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.