9 Funktionale Testmethoden

Wie wir bereits in den früheren Kapiteln gesehen haben, sind Methoden zur Testfall-Spezifikation wichtig für eine vernünftige Test-Strategie. Das Wissen, wie Dinge zu tun sind, bringt Struktur und erhöht die Effizienz. In diesem Kapitel werden einige Prinzipien zur Ableitung von Testfällen sowie ein paar repräsentative Beispiele für Testfall-Spezifikationstechniken beschrieben. Für mehr Details zu Testfall-Spezifikationstechniken wird ein Blick in die Literatur empfohlen, z.B. [Myers, 1979], [Beizer, 1984], [Kit, 1995] oder [Pol et al., 2000].

Unter funktionalen Testmethoden versteht man all diejenigen Testmethoden, die dem Test funktionaler Anforderungen dienen. Häufig sind damit nur die Methoden zur Spezifikation von funktionalen Testfällen gemeint. Methoden zum Testen nicht-funktionaler Qualitätskriterien werden in Kapitel 10 beschrieben.

Die Durchführung der verschiedenen funktionalen Testmethoden ist allerdings nicht auf allen Teststufen möglich. So gibt es zum Beispiel Testmethoden, die eher für das Testen von Einzelmodulen (Modultests) verwendet werden können, und solche, die erst nach der Integration möglich sind.

Dieses Kapitel beschreibt zunächst einige grundsätzliche Möglichkeiten, funktionale wie nicht-funktionale Testmethoden zu klassifizieren, sodann einige Qualitätsanforderungen an Testfälle, und zuletzt einige beispielhafte Methoden zur Spezifikation von funktionalen Testfällen.

9.1 Klassifikation von funktionalen Testmethoden

Funktionale Testmethoden besitzen einige wesentliche Eigenschaften, nach denen sie klassifiziert werden können.

- Die Frage, was als Basis für die Ableitung der Testfälle dient, die Spezifikation oder das Produkt selbst, führt zur Einteilung in Black-Box- und White-Box-Techniken.
- Die Testmethoden können die Ableitung der Testfälle entweder sehr detailliert vorgeben oder eher der Intuition des Testers überlassen. Dies führt zur Unterscheidung zwischen formalen und nicht formalen Testmethoden.
- Die Durchführung der Tests kann mit oder ohne Start des Systems geschehen, was zur Einteilung in dynamische und statische Testmethoden führt.
- Schließlich kann man Testmethoden nach ihrem Ziel klassifizieren, nämlich ob sie eher Fehler auffinden oder die Funktion des Systems beweisen wollen, was eher eine historische Sichtweise ist, aber grundlegende Qualitätsanforderungen an Testfälle offenbart.

9.1.1 Black-Box vs. White-Box

Zu einer Möglichkeit, Test-Methoden zu klassifizieren, gelangt man, wenn man sich überlegt, wie Testfälle abgeleitet werden. Abbildung 9.1 zeigt den Unterschied zwischen Black-Box- und

White-Box-Methoden. Reine Black-Box-Methoden fokussieren auf die Benutzeranforderungen und ignorieren die interne Programm-Struktur. Reine White-Box-Methoden benutzen die Programm-Struktur und vernachlässigen die Anforderungen.

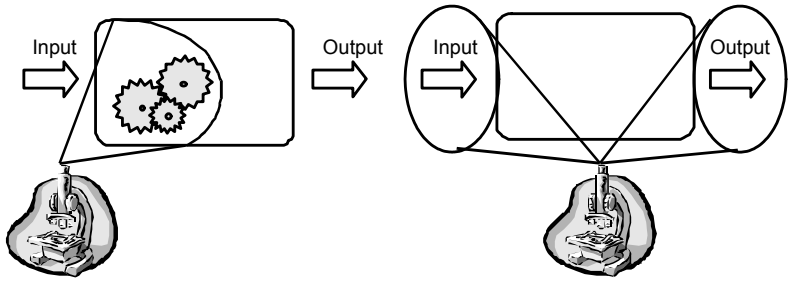

Abbildung 9.1: White-Box und Black-Box Testfälle

Myers definiert Black-Box- und White-Box-Methoden wie folgt [Myers, 1979]:

> *„One way to examine this issue is to explore a testing strategy called black-box, data-driven, functional or input/output-driven testing. In using this strategy, the tester views the program as a black box. That is, the tester is completely unconcerned about the internal behavior and structure of the program. Rather, the tester is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived solely from the specification (i.e., without taking advantage of knowledge of the internal structure of the program)."*
>
> *"Another testing strategy, white-box, structural or logic-driven testing, permits one to examine the internal structure of the program. In using this strategy, the tester derives test data from an examination of the program's logic (and often, unfortunately, at the neglect of the specification)."*

Beide Arten von Test-Methoden haben ihre Vor- und Nachteile. Der Haupt-Vorteil von Black-Box-Techniken ist deren Fokus auf die Spezifikation und damit die Benutzeranforderungen. Allerdings vernachlässigen sie die interne Programm-Struktur. Wenn z.B. drei ähnliche Anforderungen durch dasselbe Programmstück abgedeckt werden, würde unter Umständen ein Testfall genügen. Wenn nur die Anforderungen herangezogen werden, sind drei nötig. Zusätzlich sind Black-Box-Methoden nur für explizit spezifizierte Anforderungen möglich. Anforderungen, die implizit vorausgesetzt werden – und sich daher auch im Code manifestieren – werden nicht getestet.

White-Box-Methoden haben ihren Fokus auf die Programm-Struktur. Sie testen ein Programm so wie es ist, leider oft unter Vernachlässigung der Anforderungen, d.h. was das Programm tun sollte. Der Hauptvorteil von White-Box-Techniken entspringt ihrer Überdeckung von Code anstelle von Anforderungen. Sie bieten Möglichkeiten, die Code-Überdeckung zu variieren und erlauben Tests von Anforderungen, die nicht explizit genannt, aber im Programm vorhanden sind.

Die meisten Test-Methoden kombinieren Black-Box- und White-Box-Methoden. Testfälle können beispielsweise auf die Anforderungen basierend definiert werden, und dann kann ihre Anzahl

durch Wissen über die interne Programm-Struktur optimiert werden. Der Prozess der Kombination von Black-Box- und White-Box-Methoden kann oft als „Vorab-Test“ gesehen werden: Die Programm-Struktur muss zu der durch die Anforderungen verlangte Struktur passen, ansonsten kann das Programm auf keinen Fall korrekt sein, selbst wenn es alle richtigen Ergebnisse liefert – es gibt schließlich noch andere Kriterien als Funktionalität.

Beide Methoden haben ein Problem gemeinsam: um alle Fehler zu finden, verlangen Black-Box-Methoden umfangreiche Input-Tests, White-Box-Methoden umfangreiche Pfad-Tests. Dies bedeutet, dass entweder jede mögliche Eingabe-Bedingung als Testfall verwendet wird, oder dass Testfälle basierend auf allen möglichen Ausführungs-Pfaden definiert werden. Die damit verbundene Anzahl von Testfällen ist zu groß für einen ernsthaft gemeinten Test. Es gibt Methoden, die Anzahl der Testfälle ohne (große) Verluste bei der Überdeckung zu minimieren.

Die empfohlene Vorgangsweise ist, Testfälle anhand von Black-Box-Methoden zu erzeugen, und zusätzliche Testfälle nach Notwendigkeit durch White-Box-Methoden zu ergänzen [Myers, 1979]. Es gibt keinen Konkurrenzkampf zwischen beiden Methoden: beide sind essenziell, beide sind effektiv, aber beide haben Grenzen. Die zwei Konzepte stellen die Extrempunkte zwischen Struktur und Funktion dar. Modul-Tests tendieren zu mehr Struktur, Systemtests eher zur Funktion. Black-Box-Tests können im Prinzip alle Fehler finden, was allerdings eine unendliche Menge an Zeit benötigen würde. Strukturelle Tests sind finit, können aber nicht alle Fehler finden, selbst wenn sie komplett ausgeführt werden [Beizer, 1984].

9.1.2 *Formale* versus *Nicht-Formale* Testmethoden

Eine weitere Klassifizierung betrachtet die mathematische Formalität einer Test-Methode. Ein wichtiger Faktor für die Formalität ist die Test-Basis, d.h. die Grundlage, auf der die Testfälle basieren. Formale Spezifikations-Techniken leiten Testfälle von der System-Spezifikation, der Dokumentation oder dem Source-Code ab, während nicht formale Techniken vom Wissen des Testers über das System und seiner Erfahrung abhängen.

Formale Test-Methoden folgen strikten Regeln, die beschreiben, wie man Testfälle aus den gegebenen Dokumenten oder dem Source-Code erhält. Die Definition eines Testfalles für jeden Pfad der Länge zwei ist beispielsweise eine formale Methode. Wegen dieser strikten Regeln ist die Freiheit für Abweichungen davon limitiert, aber die Überdeckung kann exakt berechnet werden. Die Qualität von formalen Tests hängt einzig von der Qualität der Test-Basis ab.

Nicht formale Methoden basieren auf Kreativität. Es existieren viele Freiheitsgrade im Ableitungsprozess. Dies führt allerdings zu einer unsicheren Überdeckung. Die Qualität von nicht formalen Tests hängt vom Wissen des Testers und seiner Erfahrung ab.

In vielen Software-Projekten ist die Qualität der Systemspezifikation und/oder der Dokumentation eher niedrig, oder die verbleibende Zeit reicht kaum für Tests aus. Daher entscheidet sich ein Tester oft, nicht formale Test-Methoden (sprich: *Error Guessing*) zu verwenden. Allerdings sollte er berücksichtigen, dass in diesem Fall allein sein Wissen und seine Erfahrung die Test-Qualität bestimmen, und dass nicht formale Methoden zu unbekannter, möglicherweise sehr niedriger Überdeckung führen.

9.1.3 *Statische* versus *Dynamische* Testmethoden

Tests können abhängig davon eingeteilt werden, ob der Tester das Programm ausführt oder nicht. Methoden, bei denen das Programm ausgeführt werden muss, nennt man dynamisch, die anderen statisch.

Bei dynamischen oder „Computerbasierten“ Tests führt der Tester Funktionen aus und untersucht deren Eigenschaften und Verhalten. Um das zu erreichen, muss er Aktionen in bestimmter Reihenfolge ausführen. Eine Menge solcher Aktionen, die für die Ausführung eines einzelnen Testfalls nötig sind, nennt man Test-Skript [Pol et al., 2000].

Statische oder „*Menschenbasierte*“ Tests sind alle Arten von Tests, bei denen das Programm nicht ausgeführt wird. Sie werden benutzt, um bestimmte statische Eigenschaften des Systems, der von ihm erzeugten Daten oder des Source-Codes zu untersuchen. Der größte Vorteil von statischen Methoden ist, dass sie durchgeführt werden können, bevor die Implementierung beginnt, wodurch Fehler früh gefunden werden können.

Zusätzlich decken statische Tests eine Menge von Fehlern auf, wodurch diese en masse korrigiert werden können. Computerbasiertes Testen deckt üblicherweise nur ein Symptom eines Fehlers auf, und diese werden nacheinander gefunden und korrigiert. Die Erfahrung zeigt, dass diese „*Menschenbasierten*“ Tests sehr effizient bei der Fehlerfindung sind, sodass ein oder mehrere davon in jedem Software-Projekt verwendet werden sollten [Myers, 1979].

Beispiele für statische Tests sind Tests unter Verwendung von Checklisten, Reviews (siehe Kapitel 3), Syntax-Prüfungen und Konsistenz-Prüfungen.

9.1.4 *Error Finding* versus *Function Detection*

Eine weitere Klassifikation von Tests hat ihren Ursprung in den Zielen. Die Unterscheidung von *Error Finding* und *Function Detection* ist eher hypothetisch, da beide Ziele nur in der Theorie oder für sehr kleine Programme vollständig erreichbar sind. Allerdings führen uns Überlegungen über beide Extreme zu zwei wünschenswerten Eigenschaften von Testfall-Spezifikationen.

Das Ziel von Error Finding ist es, Fehlverhalten des Systems aufzudecken, d.h. alle Fehler. Es ist selbst in sehr kleinen Programmen sehr schwer, jeden einzelnen Fehler zu finden, also wird in den meisten Fällen dieses Ziel nicht erreicht. Um jedoch in einem gegebenen Zeitraum so viele Fehler wie möglich zu finden, muss der Tester den Fokus auf Funktionen mit der größten Fehlerwahrscheinlichkeit legen.

Function Detection hat als Ziel, die Korrektheit aller Funktionen im System zu zeigen. Der Aufwand, Korrektheit für alle Funktionen in allen Umständen zu zeigen, ist bis auf sehr kleine Systeme zu hoch. Die Korrektheit mancher Funktionen ist jedoch wichtiger als die anderer. Daher muss der Tester bei begrenzter Zeit (also immer) den Fokus auf die wichtigsten Funktionen legen.

Die beiden sich ergebenden Eigenschaften für Testfall-Spezifikationen sind Teil jeder vernünftigen Teststrategie.

9.2 Eigenschaften von Testfällen

Testfälle sind nicht einfach nur eine Menge mit voneinander ununterscheidbaren Elementen. Testfälle haben genauso Eigenschaften, welche ihre Qualität bestimmen, wie jedes andere Zwischenprodukt des Software-Entwicklungszyklus. Dieses Kapitel gibt einen Überblick über Qualitätsanforderungen an Testfälle und geht auf eine davon, genauer gesagt auf die Möglichkeit, diese zu quantifizieren, näher ein, nämlich auf die Überdeckungsmaße.

9.2.1 Qualitätsanforderungen an Testfälle

Testfälle können sehr stark in ihrer Qualität variieren. Um die Qualität besser bestimmen zu können, empfiehlt es sich auch hier, einzelne Qualitätsmerkmale einzuführen. Je mehr dieser Qualitätsmerkmale ein Testfall zu einem hohen Grad erfüllt, desto besser ist er.

Fehlerfindungsrate. Gute Testfälle haben eine hohe Wahrscheinlichkeit, Fehler zu finden. Es ist besser, einige wenige Testfälle mit hoher Fehlerfindungsrate durchzuführen, als viele, die kaum eine Chance haben, einen Fehler aufzudecken.

Fehlerlokalisierung. Testfälle können so konzipiert sein, dass aus der Tatsache, welche davon Fehler gefunden haben und welche nicht, Rückschlüsse gezogen werden können, wo im System sich ein Fehler tatsächlich befindet.

Überdeckung. Die Überdeckung gibt einen Prozentsatz der vom Testfall abgedeckten Testbasis an, z.B. der Codezeilen oder der Anforderungen. Eine hohe Überdeckung geht meist mit einer hohen Fehlerfindungsrate einher, aber nicht immer. Wenn ein Testfall mit hoher Überdeckung keinen Fehler findet, kann daraus auf eine gewisse Reife des Systems geschlossen werden.

Komplexität. Testfälle können einfach oder komplex sein. Die Komplexität kann in der Anzahl der nötigen Eingabedaten, der Anzahl der Aktionen, die zur Eingabe nötig sind, und auch in der Menge der notwendigen Vorbereitungen für den Testfall variieren. Ist ein Testfall zu komplex, kann es vorkommen, dass Fehler durch andere verdeckt werden. Tritt ein Fehler früh bei der Testfall-Durchführung auf, so können Fehler, die erst später auftreten, erst nach Korrektur des ersten Fehlers gefunden werden. Allerdings können komplexe Testfälle oft eine ganze Schar von einfachen Testfällen ersetzen. Es gilt also, einen Kompromiss zwischen der Anzahl und der Komplexität von Testfällen zu schließen

Generalität. Diese Eigenschaft betrifft die Beschreibung des Testfalls. Als Faustregel gilt, dass Testfälle so allgemein wie möglich und so genau wie nötig beschrieben werden sollten. Zu bevorzugen ist also eine generische Beschreibung der relevanten Eigenschaften ohne konkrete Werte. Diese werden erst beim Test vom Tester ausgewählt.

9.2.2 Überdeckungsmaß

Die Anzahl der Kombinationen von Aktionen, Bedingungen, Pfaden, Anforderungen etc., die getestet werden, bestimmen die Test-Intensität (Testüberdeckung, Coverage), die ihrerseits wieder Einfluss auf die Anzahl der einzigartigen Testfälle hat. Die Überdeckung kann durch genauere Betrachtung von einzelnen Bedingungen erhöht werden, oder durch größere Länge der

betrachteten Pfadabschnitte. Tabelle 9.1 zeigt einige bekannte Überdeckungsmaße für White-Box Tests.

Test-Intensität (Coverage)	**Beschreibung**
Statement coverage (Anweisungs-, C_0-Überdeckung)	Jede Aktion (= Anweisung) wird mindestens einmal ausgeführt.
Decision coverage (Zweig-, C_1-Überdeckung)	Jede Aktion wird mindestens einmal ausgeführt und jedes mögliche Ergebnis („wahr“ oder „falsch“) einer Entscheidung mindestens einmal erzeugt.
Condition coverage (einfache Bedingungsüberdeckung)	Jede Aktion wird mindestens einmal ausgeführt und jedes mögliche Ergebnis einer Bedingung (= Teil einer zusammengesetzten Entscheidung) wird zumindest einmal erzeugt.
Decision/condition coverage (Zweigüberdeckung mit Bedingungsüberdeckung)	Jede Aktion wird mindestens einmal ausgeführt und jedes mögliche Ergebnis einer Bedingung und einer Entscheidung mindestens einmal erzeugt. Das beinhaltet also sowohl condition als auch decision coverage.
Modified decision/condition coverage (minimale Mehrfach-Bedingungsüberdeckung)	Jede Aktion wird mindestens einmal ausgeführt, und jedes mögliche Ergebnis einer Bedingung bestimmt mindestens einmal unabhängig von anderen Bedingungsergebnissen das Ergebnis der Entscheidung. Das impliziert decision/condition coverage.
Multiple condition coverage (Mehrfach-Bedingungsüberdeckung)	Alle möglichen Kombinationen von Ergebnissen von Bedingungen in einer Entscheidung werden mindestens einmal erzeugt. Das impliziert modified decision/condition coverage.
$Path^n$ coverage mit n=1, 2, …, ∞ (Segmentpaarüberdeckung CSP, CS(n)-Überdeckung)	Jede mögliche Kombination von n aufeinander folgenden Anweisungen wird mindestens einmal ausgeführt.

Tabelle 9.1: Überdeckungsmaße für White-Box Tests [Pol et al., 2000]

Gewissheit, dass Pfade ausgeführt wurden, kann durch Instrumentierung der Pfade erreicht werden. Die einfachste Methode einen Pfad zu instrumentieren ist die Routine unter einem Trace laufen zu lassen [Beizer, 1984]. Bessere Methoden sind *Assertion-Statements* oder Werkzeuge, die eine Überdeckungsanalyse für eine gegebene Programmiersprache erlauben (siehe Kapitel 11).

Bei Black-Box Tests spricht man meist von Anforderungsüberdeckung, d.h. jede Anforderung muss mindestens einmal getestet worden sein. Hier kann analog zu den White-Box Tests die Überdeckung von Kombinationen von Anforderungen verlangt werden, um die Test-Intensität zu erhöhen. Allerdings gibt es hierfür keine eigenen Bezeichnungen.

9.3 Ausgewählte funktionale Spezifikationsmethoden für Testfälle

Bei der Definition von Testfällen muss der Tester immer berücksichtigen, dass er die wichtigsten Fehler so früh und so billig wie möglich finden will. Daher ist nicht nur eine Test-Strategie vorteilhaft, sondern auch die richtige Menge an Testfällen. Testfälle sollten so definiert werden, dass:

- sie mit einer hohen Wahrscheinlichkeit Fehler finden, d.h. dass sie einen großen Teil des Systems überdecken und eine hohe Anzahl von wichtigen und verschiedenen Fehlern finden,
- so wenige Aktionen wie möglich doppelt ausgeführt werden müssen, d.h. dass sie so billig wie möglich sind,
- sie so detailliert wie nötig sind aber so viele Freiheiten wie möglich bieten,
- der Prozess der Definition selbst so billig wie möglich ist.

Verwendung von Testfall-Spezifikationsmethoden ist ein einfacher Ansatz, um hohe Überdeckung des Systems zu erreichen. Viele Methoden bieten Möglichkeiten, die Überdeckung zu erhöhen oder zu verringern, wodurch Kostenaspekte berücksichtigt werden können. Manche Methoden sind teurer als andere, bieten dafür einen höheren Grad an Überdeckung oder einfach eine andere Art davon.

Eine Menge von Testfällen, die so wenige Redundanzen wie möglich beinhaltet, ist nicht leicht zu finden. Für manche Spezifikations-Methoden existieren Methoden zur Optimierung, bei manchen anderen ist die optimale Menge inhärent gegeben.

Um Testfälle zu erhalten, die einen bestimmten Freiheitsgrad bieten, aber detailliert genug sind, um einfach ausgeführt zu werden, müssen die notwendigen Anforderungen für Eingabedaten, Aktionen etc. in der Testfall-Spezifikation inkludiert werden. Dafür können Daten, die während der Spezifikationsphase des Projekts gewonnen wurden, verwendet werden. Man kann z.B. schreiben: „Eingabe eines Wertes aus der Äquivalenzklasse 1-99“ anstatt „Eingabe des Wertes 5“.

Der Prozess der Testfall-Spezifikation kann durch Automatisierungs-Mechanismen einfacher und billiger gemacht werden. Es existieren Methoden zur automatischen Testfall-Generierung, die ein Modell der Applikation verwenden (siehe z.B. [Howe et al, 1995]).

9.3.1 Prinzipien zur Ableitung von Testfällen

Wie in Kapitel 9.1 erwähnt, kann die Anzahl der Testfälle, die von einer gegebenen Test-Basis abgeleitet werden kann, sehr groß sein. Mit einigen wenigen grundlegenden Prinzipien kann eine eher bewältigbare Anzahl an Testfällen hergeleitet werden. Diese sind sowohl in Black-Box- als auch White-Box-Methoden, ob formal oder nicht, anwendbar. Einige dieser Basis-Prinzipien zur Ableitung von Testfällen sind:

- Zerlegung in Äquivalenzklassen
- Grenzwertanalyse

- Analyse der Verarbeitungslogik
- CRUD-Matrix
- Ursache-Wirkungs-Graphen
- Entscheidungstabellen
- Zustandsübergänge

In den folgenden Kapiteln werden einige Testmethoden vorgestellt, die auf diesen Prinzipien aufbauen.

9.3.2 Äquivalenzklassen-Zerlegung

Unter der Äquivalenzklassen-Zerlegung (eng. *equivalence partitioning*) versteht man die Zerlegung einer Menge von Daten (Input oder Output) in Untermengen (Klassen), sodass der Test eines Wertes einer Klasse äquivalent zum Test jedes anderen Wertes der Klasse ist [Myers, 1979]. Klassen umfassen sowohl gültige als auch ungültige Werte. Tests werden für zumindest einen Wert jeder Klasse durchgeführt. Von jedem Wert der Klasse wird angenommen, dass er mit derselben Wahrscheinlichkeit Fehler aufdeckt[1].

Üblicherweise wird die Äquivalenzklassenzerlegung als Black-Box-Methode auf Eingabe- oder Ausgabedaten von Geschäftsfällen angewendet. Sie kann aber genauso in Testmethoden Anwendung finden, welche Verzweigungen in einem formalen Modell des Programms oder den Programmcode selbst als Grundlage haben.

Der Entwurf von Test-Fällen auf Basis der Äquivalenzklassen-Zerlegung erfolgt in zwei Schritten [Myers, 1979]: der Identifikation der Äquivalenzklassen und der Definition der Testfälle. Für die Zerlegung von Eingabe-Daten eines Geschäftsfalls sehen diese Schritte folgendermaßen aus:

1. Die Identifikation der Äquivalenzklassen erfolgt in mehreren Schritten:
   a. Identifiziere die Eingabedaten für den Geschäftsfall: die Menge der Eingabedaten beschreibt einen *Eingabevektor*. Die Dimensionen des Vektors wollen wir *Eingabefeld* nennen. Zur Buchung einer Reise ist der Eingabevektor zum Beispiel definiert durch die Eingabefelder Nachname, Vorname, Anzahl Personen, Reiseziel, Reisedatum, Rückreisedatum, Preis, Rabatte, Sonderwünsche, etc.
   b. Identifiziere die Eingabe-Klassen: Zwei Typen von Äquivalenzklassen werden unterschieden: gültige Äquivalenzklassen repräsentieren gültige Eingaben an das Programm, und ungültige Äquivalenzklassen repräsentieren alle anderen möglichen Eingaben (z.B. fehlerhafte Eingabedaten). Je Eingabefeld sind entweder diskrete Eingaben (z.B. Werte in einer Combobox) oder kontinuierliche Eingaben (z.B. Geldbeträge) möglich. Alle möglichen Eingaben (sowohl gültige als auch ungültige) eines Eingabefeldes werden zu Eingabe-Klassen zusammengefügt, welche dieselben

[1] Das ist nicht ganz richtig, denn manche Werte haben eine höhere Wahrscheinlichkeit, nämlich die Grenzwerte. Die Grenzwertanalyse nutzt diese Tatsache.

Auswirkungen auf das System haben. Die Vereinigung aller Klassen muss die Menge aller Eingabemöglichkeiten ergeben, und die Klassen müssen paarweise disjunkt sein (daher der Begriff Äquivalenzklasse).

2. Definition der Testfälle: die Äquivalenzklassen werden herangezogen, um Testfälle zu finden. Die nötigen Schritte dazu sind:
   a. Weise jeder Klasse eine eindeutige Nummer zu.
   b. Schreibe einen neuen Testfall, der so viele der bisher nicht abgedeckten gültigen Klassen abdeckt wie möglich, bis alle gültigen Klassen durch mindestens einen Testfall abgedeckt sind.
   c. Schreibe einen neuen Testfall, der genau eine ungültige Klasse abdeckt, solange bis alle ungültigen Klassen durch einen Testfall abgedeckt sind.

Natürlich kann man, statt einfach nur jede Klasse mindestens einmal zu verwenden (entspricht C0-Überdeckung bei Tests basierend auf der Verarbeitungslogik, siehe Tabelle 9.1), auch jede mögliche Kombination von Klassen(-untermengen) verlangen (entspricht C_1 und höher). Dies führt dann allerdings zu einer sehr hohen Anzahl an Testfällen, vor allem bei Eingabevektoren mit vielen Dimensionen. Die Äquivalenzklassenzerlegung erstellt Testfälle, welche rein durch die Klassen beschrieben sind. Die Testfälle können durch Auswahl von Werten aus der Klasse weiter konkretisiert werden. Oft ist die generische Beschreibung vorteilhaft, bei welcher ein Tester erst bei der tatsächlichen Durchführung des Tests die konkreten Werte auswählt.

Beispiel ([Pol et al., 2000]). Eine Anforderung besagt $18 < \text{Alter} \leq 65$

Für "Alter" können wir drei unterschiedliche Klassen finden:

- A1 (ungültig): Alter ≤ 18
- A2 (gültig): $18 <$ Alter ≤ 65
- A3 (ungültig) Alter > 65

Nun definieren wir drei Testfälle, jeden durch einen Wert aus jeder Klasse, z.B. 10, 35 und 70.

Beispiel. Nehmen wir einmal an, ein Geschäftsfall verlangt als Eingabe ein Bundesland und einen Betrag. Bei den Bundesländern werden Vorarlberg, Tirol und Salzburg nach Variante 1 behandelt, die restlichen nach Variante 2. Bei den Beträgen gibt es für Variante 1 eine Staffelung in 1000er-Beträgen, von 0 bis 5000, unabhängig von den Varianten (d.h. es existiert keine Abhängigkeit vom Bundesland). Somit ergibt sich der Eingabevektor (A, B), wobei A für Bundesland steht, und B für den Betrag. Die Klassen von A sind dann A1 (westliche Bundesländer) und A2 (alle übrigen). Die Klassen von B sind B1 (ungültige Werte < 0), B2 (Werte x mit $0 < x < 1000$), B3 (Werte x mit $1000 \leq x < 2000$), B4 (Werte x mit $2000 \leq x < 3000$), B5 (Werte x mit $3000 \leq x < 4000$), B6 (Werte x mit $4000 < x \leq 5000$) und B7 (ungültige Werte > 5000). Als Testfälle ergeben sich dann: (A1, B1), (A2, B2), (A1, B3), (A1, B4), (A1, B5), (A1, B6), (A1, B7)

Bei Abhängigkeit zwischen den Eingabedaten A und B, beispielsweise einer Betrags-Staffelung wie oben für Klasse A1, und einer anderen Staffelung für Klasse A2, nämlich für Werte < 4000 (B8) und ≥ 4000 (B9), ergeben sich andere Testfälle:

(A1, B1), (A1, B2), (A1, B3), (A1, B4), (A1, B5), (A1, B6), (A1, B7)

(A2, B8), (A2, B9)

9.3.3 Grenzwert-Analyse

Die Grenzwertanalyse (engl. *boundary analysis*) stellt eine Erweiterung der Äquivalenzklassenzerlegung dar. Die Erfahrung zeigt, dass Testfälle, welche die Grenzwerte genauer unter die Lupe nehmen, eher Fehler finden als solche, die das nicht tun. Grenzwerte sind jene Situationen direkt auf, über oder unter den Grenzen einer Äquivalenzklasse. Anstatt einfach irgendein Element aus der Klasse zu wählen, werden bei der Grenzwert-Analyse ein oder mehrere Werte der Klasse gewählt, so dass jede Begrenzung der Klasse Teil eines Tests ist [Myers, 1979].

Beispiel ([Pol et al., 2000]). Eine Anforderung besagt: $18 < \text{Alter} \leq 65$

Laut Grenzwertanalyse werden für „Alter“ die Werte 18 (ungültig), 19 (gültig), 65 (gültig) und 66 (ungültig) gewählt.

Eine mögliche Erweiterung ist die Auswahl von Werten auf beiden Seiten der Grenze:

Beispiel ([Pol et al., 2000]). Eine Anforderung besagt: $\text{Alter} \leq 18$

Die für „Alter“ gewählten Werte sind: 17 (gültig), 18 (gültig) und 19 (ungültig). Eine falsch implementierte Bedingung „Alter = 18“) könnte mit den beiden Grenzwerten 18 und 19 nicht gefunden werden, nur durch Verwendung des zusätzlichen Werts 17.

9.3.4 CRUD-Matrix

Pol et al. beschreiben diese Art der Testfall-Ableitung so:

> *„Testfälle können sich auch auf den Lebenszyklus von Daten gründen (Create, Read, Update und Delete – CRUD). Daten entstehen, werden abgefragt und geändert und schließlich häufig wieder entfernt. Testfälle, die auf diesem Prinzip beruhen, untersuchen, ob die Daten von den Funktionen korrekt verarbeitet werden und ob den Beziehungskontrollen (Konsistenzprüfungen des Datenmodells) entsprochen wird. Auf diese Weise erhält man Einblick in den Lebenslauf (und dessen Vollständigkeit) der Daten bzw. Entitäten.“ [Pol et al., 2000]*

Zur Testfall-Ableitung wird eine Matrix mit den Funktionen als Zeilen und Entitäten (Klassen, Datenbank-Entitäten, etc.) als Spalten angelegt[2]. In die Schnittpunkte von Zeilen und Spalten wird eingetragen, was eine Funktion mit den Entitäten macht: C, R, U oder D. Tabelle 9.2.

Testfälle, die von einer CRUD-Matrix abgeleitet werden, bestehen aus den folgenden Grundelementen je Entität:

---

[2] Eine CRUD-Matrix während der Design-Phase anzulegen ist vernünftig, selbst wenn sie nicht für die Testfall-Spezifikation verwendet wird. Eine CRUD-Matrix kann Fehler und Unklarheiten aufdecken, die anders nicht so früh gefunden werden könnten.

1. Eingabe von Daten (pro Erzeugungsmöglichkeit ein Testfall)
2. Änderung (pro Änderungsmöglichkeit ein Testfall)
3. Löschung (mit jeder Löschfunktion) der Daten
4. Überprüfung von Daten nach jeder Aktion anhand der Abfragefunktionen

	Entität 1	**Entität 2**	**Entität 3**
Funktion 1	R	C,U,D	-
Funktion 2	C	R	-
Funktion 3	C,R,D	-	-

Tabelle 9.2: Beispiel für eine CRUD-Matrix

Die Generierung der Testfälle erfolgt durch ähnliche, bereits beschriebene Methoden. Zuerst wird die Matrix „geglättet“, so dass man eine Liste der Funktionen auf Entitäten erhält. Test-Skripts mit Funktionen für *Create*, *Read*, *Update* und *Delete* werden aus dieser Liste erzeugt. Die benutzten Funktionen werden von der Liste gestrichen bis keine Kombination Funktion/Entität mehr vorhanden ist.

Durch die Beziehungen zwischen Entitäten (z.B. *cascaded delete* in relationalen Datenbanken) können zusätzliche Testfälle notwendig werden.

9.3.5 Exploratives Testen

Exploratives Testen oder *Error Guessing*, d.h. das Raten von Fehlern, ist eine nicht formale Black-Box-Technik, die in jeder Test-Stufe und für beinahe alle Qualitätskriterien einsetzbar ist. Sie ist die einfachste Spezifikationstechnik. In ihrer simpelsten Form werden bloß zufällige Tests durchgeführt. Der Tester definiert und exekutiert Testfälle ad hoc.

Eine Verbesserung zu simplen zufälligen Tests stellt die Identifikation von möglichen Schwachstellen dar. Wahrscheinlichste Kandidaten für Schwachstellen sind:

- Teile des Systems, die häufig geändert wurden, als Konsequenz von Änderungen in den Anforderungen oder von Fehlerkorrekturen
- Teile des Systems, in welchen schon viele Fehler gefunden wurden
- Fehlerbehandlung: auf Fehler folgende Fehler, Abbruch eine Prozesses zu einem unerwarteten Zeitpunkt etc.
- Ungültige Werte: negative Werte, zu hohe oder zu niedrige Werte, Null, Leerstrings, etc.
- Sicherheit
- Systemverhalten im Falle von hoher Last
- Anforderung von zu vielen Ressourcen

Beim explorativen Testen sollte der Tester immer niederschreiben, was er tut und was geschieht [Kaner et al., 1999]. Das macht es leichter für ihn, reproduzierbare Fehlerberichte zu erzeugen.

Error Guessing ist diejenige Testfall-Spezifikationsmethode, die am häufigsten Anwendung findet, weil sie keiner Vorbereitung bedarf und nur minimal ausgebildetes Personal verlangt. Allerdings sollte sich der Testverantwortliche bewusst sein, dass – wie bei allen nicht formalen Techniken – die erreichte Testabdeckung unbekannt ist und sehr niedrig sein kann. Selbst ein Tester, der sehr erfahren sowohl im Test als auch in der Anwendung der Applikation ist, ist keine Sicherheit für eine hohe Fehlerfindungsrate. Daher sollte Error Guessing nur als Ergänzung zu formalen Methoden verwendet werden.

Was noch schlimmer ist: wenn nur Error Guessing verwendet wird, macht die Wichtigkeit von Erfahrung und Verständnis der Applikation ausgebildetes Personal zu einer Notwendigkeit für einen erfolgreichen Testprozess. Allerdings ist ausgebildetes Personal oft teuer oder einfach nicht verfügbar. In diesem Fall werden oft unerfahrene Tester für Error Guessing eingesetzt, was die schlechteste Wahl überhaupt ist.

9.3.6 Strukturtest

Der Struktur- oder Pfad-Test ist eine formale White-Box-Spezifikationsmethode. Er wird hauptsächlich für Modultests und Integrationstests eingesetzt und basiert auf Kontroll-Strukturen aus dem Systementwurf oder dem Source-Code. Im günstigsten Fall ist die Struktur als Graph gegeben. Ein Beispiel aus [Pol et al., 2000] ist in Abbildung 9.2 gegeben. Elemente des Graphen sind Entscheidungen, Anweisungsblöcke und Kanten zwischen diesen Elementen. Ein Testfall stellt dabei einen Pfad durch ein Programm oder eine Routine dar.

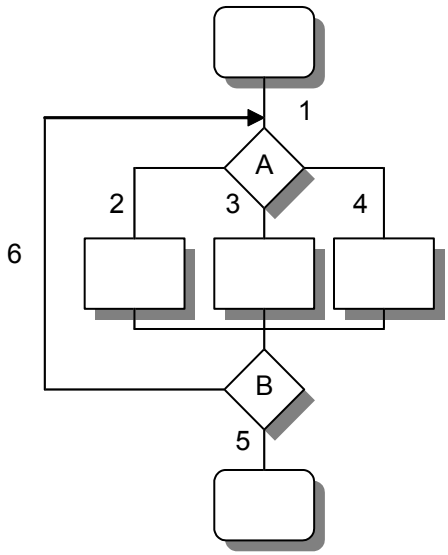

Abbildung 9.2: Beispiel-Programmstruktur [Pol et al., 2000]

Der Strukturtest baut auf Pfade von einem Start- zu einem Endpunkt auf. Ein Pfad durch eine Routine ist jede ausführbare Sequenz von Instruktionen durch diese Routine, die bei einem Eingangspunkt in die Routine beginnt und an einem der Austrittspunkte endet. Testfälle sind alle verschiedenen Pfade, die bei Betrachtung von Aktionskombinationen (Pfad-Teilstücken) der Länge n möglich sind. Die Länge n bestimmt das Testmaß des Strukturtests (*Path^n coverage*).

Die notwendigen Schritte für diese Spezifikationsmethode sind wie folgt:

Ermittlung der Entscheidungspunkte. Zuerst müssen die Bedingungen gefunden werden. Schleifen werden in Bedingungen übersetzt. Wenn ein Graph wie in Abbildung 9.2 existiert, müssen bloß die Entscheidungselemente genommen werden. Benennung der Entscheidungspunkte macht die folgenden Schritte leichter.

Ermittlung der Aktionskombinationen (AK). Aktionskombinationen sind alle Pfadstücke der Länge n. Wenn z.B. Statement-Überdeckung gefragt ist, dann sind Pfade der Länge eins die folgenden, geordnet nach den Entscheidungspunkten:

- -: (1)
- A: (2); (3); (4)
- B: (5); (6)

Spezifikation der Testfälle. Mögliche Testfälle sind alle Pfade, die beim Startknoten beginnen und beim Endknoten enden. Im Beispiel sind das (1) und (5). Testfälle werden nach folgendem Algorithmus gebildet:

1. Wähle eine beliebige, bisher nicht benutzte AK, beginnend bei einem Anfangszustand (z.B. (1)). Markiere diese AK als benutzt.

2. Wähle eine unbenutzte AK beginnend mit dem Endpunkt der zuletzt gewählten Aktionskombination (z.B. (2,5)). Markiere diese AK als benutzt. Falls keine unbenutzte AK mehr verfügbar ist, wähle den kürzesten Weg, der zu einer unbenutzten AK oder einem Endzustand führt.

3. Wiederhole Schritt 2 bis ein Endzustand erreicht ist. Falls keine unbenutzte AK mehr übrig ist, endet der Algorithmus hier.

4. Wiederhole Schritte 2 bis 3 bis keine unbenutzten Pfadkombinationen mehr übrig sind.

Betrachten wir das Beispiel mit Pfaden der Länge 1: Sofern möglich, sollten so viele noch nicht verwendete Aktionskombinationen wie möglich verwendet werden. Der erste Testfall ist (1, 2, 6, 3, 6, 4, 5). Mit diesem Testfall sind bereits alle AK der Länge 1 abgedeckt.

Unter Umständen ist ein einzelner Testfall dieser Komplexität nicht möglich, weil z.B. Abhängigkeiten innerhalb von oder zwischen Entscheidungen vorhanden sind. Für diesen Fall ist es zweckmäßiger, kürzere Testfälle zu definieren: (1, 2, 5) und (1, 3, 6, 4, 5) oder (1, 2, 5), (1, 3, 5), (1, 4, 5) und (1, 2 ,6 ,4, 5). Kurze Testfälle haben jedenfalls den Vorteil, dass sie Fehlerlokalisierungen genauer festlegen, und dass bei Auftreten eines Fehlers in einem Testfall noch andere Testfälle zur Verfügung stehen, die eine gewisse Wahrscheinlichkeit haben, weitere Fehler zu finden. Lange Testfälle decken möglicherweise einen Fehler auf und finden erst nach Korrektur des Fehlers den nächsten Fehler.

Beschreibung der initialen Datenbank. Die initiale Datenbank besteht aus den Eingabe-Daten, die benötigt werden, um Testfälle entlang der spezifizierten Pfade ausführen zu können. Die Beschreibung der Testfälle als Folge von Aktionskombinationen ist komplett generisch, ohne konkrete Daten. Diese müssen erst zugewiesen werden.

Erstellung der Test-Skripts. Die Test-Skripts bestehen aus Aktionen, Eingabedaten und Ergebnis-Prüfungen, die notwendig sind, um den Test durchzuführen.

Beispiel (aus [Pol et al., 2000]): Struktur aus Abbildung 9.2, Testmaß 2.

Die Aktionskombinationen der Länge 2 sind:

- A: (1, 2); (1, 3); (1, 4); (6, 2); (6, 3); (6, 4)
- B: (2, 5); (3, 5); (4, 5); (2, 6); (3, 6); (4, 6)

Wenn man versucht, eine möglichst große Anzahl an Aktionskombinationen in einen Testfall aufzunehmen, gelangt man zu folgendem Ergebnis:

- (1, 2, 6, 3, 6, 4, 6, 2, 5)
- (1, 3, 5)
- (1, 4, 5)

Die Auswahl der nächsten freien Aktionskombination ist nicht durch bestimmte Regeln festgelegt. Das optimale Set an Testfällen hat jedenfalls eine minimale Anzahl an mehrfach auszuführenden Aktionskombinationen; bei der zweiten Ausführung derselben Aktionskombination können mit hoher Wahrscheinlichkeit nur dieselben Fehler gefunden werden wie bei der ersten Ausführung.

Mit bestimmten Methoden kann man bei gegebenem Set an Aktionskombinationen bestimmen, ob und welche Testfälle überflüssig sind. Eine solche Methode wird im nächsten Kapitel beschrieben.

9.3.7 Geschäftsprozess-Test

Der Geschäftsprozess-Test [Pol et al., 2000] ist eine formale Black-Box-Methode. Er wird hauptsächlich zum Test der Integrierbarkeit eines automatisierten Prozesses in bestehende Geschäftsprozesse verwendet. Diese Test-Methode ist dem Strukturtest sehr ähnlich, aber hier werden die automatisierten Prozesse als Black Boxes gesehen. Als Basis dient der Kontrollfluss-Graph – oder andere formale Beschreibung – des Geschäftsprozesses.

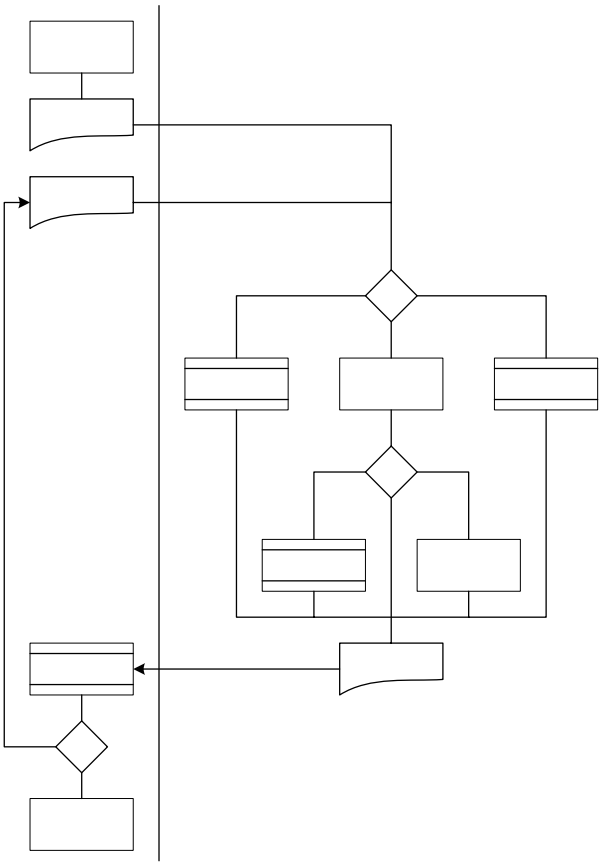

Abbildung 9.3: Beispiel: Kontrollflussgraph für Geschäftsprozesse [Pol et al., 2000]

Beim Geschäftsprozess-Test gibt es keine expliziten Prüfungen. Durch die Ausführung einer Aktion nach der anderen, wie es die Beschreibung des Geschäftsprozesses vorgibt, wird die

Integration jeder Aktion im Prozess implizit getestet. Übergänge von händischen zu automatisierten Prozessen und umgekehrt bedürfen besonderer Aufmerksamkeit.

Die Testfälle werden jetzt gleich gebildet wie beim Strukturtest. Der Algorithmus liefert Tabelle 9.3 als Ergebnis für den Kontrollfluss-Graphen in Abbildung 9.3.

#	Testmaß 1	Testmaß 2
1	(1, 2, 7)	(1, 2, 7)
2	(1, 2, 8, 3, 5, 7)	(1, 3, 5, 7)
3	(1, 3, 6, 8, 4, 7)	(1, 4, 7)
4		(1, 2, 8, 2, 7)
5		(1, 4, 8, 3, 5, 8, 4, 7)
6		(1, 3, 6, 8, 2, 7)

Tabelle 9.3: Testfälle für den Geschäftsprozesstest

Bei näherer Betrachtung sehen wir, dass manche Testfälle für das Testmaß 2 redundant sind, z.B. Testfall #1. Die Anzahl der Testfälle kann durch eine Optimierungstabelle[3] verringert werden, welche die Verwendung von Aktionskombinationen in Pfaden, d.h. Testfällen, darstellt. Tabelle 9.4 gibt ein Beispiel.

AK / Testfall	12	13	14	27	28	35	36	47	48	57	58	67	68	82	83	84
~~127~~	~~X~~			~~X~~												
1357		X				X				X						
~~147~~			~~X~~					~~X~~								
12827	X			X	X									X		
1367		X					X					X				
14835847			X			X		X	X		X				X	X
136827		X		X			X						X	X		

Tabelle 9.4: Matrix zur Testfall-Optimierung

Um die optimale Anzahl von Testfällen zu erhalten, muss jede Aktionskombination mindestens einmal verwendet werden, was 100% Überdeckung bedeutet, und jeder Testfall muss mindestens

[3] Solch eine Tabelle kann auch im Algorithmus für die Testfall-Spezifikation verwendet werden. Starte mit einer Tabelle nur mit Überschriften. Alle Aktionskombinationen werden in der Überschriften-Zeile eingetragen. Füge eine Zeile für jeden neuen Testfall ein, und markiere die benutzten Aktionskombinationen in dieser Zeile mit X. Der Algorithmus endet, wenn alle Spalten mindestens ein X enthalten.

eine AK enthalten, die sonst kein Testfall verwendet. Testfälle, die keine neuen AK durchführen erhöhen die Überdeckung nicht. Im gegebenen Beispiel sind die Testfälle (127) und (147) redundant: (127) ist in (12827) enthalten, und (147) in (14835847).

In manchen Fällen ist eine solche Optimierung nicht ideal, z.B. wenn die Applikation noch nicht sehr stabil ist. Komplexe Testfälle wie (14835847) könnten noch nicht ausführbar sein, aber (147) findet möglicherweise einen neuen Fehler. In solchen Fällen muss ein Kompromiss zwischen der Anzahl der Testfälle und deren Komplexität geschlossen werden.

9.3.8 Cause-Effect Graphing Test

Der *Cause-Effect Graphing Test* modelliert Testfälle in formaler Weise aus der Spezifikation, ist also eine formale Black-Box-Methode.

Die Testfälle werden wie folgt entwickelt:

1. Die Spezifikation wird in kleine überschaubare Stücke unterteilt, z.B. ein Programm in einzelne Methoden. Der Ursache-Wirkungs-Graph wird sonst sehr schnell unhandlich.
2. Ursachen und Wirkungen der Spezifikation festlegen. Eine Ursache ist eine Eingangsbedingung oder eine Äquivalenzklasse von Eingangsbedingungen. Eine Wirkung ist eine Ausgangsbedingung oder eine Systemtransformation (eine Nachwirkung, die die Eingabe auf den Zustand des Programms oder Systems hat). Ursache und Wirkung werden festgelegt, indem man die Spezifikation Wort für Wort liest und jedes Wort und jede Phrase, die eine Ursache bzw. Wirkung beschreibt, markiert und durch eine eindeutige Zahl kennzeichnet.
3. Der semantische Inhalt der Spezifikation wird analysiert und in einen booleschen Graphen transformiert, der die Ursachen und Wirkungen verbindet. Das wird als Ursache-Wirkungs-Graph bezeichnet.
4. Der Graph wird mit Kommentaren versehen, die Kombinationen von Ursachen und/oder Wirkungen, die aufgrund syntaktischer oder kontextabhängiger Beschränkungen nicht möglich sind, werden angegeben.
5. Der Graph wird, nachdem man die Zustandsbedingungen im Graphen methodisch verfolgt, in eine Entscheidungstabelle umgesetzt. Jede Spalte oder Tabelle stellt einen Testfall dar.
6. Die Testfälle werden aus den Spalten der Entscheidungstabelle gebildet. Da der Ursache-Wirkungs-Graph eine Umsetzung der Spezifikation in die Boolesche Logik ist, ist er ein guter Weg, Zweideutigkeiten und Unvollständigkeiten in der Spezifikation zu entdecken.

9.3.9 Transaktionsflussbasierter Test

Eine Transaktion ist ein Verarbeitungsmodul aus der Sicht eines Systembenutzers. Transaktionen bestehen aus einer Sequenz von Verarbeitungsschritten. Da es sich bei dieser Systemsicht um die Wahrnehmung eines Systems von außen handelt, passen transaktionsflussbasierte Tests besonders gut zum Systemtest.

Zur Notation von Transaktionsflüssen können verschiedene Darstellungsformen verwendet werden, wie zum Beispiel Flussdiagramme oder Sequenzdiagramme. Hier ein Beispiel für ein Sequenzdiagramm:

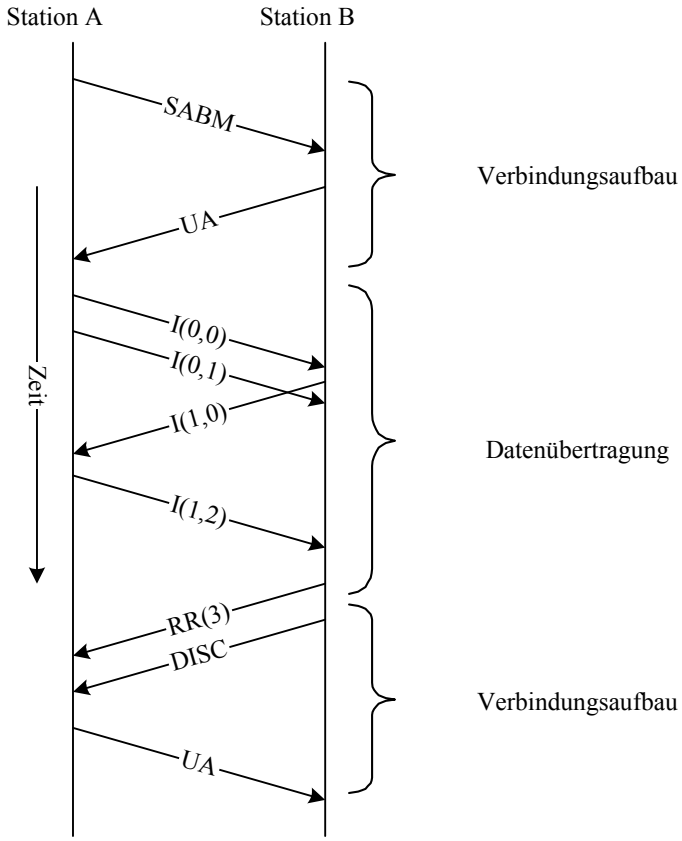

Abbildung 9.4: Sequenzdiagramm – Transaktionsfluss

In Sequenzdiagrammen werden die unterschiedlichen beteiligten Instanzen in der Horizontalen nebeneinander geschrieben (Station A, Station B). Die Zeit läuft von oben nach unten in vertikaler Richtung voran. Dieses Beispiel zeigt den Nachrichtenaustausch zwischen zwei HDLC-Protokollen auf zwei unterschiedlichen Stationen. Der Verbindungsaufbau geschieht durch eine einschlägige Aufforderung (SABM), die dann quittiert wird (UA). Anschließend werden die Daten transferiert. Am Ende wird die Verbindung abgebaut (DISC) und der Abbau wird quittiert (UA).

Einerseits ist ein solches Diagramm eine gute Ausgangsbasis für die Erzeugung von Testfällen - es gibt mögliche Testfälle direkt an. Andererseits haben Sequenzdiagramme in der Regel nur einen exemplarischen Charakter. So hätte in dem Beispiel die Aufforderung zum Verbindungsaufbau (SABM) nicht notwendig positiv quittiert werden müssen, sondern der Verbindungsaufbau hätte auch möglicherweise nicht zustande kommen können. Trotz dieses exemplarischen Charakters

sind sie eine gute Ausgangsbasis, denn es ist zu erwarten, dass diese Diagramme die besonders wichtigen Situationen beschreiben, deren Test notwendig geleistet werden muss. In Hinblick auf die Vollständigkeit des funktionsorientierten Tests ist ein transaktionsflussbasierter Ansatz aber als nicht hinreichend zu bewerten.

Der Test besteht nun in der Erzeugung der angegebenen Nachrichten (I(0,0), ...) und der Überprüfung, ob die erzeugten Antworten, so wie im Sequenzdiagramm angegeben, zustande kommen.

Der Vorteil der Sequenzdiagramme besteht darin, dass grundsätzlich beliebig viele beteiligte Instanzen dargestellt werden können. Die Grenze ist nur durch den Wunsch bestimmt, Übersichtlichkeit zu wahren.

9.3.10 Test auf Basis von Entscheidungstabellen oder –bäumen

Aus der Spezifikation leitet der Tester eine so genannte Entscheidungstabelle ab. Aus dieser Entscheidungstabelle können dann die anstehenden Testfälle herausdefiniert werden. Einerseits gewährleisten Entscheidungstabellen und Entscheidungsbäume durch ihre Systematik eine gewisse Testvollständigkeit. Andererseits steigt der Umfang dieser Darstellungen exponentiell mit der Anzahl der Bedingungen. Für umfangreiche Darstellungen sind diese Techniken daher nur begrenzt geeignet.

Beispiel. In einer Spezifikation wird beschrieben, wann die Bezahlung per Rechnung möglich ist. Ob eine Zahlung per Rechnung möglich ist, richtet sich danach, ob der Kunde Erstkunde ist, ob der Bestellwert größer als 1000 Euro ist und ob es sich um einen Privatkunden handelt.

In diesem Beispiel ergeben sich in der Entscheidungstabelle aus drei Bedingungen acht zu testende Fälle.

Bedingungen	Kunde = Erstkunde	N	N	N	N	J	J	J	J
	Bestellwert > 1000 Euro	N	N	J	J	N	N	J	J
	Kundentyp = Privatkunde	N	J	N	J	N	J	N	J
Aktion	Zahlung per Rechnung ist möglich	J	J	J	J	J	J	J	N

Es ist oft möglich, Entscheidungstabellen zu vereinfachen. Die oben dargestellte Tabelle kann ebenso vereinfacht werden.

Bedingungen	Kunde = Erstkunde	N	-	-	J
	Bestellwert > 1000 Euro	-	-	N	J
	Kundentyp = Privatkunde	-	N	-	J
Aktion	Zahlung per Rechnung ist möglich	J	J	J	N

Die optimierte Entscheidungstabelle stellt den gleichen Sachverhalt dar, wie die weiter oben dargestellte, umfangreichere Entscheidungstabelle. Eine Zahlung per Rechnung ist nur dann nicht

möglich, wenn ein Privatkunde erstmalig eine Bestellung im Wert von mehr als 1000 Euro tätigt. In allen anderen Fällen ist die Zahlung per Rechnung möglich.

Da die Größe solcher Entscheidungstabellen exponentiell mit der Anzahl der Bedingungen wächst, ist der Entscheidungstabellentest nur für Modultests geeignet.

9.4 Zusammenfassung

Unter funktionalen Testmethoden versteht man all diejenigen Testmethoden, die dem Test funktionaler Anforderungen dienen. Häufig sind damit nur die Methoden zur Spezifikation von funktionalen Testfällen gemeint. Nicht alle Methoden sind für die gleichen Teststufen sinnvoll. So sind z.B. manche Testmethoden, eher für Modultests geeignet, andere mehr für Integrationstests.

Bestimmte Eigenschaften von funktionalen Testmethoden führen zu folgender Klassifizierung:

- *Black-Box vs. White-Box*
  Black-Box-Methoden konzentrieren sich auf die Benutzeranforderungen und ignorieren die interne Programmstruktur. Reine White-Box-Methoden benutzen die Programm-Struktur und vernachlässigen die Anforderungen. Meist findet man eine Mischform: Testfälle können beispielsweise auf die Anforderungen basierend definiert werden, und dann kann ihre Anzahl durch Wissen über die interne Programm-Struktur optimiert werden.
- *formal vs. nicht-formal*
  Formale Spezifikations-Techniken leiten Testfälle von der System-Spezifikation, der Dokumentation oder dem Source-Code ab, während nicht formale Techniken vom Wissen des Testers über das System und seiner Erfahrung abhängen, also Kreativität verlangen. Oftmals wird aus Zeit- und/oder Kostengründen eine nicht-formale Methode wie z.B. Error Guessing verwendet. Die Test-Qualität hängt dann nur vom Wissen des Testers ab und liefert nur eine sehr geringe Überdeckung.
- *dynamisch vs. statisch*
  Test-Methoden, bei denen das Programm ausgeführt werden muss, nennt man dynamisch, die anderen statisch. Bei erstgenannten führt meist ein Test-Skript eine Menge von Aktionen in einer bestimmten Reihenfolge aus. Letztere haben den Vorteil, dass bereits vor Beginn der Implementierung getestet werden kann. Diese sogenannten Menschenbasierten Tests sind sehr effizient bei der Fehlerfindung wobei Computerbasierte Tests eher Symptome aufdecken.
- *Error-Finding vs. Function-Detection*
  Das Ziel von Error Finding ist es, Fehlverhalten des Systems aufzudecken, d.h. alle Fehler zu finden. Da dies meist unmöglich ist, liegt es am Tester, den Fokus auf die Funktionen mit der höchsten Fehlerwahrscheinlichkeit zu richten. Function Detection hat als Ziel, die Korrektheit aller Funktionen im System zu zeigen, was ebenfalls nahezu unmöglich ist. Deshalb muss der Test diejenigen Funktionen auswählen und deren Korrektheit zeigen, die für das System am wichtigsten sind.

Da auch Testfälle stark in ihrer Qualität variieren können, empfiehlt es sich auch hier Qualitätsmerkmale einzuführen. Denkbar wären z.B. die folgenden:

Fehlerfindungsrate. Gute Testfälle haben eine hohe Wahrscheinlichkeit, Fehler zu finden.

Fehlerlokalisierung. Es können Rückschlüsse gezogen werden können, wo im System sich ein Fehler tatsächlich befindet.

Überdeckung. Die Überdeckung gibt einen Prozentsatz der vom Testfall abgedeckten Testbasis an.

Komplexität. Komplexe Testfälle können oft eine ganze Schar von einfachen Testfällen ersetzen. Es gilt also, einen Kompromiss zwischen der Anzahl und der Komplexität von Testfällen zu schließen

Generalität. Als Faustregel gilt, dass Testfälle so allgemein wie möglich und so genau wie nötig beschrieben werden sollten.

Die Anzahl der Kombinationen von Aktionen, Bedingungen, Pfaden, Anforderungen etc., die getestet werden, bestimmen die Test-Intensität (Testüberdeckung, Coverage), die ihrerseits wieder Einfluss auf die Anzahl der einzigartigen Testfälle hat.

Bei der Spezifikation von Testfällen ist darauf zu achten, dass sie Fehler so früh und so billig wie möglich finden. Durch die Verwendung von Testfall-Spezifikationsmethoden kann der richtige Mix aus Überdeckung, Zeit und Geld gefunden werden.

Die folgenden Methoden sind nur ein Beispiel für Prinzipien, durch die die Menge an Testfällen gesenkt werden kann und somit der Testaufwand deutlich reduziert werden kann:

- Äquivalenzklassen-Zerlegung
- Grenzwert-Analyse
- Analyse der Verarbeitungslogik
- CRUD-Matrix
- Exploratives Testen (Error Guessing)
- Strukturtests
- Geschäftsprozess-Tests
- Ursache-Wirkungs-Graphen
- Transaktionsflussbasierte Tests
- Entscheidungstabellen oder -bäume
- Zustandsübergänge

9.5 Literaturreferenzen

[Beizer, 1984] Boris Beizer: "System Testing and Quality Assurance", van Nostrand, 1984, ISBN 0-442-21306-9

[Howe et al, 1995] Howe, A.E.; von Mayrhauser, A.; Mraz, R.T.: "Test sequences as plans: an experiment in using an AI planner to generate system tests", Proceedings of the 10th Knowledge-Based Software Engineering Conference, 1995, Pages: 184 –191

[Kaner et al., 1999] Cem Kaner, Jack Falk, Hung Quoc Nguyen: “Testing Computer Software", Wiley, 1999, ISBN 0-471-35846-0

[Kit, 1995] Edward Kit: "Software Testing in the Real World", Addison-Wesley, 1995, ISBN 0-201-87756-2

[LIGGES2002] Peter Liggesmeyer „Software-Qualität: Testen, Analysieren und Verifizieren von Software“, Spektrum Akademischer Verlag, 2002, ISBN 3-8274-1118-1)

[Myers, 1979] Glenford J. Myers: "The Art of Software Testing", Wiley, 1979, ISBN 0-471-04328-1

[Pol et al., 2000] Martin Pol, Tim Koomen, Andreas Spillner: “Management und Optimierung des Testprozesses: ein praktischer Leitfaden für Testen von Software, mit TPI und TMap”, dpunkt.verlag, 2000, ISBN 3-932588-65-7

[Thaller1997] Georg Erwin Thaller: „Software-Test: Verifikation und Validation“, Heise, 1997, ISBN 3-88229-183-4)

9.6 Übungsaufgaben