mitp

Roy
Osherove

2. Auflage

The Art of
Unit Testing
Deutsche Ausgabe

Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Roy Osherove

The Art of Unit Testing

Deutsche Ausgabe

Übersetzung aus dem Englischen
von Olaf Neuendorf

mitp

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 978-3-8266-8721-1
2. Auflage 2015

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Übersetzung der amerikanischen Originalausgabe:
Roy Osherove: The Art of Unit Testing: With Examples in C#,
Second Edition, ISBN 978-1617290893.
Original English language edition published by Manning Publications,
178 South Hill Drive, Westampton, NJ 08060 USA.

Lektorat: Sabine Schulz
Sprachkorrektorat: Petra Heubach-Erdmann, Maren Feilen
Coverbild: © Sebastian Duda, Bildnummer 33776850
Satz: III-satz, Husby, www.drei-satz.de

Inhaltsverzeichnis

Vorwort zur zweiten Auflage

Es muss im Jahre 2009 gewesen sein. Ich hielt einen Vortrag auf der Norwegian Developers Conference in Oslo. (Ah, Oslo im Juni!) Die Veranstaltung fand in einer großen Sportarena statt. Die Konferenz-Organisatoren unterteilten die überdachte Tribüne in Abschnitte und bauten jeweils ein Podium vor ihnen auf. Dann hängten sie dicke schwarze Stoffbahnen auf und erstellten auf diese Weise acht verschiedene »Konferenzräume«. Ich erinnere mich, dass ich gerade dabei war, meinen Vortrag zu beenden – er handelte von TDD, oder SOLID, oder Astronomie, oder irgendwas – als plötzlich von der Nachbarbühne ein lautes und raues Singen und Gitarrenspiel ertönte.

Die Stoffbahnen hingen so, dass ich zwischen ihnen hindurch spähen und den Burschen auf der Nachbarbühne sehen konnte. Natürlich war es Roy Osherove.

Nun, diejenigen von Ihnen, die mich kennen, wissen, dass das Anstimmen eines Liedes mitten in einem technischen Vortrag über Software zu den Dingen gehört, die *ich* vielleicht tue, wenn mir danach ist. Als ich mich wieder meinem Publikum zuwandte, dachte ich so bei mir, dass dieser Osherove wohl ein Gleichgesinnter sei und ich ihn besser kennenlernen müsse.

Und ich lernte ihn besser kennen. In der Tat trug er erheblich zu meinem aktuellen Buch *Clean Coder* bei und unterstützte mich drei Tage lang dabei, einen Kurs in TDD zu geben. Alle meine Erfahrungen mit Roy waren sehr positiv und ich hoffe, es wird noch viele weitere geben.

Ich schätze, dass Ihre Erfahrungen mit Roy beim Lesen dieses Buch auch sehr positiv sein werden, denn dieses Buch ist etwas Besonderes.

Haben Sie jemals einen Roman von Michener gelesen? Ich jedenfalls nicht – aber mir wurde gesagt, sie alle würden »mit dem Atom« beginnen. Das Buch, das Sie in Händen halten, ist kein James-Michener-Roman, aber es beginnt mit dem Atom – dem Atom des Unit Testings.

Lassen Sie sich nicht beirren, wenn Sie die ersten Seiten durchblättern. Dies ist *keine* bloße Einführung ins Unit Testing. Es beginnt so, aber wenn Sie bereits Erfahrung haben, können Sie diese ersten Kapitel schnell überfliegen. Während das Buch dann aber voranschreitet, beginnen die Kapitel aufeinander aufzubauen und entwickeln eine erstaunliche Tiefe. Und als ich das letzte Kapitel las (ich wusste nicht, dass es das letzte war), dachte ich tatsächlich, dass das nächste wohl vom Weltfrieden handeln müsse – denn im Ernst, womit sonst kann man weitermachen, nachdem man das Problem gelöst hat, Unit Testing in starrsinnige Organisationen mit alten, überkommenen Systemen einzuführen.

Dieses Buch ist ein Fachbuch – und zutiefst ein Fachbuch. Es gibt eine Menge Code. Und das ist gut so. Doch Roy beschränkt sich nicht auf die technischen Aspekte. Gelegentlich holt er seine Gitarre hervor, beginnt zu singen und erzählt Geschichten aus seinem Berufsleben. Oder er stellt philosophische Betrachtungen an über die Bedeutung von Design oder

die Definition von Integration. Er scheint Gefallen daran zu finden, uns mit Geschichten aus jenen dunklen, längst vergangenen Tagen des Jahres 2006 zu verwöhnen, als ihm so manches misslungen ist.

Ach, und machen Sie sich keine allzu großen Sorgen, dass der komplette Code in C# geschrieben ist. Im Ernst, wer könnte überhaupt den Unterschied zwischen C# und Java erklären? Oder? Und abgesehen davon spielt es auch keine Rolle. Er mag zwar C# als Vehikel benutzen, um seine Intention klar zu machen, aber die Lektionen in diesem Buch gelten genauso für Java, C, Ruby, Python, PHP oder jede andere Programmiersprache (vielleicht mit Ausnahme von COBOL).

Egal, ob Sie ein Anfänger auf dem Gebiet des Unit Testings und der testgetriebenen Entwicklung sind oder ein alter Hase, dieses Buch hält für Sie alle etwas bereit. Also viel Vergnügen, während Roy für Sie sein Lied singt »The Art of Unit Testing«.

Und bitte Roy, stimme diese Gitarre!

Robert C. Martin (Uncle Bob)
CLEANCODER.COM

Vorwort zur ersten Auflage

Als mir Roy Osherove erzählte, er würde an einem Buch über das Unit Testing arbeiten, war ich sehr froh, dies zu hören. Das Test-Mem ist über die Jahre in der Industrie gewachsen, aber es gibt einen relativen Mangel an Material zum Unit Testing. Wenn ich mein Bücherregal betrachte, dann sehe ich Bücher zur testgetriebenen Entwicklung im Besonderen und Bücher zum Testen im Allgemeinen, aber bis jetzt gab es keine umfassende Referenz zum Unit Testing – kein Buch, das in das Thema einführt, den Leser an die Hand nimmt und von den ersten Schritten bis zu den allgemein akzeptierten besten Vorgehensweisen führt. Diese Tatsache ist verblüffend. Unit Testing ist keine neue Praxis. Wie konnte es dazu kommen?

Es ist beinahe ein Klischee, wenn man sagt, dass wir in einer neuen Industrie arbeiten, aber es ist wahr. Mathematiker legten die Grundlagen unserer Arbeit vor weniger als 100 Jahren, aber erst seit 60 Jahren haben wir die Hardware, die schnell genug ist, um ihre Erkenntnisse auch nutzen zu können. In unserer Industrie bestand von Anfang an eine Lücke zwischen Theorie und Praxis und erst jetzt entdecken wir, wie sich das auf unser Fachgebiet ausgewirkt hat.

In den frühen Jahren waren Rechenzeiten teuer. Wir ließen die Programme im Batchbetrieb laufen. Programmierer hatten ein planmäßiges Zeitfenster, sie mussten ihre Programme in Kartenstapel stanzen und in den Maschinenraum tragen. War ein Programm nicht in Ordnung, hatten sie ihre Zeit vergeudet, also überprüften sie es zuvor mit Papier und Stift am Schreibtisch und gingen in Gedanken alle möglichen Szenarien, alle Grenzfälle durch. Ich bezweifle, dass der Begriff des automatisierten Unit Testings zu dieser Zeit überhaupt vorstellbar war. Warum sollte man die Maschine zum Testen nutzen, wo sie doch zur Lösung bestimmter Probleme gebaut worden war? Der Mangel hielt uns in der Dunkelheit gefangen.

Später wurden die Maschinen schneller und wir berauschten uns am interaktiven Computing. Wir konnten einfach Code eintippen und ihn aus Jux und Tollerei ändern. Die Idee, den Code am Schreibtisch zu überprüfen, schwand allmählich dahin und wir verloren etwas von der Disziplin der frühen Jahre. Wir wussten, dass Programmieren schwierig war, aber das bedeutete nur, dass wir mehr Zeit am Computer verbringen mussten und Zeilen und Symbole änderten, bis wir die magische Zauberformel, die funktionierte, gefunden hatten.

Wir kamen vom Mangel zum Überfluss und verpassten den Mittelweg, aber nun erobern wir ihn zurück. Automatisiertes Unit Testing verbindet die Disziplin der Schreibtischprüfung mit einer neu gefundenen Wertschätzung des Computers als einer Entwicklungsressource. Wir können automatische Tests in der Sprache schreiben, in der wir entwickeln, um unsere Arbeit zu überprüfen – und das nicht nur einmal, sondern so oft wir in der Lage sind, die Tests laufen zu lassen. Ich glaube nicht, dass es irgendein anderes Verfahren in der Softwareentwicklung gibt, das derartig mächtig ist.

Während ich dies im Jahre 2009 schreibe, bin ich froh zu sehen, wie Roys Buch in den Druck geht. Es ist eine praktische Anleitung, die Ihnen helfen wird, loszulegen, und es ist auch eine großartige Referenz, wenn Sie Ihre Aufgaben beim Testing angehen. *The Art of Unit Testing* ist kein Buch über idealisierte Szenarien. Es zeigt Ihnen, wie man Code testet, der im praktischen Einsatz existiert, wie man Nutzen aus weitverbreiteten Frameworks zieht und – was das Wichtigste ist – wie man Code schreibt, der wesentlich einfacher zu testen ist.

The Art of Unit Testing ist ein wichtiger Titel, der schon vor Jahren hätte geschrieben werden sollen, aber damals waren wir noch nicht bereit dafür. Jetzt sind wir bereit. Genießen Sie es!

Michael Feathers

Senior Consultant
Object Mentor

Einleitung

Eines der größten Projekte, an dem ich mitgearbeitet habe und das schiefgelaufen ist, beinhaltete auch Unit Tests. Das dachte ich zumindest. Ich leitete eine Gruppe von Entwicklern, die an einer Abrechnungssoftware arbeiteten, und wir machten es komplett auf die testgetriebene Weise – wir schrieben zuerst die Tests, dann den Code, die Tests schlugen fehl, wir sorgten dafür, dass sie erfolgreich verliefen, führten ein Refactoring durch und fingen wieder von vorne an.

Die ersten paar Monate des Projekts waren großartig. Alles lief gut und wir hatten Tests, die belegten, dass unser Code funktionierte. Aber im Laufe der Zeit änderten sich die Anforderungen. Wir waren also gezwungen, unseren Code zu ändern, um diesen neuen Anforderungen gerecht zu werden, doch als wir das taten, wurden die Tests unzuverlässig und mussten nachgebessert werden. Der Code funktionierte immer noch, aber die Tests, die wir dafür geschrieben hatten, waren so zerbrechlich, dass jede kleine Änderung in unserem Code sie überforderte, auch wenn der Code selbst prima funktionierte. Die Änderung des Codes in einer Klasse oder Methode wurde zu einer gefürchteten Aufgabe, weil wir auch alle damit zusammenhängenden Unit Tests entsprechend anpassen mussten.

Schlimmer noch, einige Tests wurden sogar unbrauchbar, weil diejenigen, die sie geschrieben hatten, das Projekt verließen und keiner wusste, wie deren Tests zu pflegen waren oder was sie eigentlich testeten. Die Namen, die wir unseren Unit-Testing-Methoden gaben, waren nicht aussagekräftig genug und wir hatten Tests, die auf anderen Tests aufbauten. Schließlich warfen wir nach weniger als sechs Monaten Projektlaufzeit die meisten der Tests wieder hinaus.

Das Projekt war ein erbärmlicher Fehlschlag, weil wir zugelassen hatten, dass die Tests, die wir schrieben, mehr schadeten als nützten. Langfristig brauchten wir mehr Zeit, um sie zu pflegen und zu verstehen, als sie uns einsparten. Also hörten wir auf, sie einzusetzen. Ich machte mit anderen Projekten weiter und dort haben wir unsere Arbeit beim Schreiben der Unit Tests besser erledigt. Ihr Einsatz brachte uns großen Erfolg und sparte eine Menge Zeit beim Debuggen und bei der Integration. Seitdem dieses erste Projekt fehlschlug, trage ich bewährte Vorgehensweisen für das Unit Testing zusammen und wende sie auch im nachfolgenden Projekt an. Im Laufe jedes Projekts, an dem ich arbeite, entdecke ich weitere gute Vorgehensweisen.

Zu verstehen, wie man Unit Tests schreibt – und wie man sie wartbar, lesbar und vertrauenswürdig macht –, ist das, wovon dieses Buch handelt. Egal, welche Sprache oder welche integrierte Entwicklungsumgebung (IDE) Sie verwenden. Dieses Buch behandelt die Grundlagen für das Schreiben von Unit Tests, geht dann auf die Grundlagen des Interaction Testings ein und stellt schließlich bewährte Vorgehensweisen für das Schreiben, das Verwalten und das Warten der Unit Tests in echten Projekten vor.

Über dieses Buch

Dass man das, was man wirklich lernen möchte, unterrichten sollte, ist das vielleicht Klügste, was ich jemals irgendwen über das Lernen sagen hörte (ich vergaß, wer das war). Das Schreiben und Veröffentlichen der ersten Ausgabe dieses Buches im Jahre 2009 war nicht weniger als eine echte Lehre für mich. Ursprünglich schrieb ich das Buch, weil ich keine Lust mehr hatte, die gleichen Fragen immer wieder zu beantworten. Aber natürlich gab es auch andere Gründe. Ich wollte etwas Neues ausprobieren, ich wollte experimentieren, ich wollte herausfinden, was ich lernen konnte, indem ich ein Buch schrieb – irgendein Buch. Im Bereich Unit Testing war ich gut. Dachte ich. Aber es ist ein Fluch: Je mehr Erfahrung man hat, desto dümmer kommt man sich vor.

Es gibt Teile in der ersten Ausgabe, denen ich heute nicht mehr zustimme – bespielsweise, dass sich eine *Unit* auf eine Methode bezieht. Das ist einfach nicht richtig. Eine Unit ist eine Arbeitseinheit, was ich in Kapitel 1 dieser zweiten Auflage diskutiere. Sie kann so klein sein wie eine Methode oder so groß wie mehrere Klassen (möglicherweise Assemblies). Und wie Sie noch sehen werden, gibt es weitere Dinge, die sich ebenfalls geändert haben.

Was neu ist in der zweiten Auflage

In dieser zweiten Auflage habe ich Material über die Unterschiede zwischen beschränkten und unbeschränkten Isolation-Frameworks hinzugefügt. Es gibt ein neues Kapitel 6, das sich mit der Frage beschäftigt, was ein gutes Isolation-Framework ausmacht und wie ein Framework wie Typemock im Inneren funktioniert.

Ich verwende RhinoMocks nicht mehr. Lassen Sie die Finger davon. Es ist tot. Zumindest für den Augenblick. Ich benutze NSubstitute für Beispiele zu den Grundlagen von Isolation-Frameworks und ich empfehle auch FakeItEasy. Ich bin immer noch nicht begeistert von MOQ, was ich detaillierter in Kapitel 6 erläutern werde.

Dem Kapitel über die Implementation des Unit Testings auf der Organisationsebene habe ich weitere Techniken hinzugefügt.

Es gibt eine Reihe von Design-Änderungen im Code, der in diesem Buch abgedruckt ist. Meist habe ich aufgehört, Property Setters zu verwenden, und benutze häufig Constructor Injection. Einige Diskussionen zu den Prinzipien von SOLID habe ich hinzugefügt – aber nur so viel, um Ihnen Appetit auf das Thema zu machen.

Die auf das Build bezogenen Teile von Kapitel 7 enthalten ebenfalls neue Informationen. Seit dem ersten Buch habe ich eine Menge zur Build-Automatisierung und zu Mustern gelernt.

Ich rate von Setup-Methoden ab und stelle Alternativen vor, um Ihre Test mit der gleichen Funktionalität auszustatten. Ich benutze auch neuere Versionen von Nunit, sodass sich einige der neueren Nunit APIs in diesem Buch geändert haben.

Den Teil von Kapitel 10, der sich mit Tools im Hinblick auf Legacy-Code beschäftigt, habe ich aktualisiert.

Die Tatsache, dass ich in den letzten drei Jahren neben .NET mit Ruby gearbeitet habe, führte bei mir zu neuen Einsichten in Bezug auf Design und Testbarkeit. Das spiegelt sich in Kapitel 11 wider. Den Anhang zu Werkzeugen und Frameworks habe ich aktualisiert, neue Tools hinzugefügt und alte entfernt.

Wer dieses Buch lesen sollte

Dieses Buch ist für jeden geeignet, der Code entwickelt und daran interessiert ist, die besten Methoden für das Unit Testing zu erlernen. Alle Beispiele sind mit Visual Studio in C# geschrieben, weshalb die Beispiele insbesondere für .NET-Entwickler nützlich sein werden. Aber was ich unterrichte, passt genau so gut auf die meisten, wenn nicht auf alle objektorientierten und statisch typisierten Sprachen (VB.NET, Java und C++, um nur ein paar zu nennen). Egal, ob Sie ein Architekt sind, ein Entwickler, ein Teamleiter, ein QS-Mitarbeiter (der Code schreibt) oder ein Anfänger in der Programmierung, dieses Buch wurde für Sie geschrieben.

Meilensteine

Wenn Sie noch nie einen Unit Test geschrieben haben, dann ist es am besten, Sie lesen das Buch vom Anfang bis zum Ende, um das komplette Bild zu erhalten. Wenn Sie aber schon Erfahrung haben, dann fühlen Sie sich frei, so in den Kapiteln zu springen, wie es Ihnen gerade passt. Das vorliegende Buch ist in vier Hauptteile gegliedert.

Teil I bringt Sie beim Schreiben von Unit Tests von 0 auf 100. Kapitel 1 und 2 beschäftigen sich mit den Grundlagen, wie etwa der Verwendung eines Test-Frameworks (NUnit), und führen die grundlegenden Testattribute ein, wie z. B. `[Test]` und `[TestCase]`. Darüber hinaus werden hier auch verschiedene Prinzipien erläutert im Hinblick auf die Assertion, das Ignorieren von Tests, das Testen von Arbeitseinheiten (Unit of Work Testing), die drei Typen von Endergebnissen eines Unit Tests und der drei dazu benötigten Arten von Tests, nämlich Value Tests, State-Based Tests und Interaction Tests.

Teil II diskutiert fortgeschrittene Methoden zum Auflösen von Abhängigkeiten: Mock-Objekte, Stubs, Isolation-Frameworks und Muster zum Refactoring Ihres Codes, um diese nutzen zu können. Kapitel 3 stellt das Konzept der Stubs vor und veranschaulicht, wie man sie von Hand erzeugen und benutzen kann. In Kapitel 4 wird das Interaction Testing mit handgeschriebenen Mock-Objekten beschrieben. Kapitel 5 führt diese beiden Konzepte schließlich zusammen und zeigt, wie sie sich mithilfe von Isolation-(Mock-)Frameworks kombinieren lassen und ihre Automatisierung erlauben. Kapitel 6 vertieft das Verständnis für beschränkte und unbeschränkte Isolation-Frameworks und wie sie unter der Haube funktionieren.

Teil III beschäftigt sich mit verschiedenen Möglichkeiten, den Testcode zu organisieren, mit Mustern, um ihn auszuführen und seine Strukturen umzubauen (Refactoring), und mit bewährten Methoden zum Schreiben von Tests. In Kapitel 7 werden Testhierarchien vorgestellt und es wird dargestellt, wie Testinfrastruktur-APIs verwendet und wie Tests in den automatisierten Build-Prozess eingebunden werden. Kapitel 7 diskutiert bewährte Vorgehensweisen beim Unit Testing, um wartbare, lesbare und vertrauenswürdige Tests zu entwickeln.

Teil IV beschäftigt sich damit, wie sich Veränderungen in einer Organisation umsetzen lassen und wie man mit existierendem Code umgehen kann. In Kapitel 9 werden Probleme und Lösungen im Zusammenhang mit der Einführung des Unit Testings in einem Unternehmen diskutiert. Dabei werden auch einige Fragen beantwortet, die Ihnen in diesem Zusammenhang vielleicht gestellt werden. Kapitel 10 behandelt die Integration des Unit Testings in vorhandenen Legacy-Code. Es werden mehrere Möglichkeiten aufgezeigt, um festzulegen, wo mit dem Testen begonnen werden sollte, und es werden mehrere Tools vor-

gestellt, um »untestbaren« Code zu testen. Kapitel 11 diskutiert das vorbelastete Thema des Designs um der Testbarkeit willen und die Alternativen, die derzeit existieren.

Der Anhang listet eine Reihe von Tools auf, die Sie bei Ihren Testanstrengungen hilfreich finden könnten.

Codekonventionen und Downloads

Sie können den Quellcode von der Verlagsseite unter `www.mitp.de/9712` herunterladen. Hier finden Sie den Quellcode so wie er in diesem Buch abgedruckt ist (bitte beachten Sie hier auch die Readme-Datei zu den verschiedenen Versionen des Visual Studios). Den Quellcode des Originalbuches können Sie auch auf den folgenden Webseiten herunterladen: `https://github.com/royosherove/aout2`, `www.ArtOfUnitTesting.com` oder `www.manning.com/osherove2`, wo Sie auch weitere englischsprachige Informationen zum Buch, zu dessen Themen und zum Autor finden.

Sämtliche in diesem Buch enthaltenen Listings sind vom normalen Text klar zu unterscheiden und `in einer anderen Schriftart wie dieser` gesetzt. Manche Passagen sind **fett** gedruckt, um sie besonders hervorzuheben, oder auch mit einem besonderen Hinweissymbol gekennzeichnet, wenn sie gesondert referenziert werden.[I]

Softwareanforderungen

Um den Code in diesem Buch benutzen zu können, benötigen Sie zumindest Visual Studio C# Express (das kostenlos ist) oder die umfangreicheren (und kostenpflichtigen) Versionen[2]. Darüber hinaus sind auch NUnit (ein freies Open-Source-Framework) und andere Tools erforderlich, auf die an den entsprechenden Stellen hingewiesen wird. Alle erwähnten Tools sind entweder kostenlos, als Open-Source-Versionen oder aber als Testversionen verfügbar, die Sie ausprobieren können, während Sie dieses Buch lesen.

Danksagung

Ein großes Dankeschön geht an Michael Stephens und Nermina Miller von Manning Publications für ihre Geduld auf dem langen Weg beim Schreiben dieses Buches. Mein Dank geht ebenso an jeden bei Manning, der an der zweiten Auflage in der Produktion und hinter den Kulissen mitgearbeitet hat.

1 Anmerkung des Übersetzers: Die Kommentare innerhalb der Listings und auch der größte Teil der Strings wurden – der größeren Klarheit und der besseren Lesbarkeit wegen – übersetzt. In dieser Hinsicht unterscheiden sich die abgedruckten Listings vom englischen Quelltext, den Sie von den amerikanischen Webseiten herunterladen können. Der Quelltext auf der deutschen Seite des mitp-Verlages wurde genauso angepasst, wie Sie ihn in diesem Buch vorfinden. Sie haben also die Möglichkeit, wahlweise die unveränderten Original-Dateien von den amerikanischen Seiten oder die überarbeiteten von der deutschen Seite zu verwenden.

2 Anmerkung des Übersetzers: Zum Zeitpunkt der Veröffentlichung des amerikanischen Originaltitels war VS 2010 die aktuelle Version von Microsoft, auf die auch in diesem Buch Bezug genommen wurde. Hier wurden in der vorliegenden deutschen Ausgabe Verweise auf die inzwischen veröffentlichte Version 2013 ergänzt oder ersetzt. Beachten Sie bitte auch die der deutschen Version des Quellcodes beiliegende Readme-Datei, die Hinweise auf die verschiedenen Versionen enthält.

Dank gebührt auch Jim Newkirk, Michael Feathers, Gerard Meszaros und vielen anderen, die mich mit Inspiration und Ideen unterstützt und dieses Buch zu dem gemacht haben, was es ist. Besonders danke ich dir, Uncle Bob Martin, dafür, dass du es übernommen hast, das Vorwort für die zweite Auflage zu schreiben.

Die folgenden Rezensenten haben das Manuskript in verschiedenen Phasen seiner Entwicklung gelesen. Ich möchte ihnen für ihre wertvollen Anmerkungen danken: Aaron Colcord, Alessandro Campeism, Alessandro Gallo, Bill Sorensen, Bruno Sonnino, Camal Cakar, David Madouros, Dr. Frances Buontempo, Dror Helper, Francesco Goggi, Iván Pazmino, Jason Hales, Joao Angelo, Kaleb Pederson, Karl Metivier, Mertin Skurla, Martyn Fletcher, Paul Stack, Philip Lee, Pradeep Chellappan, Raphael Faria und Tim Sloan. Dank auch an Rickard Nilsson, der das abschließende Manuskript vor der Drucklegung Korrektur gelesen hat.

Ein abschließendes Wort des Dankes auch an die ersten Leser des Buches in Mannings »Early Access Program« für ihre Kommentare im Online-Forum. Sie haben dabei geholfen, dem Buch seine endgültige Form zu geben.

Teil I

Erste Schritte

Dieser Teil des Buches behandelt die Grundlagen des Unit Testings.

In Kapitel 1 werde ich definieren, was eine *Unit* ist und was »gutes« Unit Testing bedeutet und ich werde das Unit Testing mit dem Integration Testing vergleichen. Anschließend werfen wir einen kurzen Blick auf das Test-Driven Development und dessen Rolle in Bezug auf das Unit Testing.

In Kapitel 2 werden Sie damit loslegen, Ihren ersten Unit Test mithilfe von NUnit zu schreiben. Sie werden die grundlegende API von NUnit kennenlernen und Sie werden sehen, wie Asserts verwendet und wie Tests mit dem NUnit Test Runner durchgeführt werden.

In diesem Teil:

Kapitel 1

Die Grundlagen des Unit Testings

Dieses Kapitel behandelt

- die Definition eines Unit Tests
- das Unit Testing im Gegensatz zum Integration Testing
- ein einfaches Unit-Testing-Beispiel
- einen Einblick in die testgetriebene Entwicklung (Test-Driven Development)

Es beginnt immer mit dem ersten Schritt: das erste selbst geschriebene Programm, das erste in den Sand gesetzte Projekt – und das erste erfolgreiche. Das erste Mal vergisst man nicht und ich hoffe, Sie werden Ihren ersten Test auch nicht vergessen. Vielleicht haben Sie auch schon einige Tests geschrieben und erinnern sich an diese sogar als schlecht, misslich, langsam oder nicht wartbar (wie die meisten). In einem günstigeren Fall haben Sie aber möglicherweise eine großartige erste Erfahrung mit Unit Tests gemacht und lesen dies nun, um zu sehen, was da noch so kommen mag.

Dieses Kapitel wird zunächst die »klassische« Definition des Unit Tests analysieren und mit dem Konzept des Integration Testings vergleichen. Diese Unterscheidung ist für viele verwirrend. Dann schauen wir auf die Vor- und Nachteile des Unit Testing im Vergleich zum Integration Testing und entwickeln eine bessere Definition für einen »guten« Unit Test. Wir enden schließlich mit einem Blick auf die testgetriebene Entwicklung (»Test-Driven Development«), weil sie oft mit dem Unit Testing verbunden ist. Das ganze Kapitel über werde ich Konzepte anreißen, die im weiteren Verlauf des Buches genauer erläutert werden.

Lassen Sie uns mit der Definition dessen beginnen, was ein Unit Test sein sollte.

1.1 Unit Testing – Schritt für Schritt definiert

Unit Testing ist kein neues Konzept in der Softwareentwicklung. Es ist seit den frühen Tagen der Programmiersprache Smalltalk in den 1970er Jahren im Umlauf und erweist sich für Entwickler immer wieder als eine der besten Möglichkeiten, die Code-Qualität zu verbessern und gleichzeitig ein tieferes Verständnis für die funktionalen Anforderungen einer Klasse oder Methode zu entwickeln.

Kent Beck führte das Konzept des Unit Testings für Smalltalk ein und es hielt dann Einzug in viele andere Programmiersprachen, wodurch das Schreiben von Unit Tests zu einer ausgesprochen nützlichen Praxis in der Softwareentwicklung wurde. Bevor ich weitermache, sollte ich das Unit Testing zunächst genauer definieren. Hier die klassische Definition (basierend auf dem entsprechenden Wikipedia-Artikel[1]). Sie wird sich im Lauf des Kapitels weiterentwickeln bis zur abschließenden Definition in Abschnitt 1.4.

1 Dies bezieht sich auf den englischsprachigen Wikipedia-Artikel zum Thema Unit Testing.

Definition 1.0

Ein *Unit Test* ist ein Stück Code (meist eine Methode), das ein anderes Stück Code aufruft und anschließend die Richtigkeit einer oder mehrerer Annahmen überprüft. Falls sich die Annahmen als falsch erweisen, ist der Unit Test fehlgeschlagen. Eine *Unit* ist eine Methode oder Funktion.

Das Ding, für das Sie die Tests schreiben, wird »System Under Test« (SUT) genannt.

Definition

SUT steht für »System Under Test«, manche verwenden auch den Begriff *CUT* (»Class Under Test« oder »Code Under Test«). Wenn Sie etwas testen, bezeichnen Sie das, was Sie testen, als das SUT.

Eine ganze Zeit lang hatte ich das Gefühl (ja, Gefühl. In diesem Buch steckt keine Wissenschaft. Nur Kunst), dass diese Definition eines Unit Tests zwar technisch korrekt war, aber über die letzten paar Jahre änderte sich meine Idee davon, was eine *Unit* ist. Für mich bedeutet der Begriff Unit eine »Unit of Work« oder ein »Use Case« innerhalb des Systems.

Definition

Eine »Unit of Work« ist die Summe aller Aktionen, die zwischen dem Aufruf einer öffentlichen Methode innerhalb eines Systems und einem erkennbaren Endresultat beim Test dieses Systems stattfinden. Ein erkennbares Endresultat kann ausschließlich durch die öffentlichen APIs und das öffentliche Verhalten beobachtet werden, ohne den internen Zustand des Systems zu betrachten. Ein Endresultat ist jedes der folgenden:

- Die aufgerufene öffentliche Methode gibt einen Wert zurück (eine Funktion, die nicht vom Typ `void` ist).
- Es gibt eine erkennbare Änderung im Zustand oder Verhalten des Systems vor und nach dem Aufruf, die ohne das Abfragen eines privaten Zustands bestimmt werden kann. (Beispiel: Ein zuvor nicht existierender Benutzer kann sich am System anmelden oder die Eigenschaften des Systems, falls es sich dabei um einen endlichen Automaten handelt, ändern sich.)
- Es gibt einen Aufruf zu einem Fremdsystem, über das der Test keine Kontrolle hat und dieses Fremdsystem gibt entweder keinen Wert zurück oder der Rückgabewert wird ignoriert. (Beispiel: der Aufruf eines Logging-Systems, das nicht von Ihnen geschrieben wurde und zu dem Sie keinen Quellcode haben.)

Dieser Begriff von der Unit of Work bedeutet für manche, dass eine *Unit*, um ihren Zweck zu erreichen, so wenig beinhalten kann wie einen einzigen Methodenaufruf oder so viel wie mehrere Klassen und Funktionen.

Vielleicht kommt es Ihnen so vor, dass Sie die Größe einer zu testenden Unit of Work minimieren sollten. Mir zumindest kam es so vor, aber inzwischen habe ich meine Meinung geändert. Wenn eine Unit of Work größer ist und das Endresultat vom Benutzer der API leichter beobachtet werden kann, dann glaube ich, dass Ihre Tests besser zu warten sind. Wenn Sie stattdessen versuchen, die Größe einer Unit of Work zu minimieren, dann werden Sie schließlich die ganze Zeit über dem Benutzer der öffentlichen API Endresultate

vortäuschen, die gar keine sind, sondern lediglich Zwischenstopps auf dem Weg zum Hauptbahnhof. Ich werde dazu später mehr unter dem Thema der Überspezifizierung sagen (hauptsächlich in Kapitel 8).

Aktualisierte Definition 1.1

Ein *Unit Test* ist ein Stück Code (meist eine Methode), das eine Unit of Work aufruft und ein spezifisches Endresultat dieser Unit of Work überprüft. Falls sich die Annahmen über das Endresultat als falsch erweisen, ist der Unit Test fehlgeschlagen. Der Bereich eines Unit Tests kann alles umfassen von einer einzigen Methode bis hin zu einer Vielzahl von Klassen.

Egal, welche Programmiersprache Sie benutzen, einer der schwierigsten Aspekte der Definition des Unit Testings ist die Definition dessen, was in diesem Zusammenhang mit »gut« gemeint ist.

1.1.1 Die Bedeutung guter Unit Tests

Allerdings reicht es nicht aus zu verstehen, was eine Unit of Work ist.

Die meisten, die versuchen, Unit Tests für ihren Code zu schreiben, geben an irgendeinem Punkt auf oder führen die Unit Tests nicht wirklich aus. Stattdessen verlassen sie sich auf System- und Integrationstests, die viel später im Entwicklungszyklus des Produkts durchgeführt werden. Oder sie flüchten sich in manuelle Tests mithilfe selbst geschriebener Testanwendungen oder indem sie den zu testenden Code von dem Produkt, das sie entwickeln, ausführen lassen.

Es macht keinen Sinn, einen schlechten Unit Test zu schreiben, außer man bewegt sich zum ersten Mal auf diesem Gebiet und lernt dadurch, wie ein guter geschrieben wird. Wenn Sie einen Unit Test »schlecht schreiben«, ohne es zu merken, hätten Sie es genauso gut sein lassen und sich den Ärger sparen können, den dies später hinsichtlich der Wartbarkeit und der Einhaltung von Zeitplänen einbringt. Indem Sie definieren, was ein »guter« Unit Test ist, können Sie sicherstellen, dass Sie nicht mit einer falschen Vorstellung davon starten, was Ihr Ziel ist.

Um zu verstehen, was ein »guter« Unit Test ist, müssen Sie einen Blick darauf werfen, was Entwickler tun, wenn sie etwas testen.

Wie stellt man sicher, dass der Code richtig funktioniert?

1.1.2 Wir alle haben schon Unit Tests geschrieben (irgendwie)

Es mag Sie überraschen, dies zu hören, aber Sie haben schon einige Arten von Unit Tests selbst implementiert. Haben Sie je einen Entwickler getroffen, der seinen Code nicht getestet hat, bevor er ihn weitergab? Nun, ich auch nicht.

Sie mögen eine Konsolenanwendung benutzt haben, die verschiedene Methoden einer Klasse oder Komponente aufruft, oder Sie haben möglicherweise ein spezifisches WinForms oder WebForms UI geschrieben, das die Funktionalität der Klasse oder Komponente testet. Oder vielleicht haben Sie manuelle Testläufe mit verschiedenen Aktionen im UI des realen Produkts durchgeführt. Im Ergebnis haben Sie bis zu einem gewissen Grad sichergestellt, dass der Code gut genug funktioniert, um ihn an jemand anderen weitergeben zu können.

Abbildung 1.1 zeigt, wie die meisten Entwickler ihren Code testen. Das UI mag sich ändern, aber das Muster ist gewöhnlich das gleiche: Man verwendet von Hand ein externes Tool, um etwas wiederholt zu testen, oder man lässt die komplette Anwendung laufen und untersucht das Verhalten ebenfalls manuell.

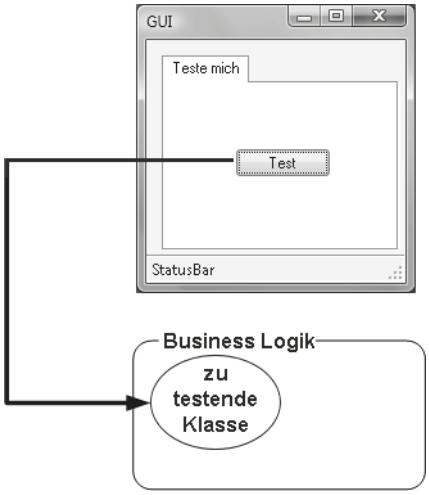

Abb. 1.1: Im klassischen Test verwenden Entwickler ein GUI (Graphical User Interface), um eine Aktion für die Klasse, die sie testen wollen, anzustoßen. Anschließend untersuchen sie das Resultat.

Diese Tests mögen nützlich sein und vielleicht auch der klassischen Definition eines Unit Tests nahekommen, aber sie sind weit davon entfernt, wie ich einen »guten« Unit Test in diesem Buch definieren werde. Das bringt uns zu der ersten und wichtigsten Frage, mit der sich ein Entwickler beschäftigen muss, wenn er die Eigenschaften eines »guten« Unit Tests definieren will: Was ist ein Unit Test und was ist er nicht?

1.2 Eigenschaften eines »guten« Unit Tests

Ein Unit Test *sollte* die folgenden Eigenschaften haben:

- Er sollte automatisiert und wiederholbar sein.
- Er sollte einfach zu implementieren sein.
- Einmal geschrieben, sollte er auch morgen noch relevant sein.
- Jeder sollte in der Lage sein, den Test auf einen Knopfdruck hin laufen zu lassen.
- Er sollte schnell ablaufen.
- Die Resultate sollten konsistent sein (die Tests geben immer den gleichen Wert zurück, wenn zwischen den Testläufen nichts geändert wird).
- Er sollte die volle Kontrolle über die getestete Unit haben.
- Er sollte komplett isoliert sein (er läuft völlig unabhängig von anderen Tests).
- Wenn er fehlschlägt, sollte es einfach zu erkennen sein, was erwartet wurde und wie das Problem lokalisiert werden kann.

Viele verwechseln den Akt des Testens ihrer Software mit dem Konzept des Unit Testings. Stellen Sie sich zu Anfang die folgenden Fragen zu den bisher von Ihnen selbst geschriebenen Tests:

- Kann ich einen Unit Test, den ich vor Wochen, Monaten oder Jahren geschrieben habe, laufen lassen und auswerten?
- Können alle Mitglieder meines Teams die Tests, die ich vor zwei Monaten geschrieben habe, ausführen und auswerten?
- Kann ich all die Tests, die ich geschrieben habe, innerhalb weniger Minuten durchlaufen lassen?
- Kann ich all die Tests, die ich geschrieben habe, auf Knopfdruck starten?
- Kann ich einen einfachen Test innerhalb weniger Minuten schreiben?

Falls Sie irgendeine dieser Fragen mit »Nein« beantwortet haben, ist es sehr wahrscheinlich, dass Ihre Implementierung kein Unit Test ist. Sie ist sicherlich *eine* Art von Test und *genauso* bedeutend wie ein Unit Test, aber sie hat ihre Nachteile im Vergleich zu Tests, bei denen alle Fragen mit »Ja« hätten beantwortet werden können.

»Was habe ich denn bisher gemacht?«, mögen Sie fragen. Sie haben *Integrationstests* durchgeführt.

1.3 Integrationstests

Für mich sind Integrationstests alle Tests, die nicht schnell sind, die nicht konsistent sind und die eine oder mehrere reale Abhängigkeit der zu testenden Unit beinhalten. Wenn also beispielsweise der Test die reale Systemzeit, das reale Dateisystem oder die reale Datenbank verwendet, dann hat er sich schon auf das Gebiet des Integration Testing begeben.

Wenn etwa der Test keine Kontrolle über die Systemzeit hat und der Test-Code die aktuelle Zeit über `DateTime.Now` bestimmt, dann ist der Test jedes Mal ein anderer, wenn er ausgeführt wird, denn er verwendet jedes Mal eine andere Zeit. Er ist nicht mehr konsistent.

Das ist per se nicht schlecht. Ich halte Integrationstests für wichtige Gegenstücke zu Unit Tests, aber sie sollten klar voneinander getrennt sein, um ein Gefühl für die »sichere grüne Zone« zu erhalten, über die ich später in diesem Buch schreiben werde.

Wenn ein Test eine reale Datenbank verwendet, dann läuft er nicht nur im Arbeitsspeicher ab und Aktionen sind schwerer zu löschen als bei reinen In-Memory Pseudo-Daten. Der Test wird mehr Zeit benötigen, was er wiederum nicht unter Kontrolle hat. Unit Tests sollten schnell sein. Integrationstests sind gewöhnlich viel langsamer. Sobald Sie Hunderte von Tests haben, zählt jede Sekunde.

Integrationstests erhöhen das Risiko für ein anderes Problem: das Testen zu vieler Dinge auf einmal.

Was passiert, wenn Ihr Auto eine Panne hat? Wie finden Sie heraus, was das Problem ist, ganz zu schweigen davon, wie es sich beheben lässt? Eine Maschine besteht aus vielen Teilsystemen, die zusammenarbeiten. Jedes ist abhängig von anderen, um letztlich das eine Ziel zu erreichen: ein fahrendes Auto. Wenn sich das Auto nicht mehr bewegt, kann die Ursache in einem dieser Teilsysteme liegen oder in mehr als nur einem. Es ist die *Integration* dieser Teilsysteme (oder Ebenen), die das Auto fahren lässt. Sie können sich die Bewegung des Autos als den ultimativen Integrationstest dieser Teile vorstellen, während das

Auto die Straße entlangfährt: Wenn der Test scheitert, versagen alle Teile zusammen; wenn er gelingt, sind alle Teile zusammen erfolgreich.

Das Gleiche passiert in der Softwareentwicklung. Die meisten Entwickler testen die Funktionstüchtigkeit der Software durch einen abschließenden Funktionalitätstest ihres UI. Das Klicken auf einen Button löst eine Serie von Ereignissen aus – Klassen und Komponenten arbeiten zusammen, um ein Resultat zu erzielen. Wenn der Test scheitert, scheitern alle diese Softwarekomponenten als ein Team und es ist möglicherweise schwierig, herauszubekommen, was genau das Scheitern der Gesamtoperation verursacht hat (siehe Abbildung 1.2).

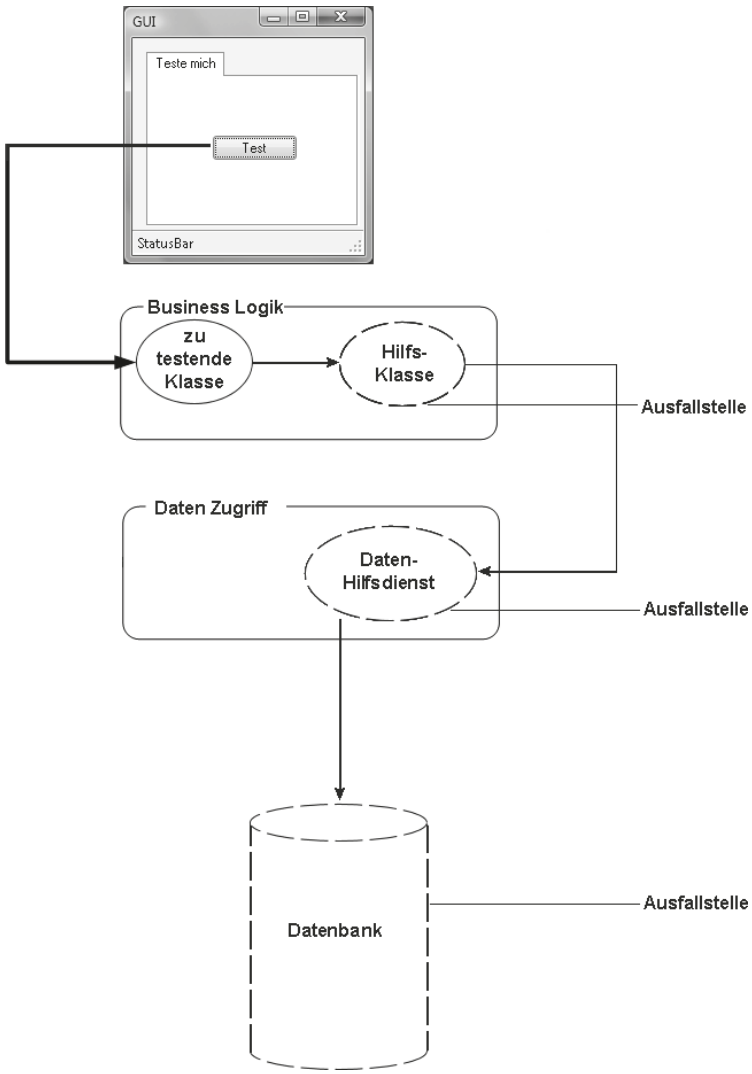

Abb. 1.2: Es gibt viele mögliche Schwachstellen in einem Integrationstest. Alle Komponenten müssen zusammenarbeiten und jede von ihnen kann fehlerhaft sein, was es schwieriger macht, die Ursache eines Fehlers zu finden.