Ulrich Breymann
(breymann@hs-bremen.de)
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 Autor zu den Themen Programmierung in C++, C++ Standard Template Library (STL) und Java ME (Micro Edition).

Andreas Spillner
(andreas.spillner@hs-bremen.de)
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.

„Lean Testing"

Entwickler testen ihre Software, bevor diese eingecheckt oder verwendet wird. Die Frage ist, wie umfangreich muss der Test sein? „Lean Testing" ist ein Ansatz, der den Entwickler dabei unterstützt, einen angemessenen und nicht zu aufwendigen Test durchzuführen. Der Entwickler soll sich beim Testen auf das Wichtige konzentrieren und dabei systematisch vorgehen. An zwei konkreten Beispielen wird dieser Ansatz verdeutlicht.

Fehlerreduktion zu angemessenen Kosten

Das Ziel eines jeden Softwareentwicklers* ist es, 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. Aber: Wie prüfe ich mein Programm(teil) auf Fehler und wie groß darf ein vertretbarer Testaufwand sein?

Es ist möglich, die Anzahl der Fehler zu reduzieren. Dabei helfen erstens konstruktive Maßnahmen. Dazu gehört die Einhaltung von Programmierrichtlinien ebenso wie das Schreiben eines verständlichen, wartungsfreundlichen Programmtextes (Stichwort *Clean Code*). Zweitens hilft das Testen, also die Prüfung der Software, ob sie den Anforderungen genügt und ob sie Fehler enthält.

Wie viel Aufwand soll in den Test gesteckt werden? 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 aus dem agilen Umfeld bekannte Begriff „lean" bedeutet beim Testen, sich auf das Wichtige zu konzentrieren, um diesen Kompromiss zu erreichen.

Die Frage des Aufwands ist aber nur vordergründig ausschließlich für Tester von Bedeutung – sie betrifft auch Softwareentwickler, da sie auch testen. Manche Softwareentwickler meinen, sie müssten ein Programm *austesten* oder gar nicht testen, im Vertrauen auf ihre sehr guten Programmierfähigkeiten. Jeff Langr schreibt beispielsweise „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." [Lan13] Keine Motivation für Entwickler, sich mit dem Testen zu beschäftigen.

„Lean Testing" [Spi16] steht dagegen für einen Ansatz, der

- auf der einen Seite alle wichtigen Testfälle zur Prüfung der Software berücksichtigt,
- auf der anderen Seite aber den Testaufwand in einem überschaubaren Rahmen hält.

Ein Ansatz zur Erstellung einer ausreichenden Anzahl von Testfällen wird an zwei Beispielen in diesem Beitrag vorgestellt.

Brücke zwischen Programmierung und Test

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. Die aktuelle Entwicklung, die bei der Softwareerstellung keine strikte (personenbezogene) Trennung zwischen Implementierung und Test auf Unit-Ebene vorsieht (z. B. *Test-Driven Development* und *Test-first*-Ansatz), erfordert eine Brücke zwischen Programmierung und Testen für den Entwickler. Ihm soll bekannt sein, welche Testverfahren es gibt und wie sie mit vertretbarem Aufwand auf seiner Ebene eingesetzt werden können.

Mit dem Begriff „Lean Testing" möchten wir verdeutlichen, dass das Testen nicht zwangsläufig mit hohem Aufwand und mit vielen Testfällen verknüpft ist. Durch die systematische Verwendung von Testverfahren, die der Tester kennt und einsetzt, ist es auch dem Entwickler möglich, die Prüfung seiner Software effektiv und effizient durchzuführen – eben „lean".

Lean Testing 1: Von 27 zu 13 Testfällen

Es sei eine (fehlerhafte) C++-Funktion zum Finden des maximalen Werts dreier Zahlen gegeben [Kle13]:

```
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;
}
```

Sicherlich sehen Sie bereits, dass dieses Programmstück seine Tücken hat. Wenn für einen ***int***-Wert 32 Bits angenommen werden, ergibt sich eine riesige Zahl möglicher Kombinationen (2^{96}). Offensichtlich ist es sinnlos, diese Kombinationen alle testen zu wollen. Nehmen wir nun weiter an, wir entscheiden uns dafür, den Test mit nur drei 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.

Unzureichend: Ad-hoc-Test

Man könnte nun auf die Idee kommen, einfach drei verschiedene Zahlen herauszugreifen und die Funktion damit zu testen, zum Beispiel 4, 5 und 7. Die Tests mit ***max (4, 5, 7)***, ***max(5, 7, 4)*** und ***max(7, 5, 4)*** sind alle erfolgreich. Darüber hinaus wird jede Entscheidung mit ***true*** und ***false*** ausgewertet. Wir haben also 100 Prozent Entscheidungsüberdeckung! Tatsächlich ist das nicht ausreichend, schon ***max(7, 4, 5)*** schlägt fehl.

Systematisch herangehen

Anstelle des Ad-hoc-Vorgehens oder des Austestens ist ein *systematisches* Vorgehen sinnvoll. Dazu wird angenommen, dass die Funktion bei jeder ganzen Zahl korrekt arbeitet, ob sie nun 9999 oder -3 ist. Das heißt, die Zahlen sind *äquivalent*. Damit reichen tatsächlich drei konkrete Zahlen aus. Aus diesen lassen sich dann $3^3 = 27$ Kombinationen bilden – schon erheblich weniger als 2^{96}.

Eine weitere Reduktion ergibt sich aus der Überlegung, dass es beim Finden des Maximums nur um die relativen Größenverhältnisse geht. Das führt auf die folgenden Testfälle:

- Alle Zahlen sind gleich, ein möglicher Repräsentant für ein zu testendes Tripel ist 7, 7, 7.
- Zwei Zahlen sind gleich und die dritte ist größer: 7, 5, 5; 5, 7, 5; 5, 5, 7.**
- 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.

In der Summe ergeben sich nun vier Klassen äquivalenter Werte. Das heißt, dass sich die Funktion bei jedem Zahlentripel einer Klasse gleich verhält. Solche Klassen heißen Äquivalenzklassen. Unter *möglicher Repräsentant* wird ein konkreter Fall von mehreren einer Äquivalenzklasse verstanden: Statt 7, 7, 7 wäre auch 4, 4, 4 oder 5, 5, 5 möglich. In der dritten Klasse sind auch 5, 5, 4 und 7, 7, 4 mögliche Repräsentanten. Es gibt nun insgesamt 13 Testfälle. Sie enthalten auch die Fälle, bei denen die Funktion scheitert. Werte in Äquivalenzklassen zu unterteilen, liegt hier nahe.

Bei der Unterteilung in Äquivalenzklassen ist es üblich, auch ungültige Werte zu betrachten, also Werte, bei denen eine Funktion auf jeden Fall scheitern muss. Das ist hier nicht erforderlich, weil jede Bit-Kombination einen gültigen ***int***-Wert repräsentiert. Im Allgemeinen ist die Aufteilung in Äquivalenzklassen nur *ein* Weg, systematisch an das Testen heranzugehen.

Lean Programming unterstützt Lean Testing

Um mit wenig Testeinsatz viel überprüfen zu können, muss der Code – das Testobjekt – möglichst einfach sein. Trickreiche und *künstlerische*, *freie* Programmierstile 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 Verfahren und Ansätzen durchgeführt werden kann und damit „lean" ist. Auch *Refactoring* ist ein wichtiger Pfeiler für eine einfache Testbarkeit.

Zuständigkeiten klären

Betrachten wir einen weiteren Aspekt: Es könnte sein, dass der Aufrufer der Funktion ***long***-Werte anstelle von ***int***-Werten übergibt. Wenn ***long*** mehr Bits hat als ***int*** (systemabhängig), werden die überschüssigen Bits abgeschnitten, sodass sich innerhalb der Funktion scheinbar gültige Werte ergeben können, obwohl das übergebene Argument zu groß ist. Dieser Fall ist innerhalb der Funktion nicht feststellbar, nur außerhalb durch Überprüfung eines jeden Aufrufs. Das ist jedoch nicht mehr die Aufgabe des Entwicklers der Funktion, sondern die der Tester des Systems und der Nutzer der Funktion. Der (System-)Tester kann das Problem feststellen und dafür sorgen, dass die Benutzungsdokumentation angepasst wird.

Das führt zu der Forderung, die Zuständigkeiten klar zu trennen. Das geeignete Mittel ist das *Design by Contract* [Mey09]. Es bedeutet, dass in einer Art Vertrag festgelegt wird, wofür der Aufrufer, der Dienstnehmer, verantwortlich ist und mit welchen Ergebnissen er vom Dienstanbieter (der Software des Entwicklers) 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.

Durch *Design by Contract* werden die Verantwortlichkeiten bei der Nutzung von Schnittstellen auf beiden Seiten festgelegt. Tests, die die Einhaltung der Schnittstelle einer Unit durch den Aufrufer prüfen, fallen nicht in die Zuständigkeit des Entwicklers der Unit und müssen von ihm nicht in Betracht gezogen werden – sonst könnte er nicht „lean" testen.

Lean Testing 2: Von 16 zu 5 und von 1024 zu 13 Testfällen

Sehen wir uns ein weiteres Vorgehen zur Erstellung von Testfällen an. Bei Funktionen kommt es häufig vor, dass Parameter unabhängig voneinander sind und unterschiedliche Werte annehmen können. Die Frage, die sich beim Testen stellt, ist, „Welche Werte der Parameter sollen miteinander kombiniert werden?" Auf der sicheren Seite ist man, wenn alle Werte miteinander kombiniert werden. Dies übersteigt in aller Regel aber den vertretbaren Aufwand. Was wäre nun ein *Lean-Testing*-Ansatz?

2er-Kombinationen bei 4 booleschen Parametern

Nehmen wir ein einfaches Beispiel mit vier booleschen Parametern A, B, C und D zur Verdeutlichung des Ansatzes. Es ergeben sich $2^4 = 16$ Kombinationen. Sehen wir uns die fünf Kombinationen beziehungsweise Testfälle in Tabelle 1 an.

	A	B	C	D
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	0	1
5	1	1	1	0

Tabelle 1: 2er-Kombinationen mit vier booleschen Parametern.

Die fünf Testfälle scheinen relativ willkürlich gewählt zu sein. Jeder Parameter erhält beide möglichen Werte – schon mal ein erster sinnvoller Ansatz! Aber dafür würden auch zwei Testfälle (alle Parameterwerte einmal ***true*** und einmal ***false***) ausreichen.

Betrachten wir nun die einzelnen Kombinationen der jeweiligen Parameter*paare*, also Parameter A mit Parameter B (farblich hervorgehoben), Parameter A mit Parameter C, ... , Parameter B mit Parameter D, Parameter C mit Parameter D (farblich hervorgehoben).

Fällt Ihnen was auf? Auf die Parameterpaare bezogen kommen alle vier möglichen Kombinationen in den fünf Testfällen vor! Das ist ein *Lean-Testing*-Ansatz: Nicht alle 16 möglichen Kombinationen, sondern eine sinnvolle, nachvollziehbare Auswahl von Testfällen wird zum Testen herangezogen. Die Ersparnis ist in diesem Beispiel nicht sehr groß – von 16 auf 5. Dies liegt aber daran, dass es bei den Parametern nur zwei Möglichkeiten (***true***, ***false***) gibt und die Anzahl der Parameter gering ist.

3er-Kombinationen bei 10 booleschen Parametern

Nehmen wir ein umfangreicheres Beispiel mit 10 booleschen Parametern A, B ... bis J zur weiteren Klärung des Ansatzes. Es ergeben sich $2^{10} = 1024$ Kombinationen – zu viele! Sehen wir uns die dreizehn Kombinationen beziehungsweise Testfälle aus Tabelle 2 an.

Es geht hier nicht mehr um Paare von Parametern, die miteinander vollständig kombiniert werden, sondern es sind *Tripel*. Drei beliebige verschiedene Spalten enthalten alle acht möglichen Kombinationen für drei binäre Variablen! In der Tabelle sind einige Spalten entsprechend farbig markiert, man könnte aber auch beliebige andere Spalten kombinieren, etwa F, G und H. Es werden immer alle acht Möglichkeiten mit den 13 Testfällen abgedeckt.

N-weises Testen

Im obigen Beispiel mit den vier Parametern haben wir alle 2er-Kombinationen mit fünf Testfällen geprüft. In diesem Beispiel sind es alle 3er-Kombinationen für drei beliebige der zehn Parameter. Aber warum gerade zwei oder drei Parameter kombinieren? Es können auch Vierer- oder höhere Kombinationen gewählt werden (N-weises Testen mit N=4, 5, ...).

Das hier skizzierte Verfahren gehört zu den kombinatorischen Testverfahren [Kuh10]. Grundlage sind sogenannte *Covering Arrays*. Mit dem Verfahren kann die Anzahl der Testfälle variiert werden, je nach geforderter Intensität des Testens. Die Anzahl bleibt aber immer noch weit unter der Anzahl, wenn alle möglichen Kombinationen herangezogen werden.

Das ist „Lean Testing"! Nicht alle möglichen Kombinationen, sondern eine sinnvolle, nachvollziehbare Auswahl von Testfällen wird zum Testen verwendet. Die Idee ist dabei, dass es meistens nur begrenzte Abhängigkeiten der Parameter untereinander gibt und dass man die meisten Fehler findet, wenn die Parameter wie beschrieben variiert und kombiniert werden. Das Vorgehen wird in [Spi16] ausführlich beschrieben und es wird eine Werkzeugunterstützung vorgestellt, die die entsprechenden Kombinationen erzeugt.

Aber wie ist zu beurteilen, ob 2er-, 3er- oder 4er-Kombinationen das Richtige sind? Dazu gibt es keine einfache Antwort. Softwarefehler stellen ein Risiko dar, das auch noch kaum quantifizierbar ist. Das Risiko zu minimieren heißt, den Testaufwand zu maximieren. In der Praxis wird man versuchen, den Testaufwand zum vermuteten Risiko in ein akzeptables Verhältnis zu setzen. „Lean Testing" versucht, den Testaufwand für ein gegebenes Risiko zu minimieren, oder anders gesagt, das Testen effizienter zu gestalten.

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Tabelle 2: 3er-Kombinationen mit zehn booleschen Parametern. (Beispiel in Anlehnung an Kuhn et al. [Kuh10])

Fazit

Wir haben zwei Beispiele zum „Lean Testing" vorgestellt und damit verdeutlicht, dass bei systematischen Vorgehen mit wenig Aufwand ein angemessener Test durchgeführt werden kann.

In [Spi16] finden sich weitere Verfahren mit ausführlichen Beispielen, um den richtigen Mittelweg zwischen zu wenig und zu viel Testen herauszufinden. Es werden Testverfahren angesprochen, die zwar dem professionellen Tester gut bekannt sind, nicht jedoch vielen Softwareentwicklern. Diesen soll eine Handreichung für die tägliche Arbeit gegeben werden, ganz im Sinn der oben angesprochenen Brücke zwischen Programmierung und Testen. Voraussetzung ist, dass die Programmierung sauber (*Clean Code*) durchgeführt wird und die Zustän-

digkeiten (*Design by Contract*) geklärt sind. Dann lässt sich der Testaufwand unter Verwendung von etablierten Testverfahren angemessen – „lean“ – gestalten.

Testverfahren sind im aktuellen ISO-Standard 29119 zu finden, der viele für den Unittest relevante Verfahren enthält ([Soft], [Spi16], Kap. 7). ■

* *Geschlechtsbezogene Formen meinen im gesamten Text Frauen, Männer und alle anderen.*
** *Statt 5 kann auch 4 als Testdatum ausgewählt werden, dies hat aber vermutlich keinen Einfluss auf das Testergebnis.*
*** *Teile des Beitrags sind dem Buch [Spi16] entnommen.*

Literatur & Links

[Kle13] St. Kleuker, Qualitätssicherung durch Softwaretests, Springer Vieweg, 2013

[Kuh10] D. R. Kuhn, R. N. Kacker, Yu Lei, Practical Combinatorial Testing, National Institute of Standards and Technology (NIST), Special Publication 800-142, 2010 (siehe: http://dx.doi.org/10.6028/NIST.SP.800-142)

[Lan13] J. Langr, Modern C++ Programming with Test-Driven Development, O'Reilly, 2013

[Mey09] B. Meyer, Touch of Class, Springer Verlag, 2009

[Soft] http://softwaretestingstandard.org/

[Spi16] A. Spillner, U. Breymann, Lean Testing für C++-Programmierer, dpunkt.verlag, 2016 (siehe auch http://www.leantesting.de/)***