Andreas Spillner · Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

Konform zu ISO 29119

dpunkt.verlag

Was sind E-Books von dpunkt?

Unsere E-Books sind Publikationen im PDF- oder ePub-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken.

Sie benötigen zum Ansehen den Acrobat Reader oder ein anderes adäquates Programm bzw. einen E-Book-Reader.

E-Books können Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt.büchern her kennen.

Was darf ich mit dem E-Book tun?

Wie erhalte ich das E-Book von dpunkt?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene Adresse eine Bestätigung. Außerdem erhalten Sie von dpunkt eine E-Mail mit den Downloadlinks für die gekauften Dokumente sowie einem Link zu einer PDF-Rechnung für die Bestellung.

Die Links sind zwei Wochen lang gültig. Die Dokumente selbst sind mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag
e-mail: ebooks@dpunkt.de
fon: 06221/1483-0.

Andreas Spillner ist Professor für Informatik an der Hochschule Bremen. Er war über 10 Jahre Sprecher der Fachgruppe TAV »Test, Analyse und Verifikation von Software« der Gesellschaft für Informatik e.V. (GI) und bis Ende 2009 Mitglied im German Testing Board e.V. Im Jahr 2007 ist er zum Fellow der GI ernannt worden. Seine Arbeitsschwerpunkte liegen im Bereich Softwaretechnik, Qualitätssicherung und Testen.

Ulrich Breymann war als Systemanalytiker und Projektleiter in der Industrie und der Raumfahrttechnik tätig. Danach lehrte er als Professor Informatik an der Hochschule Bremen. Er arbeitete an dem ersten C++-Standard mit und ist ein renommierter Autor zu den Themen Programmierung in C++, C++ Standard Template Library (STL) und Java ME (Micro Edition).

Andreas Spillner · Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

dpunkt.verlag

Prof. Dr. Andreas Spillner
Andreas.Spillner@hs-bremen.de

Prof. Dr. Ulrich Breymann
breymann@hs-bremen.de

http://leantesting.de

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: die Autoren mit LaTeX
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Print 978-3-86490-308-3
PDF 978-3-86491-967-1
ePub 978-3-86491-968-8
mobi 978-3-86491-969-5

1. Auflage 2016

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Vorwort

Liebe Leserinnen und Leser,

es ist das Ziel eines jeden Softwareentwicklers[1], Programme mit möglichst wenigen Fehlern zu schreiben. Wie man weiß, ist das weiter gehende Ziel einer fehlerfreien Software nicht zu erreichen, von sehr kleinen Programmen abgesehen. Es ist aber möglich, die Anzahl der Fehler zu reduzieren. Dabei helfen erstens konstruktive Maßnahmen. Dazu gehört die Einhaltung von Programmierrichtlinien ebenso wie das Schreiben eines verständlichen Programmtextes. Zweitens hilft das Testen, also die Prüfung der Software, ob sie den Anforderungen genügt und ob sie Fehler enthält.

Die beim Testen häufig auftretende Frage ist, wie viel Aufwand in einen Test gesteckt werden soll. Einerseits möglichst wenig, um die Kosten niedrig zu halten, andererseits möglichst viel, um dem Ziel der Fehlerfreiheit nahezukommen. Letztlich geht es darum, einen vernünftigen Kompromiss zwischen diesen beiden Extremen zu finden. Der Begriff »lean« im Buchtitel bedeutet, sich auf das Wichtige zu konzentrieren, um diesen Kompromiss zu erreichen. Die Frage des Aufwands ist aber nur vordergründig ausschließlich für Tester von Bedeutung.

Tatsächlich checkt ein Softwareentwickler seinen Code erst ein, wenn er ihn auf seiner Ebene, also der Ebene der Komponente oder Unit, getestet hat. Er ist interessiert an der Ablieferung guter Software und an der Anerkennung dafür. Er muss aber auch darauf achten, nicht mehr Zeit als angemessen zu investieren. Dieses Buch soll eine Brücke zwischen Programmierung und Testen für den C++-Entwickler bauen und ihm zeigen, welche Testverfahren es gibt und wie sie mit vertretbarem Aufwand auf seiner Ebene eingesetzt werden können.

Zum fachlichen Hintergrund der Autoren: Ulrich Breymann ist mit seinem Standardwerk »Der C++-Programmierer« [Breymann 15] in C++-Programmierer-Kreisen bekannt. Damit lernen Leser, wie sie in C++ programmieren können und dabei durch guten Programmierstil Qualität in ihre Programme bekommen. Andreas Spillner hat mit »Basiswissen Softwaretest« [Spillner & Linz 12] im Bereich des Testens ebenfalls ein grund-

[1] Geschlechtsbezogene Formen meinen hier und im Folgenden stets Frauen, Männer und alle anderen.

legendes Buch geschrieben. Der Inhalt seines Buches orientiert sich am internationalen Lehrplan »Certified Tester – Foundation Level« und umfasst neben einigen der hier aufgeführten Testverfahren noch weitere Themen.

Das vorliegende Buch zeigt die praktische Anwendung der Testverfahren für C++-Programme mit zahlreichen ausführlichen Beispielen. Dabei liegt der Fokus auf »Lean Testing«, also dem Versuch, einen guten Kompromiss zwischen angestrebter Qualität und Testaufwand zu finden.

Wir hoffen, Ihnen beim Durcharbeiten der folgenden Kapitel viele Hinweise und Anregungen für den Test Ihrer Software als Teil der täglichen Arbeit geben zu können.

Unserer Lektorin Frau Preisendanz und dem dpunkt-Team danken wir für die sehr gute Zusammenarbeit.

Bremen, im April 2016

Andreas Spillner & Ulrich Breymann

Inhaltsverzeichnis

1 Einleitung

Worum geht's

Kommt Ihnen die folgende Situation bekannt vor? Die User Story (das Feature, die Klasse, das Anforderungsdetail) ist fertig programmiert und bereit zum Einchecken für den Continuous-Integration-Prozess. Ich habe alles sorgfältig bedacht und ordentlich programmiert, aber bevor ich den Code einchecke, möchte ich noch meinen Systemteil testen. Es wäre ja zu ärgerlich, wenn es später Fehler gibt, deren Ursache in meinem Teil liegt; muss ja nicht sein! Also los geht's mit dem Testen. Aber wie und womit fange ich an und wann habe ich ausreichend genug getestet?

Genau hierfür gibt das Buch Hinweise! Es beantwortet die Fragen: Wie erstelle ich Testfälle[1]? Welche Kriterien helfen, ab wann ein Test als ausreichend angesehen und damit beendet werden kann? Wir meinen, dass jeder Entwickler auch testet, zumindest seinen eigenen Programmcode, wie in der oben beschriebenen Situation. Und wenn der Entwickler dann praktische Hinweise in seiner vertrauten Programmiersprache, hier C++, bekommt, so hoffen wir, dass es zur Verbesserung des Testvorgehens bei ihm führt und ihm bei seiner täglichen Arbeit hilft.

Wie wäre es mit folgendem Ablauf: Die User Story, das Feature, die Klasse, das Anforderungsdetail ist fertig programmiert und vor dem Einchecken für den Continuous-Integration-Prozess erfolgten die Schritte:

- Mit dem statischen Analysewerkzeug Scan-Build habe ich keine Hinweise auf Fehler erhalten, ebenso erzeugte der Compilerlauf in der höchsten Warnstufe keine Meldungen.
- Drei systematische Testverfahren (Äquivalenzklassenbildung, Grenzwertanalyse und zustandsbasierter Test) habe ich durchgeführt und dabei die geforderten Kriterien zu einer Beendigung der Tests erfüllt (die beiden dabei aufgedeckten Fehler habe ich beseitigt und der anschließende Fehlernachtest lief zufriedenstellend, also fehlerfrei).

[1] Siehe Glossar Seite 229. Viele weitere Begriffe sind im Glossar aufgeführt. Wir verzichten aber wegen der besseren Lesbarkeit auf weitere Fußnoten mit den entsprechenden Hinweisen.

- Zum Abschluss habe ich explorativ getestet. Dabei legte ich das Augenmerk besonders auf die Programmstelle, bei der ich mir beim Programmieren nicht so ganz sicher war, ob sie auch richtig funktionieren wird. Der Code war in Ordnung und ich habe keine weiteren Fehler gefunden. Ich habe alles sorgfältig dokumentiert, sodass ich nachweisen kann, dass ich nicht nur ordentlich programmiert, sondern auch ausreichend getestet habe, bevor ich den Code einchecke. Klar kann ich keine Fehlerfreiheit damit nachweisen, aber ich habe ein sehr gutes Gefühl und hohes Vertrauen, dass mein Stück Software zuverlässig läuft und nicht gleich nach dem Einchecken einen *Bug* produziert. Und ich habe nicht viel Zeit für die Tests verschwendet!

Das Buch beschränkt sich auf den sogenannten Entwicklertest, also den Test, den der Entwickler direkt nach der Programmierung durchführt. Andere geläufige Bezeichnungen sind Unit Test, Komponententest oder Modultest, um nur einige zu nennen.

Es geht darum, die kleinste Einheit zu wählen, bei der es Sinn macht, einen separaten Test durchzuführen. Dies kann eine Methode oder Funktion einer Klasse sein oder auch eine Zusammenstellung von mehreren Klassen, die eng miteinander in Kommunikation stehen.

> Wir wollen den Entwickler nicht zum Tester umschulen. Es geht vielmehr darum, den Entwickler beim Test seiner Software zu unterstützen und ihm sinnvolle und unterschiedliche Vorgehensweisen an die Hand zu geben.

TDD (Test Driven Development)

Viele Autoren empfehlen die testgetriebene Entwicklung[2] – zu Recht! Damit ist gemeint, dass zuerst die Testfälle auf Basis der Spezifikation entwickelt werden, und erst danach der damit zu testende Programmcode geschrieben wird. Dieses Buch konzentriert sich nicht auf TDD, aber fast alle der genannten Verfahren sind dafür sehr gut geeignet. Letztlich sind sie unabhängig davon, ob noch zu schreibender oder schon vorhandener Code damit getestet werden soll. Eine Ausnahme sind die Verfahren zur statischen Analyse von Programmen sowie die Whitebox-Tests, die vorhandenen Programmcode voraussetzen.

[2] Andere Bezeichnung ist Test-first Programming (siehe Glossar).

Test-Büffet

Wir sehen den Inhalt des Buches als »Test-Büffet«. Wie bei einem Büffet gibt es reichlich Auswahl und der Hungrige entscheidet, welche Wahl er trifft und wie viel er sich von jedem Angebot nimmt, auch wie viel Nachschlag er noch »verträgt«. Ähnlich ist es auch mit dem Testen: Es gibt nicht das eine Testverfahren, mit dem alle Fehler aufgedeckt werden; sinnvoll ist immer eine Kombination mehrerer Verfahren, die der Entwickler passend zum Problem aussucht. Wie intensiv und ausgiebig die einzelnen Verfahren anzuwenden sind – wie viel sich jeder vom Büffet von einer Speise auftut – ist ihm überlassen, er kennt sein Testobjekt – seinen Geschmack – am besten. Wir geben Empfehlungen, welche Reihenfolge anzuraten ist und welche Speisen in welchem Umfang gut zusammenpassen – ein Eis vorweg und danach fünf Schweineschnitzel und eine Karotte scheint uns keine ausgewogene Zusammenstellung.

Beim Büffet gibt es Vor- und Nachspeisen sowie Hauptgerichte. Ebenso verhält es sich beim Testen: Statische Analysen sind vor dem eigentlichen Testen besonders sinnvoll, die Haupttestverfahren sind die Verfahren, bei denen die Testfälle systematisch hergeleitet werden. Erfahrungsbasierte Verfahren runden das Menü ab. Eine ausgewogene Zusammenstellung ist die Kunst – nicht nur beim Büffet, sondern auch beim Testen. Nur mit Nachspeisen seinen Hunger zu stillen, ist sicherlich verlockend, aber rächt sich meist später. Ausschließlich auf die eigene Erfahrung beim Testen der eigenen Software zu setzen, birgt das Problem der Blindheit gegenüber den eigenen Fehlern. Wenn ich als Entwickler die User Story falsch interpretiert oder etwas nicht bedacht habe, werde ich, wenn ich meine Rolle als Entwickler mit der Rolle des Testers vertausche, nicht automatisch die Fehlinterpretation durch die korrekte in meinem Kopf ausgetauscht bekommen. Wenn ich aber systematische Testverfahren verwende, dann erhalte ich durch die Verfahren möglicherweise Testfälle, die mich auf meine Fehlinterpretation oder die nicht bedachte Lücke hinweisen oder mich zumindest zum Nachdenken animieren.

»Lean Testing«

Beim Büffet wird wohl keiner auf die Idee kommen, das Büffet komplett leer essen zu wollen. Uns scheint es, dass beim Testen aber ein ähnliches Bild in manchen Köpfen noch vorherrscht.

So schreibt Jeff Langr beispielsweise [Langr 13, S. 35]: »Using a testing technique, you would seek to exhaustively analyze the specification in question (and possibly the code) and devise tests that exhaustively cover the behavior.« Frei übersetzt: »Beim Testen versuchen Sie, die zugrunde liegende Spezifikation (und möglicherweise den Code) vollständig zu analysieren

und Tests zu ersinnen, die das Verhalten vollständig abdecken.«[3]

Er verbindet das Testen mit dem Anspruch der Vollständigkeit. Dies ist aber unrealistisch und kann in der Praxis in aller Regel nie erfüllt werden.

> Schon bei kleineren, aber erst recht bei hochkritischen Systemen ist ein »Austesten«, bei dem alle Kombinationen der Systemumgebung und der Eingaben berücksichtigt werden, nicht möglich.

Es ist aber auch gar nicht erforderlich, wenn einem bewusst ist, dass ein Programmsystem während seiner Einsatzzeit nie mit allen möglichen Kombinationen ausgeführt werden wird.

Ein kurzes Rechenbeispiel soll dieses veranschaulichen: Nehmen wir an, wir hätten ein sehr einfaches System, bei dem 3 ganze Zahlen einzugeben sind. Jede dieser Zahlen kann 2^{16} unterschiedliche Werte annehmen, wenn wir von 16 Bit pro Zahl ausgehen. Bei Berücksichtigung aller Kombinationen ergeben sich dann $2^{16} \cdot 2^{16} \cdot 2^{16} = 2^{48}$ Möglichkeiten. Dies sind 281.474.976.710.656 unterschiedliche Kombinationen der drei Eingaben. Damit die Zahl greifbarer wird, nehmen wir an, dass in einer Sekunde 100.000 unterschiedliche Programmläufe durchgeführt werden. Nach 89,2 Jahren hätten wir jede mögliche Kombination einmal zur Ausführung gebracht. Bei 32 Bit pro Zahl ergäben sich sogar $2{,}5 \cdot 10^{16}$ Jahre. Noch Fragen?

Es muss daher eine Beschränkung auf wenige Tests vorgenommen werden. Es gilt, einen vertretbaren und angemessenen Kompromiss zwischen Testaufwand und angestrebter Qualität zu finden. Dabei ist die Auswahl der Tests das Entscheidende! Eine Konzentration auf das Wesentliche, auf die Abläufe, die bei einem Fehler einen hohen Schaden verursachen, ist erforderlich.

> Es müssen die richtigen Tests und nicht alle möglichen und vorstellbaren Tests ausgeführt werden.

Zu diesem Zweck gibt es Testverfahren, die eine Beschränkung auf bestimmte Testfälle vorschlagen. Wir haben unserem Buch den Titel »Lean Testing« gegeben, um genau diesen Aspekt hervorzuheben. Wir wollen dem Entwickler Hilfestellung geben, damit er die für sein Problem passenden Tests in einem angemessenen Zeitaufwand durchführen kann, um die geforderte

[3] Wir verwenden das Zitat aus dem englischen Buch und haben es selbst übersetzt, da die vorhandene Übersetzung [Langr 14] in diesem Punkt den Sachverhalt nach unserer Meinung nur in abgeschwächter Form wiedergibt.

Qualität mit den Tests nachzuweisen. Ein vollständiger Test wird nicht angestrebt. Wir wollen unser Essen vom Büffet beenden, wenn wir ausreichend gesättigt sind und eine für unseren Geschmack passende Auswahl von Speisen – nicht alle – probiert haben.

»Lean Testing« setzt »Lean Programming« voraus

Um mit wenig Testeinsatz viel überprüfen zu können, muss der Code – das Testobjekt – möglichst einfach sein. Trickreicher und »künstlerischer, freier« Programmierstil sind da nicht gewünscht. Aber glücklicherweise hat sich in den letzten Jahren ein Wandel hin zum einfachen guten Programmierstil ergeben.

Die Beachtung der »Clean-Code-Prinzipien« schafft eine wichtige Voraussetzung, den Test angemessen aufwendig gestalten zu können. Erst durch eine einfache Programmstruktur ist eine einfache Testbarkeit gegeben. Die einfache Testbarkeit garantiert, dass der Test mit einfachen Methoden und Ansätzen durchgeführt werden kann und damit »lean« ist. Auch Refactoring ist ein wichtiger Pfeiler für eine einfache Testbarkeit. Wenn der Code unübersichtlich wird, sind Vereinfachungen vorzunehmen. Listing 1.1 zeigt ein Beispiel für einen Programmcode, bei dem sich Refactoring lohnt:

```
// gibt Preis in Eurocent zurück
int fahrpreis(int g,                    // Grundpreis
              int c,                    // Preis pro KM
              int s,                    // Strecke
              bool n,                   // Nachtfahrt
              bool gp) {                // Gepäck ja/nein
  int b = c * s;                        // Basispreis
  int r = 0;                            // Rabatt
  int z = 0;                            // Zuschlag für Nachtfahrt
  if(s > 50)
    r = static_cast<int>(0.1 * b + 0.5);
  else if(s > 10)
    r = static_cast<int>(0.05 * b + 0.5);
  if(n)
    z = static_cast<int>(0.2 * b + 0.5);
  if(gp)
    z += 300;                           // Zuschlag für Gepäck
  return  g + b + z - r;
}
```

Listing 1.1: »Unsauberer« Code

Die an diesem Listing zu kritisierenden Punkte sind:

1. Die Variablennamen werden kommentiert, sind aber sehr kurz. Besser ist es, Namen zu verwenden, die die Kommentierung überflüssig machen. Wenn der Code beim Lesen über eine Seite geht, sind die Kommentare verschwunden, und es muss möglicherweise umständlich zurückgeblättert werden.
2. Die Anweisungen nach den `if`-Bedingungen sind nicht in geschweifte Klammern eingeschlossen – eine mögliche Fehlerquelle, wenn die Anweisungen durch weitere ergänzt werden sollen.
3. Dreimal wird `static_cast` verwendet. Die 0.5 deutet darauf hin, dass ein `double`-Wert gerundet werden soll. Besser wäre es, den Vorgang des Rundens in eine eigene Funktion mit geeignetem Namen auszulagern, damit beim Lesen klar wird, was geschehen soll. Anstelle einer eigenen Funktion eignet sich dafür die C++-Funktion `std::round()`. Im Beispiel fällt auf, dass die Rundung nur für positive Werte von `b` korrekt ist. `std::round()` rundet auch negative Werte korrekt.
4. `z` wird zu Beginn deklariert, nicht kurz vor der Stelle der ersten Verwendung – das Lokalitätsprinzip wird verletzt.

Nach dem Refactoring könnte die Funktion so aussehen, wie sie in Listing 4.35 auf Seite 102 abgedruckt ist.

Beide Ansätze – Clean Code und Refactoring – sehen wir nicht nur im agilen Umfeld als sinnvoll an, ganz im Gegenteil: Einfache Programmierung ist in allen Bereichen und überall anzustreben.

Worum geht's nicht

Auf einem Büffet, sei es auch noch so umfangreich oder von der Zusammenstellung her thematisch begrenzt (z.B. ein Fisch- oder Vegan-Büffet), sind nie alle möglichen Speisen zu finden. So verhält es sich auch mit diesem Buch. Folgendes wird nicht behandelt:

Qualität wird bei der Entwicklung von Software produziert. Mit Testen kann nur die erreichte Qualität nachgewiesen, aber nicht verbessert werden. Stichpunkte sind die Vermeidung von unsicheren Sprachkonstrukten, das defensive Programmieren, die Einhaltung der Clean-Code-Empfehlungen, für Testbarkeit des Programms zu sorgen und Robustheit zu schaffen, um nur einige Ansätze zu nennen. Zu all diesen wichtigen Punkten finden Sie nichts in diesem Buch, wir verweisen aber – wie auch bei den anderen Punkten – auf die entsprechende Literatur (siehe Anhang).

Wir setzen kein Vorgehensmodell der Softwareentwicklung voraus, da nach unserer Einschätzung Unit Tests durch die Entwickler in jedem Modell durchgeführt werden, auch wenn sie vom Modell her explizit gar nicht

vorgeschrieben werden. Agiles Vorgehen und die Auswirkungen auf den Test werden daher ebenfalls nicht diskutiert. Test Driven Development sehen wir, wie inzwischen viele andere Autoren, nicht als Test-, sondern als Designkonzept und gehen darauf nicht näher ein.

Wie Testrahmen aufzubauen sind, damit das Testobjekt – Ihr programmiertes Stück Software, was getestet werden soll – überhaupt mit Testeingabedaten versorgt und ausgeführt werden kann, wird nur indirekt durch die Verwendung entsprechender Frameworks beschrieben. Wir nutzen im Buch Google Test [URL: googletest]. Werkzeuge zur Fehlerverwaltung (»bugtracker«) werden ausgeklammert.

Da Entwicklertests direkt nach der Programmierung folgen, werden die weiteren Teststufen wie Integrationstest, Systemtest, Abnahmetest, Akzeptanztest, die anschließend durchgeführt werden, im Buch nicht behandelt. Damit finden Sie auch zu GUI-Tests, Usability-Tests, Performanztests und weiteren Tests, die eher den höheren Teststufen zuzuordnen sind, keine Informationen in diesem Buch. Testprozesse sowie deren Bewertung und Verbesserung gehören ebenfalls nicht zum Fokus des Entwicklertests.

Der Test von parallelen bzw. nebenläufigen Programmen erfordert weitere Ansätze, die hier auch nicht behandelt werden. Wir beschränken uns auf sequenzielle Programme.

2 Test gegen die Anforderungen

Um beim Testen entscheiden zu können, ob ein fehlerhaftes Verhalten vorliegt oder nicht, werden entsprechende Informationen benötigt. Diese Informationen sind in den Anforderungen oder in der Spezifikation zu finden. Getestet wird somit immer »gegen« ein vorab festzulegendes Verhalten oder Ergebnis des Testobjekts. Anforderungen oder Spezifikationen enthalten nur ganz selten alle zu berücksichtigenden Informationen. Die fehlenden Festlegungen sind vom Programmierer zu ergänzen oder durch Rückfrage beim Kunden zu klären, was die bessere Option ist. Domain-Fachwissen und gesunder Menschenverstand sind sicherlich recht hilfreich dabei.

Als Programmierer kann man auch folgende Meinung vertreten: »Alles, was nicht spezifiziert ist, gehört nicht zu meinen Aufgaben und brauche ich nicht zu berücksichtigen. Schließlich bezahlt mir niemand die Extra-Arbeit. Und jedes Programmverhalten bei Übergabe eines nicht spezifizierten Wertes – ob Absturz oder fehlerhafte Berechnung – ist o.k.« Eine solche Einstellung ist nur bei wirklich unkritischen Programmen, wie beispielsweise einem Spiel auf dem Handy, tolerierbar. In allen anderen Fällen muss überlegt werden, wie vom Programm aus auf mögliche, auch nicht spezifizierte Eingaben zu reagieren ist.

Anforderungen müssen in überschaubare Aufgaben aufgeteilt werden, um diese dann in Programmtext umzusetzen. Für jeden dieser Programmteile sind dann entsprechende Vor- und Nachbedingungen zu spezifizieren.

Betrachten wir folgende Anforderung an eine Funktion: Eine Prozentzahl, die als positiver ganzzahliger Wert übergeben wird, soll als Text (String) umgeformt und ausgegeben werden. Beispiel: Die Zahl 13 soll umgeformt werden zu »dreizehn«. Es handelt sich hier um eine relativ einfache Aufgabe, für die es auch entsprechende Bibliotheken gibt, aber darauf kommt es in unserem Beispiel nicht an. Wir möchten hier folgende Frage diskutieren:

- Wer ist dafür verantwortlich, dass nur ganze positive Werte übergeben werden?
- Wenn nur Werte zwischen 0 und 100 in Strings umgeformt werden sollen, wer prüft die Einhaltung des Wertebereichs?

- Der übergebene Wert muss nicht vom Typ des Parameters sein, und der Compiler akzeptiert den Wert, wenn er eine entsprechende Typumwandlung kennt. Eine Typumwandlung kann mit Informationsverlust verbunden sein. Wer überprüft den Parametertyp (wenn nicht der Compiler bereits Fehler meldet) und fängt fehlerhafte Übergaben mit einer aussagekräftigen Fehlermeldung ab?

Kann sich der Programmierer also auf die Einhaltung der Anforderungen verlassen? Kann er sich im Beispiel ausschließlich auf die Umsetzung der ganzzahligen Werte zwischen 0 und 100, denn andere Werte ergeben als Prozentzahlen keinen Sinn, konzentrieren und dann auch nur diese Werte beim Testen berücksichtigen?

Design by Contract

Eine sinnvolle und in der Praxis durchaus übliche Vorgehensweise zur Klärung des Problems ist das Prinzip »Design by Contract« [Meyer 13]. Es wird in einer Art Vertrag festgelegt, wofür der Aufrufer, der Dienstnehmer, verantwortlich ist und mit welchen Ergebnissen er vom Dienstanbieter nach dem Aufruf rechnen kann. Der Dienstanbieter kann eine größere Komponente, aber auch nur eine einfache Funktion sein. Vom Aufrufer sind die vereinbarten Vorbedingungen einzuhalten, der Dienstanbieter garantiert die Einhaltung der Nachbedingung. In unserem Beispiel wäre im Vertrag festzulegen, dass der Aufrufer garantiert, dass nur ganzzahlige Werte zwischen 0 und 100 (inklusive der beiden Werte) als Parameter übergeben werden. Der Dienstanbieter garantiert für diesen Fall, dass ein entsprechender String als Ergebnis zurückgegeben wird.

> Durch »Design by Contract« werden die Verantwortlichkeiten bei der Nutzung von Schnittstellen – auf beiden Seiten – festgelegt. Dies ist von Vorteil für den Test, da bei Vertragseinhaltung ein Lean-Testing-Ansatz ausreicht.

In der Praxis: Robustheit erwünscht

»Design by Contract« sagt nichts darüber aus, wie das Ergebnis aussieht, wenn die Vorbedingung *nicht* eingehalten wird. In der Praxis ist oft Robustheit erwünscht, d.h., dass Fehler möglichst abgefangen werden. Das bedeutet, dass die Vorbedingung geprüft wird, entweder vom Aufrufer oder vom Dienstanbieter. Vordergründig betrachtet ist die Konzentration der Prü-

fung an nur einer Stelle, dem Dienstanbieter, sinnvoll. Das ist aber nicht immer möglich. Dazu zwei Beispiele mit einfachen Funktionen:

1. Eine Funktion `double sqrt(double arg)` soll die Quadratwurzel der Zahl `arg` zurückgeben. Vorbedingung sei, dass `arg` nicht negativ ist. In diesem Fall kann die Vorbedingung in der Funktion leicht geprüft und bei einem negativen Argument eine Exception geworfen werden.
2. Eine Funktion `bool binary_search(Iterator first, Iterator last, const T& value)` soll zurückgeben, ob der Wert `value` im Bereich `[first, last)`[1] enthalten ist. `first` und `last` sind Iteratoren, die in einen Container oder ein Array verweisen. Die Anzahl der Elemente im zu durchsuchenden Bereich sei mit n = `last - first` abgekürzt. Wesentlicher Vorteil des bekannten Algorithmus zur binären Suche ist seine Schnelligkeit. Er benötigt nur etwa $\log_2 n$ Schritte. Die Vorbedingung ist jedoch, dass der Bereich zwischen `first` und `last` *sortiert* ist. Wenn diese Vorbedingung nicht erfüllt ist, ist das Ergebnis undefiniert. In diesem Fall wäre die Prüfung der Vorbedingung innerhalb der Funktion nur theoretisch möglich: Sie würde nämlich n Schritte erfordern, sodass der eigentliche Vorteil des Algorithmus, seine Schnelligkeit, dahin wäre und man den Bereich ebenso gut linear durchsuchen könnte. Die Prüfung der Vorbedingung ist in diesem Fall also nicht sinnvoll.

Auswirkungen auf den Test

Die im Buch vorgestellten Testverfahren berücksichtigen nicht, ob die Software nach »Design by Contract« strukturiert und aufgeteilt ist oder nicht. Die Verfahren unterstützen die systematische Herleitung von Testfällen. Im obigen Beispiel der Umformung einer Zahl zwischen 0 und 100 in einen Text verlangt ein systematischer Test auch die Prüfung mit ganzzahligen Werten kleiner als 0 und größer als 100. Auch ist zu prüfen, wie sich das Programm bei fehlerhaften Eingaben verhält (`double`, `float`, negative `int`-Werte statt `unsigned`...). Da bei »Design by Contract« der Aufrufer für die Einhaltung der Vorbedingung verantwortlich ist, muss der Test mit den »falschen« Werten an jeder Aufrufstelle durchgeführt werden. Der Test beim Dienstanbieter kann sich dann darauf beschränken, dass nur ganzzahlige Werte zwischen 0 und 100 an der Schnittstelle übergeben werden. Die Fehlerbehandlung bei falschen Werten obliegt somit dem Aufrufer.

»Design by Contract« vereinfacht den Test beim Dienstanbieter, verlagert aber die Prüfung der Einhaltung der Vorbedingung an jede Aufrufstelle. Hier sind gegebenenfalls die falschen Werte abzufangen und mit einer aussagekräftigen Fehlermeldung abzulehnen. Diese Aufteilung muss jedem

[1] `[)` ist ein halboffenes Intervall: `first` ist enthalten, `last` nicht.

Programmierer bewusst sein, um seine Testaktivitäten entsprechend zu fokussieren. Ohne die Vereinbarung von Vor- und Nachbedingungen wäre im Beispiel die Prüfung der Einhaltung der Spezifikation des Parameterwertes Aufgabe der Funktion selbst, hier wären dann alle Überprüfungen zu programmieren.

3 Statische Verfahren

Der Begriff Software umfasst vieles. So gehören sowohl die Design- als auch die Programmdokumentation dazu und natürlich auch der Programmcode. Nur dieser wird im Folgenden betrachtet. Statische Verfahren analysieren den Programmcode, ohne ihn auszuführen – daher der Name. Zu den statischen Verfahren gehören sowohl Reviews zur Bewertung des Programmcodes und zur Aufdeckung von Fehlern als auch die automatisierte Analyse mit Werkzeugen. Es gibt verschiedene Arten von Reviews, auf die unten kurz eingegangen wird. Allen Reviews ist gemeinsam, dass sie Arbeitszeit kosten, nicht nur die des Autors, sondern auch die von Kollegen, die als Gutachter tätig werden. Uns geht es darum, die Arbeitszeit aller Beteiligten zu reduzieren, um mit demselben Aufwand bessere Qualität zu erzielen. Deshalb liegt der Schwerpunkt dieses Kapitels nicht auf Reviews, die nur der Vollständigkeit halber erwähnt werden, sondern auf Verfahren zur statischen Analyse, die automatisiert ablaufen und daher wenig Arbeitszeit kosten. Reviews werden dadurch nicht überflüssig, aber weniger aufwendig, weil ein Teil der Fehler oder Schwächen schon vorher durch die statische Analyse aufgedeckt und anschließend korrigiert werden kann. Je früher ein Fehler gefunden wird, desto leichter (und billiger) ist seine Korrektur.

> Bevor es mit dem Testen unserer Software – also der Ausführung des Testobjekts mit Testdaten auf dem Rechner – losgeht, ist es sinnvoll, vorher so viele Fehler wie möglich zu entdecken, um den Aufwand für das Testen gering (»lean«) zu halten. Daher werden hier die statischen Analyseverfahren ausführlich vorgestellt.

Statische Verfahren werden typischerweise von Entwicklern eingesetzt. Diese Verfahren können natürlich nicht alle Fehler finden, insbesondere nicht diejenigen, die von externen Daten herrühren, die erst zur Laufzeit vom Programm eingelesen werden. Dafür gibt es die dynamischen Verfahren[1],

[1] Dynamische Verfahren werden auf dem Rechner ausgeführt; im Gegensatz zu den statischen Verfahren, bei denen das Testobjekt nicht ausgeführt, sondern durch Personen oder Werkzeuge analysiert wird.

insbesondere die Unit Tests auf der Ebene des Entwicklers. Umgekehrt können Unit Tests verschiedene statische Eigenschaften nicht prüfen, wie etwa die Einhaltung von Programmierrichtlinien. Statische Verfahren sind besonders geeignet zur Prüfung der folgenden Elemente:

- Syntax (Grammatik) des Programms. Bei einer nicht der Spezifikation der Programmiersprache entsprechenden Syntax gibt schon der Compiler eine Fehlermeldung aus. Voraussetzung ist natürlich, dass der Compiler die Sprachspezifikation richtig implementiert.
- Einhaltung anerkannter Prinzipien zur Softwareentwicklung bzw. Programmierung. So kann ein Verfahren das (im Allgemeinen unerwünschte) Vorhandensein globaler Variablen entdecken oder unbeabsichtigte fehlerhafte Typumwandlungen.
- Einhaltung der Programmierrichtlinien (sofern es welche gibt). Diese schreiben zum Beispiel vor, wie Namen von Klassen und Objekten zu bilden sind, wie Kommentare gestaltet werden sollen und ob Exceptions verwendet werden dürfen.
- Kontrollfluss. So können nicht erreichbarer Code oder fehlerhafte Ablaufstrukturen entdeckt werden.
- Datenfluss. Dabei wird zum Beispiel geprüft, ob eine Variable vor der Verwendung mit einem Wert initialisiert wurde oder ob ein zugewiesener Wert überhaupt verwendet wird. Wenn nicht, muss das kein Fehler sein, deutet aber daraufhin, dass der Programmcode nicht der eigentlichen Absicht entspricht.

Konkrete Beispiele finden Sie weiter unten.

Statische Verfahren können nicht nur auf den Quellcode angewendet werden, sondern auch auf Bytecode (wie etwa das Werkzeug FindBugs[2] für die Programmiersprache Java) oder auf binäre ausführbare Programme. Wir beschränken uns hier auf den Quellcode.

Beschränkungen in der Praxis

Ein Werkzeug ist ein Werkzeug, nicht mehr und nicht weniger. Insbesondere kann es nicht die Gedanken des Programmierers lesen und daraufhin die korrekte Umsetzung in Programmcode überprüfen. Das bedeutet, dass die Entwickler eines Werkzeugs bestimmte Vorstellungen haben, wie Anweisungen und Programmstrukturen aussehen sollen, und diese Annahmen im Werkzeug implementieren. Diese Annahmen können sich an manchen Stellen als falsch erweisen. Aus diesem Grund kann es sein, dass ein Werkzeug

[2] *http://findbugs.sourceforge.net/*

- einen »Fehler« meldet, der tatsächlich keiner ist (falsch positive Meldung), oder
- einen vorhandenen Fehler nicht meldet (falsch negative Meldung).

Ein besonders gründliches Werkzeug erzeugt möglicherweise eine Menge falsch positiver Meldungen, sodass die Gefahr besteht, das wichtige Meldungen in der schieren Menge untergehen. Die Entwickler solcher Werkzeuge bemühen sich, die Anzahl der falsch positiven Ergebnisse zu reduzieren.

Praktischer Einsatz

Um die Menge an Fehlermeldungen und Warnungen zu reduzieren und die Anzahl der Tests zu beschränken, empfiehlt sich die nachstehende Reihenfolge:

1. Das Testobjekt (Programmcode) compilieren und ggf. korrigieren, bis die Compilation fehlerlos durchläuft. Dabei die höchste Warnstufe einschalten und die Warnungen des Compilers berücksichtigen.
2. Statische Verfahren einsetzen und alle gefundenen Fehler korrigieren. Die Anzahl der jetzt noch notwendigen Tests wird durch jeden in dieser Phase gefundenen Fehler reduziert.
3. Erst dann die Unit Tests durchführen.

3.1 Codereview

Review ist der Oberbegriff für verschiedene statische Prüfverfahren, die von Personen durchgeführt werden. Das Prüfobjekt kann eine Designdokumentation sein, ein zu erstellendes Produkt oder ein Teil davon oder auch der Ablauf eines Prozesses. Ein Autor ist oft »betriebsblind« und sieht bestimmte Dinge nicht mehr, deshalb ist es wichtig, dass er den Code einem Kollegen zeigt. Ein großer Vorteil eines Reviews: Andere Personen sehen das Prüfobjekt unter einem ganz anderen Blickwinkel. In diesem Abschnitt geht es aber nur um Codereviews, die hauptsächlich in zwei Arten vorkommen:

Walkthrough

Das Vorgehen ist für kleine Teams von bis zu fünf Personen geeignet und verursacht relativ wenig Aufwand. Dabei stellt der Programmautor den Code einigen Experten vor, zum Beispiel fachlich versierten Kollegen – möglichst aus anderen Projekten – oder Testern. Mit ihnen zusammen werden verschiedene Benutzungsabläufe durchgespielt. Ziel ist das gegenseitige Lernen und Verständnis über das Prüfobjekt und natürlich, Fehler zu finden.

Inspektion

Eine Inspektion folgt einem formalisierten Ablauf, in dem es verschiedene Rollen gibt, wie etwa einen Moderator und verschiedenen Gutachter (ebenfalls fachlich versierte Kollegen). Einer der Gutachter trägt den Inhalt des Prüfobjekts vor, wobei die anderen entsprechend Fragen stellen. Die Gutachter bereiten sich auf die Sitzung vor, was beim Walkthrough entfallen kann. Während der Inspektion werden auch Daten gesammelt, die zur Qualitätsbeurteilung des Entwicklungs- und Inspektionsprozesses herangezogen werden. Ziel der Inspektion ist das Finden von Fehlern und ggf. deren Ursachen.

> Codereviews tragen sehr zur Qualität einer Software bei. Deswegen werden sie hier als statisches Verfahren erwähnt, auch wenn sie nicht weiter behandelt werden.

Ausführliche Beschreibungen zu den unterschiedlichen Arten der Reviews sind in [Spillner & Linz 12, Kap. 4] und in [Rösler et al. 13] zu finden.

3.2 Compiler

Ein Compiler muss zwangsläufig ein Programm analysieren, wenn er lauffähigen Code daraus herstellen soll. Typischerweise baut er einen abstrakten Syntaxbaum auf, abgekürzt AST (abstract syntax tree) genannt. Im Prinzip hat ein Compiler alle im Programmcode liegenden Informationen zur Verfügung, sodass er einige Fehler leicht entdecken kann, wie etwa die fehlende Deklaration einer Variablen oder den Aufruf einer Funktion mit einem falschen Parametertyp.

Was ein Compiler aber nicht oder nur zum Teil kennt, sind Empfehlungen für einen defensiven Programmierstil oder die gewünschten Programmierrichtlinien.

Es gibt eine Menge verschiedener Compiler, sowohl Open Source als auch kommerziell. Die hier und im Folgenden beschriebene Problematik ist bei allen Compilern strukturell ähnlich, weswegen wir uns auf die Open-Source-Compiler GNU C++ (G++) und Clang (Aufruf clang++) beschränken.

G++

G++ gibt es für Mac OS und Linux [URL: gcc]. Die Distribution für Mac OS hinkt bezüglich der Versionsnummern gegenüber Linux etwas hinterher. Unsere Empfehlung: Auf dem Mac lieber Clang verwenden (siehe unten). In

beiden Fällen wird die mehrere Gigabytes umfassende Entwicklungsumgebung Xcode benötigt.

G++ gibt es auch für Windows, wobei die MinGW- und die Cygwin-Distribution am weitesten verbreitet sind. Aber auch hier hinken die Versionsnummern denen für Linux hinterher.

Clang

Clang [URL: clang] ist ein Frontend für die C-Sprachfamilie (C/C++, Objective C/C++), das für LLVM[3], eine modulare Compilerarchitektur, entwickelt wurde. Clang/LLVM ist Open Source. LLVM ist aus einem Projekt der Compilerforschung entstanden. Clang/LLVM ist besser als G++ für die statische Analyse eines Programms geeignet und hat im Allgemeinen auch eine bessere Performanz. Clang ist als Ersatz für G++ konzipiert und akzeptiert daher dieselben und ähnliche Optionen. Clang wurde in Xcode integriert. Für Mac OS und Linux ist Clang sehr gut geeignet.

Windows wird ebenfalls unterstützt, allerdings beschränkt auf die Integration in Visual C++. Ein problemloses Zusammenwirken mit MinGW oder Cygwin ist derzeit nicht in Sicht (Clang nutzt Teile von Visual C++ bzw. MingW, arbeitet also nicht alleinstehend).

Warnungen des Compilers einschalten

Um die meisten Warnungen der genannten Compiler einzuschalten, wird die Option `-Wall` bei der Übersetzung übergeben. Aber trotz des Namens `-Wall` werden nicht alle Warnungen ausgegeben, mit `-Wextra` gibt es weitere Überprüfungen. Zusätzlich hilft die Option `-pedantic`, die compilerspezifische Erweiterungen meldet, die nicht dem C++-Sprachstandard entsprechen. `-Wall` und `-Wextra` umfassen jeweils viele Optionen. Bezüglich ihrer Beschreibung verweisen wir auf die Dokumentation zum Compiler. Clang stellt weitere Warn-Optionen bereit, die den Compiler als Analysewerkzeug nutzbar machen; mehr dazu erfahren Sie weiter unten. Sie müssen bei der Prüfung nicht das ausführbare Programm erzeugen lassen: Mit der Option `-c` wird das Linken weggelassen. Geben Sie allerdings nicht `-fsyntax-only` ein: Damit wird nur die Syntax geprüft und sonst nichts, d.h., es werden keine Templates instanziiert und daher auch nicht geprüft.

Nachstehend finden Sie einige Beispiele für (wahrscheinliche) Fehler, die ohne die genannten Optionen *nicht* gemeldet werden (verwendet wurden g++ 5.2 und Clang 3.8).

Im ersten Beispiel wird die Variable `b` der Variablen `a` zugewiesen, die dann als Bedingungsausdruck interpretiert wird:

[3] Früher: Low Level Virtual Machine, *http://llvm.org/*.

```
if( a = b ) {
   // ...
}
```

Listing 3.1: Vermutlich fehlerhafter Vergleich

Dies ist legales C++. Vermutlich handelt es sich jedoch um einen Schreibfehler, der durch `-Wall` aufgedeckt wird. Die Warnung kann abgeschaltet werden, wenn es sich tatsächlich um eine Zuweisung und nicht um den Vergleich `(a == b)` handeln soll, indem ein weiteres Klammerpaar spendiert wird: `if((a = b))`. In unseren Augen ist dies allerdings schlechter Programmierstil: Besser ist es, Zuweisung und Vergleich in zwei Anweisungen zu zerlegen.

Leicht zu übersehen ist das überflüssige Semikolon in folgender Anweisung:

```
if ( a==b );
{
   // ...
}
```

Listing 3.2: Wirkungslose `if`-Abfrage

Es führt dazu, dass der Vergleich keine Wirkung hat, der Codeblock wird in jedem Fall ausgeführt. Dieser Fehler wird mit `-Wall` nicht entdeckt, aber mit `-Wextra`.

In der folgenden Anweisung ist eine Zahl versehentlich als `int` statt als `double` deklariert worden:

```
int z = 3.14;
```

Listing 3.3: Informationsverlust ohne Fehlermeldung

Diesen Fehler findet g++ trotz der eingeschalteten Optionen nicht. Clang meldet eine Warnung auch ohne die genannten Optionen. Guter Programmierstil ist es, solche Initialisierungen mit geschweiften Klammern zu schreiben:

```
int z {3.14};
```

Listing 3.4: Informationsverlust mit Fehlermeldung

Dann wird von jedem Compiler auch ohne Optionen als Fehler gemeldet, dass der Typ auf der rechten Seite auf den auf der linken Seite verengt wird. Das Umgekehrte (`double z {3};`) hingegen ist legal, weil kein Informationsverlust stattfindet.

Beim nächsten Beispiel ist ein Buffer-Overflow möglich:

```
void f(int a) {
  char buf[] = "Buffer-Overflow?";
  char c = buf[a];
  // ...
}
```

Listing 3.5: Möglicher Buffer-Overflow

Dieser potenzielle Fehler wird ohne die genannten Optionen nicht gemeldet. Allerdings wird nicht vor jedem Buffer-Overflow gewarnt: Die folgenden Zeilen werden klaglos übersetzt.

```
const char* name = "Georg Christoph Lichtenberg";
char buf[10];
std::strcpy(buf, name);
```

Listing 3.6: Buffer-Overflow

Also doch `std::strncpy()` oder noch besser `std::string` verwenden.

> Die wenigen Beispiele zeigen, dass es sich lohnt, die Warnungen des Compilers einzuschalten. Mancher Tipp- oder Flüchtigkeitsfehler wird schon damit gefunden.

Aber es gibt noch bessere Möglichkeiten, wie der nächste Abschnitt zeigt.

3.3 Analysewerkzeuge

Werkzeuge zur statischen Analyse verarbeiten den Programmcode ähnlich wie ein Compiler. Sie erkennen Muster und bauen geeignete Datenstrukturen auf, wie etwa den oben erwähnten abstrakten Syntaxbaum. Variablen und ihre Typen werden in einer Symboltabelle gespeichert. Letztlich sind es dieselben Informationen, die ein Werkzeug zur statischen Analyse und ein Compiler benötigen. Da liegt es nahe, beide Eigenschaften zu verknüpfen. Ein moderner Compiler liefert daher Informationen, die zum Beispiel von einer Entwicklungsumgebung genutzt werden, um mögliche Fehler anzuzeigen. Im Clang-Projekt ist der Clang Static Analyzer von vornherein ein fester Bestandteil, auch wenn er allein aufgerufen werden kann. Trotz der Gemeinsamkeiten sind die Compilation und die statische Analyse nicht dasselbe. Die statische Analyse versucht, Fehler zu finden, und muss zu diesem Zweck eine manchmal sehr tief gehende Analyse durchführen. Dabei spielt

das eingebaute Wissen über mögliche Fehler eine Rolle. Die statische Analyse dauert daher normalerweise deutlich länger als die Compilation eines Programms.

Werkzeugauswahl

Da es hier nur um die wesentlichen Eigenschaften von Werkzeugen geht und nicht um eine vergleichende Betrachtung, beschränken wir uns bei der folgenden Liste auf eine kleine (relativ willkürliche) Auswahl:

- *lint* ist der Klassiker. Es ist ein Programm, das zur Analyse von C-Programmen entwickelt wurde und dessen Name sich in heutigen Produkten wiederfindet. Das englische Wort »lint« bedeutet Fussel oder Fluse, die für die möglichen Fehler im zu analysierenden Programm stehen.
- *PC-Lint* (kommerziell) von Gimpel Software bietet die Möglichkeit, Code in ein Fenster zu kopieren und auf Fehler untersuchen zu lassen[4].
- *flint* (Open Source) steht für »Facebook lint«[5]. Es wurde bei Facebook von dem bekannten C++-Autor Andrei Alexandrescu entwickelt und wird bei Facebook eingesetzt.
- *cppcheck*[6] (Open Source) ist in einigen Entwicklungsumgebungen integriert, kann aber auch allein aufgerufen werden. Es hat den Anspruch, nur wirkliche Fehler zu finden, also möglichst keine falsch positiven Meldungen zu erzeugen.
- *coverity* ist ein kommerzieller Satz von Werkzeugen[7] (Preise nicht veröffentlicht). Einige Mitarbeiter von Coverity haben einen sehr lesenswerten Artikel über die Probleme der statischen Analyse in der realen Welt verfasst [URL: Bessey et al. 10]. Dabei geht es nur am Rande um das Produkt der Firma. Der Schwerpunkt liegt auf dem, was die Autoren alles mit Entwicklern, Managern und deren Erwartungen und Geisteshaltungen erlebt haben.
- *Visual Studio 2013*[8] von Microsoft (kommerziell; Express-Version ist kostenlos) ist eine Entwicklungsumgebung und enthält ein integriertes Tool zur statischen Analyse.
- *clang* ist der schon erwähnte Compiler, der mit erweiterten Optionen auch als Programm zur statischen Analyse verwendet werden kann. Beispiele finden Sie unten.
- *scan-build* ist ein Programm zur statischen Analyse, dem der zu benutzende Compiler und das zu benutzende Analyseprogramm mitgegeben

[4] *http://www.gimpel-online.com/OnlineTesting.html*
[5] *https://github.com/facebook/flint*
[6] *http://cppcheck.sourceforge.net/*
[7] *https://www.coverity.com/*
[8] *https://www.visualstudio.com/*

werden kann. Es gehört zu *clang* und übernimmt zusätzliche Aufgaben. So kann z.B. neben verschiedenen Optionen der Befehl *make* übergeben werden. Das bewirkt, dass *scan-build* das Makefile liest und das gesamte dazugehörige Projekt analysiert.

3.4 Analysebeispiele

Im Folgenden zeigen wir einige Beispiele für die Wirkung der statischen Analyse, wobei wir uns auf *clang* und *scan-build* beschränken. Das Prinzip ist bei anderen Werkzeugen ähnlich.

3.4.1 Clang als Analysewerkzeug

Clang kann mit der Option `-Weverything` aufgerufen werden, die alle verfügbaren Warnungen einschaltet. Damit werden etliche weitere mögliche Schwächen eines zu untersuchenden Programms gefunden. Allerdings werden auch unerwünschte Meldungen erzeugt.

Ein Beispiel: In der folgenden Zeile wird die Variable `a` mit einer binären Konstanten initialisiert.

```
int a {0B1111};                                    // C++14
```

Listing 3.7: Nur mit C++14 kompatibel

Das Präfix `0B` gibt es erst seit C++14, sodass zur Compilation die Option `-std=c++14` verwendet werden muss. Die Initialisierung mit geschweiften Klammern gibt es erst seit C++11. Mit `-Weverything` wird gemeldet, dass die Zeile nicht zu den C++-Versionen vor C++14 kompatibel ist, was auf die in `-Weverything` enthaltenen Optionen `-Wc++98-compat` und `-Wc++11-compat-pedantic` zurückzuführen ist. Diese müssen explizit ausgeschaltet werden.

Ähnliches gilt auch für andere Optionen, deren Ergebnis Sie nicht sehen möchten. Für jede unerwünschte Option ist dabei nach dem `-W` ein `no-` einzufügen, wenn `-Weverything` verwendet wird. Wenn also C++14-Programme analysiert werden sollen, empfiehlt es sich, beim Aufruf von `clang++` die folgende Kombination zu den Optionen hinzuzufügen:

```
-std=c++14 -Weverything -Wno-c++98-compat -Wno-c++11-compat-pedantic
```

Der Memory-Leak-Detektor ist aus dem statischen Analyseteil von Clang entfernt worden, weil die Tools *AddressSanitizer* (kurz: Asan, integriert in Clang und GNU C++) und *Valgrind*[9], die allerdings dynamisch arbeiten, bei Weitem mehr Fehler finden.

[9] *http://valgrind.org/*

Der statische Analysator von Clang kann mit der Option `--analyze` direkt aufgerufen werden. Es wird dann kein ausführbares Programm erzeugt.

```
const char* name = "Georg Christoph Lichtenberg";
char buf[10];
std::strcpy(buf, name);
```

Listing 3.8: Buffer-Overflow

Wenn das obige Programmsegment in einer Datei *test.cpp* vorliegt, zeigt der Aufruf `clang++ --analyze test.cpp` den Buffer-Overflow mit der folgenden Meldung an:

```
test.cpp:9:3 warning: String copy function overflows destination buffer
  std::strcpy(buf, name);
   ^~~~~~~~~~~~~~~~~~~~~
```

Gleichzeitig wird eine XML-Datei mit dem Ergebnis erzeugt.

Ein weiteres Beispiel:

```
void test(int *p) {
  if (p)                          // sollte if (!p) heißen ...
    return;
  int x = p[0];
  // ...
}
```

Listing 3.9: Fehlerhafte Zeigerabfrage

Hier wird abgesehen von der nicht weiter verwendeten Variablen `x` herausgefunden, dass ein Null-Zeiger dereferenziert wird. Der statische Analysator von Clang findet eine Menge möglicher Fehler, die hier nicht alle genannt werden können. Daher folgt nur noch ein weiteres Beispiel.

Weiter oben haben wir Datenflussanomalien erwähnt. Man unterscheidet dabei drei Arten, die sich in der folgenden fehlerhaften Funktion in konzentrierter Form zeigen und die direkt danach erläutert werden:

```
void tausch(int& min, int& max) {       // nicht benutzen!
  int hilf;
  if (min > max) {
    max = hilf;                         // ur-Anomalie
    max = min;                          // dd-Anomalie
    hilf = min;                         // du-Anomalie
  }
}
```

Listing 3.10: Fehlerhafte Tauschfunktion

- *ur-Anomalie*: Ein undefinierter Wert (*u*) wird auf einem Programmpfad gelesen (*r* für »read«). Im Beispiel wird `hilf` lesend verwendet, ohne dass vorher der Variablen ein Wert zugewiesen wurde. Dieser Fehler wird schon mithilfe der Option `-Wall` entdeckt, also auch mit `--analyze`.
- *du-Anomalie*: Die Variable erhält einen Wert, sie ist also definiert (*d*), der Wert wird aber nicht verwendet. Im Beispiel wird `hilf` ein Wert zugewiesen, ohne dass dieser verwendet wird. `clang++ --analyze` zeigt den möglichen Fehler an. Wenn die Funktion als Template geschrieben wird, gibt es nur eine Meldung, wenn das Template instanziiert wird. Clang (Version 3.8) ist nicht perfekt: Im Template wird dann nur die *ur*-Anomalie gefunden, nicht die *du*-Anomalie. PC-Lint findet auch im Template die *ur*- und die *du*-Anomalie.
- *dd-Anomalie*: Die definierte Variable erhält ein zweites Mal einen Wert (*d*), ohne dass der erste Wert (*d*) verwendet wurde. Im Beispiel trifft dies auf die Variable `max` zu. Diese Anomalie wird von Clang nicht gefunden.

PC-Lint liefert immerhin den Hinweis, dass `min` in der Parameterliste vom Typ `const int&` sein könnte (weil `min` nicht geändert wird). Das kann natürlich in einer Funktion, die eventuell `min` mit `max` vertauschen soll, nicht sein. Deswegen ist die Warnung ein indirekter Hinweis auf einen Fehler.

Die letzten beiden Anomalien müssen keinen Fehler darstellen. Sie deuten jedoch darauf hin, dass an diesen Stellen vielleicht nicht richtig überlegt wurde. Weil zwischen den Anweisungen noch viele andere stehen könnten, ist das Erkennen solcher Anomalien durch Lesen des Programmcodes möglicherweise erschwert. Eine automatisierte Analyse ist einfacher.

3.4.2 Scan-Build

scan-build kann nicht nur mit Clang, sondern auch mit G++ und einem anderen Analysierer arbeiten. Wir haben es nicht ausprobiert, sondern empfehlen Clang in beiden Fällen, sodass *scan-build* die folgenden Optionen übergeben werden:

```
--use-c++=/usr/local/bin/clang++\
--use-analyzer=/usr/local/bin/clang++\
-enable-checker core,cplusplus,deadcode,security\
-o /home/user/tmp/scanergebnis
```

Die Option `-enable-checker` wählt die zu verwendenden Prüfmodule, Checker genannt, aus. Die Option `-o` gibt das Verzeichnis an, in welches das Ergebnis im HTML-Format ausgegeben werden soll. Der Aufruf

```
scan-build make
```

in einem Verzeichnis analysiert ein Projekt, basierend auf dem Makefile. Wenn zum Beispiel die etwas unsichere Funktion `strcpy()` verwendet wird, gibt es einen Warnhinweis, auch wenn kein Buffer-Overflow entdeckt wurde:

> test.cpp:7:3: warning: Call to function 'strcpy' is insecure as it does not provide bounding of the memory buffer. Replace unbounded copy functions with analogous functions that support length arguments such as 'strlcpy'. CWE-119

Ohne den Checker `security` würde diese Warnung entfallen. Die Abkürzung CWE (Common Weakness Enumeration) in der im Warnhinweis angegebenen Fehlernummer CWE-119 bezieht sich auf die Klassifikation der Mitre-Corporation[10]. Die empfohlene Funktion `strlcpy()` ist allerdings nicht im C++-Standard enthalten und somit nicht portabel.

Bei der Analyse wird *make* tatsächlich ausgeführt. Wenn *make* nichts zu tun hat, weil alle zu erzeugenden Dateien aktuell sind, gibt auch *scan-build* nichts aus. In so einem Fall geben Sie einfach vorher `make clean` ein.

So viel zur statischen Analyse.

> Wir empfehlen, Werkzeuge zur statischen Analyse unbedingt und auch vor dem eigentlichen Testen zu nutzen, um schon vorab mögliche Fehler erkennen und eliminieren zu können.

[10] *http://cwe.mitre.org/*

4 Testentwurf und Testdurchführung

In diesem Kapitel wird das »eigentliche« Testen beschrieben, also wie auf systematische Art und Weise Überlegungen anzustellen sind, um Testfälle zu spezifizieren. Die Testfälle sind dann auf dem Rechner auszuführen, um festzustellen, ob das Testobjekt sich wie erwartet verhält oder ob Fehler gefunden werden. Aber Vorsicht, ein Fehler könnte auch darin begründet sein, dass der Testfall fehlerhaft ist! Daher ist es durchaus sinnvoll, auch die Testfälle einer Qualitätskontrolle – z.B. einem Review – zu unterziehen.

In aller Regel wird beim Entwicklertest das Testobjekt kein ausführbares Programm sein, daher muss ein Testrahmen erstellt werden. Die Hauptaufgabe des Testrahmens ist der Aufruf des Testobjekts inklusive der Übergabe der Testdaten. Inzwischen gibt es eine ganze Reihe von solchen Testrahmen, auch als Test-Frameworks bezeichnet, für nahezu alle Programmiersprachen. Die Frameworks bieten dabei meist noch weitere nützliche Funktionalität an, wie beispielsweise der Vergleich von erwarteten und tatsächlichen Ergebnissen bei jedem Testlauf.

Wir haben uns für das Google-Test-Framework entschieden, für das in den folgenden Abschnitten die Installation und Anwendung beschrieben werden. Sollten Sie bereits ein Test-Framework verwenden und damit zufrieden sein, können Sie diese Information ignorieren. Alle C++-Beispiele im Buch und auf der Internetseite zum Buch [URL: leantesting] wurden zwar mit Google Test erstellt, die Übertragung auf ein anderes Framework ist jedoch unproblematisch.

4.1 Das Google-Test-Framework

Das Google-Test-Framework [URL: googletest] erlaubt auf einfache Weise, Tests in C++ für eine ganze Reihe verschiedener Plattformen zu schreiben. Es gibt auch eine Benutzungsoberfläche für Google Test, die wir in diesem Buch der Einfachheit halber nicht verwenden.[1]

[1] Eine Einführung und wichtige Prüfmakros finden Sie auf *https://code.google.com/p/googletest/wiki/Primer*. Wenn Sie am Ende nicht */Primer*, sondern */AdvancedGuide* schreiben, erhalten Sie weitere Details.

4.1.1 Installation

Laden Sie sich Google Test herunter und entpacken Sie die zip-Datei an geeigneter Stelle, zum Beispiel in einem lokalen Benutzerverzeichnis für Programme, etwa */home/user/programme*. Das entstehende Verzeichnis *gtest-1.7.0* sollten Sie in *gtest* umbenennen, damit die Verwendung des Verzeichnisses frei von Versionsnummern bleibt. Die folgenden Hinweise gelten für ein Unix-artiges System mit dem GNU-C++-Compiler. In der Datei README finden Sie Hinweise auch für andere Systeme. Öffnen Sie eine Konsole (Terminal) in dem Verzeichnis *gtest* und tippen Sie Folgendes ein:

```
./configure
make
```

Fertig ist die Installation! Im Unterverzeichnis *make* finden Sie ein *Makefile*, das Sie abwandeln und für eigene Zwecke verwenden können. Das folgende Listing zeigt ein so abgewandeltes Makefile. Weil die konkreten Namen ersetzt worden sind, eignet es sich für eine Vielzahl kleiner Projekte. Das normalerweise unsichtbare Tabulatorzeichen wird nachfolgend mit dem Symbol ⟶| gekennzeichnet.

```
1 CXX := g++
2 CXXFLAGS+= -Wall -Wextra -pthread -std=c++14 -pedantic -fmax-errors=1
3 # A sample Makefile for building Google Test and using it in user
4 # tests.  Please tweak it to suit your environment and project.  You
5 # may want to move it to your project's root directory.
6 # SYNOPSIS:
7 #   make [all]  - makes everything.
8 #   make TARGET - makes the given target.
9 #   make clean  - removes all files generated by make.
10 # Please tweak the following variable definitions as needed by your
11 # project, except GTEST_HEADERS, which you can use in your own targets
12 # but shouldn't modify.
13
14 # Points to the root of Google Test, relative to where this file is.
15 # Remember to tweak this if you move this file.
16 GTEST_DIR = /home/users/programme/gtest
17 TEST = dertest
18
19 # Flags passed to the preprocessor.
20 # Set Google Test's header directory as a system directory, such that
21 # the compiler doesn't generate warnings in Google Test headers.
22 CPPFLAGS += -isystem $(GTEST_DIR)/include
23
24 # All Google Test headers.  Usually you shouldn't change this
```

```
25 # definition.
26 GTEST_HEADERS = $(GTEST_DIR)/include/gtest/*.h \
27                 $(GTEST_DIR)/include/gtest/internal/*.h
28 GTEST = $(GTEST_DIR)/lib/gtest
29 GTESTALL = $(GTEST_DIR)/lib/gtest-all
30 GTESTMAIN = $(GTEST_DIR)/lib/gtest_main
31 # Usually you shouldn't tweak such internal variables, indicated by a
32 # trailing _.
33 GTEST_SRCS_ = $(GTEST_DIR)/src/*.cc $(GTEST_DIR)/src/*.h \
34               $(GTEST_HEADERS)
35 SOURCES := $(wildcard *.cpp)
36 HEADERS := $(wildcard *.h)
37 OBJS := $(patsubst %.cpp,%.o,$(wildcard *.cpp))
38 TEMPLATES := $(wildcard *.t)
39 # House-keeping build targets
40 all : $(TEST)
41
42 clean :
43 ⟶|rm -f $(TEST) *.o
44
45 cleanall :
46 ⟶|rm -f $(TEST) *.o $(GTEST_DIR)/lib/*
47
48 # Builds gtest.a and gtest_main.a.
49 # For simplicity and to avoid depending on Google Test's
50 # implementation details, the dependencies specified below are
51 # conservative and not optimized.  This is fine as Google Test
52 # compiles fast and for ordinary users its source rarely changes.
53 $(GTESTALL).o : $(GTEST_SRCS_)
54 ⟶|$(CXX) $(CPPFLAGS) -I$(GTEST_DIR) $(CXXFLAGS) -c \
55             $(GTEST_DIR)/src/gtest-all.cc -o $(GTESTALL).o
56
57 $(GTESTMAIN).o : $(GTEST_SRCS_)
58 ⟶|$(CXX) $(CPPFLAGS) -I$(GTEST_DIR) $(CXXFLAGS) -c \
59             $(GTEST_DIR)/src/gtest_main.cc -o $(GTESTMAIN).o
60
61 $(GTESTALL).a : $(GTESTALL).o
62 ⟶|$(AR) $(ARFLAGS) $@ $^
63
64 $(GTESTMAIN).a : $(GTESTALL).o $(GTESTMAIN).o
65 ⟶|$(AR) $(ARFLAGS) $@ $^
66
67 # Builds a sample test.  A test should link with either gtest.a or
```

```
68 # gtest_main.a, depending on whether it defines its own main()
69 # function.
70 $(TEST) : $(OBJS) $(GTEST_DIR)/lib/gtest_main.a
71 ⟶|-$(CXX) -lpthread $^ -o $(TEST)
72
73 $(OBJS): $(SOURCES) $(HEADERS) $(TEMPLATES) $(GTEST_HEADERS)
74 ⟶|-$(CXX) $(CPPFLAGS) $(CXXFLAGS) $(INCLUDE) -c $(SOURCES)
```

Listing 4.1: Makefile für Google Test

Dieses Makefile unterscheidet sich wie folgt von *gtest/make/Makefile*: Die Dateinamen wurden durch symbolische Namen ersetzt, etwa `gtest-all.o` durch `$(GTESTALL).o`. Die symbolischen Platzhalter für die vorherigen konkreten Dateinamen (im Original `sample1.cc` usw.) werden in den Zeilen 35-38 ermittelt. Dadurch ist das Makefile für beliebige andere Dateinamen geeignet. In Zeile 16 wird das Verzeichnis, in dem sich Google Test befindet, eingetragen. Zeile 17 definiert den Namen des zu erzeugenden ausführbaren Programms.[2]

4.1.2 Anwendung

Die Anwendung ist denkbar einfach. Um das zu zeigen, definieren wir eine einfache Klasse mit nur einem Attribut und einer zugegebenermaßen relativ sinnfreien Funktion, die das Quadrat des Attributs zurückgibt.

```
#ifndef PRUEFLING_H
#define PRUEFLING_H

class pruefling {
public:
  pruefling(double attribut);
  double attributquadrat() const;
private:
  double attribut;
};

#endif
```

Listing 4.2: Deklaration der Klasse `pruefling` *(beispiele/gtest/pruefling.h)*

Anmerkung: Wenn in der Listing-Unterschrift ein Dateiname in runden Klammern angegeben wird, ist die betreffende Datei in den von der Internetseite *http://www.leantesting.de* downloadbaren Beispielen enthalten.

[2] Das Makefile und alle Beispiele können von der Seite *http://www.leantesting.de* heruntergeladen werden. Eine ausführliche Einführung in die effiziente Programmerzeugung mit Makefiles finden Sie zum Beispiel in [Breymann 15, Kap. 18].

```
#include "pruefling.h"

pruefling::pruefling(double attribut_)
  : attribut(attribut_) {
}

double pruefling::attributquadrat() const {
  return attribut*attribut;
}
```

Listing 4.3: Implementierung der Klasse `pruefling` *(beispiele/gtest/pruefling.cpp)*

Die folgende Datei enthält einige einfache Tests. Ein `main()`-Programm ist nicht notwendig; es wird vom Framework bereitgestellt. Wichtig ist die erste Zeile, die *gtest.h* inkludiert. Die Include-Anweisung kann mit spitzen Klammern statt Anführungszeichen geschrieben werden, weil das Google-Test-Verzeichnis wie ein Systemverzeichnis behandelt wird. Maßgeblich dafür ist die Option `-isystem` in Zeile 22 des ab Seite 26 abgedruckten Makefiles. Das Makro `EXPECT_DOUBLE_EQ(par1, par2)` erwartet, dass die Parameter `par1` und `par2` nahezu gleich sind (mehr dazu folgt unten). Wenn das nicht der Fall ist, gilt der Test als gescheitert.

```
#include <gtest/gtest.h>
#include "pruefling.h"

TEST(Pruefling_Test, 1_null) {
  pruefling p(0.0);
  EXPECT_DOUBLE_EQ(.0, p.attributquadrat());
}
TEST(Pruefling_Test, 2_eins) {
  pruefling p(1.0);
  EXPECT_DOUBLE_EQ(1.0, p.attributquadrat());
}
TEST(Pruefling_Test, 3a_positiveZahl) {
  pruefling p(2.5);
  EXPECT_DOUBLE_EQ(6.25, p.attributquadrat());
}
TEST(Pruefling_Test, 3b_positiveZahl) {
  pruefling p(10.0);
  EXPECT_DOUBLE_EQ(1000.0, p.attributquadrat());
}

TEST(Pruefling_Test, 4_ULP_test) {
  pruefling p(2.5);
  EXPECT_DOUBLE_EQ(6.25 + 1e-15, p.attributquadrat());
```

```
  if(6.25 + 1e-15 != 6.25) {
    std::cout << "UNGLEICH!\n";
  }
}

// .. usw.
```

Listing 4.4: Die Testfälle *(beispiele/gtest/dertest.cpp)*

Alle drei gezeigten Dateien sowie das Makefile sollen in einem Verzeichnis liegen, das keine anderen Dateien enthält. Das ausführbare Testprogramm *dertest* wird erzeugt, indem in einer in diesem Verzeichnis geöffneten Konsole (Terminal) der Befehl `make` eingegeben wird.

Der Aufruf von *dertest* startet das Programm. Die ersten drei Tests sind erfolgreich, der letzte schlägt fehl. Listing 4.5 zeigt die Ausgabe des Programms.

```
Running main() from gtest_main.cc
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from Pruefling_Test
[ RUN      ] Pruefling_Test.1_null
[       OK ] Pruefling_Test.1_null (0 ms)
[ RUN      ] Pruefling_Test.2_eins
[       OK ] Pruefling_Test.2_eins (0 ms)
[ RUN      ] Pruefling_Test.3a_positiveZahl
[       OK ] Pruefling_Test.3a_positiveZahl (0 ms)
[ RUN      ] Pruefling_Test.3b_positiveZahl
dertest.cpp:18: Failure
Value of: p.attributquadrat()
  Actual: 100
Expected: 1000.0
Which is: 1000
[  FAILED  ] Pruefling_Test.3b_positiveZahl (0 ms)
[----------] 4 tests from Pruefling_Test (1 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Pruefling_Test.3b_positiveZahl

 1 FAILED TEST
```

Listing 4.5: Ausgabe des Programms `dertest`

Beim Vergleich zweier `double`-Zahlen wie im Test 3a gibt es eine Besonderheit: Wenn sich zwei `double`-Zahlen auch nur im niedrigstwertigen Bit unterscheiden, gelten sie normalerweis als ungleich. Um den Vergleich von `double`-Zahlen, die gleich sein müssten, aber auf verschiedene Weise berechnet wurden, nicht häufig scheitern zu lassen, ist das Google-Test-Makro `EXPECT_DOUBLE_EQ(...)` mit »fast gleichen« Zahlen zufrieden. Das bedeutet, dass die Zahlen innerhalb von vier ULPs (*unit in the last place*) liegen. Dabei ist ein ULP definiert als der Abstand einer Ergebniszahl zur nächsten, direkt benachbarten Gleitkommazahl (`double` oder `float`).

Makros, die mit `EXPECT` beginnen, lassen den Test auch bei Fehlschlägen weiterlaufen. Wenn das nicht sinnvoll ist, wird stattdessen `ASSERT` verwendet. Ein fehlgeschlagener Test führt dann zum Abbruch des gesamten Tests. Bezogen auf das obige Programm wäre dann etwa `ASSERT_DOUBLE_EQ(6.25, p.attributquadrat());` zu schreiben.

Weitere Informationen zu den Google-Test-Makros finden Sie auf den Internetseiten, die am Anfang des Abschnitts angegeben sind.

> Den Entwicklertest ohne ein Test-Framework durchzuführen, ist wenig sinnvoll, da das Framework eine große Unterstützung bietet. Das Argument *»für die paar Testfälle lohnt sich ein Framework nicht«* zählt nicht. Sie werden bei der Lektüre des Buchs merken, dass immer noch ein paar Testfälle hinzukommen. Wir wollen schließlich ausreichend genug testen, bevor wir unsere Software freigeben.

4.2 Happy-Path-Test

Vorbemerkung: »Pfad« (engl. *path*) heißt der Weg von der ersten ausgeführten Anweisung einer Komponente bis zur letzten. Typischerweise werden nicht alle möglichen Pfade durchlaufen. So wird bei einer einmaligen Ausführung einer `if-else`-Anweisung des Programms nur einer der beiden möglichen Pfade der Anweisung durchlaufen.

Mit Happy Path wird ein typisches vorgegebenes Szenario bezeichnet, das vom Testobjekt problemlos verarbeitet wird und ein erwartetes Ergebnis erzeugt, ganz ohne Fehlermeldungen und Ausnahmefälle. Für dieses Testvorgehen gibt es eine Reihe von unterschiedlichen Namen. Zum Beispiel wird beim Smoke-Test das gleiche Konzept zur Testfallermittlung verfolgt. Oft wird Happy-Path-Testing erst auf höheren Teststufen, wie Integrations- oder Systemtest, angewendet, allerdings kann es auch beim Unit Test eingesetzt werden. Fokus dieses Testverfahrens ist: Funktioniert es überhaupt?

Wann ist der Einsatz sinnvoll?

Wenn bei einer Funktionalität auch Ausnahmefälle oder Besonderheiten zu berücksichtigen sind, ist die korrekte Arbeitsweise der »Haupt«-Funktionen wichtig und gleich zu Beginn der Testaktivitäten sicherzustellen, bevor die Sonderfälle getestet werden.

Grundidee

Die Hauptfunktion muss funktionieren! Erst dann lohnt sich ein weiterer Einstieg in das Testobjekt. Wenn die Hauptfunktion fehlerhaft ist, sind zuerst entsprechende Korrekturen vorzunehmen, bevor das Testen fortgesetzt werden kann.

Ein einfaches Beispiel

Eine Klausur ist bestanden, wenn von den insgesamt möglichen 100 Punkten mindestens 50% erreicht werden. Die erreichte Punktzahl soll einer Funktion, die ermitteln soll, ob die Klausur bestanden ist, als Parameter übergeben werden. Der Happy Path wäre ein Ablauf der Software bei einer korrekten Angabe, d.h., dass die übergebene Zahl nach der Berechnungsvorschrift ein korrektes Ergebnis liefert. Falsche Angaben, wie beispielsweise eine Bereichsüberschreitung und deren Erkennung durch die Software, werden nicht getestet, jedenfalls nicht als Happy Path. *Anmerkung*: Sicherlich sind auch diese Angaben zu testen, aber wenn schon eine korrekte Angabe nicht funktioniert, ist der Test mit fehlerhaften Daten wenig sinnvoll.

Testendekriterium

Zuerst muss ermittelt werden, was als Hauptfunktion anzusehen ist und ob es mehr als eine gibt. Jede Hauptfunktion muss dann mit mindestens einem Testfall überprüft werden. Das ist für den Happy-Path-Test ausreichend (aber nur für diesen!).

Bewertung

Wenn es hier schon Fehler gibt, brechen Sie den Test ab! Als einziger Test ist der Happy-Path-Test nicht ausreichend, aber er gibt einen ersten Eindruck.

Bezug zu anderen Testverfahren

Als Einstieg in den Test raten wir zum Happy-Path-Test. Danach sind weitere Testverfahren zur Ermittlung von Testfällen anzuwenden. Welche, das

hängt ganz vom Testobjekt ab und kann nicht allgemein gültig angegeben werden.

C++-Beispiel

Hier wird das Beispiel der Prüfung des Klausurergebnisses von oben aufgegriffen. Die Funktion könnte lauten:

```
bool istBestanden(int punkte) {
    if (punkte < 0) {
       throw std::invalid_argument("negative Punktzahl!");
    }
    if (punkte > 100) {
       throw std::invalid_argument("zu hohe Punktzahl!");
    }
    return punkte >= 50;
}
```

Listing 4.6: Mögliche Implementierung der Funktion `istBestanden()`

Anmerkung: Nach dem Prinzip »Design by Contract[3]« könnten die `if`-Abfragen entfallen. Sie erhöhen jedoch die Robustheit bei Fehleingaben.

Der Happy-Path-Test ist dementsprechend sehr einfach. Hier wird die Zahl 88 als Stellvertreter für eine typische Punktzahl genommen:

```
TEST(istBestandenTest, ausreichende_Punktzahl) {
  EXPECT_TRUE(istBestanden(88));
}
```

Listing 4.7: Einfacher Happy-Path-Test

> Der Happy-Path-Test ist als Nachweis für die Korrektheit des Testobjekts bei Weitem nicht ausreichend, aber er zeigt schon das grundsätzliche Funktionieren. Es wird nicht geprüft, welche anderen Werte zu einem richtigen Ergebnis führen und welche Werte zu einem falschen Ergebnis oder einem Ausnahmefall (Exception) führen – das widerspräche der Definition des Begriffs »Happy Path«.

[3] Siehe Seite 10.

4.3 Äquivalenzklassentest

Der Äquivalenzklassentest ist ein weitverbreitetes durchaus übliches Vorgehen. Oft ist es nicht unter diesem Namen bekannt. Auch die vom Verfahren vorgesehene systematische Einteilung in gültige und ungültige Äquivalenzklassen[4] (s.u.) findet in der Praxis oft keine ausreichende Berücksichtigung.

Wann ist der Einsatz sinnvoll?

Die von einer Komponente zu erbringende Funktionalität wird durch einen oder mehrere Parameter gesteuert. Je nach dem Wert eines oder mehrerer Parameter ändert sich der Ablauf in der Komponente und unterschiedliche Ergebnisse werden geliefert. Es kann Klassen von Parametern geben, die zu ähnlichen Ergebnissen führen. Der Äquivalenzklassentest lässt sich in solchen Fällen sinnvoll anwenden, da die Steuerung des Ablaufs über Parameterwerte ein übliches Vorgehen bei der Programmierung ist.

Grundidee

Wird bei Eingabewerten (und auch Ausgabewerten) gleiches Verhalten (bzw. der gleiche Ablauf) des Testobjekts erwartet, so können diese Eingabewerte in Äquivalenzklassen[5] zusammengefasst werden. Diese Erwartung muss sich aus der Spezifikation ableiten lassen. Es wird also davon ausgegangen, dass jeder Wert einer Äquivalenzklasse zu einem gleichen Verhalten des Testobjekts führt. Testfälle sind so zu wählen, dass jeweils ein Repräsentant aus jeder Äquivalenzklasse herangezogen wird.

In aller Regel wird zwischen gültigen und ungültigen Äquivalenzklassen unterschieden. Als »gültige Äquivalenzklassen« – oder genauer formuliert Äquivalenzklassen mit gültigen Werten – werden solche bezeichnet, für die in der Spezifikation des Testobjekts eine entsprechende Funktionalität vorgesehen ist. »Ungültige Äquivalenzklassen« – oder solche mit ungültigen Werten – prüfen, inwieweit das Testobjekt auf Eingabewerte reagiert, für die keine Funktionalität vorgesehen ist und die mit einer Fehlermeldung oder einer anderen Mitteilung vom Testobjekt abzulehnen sind.

Für Ausgabewerte kann die Zerlegung in Äquivalenzklassen ebenso genutzt werden. Das Vorgehen ist aufwendiger, da für jeden »Ausgaberepräsentanten« (Ergebniswert) die korrespondierenden Eingabewerte zu ermitteln sind, um die Testfälle ausführen zu können. Auch bei den Ausgabewerten ist zwischen gültigen und ungültigen Werten zu unterscheiden.

[4] Dieser Sprachgebrauch ist üblich, aber nicht ganz korrekt, denn die Äquivalenzklasse selbst ist nicht ungültig, nur die Werte in Bezug auf die spezifizierte Eingabe.

[5] Es wird gleichartiges, äquivalentes Verhalten erwartet, daher der Name. Der Begriff »Klasse« wird nicht im Sinne der Objektorientierung verwendet!

Der Äquivalenzklassentest ist ein grundlegendes Verfahren, das auch Sie bestimmt schon – wenn auch nicht unter diesem Namen – angewendet haben. Wird bei unterschiedlichen Testdaten gleiches oder äquivalentes Verhalten des Testobjekts erwartet, dann reicht der Test mit einem dieser Testdaten aus. Ein sehr sinnvoller Lean-Ansatz zum Testen.

4.3.1 Ein Beispiel mit einem Parameter

Wir greifen das Beispiel vom Happy-Path-Test auf: Eine Klausur ist bestanden, wenn von den insgesamt möglichen 100 Punkten mehr als 50% erreicht werden. Die erreichte Punktzahl soll einem Programm als Parameter übergeben werden und es soll ermittelt werden, ob die Klausur bestanden oder nicht bestanden ist.

Die Äquivalenzklasse für gültige Eingabewerte ist: $0 \leq$ erreichte Punktzahl ≤ 100[6]. Eine Unterteilung in halbe Punkte oder noch feiner sei ausgeschlossen. Aus der Spezifikation ergibt sich, dass eine Aufteilung der Klasse vorgenommen werden muss, da ja zwischen Bestehen und Nichtbestehen der Klausur je nach Punktzahl (Eingabeparameter) unterschieden werden soll:

gÄK1[7]: $0 \leq$ erreichte Punktzahl ≤ 50 (nicht bestanden)
gÄK2: $50 <$ erreichte Punktzahl ≤ 100 (bestanden)

Daraus ergeben sich zwei Testfälle:

TF1: Eingabewert: 44; erwartetes Ergebnis: »nicht bestanden«
TF2: Eingabewert: 88; erwartetes Ergebnis: »bestanden«

Für ungültige Werte sind in aller Regel zwei Äquivalenzklassen zu bilden: eine, die über dem gültigen Wertebereich liegt, und eine darunter:

uÄK1[8]: Min_Int[9]$\leq$ erreichte Punktzahl < 0
uÄk2: $100 <$ erreichte Punktzahl $\leq$ Max_Int
TF3: Eingabewert: -2;
erwartetes Ergebnis: Fehlermeldung bzw. Fehlerbehandlung
TF4: Eingabewert: 111;
erwartetes Ergebnis: Fehlermeldung bzw. Fehlerbehandlung

[6] Die untere Grenze 0 ergibt sich nicht direkt aus der Spezifikation, wurde aber als sinnvoll angenommen.

[7] gÄK: gültige Äquivalenzklasse

[8] uÄK: ungültige Äquivalenzklasse

[9] Min_Int: kleinster ganzzahliger Wert, Max_Int: größter ganzzahliger Wert (je nach Rechnerhardware bzw. gewähltem Datentyp unterschiedlich).

Testfälle mit ungültigen Werten sind immer anzuraten, wenn nicht 100%ig garantiert werden kann, dass die Werte des Eingabeparameters immer im gültigen Bereich liegen (vgl. Kapitel 2).

Der gesamte Eingabebereich von Min_Int bis Max_Int ist in vier Äquivalenzklassen eingeteilt, diese müssen überschneidungsfrei sein. Theoretisch wären noch Eingaben anderer Datentypen denkbar, die kleiner als Min_Int bzw. größer als Max_Int sind. Bei kritischen Systemen sind gegebenenfalls auch solche Überlegungen einzubeziehen. Meist besteht aber gar nicht die Möglichkeit, solche Werte an das Testobjekt zu übergeben.

Weitere Testfälle mit beispielsweise den Eingabewerten 35, 75, -50, 150 werden als unnötig angesehen, da sie zu keinen neuen Erkenntnissen führen.

Testendekriterium

Jeweils ein beliebiger Repräsentant einer Äquivalenzklasse – sowohl gültige als auch ungültige – soll zur Erstellung mindestens eines Testfalls herangezogen werden. In unserem einfachen Beispiel sind die vier oben aufgeführten Testfälle als ausreichend anzusehen.

Bewertung

Das Verfahren steht und fällt mit der Güte der Einteilung der Wertebereiche in Äquivalenzklassen. Wird die Einteilung zu grob vorgenommen, dann wird unterschiedliches Verhalten des Testobjekts nicht getestet. Bei einer zu feingranularen Einteilung werden unnötige Tests durchgeführt. Auch lässt sich nicht immer aus der Spezifikation eindeutig eine Einteilung in Äquivalenzklassen ableiten. Bestehen Zweifel an der Gleichbehandlung von Werten innerhalb einer Äquivalenzklasse, ist die Äquivalenzklasse entsprechend weiter zu unterteilen.

Ein zusätzlicher Vorteil ist die Umsetzung der meist umgangssprachlichen Spezifikation in Äquivalenzklassen. Diese definieren exakt, welche Werte in welche Äquivalenzklasse fallen. So kann im obigen Beispiel unklar sein, was bei genau 50 Punkten als Ergebnis erwartet wird. Die Umsetzung der Spezifikation in Äquivalenzklassen lässt keinen Interpretationsspielraum zu, die 50 gehört zu gÄk1 und damit ist die Klausur nicht bestanden.

Bezug zu anderen Testverfahren

Eine sinnvolle Ergänzung zum Äquivalenzklassentest ist die Grenzwertanalyse (siehe Abschnitt 4.4), da bei den Übergängen von einer Äquivalenzklasse in die benachbarte oft Fehler auftreten.

Als Einstieg in die systematische Herleitung von Testfällen ist die Verwendung des Äquivalenzklassentests empfehlenswert. Insbesondere die ungültigen Äquivalenzklassen helfen, nicht bedachte Ausnahmebehandlungen durch entsprechende Testfälle aufzudecken. Als alleiniges Testentwurfsverfahren ist es in aller Regel nicht ausreichend, es muss durch weitere Testfälle, hergeleitet aus anderen Testentwurfsverfahren, z.B. der Grenzwertanalyse (s.u.), ergänzt werden.

Hinweise für die Praxis

Steuert der übergebene Wert eines Parameters den Ablauf des Testobjekts und ist für den Parameterwert ein zusammenhängender Definitionsbereich spezifiziert – wie im oberen Beispiel ([0,100]) – dann sind eine gültige und zwei ungültige Äquivalenzklassen zu berücksichtigen. Die gültige Äquivalenzklasse ist unter Beachtung der Spezifikation weiter zu unterteilen. Die ungültigen Äquivalenzklassen ergeben sich durch Über- bzw. Unterschreitung des Definitionsbereichs. Selbst wenn es keine Einschränkung gibt, wenn z.B. der gesamte ganzzahlige Bereich genutzt werden kann ([Min_Int, Max_Int]), sind ungültige Äquivalenzklassen mit Werten, die kleiner Min_Int bzw. größer Max_Int sind, gegeben und eventuell beim Test zu berücksichtigen.

Wird über einen Parameter spezifiziert, dass eine Anzahl von Werten einzugeben ist, sind eine gültige (mit allen möglichen gültigen Werten) und zwei ungültige Äquivalenzklassen (Unter- und Überschreitung der gültigen Anzahl) zu bilden. Sind beispielsweise in einem Sportverein mindestens eine und maximal vier Sportarten von einem Mitglied zu belegen, so sind folgende Äquivalenzklassen zu bilden:

gÄk1: $1 \leq$ Anzahl Sportart ≤ 4
uÄk1: Min_Int $\leq$ Anzahl Sportart < 1
uÄk2: $4 <$ Anzahl Sportart $\leq$ Max_Int.

Ist eine Menge von Werten spezifiziert, die möglicherweise unterschiedlich zu behandeln sind, so ist für jeden Wert der Menge eine gültige Äquivalenzklasse (bestehend aus diesem einen Wert) vorzusehen und eine zusätzliche ungültige Äquivalenzklasse für alle anderen Werte. Bietet der Sportverein beispielsweise die Sportarten Tischtennis, Turnen, Volleyball, Basketball, Handball und Fitnesstraining, dann ist für jede Sportart eine gültige Äquivalenzklasse und somit ein Testfall zu bilden. Alle nicht aufgeführten Sportarten können zu einer ungültigen Äquivalenzklasse zusammengefasst werden und der Test mit einem Repräsentanten, zum Beispiel Schach, wird als ausreichend angesehen. Sind die sechs angebotenen Sportarten von der

Spezifikation her nicht unterschiedlich zu behandeln, sind z. B. keine Unterschiede im Mitgliedsbeitrag gegeben, reicht es aus, nur eine gültige Äquivalenzklasse mit allen sechs Sportarten zu bilden, und einen beliebigen Repräsentanten für den Test daraus zu wählen.

Falls die Einschränkung oder Bedingung eine Situation beschreibt, die zwingend erfüllt werden muss, ist jeweils eine gültige und eine ungültige Äquivalenzklasse zu berücksichtigen. Beim Sportverein sind die Mitgliedskennungen auf sieben Zeichen beschränkt, wobei das erste Zeichen der Anfangsbuchstabe des Nachnamens des Mitglieds ist, gefolgt von sechs Ziffern (die jeweils bei einem neuen Mitglied fortlaufend hochgezählt werden). Eine gültige Äquivalenzklasse umfasst alle Mitgliedskennungen, die mit einem Buchstaben beginnen (ohne »ß«, aber mit Umlauten), die ungültige Äquivalenzklasse alle Kennungen ohne Buchstaben an der 1. Stelle. Am Beispiel wird deutlich, dass ergänzende Tests sinnvoll sind, z.B. mit »ß« (ungültig), »Ä«, »Ü« usw. (gültig) als Anfangszeichen. Auch ist zu prüfen, ob die Länge der Mitgliedskennung korrekt ist. Hier wären drei Äquivalenzklassen zu bilden: eine mit der korrekten Länge (sieben Zeichen) und zwei ungültige Äquivalenzklassen mit Unter- bzw. Überschreitung der geforderten Länge.

4.3.2 Das Beispiel in C++

Im Folgenden sehen Sie eine einfache C++-Funktion zur obigen Fragestellung, ob eine Klausur bestanden ist:

```
#include "istBestanden.h"
#include <stdexcept>
#include <string>

bool istBestanden(int punkte) {
  if (punkte < 0) {
    throw std::invalid_argument("zu kleine Punktzahl: "
                            + std::to_string(punkte));
  }
  if (punkte > 100) {
    throw std::invalid_argument("zu hohe Punktzahl: "
                            + std::to_string(punkte));
  }
  return punkte >= 50;
}
```

Listing 4.8: Klausur bestanden? *(beispiele/klausur/istBestanden.cpp)*

Im Fehlerfall wird eine Exception geworfen. Um den Fehler leichter zu finden, wird der Exception die fehlerhafte Punktzahl mitgegeben. Die Testfälle

können entsprechend den oben erwähnten Äquivalenzklassen wie folgt aussehen:

```
TEST(istBestandenTest, 1_zu_kleine_Punktzahl) {
   ASSERT_THROW(istBestanden(-2), std::invalid_argument);
}

TEST(istBestandenTest, 2_zu_hohe_Punktzahl) {
   ASSERT_THROW(istBestanden(111), std::invalid_argument);
}

TEST(istBestandenTest, 3_kleine_Punktzahl) {
  EXPECT_FALSE(istBestanden(44));
}

TEST(istBestandenTest, 4_ausreichende_Punktzahl) {
  EXPECT_TRUE(istBestanden(88));
}
```

Listing 4.9: Mögliche Testfälle

Die ersten beiden Testfälle entsprechen den ungültigen Äquivalenzklassen. Der Test stellt sicher, dass eine `invalid_argument`-Exception geworfen wird. Die letzten beiden Tests entsprechen den gültigen Äquivalenzklassen.

Das war einfach – ganz im Sinne von »Lean Testing«! Wie bei einer so einfachen Funktion zu erwarten war, werden alle vier Tests bestanden. Mehr Aufwand wird normalerweise nicht notwendig sein. Aber: Oben heißt es: Theoretisch wären noch Eingaben anderer Datentypen denkbar, die kleiner als Min_Int bzw. größer als Max_Int sind. Bei kritischen Systemen sind gegebenenfalls auch solche Überlegungen einzubeziehen. Meist besteht aber gar nicht die Möglichkeit, solche Werte an das Testobjekt zu übergeben.

4.3.3 Erweiterung auf andere Datentypen

In diesem Abschnitt geht es um Fehler, die entstehen, wenn der Aufrufer ein Argument übergibt, das eine größere Bitbreite hat und daher den Zahlenbereich des Parametertyps überschreitet.

In unserem Fall handelt es sich definitiv nicht um ein kritisches System. Dennoch lässt sich auch an einer so einfachen Funktion gut das Problem zeigen, sodass wir hier diesen Aspekt genauer untersuchen, obwohl es wegen des höheren Aufwandes mit dem »Lean« in »Lean Testing« nichts mehr zu tun hat. Zunächst stellen wir Ihnen eine sehr einfache Möglichkeit vor, auf Min_Int zu prüfen, indem eine zusätzliche Anweisung aufgenommen wird:

```
bool istBestanden(int punkte) {
  if(punkte < std::numeric_limits<int>::min()){
    // problematisch, siehe Text unten
    throw std::invalid_argument("Min_Int unterschritten!");
  }
// .. Rest wie oben
}
```

Listing 4.10: Fehlerhafte Abfrage

Einfach, oder? Der C++-Kenner sieht sofort, was da faul ist: `punkte` kann als `int`-Wert prinzipiell niemals kleiner als `std::numeric_limits<int>::min()` (d.h. Min_Int) werden. Die Bedingung ist immer falsch. Nun mögen Sie uns für Scherzkekse halten, aber tatsächlich haben wir solche und ähnliche Vergleiche schon gesehen. Auch bei Experten kommt es vor: So steht in einem Lehrbuch die Abfrage `if(i<0 || ...)` für die `unsigned int`-Variable `i`, die niemals kleiner als null sein kann [Stroustrup 08, S. 671]. Natürlich ist das ein Versehen, aber es zeigt deutlich, dass so etwas wirklich jedem passieren kann. Insofern kann unser Hinweis nicht schaden.

Um Werte < Min_Int und > Max_Int zu erzeugen, verwenden wir den Datentyp `long`, wobei vorausgesetzt wird, dass eine `long`-Variable mehr Bits als eine `int`-Variable benötigt. Dies wird durch eine `static_assert`-Anweisung geprüft. Falls sie fehlschlägt, nehmen Sie einfach `long long` statt `long`. `int` hat in C++ meistens 32 Bits und `long long` muss mindestens 64 Bits haben. Im Folgenden sei vorausgesetzt, dass `int` 32 Bits benötigt und `long` 64 Bits. Sehen wir uns den folgenden Test an:

```
TEST(istBestandenTest, 5_MAX_INT_Punktzahl) {
  static_assert(sizeof(long) > sizeof(int),
                "long hat zu wenige Bits!");
  long punkte = 55L + std::numeric_limits<int>::max();  // = 55 + Max_Int
  std::cout << "aufgerufen mit " << punkte << " Punkten\n";
  ASSERT_THROW(istBestanden(punkte), std::invalid_argument);
}
```

Listing 4.11: Test auf Überlauf

Max_Int ist 2147483647. Das Suffix L nach der Zahl 55 sorgt dafür, dass 55 als `long`-Zahl interpretiert und deshalb eine `long`-Addition (statt `int`) durchgeführt wird. Die Anzahl der übergebenen Punkte ist 2147483702, es sind also eindeutig zu viele, d.h. mehr als 100 und auch mehr als Max_Int. Es wird eine Exception geworfen und der Test ist damit bestanden. Der Inhalt der Exception wird bei Google Test leider nicht angezeigt, weswegen tatsächlich nicht klar ist, ob die Exception den richtigen Inhalt enthält (zu kleine oder zu hohe Punktzahl). Um das herauszufinden, wird der Testfall 5 erweitert (Testfall 5A):

```
namespace {
  void checkException(int punkte) {
    try {
      istBestanden(punkte);
    } catch(const std::exception& e) {
      std::cout << "Exception: " << e.what() << '\n';
    }
  }
}
// ...

TEST(istBestandenTest, 5A_MAX_INT_Punktzahl) {
  static_assert(sizeof(long) > sizeof(int),
                      "long hat zu wenige Bits!");
  long punkte = 55L + std::numeric_limits<int>::max();
  std::cout << "aufgerufen mit " << punkte << " Punkten\n";
  checkException(punkte);    // Aufruf der oben definierten Funktion
  ASSERT_THROW(istBestanden(punkte), std::invalid_argument);
}
```

Listing 4.12: Erweiterter Test auf Überlauf

Überraschung: Obwohl auf Max_Int geprüft wird, ist der Inhalt der Exception »zu kleine Punktzahl: -2147483594«! Man ahnt, woher das kommt: Es hat etwas mit der Umwandlung der `long`-Zahl in eine `int`-Zahl beim Aufruf von `istBestanden()` zu tun. Tatsächlich werden bei der Typumwandlung einfach die überzähligen Bits abgeschnitten. Der Compiler meldet dies nicht, auch wenn ihm die Option `-Wall` übergeben wird[10]. Basierend auf diesem Ergebnis kann man einen weiteren Testfall für die Überschreitung von Max_Int entwerfen:

```
TEST(istBestandenTest, 6_MAX_INT_Punktzahl) {
  static_assert(sizeof(long) > sizeof(int),
                "long hat zu wenige Bits!");
  long punkte =   2L * std::numeric_limits<int>::max() + 70;
  std::cout << "aufgerufen mit " << punkte << " Punkten\n";
  ASSERT_THROW(istBestanden(punkte), std::invalid_argument);
}
```

Listing 4.13: Weiterer Test auf Überlauf

Hier wird `istBestanden(4294967364)` aufgerufen, aber es gibt keine Exception, obwohl es eine hätte geben müssen. Der Test schlägt fehl. Der Grund:

[10] Ein gutes Werkzeug zur statischen Analyse meldet den Fehler zuverlässig.

Innerhalb der Funktion wird die Zahl wegen der Typumwandlung als 68 interpretiert. Die Erklärung dafür: Unser Computer ist ein 64-Bit-System, auf dem `long` acht Bytes und `int` vier Bytes groß sind. Bei der Typumwandlung von `long` nach `int` werden einfach die überzähligen Bits abgeschnitten. Die folgenden Zahlen zeigen die `long`-Zahl 4294967364 in binärer Darstellung und die entstehende `int`-Zahl in binärer Darstellung nach Abschneiden der höchstwertigen 32 Bits:

```
00000000000000000000000000000001000000000000000000000001000100
                                00000000000000000000000001000100
```

So wird aus der `long`-Zahl 4294967364 die `int`-Zahl 68. Der Wert liegt zwischen 0 und 100 und führt deswegen nicht zu einer Exception. `istBestanden()` meldet keinen Fehler, obwohl einer vorliegt (falsch negative Meldung), weil die Funktion diesen Fehler nicht erkennen kann.

Entsprechende Überlegungen gelten auch für Min_Int sowie für den Fall, dass die Punktzahl nicht als `int`, sondern als `unsigned int` oder `std::size_t` übergeben wird, weil die wirkliche Punktzahl nicht negativ sein kann. Min_Int wäre dann 0.

Schlussfolgerung 1: Bei kritischen Systemen reicht ein einfacher Unit Test einer Funktion nicht aus. Man muss auch sicherstellen, dass die aufrufende Software einen ganzzahligen Datentyp verwendet, der maximal die Bitbreite des Datentyps in der Parameterliste hat und der dasselbe Vorzeichenverhalten zeigt. Andernfalls kann es falsche Resultate geben, die mit einfachen Tests nicht als solche erkannt werden können. Das liegt außerhalb des reinen Unit Test der Funktion. Wenn das nicht garantiert ist, wäre ein Typ `double` oder gar `long double` als Parametertyp sicherer.

Die Problematik der Typumwandlung ist damit deutlich geworden. Sie ist eine Eigenschaft der Sprache C++. Die Problematik tritt natürlich an vielen Stellen der Softwarewelt auf, aber weil sie nun bekannt und begründet ist, gehen wir in den folgenden Beispielen des Buchs nur noch ausnahmsweise darauf ein.

Schlussfolgerung 2: Normalerweise soll nicht mit gemischten Datentypen gearbeitet werden. Wenn sich alle daran halten und Ausnahmen sorgfältig behandelt werden, genügen die einfachen vier ersten der oben gezeigten Tests vollständig.

4.3.4 Mehrere Parameter

Bisher wurde der Äquivalenzklassentest für einen Parameter beschrieben. Wie sieht es nun aus, wenn die zu testende Funktion mehrere Parameter besitzt? Zuerst sind für jeden einzelnen Parameter die gültigen und ungültigen Äquivalenzklassen zu bilden. Wie sind die dann für jede Äquivalenzklasse zu wählenden Repräsentanten miteinander zu kombinieren?

Nehmen wir an, wir haben drei Parameter und zwei dieser Parameter haben jeweils eine gültige und zwei ungültige Äquivalenzklassen (insgesamt je 3) und der dritte Parameter hat zwei gültige und zwei ungültige Äquivalenzklassen (insgesamt 4). Eine mögliche Strategie ist die vollständige Kombination der Äquivalenzklassen, im Beispiel ergeben sich dann $3 \cdot 3 \cdot 4 = 36$ Testfälle. Diese Strategie ist allerdings wenig sinnvoll, schon gar nicht, wenn wir gemäß »Lean Testing« vorgehen wollen! Wenn zum Beispiel für alle drei Parameter Repräsentanten aus den ungültigen Äquivalenzklassen für einen Testfall kombiniert werden, und das Testobjekt bei Ausführung des Testfalls einen Fehler meldet, wie bei ungültigen Repräsentanten ja erwartet, ist nicht klar, welcher der drei ungültigen Werte die Fehlermeldung bewirkt hat. Der Testfall führt somit zu keiner wirklichen Erkenntnis.

Lean Testing: von 36 zu 8 Testfällen

Um die korrekte Behandlung der Repräsentanten der ungültigen Äquivalenzklassen zu prüfen, sind bei den Testfällen nur für einen Parameter ein ungültiger Wert und für die anderen Parameter gültige Werte zu wählen. Zugrunde liegt die folgende vernünftige Annahme: Wenn gleichzeitig für zwei (oder mehr) Parameter ein ungültiger Wert gewählt wird, ist es extrem unwahrscheinlich, dass der eine Fehler den anderen kompensiert. Es ist anzunehmen, dass einer von den beiden ungültigen Werten allein zur Fehlermeldung führt. Kommt es bei der Ausführung der Testfälle, bei denen nur ein Parameterwert aus einer ungültigen Äquivalenzklasse gewählt wird, zu der erwarteten Fehlermeldung, ist klar, dass dies durch den einen ungültigen Wert verursacht wurde. Wir benötigen somit $2 + 2 + 2 = 6$ Testfälle mit jeweils einem Repräsentanten aus einer ungültigen Äquivalenzklasse, um für jeden falschen Wert die entsprechende Prüfung durchzuführen.

Im Beispiel haben wir zwei Parameter mit einer und einen mit zwei gültigen Äquivalenzklassen. Werte aus gültigen Äquivalenzklassen können ohne Einschränkung miteinander kombiniert werden. Ziel beim Äquivalenzklassentest ist ja der Test von einem Repräsentanten aus jeder Äquivalenzklasse. Somit ergeben sich zwei weitere Testfälle aus den beiden gültigen Äquivalenzklassen des dritten Parameters.

> Werte aus ungültigen Äquivalenzklassen sollen in einem Testfall nicht miteinander kombiniert werden – für jeden Wert einer ungültigen Äquivalenzklasse ist ein Testfall zu spezifizieren. Für Werte aus gültigen Äquivalenzklassen gilt diese Einschränkung nicht, sie können miteinander kombiniert werden.

Mit insgesamt acht Testfällen haben wir die Funktion mit den drei Parametern nach dem Äquivalenzklassentest ausreichend getestet.

Wir zeigen den Äquivalenzklassenentwurf und die resultierenden Testfälle an zwei konkreten Beispielen mit mehreren Parametern. Im ersten Beispiel, einer Funktion zur Berechnung von Rabatten, überschneiden sich die Testfälle nicht (sie sind disjunkt). Im zweiten Beispiel wird die Gültigkeit eines Datums geprüft. Dort hängen manche Testfälle voneinander ab.

Beispiel: Berechnung von Rabatten

Ein Supermarkt ist an den Wochentagen (Mo.-Sa.) von 08:00 bis 20:00 Uhr geöffnet. Ein Kassenprogramm in dem Supermarkt hat folgende Anforderungen:

1. Ab einem Rechnungsbetrag von insgesamt 150,00 € wird ein Rabatt von 10,00 € gewährt.
2. Da es morgens immer recht leer im Supermarkt ist, soll ein weiterer Rabatt für größeren Umsatz sorgen: Auf alle Waren, die morgens bis 10:00 Uhr eingekauft werden, wird ein Rabatt von 5% gewährt (Frühkaufrabatt).
3. Der Frühkaufrabatt kann nicht mit dem Rechnungsbetragrabatt kombiniert werden. Es soll der für den Kunden günstigere der beiden Rabatte gelten.

Der für den Rabatt entscheidende Baustein des Kassenprogramms sei die Funktion `size_t rabatt(size_t rechnungsbetrag, size_t stunde, size_t minute)`. Um die Funktion unabhängig von der Systemzeit aufrufen zu können, wird die Tageszeit übergeben, indem jeweils die Stunde und die Minute angegeben werden. Zwar könnte die Tageszeit ein eigener Datentyp sein, aber zugunsten der Einfachheit des Entwurfs der Äquivalenzklassen und der Funktionsinterna wird darauf verzichtet. Der zurückgegebene Rabatt und der Rechnungsbetrag werden als Centbeträge angesehen, weil es keine kleinere Stückelung gibt. Der Datentyp `size_t` wird gewählt, weil mit ihm die größtmöglichen `unsigned`-Zahlen darstellbar sind. *Eine* mögliche Ursache für Fehler, die durch eine Typumwandlung entstehen, entfällt dadurch (vgl. Abschnitt 4.3.3). Aus Plausibilitätsgründen wird eine Obergrenze von 10.000,00 € festgelegt. Ein höherer Wert gilt als Fehler. Pfandgeld wird nicht berücksichtigt, um negative Beträge auszuschließen und das Beispiel einfach zu halten. Deshalb muss der Rechnungsbetrag $> 0{,}00$ € sein. Aus Punkt 1 der Anforderungen ergeben sich die Äquivalenzklassen[11] der folgenden Tabelle 4-1:

[11] gÄK – gültige, uÄK – ungültige Äquivalenzklasse.

Äquivalenzklasse	Betrag B	Rabatt
gÄK11	0,00 € < B < 150,00 €	0,00 €
gÄK12	150,00 € ≤ B ≤ 10.000,00 €	10,00 €
uÄK11	B > 10.000,00 €	irrelevant
uÄK12	B = 0,00 €	irrelevant

Tabelle 4-1: Äquivalenzklassen für den Rechnungsbetrag

Wegen des Datentyps `size_t` brauchen negative Zahlen nicht berücksichtigt zu werden. Min_Int ist 0. Es wird angenommen, dass der rechnerabhängige Wert von Max_Int mindestens 4 Bytes umfasst und damit weitaus größer als 1.000.000 ist (= 10.000 € in Cent). Aus Punkt 2 der Anforderungen ergeben sich die folgenden Äquivalenzklassen:

Äquivalenzklasse	Uhrzeit Z
gÄK21	08:00 ≤ Z ≤ 10:00
gÄK22	10:01 ≤ Z ≤ 20:00
uÄK21	20:01 ≤ Z < 24:00
uÄK22	00:00 ≤ Z < 08:00
uÄK23	Z ≥ 24:00

Tabelle 4-2: Äquivalenzklassen für die Uhrzeit

Auch hier brauchen wegen des Datentyps `size_t` negative Zahlen nicht berücksichtigt zu werden. 24:00 wird nicht als korrekte Zeit gewertet (Mitternacht ist 00:00).

Nun ist noch zu berücksichtigen, dass der für den Kunden günstigere der beiden Rabatte gelten soll. Das bedeutet, dass die Parameter nicht mehr als unabhängig voneinander betrachtet werden können. Bei einem Rechnungsbetrag von 200 € am Morgen zwischen acht und zehn Uhr ergibt sich ein Frühkaufrabatt von 5% = 10 €. Das ist derselbe Rabatt, den es nach Punkt 1 oben bei einem Rechnungsbetrag von 150 € gibt. Für den Uhrzeitbereich von 08:00 bis 10:00 ergibt sich damit die Tabelle 4-3, die eine Aufteilung der gültigen Äquivalenzklassen der Tabelle 4-1 darstellt.

Äquivalenzklasse	Uhrzeit Z	Betrag B	Rabatt
gÄK31	08:00 ≤ Z ≤ 10:00	0,00 € < B < 150,00 €	5%
gÄK32	08:00 ≤ Z ≤ 10:00	150,00 € ≤ B ≤ 200,00 €	10 €
gÄK33	08:00 ≤ Z ≤ 10:00	200,00 € < B ≤ 10.000,00 €	5%
gÄK34	10:01 ≤ Z ≤ 20:00	0,00 € < B < 150,00 €	0%
gÄK35	10:01 ≤ Z ≤ 20:00	150,00 € ≤ B ≤ 10.000,00 €	10 €

Tabelle 4-3: Neue gültige Äquivalenzklassen für den Betrag mit Uhrzeiten

Für die Testfälle sind damit die folgenden Äquivalenzklassen zu berücksichtigen:

- Alle Äquivalenzklassen der Tabelle 4-3
- Alle ungültigen Äquivalenzklassen der Tabellen 4-1 und 4-2

Wegen der notwendigen Kombination der Parameter entfallen alle gültigen Äquivalenzklassen der Tabellen 4-1 und 4-2. Sie werden durch die in Tabelle 4-3 ersetzt. Es ist zu berücksichtigen, dass die Repräsentanten der ungültigen Äquivalenzklassen jeweils zu nur einem Testfall führen. Insgesamt kommt man mit zehn Testfällen aus:

```
#include <gtest/gtest.h>
#include "rabatt.h"
// Beträge sind in Cent!

// gültige Äquivalenzklassen
TEST(RabattTest, gAeK31_Morgens_0_B_150) {
   EXPECT_EQ(rabatt(10000, 9, 0), 500);
}

TEST(RabattTest, gAeK32_Morgens_150_B_200) {
   EXPECT_EQ(rabatt(17000, 9, 30), 1000);
}

TEST(RabattTest, gAeK33_Morgens_200_B_10000) {
   EXPECT_EQ(rabatt(35000, 8, 27), 1750);
}

TEST(RabattTest, gAeK34_Tag_0_B_150) {
   EXPECT_EQ(rabatt(8000, 17, 5), 0);
}

TEST(RabattTest, gAeK35_Tag_150_B_10000) {
   EXPECT_EQ(rabatt(16000, 16, 5), 1000);
}

// ungültige Äquivalenzklassen
TEST(RabattTest, uAeK11_zuhoherBetrag) {
   ASSERT_THROW(rabatt(1000001, 12, 0), std::invalid_argument);
}

TEST(RabattTest, uAeK12_Betrag_0) {
   ASSERT_THROW(rabatt(0, 12, 0), std::invalid_argument);
```

```
}

TEST(RabattTest, uAeK21_UhrzeitZuspaet) {
   ASSERT_THROW(rabatt(50000, 20, 1), std::invalid_argument);
}

TEST(RabattTest, uAeK22_UhrzeitZufrueh) {
   ASSERT_THROW(rabatt(50000, 7, 30), std::invalid_argument);
}

TEST(RabattTest, uAeK23_UhrzeitZugross) {
   ASSERT_THROW(rabatt(50000, 24, 0), std::invalid_argument);
}
```

Listing 4.14: Die zehn Testfälle *(beispiele/rabatt/dertest.cpp)*

Nach dem Prinzip »Design by Contract[12]« würden die Prüfungen der Vorbedingungen entfallen, aber wir ziehen robuste Programme vor, insbesondere, wenn die Prüfung wenig Laufzeit kostet.

Man kann grundsätzlich zuerst die Funktionsfähigkeit prüfen, denn um die geht es in erster Linie, und dann erst auf ungültige Äquivalenzklassen testen. In einem Programm wird man aber zuerst die Vorbedingungen prüfen, um gegebenenfalls den Code für die eigentliche Funktion gar nicht erst auszuführen – schon um Folgefehler zu verhindern. Insofern ist es ebenso berechtigt, erst auf ungültige Äquivalenzklassen und danach auf gültige zu testen. In den Beispielen finden Sie beide Varianten.

```
std::size_t rabatt(std::size_t rechnungsbetrag,
                   std::size_t stunde, std::size_t minute) {
  // Prüfen der Vorbedingungen
  // (Repräsentanten ungültiger Äquivalenzklassen)
  if(rechnungsbetrag > 1000000)                          // uAeK11
    throw std::invalid_argument("Rechnungsbetrag zu hoch");
  if(rechnungsbetrag == 0)                               // uAeK12
    throw std::invalid_argument("Rechnungsbetrag ist 0");
  if((stunde == 20 && minute > 0)                        // uAeK21
     ||  (stunde > 20 && stunde < 24))
    throw std::invalid_argument("Uhrzeit zu spät");
  if(stunde < 8)                                         // uAeK22
    throw std::invalid_argument("Uhrzeit zu früh");
  if(stunde >= 24)                                       // uAeK23
    throw std::invalid_argument("Stunde >= 24");
```

[12] Siehe Diskussion auf Seite 10.

```
  // Prüfung von Repräsentanten gültiger Äquivalenzklassen gAeK3x (x=1..5)
  // (eigentliche Rabattberechnung)
  std::size_t rabatt1 = rechnungsbetrag >= 15000 ? 1000 : 0;
  std::size_t rabatt2 = 0;
  if((stunde == 10 && minute == 0) || stunde < 10) {
    double temp = 0.05 * rechnungsbetrag;
    rabatt2 = static_cast<std::size_t>(temp + 0.5); // Rundung
  }
  return std::max(rabatt1, rabatt2);
}
```

Listing 4.15: Mögliche Definition der Funktion `rabatt()`

Im Folgenden sehen Sie die Ausgabe des Testprogramms. Bitte wundern Sie sich nicht, dass die von uns produzierten Tests alle erfolgreich sind, es sei denn, wir wollen einen Fehler demonstrieren. Übrigens waren sie es anfangs nicht immer ...

```
Running main() from gtest_main.cc
[==========] Running 10 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 10 tests from RabattTest
[ RUN      ] RabattTest.gAeK31_Morgens_0_B_150
[       OK ] RabattTest.gAeK31_Morgens_0_B_150 (0 ms)
[ RUN      ] RabattTest.gAeK32_Morgens_150_B_200
[       OK ] RabattTest.gAeK32_Morgens_150_B_200 (0 ms)
[ RUN      ] RabattTest.gAeK33_Morgens_200_B_10000
[       OK ] RabattTest.gAeK33_Morgens_200_B_10000 (0 ms)
[ RUN      ] RabattTest.gAeK34_Tag_0_B_150
[       OK ] RabattTest.gAeK34_Tag_0_B_150 (0 ms)
[ RUN      ] RabattTest.gAeK35_Tag_150_B_10000
[       OK ] RabattTest.gAeK35_Tag_150_B_10000 (0 ms)
[ RUN      ] RabattTest.uAeK11_zuhoherBetrag
[       OK ] RabattTest.uAeK11_zuhoherBetrag (0 ms)
[ RUN      ] RabattTest.uAeK12_Betrag_0
[       OK ] RabattTest.uAeK12_Betrag_0 (0 ms)
[ RUN      ] RabattTest.uAeK21_UhrzeitZuspaet
[       OK ] RabattTest.uAeK21_UhrzeitZuspaet (0 ms)
[ RUN      ] RabattTest.uAeK22_UhrzeitZufrueh
[       OK ] RabattTest.uAeK22_UhrzeitZufrueh (0 ms)
[ RUN      ] RabattTest.uAeK23_UhrzeitZugross
[       OK ] RabattTest.uAeK23_UhrzeitZugross (0 ms)
[----------] 10 tests from RabattTest (0 ms total)

[----------] Global test environment tear-down
```

```
[==========] 10 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 10 tests.
```

Listing 4.16: Testergebnis

Beispiel: Gültigkeit eines Datums

Ein beliebiges Datum soll von einer Funktion `bool istGueltigesDatum(size_t tag, size_t monat, size_t jahr)` geprüft werden. Die Funktion gibt `true` zurück, wenn das Datum gültig ist, andernfalls `false`. Hier ist schon zu sehen, dass im Unterschied zur Rabattfunktion des vorhergehenden Abschnitts keine besondere Prüfung der Vorbedingungen mit Werfen einer Exception notwendig ist. Das Ergebnis `false` reicht dem Aufrufer, um zu erkennen, dass ein Datum falsch ist. Dasselbe Vorgehen bei der Rabattfunktion kann bei der Datumsprüfung durch einen zusätzlichen Parameter erreicht werden, der im Fehlerfall entsprechend gesetzt wird. So eine Vermischung zwischen Fehler- und richtigen Werten ist jedoch aus softwaretechnischer Sicht nicht empfehlenswert.

Die Frage ist: Wie könnte die Funktion `istGueltigesDatum(...)` systematisch so getestet werden, dass der Test alle wesentlichen Konstellationen berücksichtigt und dass trotzdem die Anzahl der Testfälle nicht ausufert (Stichwort »Lean Testing«). Zunächst muss klar sein, welche Zahlenwerte für die Parameter überhaupt sinnvoll möglich sind. Dazu legen wir fest:

Der Wert für den Tag darf zwischen 1 und 31 liegen.
Der Wert für den Monat darf zwischen 1 und 12 liegen.
Der Wert für das Jahr darf zwischen 1583 und 2399 liegen.

1583 soll der kleinstmögliche Wert sein, weil unser gregorianischer Kalender im Jahre 1582 eingeführt wurde. Der größtmögliche Wert für ein Jahr wird begrenzt, um eine Plausibilitätsprüfung einzubauen. Die Grenze 2399 ist willkürlich festgelegt. Mit den Kombinationen dieser Zahlen für Tag, Monat und Jahr lassen sich alle in unserem praktischen Leben vorkommenden Daten[13] bilden – allerdings auch jede Menge ungültige Kombinationen wie etwa der 30.2.1999. Aus den gegebenen Angaben lassen sich die in der Tabelle 4-4 aufgeführten Äquivalenzklassen ableiten (Seite 50).

Nach dem bisherigen Vorgehen ergibt sich die Zahl der Testfälle aus dem Produkt der Zahl der gültigen Äquivalenzklassen je Parameter (3·1·1) plus Anzahl der ungültigen Äquivalenzklassen (6), also zusammen neun. Jedoch: Die Tatsache, dass *Kombinationen* verschiedener gültiger Äquivalenzklassen ungültige Ergebnisse erzeugen können, wird dabei überhaupt nicht berücksichtigt. Ein Beispiel: gÄKT2 kombiniert mit gÄKM1 und gÄKJ1, etwa

[13] Historiker mögen dies anders sehen. Es ist für die Zwecke dieses Buchs zu kompliziert, die Kalender vor der Reform im Jahr 1582 zu berücksichtigen.

Parameter	Äquivalenzklasse	Repräsentant	Anmerkung
Tag	gÄKT1: [1..28]	10	geht immer
	gÄKT2: [30]	30	Monate 1, 3..12
	gÄKT3: [31]	31	Monate 1,3,5,7,8,10,12
	uÄKT1: [0]	0	= Min_Int (wg. `size_t`)
	uÄKT2: [32..Max_Int]	60	
Monat	gÄKM1: [1..12]	6	geht immer
	uÄKM1: [0]	0	= Min_Int
	uÄKM2: [13..Max_Int]	33	
Jahr	gÄKJ1: [1583..2399]	2015	laut Spezifikation
	uÄKJ1: [0..1582]	1000	laut Spezifikation
	uÄKJ2: [2400..Max_Int]	2500	laut Spezifikation

Tabelle 4-4: Äquivalenzklassen für die Datum-Parameter

der 30.2.2016. Das obige Schema ist offensichtlich nur anwendbar, wenn alle Parameter bezogen auf das Ergebnis als vollständig unabhängig betrachtet werden können.

Was ist zu tun? Wir müssen die Äquivalenzklassen weiter aufteilen und dabei die möglichen gültigen und ungültigen Kombinationen von Tag, Monat und Jahr berücksichtigen. Wir teilen aber nicht nur weiter auf, sondern schaffen Äquivalenzklassen, die alle drei Parameter (Tag, Monat, Jahr) umfassen. Die drei Teile dieser Äquivalenzklassen können dann aus einzelnen Werten oder – wie bisher – aus Wertebereichen bestehen. Einzelne Werte bei Äquivalenzklassen sind durchaus üblich. Wenn beispielsweise die Null ein wichtiges Eingabedatum für den Test ist, ist eine Äquivalenzklasse mit nur der Null durchaus sinnvoll. Wie sehen nun die gültigen und ungültigen Äquivalenzklassen unter Berücksichtigung der Kombinationen aus?

Gültige Äquivalenzklassen zur Parameterkombination

Zur Bildung der gültigen Äquivalenzklassen (gÄK) gehen wir zunächst vom einfachsten Fall aus, nämlich von den Tagen 1 bis 28, mit denen bei gültigen Monats- und Jahresangaben kein ungültiges Datum erzeugt werden kann. Der entsprechende Bereich sei im Folgenden <gültiger Tag> genannt. Dementsprechend heißt der Bereich 1 bis 12 <gültiger Monat> und der Bereich 1583 bis 2399 <gültiges Jahr>. Die Äquivalenzklasse dazu ist

gÄK1: <gültiger Tag>.<gültiger Monat>.<gültiges Jahr> (»normales« Datum)

Jetzt bleiben noch die Tage 29., 30. und 31. abzuhandeln. Fangen wir mit den einfacheren von diesen an, also 30. und 31.:

gÄK2: 30.[4, 6, 9, 11].<gültiges Jahr>

Hier bezeichnen die eckigen Klammern ein geschlossenes Intervall. Es ist gemeint, dass für den Monat einer der Werte 4, 6, 9 oder 11 infrage kommt. Wenn 3-12 die Monate März bis Dezember meint, ist 30.[1,3-12].<gültiges Jahr> auch richtig. Aber im Sinne von »Lean Testing« wird angenommen, dass es nichts bringt, den 30. zu testen, wenn der 31. getestet wird (siehe unten gÄK3). Schließlich ist für jeden Monat, der 31 Tage hat, auch der 30. ein richtiges Datum.

gÄK3: 31.[1, 3, 5, 7, 8, 10, 12].<gültiges Jahr>

Nun kommen wir zu dem etwas schwierigeren Fall, dem 29. Dieser Fall spielt nur für den Monat Februar eine Rolle, und er ist nur im Schaltjahr ein gültiges Datum. Zur Erinnerung: Ein Jahr ist dann ein Schaltjahr, wenn die Jahreszahl sich ohne Rest durch 4, aber nicht durch 100 teilen lässt, es sei denn durch 400. So ist 1900 kein Schaltjahr, 2000 aber sehr wohl. Die Äquivalenzklasse ist:

gÄK4: 29.2.<Schaltjahr>

Ungültige Äquivalenzklassen zur Parameterkombination

Hier geht es um die Prüfung, ob die spezifizierten Grenzwerte überschritten werden. Aus den oben angegebenen Zahlenbereichen lassen sich die ungültigen Äquivalenzklassen (uÄK) leicht ableiten. Wegen des Datentyps `size_t` ist Min_Int = 0, und Max_Int die größtmögliche Zahl für diesen Datentyp.

uÄK1: 0.<gültiger Monat>.<gültiges Jahr>
uÄK2: [32..Max_Int].<gültiger Monat>.<gültiges Jahr>
uÄK3: <gültiger Tag>.0.<gültiges Jahr>
uÄK4: <gültiger Tag>.[13..Max_Int].<gültiges Jahr>
uÄK5: <gültiger Tag>.<gültiger Monat>.[0..1582]
uÄK6: <gültiger Tag>.<gültiger Monat>.[2400..Max_Int]

Es fehlen noch die Tage 29., 30. und 31., kombiniert mit den Monaten:

uÄK7: [30,31].2.<gültiges Jahr>
uÄK8: 31.[4,6,9,11].<gültiges Jahr>
uÄK9: 29.2.<kein Schaltjahr>

Die Frage ist, ob das Schaltjahr intern korrekt berechnet wird. Das wird bisher nicht berücksichtigt. Um dies zu prüfen, sind aus den obigen Äquivalenzklassen gÄK4 und uÄK9 mehrere Testfälle abzuleiten. Ein solches Vorgehen ist aber nicht konform zur Grundidee der Äquivalenzklassenbildung: Nur ein Repräsentant einer Klasse ist beim Testen zu berücksichtigen.

Wenn nun mehrere Repräsentanten einer Klasse zu wählen sind, könnten wichtige vergessen werden. Deswegen ergänzen wir die Äquivalenzklassen um zwei weitere:

gÄK5: 29.2.<Jahr, das ohne Rest durch 400 teilbar ist>

uÄK10: 29.2.<Jahr, das ohne Rest durch 100, aber nicht durch 400, teilbar ist>

Somit ergeben sich insgesamt 5 gültige und 10 ungültige Äquivalenzklassen. Sie werden in der folgenden Tabelle 4-5 zusammengefasst.

Äquivalenzklasse	Tag.Monat.Jahr
gÄK1:	<gültiger Tag>.<gültiger Monat>.<gültiges Jahr> (»normales« Datum)
gÄK2:	30.[4,6,9,11].<gültiges Jahr>
gÄK3:	31.[1,3,5,7,8,10,12].<gültiges Jahr>
gÄK4:	29.2.<Schaltjahr>
gÄK5:	29.2.<Jahr, das ohne Rest durch 400 teilbar ist>
uÄK1:	0.<gültiger Monat>.<gültiges Jahr>
uÄK2:	[32..Max_Int].<gültiger Monat>.<gültiges Jahr>
uÄK3:	<gültiger Tag>.0.<gültiges Jahr>
uÄK4:	<gültiger Tag>.[13..Max_Int].<gültiges Jahr>
uÄK5:	<gültiger Tag>.<gültiger Monat>.[0..1582]
uÄK6:	<gültiger Tag>.<gültiger Monat>.[2400..Max_Int]
uÄK7:	[30,31].2.<gültiges Jahr>
uÄK8:	31.[4,6,9,11].<gültiges Jahr>
uÄK9:	29.2.<kein Schaltjahr>
uÄK10:	29.2.<Jahr, das ohne Rest durch 100, aber nicht durch 400, teilbar ist>

Tabelle 4-5: Äquivalenzklassen für den Datumtest

Die Tabelle 4-5 ist die Basis zur Aufstellung der Testfälle. Wir haben zur Erstellung der Testfälle beispielsweise bei gÄK1 nahezu »freie« Auswahl, solange die Kombination von Tag, Monat und Jahr ein gültiges Datum ergibt. Bei der gÄK5 ist nur das Jahr »frei« wählbar, allerdings mit der Einschränkung, dass es ohne Rest durch 400 teilbar ist und da wir nur die Prüfung zwischen den Jahren 1583 und 2399 vornehmen, sind es nicht so viele – nämlich genau zwei (1600, 2000)! Die gedankliche Hauptarbeit ist geleistet, die Umsetzung in das Testprogramm nur noch reine Fleißarbeit. Die Testfälle sind zur Wiedererkennung wie oben mit den entsprechenden Äquivalenzklassen gekennzeichnet.

```
#include <gtest/gtest.h>
#include "istGueltigesDatum.h"

// gültige Äquivalenzklassen
TEST(istGueltigesDatumTest, gAeK1_normal) {
  EXPECT_TRUE(istGueltigesDatum(15, 3, 2016));
}

TEST(istGueltigesDatumTest, gAeK2_30) {
  EXPECT_TRUE(istGueltigesDatum(30, 9, 2018));
}

TEST(istGueltigesDatumTest, gAeK3_31) {
  EXPECT_TRUE(istGueltigesDatum(31, 8, 2017));
}

TEST(istGueltigesDatumTest, gAeK4_29_Schaltjahr) {
  EXPECT_TRUE(istGueltigesDatum(29, 2, 2016));
}

TEST(istGueltigesDatumTest, gAeK5_29_Schaltjahr400) {
  EXPECT_TRUE(istGueltigesDatum(29, 2, 2000));
}

// Äquivalenzklassen mit ungültigen Werten
TEST(istGueltigesDatumTest, uAeK1_Tag0) {
  EXPECT_FALSE(istGueltigesDatum(0, 5, 2019));
}

TEST(istGueltigesDatumTest, uAeK2_TagZuGross) {
  EXPECT_FALSE(istGueltigesDatum(1000, 5, 2019));
}

TEST(istGueltigesDatumTest, uAeK3_Monat0) {
  EXPECT_FALSE(istGueltigesDatum(1, 0, 1899));
}

TEST(istGueltigesDatumTest, uAeK4_MonatZuGross) {
  EXPECT_FALSE(istGueltigesDatum(1, 13, 1899));
}

TEST(istGueltigesDatumTest, uAeK5_JahrZuKlein) {
  EXPECT_FALSE(istGueltigesDatum(1, 12, 1580));
```

```
}

TEST(istGueltigesDatumTest, uAeK6_JahrZuGross) {
  EXPECT_FALSE(istGueltigesDatum(1, 12, 3580));
}

TEST(istGueltigesDatumTest, uAeK7_30_Februar) {
  EXPECT_FALSE(istGueltigesDatum(30, 2, 1752));
}

TEST(istGueltigesDatumTest, uAeK8_31_April) {
  EXPECT_FALSE(istGueltigesDatum(31, 4, 2087));
}

TEST(istGueltigesDatumTest, uAeK9_29_keinSchaltjahr) {
  EXPECT_FALSE(istGueltigesDatum(29, 2, 2025));
}

TEST(istGueltigesDatumTest, uAeK10_29_Jahr100) {
  EXPECT_FALSE(istGueltigesDatum(29, 2, 2100));
}
```

Listing 4.17: Programm nach Tabelle 4-5 *(beispiele/gueltigesDatum/dertest.cpp)*

Eine mögliche Implementierung der Funktion `istGueltigesDatum()` folgt abschließend in Listing 4.18. Der besseren Lesbarkeit halber wird die Feststellung, ob ein Jahr ein Schaltjahr ist, in eine eigene Funktion ausgelagert. Diese Funktion wird beim Test der aufrufenden Funktion und den genannten Testfällen mitgetestet.

```
bool istSchaltjahr(std::size_t jahr) {
  return (jahr % 4 == 0 && jahr % 100 != 0) || jahr % 400 == 0;
}

bool istGueltigesDatum(std::size_t tag, std::size_t monat,
                       std::size_t jahr) {
  bool ergebnis = false;
  if( (monat >=1 && monat <=12)
      && (jahr > 1582        // 1582: Beginn des gregorianischen Kalenders
          && jahr < 2400)    // Plausibilitätskontrolle
      ) {
    std::size_t letzterTagImMonat = 29; // nur Februar im Schaltjahr
    if (monat != 2 || !istSchaltjahr(jahr)) {   // sonst
      constexpr std::size_t tageImMonat[] {31, 28, 31, 30, 31, 30, 31,
                                           31, 30, 31, 30, 31};
```

```
      letzterTagImMonat = tageImMonat[monat - 1];
    }
    ergebnis = (tag >= 1 && tag <= letzterTagImMonat);
  }
  return ergebnis;
}
```

Listing 4.18: Mögliche Implementierung von `istGueltigesDatum()`

4.4 Grenzwertanalyse

Vermutlich haben Sie sich beim Durcharbeiten des Äquivalenzklassentests schon gefragt, ob es nicht eine sinnvollere Auswahl des Repräsentanten der Äquivalenzklasse gibt, als einfach einen beliebigen Repräsentanten zu wählen. Es gibt doch sicher spannendere Werte der Äquivalenzklasse! Ja, genauso ist es auch: Nutzen Sie die Grenzwertanalyse – der Name sagt eigentlich alles!

Wann ist der Einsatz sinnvoll?

Beim Äquivalenzklassentest wird davon ausgegangen, dass die Werte einer Äquivalenzklasse gleiches Programmverhalten bewirken. Interessant sind die Werte eines Eingabeparameters, bei denen sich durch eine kleine Änderung des Wertes das Programmverhalten ändert. Greifen wir das Beispiel der Klausurauswertung erneut auf. Bei 50 erreichten Punkten gilt die Klausur als (noch) »nicht bestanden«, bei 51 Punkten ändert sich das Programmverhalten und die Auswertung ergibt (knapp) »bestanden«. Tests mit den Werten 50 und 51 geben Auskunft, ob der Übergang, die Grenze, von »nicht bestanden« zu »bestanden« korrekt implementiert wurde.

Solche Übergänge von einer Äquivalenzklasse in die benachbarte (egal ob gültig oder ungültig) sind kritische Stellen, daher empfiehlt es sich, den Test auf diese Übergänge zu fokussieren, denn oft lassen sich an diesen Stellen Fehler nachweisen. Die Grenzwertanalyse ist eine sehr sinnvolle und anzuratende Erweiterung zum Äquivalenzklassentest. Sie kann aber nur bei Äquivalenzklassen verwendet werden, die eine geordnete Menge von Daten umfassen, wodurch sich Grenzwerte überhaupt erst ermitteln lassen. Im obigen Beispiel des Sportvereins mit den Sportarten Tischtennis, Turnen, Volleyball, Basketball, Handball und Fitnesstraining können alle Sportarten zu einer gültigen Äquivalenzklasse zusammengefasst werden. Es gibt aber keine wie auch immer geartete *Ordnung* zwischen den Sportarten und somit auch keine Grenzen.

Grundidee

Wenn eine Äquivalenzklasse einen Datenbereich definiert und der kleinste Wert der Äquivalenzklasse beim Test zum erwarteten Ergebnis führt und ebenso der größte, dann kann angenommen werden, dass auch die Werte zwischen dem kleinsten und größten Wert nicht zu einem anderen Verhalten führen. Die dazwischen liegenden Datenwerte bringen beim Test keine zusätzliche Information. *Anmerkung*: Dies trifft nur dann zu, wenn die Äquivalenzklassen korrekt nach der Spezifikation oder den Anforderungen aufgeteilt wurden.

Was ist eine Grenze?

Nehmen wir an, dass bei einem Konto Guthaben- und Überziehungszinsen ab dem ersten Cent zu berechnen sind. Bei 0,01 € ist der derzeit gültige Guthabenzins zu berechnen und dem Konto gutzuschreiben, bei 0,00 € ist keine Berechnung vorzunehmen und ab -0,01 € ist der Dispo-Satz anzuwenden. Im Beispiel haben wir auch gleich die drei sinnvollen Werte zur Prüfung der Grenze angegeben, also die Werte, die »ganz dicht« an der Grenze sind. Aber Vorsicht, das Beispiel ist nicht wirklich korrekt. Wie wir wissen, sind Äquivalenzklassen immer dann zu bilden, wenn sich das Programmverhalten ändert. Im Beispiel sind somit drei Äquivalenzklassen (um null herum) zu bilden: positiver Kontostand, Null-Kontostand, negativer Kontostand. Nun haben wir nicht eine, sondern zwei Grenzen zwischen den Äquivalenzklassen zu prüfen, aber wir benötigen keine weiteren Werte als die oben angegebenen. Null ist der einzige Repräsentant der Äquivalenzklasse und damit auch gleichzeitig »Grenzwert«. Der größte Wert bei negativem Kontostand und damit die obere zu testende Grenze ist -0,01 €. Wir prüfen den kleinsten Wert der Äquivalenzklasse mit Guthaben auf dem Konto mit 0,01 €. Die beiden anderen Grenzen – größter Wert der Äquivalenzklasse mit Guthaben und kleinster Wert (d.h. negativer Wert mit dem größten Absolutbetrag) der Äquivalenzklasse bei Kontoüberziehung – sind bei den bisherigen Überlegungen nicht berücksichtigt und somit noch separat zu prüfen.

Zwei oder drei Testfälle pro Grenze?

Unter Testern gibt es häufig die Diskussion, ob eine Grenze mit zwei oder mit drei Testfällen zu prüfen ist. Nehmen wir an, dass das Konto bis maximal -1000,00 € überzogen werden darf. Damit ergeben sich für den negativen Kontostand zwei Äquivalenzklassen:

Ist der Kontostand zwischen -0,01 und -1000,00 € einschließlich, wird der Dispozins berechnet. Ab -1000,01 € werden keine Auszahlungen und

Abhebungen mehr durchgeführt und die Bank sowie der Kontoinhaber werden informiert. Mit welchen Werten ist nun die Grenze zwischen den beiden Äquivalenzklassen zu testen? Wie wäre es mit -1000,00 als Grenzwert (aus der Spezifikation) sowie -999,99 und -1000,01 als kleinster bzw. größter Wert der jeweiligen Äquivalenzklasse. Aber, werden Sie einwenden, -999,99 bringt keine neue Information, weil bei -1000,00 das gleiche Verhalten erwartet wird – schließlich gehören beide Werte zur gleichen Äquivalenzklasse. Stimmt! Diese gewählte Vorgehensweise der Bestimmung der Grenzwerte ist eben recht einfach und naheliegend, da der Wert aus der Spezifikation genommen wird und die »rechts und links liegenden« (kleinstmöglichen) Werte zur Prüfung herangezogen werden. Nehmen wir an, dass die Spezifikation leicht verändert ist und »ab« -1000,00 € keine Auszahlung und Abhebung mehr erfolgen kann. Der genannte Grenzwert ist der gleiche. Nun bringt beim einfachen Verfahren der Ermittlung der drei Grenzwerte der Wert -1000,01 keine zusätzliche Information, da er bei dieser Spezifikation mit dem Wert -1000,00 in der gleichen Äquivalenzklasse liegt.

Was ist also zu tun? Bei der Programmierung muss sehr genau die Spezifikation gelesen und verstanden werden, um zu entscheiden, ab welchem Wert welches Verhalten vom Programm erwartet wird. Der Programmierer kennt also die Grenze, bei der sich das Programmverhalten ändert. Dann reichen zwei Werte aus, nämlich genau die, bei denen sich das Verhalten gerade ändert.

Es gibt aber noch Weiteres zu beachten. Sehen wir uns das letzte Beispiel nochmals an: »ab« -1000,00. Im Programm haben die folgenden beiden Abfragen die gleiche Wirkung:

```
if (berechneterKontostand <  -100000) ... // Cent, d.h. -1000,00 €
if (berechneterKontostand <=  -99999) ... // Cent, d.h. -999,99 €
```

Listing 4.19: Gleiche Wirkung zweier Abfragen

Aber es werden unterschiedliche Grenzwerte zur Entscheidung herangezogen. Egal welche Variante gewählt wurde, die Grenze ist mit -999,99 und -1000,00 zu testen.

Schlussfolgerung 1: Da wir als Entwickler den Programmcode kennen, reichen für den Test einer Grenze zwei Testfälle aus.

Schlussfolgerung 2: Mit drei Testfällen an jeder Grenze sind wir auf der sicheren Seite. Da wir unsere Testfälle automatisiert ablaufen lassen, ist der Aufwand vertretbar.

Bei der ersten Schlussfolgerung bezieht sich »lean« auf die Anzahl der Testfälle, bei der zweiten betrifft »lean« die Vorgehensweise (Grenzwert und rechts und links daneben). Eine ausführliche Diskussion, wann zwei Grenzwerte ausreichen und wann es sich empfiehlt, mit drei Werten eine Grenze zu prüfen, findet sich in »Basiswissen Softwaretest« [Spillner & Linz 12, Abschnitt 5.1.2].

> Die Grenzen aus der Spezifikation sind eins zu eins im Programmtext umzusetzen. Und mit drei Testfällen je Grenze ist man gut beraten!

4.4.1 Ein Beispiel

Für das aus Abschnitt 4.3 bekannte Klausurproblem lassen sich die Tests der Grenzwerte sehr einfach formulieren:

```
TEST(istBestandenTest, Grenzwert50) {
  EXPECT_FALSE(istBestanden(50));
}

TEST(istBestandenTest, Grenzwert51) {
  EXPECT_TRUE(istBestanden(51));
}
```

Listing 4.20: Test der Grenzwerte beim Klausurproblem

Testendekriterium

Bei den folgenden Überlegungen gehen wir davon aus, dass zwei Testfälle pro Grenze ausreichen. Wenn wir drei (eine gültige und zwei ungültige) Äquivalenzklassen haben und jede Äquivalenzklasse eine obere und eine untere Grenze hat, kommen wir auf 3·2 Grenzen. Da jede Grenze mit zwei Testfälle zu prüfen ist, wären insgesamt 12 Tests durchzuführen. Aber die Grenzen der benachbarten Äquivalenzklassen fallen zusammen oder besser »übereinander«. So ist zum Beispiel der größte Wert innerhalb einer Äquivalenzklasse derselbe Wert wie der kleinste Wert der benachbarten Äquivalenzklasse, der gerade außerhalb dieser Klasse liegt (siehe folgende Abbildung 4-1).

Damit ergeben sich pro Äquivalenzklasse zwei Grenzwerte (kleinster und größter Wert innerhalb der Äquivalenzklasse) ergänzt um die beiden Grenzwerte an den »äußeren« Rändern, sofern deren Werte überhaupt für Testfälle herangezogen werden können (s.o.). Das führt bei drei Äquivalenzklassen maximal zu 3·2 + 2 = 8 Testfällen. Diese acht (oder sechs) Testfälle können als Kriterium für das Testende herangezogen werden. Wenn alle Testfälle durchgeführt wurden, kann der Test der Grenzwerte der Äquivalenzklassen als ausreichend angesehen werden.

Hinweis: Werden die Ränder bzw. die Grenzen der jeweiligen Äquivalenzklassen geprüft, kann auf den Test mit einem beliebigen Repräsentanten (aus der »Mitte«) verzichtet werden. Es ist aber nicht verboten, weitere Test-

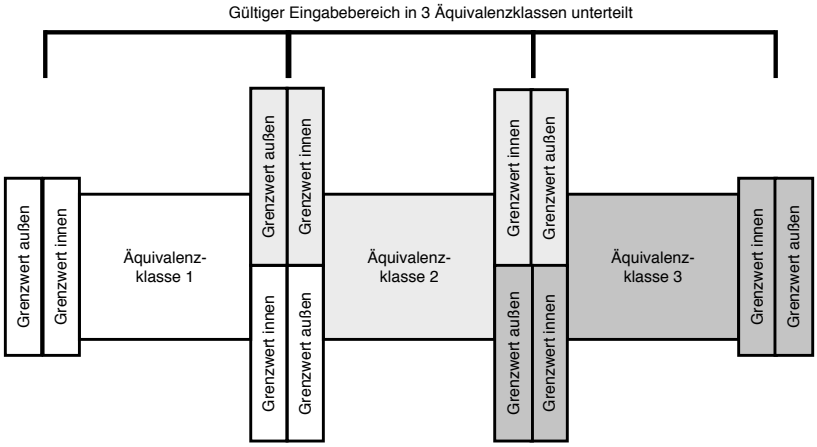


Abbildung 4-1: Zusammenfallende Grenzen benachbarter Äquivalenzklassen

fälle auch mit Nicht-Grenzwerten durchzuführen, wenn damit vermutlich kritische Eingabewerte überprüft werden.

Bewertung und Bezug zu anderen Testverfahren

Ist bei einem Testobjekt der Äquivalenzklassentest passend und lassen sich bei jeder Klasse obere und untere Grenzen festlegen, dann ist die Anwendung der Grenzwertanalyse uneingeschränkt zu empfehlen. Die Grenzen, bei denen durch eine kleine Änderung des Wertes sich eine Änderung im Programmverhalten ergibt, sind die »spannenden« Testfälle, da bei den Übergängen von einer Äquivalenzklasse in die benachbarte oft Fehler auftreten.

4.4.2 Mehrere Parameter

Wie sieht es nun mit der Grenzwertanalyse aus, wenn mehrere Parameter dem Testobjekt übergeben werden? Nun, da gibt es im Prinzip keinen Unterschied, ob ich einen beliebigen Repräsentanten aus einer Äquivalenzklasse für meinen Testfall verwende oder auf die jeweiligen Grenzwerte zurückgreife. Auch bei den Grenzwerten gilt, dass Werte aus ungültigen Äquivalenzklassen nicht mit anderen Werten aus ungültigen Äquivalenzklassen kombiniert werden sollen (Begründung siehe Abschnitt 4.3.4).

Ein einfaches Beispiel ist die Erweiterung der Testfälle zum Problem der Rabattberechnung (siehe Seite 46). Statt der Erweiterung könnte auch an den Ersatz einiger Testfälle gedacht werden. Ein Beispiel:

```
// Bis zu einem Rechnungsbetrag von 149,99 € gibt es zwischen 8 und 10 Uhr 5 %
TEST(RabattTest, gAeK31A_Morgens_149_99) {
```

```
   EXPECT_EQ(rabatt(14999, 9, 0), 750);       // 9.00 Uhr
}

// Ab 150,00 € gibt es 10 € (= 1000 Cent)
TEST(RabattTest, gAeK31B_Morgens_150_00) {
   EXPECT_EQ(rabatt(15000, 9, 0), 1000);
}
```

Listing 4.21: Untersuchung des Preissprungs

Hier wird der Sprung im Preis untersucht. Gleichzeitig wird der Sinn des Grenzwerttests deutlich: Wenn im Programm auf Seite 47 der Rabatt so berechnet würde:

```
std::size_t rabatt1 = rechnungsbetrag > 15000 ? 1000 : 0;
```

liefen die Tests von Seite 46 ohne Probleme durch! Erst der Grenzwerttest bringt an den Tag, dass es richtig

```
std::size_t rabatt1 = rechnungsbetrag >= 15000 ? 1000 : 0;
```

heißen muss (>= statt >).

Auf dieselbe Art kann der Sprung in der Zeit geprüft werden, hier von 10.00 Uhr (dann gilt noch der 5%-Rabatt) auf 10.01 Uhr (es gibt keinen Rabatt mehr):

```
// Bis 149,99 € gibt es zwischen 8 und 10 Uhr 5 %
TEST(RabattTest, gAeK31C_10Uhr_149_99) {
   EXPECT_EQ(rabatt(14999, 10, 0), 750);      // 10.00 Uhr
}
// Nach 10.00 Uhr (d.h. ab einschließlich 10.01 Uhr) gibt es nichts bei nur 149,99 €
TEST(RabattTest, gAeK31D_nach10Uhr_149_99) {
   EXPECT_EQ(rabatt(14999, 10, 1), 0);        // 10.01 Uhr
}
```

Listing 4.22: Untersuchung des Zeitsprungs

4.4.3 Ergänzung: Grenzen im Programmtext

Da wir Entwickler beim Testen unterstützen wollen, kann die Grenzwertanalyse auch auf den Programmtext übertragen werden: Auch hier gibt es Variablenwerte, die bei kleinen Änderungen ein unterschiedliches Verhalten bzw. einen unterschiedlichen Ablauf bewirken. Prüft eine Abfrage einen Wert, ob dieser größer oder gleich null ist, dann wird bei allen positiven Werten (und der null) der *Then*-Zweig durchlaufen und bei negativen Werten der *Else*-Zweig. Ein Wert knapp unterhalb von null (*Else*-Zweig) und der Wert null (*Then*-Zweig) sind nach der Grenzwertanalyse die Werte, mit denen der Test durchzuführen ist. Im Idealfall stimmen die Grenzen bei den

Äquivalenzklassen mit denen im Programmcode überein! Wenn nicht, muss beides noch mal geprüft werden, wie das obige Beispiel zeigt.

> Der Äquivalenzklassentest lässt sich bei sehr vielen Testüberlegungen anwenden. Durch die Grenzwertanalyse wird der Fokus bei der Auswahl der Repräsentanten der jeweiligen Äquivalenzklasse auf »spannende« Werte gelegt und »wichtige« Testfälle werden so nicht vergessen.

4.5 Klassifikationsbaummethode

Hatten Sie auch schon das Problem, dass viele unterschiedliche Kombinationen beim Testen zu berücksichtigen waren, und Sie nicht genau wussten, ob sie alle »wichtigen« getestet oder doch einige übersehen haben? Wenn die Übersicht verloren geht oder verloren zu gehen droht, dann ist die Klassifikationsbaummethode der richtige Ansatz, um das Problem gar nicht erst entstehen zu lassen.

Wann ist der Einsatz sinnvoll?

Beim Äquivalenzklassentest hatten wir schon das Problem der Kombinationen der Äquivalenzklassen untereinander, zum Beispiel bei der Bestimmung des korrekten Datums. Tag, Monat und Jahr waren zu kombinieren. Wenn es aber mehrere Kombinationsmöglichkeiten gibt und keine Abhängigkeiten vorhanden sind, ist der Äquivalenzklassentest eher ungeeignet, da er keine bestimmten Kombinationen erzwingt, die Repräsentanten ja frei wählbar sind. Bei der Datumsprüfung haben wir das Problem durch Vereinigung (Tag, Monat, Jahr) und weitere Aufteilung der Äquivalenzklassen gelöst.

Mit der Klassifikationsbaummethode [Grochtmann & Grimm 93] kann festgelegt werden, welche Kombinationen von Ein- und Ausgabebereichen[14] wie intensiv zu testen sind. Die grafische Darstellung bietet dabei einen sehr guten Überblick. Eine erste Strukturierung kann auf einen relativ hohen Abstraktionsniveau erfolgen. Der Klassifikationsbaum kann aber auch ganz konkret aufgebaut sein und einzelne Testfälle mit den gewählten Repräsentanten der Äquivalenzklassen darstellen. Wir konzentrieren uns auf die konkrete Darstellung, da wir den Entwickler beim Komponententest unterstützen wollen und hier Parameter und Parameterwerte im Fokus der Testüberlegungen stehen.

[14] Wir beschränken uns im Folgenden zur Vereinfachung auf die Darstellung der Eingabebereiche.

Grundidee

Die Klassifikationsbaummethode basiert auf der Spezifikation des Testobjekts. Der Eingabedatenraum des Testobjekts wird dabei nach Aspekten strukturiert, die als relevant eingeschätzt werden. Ein Aspekt oder Gesichtspunkt wird als *Klassifikation* bezeichnet. Auf der konkreten Ebene der Testfälle entsprechen die Klassifikationen den Parametern eines Testfalls und die Werte – oder genauer die Menge von Werten –, die ein Parameter annehmen kann, heißen in der Terminologie der Klassifikationsbaummethode *Klassen*. Klasse ist hier nicht als Begriff aus der objektorientierten Programmierung zu verstehen, sondern die Klassen entsprechen den oben beschriebenen Äquivalenzklassen. Zur besseren Strukturierung können Klassifikationen zu *Kompositionen* zusammengefasst werden.

4.5.1 Ein Beispiel

Zur Erläuterung der Klassifikationsbaummethode soll folgende Aufgabenstellung diskutiert werden: Nehmen wir an, es soll eine statistische Auswertung über die Studiendauer der Studierenden erstellt werden. Geschlecht, Eintrittsalter bei Studiumsbeginn und Herkunftsland des Studenten sollen ebenso erhoben und ausgewertet werden wie die Dauer seines Studiums und das Studienfach. Aufgabe in diesem Test sei jedoch nur, die implementierte Schnittstelle für das Statistikprogramm zu testen, nicht die Statistik selbst. Dazu werden verschiedene Daten von Studierenden in eine Datenbank geschrieben, und danach wird geprüft, ob die Daten korrekt gespeichert worden sind. Der Klassifikationsbaum wird hier auf einer sehr konkreten Ebene genutzt: Zur Ermittlung der Kombinationen einer zu testenden Schnittstelle mit fünf Parametern.

Auf eine präzise Spezifikation verzichten wir und verweisen auf die Abbildung 4-2 mit den Angaben der einzelnen Klassifikationen und Klassen. Zur besseren Strukturierung wurden die fünf genannten testrelevanten Aspekte (Klassifikationen) zu zwei Kompositionen zusammengefasst: Person und Studium.

Beim Geschlecht werden zwei Möglichkeiten[15], beim Eintrittsalter vier und beim Herkunftsland drei Klassen unterschieden. Studiendauer und Studienfach werden grob in jeweils drei Klassen unterteilt. In Abbildung 4-2 sind die fünf Klassifikationen (Kästchen mit dünnem Rand) mit ihren jeweiligen Klassen (Mengen von möglichen Parameterwerten) dargestellt.

Person und Studium sind die strukturierenden Kompositionen (Kästchen mit dickem Rand, die Wurzel des Klassifikationsbaums hat abgerundete Ecken).

[15] Wir wissen, dass diese Einteilung nicht gendergerecht ist, bleiben wegen der Einfachheit aber bei den beiden Möglichkeiten.

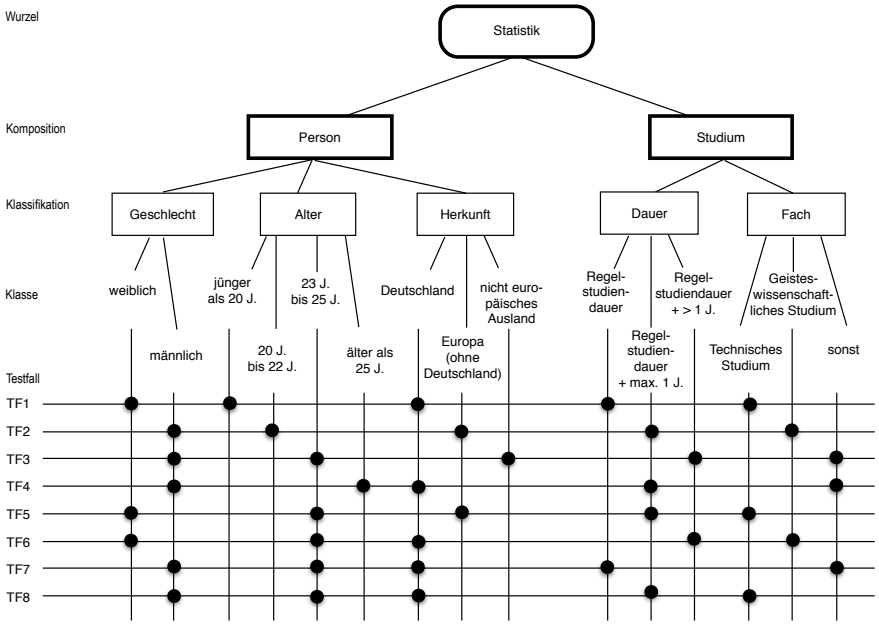


Abbildung 4-2: Beispiel mit fünf Klassifikationen

Die Testfälle (TF1-TF8) sind wie folgt zu interpretieren: Jede waagerechte Linie repräsentiert einen Testfall. Zu jeder Klassifikation muss auf dieser Linie ein (und nur ein) Auswahlpunkt aufgeführt sein, der die Belegung des Parameters wiedergibt. Im Beispiel gibt es fünf Parameter (Klassifikationen), somit müssen auf jeder Linie – in jedem Testfall – fünf Parameter mit Werten (Klassen) belegt werden. Welche Werte einer Klassifikation mit welchen Werten der anderen Klassifikationen kombiniert werden, ist in der grafischen Darstellung sehr gut ersichtlich.

In Abbildung 4-2 sind acht Testfälle durch Kombinationen aus den fünf Klassifikationen mit ihren insgesamt fünfzehn Klassen spezifiziert worden. Diese Auswahl der Testfälle erfolgte relativ willkürlich und dient in erster Linie für die folgende Diskussion:

- Alle Klassen sind in mindestens einem Testfall vertreten, da alle senkrechten Linien mit mindestens einem Auswahlpunkt versehen sind.
- Die Testfälle TF1 bis TF4 enthalten alle fünfzehn Klassen der fünf Klassifikationen. Eine erste, wenn auch sehr einfache Überdeckung ist mit diesen ersten vier Testfällen erreicht.
- Sehen wir uns die gewählten Testfälle, also die Klassen und deren Kombinationen, im Einzelnen an:

- Geschlecht: Ist mit drei Testfällen für weiblich und fünf Testfällen für männlich relativ gleichverteilt und daher unter Testgesichtspunkten eine akzeptable Auswahl.
- Alter: Hier gibt es ein klares Übergewicht der Studierenden, deren Eintrittsalter zwischen 23 und 25 Jahren liegt (fünf von acht Testfällen). Dies kann gewollt oder Zufall sein. Ob es sinnvoll ist, wäre zu entscheiden und ggf. zu ändern.[16]
- Herkunft: Auch hier ist ein klares Übergewicht für Deutschland als Herkunftsland gegeben, dies entspricht der Realität und ist daher ein guter Testansatz, da bei der späteren realen Nutzung des Programms auch viele Studierende mit Deutschland als Herkunftsland zu berücksichtigen sind.[17]
- Dauer und Fach: Beide sehen relativ gleich verteilt aus, was der Realität weitgehend entspricht.

Dank der grafischen Darstellung sind die eben getroffenen Erörterungen relativ einfach zu erkennen oder im wahrsten Sinne des Wortes »zu sehen«. Dies ist ein großer Vorteil der Klassifikationsbaummethode. Auch kann schnell geprüft werden, ob Testfälle Kombinationen von Klassen enthalten, die nicht kombiniert werden können. In unserem Beispiel ist dies nicht der Fall, da alle Klassen frei kombinierbar sind.

Die Aufgabe ist der Test der Schnittstelle zur Datenbank. Die Datenbank dient als Datenquelle für die statistische Auswertung, die nicht Teil unseres Tests ist. In der Statistik wird beispielsweise nicht differenziert, welches Alter der Student genau hat, sondern es wird lediglich der Altersbereich (zum Beispiel 20-22 Jahre) in die Datenbank eingetragen. Diese Information wird bei der statistischen Auswertung als ausreichend angesehen. In unserem Beispiel sind die Klassen daher ausreichend genau angegeben und können direkt für die Testfälle verwendet werden. Ein C++-Programm dazu – allerdings für ein anderes Testverfahren – finden Sie in Abschnitt 4.6.6.

Bei einer anderen Aufgabenstellung, zum Beispiel dem Test einer Eingabeschnittstelle zur Erfassung der Studentendaten, können die Angaben im Klassifikationsbaum als Äquivalenzklassen interpretiert werden. Dann muss zum Beispiel beim Alter ein Repräsentant – eine konkrete Zahl – für den Test ausgewählt werden, beispielsweise 21 für den Testfall TF2. Bei

[16] Das Durchschnittsalter der Studenten bei Studienbeginn liegt seit Jahren zwischen 20 und 21 Jahren und ist in den letzten Jahren durch den Wegfall des Wehrdienstes weiter gesunken. Das bei den Testfällen vorhandene Übergewicht spiegelt daher nicht die Realität wider.

[17] Die Testfälle wurden so gewählt, dass sie den beim Einsatz des Systems sehr häufig zu erwartenden Werten entsprechen. Es können aber auch andere Testfälle gewählt werden, um beispielsweise höchst seltene Kombinationen und deren korrektes Übertragen zur Datenbank zu prüfen.

der Klassifikation Geschlecht bestehen die beiden Klassen jeweils nur aus einem Repräsentanten (männlich bzw. weiblich), d.h., die Klasse im Klassifikationsbaum liefert direkt den konkreten Wert für den betreffenden Parameter unserer zu testenden Eingabeschnittstelle. Für die Klassifikation Alter bestehen die Klassen wieder aus Mengen von Werten. Aus den Mengen (Äquivalenzklassen) sind Repräsentanten zu wählen, sinnvollerweise auch Grenzwerte. Bei der Klasse Herkunft haben wir beide Formen gemischt: Deutschland ist der einzige Repräsentant der Klasse, wohingegen bei den beiden anderen Klassen noch konkrete Werte überlegt werden müssen (z.B. England, Australien). Entsprechendes gilt für die Klassen der Komposition Studium.

Der Klassifikationsbaum kann so weit verfeinert werden, dass direkt die konkreten Werte für den Test abgelesen werden können (wie in Abschnitt 4.5.2). So kann beispielsweise die bisherige Klasse Technisches Studium als weitere Klassifikation angesehen werden mit den Klassen Informatik, Elektrotechnik und Mechatronik (wenn diese die drei einzigen Studienrichtungen sind). Somit wären dann auch hier die konkreten Testdaten bereits im Klassifikationsbaum vorhanden.

Zurück zu unserem Beispiel: Reichen die in Abbildung 4-2 aufgeführten acht Testfälle aus? Schwer zu entscheiden. Minimal wären es ja vier und damit sind es schon doppelt so viele. Wir haben mit den Tests (nach Änderung und gleichmäßiger Verteilung des Alters) die Realität gut abgebildet. Eine vollständige Kombination aller Klassifikationen ergibt 2·4·3·3·3 = 216 Testfälle. Dann kann von »Lean Testing« wohl nicht mehr die Rede sein. Im folgenden C++-Beispiel werden wir die Diskussion, wie wir zu Lean-Testfällen kommen, fortführen. Im Unterschied zum vorangegangenen Beispiel gibt es Abhängigkeiten der beteiligten Klassen untereinander. Sie sind also nicht frei kombinierbar.

Testendekriterium

Bei der Klassifikationsbaummethode ist eine Abstufung der Kriterien möglich. Die Maximalforderung ist, dass für jede mögliche Kombination der Klassen aus unterschiedlichen Klassifikationen ein Testfall existiert. Minimal wäre, dass eine Klasse mindestens in einem Testfall geprüft werden muss. Die minimale Anzahl ergibt sich aus der höchsten Anzahl von Klassen einer Klassifikation. Auch unter dem Ziel »Lean Testing« sehen wir die minimale Anzahl von Testfällen als nicht immer ausreichend an. Ein angemessenes und »lean«-Vorgehen wird in Abschnitt 4.6.3 auf Seite 78 beschrieben.

Bewertung

Die Klassifikationsbaummethode ist sehr gut geeignet, wenn verschiedene testrelevante Aspekte auch in Kombination miteinander betrachtet werden müssen.

> Um bei der Vielzahl von Kombinationsmöglichkeiten noch die Übersicht zu behalten, hilft die grafische Darstellung der Klassifikationen, der Klassen und deren Kombinationen zu Testfällen. Bei jedem Testfall ist ersichtlich, aus welchen Klassen er zusammengesetzt ist. Ein einfacher, schneller Überblick ist gegeben.

Ebenso ist eine gute Beurteilung des Testumfangs möglich, da die Verteilung der Testfälle auf die Klassen geeignet gesteuert und nachvollzogen werden kann. Bei einer großen Anzahl von Klassifikationen, Klassen und Testfällen wird die grafische Darstellung unübersichtlich, aber durch die Werkzeugunterstützung (siehe Hinweise für die Praxis unten) bleibt sie noch handhabbar.

Sind die Klassifikationen unpassend oder falsch gewählt und sind bei der Zuordnung der Klassen zu den Klassifikationen Fehler unterlaufen, dann setzt die Methode auf falschen Voraussetzungen auf und liefert keine passenden Testfälle. Aber dieses Problem trifft für alle Testansätze zu: Wenn die Grundlagen nicht korrekt sind, werden keine korrekten Ergebnisse geliefert.

Bezug zu anderen Testverfahren

Auf den ersten Blick ähnelt die Methode sehr dem Äquivalenzklassentest. Allerdings gibt es beim Äquivalenzklassentest keine Unterstützung für die Kombination von Repräsentanten aus unterschiedlichen Äquivalenzklassen bei mehreren Parametern.

Darüber hinaus können mit der Klassifikationsbaummethode sehr gut Dateneingaben zu Testfällen bearbeitet werden, die keine geordneten Bereiche darstellen, wie beispielsweise die drei Unterscheidungen beim Herkunftsland der Studierenden. Auch sind beim Äquivalenzklassentest stets die ungültigen Äquivalenzklassen zu berücksichtigen. Bei der Klassifikationsbaummethode muss explizit entschieden werden, ob Parameter »ungültige Werte« annehmen können. Ungültige Werte sollen zu einer Fehlerbehandlung führen. Im Beispiel oben kann eine weitere Klasse für die Klassifikation »Alter« eingefügt werden, beispielsweise älter als 100 Jahre. In diesem Fall müssen dann Testfälle mit dieser Klasse zu einer Fehlermeldung führen.

Beim Äquivalenzklassentest können auch nutzlose Eingabekombinationen durch die zufällige Wahl der Repräsentanten entstehen (z.B. 31.2.2015 beim Datum). Bei der Klassifikationsbaummethode wird explizit ausgewählt, welche Klasse einer Klassifikation mit welchen anderen Klassen kombiniert werden soll und ob diese zulässig sind oder nicht. Unzulässige Kombinationen müssen dann zu Fehlermeldungen führen.

Die Klassifikationsbaummethode fokussiert eher auf die Prüfung der Funktionalität, der Äquivalenzklassentest, besonders in Kombination mit der Grenzwertanalyse, testet die Korrektheit an »Übergängen« und bei ungültigen Werten eher die Robustheit.

Hinweise für die Praxis

Wie bereits oben erwähnt, ist der Klassifikationsbaum werkzeuggestützt zu erstellen. Die Firma Berner & Mattner hat das Werkzeug TESTONA entwickelt, dessen einfache Version TESTONA Light[18] frei verfügbar ist. Für das folgende Beispiel wurde das Werkzeug von uns genutzt und damit wurden auch die Abbildungen 4-3 und 4-4 (ab Seite 70) erstellt.[19]

4.5.2 Das Beispiel in C++

Greifen wir ein bereits eingeführtes Beispiel zur Erläuterung auf: Ein Sportverein bietet die Sportarten Tischtennis, Turnen, Volleyball, Basketball, Handball und Fitnesstraining an. Jede Sportart ist aus organisatorischen Gründen einer Abteilung zugeordnet, wobei die Sportarten Volleyball, Basketball und Handball zu einer Abteilung Ballsport zusammengefasst werden. Damit wird auch den verschiedenen Kosten der Sportarten Rechnung getragen.

Der monatliche Mitgliedsbeitrag setzt sich zusammen aus einem Grundbetrag und einem Betrag für die in Anspruch genommene Abteilung (mindestens eine). Dabei kann mehr als eine Abteilung gewählt werden. Die monatlichen Zusatzbeiträge für die Abteilungen sind: Tischtennis 5 €, Turnen 11 €, Ballsport 9 €, Fitnesstraining 10 €. Die zu testende Anwendung soll den monatlichen Mitgliedsbeitrag berechnen.

Bei dem Grundbeitrag der Mitgliedsgebühr werden verschiedene Kategorien (Mitgliedsstatus) unterschieden:

[18] *http://www.testona.net/de/produktuebersicht/klassifikationsbaum-methode/index.html*

[19] Wir sind im Buch mit Empfehlungen für Testwerkzeuge bewusst zurückhaltend. Für alle Verfahren gibt es eine Reihe von unterstützenden Werkzeugen. Wir weisen auf entsprechende Werkzeuge hin, wenn ein Einsatz unbedingt anzuraten ist, da das Verfahren nur mit einer Werkzeugunterstützung sinnvoll durchgeführt werden kann.

- Kind oder Jugendlicher. Monatlicher Grundbeitrag: 7 €.
- Erwachsener. Monatlicher Grundbeitrag: 18 €.
- Ermäßigter Beitrag (für Studierende, Rentner und Sozialhilfeempfänger). Monatlicher Grundbeitrag: 8 €.
- Passives Mitglied. Monatlicher Grundbeitrag: 20 €.
  So ein Mitglied fördert den Verein durch seine Mitgliedschaft, ohne aktiv in einer der Abteilungen mitzuwirken.

Damit die Namen der Abteilungen und der Mitgliedsstatus flexibel änderbar sind, sind die Namen nicht fest einzuprogrammieren, sondern mit Strings zu realisieren. Die Implementierung in C++ könnte etwa so aussehen:

```
#ifndef MONATSBEITRAG_H
#define MONATSBEITRAG_H
#include <cstddef>
#include <map>
#include <stdexcept>
#include <string>
#include <set>
#include <utility>

class Monatsbeitrag {
public:
  Monatsbeitrag()
  /* Der Kürze wegen sind die Basisdaten hier fest einprogrammiert.
     Normalerweise würden sie durch Einlesen einer Konfigurationsdatei
     oder Datenbank initialisiert werden.
     Die Preise sind der Einfachheit halber in ganzen Euro.
  */
      : dieAbteilungen{{"Tischtennis", 5},
                       {"Turnen", 11},
                       {"Ballsport", 9},
                       {"Fitness", 10}},
                                                  // aktiv, Grundbeitrag
        mitgliedsstatus{{"Kind/Jugendlicher", {true, 7}},
                       {"Erwachsener", {true, 18}},
                       {"Ermaessigt", {true, 8}},
                       {"Passiv", {false, 20}}} {
        }

  // Die eigentliche Funktion zur Beitragsberechnung
  std::size_t beitrag(const std::string& status,
                      const std::set<std::string>& abteilungen =
                            std::set<std::string>()) const {
```

```
    // Prüfung auf bestimmte Fehler
    bool aktiv = mitgliedsstatus.at(status).first;
    if(aktiv && abteilungen.size() == 0) {
      throw std::logic_error("Aktiv ohne Abteilungszuordnung!");
    }
    if(!aktiv && abteilungen.size() > 0) {
      throw std::logic_error("Passiv mit Abteilungszuordnung!");
    }
    // Preisberechnung
    auto preis = mitgliedsstatus.at(status).second; // Grundpreis
    for (const auto& abt : abteilungen) {            // Zusatzpreis(e)
      preis += dieAbteilungen.at(abt);
    }
    return preis;
  }
private:
                  // < Name, Zusatzpreis >
  const std::map<std::string, std::size_t> dieAbteilungen;
                  // < Name, <aktiv ja/nein, Grundpreis> >
  const std::map<std::string, std::pair<bool, std::size_t>>
                mitgliedsstatus;
};
#endif
```

Listing 4.23: Mitgliedsbeitrag berechnen *(beispiele/sportverein/monatsbeitrag.h)*

Die Realisierung der Menge der Abteilungen als `set` (statt zum Beispiel als `vector`) garantiert, dass versehentliche Doppelnennungen eliminiert werden. Die eigentliche Berechnung ist sehr kurz: Der Grundpreis wird ermittelt, und auf ihn werden die Preise der gewählten Abteilungen addiert. Bei sich ändernder Zahl von Abteilungen oder Mitgliedsstatus oder bei Namensänderungen braucht die Funktion wegen der Realisierung mit Strings nicht geändert zu werden. Allerdings muss dann die Information, welcher dieser Strings »Passiv« bedeutet, den Werten mitgegeben werden. Eine Alternative ist, die Flexibilität nur bezüglich des passiven Status einzuschränken, sodass »Passiv« als einziges Schlüsselwort fest einprogrammiert wird.

Die einzelnen Abteilungen werden als relevant für den Test angesehen und bilden jeweils eine Klassifikation. Der Oberbegriff »Abteilungen« kann als strukturierende Komposition gesehen werden (siehe Abbildung 4-3). Zu jeder Klassifikation gehören zwei Klassen (Werte), nämlich ja und nein. Die Werte geben für ein bestimmtes Mitglied an, ob es in der betreffenden Abteilung aktiv ist. Dem Klassifikationsbaum können direkt die benötigten Testwerte entnommen werden, eine Auswahl von Repräsentanten einer Klasse muss somit nicht erfolgen.

Die zum Mitgliedsstatus gehörenden vier Klassen (Werte) sind Kind/Jugendlicher, Erwachsener, Ermäßigt, Passiv. Ein bestimmtes Mitglied kann nur zu einer der vier Klassen gehören.

Was ist nun beim Testen zu tun? Die Klassen der unterschiedlichen Abteilungen sind mit den unterschiedlichen Mitgliedsstatus zu kombinieren und entsprechende Testfälle daraus abzuleiten. Das heißt, durch die Kombination der Klassen unterschiedlicher Klassifikationen werden Testfälle definiert.

Die fünf Testfälle (TF1-TF5) in Abbildung 4-3 wurden zunächst spontan so gewählt, dass jede Klasse wenigstens einmal vorkommt. Es fehlt zum Beispiel ein Fall, in dem zwei oder mehr Abteilungen belegt werden. Auch sind die Testfälle nur eine sehr kleine Untermenge aller möglichen Kombinationen.

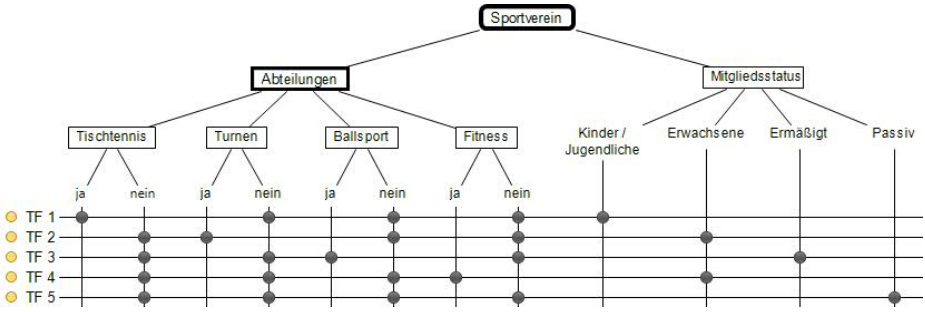


Abbildung 4-3: Klassifikationsbaum zum Sportverein

Welche Überlegungen sind anzustellen, um mit möglichst wenig Testfällen die Anwendung als ausreichend getestet anzusehen? Klar ist, dass jede Klasse in mindestens einem Testfall vorkommen muss.

Zuerst ist der Sonderfall »passives Mitglied« zu untersuchen. Hier bestehen Abhängigkeiten zu den anderen Klassen, da ein passives Mitglied zu keiner Abteilung zugeordnet werden kann. Damit haben wir schon einen wichtigen Testfall (TF5 in Abbildung 4-3). Sind Testfallkombinationen mit dem Mitgliedsstatus Passiv und einer (oder mehreren) Zuordnungen zu den Abteilungen erforderlich? Erwartet wird eine Fehlermeldung, da diese Kombination nicht den Anforderungen entspricht. Ein Testfall zur Prüfung der Ablehnung einer solchen fehlerhaften Kombination ist sinnvoll, sofern er überhaupt durchführbar ist. Auf diesen Punkt gehen wir später noch ein.

Drei unterschiedliche Mitgliedsstatus und vier Abteilungen, die untereinander beliebig kombiniert werden können, sind nun zu untersuchen. Aus den insgesamt 16 Möglichkeiten für die Belegung der Abteilungen kombiniert mit den 3 Status ergeben sich 48 Testfälle. In den 16 Kombinationen der Abteilungen ist auch die Kombination enthalten, die keine Sportart auswählt (viermal »nein«). Dies ist wieder ein Sonderfall, denn diese Kom-

bination entspricht dem Status Passiv. Es bleiben somit 15·3 = 45 Kombinationen von Abteilungszugehörigkeiten mit aktiven Mitgliedschaften – eine (zu) hohe Anzahl von Testfällen für »Lean Testing«.

Was muss eigentlich geprüft werden, was ist der »Kern« der Anwendung? Der Mitgliedsbeitrag soll ermittelt werden, also konzentrieren wir uns bei unseren Testüberlegungen auf diesen Aspekt. Der Beitrag ist abhängig vom Status und von der Wahl der Abteilungen.

Betrachten wir hierzu die Testfälle aus Abbildung 4-3: Die Testfälle TF1-TF4 prüfen jeweils eine Zugehörigkeit zu einer Abteilung und kombinieren diese relativ willkürlich mit einem Mitgliedsstatus, allerdings so, dass jeder Status in einem Testfall vorkommt. Testfall 5 ist der oben bereits diskutierte Sonderfall. Die Testfälle prüfen in keinem Fall eine Zugehörigkeit zu mehreren Abteilungen.

Müssen nun doch alle 45 Kombinationen geprüft werden? Nein, bei dieser Anwendung halten wir folgende Testüberlegungen für ausreichend:

- Ein Testfall TF6 ist zu erstellen, bei dem alle vier Abteilungen ausgewählt werden, kombiniert mit einem beliebigen aktiven Status, z.B. Kinder/Jugendlicher. Dieser Testfall prüft, ob die Zusatzbeiträge bei Nutzung mehrerer Abteilungen korrekt berechnet werden. Da jede Abteilung einen unterschiedlichen monatlichen Beitrag (5 €, 11 €, 9 €, 10 €) erfordert, kann aus dem Ergebnis des Testfalls (Summe aller Zusatzbeiträge (35 €) plus Statusbeitrag (7 €)) abgelesen werden, ob alle Werte korrekt addiert wurden.

  Aus der Summe kann aber nicht erkannt werden, ob nicht zwei Beiträge vertauscht wurden (z.B. Tischtennis mit 11 € statt 5 € und Turnen mit 5 € statt 11 €).
- Für jede Abteilung ist daher einzeln zu prüfen, ob der Zusatzbeitrag korrekt zugeordnet wurde. Wobei die Wahl des Mitgliedsstatus frei wählbar ist, aber alle in einem Testfall vorkommen sollen. Die Testfälle TF1-TF4 in Abbildung 4-3 erfüllen diese Anforderung.
- Hinzu kommt der Sonderfall TF5: Passives Mitglied und keine Abteilung gewählt. Damit wäre die Hauptfunktionalität der Anwendung mit sechs Testfällen geprüft.
- Ergänzend können noch die zwei zusätzlichen Testfälle TF7 und TF8 durchgeführt werden, die zu einer Fehlerbehandlung führen sollen: Ein passives Mitglied kombiniert mit einer oder mehreren Abteilungen und ein aktives Mitglied ohne Auswahl einer Abteilung. Abbildung 4-4 zeigt den Klassifikationsbaum, erweitert um die Testfälle TF6 - TF8.

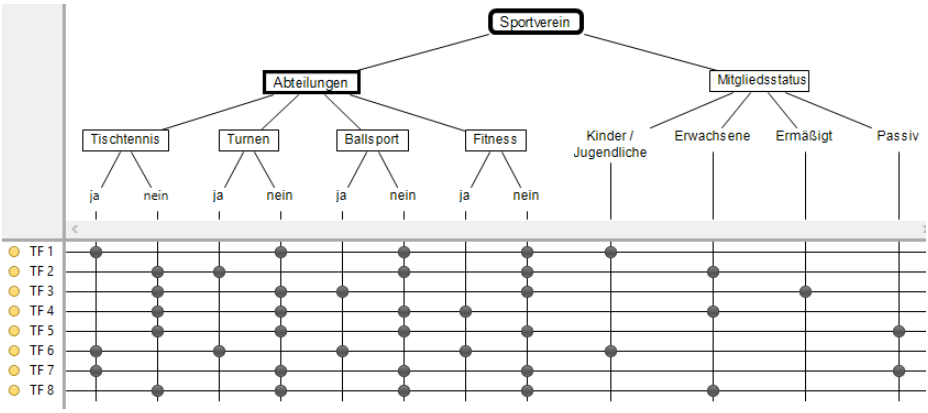


Abbildung 4-4: Klassifikationsbaum mit 8 Testfällen

Möglicherweise ist die Benutzungsschnittstelle so aufgebaut, dass die beiden hier aufgeführten Kombinationen für TF7 und TF8 gar nicht wählbar sind. Das kann etwa dadurch geschehen, dass als Vorbedingung der Funktion festgelegt wird, dass diese Fälle nicht auftreten dürfen. Es kann dann darauf verzichtet werden, die Eigenschaft aktiv/passiv den Mitgliedsstatus mitzugeben oder sie daraus zu ermitteln. Es ist dann Sache des Aufrufers, dafür zu sorgen (Stichwort »Design by Contract«, vgl. Seite 10). Das nachfolgende Listing geht davon aus.

```
class Monatsbeitrag {
public:
  Monatsbeitrag()
      : dieAbteilungen{{"Tischtennis", 5},
                       {"Turnen", 11},
                       {"Ballsport", 9},
                       {"Fitness", 10}},
        mitgliedsstatus{{"Kind/Jugendlicher", 7},
                       {"Erwachsener", 18},
                       {"Ermaessigt", 8},
                       {"Passiv", 20}} {
  }

  std::size_t beitrag(const std::string& status,
                      const std::set<std::string>& abteilungen =
                            std::set<std::string>()) const {
    auto preis = mitgliedsstatus.at(status); // Grundpreis
    for (const auto& abt : abteilungen) {     // Zusatzpreis(e)
      preis += dieAbteilungen.at(abt);
    }
```

```
    return preis;
  }
private:
  const std::map<std::string, std::size_t> dieAbteilungen;
  const std::map<std::string, std::size_t> mitgliedsstatus;
};
```

Listing 4.24: Vereinfachte Funktion ohne Fehlerprüfung

Bei dieser Vereinfachung entfallen die Testfälle TF7 und TF8. Die einzige verbleibende Fehlerprüfung ist die von C++: Wenn das Argument `x` der Funktion `at(x)` im Set nicht existiert, gibt es eine Exception. Alle genannten Testfälle sind nun leicht in C++ zu formulieren, wobei die Nummerierung aus dem Klassifikationsbaum beibehalten wird:

```
#include <gtest/gtest.h>
#include "monatsbeitrag.h"

TEST(monatsbeitragTest, TF1) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Tischtennis"};
  EXPECT_EQ(mb.beitrag("Kind/Jugendlicher", abt), 12);
}

TEST(monatsbeitragTest, TF2) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Turnen"};
  EXPECT_EQ(mb.beitrag("Erwachsener", abt), 29);
}

TEST(monatsbeitragTest, TF3) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Ballsport"};
  EXPECT_EQ(mb.beitrag("Ermaessigt", abt), 17);
}

TEST(monatsbeitragTest, TF4) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Fitness"};
  EXPECT_EQ(mb.beitrag("Erwachsener", abt), 28);
}

TEST(monatsbeitragTest, TF5_Passiv) {
  Monatsbeitrag mb;
  EXPECT_EQ(mb.beitrag("Passiv"), 20);
```

```
}

TEST(monatsbeitragTest, TF6_Summe) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Tischtennis", "Turnen",
                             "Ballsport", "Fitness"};
  EXPECT_EQ(mb.beitrag("Kind/Jugendlicher", abt), 42);
}

TEST(monatsbeitragTest, TF7_PassivMitAbteilung) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Fitness"};
  ASSERT_THROW(mb.beitrag("Passiv", abt), std::logic_error);
}

TEST(monatsbeitragTest, TF8_AktivOhneAbteilung) {
  Monatsbeitrag mb;
  ASSERT_THROW(mb.beitrag("Erwachsener"), std::logic_error);
}
```

Listing 4.25: Testfälle *(beispiele/sportverein/dertest.cpp)*

Mit insgesamt acht (oder sechs) Testfällen haben wir einen ausreichenden Test durchgeführt. Nun werden manche sagen, dass aber einiges noch nicht geprüft worden ist! Ja, das stimmt.

Wir stufen die Anwendung als unkritisch ein und sehen daher keine zwingende Notwendigkeit für einen umfassenden Test (alle Kombinationen der Abteilungen und der Mitgliedsstatus). Des Weiteren gehen wir davon aus, dass eine einfache Umsetzung der Aufgabe im Programmcode erfolgte, d.h., dass mehr oder minder »geradeaus« programmiert wurde. Es werden also sowohl der Grundbeitrag als auch der Zusatzbeitrag separat ermittelt und es wird die jeweilige Summe gebildet. Die richtige Zuordnung der Beiträge zu den Abteilungen und zum Status wird mit den aufgeführten Testfällen geprüft. Der in der obigen Aufzählung erste Testfall (d.h. TF6 auf Seite 71) prüft die Summenbildung bei der Zugehörigkeit zu mehreren (in dem Fall allen) Abteilungen. Wir gehen davon aus, dass auch die Summenbildung für zwei oder drei gewählte Abteilungen korrekt umgesetzt wurde.

Bei einer trickreichen Programmierung, die »schräge« Fehler verursachen könnte, reichen diese wenigen Testfälle nicht aus. Aber wir wollen ja mit dem Buch Testempfehlungen für den C++-Entwickler geben und erwarten einen guten und einfachen Programmierstil. Die Zeit der Programmier-»Künstler« scheint uns zu Recht schon lange vorbei zu sein.

4.6 Kombinatorisches Testen

Die Basis unserer bisherigen Überlegungen zur Kombination von Parametern waren die Spezifikation und unsere darauf aufbauenden Überlegungen. In diesem Kapitel werden wir die Mathematik zu Hilfe nehmen, um eine Auswahl von Kombinationen zu treffen.

Beginnen wir mit einem ganz einfachen Beispiel zur Motivation: Wir haben drei boolesche Parameter, die frei kombinierbar sind. Damit ergeben sich insgesamt die folgenden acht möglichen Kombinationen, dargestellt in der Tabelle 4-6.

Kombination	ParameterA	ParameterB	ParameterC
1	wahr	wahr	wahr
2	wahr	wahr	falsch
3	wahr	falsch	wahr
4	wahr	falsch	falsch
5	falsch	wahr	wahr
6	falsch	wahr	falsch
7	falsch	falsch	wahr
8	falsch	falsch	falsch

Tabelle 4-6: Vollständige Kombination von drei booleschen Parametern

Wenn wir »Lean Testing« praktizieren wollen, wie wäre es mit folgenden vier statt der vollständigen acht Kombinationen?

Kombination	ParameterA	ParameterB	ParameterC
1	wahr	wahr	wahr
4	wahr	falsch	falsch
6	falsch	wahr	falsch
7	falsch	falsch	wahr

Tabelle 4-7: Auswahl von vier Kombinationen

Sehen wir uns die vier Kombinationen näher an. Wenn wir jeweils nur zwei der drei Parameter betrachten, dann stellen wir fest, dass für die Paare ParamterA/ParameterB, ParameterB/ParameterC und ParameterA/ParameterC alle vier möglichen Kombinationen (wahr/wahr, wahr/falsch, falsch/wahr und falsch/falsch) in den vier Kombinationen der Tabelle 4-7 vorkommen. Anders formuliert: Welche zwei von den drei Spalten auch ausgewählt werden – alle vier Kombinationen kommen vor.

> Damit haben wir schon die Kernidee des kombinatorischen Testens verdeutlicht: Es sollen nicht alle möglichen Kombinationen über alle Parameter beim Testen berücksichtigt werden, sondern nur Kombinationen zwischen zwei, drei oder mehreren Parametern, diese dann aber vollständig.

Jede Zeile in den Tabellen ist als ein Testfall anzusehen. Die Werte geben vor, wie die Parameter zu belegen sind. Selbstverständlich muss, wie bei jedem anderen Testentwurfsverfahren auch, zu jedem Testfall noch das erwartete Ergebnis vor der Ausführung der Testfälle festgelegt werden.

Bei unserem Beispiel des Sportvereins gibt es vier Sportarten, die frei von den Vereinsmitgliedern gewählt werden können. Betrachten wir einmal nur die möglichen Kombinationen der Auswahl (ohne Berücksichtigung des Mitgliedsstatus), dann ergeben sich 16 mögliche Kombinationen. Mit sechs Testfällen kommen aber alle möglichen paarweisen Kombinationen (Tischtennis/Turnen, Tischtennis/Ballsport, Tischtennis/Fitness, Turnen/Ballsport, Turnen/Fitness und Ballsport/Fitness) mit ihren jeweiligen vier Möglichkeiten zur Ausführung. Die sechs Testfälle in Tabelle 4-8 enthalten alle paarweisen Kombinationen.

Test	Tischtennis	Turnen	Ballsport	Fitness
1	ja	nein	nein	nein
2	nein	ja	ja	ja
3	ja	ja	nein	ja
4	nein	nein	ja	nein
5	ja	ja	ja	nein
6	nein	nein	nein	ja

Tabelle 4-8: Auswahl von sechs Kombinationen der Sportarten

An diesem etwas umfangreicheren Beispiel ist gut zuerkennen, dass einige Kombinationen häufiger als andere enthalten sind. So kommen für das Paar Tischtennis/Turnen die Wertekombinationen ja/ja und nein/nein jeweils zweimal in den sechs Kombinationen vor, die beiden anderen Wertekombinationen nur einmal.

Auch wird deutlich, dass nicht alle »spannenden« Kombinationen berücksichtigt werden. Die Kombinationen keine Sportart (nein/nein/nein/nein) und alle Sportarten (ja/ja/ja/ja) kommen in der Tabelle 4-8 nicht vor.

Es gibt auch nicht nur diese eine Kombination mit der Eigenschaft, dass die jeweiligen Parameterpaare alle vier Kombinationen enthalten. Die folgende Tabelle 4-9 erfüllt ebenfalls diese Eigenschaft, umfasst aber eine Kombination weniger.

Test	Tischtennis	Turnen	Ballsport	Fitness
1	nein	nein	nein	nein
2	nein	ja	ja	ja
3	ja	nein	ja	ja
4	ja	ja	nein	ja
5	ja	ja	ja	nein

Tabelle 4-9: Auswahl von fünf Kombinationen der Sportarten

4.6.1 Orthogonale Arrays

Unser Eingangsbeispiel mit drei booleschen Parametern ist sehr einfach. In der Praxis sind viele Parameter mit nicht nur zwei möglichen Werten pro Parameter zu kombinieren. Die Auswahl der jeweiligen Kombinationen muss dann nach einem festgelegten Verfahren erfolgen. Grundlage sind die sogenannten orthogonalen Arrays. Ein orthogonales Array ist ein zweidimensionales Array mit folgenden speziellen mathematischen Eigenschaften: Jede Kombination von zwei beliebigen Spalten des Arrays enthält *alle* Kombinationen der Werte der beiden Spalten.

Orthogonale Arrays garantieren dabei zusätzlich eine Gleichverteilung der Kombinationen.[20] Jede Kombination kommt in einem orthogonalen Array *gleich oft* vor, egal welche zwei Spalten des Arrays ausgewählt werden. So sind die Tabellen 4-6 und 4-7 von Seite 75 orthogonale Arrays. Bei beliebigen zwei Spalten der Tabelle 4-6 kommt jede Wahr/falsch-Kombination exakt zweimal vor, siehe zum Beispiel das Spaltendoppel ParameterB/ParameterC (Zeilen 1/5, 2/6, 3/7 und 4/8). In Tabelle 4-7 kommt jede Kombination bei beliebigen zwei Spalten exakt einmal vor.

4.6.2 Covering Arrays

Covering Arrays entsprechen als Weiterentwicklung der orthogonalen Arrays ebenso der obigen Definition mit dem Unterschied, dass jede Kombination *mindestens einmal* vorkommt. Sie genügen deshalb nicht dem Anspruch der Gleichverteilung der Parameterwerte. Unter Testgesichtspunkten ist das aber kein gravierender Nachteil. Der Vorteil: Dieser Unterschied ermöglicht oft kleinere Arrays im Vergleich zu den orthogonalen Arrays. Covering Arrays werden als minimal (manchmal auch optimal) bezeichnet, wenn die Abdeckung der Kombinationen mit der geringstmöglichen Anzahl erfolgt. Das Beispiel mit den vier Sportarten würde ein orthogonales Array mit 8

[20] Ein Anwendungsgebiet der orthogonalen Arrays ist die statistische Versuchsplanung, da dort die zu untersuchenden Faktoren nicht miteinander vermengt werden dürfen und gleichverteilt sein sollen.

Zeilen (von maximal 16) erfordern, wenn zwei beliebig ausgewählte Spalten alle Wahr/falsch-Kombinationen enthalten sollen (jede käme zweimal vor).

Der Verzicht auf die Einschränkung, dass jede Kombination gleich oft vorkommen muss, führt zu dem Covering Array in Tabelle 4-8. Das Array kann noch verkleinert werden: Die Tabelle 4-9 zeigt ein minimales Covering Array. Es ergeben sich somit 5 (oder 6) Kombinationen bei den Covering Arrays im Gegensatz zu den 8 erforderlichen Kombinationen bei Nutzung der orthogonalen Arrays.

4.6.3 n-weises Testen

»Paarweises Testen« oder »Pairwise Testing« beschränkt sich auf alle diskreten Kombinationen aller Paare (2er-Tupel) der Parameterwerte, d.h., das n aus der Überschrift ist 2. Beide obigen Beispiele zeigen paarweise Kombinationen. Beim paarweisen Testen wird eine Gleichverteilung nicht garantiert, sondern nur dass jede Kombination (jedes Paar) mindestens einmal vorkommt. Ziel des paarweisen Testens ist die Entdeckung aller Fehler, die auf der Interaktion zweier Parameter beruhen. Werden drei oder mehr möglicherweise wechselwirkende Parameter in den Kombinationen berücksichtigt, dann wird das Vertrauen in die Tests weiter verstärkt.

Damit sind wir beim »n-weisen Testen«. Ermittelt werden alle möglichen diskreten Kombinationen aller n-Tupel von Parameterwerten. Zur Verdeutlichung (und zum Vergleich zu den bisherigen 2er-Kombinationen) sind in Tabelle 4-10 alle 3er-Kombinationen für die Wahl der Sportart aufgeführt.

Test	Tischtennis	Turnen	Ballsport	Fitness
1	nein	nein	nein	ja
2	nein	nein	ja	nein
3	nein	ja	nein	nein
4	nein	ja	ja	ja
5	ja	nein	nein	nein
6	ja	nein	ja	nein
7	ja	ja	nein	ja
8	ja	ja	ja	nein
9	ja	nein	ja	ja

Tabelle 4-10: Covering Array aller 3er-Kombinationen der Sportarten

Wie man sieht, sind mehr Kombinationen erforderlich. Bei vier Parametern ist n=4 das Maximum und es gäbe eine Tabelle mit allen $2^4 = 16$ möglichen Kombinationen, also eine vollständige Überdeckung. Daher wird n auch als *Stärke* eines Covering Arrays bezeichnet.

Wir wollen aber nicht zu tief in den mathematischen Hintergrund einsteigen, sondern uns mehr um die praktische Anwendung der Verfahren

kümmern. Daher konzentrieren wir uns auf das n-weise Testen, da es das paarweise Testen (2er-Kombination) umfasst, aber durch die weiteren Kombinationsmöglichkeiten darüber hinausgeht.

Was haben Covering Arrays und n-weises Testen mit »Lean Testing« zu tun? Die Antworten:

- Mit ihrer Hilfe kann die Anzahl der Testfälle teilweise drastisch reduziert werden. Oben sehen wir die Reduktion von 16 auf 5 Testfälle für das paarweise Testen. Wenn die Anzahl der möglichen Kombinationen sehr groß ist, wirkt die Reduktion noch stärker. So wird in [Kuhn et al. 10, S. 16] von einem Beispiel mit 172.800 möglichen Kombinationen berichtet, das bei n = 2 (paarweises Testen) in nur 29 (= 0,02 %) und bei n = 5 in nur 2532 (= 1,5 %) Testfällen resultierte.
- Bisher sind wir davon ausgegangen, dass ein Parameter nur die Werte wahr oder falsch annehmen kann. Es kann aber sein, dass das nur für einige Parameter zutrifft, andere aber einen von drei, vier oder noch mehr möglichen Werten annehmen können. Orthogonale Arrays lassen sich dann teilweise nicht mehr konstruieren, Covering Arrays schon.

Wann ist der Einsatz sinnvoll?

Bei den bisher vorgestellten Testverfahren mit mehreren Parametern, die zu kombinieren sind, gab es zwei Möglichkeiten:

- Jeder Parameterwert soll mindestens in einem Testfall verwendet werden. Die maximale Anzahl von Testfällen ist dann gleich der Anzahl der Parameterwerte des »größten« Parameters (der mit den meisten unterschiedlichen Werten).
- Alle möglichen Kombinationen der Werte der Parameter sollen bei den zu spezifizierenden Testfällen berücksichtigt werden. Oft ist der damit verbundene Aufwand, verursacht durch die kombinatorische Explosion bei vielen Parametern, aber nicht gerechtfertigt oder auch gar nicht leistbar.

Beide Möglichkeiten lassen sich gut veranschaulichen und erklären, dabei wird oft der Begriff »Testabdeckung« verwendet. Wenn jeder Parameterwert mindestens in einem Testfall vorkommt, dann ist eine sehr einfache Testabdeckung erreicht: Jeder Parameterwert wird beim Testen verwendet. Bei unserem Beispiel mit den drei booleschen Parametern sind die Kombinationen 1 und 8 (oder auch 2 und 7, 3 und 6 sowie 4 und 5) aus der Tabelle 4-6 zu verwenden, um diese Testabdeckung vollständig zu erfüllen. Um alle Kombinationen abzudecken, sind alle acht Kombinationen aus der Tabelle 4-6 zur Erstellung von Testfällen heranzuziehen. Die vollständige Testabdeckung wird angestrebt und dient als Endekriterium. Der Test wird als ausreichend

genug angesehen und kann daher beendet werden. Testabdeckungen sind für das jeweilige Testverfahren festzulegen.

Schon bei der Klassifikationsbaummethode wurde darauf hingewiesen, dass die minimale Testabdeckung zu »lean« ist und daher um weitere Testfälle ergänzt werden soll. Diese zusätzlichen Kombinationen können so gewählt werden, dass sie unter Testgesichtspunkten eine sinnvolle Ergänzung darstellen. In Abbildung 4-2 auf Seite 63 decken die ersten vier Testfälle jeden Wert der jeweiligen Parameter ab. Die Testfälle TF5 bis TF8 sind relativ willkürlich ausgewählt und hinzugenommen worden. Die insgesamt acht Testfälle sind nicht systematisch hergeleitet worden und deshalb lässt sich auch keine nachprüfbare Testabdeckung ermitteln.

Mit dem in diesem Kapitel vorgestellten Vorgehen gibt es einen strukturierten Ansatz, zu Testfällen zu kommen, deren Anzahl zwischen den beiden oben aufgeführten Möglichkeiten liegt. Wie umfangreich die Parameterwerte miteinander kombiniert werden sollen (2er-, 3er-, 4er-, ... Kombinationen), liegt im Ermessen des Prüfenden und hängt von der Kritikalität des Testobjekts ab. Je risikoreicher die Auswirkungen eines möglichen Fehlers sind, desto intensiver soll das Testobjekt getestet werden. Darüber hinaus werden die Testfälle durch das Verfahren so konstruiert, dass immer eine bestimmte Kombination der Parameterwerte vollständig erreicht wird. Die anzustrebende Testabdeckung lässt sich dadurch festlegen. Wenn z.B. alle 3er-Kombinationen der Sportarten getestet werden sollen, sind alle 9 Testfälle aus der Tabelle 4-10 auf Seite 78 durchzuführen. Dann ist die geforderte Testabdeckung vollständig erfüllt.

Wahrscheinlich werden Sie sich schon gefragt haben, nach welcher mathematischen Formel Sie die ganzen Kombinationen erzeugen sollen. Dafür gibt es Werkzeuge, die Ihnen die Arbeit erleichtern! Auf deren Verwendung wird weiter unten eingegangen.

Grundidee

> Das kombinatorische Testen bietet einen Mittelweg nach Lean-Kriterien zwischen den eher unbefriedigenden Ansätzen, jeden Parameterwert in nur einem Testfall zu prüfen, und der vollständigen Kombination aller Möglichkeiten, vor allem wenn es sehr viele davon gibt.

Schon ein Gerät oder eine Software, bei der eine Konfiguration mit 15 booleschen Parametern möglich ist, ist bei den sich daraus ergebenden 32.768 Möglichkeiten kaum vollständig zu testen. Wir beschränken uns allerdings auf einfachere Beispiele, um die Nachvollziehbarkeit zu erleichtern.

Der Mittelweg bietet durch seine systematische Herleitung der Kombinationen die Möglichkeit, eine Testabdeckung zu dokumentieren. Darüber hinaus ist von Vorteil, dass »aufbauende Stufen« vorhanden sind. Die kleinste Stufe ist der paarweise Test (2-weiser Test), bei dem alle 2er-Kombinationen der jeweiligen Parameterwerte miteinander kombiniert werden, und dies vollständig. Die nächste Stufe besteht darin, alle 3er-Kombinationen (3-weiser Test) mit Testfällen abzudecken. Auch hier garantiert das Verfahren die Vollständigkeit der 3er-Kombinationen.

Bisher sind wir davon ausgegangen, dass es keine Abhängigkeiten zwischen den Parametern und deren Werten gibt und alles somit frei kombinierbar ist. In der Realität ist dies aber nicht immer der Fall. Beispielsweise darf bei unserem Sportverein das passive Mitglied keine Sportart wählen und aktive Mitglieder müssen mindestens eine Sportart belegen. Diese Abhängigkeiten sind bei den Testfällen zu berücksichtigen. Dies kann entweder durch Nachbearbeitung der Testfälle erfolgen oder die Abhängigkeiten werden beim Werkzeug eingestellt und dann vom Werkzeug bei der Erstellung der Kombinationen berücksichtigt. Ebenso sind die Kombinationen zu streichen, die in der Praxis gar nicht vorkommen können.

Testendekriterium

Beim n-weisen Test ergeben sich die Kriterien zur Beendigung des Testens durch die gewählte Anzahl von Kombinationen. Je höher das n ist, desto mehr Kombinationen und damit Testfälle sind zu berücksichtigen. Damit eine 100%ige Testabdeckung erreicht wird, sind alle Testfälle durchzuführen. Vorab sind die nicht möglichen Kombinationen aus den Testfällen zu streichen. Eine 100%ige Testabdeckung wird oft nicht durchführbar oder zu aufwendig sein. Als Alternative bietet es sich an, n zu erhöhen, bis auf dieser nächsthöheren Stufe keine Fehler mehr entdeckt werden. Diese Maßnahme ist eher beim Test von kritischen System(teil)en anzuraten, da sie doch um einiges aufwendiger als ein paarweiser Test ist. Das Vorgehen beim n-weisen Testen könnte wie folgt aussehen:

1. n = 2 (paarweises Testen). Die Anzahl der Parameter sei N.
2. Erstelle Testfälle, die alle n-er-Kombinationen abdecken.
3. Wenn es Fehler gibt, beseitige diese und teste erneut die Kombinationen, bis die Testfälle fehlerfrei laufen.
4. Wenn n = 2 ist, erhöhe n um 1 und fahre fort bei Punkt 2.
5. Wenn auf der vorhergehenden und auf dieser Stufe keine Fehler mehr gefunden werden oder wenn n = N ist, brich den n-weisen Test ab. Andernfalls erhöhe n um 1 und fahre fort bei Punkt 2.

Bewertung

Das kombinatorische Vorgehen zur Erstellung der Testfälle ist einfach nachvollziehbar und überzeugend. Es gibt dem Tester die Sicherheit, wenn nicht alle möglichen Kombinationen, so doch einen systematisch hergeleiteten Ausschnitt beim Testen zu berücksichtigen. Es bietet vom Aufwand her abgestufte Möglichkeiten (2er-, 3er-, ... Kombinationen) von der einfachen Testabdeckung, dass jeder Parameterwert in mindestens einem Testfall vorkommt, hin zu der vollständigen Kombination aller Parameterwerte.

Durch die systematische Herleitung der Kombinationen entstehen Testfälle, die in der Praxis gar nicht vorkommen können. Diese sind herauszufiltern. Ebenso sind ggf. Abhängigkeiten zwischen den Parametern zu berücksichtigen und die erzeugten Testfälle entsprechend anzupassen.

Man soll dem Verfahren aber nicht zu euphorisch gegenübertreten. Die Mathematik hilft einem zwar dabei, alle abgestuften Kombinationen zu berücksichtigt, aber weitere Testfälle sind als Ergänzung hinzuzunehmen. Sehen wir uns dazu die Tabellen 4-9 mit den 2er-Kombinationen und Tabelle 4-10 mit den 3er-Kombinationen der vier Sportarten unter Testgesichtspunkten näher an.

2er-Kombinationen (Tabelle 4-9 auf Seite 77):

- Es gibt keinen Testfall, bei dem alle vier Sportarten ausgewählt werden.
- Es gibt neben dem Testfall 1 (keine der vier Sportarten ausgewählt, im konkreten Beispiel besteht hier auch eine Abhängigkeit zum Mitgliedsstatus) nur Testfälle, bei denen jeweils eine Sportart nicht ausgewählt ist, also immer drei Sportarten gewählt sind.
- Kombinationen mit einer oder zwei ausgewählten Sportarten kommen nicht vor.

3er-Kombinationen (Tabelle 4-10 auf Seite 78):

- Es gibt keinen Testfall, bei dem alle vier Sportarten ausgewählt sind.
- Es gibt keinen Testfall, bei dem keine der vier Sportarten ausgewählt wird (Abhängigkeit s.o.).
- Es gibt nur einen einzigen Testfall, bei dem zwei Sportarten ausgewählt sind: Testfall 6 mit ja/nein/ja/nein. Andere Kombinationen mit zwei Sportarten kommen nicht vor.

Trotz dieser »Nachteile« ist das n-weise Testen zur Kombination vieler Parameterwerte zu empfehlen. Der Tester muss sich über die Vor- und Nachteile des Verfahrens im Klaren sein und ggf. Ergänzungen der Testfälle vornehmen.

Generell gilt auch hier: Es gibt kein Testverfahren, das alle Fehler findet. Es muss immer eine sinnvolle Auswahl und Zusammenstellung von Testverfahren passend zum Testobjekt gewählt werden.

Bezug zu anderen Testverfahren

Das n-weise Testen ist das einzige Testverfahren, das die Kombination der Parameterwerte systematisch überprüft. Bei den anderen Testverfahren wie Äquivalenzklassentest oder Klassifikationsbaummethode wird keine systematische Herleitung von Kombinationen gefordert.

Bestehen zwischen den Parameterwerten eines Testobjekts nur wenige Abhängigkeiten, sind sie also weitgehend »frei« miteinander kombinierbar, dann ist das n-weise Testen der passende Ansatz. Besonders, wenn vermutet wird, dass durch (beliebige) Kombinationen Fehler verursacht werden können.

4.6.4 Werkzeugnutzung

Wir verwenden hier wieder das Beispiel »Sportverein«, berücksichtigen aber den Mitgliedsstatus »Passiv« und die Bedingung, dass ein nicht passives Mitglied mindestens einer Sportart zugeordnet ist. Das Werkzeug *Advanced Combinatorial Testing System (ACTS)* des amerikanischen National Institute of Standards and Technology [URL: ACTS] erzeugt uns ein Covering Array.[21] Das in Java geschriebene Werkzeug ist kostenlos erhältlich. Es wird mit folgender Anweisung gestartet:

```
java -jar acts_gui_2.92.jar
```

Nach Eingabe der Parameter, der zu berücksichtigenden Bedingungen (Konfiguration) und nach Start der Auswertung zeigt das Programm das Ergebnis an (siehe Abbildung 4-5).

Die Stärke (Strength) ist oben im Bild angegeben und kann entsprechend gesetzt werden. 2 bedeutet paarweises Testen. Wenn 3 eingestellt wird, gibt das Programm 19 Testfälle aus. Die maximale Stärke ist 5 (Anzahl der Parameter). Damit ergeben sich 46 Testfälle, also sämtliche Kombinationsmöglichkeiten. Diese setzen sich zusammen aus den auf Seite 70 berechneten 45 Möglichkeiten plus dem Fall des Mitgliedsstatus »Passiv«, dem keinerlei Sportarten zugeordnet sind.

Das Programm speichert die Konfiguration wahlweise in einer XML- oder einer Textdatei. Letztere ist hier auszugsweise abgedruckt,[22] weil sie

[21] Dazu sind natürlich auch andere Werkzeuge fähig, wie etwa das schon erwähnte TESTONA.

[22] Aus Layoutgründen leicht umformatiert.

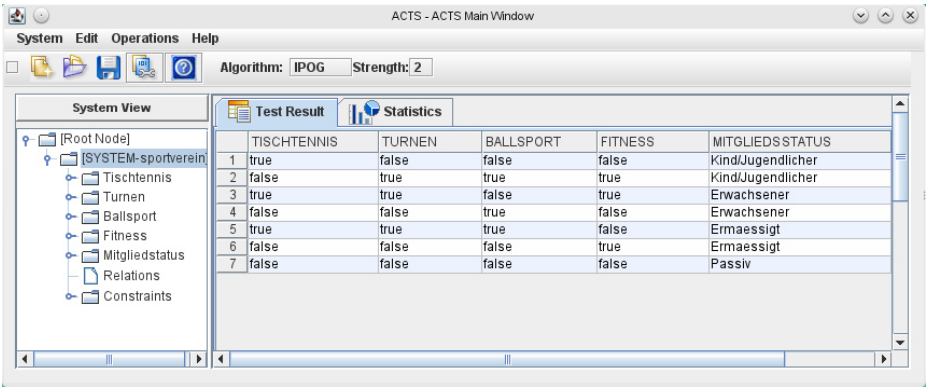


Abbildung 4-5: Covering Array für das Sportverein-Beispiel unter Berücksichtigung des Mitgliedsstatus

lesbarer ist und leichter zur Generierung von Tests ausgewertet werden kann.

```
[Constraint]
Mitgliedsstatus = "Passiv" => Tischtennis=false && Turnen=false
                          && Ballsport=false && Fitness=false
Mitgliedsstatus != "Passiv" => Tischtennis=true || Turnen=true
                            || Ballsport=true || Fitness=true
```

Listing 4.26: Bedingungen der gespeicherten Konfiguration

Die erste Bedingung drückt aus, dass dem Mitgliedsstatus »Passiv« keine Sportart zugeordnet ist. Die zweite besagt, dass wenigstens eine Sportart gewählt sein muss, wenn der Mitgliedsstatus ungleich »Passiv« ist.

Die vom Werkzeug ermittelten sieben Kombinationen der vier Sportarten und des Mitgliedsstatus sehen in der Textform wie folgt aus:

```
[Test Set]
Tischtennis,Turnen,Ballsport,Fitness,Mitgliedsstatus
true,false,false,false,Kind/Jugendlicher
false,true,true,true,Kind/Jugendlicher
true,true,false,true,Erwachsener
false,false,true,false,Erwachsener
true,true,true,false,Ermaessigt
false,false,false,true,Ermaessigt
false,false,false,false,Passiv
```

Listing 4.27: Ermittelte Kombinationen für das paarweise Testen

Das genannte Werkzeug bietet ein Java-API, sodass die entsprechenden Funktionen und Klassen dazu verwendet werden können, für Google Test

geeignete Testfälle daraus zu generieren. Wir haben jedoch festgestellt, dass der dafür notwendige Programmcode länger wird als eine auf diesen Spezialfall zugeschnittene Auswertung mit einem C++-Programm. Und es gibt noch ein weiteres Problem, wie der nächste Abschnitt zeigt.

Hinweise für die Praxis

Das oben genannte »Testset« ist tatsächlich *keine* Menge von Testfällen, denn es fehlt etwas ganz Entscheidendes: Zu jeder Kombination fehlt noch die Angabe, welches Ergebnis beim Test erwartet wird. Denn nur durch den Vergleich des berechneten Werts mit dem erwarteten lässt sich beurteilen, ob die Software richtig arbeitet. Man muss also das erwartete Ergebnis vor der Berechnung *voraussagen* können. Deswegen heißt so ein Wert in Anlehnung an das Orakel von Delphi[23] *Testorakel*. Das Orakel ergibt sich in aller Regel direkt oder indirekt aus der Spezifikation für die Software.

Um hier das jeweilige Orakel hinzuzufügen, wird der »Testset«-Teil in eine Datei *testdaten.txt* kopiert. Der erwartete Zahlenwert wird bei jeder Kombination am Ende der Zeile angehängt.

```
Tischtennis,Turnen,Ballsport,Fitness,Mitgliedsstatus
true,false,false,false,Kind/Jugendlicher,12
false,true,true,true,Kind/Jugendlicher,37
true,true,false,true,Erwachsener,44
false,false,true,false,Erwachsener,27
true,true,true,false,Ermaessigt,33
false,false,false,true,Ermaessigt,18
false,false,false,false,Passiv,20
```

Listing 4.28: Testdaten *(beispiele/sportvACTS/generator/testdaten.txt)*

4.6.5 Das Beispiel in C++

Daraus kann anschließend »zu Fuß« die C++-Datei zum Testen erzeugt werden. Bei wenigen Einträgen ist das der einfachste Weg! Nur bei vielen Einträgen oder häufigeren Änderungen lohnt es sich, die Datei mit den Testdaten automatisch auszuwerten. Das folgende Programm erzeugt aus den obigen Testdaten die Datei *dertest.cpp*, mit der der Test durchgeführt werden kann.

```
#include <fstream>
#include <iostream>
```

[23] Delphi war ein Ort im antiken Griechenland. Der griechischen Mythologie nach stiegen dort Dämpfe aus einer Erdspalte, die die Priesterin Pythia in Trance versetzten, sodass sie Prophezeiungen (Orakel) von sich gab.

```
#include "../../util/split.h" // String mit Trennzeichen in einen Vector
                              // aufsplitten

int main() {
  std::ifstream testdaten("testdaten.txt");   // Input
  std::ofstream ergebnis("../dertest.cpp");    // Output
  ergebnis << "// Von generator/generator.exe generierte Datei!\n"
    "#include <gtest/gtest.h>\n#include \"monatsbeitrag.h\"\n";
  std::string zeile;
  std::getline(testdaten, zeile);
  auto faktoren = split(zeile, ",");
  std::size_t testfall = 0;

  while(testdaten.good()) {
    std::getline(testdaten, zeile);
    if(zeile.length() > 3) {
      ergebnis << "\nTEST(monatsbeitragTest, TF" << ++testfall
               << ") {\n  Monatsbeitrag mb;\n"
                  "  std::set<std::string> abt {";
      auto testparameter = split(zeile, ",");
      auto mitgliedsstatus = testparameter.at(testparameter.size()-2);
      auto orakel = testparameter.back();
      bool ersterEintrag = true;
      for(std::size_t i = 0; i < testparameter.size() - 1u; ++i) {
        if(testparameter[i] == "true") {
          if(!ersterEintrag) {
            ergebnis << ", ";
          }
          ersterEintrag = false;
          ergebnis << '"' << faktoren[i] << '"';
        }
      }
      ergebnis << "};\n  EXPECT_EQ(mb.beitrag(\""
               << mitgliedsstatus << "\", abt), "
               << orakel << ");\n}\n";
    }
  }
}
```

Listing 4.29: Testcode erzeugen *(beispiele/sportvACTS/generator/main.cpp)*

In diesem kleinen Beispiel ist die generierte Datei noch etwas kürzer als das erzeugende Programm. Bei einem Fall mit vielen Kombinationsmöglichkeiten wird das anders sein.

```
// Von generator/generator.exe generierte Datei!
#include <gtest/gtest.h>
#include "monatsbeitrag.h"

TEST(monatsbeitragTest, TF1) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Tischtennis"};
  EXPECT_EQ(mb.beitrag("Kind/Jugendlicher", abt), 12);
}

TEST(monatsbeitragTest, TF2) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Turnen", "Ballsport", "Fitness"};
  EXPECT_EQ(mb.beitrag("Kind/Jugendlicher", abt), 37);
}

TEST(monatsbeitragTest, TF3) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Tischtennis", "Turnen", "Fitness"};
  EXPECT_EQ(mb.beitrag("Erwachsener", abt), 44);
}

TEST(monatsbeitragTest, TF4) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Ballsport"};
  EXPECT_EQ(mb.beitrag("Erwachsener", abt), 27);
}

TEST(monatsbeitragTest, TF5) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Tischtennis", "Turnen", "Ballsport"};
  EXPECT_EQ(mb.beitrag("Ermaessigt", abt), 33);
}

TEST(monatsbeitragTest, TF6) {
  Monatsbeitrag mb;
  std::set<std::string> abt {"Fitness"};
  EXPECT_EQ(mb.beitrag("Ermaessigt", abt), 18);
}

TEST(monatsbeitragTest, TF7) {
  Monatsbeitrag mb;
  std::set<std::string> abt {};
```

```
  EXPECT_EQ(mb.beitrag("Passiv", abt), 20);
}
```

Listing 4.30: Testprogramm *(beispiele/sportvACTS/dertest.cpp)*

4.6.6 Ein Beispiel ohne Orakel

Als Grundlage nehmen wir das Beispiel der statistischen Auswertung der Studiendauer von Studierenden aus Abschnitt 4.5.1 von Seite 62. Die zugehörige Abbildung 4-2 zeigt fünf *unabhängige* Klassifikationen (Geschlecht, Alter, Herkunft, Dauer, Fach). Nach Abbildung 4-2 gibt es 15 Klassen, und wie schon in Abschnitt 4.5.1 festgestellt, ergeben sich $2{\cdot}4{\cdot}3{\cdot}3{\cdot}3 = 2^1 \cdot 4^1 \cdot 3^3 = 216$ Testfälle bei vollständiger Kombination. Wenn es zu jeder der fünf Klassifikationen genau 3 Klassen gäbe (was nicht der Fall ist), könnten wir ein 3^5-orthogonales Array verwenden. Aber: Ein $2^1\ 4^1\ 3^3$-orthogonales Array gibt es nicht. Das Problem verschwindet, wenn wir stattdessen ein Covering Array nehmen.

Ein Orakel wie im vorherigen Abschnitt ist nicht notwendig, weil einfach nur die Kombinationen in eine Datenbank geschrieben werden (siehe Seite 62). Es müssen also keine zu erwartenden Testergebnisse ermittelt werden, sondern es wird nur geprüft, ob die Datenbankeinträge korrekt abgelegt sind. Abbildung 4-6 zeigt zwölf Kombinationen für das vollständige paarweise Testen.

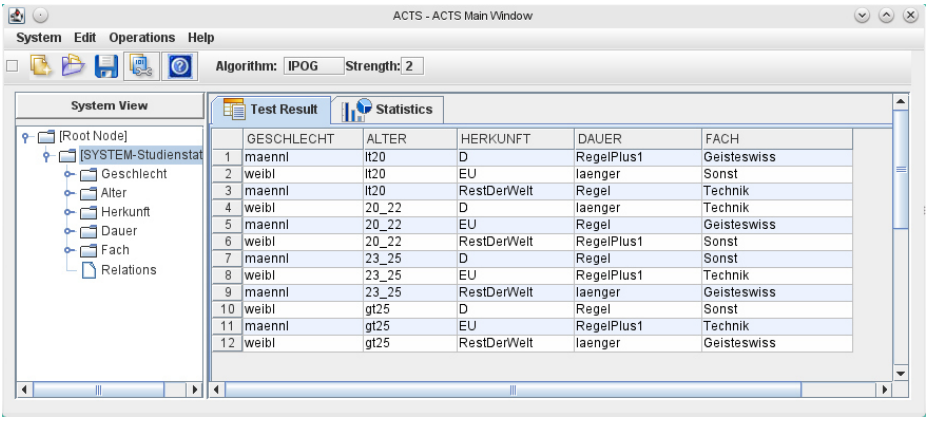


Abbildung 4-6: Covering Array für das Studienstatistik-Beispiel

Zum Testen aller 3er-Kombinationen sind 43 Testfälle erforderlich. Alle 5er-Kombinationen (Anzahl der Parameter) ergeben sämtliche oben berechneten 216 Testfälle.

Der nachfolgende Test unterscheidet sich von allen bisherigen dadurch, dass nicht jeder Test einzeln angelegt wird, sondern alle Tests laufen in ei-

ner Schleife, in der der jeweilige Testfall eingelesen wird. Im folgenden Programm wird eine Klasse `StatistikDBTest` angelegt, die von `::testing::Test` erbt. Der Konstruktor ist für das Testsetup verantwortlich. Er leert die Datenbank und füllt sie mit allen Kombinationen. Das Makro `TEST_F` (nicht nur `TEST`!) liest jeden Testfall ein und vergleicht die Daten mit dem entsprechenden Eintrag in der Datenbank. Das Makro `SCOPED_TRACE` sorgt für die Ausgabe des Fehlers. Im Unterschied zu den bisherigen Testfällen gibt es jetzt wieder eine eigene `main`-Funktion mit dem entscheidenden Ausdruck `RUN_ALL_TESTS()`.

```
#include "StatistikDB.h"
#include <gtest/gtest.h>
#include <iostream>

namespace {
  class StatistikDBTest : public ::testing::Test {
  protected:
    StatistikDBTest()   // Testsetup
      : db("Studienstatistik.db"),
        acts_dateiname("Studienstatistik.txt") {
      eingabe.open(acts_dateiname);
      // 1. Durchgang: Testdatenbank leeren und neu füllen
      db.leeren();
      db.fuellen(eingabe);
      eingabe.close();
      eingabe.clear();
    }
    virtual ~StatistikDBTest() = default;
    StatistikDB db;
    std::string acts_dateiname;
    std::ifstream eingabe;
  };

  TEST_F(StatistikDBTest, VergleichDerTestFaelle) {
    // 2. Durchgang: Eigentlicher Test: Vergleich der Einträge
    eingabe.open(acts_dateiname);
    ueberspringeKopfteil(eingabe);
    std::size_t nr = 1;
    while(true)  {
      auto testfall = liesTestfall(eingabe);
      // Nächste Zeile löschen! Fehler provozieren: test the test:
      if(nr ==3 || nr == 7) {testfall.at(1) = "murks";}
      if(testfall.size() > 0) {
```

```
        auto datenbankeintrag = db.liesTupel(nr);
        SCOPED_TRACE("Fehler in Tupel "  + std::to_string(nr));
        EXPECT_EQ(testfall, datenbankeintrag);
      }
      else {
        break;
      }
      ++nr;
    }
  }
}    // Namespace

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Listing 4.31: Testprogramm *(beispiele/statistikDB/main.cpp)*

Die Datei *Studienstatistik.txt* enthält die vom ACTS-Programm gespeicherte Konfiguration einschließlich des Testsets. Um die Fehlerlokalisation zu zeigen, wurden zwei Testfälle manipuliert (siehe Kommentarzeile im Programm). Das Ergebnis ist:

```
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from StatistikDBTest
[ RUN      ] StatistikDBTest.VergleichDerTestFaelle
main.cpp:35: Failure
Value of: datenbankeintrag
  Actual: { "maennl", "lt20", "RestDerWelt", "Regel", "Technik" }
Expected: testfall
Which is: { "maennl", "murks", "RestDerWelt", "Regel", "Technik" }
Google Test trace:
main.cpp:34: Fehler in Tupel 3
main.cpp:35: Failure
Value of: datenbankeintrag
  Actual: { "maennl", "23_25", "D", "Regel", "Sonst" }
Expected: testfall
Which is: { "maennl", "murks", "D", "Regel", "Sonst" }
Google Test trace:
main.cpp:34: Fehler in Tupel 7
[  FAILED  ] StatistikDBTest.VergleichDerTestFaelle (170 ms)
[----------] 1 test from StatistikDBTest (170 ms total)
[----------] Global test environment tear-down
```

```
[==========] 1 test from 1 test case ran. (170 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] StatistikDBTest.VergleichDerTestFaelle
```

Listing 4.32: Testergebnis mit Fehlern

Als Datenbank verwenden wir *sqlite*.[24] Die weiteren zum obigen Testprogramm gehörenden Dateien wie beispielsweise die Definition der Klasse `StatistikDB` haben wir aus Platzgründen nicht abgedruckt.

> Das kombinatorische Testen ist gut nachvollziehbar und überzeugt durch Eingrenzung der Kombinationen. Allerdings wird etwas Mathematik benötigt und am besten auch eine Unterstützung durch ein Werkzeug. Das paarweise bzw. n-weise Testen bietet einen Mittelweg nach Lean-Kriterien zwischen keiner (jeder Parameterwert kommt in einem Testfall vor) und vollständiger Kombination (jede Kombination aller Parameterwerte). Erzeugte Kombinationen, die nicht konform zu den Anforderungen sind, sind zu löschen und sinnvolle fehlende, aber durch das Verfahren nicht erzeugte Kombinationen sind zu ergänzen.

Alle Dateien sind im Verzeichnis *beispiele/statistikDB* enthalten, wenn Sie die Datei mit den Beispielen von der Webseite zum Buch herunterladen und entpacken. Sie müssen vorher *sqlite* installieren, wenn Sie das Beispiel ausprobieren wollen. Anstelle der Beschreibung des gesamten Programms geben wir hier nur eine Bedienungsanleitung:

- Datenbank neu erzeugen:
  - Die Datei *Studienstatistik.db* löschen, wenn vorhanden.
  - Mit dem Konsolenbefehl
    `sqlite3 Studienstatistik.db < createStudienstatistikDB.sql`
    alle benötigten Tabellen anlegen. *sqlite* hat keinen Datentyp `enum`, weswegen er mit eigenen Tabellen simuliert wird (siehe *createStudienstatistikDB.sql*). Die Tabelle `statistikeintrag` wird nach der Anlage die zu testenden Kombinationen enthalten.
- Das Programm *dertest* durch Eingabe von `make` erzeugen.
- *dertest* starten. Das Ergebnis erscheint auf der Konsole, wenn es nicht in eine Datei umgeleitet wird. Das Programm *dertest* leert zunächst die Tabelle `statistikeintrag` und befüllt sie dann entsprechend den Daten aus der ACTS-Datei *Studienstatistik.txt*. Anschließend wird geprüft, ob die Einträge in der Datenbank vorliegen.

[24] *http://www.sqlite.org*

4.7 Entscheidungstabellentest

Nicht immer lassen sich Parameter frei miteinander kombinieren. Beeinflussen Parameterwerte sich gegenseitig oder sind Kombinationen von Bedingungen zu berücksichtigen, hilft der Entscheidungstabellentest zur Klärung der Sachverhalte und zur Erstellung von Testfällen.

Es gibt Konstellationen, bei denen zum Beispiel ein Fakt gegeben sein muss, bevor weitere Unterscheidungen zum Tragen kommen. So muss eine Person eine gültige Bankkarte besitzen, die vom Geldautomaten akzeptiert wird. Erst dann wird die PIN abgefragt, und nach erfolgreicher Eingabe kann eine Banktransaktion durchgeführt werden. Ist die Bankkarte nicht gültig oder nicht lesbar, spielen alle folgenden möglichen Aktionen (PIN-Eingabe, Kontostandabfrage, Geldabhebung, ...) keine Rolle.

Wann ist der Einsatz sinnvoll?

Wenn bestimmte Kombinationen von Voraussetzungen erfüllt sein müssen und dann erst weitere Entscheidungen zu treffen sind, ist der Entscheidungstabellentest das passende Verfahren.

> Entscheidungstabellen können bereits bei der Spezifikation und nicht erst beim Testen eine wichtige Rolle spielen, um den Sachverhalt zu klären. Aber dies gilt auch für andere Testansätze. So kann zum Beispiel die Bildung der Äquivalenzklassen die Spezifikation verbessern und präzisieren.

Grundidee

Sind beispielsweise Berechnungen oder Ausgaben von Kombinationen von (Eingangs-)Bedingungen abhängig, so ist zu klären, welche der möglichen Kombinationen zu welchen Berechnungen oder Ausgaben führen soll. Eine Entscheidungstabelle stellt einen solchen Sachverhalt in übersichtlicher Weise dar. Die Tabelle ist in vier Bereiche unterteilt:

- Oben links: Hier werden alle Bedingungen so aufgeführt, dass sie mit *ja* oder *nein* beantwortet werden können.
- Oben rechts: In einem ersten Schritt werden hier alle Kombinationen der Bedingungen aufgeführt (bei n Bedingungen sind es 2^n Kombinationen).
- Unten links: Alle Aktionen (Ausgabe, Berechnung, ...) sind hier vermerkt.
- Unten rechts: Hier wird vermerkt, bei welcher Kombination welche Aktion erfolgen soll.

Eine Entscheidungstabelle ist wie folgt zu erstellen:

1. Ermittle alle Bedingungen, die eine Rolle bei dem Problem spielen. Formuliere die Bedingungen so, dass sie mit *ja* oder *nein* zu beantworten sind. Schreibe jede Bedingung in jeweils eine Zeile (links oben).
2. Ermittle alle möglichen Aktionen, die zu berücksichtigen sind. Schreibe eine Aktion in jeweils eine Zeile (links unten).
3. Erstelle alle Kombinationen der Bedingungen ohne Berücksichtigung von möglichen Abhängigkeiten (oben rechts).
4. Sieh dir alle Bedingungskombinationen (jede Spalte auf der rechten Seite der Tabelle) an und entscheide, welche Aktion bei der jeweiligen Kombination zur Ausführung kommen soll. Setze ein Kreuz an die Stelle der Tabelle, wo die Bedingungskombination zu einer Aktion führen soll. Es können also mehrere Kreuze bei einer Kombination vorkommen, wenn bei dieser Kombination mehrere Aktionen auszuführen sind.

Führen unterschiedliche Bedingungskombinationen zu gleichen Aktionen, kann die Tabelle konsolidiert werden. Ebenso kann geprüft werden, ob die Entscheidungstabelle vollständig, redundanzfrei und konsistent ist. Zunächst verdeutlichen wir aber die Grundidee an einem einfachen Beispiel.

4.7.1 Ein Beispiel

Inzwischen ist es übliche Praxis, Kunden an ein Geschäft zu binden, indem Kundenkarten an Stammkunden ausgegeben werden. In unserem Beispiel bekommen Stammkunden einen Rabatt von 7% auf ihren Einkauf. Um den Verkauf in den Morgenstunden zu erhöhen, wird in der Zeit von 8:00 bis 10:00 Uhr ein allgemeiner Rabatt auf alle Einkäufe von 5% gewährt. Stammkunden- und Morgenstundenrabatt werden beide gewährt (insgesamt dann 12% Rabatt). Alle Kunden, die vor 10:00 Uhr einkaufen, erhalten zusätzlich ein kleines Werbegeschenk. Aus diesen Angaben ist wie folgt eine Entscheidungstabelle zu erstellen.

Die Bedingungen sind:

- Kunde ist Stammkunde
- Zeit des Einkaufs ist zwischen 8:00 und 10:00 Uhr

Die Aktionen sind:

- Rabatt von 7% (bei Stammkunden)
- Rabatt von 5% (bei Einkauf zwischen 8:00 und 10:00 Uhr)
- Werbegeschenk (bei Einkauf zwischen 8:00 und 10:00 Uhr)

Die Entscheidungstabelle ergibt sich aus den Angaben wie folgt (siehe Tabelle 4-11).

Bedingungen				
Kunde ist Stammkunde	ja	ja	nein	nein
Einkaufszeit 8:00 bis 10:00 Uhr	ja	nein	ja	nein
Aktionen				
Rabatt von 7%	x	x		
Rabatt von 5%	x		x	
Werbegeschenk	x		x	

Tabelle 4-11: Entscheidungstabelle Rabatt für Stammkunden und Frühkäufer

Die Tabelle zeigt nun übersichtlich, welche Bedingungskombinationen zu welchen Aktionen führen. So bekommt ein Stammkunde, der um 9:15 Uhr einkauft, neben 12% Rabatt auch ein Werbegeschenk. Geht derselbe Kunde eine Stunde später einkaufen, erhält er lediglich 7% Rabatt. Eine Vereinfachung der Tabelle ist nicht möglich, da jede der vier Kombinationen zu einer anderen (oder keiner) Zusammenstellung von Aktionen führt.

Jede Spalte des rechten Teils der Tabelle entspricht einem Testfall und die erwarteten (Re-)Aktionen können an den Kreuzen in der Spalte abgelesen werden.

Nehmen wir nun an, dass Spirituosen verkauft werden. Diese dürfen aber nur an Personen über 18 Jahren abgegeben werden. Es kommt somit eine weitere Bedingung hinzu. Die Größe der Entscheidungstabelle verdoppelt sich (siehe Tabelle 4-12).

Bedingungen								
Kunde ist unter 18 Jahren	j	j	j	j	n	n	n	n
Kunde ist Stammkunde	j	j	n	n	j	j	n	n
Einkaufszeit 8:00 bis 10:00	j	n	j	n	j	n	j	n
Aktionen								
Verkauf von Spirituosen ablehnen	x	x	x	x				
Rabatt von 7%					x	x		
Rabatt von 5%					x		x	
Werbegeschenk					x		x	

Tabelle 4-12: Entscheidungstabelle für Spirituosenverkauf (j – ja, n – nein)

Allerdings führt dies nicht zwangsläufig zu einer Verdoppelung der Testfälle. Die Tabelle zeigt deutlich, dass bei Kunden unter 18 Jahren alle weiteren Aktionen beim Kauf von Spirituosen keine Rolle spielen. Die ersten vier Spalten der rechten Seite der Tabelle können daher zu einer Spalte zusammengefasst werden, da sie alle die Aktion »Verkauf von Spirituosen ablehnen« bewirken. Die Entscheidungstabelle wird konsolidiert (siehe Tabelle 4-13).

Bedingungen					
Kunde ist unter 18 Jahren	j	n	n	n	n
Kunde ist Stammkunde	–	j	j	n	n
Einkaufszeit 8:00 bis 10:00	–	j	n	j	n
Aktionen					
Verkauf von Spirituosen ablehnen	x				
Rabatt von 7%		x	x		
Rabatt von 5%		x		x	
Werbegeschenk		x		x	

Tabelle 4-13: Konsolidierte Entscheidungstabelle für Spirituosenverkauf

Die »–« in der Spalte werden als »don't care« interpretiert, d.h., ob die Bedingung zutrifft oder nicht, spielt keine Rolle für die Auswahl der Aktion. Durch die zusätzliche Bedingung ist nur ein weiterer Testfall hinzugekommen.

Testendekriterium

Jede Spalte der Entscheidungstabelle entspricht einem Testfall. In jeder Spalte der Tabelle ist einfach abzulesen, welche Bedingungen zutreffen müssen (und welche nicht) und welche Aktionen bei dieser Kombination erwartet werden. Dies entspricht den Ausgaben bzw. dem Verhalten des Testobjekts. Der Test wird als ausreichend angesehen, wenn jede Spalte zu einem Testfall führt.

Bewertung

Der große Vorteil der Entscheidungstabellen ist die Übersicht über alle relevanten Kombinationen, also solche, die zu unterschiedlichen Aktionen oder Zusammenstellungen von Aktionen führen. Dadurch wird kein relevanter Test – keine relevante Kombination von Bedingungen – übersehen. Dies wird allerdings dadurch erreicht, dass in einem ersten Schritt alle Kombinationen zu erstellen sind, wie oben im Beispiel auch durchgeführt.

Nun werden Sie zu Recht sagen: Diesen Schritt kann ich mir doch sparen. Ist doch klar: Wenn der Kunde wegen seines Alters keinen Alkohol bekommt, sind alle weiteren Rabatte überflüssig. Stimmt, aber es besteht eine gewisse Gefahr, dass bei komplizierteren Kombinationen von Bedingungen relevante Kombinationen übersehen werden.

Aber die Vollständigkeit von Entscheidungstabellen kann schnell überprüft werden: Die Anzahl der Spalten auf der rechten Seite muss eine 2er-Potenz ergeben. In Tabelle 4-11 sind es vier und in Tabelle 4-12 acht Spalten. Wie sieht es aber mit der konsolidierten Tabelle 4-13 aus? Diese enthält fünf Spalten! Wenn in einer Tabelle »–« vorkommen, dann führt jeder Strich zu einer Multiplikation mit 2. Wir haben also 1 Spalte (mit »ja« bei der ersten

Bedingung) mit zwei Strichen bei den weiteren Bedingungen in der Spalte, dies ergibt 1·2·2 = 4. Gefolgt von vier weiteren Spalten in der Tabelle, was zusammen wieder acht und somit eine 2er-Potenz ergibt. Die Tabelle ist somit vollständig und keine Kombination wurde übersehen.

Redundante Spalten können ebenso erkannt werden. Nehmen wir an, in Tabelle 4-13 wäre eine weitere Spalte enthalten mit den Bedingungen (ja, –, ja), also ein Jugendlicher will vor 10:00 Uhr Alkohol einkaufen. Diese Kombination führt zu keiner anderen Aktion als die Kombination der ersten Spalte, sie ist in der ersten Spalte durch den »–« bei der dritten Bedingung enthalten.

Ob die Bedingungen und Aktionen konsistent sind, kann ebenfalls geprüft werden. Wäre im oben geschilderten Fall der zusätzlichen Spalte eine unterschiedliche Aktion (Verkauf an Jugendliche vor 10:00 Uhr erlaubt) auszuführen, liegt eine Inkonsistenz der Tabelle vor. Beide Spalten – oder genauer die Kombination der Bedingungen und deren darausfolgenden Aktionen – widersprechen sich: In der ersten ist der Verkauf nicht erlaubt, in der zusätzlichen Spalte allerdings morgens gestattet.

Auch ergibt die Überprüfung der Vollständigkeit mit der zusätzlichen Spalte in den beiden geschilderten Fällen keinen korrekten Wert, d.h. keine 2er-Potenz.

Ein großer Nachteil von Entscheidungstabellen ist das schnelle Anwachsen des Umfangs, wenn viele Bedingungen zu berücksichtigen sind. Aber das Testen von Kombinationen vieler Bedingungen ist eben auch nicht einfach. Und in einem solchen Fall ein Mittel an der Hand zu haben, das einem die Sicherheit gibt, keine wichtige Kombination zu vergessen, ist sehr hilfreich.

Bezug zu anderen Testverfahren

Der Entscheidungstabellentest kombiniert Bedingungen und stellt die zu erfolgenden Aktionen dar. Die Bedingungen sind dabei nicht unbedingt als Parameterwerte zu sehen, sie können auf einer abstrakteren Ebene, etwa der Spezifikation, formuliert sein.

Die Einbeziehung der Aktionen (Ergebnisse) bei jedem Testfall ist ein großer Vorteil und wird bei den bisher vorgestellten Verfahren nicht (direkt) berücksichtigt. Bei diesen Verfahren ist ein Testorakel zu befragen, um die erwarteten Ergebnisse und Reaktionen vorab festzulegen.

Ein weiterer Vorteil ist die Prüfung auf Vollständigkeit, Redundanzfreiheit und Konsistenz der Entscheidungstabelle und somit der Testfälle. So kann sichergestellt werden, dass wichtige Kombinationen nicht übersehen und doppelte oder nicht mögliche Kombinationen gar nicht erst berücksichtigt werden.

Äquivalenzklassen beziehen sich auf Parameterwerte, die meist ohne Einschränkung frei zu kombinieren sind. Abhängigkeiten zwischen Parame-

tern lassen sich nur schwer bis gar nicht berücksichtigen. Die Beachtung von Grenzwerten, wenn eine Bedingung von »erfüllt« zu »nicht erfüllt« wechselt, wird beim Entscheidungstabellentest nicht verlangt, ist aber unter Testgesichtspunkten eine wichtige Ergänzung. Bei der Klassifikationsbaummethode werden die Kombinationsmöglichkeiten der Parameter bzw. der Klassifikationen explizit bei jedem Testfall verdeutlicht. Es fehlt allerdings die Angabe der Ergebnisse (Aktionen) und ein »don't care« ist nicht vorgesehen. Beim kombinatorischen Testen sind keine Abhängigkeiten gegeben. Es wird davon ausgegangen, dass alles mit allem frei kombinierbar ist.

Hinweise für die Praxis

Entscheidungstabellen sind schon bei der Spezifikation einzusetzen und dann beim Testen zur Erstellung der Testfälle heranzuziehen. Aber die Realität sieht oft so aus, dass beim Testen Lücken und Ungenauigkeiten der Spezifikation aufgedeckt werden. Hierbei hilft der Entscheidungstabellentest.

4.7.2 Ein Beispiel in C++

Der Preis für eine Taxifahrt setzt sich üblicherweise aus einem Grundpreis und einem Preis pro gefahrenen Kilometer, oft gestaffelt nach Entfernung, zusammen. Letzteres soll im Beispiel durch eine Rabattstaffelung realisiert werden. Nachtfahrten und Gepäckstücke sollen ebenso Berücksichtigung finden. Für die Berechnung des Preises einer Taxifahrt seien die folgenden Bedingungen gegeben:

- Der Grundpreis beträgt 3,50 €, der Fahrpreis pro Kilometer 2,10 €.
- Bei einer Strecke von mehr als 10 Kilometern wird ein Rabatt von 5% gewährt.
- Bei einer Strecke von mehr als 50 Kilometern wird ein Rabatt von 10% gewährt, aber nicht zusätzlich zum 5%-Rabatt.
- Findet die Fahrt zwischen 22:00 und 6:00 Uhr statt, gibt es einen Nachtzuschlag in Höhe von 20%.
- Gepäckstücke kosten unabhängig von der Anzahl 3 € zusätzlich.

Eine Fahrt ohne Gepäck tagsüber von beispielsweise genau 10 km kostet damit 3,50 € + 10 · 2,10 € = 24,50 €. Die Berechnung der Kosten für eine Taxifahrt lässt sich leicht als C++-Funktion beschreiben. Weil es beim Preis keine Cent-Bruchteile gibt, wird er in ganzen Cent zurückgegeben:

```
// gibt Preis in Eurocent zurück
int fahrpreis(int grundpreis,                    // Eurocent
              int centProKm,
```

```
              int streckeInKm,
              bool nachtfahrt,
              bool gepaeck) {
  double preis = grundpreis + centProKm * streckeInKm;
  if(streckeInKm > 50) {
    preis *= 0.9;                                  // Rabatt 10%
  } else if(streckeInKm > 10) {
    preis *= 0.95;                                 // Rabatt 5%
  }
  if(nachtfahrt) {
    preis *= 1.2;                                  // Zuschlag 20%
  }
  if(gepaeck) {
    preis += 300;                                  // Zuschlag 3 Euro
  }
  return static_cast<int>(preis + 0.5);            // Rundung
}
```

Listing 4.33: Fahrpreisberechnung

Ist dieser Algorithmus richtig? Wie viele Testfälle sind notwendig, um die richtige Berechnung des Fahrpreises zu prüfen? Was meinen Sie? Denken Sie bitte kurz nach, wie viele Testfälle Sie als ausreichend ansehen. Sind es vier, sechs, acht oder mehr?

Um die Fragen beantworten zu können, konstruieren wir eine Entscheidungstabelle. Da der Grundpreis und der Fahrpreis pro Kilometer konstant bleiben, müssen wir sie nicht in der Tabelle berücksichtigen. Von den oben genannten Bedingungen bleiben damit vier übrig, sodass die Anzahl der Kombinationen $2^4 = 16$ ist. Die Anzahl der Testfälle ist höchstens genau so hoch, bei möglicher Konsolidierung der Tabelle aber geringer.

Bedingungen	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
> 10 km	j	j	j	j	j	j	j	j	n	n	n	n	n	n	n	n
> 50 km	j	j	j	j	n	n	n	n	j	j	j	j	n	n	n	n
Nachtfahrt	j	j	n	n	j	j	n	n	j	j	n	n	j	j	n	n
Gepäck	j	n	j	n	j	n	j	n	j	n	j	n	j	n	j	n
Aktionen																
Rabatt 5%	x	x	x	x	x	x	x	x								
Rabatt 10%	x	x	x	x					x	x	x	x				
Zuschlag 20%	x	x			x	x			x	x			x	x		
Zuschlag 3 €	x		x		x		x		x		x		x		x	
Normalpreis																x

Tabelle 4-14: Erste Entscheidungstabelle für den Fahrpreistest

In Tabelle 4-14 ist bereits ein Widerspruch zu erkennen. Die Rabatte für Strecken von mehr als 10 oder mehr als 50 km sind nicht additiv. Es gilt nur der Rabatt der längeren Strecke! Das heißt, dass überall da, wo bei »> 50 km« ein »j« steht, in der Zeile »> 10 km« ein »–« (don't care) stehen muss (Spalten 1 bis 4 und 9 bis 12).

Ein genauer Blick zeigt darüber hinaus, dass die Aktionen dieser Spalten identisch sind, wenn man die nicht zu berücksichtigenden 5% Rabatt (Strecke zwischen 10 und 50 Kilometern) bei Strecken > 50 km ignoriert. Die Spalten 1 bis 4 oder die Spalten 9 bis 12 können wegfallen oder, anders ausgedrückt, zusammengelegt werden. Tabelle 4-15 zeigt die um die Spalten 9 bis 12 verminderte und damit konsolidierte Tabelle 4-14 mit fortlaufenden neuen Spaltennummern.

Bedingungen	1	2	3	4	5	6	7	8	9	10	11	12
> 10 km	–	–	–	–	j	j	j	j	n	n	n	n
> 50 km	j	j	j	j	n	n	n	n	n	n	n	n
Nachtfahrt	j	j	n	n	j	j	n	n	j	j	n	n
Gepäck	j	n	j	n	j	n	j	n	j	n	j	n
Aktionen												
Rabatt 5%					x	x	x	x				
Rabatt 10%	x	x	x	x								
Zuschlag 20%	x	x			x	x			x	x		
Zuschlag 3 €	x		x		x		x		x		x	
Normalpreis												x

Tabelle 4-15: Konsolidierte Entscheidungstabelle für den Fahrpreistest

In der Tabelle 4-15 liefert nun keine Kombination (Spalte) mehr identische Aktionen. Das heißt, es bleiben 12 Testfälle zu prüfen. Wie viele hatten Sie sich überlegt? Auch 12? Gratulation! Vermutlich waren es weniger und Sie haben Kombinationsmöglichkeiten übersehen. Wenn es mehr waren, sind doppelte und damit unnütze Testfälle dabei. Die Entscheidungstabelle hilft also, die Übersicht über die sinnvollen Kombinationen zu behalten.

Die Strecken von 5 km, 20 km und 60 km werden in den folgenden Testfällen für die drei Streckenbereiche ≤ 10 km, > 10 km und ≤ 50 km sowie > 50 km gewählt. Grenzwerte werden nicht geprüft, um das Beispiel einfach zu halten[25]. Bei jeder der drei Strecken wird zwischen einer Tag- und einer Nachtfahrt mit und ohne Gepäck unterschieden. Mithilfe einer Tabellenkalkulation werden die zu erwartenden Fahrpreise berechnet (siehe Tabelle 4-16).

Das Listing 4.34 zeigt das Testprogramm. Die Nummer eines Testfalls `TF` entspricht der Spaltennummer der Tabelle 4-16.

[25] Ergänzende Testfälle mit den Grenzwerten sind sinnvoll und durchzuführen.

Nummer	Strecke	Zeit	Gepäck	Fahrpreis
1	60 km	nachts	ja	142,86 €
2	60 km	nachts	nein	139,86 €
3	60 km	tags	ja	119,55 €
4	60 km	tags	nein	116,55 €
5	20 km	nachts	ja	54,87 €
6	20 km	nachts	nein	51,87 €
7	20 km	tags	ja	46,23 €
8	20 km	tags	nein	43,23 €
9	5 km	nachts	ja	19,80 €
10	5 km	nachts	nein	16,80 €
11	5 km	tags	ja	17,00 €
12	5 km	tags	nein	14,00 €

Tabelle 4-16: Testfälle für die Fahrpreisberechnung

```
#include "gtest/gtest.h"
#include "taxifahrt.h"

// Parameter: grundpreis, centProKm, streckeInKm, nachtfahrt, gepaeck
constexpr std::size_t grundpreis = 350;  // Eurocent
constexpr std::size_t centProKm  = 210;  // Eurocent
// erwartet wird der Fahrpreis in Eurocent

TEST(Taxifahrt, TF1) {
  EXPECT_EQ(14286, fahrpreis(grundpreis, centProKm, 60, true, true));
}

TEST(Taxifahrt, TF2) {
  EXPECT_EQ(13986, fahrpreis(grundpreis, centProKm, 60, true, false));
}
// ... Testfälle 3 bis 10 weggelassen

TEST(Taxifahrt, TF11) {
  EXPECT_EQ(1700, fahrpreis(grundpreis, centProKm, 5, false, true));
}

TEST(Taxifahrt, TF12) {
  EXPECT_EQ(1400, fahrpreis(grundpreis, centProKm, 5, false, false));
}
```

Listing 4.34: Auszug des Testprogramms *beispiele/taxi1/fahrpreistest.cpp*

Vorsicht Kunde!

Alle Test sind bestanden! Die vorab berechneten zu erwartenden Ergebnisse wurden beim Test bestätigt. Nun stellen Sie sich vor, dass der Kunde die Testfälle mit den Ergebnissen erhält. Nach ein paar Tagen meldet er sich und lobt Sie für die gründliche Testdurchführung. Dann sagt er aber: »Mir kamen die krummen Preise doch etwas merkwürdig vor. Deshalb habe ich alles nachgerechnet. Das Ergebnis meiner Rechnungen ist, dass die Testfälle 11 und 12[26] korrekt berechnet wurden, aber alle anderen Ergebnisse falsch sind!« Schreck, lass nach! Wie konnte das bei der verwendeten Sorgfalt überhaupt geschehen? Bei einem Treffen bestätigt der Kunde, dass die genannten fünf Bedingungen von Seite 97 richtig sind. Um das Rätsel zu lösen, schreiben Sie ihm genau auf, wie Sie den Preis berechnet haben.

Das ist der Moment, wo der Kunde feststellt: »Auf den *Grundpreis* gibt es natürlich *nie* Rabatt!«. Und er fügt hinzu, dass sowohl Rabatte wie auch der Nachtzuschlag sich immer auf den Basispreis beziehen! Rabatt und Nachtzuschlag sollen jeweils kaufmännisch gerundete Werte ergeben und müssten separat ausgewiesen werden können.

Natürlich haben Sie »Basispreis« in diesem Zusammenhang noch nie gehört und fragen nach. Der Basispreis entpuppt sich als der Preis, der sich aus der gefahrenen Strecke mal dem Preis pro Kilometer ergibt. Dummerweise haben Sie nicht nur den Grundpreis in die Rabatt- und Zuschlagsberechnung aufgenommen, sondern auch die 20% Zuschlag für die Nachtfahrt auf den schon rabattierten Preis bezogen und nicht auf den Basispreis.

> Ein typischer Fehler: Die Spezifikation ist nicht vollständig, lässt zu viele Interpretationen zu oder ist nicht widerspruchsfrei! Die zu erwartenden Werte wurden für eine bestimmte Interpretation der Spezifikation berechnet. Erst der Test zusammen mit einer wirklich unabhängigen Kontrolle hat die Fehlinterpretation aufgedeckt. Deshalb ist die Erstellung von Testfällen vor der Programmierung so sinnvoll als Ergänzung der Spezifikation – wenn der Kunde die Testfälle »absegnet«.

Wenn die angesprochenen Bedingungen von Seite 97 genau analysiert werden, ergibt sich, dass es mehrere Möglichkeiten der Fahrpreisberechnung gibt, unter anderem:

1. Rabatte und der Nachtzuschlag werden auf den berechneten Preis bezogen, wie es in Listing 4.33 von Seite 97 gemacht wird. Beispielrechnung für Testfall 5 aus Tabelle 4-16, eine Nachtfahrt mit Gepäck über 20

[26] Das sind die beiden einzigen Testfälle ohne Rabatt oder Zuschlag!

Kilometer:
$(3{,}50 + (2{,}10 \cdot 20)) \cdot 0{,}95 \cdot 1{,}2 + 3 = 54{,}87$

2. Wie bei Variante 1, aber der Gepäckzuschlag wird gleich in den Preis hineingerechnet und dann auch beim Rabatt und beim Nachtzuschlag berücksichtigt:
   $(3{,}50 + (2{,}10 \cdot 20) + 3) \cdot 0{,}95 \cdot 1{,}2 = 55{,}29$
3. Wie bei Variante 1 oder 2, aber der Grundpreis wird beim Rabatt und beim Nachtzuschlag nicht berücksichtigt:
   $3{,}50 + ((2{,}10 \cdot 20) + 3) \cdot 0{,}95 \cdot 1{,}2 = 54{,}80$
4. Rabattierung und Nachtzuschlag sind auf den Basispreis bezogen, werden aber nicht getrennt berechnet:
   $3{,}50 + (2{,}10 \cdot 20) \cdot 0{,}95 \cdot 1{,}2 + 3 = 54{,}38$
5. Wie sich der Kunde die Berechnung wünscht: Rabattierung und Nachtzuschlag sind separat auf den Basispreis bezogen auszurechnen:
   $2{,}10 \cdot 20 = 42{,}00$
   $42{,}00 \cdot 0{,}05 = 2{,}10$ Rabatt
   $42{,}00 \cdot 0{,}2 = 8{,}40$ Zuschlag
   Als Endpreis ergibt sich: 3,50 + 42,00 - 2,10 + 8,40 + 3 = 54,80

Es kann sein, dass trotz unterschiedlicher Berechnung dieselben Preise herauskommen – je nach Wahl der Parameter. Ein Test allein bringt noch keine Klarheit.

Hier das entsprechend Punkt 5 korrigierte Programm:

```
// gibt Preis in Eurocent zurück
int fahrpreis(int grundpreisInCent,
              int preisInCentProKm,
              int streckeInKm,
              bool nachtfahrt,
              bool gepaeckVorhanden) {
  int basispreis = preisInCentProKm * streckeInKm;
  int rabatt = 0;
  if(streckeInKm > 50) {
    rabatt = std::round(0.1 * basispreis);     // Rabatt 10%, gerundet
  } else if(streckeInKm > 10) {
    rabatt = std::round(0.05 * basispreis);    // Rabatt 5%, gerundet
  }
  int zuschlag = 0;
  if(nachtfahrt) {
    zuschlag = std::round(0.2 * basispreis); // Zuschlag 20%, gerundet
  }
  if(gepaeckVorhanden) {
    zuschlag += 300;                         // Cent Zuschlag für Gepäck
  }
```

```
  return  grundpreisInCent + basispreis + zuschlag - rabatt;
}
```

Listing 4.35: Fahrpreisberechnung nach Kundenvorstellung

Entscheidungstabellen sind nicht wirklich »lean«, sie helfen aber dabei, alle möglichen und im Ergebnis unterschiedlichen Kombinationen von Bedingungen beim Testen zu berücksichtigen. Fehler können somit frühzeitig erkannt und behoben werden. In diesem Sinne ist das Ganze dann doch »lean«, da Kosten gespart werden können im Vergleich zu den Korrekturkosten, die entstehen, wenn die Software bereits im Einsatz ist.

> Aus unserer Praxis wissen wir, dass verschiedene Interpretationen der Spezifikation oft zu fehlerhafter Umsetzung der Anforderungen des Kunden führen. Frühzeitige gute Kommunikation mit allen am Projekt beteiligten Personen kann das verhindern helfen. Nach unserer Auffassung ist das einer der Vorteile des agilen Vorgehens, bei dem Kommunikation ein ganz wichtiger Bestandteil ist. Miteinander reden ist bei allen Projekten sinnvoll.

4.8 Zustandsbasierter Test

Bei den bisher vorgestellten Verfahren zur Erstellung von Testfällen sind nur die Parameterwerte und deren Kombinationen zu berücksichtigen. Bei einer ganzen Reihe von Systemen, besonders – aber nicht nur – im technischen Bereich, hat neben den Werten der Eingabe auch der bisherige Ablauf und damit der aktuelle Zustand des Systems Einfluss auf das Systemverhalten. So darf bei einer Fahrstuhlsteuerung die Tür nur dann geöffnet werden, wenn der Fahrstuhl auf einer Etagenebene steht und sich nicht mehr bewegt. Befindet sich der Fahrstuhl im Zustand *Abwärtsfahrt* (oder *Aufwärtsfahrt*), darf die Tür sich nicht öffnen lassen (die Öffnung in Notfällen wird außer acht gelassen). Die Software, oder genauer der Teil der Software, der die Türöffnung steuert, muss den Zustand des Fahrstuhls berücksichtigen und je nachdem unterschiedlich reagieren.

Wann ist der Einsatz sinnvoll?

Wenn neben den Eingabewerten auch der Zustand des Systems (oder eines Objekts bei einem objektorientiert entwickelten System) zu berücksichtigen ist, dann ist der zustandsbasierte Test ein passendes Vorgehen. Beim zustandsbasierten Test spielt die Historie eine Rolle, also welche unterschied-

lichen Zustände im Laufe der Systemnutzung erreicht und auch wieder verlassen werden. Des Weiteren sind ebenso die Zustandsübergänge zu berücksichtigen.

Zustandsmodell als Grundlage

Zur Spezifikation und Veranschaulichung der Zustände und Übergänge wird ein Zustandsmodell (Zustandsautomat und/oder Zustandsübergangstabelle, s.u.) verwendet. Die Modellierung mündet in die Spezifikation, die sowohl Basis für die Programmierung wie auch den Test ist. Schließlich soll der Test zeigen, dass das Programm die Spezifikation erfüllt.

Vorab noch einige Anmerkungen, wobei wir voraussetzen, dass Sie im Großen und Ganzen mit dem Konzept des Zustandsautomaten vertraut sind.

Zur Erinnerung: Ein Zustandsautomat ändert seinen Zustand in Abhängigkeit von seinem aktuellen Zustand und der nächsten Eingabe. Die Eingabe kann auch ein Ereignis sein. Die möglichen Ereignisse oder Eingaben[27] werden durch Symbole repräsentiert. Es gibt nur endlich viele Zustände. Der Automat kann in jedem Zustand oder bei jedem Übergang von einem Zustand zum nächsten eine Aktion tätigen (zum Beispiel eine Ausgabe). Wir nehmen folgende Vereinfachungen an:

- Aktionen sind mit Übergängen verbunden, nicht mit Zuständen.
- Der Automat ist deterministisch.[28] Das heißt, dass sich der Automat nach einer gegebenen Folge von Eingabesymbolen (Ereignissen) und einem gegebenen Startzustand in einem eindeutig definierten Folgezustand befindet. Der Begriff »deterministischer endlicher Automat« wird mit DEA abgekürzt.
- Der Automat ist vollständig. Damit ist gemeint, dass es in jedem Zustand einen Übergang für jede mögliche Eingabe gibt. Oft ist das in der Modellierung nicht der Fall. Die Vollständigkeit kann aber leicht erreicht werden, wenn der DEA so erweitert wird, dass für bisher nicht berücksichtigte Eingabesymbole ein Übergang
  - ohne eine Aktion auf denselben Zustand führt oder
  - zu einem Fehlerzustand führt.

Grundidee

Im Test soll geprüft werden, ob der Prüfling (das zu testende Programm) sich wie der DEA laut Spezifikation verhält, also bei einer gegebenen Folge von

[27] Die Begriffe werden in diesem Abschnitt synonym benutzt.

[28] Ein nicht deterministischer Automat kann in einen deterministischen umgewandelt werden.

Ereignissen einen bestimmten Zustand erreicht. Das Problem: Der Prüfling ist normalerweise eine Blackbox. Wir können die internen Zustände nicht beobachten (siehe Abbildung 4-7).

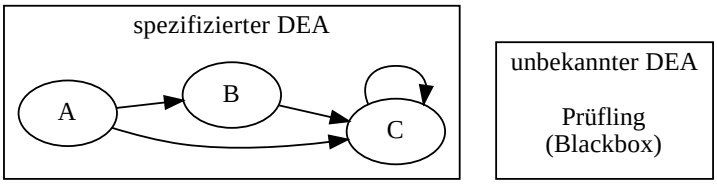


Abbildung 4-7: Sicht des Testers (der Entwickler kennt den Prüfling)

> Der Prüfling ist die Implementierung eines spezifizierten DEA. Der Zustandstest stellt fest, ob der Prüfling sich wie der spezifizierte Automat verhält. Mit passenden Folgen von Eingabesymbolen wird jeder Zustandsübergang ausgelöst. Dabei wird beobachtet, ob die Folge der Ausgaben/Aktionen dem spezifizierten DEA entspricht. Wenn ausreichend viele Übergänge getestet wurden, entspricht der Prüfling dem spezifizierten Modell. Was »ausreichend« bedeutet, hängt vom Einzelfall ab. Wenn der Automat interne Variablen benutzt, kann die Abfolge von diesen Variablen abhängen, sodass ein Übergang mehrfach getestet werden muss.

Die Folge der Eingaben muss stets beim Anfangszustand beginnen, da es bei einer Blackbox nicht möglich ist, einen bestimmten Zustand direkt zu erzeugen, um den Test mit genau diesem beginnen zu lassen. Was wir beobachten können, sind die Aktionen oder Ausgaben des Automaten. Es ist aber nicht garantiert, dass jede Eingabe zu einer Ausgabe führt. Aus den beobachteten Ausgaben können wir Rückschlüsse auf die internen Zustände und deren Reihenfolge ziehen und diese mit den spezifizierten vergleichen.

Der Entwickler hat keine Blackbox vor sich, da er das Programm selbst geschrieben hat. Er kann also während der Entwicklung bei jedem Zustand eine Meldung ausgeben, dass der Zustand erreicht wurde. Er kann auch *von außen* einen bestimmten Zustand erzwingen und den Test mit diesem internen Zustand als Startzustand beginnen. Allerdings verändert er dadurch das Testobjekt für den Test. Testobjekt und eingesetztes Programm (z. B. ohne Ausgabe der Zustandsmeldung) stimmen nicht mehr überein. Ein solches Vorgehen ist kritisch zu sehen, besonders im technischen Bereich, und soll daher vermieden werden. Auch kann es sein, dass das Programm in eine Hardware geladen wird, die keinerlei Möglichkeit für Zustandsmeldungen

vorsieht. In Einzelsituationen, etwa einer Simulation, kann es sinnvoll sein, das Testobjekt entsprechend zu manipulieren, falls der unten beschriebene Ansatz zu aufwendig oder nicht durchführbar ist.

Wir stellen zwei Ansätze zum zustandsbasierten Test vor:

- N-Switch-Überdeckung. Der Ansatz fokussiert auf einzelne Zustandsübergänge, wobei die Länge der Aneinanderreihung von Zuständen und Übergängen durch das »N« festgelegt wird.
- Testfallerzeugung mit einem Übergangsbaum. Hier beginnt jeder Testfall mit dem Startzustand, was den Test vereinfacht. Mehrfachzyklen werden bei diesem Ansatz nicht berücksichtigt.

Beide Ansätze werden im Folgenden ausführlich beschrieben und diskutiert.

4.8.1 Ein Beispiel

Mit einem einfachen Beispiel soll die Herleitung von Testfällen und die Definition von Testendekriterien beim zustandsbasierten Test verdeutlicht werden. Das Beispiel ist dem technischen Bereich entnommen, weil dort die Modellierung mit Zustandsautomaten verbreitet ist. Um testen zu können, müssen wir festlegen, welche Funktionalität getestet werden soll. Für das Beispiel brauchen wir also zunächst eine Spezifikation. Danach widmen wir uns dem Zustandstest.

Die Spezifikation – ein sehr einfacher Fahrstuhl

Dieser Fahrstuhl kennt nur zwei Stockwerke. Die Zustände könnten sein:

- oben wartend mit geschlossener Tür
- oben wartend mit offener Tür: bereit zum Ein- und Aussteigen
- nach unten fahrend
- nach oben fahrend
- unten wartend mit geschlossener Tür
- unten wartend mit offener Tür: bereit zum Ein- und Aussteigen

Bei dieser Auflistung fällt die Ähnlichkeit der Zustände auf. Deshalb werden zur Vereinfachung oben und unten zusammengelegt, sodass sich nur drei Zustände ergeben:

- wartend mit geschlossener Tür (ZWG)
- wartend mit offener Tür (ZWO)
- fahrend (ZF)

In Klammern sind abkürzende Symbole angegeben. Auch den zu betrachtenden Ereignissen sind abkürzende Symbole zugeordnet:

s Der Anforderungsknopf außerhalb des Fahrstuhls, der sich auf derselben Ebene wie der Fahrstuhl befindet, wird gedrückt. Je nach Fahrstuhlposition ist mal der Knopf oben, mal unten gemeint. Dass jemand innen im haltenden Fahrstuhl auf den Knopf »Tür öffnen« drückt, wird demselben Symbol zugeordnet.

b Der Anforderungsknopf außerhalb des Fahrstuhls, der sich auf der anderen Ebene wie der Fahrstuhl befindet, wird gedrückt. Die Tür wird geschlossen, falls sie offen ist, und die Fahrstuhlfahrt beginnt. Wenn jemand innen im haltenden Fahrstuhl auf den Knopf zum Starten der Fahrt drückt, wird dies demselben Symbol zugeordnet.

t Wenn die Tür aufgeht, wird ein Timer gestartet. Wenn die voreingestellte Zeit abgelaufen ist (timeout), wird dieses Ereignis erzeugt. Wenn die Tür schließt, wird der Timer automatisch gelöscht.

e Ein Sensor meldet, dass der fahrende Fahrstuhl seine Endposition (sein Ziel, oben oder unten) erreicht hat.

Abbildung 4-8 zeigt das Zustandsübergangsdiagramm. Start- und Endzustände werden mit einem doppelten Kreis dargestellt.

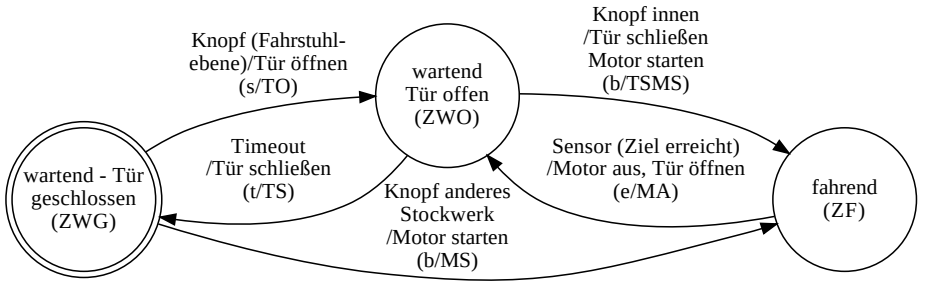


Abbildung 4-8: Zustandsautomat »Fahrstuhl« (Legende siehe Text)

Die oben beschriebenen Ereignisse lösen Zustandsübergänge aus, die einer Aktion verbunden sind. Ereignis und Aktion sind am Übergang mit einem trennenden Schrägstrich / notiert, darunter in Klammern die Abkürzung. Die noch fehlende Beschreibung der Aktionen wird hier nachgeholt:

NOP keine Aktion (No Operation).[29]
TO Tür oeffnen und Timer starten
TS Tür schließen und Timer löschen
MS Motor starten (Fahrstuhl beginnt zu fahren)
MA Motor aus (Fahrstuhl hält an), Tür öffnen
TSMS Tür schließen, dann Motor starten
X Reaktion auf schweren Fehler

[29] Kommt in der Abbildung 4-8 ebenso wie X bisher nicht vor.

Nun kann man sich überlegen, was geschehen soll, wenn das Ereignis *s* eintritt, während der Fahrstuhl sich im Zustand ZWO befindet. Es drückt also jemand auf den Anforderungskopf außen, obwohl der Fahrstuhl sich mit geöffneter Tür auf dieser Ebene befindet, er oder sie also direkt einsteigen kann. Am besten wird das Ereignis ignoriert! Das heißt, es gibt keine Aktion (NOP) und der Zustand ändert sich nicht.

Ein anderer Fall ist gegeben, wenn das Ereignis *e* im Zustand ZWG oder ZWO eintritt. Der Sensor kann aber das Ereignis »Ziel erreicht« nur im Zustand »fahrend« (ZF) melden, weswegen dieses Ereignis eigentlich unmöglich und damit ein Fehler in der Hardware ist. Der Fehler muss von einem Mitarbeiter des Fahrstuhlherstellers behoben werden. Für solche Zwecke wird ein Fehlerzustand ZX vorgesehen.

Um den DEA zu vervollständigen, werden Fehlerzustand und die bisher fehlenden Ereignisse mit den jeweiligen Aktionen an den jeweiligen Zuständen nachgetragen. Aus Platzgründen werden ab jetzt nur noch die Abkürzungen verwendet. Das Ergebnis ist in Abbildung 4-9 dargestellt.

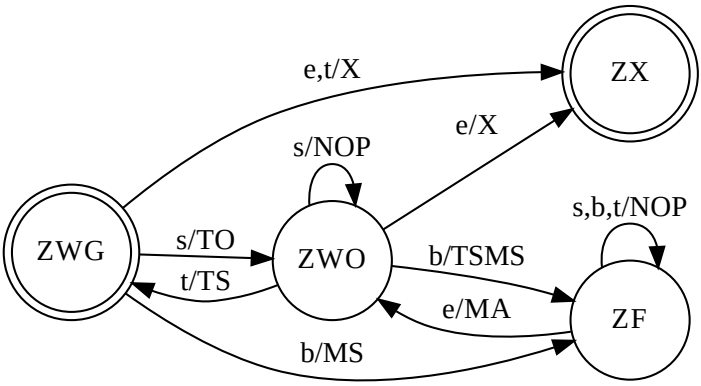


Abbildung 4-9: Vollständiger Zustandsautomat

Das Zustandsübergangsdiagramm ist äquivalent zu der Tabelle, die für jeden Zustand den Folgezustand in Abhängigkeit von den Eingaben/Ereignissen zeigt (siehe Tabelle 4-17).

	s	b	e	t
ZWG	ZWO	ZF	ZX	ZX
ZWO	ZWO	ZF	ZX	ZWG
ZF	ZF	ZF	ZWO	ZF

Tabelle 4-17: Folgezustände in Abhängigkeit vom vorherigen Zustand und den Ereignissen

Für den Test ist es jedoch interessanter, die Aktionen in Abhängigkeit von Zuständen und Ereignissen zu betrachten, weil die Folgezustände beim Test nicht beobachtet werden können (siehe Tabelle 4-18).

	s	b	e	t
ZWG	TO	MS	X	X
ZWO	NOP	TSMS	X	TS
ZF	NOP	NOP	MA	NOP

Tabelle 4-18: Aktionen bei den Zustandsübergängen (X – Reaktion auf Fehler)

Aus den Aktionen können Schlüsse auf die internen Zustandsänderungen gezogen werden. Dabei gibt es nicht immer ein eindeutiges Bild. So gibt es im fahrenden Zustand (ZF) bei den Eingaben *s*, *b* und *t* keine Aktionen (NOP) und es ist nicht direkt erkennbar, ob der Zustand unverändert geblieben ist oder ein anderer Zustand eventuell erreicht wurde.

4.8.2 Der minimale Zustandstest

Unter »minimal« (»lean«) verstehen wir, dass jeder Übergang, und damit auch jeder Zustand, mindestens einmal durchlaufen wird. Dies ist ein eher einfaches Kriterium zur Beendigung des zustandsbasierten Tests. Je nach den Anforderungen an die Qualität der Software wird dieses minimale Kriterium nicht immer ausreichen, doch dazu mehr im nächsten Abschnitt.

Wir können Zustandsübergänge und Folgen von Übergängen separat betrachten. Wenn ein Übergang von Zustand Z_i zum Zustand Z_j getestet werden soll, besteht die Folge der Eingabesymbole zum Testen des Automaten aus zwei Teilen: Die Startfolge dient dazu, den Anfangszustand Z_i der Folge ausgehend vom Startzustand zu erreichen. Sie ist leer, wenn Z_i selbst der Startzustand ist. Der zweite Teil der Folge besteht aus den Eingaben, die den Übergang vom Anfangszustand Z_i zum Endzustand Z_j bewirken – unsere eigentliche Testfolge. Dabei kann $i = j$ sein (Eingaben, die keine Zustandsänderung auslösen).

Ein Maß für ein Endekriterium beim zustandsbasierten Test ist die *N-Switch*-Überdeckung. N ist dabei die Anzahl der Zustände zwischen dem Anfangs- und dem Zielzustand der zu testenden Folge von Zuständen. Wenn nur ein Übergang von einem Zustand (Anfangszustand) direkt zum Folgezustand (Zielzustand) getestet werden soll – unser minimaler Test –, wird demzufolge von der 0-Switch-Überdeckung gesprochen. Der Name N-Switch deutet ja schon darauf hin, dass es noch weitere Kriterien gibt: Beinhaltet die zu testende Folge zwei Übergänge (einen Zustand zwischen Anfangs- und Zielzustand), wird eine 1-Switch-Überdeckung[30] angestrebt. Bei drei

[30] Das Maß wird auch als N-1-Switch-Überdeckung bezeichnet, wobei das N dann die Übergänge und nicht die Zwischenzustände repräsentiert.

Übergängen mit zwei Zuständen liegt eine 2-Switch-Überdeckung vor, usw. Dabei ist zu beachten, dass die Zustände in den Folgen auch dieselben sein können, wenn nämlich eine Eingabe den Zustand nicht ändert.

Aber kommen wir zurück zu unserem Beispiel: Wir haben die Startfolge zum Erreichen des Anfangszustands ermittelt und auch die Eingaben zum Test des Übergangs bzw. der Übergänge unserer zu testenden Folge. Nach Abarbeitung dieser Folgen landet der Prüfling in einem Zustand, der dem gewünschten Zielzustand entsprechen soll, den wir aber nicht kennen oder direkt abfragen können (Blackbox!). Es fehlt also noch der Nachweis, dass der erreichte Zustand tatsächlich dem erwarteten Zielzustand Z_j entspricht. Das wird mit einer nachgeschalteten Folge von Eingaben erreicht, die den Zielzustand Z_j verifiziert (siehe Abbildung 4-10).

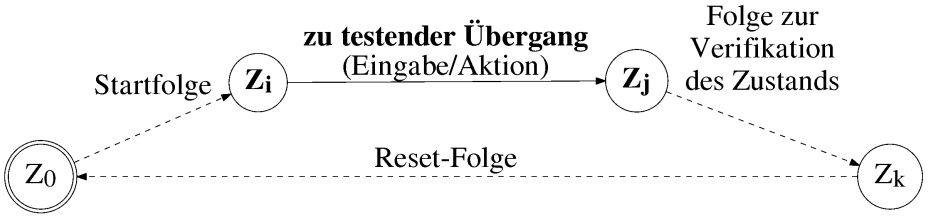


Abbildung 4-10: Test eines Übergangs

Falls der DEA eine Reset-Eingabe oder ein Reset-Symbol besitzt, kann direkt zum Startzustand des Automaten gegangen werden. Dieses ist in Abbildung 4-10 aufgenommen worden und erleichtert manchmal den Test.

Es gibt mehrere Ansätze zur Verifikation eines Zielzustands. Einer ist das Herausfinden einer eindeutigen Eingabe-/Ausgabefolge[31] für einen Zustand. Für so eine Folge gilt:

- Die Eingabe eines gültigen Symbols x erzeugt einen Übergang mit der Ausgabe y, wenn der DEA sich im Zustand Z_j befindet.
- Bei allen anderen Zuständen ergibt die Eingabe von x eine *andere*, nicht mit y übereinstimmende Ausgabe.

Damit ist klar, dass wirklich der Zielzustand Z_j erreicht wurde. Um so eine Folge für unser Fahrstuhl-Beispiel zu finden, genügt der Blick auf die Tabelle 4-18. So erzeugt die Eingabe von b im Zustand ZWO die Aktion TSMS. Das ist bei keinem anderen Zustand der Fall. Oder wenn die Aktion MS das Ergebnis der Eingabe b ist, kann der vorherige Zustand nur ZWG gewesen sein. Wenn ein Wert in der Spalte zu einer gegebenen Eingabe nur einmal auftritt, ist er zur Verifikation eines Zustands geeignet.

[31] Wir gehen nicht darauf ein, dass es Zustände geben kann, für die so eine Folge nicht existiert.

Wenn ausgehend von ZF *s* eingegeben wird, ist das Ergebnis hingegen nicht eindeutig. Die NOP-Aktion kann sowohl bei ZWO als auch ZF auftreten. Bisher genügt ein einziges Symbol zur Verifikation eines Zustands. Manchmal müssen es mehrere sein, wie im folgenden einfachen fiktiven Beispiel gezeigt wird (siehe Abbildung 4-11):

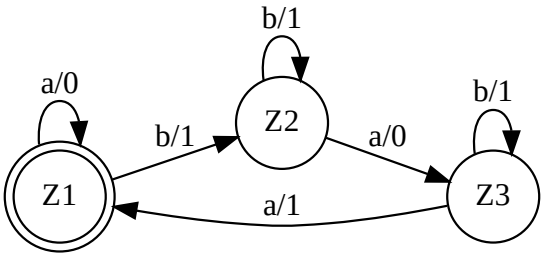


Abbildung 4-11: Verifikation von Z1 setzt zwei Eingaben voraus.

Die Tabelle 4-19 zeigt in der letzten Spalte die Folge der Ausgaben bei der Eingabefolge *aa*. Sie sehen, dass *a* allein Z1 nicht verifiziert, weil die Aktion 0 auch bei Z2 erzeugt wird. Die aus zwei Symbolen bestehende Eingabefolge *aa* liefert allerdings in jedem Zustand eine andere Folge von Aktionen und ist deshalb zur Verifizierung geeignet.

	a	b	aa
Z1	0	1	00
Z2	0	1	01
Z3	1	1	10

Tabelle 4-19: Aktionen in Abhängigkeit von Zustand und Eingabefolge

Bei unserem Fahrstuhl-Beispiel spielt das keine Rolle, aber es kann in anderen Fällen auftreten – deswegen unser Hinweis.

0-Switch-Überdeckung

Wie sehen nun die Eingaben für den Fahrstuhl aus, wenn eine 0-Switch-Überdeckung erreicht werden soll? Bei der 0-Switch-Überdeckung wird jeder Übergang separat betrachtet. Sehen wir uns den Zustandsautomaten in Abbildung 4-9 auf Seite 108 an, dann sind insgesamt zwölf[32] Übergänge im Automaten vorhanden. Somit sind zwölf Testfälle zu spezifizieren. Wir brauchen zunächst die Startfolge, um den jeweiligen Anfangszustand für den zu testenden Übergang zu erreichen, dann das Eingabesymbol zum Auslösen

[32]Einige Übergänge sind in der Abbildung zusammengefasst, daher sind nur 9 Pfeile/Kanten dargestellt.

des Übergangs und damit das Erlangen des Zielzustands. Abschließend benötigen wir noch die Eingaben zur Verifikation des Zielzustands, um den Nachweis zu erbringen, dass wirklich der zu testende Übergang zur Ausführung gekommen ist. Tabelle 4-20 zeigt die zwölf vorhandenen Übergänge und die erforderlichen Testsequenzen. Es sind alle Testfälle für das »Lean Testing«.

Nr.	Übergang	Start-folge	Eingabe-symbol	Verifikation mit	erwartete Ausgangsfolge
1	ZWG → ZWO	ϵ	s	b	TO-TSMS
2	ZWG → ZF	ϵ	b	e	MS-MA
3	ZWO → ZWG	s	t	s	TO-TS-TO
4	ZWO → ZWO	s	s	b	TO-NOP-TSMS
5	ZWO → ZF	s	b	e	TO-TSMS-MA
6	ZF → ZF	b	s	e	MS-NOP-MA
7	ZF → ZF	b	b	e	MS-NOP-MA
8	ZF → ZF	b	t	e	MS-NOP-MA
9	ZF → ZWO	b	e	b	MS-MA-TSMS
10	ZWG → ZX	ϵ	e	-	X
11	ZWG → ZX	ϵ	t	-	X
12	ZWO → ZX	s	e	-	TO-X

Tabelle 4-20: 0-Switch-Überdeckung für die Fahrstuhlsteuerung

Das Symbol ϵ zeigt eine leere Folge an (d.h., der Startzustand ist identisch mit dem Anfangszustand unseres zu testenden Übergangs). Weil der Fehlerzustand ein Endzustand ist, von dem kein Übergang wegführt, kann er nicht auf die beschriebene Art verifiziert werden. Man kann annehmen, dass er erreicht wurde, wenn es die Fehlerreaktion X gab. Weil bei manchen Übergängen verschiedene Eingaben dieselbe Aktion erzeugen, müssen manche Ausgangsfolgen gleich sein. Dies betrifft hier hauptsächlich die zustandserhaltenden NOP-Aktionen.

Sehen wir uns den ersten Testfall im Detail an: Der Fahrstuhl steht wartend mit geschlossener Tür (Anfangszustand), der Übergang zum Zielzustand – Fahrstuhl steht wartend mit geöffneter Tür – ist der zu testende Übergang. Anfangszustand ist auch der Startzustand des Automaten, wir benötigen somit keine Eingaben, um zum Anfangszustand des zu testenden Übergangs zu gelangen. Das Ereignis *s* ist die Eingabe für den Test, d.h., der Anforderungsknopf außerhalb des Fahrstuhls auf derselben Ebene wird gedrückt oder im haltenden Fahrstuhl wird innen »Tür öffnen« gedrückt. Es wird erwartet, dass als Aktion die Tür geöffnet und somit der Zustand »wartend mit geöffneter Tür« (ZWO) erreicht wird. Zum Nachweis erfolgt die Eingabe *b*, denn *b* ist die einzige Eingabe, die einen Zustandswechsel vom wartenden Fahrstuhl mit offener Tür zum fahrenden Fahrstuhl bewirkt.

Als Aktion wird somit erwartet, dass die Tür geschlossen wird, der Motor startet und der Fahrstuhl sich in Bewegung setzt. Sind die Aktionen Tür öffnen, Tür schließen, Motor starten in der Reihenfolge gegeben, ist der Testfall konform zu den Erwartungen und es liegt kein Fehlverhalten vor. Das Programm verhält sich somit konform zum Zustandsautomaten.

Mit den ersten beiden Testfällen werden die Eingaben von s und b geprüft. Beide Eingaben dienen für weitere Testfälle als Startfolgen. Daher ist es sehr sinnvoll, mit dem Test dieser beiden Eingaben zu beginnen und somit deren korrekte Bearbeitung im Programm zu prüfen.

4.8.3 Das Beispiel in C++

Der Prüfling wird durch ein Objekt der Klasse `Fahrstuhlsteuerung`[33] repräsentiert. Die Klasse hat eine Methode `char uebergang(char input)`, die als Parameter ein Zeichen des Typs `char` für das Eingabesymbol nimmt. Das zugehörige Ereignis wird von der Hardware geliefert und von der Schnittstelle in das betreffende Zeichen umgewandelt. Die Ausgabe ist ebenfalls ein Zeichen. Die Aktion 1 wird somit durch das Zeichen '1' repräsentiert. Von der Hilfsfunktion `run()` des Testprogramms (siehe unten) wird die Ausgabe in die Abkürzungen der Spezifikation transformiert, um diese Umsetzung nicht manuell vornehmen zu müssen. Die Ausgabe wird von der unterlagerten Steuerung in entsprechende Befehle zur Steuerung des Fahrstuhls umgesetzt. Die C++/Hardware-Schnittstelle spielt im Folgenden keine Rolle. Es soll vielmehr geprüft werden, ob die Methode `uebergang()` sich genau so wie der spezifizierte DEA verhält. Über die Interna der Methode haben wir keine Kenntnis. Die Testfälle der Tabelle 4-20 können leicht in ein Testprogramm umgesetzt werden (siehe Listing 4.36).

```
#include "gtest/gtest.h"
#include <string>
#include "Fahrstuhlsteuerung.h"
namespace {
  // Diese Funktion dient dazu, den Programmcode der Testfälle
  // zu verkürzen. Sie transformiert gleichzeitig den
  // internen Zifferncode in die Bezeichnung der Aktionen.
  const std::map<char, const std::string> zifferZuAktion
    { '0', "NOP"}, {'1', "TO"}, {'2', "TS"}, {'3', "MS"},
    {'4', "MA"}, {'5', "TSMS"}, {'X', "X"} };
  std::string run(Fahrstuhlsteuerung& fst,
                  const std::string& eingabefolge) {
    std::string erg;
    for (auto input : eingabefolge) {
```

[33] Das vollständige Beispiel finden Sie im Verzeichnis *beispiele/fahrstuhl*.

```
      erg += zifferZuAktion.at(fst.uebergang(input)) + " ";
    }
    erg.erase(--erg.end());  // letztes Leerzeichen wieder entfernen
    return erg;
  }
}

TEST(Zustand, TF1) {
  Fahrstuhlsteuerung fst;
  //              Eingabefolge   erwartete Ausgabefolge
  EXPECT_EQ( run(fst, "sb"), "TO TSMS");
}
TEST(Zustand, TF2) {
  Fahrstuhlsteuerung fst;
  EXPECT_EQ( run(fst, "be"), "MS MA");
}
TEST(Zustand, TF3) {
  Fahrstuhlsteuerung fst;
  EXPECT_EQ( run(fst, "sts"), "TO TS TO");
}
TEST(Zustand, TF4) {
  Fahrstuhlsteuerung fst;
  EXPECT_EQ( run(fst, "ssb"), "TO NOP TSMS");
}
// usw.
```

Listing 4.36: Auszug des Testprogramms

Wenn es ein Reset-Ereignis *r* gäbe, könnten alle Tests in eine einzige Folge gepackt werden. Allerdings wäre dann ein Fehler vermutlich schwerer zu finden.

4.8.4 Test von Übergangsfolgen

Bei der 0-Switch-Überdeckung wird jeder Übergang und damit auch jeder Zustand mindestens einmal durchlaufen. Im Folgenden wird der Test von längeren (>1) Folgen von Übergängen vorgestellt, falls der Lean-Ansatz als nicht ausreichend angesehen wird.

N-Switch-Überdeckung

Wie bereits oben erwähnt, kann die geforderte Switch-Überdeckung sich auch auf zwei oder mehr Übergänge beziehen. Damit erfolgt ein umfassenderer, intensiverer Test.

Bei unserem Fahrstuhl sind folgende Übergänge der Länge zwei vom Startzustand »wartend mit geschlossener Tür« (ZWG) aus gegeben (in Klammern sind die jeweiligen Eingaben aufgeführt):[34]

ZWG → ZWO → ZF (s,b)
ZWG → ZWO → ZWG (s,t)
ZWG → ZWO → ZWO (s,s)
ZWG → ZWO → ZX (s,e)
ZWG → ZF → ZWO (b,e)
ZWG → ZF → ZF (b,s)
ZWG → ZF → ZF (b,b)
ZWG → ZF → ZF (b,t)

Bei der 0-Switch-Überdeckung waren vier Übergänge (Testfälle, inkl. der Übergänge zum Fehlerzustand) vom Anfangszustand aus zu testen. Bei der 1-Switch-Überdeckung sind es acht, allein vom Anfangszustand aus (ohne die Übergänge zum Fehlerzustand der Länge 1). Bei den letzten drei aufgeführten Folgen liegen keine Unterschiede in den Aktionen vor, daher ist zu überlegen, ob alle drei Testfälle auszuführen sind oder ein Testfall als ausreichend angesehen werden kann.

Dies sind aber noch nicht alle möglichen Übergänge der Länge zwei. Vom Zustand »wartend mit offener Tür« (ZWO) sind folgende Testfälle erforderlich (die Eingaben zum Erreichen des Anfangszustands ZWO vom Startzustand ZWG aus sind nicht mit angegeben):

ZWO → ZWO → ZWO (s,s)
ZWO → ZWO → ZWG (s,t)
ZWO → ZWO → ZF (s,b)
ZWO → ZWO → ZX (s,e)
ZWO → ZWG → ZWO (t,s)
ZWO → ZWG → ZF (t,b)
ZWO → ZWG → ZX (t,e)
ZWO → ZWG → ZX (t,t)
ZWO → ZF → ZWO (b,e)
ZWO → ZF → ZF (b,s)
ZWO → ZF → ZF (b,b)
ZWO → ZF → ZF (b,t)

Bei den letzten drei Folgen und bei den beiden Folgen mit den Teilfolgen ZWG → ZX gibt es ebenfalls keine Unterschiede in den Aktionen.

Bleibt noch der Zustand »fahrend« (ZF) als Anfangszustand der zu testenden Folgen:

[34] Die Übergänge vom Startzustand zum Fehlerzustand (nur *e*, Länge 1) sind nicht aufgeführt.

ZF → ZWO → ZF (e,b)
ZF → ZWO → ZWG (e,t)
ZF → ZWO → ZX (e,e)
ZF → ZWO → ZWO (e,s)
ZF → ZF → ZF (s,s)

Für den letzten Übergang sind auch folgende weitere Eingabefolgen möglich: s,b s,t b,s b,t b,b t,s t,b und t,t (hier gibt es wieder keine Unterschiede in den Aktionen).

Wenn eine 1-Switch-Überdeckung zu 100% erreicht werden soll, kommen wir auf insgesamt 33 Testfälle (oder 20, bei Zusammenlegung der Testfälle im Zustand fahrend (ZF) bzw. wartend mit geschlossener Tür (ZWG)).

Im Gegensatz zur 0-Switch-Überdeckung sind bei der 1-Switch-Überdeckung auch Übergangsfolgen der Länge zwei ohne Zustandsänderungen zu berücksichtigen. Die Folge

ZWO → ZWO → ZWO (s,s)

ist beispielsweise so ein Testfall. Für unser Fahrstuhl-Beispiel mag der Testfall – Anforderungsknopf auf derselben Etage bei geöffneter Fahrstuhltür zweimal hintereinander drücken – wenig sinnvoll erscheinen. Bei anderen Testobjekten kann ein solcher Test aber durchaus ratsam sein, wenn beispielsweise interne Variablen den Zustand beeinflussen (oder speichern, als internes Gedächtnis) und bei einer wiederholten Eingabe ohne Zustandsänderung nicht richtig gesetzt bzw. zurückgesetzt werden.

Sehen wir uns noch einen Testfall für eine 2-Switch-Überdeckung mit den Eingaben und Aktionen genauer an:

ZWG → ZF → ZWO → ZWG (b,e,t) MS-MA-TS

Mit diesem Testfall werden alle Zustände (ohne Fehlerzustand) überdeckt. Mit ihm wird eine »komplette« Fahrstuhlfahrt mit folgendem Ablauf überprüft:

- Der Fahrstuhl steht wartend mit geschlossener Tür (ZWG), und es gibt eine Anforderung an der Fahrstuhl auf der anderen Ebene (*b*).
- Der Fahrstuhl setzt sich in Bewegung (MS) und ist im Zustand »fahrend« (ZF).
- Der Sensor meldet das Erreichen des Ziels (*e*), der Motor wird ausgestellt und die Tür geöffnet (MA). Dabei wird der Timer gestartet. Der erreichte Zustand ist ZWO.
- Der Ablauf des Timers erzeugt das Ereignis *t*. Die Tür schließt sich (TS) und der Fahrstuhl geht in den Zustand »wartend mit geschlossener Tür« (ZWG).

Ein Testfall, der sicherlich sinnvoll zur Ergänzung mit aufgenommen werden kann. Auf weitere Testfälle und längere Switch-Überdeckungen wird nicht weiter eingegangen. Das Prinzip zur Erzeugung längerer Zustandsübergangsfolgen ist hoffentlich klar geworden.

Testfallerzeugung mit einem Übergangsbaum

Ein weiterer Ansatz zum zustandsbasierten Test konzentriert sich nicht auf den separaten Test von einzelnen Übergängen, sondern auf Abfolgen von Zustandsübergängen.

Aus dem Zustandsautomaten wird ein sogenannter Übergangsbaum erstellt. Ziel dabei ist die Aufstellung von Ereignis- oder Eingabefolgen und die Eliminierung von Mehrfachzyklen. Wenn ein in der bisher erzeugten Folge bereits vorliegender Zustand wieder erreicht wird (einfacher Zyklus), wird die weitere Expandierung des Baums abgebrochen. In der Regel wird der noch nicht vervollständigte Zustandsautomat (siehe Abbildung 4-8 auf Seite 107) als Grundlage verwendet. Der Zustandsautomat wird durchlaufen und ein Übergangsbaum dabei wie folgt erzeugt:

1. Als Wurzel des Baums wird der Startzustand verwendet.
2. Jeder mögliche Zustandsübergang im Automaten zu einem Folgezustand wird im Übergangsbaum zu einem neuen Knoten, der den Folgezustand repräsentiert. Die Kante zwischen den beiden Knoten ist mit dem Ereignis (der Eingabe) verbunden, das den Übergang auslöst. Dieser Schritt wird so lange wiederholt, bis
   - ein bereits im Automaten besuchter Zustand erreicht wird (dadurch werden mehrfache Zyklen vermieden) oder
   - ein Zustand keine abgehenden Übergänge besitzt, also ein Endzustand ist.

Der so erzeugte Baum in Abbildung 4-12 *repräsentiert* somit alle möglichen Pfade[35] im Automaten vom Startzustand bis zu einem Endzustand bzw. bis zu einem Zustand, der bereits Bestandteil des bis dahin *aufgesammelten* Pfades ist. Jeder Pfad von der Wurzel des Baums zu einem Blatt *repräsentiert* einen Testfall, eine Folge von Eingaben (in Abbildung 4-12 mit römischen Zahlen gekennzeichnet). Diese Folge ist frei von zwei- oder mehrfachen Zyklen, da nach dem zweiten Erreichen eines Zustands keine weiteren Übergänge mehr im Übergangsbaum hinzu kommen.

Der Testfall V entspricht dem oben aufgeführten Testfall der 2-Switch-Überdeckung, also mit zwei Zwischenzuständen und insgesamt drei Übergängen. Die Testfälle III, IV, VI und VII enthalten ebenfalls drei Übergänge.

[35] Unter Pfad wird hier eine Folge von Zuständen und Zustandsübergängen verstanden.

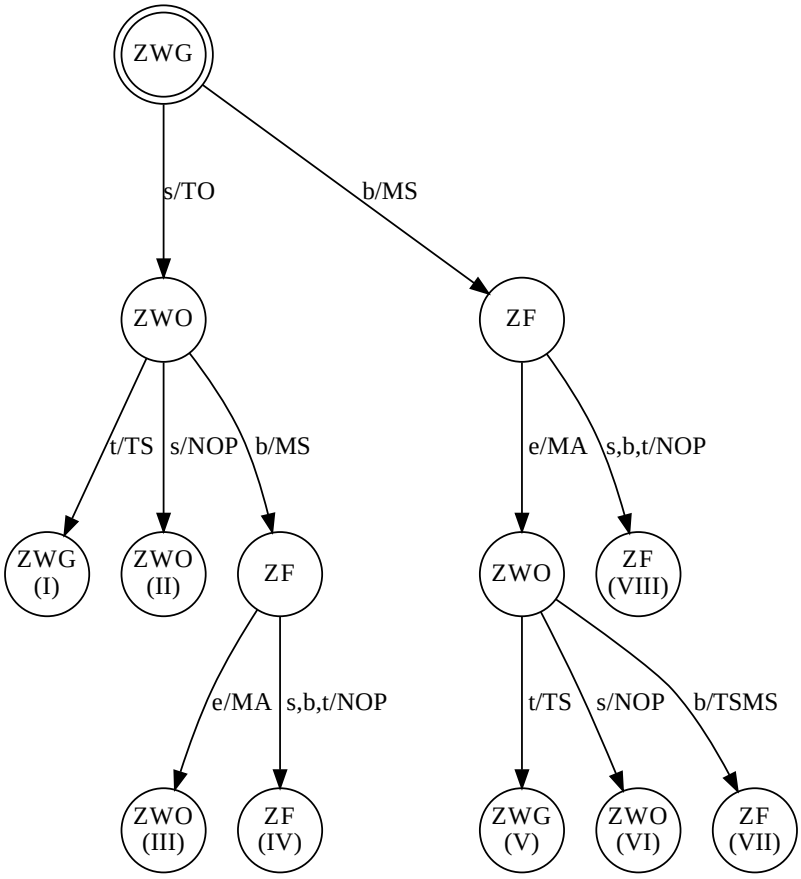


Abbildung 4-12: Übergangsbaum

Die verbleibenden drei Testfälle (I, II und VIII) enthalten nur einen Zwischenzustand (1-Switch-Überdeckung). Testfälle mit nur einem Übergang (ohne Zwischenzustand, 0-Switch-Überdeckung) werden bei diesem Vorgehen für unser Fahrstuhl-Beispiel nicht erzeugt.

Ausgangspunkt für die Erstellung des Übergangsbaums ist der noch nicht vervollständigte Automat. Der Übergangsbaum kann aber wie folgt ergänzt werden: Bei jedem Zustand (Knoten) wird geprüft, welche Eingaben bereits zu einem Zustandsübergang führen. Wenn Eingaben fehlen, werden diese entsprechend ergänzt. So fehlen beim Startzustand ZWG die Eingaben *e* und *t*, diese sind entsprechend aufzunehmen. Beide führen zum Fehlerzustand ZX, einem Endzustand. Der Übergang zum Fehlerzustand fehlt bisher auch beim Zustand ZWO (durch die Eingabe von *e*). An zwei Stellen im Baum ist diese Ergänzung vorzunehmen.

Durch die Ergänzung der fehlenden Eingaben werden auch die Fehlersituationen berücksichtigt, die im bisherigen Übergangsbaum nicht vorkommen. Der neue Übergangsbaum, in Abbildung 4-13 dargestellt, wird als erweiterter Übergangsbaum bezeichnet. Da nun auch Fehlersituationen überprüft werden, wird in diesem Zusammenhang auch vom Test auf Robustheit gesprochen.

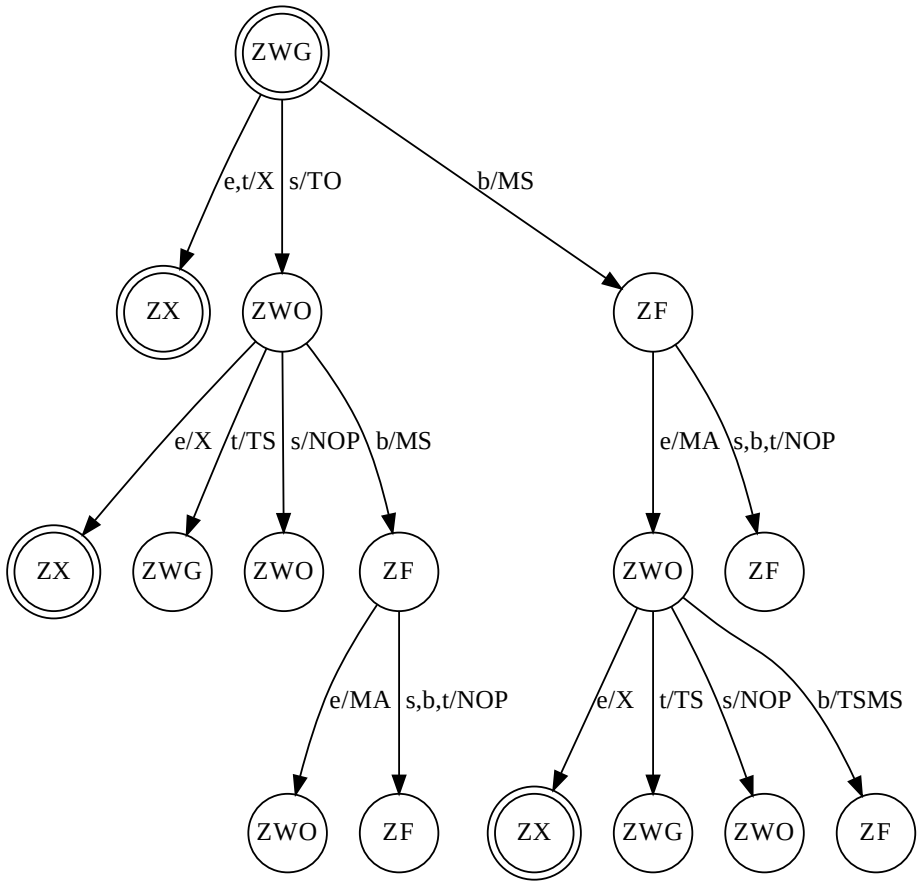


Abbildung 4-13: Erweiterter Übergangsbaum

Zur Vereinfachung sind in Abbildung 4-13 Eingaben, die zu gleichen Übergängen und Aktionen führen, zusammengefasst. Dies betrifft *e*,*t*/X von ZWG nach ZX und *s*,*b*,*t*/NOP von ZF nach ZF an zwei Stellen im Baum.

Wenn als Ausgangspunkt für die Erstellung des Übergangsbaums der vollständige Automat aus der Abbildung 4-9 auf Seite 108 verwendet wird, entsteht direkt der erweiterte Übergangsbaum mit allen möglichen Eingaben an allen Zuständen. Aus dem Übergangsbaum lassen sich die Testfälle

(Pfade von der Wurzel zum Blatt) ermitteln. Die Testfälle sind in der Tabelle 4-21 aufgelistet.

Nr.	Übergänge	Eingabe(n)	Aktion(sfolgen)
1	ZWG → ZX	e	X
2	ZWG → ZX	t	X
3	ZWG → ZWO → ZWG	s,t	TO-TS
4	ZWG → ZWO → ZWO	s,s	TO-NOP
5	ZWG → ZWO → ZX	s,e	TO-X
6	ZWG → ZWO → ZF → ZWO	s,b,e	TO-MS-MA
7	ZWG → ZWO → ZF → ZF	s,b,s	TO-MS-NOP
8	ZWG → ZWO → ZF → ZF	s,b,b	TO-MS-NOP
9	ZWG → ZWO → ZF → ZF	s,b,t	TO-MS-NOP
10	ZWG → ZF → ZWO → ZWG	b,e,t	MS-MA-MS
11	ZWG → ZF → ZWO → ZWO	b,e,s	MS-MA-NOP
12	ZWG → ZF → ZWO → ZWX	b,e,e	MS-MA-X
13	ZWG → ZF → ZWO → ZF	b,e,b	MS-MA-TSMS
14	ZWG → ZF → ZF	b,s	MS-NOP
15	ZWG → ZF → ZF	b,b	MS-NOP
16	ZWG → ZF → ZF	b,t	MS-NOP

Tabelle 4-21: Aus dem Übergangsbaum erzeugte Testfälle

Aus dem erweiterten Übergangsbaum ergeben sich 11 Testfälle bzw. 16 Testfälle, wenn die Eingaben *e*,*t* im Zustand ZWG und die Eingaben *s*,*b*,*t* im Zustand ZF (kommt zweimal vor) einzeln getestet werden. Die Testfälle sind leicht zu erzeugen: Das schon vorgestellte C++-Programm wird einfach um die Testfälle erweitert. Zum Beispiel sieht die Erweiterung mit Testfall 13 der Tabelle 4-21 so aus:

```
TEST(Zustand, TF13_Uebergangsbaum) {
  Fahrstuhlsteuerung fst;
  EXPECT_EQ( run(fst, "beb"), "MS MA TSMS");
}
```

Listing 4.37: Testfallerweiterung

Zur Erinnerung: Bei der 0-Switch-Überdeckung sind 12 Testfälle ausreichend und »lean«! Beim erweiterten Übergangsbaum sind hierfür 16 Testfälle erforderlich. Es steht damit 12 zu 16! Worin liegt also der Unterschied und welcher Ansatz ist zu bevorzugen?

Diskussion der beiden Ansätze

Mit der Erstellung des Übergangsbaums aus dem Zustandsdiagramm werden Mehrfachzyklen eliminiert und die Folgen von Zuständen und Über-

gängen (Pfad von der Wurzel zu einem Blatt) zur Erstellung von Testfällen sind einfach dem Baum zu entnehmen. Auf diese Weise entsteht eine minimale Anzahl von Testfällen, mit denen alle Zustandsübergänge (und somit alle Zustände) beim Test mindestens einmal zur Ausführung kommen. Da immer Folgen von Übergängen in einem Testfall überprüft werden, ist es schwierig nachzuprüfen, ob die tatsächliche Abfolge von einem Zustand zum nächsten – also ein einzelner Übergang – auch Teil dieser im Automaten so spezifizierten Folge ist. Dies ist nur an der Reihenfolge der Aktionen abzulesen. Dies gilt auch für die Prüfung, wie der letzte Zustand des Testfalls (Blatt im Übergangsbaum) zu verifizieren ist. Hier kann das bei der N-Switch-Überdeckung vorgestellte Vorgehen verwendet werden. Vorteil ist beim Übergangsbaum, dass alle Testfälle beim Startzustand beginnen und somit keine vorgeschalteten Startfolgen beim Test erstellt werden müssen.

Mit der N-Switch-Überdeckung werden die Längen von Folgen aus Zuständen und Zustandsübergängen vorgegeben. Dabei werden alle Folgen berücksichtigt, die sich aus dem Automaten ableiten lassen, und nicht nur solche, die beim Startzustand beginnen. Durch die Vorgabe der Länge der Folge werden auch Zyklen – und dies auch mehrfach bei langen Folgen – getestet. Da die Übergänge oder Folgen von Übergängen separat betrachtet werden, sind Startfolgen und Verifikationsfolgen erforderlich. Die separate Betrachtung ist von Vorteil, da auftretende Fehler sich meist direkt auf den getesteten Übergang beziehen.

> Leider lässt sich hier keine allgemein gültige Empfehlung für eines der beiden Verfahren zum zustandsbasierten Test geben. Wenn die Reihenfolgen der Zustände und Übergänge als »relativ sicher« angesehen werden können, reicht der Test auf Grundlage des Übergangsbaums aus. Es soll aber der erweiterte Übergangsbaum erzeugt werden, um auch Fehlersituationen zu prüfen. Besteht eine gewisse Unsicherheit, ob die Zustände und Übergänge in der spezifizierten Reihenfolge zur Ausführung kommen, lohnt sich ein separater Blick auf jeden Übergang bzw. jede Folge von Übergängen, und die N-Switch-Überdeckung ist dann die bessere Wahl.

Testendekriterium

Die Kriterien lassen sich je nach verwendetem Ansatz der Testfallerstellung unterschiedlich festlegen:

- Zustandsautomat: Die Überdeckung aller Zustände (Knotenüberdeckung) kann ein Endekriterium sein. Damit ist aber nicht sicherge-

stellt, dass auch alle Übergänge zur Ausführung kommen. Ein sinnvolles Kriterium ist daher die Überdeckung aller Zustandsübergänge.

- N-Switch-Überdeckung: Je nachdem welcher Wert für *N* gewählt wird, entstehen entsprechend lange Abfolgen von Zuständen und Übergängen. Mit N=0 wird jeder Übergang separat getestet. Dies wäre ein erstes minimales Lean-Kriterium zur Beendigung des Tests. Mit größerem N steigt die Zahl der erforderlichen Testfälle erheblich an, wie oben im Beispiel gezeigt. Mit größerem N werden auch Zyklen und damit das wiederholte Erreichen und Verlassen von Zuständen einem Test unterzogen. Oft treten aber gerade dabei Fehler auf, da bestimmte Konstellationen bei der Realisierung nicht bedacht wurden (z. B. Variablen werden nicht zurückgesetzt oder jedes Mal beim Erreichen des Zustands neu initialisiert). Durch die Wahl von N wird die Anzahl der zu erstellenden Testfälle festgelegt. Dadurch lässt sich der Testaufwand und damit die Testintensität sehr gut steuern.
- Übergangsbaum: Die Testfälle ergeben sich aus den Pfaden von der Wurzel zu einem Blatt. Abstufungen der Testintensität sind nicht möglich.
- Übergangstabelle: Wird der separate Test jedes Übergangs als ausreichend angesehen, kann die Zustandsübergangstabelle herangezogen werden. Der Test jedes Übergangs gilt dann als ausreichend. Aus der Übergangstabelle lassen sich auch längere Folgen von Übergängen ablesen, allerdings ist dieses Vorgehen sehr mühsam und fehleranfällig und liefert die gleichen Folgen wie die N-Switch-Überdeckung.

Bewertung

Beim Test von Software mit Zuständen und Übergängen wird der Einsatz des zustandsbasierten Tests zur systematischen Herleitung der Testfälle empfohlen, da die anderen Testentwurfsverfahren nicht auf das unterschiedliche Verhalten in Abhängigkeit vom Zustand eingehen.

Welcher der vorgestellten Ansätze und welche Überdeckungskriterien sind »lean«? Alle Zustandsübergänge zu testen, ist die Mindestforderung. Ob diese Übergänge separat getestet werden (0-Switch-Überdeckung) oder vom Startzustand aus (Übergangsbaum), kommt auf die Beschaffenheit des Testobjekts an (siehe obige Diskussion der beiden Ansätze).

Fehlersituationen sind beim zustandsbasierten Test – besonders bei technischen Systemen – stets zu berücksichtigen. Der Automat als Grundlage für die N-Switch-Überdeckung und zur Erzeugung des Übergangsbaums ist auf Vollständigkeit zu prüfen und ggf. vorab zu ergänzen. Die Zustandsübergangstabelle kann sehr gut zur Prüfung des Automaten dienen und fehlerhafte oder fehlende Übergänge aufdecken.

Bezug zu anderen Testverfahren

Direkte Bezüge zu anderen Testverfahren sind nicht gegeben. Jedoch kann zum Grenzwerttest eine Analogie gesehen werden, wenn der Übergang von einem Zustand in einen Folgezustand beispielsweise durch eine kleine Änderung erfolgt.

Ein Beispiel: Ein Stapel hat eine begrenzte Aufnahme. Er kann die Zustände *leer*, *gefüllt* und *voll* annehmen. Aus einem gefüllten Stapel wird ein voller Stapel (Zustandsübergang), wenn der letzte freie Platz im Stapel durch ein Element belegt wird. Die Grenze zwischen den beiden Zuständen *gefüllt* und *voll* wird somit überschritten. Insofern gibt es Analogien zum Grenzwerttest.

Hinweise für die Praxis

Der zustandsbasierte Test ist überwiegend bei technischen Systemen anwendbar. Hier ist sein Einsatz auch zu empfehlen, aber vermutlich nicht ausschließlich der Lean-Ansatz, sondern auch eine 1-Switch-Überdeckung, um das Verhalten des Systems bei Eingaben ohne Zustandsänderung (Zyklen) zu testen. Es muss ja keine 100% Überdeckung angestrebt werden. In der Praxis reicht es möglicherweise aus, die kritischen Übergänge zu identifizieren und diese intensiv zu testen. Ein solch aufwendiger Test ist bei technischen Systemen gerechtfertigt, da Fehler gravierende Auswirkungen haben können.

Grundlage des zustandsbasierten Tests ist der Zustandsautomat. Fehler in der Beschreibung des Automaten können durch Tests, die auf diesem Automaten beruhen, nicht erkannt werden. Vor der Nutzung des Automaten ist seine Qualität zu sichern, z. B. durch Erstellung einer Zustandsübergangstabelle zur Prüfung des Automaten oder durch ein Review mit mehreren Kollegen (4-Augen-Prinzip).

Der Zustandsautomat soll auch Grundlage für die Implementierung sein, also als Teil der Spezifikation zur Verfügung stehen. Leider ist dies in der Praxis nicht immer der Fall. Wenn der Automat erst zu Testzwecken erstellt wird, besteht das Risiko, dass die Implementierung auf einem anderen Automaten – der vielleicht nur im Kopf des Entwicklers existiert – erfolgte. Entwicklung und Test setzen dann auf unterschiedlichen Automaten auf.[36] Die Ergebnisse des Tests lassen dann keine direkten Rückschlüsse auf die Implementierung zu.

In der Praxis sind Zustandsmodelle oft sehr groß, es existieren viele Zustände und Übergänge. Mit hierarchischen Zustandsautomaten kann die Komplexität aufgeteilt und damit handhabbarer gestaltet werden.

[36] Kann beim Entwicklertest nicht vorkommen, da Programmierung und Test von einer Person – dem Entwickler – durchgeführt wird.

4.9 Syntaxtest

Beim zustandsbasierten Test ist das Zustandsmodell Ausgangsbasis für die Ermittlung der Testfälle. Zustandsübergänge können durch Eingaben ausgelöst werden. Womit wir wieder bei der grundsätzlichen Herausforderung beim Testen sind: Wie kann mit einer sinnvoll gewählten kleinen Zahl von unterschiedlichen Eingaben das Testobjekt ausreichend getestet werden?

> Müssen die Eingaben einer definierten Syntax genügen, sind sie also in einer festgelegten Art und Weise aufgebaut und strukturiert, ist der Syntaxtest ein adäquates Vorgehen zur Ermittlung von Testfällen.

Wann ist der Einsatz sinnvoll?

Daten unterliegen oft einer festgelegten Struktur, die bei den Anforderungen zu definieren ist. Beispiele sind die deutschen fünfstelligen Postleitzahlen, ein Tagesdatum, die IBAN im Bankbereich[37] oder auch die Angabe von Geldbeträgen.

Bei den fünfstelligen Postleitzahlen, die nur aus Ziffern bestehen – also konform zur Syntax sind –, ist aber noch zu testen, ob eine Ziffernfolge auch einer Postleitzahl entspricht. Diese Prüfung ist dann aber nicht mehr Teil des Syntaxtests. Die einzuhaltende Syntax für Postleitzahlen lässt ist aber präziser formulieren, sodass der Syntaxtest sich nicht nur auf die Anzahl und die Art der Zeichen beschränkt. So kann die Syntax festlegen, dass eine Postleitzahl nicht mit den Ziffernfolgen 00, 05, 43 und 62 beginnt.[38] Präzisierungen, die inhaltliche Aussagen in die Syntax verlegen, können sehr aufwendig sein. So ist die Postleitzahl 10000 syntaktisch korrekt, aber ein Sonderfall und keinem Postzustellbereich zugeordnet. In Abschnitt 4.9.2 wird auf die Problematik der einfachen oder umfassenderen Syntax eingegangen.

Reguläre Ausdrücke als Grundlage

Zur Formulierung der Syntax werden meist reguläre Ausdrücke verwendet. Der Ausdruck definiert, wie das Datum strukturiert ist, wann welche

[37] Die Syntax des Tagesdatums und der IBAN werden bei den folgenden C++-Beispielen ausführlich diskutiert.

[38] Die ersten zwei Ziffern bezeichnen die Postleitregionen. Für die genannten Ziffernfolgen ist keine Postleitregion definiert. Quelle: *https://de.wikipedia.org/wiki/Postleitzahl_%28Deutschland%29*.

Zeichen an welchen Stellen vorkommen dürfen oder müssen. Aus dem Ausdruck geht hervor, welche Struktur und/oder welcher Inhalt syntaktisch korrekt ist und welche bzw. welcher nicht. So ist beispielsweise bei einem Eurobetrag festzulegen, ob Eurocent durch ein Komma abzutrennen und auszuweisen sind (z.B. 1,00 Euro statt 1 Euro).

Wir gehen davon aus, dass der Leser mit regulären Ausdrücken vertraut ist. Daher werden diese hier nicht detailliert erläutert. In den beiden folgenden Beispielen in C++ werden die regulären Ausdrücke schrittweise abgeleitet, sodass deren Aufbau leicht nachvollzogen werden kann.

Grundidee und Beispiel

Ausgehend von der Syntax können zwei Testansätze unterschieden werden:

- Testfälle, die die Syntax einhalten, und
- solche, die die Syntax verletzen.

Ein Testfall für die Einhaltung der Syntax bei den Postleitzahlen ist »28199«, für die Postleitzahl der Hochschule Bremen. Das Testdatum hat die korrekte Länge 5 und enthält nur Ziffern. Beide Aspekte der Syntaxbeschreibung sind damit erfüllt. Ob es auch die Postleitzahl »12345« gibt, ist mit anderen Testansätzen zu kontrollieren, syntaktisch ist dieses Testdatum ebenfalls korrekt.

Für die zweite Gruppe von Testfällen ist zuerst die Syntax zu untersuchen, um festzustellen, welche einzelnen Festlegungen definiert sind. Für die Postleitzahl ist es die Länge (genau 5) und die Art der Zeichen (Ziffern von 0-9). Die Verletzung der beiden Aspekte ist zu prüfen.

Bei der Länge ist eine Unter- und Überschreitung mit jeweils einem Testfall zu kontrollieren. Als Längen bieten sich hier die Länge 4 mit dem Testdatum »1234« und die Länge 6 mit dem Testdatum »123456« an. Bei beiden Testfällen wird erwartet, dass eine Fehlermeldung ausgegeben wird, eventuell mit dem Hinweis der Unter- oder Überschreitung der Länge der Postleitzahl. Auf weitere Testfälle mit anderen Längen (< 4 und > 6) kann verzichtet werden, weil davon auszugehen ist, dass wenn eine kleine Über- oder Unterschreitung erkannt wird, dies auch bei größeren der Fall sein wird.

Wie sehen nun die Testfälle aus, die nachweisen, dass andere Zeichen außer Ziffern erkannt und mit einer entsprechenden Fehlermeldung abgewiesen werden? Hier eine kleine Auswahl von Möglichkeiten:

1. 1234A
2. 123A4
3. 12A34
4. 1A234

5. A1234
6. 1234?
7. 123??
8. 12???
9. 1????
10. ?????
11. ?1234
12. ??123
13. ???12
14. ????1
15. ?2?4?
16. ...

Es sind nahezu beliebig viele Testfälle möglich, aber sicherlich nicht unbedingt alle sinnvoll. Wie sieht ein Lean-Testing-Ansatz aus? Wovon können wir ausgehen? Vermutlich wird in unserem Testobjekt geprüft, ob eine Ziffer vorliegt, alle anderen Zeichen werden als fehlerhaft zurückgewiesen. Damit reicht eine beliebige *Nicht-Ziffer* aus, um das korrekte Arbeiten des Testobjekts nachzuweisen. Welches Sonderzeichen oder welcher Groß- oder Kleinbuchstabe verwendet wird, ist dabei belanglos.

Wie sieht es aber mit der Position und der Anzahl von Nicht-Ziffern im Testdatum aus? Was müssen wir hier bei unseren Testüberlegungen berücksichtigen? Vermutlich erfolgt die Prüfung ob Ziffer oder nicht in einer Schleife, dann hat auch die Position der Nicht-Ziffer keine Relevanz. Eine beliebige Position in einem Testfall reicht aus. Aber die Ränder sind spannende Testfälle (Nummern 1 und 5 in der Aufzählung oben).

Sind auch Testfälle durchzuführen, die mehrere Nicht-Ziffern in der Postleitzahl enthalten? Vermutlich wird die Schleife der Ziffernprüfung mit einem Fehlercode verlassen, wenn eine Nicht-Ziffer erkannt wird. Eine weitere Prüfung des folgenden Zeichens wird dann nicht erfolgen.

Wir haben zur Auswahl der Testfälle oder genauer zur Beschränkung der Anzahl der Testfälle Annahmen über das Testobjekt getroffen, die spekulativ sind. Wir gehen dabei von einem sauberen und einfachen Programmierstil aus. Unser Buch richtet sich in erster Linie an die Entwickler. Der Entwickler weiß ja, wie er das Testobjekt programmiert hat, und kann die von uns getroffenen Spekulationen als richtig oder falsch einstufen. Wenn die Annahmen nicht bestätigt werden können, verursacht dies einen höheren Testaufwand. Oder die bessere Alternative: eine Änderung und damit Verbesserung des Programmtextes.

Bitte beachten Sie, dass jeder Testfall nur einen der beiden zu prüfenden Aspekte (Länge, Ziffer) untersucht. Eine Mischung der beiden Aspekte (»1?34« – Sonderzeichen und falsche Länge) in einem Testfall ist zu vermeiden, da bei einer Fehlerrückmeldung nicht garantiert werden kann, wo-

durch (Länge oder Nicht-Ziffer) der Fehler ausgelöst wurde. Dann werden zusätzliche Testfälle mit getrennter Prüfung der Aspekte benötigt. Ein Lean-Syntaxtest umfasst damit die folgenden, in Tabelle 4-22 aufgeführten fünf Testfälle:

	Testdatum	Erwartetes Ergebnis
1	12345	Syntax eingehalten
2	1234	Syntax verletzt, zu kurz
3	123456	Syntax verletzt, zu lang
4	1234A	Syntax verletzt, Nicht-Ziffer
5	?2345	Syntax verletzt, Nicht-Ziffer

Tabelle 4-22: Testfälle für den Syntaxtest von Postleitzahlen

Auffällig dabei ist, dass für die Überprüfung der Einhaltung der Syntax nur ein Testfall erforderlich ist. Für den Nachweis der Verletzung der Syntax sind in aller Regel mehr Testfälle notwendig, da jede Verletzung eines Aspekts einzeln zu testen ist und es dabei oft mehrere Varianten gibt. Beim fünften Testfall wurde eine andere Nicht-Ziffer (?) als beim vierten Testfall (A) verwendet, um eine Variation bei den Nicht-Ziffern zu testen. Dadurch wird eine unserer Vermutungen über den Programmtext, dass nämlich jede beliebige Nicht-Ziffer erkannt wird, erhärtet.

4.9.1 Das Beispiel in C++ – Variante 1

Ein jeder kennt wohl die IBAN (International Bank Account Number). Sie besteht aus dem Ländercode, einer zweistelligen Prüfziffer, der Kennzeichnung des Bankinstituts und der Kundenkontonummer. Die beiden letzten Größen können aus insgesamt maximal 30 alphanumerischen Zeichen bestehen. So sind etwa in der Schweiz Kontonummern möglich, die Großbuchstaben enthalten. Der Einfachheit halber beschränken wir uns im Folgenden auf die Syntax der deutschen IBAN. Sie besteht nach [URL: IBAN] aus dem Ländercode `DE`, einer zweistelligen Prüfziffer, der 8-stelligen Bankleitzahl und der 10-stelligen Kontonummer. Bis auf den Ländercode sind alle Zeichen Ziffern, die Syntax ist also sehr einfach. Manchmal wird die IBAN zur besseren Lesbarkeit in Vierergruppen, die durch ein Leerzeichen getrennt sind, notiert.

Um eine Bankleitzahl zu prüfen, genügt es normalerweise nicht, nur die Korrektheit der Syntax festzustellen. Die Prüfziffer, die nach einem bestimmten Verfahren aus den anderen Zeichen berechnet wird, muss ebenfalls stimmen. Weil es in diesem Abschnitt um den Syntaxtest geht, wird auf die Ermittlung der Prüfziffer verzichtet.[39]

[39] Sie finden dazu eine Funktion in den downloadbaren Beispielen.

Die Prüfung, ob die Syntax stimmt, ist sehr einfach: Nach dem Entfernen der Leerzeichen muss die verbleibende Zeichenkette aus 22 Zeichen bestehen. Dabei bilden die ersten beiden Zeichen den Ländercode `DE` und die restlichen 20 Zeichen sind Ziffern. Als regulärer Ausdruck formuliert: `DE\d{20}`. Dabei steht `\d` für eine Ziffer, ist also dasselbe wie `[0-9]`. Wenn zwischen den Ziffern beliebig viele Leerzeichen möglich sein sollen, lautet die Syntax: `DE(\d *){20}`. Die Gruppe `(\d *)` kommt genau 20 Mal vor. Das *-Symbol bedeutet, dass das davorstehende Element, hier ein Leerzeichen, keinmal oder aber beliebig oft vorkommen kann. Im Testprogramm kommt die Syntax nicht vor, aber in der Funktion, die die Syntax prüft. Das folgende Testprogramm zeigt zwei gültige und einige ungültige IBANs:

```
#include <gtest/gtest.h>
#include "istIBANsyntaxgueltig.h"

TEST(IBAN_Syntax, TF1) {
  EXPECT_TRUE(istIBANsyntaxGueltig("DE10123456780123456789"));
}

TEST(IBAN_Syntax, TF2_Leerzeichen) {
  EXPECT_TRUE(istIBANsyntaxGueltig("DE10 12345678  0123456789"));
}

TEST(IBAN_Syntax, TF3_Schweizer_IBAN) {
  EXPECT_FALSE(istIBANsyntaxGueltig("CH10 002300A1 02350 2601"));
}

TEST(IBAN_Syntax, TF4_Ziffer_zu_viel) {
  EXPECT_FALSE(istIBANsyntaxGueltig("DE10 192345678  0123456789"));
}

TEST(IBAN_Syntax, TF5_Ziffer_zu_wenig) {
  EXPECT_FALSE(istIBANsyntaxGueltig("DE10 12345678  01256789"));
}

TEST(IBAN_Syntax, TF6_enthaelt_Buchstabe) {
  EXPECT_FALSE(istIBANsyntaxGueltig("DE10 192345678  01A3456789"));
}
```

Listing 4.38: IBAN-Syntaxtest *(beispiele/syntax/iban/testibansyntax.cpp)*

Die dazu passende C++-Funktion ist sehr kurz. Beachten Sie, dass der Backslash im regulären Ausdruck bei der Übergabe als Parameter verdoppelt werden muss.

```
#include <regex>
// ...
bool istIBANsyntaxGueltig(const std::string& iban) {
  return std::regex_match(iban, std::regex("DE(\\d *){20}"));
}
```

Listing 4.39: Funktion zur Syntaxprüfung

Das Ergebnis der IBAN-Syntaxprüfung mit dem obigen Programm ist:

```
Running main() from gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from IBAN_Syntax
[ RUN      ] IBAN_Syntax.TF1
[       OK ] IBAN_Syntax.TF1 (1 ms)
[ RUN      ] IBAN_Syntax.TF2_Leerzeichen
[       OK ] IBAN_Syntax.TF2_Leerzeichen (1 ms)
[ RUN      ] IBAN_Syntax.TF3_Schweizer_IBAN
[       OK ] IBAN_Syntax.TF3_Schweizer_IBAN (1 ms)
[ RUN      ] IBAN_Syntax.TF4_Ziffer_zu_viel
[       OK ] IBAN_Syntax.TF4_Ziffer_zu_viel (1 ms)
[ RUN      ] IBAN_Syntax.TF5_Ziffer_zu_wenig
[       OK ] IBAN_Syntax.TF5_Ziffer_zu_wenig (1 ms)
[ RUN      ] IBAN_Syntax.TF6_enthaelt_Buchstabe
[       OK ] IBAN_Syntax.TF6_enthaelt_Buchstabe (1 ms)
[----------] 6 tests from IBAN_Syntax (6 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (6 ms total)
[  PASSED  ] 6 tests.
```

Listing 4.40: Testergebnis der IBAN-Syntaxprüfung

4.9.2 Das Beispiel in C++ – Variante 2

Der obige Abschnitt zeigt das Prinzip des Syntaxtests. Das nun folgende Beispiel lenkt den Fokus darauf, dass »Lean Testing« durch »Lean Programming« stark unterstützt wird.

Ein Datum besteht in der deutschen Schreibweise zunächst einmal aus ein bis zwei Ziffern für den Tag, gefolgt von einem Punkt, ein bis zwei Ziffern für den Monat, gefolgt von einem Punkt, und vier Ziffern für das Jahr. Ein passender regulärer Ausdruck (von vielen möglichen) ist `\d\d?\.\d\d?\.\d\d\d\d` oder kürzer `(\d\d?\.){2}\d{4}`, weil Tag und Monat derselben Syntax unterliegen. Das Fragezeichen bedeutet, dass das direkt

davor stehende Element, hier das Ziffernzeichen, entweder nicht auftritt oder genau einmal. Ein Punkt ohne vorangestellten Backslash steht für ein beliebiges Zeichen, `\.` jedoch für einen Punkt.

Am obigen Beispiel wird sofort ein Problem klar: 99.9.9999 ist ein ungültiges Datum, wird aber von der gegebenen Syntax als gültig akzeptiert. Der Grund liegt darin, dass die alleinige Betrachtung der Syntax die Semantik (Bedeutung) des Inhalts ignoriert. Was tun? Es gibt verschiedene Abstufungen der Prüfung:

1. Die einfachste Prüfung ist die der oben dargestellten Syntax. Zusätzlich müssen Tag, Monat und Jahr aus dem Datumstring extrahiert werden, um dann die Gültigkeit zu prüfen.
2. Man kann aber auch in Erwägung ziehen, bereits bei der syntaktischen Analyse die Gültigkeit der Daten zu überprüfen.

   Für den Tag gibt es folgende Möglichkeiten:

   - Wenn der Tag nur eine Ziffer hat, liegt sie im Bereich `[1-9]`. Der 0.4.2016 ist nicht möglich.
   - Wenn bei zwei Ziffern die erste eine 0 ist, liegt die zweite im Bereich `[1-9]`, etwa beim 09.4.2016.
   - Ist die erste Ziffer eine 1 oder eine 2, ist als zweite zusätzlich die 0 möglich. Der erlaubte Bereich für die zweite Ziffer ist damit `[0-9]`.
   - Ist die erste Ziffer aber eine 3, kann die zweite nur 0 oder 1 sein.

   Der erste und der zweite Punkt werden mit dem regulären Ausdruck `0?[1-9]` realisiert, der dritte mit `(1|2)\d`, der vierte mit `3[01]`. Der reguläre Ausdruck für den Tag einschließlich des Punktes kann damit als `(0?[1-9])|((1|2)\d)|(3[01])\.` formuliert werden. Der senkrechte Strich `|` bedeutet »oder«.

   Für den Monat gilt:

   - Wenn der Monat einstellig ist oder mit einer 0 beginnt, liegt die Ziffer vor dem Punkt im Bereich `[1-9]`.
   - Wenn der Monat zweistellig ist, kann die erste Ziffer nur eine 1 sein und die zweite liegt im Bereich `[0-2]`.

   Der reguläre Ausdruck für den Monat einschließlich des Punktes kann damit als `(0?[1-9])|(1[0-2])\.` formuliert werden.
3. Mit der vorstehenden Lösung ist aber noch nicht alles erreicht, denn manche Monate haben nur 30 Tage und der Februar sogar nur 28 bzw. 29 Tage in einem Schaltjahr. 30 oder 28 Tage lassen sich mit einigem Aufwand in der Syntax berücksichtigen, aber für das Schaltjahr ist ein sehr großer, unverhältnismäßiger Aufwand erforderlich.

Sie sehen, man kann einen recht hohen Aufwand treiben. Bei der Überprüfung von Dialogeingaben oder von einer Datei eingelesenen Informationen ist daher abzuwägen, wie sich der Gesamtaufwand aus Syntaxprüfung und inhaltlicher Prüfung minimieren lässt.

> Eine komplexe Syntax ist schwieriger zu programmieren und zu testen als eine einfache. Deshalb unsere Empfehlung: Bevorzugen Sie einfache reguläre Ausdrücke, ergänzt durch eine inhaltliche Prüfung. Letztere würde durch eine komplexere Syntaxprüfung nicht wesentlich vereinfacht.

In unserem Beispiel gehen wir genau diesen Weg. Dabei lassen wir die inhaltliche Prüfung an dieser Stelle weg, weil sie in Abschnitt 4.3.4 beschrieben wird. Grundlage der Syntaxprüfung ist der oben erwähnte reguläre Ausdruck `(\d\d?\.){2}\d{4}`. Im Testprogramm kommt die Syntax nicht vor, aber in der Funktion, die die Syntax prüft. Das folgende Testprogramm zeigt mehrere Datumsangaben mit gültiger (Testfälle 1 bis 5) und mit ungültiger Syntax (Testfälle 6 bis 10):

```
#include <gtest/gtest.h>
#include "istDatumSyntaxGueltig.h"

TEST(Datumsyntax, TF1_korrekterWert) {
  EXPECT_TRUE(istDatumSyntaxGueltig("11.11.2016"));
}

TEST(Datumsyntax, TF2_SyntaxOK_falscherWert) {
  EXPECT_TRUE(istDatumSyntaxGueltig("30.2.2017"));
}

TEST(Datumsyntax, TF3) {
  EXPECT_TRUE(istDatumSyntaxGueltig("0.0.1111"));
}

TEST(Datumsyntax, TF4) {
  EXPECT_TRUE(istDatumSyntaxGueltig("99.99.9999"));
}

TEST(Datumsyntax, TF5) {
  EXPECT_TRUE(istDatumSyntaxGueltig("00.88.0000"));
}
```

```
TEST(Datumsyntax, TF6_Buchstabe) {
  EXPECT_FALSE(istDatumSyntaxGueltig("x0.2.2016"));
}

TEST(Datumsyntax, TF7_Jahr_zu_kurz) {
  EXPECT_FALSE(istDatumSyntaxGueltig("1.1.123"));
}

TEST(Datumsyntax, TF7_Jahr_zu_lang) {
  EXPECT_FALSE(istDatumSyntaxGueltig("1.1.12345"));
}

TEST(Datumsyntax, TF8_Tag_fehlt) {
  EXPECT_FALSE(istDatumSyntaxGueltig(".12.2001"));
}

TEST(Datumsyntax, TF9_Monat_fehlt) {
  EXPECT_FALSE(istDatumSyntaxGueltig("1..2001"));
}

TEST(Datumsyntax, TF10_Komma_statt_Punkt) {
  EXPECT_FALSE(istDatumSyntaxGueltig("23,12,1995"));
}
```

Listing 4.41: Datum-Syntaxtest *(beispiele/syntax/datum/testdatumsyntax.cpp)*

Die dazu passende C++-Funktion ist sehr kurz – eben »lean«! Auch hier wird der Backslash im regulären Ausdruck bei der Übergabe als Parameter verdoppelt.

```
#include <regex>
// ...
bool istDatumSyntaxGueltig(const std::string& datum) {
  return std::regex_match(datum, std::regex("(\\d\\d?\\.){2}\\d{4}"));
}
```

Listing 4.42: Mögliche Funktion zur Syntaxprüfung

Eine ausführlichere syntaktische Analyse würde den regulären Ausdruck verkomplizieren, ihn schwerer lesbar und schwerer testbar machen. Einfacher und verständlicher ist eine nachgeschaltete inhaltliche Prüfung.

4.10 Zufallstest

Der Zufallstest ist ein Verfahren, bei dem Programme mit (pseudo-)zufällig erzeugten Eingangsdaten »gefüttert« werden.

Wann ist der Einsatz sinnvoll?

Zufallstests können sinnvoll eingesetzt werden, zum Beispiel um das Belastungsverhalten eines Servers zu prüfen, der mit (pseudo-)zufälligen Nutzeranfragen bombardiert wird. Die Verteilung der Anfragen kann den Häufigkeiten je nach Tageszeit entsprechen. Solche Tests sind hier nicht das Thema. Wir beschränken uns auf funktionale Tests. Dafür scheinen Zufallstests weniger geeignet.

Grundidee

Die Eingabedaten werden mithilfe eines Programms gebildet, das einen Zufallszahlengenerator zur Erzeugung benutzt. C++ bietet mehrere Generatoren für Pseudozufallszahlen an. »Pseudo« deshalb, weil die Generatoren deterministisch arbeiten. Damit kann eine Reproduzierbarkeit der Ergebnisse gewährleistet werden. Nicht reproduzierbare Zufallszahlen sind möglich, aber für Tests kaum sinnvoll.

Ein offensichtlicher Vorteil scheint zu sein, dass man sich keine Gedanken machen muss – *ein* fehlerproduzierender Wert wird schon dabei sein, wenn sehr viele verschiedene Daten eingegeben werden! Tatsächlich kann diese Annahme jedoch falsch sein, wie unten zu sehen ist.

Ein Nachteil ist ebenso offensichtlich: Was ist das Testorakel, d.h., welche Ausgabewerte muss das Programm bei zufälligen Eingabewerten liefern? Das ist oft nicht oder nur mit sehr großem Aufwand zu ermitteln. Eine Ausnahme davon sehen Sie im Beispiel unten.

Zufallstests sind *in der Regel* nicht geeignet, um die Funktionalität eines Programms gegen die Spezifikation zu testen. Der nächste Abschnitt zeigt dennoch eine mögliche sinnvolle Nutzung – allerdings mit Einschränkungen.

Ein Beispiel in C++

Als Beispiel wählen wir den fehlerhaften Bubblesort, der in einem anderen Zusammenhang auf Seite 180 vorkommt. Bei einem Sortierprogramm ist das Orakel kein Problem: Erstens muss die Ergebnisfolge sortiert sein und zweitens muss sie dieselben Elemente wie die Eingangsfolge enthalten, wenn auch ggf. in anderer Reihenfolge. Der fehlerhaften Bubblesort sortiert

korrekt, wenn alle Zahlen verschieden sind, jedoch nicht, wenn beispielsweise die ersten beiden Zahlen einer zu sortierenden Folge erstens gleich und zweitens die größten Zahlen der Folge sind. Auf weitere Einzelheiten muss hier nicht eingegangen werden. Wir kennen also den Fehler – im Unterschied zu dem Praktikanten, der ab Seite 180 den Fehler suchen soll. Die Kenntnis wird hier benutzt, um für den Zufallstest Überlegungen zur Wahrscheinlichkeit anstellen zu können.

Die relative Größe der Werte zueinander bestimmt die Sortierung. Nehmen wir eine Folge mit nur drei Zahlen an, um einen möglichst einfachen Fall zu haben. Bei längeren Folgen kann es weitere Fehlerkonstellationen geben, die wir hier der Einfachheit halber ignorieren. Wenn es sich um drei *ganze* Zahlen handelt, sagen wir 1, 2 und 3, gibt es nur $3^3 = 27$ verschiedene Werte für die Folge, also (1, 1, 1), (1, 1, 2) usw. bis (3, 3, 3). Darunter sind dann auch die drei Fehlerkonstellationen (2, 2, 1), (3, 3, 1) und (3, 3, 2). Wenn die Folgen zufällig aus den drei Zahlen 1, 2 und 3 erzeugt werden, wobei eine Gleichverteilung angenommen wird, ist die Wahrscheinlichkeit 3/27 = 1/9 bzw. etwa 11,11%, dass eine der Fehlerkonstellationen erzeugt wird bzw. 8/9, dass sie nicht erzeugt wird. Mit anderen Worten: Schon bei einer geringen Anzahl N von Versuchen ist die Wahrscheinlichkeit, den Fehler zu finden, recht hoch. Genau gesagt: $(1 - (8/9)^N)$, zum Beispiel 95,8% bei 27 und 99,999% bei 100 Versuchen. Das folgende Programm zeigt einen Zufallstest mit 1000 Versuchen für eine Folge der Länge 3. Beide Parameter können angepasst werden.

```
#include "zufallstest.h"

int main() {
  Zufallstest testobjekt(3, 1000);     // Länge der Folge, Zahl der Versuche
  testobjekt.execute();
}
```

Listing 4.43: Zufallstest *(beispiele/zufallstest/mainZint.cpp)*

Die Klasse `Zufallstest` enthält die in `main()` aufgerufene Funktion `execute()`, die N Folgen erzeugt, das Ergebnis auswertet und eine Zusammenfassung anzeigt. Weil die Attribute, die Auswertung und die zusammenfassende Anzeige auch noch für andere Tests (siehe unten) benötigt werden, sind sie in eine Oberklasse `FastbubblesortTest` ausgelagert. `execute()` ist dabei rein virtuell, muss also von den abgeleiteten Klassen implementiert werden:

```
template<typename T>     // T: Typ eines Elements der Folge
class FastbubblesortTest {
using Container = std::vector<unsigned int>;
public:
  FastbubblesortTest(std::size_t L, std::size_t anz = 0)
```

```
    : laenge(L), folge(L), eingabe(L), referenz(L), N(anz) {
  }

  virtual void execute() = 0;    //von den abgeleiteten Klassen zu
                                 // implementieren

  virtual ~FastbubblesortTest() = default;

  void zusammenfassung() const {
    std::cout << fehlerzahl << " Ergebnisse von " << N
              << " sind falsch, d.h. "
              << 100.0*fehlerzahl/N << "%\n";
  }
  void auswertung() {
    eingabe = folge;  // Eingabewerte sichern
    // korrektes Ergebnis mit Standardfunktion erzeugen:
    referenz = folge;
    std::sort(referenz.begin(), referenz.end());
    // Folge mit zu testender Funktion sortieren und vergleichen:
    fastbubblesort(folge);    // Aufruf der zu prüfenden Funktion
    if(folge != referenz) {
      ++fehlerzahl;
      std::cout << "Eingabewerte: ";   show(eingabe);
      std::cout << "  erwartet: ";     show(referenz);
      std::cout << "  tatsächlich: ";  show(folge);
      std::cout << '\n';
    }
  }
  void show(const Container& cont) {  // Container anzeigen
    for(auto w : cont) {
      std::cout << w << ' ';
    }
  }
protected:
  std::size_t laenge;
  Container folge;
  Container eingabe;
  Container referenz;
  std::size_t fehlerzahl = 0;
  std::size_t N;
};
```

Listing 4.44: Oberklasse (Auszug aus *beispiele/zufallstest/fastbubblesorttest.h*)

Der eigentliche Zufallstest findet in der Methode `execute()` der abgeleiteten Klasse `Zufallstest` statt:

```
#ifndef ZUFALLSTEST_H
#define ZUFALLSTEST_H
#include <random>
#include "fastbubblesorttest.h"

class Zufallstest : public FastbubblesortTest<unsigned int> {
public:
  using FastbubblesortTest::FastbubblesortTest; // Konstruktor erben
  void execute() override {
    std::mt19937 generator;        // einer der C++-Zufallszahlengeneratoren
    std::uniform_int_distribution<unsigned int> verteilung(1, laenge);
    for(std::size_t i = 0u; i < N; ++i) {  // N Folgen untersuchen
      for(auto& wert : folge) {       // Zufallswerte für jede Folge erzeugen
        wert = verteilung(generator);
      }
      auswertung();
    }
    zusammenfassung();
  }
};
#endif
```

Listing 4.45: Klasse `Zufallstest` *(beispiele/zufallstest/zufallstest.h)*

Der `generator` erzeugt die Zufallszahlen über einen sehr großen Bereich. Der Bereich wird mithilfe der `verteilung` eingegrenzt. Es wird eine Gleichverteilung zugrunde gelegt, andere Verteilungen, wie etwa eine Normalverteilung, wären für Problemstellungen anderer Art auch möglich.

> In den Beispielen werden alle Fehlversuche realisiert, um die Wahrscheinlichkeitsaussagen zu verifizieren. In der Praxis wird man Zeit sparen wollen und den Test nach wenigen Fehlschlägen abbrechen, um zunächst die Ursache des Fehlers herauszufinden.

Die Anzeigen des Programms bei mehreren Läufen, ohne die Anzeige der fehlgeschlagenen Versuche, sehen folgendermaßen aus:

```
16 Ergebnisse von 100 sind falsch, d.h. 16%
106 Ergebnisse von 1000 sind falsch, d.h. 10.6%
1090 Ergebnisse von 10000 sind falsch, d.h. 10.9%
11118 Ergebnisse von 100000 sind falsch, d.h. 11.118%
```

Mit zunehmender Zahl der Versuche nähert sich die Zahl fehlgeschlagener Versuche dem theoretischen Wert 11,11%. Bei längeren Folgen sind die Ergebnisse natürlich anders. So ergeben 1000 Läufe bei Folgen mit 100 Elementen 2,9% Fehlschläge. Eine tief gehende Analyse ist hier jedoch nicht sinnvoll, solange schon Folgen der Länge 3 Fehler ergeben.

Die Schwäche des Zufallstests zeigt sich im Vergleich zu einem *systematischen* Vorgehen: Die systematische Erzeugung aller 27 Kombinationen[40] ist genau so einfach, findet aber den Fehler *mit Sicherheit*. Sinnvollerweise werden die Eingabewerte von einem Programm erzeugt. Ein Programm zur Erzeugung aller 27 Kombinationen verursacht ungefähr genau so viel Arbeit wie ein Programm, das die Eingabedaten mithilfe eines Zufallszahlengenerators erzeugt. Die Funktion `execute()` der Klasse `Kombinationstest` testet sämtliche Kombinationen.

```
// Auszug aus beispiele/zufallstest/kombinationstest.h
#include "fastbubblesorttest.h"

class Kombinationstest : public FastbubblesortTest<unsigned int> {
public:
  using FastbubblesortTest::FastbubblesortTest; // Konstruktor erben

  void execute() override {
    N = laenge;     // Anzahl der Kombinationen laenge^laenge berechnen:
    for(std::size_t i = 1u; i < laenge; ++i) {
      assert((N*laenge)/laenge == N);  // Überlaufprüfung
      N *= laenge;                     // Anzahl = laenge^laenge
    }
    // Schleife über alle Kombinationen
    for(std::size_t n = 0u; n < N; ++n) {
      auto zahl = n;
      for(auto& wert : folge) {    // Kombination erzeugen
        wert = 1 + zahl % laenge;
        zahl /= laenge;
      }
      auswertung();
    }
    zusammenfassung();
  }
};
```

Listing 4.46: Klasse `Kombinationstest`

[40] Spezialfall des kombinatorischen Testens, vgl. Abschnitt 4.6.

Die Länge der Folge kann bei der Erzeugung eines Objekts der Klasse `Kombinationstest` angepasst werden.

```
#include "kombinationstest.h"

int main() {
  Kombinationstest testobjekt(3);        // Länge der Folge
  testobjekt.execute();
}
```

Listing 4.47: Test aller Kombinationen *(beispiele/zufallstest/mainK.cpp)*

Bei einer Folge der Länge drei gibt es $3^3 = 27$ Kombinationen. Das Programm hat folgende Ausgabe:

```
Eingabewerte: 2 2 1   erwartet: 1 2 2   tatsächlich: 2 1 2
Eingabewerte: 3 3 1   erwartet: 1 3 3   tatsächlich: 3 1 3
Eingabewerte: 3 3 2   erwartet: 2 3 3   tatsächlich: 3 2 3
3 Ergebnisse von 27 sind falsch, d.h. 11.1111%
```

Längere Folgen ergeben andere Werte. So ergibt der Test von Folgen der Länge vier ($4^4 = 256$ Kombinationen) eine Fehlerrate von 16,4%, und bei einer Länge von fünf ($5^5 = 3125$ Kombinationen) steigt die Fehlerrate auf 18,4%. Das liegt daran, dass bei längeren Folgen noch andere Fehlerkonstellationen möglich sind, die von der obigen Betrachtung nicht erfasst werden. Aber auch hier gilt, dass eine Analyse dieser Fälle nicht sinnvoll ist, solange der Fehler schon bei einer Länge von drei auftritt.

Bei der Untersuchung der Sortierung von `double`-Zahlen wird die Bilanz für den Zufallstest erheblich schlechter. In der internen Zahlendarstellung[41] werden für eine 8 Byte große `double`-Zahl insgesamt 64 Bits verwendet, davon 11 Bits für den Exponenten (inkl. Vorzeichen des Exponenten) und ein Bit für das Vorzeichen der Zahl. Damit bleiben 52 Bits für die Mantisse. Wir grenzen den Bereich der Zufallszahlen von 0,0 bis 1,0 ein, damit wir uns nicht um die Exponenten kümmern müssen. Die Wahrscheinlichkeit, dass ein Zufallszahlengenerator mit guter Gleichverteilung als zweite Zahl dasselbe Bitmuster wie bei der ersten Zahl produziert, liegt bei 2^{-52} oder dezimal etwa bei $0{,}2 \cdot 10^{-16}$. Ganz abgesehen vom Problem, dass diese Zahlen auch noch die größten sein sollen, werden also deutlich mehr als 10^{16} Versuche benötigt, den genannten Fehler zu finden – eine absurd hohe Zahl. Die Wahrscheinlichkeit, den Fehler der genannten Sortierfunktion mit einem Zufallstest aufzudecken, liegt für eine Folge von `double`-Zahlen nahezu bei null.[42]

[41] nach IEEE 754.

[42] Siehe *mainZdouble.cpp* im Verzeichnis *beispiele/zufallstest/*.

Testendekriterium

Abgesehen von anderen bisher diskutierten Testendekriterien spielt die Wahrscheinlichkeit, einen Fehler zu finden, eine Rolle. Sie hängt von der Anzahl N der Tests und der Art der Verteilung der Eingabedaten ab. Genügen 90%, 99% oder 99,9%? Dabei ist immer zu beachten, dass eine Fehlerkonstellation möglicherweise nicht gefunden wird.

Bewertung

In der Regel eignet sich ein Zufallstest nicht für funktionale Tests. Systematische Tests sind vorzuziehen. Auf anderen Gebieten mag der Einsatz berechtigt sein, zum Beispiel um das Belastungsverhalten eines Servers zu prüfen, wie oben schon erwähnt. Auch kann es sinnvoll sein, bei Simulationen das Verhalten eines komplexen Systems mithilfe von Zufallseingaben zu analysieren.

Bezug zu anderen Testverfahren

Zu anderen Testverfahren gibt es keinen Bezug, weil die anderen systematischer Natur sind.

Hinweise für die Praxis

Die Möglichkeit, die Anzahl der Tests mit wenig Aufwand nahezu beliebig zu erhöhen, scheint sehr verlockend. Ein »Nachdenken« über jeden einzelnen Testfall und das erwartete Ergebnis ist beim Zufallstest ja auch nicht erforderlich! Die generierten, zufälligen Werte werden schon für eine breite Abdeckung aller möglichen Eingabewerte sorgen – was ja angenommen werden könnte.

> Vorsicht! Wir haben hoffentlich klarstellen können, dass der Zufallstest kein Allheilmittel ist, sondern dass in jedem Fall überlegt werden soll, ob nicht mit einem systematischen Verfahren weit mehr erreicht werden kann.

5 Strukturbasierte Testverfahren

Alle bisher vorgestellten Testverfahren beruhen auf den Anforderungen an das Testobjekt. Grundlage für die Erstellung der Testfälle sind zum Beispiel Spezifikationen, Zustandsautomaten, Klassifikationsbäume oder auch die Syntaxbeschreibung der Eingaben. Interna des Testobjekts werden dabei nicht genutzt, weshalb diese Ansätze auch als Blackbox-Testverfahren bezeichnet werden. Das Testobjekt ist ein schwarzer Kasten, dessen innere Struktur unbekannt ist.

Im Folgenden werden Ansätze vorgestellt, die die innere Struktur des Testobjekts verwenden, d.h., der Programmcode muss für den Tester zugänglich und ersichtlich sein. Diese Ansätze werden als Whitebox-Testverfahren bezeichnet. Ein weißer Kasten ist aber nicht wirklich durchsichtig.[1] Wir bevorzugen daher den Begriff »strukturbasierte Testverfahren«.

Da die Struktur des Testobjekts – oder genauer der Programmtext – für Überlegungen zur Erstellung von Testfällen genutzt wird, muss detailliertes Wissen über das Programm vorhanden sein. Oft werden die strukturbasierten Testverfahren daher zum Aufgabenbereich des Entwicklers gezählt. Er soll seine Software mit diesen Verfahren prüfen.

Die Testfälle werden auf Grundlage des Programmtextes erstellt, der dann mit diesen Testfällen auf Fehler geprüft wird. Sicherlich nicht der beste Ansatz! Es ist dann wohl ein Test *gegen sich selbst*. So können Anforderungen an das Testobjekt, die nicht berücksichtigt – also vergessen – wurden und somit nicht im Programmtext zu finden sind, mit dem strukturbasierten Ansatz nicht aufgedeckt werden. Auch eine falsche Interpretation und damit Umsetzung von Anforderungen bleibt unerkannt. Die Fehlinterpretation ist ja in den Programmtext eingeflossen, der als Grundlage für die Testüberlegungen dient. Eine Differenz zwischen (fehlerhaftem) Programmtext und den aus dem Programmtext gewonnenen (fehlerhaften) Testfällen liegt nicht vor. Die fehlerhafte Umsetzung kann mit diesem Ansatz nicht aufgedeckt werden.

Auf welcher Basis sollen die erwarteten Ergebnisse der einzelnen Testfälle abgeleitet werden? Dies ist ein weiteres Problem, da der Programmtext

[1] In der Literatur findet sich auch der passendere Begriff »Glassbox«, der sich allerdings nicht durchgesetzt hat.

dazu keine Hilfe bietet. Der tatsächliche Ergebniswert wird vom Programm berechnet, aber woher kommt der erwartete Vergleichswert? Spezifikations- und strukturbasierte Testverfahren sind aus den oben genannten Gründen nicht als gleich wichtig für die Qualitätssicherung anzusehen.

Was bringen aber dann die strukturbasierten Ansätze? Sie bieten eine Kontrollmöglichkeit, inwieweit die spezifikationsbasierten Verfahren eine Auswahl von möglichen Abläufen des Testobjekts zur Ausführung gebracht haben. Tatsächlich fordern verschiedene industrielle Normen, dass die in den folgenden Abschnitten beschriebenen Codeüberdeckungsmaße mit Tests nachgewiesen werden. So fordert etwa die amerikanische Norm DO-178 für den Luftfahrtbereich den Nachweis der modifizierten Bedingungs-/Entscheidungs-Überdeckung (siehe Abschnitt 5.2.3). Entsprechende Dokumente gibt es auch für die Entwicklung von Eisenbahn- und Automobilsoftware. Da dies nicht nur für Tester von Bedeutung ist, sondern auch für Entwickler, gehen wir auf die strukturbasierten Ansätze relativ ausführlich ein.

> Der Hauptgedanke der strukturbasierten Verfahren: Es lässt sich keine Aussage über Teile des Testobjekts treffen, die beim Testen nicht ausgeführt wurden. Auch wenn alle Teile zur Ausführung gekommen sind, ist dies keine Garantie für Fehlerfreiheit. Strukturbasierte Verfahren bieten aber eine Kontrollmöglichkeit, inwieweit die durchgeführten Testfälle die Teile des Testobjekts durchlaufen haben.

Wann ist der Einsatz sinnvoll?

Zuerst werden die für das Testobjekt passenden spezifikationsbasierten Testverfahren ausgewählt und die entsprechenden Testfälle erstellt. Bei der Durchführung dieser Testfälle ist durch ein geeignetes Werkzeug zu ermitteln, welche Teile des Testobjekts tatsächlich ausgeführt wurden. Zu untersuchen sind die Teile, die bisher nicht zur Ausführung gekommen sind. Es ist dann zu überlegen, wie und mit welchen Testdaten diese verbleibenden Teile ausgeführt werden können, um auch für diese zu ermitteln, ob sie korrekt oder fehlerhaft ablaufen.

Grundidee

Bei jedem Testlauf kommen im Testobjekt bestimmte Teile des Programmtextes zur Ausführung. Besteht das Testobjekt aus einer Sequenz von Anweisungen ohne Entscheidungen und Schleifen, werden alle Anweisungen durch einem Testfall ausgeführt. Ein Ziel der strukturbasierten Testent-

wurfsverfahren, die Ausführung aller Anweisungen, ist damit bereits erreicht.

Enthält das Testobjekt als Kontrollstruktur eine *If*-Abfrage, so kommt je nach Auswertung der Entscheidung der *Then*- oder *Else*-Teil zur Ausführung. Um hier beide Möglichkeiten zu prüfen, sind mindestens zwei Testfälle erforderlich. Ein Testfall muss Eingabedaten enthalten, die bei der Auswertung der Entscheidung der *If*-Abfrage `true` ergeben, damit der *Then*-Teil ausgeführt wird. Bei einem weiteren Testfall müssen die Eingabewerte so gewählt werden, dass die Entscheidung zu `false` ausgewertet wird und somit der *Else*-Teil zur Ausführung kommt. Dabei spielt es keine Rolle, ob der *Else*-Teil Anweisungen enthält oder leer ist. Ausschlaggebend ist, dass die Entscheidung – oder genauer der Entscheidungsausgang – bei der Durchführung der Testfälle beide Wahrheitswerte annimmt.

Nehmen wir an, dass unser Testobjekt drei voneinander unabhängige *If*-Abfragen enthält. Dann können die Eingaben so gewählt werden, dass jede der drei Entscheidungen zu `true` ausgewertet wird, also die drei *Then*-Teile mit einem Testfall durchlaufen werden. Bei passender Wahl des zweiten Testfalls liefert jede der drei Entscheidungen `false` und alle *Else*-Teile werden geprüft.

Soll nun jede mögliche Kombination getestet werden, sind $2^3 = 8$ Testfälle erforderlich, da dann alle `true`/`false`-Kombinationen geprüft werden. Es sind acht unterschiedliche Pfade durch das Testobjekt möglich. Wie an diesem Beispiel deutlich wird, ist dies eine umfassendere Forderung zur Prüfung des Testobjekts auf der Grundlage des Programmtextes.

Enthält das Testobjekt Schleifen, so bewirkt jede unterschiedliche Anzahl von Schleifendurchläufen ein unterschiedliches Programmverhalten und somit einen unterschiedlichen Pfad durch das Testobjekt. Wird der Schleifenkörper einer Schleife zwei-, drei- oder zwanzigmal wiederholt, entsteht jeweils ein neuer Ablauf des Testobjekts, ein neuer Pfad und – in aller Regel – auch ein neues Ergebnis. Die Anzahl von Schleifendurchläufen, mit Ausnahme von Zählschleifen, ist nahezu unbegrenzt und von den gewählten Eingaben abhängig.

Auf die eben kurz skizzierten Ansätze wird in den folgenden Abschnitten näher eingegangen.

5.1 Kontrollflussbasierter Test

Die oben erwähnten Ansätze beruhen alle auf dem Kontrollfluss durch das Testobjekt. Der Kontrollfluss wird durch Abfragen und Schleifen – genauer die Auswertung der Entscheidungen der Abfragen und der Schleifenbedingungen – gesteuert.

Der Kontrollfluss[2] eines Testobjekts kann als Kontrollflussgraph dargestellt werden. Der Graph ist ein gerichteter Graph, der aus Knoten und Kanten sowie aus einem Start- und Endknoten besteht. Die Knoten repräsentieren die Anweisungen bzw. eine Sequenz von Anweisungen des Testobjekts. Die Kanten stellen den Kontrollfluss dar. Ein Knoten mit zwei ausgehenden Kanten, eine Verzweigung, repräsentiert eine *If*-Abfrage mit dem *Then*- und dem *Else*-Teil. Schleifen enthalten rückführende Kanten zum Schleifenanfang. Ein ausgeführter Testfall ist im Graphen eine Folge von Knoten und Kanten und wird als Pfad bezeichnet. Die nachfolgenden Ansätze zur Überdeckung des Testobjekts werden häufig auf Grundlage des Kontrollflussgraphen definiert. Wir beschränken uns auf den Programmtext und erörtern die Vorgehensweisen nicht am Graphen.

Bei allen Ansätzen kann ein Grad der Überdeckung bei der Ausführung der Testfälle ermittelt werden. Vorab kann festgelegt werden, wie hoch der Grad der Überdeckung sein soll, der bei der Durchführung der Testfälle zu erreichen ist, z.B. können 80% statt 100% als ausreichend angesehen werden. Nach Ausführung der Testfälle ist dann der erreichte Überdeckungsgrad auszuweisen. Hierzu wird in aller Regel ein Werkzeug genutzt, das die Überdeckung ermittelt. Liegt der erreichte Überdeckungsgrad unterhalb der zu erzielenden Überdeckung, sind weitere Testfälle zu spezifizieren und auszuführen. Durch eine Analyse des Programmtextes – oder genauer der Teile, die bisher nicht ausgeführt wurden – ergeben sich weitere Testfälle. Sehen wir uns nun zunächst eine Werkzeugunterstützung näher an.

5.1.1 Werkzeugunterstützung

Um zu erkennen, welche Anweisungen wie oft durchlaufen werden, wird der Programmcode durch entsprechende Zähler ergänzt. Änderungen des Programmcodes für Messzwecke werden »Instrumentierung« genannt und lassen sich am besten von einem Werkzeug durchführen.

Open-Source-Werkzeug gcov mit lcov und genhtml

Wir haben im Folgenden die Open-Source-Werkzeuge *lcov*[3] und *genhtml* zur Feststellung der Zeilenüberdeckung[4] verwendet. *lcov* ist ein Frontend für *gcov*, das allerdings nur mit dem g++-Compiler zusammenarbeitet. Die von

[2] Neben dem Kontrollfluss kann auch der Datenfluss im Testobjekt analysiert und für die Ermittlung von Testfällen herangezogen werden. Auf diese Möglichkeit wird hier nicht weiter eingegangen.

[3] *http://ltp.sourceforge.net/coverage/lcov.php*

[4] Die Werkzeuge ermitteln, ob eine Zeile (nicht immer gleichzusetzen mit einer Anweisung) ausgeführt wurde.

lcov erzeugten Informationen werden mit *genhtml* in html-Dateien umgeformt. Die Verarbeitungsschritte sind:

1. Übersetzen und linken des Testprogramms. Dabei ist die Optimierung mit `-O0` auszuschalten. Die zu übergebenden Optionen sind `-fprofile-arcs` und `-ftest-coverage`.
2. Testprogramm mit den Testfällen ausführen. Dabei werden Dateien mit der Endung *.gcda* erzeugt. Vorher existierende *.gcda*-Dateien müssen gelöscht werden!
3. `lcov -d . --no-external -c -o ergebnis.info`
   erzeugt die Informationen, die für die Darstellung mit dem Browser benötigt werden. `-d` steht für directory, der Punkt ist das aktuelle Verzeichnis. `--no-external` sorgt dafür, dass Systemdateien ausgeklammert werden.
4. `genhtml -o html ergebnis.info` erzeugt das Verzeichnis *html*.
5. Anklicken der Datei *html/index.html* zeigt das Ergebnis im Browser.

In unseren herunterladbaren Beispielen wird CTC++ mit Google Test kombiniert. Es genügt, `make clean` vor jedem neuen Test und dann `make run` einzugeben, sodass nur noch der letzte Schritt, die Anzeige mit dem Browser, ausgeführt werden muss.

Testwell CTC++

Testwell CTC++ ist ein kommerzielles Werkzeug[5] und erheblich leistungsfähiger als *gcov*, wenn mehr als nur die Zeilenüberdeckung gefragt ist. Für das Werkzeug ist eine kostenlose befristete Evaluierungslizenz erhältlich. Sie wurde uns freundlicherweise von der Firma Verifysoft Technology GmbH[6] zur Verfügung gestellt. Die Bedienung des Tools ist ebenso einfach wie die von *lcov*. Sie wird in der ausführlichen Benutzungsdokumentation erläutert.

In unseren herunterladbaren Beispielen genügt es, `make run` einzugeben, sodass ebenfalls nur noch der letzte Schritt – Anzeige im Browser – ausgeführt werden muss.

5.1.2 Anweisungstest

Über Anweisungen, die während des Tests nicht zur Ausführung gekommen sind, lässt sich keinerlei Aussage treffen – eine einsichtige und leicht nachvollziehbare Aussage. Beim Anweisungstest (statement testing)[7] ist also anzustreben, dass alle Anweisungen des Testobjekts mindestens einmal

[5] Die Verwendung von CTC++ für die Zwecke dieses Buchs ist keine Aussage über die Qualitäten und Fähigkeiten anderer kommerzieller Werkzeuge.

[6] *http://www.verifysoft.com/de.html*

[7] Wir führen die englischen Begriffe jeweils mit auf, da diese auch häufig verwendet werden.

ausgeführt werden. Wenn das Testobjekt *If*-Abfragen mit Anweisungen im *Then*- und *Else*-Teil enthält, sind mindestens zwei Testfälle erforderlich. Das Maß für die Überdeckung der Anweisungen (statement coverage) errechnet sich folgendermaßen:

Anweisungsüberdeckung = S/A mit
S = Anzahl der ausgeführten Anweisungen
A = Anzahl aller Anweisungen des Testobjekts

Ein Überdeckungsgrad von 1.0, also 100% Überdeckung, bedeutet, dass jede Anweisung des Testobjekts mindestens in einem Testfall ausgeführt wurde. Wie häufig eine Anweisung bei der Durchführung aller Testfälle ausgeführt wurde, spielt bei der Überdeckung keine Rolle.[8] 100% Überdeckung ist anzustreben. Wird dieser Wert bei den bisher durchgeführten Testfällen nicht erreicht, ist zu prüfen, woran es liegt. Die bisher nicht ausgeführten Anweisungen des Testobjekts sind genauer zu untersuchen. Es gibt zwei mögliche Ursachen:

1. Es fehlen Testfälle, die diese Anweisung(en) zur Ausführung bringen.
2. Es ist nicht möglich, diese Anweisung(en) zur Ausführung zu bringen.

Im ersten Fall ist zu überlegen, mit welchen Eingabedaten die bisher noch nicht ausgeführten Anweisungen zur Ausführung gebracht werden können. Diese Aufgabe, an die betreffende Stelle im Testobjekt mit den geforderten Werten interner Variablen zu gelangen, kann sich als recht schwierig erweisen.

Zu den nicht ausgeführten Programmteilen gehört oft das Abfangen von Fehlersituationen (exceptions), da es meist sehr aufwendig ist, eine solche Fehlersituation herzustellen bzw. zu simulieren. Wenn dieser Aufwand unverhältnismäßig hoch ist, kann auf die Ausführung dieser Programmteile verzichtet werden (»Lean Testing«). Ein Codereview zur Kontrolle dieser nicht ausgeführten Anweisungen ist dann eine sinnvolle qualitätssichernde Maßnahme.

Im zweiten Fall können die Anweisungen nicht erreicht werden, weil zum Beispiel verschachtelte Abfragen die Ausführung nicht ermöglichen. Nehmen wir ein einfaches Beispiel: Eine *If*-Abfrage prüft, ob der Wert einer Variablen größer als 100 ist. Im *Then*-Teil wird diese Variable erneut abgefragt, ob ihr Wert größer als 50 ist. Der *Else*-Teil der zweiten verschachtelten Abfrage kann nie erreicht werden, da ein Wert nicht größer als 100 und kleiner oder gleich 50 sein kann (siehe Listing 5.1).

```
if(x > 100) {
  if(x > 50) {
```

[8] Diese Anmerkung gilt auch für alle weiteren Überdeckungsmaße.

```
    // Anweisungen ...
  }
  else {
    y = 1;   // unerreichbare Anweisung
  }
}
```

Listing 5.1: Unerreichbare Anweisungen

Diese unerreichbaren Anweisungen werden auch als »dead code« bezeichnet. Sie deuten auf fehlerhafte Programmierung hin. Ursache können aber auch Missverständnisse oder Denkfehler bei der Umsetzung der Anforderungen sein. Durch »copy and paste« von Programmteilen kann ebenfalls unerreichbarer Programmcode entstehen. Nicht immer führt »dead code« auch zu einem fehlerhaften Programmverhalten, aber eine genauere Untersuchung der unerreichbaren Programmstellen ist in jedem Fall sinnvoll.

C++-Beispiel

Aus Abschnitt 4.8.3 kennen Sie die zustandsbasierten Testfälle zur Prüfung der Methode `uebergang()`. Uns interessiert nun, welche Teile des Testobjekts bei der Durchführung der Tests zur Ausführung gekommen sind und ob es Programmteile gibt, die durch diese Tests noch nicht ausgeführt wurden. Parallel zur Durchführung der Tests wurde die Zeilenüberdeckung gemessen.[9] Abbildung 5-1 zeigt eine vom Browser angezeigte Übersicht des Ergebnisses. Die Überdeckung der Zeilen zeigt auch die Überdeckung der Anweisungen, wenn nicht mehr als eine Anweisung pro Zeile vorkommt. In diesem Fall entsprechen 100% Zeilenüberdeckung auch 100% Anweisungsüberdeckung.

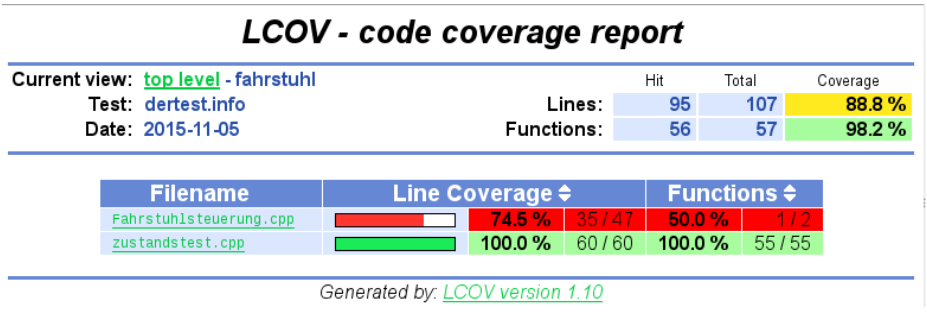
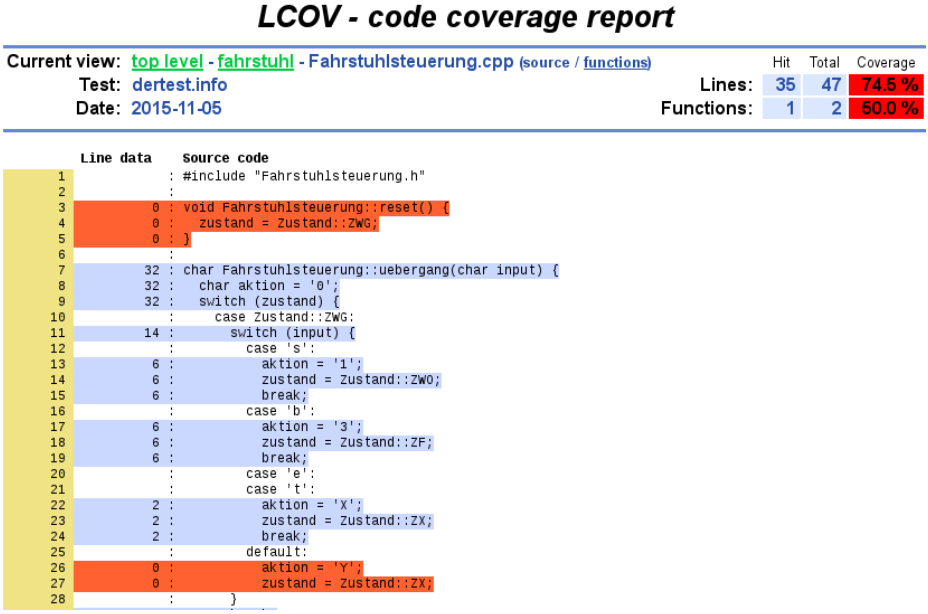


Abbildung 5-1: Zeilenüberdeckung des Zustandstests »Fahrstuhlsteuerung«

In der Übersicht findet man rechts oben eine Zusammenfassung der Überdeckung. Wird in der Darstellung links auf *fahrstuhlsteuerung.cpp* geklickt,

[9] Siehe Verzeichnis *beispiele/gcov/fahrstuhl*.

wird der Quellcode mit den Einzelheiten angezeigt – und nur diesen sehen wir uns näher an (in Abbildung 5-2 ist nur ein Ausschnitt aus dem Report dargestellt).



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - fahrstuhl - Fahrstuhlsteuerung.cpp (source / functions)	Lines:	35	47	74.5 %
Test:	dertest.info	Functions:	1	2	50.0 %
Date:	2015-11-05				

	Line data	Source code
1	:	#include "Fahrstuhlsteuerung.h"
2	:	
3	0 :	void Fahrstuhlsteuerung::reset() {
4	0 :	zustand = Zustand::ZWG;
5	0 :	}
6	:	
7	32 :	char Fahrstuhlsteuerung::uebergang(char input) {
8	32 :	char aktion = '0';
9	32 :	switch (zustand) {
10	:	case Zustand::ZWG:
11	14 :	switch (input) {
12	:	case 's':
13	6 :	aktion = '1';
14	6 :	zustand = Zustand::ZW0;
15	6 :	break;
16	:	case 'b':
17	6 :	aktion = '3';
18	6 :	zustand = Zustand::ZF;
19	6 :	break;
20	:	case 'e':
21	:	case 't':
22	2 :	aktion = 'X';
23	2 :	zustand = Zustand::ZX;
24	2 :	break;
25	:	default:
26	0 :	aktion = 'Y';
27	0 :	zustand = Zustand::ZX;
28	:	}

Abbildung 5-2: Zeilenüberdeckung (Ausschnitt aus dem Report)

Die zweite Spalte gibt an, wie oft die betreffende Zeile ausgeführt wurde. Die Zeilen 3 bis 5 sowie die Zeilen 26 und 27 wurden nicht angesprochen, weil die Funktion `reset()` nicht aufgerufen wurde und keine unspezifizierten Eingabesymbole im Test auftraten. Die bei der Durchführung der Testfälle erreichte Zeilenüberdeckung (line coverage) beträgt 74,5% für *fahrstuhlsteuerung.cpp*. Unser Zustandstest hat somit nicht alle Teile des Testobjekts geprüft. Es gibt Anweisungen, die noch nicht zur Ausführung gekommen sind.

Außer der Funktion `uebergang()` gibt es eine zweite Funktion `reset()` in den Zeilen 3 bis 5 – diese wurde bisher nicht ausgeführt. Das Werkzeug zählt die Funktionen des Testobjekts und errechnet eine Funktionsüberdeckung, also welcher Anteil aller im Testobjekt vorkommenden Funktionen bei den Tests aufgerufen wurde. Dabei wird aber nicht geprüft, welche Teile innerhalb der Funktion zur Ausführung kommen. Von den zwei im Testobjekt vorhandenen Funktionen wurde bisher nur eine aufgerufen; die Funktionsüberdeckung ist damit 50%.

5.1.3 Entscheidungstest

Der Anweisungstest ist eine schwache Anforderung an die zu erreichende Codeüberdeckung. So bleiben leere *Else*-Teile von *If*-Abfragen unberücksichtigt, da sie keine Anweisungen enthalten. Die *If*-Abfrage muss beim Test nicht den Wert `false` liefern. Dies ist kein ausreichender Test. Beim Entscheidungstest[10] (decision testing) wird daher gefordert, dass jede im Testobjekt vorhandene Entscheidung mindestens einmal zu `true` und einmal zu `false` ausgewertet wird. Damit kommen auch leere *Else*-Teile zur »Ausführung«. Genauer betrachtet geht es um die beiden *Entscheidungsausgänge* (`true`, `false`), die beim Test zu überdecken sind.

Der Entscheidungstest ist damit umfassender als der Anweisungstest, er schließt den Anweisungstest mit ein. Das Maß für die Überdeckung der Entscheidungsausgänge (decision coverage) errechnet sich analog zur Überdeckung der Anweisungen:

Entscheidungsüberdeckung = E/A mit
E = Anzahl der ausgeführten Entscheidungsausgänge
A = Anzahl aller Entscheidungsausgänge des Testobjekts

Oft wird der Entscheidungstest als ein Kriterium zur Beendigung der Testaktivitäten angesehen, wenn jede Entscheidung beide Wahrheitswerte angenommen hat und somit alle Entscheidungsausgänge getestet wurden.

Sehen wir uns hierzu ein triviales Beispiel an: Aus drei Parametern soll das Maximum ermittelt werden (siehe Listing 5.2[11]).

```
int max (int x, int y, int z) {
  int max=0;
  if (x > z)
    max = x;
  if (y > x)
    max = y;
  if (z > y)
    max = z;
  return max;
}
```

Listing 5.2: Maximum von drei Zahlen finden

Sicherlich sehen Sie bereits, dass dieses Programmstück seine Tücken hat. Nehmen wir nun weiter an, wir entscheiden uns dafür, den Test mit drei

[10] Wenn das Verfahren auf dem Kontrollflussgraphen basiert, wird es als Zweigtest bezeichnet. Die Unterschiede werden kurz in Abschnitt 7.1.2 auf Seite 201 beschrieben.

[11] Quelle: Stephan Kleuker: Qualitätssicherung durch Softwaretests, Springer Vieweg, Wiesbaden, 2013.

Testfällen durchzuführen. Bei jedem Testfall wird der zu ermittelnde Maximalwert durch einen anderen Parameter übergeben. Dies scheint eine sinnvolle Wahl von Testfällen zu sein, wenn das Maximum ermittelt werden soll.

```
// Auszug aus beispiele/max3/max3.cpp
TEST(Maximum_dreier_Zahlen, TF1) {
  EXPECT_EQ(7, max(7, 5, 4));
}
TEST(Maximum_dreier_Zahlen, TF2) {
  EXPECT_EQ(7, max(5, 7, 4));
}
TEST(Maximum_dreier_Zahlen, TF3) {
  EXPECT_EQ(7, max(4, 5, 7));
}
```

Listing 5.3: Drei Testfälle

Parallel zur Ausführung der drei Testfälle haben wir mit CTC++ die erreichte Überdeckung gemessen.[12] Das Werkzeug liefert textuelle Ausgaben, die in eine HTML-Darstellung umgewandelt werden können – nur diese zeigen wir hier:

CTC++ Coverage Report - Execution Profile #1/2

Directory Summary | Files Summary | Functions Summary | Untested Code | Execution
To files: First | Previous | Next | Last | Index | No Index

Source file: ./max3.cpp
Instrumentation mode: multicondition **Reduced to:** decision coverage
TER: 100 % (8/8) structural, 100 % (8/8) statement

Hits/True	False	Line	Source
		1	#include "max3.h"
		2	
Top			
3		3	int max(int x, int y, int z) {
		4	int max = 0;
2	1	5	if (x > z)
		6	max = x;
2	1	7	if (y > x)
		8	max = y;
1	2	9	if (z > y)
		10	max = z;
3		11	return max;
		12	}
		13	

Abbildung 5-3: Entscheidungsüberdeckung des Tests »Maximum dreier Zahlen«

[12] Siehe Verzeichnis *beispiele/ctc/max3*.

Die Abkürzung TER steht für *test effectiveness ratio*, ein Testmaß des Werkzeugherstellers. Es wird in Prozent angegeben. »statement« meint Anweisungsüberdeckung. Wenn 8/8 (also 8 von 8) Anweisungen ausgeführt werden, ergibt das ein TER von 100%. Das Werkzeug liefert die Anzahl, wie oft eine Anweisung bei der Durchführung der Testfälle ausgeführt wurde. Eine Anweisung ist durch ein abschließendes Semikolon gekennzeichnet. Kontrollanweisungen ohne abschließendes Semikolon, wie die Auswertung einer `if`-Bedingung, zählen ebenfalls als Anweisung. »structural« ist die Überdeckung basierend auf der Programmstruktur. Darunter fallen je nach Einstellung Funktionen, Entscheidungen und Mehrfachbedingungen. Bei drei `if`-Abfragen mit je einem *Then*- und je einem *Else*-Zweig müsste sich eigentlich 6/6 für die Entscheidungsüberdeckung ergeben. Es werden aber der Eingang in die Funktion und der Ausgang mitgezählt, sodass 8/8 in Abbildung 5-3 angegeben ist. Die Abbildung zeigt, welcher Entscheidungsausgang bei einer *If*-Abfrage wie oft zur Ausführung gekommen ist. In Zeile 5 ist die *If*-Abfrage 3-mal ausgeführt worden. Dabei ergab sich 2-mal der Wert `true` und einmal der Wert `false`. Für alle drei Abfragen trifft zu, dass sowohl der *Then*-Zweig als auch der (leere) *Else*-Zweig durchlaufen wurden. Damit haben alle Entscheidungsausgänge die beiden möglichen booleschen Werte angenommen, es liegt mit den drei durchgeführten Testfällen eine 100%ige Entscheidungsüberdeckung vor.

Wir sollten doch jetzt ein gutes Gefühl haben und den Test als ausreichend ansehen und beenden können, oder? Aber: Wie sieht es mit den folgenden zwei Testfällen aus?

```
TEST(Maximum_dreier_Zahlen, TF4) {
  EXPECT_EQ(7, max(7, 4, 5));
}
TEST(Maximum_dreier_Zahlen, TF5) {
  EXPECT_EQ(7, max(7, 7, 7));
}
```

Listing 5.4: Testfälle, die zu Fehlern führen

Der Aufruf der Funktion `max()` in Testfall TF4 liefert 5 als Ergebnis, der Aufruf in TF5 liefert 0, wie der Auszug des Testprotokolls zeigt:

```
[ RUN      ] Maximum_dreier_Zahlen.TF4
max3_test.cpp:15: Failure
Value of: max(7, 4, 5)
  Actual: 5
Expected: 7
[  FAILED  ] Maximum_dreier_Zahlen.TF4 (0 ms)
[ RUN      ] Maximum_dreier_Zahlen.TF5
max3_test.cpp:18: Failure
```

```
Value of: max(7, 7, 7)
  Actual: 0
Expected: 7
[  FAILED  ] Maximum_dreier_Zahlen.TF5 (0 ms)
```

Beide Ergebnisse sind falsch. Das sieht nicht gut aus! Fehler können durch unterschiedliche Kombinationen der drei Entscheidungen auftreten. In Testfall TF5 liefern alle drei Entscheidungsausgänge `false`. Die Variable `max` wird an keiner Stelle geändert und der initiale Wert (0) wird somit als Maximum ausgegeben.

Wie kann nun ein systematischer Ansatz zum Testen aussehen? Wir können die Überlegungen zur Äquivalenzklassenbildung[13] nutzen, um den Eingabebereich (drei Zahlen) zu strukturieren. Wir bleiben bei unseren Zahlenwerten (4,5,7) mit 7 als Maximalwert, der bei jedem Testfall vorkommt, um den Test noch »lean« zu gestalten. Ebenso bleiben wir bei der Vermutung, die auch zu unseren ersten drei Testfällen führte, dass die Position des Parameters des gesuchten Maximums von Bedeutung ist. Die mögliche Strukturierung der Testfälle ist:

- Alle Zahlen sind gleich, ein möglicher Repräsentant ist 7: 7,7,7
- Zwei Zahlen sind gleich und die dritte ist größer: 7,5,5; 5,7,5; 5,5,7[14]
- Zwei Zahlen sind gleich und die dritte ist kleiner: 7,7,5; 7,5,7; 5,7,7
- Alle Zahlen sind ungleich: 7,5,4; 7,4,5; 5,7,4; 4,7,5; 4,5,7; 5,4,7

Hieraus ergeben sich unter Berücksichtigung der angegebenen Permutationen insgesamt 13 Testfälle. Die beiden fehleraufdeckenden Testfälle (7,4,5 und 7,7,7) werden bei diesem systematischen Vorgehen mit berücksichtigt. Eine hohe Codeüberdeckung – erreicht mit den ersten drei oben angegebenen Testfällen – ist also kein Garant dafür, dass ausreichend getestet wurde.

> 100% Entscheidungsüberdeckung garantiert keine Fehlerfreiheit.

[13] Die Äquivalenzklassenbildung bezieht sich auf einzelne Eingabebereiche. Hier analysieren wir die drei Eingabeparameter und teilen die möglichen Konstellationen der Parameter sozusagen übergreifend in Klassen ein. Im Gegensatz zum Äquivalenzklassenansatz werden keine Datenbereiche angegeben, sondern direkt unterschiedliche Repräsentanten.

[14] Statt 5 kann auch 4 als Testdatum ausgewählt werden, dies hat aber vermutlich keinen Einfluss auf das Testergebnis.

5.1.4 Pfadtest

Eine mögliche Folge von Anweisungen des Testobjekts wird als Pfad[15] bezeichnet. Jeder Unterschied in der Abfolge von Anweisungen führt zu einem neuen Pfad. Der Pfadtest (path testing) ist eher ein theoretisches Maß, da eine 100%ige Pfadüberdeckung (path coverage) nur bei trivialen (meist rein sequenziellen) Programmen erreichbar ist.

Greifen wir das obige Beispiel der Ermittlung des Maximalwertes mit den drei Abfragen im Programmtext erneut auf. Jede Abfrage hat zwei Entscheidungsausgänge, insgesamt ergeben sich bei drei Abfragen damit $2^3 = 8$ mögliche Pfade. Demnach müssen beim Pfadtest acht Testfälle spezifiziert werden, wobei jeder Testfall genau einen der acht Pfade zur Ausführung bringt. Bei einem Pfad liefern somit alle drei Entscheidungsausgänge `true`: x > z und y > x und z > y. Es lassen sich keine Werte für x, y und z finden, die diese drei Ungleichungen gleichzeitig erfüllen. Dieser Pfad ist nicht ausführbar.

Die restlichen sieben unterschiedlichen Pfade durch das Testobjekt ergeben sich wie folgt: Alle Entscheidungen liefern den Wert `false` (Testdaten: 7,7,7). Die weiteren sechs Pfade ergeben sich durch die Kombinationen von ungleichen Zahlen: 7,5,4; 7,4,5; 5,7,4; 4,7,5; 4,5,7; 5,4,7. Damit sind die sieben ausführbaren Pfade durch das Testobjekt überdeckt.

Die Testüberlegung, dass zwei Zahlen gleich sind – egal ob größer oder kleiner als die maximale Zahl –, die wir oben getroffen haben, ist beim Pfadtest für dieses Beispiel ohne Bedeutung. Die drei Abfragen prüfen auf *größer*, und bei Gleichheit der Werte ergibt sich als Entscheidungsausgang `false`, derselbe Wert, wie wenn die erste Zahl kleiner als die zweite ist. Für den Pfadtest ergeben sich bei Prüfung auf Gleichheit somit keine zusätzlichen Pfade. Die Prüfung auf Gleichheit wird vom Pfadtest daher nicht erzwungen, eine höhere Überdeckung wird mit diesen zusätzlichen Tests nicht erreicht. Das heißt aber nicht, dass der Test auf zwei gleiche Zahlen nicht sinnvoll und daher nicht durchzuführen ist – nur für den Pfadtest bringt es keine zusätzliche Einsicht. Das ist eine ganz grundsätzliche Erkenntnis:

> Jedes Verfahren fokussiert auf einen bestimmten Aspekt, den es prüft. Nur die Kombination unterschiedlicher Verfahren ergibt ausreichende Tests.

Im Beispiel wurde deutlich, dass sobald Abhängigkeiten zwischen den einzelnen Entscheidungen im Testobjekt vorhanden sind, eine vollständige Kombi-

[15] Beim Kontrollflussgraphen wird eine beliebige Folge verbundener Knoten und Kanten als Pfad bezeichnet. Diese beginnt beim Start- und endet beim Endknoten.

nation aller Entscheidungsausgänge nicht garantiert werden kann, sondern im Einzelfall nachgewiesen werden muss.

Nicht ausführbare Pfade sind aber nicht der Hauptgrund, weshalb der Pfadtest keine sinnvolle Vorgehensweise für die Praxis bietet. Jede unterschiedliche Anzahl von Schleifendurchläufen führt zu einer unterschiedlichen Abfolge von Anweisungen und somit zu einem unterschiedlichen Pfad. Sobald Schleifen mit variablen Abbruchbedingungen im Testobjekt vorkommen, lässt sich die Anzahl aller Pfade des Testobjekts nicht ermitteln und damit kann kein Grad der Pfadüberdeckung bestimmt werden.

Ein Ansatz, das Problem in den Griff zu bekommen, ist der sogenannte strukturierte Pfadtest. Die Länge der Pfade – insbesondere die Anzahl von Schleifenwiederholungen – wird beschränkt und nur Pfade mit maximal dieser Länge werden beim Test berücksichtigt. Ein weiterer Ansatz, um Schleifen zu testen, wird im Folgenden vorgestellt.

5.1.5 Schleifentest

Um die meist beliebige Anzahl von Wiederholungen bei Schleifen[16] für den Test einzuschränken, kann der sogenannte Boundary-Interior-Test[17] verwendet werden. Jede Schleife im Testobjekt wird einzeln betrachtet. Es werden dabei drei Ausführungen der Schleife unterschieden:

1. Der Schleifenkörper wird nicht ausgeführt[18].
2. Der Schleifenkörper wird genau einmal ausgeführt (boundary).
3. Der Schleifenkörper wird (mindestens) zweimal ausgeführt (interior).

In allen drei Fällen wird davon ausgegangen, dass eine spezielle fehlerhafte Programmierung der Schleife erkannt wird. Im 1. Fall kann es zu Fehlern kommen, wenn der Schleifenkörper notwendige Berechnungen oder Zuweisungen enthält, die außerhalb der Schleife erforderlich sind, und dann bei der Umgehung der Schleife fehlen und somit zu Fehlern führen.

Bei einer einmaligen Ausführung kann der Abbruch zu früh erfolgt sein, und es wird statt mit einem Endergebnis mit einem Zwischenergebnis weiter gearbeitet.

Die dritte Überprüfung ist wichtig, da es vorkommen kann, dass eine eigentlich einmalige Initialisierung (vor Betreten der Schleife) versehentlich bei jedem Schleifendurchlauf erfolgt. Diese Art von Fehlern kommt nur bei einer Wiederholung des Schleifenkörpers zum Tragen.

[16] Zählschleifen, bei denen die Anzahl der Durchläufe festliegt, sind davon ausgenommen.

[17] Eine gebräuchliche Übersetzung ins Deutsche existiert leider nicht.

[18] Nur bei abweisenden Schleifen möglich.

Beispiel in C++

Als Beispiel dient eine Funktion zum Zählen der in einem String enthaltenen Vokale. Der Einfachheit halber wird auf die Erkennung von Umlauten, die einen anderen Zeichensatz als ASCII voraussetzt, verzichtet. Um die Anzahl der Abfragen zu reduzieren, wird jedes Zeichen gegebenenfalls in einen Kleinbuchstaben umgewandelt.

```
// Auszug aus beispiele/gcov/vokale/vokale.cpp
// Vokale zählen (ohne Umlaute)
std::size_t vokale(const std::string& str) {
  std::size_t z = 0;
  for(auto c : str) {
    if(isupper(c)) {
      c = std::tolower(c);    // Umwandlung in Kleinbuchstaben
    }
    if(   c == 'a'
       || c == 'e'
       || c == 'i'
       || c == 'o'
       || c == 'u') {
      ++z;
    }
  }
  return z;
}
```

Listing 5.5: Vokale zählen

Die drei Teile des Boundary-Interior-Tests werden entsprechend durchnummeriert. Im ersten Test wird ein leerer String übergeben, sodass die Schleife nicht ausgeführt wird.

```
// Auszug aus beispiele/gcov/vokale/vokale_test.cpp
TEST(Vokale_zaehlen, Boundary_Interior_Test_0) {
  EXPECT_EQ(0, vokale(""));
}
```

Listing 5.6: Boundary-Interior-Test 0

Da wir bei jedem der Testfälle überprüfen müssen, auf welche Art und Weise die Schleife zur Ausführung gekommen ist, ist es erforderlich, die Überdeckung pro Testfall zu analysieren. Abbildung 5-4 zeigt die erreichte Überdeckung beim ersten Testfall. Der Wert 0 in der zweiten Spalte gibt an, dass der Schleifenkörper nicht ausgeführt wird.

Der String im nächsten Test enthält nur ein Zeichen, sodass die Schleife genau einmal ausgeführt wird:

Current view:	top level - vokale - vokale.cpp (source / functions)		Hit	Total	Coverage
Test:	dertest.info	Lines:	4	12	33.3 %
Date:	2015-11-06	Functions:	1	1	100.0 %

```
      Line data    Source code
    1            : #include "vokale.h"
    2            : #include <cstring>
    3            :
    4            : // Vokale zählen (ohne Umlaute)
    5          1 : std::size_t vokale(const std::string& str) {
    6          1 :   std::size_t z = 0;
    7          1 :   for(auto c : str) {
    8          0 :     if(isupper(c)) {
    9          0 :       c = std::tolower(c);   // Umwandlung in Kleinbuchstaben
   10            :     }
   11          0 :     if(c == 'a'
   12          0 :        || c == 'e'
   13          0 :        || c == 'i'
   14          0 :        || c == 'o'
   15          0 :        || c == 'u') {
   16          0 :       ++z;
   17            :     }
   18            :   }
   19          1 :   return z;
   20            : }
   21            :
```

Abbildung 5-4: Schleifenkörper wird nicht durchlaufen.

```
TEST(Vokale_zaehlen, Boundary_Interior_Test_1) {
  EXPECT_EQ(1, vokale("A"));
}
```

Listing 5.7: Boundary-Interior-Test 1

Die Abbildung 5-5 zeigt die gemessene Überdeckung für diesen einen Testfall. Der Schleifenkörper (Zeilen 8 bis 18) wird nur einmal ausgeführt, die Schleifenentscheidung (Zeile 7) wurde zweimal ausgeführt und ergab einmal den Wert `true` und beim zweiten Erreichen den Wert `false`, abzulesen in Zeile 8, die als erste Zeile in der Schleife nur einmal ausgeführt wurde. Unser Ziel, die Schleife nur einmal auszuführen, ist erfüllt.

Anmerkung: Die Abfrage in Zeile 11 besteht aus mehreren Bedingungen, die alle mit einem logischen *oder* verknüpft sind. Um zu erkennen, welche Bedingungen zur Ausführung kommt, wurde jede Bedingung in eine extra Zeile geschrieben, da das Werkzeug ja das Erreichen von Zeilen prüft. In Zeile 11 wird angezeigt, dass der Vokal 'a' erkannt wurde. Die folgenden Bedingungen in den Zeilen 12 bis 15 werden nicht ausgeführt, weil der Test auf 'a' `true` ergab und die Auswertung weiterer mit einem logischen *oder* verbundenen Bedingungen daher überflüssig ist (Kurzschlussauswertung). *Hinweis*: Die ermittelte Zeilenüberdeckung für den Testfall beträgt 66,7%, was an der Aufteilung der Abfrage in mehrere Zeilen liegt. Wäre die Abfrage in einer Zeile aufgeführt, würde 100% erreicht. Der String im letzten Test enthält zwei Zeichen, sodass die Schleife genau zweimal ausgeführt wird. Für diesen Test könnten es auch mehr als zwei Zeichen sein.

Current view:	top level - vokale - vokale.cpp (source / functions)		Hit	Total	Coverage
Test:	dertest.info	Lines:	8	12	66.7 %
Date:	2015-11-06	Functions:	1	1	100.0 %

```
      Line data    Source code
    1            : #include "vokale.h"
    2            : #include <cstring>
    3            :
    4            : // Vokale zählen (ohne Umlaute)
    5          1 : std::size_t vokale(const std::string& str) {
    6          1 :   std::size_t z = 0;
    7          2 :   for(auto c : str) {
    8          1 :     if(isupper(c)) {
    9          1 :       c = std::tolower(c);   // Umwandlung in Kleinbuchstaben
   10            :     }
   11          1 :     if(c == 'a'
   12          0 :        || c == 'e'
   13          0 :        || c == 'i'
   14          0 :        || c == 'o'
   15          0 :        || c == 'u') {
   16          1 :       ++z;
   17            :     }
   18            :   }
   19          1 :   return z;
   20            : }
   21            :
```

Abbildung 5-5: Schleifenkörper wird genau einmal durchlaufen.

```
TEST(Vokale_zaehlen, Boundary_Interior_Test_2) {
  EXPECT_EQ(1, vokale("AX"));
}
```

Listing 5.8: Boundary-Interior-Test 2

Abbildung 5-6 zeigt die gemessene Überdeckung.

Current view:	top level - vokale - vokale.cpp (source / functions)		Hit	Total	Coverage
Test:	dertest.info	Lines:	12	12	100.0 %
Date:	2015-11-06	Functions:	1	1	100.0 %

```
      Line data    Source code
    1            : #include "vokale.h"
    2            : #include <cstring>
    3            :
    4            : // Vokale zählen (ohne Umlaute)
    5          1 : std::size_t vokale(const std::string& str) {
    6          1 :   std::size_t z = 0;
    7          3 :   for(auto c : str) {
    8          2 :     if(isupper(c)) {
    9          2 :       c = std::tolower(c);   // Umwandlung in Kleinbuchstaben
   10            :     }
   11          2 :     if(c == 'a'
   12          1 :        || c == 'e'
   13          1 :        || c == 'i'
   14          1 :        || c == 'o'
   15          1 :        || c == 'u') {
   16          1 :       ++z;
   17            :     }
   18            :   }
   19          1 :   return z;
   20            : }
   21            :
```

Abbildung 5-6: Schleifenkörper wird zweimal durchlaufen.

Weil das zweite Zeichen im String kein Vokal ist, müssen die mit *oder* verbundenen Bedingungen sämtlich ausgeführt werden. Der Wert in Zeile 11 ist um 1 größer als die Werte in den Zeilen 12 bis 15, weil beim ersten Durchlauf der Schleife der Vokal 'a' erkannt wurde. Die Schleife wurde somit zweimal ausgeführt (siehe Zeilen 7 und 8).

Wir raten, Schleifen auf diese Art und Weise einem separaten und genauerem Test zu unterziehen. Aber auch hier besteht die Schwierigkeit, die Eingabewerte des Testobjekts so zu wählen, dass die untersuchte Schleife auf genau die geforderte Weise zur Ausführung kommt (siehe Wahl des dritten Testfalls mit den Werten 'AX').

Hinweis: Bei jedem der ausgeführten Testfälle ist nicht nur zu kontrollieren, ob die geforderte Überdeckung damit erreicht wird, sondern es ist auch zu prüfen, ob der eigentliche Test auch das erwartete Ergebnis liefert!

5.2 Test komplexer Entscheidungen

Bisher haben wir von Entscheidungen, Entscheidungsausgängen und Bedingungen gesprochen. Es ist an der Zeit, die Begriffe genauer zu definieren: Eine Entscheidung steuert den Ablauf des Testobjekts. Eine Entscheidung ist beispielsweise eine *If*-Abfrage oder die Wiederholung einer Schleife. Eine Entscheidung hat immer zwei Ausgänge (`true` und `false`). Eine Entscheidung kann aus nur einer Bedingung, zum Beispiel `(x > 0)`, oder auch aus mehreren Bedingungen verknüpft mit booleschen Operatoren bestehen, zum Beispiel `((x > 100) && (y > 100))` (die inneren Klammern sind nicht notwendig, sie dienen der Lesbarkeit.). Jede der Bedingungen hat ebenfalls zwei Ausgänge, jeweils einen der beiden booleschen Werte. Die Bedingungen werden auch als atomare Bedingungen bezeichnet, da sie nicht weiter zerlegt werden können, im Gegensatz zu einer Entscheidung, die aus mehreren Bedingungen besteht. Das Ergebnis der Entscheidung – der Entscheidungsausgang – ergibt sich aus der Auswertung der einzelnen Bedingungsausgänge verknüpft mit den booleschen Operatoren. Soweit die Begriffserörterungen.

Der oben beschriebene kontrollflussbasierte Entscheidungstest betrachtet nur den Entscheidungsausgang, aber nicht, wie der Wert entstanden ist. Das ist aber von Interesse, wenn die Entscheidung aus mehreren Bedingungen zusammengesetzt ist. Im vorherigen Abschnitt gibt es schon eine zusammengesetzte Entscheidung (die *If*-Abfrage zur Vokal-Bestimmung), ohne dass im Detail darauf eingegangen wurde. Das wird hier nachgeholt.

```
if((x > 100) && (y > 100)) {
  ...
}
```

Listing 5.9: Zusammengesetzte Entscheidung

Die in Listing 5.9 aufgeführte Entscheidung besteht aus zwei atomaren Bedingungen. Die einzelnen Wahrheitswerte werden bei zusammengesetzten Entscheidungen mit booleschen Operatoren (logisches *und*, logisches *oder*, ...) verknüpft und der Wahrheitswert der Entscheidung – der Wert des Entscheidungsausgangs – wird ermittelt. Nur diesen Entscheidungsausgang – wie beim Entscheidungstest – zu berücksichtigen, wird der Komplexität von zusammengesetzten Entscheidungen nicht gerecht. Die Wahrheitswerte der Bedingungen und deren Kombination mit den booleschen Operatoren wird für die Ermittlung von weiteren Testfällen genutzt. Dabei wird jede Entscheidung – genauer jede Entscheidung, die aus mehreren Bedingungen zusammengesetzt ist – im Testobjekt separat betrachtet. Die unterschiedlichen Ansätze werden im Folgenden an Beispielen erörtert.

Noch eine Anmerkung vorweg: Man kann eine komplexe `if`-Abfrage sehr einfach gestalten, wenn statt einer zusammengesetzten booleschen Entscheidung nur eine einfache Variable verwendet wird. Dieser muss vor der Abfrage ein passender Wert zugewiesen werden.

```
bool bedingung = (x > 100) && (y > 100);
if(bedingung) {           // fraglich
  ...
}
```

Listing 5.10: `if` mit einfacher Variable

Das sieht vielleicht einfacher aus – die `if`-Abfrage besteht ja nur noch aus einer booleschen Variablen –, verlagert das Problem aber nur auf eine andere Ebene.

5.2.1 Einfacher Bedingungstest

Der einfache Bedingungstest (condition testing) fordert, dass jede Bedingung einer Entscheidung beide Wahrheitswerte im Test angenommen haben soll (condition coverage).

```
if((x > 100) && (y > 100)) {
  ...
}
```

Listing 5.11: Beispiel für einen einfachen Bedingungstest

Bei der Entscheidung in Listing 5.11 können beispielsweise die Werte 99 und 101 für x und y zum Test herangezogen werden. Es müssen somit Testfälle spezifiziert werden, bei denen die Variablen x und y mit den angegebenen Werten (99, 101) bei der Abfrage zur Ausführung kommen.

Mit den Werten x=101 und y=101 werden beide Bedingungen der Entscheidung jeweils zu `true` ausgewertet. Der Entscheidungsausgang ist ebenfalls `true`. Mit den Werten x=99 und y=99 ergibt sich sowohl für die beiden

Bedingungen als auch für den Entscheidungsausgang der Wert `false`. Das Maß der Überdeckung ist festgelegt als:

Einfache Bedingungsüberdeckung = E/A mit
E = Anzahl der ausgeführten Bedingungsausgänge
A = Anzahl aller Bedingungsausgänge einer Entscheidung

Im Beispiel gibt es vier Bedingungsausgänge (je zwei pro Bedingung). Mit den beiden oben angegebenen Werten für x und y werden alle vier Bedingungsausgänge ausgeführt. Wir erreichen somit eine 100%ige Überdeckung.

Bei ungeschickter Wahl der Werte mit x=99 und y=101 sowie den Werten x=101 und y=99 ist zwar die Anforderung an den einfachen Bedingungstest – jede atomare Bedingung muss einmal zu `true` und einmal zu `false` ausgewertet worden sein – ebenfalls zu 100% erfüllt, aber der Entscheidungsausgang nimmt bei beiden Kombinationen den Wert `false` an.

Tabelle 5-1 zeigt die möglichen Testfälle. Bei beiden Testfallkombinationen (a und b) ist der einfache Bedingungstest erfüllt – jede Bedingung hat beide booleschen Werte angenommen. Aber nur bei den beiden Testfällen 1a und 2a nimmt auch der Entscheidungsausgang beide booleschen Werte an.

	(x > 100)	(y > 100)	Wert von x	Wert von y	Entscheidungsausgang
1a	true	true	101	101	true
2a	false	false	99	99	false
1b	false	true	99	101	false
2b	true	false	101	99	false

Tabelle 5-1: Testfälle für den einfachen Bedingungstest

Das Verfahren erzwingt also nicht, dass bei der Durchführung der Testfälle der Entscheidungsausgang mindestens einmal den Wert `true` und mindestens einmal den Wert `false` angenommen hat. Es wird ausschließlich auf die atomaren Bedingungen fokussiert, und dass diese beide booleschen Werte mindestens einmal angenommen haben. Auch wenn eine Entscheidung aus mehr als zwei Bedingungen besteht, reichen zwei Testfälle aus, wenn zum Beispiel jede Bedingung in einem Testfall zu `true` ausgewertet wird und in einem zweiten zu `false`. Dazu müssen die Bedingungen der Entscheidung voneinander unabhängig sein. Kombinationen werden nicht betrachtet oder vorgeschrieben.

> Der einfache Bedingungstest ist kein zu empfehlendes Verfahren zur Überprüfung komplexer Entscheidungen, wenn nicht garantiert wird, dass der Entscheidungsausgang – der ja den weiteren Ablauf des Testobjekts steuert – beide möglichen Varianten annimmt!

5.2.2 Mehrfachbedingungs- oder Bedingungskombinationstest

Der nächste Ansatz, der Mehrfachbedingungstest (multiple condition testing), ist naheliegend: Alle Kombinationen der Wahrheitswerte der Bedingungsausgänge sind zu testen (multiple condition coverage). Bei zwei Bedingungen sind es $2^2 = 4$ Kombinationen. Bleiben wir bei unserem Beispiel und den gewählten Zahlenwerten, dann ergeben sich folgende vier Möglichkeiten der Auswertung der Entscheidung (siehe Tabelle 5-2):

	(x > 100)	(y > 100)	Wert von x	Wert von y	Entscheidungsausgang
1	true	true	101	101	true
2	true	false	101	99	false
3	false	true	99	101	false
4	false	false	99	99	false

Tabelle 5-2: Auswertung der Entscheidung

Hinweis: Für den kontrollflussbasierten Entscheidungstest reicht die erste und eine der drei anderen Kombinationen aus, da nur der Entscheidungsausgang betrachtet wird. Für den einfachen Bedingungstest sind es zum Beispiel die Testfälle 2 und 3!

Besteht die Entscheidung aus mehr als zwei Bedingungen, erhöht sich der Aufwand mit jeder hinzukommenden Bedingung um den Faktor 2. Bei 4 Bedingungen sind es 16 mögliche Kombinationen.

> Beim Mehrfachbedingungstest sind alle Kombinationen der Bedingungsausgänge miteinander zu kombinieren. Kein wirklicher Lean-Ansatz zur Prüfung komplexer Entscheidungen.

Bestehen Abhängigkeiten zwischen den Bedingungen, kommt der Mehrfachbedingungstest an seine Grenzen. Sehen wir uns das folgende Beispiel an:

```
if((x > 0) && (x < 101)) {
  ...
}
```

Listing 5.12: Prüfung eines Wertebereichs

Es soll geprüft werden, ob x im Wertebereich 1 bis 100 liegt. Wie sehen hier die Daten (d.h. der Wert von x) aus, damit alle Kombinationen der Bedingungsausgänge berücksichtigt werden (siehe Tabelle 5-3)?

	(x > 0)	(x < 101)	Wert von x	Entscheidungsausgang
1	true	true	3	true
2	true	false	103	false
3	false	true	0	false
4	false	false	?	?

Tabelle 5-3: Auswertung der Entscheidung mit abhängigen Bedingungen

Es gibt keinen Wert für x, bei dem die erste Bedingung (x > 0) zu `false` und die zweite Bedingung (x < 101) ebenfalls zu `false` ausgewertet werden kann. Ein Wert kann nicht kleiner als 1 und zugleich größer als 100 sein!

Es ist daher zu prüfen, ob alle geforderten Kombinationen überhaupt realisierbar sind. Die Mehrfachbedingungsüberdeckung ist somit nicht immer zu 100% erreichbar. Der nächste Ansatz beseitigt dieses Problem, erfordert aber mehr Überlegungen, da eine Auswahl aus den möglichen Kombinationen zu treffen ist.

5.2.3 Modifizierter Bedingungs-/Entscheidungstest

Im Gegensatz zum Mehrfachbedingungstest verlangt der modifizierte Bedingungs-/Entscheidungstest (modified condition decision testing) nicht, alle Kombinationen der einzelnen Bedingungsausgänge zu testen.

> Kernidee des Verfahrens ist folgende: Wenn eine einzelne Bedingung zu `false` statt zu `true` (oder umgekehrt) ausgewertet wird, sich aber der Entscheidungsausgang nicht verändert, also völlig unabhängig von der Auswertung dieser einen Bedingung ist, dann ist es wenig sinnvoll, diese Änderung mit Testfällen zu prüfen. Der modifizierte Bedingungs-/Entscheidungstest beschränkt sich auf solche Testfälle, bei denen die Änderung eines einzelnen Bedingungsausgangs den Entscheidungsausgang ebenfalls ändert.

Sehen wir uns zur Verdeutlichung das oben beim einfachen Bedingungstest (Seite 159) beschriebene Beispiel nochmals an:

```
if((x > 100) && (y > 100)) {
   ...
}
```

Listing 5.13: Beispiel für einen einfachen Bedingungstest

Grundlage unserer Erläuterung ist die Tabelle 5-2 mit den vier Kombinationen. Ausgehend von der letzten Kombination mit `false` und `false` für die beiden Bedingungen in Zeile 4 nehmen wir an, dass die erste Bedingung

(x > 100) als Ergebnis `true` statt `false` liefert. Die zweite (y > 100) liefert wie bisher `false`. In der Tabelle 5-2 betrachten wir somit die Zeilen 2 und 4. Das Ergebnis der Entscheidung ändert sich durch Änderung der ersten Bedingung von `false` nach `true` nicht, es bleibt `false`. Die gleiche Überlegung gilt für die zweite Bedingung (y > 100). Wechselt hier der Wahrheitswert von `false` auf `true` und der Wert der ersten Bedingung (x > 100) bleibt unverändert bei `false`, so ändert sich der Entscheidungsausgang nicht, er bleibt bei `false` (Zeile 3 und 4 in der Tabelle 5-2).

Beim modifizierten Bedingungs-/Entscheidungstest kann die Kombination der beiden Bedingungen – genauer der Bedingungsausgänge – (`false && false`) entfallen und muss beim Test nicht berücksichtigt werden, denn wie eben dargelegt, verursacht die Änderung des Bedingungsausgangs bei dieser Kombination (`false && false`) – bei beiden Bedingungsausgängen – keine Änderung des Entscheidungsausgangs.

Allgemein: Bei einer logischen *und*-Verknüpfung zwischen zwei Bedingungen ist die Kombination `false && false` nicht beim Test zu berücksichtigen. Analog dazu ist bei der logischen *oder*-Verküpfung die Kombination `true || true` beim Testen überflüssig, da sich bei der Änderung einer der beiden Bedingungen der Entscheidungsausgang nicht ändert.

Prima, das hilft uns ja bei unserem anderen Beispiel mit `if((x > 0) && (x < 101))`. Für die Kombination `false` && `false` haben wir ja sowieso kein Testdatum für x parat gehabt!

Die Ersparnis an Testfällen scheint nicht sehr groß zu sein, aber immerhin können nun auch alle geforderten Kombinationen tatsächlich mit Testdaten ausgeführt werden. Somit ist eine 100%ige Erfüllung der geforderten Kombinationen möglich.

Verknüpfen dreier Bedingungen

Wie sieht es aus, wenn drei Bedingungen mit logischen Operatoren in einer Entscheidung verknüpft sind? Sehen wir uns hierzu zunächst ein einfaches Beispiel als Programmtext (siehe Listing 5.14) an:

```
if((x > 0) && (y > 0) && (z > 0)) {
  ...
}
```

Listing 5.14: Verknüpfen dreier Bedingungen

Tabelle 5-4 zeigt dazu alle möglichen Kombinationen der Bedingungsausgänge und der Entscheidungsausgänge an.

Starten wir bei der Kombination 1 mit unseren Überlegungen. Ändert sich das Ergebnis der letzten Bedingung (z > 0) von `true` zu `false`, so ändert sich auch der Entscheidungsausgang von `true` zu `false`. Die Kombinationen 1 und 2 sind somit beim Test zu berücksichtigen.

	(x > 0)	(y > 0)	(z > 0)	Entscheidungsausgang
1	true	true	true	true
2	true	true	false	false
3	true	false	true	false
4	true	false	false	false
5	false	true	true	false
6	false	true	false	false
7	false	false	true	false
8	false	false	false	false

Tabelle 5-4: Drei Bedingungen mit logischer *und*-Verknüpfung

Das Gleiche trifft zu für die Bedingung (y > 0) mit der Kombination 1 und 3 und für die Bedingung (x > 0) mit der Kombination 1 und 5. Zu beachten ist, dass sich die ausgewählten Kombinationen jeweils nur durch den Wert einer Bedingung – eines Bedingungsausgangs – unterscheiden. *Dieser Unterschied bewirkt aber jeweils eine Änderung des Entscheidungsausgangs.*

Alle anderen Kombinationen bewirken bei Änderung nur eines Bedingungsausgangs keine Änderung des Entscheidungsausgangs. Wenn sich beispielsweise bei den Kombinationen 7 und 8 der Bedingungsausgang der Bedingung (z > 0) ändert, behält der Entscheidungsausgang unverändert den Wert `false`. Diese Änderung muss daher nicht getestet werden.

Von den insgesamt acht möglichen Kombinationen, die bei diesem Beispiel auch alle realisierbar wären, fordert der modifizierte Bedingungs-/Entscheidungstest die Ausführung der vier Testfälle 1, 2, 3 und 5 (siehe Tabelle 5-5). Dabei ist sichergestellt, dass auch der Entscheidungsausgang beide booleschen Werte annimmt.

	(x > 0)	(y > 0)	(z > 0)	Entscheidungsausgang
1	**true**	**true**	**true**	true
2	true	true	**false**	false
3	true	**false**	true	false
5	**false**	true	true	false

Tabelle 5-5: Die vier Testfälle zur Erfüllung der Überdeckung

Zusammengefasst: Bei den Kombinationen 1 und 2 ändert sich nur der Bedingungsausgang der Bedingung (z > 0) (in der Tabelle jeweils fettgedruckt dargestellt) und bewirkt eine Änderung des Entscheidungsausgangs (von `true` auf `false`). Bei den Kombinationen 1 und 3 ändert sich nur die Bedingung (y > 0) und bewirkt ebenfalls die Änderung des Entscheidungsausgangs.

Und bei den Kombinationen 1 und 5 ist es die Bedingung (x > 0), die auch als einzige sich ändert und den Entscheidungsausgang ebenfalls verändert.

Zum Vergleich: Für den einfachen Bedingungstest (alle Bedingungen nehmen beide möglichen Wahrheitswerte an) reichen zwei Kombinationen aus (z.B. 1 und 8 aus Tabelle 5-4). Die Wahl der Kombinationen 4 und 5 ist auch möglich, da alle drei atomaren Bedingungen beide Wahrheitswerte annehmen, allerdings nimmt der Entscheidungsausgang bei beiden Kombinationen den Wert `false` an. Beim Mehrfachbedingungstest sind alle acht Kombinationen zu testen, um dieses Kriterium zu 100% zu erfüllen. Beim modifizierten Bedingungs-/Entscheidungstest reduzieren sich die Testfälle auf vier.

> Damit kann der modifizierte Bedingungs-/Entscheidungstest als »lean« angesehen werden, wenn komplexe Entscheidungen zu testen sind.

Anforderungen an die Überdeckung

Der modifizierte Bedingungs-/Entscheidungstest führt zur modifizierten Bedingungs-/Entscheidungs-Überdeckung (modified condition/decision coverage (MC/DC)). Bei diesem Ansatz zum Testen von komplexen Entscheidungen muss nachgewiesen werden, dass jede Bedingung alleine – also bei Gleichbleiben der anderen Bedingungen – den Entscheidungsausgang verändert. Welche Werte die anderen Bedingungen dabei haben, spielt keine Rolle. Für einen Test zum Erreichen dieser Überdeckung wird Folgendes verlangt (angepasst an eine C++-Funktion):

- Jeder Ausgangspunkt der Funktion (`return`, Exception) muss mindestens einmal zum Tragen kommen.
- Jede (atomare) Bedingung in einer Entscheidung muss mindestens einmal `true` und mindestens einmal `false` sein.
- Jeder Entscheidungsausgang muss mindestens einmal `true` und mindestens einmal `false` sein.
- Für jede (atomare) Bedingung in einer Entscheidung muss es einen Test geben, bei dem die Änderung des Bedingungsausgangs zu einer Änderung des Entscheidungsausgangs führt. Dabei müssen die anderen Bedingungsausgänge unverändert bleiben.

Im folgenden Abschnitt werden diese Anforderungen an Beispielen gezeigt, einschließlich der Analyse der zusammengesetzten Entscheidungen durch ein Analysewerkzeug, damit Sie sehen, wie MC/DC funktioniert. Auch für die Testfallerzeugung bei komplexeren Bedingungen empfehlen wir, ein Werkzeug zu Hilfe zu nehmen.

Verknüpfen von drei Bedingungen: Variante mit *und* und *oder*

Sehen wir uns noch ein weiteres Beispiel an, und zwar eine Entscheidung, die eine Kombination von *und*- und *oder*-Verknüpfungen enthält. Listing 5.15 zeigt den Programmtext und die dazugehörige Tabelle 5-6 die möglichen Kombinationen:

```
if(((x > 0) && (y > 0)) || (z > 0)) {
  ...
}
```

Listing 5.15: Verknüpfen dreier Bedingungen

	(x > 0)	(y > 0)	(z > 0)	Entscheidungsausgang
1	true	true	true	true
2	**true**	**true**	false	true
3	true	false	**true**	true
4	true	**false**	**false**	false
5	false	true	**true**	true
6	**false**	true	**false**	false
7	false	false	**true**	true
8	false	false	**false**	false

Tabelle 5-6: Drei Bedingungen mit logischer *und*- und *oder*-Verknüpfung

Die fettgedruckten Einträge bedeuten, dass sich der Entscheidungsausgang ändert, wenn der fettgedruckte Wert der Bedingung sich ändert. In Zeile 2 und 6 ändert sich der Wahrheitswert der ersten Bedingung (x > 0) von `true` auf `false`, ebenso der Entscheidungsausgang.[19] Die Erläuterungen zu den einzelnen Kombinationen folgen weiter unten. Auch hier müssen wir systematisch die Kombinationen betrachten, um die erforderlichen Testfälle herauszufiltern. Und damit wird auch schon ein großer Nachteil deutlich: Bei jeder Entscheidung ist zu analysieren, welche Kombinationen der booleschen Ergebnisse der Bedingungen – welche Bedingungsausgänge – beim Testen zu berücksichtigen sind.

Fangen wir mit der Kombination 1 an und betrachten jede Änderung eines Bedingungsausgangs, und ob diese Änderung auch den Entscheidungsausgang verändert. Kombinationen 1 (`true, true, true -> true`) und 2 (`true, true, false -> true`) ergeben keine Änderung des Entscheidungsausgangs, 1 und 3 (`true, false, true -> true`) sowie 1 und 5 (`false, true, true -> true`) ebenfalls nicht. Damit ist Kombination 1 beim Test nicht zu berücksichtigen.

[19] Für die zweite Bedingung (y > 0) sind es die Zeilen 2 und 4 und für die dritte Bedingung (z > 0) die Zeilen 3 und 4, ebenso 5 und 6 sowie 7 und 8.

Wie sieht es mit der Kombination 2 aus? Bei 2 und 1 (Änderung des Ergebnisses der Bedingung (z > 0)) tritt keine Änderung des Entscheidungsausgangs ein.[20] Anders sieht es bei 2 (`true, true, false -> true`) und 4 (`true, false, false -> false`) und bei 2 und 6 (`false, true, false -> false`) aus. Bei 2 und 4 bewirkt die Änderung des Bedingungsausgangs von (y > 0) und bei 2 und 6 bewirkt die Änderung des Bedingungsausgangs der Bedingung (x > 0) die Veränderung des Entscheidungsausgangs (bei beiden von `true` nach `false`). Damit sind die Kombinationen 2, 4 und 6 beim Test zu berücksichtigen. Wir haben nun die Bedingungen (x > 0) und (y > 0) untersucht und dabei alle Testfälle ermittelt, bei denen eine Änderung eines Bedingungsausgangs auch den Entscheidungsausgang verändert.

> Zur Feststellung der MC/DC-Überdeckung bezüglich einer einzelnen Bedingung wird stets ein *Paar* von Tests benötigt. Dabei ergeben sich auch Wahlmöglichkeiten, da es mehrere Paare gibt, die das MC/DC-Kriterium erfüllen (s.u.).

Nun müssen wir uns um die Bedingung (z > 0) kümmern, da diese bei Änderung des Bedingungsausgangs noch keine Änderung des Entscheidungsausgangs bewirkt hat. Betrachten wir die Kombinationen 3 (`true, false, true -> true`) und 4 (`true, false, false -> false`). Eine Änderung von Bedingung (z > 0) bewirkt eine Änderung des Entscheidungsausgangs. Wir müssen diese beiden Kombinationen bei unserem Test hinzunehmen. Aber auch die Kombinationen 5 (`false, true, true -> true`) und 6 (`false, true, false -> false`) sowie 7 (`false, false, true -> true`) und 8 (`false, false, false -> false`) erzielen den gleichen Effekt – Änderung des Entscheidungsausgangs durch alleinige Änderung des Bedingungsausgangs der Bedingung (z > 0). Der Unterschied zwischen den drei zuletzt genannten Kombinationen besteht in den Ergebniswerten der anderen beiden Bedingungen.[21]

Bezüglich der Bedingung (z > 0) kann also statt der Kombinationen 3 und 4 auch ein anderes der oben angegebenen Paare genommen werden. Der modifizierte Bedingungs-/Entscheidungstest schreibt nur vor, dass eine – beliebige – Kombination zum Nachweis herangezogen werden muss. Auf die beiden anderen Kombinationen kann verzichtet werden, um die Anzahl der Testfälle klein zu halten.

[20] Diese Kombination ist ja eben mit 1 und 2 bereits überprüft worden. Der Systematik halber ist sie hier erneut aufgeführt. 2 und 3 wird nicht betrachtet, da sich zwei Bedingungsausgänge ((y>0) und (z>0)) unterscheiden und ja nur eine Änderung Einfluss auf den Entscheidungsausgang haben soll.

[21] Es gibt Verfahren, bei denen diese Kombinationen ebenfalls zu berücksichtigen sind. Auf diese Verfahren wird hier nicht näher eingegangen.

Statt der Kombinationen 3 und 4 braucht nur die Kombination 3 hinzugenommen werden. Kombination 3 unterscheidet sich von Kombination 4, die wir für den Test ja bereits ausgewählt haben, in passender Weise – nur das Ergebnis der Bedingung (z > 0) ist anders, ebenso der Entscheidungsausgang.

Damit haben wir die Kombinationen 2, 3, 4 und 6 ausgewählt (siehe Tabelle 5-7). Wie Sie sehen, hat jede Bedingung bei den vier Kombinationen beide Wahrheitswerte angenommen und auch der Entscheidungsausgang nimmt beide Werte an. Ebenso beeinflusst eine Änderung einer Bedingung die Änderung des Entscheidungsausgangs, wie oben gezeigt.

	(x > 0)	(y > 0)	(z > 0)	Entscheidungsausgang
2	**true**	**true**	false	true
3	true	false	**true**	true
4	true	**false**	**false**	false
6	**false**	true	false	false

Tabelle 5-7: Testfälle für den modifizierten Bedingungs-/Entscheidungstest

Nochmals zusammengefasst: Bei den Kombinationen 2 und 6 ändert sich nur der Bedingungsausgang von (x > 0), dieses bewirkt eine Änderung des Entscheidungsausgangs (von `true` auf `false`). Bei den Kombinationen 2 und 4 ändert sich nur die Bedingung (y > 0) und bewirkt ebenfalls die Änderung des Entscheidungsausgangs. Und bei 3 und 4 ist die Bedingung (z > 0) die entscheidende Bedingung, die sich als einzige ändert und den Entscheidungsausgang ebenfalls verändert.

Nun müssen die Eingabeparameter des Testobjekts nur noch so gewählt werden, dass die atomaren Bedingungen innerhalb der untersuchten Entscheidung in der geforderten Art und Kombination zur Ausführung kommen. Dies ist keine leichte Aufgabe in der Praxis!

Die vier ausgewählten Kombinationen genügen der modifizierten Bedingungs-/Entscheidungs-Überdeckung (modified condition/decision coverage (MC/DC)).

Prüfung mit dem Werkzeug

Um zu sehen, wie CTC++ die Analyse vornimmt, wird die obige `if`-Bedingung in einer Funktion isoliert:[22]

```
bool undoder(int x, int y, int z) {
  bool ergebnis = false;
  if(((x > 0) && (y > 0)) || (z > 0)) {
    ergebnis = true;
```

[22] Siehe Verzeichnis *beispiel/ctc/undoder/*.

```
  }
  return ergebnis;
}
```

Listing 5.16: Mehrfachbedingung in einer Funktion

Die `if`-Anweisung ist die zu prüfende Entscheidung.[23] Wir haben die Dateien mit dem Werkzeug CTC++ instrumentiert und die Testfälle entsprechend Tabelle 5-7 durchgeführt. Mit der Benennung der Testfälle (letzte Ziffer) in Listing 5.17 wird der Bezug zur entsprechenden Tabellenzeile hergestellt:

```
// Auszug aus beispiel/ctc/undoder/undodertest.cpp
TEST(komplexeBedingung, Tab5_7_Test_2) {
  EXPECT_EQ(true, undoder(1, 1, -1));
}

TEST(komplexeBedingung, Tab5_7_Test_3) {
  EXPECT_EQ(true, undoder(1, -1, 1));
}

TEST(komplexeBedingung, Tab5_7_Test_4) {
  EXPECT_EQ(false, undoder(1, -1, -1));
}

TEST(komplexeBedingung, Tab5_7_Test_6) {
  EXPECT_EQ(false, undoder(-1, 1, -1));
}
```

Listing 5.17: Testfälle

Abbildung 5-7 zeigt die Ausgabe des Werkzeugs. Die Frage ist – wie ist sie zu interpretieren? Der Programmcode hat einen weißen Hintergrund, die werkzeuggenerierte Dokumentation ist hellgrau hinterlegt.

Das Werkzeug zeigt für die MC/DC-Überdeckung 100% an, damit sind die vier von uns ermittelten und ausgeführten Testfälle als ausreichend anzusehen. Die zu testende Entscheidung in Zeile 5 wurde viermal ausgewertet, je zweimal zu `true` und `false`. So weit, so gut.

Das Werkzeug ermittelt aber insgesamt fünf Tests – genauer Kombinationen der Bedingungsausgänge (direkt nach der Entscheidung angegeben mit 1: bis 5:). Dabei werden die jeweiligen Bedingungsausgänge mit `(T)` und `(F)` aufgeführt. Spielt ein Bedingungsausgang keine Rolle für den Entscheidungsausgang, wird dies mit `(_)` gekennzeichnet. Es gibt drei solche Stellen

[23] Es reicht nicht als Rückgabewert der Funktion nur den Wert der Entscheidung direkt zu nehmen, um die Funktion zu vereinfachen, wenn die Entscheidungsüberdeckung ermittelt werden soll.

Source file: ./undoder.cpp
Instrumentation mode: multicondition **Reduced to:** MC/DC coverage
TER: 100 % (7/7) structural, 100 % (4/4) statement

Hits/True	False	-Line	Source
		1	#include "undoder.h"
		2	
Top			
4		3	bool undoder(int x, int y, int z) {
		4	bool ergebnis = false;
2	2	5	if(((x > 0) && (y > 0)) || (z > 0)) {
1		5	1: ((T) && (T)) || (_)
1		5	2: ((T) && (F)) || (T)
0		5	3: ((F) && (_)) || (T)
	1	5	4: ((T) && (F)) || (F)
	1	5	5: ((F) && (_)) || (F)
		5	MC/DC (cond 1): 1 + 5
		5	MC/DC (cond 2): 1 + 4
		5	MC/DC (cond 3): 2 + 4, 3 - 5
		6	ergebnis = true;
		7	}
4		8	return ergebnis;
		9	}

Abbildung 5-7: MC/DC-Überdeckung

in den fünf Kombinationen. Damit sind es insgesamt wieder alle acht überhaupt möglichen Kombinationen der drei Bedingungen. Nach der Auflistung folgen drei Zeilen mit den Angaben zur MC/DC-Überdeckung. Für jede der drei Bedingungen (cond 1 - 3) wird angegeben, welche Kombinationen zu berücksichtigen sind.

Um die Zuordnung zu erleichtern, haben wir eine weitere Tabelle 5-8 eingefügt und als Ergänzung die oben angegebene Nummerierung vom CTC++-Report mit aufgenommen. Zur Unterscheidung im Text haben wir das Kürzel »ctc« davor gesetzt.

Sehen wir uns die erste Kombination des Werkzeugs genauer an:

```
1: ((T) && (T)) || (_)
```

Die Bedingungen (x > 0) und (y > 0) sind beide `true`. Ob (z > 0) `true` oder `false` ist, spielt wegen der Kurzschlussauswertung keine Rolle – der Entscheidungsausgang ist unabhängig davon und steht durch die ersten beiden Bedingungen fest: `true`. Daher ist bei den ersten beiden Zeilen der Tabelle jeweils die 1 aus dem CTC++-Report angegeben – ctc1. Wie auf Seite 166

	(x > 0)	(y > 0)	(z > 0)	Entscheidungsausgang	CTC++-Report
1	true	true	true	true	(ctc1)
2	**true**	**true**	false	true	ctc1
3	true	false	**true**	true	ctc2
4	true	**false**	**false**	false	ctc4
5	false	true	**true**	true	ctc3
6	**false**	true	**false**	false	ctc5
7	false	false	**true**	true	ctc3
8	false	false	**false**	false	ctc5

Tabelle 5-8: Tabelle 5-6, ergänzt um die Nummern des CTC++-Reports

erläutert, spielt die Zeile 1 keine Rolle für die Tests, weswegen ctc1 eingeklammert ist.

Nummer ctc2 im Report entspricht Zeile 3 in der Tabelle, Nummer ctc3 den Zeilen 5 und 7, ctc4 der Zeile 4 und Nummer ctc5 den Zeilen 6 und 8.[24]

Nach dieser Auflistung folgt im Report die Übersicht über die `MC/DC (cond x)`-Überdeckung in Bezug auf die drei Bedingungen. Diese sind wie folgt zu interpretieren:

1. Das MC/DC-Kriterium für die Bedingung 1, also (x > 0), wird durch die Kombination der Tests ctc1 und ctc5 erreicht (Zeilen 2 und 6 der Tabelle 5-8 unterscheiden sich nur bei (x > 0)).
2. Das MC/DC-Kriterium für die Bedingung 2, also (y > 0), wird durch die Kombination der Tests ctc1 und ctc4 erreicht (Zeilen 2 und 4 der Tabelle 5-8).
3. Das MC/DC-Kriterium für die Bedingung 3, also (z > 0), wird durch die Kombination der Tests ctc2 und ctc4 erreicht (Zeilen 3 und 4 der Tabelle 5-8).

Unsere vier hergeleiteten Testfälle (siehe Tabelle 5-7) erfüllen somit das MC/DC-Kriterium!

Im Report ist noch ctc3 - ctc5 angegeben – warum? Bei beiden ist die zweite Bedingung ohne Relevanz (beides kann jeweils `true` oder `false` sein; mit `(_)` gekennzeichnet). Es können als Alternative (oder Ergänzung, falls gewünscht) zu ctc2 + ctc4 ebenso die Kombination der Tests in Zeilen 5 und 6 oder die Tests in den Zeilen 7 und 8 der Tabelle 5-8 gewählt werden. Diese Wahl/Alternativen wäre aber nicht »lean«, da Testfall 5 (aus der Tabelle) neu hinzukäme oder bei der anderen Wahl sogar zwei Testfälle: 7 und 8. Deshalb prüfen wir die Bedingung (z > 0) mit den Tests 3 und 4 (ctc2 und

[24] Wenn in der Bedingung ein `(_)` vorhanden ist, dann muss diese Zeile aus dem Report zweimal in der Tabelle aufgeführt werden, da in der Tabelle alle Kombinationen vorhanden sind.

ctc4 im Report), da wir Test 4 ja schon nutzen und nur Test 3 (bzw. ctc2) hinzukommt.

Nun sehen wir im Report noch eine 0 bei ctc3! Diese Kombination 3: wurde also bei den vier durchgeführten Tests nie ausgeführt. Eigentlich sind es ja zwei Kombinationen wegen `(_)` bei der Bedingung (y>0). Die erste Bedingung (x > 0) liefert `false`, der Wert der zweiten Bedingung (y > 0) ist egal und der Wert der dritten Bedingung (z > 0) ist `true`. Sehen wir uns Tabelle 5-5 an, dann stellen wir fest, dass genau diese Kombinationen nicht vorkommen. Bedingung (x > 0) mit `false` wird nur mit Bedingung (z > 0) mit `true` und nie mit `false` kombiniert, egal wie Bedingung (y > 0) ausgewertet wird. Da aber diese Kombinationen vom Werkzeug nur als zusätzliche Möglichkeit bei der dritten Bedingung (z>0) aufgeführt wurden (ctc3 – ctc5), spielen sie für das Erreichen der 100% MC/DC-Überdeckung keine Rolle. Es ist aber ein Hinweis auf weitere Möglichkeiten für Testfälle (wie oben bereits erwähnt). Die folgende beiden Testfallpaare bilden eine solche Ergänzung:

```
TEST(komplexeBedingung, Tab5_8_Test_5) {
  EXPECT_EQ(true, undoder(-1, 1, 1));
}

TEST(komplexeBedingung, Tab5_8_Test_6) {
  EXPECT_EQ(true, undoder(-1, 1, -1));
```

Listing 5.18: Ergänzende Tests

oder

```
TEST(komplexeBedingung, Tab5_8_Test_7) {
  EXPECT_EQ(true, undoder(-1, -1, 1));
}

TEST(komplexeBedingung, Tab5_8_Test_8) {
  EXPECT_EQ(true, undoder(-1, -1, -1));
}
```

Listing 5.19: Weitere ergänzende Tests

Dabei ist das erste Paar vorzuziehen, weil nur der Test aus Zeile 5 der Tabelle 5-8 hinzukäme – der Test aus Zeile 6 wurde ja schon berücksichtigt.

Zur Abschreckung: Verknüpfung mehrerer Bedingungen

Um die obige Empfehlung zu untermauern, komplexe Entscheidungen möglichst zu vermeiden, haben wir ein weiteres Beispiel beleuchtet. Für die Analyse wird die Entscheidung in eine Funktion gepackt:

Wir haben uns bisher Entscheidungen angesehen, die aus maximal drei Bedingungen zusammengesetzt sind. Mit jeder weiteren Bedingung erhöht sich der Aufwand entsprechend. Es ist daher ratsam, keine Entscheidungen zu programmieren, die aus vielen Bedingungen bestehen. Mit geschachtelten Abfragen lässt sich von der Logik her die gleiche Auswahl treffen. Eine solche Konstellation ist aber viel einfacher zu überprüfen, z. B. mit dem Entscheidungstest.

```
bool vielfachbedingung1(bool a, bool b, bool c, bool d, bool e) {
  bool ergebnis = false;
  if( (a && !b) || (!c && d) || !e ) {
    ergebnis = true;
  }
  return ergebnis;
}
```

Listing 5.20: Verknüpfung von fünf Bedingungen

Eine Entscheidung kann aus mehreren *Bedingungen* bestehen, die über boolesche Operatoren verknüpft sind. Im Beispiel liefern die Parameter bereits Werte, die teilweise direkt und teilweise nach Negation für die Auswertung der Entscheidung verwendet werden.

Die `if`-Entscheidung wurde mit nur fünf Testfällen von $2^5 = 32$ möglichen überprüft. Dabei waren alle Parameter bis auf einen `true` (siehe Tabelle 5-9).

	a	b	c	d	e	Entscheidungsausgang
1	false	true	true	true	true	false
2	true	false	true	true	true	true
3	true	true	false	true	true	true
4	true	true	true	false	true	false
5	true	true	true	true	false	true

Tabelle 5-9: Werte der fünf Parameter

Es gibt hier die fünf Bedingungen `a`, `!b`, `!c`, `d` und `!e`. Für die fünf Bedingungen (als bed-Parameter in Tabelle 5-10 angegeben) der Entscheidung – und auf die kommt es ja an – sehen die Testfälle mit den gewählten Parameterwerten wie folgt aus (siehe Tabelle 5-10).

	bed-a	bed-b	bed-c	bed-d	bed-e	Entscheidungsausgang
1	false	false	false	true	false	false
2	true	true	false	true	false	true
3	true	false	true	true	false	true
4	true	false	false	false	false	false
5	true	false	false	true	true	true

Tabelle 5-10: Werte der fünf Bedingungen

Wie in Tabelle 5-10 zu sehen, sind nur noch zwei Testfälle (1 und 4) vorhanden, bei denen eine Bedingung `true` annimmt und alle anderen Bedingungen `false` liefern. Bei diesem Beispiel erzwingt bereits die Wahl der Parameterkombinationen genaue Überlegungen, um die einzelnen Bedingungen der Entscheidung *passend* zu erhalten – wir wollten ja, dass jeweils nur eine Bedingung der Entscheidung `true` liefert.

Die fünf Testfälle ersetzen im Ernstfall keine Herleitung eines minimalen Satzes von Testfällen, die den MC/DC-Kriterien genügen. Es geht nur um den ersten Eindruck und darum, ob man dem Ergebnis des Werkzeugs Empfehlungen für weitere Testfälle entnehmen kann. Die MC/DC-Überdeckung wurde noch nicht erreicht. Das Ergebnis sehen Sie in Abbildung 5-8.

Instrumentation mode: multicondition **Reduced to:** MC/DC coverage
TER: 65 % (15/23) structural, 100 % (8/8) statement

Hits/True	False	-	Line	Source
			1	#include "vielfachbedingung.h"
			2	
Top				
5			3	bool vielfachbedingung1(bool a, bool b, bool c, bool d, bool e) {
			4	bool ergebnis = false;
3	2		5	if((a && !b) || (!c && d) || !e) {
1			5	1: (T && T) || (_ && _) || _
1			5	2: (T && F) || (T && T) || _
0			5	3: (T && F) || (T && F) || T
1			5	4: (T && F) || (F && _) || T
0			5	5: (F && _) || (T && T) || _
0			5	6: (F && _) || (T && F) || T
0			5	7: (F && _) || (F && _) || T
	0		5	8: (T && F) || (T && F) || F
	1		5	9: (T && F) || (F && _) || F
	0		5	10: (F && _) || (T && F) || F
	1		5	11: (F && _) || (F && _) || F
			5	MC/DC (cond 1): 1 + 11, 1 - 10
			5	MC/DC (cond 2): 1 + 9, 1 - 8
			5	MC/DC (cond 3): 2 + 9, 5 - 11
		-	5	MC/DC (cond 4): 2 - 8, 5 - 10
			5	MC/DC (cond 5): 4 + 9, 3 - 8, 6 - 10, 7 - 11
			6	ergebnis = true;
			7	}
5			8	return ergebnis;
			9	}

Abbildung 5-8: Ausgabe des Werkzeugs

Wer es genau wissen will: Der Testfall 1 aus Tabelle 5-10 findet sich bei der Werkzeugausgabe (siehe Abbildung 5-8) als 11: wieder: bed-a, bed-c und bed-e sind F, die beiden anderen werden nicht berücksichtigt (_). Testfall 2 entspricht 1: mit T für bed-a und bed-b sowie ohne Bedeutung für die drei anderen Bedingungen. Testfall 3 ist 2: mit T für bed-a, bed-c und bed-d, F für bed-b sowie (_) für bed-e. Testfall 4 entspricht 9: und sieht wie folgt aus: bed-a ist T, bed-b, bed-c und bed-e sind F und bed-d ist ohne Bedeutung (_). Der letzte Testfall 5 findet sich in der Werkzeugausgabe als 4: wieder. bed-a und bed-e sind T, bed-b und bed-c sind F, bed-d ist (_).

Auf den ersten Blick wird klar, dass es einer erheblichen Präzision und Konzentration beim Lesen bedarf, um weitere Schlüsse zu ziehen – und dies bei nur fünf verknüpften Bedingungen! Es wird deutlich, dass noch ein erheblicher Aufwand erforderlich ist, um 100% MC/DC-Überdeckung zu erreichen, ohne einfach alle 32 möglichen Kombinationen durchzuprobieren. Bei N verknüpften Bedingungen ist die vollständige Anzahl aller Testfälle 2^N. Für N = 10 ergeben sich schon 1024 Kombinationsmöglichkeiten. Das MC/DC-Verfahren bewirkt eine deutliche Verminderung der Zahl der Testfälle. Andererseits ist der Aufwand zur Testfallerzeugung entsprechend den MC/DC-Kriterien nicht unerheblich. Aus diesen Gründen empfehlen wir, die Anzahl verknüpfter Bedingungen zu reduzieren und für die Erzeugung der Testfälle ein Werkzeug einzusetzen.

5.3 Bewertung

Die strukturbasierten Testverfahren nutzen Informationen des Programmtextes zur Gewinnung von Testfällen. Bereits im einleitenden Abschnitt des Kapitels wurde auf die Schwierigkeiten hingewiesen, dass damit ein Test *gegen sich selbst* durchgeführt wird.

Bei allen strukturbasierten Ansätzen lässt sich die bei der Ausführung der Testfälle erreichte Codeüberdeckung mit entsprechenden Werkzeugen ermitteln. Mit der erreichten Überdeckung können Rückschlüsse auf die Qualität der bisher durchgeführten Testfälle gezogen werden. Eine niedrige Überdeckung deutet auf Lücken bei den Testfällen hin und es müssen entsprechende Ergänzungen vorgenommen werden, um eine höhere Überdeckung zu erreichen.

Wenn beispielsweise 100% *Anweisungsüberdeckung* nicht vorliegt und auch durch weitere Testfälle nicht erreicht wird, sind unerreichbare Programmteile (dead code) vorhanden – ein sehr nützlicher Hinweis. Die nicht erreichbaren Programmteile müssen dann genauer untersucht werden. Die Anweisungsüberdeckung bietet darüber hinaus keine weiteren nützlichen Informationen und ist ein sehr schwacher Ansatz.

Der *Entscheidungstest* prüft, ob jede Entscheidung im Testobjekt beide Wahrheitswerte bei der Durchführung der Testfälle angenommen hat. Dabei wird jede Entscheidung separat betrachtet. Und darin liegt der große Nachteil, denn oft treten Fehler erst bei der Kombination von mehreren Entscheidungen auf, wie wir es bei dem fehlerhaften Beispiel der Maximumbestimmung von drei Zahlen hoffentlich verdeutlichen konnten.

Der *Pfadtest* prüft keine einzelnen Konstrukte wie Anweisungen oder Entscheidungen des Testobjekts, sondern prüft einzelne mögliche Abläufe vom Eintritt oder Aufruf bis zur Beendigung oder dem Verlassen des Testobjekts. Jeder einzelne Testfall führt genau einen Pfad aus; unterschiedliche Testfälle führen unterschiedliche Pfade aus. In aller Regel kann die Summe aller möglichen Pfade nicht ermittelt werden und somit kann auch kein Grad der Überdeckung angegeben werden. Die weiteren strukturbasierten Testverfahren fokussieren auf einzelne Programmkonstrukte wie Schleifen und komplexe Entscheidungen.

Beim *Schleifentest* soll jede Schleife des Testobjekts durch drei Testfälle überprüft werden. Der Schleifenkörper soll gar nicht, einmal und mindestens zweimal ausgeführt werden. Parallel zur Durchführung der Testfälle ist zu prüfen, ob die geforderte Überdeckung erreicht wird. Gegebenenfalls sind ergänzende Testfälle zur Prüfung der einzelnen Schleifen zu ermitteln und auszuführen. Da bei Schleifen häufig Fehler auftreten, ist dieses Vorgehen anzuraten.

Entscheidungen steuern den Ablauf des Testobjekts. Sie sind damit das »Herzstück« und müssen besonders intensiv geprüft werden. Der Entscheidungstest, der nur den Entscheidungsausgang betrachtet, ist bei einfachen Entscheidungen, die nur aus einer Bedingung bestehen, ausreichend. Er greift aber bei komplexen Entscheidungen zu kurz. Drei Ansätze wurden vorgestellt: einfacher Bedingungstest, Mehrfachbedingungs- oder Bedingungskombinationstest und der modifizierte Bedingungs-/Entscheidungstest.

Der *einfache Bedingungstest* betrachtet jede Bedingung einer komplexen Entscheidung separat und fordert, dass jeder Bedingungsausgang bei der Durchführung der Testfälle beide Wahrheitswerte angenommen hat. Der Entscheidungsausgang bleibt dabei unberücksichtigt – kein adäquates Vorgehen zur Prüfung komplexer Entscheidungen.

Der *Mehrfachbedingungs- oder Bedingungskombinationstest* fordert die Prüfung aller Kombinationen und ist damit sehr aufwendig. Ein Problem besteht darin, dass bei Abhängigkeiten der Bedingungen einer Entscheidung eine 100%ige Überdeckung aller Kombinationen gar nicht erreichbar ist. Auch das ist kein sinnvolles Vorgehen für die Praxis.

Bleibt noch der *modifizierte Bedingungs-/Entscheidungstest*, den wir ausführlich beschrieben haben. Die Feststellung der erreichten Überdeckung ist dabei sehr aufwendig. Zuerst muss ermittelt werden, welche Wahrheitswerte für die einzelnen Bedingungsausgänge in welcher Kombination

zur Ausführung kommen sollen. Mit den entsprechenden Werkzeugen kann dann die Erfüllung der geforderten Überdeckung nachgewiesen werden.

Aber auch dieses Vorgehen hat seine Tücken, da auch hier die Bedingungen einzeln im Hinblick auf ihre Wirksamkeit auf den Entscheidungsausgang analysiert werden und nicht in Kombination mit anderen Bedingungen der komplexen Entscheidung. Es gibt unter anderem das Problem der Abhängigkeiten wie in `(A && B) || (A && C)`, wo das zweite A nicht unabhängig vom ersten variiert werden kann. Ein anderer Fall sind sogenannte degenerierte Fälle, logische Ausdrücke, die vereinfacht werden könnten, wie etwa `(A && B) || (!A && C) || (A && C && D) || (C && D && E)`. Sie sind redundant und sollten besser zu kürzeren, aber äquivalent logischen Ausdrücken umgeformt werden. Letzteres mag dann allerdings die Nachvollziehbarkeit beim Lesen erschweren, auch müsste die Äquivalenz nachgewiesen werden.

Der einfache MC/DC-Test ist überdies restriktiv, weil gefordert wird, dass eine Bedingung geändert, aber alle anderen gleich gehalten werden sollen. Letzteres ist aber nur bedingt sinnvoll, wenn eine oder mehrere der anderen Bedingungen den Ausgang gar nicht beeinflussen *können*. Der modifizierte Bedingungs-/Entscheidungstest ist daher nicht unumstritten; darauf wollen wir hier aber nicht weiter eingehen.

> Es gibt in Bezug auf komplexe Entscheidungen nur eine Empfehlung: *vermeiden*! Wenn es sich nicht verhindern lässt, dann sollen die komplexen Entscheidungen aus möglichst wenig Bedingungen bestehen und auch die Verwendung von unterschiedlichen booleschen Operatoren soll zurückhaltend erfolgen. Nur »Lean Programming« ermöglicht »Lean Testing«!

5.4 Bezug zu anderen Testverfahren

Einen direkten Bezug zu einzelnen Testverfahren gibt es nicht. Es besteht ein Zusammenhang zu allen Verfahren. Egal nach welchem oder welchen Testentwurfsverfahren die Testfälle spezifiziert wurden: Bei deren Ausführung ist die erreichte Codeüberdeckung zu messen und aus den Ergebnissen sind die oben bereits erwähnten Schlussfolgerungen – meist Ergänzung durch weitere Testfälle – zu ziehen.

5.5 Hinweise für die Praxis

Der Codeüberdeckung wird in der Praxis meist eine zu hohe Bedeutung beigemessen. 100% wird als zu erreichendes Ziel vorgegeben und bei dessen Erreichen werden die Testaktivitäten als ausreichend angesehen und beendet. Vorsicht! Dies ist kein sinnvolles Kriterium zur Beendigung des Tests, wie wir oben hoffentlich verdeutlichen konnten!

> Umgekehrt wird ein Schuh daraus: Sind die 100% nicht erreicht, dann fehlen noch Testfälle, die entsprechend zu ergänzen sind.

Diese Ergänzungen sind aber nicht einfach. Es müssen solche Testfälle zusätzlich spezifiziert werden, die genau einen bestimmten – bisher nicht ausgeführten – Teil des Testobjekts zur Ausführung bringen. Denken wir an den modifizierten Bedingungs-/Entscheidungstest, dann ist die Herausforderung darin zu sehen, eine einzelne Bedingung einer Entscheidung zu einem geforderten Wahrheitswert auswerten zu lassen, ohne dabei die anderen Bedingungen zu verändern. Und welche Eingabeparameter müssen mit welchen Werten belegt werden, um genau diesen Effekt – den Ablauf – zu erreichen? Dabei kann der Debugger eine große Hilfe sein, sich an die betreffenden Stellen »heranzuarbeiten« und damit Ideen für die Eingabewerte zu bekommen.

> Codeüberdeckung ist kein sinnvolles Kriterium zur Beendigung der Testaktivitäten! Sondern Codeüberdeckung liefert Hinweise über die Qualität der bisher durchgeführten Testfälle in Bezug auf Strukturelemente des Testobjekts.

6 Erfahrungsbasiertes Testen

Zunächst ein allgemeiner Hinweis zu den erfahrungsbasierten Testansätzen: Wir wollen Entwicklern eine Unterstützung an die Hand geben, wenn sie ihr Programm direkt nach der Programmierung testen. Hierzu haben wir eine Auswahl von systematischen Ansätzen zur Herleitung von Testfällen in den bisherigen Kapiteln vorgestellt. Die erfahrungsbasierten Ansätze liefern weitere Ideen für Tests! Hauptquelle für diese Ideen ist die eigene Erfahrung, die der Programmierer mitbringt. Nun stellt sich aber die Frage, ob er diese Erfahrung nicht auch bereits bei der Programmierung nutzt. Sicherlich wird er dies tun. Er kennt zum Beispiel die Problematik mit dem Wert null. Bei der Programmierung wird er darauf achten, dass keine Division durch null erfolgen kann. All seine Erfahrung wird er bei der Programmierung bereits berücksichtigen.

Wenn er nun nach der Programmierung erfahrungsbasiert testet, greift er ebenfalls auf seine Erfahrungen und Erkenntnisse zurück. Wenn er aber bei der Programmierung etwas übersehen oder falsch bedacht hat, wieso sollte er beim Testen schlauer sein?

Auch beim erfahrungsbasierten Testen werden wir Regeln und Hilfestellungen vorstellen, die dem Entwickler die ein oder andere zusätzliche Idee für Testfälle an die Hand geben, aber der große Fundus ist und bleibt seine Erfahrung. Insofern sind die erfahrungsbasierten Testansätze für den Entwicklertest nicht direkt geeignet.

Wir haben sie trotz dieser Einschränkung aufgenommen, da sie eine sehr sinnvolle Ergänzung zu den systematisch hergeleiteten Testfällen sind, und weil in der Praxis das »gegenseitige Testen« im Team durchaus verbreitet ist, um die Blindheit gegenüber den eigenen Fehlern zu beseitigen. In diesem Zusammenhang – Test von nicht selbst programmierter Software – sind die erfahrungsbasierten Ansätze sehr sinnvoll und nützlich.

Ein Tag im Leben des Praktikanten Otto

Zum Einstieg dient folgender Dialog zwischen dem Praktikanten Otto und der Softwareentwicklerin Paula.

Paula: Guten Tag, du bist also der neue Praktikant? Ich heiße Paula.

Otto: Ja, ich heiße Otto und bin seit heute in der Firma. Zuerst soll ich mich mit dem Testen von Software beschäftigen.
Paula: Ich weiß. Ich habe die Aufgabe, dich da ein wenig zu betreuen. Am besten fangen wir gleich an. Du kennst doch vermutlich das Buch 'Der C++-Programmierer' von Breymann, ich habe es hier im Regal stehen. Am Ende des Abschnitts 2.4 gibt es eine Funktion `fastbubblesort()`, die nach Angaben des Autors schneller als der normale Bubblesort ist, aber zwei schwere Fehler enthält, die es herauszufinden gilt. Lies' die Aufgabe durch und überlege, wie du die Funktion testen könntest. Wir benutzen übrigens hier das Google-Test-Framework, auch damit sollst du dich befassen. Ich glaube, du brauchst so eine halbe Stunde, um mit dem Test-Framework klarzukommen. Wenn du fertig bist, besprechen wir deine Vorschläge.

```
void fastbubblesort(std::vector<int>& feld) {
   int temp;
   do {
      temp = feld[0];
      for(std::size_t j = 1; j < feld.size(); j++) {
         if(feld[j] < feld[j-1]) {          // vertauschen
            temp     = feld[j-1];
            feld[j-1] = feld[j];
            feld[j]   = temp;
         }
      }
   } while(temp != feld[0]);            // false = keine Vertauschung mehr
}
```

Listing 6.1: BubbleSort-Übungsaufgabe mit absichtlichem Fehler

[Nach einiger Zeit...]

Otto: Hallo Paula, Google Test habe ich mir angesehen. Ich würde die Funktion `fastbubblesort()` so testen: Es werden verschiedene Vektoren angelegt und mit `fastbubblesort()` sortiert. Natürlich soll der Vorgang möglichst automatisiert ablaufen und wenig Schreibarbeit machen. Deshalb prüfe ich mit einer Funktion `bool istAufsteigendSortiert(const std::vector<int>& feld)`, ob das Ergebnis aufsteigend sortiert ist. Wenn ja, ist alles in Ordnung. Was meinst du dazu?
Paula: Ja, versuche es einfach.
Otto (nach einiger Zeit): Ich zeige dir mal meine Funktion:

```
bool istAufsteigendSortiert(const std::vector<int>& feld) {
   for(std::size_t j = 1; j < feld.size(); j++) {
      if(feld[j] < feld[j-1]) {
         return false;
       }
```

```
   }
   return true;
}
```

Listing 6.2: Ottos Funktion zum Prüfen auf korrekte Sortierung

Otto: Auch habe ich schon einen Test geschrieben:

```
TEST(istaufsteigendsortiertTest, zweiElemente) {
   EXPECT_TRUE(istAufsteigendSortiert(std::vector<int> ( {1, 2 } )));
   EXPECT_FALSE(istAufsteigendSortiert(std::vector<int> ( {2, 1 } )));
}
```

Listing 6.3: ... und ein Test dazu

Paula: Prima. Ein wenig Kommentar im Code wäre schön... Aber du brauchst das Rad nicht neu zu erfinden: Nimm die Bibliotheksfunktion `is_sorted()`. Ich sehe, dass du dich mit Google Test beschäftigt hast. Aber meinst du, dass ein Test mit zwei Elementen reicht? Immerhin kommt in der Funktion eine Schleife vor, deren Laufvariable einen Anfangswert und einen Endwert hat, und vielleicht noch einen in der Mitte.
Otto: Ja, das stimmt. Du meinst, ich müsste mindestens mit drei Elementen arbeiten?
Paula: Nicht nur, aber auch. Schließlich könnte es auch sein, dass der Vektor gar kein Element enthält oder nur eins.
[Otto wirkt nachdenklich.]
Otto: Dann müsste ich für diese Fälle weitere Tests vorsehen. Aber was ist mit vier oder fünf oder noch mehr Elementen?
Paula: Überleg' doch mal. Verhält sich eine Schleife mit noch mehr Elementen im Prinzip anders?
Otto: Nein, wohl nicht. Wenn eine Schleife mit drei Elementen funktioniert, tut sie das auch mit vier.
Paula: Richtig, jedenfalls in diesem Fall. Man sagt, dass solche Fälle *äquivalent* sind. Aber es gibt noch einen Einwand. `fastbubblesort()` sortiert einen Vektor. Das heißt aber auch, dass der Vektor *verändert* wird. Die von dir vorgeschlagene Funktion `istAufsteigendSortiert()` prüft nur die korrekte Sortierung. Was ist aber, wenn (3, 2, 1) sortiert werden soll und das Ergebnis (1, 2, 2) wäre? Es wäre zwar richtig sortiert, aber falsch. Denk doch darüber mal nach, Otto. Wir sehen uns nach dem Mittagessen. Oder kommst du mit in die Kantine?

[Eine Stunde später ...]

Otto: Hallo Paula. Ich meine, dass der Vektor im Prinzip gar nicht verändert werden kann, weil die einzige verändernde Operation in der Funktion `fastbubblesort()` eine *Vertauschung* ist. Alle Elemente bleiben erhalten, abgesehen von ihrer Reihenfolge.

Paula: Und wie würdest du das mit einem Test nachweisen?
Otto: Mit dieser Frage habe ich jetzt nicht gerechnet ... Da bin ich überfragt – ich glaube, ich brauche noch etwas Zeit!

[Stunden später ...]

Otto: Hallo Paula, man könnte das erwartete Ergebnis aufschreiben und dann vergleichen. Das Problem ist dabei die große Menge der Kombinationsmöglichkeiten, wenn der Vektor mehr als zwei Elemente hat. Eine elende Schreibarbeit, und mit hoher Wahrscheinlichkeit vertippt man sich oft.
Paula: Ja, da hast du recht. Ist dir eine Lösung eingefallen?
Otto: Ja. Ich habe mir gedacht, mit Schleifen alle möglichen Kombinationen zu erzeugen, aber das Ergebnis, mit dem verglichen werden soll, nicht aufzuschreiben, sondern mit einem *anderen* Sortierprogramm zu berechnen. Ich habe da an die Bibliotheksfunktion `std::sort()` gedacht. Es ist sehr unwahrscheinlich, dass ein Test nur deswegen gelingt, weil beide denselben Fehler haben. So wird die Bibliotheksfunktion `std::sort()` gleich mitgetestet. Dass alle Kombinationen berechnet werden, ist bei kleinen Zahlen noch tragbar. Mein Test sieht jetzt so aus:

```
TEST(fastbubblesortTest, vector_leer) {
   std::vector<int> v;
   std::vector<int> ergebnis;
   fastbubblesort1(v);
   ASSERT_EQ(ergebnis, v);
}

TEST(fastbubblesortTest, einElement) {
   std::vector<int> v { 1000 };
   std::vector<int> ergebnis { 1000 };
   fastbubblesort(v);
   ASSERT_EQ(ergebnis, v);
}

TEST(fastbubblesortTest, zweiElementeAlleKombinationen) {
   for(int i = 0; i < 2; ++i) {
      for(int j = 0; j < 2; ++j) {
         std::vector<int> v {i, j};
         std::vector<int> ergebnis = v;
         std::sort(ergebnis.begin(), ergebnis.end());
         fastbubblesort(v);
         ASSERT_EQ(ergebnis, v);
      }
   }
}
```

```
TEST(fastbubblesortTest, dreiElementeAlleKombinationen) {
   for(int i = 0; i < 3; ++i) {
      for(int j = 0; j < 3; ++j) {
         for(int k = 0; k < 3; ++k) {
               std::vector<int> v {i, j, k};
               show(v);                    // zeigt Vektor nur an
               std::vector<int> ergebnis = v;
               std::sort(ergebnis.begin(), ergebnis.end());
               fastbubblesort(v);
               ASSERT_EQ(ergebnis, v);
         }
      }
   }
}
```

Listing 6.4: Der Test mit den Kombinationen

Otto: Der erste Test mit dem leeren Vektor schlägt fehl – aber nicht mit einer Fehlermeldung des Test-Frameworks, sondern mit einem Speicherzugriffsfehler! Das Programm stürzt ab. Wenn auf ein nicht existierendes Vektorelement zugegriffen wird, gibt es in C++ nicht automatisch eine Exception, sondern ein undefiniertes Verhalten, auf Englisch *undefined behaviour*! Die Anweisung `temp = feld[0];` in `fastbubblesort()` ist der Übeltäter.
Paula: Prima! Wie kannst du das umgehen?
Otto: Entweder schreibe ich `temp = feld.at(0);`, dann gibt es eine Exception, oder besser noch, die Funktion wird bei einem leeren Vektor gar nicht erst ausgeführt. Ich habe `if(feld.size() == 0) return;` als erste Zeile eingefügt. Aber dann gab es einen weiteren Fehler!
Paula: Welchen?
Otto: Die Kombination (1, 1, 0) ergab (1, 0, 1) statt (0, 1, 1)! Damit ist nachgewiesen, dass diese Datenkonstellation zum Scheitern führt, die getestete Funktion also fehlerhaft ist. Der Algorithmus verwendet anscheinend die Überlegung: Nur eine Vertauschung ändert `temp`, weswegen es als Indikator für das Ende der Sortierung genommen werden kann. Der Fehler: Dies gilt nicht, wenn nach der letzten Vertauschung `temp` genau den Wert hat, den auch `feld[0]` hat. Die Behauptung im Quellcode (Seite 180) »`false` = keine Vertauschung mehr« ist also falsch.
Paula: Gutes Ergebnis. Gibt es noch weitere fehlerhafte Datenkonstellationen?
Otto: Möglicherweise. Sie werden im Test nicht sichtbar, weil ich `ASSERT_EQ` statt `EXPECT_EQ` geschrieben habe. Bei `ASSERT_EQ` wird der Test bei einem Fehler sofort abgebrochen.
Paula: Was meinst du? Wie könnte der Fehler in der Funktion korrigiert werden?

Otto: Durch Einfügen einer booleschen Variablen, die feststellt, ob tatsächlich eine Vertauschung stattgefunden hat. Das hätte den weiteren Vorteil, dass der Zugriff auf `feld[0]` entfiele. Etwa so:

```
void fastbubblesort1(std::vector<int>& feld) {
   bool vertauscht;
   do {
      vertauscht = false;
      for(std::size_t j = 1; j < feld.size(); j++) {
         if(feld[j] < feld[j-1]) {          // vertauschen
            int temp  = feld[j-1];
            feld[j-1] = feld[j];
            feld[j]   = temp;
            vertauscht = true;
         }
      }
   } while(vertauscht);                 // false = keine Vertauschung mehr
}
```

Listing 6.5: Der korrigierte BubbleSort

Paula: Ja gut, teste das doch mal. Bis nachher – jetzt habe ich gerade keine Zeit mehr.

[Später ...]

Paula: Hi Otto! Wie ist der Test ausgegangen?
Otto: Sehr gut! Alle Test waren erfolgreich!
Paula: Herzlichen Glückwunsch! Die Funktion ist jedoch nur für `int`-Vektoren geeignet. Ist das nicht ein wenig beschränkt? Wie könntest du sie für beliebige Typen verallgemeinern?
Otto: Ist doch klar. Einfach nur die Schnittstelle gegen ein Template austauschen, nämlich so:

```
template<typename T>                                    // neu
void fastbubblesort1(std::vector<T>& feld) {            // neu
   bool vertauscht;
   do {
      vertauscht = false;
      for(std::size_t j = 1; j < feld.size(); j++) {
         if(feld[j] < feld[j-1]) {          // vertauschen
            int temp  = feld[j-1];
            feld[j-1] = feld[j];
            feld[j]   = temp;
            vertauscht = true;
         }
```

```
      }
   } while(vertauscht);                    // false = keine Vertauschung mehr
}
```

Listing 6.6: Ottos Versuch der generischen Programmierung

Paula: Gut. Dann teste doch mal mit Vektoren, die `double`-Werte enthalten. Sag Bescheid, wenn du fertig bist.
Otto: Okay, bis gleich.

[Eine halbe Stunde später ...]

Otto: Hallo Paula! Es funktioniert alles weiterhin! Ich habe einfach die Variablen `i`, `j` und `k` des Tests in `double`-Werte umgewandelt. Zwar ist ein Vergleich von `double`-Werten untereinander kritisch, wenn sie auf verschiedenen Wegen berechnet wurden, hier ist das aber nicht der Fall, weil es nur Vertauschungen gibt. Wenn nicht, würde der Test auch deswegen scheitern.
Paula: Tja, damit hast du recht! Aber meinst du, dass du damit wirklich `double`-Eigenschaften geprüft hast? Meine Erfahrung sagt mir, dass man da nochmal genau hinschauen muss! Teste doch mal mit `double`-Werten, deren Wert keine ganze Zahl ist, zum Beispiel, indem du auf jedes Element im Vektor 0,1 addierst. Oder indem du nur Zahlen nimmst, die zwischen 0 und 1 liegen.

[Nach einer halben Stunde ...]

Otto: Paula, du hattest recht! Die Tests schlagen fehl! Woran kann das nur liegen? Es wird alles korrekt ohne Fehler und Warnungen compiliert, ich habe sogar mit `-Wall` die höchste Warnstufe eingeschaltet! Und mit `int`-Werten funktioniert alles perfekt.
Paula: Geh' das Programm sorgfältig durch! Und wenn du nicht weiter weißt, lass' dir die Variable `temp` ausgeben und vergleiche sie mit ihrem Sollwert.

[Nach einer Stunde ...]

Otto: Paula, ich hab's! Es ist oberpeinlich! Ich hätte nicht nur die Schnittstelle gegen ein Template austauschen, sondern auch auf die Verwendung des generischen Typs achten müssen. Es muss `T temp = feld[j-1]` heißen, nicht `int temp = feld[j-1]`! Leider ist hier die automatische Umwandlung von `double` nach `int` ohne Warnung erlaubt. *So* typsicher ist C++ eben doch nicht.
Paula: Immerhin, wieder ein Fehler gefunden! Auch wenn du jetzt nicht sehr zufrieden scheinst: Die Ausdauer hat sich gelohnt. Du weißt jetzt, dass allein der Unit Test nicht reicht, sondern dass man auch die Eigenheiten der Programmiersprache kennen muss. Welche Erkenntnis hast du heute hauptsächlich gewonnen?

Otto: Ich wundere mich, dass bei einem so kleinen Programm so viele Fehler zutage treten – und zum Sortieren nehme ich nur noch die Bibliotheksfunktion `sort()`! Tschüss bis morgen!

Was können wir aus diesem Dialog lernen? Erfahrung ist wichtig! Aber wie kann ich mir als Tester oder als testender Entwickler Erfahrung aneignen, wenn nicht durch jahrelanges Testen?

Es gibt auch hier eine ganze Reihe von Ansätzen, die im Folgenden vorgestellt werden. Allen gemein ist, dass es keine vorab definierbaren und nachprüfbaren Kriterien zur Beendigung des Testens gibt. Meist ist es die zur Verfügung stehende Zeit, die die Beendigung der Testaktivitäten erzwingt. Darüber hinaus gibt es bei den erfahrungsbasierten Verfahren kein strukturiertes Vorgehen, wie die einzelnen Testfällen zu erstellen sind.

> Die erfahrungsbasierten Testansätze sind als ergänzende und nicht als alleinige Testaktivitäten anzusehen.

6.1 Exploratives Testen

Exploratives Testen ist ein erfahrungsbasierter Testansatz, der in den letzten Jahren viel Verbreitung und Anwendung – besonders im agilen Umfeld – gefunden hat. Wie der Name bereits andeutet (lat. explorare = erforschen), geht es darum, die zu testende Software zu *erkunden*. Wie arbeitet die Software, wie behandelt sie schwierige, komplexe Eingaben, wie geht sie mit eher einfachen Konstellationen um – das sind beispielsweise Fragestellungen, die beim explorativen Testen geklärt werden sollen.

Wann ist der Einsatz sinnvoll?

Für den Einsatz des explorativen Testens gibt es keine Einschränkung, es kann beim Testen jedes Programms verwendet werden. Diese Aussage trifft ebenso auf alle anderen erfahrungsbasierten Testansätze zu. Explorativ testen ist eine sinnvolle Ergänzung zu den anderen Testverfahren, die wir in den vorangegangenen Kapiteln vorgestellt haben. Je besser sich der explorative Tester mit dem Testobjekt und im Testen allgemein auskennt, je mehr Erfahrung und intellektuelle Neugierde – Forschergeist – er hat, desto erfolgreicher wird er explorativ testen! In folgenden Situationen ist exploratives Testen besonders geeignet:

- Es ist wenig Zeit für das Testen vorgesehen oder vorhanden.
- Die korrekte Arbeitsweise einer geänderten Funktion soll nachgewiesen werden.
- Für eine neuprogrammierte Funktion ist kurzfristig Feedback erforderlich.
- Risiken sollen durch Testfälle erkannt werden.
- Eine fehlerhafte Situation soll weiter analysiert und eingekreist werden.

Aus der Aufzählung wird ein Vorteil des explorativen Testens ersichtlich: Mit explorativem Testen kann sofort begonnen werden, da keine Vorbereitung erforderlich ist.

Grundidee

> Testfälle werden – im Gegensatz zu den bisher vorgestellten Verfahren – nicht vorab spezifiziert. Vielmehr ist die Kernidee des explorativen Testens, aus der Durchführung von Testfällen zu lernen und somit zu neuen Ideen für weitere Testfälle zu gelangen.

Jeder kennt die folgende Situation: Man hat einen Testfall ausgeführt und bekommt dadurch eine Idee, mit welchem weiteren Test es sich lohnt, die Software zu überprüfen. Dieses Vorgehen wird beim explorativen Testen zur Prämisse erhoben. Der Tester steuert den Entwurf der Tests aktiv, indem er die Informationen, die er während des Testens erhält, zum Entwurf weiterer Tests verwendet.

Auch ist es nicht zwingend erforderlich, bei jedem Testfall das erwartete Ergebnis vorab zu definieren. *Erkundet* werden kann somit auch das Ergebnis der Testfälle. Damit wird ein weiterer Vorteil des Ansatzes deutlich: Wenn die Spezifikation des Testobjekts veraltet, lückenhaft oder gar nicht existent ist, kann eine Prüfung des Testobjekts explorativ vorgenommen werden.[1]

Testfallerstellung, Testdurchführung und die Ideen für weitere Tests sind beim explorativen Testen nicht strikt getrennt, sondern gehen ineinander über. In welche Richtung sich die Tests dabei entwickeln, obliegt dem Tester. Diese Freiheit kann leicht zu einem Verzetteln führen. Daher ist beim explorativen Testen ein Testziel zu definieren, das als Test-Charta bezeichnet wird. Eine Charta beschreibt das Ziel oder die Mission und auch den Weg, um das Ziel zu erreichen. Sie beschreibt keine einzelnen Testfälle.

[1] Beim Entwicklertest kann es diese Situation nicht geben, da ohne genaue Spezifikation keine Implementierung vorgenommen werden kann.

Die Charta muss so verfasst sein, dass dem *Erforschen* noch genug Freiraum bleibt. Es empfiehlt sich, die Charta schriftlich zu fixieren, also eine Art »Testauftrag« zu verfassen.

Im Buch von Elisabeth Hendrickson »Explore It!« [Hendrickson 14] wird eine Vorlage für Test-Chartas vorgestellt, die im Folgenden kurz erörtert wird. Eine Charta besteht im Wesentlichen aus drei Teilen:

1. Wo soll erkundet werden?
2. Welche Hilfsmittel stehen zur Verfügung?
3. Welche Erkenntnisse sollen gewonnen werden?

Wo geforscht werden soll, kann sich sowohl aus den Anforderungen als auch aus dem Programmcode ergeben. So kann eine Anforderung, bei deren Nichterfüllung ein hohes Risiko besteht, explorativ getestet werden. Ein Beispiel wäre eine mögliche Umgehung eines Login-Vorgangs. Eine C++-Funktion oder Klasse kann ebenso Gegenstand des Testens sein. Als Hilfsmittel gelten sowohl unterstützende Werkzeuge als auch Techniken oder die Strategie, die genutzt werden soll. Ebenso können vorhandene Datensätze und Konfigurationen hier vermerkt bzw. festgelegt werden. Das eigentliche Testziel, was untersucht werden soll, wird als dritter Punkt vermerkt. Die Prüfung der Benutzbarkeit, Sicherheit, Geschwindigkeit oder der Datenvolumina sind Beispiele für solche Testziele.

Für die Durchführung einer Test-Charta steht ein festes Zeitkontingent zur Verfügung. In der Regel sind es 90 bis 120 Minuten. Ein Arbeiten an der Charta soll dabei am Stück und störungs- und unterbrechungsfrei erfolgen. Die Charta kann von einer oder zwei Personen bearbeitet werden.

Die Dokumentation beim explorativen Testen variiert stark. Eine zusammenfassende Beschreibung der aufgedeckten Fehler kann ebenso als Dokumentation dienen wie die Protokollierung aller ausgeführten Testfälle. Art und Umfang der Dokumentation ist vorab festzulegen.

Testendekriterium

Bei den bisher vorgestellten Testverfahren kann vorab festgelegt werden, wie intensiv getestet werden soll. Messbare Kriterien dienen zur Bestimmung und zum Nachweis der Testintensität und als Kriterium zur Beendigung der Testaktivitäten für das einzelne Testverfahren. Bei den erfahrungsbasierten Testansätzen lassen sich solche Kriterien nicht festlegen.

Beim explorativen Testen ist der Test beendet, wenn die Test-Chartas abgearbeitet sind, oder wenn die zur Verfügung stehende Zeit aufgebraucht ist. Dies ist kein im eigentlichen Sinne *messbares* Kriterium, das über die Intensität der Tests eine Aussage zulässt. In 90 Minuten können viele sinnvolle Testfälle durchgeführt und viele Fehler gefunden werden, es kann aber

auch das Gegenteil sein, wenn der Tester über zu wenig Erfahrung verfügt und wenig kreativ ist!

Bewertung

Auf Erfahrung zu verzichten, ist sicherlich keine gute Idee. Exploratives Testen findet Fehler, die bei den strukturierten Verfahren nicht ohne Weiteres auffindbar sind. Es untersucht – bei entsprechender Wahl der Charta – kritische und risikoreiche Stellen im Testobjekt, bei denen sich ein intensiver Test lohnt. Durch die Vorgehensweise des flexiblen tiefen Einsteigens in einen *Problembereich* können so ganz gezielt Fehler oder das korrekte Arbeiten des Testobjekts nachgewiesen werden. Dieser Vorteil ist aber auch ein Nachteil, da die Prüfung nur punktuell erfolgt. Für den Nachweis des korrekten Arbeitens des Testobjekts *in der Breite* sind die strukturierten Testverfahren durchzuführen.

Bezug zu anderen Testverfahren

Einen direkten Bezug zu anderen Testverfahren gibt es nicht. Im Buch von Elisabeth Hendrickson »Explore It!« [Hendrickson 14] werden zwei Seiten des Testens beschrieben. Die eine dient zum Nachweis, dass das Testobjekt sich so wie spezifiziert verhält, hierzu sind die Testentwurfsverfahren einzusetzen. Die andere Seite des Testens prüft, ob weitere Risiken bestehen, hier ist exploratives Testen der geeignete Ansatz.

Hinweise für die Praxis

Grundlage des explorativen Testens sind die Test-Chartas, die auch eine gewisse Strukturierung des Ansatzes bilden. Zum Testobjekt *passende* Chartas zu definieren, die in der zur Verfügung stehenden Zeit sinnvoll und ausreichend bearbeitet werden können, ist keine einfache Aufgabe. Eine immer wieder aktualisierte Aufstellung von erfolgreich genutzten Test-Chartas ist für die Praxis ein nützliches Nachschlagewerk. Ihr können Ideen für Test-Chartas entnommen werden und die Liste bietet auch unerfahrenen Entwicklern einen schnellen Einstieg ins explorative Testen.

Beispiele für Test-Chartas

Wir greifen auf vorhandene Beispiele im Buch zurück und formulieren einige Test-Chartas, um einen Eindruck zu geben, welchen Umfang und welche Ziele Test-Chartas haben können. Wir nennen dabei eher die Fragestellung bzw. den Ausgangspunkt, die bzw. der in der Test-Charta behandelt wer-

den soll. Die Einteilung der Charta in drei Punkte (s.o.) wird dabei nicht vorgenommen, um die Beispiele einfach zu halten.

- Der Praktikant Otto vom Anfang dieses Kapitels hat die Aufgabe, ein Sortierprogramm für einen Vektor zu testen. Paula fordert ihn auf, herauszubekommen, ob der Vektor nach der Sortierung dieselben Elemente enthält. Dieser Testauftrag als Charta formuliert lautet:

  Erkunde[2] *im Programm, ob die einzelnen Elemente des Vektors durch die Sortierung unverändert bleiben.*
- In Abschnitt 4.3.4 zum Äquivalenzklassentest haben wir ein Beispiel zur Berechnung von zu gewährendem Rabatt in Abhängigkeit von Einkaufssumme und Uhrzeit vorgestellt. Dabei haben wir auch das Datum überprüft und uns dabei um die korrekte Berechnung des Schaltjahrs gekümmert. Die Test-Charta lautet:

  Erkunde, ob die Berechnung des Schaltjahrs korrekt erfolgt. Hilfsmittel: Bitte die Berechnungsvorschrift für ein Schaltjahr vorab klären, da sie einige Sonderfälle enthält.

  Eine ergänzende Test-Charta für das Beispiel ist folgende:

  Erkunde, ob die Umstellung von Sommer- auf Winterzeit (und umgekehrt) Auswirkungen auf die Rabattberechnung und die Ergebnisse hat. Hinweis: Diese Überlegung haben wir bei den bisher spezifizierten Testfällen nicht berücksichtigt. Der Testauftrag verdeutlicht, dass beim explorativen Test weitere Aspekte in den Fokus treten können.
- Im Beispiel zur Klassifikationsbaummethode (siehe Abschnitt 4.5.1) haben wir eine Schnittstelle zu einem Statistikprogramm besprochen. Beim Studium wurde zwischen Technischem Studium, Geisteswissenschaftlichem Studium und sonstigem Studium unterschieden. Eine differenziertere Statistik ist erforderlich und wurde entsprechend umgesetzt. Das Studienfach Medizin ist hinzugekommen und muss ab sofort berücksichtigt werden. Um die Ergänzung kurzfristig zu prüfen, ist folgende Test-Charta durchzuführen:

  Erkunde, ob die Ergänzung um das Studienfach Medizin korrekt bearbeitet und an die Schnittstelle weitergegeben wird.

  Sicherlich ist auch zu testen, ob die bisherige Bearbeitung der anderen Studienfächer weiterhin korrekt vorgenommen wird. Diese Prüfung kann aber nach Anpassung der vorliegenden Testfälle, die mit der Klassifikationsbaummethode erstellt wurden, durchgeführt werden. Ein »Erforschen« oder »Erkunden« ist hier nicht erforderlich.
- Im Beispiel des Sportvereins (siehe Abschnitt 4.5.2) wurden die Sportarten Tischtennis, Turnen, Volleyball, Basketball, Handball und Fitnesstraining angeboten. Es ist folgende Veränderung eingetreten: Wegen der

[2] Eigentlich beginnen Test-Chartas mit »Erforsche ...«, wir halten aber »Erkunde« für den passenderen Begriff, da Forschung doch mehr umfasst.

geringen Nachfrage wird Turnen nicht mehr angeboten. Auch in dem Fall sind Test-Charta und die Wiederholung von Testfällen nach der programmtechnischen Umsetzung der geänderten Anforderung durchzuführen:

Erkunde, ob eine Auswahl der Sportart Turnen möglich ist.

Es ist auch zu testen, ob die bisherige Bearbeitung der anderen Sportarten weiterhin korrekt erfolgt. Zu wiederholen (ggf. nach Anpassung) sind die Testfälle, die mit der Klassifikationsbaummethode und dem kombinatorischen Testen erstellt worden sind.

6.2 Freies Testen

Freies Testen bedeutet: einfach mal drauflos ausprobieren! Hier ist Kreativität und Fantasie gefragt, unter Berücksichtigung von Erfahrung. Beim explorativen Test ist durch die Test-Chartas und die zeitliche Beschränkung eher noch ein festgelegter Ablauf vorgegeben. Ein Prozess oder ein strukturiertes Vorgehen ist bei den folgenden freien Ansätzen nicht unbedingt vorhanden. Eine kleine Auswahl von Ansätzen zum freien Testen wird in diesem Kapitel zusammengefasst und kurz dargestellt. Die Ansätze unterscheiden sich hauptsächlich durch die jeweilige Zielsetzung bzw. durch die zugrunde liegende Idee. Die folgenden Ausführungen gelten für alle Ansätze zum freien Testen gleichermaßen.

Wann ist der Einsatz sinnvoll?

Für freies Testen gibt es keine Einschränkungen oder Voraussetzungen, die einzuhalten wären. Die Ansätze können stets angewendet werden.

Grundidee

Die Grundidee aller Ansätze zum freien Testen ist oben bereits genannt: mit Kreativität, Fantasie und Erfahrung Testfälle überlegen. Deren Ausführung ähnelt dabei dem explorativen Ansatz: Die Testfälle werden meist einzeln überlegt und ausgeführt. Danach erfolgt die Ausführung weiterer Testfälle.

Testendekriterium

Wenn keine weiteren Ideen für Testfälle vorhanden sind, wird das Testen beendet – das kann man nicht als messbares Kriterium bezeichnen. Falls Checklisten oder Heuristiken eingesetzt werden (s.u.), kann die Abarbeitung solcher Listen als Kriterium herangezogen werden, wobei aber nicht

jeder Punkt der Checkliste oder jede Heuristik auch genutzt werden muss, sondern nur die Auswahl, die für das Testobjekt passend und sinnvoll ist.

Bewertung

Die Bewertung ist ähnlich wie beim explorativen Testen: Freies Testen ist eine sinnvolle Ergänzung. Auch wenn die Testfälle nicht systematisch hergeleitet werden, soll freies Testen genutzt werden. Es werden so Testfälle erstellt, die sich mit systematischer Herleitung nicht unbedingt ergeben. Freies Testen soll aber nicht die einzige Testaktivität sein!

Bezug zu anderen Testverfahren

Es gibt keinen direkten Bezug zu anderen Verfahren. Freies Testen ähnelt aber sehr dem explorativen Testen. Neben dem Einsatz von Test-Chartas und der zeitlichen Beschränkung gibt es so gut wie keine Unterschiede.

Hinweise für die Praxis

Einfach mal »drauflosprobieren« kann erfolgreich sein, muss es aber nicht. Und ganz ohne Erfahrung werden die Erfolgsaussichten noch geringer. Auch hier helfen wieder Listen, die erfolgreiche Testideen beinhalten und als Grundlage dienen können. Oft werden solche Listen auch als Heuristiken bezeichnet. Sie dienen zur Gewinnung neuer Erkenntnisse bzw. Ideen für weitere Testfälle. Im Internet[3] und in Büchern sind solche Heuristiken zu finden (zum Beispiel in [Hendrickson 14, Anhang 2: Testheuristiken – Spickzettel]).

Wir empfehlen, diese Listen als nützliche Nachschlagewerke und Ideensammlungen zu nutzen. Darüber hinaus raten wir, die Listen um eigene Erfahrungen zu ergänzen und für das Projekt oder das Testobjekt unpassende Ideen aus den Listen zu streichen. Dadurch bleibt eine solche Sammlung von Heuristiken aktuell und von der Länge her überschaubar. Möglicherweise sind auch unterschiedliche Listen für unterschiedliche Testaktivitäten praktikabel und sinnvoll.

Unterschiedliche Ansätze

Im Folgenden werden einige Ansätze kurz vorgestellt. Es ist eine relativ willkürliche Auswahl aus einer ganzen Reihe von Heuristiken zum freien Testen. Die Auswahl ist nicht als Empfehlung von unbedingt einzusetzenden Heuristiken zu verstehen.

[3] Siehe z.B. *http://www.it-agile.de/fileadmin/docs/Heuristic-Sheet_deutsch.pdf*.

Fehlererwartungsmethode

Wie der Name schon andeutet, ist der Ansatz hier »anders herum«. Beim Testen wird üblicherweise von den Anforderungen ausgegangen und es werden entsprechende Testfälle zur Überprüfung der korrekten Umsetzung entworfen. Bei der Fehlererwartungsmethode wird überlegt, welche Fehler auftreten können. Es wird abgeschätzt, wo bei der Programmierung falsche Entscheidungen getroffen oder Dinge möglicherweise übersehen wurden, die zu Fehlern im Programm führen würden. Diese Überlegungen sind die Grundlage zum Erstellen der Testfälle nach der Fehlererwartungsmethode.

Aus der Erfahrung kennt man bestimmte Situationen, die in der Vergangenheit häufig zu Fehlern geführt haben. Das Paradebeispiel ist die Null. Wenn bei den bisher durchgeführten Testfällen die Null als Parameterwert noch nicht verwendet wurde, ist dies spätestens hier nachzuholen. Die Null ist immer ein spannendes Testdatum, bei dem häufig Fehler auftreten.

Nehmen wir das einleitende Beispiel aus diesem Kapitel: Test eines Sortieralgorithmus. Fehler können bei folgenden Eingaben auftreten:

- Eine bereits sortierte Liste
- Eine umgekehrt sortierte Liste
- Eine Liste, die aus nur gleichen Elementen besteht
- Eine Liste, die nur ein Element umfasst
- Eine Liste, die kein Element enthält
- ...

Auf einige dieser Ideen ist unser Praktikant Otto bereits gekommen. Auch bei diesem Ansatz des erfahrungsbasierten freien Testens ist eine Liste von denkbaren Fehlern sehr hilfreich. Dadurch kann die Erfahrung im Team verteilt werden. Fehlererwartungen aus anderen Projekten werden berücksichtigt, wenn die Liste projektübergreifend angelegt ist und fortlaufend aktualisiert wird.

DAU-Test

Vielleicht haben einige von Ihnen diese Situation bereits erlebt: Die Software ist bereit zur Abnahme und wird dem Team vorgestellt. Die Eingabemaske ist aufgerufen und Werte sind einzugeben. Ein Kollege setzt sich auf die Tastatur und ist gespannt, was passiert! Eine solche Eingabemöglichkeit ist in den Anforderungen mit Sicherheit nicht vorgesehen. Trotzdem wäre es doch gut, wenn die Software auch mit einer solchen doch recht exotischen Eingabesituation klarkommt, oder?

Die Bezeichnung für ein solches Vorgehen, wenn man dem einen Namen geben will, lautet DAU-Test. DAU steht für »dümmster anzunehmender User«. Das englische Pendant ist »Random Monkey Testing«.

Beim DAU-Test geht es darum, nahezu beliebige zufällige Eingaben vorzunehmen, ohne Sinngehalt und ohne Verwendung einer Systematik. Einfach mal wild drauflos eingeben! Listen mit möglichen Eingaben, die sich in der Vergangenheit bewährt haben, ergeben hier keinen Sinn. Geprüft wird in aller Regel die Robustheit der Software. Aber bitte diesen Aspekt beim Testen nicht völlig vernachlässigen: Wenn der Nutzer aus Versehen sinnlose Eingaben vornimmt – aus welchem Grund auch immer –, ist er sicherlich nicht erfreut, wenn die Software auf solche Eingaben nicht adäquat reagiert und möglicherweise sogar abstürzt.

Seifenopern-Test

Einen ähnlichen Ansatz verfolgt der Seifenopern-Test (Soap Opera Testing). Der Begriff ist von den täglichen oder wöchentlich ausgestrahlten Serien abgeleitet, mit denen uns die Fernsehanstalten versorgen.

In den einzelnen Folgen einer solchen Seifenoper wird das (vermeintlich) alltägliche Leben dargestellt. Allerdings passieren in einer einzigen Folge Ereignisse, die im realen Leben – wenn sie überhaupt alle vorkommen – über einen viel längeren Zeitraum verteilt auftreten. Kennzeichen der Seifenopern ist ferner, dass es unterschiedliche Handlungsstränge gibt, die untereinander in Beziehung stehen.

Was bedeutet dies für den Seifenopern-Test? Hier geht es weniger um die Eingabe von einzelnen Daten, sondern es sind Szenarien bzw. Abläufe zu überlegen, die dann zu Testfällen – genauer einer Reihe von nacheinander auszuführenden Testfällen – führen. Die zu überlegenden Tests sollen realitätsnah sein, allerdings bestückt mit unvorhergesehenen Ereignissen und komplexen Abhängigkeiten.

Wie könnte nun ein solches Szenario aussehen? Hier ein Beispiel einer Autovermietung: Herr Müller leiht sich für drei Tage einen Mietwagen bei einer Autovermietung. Am zweiten Tag stellt er fest, dass er das Auto noch länger benötigt und teilt dies der Autovermietung mit. Am Folgetag meldet er das Auto als gestohlen und verlangt von der Autovermietung umgehend Ersatz. Er hatte seine Fahrt in Bremen begonnen und ist jetzt in Heidelberg. Er benötigt dort vor Ort das Fahrzeug. Dieses wird ihm auch von der Autovermietung zugesichert, da er ein guter Stammkunde ist. Da die Autovermietung in Heidelberg kein passendes Fahrzeug anbieten kann, wird ein Fahrzeug von der Frankfurter Filiale nach Heidelberg gebracht. Am nächsten Tag stellt Herr Müller fest, dass der erste Leihwagen gar nicht gestohlen ist, er hatte nur vergessen, wo er es geparkt hatte – der Kneipenbesuch war wohl doch nicht folgenlos ...

Dieser Fall wird im realen Leben wohl nicht oder doch höchst selten vorkommen. Die einzelnen Aspekte (Verlängerung, Ersatz, Diebstahl) gehören zum Alltag einer Autovermietung und sind durch Tests auch bereits einzeln

geprüft. Beim Seifenopern-Test sollen nun genau solche »irrwitzigen« Szenarien überlegt und getestet werden. Idee dahinter: Durch die Kombination und Konstellation der Ereignisse können Fehler auftreten, die beim Test der einzelnen Ereignisse nicht vorkommen.

Der Seifenopern-Test beruht auf der Fantasie und Kreativität des Testers. Aber es macht ja vielleicht auch Spaß, sich solche irrwitzigen Konstellationen und Abläufe zu überlegen und zu testen. Und manchmal hält das wirkliche Leben noch irrwitzigere Situationen für uns bereit.

Der Seifenopern-Test umfasst Abfolgen und Kombinationen von Einzelereignissen. Diese sind meist nicht Gegenstand des Entwicklertests, da das Testobjekt nicht all diese Konstellationen umfassen wird. Trotzdem haben wir den Seifenopern-Test hier aufgeführt, um zu zeigen, welche Ideen beim freien Testen umgesetzt werden können.

> Freies Testen – mit Kreativität, Fantasie und Erfahrung Testfälle überlegen – ist ein sinnvolles Vorgehen, um weitere Testfälle durchzuführen. *Mal sehen, was passiert, wenn ...*, ist die Fragestellung, die den Tester beim freien Testen beschäftigt.

7 Softwareteststandard ISO 29119

Wir haben eine Vielzahl von Ansätzen zur Erstellung von Testfällen in den vorherigen Kapiteln vorgestellt. Vielleicht werden Sie sich schon gefragt haben, ob es keine Standards gibt, an denen man sich orientieren kann und sollte. In Testerkreisen ist der »Certified Tester« recht verbreitet. Dies ist aber kein Standard im eigentlichen Sinne, sondern in den Certified-Tester-Lehrplänen werden »Best Practices« formuliert.[1]

Im Testbereich gibt es eine ganze Reihe von Standards. Zu ihnen gehören unter anderen: IEEE 829 Test Documentation, IEEE 1008 Unit Testing und die beiden britischen Standards BS 7925-1 Glossary of Software Testing Terms und BS 7925-2 Software Component Testing Standard. Mit dem aktuellen Standard »ISO 29119 Software Testing«[2] wird eine Vereinheitlichung der verschiedenen Standards angestrebt, sodass dieser als alleiniger Standard für den Test von Software ausreicht.

Der Standard besteht aus mehreren Teilen. Derzeit sind veröffentlicht:

- ISO 29119-1 – Concepts and Definitions
- ISO 29119-2 – Test Processes
- ISO 29119-3 – Test Documentation
- ISO 29119-4 – Test Techniques[3]

Themen, die zu weiteren Teilen des Standards in Zukunft führen werden und bei der Bearbeitung des Standards identifiziert wurden, sind folgende:

- Schlüsselwortbasierter Test
- Prozessbewertung und -verbesserung
- Statischer Test
- Modellbasierter Test

[1] Näheres zum Certified Tester finden Sie unter *http://www.german-testing-board.info/* und international unter *http://www.istqb.org/*.

[2] Die vollständige Bezeichnung des Standards ist ISO/ICE/IEEE 29119 Software Testing. Wegen der besseren Lesbarkeit verwenden wir »ISO 29119«, ohne dabei die beiden anderen Organisationen diskreditieren zu wollen.

[3] Die ersten drei Teile wurden im September 2013 und der vierte Teil im Dezember 2015 publiziert.

Unser Buch beschäftigt sich mit Testverfahren, wir werden uns daher Teil 4 näher ansehen. Für ausführliche Informationen zum Standard verweisen wir auf das Buch von Matthias Daigl und Rolf Glunz »ISO 29119 – Die Softwaretest-Normen verstehen und anwenden« [Daigl & Glunz 16].

> Wir stellen den Standard vor, um dem Leser eine Einordnung der im Buch präsentierten Testverfahren an die Hand zu geben, und weil der Standard nach unserer Einschätzung praxisnahe und alltagstaugliche Testverfahren beinhaltet.

7.1 Testverfahren nach ISO 29119

Der Standard fasst den Begriff »Testverfahren« sehr eng:

Aktivitäten, Konzepte, Prozesse und Muster, die verwendet werden, um ein Testmodell zu erstellen, das wiederum verwendet wird, um nacheinander Testbedingungen für ein Testobjekt, dann die entsprechenden Testüberdeckungselemente und schließlich Testfälle abzuleiten oder auszuwählen (zitiert nach [Daigl & Glunz 16, S. 114])[4].

In unserem Buch sind wir nicht auf ein Testmodell eingegangen, sondern haben direkt die jeweiligen Testverfahren mit ihren Überdeckungen vorgestellt und das Vorgehen, um daraus Testfälle abzuleiten. Wir halten ein Testmodell nicht für zwingend erforderlich, um die Testverfahren in der Praxis anzuwenden. Die engere Sichtweise des Standards hat zur Folge, dass zum Beispiel exploratives Testen nicht zu den Testverfahren zählt, da kein grundlegendes Modell genutzt wird, sondern die Testfälle direkt erstellt werden.

Im Standard werden drei Kategorien von Testverfahren unterschieden: spezifikationsbasierte, strukturbasierte und erfahrungsbasierte. Sehen wir uns diese im Einzelnen an.

7.1.1 Spezifikationsbasierte Testverfahren

Folgende spezifikationsbasierte Testverfahren werden im Standard behandelt (in Klammern die englischen Fachbegriffe):

[4] *Hinweis*: Der Begriff Testbedingung kommt vom englischen Fachbegriff *test condition*. Eine möglicherweise passendere Übersetzung wäre Testaspekt oder Testgegenstand. Testbedingung ist als Begriff bereits sehr verbreitet, da er auch bei den Certified-Tester-Lehrplänen verwendet wird.

- Äquivalenzklassentest (Equivalence Partitioning)
- Klassifikationsbaummethode (Classification Tree Method)
- Grenzwertanalyse (Boundary Value Analysis)
- Syntaxtest (Syntax Testing)
- Kombinatorisches Testen (Combinatorial Test Design Techniques)
- Entscheidungstabellentest (Decision Table Testing)
- Ursache-Wirkungs-Graph-Analyse (Cause-Effect Graphing)
- Zustandsbasierter Test (State Transition Testing)
- Szenariotest (Scenario Testing)
- Zufallstest (Random Testing)

Bis auf die Ursache-Wirkungs-Graph-Analyse und den Szenariotest sind alle Testverfahren des Standards im Buch ausführlich beschrieben. Um das Bild zu vervollständigen, werden die fehlenden Verfahren kurz vorgestellt.

Ursache-Wirkungs-Graph-Analyse

Glenford J. Myers hat Ende der 1970er-Jahre das Verfahren vorgestellt. Grundlage ist eine grafische Darstellung, in der Ursachen und Wirkungen miteinander in Beziehungen gesetzt werden. Ursachen sind dabei die Eingaben oder Vorbedingungen, die ein unterschiedliches Verhalten bewirken können und zu unterschiedlichen Ausgaben bzw. Wirkungen führen. Ursachen und Wirkungen werden im Graphen gegenübergestellt und miteinander durch grafische Elemente (Linien) in Beziehung gesetzt. Sind beispielsweise zwei bestimmte Eingaben zwingend erforderlich, um eine Wirkung zu erzielen, dann sind im Graphen von jeder Ursache eine Verbindung zur Wirkung zu zeichnen und es besteht eine »UND«-Verbindung zwischen den beiden Beziehungen, die ebenfalls im Graphen durch ein entsprechendes Symbol darzustellen ist. »ODER«, »NEGATION« und »IMPLIKATION« sind weitere Verknüpfungsoptionen.

Die Erstellung der Graphen kann sehr aufwendig sein, hat aber den Vorteil, dass eine genaue Grundlage für die weiteren Überlegungen vorliegt. Wenn viele Ursachen, Wirkungen und unterschiedliche Beziehungen vorhanden sind, wird der Graph allerdings unübersichtlich.

Der Ursache-Wirkungs-Graph wird anschließend in eine Entscheidungstabelle umgewandelt und zur Ermittlung von Testfällen wird dann wie beim Entscheidungstabellentest vorgegangen.

Der Einsatz der Ursache-Wirkungs-Graph-Analyse beim Entwicklertest ist meist nicht sinnvoll, da hier die Wechselwirkungen zwischen den Eingaben und den Ausgaben überschaubar sind und direkt in eine Entscheidungstabelle übernommen werden können. Eine vorherige grafische Aufbereitung der Abhängigkeiten ist nach unserer Meinung beim Komponententest nicht

erforderlich. Deshalb haben wir die Ursache-Wirkungs-Graph-Analyse nicht näher erörtert.

Szenariotest

Im Standard werden unterschiedliche Varianten von Testansätzen unter dem Begriff Szenariotest zusammengefasst:

- Anwendungsfallbasierter Test (Use Case Testing)
- Geschäftsprozessbasierter Test
- Transaktionsbasierter Test

All diese Testansätze gehören nicht zum Bereich des Entwicklertests. Sie spielen beim End-to-End-Test oder auch beim Systemintegrationstest eine Rolle.

Ergänzung zum kombinatorischen Testen

Im Standard ISO 29119 wird das kombinatorische Testen in vier Variationsmöglichkeiten unterteilt:

- Vollständige Kombinatorik (All Combination Testing)
- Paarweises Testen (Pair-wise Testing)
- Einfache Kombinatorik (Each Choice Testing)
- Basiskombinatorik (Basic Choice Testing)

Die ersten drei Verfahren sind in Abschnitt 4.6 zum kombinatorischen Testen beschrieben. Zur Erinnerung: Die vollständige Kombinatorik verlangt, dass alle möglichen Kombinationen aller Parameterwerte beim Testen berücksichtigt werden. Beim paarweisen Testen sind alle möglichen Kombinationen zwischen jeweils zwei Parametern mit Testfällen zu prüfen. Dieses Vorgehen sehen wir als »lean« an und empfehlen den Einsatz. Einfache Kombinatorik verlangt nur, dass jeder Wert jedes Parameters in mindestens einem Testfall vorkommt, ohne jede Berücksichtigung von parameterübergreifenden Kombinationen.

Auf die Basiskombinatorik sind wir bisher nicht eingegangen. Das Vorgehen ist wie folgt: Es wird zunächst für jeden Parameter ein Wert aus seinen möglichen Werten festgelegt, eine Art Basiswert. Dies kann der in der Anwendung vermutlich meist verwendete Wert, ein »wichtiger« oder ein anderer sinnvoller Wert sein. Testfälle werden dann so erstellt, dass alle Parameter mit ihren jeweiligen Basiswerten belegt werden und jeweils nur der Wert eines Parameters von Testfall zu Testfall verändert wird. Die anderen Parameter bleiben unverändert mit ihren jeweiligen Basiswerten

belegt. Das Vorgehen wird so lange fortgesetzt, bis alle Parameter separat verändert wurden.

7.1.2 Strukturbasierte Testverfahren

Folgende strukturbasierte Testverfahren werden im Standard behandelt:

- Anweisungstest (Statement Testing)
- Zweigtest (Branch Testing)
- Entscheidungstest (Decision Testing)
- Bedingungstest (Branch Condition Testing)
- Mehrfachbedingungstest (Branch Condition Combination Testing)
- Modifizierter Bedingungs-/Entscheidungstest (Modified Condition Decision Coverage (MCDC) Testing)
- Datenflusstest (Data Flow Testing)

Auch hier findet der Leser alle aufgeführten strukturbasierten Verfahren in diesem Buch bis auf den Datenflusstest.

Bevor dieser kurz vorgestellt wird, ist hier noch eine Klarstellung zum Unterschied zwischen Zweigtest und Entscheidungstest angebracht: Beide fokussieren auf Entscheidungen im Testobjekt. Die Überlegungen und die Messung der erreichten Überdeckung basieren beim Zweigtest auf dem Kontrollflussgraphen, der eine 1-zu-1-Abbildung[5] des Programmtextes darstellt. Der Kontrollflussgraph besteht aus Knoten, die die Anweisungen bzw. Sequenzen von Anweisungen repräsentieren, und aus Kanten zwischen den Knoten, die den Kontrollfluss darstellen. Die Überdeckung wird durch Zählen aller Kanten (Zweige) und der bisher beim Testen überdeckten Kanten ermittelt. Grundlage des Entscheidungstests ist der Programmtext – genauer die Entscheidungen im Testobjekt. Für die Überdeckung wird lediglich der Entscheidungsausgang (`true` oder `false`) ermittelt. Wir haben uns im Buch auf den Entscheidungstest konzentriert.

Datenflusstest

Programme bearbeiten Daten. Der Datenflusstest stellt die Verwendung von Daten – also die Verwendung einzelner Variablen im Programm – in den Mittelpunkt der Testüberlegungen. Für jede Variable des Testobjekts wird ermittelt, wie und an welcher Stelle im Programm sie verwendet wird. Die drei folgenden Verwendungsarten werden unterschieden:

[5] Die Abbildung kann zu unterschiedlichen Graphen führen, je nachdem welche verzweigungsfreien Teile des Testobjekts zu wie vielen Knoten zusammengefasst werden.

- Definition (def, define) – Eine Variable bekommt einen Wert zugewiesen: Sie wird definiert. Dies ist z.B. der Fall, wenn die Variable auf der linken Seite einer Zuweisung aufgeführt ist.
- Berechnung (c-use, computational use) – Der Wert der Variablen wird in einer Berechnung verwendet. Dies ist z.B. der Fall, wenn die Variable auf der rechten Seite einer Zuweisung aufgeführt ist.
- Entscheidung (p-use, predicate use) – Der Wert der Variablen wird zur Bestimmung eines Wahrheitswertes innerhalb einer Entscheidung verwendet. Dies ist z.B. der Fall, wenn die Variable bei einer *If*-Anweisung innerhalb der Entscheidung aufgeführt ist.

Die Kernidee beim Datenflusstest ist folgende: Zwischen der Definition und der Verwendung einer Variablen besteht eine Beziehung, die beim Testen überprüft werden soll. Oder genauer: Zwischen Definition und Verwendung existiert ein Weg – ein Pfad – durch das Testobjekt, der die beiden Benutzungen der Variablen verbindet. Ein solcher Pfad soll beim Datenflusstest zur Ausführung gebracht werden.

Nehmen wir an, eine Variable bekommt bei Eintritt in das Testobjekt einen initialen Wert zugewiesen (sie wird definiert). Die Variable wird an zwei Stellen im Testobjekt verwendet. Beide Stellen sind von der Initialisierung aus erreichbar, ohne dass der Wert zwischendurch in einer anderen Anweisung verändert wird. Zwei Testfälle sind zu erstellen, wobei jeder Testfall den jeweiligen, entsprechenden Pfad von der Definition (Initialisierung) der Variablen zur Verwendung zur Ausführung bringt.

Beim Datenflusstest unterscheidet der Standard ISO 29119 fünf Abstufungen, die die Intensität des Tests betreffen. Es werden dabei immer die Pfade von einer Definition zu einer Verwendung einer Variablen analysiert. Danach sind Testfälle zu spezifizieren, die genau diese Pfade durch das Testobjekt zur Ausführung bringen.

- All-definitions-testing: Von jeder Definition einer Variablen soll ein Pfad zu irgendeiner der beiden möglichen Verwendungen (c-use, p-use) durch wenigstens einen Testfall ausgeführt (überdeckt) werden. Im Fokus stehen die Definitionen, sind alle abgearbeitet – alle einmal verwendet, egal wie –, ist eine 100%ige Überdeckung erreicht.
- All-c-uses-testing: Von jeder Definition einer Variablen soll mindestens ein Pfad zu jeder berechnenden Verwendung (c-use) durch wenigstens einen Testfall ausgeführt (überdeckt) werden. Im Fokus stehen die unterschiedlichen berechnenden Verwendungen, die von einer Definition aus erreichbar sind. Sind alle Pfade abgearbeitet, ist eine 100%ige Überdeckung erreicht.
- All-p-uses-testing: Analog zu All-c-uses-testing, nur dass nicht die berechnenden Verwendungen, sondern die Verwendungen innerhalb der

Entscheidungen im Fokus stehen und zur Ermittlung der Testfälle herangezogen werden.

- All-uses-testing: Die Vereinigung von All-c-uses-testing und All-p-uses-testing. Wird eine Definition einer Variablen sowohl zu Berechnungen als auch in einer Entscheidung verwendet, sind alle Verwendungen für die Ermittlung der auszuführenden Pfade und somit der Testfälle heranzuziehen.
- All-DU-path-testing (DU: define, use): Die umfassende Überdeckung verlangt, dass jeder mögliche Pfad von der Definition einer Variablen zu allen Verwendungen (sowohl zur Berechnung als auch zur Entscheidungsbestimmung) durch wenigstens einen Testfall ausgeführt (überdeckt) wird. Die Erweiterung zu All-uses-testing besteht darin: Wenn es mehrere verschiedene Pfade von einer einzelnen Definition zu einer einzelnen Verwendung gibt, dann sind alle diese unterschiedlichen Pfade durch Testfälle abzudecken.

Die Ermittlung der jeweiligen zu prüfenden Pfade ist recht aufwendig. Darüber hinaus müssen dann noch die passenden Eingabedaten überlegt werden, die genau diese Pfade zur Ausführung bringen. Ohne adäquate Werkzeugunterstützung ist ein sinnvoller Datenflusstest nicht durchzuführen. Wir haben uns entschieden, den Datenflusstest nicht aufzunehmen, da er nicht wirklich »lean« ist.

7.1.3 Erfahrungsbasierte Testverfahren

Im Standard ISO 29119 wird die Fehlererwartungsmethode (Error Guessing) als einziger Ansatz bei den erfahrungsbasierten Testverfahren aufgeführt. In diesem Buch wird die Fehlererwartungsmethode beim freien Testen vorgestellt. Die Fehlererwartungsmethode beruht überwiegend auf der Erfahrung der Tester. Die Methode unterliegt aber einer gewissen Systematik – im Vergleich zu anderen erfahrungsbasierten Testverfahren – und ist daher im Standard aufgenommen.

Im Standard wird gefordert, dass eine Aufstellung von potenziellen Fehlern, die im Testobjekt erwartet werden, angefertigt wird. Welche möglichen Fehler in der Liste aufgenommen werden, ist abhängig von der Erfahrung der Tester. Die Aufstellung wird genutzt, um Testfälle zu erstellen, die dazu führen sollen, dass der Fehler im Testobjekt auftritt. Sind alle Fehleroptionen aus der Aufstellung geprüft, wird die Fehlererwartungsmethode als ausreichend durchgeführt angesehen und beendet. Wir sehen die Aufstellung der Liste als sehr nützlich an, sie ist aber nach unserem Dafürhalten nicht zwingend erforderlich, um die Fehlererwartungsmethode durchführen zu können.

Der Standard ISO 29119 enthält eine Reihe sehr nützlicher Testverfahren, die fast alle in diesem Buch ausführlich behandelt werden. Wenn die Testverfahren vom Entwickler zur Spezifikation der Testfälle genutzt werden, kann er mit gutem Gewissen behaupten: »Ich teste meine Software nach dem aktuellen Standard ISO 29119!«

8 Ein Leitfaden zum Einsatz der Testverfahren

In der Einleitung haben wir den Inhalt des Buchs als »Test-Büffet« vorgestellt. Wir haben darauf hingewiesen, dass beim Testen – wie auch bei einem Speisebüffet – eine Auswahl aus den Angeboten (Speisen oder Testverfahren) zu treffen ist. Es drängt sich daher die folgende Frage auf: »Wann soll ich welches Testverfahren einsetzen?« Dieser Frage gehen wir in diesem Kapitel nach.

Im Folgenden wird ein mögliches Vorgehen beschrieben, das dem Leser eine Art Leitfaden an die Hand geben soll. Es besteht aber keine Garantie, dass bei Befolgung des Leitfadens ein fehlerfreies Programm entsteht! Vielmehr wird aufgeführt, was wir für ein sinnvolles Vorgehen erachten. Sie haben aber jeweils zu entscheiden, ob dieses Vorgehen bei Ihrem Problem sinnvoll ist! Mit der Zeit und dem Sammeln von Erfahrung beim Testen werden Sie selbst gut einschätzen und entscheiden können, welche Testverfahren bei welchen Konstellationen geeignet sind. Den Leitfaden werden Sie dann nicht mehr benötigen.

Vorab noch ein paar Hinweise: Bei der Nennung der Testverfahren ist im Leitfaden eher die Testdurchführung gemeint. Die Erstellung der Testfälle soll ja frühzeitig erfolgen – aus den bereits erwähnten Vorteilen! Auch ist vorab festzulegen, wie intensiv getestet werden soll, d.h., welcher Überdeckungsgrad bei den einzelnen Testverfahren zu erreichen und somit nachzuweisen ist. Der Leitfaden ist als WENN-DANN-Folge beschrieben. In aller Regel werden mehrere WENN-Bedingungen zutreffen und somit sind auch die zutreffenden DANN-Empfehlungen umzusetzen.

Prolog

- **WENN** die Programmierung abgeschlossen ist, der Compiler keine Meldungen mehr liefert und mit der Testdurchführung begonnen werden soll,
  DANN ist der Code durch ein statisches Analysewerkzeug zu überprüfen.
- **WENN** alle Hinweise des Werkzeugs abgearbeitet sind,
  DANN kann ein Codereview sehr sinnvoll sein.
- **WENN** alle Befunde des Reviews geklärt sind,
  DANN kann mit der Testausführung begonnen werden.
- **WENN** es mit der Testausführung losgehen soll,
  DANN ist ein Werkzeug zur Ermittlung der Codeüberdeckung bei jedem Testlauf einzusetzen, um nach den Testläufen zu ermitteln, ob die vorab festgelegte Überdeckung auch erreicht wurde.

Jetzt gehts los!

- **WENN** das Testobjekt eine oder mehrere Hauptfunktionen hat,
  DANN sind diese mit Happy-Path-Tests[1] zu prüfen.
- **WENN** der Happy-Path-Test scheitert,
  DANN lohnt es sich nicht, die Auswahl von Testverfahren fortzusetzen. Es liegen dann vermutlich gravierende Missverständnisse vor, die erst zu klären sind. Danach muss der Programmtext entsprechend angepasst bzw. korrigiert werden.
- **WENN** das Testobjekt Parameter hat, die weitgehend voneinander unabhängig sind und sich nicht gegenseitig beeinflussen, und wenn diese Parameter für bestimmte Datenbereiche oder Werte definiert sind,
  DANN ist der Äquivalenzklassentest zur Erstellung der Testfälle durchzuführen. Sinnvollerweise wird die Grenzwertanalyse gleich mitberücksichtigt, da die Grenzen der Äquivalenzklassen die »spannenden« Testdaten sind.
- **WENN** das Testobjekt viele Parameter hat und diese auf die ein oder andere Art miteinander zu kombinieren sind,
  DANN ist die Klassifikationsbaummethode sehr gut geeignet, nicht den Überblick zu verlieren und mithilfe des Baums die Testfälle zusammenzustellen.

[1] Im Folgenden wird nicht mehr aufgeführt, dass beim Test gefundene Fehler zu beseitigen sind, bevor das nächste Testverfahren genutzt wird, und dass alle Tests nach einer Änderung oder einem Refactoring zu wiederholen sind.

- **WENN** die Parameter des Testobjekts miteinander frei kombinierbar sind und dies bei der Anwendung des Programms in der Praxis auch so vorkommen wird,

  DANN ist das kombinatorische Testen durchzuführen. Da der Test aller möglichen Kombinationen in aller Regel zu aufwendig ist, wird das Vorgehen zum paarweisen Testen empfohlen.
- **WENN** die Parameter des Testobjekts nicht völlig frei kombinierbar sind, sondern Abhängigkeiten bestehen,

  DANN ist der Entscheidungstabellentest einzusetzen. Aus der Entscheidungstabelle kann abgelesen werden, welche der möglichen Kombinationen für das Testobjekt relevant und somit zu testen sind.
- **WENN** die Historie – der bisherige Ablauf des Testobjekts – oder der aktuelle Zustand des Systems oder eines Objekts eine Rolle spielt,

  DANN ist der zustandsbasierte Test durchzuführen. Es sind Folgen von Testfällen zur Prüfung der Zustände und der Zustandsübergänge auszuführen. Ob dabei ein N-Switch-Verfahren oder der Übergangsbaum oder beide zu verwenden sind, ist von Fall zu Fall zu entscheiden.
- **WENN** die Eingaben einer Syntax unterliegen, also einem vorgegebenen Format oder bestimmten Regeln entsprechen müssen,

  DANN ist der Syntaxtest mit der Prüfung der Einhaltung bzw. der Verletzung der Syntaxregeln durchzuführen.
- **WENN** zufällige Eingaben sinnvoll erscheinen,

  DANN ist der Zufallstest durchzuführen. Hierbei ist zu bedenken, dass der Vergleich zwischen den erwarteten Ergebnissen und den durch den Testlauf erzeugten Ergebnissen meist nicht durchgeführt werden kann, da keine erwarteten Ergebnisse für die zufällig erzeugten Eingaben vorliegen.
- **WENN** ein oder mehrere Testverfahren ausgewählt wurden,

  DANN ist zu prüfen, ob nicht noch weitere Testverfahren sinnvoll für die Prüfung des Testobjekts eingesetzt werden können.

Sind Sie nun fertig?

- **WENN** das Testobjekt mit Testfällen, die in der Regel durch die Anwendung von mehreren Testverfahren erstellt wurden, ausgeführt wurde,

  DANN ist je nach vorab festgelegter Anforderung an die zu erreichende Überdeckung (Anweisung, Entscheidung, ...) festzustellen, welche Teile des Testobjekts noch nicht ausgeführt wurden.
- **WENN** noch nicht ausgeführte Programmteile vorhanden sind,

  DANN sind ergänzende Testfälle zu spezifizieren und auszuführen, um diese Programmteile zur Ausführung zu bringen.

- **WENN** dies nicht gelingt,
  DANN ist zu untersuchen, welche Ursachen vorliegen, warum die Programmteile nicht ausgeführt werden können. Entsprechende Korrekturen des Progammtextes sind vorzunehmen.
- **WENN** im Testobjekt Entscheidungen enthalten sind, die aus mehreren Bedingungen bestehen, die über boolesche Operatoren miteinander verknüpft sind,
  DANN sind diese separat mit einem der drei Bedingungstests – je nach geforderter Intensität (Empfehlung: modifizierter Bedingungs-/Entscheidungstest) – zu prüfen.
- **WENN** im Testobjekt Schleifen eine wichtige Rolle spielen,
  DANN ist zu untersuchen, ob mit den bisher durchgeführten Testfällen die Anforderungen an den Schleifentest bereits erfüllt sind oder ob ergänzende Testfälle durchzuführen sind.

Aber jetzt – oder?

Nein, jetzt sind Sie, Ihre Erfahrung, Fantasie und Ihr Ideenreichtum gefragt!

- **WENN** Sie noch eine Idee haben, wo es *krachen* könnte – und so eine Idee wird in aller Regel immer vorhanden sein –,
  DANN formulieren Sie Ihre Idee als Test-Charta und testen Sie explorativ. Erst wenn Sie sicher[2] sein können, dass es nicht zum »Krachen« der Software kommt, kann das explorative Testen beendet werden.
- **WENN** Ihnen dann immer noch Ideen einfallen, die sich nicht als Test-Charta formulieren lassen,
  DANN ist freies Testen angesagt! Wenn Ihnen die Ideen ausgehen, ist das freie Testen beendet.
- **WENN** Ihnen dann noch Fehlersituationen aus vergangenen Projekten in Erinnerung sind oder eine Liste mit solchen Fehlern existiert,
  DANN ist der Einsatz der Fehlererwartungsmethode eine sinnvolle Ergänzung für die Komplettierung der Testaktivitäten.

Epilog

- **WENN** Sie an alles gedacht haben und somit ausreichend und angemessen Ihre Software getestet haben,
  DANN können Sie ganz beruhigt und zufrieden die Software einchecken und Ihren Feierabend genießen ...

[2] »Sicher« kann man nie sein. Hier ist damit gemeint, dass Sie das explorative Testen als ausreichend ansehen.

... wenn Sie auch von Anfang an die paar C++-Besonderheiten berücksichtigt haben, die im nächsten Kapitel beschrieben werden.

1. Compiler und Analysewerkzeuge nutzen – evtl. Fehler korrigieren
2. Codereview durchführen – evtl. Fehler korrigieren
3. Werkzeug zur Ermittlung der Codeüberdeckung einsetzen
4. Mehrere »passende« Testverfahren auswählen, systematisch Testfälle erstellen und ausführen – evtl. Fehler korrigieren
5. Erreichte Codeüberdeckung ermitteln und ggf. weitere Testfälle durchführen – Ursache von »dead code« klären und beseitigen
6. Erfahrungsbasierte Testfälle überlegen und ausführen – evtl. Fehler korrigieren

9 Zu berücksichtigende C++-Eigenschaften

Dieses Kapitel nennt einige C++-spezifische Punkte, die bei Tests möglicherweise beachtet werden müssen und bisher nur zu einem kleinen Teil behandelt wurden.

9.1 Automatische Typumwandlung

C++ erlaubt die Umwandlung integraler Datentypen in integrale Datentypen anderer Bitbreite. Dabei gibt der Compiler nicht automatisch eine Warnung aus. Auch wird ignoriert, dass der eine Typ vorzeichenbehaftet ist und der andere nicht. Dadurch kann es zu Informationsverlust kommen. An dieser Stelle gehen wir nicht ins Detail, weil bereits Abschnitt 4.3.3 auf Seite 39 zeigt, zu welchen Problemen es dabei kommen kann.

9.2 Undefinierte Bitbreite

In Java gibt es unter anderem die integralen Datentypen `short`, `int` und `long` mit den Größen 16 Bit, 32 Bit und 64 Bit. Das gilt auf jedem Java-System. C++ hat dieselben Datentypen, legt jedoch die Bitbreite nur für `short` fest (16 Bit). Ansonsten gilt Bits(`short`) $\leq$ Bits(`int`) $\leq$ Bits(`long`). Damit sind C++-Programme, die sich auf den Platzbedarf von Variablen dieser Datentypen oder auf bestimmte darstellbare Zahlenbereiche verlassen, nicht portabel. Die Konsequenz: Ein C++-Quellprogramm, das auf einen anderen Rechner portiert wird, muss nicht nur neu übersetzt, sondern auch neu getestet werden. Es gibt auch in C++ die Möglichkeit, bestimmte Bitbreiten zu garantieren, indem etwa die Typen `int_least8_t`, `int_least16_t`, `int_least32_t` usw. verwendet werden. Jeder dieser Typen hat mindestens die angegebene Bitbreite. Allerdings gibt es sie erst seit C++11.

9.3 Alignment

Das genannte Problem der undefinierten Bitbreite wirkt sich auch auf den Platzbedarf von Objekten von Klassen aus. Um den Zugriff der CPU auf einzelne Attribute schnell zu gestalten, werden sie auf bestimmte Grenzen im Speicher gelegt. Dabei wird gegebenenfalls bewusst Speicherplatz zugunsten der Geschwindigkeit verschenkt.

Ein Beispiel:

```
struct Struct {
  char a[1];
  double b;
};
```

Listing 9.1: Struktur mit 16 Bytes Platzbedarf

Die Größe dieser Struktur ist nicht `sizeof(char)` plus `sizeof(double)`, also 1 + 8 = 9, sondern 16 (auf unserem Computer, auf Ihrem System mag sich ein anderer Wert ergeben). Damit wird deutlich, dass sieben Bytes der Struktur ungenutzt sind. Das `char`-Array könnte vergrößert werden, ohne dass sich `sizeof(Struct)` ändert. Erst wenn die Anzahl der Zeichen 8 überschreitet, wird `sizeof(Struct)` größer. Die Funktion `alignof(Datentyp)` gibt den Abstand der Wortgrenzen[1] für einen Datentyp in Bytes an. Er ist vom jeweiligen System abhängig und möglicherweise zu berücksichtigen, etwa bei einer eigenen Speicherverwaltung.

9.4 32- oder 64-Bit-System?

Die Größe mancher Datentypen ist auf einem 32-Bit-System anders als auf einem 64-Bit-System. Auf einem 32-Bit-System erfolgreich getestete und korrekte Programme können bei der Neuübersetzung auf einem 64-Bit-System falsch werden. Listing 9.2 zeigt ein harmlos erscheinendes Beispiel:

```
std::string zudurchsuchenderText("ein Text");
unsigned int punktPosition = zudurchsuchenderText.find(".");   //?
if(punktPosition == std::string::npos) {
  std::cout << "'.' nicht gefunden!\n";
}
else {
  std::cout << "'.' gefunden an Position " << punktPosition << '\n';
}
```

Listing 9.2: Problem bei einem Systemwechsel

[1] *http://www.ibm.com/developerworks/library/pa-dalign/* gibt weitere Informationen zum Thema Alignment.

Auf einem 32-Bit-System gibt das Programm korrekt *'.' nicht gefunden!* aus. Auf unserem 64-Bit-System ist die Ausgabe jedoch *'.' gefunden an Position 4294967295*. Die Ursache: Der Typ von `npos` ist der `unsigned`-Datentyp `string::size_type`. Der Wert ist die maximal mögliche `unsigned`-Zahl für diesen Typ. `sizeof(unsigned int)` ist jedoch 4 und `sizeof(string::size_type)` ist 8 auf unserem System. Der Fehler hätte sich bei sauberer Programmierung vermeiden lassen, wenn nämlich `unsigned int` durch `std::string::size_type` oder noch besser durch `auto` ersetzt worden wäre.

Ein ähnliches Problem kann bei Zeigern auftreten, wenn die Zeiger, wie es in manchen älteren Programmen vorkommt, in einer `unsigned int`-Zahl (oder `long`) gespeichert oder als `unsigned int`-Parameter einer Funktion übergeben werden. Ähnlich wie im obigen Beispiel werden bei der Typumwandlung Bits abgeschnitten, und der Zeiger zeigt ins Nirwana, sobald Adressen im Spiel sind, die den Bereich von `unsigned int` überschreiten.

9.5 static-Missverständnis

Im C++-Standardkomitee gab es das Bestreben, neue Schlüsselwörter möglichst zu vermeiden. Das führte dazu, dass die Bedeutung mancher Schlüsselwörter (oder Symbole) vom Umfeld abhängt, in dem sie benutzt werden. Missverständnisse sind damit vorprogrammiert. In einem C++-Buch von 2009[2] haben wir das folgende Beispiel für einen überladenen ++-Operator in der Postfix-Form gefunden:

```
// Postfix-Operator ++ überladen
myComplex& operator++(int) {
  static myComplex tmp(*this);
  _real++;
  _image++;
  return tmp;
}
```

Listing 9.3: Fehlerhafter ++-Operator

Sicher ist es fraglich, ob ein ++-Operator für komplexe Zahlen überhaupt sinnvoll ist. Für die folgende Betrachtung seien zwei Variablen gegeben:

```
myComplex c1(1.1, 2.2);
myComplex c2 = c1;
```

Listing 9.4: Zwei Variablen für den Test

Abgesehen davon, dass eine Referenz als Rückgabetyp nicht angebracht ist, weil man dann `c1++ = c2;` schreiben könnte, gibt es hier ein weiteres Miss-

[2] Wegen der vielen weiteren Fehler in diesem Werk geben wir die Quelle nicht an.

verständnis. Es hat vermutlich damit zu tun, dass `static` ein Schlüsselwort ist, das je nach Kontext verschiedene Bedeutungen haben kann. Wenn nun der obige Operator getestet werden soll, kann man Folgendes schreiben:

```
c2 = c1++;              // 1. Postfix-Operator ausführen
c2.printComplex();      // 2. okay, c2 ist jetzt (1.1, 2.2) (d.h. alter Wert von c1)
c1.printComplex();      // 3. okay, c1 ist jetzt (2.1, 3.2)
```

Listing 9.5: Test des Postfix-++-Operators, Teil 1

Alles wunderbar, Test bestanden! Aber wenn man den Test wiederholt, also die folgenden beiden Zeilen dem obigen Programmcode hinzufügt:

```
c2 = c1++;              // 4. Postfix-Operator 2. Mal ausführen
c2.printComplex();      // 5.
```

Listing 9.6: Test des Postfix-++-Operators, Teil 2

dann stellt man fest, dass `c2` *unverändert* (1.1, 2.2) bleibt, d.h. *ungleich* dem vorherigen Wert von `c1` ist. Die Ursache besteht in einem Missverständnis. Eine innerhalb einer Funktion deklarierte Nicht-`static`-Variable wird bei jedem Betreten der Funktion neu initialisiert. Mit dem Schlüsselwort `static` sollte wohl erreicht werden, dass die Variable zusätzlich zu dieser Eigenschaft einen Speicherplatz außerhalb der funktionslokalen Variablen erhält. Normale funktionslokale Variablen liegen auf dem Laufzeit-Stack und sind nach Ende der Funktion verschwunden. Schließlich ist die Rückgabe der Referenz auf eine lokale Variable ein schwerer Fehler. Tatsächlich aber wird eine innerhalb einer Funktion deklarierte `static`-Variable nur genau einmal initialisiert, und zwar beim erstmaligen Betreten einer Funktion. Sie behält den Wert bei allen nachfolgenden Aufrufen der Funktion, wenn er nicht explizit geändert wird, etwa durch eine Zuweisung `tmp = *this;` im obigen Programmfragment.

9.6 Memory Leaks

Die Tests in diesem Buch prüfen das Verhalten der Software bei Variation der Eingabedaten. Danach kann es sein, dass ein Programm zwar alle Tests besteht, aber trotzdem fehlerhaft ist. Das ist dann der Fall, wenn es Speicher verbraucht, ohne ihn wieder freizugeben, wenn es also zu Speicherlecks (*memory leaks*) kommt. Im folgenden einfachen Beispiel soll eine Funktion einen Text als Argument entgegennehmen und den in Großbuchstaben umgewandelten Text zurückgeben:

```
const char* grossbuchstaben(const char* text) {
  char* temp = new char[strlen(text)+1];
  std::strcpy(temp, text);
```

```
  for(int i = 0; i < strlen(text); ++i) {
    temp[i] = std::toupper(temp[i]);
  }
  return temp;
}

// Anwendung zum Beispiel
std::string s(grossbuchstaben("memory leak"));
```

Listing 9.7: Umwandlung in Großbuchstaben

Weil der übergebene Text nicht geändert werden kann, wird mit `new` ein Zwischenspeicher beschafft, in dem die einzelnen Buchstaben umgewandelt werden. Das Problem: Der Speicher wird nicht wieder freigegeben. Die Verantwortung darüber liegt beim Aufrufer der Funktion[3]. Bei der obigen unschuldig aussehenden, aber fehlerhaften Anwendung ist der zurückgegebene Zeiger ein temporärer Parameter, der nach der Initialisierung des Strings `s` nicht mehr zugreifbar ist.

Wenn der Rechner mit ausreichend Speicher ausgestattet ist, fällt beim Test nichts auf – die Funktion läuft einwandfrei. Wenn sie aber in einem Programm verwendet würde, das rund um die Uhr läuft, etwa in einer Überwachungssoftware in einem Atomkraftwerk, kann der Speicher nach und nach volllaufen und das Programm bleibt irgendwann, möglicherweise erst nach Wochen, stehen. Die Fehlerdiagnose ist dann schwierig.

In anderen Programmiersprachen, zum Beispiel Java, ist ein sogenannter Garbage Collector integriert, der beim Laufen eines Programms im Hintergrund den nicht benutzten Speicher freigibt. Das kann allenfalls dazu führen (und ggf. problematisch sein), dass das Programm für kurze Zeit angehalten wird.

Aus diesem Grund ist es sinnvoll, C++-Programme auf Memory Leaks zu prüfen, wenn `new` und `delete` oder die C-Äquivalente `malloc` und `free` verwendet werden. Einigen Programmierern ist dieses Problem nicht bewusst, besonders wenn die von ihnen verwendete Literatur diesen Punkt ignoriert. So habe ich in einem verbreiteten C++-Buch[4] das folgende Beispiel gefunden (wörtlicher Auszug):

```
template <typename T>
class Liste {
   private:
   AnfangsKnoten<T> *anfang;
   public:
   // Bei Anlegen gleich ein Objekt "Anfangsknoten" erzeugen
```

[3] Manche kennen das Verhalten von der C-Funktion `strdup()`.

[4] Ein aktuelleres Buch desselben Autors, siehe Fußnote 2 auf Seite 213.

```
    // der Konstruktor von "AnfangsKnoten" erzeugt wiederum
    // ein Objekt "EndKnoten", auf dass dieser gleich zeigt.
    Liste() { anfang = new AnfangsKnoten<T>; }
    ~Liste() { delete anfang; }
    void einfuegen( T* d ) {
       anfang->einfuegen(d);
    }
    void alles_anzeigen() {
       anfang->anzeigen();
    }
};
```

Listing 9.8: Klasse mit Memory Leak

Es sind hier gleich mehrere grobe Fehler zu sehen. Der Kopierkonstruktor und der Zuweisungsoperator fehlen, sodass es bei einer Kopie der Liste zum Speicherzugriffsfehler (segmentation fault) kommt. Wir möchten aber das Augenmerk auf den Destruktor lenken. Er löscht mit `delete` den ersten Knoten in der Liste. Was aber fehlt, ist die Freigabe des Speichers für den Knoten, auf den dieser Anfangsknoten verweist, sowie alle daran hängenden Knoten. Je länger die Liste ist und je öfter eine Instanz davon erzeugt wird, desto größer ist das Memory Leak.

Dabei gibt es ein gutes Open-Source-Werkzeug zur Entdeckung solcher Fehler: das schon auf Seite 21 erwähnte Valgrind. Wenn die Datei *a.out* das ausführbare Programm ist, überwacht Valgrind die Ausführung mit dem folgenden Befehl:

```
valgrind --leak-check=yes a.out
```

Im Anschluss gibt Valgrind die Fehler im Detail und auch in einer Zusammenfassung aus. Dies sind die letzten Zeilen der Ausgabe [5]:

```
==4297== LEAK SUMMARY:
==4297==    definitely lost: 24 bytes in 1 blocks
==4297==    indirectly lost: 96 bytes in 8 blocks
==4297==      possibly lost: 0 bytes in 0 blocks
==4297==    still reachable: 72,704 bytes in 1 blocks
==4297==         suppressed: 0 bytes in 0 blocks
==4297== Reachable blocks (those to which a pointer was found) are not shown.
==4297== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==4297==
==4297== For counts of detected and suppressed errors, rerun with: -v
==4297== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Der Aufwand für so eine dynamische Analyse ist wirklich nicht groß! Noch besser ist es allerdings, Speicherlecks prinzipiell zu vermeiden, etwa durch

[5] Die Zahl ganz links ist die Prozess-Id.

die Verwendung sogenannter »smarter« Zeiger wie `std::unique_ptr` oder `std::shared_ptr`.

> Das ist nicht das Ende ...
> ... denn es gibt viele Möglichkeiten, versehentlich durch unsaubere Programmierung Fehler zu machen, wie die Beispiele oben zeigen. Sie alle hier aufzuzählen, ist unmöglich. Umso wichtiger sind systematische Tests.

Glossar

Dieses Glossar enthält Kurzdefinitionen der wichtigsten Begriffe, die in diesem Buch verwendet werden. Ein Teil der Definitionen ist von [Spillner & Linz 12] und aus dem ISTQB®-Glossar [URL: ISTQB] übernommen worden.

Abnahmetest

Formales Testen hinsichtlich der Benutzeranforderungen und -bedürfnisse bzw. der Geschäftsprozesse, das durchgeführt wird, um einen Auftraggeber oder eine bevollmächtigte Instanz in die Lage zu versetzen, entscheiden zu können, ob ein System anzunehmen ist oder nicht.

Akzeptanztest

1. → *Abnahmetest* aus Sicht des Benutzers.
2. Teilmenge aller vorhandenen Testfälle, die das Testobjekt als Eintrittskriterium in eine bestimmte → *Teststufe* bestehen muss.

Anomalie

Abweichung von den Erwartungen aufgrund von → *Spezifikation*en und Erfahrungen. In diesem Buch spielen → *Datenflussanomalien* eine Rolle.

Anweisungstest

Kontrollflussbasiertes Testverfahren (→ *kontrollflussbasierter Test*), das die mindestens einmalige Ausführung aller Anweisungen des Testobjekts fordert.

Äquivalenzklasse

Teil des Wertebereichs von Ein- oder Ausgabewerten, für den ein gleichartiges Verhalten der Komponente oder des Systems angenommen wird, basierend auf der zugrunde liegenden → *Spezifikation*.

Äquivalenzklassenbildung

Zerlegung von Ein- oder Ausgabewerten eines Programms in eine endliche Menge von Klassen, deren Elemente jeweils ein gleiches funktionales Verhalten zeigen.

Bedingungstest (einfacher)

Kontrollflussbasiertes → *Whitebox-Testentwurfsverfahren*, das die → *Überdeckung* der (Teil-)Bedingungen einer Entscheidung mit *wahr* und *falsch* fordert.

Blackbox-(Testentwurfs-)Verfahren / Blackbox-Test

Nachvollziehbares Verfahren zur Herleitung und Auswahl von Testfällen basierend auf einer Analyse der funktionalen oder nicht funktionalen → *Spezifikation* einer Komponente oder Systems ohne Berücksichtigung der internen Struktur.

Clean Code

Clean Code ist Programmcode, der leicht verständlich und gut wartbar ist. Der Name geht auf das Buch von Robert Martin [Martin 09] zurück, das allerdings auf die Programmiersprache Java bezogen ist. Clean Code heißt unter anderem, Codedoppelungen zu vermeiden, Entwurfsmuster zu benutzen und sichere Programmkonstruktionen zu verwenden, zum Beispiel Speicherbeschaffung und Freigabe in einer Klasse zu kapseln. »Unsauberer« Code kann bereinigt werden. Dieser Vorgang wird → *Refactoring* genannt.

Datenfluss

Eine abstrakte Darstellung der Abfolge von Zustandsänderungen eines Datenobjekts, bei der das Objekt folgende Zustände annimmt: Definition/Neuanlage, Verwendung oder Löschung.

Datenflussanomalie

Eine aller Erfahrung nach fehlerhafte oder nicht erwartete Abfolge von Operationen auf Daten.

Datenflusstest

Ein → *Whitebox-Testentwurfsverfahren*, bei dem Testfälle entworfen werden, um Definition-Verwendungspaare von Variablen auszuführen.

DAU-Test – Dümmster Anzunehmender User

Ein Test, bei dem aus einer größeren Menge von möglichen Eingaben diese zufällig ausgewählt und Tasten zufällig betätigt werden, unabhängig davon, wie das Produkt im Betrieb tatsächlich verwendet wird.[6]

Design by Contract

Eine → *Spezifikation* kann als *Vertrag* zwischen Aufrufer und Funktion aufgefasst werden. Design by Contract meint: Die Funktion gewährleistet die → *Nachbedingung*, wenn der Aufrufer die → *Vorbedingung* einhält. Die Analogie zu einem Vertrag zwischen Kunde und Softwarehaus liegt auf der Hand (siehe [Meyer 13]).

Entscheidungstabelle

Eine Tabelle von Regeln, die jeweils aus einer Kombination von Bedingungen (z.B. Eingaben und/oder Auslösern) und den dazugehörigen Aktionen (z.B. Ausgaben und/oder Wirkungen) bestehen. Entscheidungstabellen können zum Entwurf von Testfällen verwendet werden.

Entscheidungstest

Kontrollflussbasiertes → *Whitebox-Testentwurfsverfahren*, das die mindestens einmalige »Verwendung« aller Ausgänge von Entscheidungen des Testobjekts fordert. (Eine *If*-Abfrage hat 2 Ausgänge, `true` und `false`, bei einer *Case*-Anweisung ist jede aufgeführte Auswahl ein Ausgang.)

Entwicklertest

Test, der in der (ausschließlichen) Verantwortung des Entwicklers bzw. der Entwicklungsgruppe des Testobjekts durchgeführt wird (oft mit Komponententest gleichgesetzt).

Exploratives Testen

Testen, bei dem der Tester den Entwurf der Tests aktiv steuert, indem er testet und die Informationen, die er während des Tests erhält, zum Entwurf weiterer Tests verwendet.

[6] Aus dem ISTQB®-Glossar übernommen, dort aber unter »Affentest« als Übersetzung von »monkey testing« aufgeführt.

Fehler

Nichterfüllung einer festgelegten Anforderung[7].

Fehlererwartungsmethode

Ein Testentwurfsverfahren, bei dem die Erfahrung und das Wissen der Tester genutzt werden, um vorherzusagen, welche Fehlerzustände in einer Komponente oder einem System aufgrund der Fehlhandlungen vorkommen könnten, und um Testfälle so abzuleiten, dass diese Fehlerzustände aufgedeckt werden[8].

Fehlernachtest

Wiederholung aller Testfälle, die vor der Fehlerkorrektur einen Fehler erzeugt haben; dient der Überprüfung, ob die Korrektur des ursächlichen Fehlers erfolgreich war.

Funktionalität

Spezifiziert das Verhalten, welches das System erbringen muss; beschreibt, »was« das System leisten soll. Umsetzung ist Voraussetzung dafür, dass das System überhaupt einsetzbar ist.

Grenzwertanalyse

Fehlerorientiertes → *Blackbox-Testentwurfsverfahren*, bei dem die Testfälle unter Nutzung von Grenzwerten (die auf bzw. knapp inner- und außerhalb der Randbereiche der → *Äquivalenzklassen* liegen) entworfen werden.

GUI-Test

Test der grafischen Benutzungsoberfläche (graphical user interface – GUI).

---

[7] Im ISTQB®-Glossar wird zwischen Fehlhandlung (error – Eine menschliche Handlung, die zu einem falschen Ergebnis führt), Fehlerzustand (fault, defect – Innerer Defekt in einer Komponente oder einem System, der eine geforderte Funktion des Produkts beeinträchtigen kann) und Fehlerwirkung (failure – Nach außen sichtbare Abweichung einer Komponente/eines Systems von der erwarteten Lieferung, Leistung oder dem Ergebnis) unterschieden. Wir nutzen im Buch den Oberbegriff Fehler.

[8] Aus dem ISTQB®-Glossar übernommen, dort aber unter »intuitive Testfallermittlung« aufgeführt. Zu den Begriffen Fehlerzustand und Fehlhandlung siehe vorherige Fußnote.

Happy Path

Typisches vorgegebenes Szenario, das vom Testobjekt problemlos verarbeitet wird und ein erwartetes Ergebnis erzeugt, ganz ohne Fehlermeldungen und Ausnahmezustände (Happy-Path-Test – Prüfung der Hauptfunktionen).

Inspektion

Eine Reviewart, die auf einer Sichtprüfung von Dokumenten beruht, um Mängel zu finden, zum Beispiel Nichteinhaltung von Entwicklungsstandards oder Nicht-Konformität gegenüber den zugrunde liegenden Dokumenten. Es ist die formalste Reviewart und folgt stets einem dokumentierten Vorgehen.

Instrumentierung

(Werkzeuggestütztes) Einfügen von Protokoll- oder Zählanweisungen in den Quell- oder Objektcode eines Testobjekts, um während der Ausführung Informationen über das Programmverhalten zu sammeln.

Integrationstest

Testen mit dem Ziel, Fehler in den Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten aufzudecken.

Komponente[9]

1. Kleinste Softwareeinheit, für die eine separate → *Spezifikation* verfügbar ist oder die separat getestet werden kann.
2. Softwareeinheit, die die Implementierungsstandards eines Komponentenmodells (EJB, CORBA, .NET) erfüllt.

Kontrollfluss

Eine abstrakte Repräsentation von allen möglichen Reihenfolgen von Ereignissen (Pfaden) während der Ausführung einer Komponente oder Systems.

Kontrollflussbasierter Test

Dynamisches → *Whitebox-Testentwurfsverfahren*, bei dem die Testfälle unter Berücksichtigung des → *Kontrollflusses* des Testobjekts hergeleitet werden

[9] Komponente, Unit, Modul sind Begriffe, für die es keine einheitlichen Definitionen gibt. Die hier gegebene Definition bezieht sich auf die Testaspekte.

und die Vollständigkeit der Prüfung (→ *Überdeckungsgrad*) anhand des → *Kontrollflussgraphen* bewertet wird.

Kontrollflussgraph

Ein gerichteter Graph G = (N, E, nstart, nfinal). N ist die endliche Menge der Knoten. E ist die endliche Menge der gerichteten Kanten (zwischen den Knoten). »nstart« ist der Startknoten. »nfinal« ist der Endknoten. Kontrollflussgraphen dienen zur Darstellung der Kontrollstruktur von → *Komponenten*.

Kurzschlussauswertung (short-circuit evaluation)

Die Kurzschlussauswertung sorgt dafür, dass Teile einer Bedingung, die nicht zu einer Änderung des Ergebnisses führen, nicht ausgeführt werden. Das spart CPU-Zeit. Ein Beispiel:

```
if(a || b && c) {
...
}
```

Wenn `a` den Wert `true` hat, wird `b && c` nicht mehr ausgewertet. Wenn `a` und `b` beide den Wert `false` haben, wird `c` nicht mehr ausgewertet.

Lean

Der Begriff »lean« (dt. etwa schlank) stammt ursprünglich aus der Automobilindustrie, als es Toyota in den 1950er-Jahren gelang, Autos schnell und mit wenig Ressourceneinsatz zu bauen. »Lean Manufacturing« wurde auf andere Produktionsweisen übertragen, und auch die Softwarewelt bedient sich des Begriffs »lean«, wie sie auch gern andere klangvolle Schlagworte der japanischen Automobilfertigung übernommen hat (z.B. Kanban). Es gibt jedoch keine eindeutige, allseits anerkannte Definition. In diesem Buch verstehen wir unter »lean«, sich auf das Wichtige zu konzentrieren mit dem Ziel, einen akzeptablen Kompromiss zwischen Testaufwand und möglichst geringer Fehleranzahl in einem Programm zu erreichen.

Mehrfachbedingungstest

Ein → *Whitebox-Testentwurfsverfahren*, das die → *Überdeckung* der Bedingungen bzw. Teilbedingungen einer Entscheidung mit *wahr* und *falsch* in allen Kombinationen fordert.

Modifizierter Bedingungs-/Entscheidungstest

Ein → *Whitebox-Testentwurfsverfahren*, bei dem Testfälle so entworfen werden, dass diejenigen Bedingungsergebnisse zur Ausführung kommen, die unabhängig voneinander einen Entscheidungsausgang beeinflussen.

Nachbedingung

Spezifikation eines Programmschritts oder einer Funktion. Die Nachbedingung beschreibt ausgehend vom Zustand des Programm(teil)s *vor* Ausführung (→ *Vorbedingung*), welchen Zustand ein Programm(teil) *nach* der Ausführung hat.

Orakel (Testorakel)

Informationsquelle zur Ermittlung der jeweiligen Sollergebnisse eines Testfalls (in der Regel die Anforderungsdefinitionen oder → *Spezifikation*en). Ein Testorakel kann auch ein existierendes System (als Benchmark), ein Benutzerhandbuch oder das Spezialwissen einer Person sein. Es sollte aber nicht der Code sein, da dann der Code als Grundlage zum Testen genommen, also *gegen sich selbst* getestet wird.

Orthogonale Arrays

Ein zweidimensionales Array mit speziellen mathematischen Eigenschaften, bei dem jede Kombination von zwei Spalten alle Kombinationen der Werte enthält.

Paarweises Testen (Pairwise Testing)

Ein → *Blackbox-Testentwurfsverfahren*, bei dem die Testfälle so entworfen werden, dass alle möglichen diskreten Kombinationen aller Paare von Eingabeparametern ausgeführt werden.

Performanztest (Performance Testing)

Prüfung der Performanz für bestimmte Anwendungsfälle, in der Regel in Abhängigkeit von steigender Last.

Pfad

Weg von der ersten ausgeführten Anweisung einer → *Komponente* bis zur letzten. Typischerweise werden nicht alle möglichen Pfade durchlaufen. So

wird bei einer einmaligen Ausführung einer *If-Else*-Anweisung des Programms nur einer der beiden möglichen Pfade der Anweisung durchlaufen.

Pfadüberdeckung

Kontrollflussbasiertes → *Whitebox-Testentwurfsverfahren*, das die Ausführung aller unterschiedlichen vollständigen Pfade eines Testobjekts fordert. In der Praxis ist es durch die hohe Anzahl an möglichen Pfaden in der Regel nicht anwendbar.

Platzhalter (stub)

Rudimentäre oder spezielle Implementierungen von Softwarekomponenten, die beim Komponenten- und → *Integrationstest* benötigt werden, um noch nicht implementierte → *Komponenten* für die Testdurchführung zu ersetzen bzw. zu simulieren.

Refactoring

Refactoring ist die Verbesserung des Programmtextes ohne Änderung seines Verhaltens, um → *Clean Code* zu erreichen. Ein typisches Beispiel »unsauberen« Codes ist die Codedoppelung, also etwa zwei Funktionen, deren Programmcode nur wenig verschieden ist. Refactoring bedeutet, den gemeinsamen Teil herauszuziehen und in eine eigene Funktion auszulagern. Ein anderes Beispiel ist der Ersatz kurzer kommentierter Variablennamen durch Namen, welche die Kommentare überflüssig machen.

Review

Bewertung eines Softwareprodukts oder eines Projektstatus zur Aufdeckung von Diskrepanzen der geplanten Arbeitsergebnisse und der Identifizierung von Verbesserungspotenzialen. Review wird auch als Oberbegriff für unterschiedliche Arten von statischen Analysen, die von Personen durchgeführt werden, verwendet.

Robustheit

Robustheit ist der Grad, zu welchem Ausmaß das Programm bei ungültigen Eingaben oder Daten noch korrekt funktioniert.

Smoke-Test

1. In der Regel automatisierter Test (Teilmenge aller definierten/geplanten Testfälle), der die Hauptfunktionalitäten einer → *Komponente* oder

eines Systems abdeckt, um sicherzustellen, dass die wichtigen Funktionen eines Programms korrekt arbeiten, jedoch ohne Berücksichtigung einzelner Details.
2. Wird oft realisiert, ohne die Ausgaben des Testobjekts mit vorgegebenen Sollergebnissen zu vergleichen. Stattdessen wird versucht, offensichtliche Fehlerwirkungen (beispielsweise einen Absturz) des Testobjekts zu erzeugen. Dient vorwiegend zur Prüfung der → *Robustheit*.

Spezifikation

Ein Dokument, das die Anforderungen, den Aufbau, das Verhalten oder andere Charakteristika des Systems bzw. der → *Komponente* beschreibt, idealerweise genau, vollständig, konkret und nachprüfbar. Dient Entwicklern als Grundlage für die Programmierung und Testern als Grundlage für die Herleitung von Testfällen mittels → *Blackbox-Testentwurfsverfahren*.

Statische Analyse

Die Durchführung der Analyse (Überprüfung, Vermessung und Darstellung) eines Dokuments (z.B. Anforderung oder Quelltext), ohne dessen Ausführung.

Strukturbasierter Test

→ *Whitebox-Test*

Syntaxtest

Verfahren zur Ermittlung der Testfälle, das bei Vorliegen einer formalen Spezifikation der Syntax für die Eingaben angewendet werden kann.

Systemtest

Test eines integrierten Systems, um sicherzustellen, dass es spezifizierte Anforderungen erfüllt.

Szenariotest

Ein → *Blackbox-Testentwurfsverfahren*, bei dem Testfälle so entworfen werden, dass damit Szenarien der Anwendungsfälle durchgeführt werden.[10]

[10] Aus dem ISTQB®-Glossar übernommen, dort aber unter »anwendungsfallbasierter Test« aufgeführt.

Testbarkeit

1. Mühelosigkeit und Geschwindigkeit, mit der die Funktionalität und das Leistungsniveau des Systems (auch nach jeder Anpassung) getestet werden können.
2. Zugänglichkeit des zu testenden Systems für den Test (Aspekte sind: Offenheit der Schnittstellen, Dokumentationsqualität, Zerlegbarkeit in kleinere Einheiten, Nachbildbarkeit der Produktivumgebung in der Testumgebung).

Testbasis

Alle Dokumente, welche die Anforderungen an ein System oder eine Komponente eines Systems beschreiben, die zur Herleitung oder Auswahl von Testfällen herangezogen werden können.

Testbedingung

Eine Einheit oder ein Ereignis, z.B. eine Funktion, eine Transaktion, ein Feature, ein Qualitätsmerkmal oder ein strukturelles Element einer → *Komponente* oder eines Systems, die bzw. das durch einen oder mehrere Testfälle geprüft/verifiziert werden kann.

Test-Charta

Eine Anweisung von Testzielen und möglichen Testideen wie getestet werden soll. Test-Chartas werden oft im → *explorativen Testen* verwendet.

Testen

Der Prozess beinhaltet alle statischen und dynamischen Aktivitäten, die sich mit der Planung, Vorbereitung und Bewertung eines Softwareprodukts (und der dazugehörigen Arbeitsergebnisse) befassen. Ziel des Prozesses ist der Nachweis, dass das Softwareprodukt allen festgelegten Anforderungen genügt und dass es seinen Zweck erfüllt, sowie das Aufdecken von etwaigen Fehlern.

Testentwurf/Testentwurfsverfahren

Die Ableitung oder die Auswahl von Testfällen, basierend zum Beispiel auf der → *Spezifikation* oder der Struktur des Programmcodes. Wenn der Testentwurf nach einem planmäßigen oder einem regelbasierten Verfahren abläuft, ist dieses Verfahren das *Testentwurfsverfahren*.

Testfall

Umfasst folgende Angaben: die für die Ausführung notwendigen → *Vorbedingungen*, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts) und die Menge der vorausgesagten Sollwerte, die Prüfanweisung (wie Eingaben an das Testobjekt übergeben und Sollwerte abzulesen sind) sowie die erwarteten → *Nachbedingungen*. Testfälle werden definiert im Hinblick auf ein bestimmtes Ziel bzw. auf eine → *Testbedingung*, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen.

Test-first Programming

Vorgehensweise bei der Entwicklung von Software, die vor der Codierung das Ergebnis eines beherrschbar kleinen Implementierungsschrittes durch einen Test spezifiziert. Andere Bezeichnungen bzw. sehr ähnliche Vorgehensweisen sind Test-first Design, Test-first Development oder Test Driven Development.

Testlauf

Die Ausführung eines oder mehrerer Testfälle mit einer bestimmten Version eines bestimmten Testobjekts.

Testmittel

Alle Artefakte, die während des Testprozesses erstellt werden und die erforderlich sind, um die Tests zu planen, zu entwerfen oder auszuführen. Dazu gehören: Dokumente, Skripte, Eingabedaten, erwartete Ergebnisse, Prozeduren zum Aufsetzen und Aufräumen von Testdaten, Dateien, Datenbanken, Umgebungen und weitere zusätzliche Software- und Dienstprogramme, die für das Testen verwendet werden.

Testmodell

Ein Modell, das die → *Testmittel* beschreibt, die zum Testen einer → *Komponente* oder eines zu testenden Systems genutzt werden.

Testprozess

Umfasst alle Aktivitäten, die zur Planung und Steuerung, Analyse und Design, Realisierung und Durchführung, Auswertung und Bericht sowie zum Abschluss der Testaktivitäten in einem Projekt benötigt werden.

Testrahmen

Umfasst alle → *Testtreiber* und → *Platzhalter*, die notwendig sind, um Testfälle auszuführen. Es können auch Protokollierungs- und Auswertungsaufgaben im Testrahmen integriert sein.

Teststufe

Eine Teststufe ist eine Gruppe von Testaktivitäten, die gemeinsam ausgeführt und verwaltet werden. Zuständigkeiten in einem Projekt können sich auf Teststufen beziehen. Beispiele von Teststufen sind der Komponententest, der → *Integrationstest*, der → *Systemtest* und der → *Abnahmetest* (nach dem allg. V-Modell).

Testtreiber (driver)

Programm bzw. Werkzeug, das es ermöglicht, ein Testobjekt ablaufen zu lassen, mit Testdaten zu versorgen und Ausgaben/Reaktionen des Testobjekts entgegenzunehmen.

Testverfahren

Eine Kombination von Tätigkeiten zum systematischen Erzeugen eines Testergebnisses. Dazu gehören der → *Testentwurf* und die Testentwurfsverfahren.

Überdeckung / Überdeckungsgrad

Kriterium zur Intensität des Tests (ausgedrückt in Prozent), unterschiedlich je nach → *Testverfahren*, in der Regel durch Werkzeuge zu ermitteln.

Ursache-Wirkungs-Analyse/-Diagramm

Eine grafische Darstellung zur Organisation und Darstellung der Zusammenhänge verschiedener möglicher Ursachen eines Problems. Mögliche Gründe einer echten oder potenziellen Fehlerursache oder -wirkung sind in Kategorien und Subkategorien einer horizontalen Baumstruktur organisiert, deren Wurzelknoten die (potenzielle) Fehlerursache/-wirkung darstellt.

Usability-Test (Benutzbarkeitstest)

Testen, um zu bestimmen, inwieweit ein Softwareprodukt unter spezifizierten Bedingungen für einen Benutzer verständlich, leicht erlernbar, leicht anwendbar und attraktiv ist.

Vorbedingung

Die Vorbedingung beschreibt den Zustand eines Programms, der notwendig ist, um den nächsten Programmschritt korrekt durchführen zu können. Der Programmschritt kann eine einzelne Anweisung, aber auch der Aufruf einer → *Komponente*/Funktion sein.

Walkthrough

Manuelle, informale Prüfmethode, um Fehler, Defekte, Unklarheiten und Probleme in schriftlichen Dokumenten zu identifizieren durch eine schrittweise Präsentation eines Dokuments durch den Autor. Dient darüber hinaus dazu, Informationen zu sammeln und ein gemeinsames Verständnis des Dokumenteninhalts aufzubauen.

Whitebox-(Testentwurfs-)Verfahren / Whitebox-Test

Alle Verfahren, die zur Herleitung oder Auswahl der Testfälle Information über die innere Struktur des Testobjekts benötigen.

Zufallstest

Ein → *Blackbox-Testentwurfsverfahren*, bei dem Testfälle zufällig, eventuell unter Verwendung eines pseudozufälligen Generierungsalgorithmus, ausgewählt werden, um einem Nutzungsprofil in der Produktivumgebung zu entsprechen. *Anmerkung*: Dieses Verfahren kann u.a. zum Test nicht funktionaler Qualitätsmerkmale wie z.B. Zuverlässigkeit und Performanz eingesetzt werden.

Zustandsautomat

Ein Modell eines Verhaltens, bestehend aus einer endlichen Anzahl von Zuständen und Zustandsübergängen, ggf. mit begleitenden Aktionen.

Zustandsbasierter Test

Ein → *Blackbox-Testentwurfsverfahren*, bei dem die Testfälle unter Berücksichtigung der Zustände und (gültigen und ungültigen) Zustandsübergänge

des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (→ *Überdeckungsgrad*) anhand der Zustände und Zustandsübergänge bewertet wird.

Zweigtest

Kontrollflussbasiertes → *Whitebox-Testentwurfsverfahren*, das die mindestens einmalige Ausführung aller Ausgänge von Entscheidungen (Zweige im → *Kontrollflussgraphen*) des Testobjekts fordert.

Literaturverzeichnis

[Breymann 15] Ulrich Breymann: *Der C++-Programmierer*. Hanser Verlag, 2015.

[Daigl & Glunz 16] Matthias Daigl, Rolf Glunz: *ISO 29119 – Die Softwaretest-Normen verstehen und anwenden*. dpunkt.verlag, 2016.

[Grochtmann & Grimm 93] Matthias Grochtmann, Klaus Grimm: *Classification trees for partition testing*. Journal of Software Testing, Verification and Reliability, 3(2):63–82, John Wiley & Sons, 1993.

[Hendrickson 14] Elisabeth Hendrickson: *Explore It!*, dpunkt.verlag, 2014.

[Kuhn et al. 10] D. Richard Kuhn, Raghu N. Kacker, Yu Lei: *Practical Combinatorial Testing*, National Institute of Standards and Technology (NIST). Special Publication 800-142, 2010, *http://dx.doi.org/10.6028/NIST.SP.800-142*.

[Langr 13] Jeff Langr: *Modern C++ Programming with Test-Driven Development*. O'Reilly, 2013.

[Langr 14] Jeff Langr, *Testgetriebene Entwicklung mit C++*. dpunkt.verlag, 2014.

[Martin 09] Robert C. Martin: *Clean Code*. Prentice-Hall, 2009.

[Meyer 13] Bertrand Meyer: *Touch of Class*. Springer-Verlag, 2013.

[Rösler et al. 13] Peter Rösler, Maud Schlich, Ralf Kneuper: *Reviews in der System- und Softwareentwicklung*. dpunkt.verlag, 2013.

[Spillner & Linz 12] Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest*. dpunkt.verlag, 2012.

[Stroustrup 08] Bjarne Stroustrup: *Programming – Principles and Practice Using C++*. Pearson Education, First Printing Dec. 2008.

[URL: ACTS] Advanced Combinatorial Testing System (ACTS). *http://csrc.nist.gov/groups/SNS/acts/download_tools.html*.

[URL: Bessey et al. 10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler : *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*. CACM, Vol. 53 No. 2, 2010, S. 66-75, auch online: *http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext*.

[URL: clang] *clang: a C language family frontend for LLVM*, *http://clang.llvm.org/*.

[URL: gcc] *GCC, the GNU Compiler Collection*, *http://gcc.gnu.org/*.

[URL: googletest] *Google C++ Testing Framework*, *https://code.google.com/p/googletest*.

[URL: IBAN] *IBAN (International Bank Account Number)*, *https://de.wikipedia.org/wiki/IBAN*.

[URL: ISTQB] *ISTQB® – International Software Testing Qualifications Board*, Glossar von der Internetseite des German Testing Board, *http://glossar.german-testing-board.info/*.

[URL: leantesting] Internetseite zu diesem Buch. Sie enthält einen Link auf die Beispiele, *http://www.leantesting.de*.

Hinweis: Internetverweise unterliegen häufig Änderungen. Es kann daher sein, dass einige der angegebenen Links nach Druck des Buchs nicht mehr auffindbar sind.

Stichwortverzeichnis

Rezensieren & gewinnen!

Besprechen Sie dieses Buch und helfen Sie uns und unseren Autoren, noch besser zu werden.

Als Dankeschön verlosen wir jeden Monat unter allen neuen Einreichungen fünf dpunkt.bücher. Mit etwas Glück sind dann auch Sie mit Ihrem Wunschtitel dabei.

Wir freuen uns über eine aussagekräftige Rezension, aus der hervorgeht, was Sie an diesem Buch gut finden, aber auch was sich verbessern lässt. Dabei ist es egal, ob Sie den Titel auf Amazon, in Ihrem Blog oder bei YouTube besprechen.

Schicken Sie uns einfach den Link zu Ihrer Besprechung und vergessen Sie nicht, Ihren Wunschtitel anzugeben:
www.dpunkt.de/besprechung oder besprechung@dpunkt.de

dpunkt.verlag

dpunkt.verlag GmbH · Wieblinger Weg 17 · 69123 Heidelberg
fon: 0 62 21/14 83 22 · fax: 0 62 21/14 83 99