Andreas Spillner · Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

Konform zu ISO 29119

dpunkt.verlag

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

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

Papier plus PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.de/plus

Andreas Spillner · Ulrich Breymann

Lean Testing für C++-Programmierer

Angemessen statt aufwendig testen

dpunkt.verlag

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

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

http://leantesting.de

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

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

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

1. Auflage 2016
Copyright © 2016 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.

5 4 3 2 1 0

Vorwort

Liebe Leserinnen und Leser,

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

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

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

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

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

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

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

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

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

Bremen, im April 2016

Andreas Spillner & Ulrich Breymann

Inhaltsverzeichnis

1 Einleitung

Worum geht's

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

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

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

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

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

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

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

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

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

TDD (Test Driven Development)

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

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

Test-Büffet

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

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

»Lean Testing«

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

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

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

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

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

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

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

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

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

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

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

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

»Lean Testing« setzt »Lean Programming« voraus

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

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

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

Listing 1.1: »Unsauberer« Code

Die an diesem Listing zu kritisierenden Punkte sind:

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

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

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

Worum geht's nicht

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

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

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

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

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

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

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

2 Test gegen die Anforderungen

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

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

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

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

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

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

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

Design by Contract

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

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

In der Praxis: Robustheit erwünscht

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

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

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

Auswirkungen auf den Test

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

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

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

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

3 Statische Verfahren

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

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

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

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

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

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

Konkrete Beispiele finden Sie weiter unten.

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

Beschränkungen in der Praxis

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

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

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

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

Praktischer Einsatz

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

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

3.1 Codereview

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

Walkthrough

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