, San Francisco, May, 1994

[Sie96] Siegel,S.: Object-Oriented Software Testing – A Hierarchical Approach, John Wiley & Sons, New York, 1996, S. 93, 163, 175, 225

[SiNe94] Siepmann, A.; Newton, E. R.: “TOBAC: A Test Case Browser for Testing Object-Oriented Software”, in Proc. 1994 International Symposium on Software Testing and Analysis (ISSTA), 1994, S. 154–168

[SmRo90] Smith,M., Robson,D.: “Object-Oriented Programming – The Problems of Validation” in: Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, San Diego, 1990, S. 272

[SmRo92] Smith, M.; Robson, D.: “A Framework for Testing Object-Oriented Programs”, Journal of OO-Programming, Vol. 5, Nr. 3, 1992, S. 45

[Sne83] Sneed, H. Software-Qualitätssicherung für kommerzielle Anwendungssysteme, Rudolf Müller Verlag, Köln, 1983, S. 102

[Sne83a] Sneed, H.: “PRÜFSTAND: A System for Testing Software Components”, in: Proc. of 1st Int. Conf. on Software Maint., IEEE Press, Monterey, Cal., Dec. 1983, S. 281

[Sne86] Sneed, H.: “Data Coverage Measurement in Program Testing”, in Proc. of ACM/IEEE Workshop on Software Testing, Banff, Canada, IEEE Press, 1986, S. 34

[Sne88] Sneed, H.: Software-Qualitätssicherung, Rudolf-Müller Verlag, Köln, 1988, S. 256

[Sne90] Sneed, H.: Softwarewartung, Rudolf Müller Verlag, Köln, 1990, s. 63

[Sne92] Sneed, H.: “Regression Testing in Reengineering Projects” in Proc. 9th Int Conf. on Testing Computer Programs, U.S. Prof. Development Institute, Washington, D.C., June 1992, S. 161

[Sne92] Sneed,H.: “Regression Testing in Reengeering Projects”, in Proc. of STAR Conference on Testing Computer Software, Washington, D.C., June 1992, S. 161

[Sne94] Sneed, H.: “Validating Functional Equivalence of Reengineered Programs via Control Path, Result and Data Flow Comparison”, Journal of Software Testing, Verification & Reliability, Vol. 4, Nr. 1, 1994, S. 33

[Sne95] Sneed,H. „Objektorientiertes Testen”, Informatik Spektrum, Nr. 18, Jan. 1995, S. 6, 12ff.

[Sne96] Sneed,H.: “Ein objektorientiertes Testverfahren” in GMD Bericht Nr. 260 – Test, Analyse und Verifikation von Software, Oldenbourg Verlag, München, 1996, S. 25

[Sne98] Sneed, H. „Automated Test Case Specification for Integration Testing of Distributed Objects”, in Proc. of EuroStar-98, München, Dez. 1998, S. 137

[Sne98a] Sneed, H.: Objektorientierte Softwaremigration, Addison-Wesley, Bonn, 1998, S. 227

[Sne99] Sneed, H.: “Control Flow Animation as a means of Class Testing”, in Proc. of Quality Week, Europe, Software Research, Brüssels, Nov. 1999

[SnKi83] Sneed, H., Kirchhof, K.:“Prüfstand – A system for testing Software Components” in: Proc. of 1st Int. Conf. on Software Maint., Monterey, Dec. 1983, S. 281

[SnMa83] Sneed, H.; Majoros, M.: “Testing Programs against a formal Specification”, in Proc. of COMPSAC-93, IEEE Computer Society Press, Nov. 1983, S. 512

[SnRi93] Sneed, H.; Ritsch, H.: “Reverse Engineering Programs via Dynamic Analysis” in Proc. IEEE Working Conf. on Reverse Eng., IEEE Computer Society Press, Baltimore, 1993, S. 192

[Spi90] Spillner, A.: Dynamischer Integrationstest modularer Softwaresysteme, Dissertation, Univ. Bremen, 1990

[Spi00] Spillner, A.: “From V-Model to W-Model – Establishing the Whole Test Proces”, in Proc. CONQUEST 2000, ASQF, Nürnberg, Sep. 2000

[Sta96] Statz, J.: “Getting started with Software Risk Management”, American Programmer, Vol. 8, Nr. 3, March 1995, S. 23

[Str92] Stroustroup,B.: The Design and Evolution of C++, Addison-Wesley, Reading, Mass., 1992, S. 35

[Thu92] Thuy, N.: “Testability and Unit tests in large object-oriented Software”, Proc of 5th Int. Software Quality Week, Software Research, San Francisco, May, 1992

[Thu93] Thuy, N.: “Design for Quality in large object-oriented Software”, in Proc. of 6th Int. Quality Week, Software Research, San Francisco, May, 1993

[Tro95] Trower, T.: “User Interface Design for Windows-95”, American Programmer, Vol. 8, Nr. 4, April 1995, S. 8

[TuRo92] Turner, C. D.; Robson, D. J.: “A Suite of Tools for the State Based Testing of Object-Oriented Programs”, Tech. Rep. TR-14/93, School of Engineering and Computer Science, University of Durham, Durham, GB, Apr. 1992

[TuRo93] Turner,C., Robson,D.: “State-Based Testing of object-oriented Programs”, in: Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, Montreal, 1993, p. 143

[VM97] Entwicklungsstandard für IT-Systeme des Bundes – Vorgehensmodell, Allgemeiner Umdruck Nr. 250/1 (Regelungsteil), Juni 1997

[VoMi95] Voas, J. M.; Miller K. W: “Software Testability: The New Verification” IEEE Software, Jan. 1995, S. 28ff.

[WaKl99] Warmer, J.; Kleppe, A.: The Object Constraint Language – Precise Modeling with UML, Addison-Wesley, Reading, Mass., 1999

[Wei84] Weiser, M.: “Program Slicing”, IEEE Trans. on SE, Vol. 10, Nr. 4, Juli, 1984, S. 352

[Wey90] Weyuker, E.: “The Cost of Dataflow Testing – An Empirical Study”, IEEE Trans on S.E., Vol. 16, Nr. 2, Feb., 1990, S. 121

[Wey93]Weyuker, E.: “Experience with Data Flow Testing”, IEEE Trans on SE, Vol. 19, Nr. 9, Sept. 1993, S. 912

[Wey98] Weyuker, E.: “Testing component-based Software”, IEEE Software, Sept. 1998, S. 54

[WGM95] Weiser, M.; Gannon, J.; McMullin, P.: “Comparison of structural Test Coverage Metrics”, IEEE Software, März 1995, S. 80

[WHB01] Weitzel,T., Harder,T., Buxmann,P.: Electronic Business und EDI mit XML, dpunkt.verlag, Heidelberg, 2001, p. 15

[WhCo80] White, L., Cohen, E.: “A Domain Strategy for Computer Testing”, IEEE Trans. on SE, Vol. 6, Nr. 3, 1980, S. 247

[WHH80] Woodward, M., Hedley, D., Hennel, M.: “Experience with Path Analysis and Testing of Programs”, IEEE Trans on SE, Vol. 6, Nr. 3, March 1980, S. 278

[Win00] Winter, M.: „Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest“, Informatik Forschung und Entwicklung, Vol. 15, Nr. 3, 2000

[Win01] Winter, M.: „Testen“, Kapitel 21 in: J. Noack (Hrsg): Techniken der objektorientierten Softwareentwicklung, Springer, Berlin, 200, S. 474–540

[Win93] Winder, R.: Developing C++ Software, John Wiley & Sons, Chichester, G.B., 1993, S. 209

[Win98] Winter, M.: “Managing Object-Oriented Integration and Regression Testing”, in: Proc. of EuroSTAR 98, München, Dez. 1998, S. 189

[Win99] Winter, M.: Qualitätssicherung für Objektorientierte Software – Anforderungsermittlung und Test gegen die Anforderungsspezifikation, dissertation.de, Berlin, 2000, zugl. Dissertation, Fern Univ. Hagen, 1999

[YaKi87] Yau, S., Kishimoto, Z.: “A Method for Revalidating modified Programs in the Maintenance Phase”, in: Proc. of COMPSAC, IEEE Computer Society Press, Chicago, 1987, S. 272

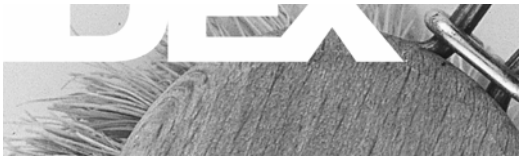
[Yam98] Yamura, T.: “How to design practical Test Cases”, IEEE Software, Nov. 1998, S. 30

[You89] Yourdon, E.: Modern Structured Analysis, Yourdon Press, Englewood Cliffs, N.J., 1989, S. 51

[Zei86] Zeil, S.: “The EQUATE Testing Strategy”, in Proc of ACM/IEEE Workshop on Software Testing, IEEE Computer Society Press, Banf Canada, 1986, S. 142

[Zei89] Zeil, S.: “Perturbation techniques for detecting Domain errors”, IEEE Trans. on S.E., Vol. 15, Nr. 6., June 1989, S. 737

[Zel90] Zelkowitz, M.: “A Functional Correctness Model of Program Verification”, IEEE Computer, Nov. 1990, S. 30



Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

Ü

U

V

W

X

Z