Kurzanleitung Testen

Dieses Dokument gibt eine Einführung in gebräuchliche Vorgehensweisen zum Testen prozeduraler und objektorientierter Programmen. Zwei Fallbeispiele verdeutlichen die Testvorbereitung für den Modultest und den anwendungsfallbezogenen Systemtest.

Inhaltsverzeichnis

1 Überblick

Wie stelle ich sicher, dass ein Programm das tut, was es tun soll? Ich muss das Programm überprüfen. Dazu gibt es im wesentlichen drei Möglichkeiten:

- **Review** Die Entwurfsdokumente und der Quellcode des Programms werden kritisch durchgesehen, ohne das Programm selbst laufen zu lassen.
- **Test** Das Programm wird mit Eingabedaten betrieben und das Verhalten des Programms wird mit dem erwarteten Verhalten verglichen.
- **Verifikation** Ohne das Programm laufen zu lassen, wird mit Hilfe formaler Methoden bewiesen, dass der Programmcode korrekt ist, d.h. das er die Spezifikation erfüllt.

Beim Testen können sowohl die Funktionalität des Programms als auch nichtfunktionale Eigenschaften, wie etwas das Verhalten unter großer Last, überprüft werden. Tests können zufällig ohne Vorbereitung, aber auch sorgfältig vorbereitet und strukturiert durchgeführt werden. Wir beschäftigen uns in diesem Dokument mit strukturierten, funktionalen Tests.

Dabei betrachten wir verschiedene Testansätze:

- Außen-/Innensicht (*black box*, *white box* und *grey box*)
- Klassifizierung der Eingabedaten (*Äquivalenzklassenbildung* und *Randwertanalyse*) und des Verhaltens (*Überprüfung von Zustandsübergängen*)
- Code-Abdeckung (*test coverage*)

und verschiedene Testebenen:

- *Modultest* (unterste Ebene: einzelne Funktionen bei prozeduralen Programmiersprachen bzw. einzelne Methoden/Klassen bei objektorientierten Programmiersprachen)
- *Integrationstest* (Zwischenebene)
- *Systemtest* und *Abnahmetest* (oberste Ebene: Gesamtprogramm)

In den Abschnitten 2 und 3 geht es um die Testvorbereitung. In Abschnitt 2 erläutern wir die verschiedenen Testansätze, in Abschnitt 3 die verschiedenen Testebenen. Abschnitt 4 erläutert schließlich die Tätigkeiten, die bei der Testdurchführung anfallen. Abschnitt 5 enthält zwei Fallbeispiele, einen für den Modultest (Klassentest), einen für den Systemtest.

2 Testansätze

In diesem Abschnitt beschäftigen wir uns mit der Testvorbereitung auf der Modultestebene (= unterste Testebene), also mit der Testvorbereitung für die kleinsten Programmeinheiten: Funktionen bei prozeduralen Programmiersprachen bzw. Methoden bei objektorientierten Programmiersprachen. Dabei lernen wir Testansätze kennen, die auch auf den weiter oben gelegenen Testebenen (Integrations-, System- und Abnahmetest) Verwendung finden. Zum Schluss dieses Abschnitts gehen wir noch darauf ein, was bei der Testvorbereitung für Klassen zu beachten ist.

Die drei Testansätze (1) „nichts testen“, (2) „chaotisch bzw. rein zufällig testen“, (3) „alles testen (*exhaustive testing*)“ sind in der Praxis unbrauchbar. Bei (1) und (2) mag das ohne weitere Diskussion einleuchten. (3) sieht auf den ersten Blick nach einem guten Ansatz aus, da wir nur bei einem vollständigen Test sicher sein können, dass wir alle Fehler aufspüren. Warum (3) dennoch ausscheidet, soll uns ein Beispiel zeigen:

Beispiel 2.1
Eine Methode `double sqrt(double a)` *soll mit sämtlichen Eingabewerten getestet werden. Die Eingabeparamter sind vom Typ 64bit-double. Die Ausgabe ist die Wurzel von* `a` *bzw. eine Fehlermeldung. Möchte man alles testen, so ist die Anzahl der Testfälle gleich der Anzahl der möglichen Eingaben. Bei 64bit-double sind* 2^{64} *(*$\approx 10^{20}$*) Zahlen nötig, um alle Eingaben abzudecken. Angenommen, es könnten 1 Mrd. Testfälle pro Sekunde durchgeführt werden, würde ein vollständiger Testdurchlauf* $10^{20}/10^9 = 10^{11}$ *Sekunden* $\approx$ *300 Mio Jahre dauern. Dieser Zeitraum ist viel zu lange und der Ansatz somit keinesfalls praktikabel.*

Wir müssen also einen sinnvollen Kompromiss finden, bei dem wir abwägen zwischen ökonomischen Rahmenbedingungen und dem Anspruch, mit Sicherheit alle Fehler aufspüren zu können. Ein solcher Kompromiss besteht auf der Modultestebene darin, zunächst aus der Außensicht (black box) mit Äquivalenzklassenbildung und Randwertanalyse die Testfälle zu finden, bei denen mit größter Wahrscheinlichkeit Fehler auftreten, und dann aus der Innensicht (white box) die Testfälle zu finden, die eine hinreichende Code-Abdeckung (test coverage) sicherzustellen. Was das im einzelnen bedeutet, werden wir gleich sehen.

Vorher noch ein paar erklärende Worte zum Begriff „Testfall“. Als Testfall bezeichnet man eine konkrete Situation, die getestet wird. Ein Testfall umfasst typischerweise folgende Information:

1. Was wird getestet?
2. Mit welchen Werten wird getestet?
3. Welche Vorbedingungen gelten?
4. Welches Verhalten wird erwartet?

Je nach Situation kann es nötig sein, weitere Information mit in einen Testfall aufzunehmen. Testfälle werden zunächst tabellarisch aufgelistet. Für die spätere (bequemere) Testdurchführung wird daraus dann ein Testskript erstellt. Es enthält jeweils die genauen Schritte inklusive Testdaten, die nötig sind, um einen Testlauf durchzuführen und später zu reproduzieren.

2.1 Black-Box-Test

Beim Black-Box-Test wird das, was getestet wird, nur von außen betrachtet. Unter Vernachlässigung aller Kenntnisse über den inneren Aufbau (Implementierung, Entwurf) werden alleine anhand der Spezifikation Eingabewerte und zugehörige erwartete Ausgabewerte festgelegt. Bei der Ausführung des Test werden dann die erwarteten mit den tatsächlichen Ausgaben verglichen. Testfälle, die auf diese Art und Weise erstellt werden, können auch dann noch verwendet werden, wenn sich die Implementierung aber nicht die Spezifikation ändert.

Betrachten wir ein Beispiel für einen Black-Box-Test:

Beispiel 2.2

Die Methode `sqrt(double a)` *besitzt folgende Spezifikation (Listing 1):*

```
1 /**
2  * @params a Wert von dem die Wurzel berechnet wird.
3  * @pre a >= 0
4  * @return returnValue * returnValue = a mit Genauigkeit +/-0.01
5  * @throws IllegalArgumentException, wenn a < 0
6  */
7 public static double sqrt(double a){
8 ...
```

Listing 1: Spezifikation der Methode `sqrt(double a)`

Für die Erstellung von Testfällen betrachten wir die möglichen Eingabewerte, also die möglichen Werte, die der Parameter `a` *annehmen kann. Da es sich um eine statische Methode handelt, gibt es keine Vorbedingungen (z.B. Werte von Attributen), die zu beachten sind.*

Anhand der Spezifikation sehen wir, dass das sich das Verhalten von `sqrt` *in Abhängigkeit des Vorzeichens von* `a` *grundlegend ändert. Dementsprechend entscheiden wir zwischen zwei Wertebereichen für* `a`*:* `a>=0` *und* `a<0`*. Schauen wir nochmal genau hin, so sehen wir aufgrund der Spezifikation, dass sich diese Wertebereiche noch weiter eingrenzen lassen. Durch den Typ* `double` *ist automatisch auch eine Untergrenze* `-MAX_DOUBLE` *und eine Obergrenze* `MAX_DOUBLE` *für die Werte, die* `a` *annehmen kann, vorgegeben. Somit erhalten wir diese zwei Wertebereiche für* `a`*:*

- *gültige Werte (Wurzelberechnung nicht möglich):* `-MAX_DOUBLE <= a < 0`
- *ungültige Werte (Wurzel wird berechnet):* `0 <= a <= MAX_DOUBLE`

Durch die Obergrenze `MAX_DOUBLE` *und die Untergrenze* `-MAX_DOUBLE` *sollten wir auch noch diese zwei Wertebereiche mit in Betracht ziehen:*

- *ungültige Werte (nicht zulässig für Datentyp* `double`*):* `a < -MAX_DOUBLE`
- *ungültige Werte (nicht zulässig für Datentyp* `double`*):* `MAX_DOUBLE < a`

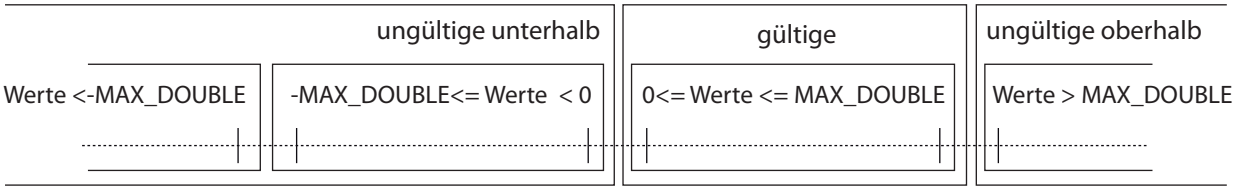

Abbildung 1: Wertebereiche für `a`

Abbildung 1 zeigt alle vier Wertebereiche und die Grenzen zwischen den Wertbereichen.

Der oben erwähnte Kompromiss besteht in diesem Beispiel darin, für jeden Wertebereich einen Vertreter aus diesem Wertebereich zu wählen (z.B. `a == 9` *aus dem Wertebereich* `0 <= a <= MAX_DOUBLE`*) und an jeder Wertebereichsgrenze Werte zu wählen (z.B.* `a == 0` *an der Grenze zwischen den Wertebereichen*

`-MAX_DOUBLE <= a < 0` *und* `0 <= a <= MAX_DOUBLE`*).*[1] *Damit erhalten wir Tabelle 1 zu sehen. Durch das Notieren der Testfälle wird gleichzeitig eine Validierung der Spezifikation durchgeführt, da noch einmal intensiv die Funktionalität des Programms durchdacht wird.*

#	Klasse/Grenze	Eingabe `a`	Ausgabe	Testdaten	erwartetes Verhalten
1	ungültig oberhalb	MAX_DOUBLE + 1	???	???	???
2	größte double	MAX_DOUBLE	$\sqrt{a}$	MAX_DOUBLE	$\sqrt{\text{MAX\_DOUBLE}}$
3	gültig	>0	$\sqrt{a}$	9.0	3.0
4	gültiger unterer Rand	0	0	0.0	0.0
5	ungültiger oberer Rand	-MIN_DOUBLE	IllegalArg	-MIN_DOUBLE	IllegalArg
6	ungültig unterhalb	<0	IllegalArg	-16.0	IllegalArg
7	kleinste double	-MAX_DOUBLE	IllegalArg	-MAX_DOUBLE	IllegalArg
8	kleiner als kleinste double	-MAX_DOUBLE - 1	???	???	???
9	betragsmäßig kleinste positive double	+MIN_DOUBLE	$\sqrt{a}$	MIN_DOUBLE	$\sqrt{\text{MIN\_DOUBLE}}$
10	betragsmäßig kleinste negative double	-MIN_DOUBLE	IllegalArg	-MIN_DOUBLE	IllegalArg

Tabelle 1: Testfälle zur Methode `sqrt(double a)`

Beim Erstellen der Tabelle stellen wir fest, dass die Testfälle 1 und 8 keine sinnvollen Tests für `sqrt(double a)` *ergeben, da wir beim Aufruf dieser statischen Methode keine Werte für* `a` *übergeben können, die außerhalb des Wertebereichs des Datentyps* `double` *liegen. Für den Test von* `sqrt(double a)` *sind die Testfälle 1 und 8 also zu verwerfen.*

Sollte der Test auf Systemtestebene (siehe Kapitel 3.3) stattfinden, dann wären die Testfälle 1 und 8 möglicherweise sinnvoll. Denn nimmt eine Oberfläche auch String-Werte entgegen, muss geprüft werden, wie sich das Programm in diesem Fall verhält. Beispiel 2.2 isoliert betrachtet kann nur auf Modultestebene getestet werden.

Die Testfälle 9 und 10 werden gerne vergessen. Beide liegen direkt am Rand 0*.*

Die in diesem Beispiel angewandten Vorgehensweisen werden als **Äquivalenzklassenbildung** (= sinnvolle Einteilung in Wertebereiche) und **Randwertanalyse** (= Auswahl von Werten an den Rändern dieser Wetebereiche) bezeichnet. Dabei haben wir bei beiden Vorgehensweisen die Spezifikation, also hier im Fall einer Methode (Funktion) die Vor- und Nachbedingungen, zugrunde gelegt.

Mit Hilfe der Äquivalenzklassenbildung kann durch eine geringe Menge an Testwerten eine hohe Testabdeckung erreicht werden. Eine vollständige Prüfung mit sämtlichen Möglichkeiten ist aufgrund der großen

[1] In Java: `MAX_DOUBLE` $\approx 10^{308}$ und `MIN_DOUBLE` $\approx 10^{-323}$.

Menge an Werten nicht möglich (siehe Beispiel 2.1 auf Seite 3). Mögliche Testwerte werden stattdessen in Klassen (= Wertebereiche) eingeteilt und es wird jeweils ein Vertreter einer solchen Klasse ausgewählt. Dabei liegt die Annahme zugrunde, dass ein Fehler für Eingabewerte aus einer Äquivalenzklasse mit großer Wahrscheinlichkeit durch einen Test mit einem einzigen Wert aus dieser Äquivalenzklasse aufgedeckt werden kann. Bei der Randwertanalyse werden die Ränder der jeweiligen Äquivalenzklassen betrachtet. Zusätzlich zu einem Vertreter der Äquivalenzklasse wird also jeweils ein Vertreter (ggfs. auch mehrere Vertreter) der Grenzbereiche ausgewählt. Der Randwertanalyse liegt die Erfahrung zugrunde, dass gerade an den Grenzen von Wertebereichen Fehler auftreten können.

In Beispiel 2.2 haben wir eine Methode mit einem Eingabeparameter und Vor-/Nachbedingungen, die sich nur auf diesen Eingabeparameter beziehen, betrachtet. Bei Methoden mit mehreren Parametern wird in der Regel zunächst für jeden Parameter einzeln die Äquivalenzklassenbildung und Randwertanalyse durchgeführt.[2] Häufig ist die Ausführung einer Methode nicht nur von den Eingabeparametern sondern auch vom Zustand des Objekts, auf dem diese Methode ausgeführt wird, abhängig. Diese Abhängigkeit spiegelt sich in den Vor-/Nachbedingungen wider. Für die dort aufgeführten Eigenschaften, die sich auf den Zustand des Objekts beziehen, wird auch einzeln die Äquivalenzklassenbildung und Randwertanalyse durchgeführt.[3]

Dann werden die so ermittelten Äquivalenzklassen und Randwerte miteinander kombiniert. Die maximal mögliche Anzahl an kombinierten Testfällen erhält man, wenn alle Testfälle aller Parameter und aller in den Vor-/Nachbedingungen aufgeführten Objekteigenschaften miteinander kombiniert werden. Damit erhält man aber zu viele kombinierte Testfälle. Die Anzahl kann deutlich reduziert werden, wenn man sicherstellt, dass jeder der einzeln ermittelten Testfälle in mindestens einem kombinierten Testfall auftaucht. Zu dieser minimalen Anzahl an kombinierten Testfällen kann man dann noch aufgrund inhaltlicher Gesichtspunkte oder aufgrund von Erfahrungswerten weitere kombinierte Testfälle hinzufügen, die mit großer Wahrscheinlichkeit Fehler aufdecken werden.[4] Das **Cause and Effect Testing**[5] ist eine (recht aufwändige) Vorgehensweise zur systematischen Auswahl kombinierter Testfälle.

Welche anderen populären Black-Blox-Verfahren gibt es noch? Da ist zum einen die **Überprüfung von Zustandsübergängen** (State Transition Testing). Darauf werden wir in Abschnitt 2.4 näher eingehen. Zum anderen ist bei der Erstellung von Testfällen auch Erfahrung hiflreich. So können Testfälle mit aufgenommen werden, die sich durch keines der o.g. Verfahren ergeben, aber dennoch mit großer Wahrscheinlichkeit Fehler im Code aufdecken können. Generell gilt: Im Zweifelsfall lieber einen Testfall mehr als einen zu wenig.

Wer sollte die Black-Box-Tests erstellen? Sie sollten möglichst durch Tester erstellt werden, die keine Kenntnisse über die internen Strukturen des Programms besitzen. So ist sicher gestellt, dass die Testkonzipierung auf Basis der Spezifikation und nicht auf Basis des Codes erfolgt.[6] Denn dafür gibt es den White-Box-Test.

[2] Sind mehrere Parameter durch Vor- oder Nachbedingungen voneinander abhängig, so kann es nötig sein, mehrere Parameter gemeinsam zu betrachten.

[3] Eine solche getrennte Betrachtung für einen Methoden-Parameter und eine Objekt-Eigenschaft findet sich im Fallbeispiel in Abschnitt 5.1. Die Resultate der getrennten Betrachtungen stehen in Tabelle 3 und Tabelle 4.

[4] Diese Vorgehensweise wird im Abschnitt 5.1 ausführlicher anhand der Methode `push` erläutert.

[5] S. [1][Seite 78 ff.].

[6] Dieses Prinzip kann auch dann eingehalten werden, wenn Entwickler Tests für den eigenen Code schreiben. Dazu ist es nur notwendig, dass sie die Tests erstellen, bevor der Code geschrieben wird. Diese Vorgehensweise wird auch als *Test-First-Ansatz* bezeichnet.

2.2 White-Box-Test

Im Gegensatz zum Black-Box-Test werden beim White-Box-Test Kenntnisse über die interne Struktur für die Erstellung des Test genutzt. Der White-Box-Test sollte als Ergänzung zum Black-Box-Test gesehen werden, um die Areale zu untersuchen, die bisher ungeprüft sind. Werden zunächst jeweils sämtliche Black-Box-Tests und sämtliche White-Box-Tests notiert, können im nächsten Schritt Dopplungen gestrichen werden. So ist eine möglichst große Testabdeckung gewährleistet. White-Box-Tests werden auch als Glas-Box-Tests bezeichnet.

Bei den White-Box-Tests werden meist vier Verfahren unterschieden. Sie bezeichnen gleichzeitig die Testabdeckung (Coverage):

- **Anweisungsüberdeckung (Statement Coverage)** Jede Anweisung des Programms wird mindestens einmal durchlaufen.
- **Zweigüberdeckung (Branch Coverage)** Jeder Zweig der Anwendung wird mindestens einmal durchlaufen. Z.B. einmal `if` und einmal `else` oder einmal `while` und einmal nicht `while`.
- **Pfadüberdeckung (Path Coverage)** Jeder mögliche Pfad der Anwendung wird mindestens einmal durchlaufen.
- **Bedingungsüberdeckung (Condition Coverage)** Jede Bedingung und jede Teilbedinung wird mindestens einmal durchlaufen und ist einmal `true` und einmal `false`.

Zur Überprüfung, welche Stufe der Testabdeckung erreicht ist, ist es hilfreich auf Tool-Unterstützung zurückzugreifen. Sogenannte Coverage-Tools bewerten die Testfälle in Relation zum Quellcode und markieren die Bereiche, die bisher unzureichend untersucht worden sind. Meist ist es angemessen, Pfadüberdeckung zu erlangen. Es bietet eine guten Kompromiss zwischen Aufwand und Abdeckung.

Beispiel 2.3

```
1 public int f(int a, int b){
2   if(a>0){ //if1
3     ...
4   }
5   if(b>0){ //if2
6     ...
7   }
8 }
```

Mit z.B. folgenden Testwerten bzw. Situationen lassen sich diese Testabdeckungen erreichen:

- *Für die **Anweisungsüberdeckung** ist nur ein Testfall nötig:* `f(1,1)`
- *Für die **Zweigüberdeckung** benötigen wir zwei Testfälle:* `f(0,0)` *und* `f(1,1)`
- *Für die **Pfadüberdeckung** brauchen wir vier Testfälle:*
  `if1, if2 mit f(1,1)`
  `!if1, if2 mit f(0,2)`
  `if1, !if2 mit f(1,0)`
  `!if1, !if2 mit f(0,0)`

- *Für die **Bedingungsüberdeckung** benötigen wir hier, wie bei der Zweigüberdeckung, nur zwei Testfälle:*
  if1, if2 für `f(0,0)`
  if1!, if2! für `f(1,1)`

Beispiel 2.4

```
1 public int g(int a, int b){
2       //cond1       cond2       cond3
3   if( (a-b)>=0 && ((a-b)<0 || (a*b)>0)) ){
4     //            <-------- cond4 -------->
5     //<---------- cond5 ------------------->
6     ...
7   }
8   ...
9 }
```

- ***Anweisungsüberdeckung:*** `g(2,1)`
- ***Zweigüberdeckung:*** `g(2,1)`, `g(-1,1)`
- ***Pfadüberdeckung:*** *hier identisch zu Zweigabdeckung.*
- ***Bedingungsüberdeckung:*** *drei Testfälle genügen, um cond1, cond2, cond3, cond4 und cond5 jeweils mindestens ein Mal wahr und ein Mal falsch werden zu lassen:*
  `cond1, cond2!, cond3, cond4, cond5 mit g(2,1)`
  `cond1!, cond2, cond3!, cond4, cond5! mit g(-1,1)`
  `cond1, cond2!, cond3!, cond4!, cond5! mit g(1,-1)`

Beispiel 2.5

```
1 public int max(int a, int b){
2   return a;
3 }
```

In diesem Beispiel wird innerhalb der Methode der Parameter `b` *aus der Signatur nicht verwendet. Hier stellt sich die Frage, ob die Methode bzw. Methodensignatur korrekt sind. Um Anweisungs-, Zweig- oder Bedinungsüberdeckung zu erreichen, würde ein Aufruf der Methode genügen. Dabei wäre es egal, welche Parameter übergeben werden. Zur Pfadüberdeckung, müsste die Methode einmal verwendet werden und einmal nicht. Bei keiner dieser Vorgehensweisen, würde auffallen, dass* `b` *als Parameter übergeben, in der Methode aber gar nicht verwendet wird. Dieses Beispiel zeigt, dass in White-Box-Tests nicht nur rein schmematisch nach Abdeckungsstrategien vorgegangen werden sollte, sondern auch durch Infragestellung des geschriebenen Codes.*[7] *Dies ist ein gewichtiger Grund, warum es sich anbietet, White-Box- und Black-Box-Tests zu kombinieren.*

[7] Wird Code intensiv durchgesehen, dann wird diese Tätigkeit auch als Code Review bezeichnet. Das Ergebnis eines Reviews ist ein Protokoll mit Anmerkungen zum Code, die danach durch den Programmierer entsprechend umgesetzt werden.

Beispiel 2.6

```
for(int i=0; i<100; i++){
  if(i>a){
    ...
  }
  ...
}
```

Die Kombination von Schleifen und Bedingungen, die zusätzlich den Schleifenzähler verwenden, erhöhen die Anzahl der Testfälle zur Pfadabdeckung nochmals. In diesem Fall existieren mehrere 100 Pfade, für die Testfälle erstellt werden müssten.

Das Beispiel 2.6 zeigt, dass die Menge der White-Box-Tests sehr schnell zu einer großen Menge an Testfällen anwachsen kann, besonders dann, wenn Pfad- oder Bedinungsüberdeckung angestrebt wird. Deshalb vereinfacht man die Regeln zur Pfadabdeckung bei Schleifen wie folgt: Bei einer festen Anzahl an Schleifeniterationen sind zwei Durchläufe nötig, genau die Anzahl und niemals. Ist die Anzahl variabel, sollten kein Durchlauf, genau ein Durchlauf und mehrere Durchläufe überprüft werden. Je nachdem, wie die Schleifenabbruchbedinung strukturiert ist, sollte für sie selbst Bedingungsabdeckung angewendet werden.

Eine Pfadabdeckung für Rekursion wird durch je einen Testfall mit einem und keinem rekursiven Absteigen erreicht. Auch das Testen von gewollt auftretenden Exceptions sollte nicht vergessen werden. Es erhöht die Anzahl der Pfade noch einmal.

2.3 Grey-Box-Tests

Black-Box- und White-Box-Tests sind klar voneinander abgegrenzt. Im einen Fall wird die Software nur von außen (Spezifikation), im anderen Fall von innen (Implementierung, Entwurf) betrachtet. Wird die Software bei der Erstellung von Testfällen sowohl von innen als auch von außen betrachtet, so spricht man von Grey-Box-Tests. Dies trifft zum Beispiel häufig auf Integrationstests (s. Abschnitt 3.2) zu.

2.4 Klassen

Klassen können für den Test wie eine Menge zusammenhängender Methoden gesehen werden, die als eine Einheit zusammen getestet werden. Das bisher für Modultests Gesagte trifft auch für Klassentest zu. Es sind jedoch ein paar Eigenheiten zu beachten.

2.4.1 Objektzustände

In die Vor-/Nachbedingungen von Methoden in einer Klasse geht in der Regel auch der Zustand des Objekts mit ein. Dementsprechend sind die Objektzustände auch in die Testvorbereitung mit einzubeziehen. Betrachten wir uns als Beispiel eine Stack-Klasse:

Beispiel 2.7

Die Klasse `MyStack` in Abbildung 2 hat eine Methode `push` zum Ablegen neuer Elemente auf dem Stapel und eine Methode `pop` zum Entnehmen des obigen Elements. Zusätzlich wird gefordert: Es dürfen maximal fünf Elemente auf den Stapel gelegt werden.

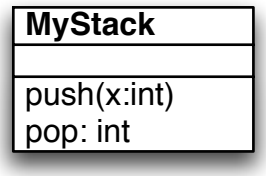

Abbildung 2: Klassendiagramm für Klasse MyStack

Das Zustandsdiagramm in Abbildung 3 veranschaulicht das Verhalten in Abhängigkeit vom Zustand eines Stapel-Objekts (Black-Box-Sichtweise): Zunächst ist der Stack leer. Es können solange Elemente mit `push`

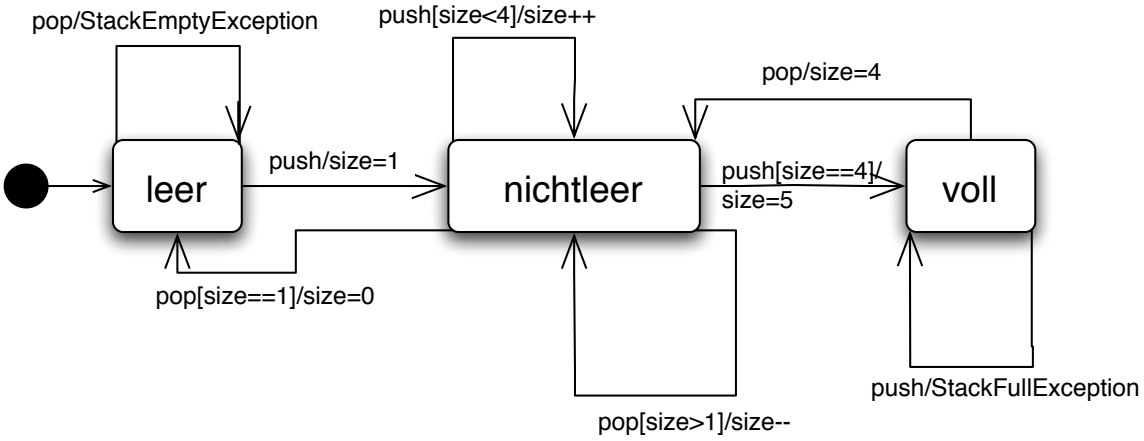

Abbildung 3: Zustandsdiagramm für Klasse MyStack

hinzugefügt werden, bis der Stack voll ist. Wenn er voll ist, dann führt der Aufruf von `push` zu einer Exception. Entsprechend führt der Aufruf von `pop` bei einem leeren Stack auch zu einer Exception. Bei einem vollen und bei einem nichtleeren Stack führt der Aufruf `pop` zur Entnahme eines Elements.

Möchten wir durch den Test sicherstellen, dass jeder Zustand mindestens einmal angenommen und dass jeder Übergang mindestens einmal durchlaufen wird, dann kommen wir auf die Testfälle gemäß Tabelle 2.

Die in Beispiel 2 durchgeführte Vorgehensweise wird als **Überprüfung von Zustandsübergängen** bezeichnet. Dabei handelt es sich um ein Black-Box-Verfahren, da nur die von außen sichtbaren Zustände und Übergänge in Betracht gezogen werden.

#	Zustand	Methode	Testdaten	erwartetes Ergebnis
1	leer	pop		Stack leer StackEmptyException
2	leer	push	x	Stack nicht leer, x ist oberstes Element
3	nicht leer, 1 Element auf Stack	pop		einziges Element wird zurückgegeben, Stack leer
4	nicht leer, mind. 2 Elemente auf Stack	pop		oberstes Element wird zurückgegeben, Stack nicht leer, 1 Element weniger auf Stack
5	nicht leer höchstens 3 Elemente auf Stack	push	x	1 Element mehr auf Stack, Stack nicht leer, x ist oberstes Element
6	nicht leer, 4 Elemente auf Stack	push	x	x ist oberstes Element, Stack voll
7	voll	push	x	Stack voll, StackFullException, Stackinhalt unverändert
8	voll	pop		oberstes Element wird zurückgegeben, Stack nicht leer, 4 Elemente auf Stack

Tabelle 2: Testfälle zur Klasse `MyStack`

Analog zu Abschnitt 2.2 können dabei auch verschiedene Verfahren zur Testabdeckung unterschieden werden:[8]

- **Zustandsüberdeckung** Jeder Zustand wird mindestens einmal durchlaufen. Dies entspricht der Anweisungsüberdeckung für White-Box-Tests.
- **Übergangsüberdeckung** Jeder Übergang wird mindestens einmal durchlaufen. Dies entspricht der Zweigüberdeckung für White-Box-Tests.
- **Pfadüberdeckung** Jeder mögliche Übergangs-Pfad der Anwendung wird mindestens einmal durchlaufen.
- **Bedingungsüberdeckung** Jede Bedingung bzw. Teilbedinung wird mindestens einmal durchlaufen und ist einmal `true` und einmal `false`.

Die so erhaltenen Testfälle können dann mit denen kombiniert werden, die wir durch die oben aufgeführten Verfahren für Modultests erhalten.

[8] In Beispiel 2.7 haben wir uns für Zweigüberdeckung entschieden.

Worauf ist sonst noch zu achten? Bei der Testdurchführung muss für jeden Testfall ggf. erst eine geeignete Testsituation hergestellt werden. Dazu ist ein Aufruf des Konstruktors und vielleicht weiterer Methoden nötig. Hier sollte nicht vergessen werden, dass auch diese fehlerhaft sein könnten und ein Fehler hierauf zurückzuführen ist. Zur Interpretation eines Testablaufs sind ggf. weitere Beobachtermethoden nötig, die Informationen über den Status des Datentyps preis geben. Außerdem sollte nach jedem Methodenaufruf die Repräsentationsinvariante überprüft werden z.B. durch den Aufruf der `repOk()`-Methode. Ein ausführliches Beispiel für den Test einer Java-Klasse wird in Abschnitt 5.1 behandelt.

2.4.2 Vererbung

In einer Vererbungshierachie sollte zunächst ein ausreichender Test für den Supertyp erstellt werden. Jede Subklasse muss mit sämtlichen Black-Box-Test der Superklasse getestet werden. Zur Testvorbeitung wird der Konstruktor der Subklasse verwendet. Sind diese Tests erfolgreich, ist sichergestellt, dass sich ein Objekt der Subklasse wie ein Objekt der Superklasse verhält (Liskovsches Substitutionsprinzip). Für neue oder überschriebene Methoden müssen eigene Black- bzw. White-Box-Tests geschrieben werden.

Abstrakte Supertypen, von denen keine Objekte erzeugt werden können, können mit Hilfe einer einfachen, konkreten Subtyp-Implementierung getestet werden. Gekoppelte Subtypen sind analog zu gekoppelten Modulen im Integrationstest (s. Abschnitt 3.2) zu behandeln und durch Teststubs (s. Abschnitt 4.1) zu ersetzten. Sollten die Interaktionen gekoppelter Datentypen überprüft werden, kann dies entweder im Integrationstest oder durch Weiterfassung eines Moduls auf mehrere Klassen, innerhalb eines Modultests erfolgen. Das ist zulässig, da die Moduleinteilung nicht auf genau eine Klasse beschränkt, sondern frei wählbar ist.

2.5 Zusammenhang zwischen den verschiedenen Testverfahren

Bei einem Modultest werden normalerweise die Äquivalenzklassenbildung, die Randwertanalyse, das Cause and Effect Testing und die Überprüfung von Zustandsübergängen für Black-Box-Tests herangezogen. Anweisungsüberdeckung, Zweigüberdeckung, Pfadüberdeckung und Bedingungsüberdeckung werden für White-Box-Tests verwendet.

Dass sich die letztgenannten vier Verfahren aber auch für Black-Box-Tests eignen, sieht man bereits daran, dass die Überdeckungsverfahren im Prinzip bei der Überprüfung von Zustandsübergängen Anwendung finden. Diese vier Verfahren können dementsprechend auch für andere Black-Box-Tests verwendet werden, z.B. auch für Black-Box-Tests auf der Systemtestebene, bei denen die Überdeckung von Aktivitätsdiagrammen oder Anwendungsfallbeschreibungen betrachtet wird.

3 Testebenen

Beim Testen werden meist drei Testebenen unterschieden (s.a. Beispiel 3.1):

- *Modultest* (unterste Ebene: einzelne Funktionen bei prozeduralen Programmiersprachen bzw. einzelne Methoden/Klassen bei objektorientierten Programmiersprachen)

- *Integrationstest* (Zwischenebene)
- *Systemtest* und *Abnahmetest* (oberste Ebene: Gesamtprogramm)

Dabei befinden sich der Systemtest und der Abnahmetest auf der gleichen Ebene. (Die Unterschiede werden kurz im Abschnitt 3.4 erläutert.)

Beispiel 3.1
Ein System besteht aus den drei Modulen A, B und C, die voneinander abhängig sind. Das Gesamtsystem nimmt Benutzereingaben entgegen und liefert Systemantworten (siehe Abbildung 4).

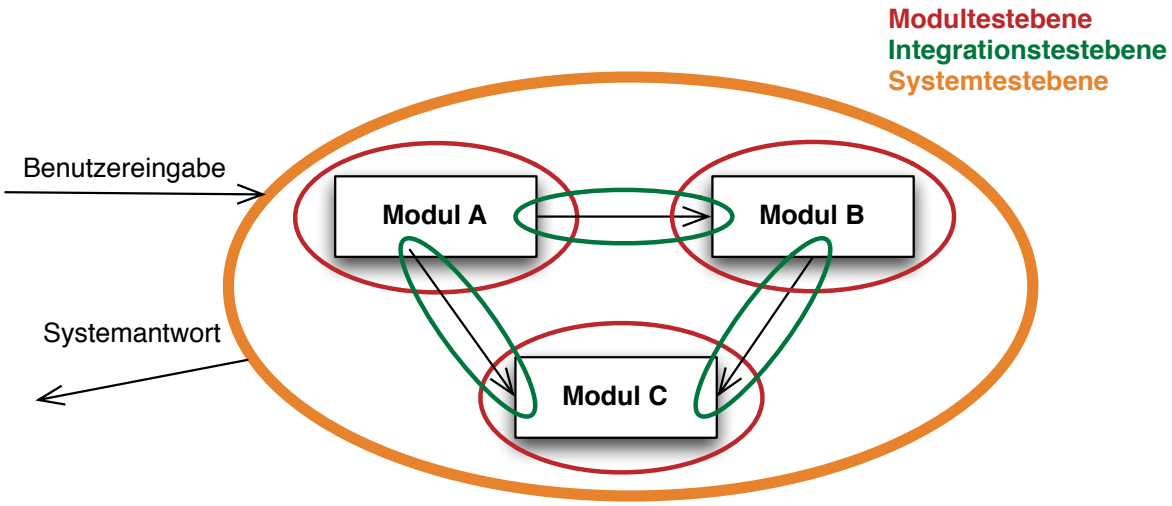

Abbildung 4: Testebenen im Gesamtsystem

Um eine Anwendung ausreichend zu testen, sollten Testfälle auf unterschiedlichen Ebenen durchgeführt werden. Dabei sollten zunächst Modultests, dann Integrationstests und erst zum Schluss Systemtests ausgeführt werden. Warum? Hält man sich an diese Reihenfolge, dann können Fehler auf unterster Ebene, nämlich auf Modulebene, sehr früh erkannt und behoben werden. Je später (in einer höheren Ebene) Fehler einer tieferen Ebene, die früher hätten bemerkt werden können, erkannt werden, desto teurer wird das Beheben sein.

3.1 Modultest

Modultests werden auch als unit tests oder component tests bezeichnet und arbeiten auf der innersten/untersten Ebene eines Softwaresystems. Im Fokus stehen Methoden, Klassen, Funktionen und Module. Es werden nur kleine Einheiten getestet. Die Abgrenzung, was zu einem Modul dazugehört, ist jeweils individuell vorzunehmen. Eine Möglichkeit ist jede Klasse als ein Modul anzusehen. Der Modultest ist im Beispiel 3.1 in Abbildung 4 jeweils innerhalb der Module A, B und C anzusiedeln.

Die Tests beschränken sich auf funktionale Aspekte und es kommen sowohl Black- als auch White-Box-Tests zum Einsatz. Fehler, die in dieser Ebene auftreten, sollten erst behoben worden sein, bevor ein Integrationstest angestrebt wird. Zu bedenken ist, dass Fehler auch auf Unklarheiten oder Fehler in der Spezifikation zurückzuführen sind. Neben den Modulen selbst wird damit auch die Spezifikation geprüft. Modultests können in Java z.B. mit JUnit[9] automatisiert werden.

Modultests können zu zwei Zeitpunkten durchgeführt werden, nach diesen sind sie auch benannt. Der „Normalfall“ ist der **Code-First-Ansatz**. Es wird zunächst der Produktivcode, also das eigentliche Programm, geschrieben und danach die Testfälle.

Der zweite Ansatz ist der **Test-First-Ansatz**. Dabei werden erst die Tests zusammen mit dem Rumpf der zu testenden Klasse(n) geschrieben. Dieser Code lässt sich kompilieren. Es schlagen jedoch alle Tests fehl. Nun wird Methode für Methode ausimplementiert. Der Codierungsfortschritt lässt sich direkt in der Zunahme erfolgreicher Tests messen. Diese Vorgehensweise hat u.a. den Vorteil, dass das Testen nicht als unliebsame Beschäftigung, die sich an das Codieren anschließt, empfunden wird. Im Gegenteil, das Testen wird als angenehmer Teil der Entwicklung empfunden, da es für Erfolgserlebnisse sorgt.

Ein ausführliches Beispiel für den Test einer Java-Klasse auf der Modultestebene wird in Abschnitt 5.1 behandelt.

3.2 Integrationstest

Integrationstests überprüfen das Zusammenspiel einzelner Systemkomponenten, die vorher einem Modultest unterzogen wurden. In Abbildung 4 auf der vorherigen Seite ist der Integrationstest durch Pfeile zwischen den Modulen symbolisiert.

Bei der Integration von Einzelkomponenten werden vier grundlegende Verfahren eingesetzt. Als **Top-Down-Integration** bezeichnet man das Vorgehen, bei dem zunächst das übergeordnete Gerüst der Komponenten erstellt wird und danach die Einzelkomponenten integriert werden. Die zweite Möglichkeit geht von unten aus und wird als **Bottom-up-Integration** bezeichnet. Hier werden zunächst die Infrastrukturkomponenten, z.B. Zugriff auf Datenbank, integriert und erst im Anschluss weitere Einzelkomponenten, die für funktionale Aspekte zuständig sind. Die dritte Variante wird als **Big-Bang-Integration** bezeichnet und bedeutet, dass sämtliche Module gleichzeitig in einem Integrationstest getestet werden. Die vierte Strategie gruppiert verschiedene Module und testet zunächst diese, bevor sie wiederum mit weiteren Gruppen gemeinsam getestet werden. Sie wird als **Cluster-Integration** bezeichnet und ist in Kombination mit Bottom-up die gebräuchlichste Form.

Es werden sowohl Black- als auch White-Box-Tests, teilweise miteinander vermischt als Grey-Box-Tests in dieser Ebene verwendet. Durch White-Box-Tests innerhalb des Integrationstests sollte zumindest Pfadabdeckung bezogen auf die Aufrufhierachie der Einzelmodule erreicht werden. Auch für Intergrationstests eignet sich der Einsatz von JUnit, solange Komponenten getestet werden, die in einer Sprache geschrieben wurden.

[9] JUnit ist eine Variante von xUnit, die für den Einsatz mit Java entwickelt wurde. Neben dieser Variante existieren auch noch weitere Implementierungen für andere Sprachen. Wenn im Folgenden von JUnit die Rede ist, dann bezieht sich dies immer auf den Einsatz mit Java. Der Einsatz in anderen Sprachen ist davon abhängig, ob eine Implementierung für diese Sprache existiert. Eine Kurzanleitung zu JUnit ist in [5] zu finden

3.3 Systemtest

Systemtests sollen möglichst realitätsnah sein und einen realistischen Betrieb der Software simulieren. In Abbildung 4 auf Seite 13 ist durch den äußeren Kreis das Gesamtsystem visualisiert. Es empfängt Benutzereingaben und liefert Systemantworten. Dementsprechend ist als Testansatz weitestgehend nur der Black-Box-Test sinnvoll, da kein Wissen über Implementierungsdetails, sondern nur das Wissen über die Spezifikation, zur Erstellung verwendet werden darf. Neben funktionalen Anforderungen werden im Systemtest meist auch nicht-funktionale Anforderungen z.B. Last- oder Performanceanforderungen überprüft. Systemtests können in der Regel nicht mit Hilfe von JUnit (allein) realisiert werden, da hier oft eine Schnittstelle, z.B. eine grafische Benutzerschnittstelle, getestet wird, auf die mit JUnit von außen nicht oder nicht geeignet zugegriffen werden kann.

Der Systemtest ist zeitlich gegen Ende des Entwicklungsprozesses anzusiedeln und dient auch als Test kurz vor der Auslieferung, für ihn sollte unbedingt **genügend Zeit** eingeplant werden. In dieser Testphase sollten möglichst keine oder nur sehr unkritische Fehler auftreten, die nicht schon in einer früheren Phase hätten erkannt werden können. Voraussetzung dafür ist, dass vorher ausreichend in der Integrations- bzw. Modultestphase getestet wurde. Häufig treten in dieser Abschlussphase dennoch kritische Fehler auf. Dadurch ist eine Verzögerung der Auslieferung nicht unwahrscheinlich, was sogar zu Vertragsstrafen führen kann. Außerdem verlängert es die Entwicklungszeit und erhöht damit die Entwicklungskosten.

Funktionale Systemtests können auf Basis der Anforderungsdefinition (z.B. Anwendungsfälle) erstellt werden. Einerseits wird diese dadurch noch einmal überprüft, andererseits wird die gesamte Anwendung überprüft. Es besteht bei der Auswahl die Schwierigkeit die richtige Menge an Testfällen zu finden. Hier ist zwischen der eigentlichen Funktionalität, die getestet werden soll, und der Menge der unterschiedlichen Testwerte zu unterscheiden. Ein systematisches Vorgehen zur Auswahl der Testfälle ist z.B. der anwendungsfallorientierte Ansatz. Dabei werden sämtliche Anwendungsfälle, die in der Anforderungsdefinition beschrieben wurden, als Testbereiche interpretiert. Zu jedem Anwendungsfall werden der Standardablauf und die Alternativabläufe überprüft. Wenn Eingabewerte und Vorbedingungen nötig sind, dann erhöht sich dadurch die Menge noch einmal. Denn auch diese müssen in ausreichendem Maße variiert werden, um ein qualifiziertes Urteil nach Abschluss der Tests treffen zu können. Eingabewerte und Vorbedingungen können mit Hilfe der Äquivalenzklassenbildung und der Randwertanalyse auf ein Minimum eingegrenzt werden.

Für einen hinreichend vollständigen Test sollten sämtliche Themenbereiche getestet werden. Trotzdem kann es sinnvoll sein zunächst nur Teilbereiche der Anwendung zu überprüfen. Wie viele Tests am Anfang durchgeführt werden, ist von der individuellen Problemstellung abhängig. Die Priorisierung der verschiedenen Testfälle unterstützt einen systematischen Aufbau des Testkonzeptes. Hier gibt es zwei grobe Vorgehensweisen:

- Die Anwendungsfälle werden nach Themenbereichen getrennt notiert. Zuerst werden die Themenbereiche getestet, die als wichtiger oder essentieller anzusehen sind.
- Die Anwendungsfälle werden nach Wichtigkeit sortiert notiert. Unabhängig vom Themenbereich wird eine Liste mit allen Anwendungsfällen notiert. Die Anwendungsfälle mit höchster Priorität sind ganz oben anzusiedeln. Zuerst werden nun die Testfälle umgesetzt, die am Anfang der Liste stehen.

Diese groben Richtungen können noch weiter verfeinert werden. Es ist auch denkbar zunächst nur den Standardablauf der Anwendungsfälle zu testen und erst im nächsten Schritt auch die Alternativabläufe zu

prüfen. Beide Konzepte können auch in Kombination verwendet werden. Dazu werden zunächst einzelne Themenbereiche gruppiert und danach wird jeweils innerhalb eines Bereiches eine Priorisierung vorgenommen.

In der Regel werden noch alle Dokumente, die zur Komplettierung der Anwendungsfalldefinition benötigt werden, mit in die Vorbereitung des Systemtests einbezogen. Ein solches Dokument ist zum Beispiel das Fachmodell.

Ein Beispiel für die Vorbereitung eines anwendungsfallbezogenen Systemtests wird in Abschnitt 5.2 behandelt.

3.4 Abnahmetest

Zeitlich nach dem Systemtest ist der Abnahme- oder Akzeptanztest anzusiedeln. Er entspricht vom Umfang her dem Systemtest, wird aber vom Kunden selbst in seiner Einsatzumgebung ausgeführt. Er dient der Übergabeprüfung durch den Kunden. Ein erfolgreich durchgeführter Abnahmetest ist oft auch mit der Bezahlung verbunden.

Werden Massentest mit vielen Kunden durchgeführt, zum Beispiel für eine Office-Suite, die in hoher Stückzahl verkauft werden soll, dann unterscheidet man zwischen:

- α-Test: Der Test wird von Kunden beim Softwarehersteller, also in dessen Räumen und auf dessen Rechnern, durchgeführt.
- β-Test: Der Test wird von Kunden in ihren eigenen Räumen und auf ihren eigenen Rechnern durchgeführt.

4 Testen

Bislang haben wir uns vor allem mit der Testvorbereitung beschäftigt. In Unterabschnitt 4.1 geben wir einen kurzen Überblick über die Tätigkeiten, die normalerweise im Rahmen der Testdurchführung anfallen, und die Werkzeuge, die zur Testunterstützung benötigt werden. Unterabschnitt 4.2 geht kurz auf eine Tätigkeit ein, die in engem Zusammenhang mit dem Testen steht: Debugging. Wir schließen diesen Abschnitt mit allgemeinen Tipps zum Testen in Unterabschnitt 4.3.

4.1 Testdurchführung

Die Durchführung eines Test gliedert sich in drei Teilschritte: Testvorbereitung, Testausführung und Testauswertung. Jeder Testdurchlauf sollte dokumentiert werden, denn nur so ist eine Bewertbarkeit und Wiederholbarkeit möglich. Eine Wiederholung der Tests ist aus zwei Gründen sinnvoll:

- Ein Fehler trat auf und soll nun lokalisiert bzw. behoben werden.

- Das Programm wurde verändert und es soll sichergestellt werden, dass sich in die unverändert gebliebenen Bestandteile kein neuer Fehler eingeschlichen hat.

Die **Testdokumentation** umfasst die Spezifikation der Testfälle, die Rahmenbedingungen, die genaue Beschreibung der zu testenden Einheit und die Ergebnisse des Tests. Nur wenn ein Testlauf ausreichend dokumentiert ist, kann der Testaufbau später auf mögliche Fehler untersucht werden.

Testwerkzeuge dienen zur Unterstützung während eines Testdurchlaufs. Besonders im Modultest liegt häufig der Fall vor, dass ein Modul ein weiters Modul benötigt. Das Beispiel 3.1 auf Seite 13 enthält drei Module, wobei Modul A von B und Modul B wiederum von C abhängig ist. Da ein Modultest aber *ein* Modul unabhängig von anderen Modulen testet, müssen die jeweils anderen simuliert werden.

Als Werkzeuge werden Testdriver und Teststub eingesetzt. Diese Werkzeuge muss sich der Entwickler i.d.R. selber schreiben. Der **Testdriver** (z.B. JUnit) ermöglicht es einen Test zu starten, in dem er die nötigen Rahmenbedingungen schafft. Der **Teststub** simuliert ein anderes Modul, das benötigt wird, um ein bestimmtes Modul zu testen. Im Beispiel 3.1 auf Seite 13 wäre ein Test-Stub für das Modul B nötig, damit das Modul A getestet werden kann.

Werden diese beiden Werkzeuge sinnvoll eingesetzt, kann der Test vollautomatisch ablaufen und regelmäßig als **Regressionstest** ausgeführt werden. Ein Regressionstest ist ein regelmäßiger, vollständiger Test, der z.B. nach Erweiterungen oder Fehlerbehebungen ausgeführt wird, um die Funktionsfähigkeit zu überwachen.

4.2 Debugging

Die Tätigkeit des Debuggings begleitet jeden Entwickler sowohl während des Programmiervorgangs als auch nachdem Fehler aus dem strukturierten Testen gemeldet worden sind. Unter Debugging versteht man zunächst einmal nur das Beheben von Fehlern im Code. Bevor ein Fehler behoben werden kann, muss erst die Fehlerursache analysiert werden. In den meisten Fällen ist es lehrreich, einfacher und auch ökonomischer, den Quellcode direkt zu analysieren.

Wenn dies nicht möglich ist, dann ist es natürlich sinnvoll, entsprechende Werkzeuge zur Fehlersuche einzusetzen. Dies ist die zweite Bedeutung des Wortes „Debugging“: Fehlersuche mit Hilfe eines geeigneten Werkzeugs, eines Debuggers. Ein Debugger begleitet die Ausführung eines Programms und zeigt dem Programmierer jeweils den aktuellen Stand des Programms und die Werte der Variablen an. Er ermöglicht ein sukzessives Durchlaufen des Codes.

Fehler, die durch strukturierte Tests ans Licht gekommen sind, können mit genau diesen Testdaten auch im Debugprozeß reproduziert werden. Der Debugvorgang, der vor einem regulären Testdurchlauf noch während der Entwicklungsphase stattfindet, sollte ebenfalls strukturiert ablaufen. Dazu ist es sinnvoll zunächst einen minimalen Satz an Testdaten zu finden, die Fehler produzieren. Davon ausgehend kann der Fehler schrittweise eingegrenzt und behoben werden, in dem Informationen gesammelt, Hypothesen aufgestellt und anschließend überprüft werden.

4.3 Tipps

Die Tätigkeit des Testens ist eine aufwendige und nicht triviale Tätigkeit. Sie sollte äußerst gewissenhaft und sorgfältig durchgeführt werden. Besser aber als Fehler im Testlauf zu erkennen ist es Fehler zu vermeiden oder schon zur Entwicklungszeit zu erkennen. Hierzu sind folgende Tipps hilfreich:

Quellcode, der als Fehlerquelle auszuschließen ist, als solchen markieren. Dieses Vorgehen wird als „Unschuldigen Code freisprechen." bezeichnet. Durch eine defensive Programmierung sind viele Fehlerquellen auszuschließen oder erleichtern zumindest das Debugging. Auch das Lesen von Quellcode, der Code Review, ermöglicht es viele Fehler im Vorfeld aufzudecken. Sehr hilfreich ist es, den Review von anderen durchführen zu lassen. Jemand, der unvoreingenommen ist, stellt eher Fehler fest, als jemand der zu sehr in den geschriebenen Quellcode eingetaucht ist. Durch ein sorgfältiges und entspanntes Vorgehen wird die Fehleranzahl verringert. Für die Konzeption von Testfällen gilt immer, lieber zunächst zu viele zu notieren und sie anschließend zu reduzieren, als Teilaspekte ungeprüft zu lassen.

Trotz all dieser Vorkehrungen, werden weiterhin Fehler gemacht. Durch ausgiebige Erkundung der Hintergründe kann für zukünftige Tätigkeiten ein **Lerneffekt** erzielt werden.

5 Fallbeispiele

In diesem Abschnitt werden zwei Fallbeispiele erläutert. Das erste Beispiel befasst sich mit einem vollständigen Modultest für eine Klasse `MyStack`. Das zweite Beispiel skizziert die Vorbereitungen für einen Systemtest auf Basis einer Anwendungsfalldefinition, wobei zur Vereinfachung auf die Darstellung und Verwendung eines zugehörigen Fachmodells verzichtet wird.

5.1 Modultest

Anhand des folgenden Beispiels wird der Vorgang zur Erstellung und Durchführung eines Modultestes erläutert.

Beispiel 5.1
Ein Java-Programm bildet eine Stack-Verwaltung ab. Der Stack akzeptiert ausschließlich positive Int-Werte. Die maximale Anzahl an Werten ist fünf. Beim Versuch, einen sechsten Wert einzufügen, wird eine Exception geworfen.

Das folgenden Listing 2 zeigt den Quellcode der Klasse `MyStack`. Die Klasse und ihre Bestandteile sind mit Hilfe von JavaDoc-Kommentaren nach [2][Seite 3] spezifiziert.

```
1  /**
2   * Objekte dieser Klasse repäsentieren Stacks, die maximal 5 positive-int-Werte aufnehmen.
3   *
4   * Mit MyStacks kann man tun, was man mit Stacks so tun kann: Elemente holen, Elemente ablegen usw.
5   * Elemente werden immer von oben geholt und oben abgelegt. Ein Element wird durch xn bezeichnet,
6   * wobei n = {0 <= n <= 4}.
7   *
8   * x4 (oben)
9   *  |
10  *  |
```

```
 *  |
 *  x0 (unten)
 * Die Klasse MyStack ist eine zustandsorientierte Klasse.
 * @author Nadja Kruemmel
 */
public class MyStack {
  /**
   * Der Stack wird intern durch einen Vector repräsentiert.
   *
   * Repräsentationsinvariante: Die Implementierung sorgt dafür, dass maximal fünf
   * Werte eingefügt werden können, und nur dann Elemente abgeholt werden dürfen,
   * wenn mindestens ein Wert auf dem Stack liegt.
   */
  private Vector<Integer> stack;

  /**
   * Konstruktor der Klasse. Erzeugt einen neues Objekt, das 5 int-Werte
   * aufnehmen kann.
   * @post Stack ist leer.
   */
  public MyStack() {
    stack = new Vector<Integer>(5);
  }

  /**
   * Legt einen neuen int-Wert >0 auf dem Stack ab.
   * @param x Wert, der auf dem Stack abgelegt werden soll
   * @pre Stack darf nicht voll sein
   * @post Das oberste Element des Stacks ist x. Die anderen Elemente des Stacks
   *  bleiben unberührt.
   * @modifies this
   * @throws IllegalArgumentException, falls x < 0 und
   *         IndexOutOfBoundsException, falls size >= capacity.
   */
  public void push(int x) {
    if (x < 0) {
      throw new IllegalArgumentException();
    }
    if (stack.size() < stack.capacity()) {
      stack.add(x);
    } else {
      throw new IndexOutOfBoundsException();
    }
  }

  /**
   * Gibt das oberste Element des Stapels zurück und löscht es.
   * @pre Stack darf nicht leer sein
   * @modifies this
   * @post oberstes Element des Stacks ist gelöscht
   * @return das oberste Element
   * @throws EmptyStackException, falls der Stack leer ist
   */
  public int pop() {
    if (stack.size() > 0) {
      int res = stack.get(stack.size() - 1);
      stack.remove(stack.size() - 1);
      return res;
    }
    throw new EmptyStackException();
  }

  /**
   * Gibt das oberste Element des Stapels zurück.
   * @pre Stack darf nicht leer sein
   * @return das oberste Element
   * @throws EmptyStackException, falls der Stack leer ist
```

```
78      */
79    public int top() {
80      if (stack.size() > 0) {
81        return stack.get(stack.size() - 1);
82      }
83      throw new EmptyStackException();
84    }
85
86    /**
87     * Liefert die momentane Größe des Stacks zurück.
88     * @return Größe des Stacks
89     */
90    public int size() {
91      return stack.size();
92    }
93
94    /**
95     * Prüft, ob der Stack leer ist
96     * @return true, wenn der Stack leer ist, sonst false
97     */
98    public boolean isEmpty() {
99      if (stack.size() == 0) {
100       return true;
101     }
102     return false;
103   }
104
105   /**
106    * Prüft, ob der Stack voll ist
107    * @return true, wenn der Stack voll ist, sonst false
108    */
109   public boolean isFull() {
110     if (stack.size() == stack.capacity()) {
111       return true;
112     }
113     return false;
114   }
115
116   /**
117    * Prüft, ob die Repräsentationsinvariante erfüllt ist
118    */
119   public boolean repOk() {
120     if (stack.capacity() >= 0 || stack.capacity() <= 5) {
121       return true;
122     }
123     return false;
124   }
125
126   /**
127    * Die überschriebene Methode toString() liefert eine String-Darstellung
128    * eines MyStack Objektes zurück.
129    */
130   @Override
131   public String toString() {
132     StringBuffer buffer = new StringBuffer();
133     buffer.append("-------------------------\n");
134     buffer.append("Size: " + stack.size() + "\n");
135     for (Integer value : stack) {
136       buffer.append(value + "\n");
137     }
138     buffer.append("-------------------------\n");
139     return buffer.toString();
140   }
141 }
```

Listing 2: Implementierung der Klasse `MyStack`

Es werden nun Black- und dann White-Box-Tests erstellt.

5.1.1 Black-Box-Tests

Black-Box-Tests werden aufgrund der Spezifikation konzipiert. Zu jeder Methode werden nun die Vor- und Nachbedingungen für die Methodenparameter und die Objekteigenschaften ausgewertet und entsprechende Testfälle erstellt. Beim Identifizieren der Testwerte kann die Technik der Äquivalenzklassenbildung und Randwertanalyse hilfreich sein.

Methode `push()` *Die einzige Methode der Klasse `MyStack`, die Inputparameter benötigt, ist die Methode `push(int x)`. Sie akzeptiert laut Spezifikation ausschließlich positive Integerwerte. Die Analyse des Programms hat zwei Äquivalenzklassen für die Eingabewerte ergeben. Diese und das Ergebnis der Randwertanalyse sind in Tabelle 3 zu finden.*

Eigenschaft	Wertebereich	Beispiel	Randwerte
positiver Integerwert	0 bis MAX_INT	5	0, MAX_INT
negativer Integerwert	-MAX_INT bis -1	-5	-1, -MAX_INT

Tabelle 3: Äquivalenzklassen der Eingabe-Parameter

Da diese Testsfälle auf Modultestebene ausgeführt werden, ist es nicht sinnvoll mit Werten ausserhalb des Wertebereichs von Integer zu testen, da diese Zahl nicht erzeugt werden könnte. Anders sieht es z.B. bei Systemtests aus. Dort sollte über den Wertebereich hinaus getestet werden, in diesem Fall MAX_INT+1 und -MAX_INT-1.

Dieselben Verfahren können zur Bestimmung der Vorbedingungen genutzt werden. Laut Spezifikation können weitere Elemente nur in einen nicht vollen Stack eingefügt werden. Daraus ergeben sich zwei Äquivalenzklassen (Tabelle 4).

Eigenschaft	Wertebereich	Beispiel	Randwerte
Stack nicht voll	$0 <= \text{Size} < 5$	Size: 3, Werte:[5, 6, 3]	Size: 0, 1, 4, 5
Stack voll	$\text{Size} = 5$	Size: 5, Werte:[1, 4, 5, 6, 3]	Size: 5

Tabelle 4: Äquivalenzklassen der Stackbefüllung

Die Äquivalenzklassen wurden auf Basis möglicher Objektzustände gebildet. Für die Methode `push()` sind zwei Zustände relevant: „Stack nicht voll“ und „Stack voll“. Durch diese Art der Betrachtung ergibt sich für die Äquivalenzklasse „Stack voll“, dass der Randbereich gleichzeitig auch das einzige Element der Klasse ist. Eine alternative Betrachtung des Sachverhaltes wäre, den möglichen Wertebereich von n zugrunde zu legen. Daraus ergibt sich eine Äquivalenzklasse deren oberer Rand 5 auch als Testwert betrachtet und somit auch die Situation „Stack voll“ abdecken wird.

Die Überprüfung von Zustandsübergängen führt uns auf die Testfälle aus Beispiel 2. Dies gibt im Vergleich zu Tabelle 4 keinen neuen Testfälle.

Anhand der Äquivalenzklassenbildung und des Randwertanalyse konnten Vorbedingungen und Eingabe-Parameter bestimmt werden. Es wurde jeweils eine gültige (positiver Integerwert, Stack nicht voll) und eine ungültige (negativer Integerwert, Stack voll) Äquivalenzklasse gefunden.

Die maximal mögliche Anzahl an kombinierten Testfällen erhält man, wenn alle Kombinationen aus den Testfällen in Tabelle 3 und Tabelle 4 gebildet werden. Dann ergeben sich wie hier im Beispiel 24 kombinierte Testfälle = 6 Eingabewerte ∗ 4 Vorbedingungen. Stellt man nur sicher, dass jeder Testfall aus beiden Tabellen mindestens ein Mal getestet wird, dann lässt sich die Menge an kombinierten Testfällen deutlich reduzieren. Man kann auch so vorgehen, dass man aus der maximal möglichen Menge an Testfallkombinationen diejenigen Paare aus Vorbedingung und Eingabeparameter streicht, die als redundant anzusehen sind.

Zunächst wird die Vorbedingung Size = 3, sie ist ein Vertreter der gültigen Äquivalenzklasse „Stack nicht leer und nicht voll“, mit sämtlichen Testfällen für den Eingabeparameter zusammen verwendet. Hieraus ergeben sich die Testfälle 1 bis 6. Die weiteren Vorbedingungen für die Stackbefüllung (Size = 0, 4 und 5) werden jeweils nur mit einem zulässigen Vertreter des Eingabeparameters kombiniert getestet (Testfälle 7 bis 9). Tabelle 5 fasst die sich daraus für die Methode `push(int x)` ergebenden Testfälle zusammen.

#	Methode	Vorbedingung	Parameter	erwarteter Zustand
1	`push(int x)`	Size: 3	5	Size: 4, bisherige Werte unverändert, oberstes Element = 5
2			0	Size: 4, bisherige Werte unverändert, oberstes Element = 0
3			MAX_INT	Size: 4, bisherige Werte unverändert, oberstes Element = MAX_INT
4			-5	IllgalArgumentException
5			-1	IllgalArgumentException
6			-MAX_INT	IllgalArgumentException
7		Size: 0	10	Size: 1, bisherige Werte unverändert, oberstes Element = 10
8		Size: 1	11	Size: 2, bisherige Werte unverändert, oberstes Element = 11
9		Size: 4	12	Size: 5, bisherige Werte unverändert, oberstes Element = 12
10		Size: 5	13	IndexOutOfBoundsException

Tabelle 5: Black-Box-Testfälle zur Methode `push(int x)`

Restliche Methoden *Die Testfälle für die anderen Methoden der Klasse werden analog erstellt. Allerdings ist eine erneute Betrachtung der Eingabeparameter nicht notwendig, da die weiteren Methoden keine Eingaben verlangen. Es genügt jeweils die Testfälle aus Tabelle 4 zugrunde zu legen. Bei manchen Methoden kann die Anzahl dieser Testfälle noch weiter reduziert werden. Z.B. ist für die Methode `isFull()` nur relevant, ob der Stack voll oder nicht voll ist. Der Stack ist auch dann nicht voll, wenn er leer ist. D.h. für diese Methode wären nur zwei Äquivalenzklassen nötig gewesen. Es ist aber zulässig feinere Klassen als nötig zu bilden, da damit auch die gröberen Klassen abgebildet werden.*

Die Visualisierung der Äquivalenzklassen aus Tabelle 6 ist in Abbildung 5 zu sehen. Sie zeigt jeweils, dass die Randwerte direkt an die Grenzen der jeweiligen Klassen reichen. Da der Wertebereich für „Stack leer“ und „Stack voll“ genau eins beträgt ist der Repräsentant der Klasse identisch mit dem Randbereich. Daraus ergeben sich fünf Testwerte für die Vorbedingungen.

Eigenschaft	**Wertebereich**	**Beispiel**	**Randwerte**
Stack nicht voll und nicht leer	$0 < \text{Size} < 5$	Size: 3, Werte:[5, 6, 3]	Size: 0, 1, 4, 5
Stack voll	$\text{Size} = 5$	Size: 5, Werte:[1, 4, 5, 6, 3]	Size: 5
Stack leer	$\text{Size} = 0$	Size: 0, Werte:[ ]	Size: 0

Tabelle 6: erweiterte Äquivalenzklassen der Vorbedingungen

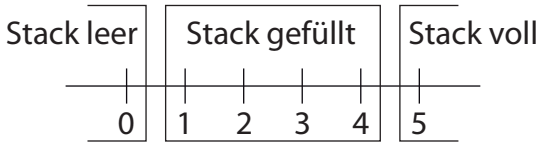

Abbildung 5: Visualisierung der Äquivalenzklassen aus Tabelle 6

Tabelle 7 ergänzt zur Tabelle 5 auf der vorherigen Seite die bisher noch fehlenden Black-Box-Testfälle. Als Vorbedingungen werden jeweils die obigen fünf Werte (3, 1, 4, 5, 0) als Repräsentanten der Äquivalenzklassen verwendet.

Nach dem Aufschreiben der Testfälle wird für die spätere Testdurchführung ein konkretes Testskript erstellt. Es stellt für jeden Testfall durch Methodenaufrufe zunächst die gewünschte Vorbedingung her und ruft dann die zu testende Methode auf. Listing 3 auf der nächsten Seite setzt den Testfall #1 aus Tabelle 5 auf der vorherigen Seite beispielhaft in JUnit-Code um.

#	**Methode**	**Vorbedingung**	**Parameter**	**erwarteter Zustand**
11	`pop()`	Size: 3, oberstes Element x	-	Size: 2, oberster Wert gelöscht, bisherigen Werte unverändert, return = x
12		Size: 1, oberstes Element x	-	Size: 0, oberster Wert gelöscht, bisherigen Werte unverändert, return = x
13		Size: 4, oberstes Element x	-	Size: 3, oberster Wert gelöscht, bisherigen Werte unverändert, return = x
14		Size: 5, oberstes Element x	-	Size: 4, oberster Wert gelöscht, bisherigen Werte unverändert, return = x
15		Size: 0	-	EmptyStackException

#	Methode	Vorbedingung	Parameter	erwarteter Zustand
16	top()	Size: 3, oberstes Element x	-	Size: 3, bisherigen Werte unverändert, return = x
17		Size: 1, oberstes Element x	-	Size: 1, bisherigen Werte unverändert, return = x
18		Size: 4, oberstes Element x	-	Size: 4, bisherigen Werte unverändert, return = x
19		Size: 5, oberstes Element x	-	Size: 5, bisherigen Werte unverändert, return = x
20		Size: 0	-	EmptyStackException
21	size()	Size: 3	-	Size: 3
22		Size: 1	-	Size: 1
23		Size: 4	-	Size: 4
24		Size: 5	-	Size: 5
25		Size: 0	-	Size: 0
26	isEmpty()	Size: 3	-	false
27		Size: 1	-	false
28		Size: 4	-	false
29		Size: 5	-	false
30		Size: 0	-	true
31	isFull()	Size: 3	-	false
32		Size: 1	-	false
33		Size: 4	-	false
34		Size: 5	-	true
35		Size: 0	-	false
36	MyStack()	-		Stack ist leer, Kapazität = 5

Tabelle 7: Black-Box-Testfälle zur Klasse `MyStack`

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class MyStackTest {
5    @Test
6    public void pushMyStack(){
7      MyStack stack = new MyStack();
8      // Stack mit 3 Werten befüllen, Werte dürfen beliebig sein
9      stack.push(43);
10     stack.push(2);
11     stack.push(7);
12
13     // Vorbedingung size = 3 erfüllt. Test für push mit Parameter = 5 durchführen
14     stack.push(5);
15
16     // Erwartetes Ergebnis überprüfen: size == 4 ?
17     assertEquals(4, stack.size());
18
19     // oberstes Element = 5
20     assertEquals(5, stack.top());
21   }
22 }
```

Listing 3: Testcode mit JUnit für den Testfall #1 zur Klasse `MyStack`

5.1.2 White-Box-Tests

zeichnen sich dadurch aus, dass sie mit Wissen über den Quellcode erstellt werden. Zumindest die Stufe „Zweigüberdeckung“ besser noch „Pfadüberdeckung“ sollte erreicht werden, um eine ausreichende Testabdeckung zu gewährleisten. Der Quellcode muss nun darauf untersucht werden, welche White-Box-Testfälle nötig sind.

Im Folgenden werden beispielhaft für die Methode `push(int x)` White-Box-Tests erstellt, mit denen Pfadabdeckung erreicht wird.

In Listing 4 wird ein Teil des Quellcodes der Klasse `MyStack` wiederholt.

```
20 public void push(int x) {
21   if (x < 0) {
22     throw new IllegalArgumentException();
23   }
24
25   if (stack.size() < stack.capacity()) {
26     stack.add(x);
27   } else {
28     throw new IndexOutOfBoundsException();
29   }
30 }
```

Listing 4: Ausschnitt aus der Klasse `MyStack`

Die Analyse ergibt, dass drei Pfade möglich sind: Pfad 1 wird durchlaufen, wenn x<0 ist. Es werden Zeile 21 und 22 ausgeführt. Pfad 2 wird durchlaufen, wenn die Größe des Stacks gleich oder größer als die Stackgesamtkapazität ist. Es werden die Zeilen 21, 25, 27 und 28 ausgeführt. Pfad 3 ist der Normalfall ohne Fehlerbehandlung. Es wird ein Element in den Stack eingefügt, da der Wert von x >= 0 ist und noch genügend Platz im Stack war. Es werden die Zeilen 21, 25 und 26 ausgeführt. Auf Basis dieser drei Abläufe müssen nun Testfälle mit passenden Testwerten aufgestellt werden.

In Tabelle 8 sind drei Testfälle zu sehen. Durch Testfall 37 wird Pfad 1, durch 38 Pfad 2 und durch 39 wird Pfad 3 abgedeckt. Vergleicht man nun die White-Box-Testfälle aus Tabelle 8 zur Methode `push()` mit denen der Black-Box-Testfälle aus Tabelle 5 auf Seite 22 wird man feststellen, dass die Testfälle aus Tabelle 8 schon in Tabelle 5 enthalten sind. Testfall 37 ist äquivalent zu Testfall 5, Testfall 38 zu Testfall 10 und Testfall 39 zu Testfall 1. Die drei White-Box-Testfälle können somit gestrichen werden.

#	Methode	Vorbedingung	Parameter	erwartetes Verhalten
37	`push(int x)`	beliebig	-1	IllegalArgumentException
38		Size: 5	beliebig	IndexOutOfBoundsException
39		Size: 3	5	Size: 4, bisherigen Werte unverändert, oberstes Element = 5

Tabelle 8: White-Box-Testfälle zur Methode `push(int x)`

Tabelle 9 auf der nächsten Seite enthält der Vollständigkeit halber die übrigen White-Box-Testfälle zur Pfadüberdeckung. Auf eine Erläuterung der Pfade wurde verzichtet. Vergleicht man diese mit Tabelle 7 auf der vorherigen Seite, wird man feststellen, dass sämtliche Testfälle aus Tabelle 9 schon in Tabelle 7 enthalten sind.

#	Methode	Vorbedingung	Parameter	erwartetes Verhalten
40	`pop()`	Size: 0	-	EmptyStackException
41		Size: 5, oberstes Element x	-	Size: 4, oberster Wert gelöscht, bisherigen Werte unverändert, return = x
42	`top()`	Size: 0	-	EmptyStackException
43		Size: 5, oberstes Element x	-	Size: 5, bisherigen Werte unverändert, return = x
44	`size()`	Size: 3	-	Size: 3
45	`isEmpty()`	Size: 5	-	false
46		Size: 0	-	true
47	`isFull()`	Size: 0	-	false
48		Size: 5	-	true
49	`MyStack()`	Size: 5	-	true

Tabelle 9: White-Box-Testfälle zur Methode `push(int x)`

In diesem Fall hat die Aufstellung der White-Box-Testfälle keine Neuerungen ergeben. Trotzdem war es gut sie aufzustellen, da so sicher überprüft worden ist, ob wirklich eine Pfadüberdeckung durch die Black-Box-Testfälle erreicht wurde.

Anstatt die White-Box-Tests manuell zu erstellen (Das ist besonders bei größeren Projekten sehr zeitaufwendig.) kann mit Hilfe so genannter Coverage-Tools[10] *die Testabdeckung der bisherigen Testfälle untersucht werden. Je nachdem welche Testabdeckungsstufe erreicht werden soll, wird der zu testende Programmcode daraufhin untersucht und entsprechend markiert. Im Anschluss daran kann der Testentwickler die bisher ungetesteten Stellen durch separate White-Box-Tests überprüfen. Auch auf diese Weise ist eine vollständige Testabdeckung zu erreichen.*

5.2 Systemtest

In diesem Unterabschnitt betrachten wir anhand einer einfachen MP3-Anwendung in groben Zügen, wie bei der Vorbereitung eines anwendungsfallbezogenen Systemtests vorzugehen ist. Dabei konzentrieren wir uns auf das Verständnis des prinzipiellen Ablaufs. So verzichten wir z.B. der Einfachheit halber auf ein zugehöriges Fachmodell und gehen davon aus, dass die funktionale Anforderungen nur durch Anwendungsfallbeschreibungen vorgegeben sind.

Im ersten Schritt der Testerstellung muss eine Priorisierung vorgenommen werden. In diesem Fall entscheiden wir uns zunächst dazu die Anwendungsfälle der MP3-Verwaltung nach Themenbereichen zu sortieren. Innerhalb eines Themenbereiches sind Anwendungsfälle mit höherer Priorität oben im Stapel zu finden. Für eine erste Auswahl an Testfällen werden nun auf horizontaler Ebene jeweils die oberen Testfälle aus den einzelnen Themenbereichen ausgewählt. Dieses Vorgehen ist in Abbildung 6 auf der nächsten Seite zu sehen.

Außerdem sollte eine inhaltliche Priorisierung vorgenommen werden, sie ergibt sich jeweils aus der konkreten Anwendungsdomäne. Um in unserem Beispiel zu bleiben: Es ist nur sinnvoll ein Album zu verändern oder

[10] Eine Open-Source-Implementierung ist z.B. Cobertura und unter http://cobertura.sourceforge.net zu finden.

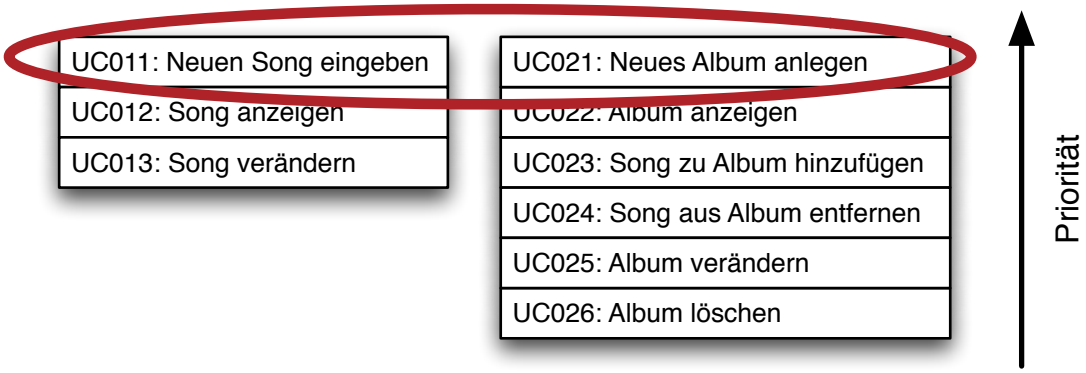

Abbildung 6: Ausschnitt der Anwendungsfälle der Song-Verwaltung

zu löschen, wenn vorher der Test des Anwendungsfalls „Neues Album anlegen“ erfolgreich wahr. Weiterhin ist es nur dann möglich Alben anzulegen, wenn man auch Songs anlegen kann und dies getestet hat.

An dieser Stelle noch der Hinweis, dass eine Software nicht nur in Teilbereichen getestet werden muss, um sie als funktionsfähig einzustufen. Wie umfangreich die Testkonzepte sein müssen, hängt individuell vom Einzelprojekt ab. Wichtig ist sich an entsprechende Absprachen mit dem Auftraggeber zu halten und möglicherweise eine Kosten-Nutzen-Aufstellung zu machen.

5.2.1 Testfallbeschreibung

In diesem Abschnitt betrachten wir beispielhaft einen Anwendungsfall. Er dient uns als Grundlage für unsere Testfallbeschreibung.

Beispiel 5.2
Für eine MP3-Anwendung sind verschiedene Anwendungsfälle definiert. Im Folgenden wird der Anwendungsfall UC021 beschrieben. Im Anschluss daran erstellen wir eine Testfallbeschreibung.

1) ***Name und Identifier*** *Neues Album anlegen, UC021*
2) ***Ziel*** *Das erworbene Album ist im System mit allen gewünschten Informationen hinterlegt.*
3) ***Beteiligte Akteure (actors)*** *Der Anwender. Natürlich könnten an dieser Stelle weitere Systeme (z.B. beim Onlinekauf ein Bezahlungssystem etc.) involviert sein, wir möchten den Anwendungsfall an dieser Stelle aber so einfach wie möglich halten.*
4) ***Status*** *Z.B. „Im Review“.*
5) ***Verwendete Anwendungsfälle (includes)*** *Der vorliegende Anwendungsfall verwendet den Anwendungsfall „Neuen Song eingeben“ (UC011) um die einzelnen Tracks, welche auf dem Album enthalten sind, in das System einzutragen.*

6) ***Auslöser (rationale oder trigger)*** *Kauf / Erhalt eines neuen Albums.*

7) ***Vorbedingungen (preconditions)*** *Keine*

8) ***Nachbedingung / Ergebnis (postconditions)***

Erfolg: Das System enthält die Informationen über das Album sowie die darauf enthaltenen Songs.

Misserfolg: Das System ist in dem Zustand, in welchem es sich vor dem Anwendungsfall befand.

9) ***Standardablauf (normal flow)***

1. *Anwender öffnet den Dialog für die Funktion „Neues Album anlegen".*
2. *Anwender gibt alle Informationen zum Album selbst ein (Titel, Interpret).*
3. *Anwender gibt alle auf dem Album vorhandenen Songs der Reihe nach ein (UC011).*
4. *Anwender überprüft die Eingaben auf Vollständigkeit und Richtigkeit und bestätigt beides.*

10) ***Alternative Ablaufschritte (alternative flow)***

2a. Ein Album mit dem gleichen Titel ist bereits vorhanden. Korrektur durch Anwender[11] oder Abbruch.

3a. Ein Song ist bereits im System vorhanden (Woran erkennt man dies?)[12]. Das System bietet an, den vorhandenen Song als Track auf der CD aufzunehmen[13].

Welche Wege durch die Anwendungsfallbeschreibung werden betrachtet? *Für eine vollständige Testvorbereitung ist es notwendig, nicht nur ein mögliches Szenario für den Standardablauf zu beschreiben, sondern eines der in Abschnitt 4 genannten Überdeckungsverfahren auf die Anwendungsfallbeschreibung anzuwenden, um auch alle relevanten Verzweigungen/Pfade/Bedingungen mit einzubeziehen. Die hier beschriebene Variante entspricht einer Mischung aus Anweisungsüberdeckung und Zweigüberdeckung: Es wird im wesentlichen sichergestellt, dass jeder Anwendungsfallschritt mindestens einmal durchgeführt wird. Zusätzlich werden noch Verzweigungen betrachtet, die sich bereits aus der Äquivalenzklassenbildung und Randwertanalyse ergeben.*

Welche Eingabewerte werden benötigt? *Als Eingabewerte sind der Titel und der Interpret des Albums nötig. Außerdem werden Daten für den Anwendungsfall UC011 benötigt. Die Daten eines Songs bestehen aus Name, Interpret (optional), Länge und Genre.*

Welche Vorbedingungen werden betrachtet? *Da unter 7) in der Anwendungsfallbeschreibung keine Vorbedingungen notiert sind, könnte der Eindruck entstehen, dass es keine Vorbedingungen gibt. Hier sind zwei Bedeutungen zu unterscheiden: Der Anwendungsfall verlangt keine besonderen Bedingungen, damit er ausgeführt werden darf. Trotzdem können unterschiedliche Bedingungen vorherrschen, wenn der Anwendungsfall ausgeführt werden soll. Eine sinnvolle Unterscheidung sollte zwischen keine und n ($n > 0$) Alben im System getroffen werden.*

[11] Z.B. kann er einen alternativen Namen wählen.
[12] Diese Klammer zeigt, dass hier eine Frage offen geblieben ist. Diese sollte mit dem Experten (Kunde) geklärt werden.
[13] Diese Ausnahme müsste bereits im Anwendungsfall „Neuen Song eingeben" (UC011) durchdacht werden

Wie lauten die Testfääe? *Wenn man die Frage nach dem Überdeckungsverfahren, den Eingabewerten und Vorbedingungen beantwortet hat, lassen sich daraus die Testfälle entwickeln. Mit Hilfe der Äquivalenklassenbildung lassen sich für sämtliche variablen Bedingungen Wertebereiche notieren. Für unser Beispiel sind sie in Tabelle 10 notiert. Beim Aufschreiben der Testfälle fällt auf, dass die vorliegenden Angaben nicht vollständig sind. So ist z.B. noch nicht festgelegt, was „normale Länge“ oder „maximale Länge“ bedeutet. Solche offenen Punkte nimmt man am besten in eine Liste offener Punkte auf und sorgt dafür, dass diese geklärt und in den entsprechenden Dokumenten nachgetragen werden. Auf diese Art dient die Systemtestvorbereitung auch gleichzeitig als Review.*

#	Bedingung	Ausprägung
1	Anzahl der Alben	0
2	Anzahl der Alben	<0
3	Titel	normale Länge
4	Tiel	minimale Länge
5	Titel	maximale Länge
6	Titel	zu lang
7	Titel	zu kurz
8	Titel	kein
9	Interpret	normale Länge
10	Interpret	minimale Länge
11	Interpret	maximale Länge
12	Interpret	zu lang
13	Interpret	zu kurz
14	Song	vorhanden
15	Song	nicht vorhanden
16	Anwendungsfall UC011	erfolgreich
17	Anwendungsfall UC011	nicht erfolgreich

Tabelle 10: Testfälle zum Anwendungfall UC021

Die Testfälle können nun mit Werten als konkrete Testfallbeschreibungen umgesetzt werden. Konkrete Werte erlangt man durch Äquivalenzklassenbildung und Randwertanalyse. Es ist möglich verschiedene Bedingungen, die nicht im Widerspruch zueinander stehen, innerhalb eines Testschritts zu behandeln.

Testschrittbeschreibung mit Testwerten für den Standardablauf von Anwendungsfall UC021 *Hier wird beispielhaft eine Abfolge von Testschritten beschrieben, die den Standardablauf mit* normalen *Werten zeigt.*

1. *Anwender öffnet den Dialog für die Funktion „Neues Album anlegen“.*
2. *Anwender gibt folgende Informationen zum Album selbst ein:*
   Titel: „Black in black“
   Interpret: „AC/DC“
3. *Anwender gibt alle auf dem Album vorhandenen Songs der Reihe nach ein (UC011):*
   1. Song:

Titel: „Hells bells“
Interpret: „AC/DC“
Länge: „2:55“
Genre: „Hardrock“
2. Song:
Titel: „Black in black“
Interpret: „AC/DC“
Länge: „3:15“
Genre: „Hardrock“

4. *Anwender überprüft die Eingaben auf Vollständigkeit und Richtigkeit und bestätigt beides.*

Der Standardablauf wird nun mit einem zweiten, anderen Album wiederholt. Dadurch hat man sowohl den Fall „Keine Alben im System“ als auch „n-Alben im System“ abgedeckt.

Die Testfallbeschreibung selbst orientiert sich sehr nah an den Anwendungsfällen. Durch Analyse der Beschreibung lassen sich die Eingabewerte und Vorbedingungen erkennen. Hierfür werden sinnvolle Testwerte notiert. Sinnvoll bedeutet an dieser Stelle, dass es möglichst realistische Werte sein sollten. Es bietet sich an in dieser Situation Informationen zu verwenden, die der Kunde schon in der Analysephase geliefert hat. Diese Testdaten können dann um weitere Daten ergänzt werden, die z.B. mit Hilfe der Äquivalenzklassenbildung gefunden wurden. Das Gleiche lässt sich auf die Vorbedingungen übertragen. Es hat sich bewährt zunächst den Standardablauf und dann die Alternativabläufe zu betrachten. Häufig sind Testdaten, die Fehler im Standardablauf hervorrufen würden, durch einen Alternativablauf schon abgedeckt.

5.2.2 Testdurchführung

Nachdem ausreichend Testdaten gesammelt wurden, wird zunächst der Standardablauf des Anwendungsfalls durchgespielt. Wurde er mit den Testdaten durchgeführt, werden die alternativen Abläufe getestet. Jede Durchführung eines Testablaufs sollte durch ein Protokoll dokumentiert werden. Denn nur so ist sichergestellt, dass Erfolg und Misserfolg mit den Ergebnissen weiterer Tests verglichen werden können.

Die Testdurchführung hat zwei Ziele: Erstens die Erkennung von Fehlern, um sie im Anschluss zu beheben und durch eine erneute Testdurchführung als behoben zu bestätigen. Zweitens einen Beleg gegenüber dem Kunden bzw. dem Vorgesetzten zu erstellen, dass man systematisch getestet hat. Beide Ziele können nur erreicht werden, wenn die Planung und Durchführung protokolliert wurde.

6 Literatur

Zur vertiefenden Lektüre sind [1] als sehr ausführliches Werk mit vielen Verweisen auf weitere Literatur und [6] mit einem Schwerpunkt auf das Testen im objektorientierten Umfeld zu empfehlen. Der Teil der xUnit-Einführung ist allerdings veraltet. Mittlerweile setzt man JUnit 4 ein, was einige grundlegender Veränderungen brachte. Im Mittelpunkt von [3] steht die Softwareprüfung, die das Testen mit umfasst. Nach wie vor lesenswert, wenn auch schwer zu bekommen, ist [4].

Literatur

[1] Ilene Burnstein. *Practical Software Testing*. Springer, 2003.

[2] B. Franzen, B. Igler, T. Letschert, B. Renz & N. Krümmel. *Java-Codierrichtlinien für den Fachbereich MNI*. FH-Gießen-Friedberg, 2007.

[3] Karol Frühauf, Jochen Ludewig & Helmut Sandmayr. *Software-Prüfung*. vdf, 2000.

[4] Glenford J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[5] Projektgruppe Softwaretechnik. *Kurzanleitung JUnit*. FH-Gießen-Friedberg, 2008.

[6] Uwe Vigenschow. *Objektorientiertes Testen und Testautomatisierung in der Praxis*. dpunkt.verlag, 2005. URL `www.oo-testen.de`.

Autoren: Nadja Krümmel & Bodo Igler, Institut für SoftwareArchitektur.