



Andreas Spillner · Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

Broschüre zu Lean Testing von
Kombinationen mit n-weisem Testen
& Entscheidungstabellen

Konform zu
ISO 29119

dpunkt.verlag



Andreas Spillner war bis 2017 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 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).

Lektorat: Christa Preisendanz

Copy-Editing: Ursula Zimpfer, Herrenberg

Satz: Andreas Spillner, Ulrich Breymann

Umschlaggestaltung: Helmut Kraus, www.exclam.de

Diese Broschüre basiert auf dem Buch

»Lean Testing für C++-Programmierer«, dpunkt.verlag, 2016 (ISBN 978-3-86490-308-3).

Copyright © 2018 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

Andreas Spillner • Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

**Lean Testing von Kombinationen
mit n-weisem Testen &
Entscheidungstabellen**



dpunkt.verlag

Vorwort

Liebe Leserinnen und Leser,

es ist das Ziel eines jeden Softwareentwicklers¹, 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 Titel des Open Books 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. Das beschriebene Vorgehen soll eine Brücke zwischen Programmierung und Testen für den C++-Entwickler bauen und ihm zeigen, welche Testverfahren es beim kombinatorischen Testen gibt und wie sie mit vertretbarem Aufwand eingesetzt werden können.

Lean Testing

Testen wird leider oft noch als eine destruktive, zeitaufwendige und wenig zielführende Aktivität gesehen. So schreibt Jeff Langr beispielsweise [Langr 13, S. 35]: »Using a testing technique, you would seek to exhaustive-

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

ly 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.« 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 drei 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!

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.

Dafür gibt es Testverfahren, die eine Beschränkung auf bestimmte Testfälle vorschlagen. Der Begriff »Lean Testing« hebt genau diesen Aspekt hervor. Wir wollen dem Entwickler Hilfestellung geben, damit er die für sein Problem passenden Tests in einem angemessenen Zeitaufwand durchführen kann, um die geforderte Qualität mit den Tests nachzuweisen. Ein vollständiger Test wird nicht angestrebt.

Kombinatorisches Testen

Ein häufiges Problem beim Testen sind die Kombinationsmöglichkeiten von Parametern. Der Entwickler muss entscheiden, welche der vielen Kombinationen er mit einem Test überprüft und welche nicht. In diesem Zusammenhang wird auch von Kombinationsexplosion gesprochen.

Beispiel: 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.

Wie intensiv soll getestet werden?

Zwei Testfälle reichen aus, dass jeder Parameter die Werte »true« und »false« annimmt. Beim ersten Testfall erhalten alle 15 Parameter den Wert »true« und beim zweiten Testfall den Wert »false«.

Die vollständige Kombination erfordert 32.768 Testfälle.

Da muss es doch noch was Sinnvolles dazwischen geben!

Wir stellen die praktische Anwendung von zwei kombinatorischen Testverfahren vor, um die Entscheidung, welche Testfälle auszuwählen sind, zu unterstützen und um Kriterien an die Hand zu geben, wann der Test als ausreichend angesehen werden kann. Dabei liegt der Fokus auf Lean Testing, also dem Versuch, einen guten Kompromiss zwischen angestrebter Qualität und Testaufwand zu finden.

Wir wollen auch zeigen, dass die Nutzung von Testverfahren eine Hilfestellung bei der Auswahl der »passenden« Testfälle bietet und keinen Zwang darstellt, alle möglichen Testfälle auszuführen, wie es Langr oben beschreibt.

Für alle Beispiele im Open Book nutzen wir Google Test [[URL: googletest](#)]. Die Programmbeispiele stehen unter [[URL: leantesting](#)] zum Download zur Verfügung.

Wir hoffen, Ihnen beim Durcharbeiten des Open Books Hinweise und Anregungen für den Test Ihrer Software als Teil der täglichen Arbeit geben zu können. Neben den hier beschriebenen Ansätzen zum kombinatorischen Testen finden Sie viele weitere in unserem Buch »Lean Testing für C++-Programmierer – Angemessen statt aufwendig testen«.

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

Bremen, im April 2018

Andreas Spillner & Ulrich Breymann

Inhaltsverzeichnis

1	n-weises kombinatorisches Testen	1
1.1	Orthogonale Arrays	3
1.2	Covering Arrays	4
1.3	n-weises Testen (N-wise Testing)	4
1.4	Ein Beispiel in C++	12
2	Entscheidungstabellentest	15
2.1	Ein Beispiel in C++	20
2.2	Vorsicht Kunde!	24
3	Zusammenfassung	27
	Literaturverzeichnis	29

1 n-weises kombinatorisches Testen

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 1-1.

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 1-1: 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 1-2: 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 ParameterA/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 1-2 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.

Erörtern wir das Vorgehen an einem komplexeren Beispiel: 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:

- Kind oder Jugendlicher. Monatlicher Grundbeitrag: 7 €.
 - Erwachsener. Monatlicher Grundbeitrag: 18 €.
 - Ermäßiger 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.

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 1-3 enthalten alle paarweisen Kombinationen.

An diesem etwas umfangreicherem Beispiel ist gut zu erkennen, dass einige Kombinationen häufiger als andere enthalten sind. So kommen für das Paar Tischtennis/Turnen die Wertekombinationen ja/ja und nein/nein

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 1-3: Sportarten (die vier möglichen Kombinationen von Turnen und Ballsport sind hervorgehoben)

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 1-3 nicht vor.

Es gibt auch nicht nur diese eine Kombination mit der Eigenschaft, dass die jeweiligen Parameterpaare alle vier Kombinationen enthalten. Die folgende Tabelle 1-4 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 1-4: Auswahl von fünf Kombinationen der Sportarten

1.1 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. Ein Anwendungsgebiet der orthogonalen Arrays ist

die statistische Versuchsplanung, da dort die zu untersuchenden Faktoren nicht miteinander vermenge werden dürfen und gleichverteilt sein sollen. Jede Kombination kommt in einem orthogonalen Array *gleich oft* vor, egal welche zwei Spalten des Arrays ausgewählt werden. So sind die Tabellen 1-1 und 1-2 von Seite 1 orthogonale Arrays. Bei beliebigen zwei Spalten der Tabelle 1-1 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 1-2 kommt jede Kombination bei beliebigen zwei Spalten exakt einmal vor.

1.2 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 acht 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 1-3. Das Array kann noch verkleinert werden: Die Tabelle 1-4 zeigt ein minimales Covering Array. Es ergeben sich somit fünf (oder sechs) Kombinationen bei den Covering Arrays im Gegensatz zu den acht erforderlichen Kombinationen bei Nutzung der orthogonalen Arrays.

1.3 n-weises Testen (N-wise Testing)

»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 1-5 alle 3er-Kombinationen für die Wahl der Sportart aufgeführt. Die acht möglichen Kombinationen von Tischtennis, Turnen und Fitness sind rot hervorgehoben (Test 6 ist diesbezüglich redundant zu Test 5).

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 1-5: 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 fünf 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 Parameter aber einen von drei, vier oder noch mehr möglichen Werten annehmen können. Orthogonale Ar-

rays lassen sich dann teilweise nicht mehr konstruieren, Covering Arrays schon.

Wann ist der Einsatz sinnvoll?

Bei Testobjekten mit mehreren Parametern, die zu kombinieren sind, gibt 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 nur die Kombinationen 1 und 8 (oder auch 2 und 7, 3 und 6 sowie 4 und 5) aus der Tabelle 1-1 zu verwenden, um diese Testabdeckung bereits vollständig zu erfüllen. Um alle Kombinationen abzudecken, sind alle acht Kombinationen aus der Tabelle 1-1 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 und damit sinnvolle Kriterien zur Beendigung des Testens lassen sich für alle Testverfahren festlegen.

Mit dem 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 Ermessens 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 1-5 auf Seite 5 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 nach Lean-Testing-Kriterien einen Mittelweg 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.

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 bei großem n oft nicht durchführbar oder zu aufwendig sein. Als Alternative bietet es sich an, nach Beseitigen

der Fehler n schrittweise zu erhöhen, bis auch auf der 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.

Bewertung

Das kombinatorische Vorgehen zur Erstellung der Testfälle ist einfach nachvollziehbar und überzeugend. Es gibt dem Entwickler 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 berücksichtigen, aber weitere Testfälle sind als Ergänzung hinzuzunehmen. Sehen wir uns dazu die Tabellen [1-4](#) mit den 2er-Kombinationen und Tabelle [1-5](#) mit den 3er-Kombinationen der vier Sportarten unter Testgesichtspunkten näher an.

2er-Kombinationen (Tabelle [1-4](#) auf Seite [3](#)):

- 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 [1-5](#) auf Seite [5](#)):

- 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 Entwickler 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 [Spillner & Breymann 16, Kap. 8].

Bezug zu anderen Testverfahren

Das n-weise Testen ist das einzige Testverfahren, das die Kombination der Parameterwerte systematisch überprüft. Bei anderen Testverfahren wie Äquivalenzklassentest¹ 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.

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.² 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 1-1).

Die Stärke (Strength) ist oben im Bild angegeben und kann entsprechend gesetzt werden. 2 (Anzahl der Parameter) bedeutet paarweises Testen. Wenn 3 eingestellt wird, gibt das Programm 19 Testfälle aus. Die maximale

¹Testverfahren, die hier genannt, aber nicht erklärt werden, wie Äquivalenzklassentest, Klassifikationsbaummethode und Grenzwertanalyse, werden in [Spillner & Breymann 16] in den Abschnitten 4.3 – 4.5 ausführlich beschrieben.

²Dazu sind natürlich auch andere Werkzeuge fähig.

	TISCHTENNIS	TURNEN	BALLSPORT	FITNESS	MITGLIEDSSTATUS
1	true	false	false	false	Kind/Jugendlicher
2	false	true	true	true	Kind/Jugendlicher
3	true	true	false	true	Erwachsener
4	false	false	true	false	Erwachsener
5	true	true	true	false	Ermaessigt
6	false	false	false	true	Ermaessigt
7	false	false	false	false	Passiv

Abbildung 1-1: Covering Array für das Sportverein-Beispiel unter Berücksichtigung des Mitgliedsstatus

Stärke ist 5. Damit ergeben sich 46 Testfälle, also sämtliche Kombinationsmöglichkeiten. Eine davon betrifft den Mitgliedsstatus »Passiv«, dem keinerlei Sportarten zugeordnet sind.

Das Programm speichert die Konfiguration wahlweise in einer XML- oder einer Textdatei. Letztere ist hier auszugsweise abgedruckt,³ weil sie 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 1.1: 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
```

³Aus Layoutgründen leicht umformatiert.

```
true,true,true,false,Ermaessigt  
false,false,false,true,Ermaessigt  
false,false,false,false,Passiv
```

Listing 1.2: 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 [[URL: googletest](#)] 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 – hier der monatliche Mitgliedsbeitrag je nach gewählten Sportarten und Mitgliedsstatus – vor der Berechnung *voraussagen* können. Deswegen heißt so ein Wert in Anlehnung an das Orakel von Delphi *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 – der jeweilige monatliche Mitgliedsbeitrag – 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 1.3: Testdaten (*beispiele/sportvACTS/generator/testdaten.txt*)

1.4 Ein Beispiel in C++

Aus den Testdaten 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>
#include "../util/split.h" // String mit Trennzeichen in einen Vektor
                         // 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
                << ")_{"_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\";\n"
}
```

Listing 1.4: 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 1.5: Testprogramm (*beispiele/sportvACTS/dertest.cpp*)

Mit den insgesamt sieben Testfällen sehen wir den Test als ausreichend an. Wir haben nicht alle Kombinationen geprüft, aber durch den methodischen Ansatz alle möglichen 2er-Kombinationen. Wir haben »lean« getestet!

Welche und wie viele Testfälle hätten Sie ausgewählt? Hätten Sie ein nachvollziehbares Kriterium zur Beendigung des Testens gehabt (außer Zeit oder Anzahl der Testfälle)?

2 Entscheidungstabellentest

Nicht immer lassen sich Parameter frei miteinander kombinieren. Beeinflussten 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.

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.

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 2-1).

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 2-1: 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 ab 18 Jahren abgegeben werden. Es kommt somit eine weitere Bedingung hinzu. Die Größe der Entscheidungstabelle verdoppelt sich (siehe Tabelle 2-2).

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

Tabelle 2-2: 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 2-3).

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 2-3: 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 2-1 sind es vier und in Tabelle 2-2 acht Spalten. Wie sieht es aber mit der konsolidierten Tabelle 2-3 aus? Diese enthält fünf Spalten! Wenn in einer Tabelle »-« vorkommen, dann führt jeder Strich zu einer Multiplikation mit 2. Wir haben also eine Spalte (mit »ja« bei der ersten Bedingung) mit zwei Strichen bei den weiteren Bedingungen in der Spalte, dies ergibt $1 \cdot 2 \cdot 2 = 4$. 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 2-3 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 ein Testorakel ist nicht extra zu befragen.

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.

Äquivalenzklassen¹ beziehen sich auf Parameterwerte, die meist ohne Einschränkung frei zu kombinieren sind. Abhängigkeiten zwischen Parametern 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.

2.1 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 Nachzuschlag in Höhe von 20%.
- Gepäckstücke kosten unabhängig von der Anzahl 3 € zusätzlich.

¹Einzelheiten zum Vorgehen bei den hier erwähnten weiteren Testverfahren werden in [Spillner & Breymann 16] ausführlich beschrieben.

Eine Fahrt ohne Gepäck tagsüber von beispielsweise genau 10 km kostet damit $3,50 \text{ €} + 10 \cdot 2,10 \text{ €} = 24,50 \text{ €}$. 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 2.1: 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.

In Tabelle 2-4 ist bereits ein Widerspruch zu erkennen. Die Rabatte für Strecken von mehr als 10 oder mehr als 50 km sind nicht additiv. Es gilt

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 2-4: Erste Entscheidungstabelle für den Fahrpreistest

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 2-5 zeigt die um die Spalten 9 bis 12 verminderte und damit konsolidierte Tabelle 2-4 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 2-5: Konsolidierte Entscheidungstabelle für den Fahrpreistest

In der Tabelle 2-5 liefert nun keine Kombination (Spalte) mehr identische Aktionen. Das heißt, es bleiben 12 Testfälle zu prüfen.

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

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 unnötze Testfälle dabei. Die Entscheidungstabelle hilft also, die Übersicht über die sinnvollen Kombinationen zu behalten.

> 50 km gewählt. Grenzwerte werden hier nicht geprüft, um das Beispiel einfach zu halten. Ergänzende Testfälle mit den Grenzwerten sind natürlich sinnvoll und durchzuführen. 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 2-6).

Das Listing 2.2 zeigt das Testprogramm. Die Nummer eines Testfalls TF entspricht der Spaltennummer der Tabelle 2-6.

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 2-6: 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 2.2: Auszug des Testprogramms *beispiele/taxi1/fahrpreistest.cpp*

2.2 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 12² 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 20 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.

²Das sind die beiden einzigen Testfälle ohne Rabatt oder Zuschlag!

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 20 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 2.1 von Seite 21 gemacht wird. Beispielrechnung für Testfall 5 aus Tabelle 2-6, eine Nachtfahrt mit Gepäck über 20 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 2.3: Fahrpreisberechnung nach Kundenvorstellung

Entscheidungstabellen, deren Umfang durch Konsolidierung nicht groß eingeschränkt werden kann, sind nicht wirklich »lean«. Entscheidungstabellen 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.

3 Zusammenfassung

In diesem Open Book haben wir uns auf den Lean-Testing-Ansatz bei kombinatorischen Tests und Entscheidungstabellentests konzentriert, weil dort die Einsparungseffekte besonders gut sichtbar werden.

Daneben gibt es weitere Testarten, die vom Lean-Testing-Ansatz profitieren, etwa die Klassifikationsbaummethode oder der zustandsbasierte Test. Im Buch »Lean Testing für C++-Programmierer – Angemessen statt aufwendig testen« [Spillner & Breymann 16] finden Sie insgesamt neun Testansätze für den spezifikationsbasierten Test und sieben Ansätze für den strukturbasierten Test, jeweils mit praktischen Beispielen in der Programmiersprache C++. Auch zwei erfahrungsbasierter Verfahren werden dort diskutiert. Bei jedem Testverfahren wird herausgearbeitet, wie ein angemessener »lean« Test aussieht. Alle Testansätze werden anhand von Programmbeispielen erklärt und sind bis hin zu den einzelnen Testfällen programmiert. Die Programmbeispiele stehen unter [[URL: leantesting](#)] zum Download zur Verfügung.

Im Buch wird auf den aktuellen Softwareteststandard ISO 29119 ebenso eingegangen wie auf die Frage »Wann soll ich welches Testverfahren einsetzen?«. Es wird ein Leitfaden zum Einsatz der Verfahren vorgestellt.

Zur praktischen Umsetzung gehört auch das Umfeld: die Eigenschaften der Programmiersprache, hier C++, und die verwendeten Werkzeuge. Diese Dinge werden ebenso beschrieben.

Wir hoffen, mit dem Open Book vielen Entwicklern eine Anregung gegeben und neben der praktischen Umsetzung gezeigt zu haben, dass es beim Testen einen zielgerichteten Weg zwischen zu wenigen und zu vielen Testfällen gibt.

Literaturverzeichnis

- [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*. The Pragmatic Programmers, 2013. [dt. Übersetzung *Testgetriebene Entwicklung mit C++ – Sauberer Code. Bessere Produkte*, dpunkt.verlag, 2014]
- [Spillner & Breymann 16] Andreas Spillner, Ulrich Breymann: *Lean Testing für C++-Programmierer – Angemessen statt aufwendig testen.* dpunkt.verlag, 2016
- [URL: ACTS] Advanced Combinatorial Testing System (ACTS).
http://csrc.nist.gov/groups/SNS/acts/download_tools.html.
- [URL: googletest] *Google C++ Testing Framework*,
<https://code.google.com/p/googletest>.
- [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 Erscheinen dieses Open Books nicht mehr auffindbar sind.

