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.

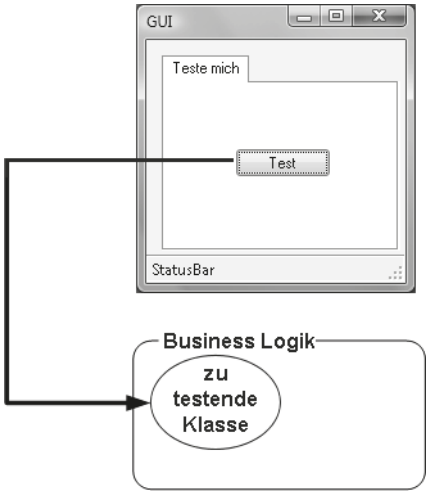


Abb. 1.1: Im klassischen Test verwenden Entwickler ein GUI (Graphical User Interface), um eine Aktion für die Klasse, die sie testen wollen, anzustoßen. Anschließend untersuchen sie das Resultat.

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).

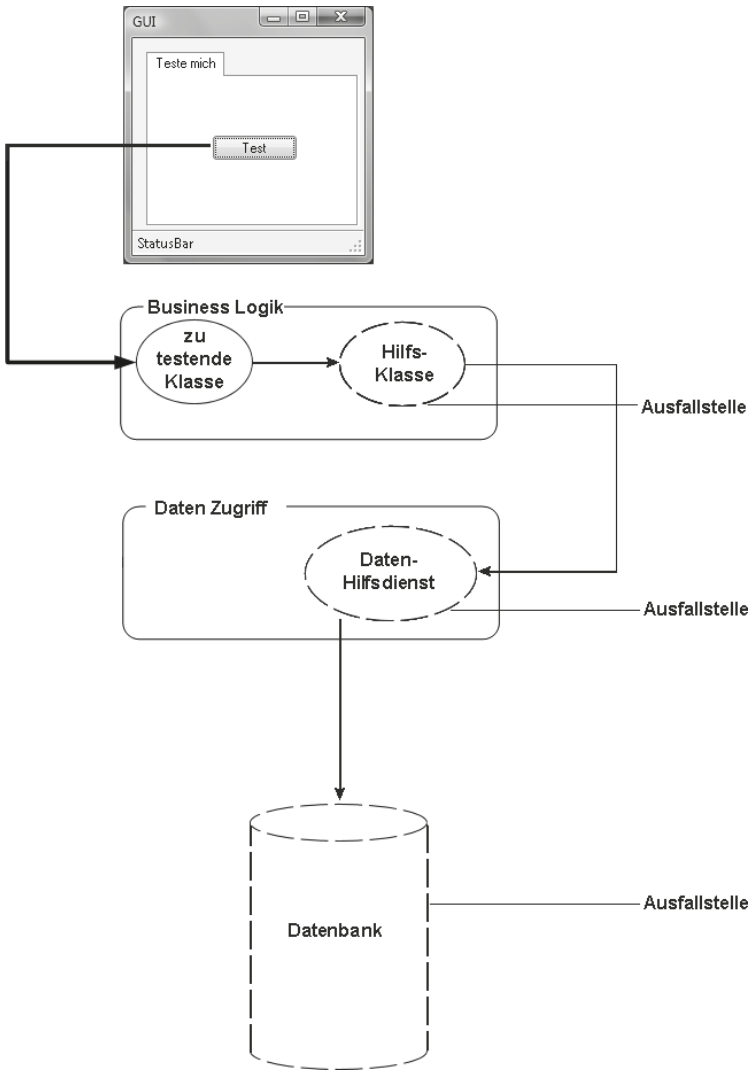


Abb. 1.2: Es gibt viele mögliche Schwachstellen in einem Integrationstest. Alle Komponenten müssen zusammenarbeiten und jede von ihnen kann fehlerhaft sein, was es schwieriger macht, die Ursache eines Fehlers zu finden.

The Complete Guide to Software Testing von Bill Hetzel (Wiley, 1993) definiert das Integration Testing als »einen systematischen Fortschritt des Testens, bei dem Software- und/oder Hardwareelemente *kombiniert und getestet* werden, bis das gesamte System integriert ist«. Diese Definition des Integration Testings greift ein bisschen zu kurz für das, was viele die ganze Zeit über tun, nicht als Teil eines System-Integrationstests, aber als Teil eines Entwicklungs- und Unit Tests.

Hier ist eine bessere Definition für das Integration Testing:

Definition

Integration Testing bedeutet, dass eine Unit of Work getestet wird, ohne die volle Kontrolle über sie oder eine oder mehrere ihrer realen Abhängigkeiten wie Zeit, Netzwerk, Datenbank, Threads, Zufallszahlengenerator usw. zu haben.

Noch mal zusammengefasst: Ein Integrationstest verwendet reale Abhängigkeiten; Unit Tests isolieren die Unit of Work von ihren Abhängigkeiten, wodurch die Konsistenz ihrer Resultate leicht erreicht und jeder Aspekt im Verhalten der Unit kontrolliert und simuliert werden kann.

Die Fragen von Abschnitt 1.2 können dabei helfen, einige der Nachteile des Integration Testings zu erkennen. Lassen Sie uns versuchen, die Qualitäten zu definieren, die für einen guten Unit Test wichtig sind.

1.3.1 Nachteile von nicht automatisierten Integrationstests im Vergleich zu automatisierten Unit Tests

Wenden wir die Fragen aus Abschnitt 1.2 auf Integrationstests an und überlegen wir, was Sie mit Unit Tests in der Praxis erreichen wollen:

- *Kann ich einen Unit Test, den ich vor Wochen, Monaten oder Jahren geschrieben habe, laufen lassen und auswerten?*

  Falls Sie das nicht können, wie können Sie dann wissen, ob ein Feature, das Sie zuvor eingebaut haben, immer noch funktioniert? Der Code einer Anwendung ändert sich regelmäßig. Aber wenn Sie, nachdem Sie den Code geändert haben, keine Testläufe für alle bisher funktionierenden Features durchführen können oder wollen, haben Sie möglicherweise neue Fehler eingebaut, ohne es zu ahnen. Ich nenne das »versehentliches Bugging« (»Accidental Bugging«) und es scheint häufig in der Endphase eines Softwareprojekts aufzutreten, wenn die Entwickler beim Beheben vorhandener Fehler unter Zeitdruck stehen. Manchmal bauen sie versehentlich neue Bugs ein, während sie alte beheben. Wäre es nicht großartig, innerhalb von drei Minuten zu wissen, dass Sie gerade Ihren Code ruiniert haben? Wie das erreicht werden kann, werden Sie etwas später in diesem Buch sehen.

Definition

Eine *Regression* ist eine oder mehrere Units of Work, die mal funktioniert haben, jetzt aber nicht mehr.

- *Können alle Mitglieder meines Teams die Tests, die ich vor zwei Monaten geschrieben habe, ausführen und auswerten?*

  Das hängt eng mit dem vorherigen Punkt zusammen, geht aber noch einen Schritt weiter. Sie möchten den Code eines anderen Entwicklers nicht beschädigen, während Sie etwas ändern. Viele Entwickler fürchten sich davor, *Legacy-Code* in älteren Systemen zu ändern, weil sie häufig nicht wissen, welcher andere Code noch von ihren Änderungen betroffen sein könnte. Im Wesentlichen riskieren sie dabei, das System in einen unbekannten Stabilitätszustand zu bringen.

  Nur wenige Dinge sind beängstigender, als nicht zu wissen, ob eine Anwendung überhaupt noch funktioniert, insbesondere wenn man den Code gar nicht selbst geschrieben hat. Wenn man weiß, dass man nichts ruiniert, wird man sich weniger zögerlich um unbekannten Programmcode kümmern, denn dann hat man das Sicherheitsnetz der Unit Tests.

  Gute Tests können von jedem verwendet und ausgeführt werden.

Definition

Wikipedia definiert *Legacy-Code* als »Quellcode, der sich auf ein nicht mehr unterstütztes oder hergestelltes Betriebssystem oder eine andere Computertechnologie bezieht«[2], aber viele Läden bezeichnen damit eine ältere Version der Anwendung als die, die derzeit noch gewartet wird. Der Ausdruck bezeichnet häufig solchen Code, der schwierig anzuwenden, schwierig zu testen und meist auch schwierig zu lesen ist.

Einer meiner Kunden hat Legacy-Code einmal ganz nüchtern so definiert: »Code, der funktioniert«. Viele definieren Legacy-Code hingegen gern als »Code ohne Tests«. In *Effektives Arbeiten mit Legacy Code* von Michael Feathers (mitp-Verlag, 2010) wird dies als die offizielle Definition für Legacy-Code verwendet und ebenso wird es auch in diesem Buch gehandhabt.

- *Kann ich all die Tests, die ich geschrieben habe, innerhalb weniger Minuten durchlaufen lassen?*

  Wenn Sie Ihre Tests nicht schnell ausführen können (Sekunden sind besser als Minuten), werden Sie sie seltener ausführen (täglich oder nur wöchentlich oder manchmal gar monatlich). Das Problem ist, dass Sie nach einer Code-Änderung so schnell wie möglich eine Rückmeldung benötigen, um zu sehen, ob immer noch alles funktioniert. Je mehr Zeit zwischen den Testläufen vergeht, desto mehr Änderungen führen Sie am System durch und desto mehr Stellen müssen Sie nach Bugs absuchen, wenn Sie feststellen, dass etwas nicht mehr funktioniert.

  Gute Tests sollten *schnell* sein.

- *Kann ich all die Tests, die ich geschrieben habe, auf Knopfdruck starten?*

  Wenn Sie es nicht können, heißt das wahrscheinlich, dass Sie Ihr Testsystem erst konfigurieren müssen, damit die Tests korrekt ablaufen (z. B. die Connection Strings für die Datenbank), oder dass Ihre Unit Tests nicht komplett automatisiert sind. Wenn Sie die

2 Aus dem englischsprachigen Wikipedia-Artikel zu »Legacy Code« übersetzt.

Unit Tests nicht komplett automatisieren können, werden Sie wahrscheinlich genauso wie jeder andere in Ihrem Team vermeiden, sie häufig laufen zu lassen.

Niemand fährt sich gerne bei der Konfiguration von Testdetails fest, nur um sicherzustellen, dass das System immer noch funktioniert. Entwickler haben Wichtigeres zu tun, etwa neue Features in das System einzubauen.

Gute Tests sollten einfach in ihrer ursprünglichen Form ausgeführt werden können und nicht von Hand ausgeführt werden müssen.

- *Kann ich einen einfachen Unit Test innerhalb weniger Minuten schreiben?*

  Eines der am einfachsten zu erkennenden Merkmale eines Integrationstests ist, dass man Zeit braucht, um ihn nicht nur auszuführen, sondern auch korrekt vorzubereiten und zu implementieren. Man braucht Zeit, um sich darüber klar zu werden, wie man ihn schreibt, denn es gibt viele interne und manchmal auch externe Abhängigkeiten (eine Datenbank kann als solche externe Abhängigkeit betrachtet werden). Wenn Sie den Test nicht automatisieren, sind solche Abhängigkeiten zwar weniger ein Problem, aber Sie verlieren auch die Vorteile eines automatisierten Tests. Je schwerer es ist, einen Test zu schreiben, desto unwahrscheinlicher ist es, dass Sie viele Tests schreiben oder sich auf etwas anderes konzentrieren als das »große Zeug«, das Ihnen gerade Sorgen bereitet. Eine der Stärken von Unit Tests ist es, dass sie dazu tendieren, jede Kleinigkeit, die schieflaufen kann, zu testen und eben nicht nur das große Zeug. Die Leute sind oft überrascht, wie viele Bugs sie in Code finden können, von dem sie dachten, er sei einfach und fehlerfrei.

  Wenn Sie sich nur auf die großen Tests konzentrieren, dann ist die *Abdeckung* der Logik in Ihren Tests geringer. Viele Teile der Kernlogik im Code werden nicht getestet (auch wenn Sie möglicherweise mehr Komponenten abdecken) und es können dort viele Bugs verborgen sein, an die Sie gar nicht gedacht haben.

  Gute Tests des Systems sollten einfach und schnell zu schreiben sein, sobald Sie herausgefunden haben, nach welchen Mustern Sie Ihr spezifisches Objektmodell testen wollen. Eine kleine Warnung: Auch erfahrene Unit Tester können leicht eine halbe Stunde und mehr Zeit benötigen, wenn sie herausfinden wollen, wie sie ihren ersten Unit Test schreiben sollen, wenn sie für das betreffende Objektmodel nie zuvor einen Unit Test geschrieben haben. Dies ist Teil der Arbeit und ganz normal. Der zweite und die weiteren Tests für dieses Objektmodell sollten dann aber leicht bewerkstelligt werden können.

Ausgehend von dem, was ich bisher zu der Frage, was ein Unit Test nicht ist und welche Eigenschaften für einen nützlichen Test vorhanden sein müssen, erläutert habe, kann ich nun damit beginnen, die primäre Frage dieses Kapitels zu beantworten: Was ist ein »guter« Unit Test?

1.4 Was Unit Tests »gut« macht

Lassen Sie mich nun, da ich die wichtigen Eigenschaften, die ein Unit Test haben sollte, erläutert habe, Unit Tests ein für alle Mal definieren.

Aktualisierte und finale Definition 1.2

Ein *Unit Test* ist ein automatisiertes Stück Code, das eine zu testende Unit of Work aufruft und dann einige Annahmen über ein einzelnes Endresultat dieser Unit prüft. Ein Unit Test wird fast immer mithilfe eines Unit-Testing-Frameworks erstellt. Er kann einfach geschrieben und schnell ausgeführt werden. Er ist vertrauenswürdig[3], lesbar und wartbar. Seine Ergebnisse sind konsistent, solange der Produktionscode nicht geändert wird.

Diese Definition scheint sicherlich ein bisschen viel verlangt, insbesondere wenn man bedenkt, wie viele Entwickler Unit Tests schlecht implementierten. Das bringt uns zu einem selbstkritischen Blick auf die Art, wie wir als Entwickler bisher Tests implementiert haben, verglichen damit, wie wir sie gern implementieren würden. (»Vertrauenswürdige, lesbare und wartbare« Tests werden im Detail in Kapitel 8 erläutert.)

In der vorigen Ausgabe dieses Buches war meine Definition eines Unit Tests ein wenig anders. Ich habe einen Unit Test so definiert, dass er nur »gegen den Ablaufsteuerungs-Code läuft«. Aber inzwischen halte ich das nicht mehr für richtig. Häufig wird Code ohne Logik als Teil einer Unit of Work verwendet. Selbst Properties ohne jegliche Logik werden von einer Unit of Work verwendet, weshalb sie von Tests nicht eigens adressiert werden müssen.

Definition

Ablaufsteuerungs-Code ist jegliches Stück Code, das irgendeine Art von Logik enthält, so klein es auch sein mag. Es enthält irgendetwas von Folgendem: eine `if`-Anweisung, eine Schleife, eine `switch`- oder `case`-Anweisung, Berechnungen oder irgendeine andere Art von entscheidungsfindendem Code.

Properties (Getters/Setters in Java) sind gute Beispiele für Code, der gewöhnlich keine Logik enthält und somit nicht eigens von den Tests adressiert werden muss. Es handelt sich um Code, der wahrscheinlich von einer Unit of Work, die Sie testen, verwendet wird. Es besteht daher keine Notwendigkeit, ihn direkt zu testen. Aber seien Sie auf der Hut: Sobald Sie irgendeine Überprüfung in eine Property einbauen, sollten Sie sicherstellen, dass die Logik getestet wird.

Im nächsten Abschnitt werfen wir einen Blick auf einen einfachen Unit Test, der komplett mit Code ohne die Hilfe eines Unit-Test-Frameworks implementiert wurde. (Wir werden uns Unit-Test-Frameworks in Kapitel 2 anschauen.)

1.5 Ein einfaches Unit-Test-Beispiel

Es ist durchaus möglich, einen automatisierten Test ohne ein Test-Framework zu schreiben. Tatsächlich habe ich viele Entwickler erlebt, die, als sie dabei waren, sich an die Automatisierung ihrer Tests zu gewöhnen, genau dies taten, bis sie schließlich die Test-Frameworks kennenlernten. In diesem Abschnitt werde ich zeigen, wie das Schreiben

3 Der im amerikanischen Originaltitel verwendete Begriff »trustworthy« wird in diesem Buch durchgängig mit »vertrauenswürdig« übersetzt. In diesem Zusammenhang geht die Bedeutung über eine reine »Zuverlässigkeit« hinaus. Siehe hierzu auch die Diskussion der »trustworthy tests« in Kapitel 7.

eines solchen Tests ohne Framework aussehen kann. Sie können dies dann mit dem Einsatz eines Frameworks in Kapitel 2 vergleichen.

Angenommen, Sie haben eine Klasse `SimpleParser` (siehe Listing 1.1), die Sie gerne testen würden. Sie hat eine Methode namens `ParseAndSum`, die als Parameter einen String der Länge 0 oder mehrere, durch Kommas getrennte Zahlen übernimmt. Falls nichts übergeben wird, gibt sie 0 zurück. Für den Fall einer einzelnen Zahl gibt sie deren Wert als ein `int` zurück. Und falls es mehrere Zahlen sind, addiert sie diese auf und gibt die Summe zurück (obwohl der Code im Augenblick nur mit den ersten beiden Fällen umgehen kann).

```
public class SimpleParser
{
    public int ParseAndSum(string numbers)
    {
        if(numbers.Length==0)
        {
            return 0;
        }
        if(!numbers.Contains(","))
        {
            return int.Parse(numbers);
        }
        else
        {
            throw new InvalidOperationException(
            "Ich kann bisher nur mit keiner oder einer Zahl umgehen!");
        }
    }
}
```

Listing 1.1: Eine einfache Parser-Klasse soll getestet werden.

Sie können das Projekt einer einfachen Konsolenanwendung anlegen, das eine Referenz auf das Assembly dieser Klasse hält, und Sie können eine Methode `SimpleParserTests` wie im folgenden Listing 1.2 schreiben. Die Testmethode ruft die *Produktionsklasse* (die zu testende Klasse) auf und prüft dann den Rückgabewert. Falls dieser nicht den Erwartungen entspricht, schreibt die Testmethode das in die Konsole. Sie fängt auch Ausnahmen auf und schreibt sie in die Konsole.

```
class SimpleParserTests
{
    public static void TestReturnsZeroWhenEmptyString()
    {
        try
        {
            SimpleParser p = new SimpleParser();
            int result = p.ParseAndSum(string.Empty);
```

```
            if(result!=0)
            {
                Console.WriteLine(
@"***SimpleParserTests.TestReturnsZeroWhenEmptyString:
-------
ParseAndSum sollte für einen leeren String 0 zurückgeben");
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

Listing 1.2: Eine einfache, codierte Methode, die die Klasse `SimpleParser` testet

Als Nächstes können Sie den gerade geschriebenen Test aufrufen, indem Sie eine einfache `Main`-Methode, wie im nächsten Listing dargestellt, in die Konsolenanwendung einbauen und ausführen lassen. Die `Main`-Methode wird hier als ein einfacher Testläufer verwendet, der die Tests nach und nach aufruft und sie in die Konsole schreiben lässt. Da es sich um ein lauffähiges Programm handelt, kann es ohne manuelle Eingriffe ausgeführt werden (jedenfalls solange die Tests keine interaktiven Benutzerdialoge öffnen).

```
public static void Main(string[] args)
{
    try
    {
        SimpleParserTests.TestReturnsZeroWhenEmptyString();
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
    }
}
```

Listing 1.3: Die codierten Tests werden über eine einfache Konsolenanwendung ausgeführt.

Es liegt in der Verantwortung der Testmethoden, alle auftretenden Ausnahmen aufzufangen und sie in die Konsole zu schreiben, damit sie die Ausführung der nachfolgenden Methoden nicht behindern. Sie können dann der `Main`-Funktion weitere Methodenaufrufe hinzufügen, wenn Sie das Projekt um neue Tests erweitern. Jeder Test trägt selber die Verantwortung dafür, die Problembeschreibung (falls es ein Problem gibt) in das Konsolenfenster auszugeben.

Offensichtlich ist das Schreiben eines solchen Tests eine Ad-hoc-Lösung. Wenn Sie eine Vielzahl derartiger Tests schreiben, werden Sie sich möglicherweise eine generische Methode `ShowProblem` wünschen, die für eine einheitliche Formatierung der Fehlermel-

dungen sorgt und von allen Tests verwendet werden kann. Sie könnten ebenso spezielle Hilfsmethoden hinzufügen, die auf Dinge, wie etwa Null-Objekte, leere Strings usw., prüfen würden, sodass Sie nicht immer wieder die gleichen langen Code-Zeilen in Ihren Tests schreiben müssen.

Das nächste Listing zeigt, wie der Test mit einer etwas allgemeineren Methode `ShowProblem` aussehen würde.

```
public class TestUtil
{
    public static void ShowProblem(string test,string message )
    {
        string msg = string.Format(@"
---{0}---
{1}
--------------------
", test, message);
        Console.WriteLine(msg);
    }
}

public static void TestReturnsZeroWhenEmptyString()
{
    //verwende .NET Reflection API, um den aktuellen Methodennamen
    //zu erhalten. Man könnte das auch direkt codieren, ist aber
    //gut, man kennt diese nützliche Technik
    string testName = MethodBase.GetCurrentMethod().Name;
    try
    {
        SimpleParser p = new SimpleParser();
        int result = p.ParseAndSum(string.Empty);
        if(result!=0)
        {
            //Wir rufen die Hilfsmethode auf
            TestUtil.ShowProblem(testName,
    "ParseAndSum sollte für einen leeren String 0 zurückgeben");
        }
    }
    catch (Exception e)
    {
        TestUtil.ShowProblem(testName, e.ToString());
    }
}
```

Listing 1.4: Die Verwendung einer etwas allgemeineren Implementierung der Methode `ShowProblem`

Unit-Testing-Frameworks können Sie dabei unterstützen, Hilfsfunktionen wie diese allgemeiner zu formulieren, wodurch es einfacher wird, Tests zu schreiben. Ich werde darüber in Kapitel 2 sprechen. Aber bevor wir dahin kommen, möchte ich noch eine wichtige Sache diskutieren: Nicht nur, *wie* Sie einen Unit Test schreiben, sondern auch, *wann* Sie ihn während des Entwicklungsprozesses schreiben. Das ist der Punkt, an dem die testgetriebene Entwicklung (Test-Driven Development) ins Spiel kommt.

1.6 Testgetriebene Entwicklung

Sobald Sie wissen, wie man strukturierte, wartbare und stabile Tests mithilfe eines Unit-Testing-Frameworks schreibt, stellt sich als Nächstes die Frage, wann man diese Tests schreibt. Viele haben das Gefühl, dass die Unit Tests am besten nach der Software geschrieben werden sollten, aber eine wachsende Anzahl bevorzugt das Schreiben der Unit Tests vor dem Schreiben des Produktionscodes. Dieser Ansatz nennt sich »Test-First« oder »Test-Driven Development« (TDD), zu Deutsch also »Testgetriebene Entwicklung«.

Anmerkung

Es gibt viele unterschiedliche Ansichten darüber, was testgetriebene Entwicklung wirklich bedeutet. Einige sagen, dass die Tests zuerst kommen, andere, dass man eine Menge Tests hat. Manche sehen sie als eine Art des Designs und wieder andere glauben, sie sei eine Art, das Verhalten des Codes mit nur einem Teil des Designs zu steuern. Um einen umfassenderen Überblick über die verschiedenen Ansichten zu TDD zu erhalten, können Sie einen Blick auf »The various meanings of TDD« in meinem Blog werfen (`http://osherove.com/blog/2007/10/08/the-various-meanings-of-tdd.html`). Im Rahmen dieses Buches meint TDD die »Test-First«-Entwicklung, wobei das Design bei dieser Technik nur eine Nebenrolle spielt (die in diesem Buch nicht diskutiert wird).

Abbildung 1.3 und Abbildung 1.4 zeigen die Unterschiede zwischen traditioneller Codierung und der testgetriebenen Entwicklung.

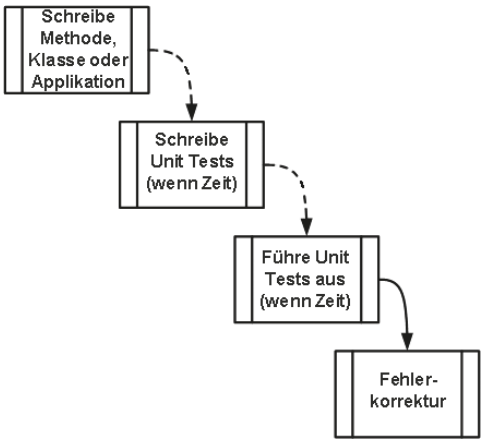


Abb. 1.3: Der traditionelle Weg, Unit Tests zu schreiben. Die gestrichelten Linien deuten optionale Aktionen an.

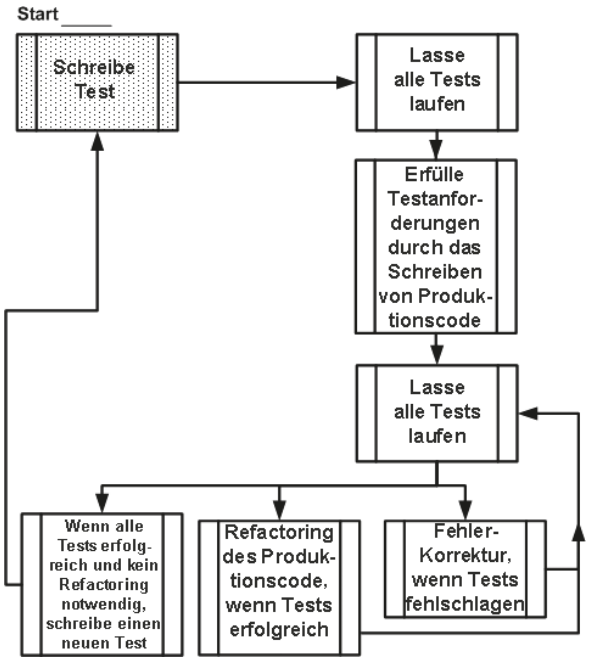


Abb. 1.4: Testgetriebene Entwicklung – ein Blick aus der Vogelperspektive. Beachten Sie den spiralförmigen Charakter des Prozesses: Schreibe einen Test, schreibe Code, passe ihn an (»Refactoring«), schreibe den nächsten Test. Die Abbildung zeigt die inkrementelle Natur von TDD: Kleine Schritte führen zu einem qualitativ guten Ergebnis.

Die testgetriebene Entwicklung unterscheidet sich in vieler Hinsicht von der traditionellen Entwicklung, wie Abbildung 1.4 zeigt. Man beginnt mit einem Test, der fehlschlägt; dann geht man weiter und schreibt den Produktionscode, schaut, ob er den Test besteht und überarbeitet dann entweder den Code oder beginnt mit dem nächsten Test, der fehlschlagen könnte.

Dieses Buch konzentriert sich auf die Technik des Schreibens »guter« Unit Tests und nicht auf die testgetriebene Entwicklung, obwohl ich ein großer Fan von TDD bin. Ich habe mehrere große Anwendungen und Frameworks mithilfe von TDD geschrieben und Teams geleitet, die es benutzten. Außerdem habe ich mehr als hundert Kurse und Workshops zum Thema TDD und Unit Testing gegeben. In meiner gesamten Laufbahn fand ich TDD nützlich, um hochwertigen Code, hochwertige Tests und ein besseres Design für meinen Code zu entwickeln. Ich bin überzeugt, dass es auch Ihnen von Nutzen sein kann, aber Sie erhalten es nicht ganz ohne Aufwand (Zeit für die Einarbeitung, Zeit für die Implementierung und anderes). Dennoch ist es die Mühe definitiv wert.

Es ist wichtig, sich klarzumachen, dass TDD weder den Projekterfolg noch robuste und wartbare Tests sicherstellt. Es ist recht einfach, sich in der Technik des TDD zu verheddern und der Art, wie Unit Tests geschrieben werden, keine Beachtung mehr zu schenken: ihrer Benennung, der Wartbarkeit und Lesbarkeit, ob sie die richtigen Dinge testen oder ob sie womöglich fehlerhaft sind.

Die Technik von TDD ist recht einfach:

1. *Schreiben Sie einen Test, der fehlschlägt, um zu zeigen, dass im Endprodukt ein Stück Code oder eine Funktionalität fehlt.*

   Der Test wird so geschrieben, als ob der Produktionscode bereits funktionieren würde, damit das Fehlschlagen des Tests wirklich bedeutet, dass ein Bug vorliegt. Wenn ich beispielsweise ein neues Feature in eine Taschenrechner-Klasse einbauen und den Wert `LastSum` speichern wollte, so würde ich einen Test schreiben, der überprüft, ob `LastSum` tatsächlich den korrekten Wert enthält. Der Test könnte erst kompiliert werden, nachdem ich den benötigten Code geschrieben habe (aber ohne die eigentliche Funktionalität, um sich die Zahl zu merken). Danach würde der Test laufen, aber fehlschlagen, weil ich diese Funktionalität noch nicht eingebaut habe.

2. *Lassen Sie den Test erfolgreich ablaufen, indem Sie Produktionscode schreiben, der die Anforderungen des Tests erfüllt.*

   Der Produktionscode sollte so einfach wie möglich gehalten werden.

3. *Überarbeiten Sie Ihren Code (Refactoring).*

   Wenn der Test erfolgreich durchläuft, können Sie entweder zum nächsten Unit Test weitergehen, oder aber Sie überarbeiten Ihren Code, um ihn beispielsweise lesbarer zu machen oder um eine Code-Duplizität zu entfernen etc.

Ein Refactoring kann nach dem Schreiben mehrerer Tests oder jedes Tests ausgeführt werden. Dies ist eine wichtige Praxis, denn sie stellt sicher, dass Ihr Code besser zu lesen und zu warten ist, während alle bisher geschriebenen Tests immer noch funktionieren.

Definition

Refactoring bedeutet die Änderung eines Code-Stücks, ohne die Funktionalität zu ändern. Wenn Sie jemals eine Methode umbenannt haben, dann haben Sie ein Refactoring durchgeführt. Wenn Sie jemals eine große Methode in mehrere kleine aufgeteilt haben, dann haben Sie ein Refactoring durchgeführt. Der Code tut immer noch das Gleiche, aber er ist einfacher zu warten, zu lesen, zu debuggen und zu ändern.

Die vorhergehenden Schritte klingen technisch, aber es steckt eine Menge Weisheit darin. Korrekt ausgeführt kann TDD Ihre Code-Qualität beträchtlich steigern, die Zahl der Bugs verringern, Ihr Vertrauen in den Code wachsen lassen, die für die Fehlersuche benötigte Zeit verkürzen, das Code-Design verbessern und Ihren Chef glücklicher machen. Schlecht durchgeführt lässt TDD den Zeitplan Ihres Projekts ins Wanken geraten, verschwendet Ihre Zeit, untergräbt Ihre Motivation und senkt die Qualität Ihres Codes. Es ist ein zweischneidiges Schwert und viele finden dies auf die harte Tour heraus.

Unter technischen Aspekten ist einer der größten Vorteile von TDD einer, von dem Ihnen keiner erzählt. Wenn Sie sehen, dass ein Test fehlschlägt, der später ohne jegliche Änderung am Test selber erfolgreich durchläuft, dann testen Sie im Wesentlichen den Test selbst. Wenn Sie erwarten, dass er fehlschlägt, und er läuft erfolgreich durch, dann haben Sie vielleicht einen Fehler in Ihrem Test oder Sie testen die falsche Sache. Wenn der Test zunächst fehlgeschlagen ist und Sie nun erwarten, dass er erfolgreich ist, aber immer noch fehlschlägt, dann könnte Ihr Test einen Fehler haben oder er erwartet das falsche Resultat.

Dieses Buch handelt von lesbaren, wartbaren und vertrauenswürdigen Tests, aber die beste Bestätigung erhalten Sie von Ihren Tests, wenn Sie sehen, dass sie dann fehlschlagen oder

erfolgreich sind, wenn sie es sollten. Dabei hilft Ihnen TDD eine ganze Menge, was einer der Gründe ist, weshalb Entwickler, die ihren Code mit TDD überprüfen, deutlich seltener debuggen, als wenn sie einfach nur Unit Tests im Nachhinein anwenden. Wenn sie dem Test vertrauen, dann haben sie nicht das Gefühl, ihn »nur sicherheitshalber« noch debuggen zu müssen. Und das ist die Art von Vertrauen, die Sie nur erhalten, wenn Sie beide Seiten des Tests sehen – das Fehlschlagen, wenn er fehlschlagen sollte, und den Erfolg, wenn er erfolgreich passieren sollte.

1.7 Die drei Schlüsselqualifikationen für erfolgreiches TDD

Um in der testgetriebenen Entwicklung erfolgreich zu sein, benötigen Sie drei Schlüsselqualifikationen: Sie müssen wissen, wie man gute Tests schreibt, Sie müssen sie vor dem Produktionscode schreiben (»test-first«) und Sie müssen sie gut designen.

- *Nur weil Sie Ihre Tests zuerst schreiben, bedeutet das nicht, dass sie wartbar, lesbar oder vertrauenswürdig sind.* In dem Buch, das Sie gerade lesen, dreht sich alles um die Fähigkeiten, die Sie für ein gutes Unit Testing benötigen.
- *Nur weil Sie lesbare und wartbare Tests schreiben, bedeutet das nicht, dass Sie die gleichen Vorteile erzielen, die Sie mit dem Test-first-Schreiben erreichen können.* Test-first-Fähigkeiten sind das, was die meisten TDD-Bücher auf dem Markt lehren, aber ohne auf die Fähigkeiten für gutes Testing einzugehen. Ich empfehle hier insbesondere Kents Becks *Test-Driven Development: by Example* (Addison-Wesley, 2002).
- *Nur weil Sie Ihre Tests zuerst schreiben und sie lesbar und wartbar sind, bedeutet das nicht, dass Sie letztlich ein gut designtes System erhalten werden.* Design-Fähigkeiten sind das, was Ihren Code elegant und wartbar macht. Ich empfehle zu diesem Thema die Bücher *Growing Object-Oriented Software, Guided by Tests* von Steve Freeman und Nat Pryce (Addison-Wesley, 2009) und *Clean Code* von Robert C. Martin (mitp-Verlag, 2009).

Ein pragmatischer Ansatz zum Einstieg in TDD besteht darin, jeden dieser drei Aspekte separat zu lernen. Das bedeutet, sich zunächst immer nur auf eine Fähigkeit auf einmal zu konzentrieren und die anderen vorläufig zu ignorieren. Ich empfehle das, weil ich schon oft gesehen habe, wie Leute versuchten, alle drei Fähigkeiten auf einmal zu lernen. Die hatten dann eine echt harte Zeit und gaben schließlich auf, weil diese Hürde einfach zu hoch war. Indem Sie mit einem inkrementellen Ansatz starten, nehmen Sie sich selbst die Furcht davor, auf den Gebieten, auf denen gerade nicht Ihr Fokus liegt, Fehler zu machen.

In Bezug auf die Reihenfolge dieses Lernansatzes schwebt mir kein bestimmtes Schema vor. Ich würde aber sehr gerne von Ihnen Ihre Erfahrungen und Empfehlungen hören, wenn Sie diese Fähigkeiten erlernen. Sie finden hier Kontakt-Links: `http://osherove.com`.

1.8 Zusammenfassung

In diesem Kapitel habe ich einen »guten« Unit Test so definiert, dass er die folgenden Eigenschaften besitzt:

- Er ist ein automatisiertes Stück Code, das eine andere Methode aufruft und dann einige Annahmen über das logische Verhalten dieser Methode oder Klasse prüft.
- Er wird mithilfe eines Unit-Testing-Frameworks geschrieben.

- Er kann sehr einfach geschrieben werden.
- Er läuft schnell ab.
- Er kann wiederholt von jedem Mitglied des Entwicklungsteams ausgeführt werden.

Um zu verstehen, was eine Unit ist, mussten Sie sich anschauen, was für eine Art von Tests Sie bisher durchgeführt haben. Sie haben sie als Integrationstests identifiziert, da ein Satz voneinander abhängiger Units getestet wird.

Es ist wichtig, den Unterschied zwischen Unit Tests und Integrationstests zu erkennen. Sie werden dieses Wissen in Ihrem Alltag als Entwickler nutzen, wenn Sie entscheiden müssen, wo Sie Ihre Tests platzieren, welche Art von Tests wann geschrieben werden müssen und welche Option sich besser für ein spezifisches Problem eignet. Es wird Ihnen ebenfalls helfen, Lösungen für Probleme mit Tests, die Ihnen schon Kopfzerbrechen bereiten, zu finden.

Wir haben ebenfalls einen Blick auf das Für und Wider von Integrationstests ohne ein unterstützendes Framework geworfen: Diese Art von Test ist schwierig zu schreiben und zu automatisieren, langsam in der Anwendung und erfordert einen gewissen Konfigurationsaufwand. Auch wenn Sie Integrationstests in Ihrem Projekt einsetzen wollen, können Unit Tests zu einem viel früheren Zeitpunkt im Prozess von Nutzen sein, wenn nämlich die Bugs kleiner und leichter zu finden sind und weniger Code überflogen werden muss.

Zu guter Letzt haben wir auch auf die testgetriebene Entwicklung geschaut, wie sie sich von der traditionellen Codierung unterscheidet und was ihre wesentlichen Vorteile sind. TDD unterstützt Sie dabei, sicherzustellen, dass die Code-Abdeckung Ihrer Tests (also wie viel des Codes Ihre Tests ausführen) sehr hoch ist (nahezu 100% des *logischen* Codes). TDD unterstützt Sie bei der Erstellung von Tests, denen man vertrauen kann. TDD »testet Ihre Tests«, indem es dafür sorgt, dass Ihre Tests fehlschlagen, wenn sie fehlschlagen sollten, und dass sie erfolgreich sind, wenn der Code funktioniert. TDD hat viele weitere Vorteile, wie die Unterstützung beim Design, die Reduzierung der Komplexität und eine Vorgehensweise, die Sie schwierige Probleme Schritt für Schritt angehen lässt. Aber Sie können TDD nicht erfolgreich über längere Zeit verwenden, ohne zu wissen, wie man gute Tests schreibt.

Im nächsten Kapitel werden Sie damit beginnen, Ihren ersten Unit Test mithilfe von NUnit, dem De-facto-Standard-Unit-Testing-Framework für .NET-Entwickler, zu schreiben.

Kapitel 2

Ein erster Unit Test

Dieses Kapitel behandelt

- Unit-Testing-Frameworks für .NET
- das Schreiben Ihres ersten Tests mit NUnit
- das Arbeiten mit NUnit-Attributen
- einen Einblick in die drei Ausgabetypen einer Unit of Work

Als ich anfing, Unit Tests mithilfe eines echten Unit-Testing-Frameworks zu schreiben, gab es dazu wenig an Dokumentation und die Frameworks, mit denen ich arbeitete, hatten keine ordentlichen Beispiele. (Zu dieser Zeit schrieb ich den Code meist mit VB 5 und 6.) Es war eine Herausforderung, den Umgang mit den Frameworks zu lernen, und ich habe mit sehr mäßigen Tests begonnen. Glücklicherweise haben sich die Zeiten geändert.

In diesem Kapitel werden Sie mit dem Schreiben von Tests anfangen, auch wenn Sie keine Ahnung haben, wo Sie starten sollen. Es wird Sie fit machen, praxisnahe Unit Tests mit einem Framework namens NUnit – einem .NET-Unit-Testing-Framework – zu schreiben. Es ist mein Lieblings-Framework, um in .NET Unit Tests durchzuführen, weil es so einfach zu benutzen ist, da man sich nicht viel merken muss und es viele großartige Features besitzt.

Es gibt andere Frameworks für .NET, auch solche mit mehr Features, aber ich beginne immer mit NUnit. Manchmal, wenn es notwendig wird, erweitere ich den Prozess später auf ein zusätzliches Framework. Wir werden uns anschauen, wie NUnit arbeitet, wie die Syntax ist und wie man es laufen lässt und ein Feedback erhält, wenn der Test fehlschlägt oder erfolgreich ist. Um das zu erreichen, werde ich ein kleines Softwareprojekt vorstellen, das wir das Buch hindurch verwenden und mit dem wir Techniken des Testens und die beste Praxis dafür untersuchen werden.

Es mag Ihnen vorkommen, als würde Ihnen NUnit in diesem Buch aufgezwungen werden. Warum nicht das in Visual Studio eingebaute MSTest-Framework verwenden? Die Antwort darauf besteht aus zwei Teilen:

- NUnit enthält bessere Funktionen als MSTest in Bezug auf das Schreiben von Unit Tests und von Testattributen, die helfen, wartbarere und lesbarere Tests zu schreiben.
- In den neueren Versionen von Visual Studio erlaubt es der eingebaute Test-Runner, auch Tests auszuführen, die mit anderen Frameworks einschließlich NUnit geschrieben wurden. Dazu müssen Sie nur den NUnit-Test-Adapter für Visual Studio via NuGet installieren. (NuGet wird später in diesem Kapitel erläutert.)

Das macht die Wahl, welches Framework ich benutzen möchte, ziemlich einfach.

Als Erstes müssen wir uns anschauen, was ein Unit-Testing-Framework ist und welche Möglichkeiten, die Sie sonst nicht hätten oder nicht nutzen würden, es Ihnen gibt.

2.1 Frameworks für das Unit Testing

Manuelle Tests sind Mist. Sie schreiben Ihren Code, Sie lassen ihn im Debugger laufen und drücken all die richtigen Tasten in Ihrer App, damit alles richtig läuft, und Sie wiederholen all das beim nächsten Mal, wenn Sie neuen Code geschrieben haben. Und Sie müssen auch dran denken, all den anderen Code zu testen, der vom neuen Code betroffen sein könnte. Noch mehr manuelle Arbeit. Großartige Sache.

Tests und Regressionstests komplett von Hand durchzuführen und die gleichen Aktionen noch mal und noch mal zu wiederholen wie ein Affe, ist fehleranfällig und zeitaufwendig. Das scheint mehr als alles andere in der Softwareentwicklung verhasst zu sein. Die Probleme werden durch den Einsatz von Tools gelindert. Unit-Testing-Frameworks helfen den Entwicklern, die Tests schneller mit einem Satz von bekannten APIs zu schreiben, sie automatisiert auszuführen und die Testresultate leichter zu begutachten. Und sie vergessen niemals etwas! Lassen Sie uns tiefer einsteigen und schauen, was sie zu bieten haben.

2.1.1 Was Unit-Testing-Frameworks bieten

Für viele von Ihnen, die das hier lesen, mögen die Tests, die sie bisher durchgeführt haben, limitiert gewesen sein:

- *Sie waren nicht strukturiert.*

  Jedes Mal mussten Sie das Rad neu erfinden, wenn Sie ein Feature testen wollten. Ein Test hat vielleicht wie eine Konsolenapplikation ausgesehen, ein anderer verwendete eine UI-Form und noch ein anderer eine Webform. Sie haben diese Zeit nicht auf das Testen verwendet und der Test hat die Anforderung, »einfach zu implementieren«, verfehlt.

- *Sie waren nicht wiederholbar.*

  Weder Sie noch die Mitglieder Ihres Teams konnten die Tests, die Sie in der Vergangenheit geschrieben haben, laufen lassen. Das verfehlt die Anforderung der »Wiederholbarkeit« und verhindert das Auffinden von Regression-Bugs. Mit einem Framework können Sie leichter und automatisch Tests schreiben, die wiederholbar sind.

- *Sie haben nicht alle wichtigen Teile Ihres Codes abgedeckt.*

  Die Tests haben nicht den ganzen Code, der eine Rolle spielt, geprüft. Damit ist der komplette Code, der die Logik enthält, gemeint, denn jeder einzelne Teil könnte einen potenziellen Bug beinhalten. (Die Getters und Setters der Properties gelten nicht als Logik, aber werden letztlich als Teil einer Unit of Work verwendet.) Wenn es einfacher wäre, die Tests zu schreiben, würden Sie dazu tendieren, mehr davon zu schreiben, und erhielten eine bessere Code-Abdeckung.

Kurz gesagt, was Sie bisher vermisst haben, war ein *Framework* für das Schreiben, Ausführen und Auswerten von Unit Tests und ihrer Ergebnisse.

Abbildung 2.1 zeigt die Bereiche in der Softwareentwicklung, in denen ein Unit-Testing-Framework eine Rolle spielt.

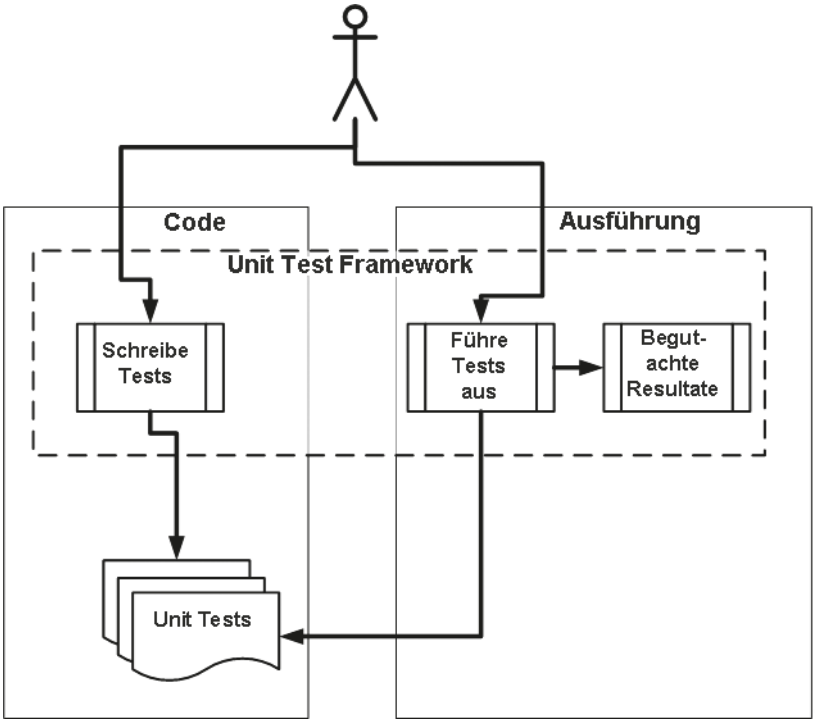


Abb. 2.1: Unit Tests werden wie Code mithilfe der Bibliotheken des Unit-Testing-Frameworks geschrieben. Dann werden die Tests von einem separaten Unit-Test-Tool oder innerhalb der IDE ausgeführt und die Resultate werden entweder vom Entwickler oder von einem automatischen Build-Prozess ausgewertet (entweder in der UI des Unit-Testing-Frameworks oder als Ausgabe-Text in der IDE).

Unit-Test-Praxis	Wie das Framework hilft
Schreiben Sie die Tests leicht und strukturiert.	Das Framework stellt dem Entwickler eine Klassenbibliothek bereit, die Folgendes enthält: ■ Basisklassen oder Interfaces, die vererbt werden können ■ Attribute, die im Code platziert werden können, um zu vermerken, welche von Ihren Methoden Tests sind ■ Assert-Klassen mit speziellen Methoden, die Sie aufrufen können, um den Code zu verifizieren
Führen Sie einen oder alle Unit Tests aus.	Das Framework stellt einen »Test-Runner« (ein Konsolen- oder GUI-Tool) zur Verfügung, der ■ die Tests im Code erkennt ■ die Tests automatisch ausführt ■ den Status während des Laufs anzeigt ■ über eine Kommandozeile automatisiert werden kann

Tabelle 2.1: Wie Unit-Testing-Frameworks den Entwicklern helfen, Tests zu schreiben, auszuführen und auszuwerten

Unit-Test-Praxis	Wie das Framework hilft
Werten Sie die Testresultate aus.	Der Test-Runner gibt Informationen wie die folgenden aus: ■ wie viele Tests ausgeführt wurden ■ wie viele Tests nicht ausgeführt wurden ■ wie viele Tests fehlschlugen ■ welche Tests fehlschlugen ■ die Gründe für die Fehlschläge ■ die Assert-Nachrichten, die Sie geschrieben haben ■ die Code-Zeilen, die fehlschlugen ■ wenn möglich, einen kompletten Stack Trace aller Ausnahmen, die zum Fehlschlag des Tests führten, und die Möglichkeit, zu allen Methodenaufrufen innerhalb des Call Stacks zu springen

Tabelle 2.1: Wie Unit-Testing-Frameworks den Entwicklern helfen, Tests zu schreiben, auszuführen und auszuwerten (Forts.)

Wie in Tabelle 2.1 gezeigt, sind Unit-Testing-Frameworks Code-Bibliotheken und Module, die den Entwicklern helfen, Unit Tests für ihren Code auszuführen. Sie haben aber auch noch einen anderen Zweck – nämlich die Durchführung der Tests als Teil eines automatisierten Builds, was ich in den späteren Kapiteln noch behandeln werde.

Während ich dieses Buch schreibe, existieren mehr als 150 Unit-Testing-Frameworks auf dem Markt – so gut wie eins für jede verwendete Programmiersprache. Eine gute Liste dieser Frameworks ist unter `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks` zu finden[1]. Beachten Sie, dass allein für .NET mindestens drei verschiedene aktiv genutzte Frameworks existieren: MSTest (von Microsoft), xUnit.net und NUnit. Unter diesen war NUnit der De-facto-Standard. Zurzeit scheint mir aber die Führung zwischen MSTest und NUnit umkämpft zu sein, einfach weil MSTest in Visual Studio eingebaut ist. Aber wenn ich die Wahl habe, nehme ich NUnit, wegen einiger der Fähigkeiten, die Sie später in diesem Kapitel sehen werden und auch im Anhang zu den Werkzeugen und Frameworks.

Anmerkung

Die Verwendung eines Unit-Testing-Frameworks stellt nicht sicher, dass Sie *lesbare*, *wartbare* oder *vertrauenswürdige* Tests schreiben oder dass sie die ganze Logik, die Sie testen möchten, auch tatsächlich abdecken. Wie Sie sicherstellen, dass Ihre Tests diese Eigenschaften haben, werden wir in Kapitel 7 und an verschiedenen anderen Stellen in diesem Buch genauer betrachten.

2.1.2 Die xUnit-Frameworks

Gemeinsam werden diese Unit-Testing-Frameworks als »xUnit-Frameworks« bezeichnet, weil ihre Namen meist mit den ersten Buchstaben der Sprache beginnen, für die sie geschrieben sind. Es gibt »CppUnit« für C++, »JUnit« für Java, »NUnit« für .NET und

1 Eine entsprechende Seite in der deutschen Wikipedia finden Sie hier: `http://de.wikipedia.org/wiki/Liste_von_Modultest-Software`

»HUnit« für die Haskell-Programmiersprache. Nicht alle folgen dieser Namenskonvention, aber die meisten.

In diesem Buch werden wir NUnit nutzen, ein .NET-Unit-Testing-Framework, das es einfach macht, Tests zu schreiben, auszuführen und die Resultate zu erhalten. NUnit ist als direkte Portierung des allgegenwärtigen JUnit für Java gestartet und hat seitdem große Fortschritte in seinem Design und in seiner Benutzerfreundlichkeit gemacht, wodurch es sich von seinem Vorläufer absetzt und neues Leben in das Ökosystem der Test-Frameworks bringt, das sich mehr und mehr ändert. Die Konzepte, die wir näher betrachten, werden für Java- und C++-Entwickler gleichermaßen verständlich sein.

2.2 Das LogAn-Projekt wird vorgestellt

Das Projekt, das wir in diesem Buch für die Tests verwenden, wird zunächst sehr einfach sein und nur eine Klasse enthalten. Während wir im Buch weiter voranschreiten, werden wir das Projekt um neue Klassen und Features erweitern. Wir nennen es das »LogAn-Projekt«, als Abkürzung für »Log and Notification« (»Protokollieren und Benachrichtigen«).

Hier das Szenario: Ihre Firma hat viele interne Produkte, um Anwendungen bei ihren Kunden vor Ort zu überwachen. Alle diese Produkte schreiben Logdateien, die sie in einem bestimmten Verzeichnis ablegen. Diese Logdateien werden alle in einem proprietären Format geschrieben, das Ihre Firma entwickelt hat und das von keinem existierenden Tool eines Drittanbieters geparst werden kann. Sie bekommen die Aufgabe, ein solches Produkt, »LogAn«, zu entwickeln. Es soll die Logdateien analysieren und besondere Fälle und Ereignisse darin finden. Wenn es diese Fälle und Ereignisse entdeckt, soll es alle Beteiligten darauf aufmerksam machen.

In diesem Buch werde ich Ihnen beibringen, Tests zu schreiben, die die Fähigkeiten von LogAn zur Analyse, Ereigniserkennung und Benachrichtigung verifizieren. Bevor wir loslegen, unser Projekt zu testen, müssen wir jedoch noch einen Blick darauf werfen, wie man einen Unit Test mit NUnit schreibt. Der erste Schritt ist seine Installation.

2.3 Die ersten Schritte mit NUnit

Wie jedes neue Tool muss es zunächst installiert werden. Weil NUnit ein Open-Source-Projekt ist und frei heruntergeladen werden kann, sollte diese Aufgabe recht simpel sein. Anschließend werden Sie sehen, wie man beginnt, mit NUnit einen Test zu schreiben, wie man die verschiedenen eingebauten Attribute, die NUnit mitbringt, nutzt und wie man seine Tests laufen lässt und echte Resultate erhält.

2.3.1 Die Installation von NUnit

Am besten und einfachsten ist es, NUnit über NuGet zu installieren. NuGet ist eine freie Erweiterung von Visual Studio, was es erlaubt, aus Visual Studio heraus mit ein paar Klicks oder einem einfachen Command-Text Referenzen auf populäre Bibliotheken zu suchen, herunterzuladen und zu installieren.

Ich rate Ihnen dringend, NuGet zu installieren, indem Sie in Visual Studio zum Erweiterungs-Manager-Menü unter EXTRAS gehen, dort auf ONLINE KATALOG klicken und dann den NuGet Package Manager installieren (wenn Sie nach »Höchster Rang« sortieren, fin-

den Sie ihn ganz oben)[2]. Vergessen Sie nicht, nach der Installation Visual Studio neu zu starten, und voilà – nun haben Sie ein sehr mächtiges und gleichzeitig einfaches Tool, um Ihren Projekten Referenzen hinzuzufügen und diese zu managen. (Wenn Sie aus der Ruby-Welt kommen, werden Sie bemerken, dass NuGet Ruby Gems und der GemFile-Idee ähnelt, obwohl es im Hinblick auf die Fähigkeiten zur Versionierung und zum Produktions-Deployment noch recht neu ist.)

Nachdem Sie nun NuGet installiert haben, können Sie das folgende Menü öffnen: EXTRAS|NUGET-PAKET-MANAGER|PAKET-MANAGER-KONSOLE. Anschließend tippen Sie `Install-Package NUnit` in das Textfenster, das erschienen ist. (Sie können auch die Tabulator-Taste verwenden, um die möglichen Kommandos oder Namen von Bibliotheks-Paketen automatisch vervollständigen zu lassen.)

Sobald alles gesagt und getan ist, sollten Sie eine hübsche Nachricht sehen: »NUnit wurde erfolgreich installiert«. NuGet hat dann eine zip-Datei mit den NUnit-Dateien heruntergeladen, einen Verweis auf das voreingestellte Projekt in der Combobox des Paket-Manager-Konsolenfensters hinzugefügt und Ihnen schließlich mitgeteilt, dass all das durchgeführt wurde. Sie sollten nun in Ihrem Projekt einen Verweis auf die `NUnit.Framework.dll` sehen.

Einen Kommentar zum NUnit-GUI – dies ist der einfache UI-Runner von NUnit. Ich werde auf dieses UI später in diesem Kapitel eingehen, aber gewöhnlich verwende ich es nicht. Betrachten Sie es eher als ein Lernwerkzeug, das Ihnen hilft zu verstehen, wie NUnit als Minimal-Werkzeug ohne Visual-Studio-Add-ons läuft. Es kommt auch nicht zusammen mit der NUnit-Version von NuGet. NuGet installiert nur die notwendigen DLLs, aber nicht das UI (das macht durchaus Sinn, denn Sie können mehrere Projekte haben, die NUnit verwenden, aber Sie brauchen nicht mehrere Versionen von seinem UI). Um das NUnit-UI zu erhalten, was ich auch etwas später in diesem Kapitel zeigen werde, können Sie `NUnit.Runners` aus NuGet installieren oder Sie können auf die Seite `NUnit.com` gehen und die vollständige Version von dort installieren. Auch diese vollständige Version kommt gemeinsam mit dem NUnit Console Runner, mit dessen Hilfe Sie Tests auf einem Build-Server laufen lassen.

Wenn Sie keinen Zugang zu NUnit haben, können Sie es von der Seite `www.NUnit.com` herunterladen und von Hand einen Verweis auf `nunit.framework.dll` hinzufügen.

Ein zusätzlicher Vorteil von NUnit ist, dass es sich um ein Open-Source-Produkt handelt und Sie den Quellcode von NUnit herunterladen, kompilieren und innerhalb der Grenzen der Open-Source-Lizenz frei verwenden. (Beachten Sie hierzu die Datei `license.txt` im Programmverzeichnis für weitere Details.)

Anmerkung

Während ich dies schreibe, ist die neueste Version von NUnit die Version 2.6.0. Die Beispiele in diesem Buch sollten mit den meisten zukünftigen Versionen des Frameworks kompatibel sein.

2 Anmerkung des Übersetzers: die verschiedenen Versionen von Visual Studio unterscheiden sich in den Bezeichnungen leicht und manche bringen auch den NuGet-Paket-Manager bereits mit.

Wenn Sie den manuellen Weg zur Installation von NUnit wählen, führen Sie das Setup-Programm aus, das Sie heruntergeladen haben. Das Installationsprogramm wird einen Shortcut zum GUI-Teil des NUnit Runners auf Ihren Desktop legen, aber die wesentlichen Programmdateien sollten in einem Ordner liegen, der sinngemäß `C:\Programme\NUnit-Net-2.6.0` heißt. Wenn Sie auf das NUnit-Desktop-Icon klicken, sehen Sie den Test-Runner wie in Abbildung 2.2.

Anmerkung

Die Verwendung der C# Express Edition von Visual Studio (oder höher) ist ausreichend für die Verwendung mit diesem Buch.

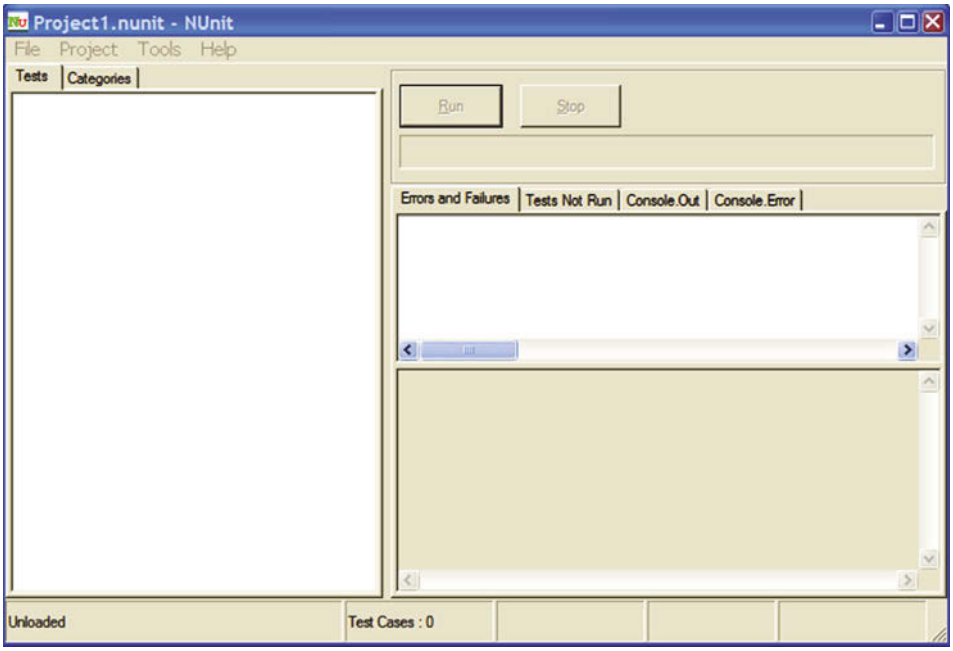


Abb. 2.2: Das NUnit-GUI ist im Wesentlichen in drei Bereiche unterteilt: Eine Baumstruktur listet die Tests an der rechten Seite auf, Nachrichten und Fehlermeldungen befinden sich rechts oben und die Stack-Trace-Informationen rechts unten.

2.3.2 Das Laden der Projektmappe

Wenn Sie den Code zu diesem Buch auf Ihrem Computer haben, dann können Sie die Projektmappe (Solution) `ArtOfUnitTesting2ndEd.Samples.sln` aus dem Code-Ordner in Ihr Visual Studio 2010 (oder höher) laden.

Wir beginnen mit dem Test der folgenden einfachen Klasse, die eine Methode (die Unit, die Sie testen) enthält:

```
public class LogAnalyzer
{
```

```
    public bool IsValidLogFileName(string fileName)
    {
        if(fileName.EndsWith(".SLF"))
        {
            return false;
        }
        return true;
    }
}
```

Bitte beachten Sie, dass ich mit Absicht ein `!` in der if-Bedingung weggelassen habe, damit diese Methode einen Fehler enthält – sie gibt `false` statt `true` zurück, wenn der Dateiname mit `.SLF` endet. Auf diese Weise können Sie sehen, was passiert, wenn ein Test im Test-Runner fehlschlägt.

Diese Methode mag nicht kompliziert aussehen, aber wir testen sie dennoch, um sicher zu sein, dass sie funktioniert. In der Praxis werden Sie alle Methoden, die eine Logik enthalten, testen wollen, egal, wie einfach sie erscheint. Die Logik kann fehlschlagen und Sie wollen es wissen, falls sie es tut. In den nächsten Kapiteln werden wir kompliziertere Szenarien und kompliziertere Logik testen.

Die Methode schaut auf die Dateierweiterung, um zu entscheiden, ob die Datei ein gültiges Log-File ist oder nicht. Unser erster Test wird sein, einen gültigen Dateinamen anzugeben und sicherzustellen, dass die Methode `true` zurückgibt.

Hier sind die ersten Schritte, um für die Methode `IsValidLogFileName` einen automatisierten Test zu schreiben:

1. Fügen Sie der Projektmappe (Solution) ein neues Projekt einer Klassenbibliothek hinzu, die Ihre Testklassen enthalten wird. Nennen Sie es `LogAn.UnitTests` (unter der Annahme, dass der ursprüngliche Projektname `LogAn.csproj` ist).
2. Fügen Sie dieser Bibliothek eine neue Klasse hinzu, die Ihre Testmethoden enthalten wird. Nennen Sie sie `LogAnalyzerTests` (unter der Annahme, dass Ihre zu testende Klasse `LogAnalyzer.cs` heißt).
3. Fügen Sie dem vorangegangenen Testfall eine neue Methode namens `IsValidLogFileName_BadExtension_ReturnsFalse()` hinzu.

Wir werden uns im weiteren Verlauf des Buches genauer mit den Standards der Testbenennung und Anordnung beschäftigen, aber die Grundregeln werden schon in Tabelle 2.2 aufgeführt.

Zu testendes Objekt	Auf der Testseite zu erzeugendes Objekt
Projekt	Erzeugen Sie ein Testprojekt namens `[ProjektImTest].UnitTests`.
Klasse	Erzeugen Sie zu einer Klasse in `ProjektImTest` eine weitere Klasse namens `[KlassenName]Tests`.

Tabelle 2.2: Die Grundregeln für die Platzierung und Benennung der Tests

Zu testendes Objekt	Auf der Testseite zu erzeugendes Objekt
Unit of Work (eine Methode oder eine logische Gruppierung von mehreren Methoden oder Klassen)	Erzeugen Sie für jede Unit of Work eine weitere Methode namens `[UnitOfWorkName]_[SzenarioImTest]_[ErwartetesVerhalten]`. Der Name der Unit of Work kann so einfach wie ein Methodenname sein (wenn das die ganze Unit of Work ist) oder auch abstrakter, wenn es sich um einen Use Case handelt, der mehrere Methoden und Klassen umfasst, so wie etwa `UserLogin` oder `RemoveUser` oder `Startup`. Vielleicht fühlen Sie sich wohler, wenn Sie mit Methodennamen beginnen und später zu abstrakteren Namen übergehen. Stellen Sie nur sicher, dass es sich auch um Methodennamen handelt und dass sie öffentlich sind, denn sonst repräsentieren sie nicht den Anfang einer Unit of Work.

Tabelle 2.2: Die Grundregeln für die Platzierung und Benennung der Tests (Forts.)

Der Name für unser LogAn-Testprojekt ist `Logan.UnitTests`. Der Name für die `LogAnalyzer`-Testklasse ist `LogAnalyzerTests`.

Hier sind die drei Teile des Testmethodennamens:

- `UnitOfWorkName` – der Name der Methode oder der Gruppe von Methoden oder Klassen, die getestet wird.
- `Szenario` – die Bedingungen, unter denen die Unit getestet wird, wie etwa »falsches Login« oder »ungültiger Benutzer« oder »gültiges Passwort«.
- `ErwartetesVerhalten` – was von der getesteten Methode unter den spezifischen Bedingungen erwartet wird. Das kann eine von drei Möglichkeiten sein: die Rückgabe eines Wertes (als echter Wert oder als Ausnahme), die Änderung des Systemzustands (wie das Hinzufügen eines neuen Benutzers, wodurch sich das System beim nächsten Login anders verhält) oder der Aufruf eines Third-Party-Systems (wie ein externer Webservice).

In unserem Test der Methode `IsValidLogFileName` ist das Szenario, dass Sie der Methode einen gültigen Dateinamen übergeben, und das erwartete Verhalten ist die Rückgabe des Wertes `true`. Der Name der Testmethode könnte `IsValidFileName_BadExtension_ReturnsFalse()` lauten.

Sollten Sie die Tests im Projekt des Produktions-Codes schreiben? Oder sollten Sie sie auslagern in ein testbezogenes Projekt? Gewöhnlich bevorzuge ich das Auslagern, denn es macht den ganzen Rest der testbezogenen Arbeit leichter. Auch sind viele nicht glücklich damit, Tests in ihrem Produktions-Code zu haben, denn das führt zu hässlichen, konditionalen Compilations-Schemata und anderen schlechten Einfällen, die den Code weniger lesbar machen.

Für mich ist das aber keine Frage des Glaubens. Mir gefällt auch die Idee, die Tests direkt neben der laufenden Produktions-App zu haben, wodurch man den Zustand auch nach dem Deployment testen kann. Das erfordert eine gewisse Vorsicht, aber es erfordert *nicht*, dass Sie Tests und Produktions-Code im gleichen Projekt haben. Sie können tatsächlich den Kuchen essen und ihn trotzdem behalten.

Sie haben das NUnit-Testing-Framework bisher noch nicht benutzt, aber sie sind nahe dran. Sie müssen noch eine Referenz auf das zu testende Projekt in das neue Testprojekt

einfügen. Führen Sie dazu einen Rechtsklick auf das Testprojekt aus und wählen Sie VERWEIS HINZUFÜGEN. Dann wählen Sie die Registerkarte PROJEKTE und anschließend das LogAn-Projekt aus[3].

Als Nächstes lernen Sie, wie die zu ladende Testmethode markiert und von NUnit automatisch ausgeführt wird. Stellen Sie aber zunächst sicher, dass Sie die NUnit-Referenz, wie in Abschnitt 2.3.1 erklärt, entweder via NuGet oder von Hand hinzugefügt haben.

2.3.3 Die Verwendung der NUnit-Attribute in Ihrem Code

NUnit verwendet ein Attribut-Schema, um Tests zu erkennen und zu laden. So, wie Lesezeichen in einem Buch, helfen Attribute dem Framework, zu erkennen, was die wichtigen Teile in dem Assembly sind, das es gerade lädt, und welche Teile Tests sind, die ausgeführt werden sollen.

NUnit stellt ein Assembly zur Verfügung, das spezielle Attribute enthält. Sie müssen nur eine Referenz auf das Assembly `NUnit.Framework` in Ihr Testprojekt (nicht in Ihren Produktionscode!) einbauen. Dieses finden Sie auf der .NET-Registerkarte im Dialog VERWEIS HINZUFÜGEN (Sie müssen das nicht machen, wenn Sie NuGet zur Installation von NUnit verwendet haben). Tippen Sie `Nunit` ein und Sie werden mehrere Assemblies sehen, die mit diesem Namen anfangen. Fügen Sie nun `nunit.framework.dll` als eine Referenz zu Ihrem Testprojekt hinzu (wenn Sie es von Hand installiert haben und nicht über NuGet).

Der NUnit Runner benötigt mindestens zwei Attribute, um zu wissen, was er ausführen soll:

- Das `[TestFixture]`-Attribut, das auf eine Klasse verweist, die die automatisierten NUnit-Tests beinhaltet. (Wenn Sie das Wort »Fixture« durch »Class« ersetzen, macht es wesentlich mehr Sinn, allerdings nur als gedankliche Übung. Der Code wird sich nicht übersetzen lassen, wenn Sie ihn auf diese Weise wörtlich ändern.) Wenden Sie das Attribut auf Ihre neue Klasse `LogAnalyzerTests` an.
- Das `[Test]`-Attribut, das auf eine Methode angewendet werden kann, um sie als einen automatischen Test zu markieren, der aufgerufen werden soll. Wenden Sie das Attribut auf Ihre neue Testmethode an.

Wenn Sie fertig sind, sollte Ihr Testcode folgendermaßen aussehen:

```
[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void IsValidFileName_BadExtension_ReturnsFalse()
    {
    }
}
```

3 Anmerkung des Übersetzers: auch hier haben sich die Bezeichnungen für Visual Studio 2013 leicht geändert. Sie gehen jetzt über Hinzufügen | Verweis | Projektmappe.

Hinweis

NUnit verlangt von den Testmethoden, dass sie öffentlich sind, den Typ `void` haben und dass sie in der einfachsten Konfiguration keine Parameter besitzen. Aber Sie werden sehen, dass sie manchmal doch Parameter haben können!

An diesem Punkt haben Sie Ihre Klasse und eine Methode zur Ausführung markiert. Was für einen Code auch immer Sie jetzt in Ihre Testmethode einbinden, er wird von NUnit ausgeführt, wann immer Sie es möchten.

2.4 Sie schreiben Ihren ersten Test

Wie testen Sie Ihren Code? Ein Unit Test umfasst gewöhnlich drei wesentliche Aktionen:

- *Arrangiere (Arrange)* die Objekte, erzeuge und konfiguriere sie wie benötigt.
- *Agiere (Act)* mit einem Objekt.
- *Bestätige (Assert)*, dass sich etwas wie erwartet verhält.

Hier ist ein einfaches Code-Stück mit allen drei Aktionen, wobei der `Assert`-Teil von der `Assert`-Klasse des NUnit-Frameworks übernommen wird:

```
[Test]
public void IsValidFileName_BadExtension_ReturnsFalse()
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName("dateimitungültigererweiterung.foo");

    Assert.False(result);
}
```

Bevor wir weitermachen, müssen Sie noch etwas mehr über die Klasse `Assert` erfahren, denn sie ist ein wichtiger Bestandteil beim Schreiben von Unit Tests.

2.4.1 Die Klasse Assert

Die Klasse `Assert` hat statische Methoden und befindet sich im Namespace `NUnit.Framework`. Sie ist die Brücke zwischen Ihrem Code und dem NUnit-Framework und ihre Aufgabe ist die Deklaration, dass eine bestimmte Annahme gelten soll. Wenn sich die Argumente, die der Klasse `Assert` übergeben werden, von dem unterscheiden, was Sie bestätigen wollen, dann bemerkt NUnit, dass der Test fehlgeschlagen ist, und informiert Sie. Sie können optional der `Assert`-Klasse einen Nachrichtentext übergeben, mit dem Sie im Fall des Fehlschlags benachrichtigt werden wollen.

Die Klasse `Assert` hat viele Methoden, wobei die wichtigste `Assert.IsTrue (`*`ein boolescher Ausdruck`*`)` ist, die einen booleschen Ausdruck verifiziert. Aber sie besitzt noch zahlreiche weitere Methoden, die Sie als syntaktische Sahnehäubchen betrachten können, um

die jeweilige Verwendung von `Assert` klarer zu machen (so wie `Assert.False`, das wir verwendet haben).

Die folgende Passage verifiziert, dass ein erwartetes Objekt oder ein erwarteter Wert gleich dem aktuellen ist:

```
Assert.AreEqual(expectedObject, actualObject, message);
```

Hier ist ein Beispiel:

```
Assert.AreEqual(2, 1+1, "Mathe klappt nicht");
```

Die nächste Methode verifiziert, dass zwei Argumente das gleiche Objekt referenzieren:

```
Assert.AreSame(expectedObject, actualObject, message);
```

Hier ist ein Beispiel:

```
Assert.AreSame(int.Parse("1"), int.Parse("1"),
               "dieser Test sollte fehlschlagen");
```

`Assert` ist einfach zu lernen, zu benutzen und zu merken.

Bitte beachten Sie auch, dass alle `Assert`-Methoden einen letzen Parameter vom Typ `string` akzeptieren, der im Fall eines Testfehlers zusätzlich zur Ausgabe des Frameworks angezeigt wird. Bitte verwenden Sie ihn niemals nie (er ist immer optional). Stellen Sie einfach sicher, dass Ihr Test-Name erklärt, welches Verhalten erwartet wird. Häufig schreiben die Leute das Offensichtliche wie »Test fehlgeschlagen« oder »x statt y erwartet«, was das Framework sowieso ausgibt. Ähnlich wie bei Kommentaren im Code sollte Ihr Methoden-Name klarer sein, wenn Sie diesen Parameter verwenden müssen.

Lassen Sie uns nun, da wir die API-Grundlagen durchgegangen sind, einen Test ausführen.

2.4.2 Sie führen Ihren ersten Test mit NUnit aus

Es wird Zeit, Ihren ersten Test auszuführen und zu sehen, ob er erfolgreich ist oder nicht.

Es gibt mindestens vier Arten, diesen Test laufen zu lassen:

- Über das NUnit-GUI
- Über den Visual-Studio-2013-Test-Runner mit der NUnit-Runner-Erweiterung namens NUnit Test Adapter (in der NUget Gallerie)
- Über den ReSharper Test Runner (ein bekanntes, kommerzielles Plug-in für VS)
- Über den TestDriven.NET Test Runner (ein anderes, bekanntes, kommerzielles Plug-in für VS)

Obwohl sich dieses Buch nur mit dem NUnit-GUI beschäftigt, verwende ich persönlich NCrunch, das schnell ist und automatisch läuft, aber auch Geld kostet. (Dieses Tool und andere werden im Anhang behandelt.) Es liefert ein einfaches, schnelles Feedback im Visual-Studio-Editor-Fenster. Ich finde, dass dieser Test-Runner einen guten Begleiter für

die testgetriebene Entwicklung in echten Anwendungen abgibt. Mehr darüber können Sie auf `www.ncrunch.net` herausfinden.

Um den Test mit dem NUnit-GUI laufen zu lassen, müssen Sie ein Assembly erstellen (in diesem Fall eine *.dll*-Datei), die Sie zur Untersuchung an NUnit übergeben können. Nachdem Sie das Projekt erstellt haben, ermitteln Sie den Pfad zu dem Assembly, das erzeugt wurde.

Rufen Sie dann das NUnit-GUI auf. (Wenn Sie NUnit von Hand installiert haben, finden Sie das Icon auf Ihrem Desktop. Wenn Sie NUnit.Runners über NUget installiert haben, finden Sie das NUnit-GUI-EXE-File im Packages-Ordner unter dem Root-Verzeichnis Ihrer Solution.) Wählen Sie FILE|OPEN. Geben Sie den Namen Ihres Test-Assemblys an. Auf der linken Seite werden Sie dann Ihren einzelnen Test mit der Klassen- und Namespace-Hierarchie Ihres Projekts sehen, wie in Abbildung 2.3 dargestellt. Klicken Sie auf den RUN-Button, um Ihre Tests zu starten. Die Tests werden automatisch nach ihrem Namespace (Assembly, Typname) gruppiert, weshalb Sie die Testläufe auf bestimmte Typen oder Namespaces einschränken können. (Gewöhnlich werden Sie alle Tests ausführen wollen, um ein besseres Feedback auf Fehlschläge zu erhalten.)

Sie haben einen Test, der fehlschlägt, was nahelegt, dass da ein Bug im Code ist. Jetzt ist es an der Zeit, den Fehler zu beheben und zu schauen, ob der Test dann erfolgreich ist. Ändern Sie den Code und fügen Sie das fehlende `!` in der if-Bedingung hinzu, damit er wie folgt aussieht:

```
if(!fileName.EndsWith(".SLF"))
{
    return false;
}
```

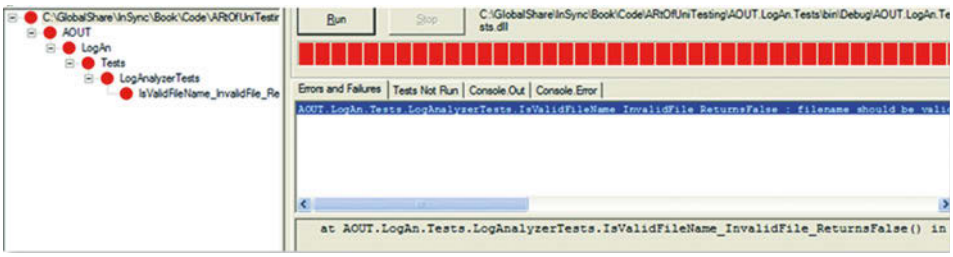


Abb. 2.3: NUnit zeigt die fehlgeschlagenen Tests an drei Stellen: die Testhierarchie auf der linken Seite wird rot, die Fortschrittsanzeige oben wird rot und alle Fehler werden auf der rechten Seite gezeigt.

2.4.3 Sie fügen positive Tests hinzu

Sie haben gesehen, dass falsche Erweiterungen markiert werden, aber wer sagt, dass die richtigen von dieser kleinen Methode akzeptiert werden? Falls wir auf die testgetriebene Art vorgehen würden, würde ein fehlender Test hier auffallen, aber da Sie die Tests erst nach dem Code schreiben, müssen Sie sich etwas einfallen lassen, um alle Wege abzudecken. Das folgende Listing fügt ein paar Tests hinzu, um zu sehen, was passiert, wenn Sie eine

Datei mit der richtigen Endung angeben. Eine Datei hat eine kleingeschriebene Endung, die der anderen ist großgeschrieben.

```
[Test] public void
IsValidLogFileName_GoodExtensionLowercase_ReturnsTrue()
{
    LogAnalyzer analyzer = new LogAnalyzer();
    bool result = analyzer.IsValidLogFileName("dateimitgültigererweiterung.slf");

    Assert.True(result);
}

[Test] public void
IsValidLogFileName_GoodExtensionUppercase_ReturnsTrue()
{
    LogAnalyzer analyzer = new LogAnalyzer();
    bool result = analyzer.IsValidLogFileName("dateimitgültigererweiterung.SLF");

    Assert.True(result);
}
```

Listing 2.1: Die Dateinamen-Überprüfungslogik von LogAnalyzer wird getestet.

Wenn Sie jetzt ein Rebuild Ihrer Solution ausführen, werden Sie sehen, dass NUnits UI die Änderung des Assemblys erkennt und es automatisch nachladen wird. Wenn Sie die Tests erneut ausführen, wird der Test mit der Kleinschreibung fehlschlagen. Sie müssen den Produktionscode so ändern, dass er eine Case-insensitive String-Überprüfung verwendet:

```
public bool IsValidLogFileName (string fileName)
{
    if (!fileName.EndsWith(".SLF",
StringComparison.CurrentCultureIgnoreCase))
    {
        return false;
    }
    return true;
}
```

Wenn Sie die Tests jetzt erneut ausführen, sollten sie alle erfolgreich durchlaufen und Sie einen hübschen grünen Balken in der NUnit-GUI sehen.

2.4.4 Von Rot nach Grün: das erfolgreiche Ausführen der Tests

Das GUI von NUnit basiert auf einer einfachen Idee: Alle Tests müssen erfolgreich sein, damit man mit einem grünen Signal weitermachen kann. Wenn nur einer der Tests fehl-

schlägt, sehen Sie eine rote Fortschrittsleiste, damit Sie wissen, dass irgendetwas mit dem System (oder Ihren Tests) nicht stimmt.

Dieses »Rot/Grün«-Konzept ist in der Unit-Test-Welt allgemein verbreitet, insbesondere in der testgetriebenen Entwicklung. Sein Mantra ist »Rot/Grün-Überarbeitung« (»Red-Green-Refactor«), was bedeutet, dass Sie mit einem Test beginnen, der zunächst fehlschlägt, danach erfolgreich durchläuft, und wo Sie zuletzt den Code überarbeiten, um ihn lesbarer und wartbarer zu machen.

Tests können auch fehlschlagen, wenn plötzlich eine unerwartete Ausnahme geworfen wird. Ein Test, der wegen einer unerwarteten Ausnahme stoppt, wird von den meisten, wenn nicht von allen, Test-Frameworks als ein fehlgeschlagener Test gewertet. Das ist Teil der Geschichte – manchmal haben Sie Fehler in Form einer Exception, die Sie nicht erwartet haben.

Später in diesem Kapitel werden Sie eine Version eines Tests sehen, der das Werfen einer Ausnahme als spezifisches Resultat oder Verhalten erwartet. Solche Tests schlagen fehl, wenn die Ausnahme nicht geworfen wird.

2.4.5 Test-Code-Gestaltung

Beachten Sie, dass die von mir geschriebenen Tests einige Charakteristiken in Bezug auf Gestaltung und Lesbarkeit enthalten, die von »Standard«-Code abweichen. Der Testname kann sehr lang sein, aber die Unterstriche helfen dabei, alle wichtigen Informationen einzubauen. Beachten Sie auch, dass zwischen den Arrange-, Act- und Assert-Abschnitten Leerzeilen in die Tests eingefügt sind. Dies hilft mir dabei, die Tests schneller zu lesen und Probleme schneller zu finden.

Ich versuche auch, die Assert-Abschnitte so weit wie möglich von den Act-Abschnitten zu trennen. Ich setze ein Assert lieber auf einen Wert und nicht direkt auf einen Funktionsaufruf. Das macht den Code wesentlich lesbarer.

Lesbarkeit ist einer der wichtigsten Aspekte beim Schreiben eines Tests. So weit wie möglich sollte er ohne Anstrengung lesbar sein, auch von jemandem, der nie zuvor einen Test gesehen hat, und ohne dass er allzu viele Fragen stellen muss – am besten gar keine. Mehr dazu folgt in Kapitel 8. Nun lassen Sie uns sehen, ob man die Tests auch ein wenig knapper und sich weniger wiederholend, aber immer noch lesbar, hinbekommt.

2.5 Refactoring zu parametrisierten Tests

Alle Tests, die Sie bisher geschrieben haben, leiden an einigen Wartbarkeitsproblemen. Stellen Sie sich vor, dass Sie nun einen Parameter zum Konstruktor der Klasse `LogAnalyzer` hinzufügen wollen. Jetzt hätten Sie drei Tests, die sich nicht kompilieren lassen. Daran zu gehen und drei Tests zu reparieren, mag nicht so schlecht klingen, aber es könnten schnell 30 oder 100 werden. In der realen Welt glauben Entwickler, dass sie Besseres zu tun haben, als den Compiler auf etwas zu hetzen, was sie für eine einfache Änderung halten. Wenn Ihre Tests ihren Lauf unterbrechen, wollen Sie sie vielleicht nicht laufen lassen oder neigen sogar dazu, nervige Tests zu entfernen.

Lassen Sie sie uns so umbauen, dass Sie gar nicht erst auf dieses Problem stoßen.

NUnit besitzt ein cooles Feature, das hier ein gutes Stück weiterhelfen kann. Es heißt *parametrisierte Tests*. Um es zu verwenden, nehmen Sie einfach eine der vorhandenen Testmethoden, die genauso aussieht wie einige der anderen, und tun das Folgende:

1. Ersetzen Sie das `[Test]`-Attribut durch das Attribut `[TestCase]`.
2. Extrahieren Sie die fest codierten Werte, die der Test verwendet, in Parameter für die Testmethode.
3. Verschieben Sie die Werte, die Sie zuvor hatten, in die Klammern der `[TestCase(param1, param2, ...)]`-Attribute.
4. Benennen Sie die Testmethode um und geben Sie ihr einen allgemeineren Namen.
5. Fügen Sie dieser Testmethode ein `[TestCase(...)]`-Attribut mit den entsprechenden Werten für jeden Test hinzu, den Sie in dieser Testmethode zusammenführen wollen.
6. Entfernen Sie die anderen Tests, damit nur noch eine Testmethode mit einer Vielzahl von `[TestCase]`-Attributen übrig bleibt.

Lassen Sie uns das Schritt für Schritt durchführen. Nach Schritt 4 wird der letzte Test so aussehen:

```
//das Attribut TestCase sendet einen Parameter an die Methode in der
//nächsten Zeile:
[TestCase("dateimitgültigererweiterung.SLF");]
public void
IsValidLogFileName_ValidExtensions_ReturnsTrue
    (string file) //der Parameter, dem TestCase Attribute einen Wert
                  //zuweisen können
{
    LogAnalyzer analyzer = new LogAnalyzer();

    //der Parameter wird generisch verwendet:
    bool result = analyzer.IsValidLogFileName(file);

    Assert.True(result);
}
```

Der Parameter, der an das `TestCase`-Attribut übergeben wird, wird vom Test-Runner zur Laufzeit auf den ersten Parameter der Testmethode abgebildet. Sie können der Testmethode und dem `TestCase`-Attribut so viele Parameter hinzufügen, wie Sie wollen. Jetzt kommt der Clou: Sie können *mehrere* `TestCase`-Attribute auf die gleiche Testmethode anwenden. Nach Schritt 6 wird der Test so aussehen:

```
[TestCase("dateimitgültigererweiterung.SLF");]
//ein weiteres Attribut bedeutet einen weiteren Test mit einem
//anderen Wert, der dem Parameter der Methode zugewiesen wird:
[TestCase("dateimitgültigererweiterung.slf");]
public void
IsValidLogFileName_ValidExtensions_ReturnsTrue (string file)
```

```
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName(file);

    Assert.True(result);
}
```

Und nun können Sie die vorherige Testmethode, die eine richtige, kleingeschriebene Endung verwendet hat, *löschen*, denn sie ist im `TestCase`-Attribut der aktuellen Testmethode enthalten. Wenn Sie die Tests wieder ausführen, werden Sie sehen, dass Sie immer noch die gleiche Anzahl von Tests haben, aber der Code ist besser wartbar und besser lesbar.

Sie können das noch einen Schritt weiter treiben und auch die negativen Tests (die Asserts, die einen falschen Wert als Endresultat erwarten) in die aktuelle Testmethode einbauen. Ich zeige hier, wie das gemacht wird, aber ich warne Sie, dass das wahrscheinlich zu einer weniger gut lesbaren Testmethode führen wird, denn der Name wird *noch* generischer werden müssen. Betrachten Sie dies einfach als eine Demo der Syntax, wohl wissend, dass damit diese Technik zu weit getrieben wird, denn es führt zu weniger verständlichen Tests.

Hier folgt, wie Sie alle Tests in der Klasse umbauen können – durch Hinzufügen eines weiteren Parameters und durch Änderung des Asserts zu `Assert.AreEqual`:

```
[TestCase("dateimitgültigererweiterung.SLF",true);]
[TestCase("dateimitgültigererweiterung.slf",true);]
//ein weiterer Parameter:
[TestCase("dateimitungültigererweiterung.foo",false);]
public void
IsValidLogFileName_VariousExtensions_ChecksThem (string file,
    bool expected) //erhält den zweiten TestCase Parameter
{
    LogAnalyzer analyzer = new LogAnalyzer();

    bool result = analyzer.IsValidLogFileName(file);

    //verwendet den zweiten Parameter:
    Assert.True(expected, result);
}
```

Mit dieser einen Testmethode können Sie all die anderen Testmethoden in dieser Klasse loswerden, aber beachten Sie, dass der Name des Tests so generisch wird, dass es schwer wird, den Unterschied zwischen gültig und ungültig zu erkennen. Diese Information muss sich selbsterklärend aus den Parameterwerten ergeben, daher sollten Sie sie so einfach wie möglich halten und es sollte so offensichtlich wie möglich sein, dass dies die einfachsten Werte sind, die ihre Aufgabe erfüllen. Wieder einmal mehr zum Ziel der Lesbarkeit in Kapitel 8.

Beachten Sie im Hinblick auf die Wartbarkeit, dass Sie jetzt nur noch einen Aufruf des Konstruktors haben. Das ist besser, aber noch nicht gut genug, denn Sie wollen ja nicht alle Ihre Tests in eine einzige, große, parametrisierte Testmethode stecken. Weitere Techniken zur Wartbarkeit folgen später. (Ja, in Kapitel 8. Sie haben es gewusst.)

Sie können noch ein anderes Refactoring an dieser Stelle durchführen, nämlich wie die `if`-Bedingung im Produktions-Code aussieht. Sie können es zu einem einzigen `return`-Statement reduzieren. Wenn Sie diese Art von Dingen mögen, dann ist es ein guter Zeitpunkt für ein Refactoring. Ich mag's nicht. Ich mag ein wenig Ausführlichkeit und möchte den Leser nicht zu sehr über den Code brüten lassen. Ich mag Code, der nicht um seiner selbst willen versucht, zu smart zu sein, und `return`-Statements, die Konditionen enthalten, gehen mir gegen den Strich. Aber dies ist kein Buch über Design, Sie erinnern sich? Also tun Sie, was Sie wollen. Wegen des »Clean Code« des Buches verweise ich Sie zunächst auf Robert Martin (Uncle Bob).

2.6 Weitere NUnit-Attribute

Nachdem Sie gesehen haben, wie einfach es ist, Unit Tests zu erzeugen, die automatisch ablaufen, werden wir nun einen Blick auf das Setup der Startbedingungen für jeden Test werfen und uns anschauen, wie der Abfall, den Ihre Tests hinterlassen, entsorgt werden kann.

Ein Unit Test weist in seinem Lebenszyklus bestimmte Punkte auf, über die Sie die Kontrolle behalten wollen. Die Ausführung des Tests ist nur einer davon, und es gibt besondere Setup-Methoden, die laufen, bevor jeder einzelne Test läuft, wie Sie im nächsten Abschnitt sehen werden.

2.6.1 Aufbau und Abbau

Für Unit Tests ist es wichtig, dass alle von vorangegangenen Tests übrig gebliebenen Daten oder Instanzen vernichtet werden und dass der Ausgangszustand des neuen Tests so wiederhergestellt wird, als hätte zuvor nie ein Test stattgefunden. Falls ein vorangegangener Test irgendwelche Spuren hinterlässt, kann es sein, dass einer Ihrer Tests fehlschlägt, aber nur wenn er nach diesem bestimmten Test ausgeführt wird, während er bei anderen Gelegenheiten erfolgreich durchläuft. Das Auffinden dieser Art von Abhängigkeits-Bugs ist schwierig und zeitaufwendig und ich empfehle es niemandem. Tests zu haben, die absolut unabhängig voneinander sind, ist eine der besten Methoden, die ich in Teil II dieses Buches behandeln werde.

In NUnit gibt es spezielle Attribute, die eine einfachere Kontrolle über den Aufbau und den Abbau eines Zustands vor und nach einem Test ermöglichen. Dies sind die Aktionsattribute `[SetUp]` und `[TearDown]`.

Abbildung 2.4 zeigt den Prozessablauf eines Tests mit Aufbau- und Abbauaktionen.

Im Augenblick genügt es, wenn jeder Test, den Sie schreiben, eine neue Instanz der jeweils zu testenden Klasse verwendet, damit keine Überbleibsel des vorhergehenden Zustands Ihre Tests verderben.

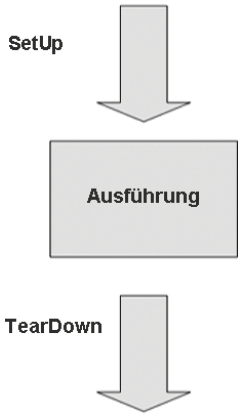


Abb. 2.4: NUnit führt Aufbau- und Abbauaktionen vor und nach jeder einzelnen Testmethode durch.

Das, was in den Schritten Aufbau und Abbau geschieht, können Sie über die Verwendung von zwei NUnit-Attributen bestimmen:

- `[SetUp]` – Dieses Attribut kann ähnlich wie das `[Test]`-Attribut auf eine Methode angewendet werden und es veranlasst NUnit, jedes Mal die Methode auszuführen, wenn ein Test in Ihrer Klasse gestartet wird.
- `[TearDown]` – Dieses Attribut bezeichnet eine Methode, die jedes Mal nach einem Test in Ihrer Klasse ausgeführt wird.

Listing 2.2 zeigt, wie Sie die Attribute `[SetUp]` und `[TearDown]` verwenden können, damit jeder Test eine neue Instanz von `LogAnalyzer` erhält, während Sie gleichzeitig noch ein wenig Schreibarbeit für redundanten Code einsparen können.

Aber bedenken Sie, je öfter Sie `[SetUp]` verwenden, umso weniger lesbar werden Ihre Tests sein, denn man muss Ihren Test-Code an zwei Stellen in der Datei lesen, um zu verstehen, wie der Test seine Instanzen erhält und welchen Typ von jedem Objekt er verwendet. Ich sage meinen Kursteilnehmern: »Malen Sie sich aus, dass die Leser Ihres Tests Sie nie kennengelernt haben und auch nie werden. Sie kommen und lesen Ihre Tests zwei Jahre, nachdem Sie die Firma verlassen haben. Jede Kleinigkeit, die Sie unternehmen, damit sie den Code verstehen können, ohne irgendeine Frage stellen zu müssen, ist eine große Hilfe. Sie haben wahrscheinlich auch niemanden, dem sie diese Fragen stellen könnten, und somit sind Sie ihre einzige Hoffnung.« Ihre Augen dauernd zwischen zwei Regionen des Codes hin und her wandern zu lassen, damit sie Ihren Test verstehen, ist keine gute Idee.

```
using NUnit.Framework;

    [TestFixture] public class LogAnalyzerTests
    {
        private LogAnalyzer m_analyzer=null;

        //ein Setup Attribut:
        [SetUp]
        public void Setup()
```

```
        {
            m_analyzer = new LogAnalyzer();
        }
        [Test]
        public void IsValidFileName_validFileLowerCased_ReturnsTrue()
        {
            bool result =
                m_analyzer.IsValidLogFileName("irgendwas.slf");

            Assert.IsTrue(result,
                "der Dateiname sollte gültig sein!");
        }

        [Test]
        public void IsValidFileName_validFileUpperCased_ReturnsTrue()
        {
            bool result =
                m_analyzer.IsValidLogFileName("irgendwas.SLF");
            Assert.IsTrue(result,
                "der Dateiname sollte gültig sein!");
        }

        //ein Teardown Attribut:
        [TearDown]
        public void TearDown()
        {
        //die folgende Zeile zeigt ein Anti-Muster. Sie wird
        //nicht benötigt. Machen Sie das nicht im wahren Leben.
            m_analyzer = null; //ein verbreitetes Anti-Muster - Sie
                               //müssen das nicht machen
        }
    }
}
```

Listing 2.2: Die Verwendung der Attribute `[SetUp]` und `[TearDown]`

Sie können sich die Aufbau- und Abbaumethoden als Konstruktoren und Destruktoren für die Tests in Ihrer Klasse vorstellen. Sie können nur je eine von beiden Methoden in jeder Testklasse haben und jede wird für jeden einzelnen Test in Ihrer Klasse ausgeführt. In Listing 2.2 haben wir zwei Unit Tests, sodass der Ausführungsablauf so ähnlich wie der in Abbildung 2.5 aussehen wird.

Im wahren Leben verwende ich keine Setup-Methoden, um meine Instanzen zu initialisieren. Ich zeige sie hier, damit Sie wissen, dass sie existieren, und sie vermeiden. Es mag wie eine gute Idee aussehen, aber schnell werden die Tests unterhalb der Setup-Methode schwerer zu lesen. Stattdessen verwende ich Fabrik-Methoden, um meine zu testenden Instanzen zu initialisieren. Lesen Sie dazu in Kapitel 7 nach.

NUnit enthält verschiedene weitere Attribute, die beim Aufbau und Abbau des Zustands helfen. Beispielsweise erlauben `[TestFixtureSetUp]` und `[TestFixtureTearDown]` den einmaligen Aufbau des Zustands vor der Ausführung aller Tests einer spezifischen *Klasse* und den Abbau nach diesen Testläufen. Das ist dann nützlich, wenn der Aufbau oder das Aufräumen lange dauert und Sie es nur einmal pro Lauf durchführen wollen. Sie müssen bei der Verwendung dieser Attribute vorsichtig sein. Andernfalls könnte es sein, dass sich mehrere Tests den Zustand teilen.

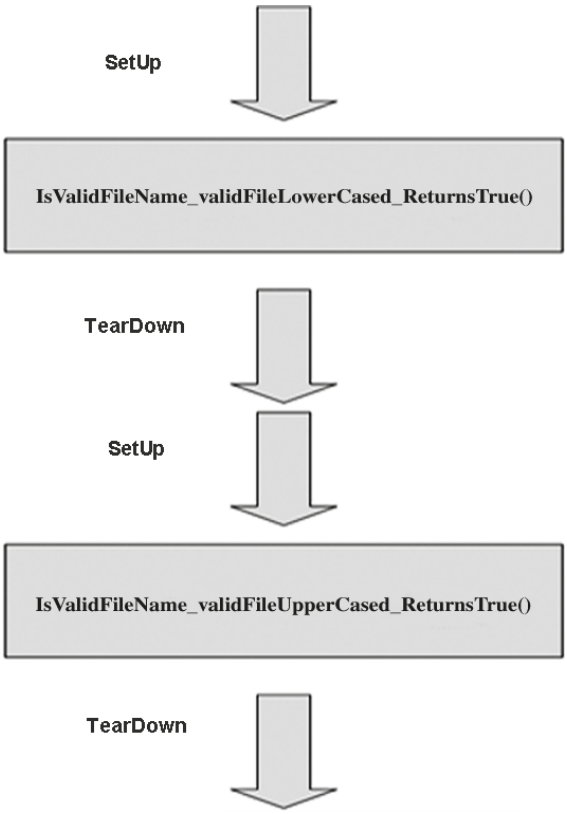


Abb. 2.5: Wie NUnit `[SetUp]` und `[TearDown]` für mehrere Unit Tests in der gleichen Klasse aufruft: Jedem Test geht zunächst die Ausführung von `[SetUp]` voran, gefolgt von einer abschließenden `[TearDown]`-Methode.

Sie verwenden so gut wie *nie, niemals* `TearDown`- oder `TestFixture`-Methoden in Unit-Test-Projekten. Falls doch, schreiben Sie sehr wahrscheinlich einen Integrations-Test, wobei Sie in Kontakt mit dem Dateisystem oder einer Datenbank kommen und die Disk oder die Datenbank nach den Tests aufräumen müssen. Das einzige Mal, das mir bisher untergekommen ist und bei dem es Sinn macht, eine `TearDown`-Methode in Unit Tests zu verwenden, ist der Fall, dass Sie den Zustand einer statischen Variablen oder eines Singletons im Arbeitsspeicher zwischen den Tests »zurücksetzen« müssen. In allen anderen Fällen machen Sie wahrscheinlich Integrations-Tests. Das ist keine schlechte Sache, aber Sie sollten das in einem separaten Projekt tun, das den Integrations-Tests gewidmet ist.

Als Nächstes schauen wir uns an, wie Sie überprüfen können, ob eine Ausnahme von Ihrem Code dann ausgelöst wird, wenn sie ausgelöst werden soll.

2.6.2 Auf erwartete Ausnahmen prüfen

Ein verbreitetes Testszenario ist es, sicherzustellen, dass die richtige Ausnahme dann ausgelöst wird, wenn es notwendig ist.

Lassen Sie uns annehmen, dass Ihre Methode eine Ausnahme `ArgumentException` auslösen soll, wenn Sie einen leeren Dateinamen angeben. Wenn Ihr Code keine Ausnahme auslöst, soll Ihr Test fehlschlagen. Wir werden die Logik der Methode im folgenden Listing testen.

```
public class LogAnalyzer
{
    public bool IsValidLogFileName(string fileName)
    {
        ...
        if(string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException("Dateiname muss angegeben werden!");
        }
        ...
    }
}
```

Listing 2.3: Die Logik der Dateinamenvalidierung in `LogAnalyzer`, die geprüft werden soll

Es gibt zwei Arten, darauf zu prüfen. Lassen Sie uns mit der beginnen, die Sie nicht verwenden sollten, denn sie ist sehr verbreitet und sie ist gewöhnlich die einzige API, um dies zu erreichen. Es gibt ein spezielles Attribut in NUnit, das Ihnen hilft, Ausnahmen zu testen: das Attribut `[ExpectedException]`. Hier kommt ein Beispiel für einen Test, der auf das Auftreten einer Ausnahme prüft:

```
[Test]
[ExpectedException(typeof(ArgumentException),
ExpectedMessage = "Dateiname muss angegeben werden!")]
public void IsValidFileName_EmptyFileName_ThrowsException()
{
    m_analyzer.IsValidLogFileName(string.Empty);
}

private LogAnalyzer MakeAnalyter()
{
    return new LogAnalyzer();
}
```

Es gibt hier einige wichtige Dinge anzumerken:

- Der Nachrichtentext der erwarteten Ausnahme wird dem `[ExpectedException]`-Attribut als Parameter hinzugefügt.
- Es gibt keinen Aufruf von `Assert` im Test selber. Das `[ExpectedException]`-Attribut enthält den Aufruf implizit.
- Es macht keinen Sinn, den booleschen Rückgabewert der Methode abzufragen, da von der Methode erwartet wird, dass sie eine Ausnahme wirft.

Das hat nichts mit diesem Beispiel zu tun, aber ich bin schon vorgeeilt und habe den Code, der eine Instanz von `LogAnalyzer` erzeugt, in eine Fabrik-Methode extrahiert. Ich benutze diese Fabrik-Methode in all meinen Tests, damit die Wartbarkeit des Konstruktors einfacher wird und nur wenige Tests angepasst werden müssen.

Wenn man jetzt den Test auf die Methode in Listing 2.3 anwendet, dann sollte er erfolgreich durchlaufen. Würde Ihre Methode *keine* `ArgumentException`-Ausnahme werfen oder würde sich die Nachricht der Ausnahme von der erwarteten unterscheiden, so wäre der Test fehlgeschlagen – was Ihnen dann sagt, dass entweder die Ausnahme nicht geworfen wurde oder dass die Nachricht nicht die erwartete ist.

Aber warum habe ich erwähnt, dass Sie diese Methode nicht verwenden sollten? Weil dieses Attribut dem Test-Runner im Wesentlichen sagt, dass er die Ausführung der ganzen Methode in einen großen `try-catch`-Block packen und dann fehlschlagen soll, wenn nichts ge-»catch«-ed wird. Das große Problem damit ist, dass Sie nicht wissen, *welche* Zeile die Ausnahme geworfen hat. In der Tat könnten Sie einen Fehler im Konstruktor haben, der eine Ausnahme wirft, und Ihr Test wird erfolgreich durchlaufen, obwohl der Konstruktor niemals diese Ausnahme werfen dürfte! Der Test könnte Sie anlügen, wenn Sie dieses Attribut verwenden, also versuchen Sie, es zu vermeiden.

Stattdessen gibt es eine neuere API in NUnit: `Assert.Catch<T>(delegate)`. Hier kommt der mithilfe von `Assert.Catch` umgeschriebene Test:

```
[Test]
//kein ExpectedException Attribut wird benötigt
public void IsValidFileName_EmptyFileName_Throws()
{
    LogAnalyzer la = MakeAnalyzer();
    //verwende Assert.Catch:
    var ex = Assert.Catch<Exception>(()=>la.IsValidLogFileName(""));

    StringAssert.Contains("Dateiname muss angegeben werden!",
        ex.Message);  //verwendet das Exception-Objekt, das von
                      //Assert.Catch zurückgegeben wird.
}
```

Hier gibt es eine Reihe von Änderungen:

- Sie haben nicht mehr das `[ExpectedException]`-Attribut.
- Sie verwenden `Assert.Catch` und einen Lambda-Ausdruck ohne Argumente, der aus dem Aufruf von `la.IsValidLogFileName("")` besteht.

- Wenn der Code innerhalb des Lambdas eine Ausnahme wirft, wird der Test erfolgreich durchlaufen. Wenn irgendeine Code-Zeile außerhalb des Lambdas eine Ausnahme wirft, wird der Test fehlschlagen.
- `Assert.Catch` ist eine Funktion, die die Instanz des Exception-Objekts zurückgibt, das innerhalb des Lambdas geworfen wurde. Das erlaubt Ihnen später, ein `Assert` auf die Nachricht des Exception-Objekts zu setzen.
- Sie verwenden `StringAssert` – eine Klasse, die Teil des NUnit-Frameworks ist, das Ihnen noch nicht vorgestellt wurde. Es enthält Hilfen, die das Testen mit Strings einfacher und leichter lesbar machen.
- Sie testen nicht auf komplette String-Gleichheit mit `Assert.AreEqual`, sondern verwenden `Assert.Contains`. Die String-Nachricht *enthält* den String, nach dem Sie Ausschau halten. Das macht den Test leichter wartbar, denn Strings sind berüchtigt dafür, sich mit der Zeit zu verändern, wenn neue Features hinzugefügt werden. Strings sind tatsächlich eine Form von UI und können extra Zeilenumbrüche oder extra Informationen, die Sie nicht interessieren, enthalten etc. Wenn Sie ein Assert auf den kompletten String setzen und darauf testen, dass er gleich einem spezifischen, von Ihnen erwarteten String ist, dann müssten Sie den Test jedes Mal anpassen, wenn Sie ein neues Feature an den Anfang oder das Ende der Nachricht setzen, obwohl Sie das in diesem Test gar nicht interessiert (wie etwa zusätzliche Zeilen oder eine andere Formatierung).

Es ist sehr viel weniger wahrscheinlich, dass dieser Test Sie »anlügen« wird, und deshalb empfehle ich die Verwendung von `Assert.Catch` statt `[ExpectedException]`.

Es gibt andere Möglichkeiten, die flüssige Syntax von NUnit zu verwenden, um die Exception-Nachricht zu überprüfen. Ich mag sie nicht sehr, aber das ist eher eine Frage des persönlichen Stils. Mehr über die Syntax von NUnit erfahren Sie auf der Seite `NUnit.com`.

2.6.3 Das Ignorieren von Tests

Manchmal hat man nicht funktionierende Tests und man muss noch den Code im Quellverzeichnis überprüfen. In solchen seltenen Fällen (und sie sollten selten sein!) kann man den Tests das Attribut `[Ignore]` hinzufügen, weil ein Problem im Test existiert und nicht im Quellcode.

Das kann dann so aussehen:

```
[Test]
[Ignore("es gibt ein Problem mit diesem Test")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
/// ...
}
```

Das Ausführen dieses Tests im NUnit-GUI erzeugt ein Resultat wie das in Abbildung 2.6 dargestellte.

Was passiert, wenn Sie die Tests nicht nach ihren Namespaces, sondern nach einem anderen Merkmal gruppiert ausführen wollen? Hier kommen die Testkategorien ins Spiel. Ich werde sie in Abschnitt 2.6.5 erläutern.

2.6.4 Die fließende Syntax von NUnit

NUnit besitzt auch eine alternative, fließendere Syntax, die Sie statt der einfachen `Assert.*`-Methoden verwenden können. Diese Syntax beginnt immer mit `Assert.That(..)`.

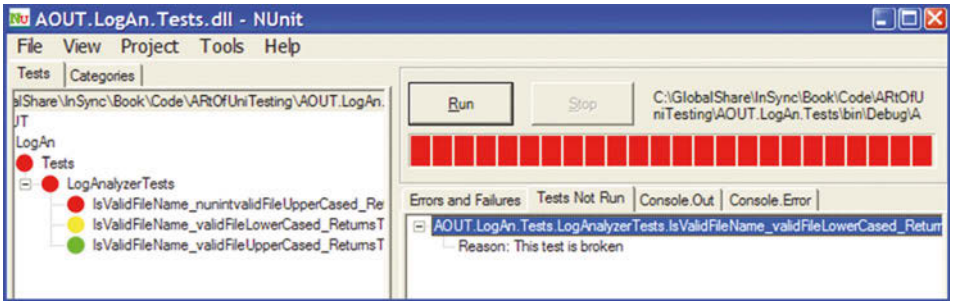


Abb. 2.6: In NUnit wird ein Test, der ignoriert werden soll, mit Gelb markiert (der mittlere Test) und der Grund, warum der Test nicht ausgeführt werden soll, wird rechts auf der Registerkarte TESTS NOT RUN angegeben.

Hier kommt der mit NUnits fließender Syntax umgeschriebene letzte Test:

```
[Test]
public void IsValidFileName_EmptyFileName_ThrowsFluent()
{
    LogAnalyzer la = MakeAnalyzer();
    var ex = Assert.Catch<ArgumentException>(() =>
        la.IsValidLogFileName(""));

    Assert.That(ex.Message,
        IsStringContaining("Dateiname muss angegeben werden!"));
}
```

Ich persönlich mag die knappere, einfachere und kürzere Syntax von `Assert.something()` mehr als `Assert.That()`. Obwohl die fließende Syntax auf den ersten Blick freundlicher erscheint, dauert es länger zu verstehen, worauf Sie testen (die ganze Strecke bis zum Ende der Zeile). Treffen Sie Ihre eigene Wahl, aber stellen Sie sicher, dass sie über das ganze Testprojekt konsistent ist, denn ein Mangel an Konsistenz führt zu einer Reihe von Lesbarkeitsproblemen.

2.6.5 Das Festlegen der Testkategorien

Sie können Ihre Tests so aufbauen, dass sie unter spezifischen Testkategorien ausgeführt werden, wie zum Beispiel langsame Tests und schnelle Tests. Sie erreichen dies über das NUnit-Attribut `[Category]`:

```
[Test]
[Category("Schnelle Tests")]
public void IsValidFileName_ValidFile_ReturnsTrue()
{
/// ...
}
```

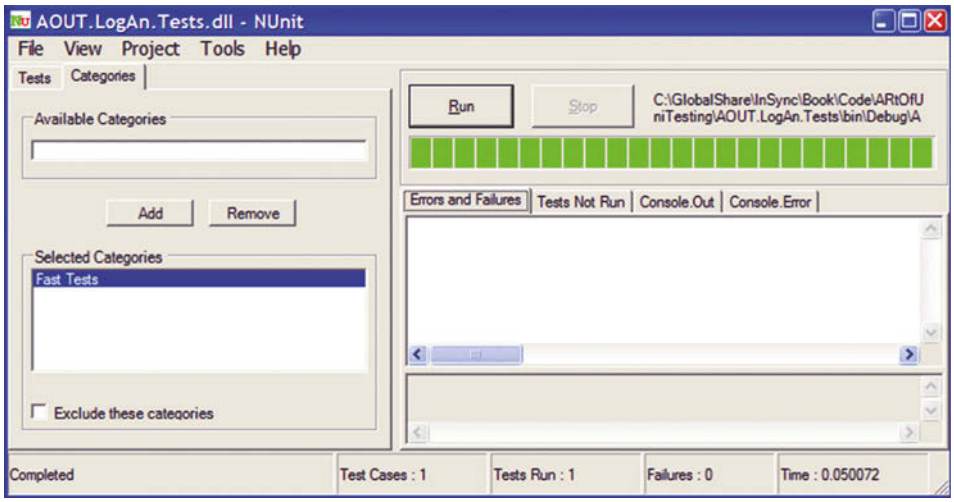


Abb. 2.7: Sie können Testkategorien in der Code-Basis aufsetzen und dann eine bestimmte Kategorie auswählen, um die zugehörigen Tests im GUI von NUnit auszuführen.

Wenn Sie Ihr Test-Assembly in NUnit laden, sehen Sie Ihre Tests nach Kategorien statt nach Namespaces sortiert. Wechseln Sie zur CATEGORIES-Registerkarte in NUnit und doppelklicken Sie auf die Kategorie, die Sie ausführen möchten, woraufhin diese im unteren Fensterbereich SELECTED CATEGORIES erscheint. Dann klicken Sie auf den RUN-Button. Abbildung 2.7 zeigt, wie die Oberfläche aussehen könnte, nachdem Sie die CATEGORIES-Registerkarte ausgewählt haben.

Bisher haben Sie einfache Tests mit Methoden ausgeführt, die einen Wert als Resultat zurückgeben. Was aber ist, wenn Ihre Methode keinen Wert zurückgibt, sondern den Zustand eines Objekts ändert?

2.7 Das Testen auf Zustandsänderungen des Systems statt auf Rückgabewerte

Bis zu diesem Abschnitt haben Sie gesehen, wie auf die einfachste Art von Resultat, das eine Unit of Work haben kann, getestet wird: auf Rückgabewerte (siehe Kapitel 1). In diesem und im nächsten Kapitel werden wir die zweite Art von Resultat diskutieren: Zustandsänderungen des Systems – die Überprüfung, dass sich das Verhalten des Systems nach der Ausführung einer Aktion auf das zu testende System geändert hat.

Definition

State-Based Testing (»Zustandsbezogenes Testen«, manchmal auch als »State-Verification« bezeichnet) prüft, ob die ausgeführte Methode korrekt arbeitet, indem das geänderte Verhalten des Testsystems und der damit verknüpften Komponenten nach Ausführung der Methode untersucht wird.

Wenn sich das System nachher genauso verhält wie zuvor, dann haben Sie entweder seinen Zustand nicht geändert oder da ist irgendwo ein Bug.

Wenn Sie bereits woanders Definitionen des State-Based Testings gelesen haben, dann wird Ihnen auffallen, dass ich es etwas anders definiere. Das liegt daran, dass ich das Ganze in einem etwas anderen Licht betrachte – dem der Wartbarkeit von Tests. Das einfache Testen des unmittelbaren Zustands (manchmal muss er ausgelagert werden, um ihn überhaupt testen zu können) ist etwas, das ich gewöhnlich nicht empfehlen würde, denn es führt zu weniger wartbarem und weniger lesbarem Code.

Lassen Sie uns ein einfaches State-Based-Testing-Beispiel betrachten, das die Klasse `LogAnalyzer` verwendet, die Sie nicht einfach durch den Aufruf einer Methode testen können. Listing 2.4 zeigt den Code für diese Klasse. In diesem Fall führen Sie eine neue Property ein, `WasLastFileNameValid`, in der der letzte Zustand der Methode `IsValidLogFileName` gespeichert wird. Ich zeige Ihnen den Code zuerst, denn ich will Ihnen an dieser Stelle nicht TDD beibringen, sondern wie gute Tests geschrieben werden. Tests *können* durch TDD besser werden, aber diesen Schritt gehen Sie erst, wenn Sie wissen, wie man die Tests *nach* dem Code schreibt.

```
public class LogAnalyzer
{
    public bool WasLastFileNameValid {get; set;}

    public bool IsValidLogFileName(string fileName)
    {
        //ändert den Systemzustand:
        WasLastFileNameValid = false;
        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException("Dateiname muss angegeben werden!"));
        }
        if (!fileName.EndsWith(".SLF"),
            StringComparison.CurrentCultureIgnoreCase))
        {
            return false;
        }

        //ändert den Systemzustand:
        WasLastFileNameValid = true;
        return true;
```

```
    }
}
```

Listing 2.4: Der Wert der Property wird über `IsValidLogFileName` getestet.

Wie Sie im Code sehen, speichert `LogAnalyzer` das Ergebnis der letzten Validierung des Dateinamens. Weil `WasLastFileNameValid` davon abhängt, dass eine andere Methode zuerst aufgerufen wird, können Sie für diese Funktionalität nicht einfach einen Test schreiben, der den Rückgabewert einer Methode prüft; Sie müssen Alternativen zum Testen der Logik finden.

Als Erstes müssen Sie die Unit of Work identifizieren, die Sie testen wollen. Ist sie in der neuen Property namens `WasLastFileNameValid`? Teilweise; sie liegt ebenso in der Methode `IsValidLogFileName`, weshalb Ihr Test mit dem Namen dieser Methode beginnen sollte, denn dies ist die Unit of Work, die Sie öffentlich aufrufen, um den Zustand des Systems zu ändern. Das folgende Listing zeigt einen einfachen Test, um zu prüfen, ob das Ergebnis gespeichert wird.

```
[Test]
public void IsValidFileName_WhenCalled_ChangesWasLastFileNameValid()
{
    LogAnalyzer la = MakeAnalyzer();

    la.IsValidLogFileName("ungültigerName.foo");

    //wende Assert auf den Systemzustand an:
    Assert.False(la.WasLastFileNameValid);
}
```

Listing 2.5: Eine Klasse wird getestet, indem eine Methode aufgerufen und der Wert einer Property geprüft wird.

Beachten Sie, dass wir die Funktionalität der Methode `IsValidLogFileName` testen, indem wir das `Assert` auf ein Stück Code anwenden, das an einer anderen Stelle liegt als die Methode, die wir testen.

Hier das Beispiel nach einem Refactoring, das einen weiteren Test für die gegenteilige Annahme zum Systemzustand hinzufügt:

```
[TestCase("ungültigeDatei.foo", false)]
[TestCase("gültigeDatei.slf", true)]
public void IsValidFileName_WhenCalled_ChangesWasLastFileNameValid
    (string file, bool expected)
{
    LogAnalyzer la = MakeAnalyzer();

    la.IsValidLogFileName(file);

    Assert.AreEqual(expected, la.WasLastFileNameValid);
}
```

Das nächste Listing zeigt ein weiteres Beispiel. Es wirft einen Blick auf die Funktionalität einer Rechenmaschine mit Gedächtnis.

```
public class MemCalculator
{
    private int sum=0;

    public void Add(int number)
    {
        sum+=number;
    }

    public int Sum()
    {
        int temp = sum;
        sum = 0;
        return temp;
    }
}
```

Listing 2.6: Die Methoden `Add()` und `Sum()`

Die Klasse `MemCalculator` arbeitet so ähnlich wie der Taschenrechner, den Sie kennen und lieben. Sie können eine Zahl eingeben, dann auf *Addieren* klicken, eine andere Zahl eingeben, wieder auf *Addieren* klicken und so weiter. Wenn Sie fertig sind, können Sie auf das Gleichheitszeichen klicken und Sie erhalten die Summe.

Wo starten Sie mit dem Test der Funktion `Sum()`? Sie sollten immer versuchen, mit dem einfachsten Test zu beginnen, so wie dem, der die Voreinstellung für die Rückgabe von `Sum()` auf den Wert 0 prüft. Dies wird im folgenden Listing gezeigt.

```
[Test]
public void Sum_ByDefault_ReturnsZero()
{
    MemCalculator calc = new MemCalculator();

    int lastSum = calc.Sum();

    //überprüfe auf den Standard-Rückgabewert:
    Assert.AreEqual(0,lastSum);
}
```

Listing 2.7: Der einfachste Test für die Funktion `Sum()` von `Calculator`

Beachten Sie hier auch die Bedeutung des Methodennamens. Sie können ihn lesen wie einen Satz.

Hier kommt eine einfache Liste von Namenskonventionen für Szenarien, die ich in solchen Fällen gerne verwende:

- `ByDefault` kann verwendet werden, wenn es einen erwarteten Rückgabewert ohne vorhergehende Aktion gibt, wie in dem eben gezeigten Beispiel.
- `WhenCalled` oder `Always` kann bei der zweiten oder dritten Art von Unit-of-Work-Resultaten (Zustandsänderungen oder Third-Party-Aufrufen) verwendet werden, wenn eine Zustandsänderung oder ein Third-Party-Aufruf ohne vorherige Konfiguration ausgeführt wird: beispielsweise `Sum_WhenCalled_CallsTheLogger` oder `Sum_Always_CallsTheLogger`.

Sie können keinen weiteren Test schreiben, ohne zunächst die Methode `Add()` aufzurufen, weshalb Ihr nächster Test zuerst `Add()` aufruft und dann `Assert` auf den von `Sum()` zurückgegebenen Wert anwendet. Listing 2.8 zeigt unsere Testklasse mit diesem neuen Test.

```
[Test]
public void Sum_ByDefault_ReturnsZero()
{
    MemCalculator calc = MakeCalc();

    int lastSum = calc.Sum();

    Assert.AreEqual(0,lastSum);
}

[Test]
public void Add_WhenCalled_ChangeSum()
{
    MemCalculator calc = MakeCalc();

    calc.Add(1);
    int lastSum = calc.Sum();

    //Das Verhalten und der Zustand des Systems ändern sich,
    //wenn Sum in diesem Test eine andere Zahl zurückgibt:
    Assert.AreEqual(1,lastSum);
}

private static MemCalculator MakeCalc()
{
    return new MemCalculator();
}
```

Listing 2.8: Die zwei Tests, wobei der zweite die Methode `Add()` aufruft

Beachten Sie, dass Sie diesmal eine Fabrik-Methode benutzen, um `MemCalculator` zu initialisieren. Das ist eine gute Idee, denn es spart Zeit beim Schreiben der Tests, macht den Code innerhalb jedes Tests kleiner und ein wenig lesbarer und stellt sicher, dass `MemCalcu-`

lator immer auf die gleiche Weise initialisiert wird. Das ist auch für die Wartbarkeit des Tests besser, denn wenn sich der Konstruktor von `MemCalculator` ändert, braucht man die Initialisierung nur an einer Stelle zu ändern, anstatt in jedem Test den `new`-Aufruf anpassen zu müssen.

So weit, so gut. Aber was geschieht, wenn die Methode, die Sie testen, von externen Ressourcen abhängt, wie etwa dem Dateisystem, einer Datenbank, einem Webservice oder irgendetwas anderem, das für Sie schwierig zu kontrollieren ist? Und wie testen Sie die dritte Art von Ergebnissen für eine Unit of Work – einen Third-Party-Aufruf? Das ist der Augenblick, in dem Sie beginnen, Test-Stubs, Fake-Objekte und Mock-Objekte zu erzeugen, die in den nächsten Kapiteln vorgestellt werden.

2.8 Zusammenfassung

In diesem Kapitel haben wir uns NUnit angeschaut, um einfache Tests für einfachen Code zu schreiben. Sie haben die Attribute `[TestCase]`, `[SetUp]` und `[TearDown]` verwendet, damit sich Ihre Tests immer in einem neuen und sauberen Zustand befinden. Sie haben Fabrik-Methoden verwendet, um das Ganze wartbarer zu machen. Sie haben `[Ignore]` verwendet, um Tests, die noch repariert werden müssen, zu überspringen. Testkategorien können Ihnen dabei helfen, die Tests nach logischen Kriterien statt nach den Klassen oder Namespaces zu gruppieren, und `Assert.Catch()` hilft Ihnen, dass Ihr Code dann Ausnahmen wirft, wenn er es auch soll. Wir haben uns auch angeschaut, was passiert, wenn Sie keiner einfachen Methode mit einem Rückgabewert begegnen und Sie den Zustand eines Objekts testen müssen.

Aber das ist nicht genug. Der meiste Testcode muss mit viel schwierigeren Situationen umgehen. Die nächsten Kapitel werden Ihnen weitere grundlegende Werkzeuge zum Schreiben von Unit Tests an die Hand geben. Sie werden aus diesen Werkzeugen auswählen müssen, wenn Sie Tests für die unterschiedlich schwierigeren Szenarien, denen Sie begegnen, schreiben wollen.

Behalten Sie Folgendes im Hinterkopf:

- Es ist eine verbreitete Praxis, eine Testklasse pro zu testender Klasse, ein Unit-Test-Projekt pro zu testendem Projekt (abgesehen von einem Integrations-Test-Projekt für das zu testende Projekt) und zumindest eine Testmethode pro Unit of Work (die so klein wie eine Methode oder so groß wie mehrere Klassen sein kann) zu haben.
- Benennen Sie Ihre Tests eindeutig nach dem folgenden Modell: `[UnitOfWork]_[Szenario]_[ErwartetesVerhalten]`.
- Verwenden Sie Fabrik-Methoden, um den Code in Ihren Tests wiederverwenden zu können, beispielsweise um Objekte, die alle Ihre Tests benötigen, zu erzeugen und zu initialisieren.
- Verwenden Sie die Attribute `[SetUp]` und `[TearDown]` nicht, wenn Sie es vermeiden können. Sie machen die Tests weniger verständlich.

Im nächsten Kapitel werden wir uns praxisnähere Szenarien anschauen, wo der zu testende Code etwas realistischer ist als der, den Sie bis hierher gesehen haben. Er enthält Abhängigkeiten und hat Probleme hinsichtlich seiner Testbarkeit. Wir beginnen mit einer Diskussion der Idee des Integrationstests im Unterschied zum Unit Test und erörtern, was das für uns als Entwickler, die Tests schreiben und die Qualität ihres Codes sicherstellen wollen, bedeutet.

Teil II

Zentrale Methoden

Nachdem ich in den vorangegangenen Kapiteln die Grundlagen behandelt habe, werde ich nun die zentralen Test- und Refactoring-Methoden einführen, die notwendig sind, um praxistaugliche Tests zu schreiben.

In Kapitel 3 beginne ich mit den Stubs (»Stummel«, »Stumpf«) und Sie lernen, wie sie dabei helfen, Abhängigkeiten zu durchbrechen. Anschließend gehen wir zu den Refactoring-Methoden über, die den Code leichter testbar machen, und Sie lernen etwas über die Seams (»Nähte«, »Nahtstellen«) innerhalb des Prozesses.

In Kapitel 4 geht es weiter mit den Mock-Objekten (»Attrappen«) und dem Interaction Testing und wir werfen einen Blick darauf, wie sich Mock-Objekte von Stubs unterscheiden, und ich werde auf das Konzept der Fake-Objekte eingehen.

In Kapitel 5 werden Sie sich Isolation-Frameworks (auch als »Mocking-Frameworks« bekannt) anschauen und ergründen, wie sie das sich wiederholende Codieren in handgeschriebenen Mock-Objekten und Stubs vermeiden helfen. Kapitel 6 vergleicht ebenfalls die wichtigsten Isolation-Frameworks für .NET und verwendet für einige Beispiele FakeItEasy, um die API für verschiedene verbreitete Use Cases zu demonstrieren.

In diesem Teil:

Kapitel 3

Die Verwendung von Stubs, um Abhängigkeiten aufzulösen

Dieses Kapitel behandelt

- die Definition von Stubs
- das Code-Refactoring, um Stubs zu verwenden
- Einkapselungsprobleme im Code
- die besten Methoden beim Einsatz von Stubs

Im vorhergehenden Kapitel haben Sie Ihren ersten Unit Test mit NUnit geschrieben und verschiedene Testattribute ausprobiert. Sie haben auch Tests für einfache Use Cases erstellt, wobei Sie nur Rückgabewerte von Objekten oder den Zustand der zu testenden Unit in einem Minimal-System prüfen mussten.

In diesem Kapitel werden wir einen Blick auf wesentlich realistischere Beispiele werfen, wo das zu testende Objekt von einem anderen Objekt abhängt, über das Sie keine Kontrolle haben (oder das noch nicht funktioniert). Dieses Objekt könnte ein Webservice sein, die Tageszeit, ein anderer Thread oder sonst irgendetwas. Der entscheidende Punkt ist, dass Ihr Test nicht kontrollieren kann, was während des Tests aus diesem Abhängigkeitsverhältnis an Ihren Code zurückgegeben wird oder wie das Verhalten ist (wenn Sie beispielsweise eine Ausnahme simulieren wollen). Hier kommen die »Stubs« ins Spiel.

3.1 Die Stubs werden vorgestellt

Menschen in das Weltall zu fliegen, stellt Ingenieure und Astronauten vor interessante Herausforderungen. Eine der schwierigeren besteht darin, sicherzustellen, dass der Astronaut für den Raumflug bereit ist und die ganze Technik auch im Orbit bedienen kann. Ein kompletter Integrationstest für das Space Shuttle hätte einen Test im All erfordert und das ist offensichtlich keine sichere Art, die Astronauten zu testen. Darum hat die NASA umfassende Simulatoren, die die Umgebung im und um das Space Shuttle nachahmen und die externe Abhängigkeit beseitigen, sich im Weltall befinden zu müssen.

Definition

Eine *externe Abhängigkeit* ist ein Objekt in unserem System, mit dem der zu testende Code interagiert und über das wir keine Kontrolle haben. (Typische Beispiele sind das Dateisystem, Threads, Hauptspeicher, Zeit usw.)

Diese externen Abhängigkeiten in unserem Code zu beherrschen, ist das Thema dieses Kapitels und des größten Teils dieses Buches. In der Programmierung verwendet man Stubs, um das Problem der externen Abhängigkeiten zu umschiffen.

Definition

Ein *Stub* ist ein kontrollierbarer Ersatz für eine vorhandene Abhängigkeit (ein *Collaborator*) im System. Durch die Verwendung eines Stubs kann der Code getestet werden, ohne die Abhängigkeit direkt handhaben zu müssen.

In Kapitel 4 werden wir eine erweiterte Definition von Stubs, Mocks und Fakes, und wie diese zusammenhängen, haben. Im Augenblick genügt es zu wissen, dass der wesentliche Unterschied zwischen Mocks und Stubs ist, dass sie ein Assert nur auf Mock-Objekte anwenden, aber *nie* auf Stubs. Ansonsten verhalten sich Mocks ganz ähnlich wie Stubs.

Lassen Sie uns einen Blick auf ein reales Beispiel werfen und die Dinge für unsere `LogAnalyzer`-Klasse, die wir im letzten Kapitel eingeführt haben, ein wenig komplizierter machen. Wir werden versuchen, eine Abhängigkeit vom Dateisystem zu entwirren.

Testmusternamen

xUnit Test Patterns: Refactoring Test Code von Gerard Meszaros (Addison-Wesley, 2007) ist ein klassisches Muster-Referenzbuch zum Unit Testing. Es definiert Muster für Dinge, die Sie in Ihren Tests auf mindestens fünf unterschiedliche Arten imitieren, wobei ich glaube, dass das eher verwirrt (obwohl es detailliert ist). In diesem Buch wähle ich nur drei Definitionen zum Imitieren von Dingen in Tests: Fakes, Stubs und Mocks. Ich glaube, dass es diese Vereinfachung der Begriffe für den Leser leichter macht, die Muster zu verdauen, und es gibt keine Notwendigkeit, mehr als diese drei zu kennen, um loszulegen und großartige Tests zu schreiben. An verschiedenen Stellen im Buch werde ich jedoch auf die Namen der Muster verweisen, die in *xUnit Test Patterns* verwendet werden, sodass Sie Meszaros' Definition dort einfach nachschlagen können, wenn Sie möchten.

3.2 Die Identifizierung einer Dateisystemabhängigkeit in LogAn

Die Anwendung, zu der unsere Klasse `LogAn` gehört, kann so konfiguriert werden, dass sie mit einer Vielzahl von Dateinamenserweiterungen zurechtkommt, indem sie jeweils einen speziellen Adapter benutzt. Der Einfachheit halber lassen Sie uns annehmen, dass die gültigen Dateinamen irgendwo auf der Festplatte in einer Konfigurationsdatei für die Applikation gespeichert sind und dass die Methode `IsValidLogFileName` folgendermaßen aussieht:

```
public bool IsValidLogFileName (string fileName)
{
    // lese die Konfigurationsdatei
    // gebe true zurück, wenn die Erweiterung unterstützt wird.
}
```

Damit taucht das Problem auf, dass Sie, sobald dieser Test, wie in Abbildung 3.1 dargestellt, vom Dateisystem abhängt, einen Integrationstest durchführen, mit all den damit verbundenen Problemen: Integrationstests sind langsamer durchzuführen, sie benötigen eine Konfiguration, sie testen mehrere Dinge gleichzeitig usw.

Dies ist die Essenz von »testhemmendem« Design: Der Code hat irgendeine Abhängigkeit von einer externen Ressource, die den Test stören kann, auch wenn die Logik im Code vollkommen korrekt ist. In Legacy-Systemen kann eine einzelne Unit of Work (*Aktion* im System) viele Abhängigkeiten von externen Ressourcen haben, über die Ihr Testcode wenig oder gar keine Kontrolle hat. Kapitel 10 geht näher auf dieses Thema ein.

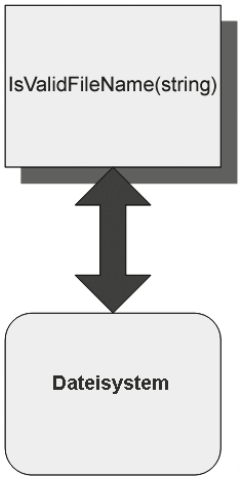


Abb. 3.1: Ihre Methode hat eine direkte Abhängigkeit vom Dateisystem. Das Design des Objektmodells im Test verhindert, dass Sie einen Unit Test durchführen – es schreit nach einem Integrationstest.

3.3 Die Entscheidung, wie LogAnalyzer am einfachsten getestet werden kann

»Es gibt kein objektorientiertes Problem, das nicht durch die Einführung einer zusätzlichen Indirektionsschicht gelöst werden kann, außer natürlich zu viele Indirektionsschichten.« Ich mag dieses Zitat (aus `http://en.wikipedia.org/wiki/Abstraction_layer`), denn eine Menge der »Kunst« in der Kunst des Unit Testings liegt im Finden der richtigen Stelle, um eine Indirektionsschicht hinzuzufügen oder zu verwenden und damit den Code zu testen.

Sie können irgendetwas nicht testen? Fügen Sie eine Schicht hinzu, die alle Aufrufe zu diesem Etwas einpackt, und lassen Sie dann den Layer das gewünschte Verhalten in Ihren Tests nachahmen. Oder machen Sie dieses Etwas ersetzbar (sodass es selber eine Indirektionsschicht ist). Die Kunst besteht auch darin, herauszufinden, wann eine Indirektionsschicht bereits vorhanden ist, statt sie neu erfinden zu müssen, oder zu wissen, wann man sie besser nicht benutzt, weil es die Dinge zu kompliziert macht. Aber lassen Sie uns einen Schritt nach dem anderen tun.

Die einzige Möglichkeit, einen Test für den Code, so wie er ist, zu schreiben, besteht in einer Konfigurationsdatei im Dateisystem. Da Sie versuchen, diese Art von Abhängigkeiten zu vermeiden, wollen Sie, dass Ihr Code einfach zu testen ist, ohne dass Sie Zuflucht zum Integration Testing nehmen müssen.

Wenn Sie auf die Analogie zum Astronauten schauen, mit der wir begonnen haben, so können Sie ein eindeutiges Muster zum Durchbrechen der Abhängigkeit erkennen:

1. Finde das *Interface* oder die *API*, mit der das zu testende Objekt zusammenarbeitet. Im Astronauten-Beispiel waren dies die Joysticks und Monitore eines Space Shuttles, wie in Abbildung 3.2 gezeigt.
2. Ersetze die dahinter liegende Implementierung dieses Interface mit etwas, über das wir die Kontrolle haben. Das schließt das Verkabeln der verschiedenen Shuttle-Monitore, der Joysticks und der Knöpfe sowie das Verbinden mit einem Kontrollraum, wo Testingenieure kontrollieren können, was das Space-Shuttle-*Interface* den Astronauten im Test zeigt, ein.

Das Übertragen dieses Musters auf Ihren Code erfordert weitere Schritte:

1. Finde das *Interface*, mit dem der Anfang der zu testenden Unit of Work zusammenarbeitet. (In diesem Fall wird der Begriff »Interface« nicht im rein objektorientierten Sinn benutzt; er bezieht sich auf eine bestimmte Methode oder Klasse, mit der zusammengearbeitet wird.) In unserem LogAn-Projekt ist das die Konfigurationsdatei im Dateisystem.

Abb. 3.2: Ein Space-Shuttle-Simulator hat realistische Joysticks und Bildschirme, um die Außenwelt zu simulieren. (Foto mit freundlicher Genehmigung der NASA)

2. Wenn das Interface mit Ihrer Unit of Work im Test nicht *direkt verbunden* ist (wie in diesem Fall – Sie rufen direkt das Dateisystem auf), dann mache den Code durch das Hinzufügen einer Indirektionsschicht, die das Interface verdeckt, testbar. In unserem Beispiel wäre es eine Möglichkeit, ein Indirektions-Level hinzuzufügen, indem der direkte Aufruf des Dateisystems in eine eigene Klasse (wie etwa `FileExtensionManager`) verschoben wird. Wir werden uns auch andere anschauen. (Abbildung 3.3 zeigt, wie das Design nach diesem Schritt aussehen könnte.)
3. Ersetze die *darunter liegende Implementierung* der interaktiven Schnittstelle mit etwas, über das wir die Kontrolle haben. In diesem Fall ersetzen Sie die Instanz der Klasse, die Ihre Methode aufruft (`FileExtensionManager`), mit einer Stub-Klasse, über die Sie die Kontrolle haben (`StubExtensionManager`), sodass Sie Ihrem Test die Kontrolle über die externen Abhängigkeiten geben.

Ihre Ersatzinstanz wird mit dem Dateisystem *nicht* kommunizieren, was die Abhängigkeit vom Dateisystem auflöst. Weil Sie die Klasse, die mit dem Dateisystem kommuniziert, nicht testen, aber den Code, der diese Klasse aufruft, ist es in Ordnung, wenn die Stub-Klasse nichts anderes macht, außer Däumchen zu drehen, während der Test läuft. Abbildung 3.4 zeigt das Design nach dieser Änderung.

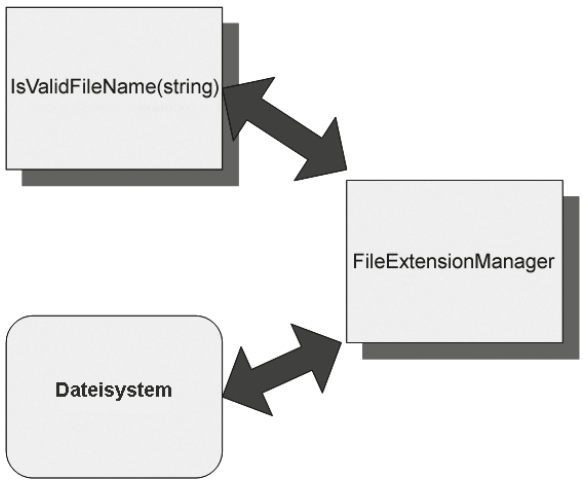


Abb. 3.3: Das Einführen einer Indirektionsschicht, um eine direkte Abhängigkeit vom Dateisystem zu vermeiden. Der Code, der das Dateisystem aufruft, wurde in eine eigene Klasse `FileExtensionManager` verlagert, die später in Ihrem Test durch einen Stub ersetzt wird.

In Abbildung 3.4 habe ich ein neues C#-Interface hinzugefügt. Dieses neue Interface erlaubt es dem Objektmodell, die Operationen, die die Klasse `FileExtensionManager` durchführt, zu verbergen, und es erlaubt dem Test, einen Stub einzufügen, der wie ein `FileExtensionManager` aussieht. Im nächsten Abschnitt werden Sie mehr über diese Methode erfahren.

Wir haben uns eine Möglichkeit angeschaut, die Testbarkeit Ihrer Code-Basis zu verbessern – durch das Hinzufügen eines neuen Interface. Lassen Sie uns nun einen Blick auf die Idee des Code-*Refactorings* werfen und *Seams* (Nahtstellen) in Ihren Code einführen.

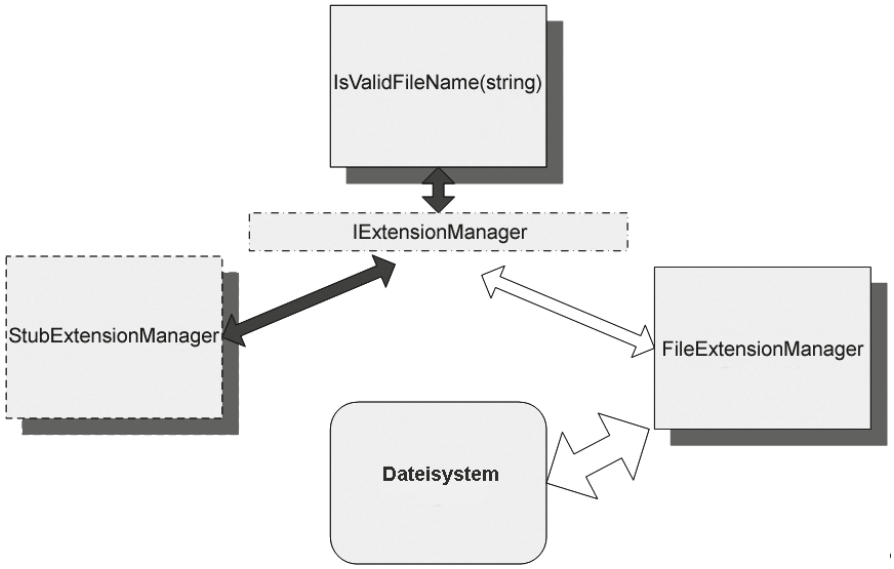


Abb. 3.4: Die Einführung eines Stubs, um die Abhängigkeit aufzulösen. Nun sollte Ihre Klasse nicht wissen oder sich darum kümmern, mit welcher Implementierung eines Extension Managers sie zusammenarbeitet.

3.4 Design-Refactoring zur Verbesserung der Testbarkeit

Es ist an der Zeit, zwei neue Begriffe einzuführen, die dieses Buch hindurch benutzt werden: *Refactoring* und *Seams*.

Definition

Refactoring ist der Vorgang der Änderung des Codes, ohne die Funktionalität des Codes zu ändern. Das bedeutet, er macht genau den gleichen Job wie zuvor. Nicht mehr und nicht weniger. Er sieht nur anders aus. Ein Refactoring-Beispiel könnte eine Methode umbenennen und eine lange Methode in mehrere kleinere aufbrechen.

Definition

Seams sind Stellen in Ihrem Code, in die Sie eine andere Funktionalität einfügen können, wie etwa Stub-Klassen. Sie könnten auch einen Konstruktor-Parameter hinzufügen, eine Property mit öffentlichem `set`, Sie könnten eine Methode virtuell machen, damit sie überschrieben werden kann, oder einen Delegaten als Parameter oder Property von außen verfügbar machen, damit von außerhalb der Klasse auf ihn zugegriffen werden kann. Seams sind das, was Sie bekommen, wenn Sie das Open-Closed-Prinzip implementieren, wo die Funktionalität einer Klasse offen ist für eine Verbesserung, aber der Quellcode abgeschlossen ist gegen eine direkte Modifikation. (Mehr zum Thema Seams erfahren Sie im Buch *Effektives Arbeiten mit Legacy Code* von Michael Feathers oder zum Open-Closed-Prinzip in *Clean Code* von Robert Martin.)

Sie können ein Refactoring durchführen, indem Sie einen neuen Seam in den Code einbauen, ohne die originale Funktionalität des Codes zu ändern. Das ist genau das, was ich mit der Einführung des `IExtensionManager`-Interface getan habe.

Und Sie werden Refactoring durchführen.

Aber zunächst möchte ich Sie daran erinnern, dass das Refactoring von Code ohne irgendeine Art von automatisierten Tests für Ihren Code (Integrationstests oder andere) Sie auf direktem Weg in eine Sackgasse und das Ende Ihrer Karriere führen kann, wenn Sie nicht aufpassen. Sie sollten immer irgendeine Art von Integrationstests haben, die Ihnen den Rücken frei halten, bevor Sie anfangen, an existierendem Code herumzubasteln, oder zumindest einen »Fluchtplan« – eine Kopie des Codes vor dem Beginn des Refactorings, am besten in Ihrem Source Control und mit einem netten, fetten Kommentar »vor dem Beginn des Refactorings«, den Sie später leicht wiederfinden können. In diesem Kapitel nehme ich an, dass Sie bereits einige dieser Integrations-Tests haben und dass Sie sie nach jedem Refactoring laufen lassen, um zu sehen, ob der Code immer noch erfolgreich durchläuft. Aber wir konzentrieren uns nicht darauf, denn dieses Buch handelt vom Unit Testing.

Wenn Sie die Abhängigkeit zwischen Ihrem zu testenden Code und dem Dateisystem auflösen wollen, können Sie einen oder mehrere *Seams* in Ihren Code einfügen. Sie müssen nur sicherstellen, dass der resultierende Code *genau* das Gleiche macht wie zuvor. Es gibt zwei Arten von Techniken zum Auflösen von Abhängigkeiten, und die eine hängt von der anderen ab. Ich nenne sie Typ-A- und Typ-B-Refactoring:

- *Typ A* – das Abstrahieren konkreter Objekte in *Interfaces* oder *Delegaten*.
- *Typ B* – das Refactoring, um die Injektion von Fake-Implementierungen dieser Delegaten oder Interfaces zu ermöglichen.

In der folgenden Liste ist nur der erste Eintrag ein Typ-A-Refactoring. Der Rest sind Typ-B-Refactorings:

- *Typ A* – Extrahiere ein Interface, um die dahinter liegende Implementierung durch eine andere ersetzen zu können.
- *Typ B* – Injiziere eine Stub-Implementierung in die zu testende Klasse.
- *Typ B* – Injiziere einen Fake auf Konstruktor-Ebene.
- *Typ B* – Injiziere ein Fake als `set` oder `get` einer Property.
- *Typ B* – Injiziere ein Fake unmittelbar vor einem Methodenaufruf.

Jede dieser Techniken werden wir uns nun im Einzelnen anschauen.

3.4.1 Extrahiere ein Interface, um die dahinter liegende Implementierung durch eine andere ersetzen zu können

Bei dieser Technik müssen Sie den Code, der das Dateisystem betrifft, herausbrechen und in eine separate Klasse auslagern. Auf diese Weise können Sie ihn leicht unterscheiden und später den Aufruf dieser Klasse von Ihren Tests aus ersetzen (wie in Abbildung 3.3 gezeigt wurde). Dieses erste Listing zeigt die Stellen, an denen Sie den Code ändern müssen.

```
public bool IsValidLogFileName(string fileName)
{
    FileExtensionManager mgr = new FileExtensionManager();
```

```
    //verwende die extrahierte Klasse:
    return mgr.IsValid(fileName);
}

//definiere die extrahierte Klasse:
class FileExtensionManager
{
    public bool IsValid(string fileName)
    {
    //lese hier eine Datei
    }
}
```

Listing 3.1: Sie extrahieren eine Klasse, die auf das Dateisystem zugreift, und rufen sie auf.

Als Nächstes können Sie Ihrer zu testenden Klasse beibringen, dass sie statt der konkreten Klasse `FileExtensionManager` irgendeine Art von `ExtensionManager` verwenden soll, ohne die konkrete Implementierung zu kennen. In .NET kann dies erreicht werden, indem entweder eine Basisklasse oder ein Interface angelegt und dann die Klasse `FileExtensionManager` davon abgeleitet wird.

Das nächste Listing zeigt die Verwendung eines neuen Interface in Ihrem Design, um es besser testen zu können. Abbildung 3.4 zeigt ein Diagramm dieser Implementierung.

```
//implementiere das Interface
public class FileExtensionManager : IExtensionManager
{
    public bool IsValid(string fileName)
    {
    ...
    }
}

//definiere das neue Interface
public interface IExtensionManager
{
    bool IsValid (string fileName);
}

//die zu testende Unit of Work:
public bool IsValidLogFileName(string fileName)
{
    //definiere eine Variable als den Typ des Interfaces:
    IExtensionManager mgr = new FileExtensionManager();
```

```
    return mgr.IsValid(fileName);
}
```

Listing 3.2: Sie extrahieren ein Interface einer bekannten Klasse.

Sie erzeugen einfach ein Interface mit einer Methode `IsValid(string)` und bringen `FileExtensionManager` bei, dieses Interface zu implementieren. Das Ganze funktioniert immer noch genauso, aber Sie können nun den »echten« Manager durch Ihren »Fake«-Manager, den Sie später erzeugen werden, ersetzen und damit Ihren Test unterstützen.

Sie haben noch nicht den Stub Extension Manager erzeugt, also lassen Sie uns das jetzt tun. Das Ergebnis zeigt das folgende Listing.

```
//implementiere IExtensionManager:
public class AlwaysValidFakeExtensionManager:IExtensionManager
{
    public bool IsValid(string fileName)
    {
        return true;
    }
}
```

Listing 3.3: Ein einfacher Code für den Stub, der immer `true` zurückgibt

Beachten Sie zunächst den eindeutigen Namen dieser Klasse. Das ist sehr wichtig. Er ist nicht `StubExtensionManager` oder `MockExtensionManager`. Er ist `FakeExtensionManager`. Ein Fake bezeichnet ein Objekt, das wie ein anderes Objekt aussieht, aber wie ein *Mock* oder *Stub* verwendet werden kann. (Das nächste Kapitel behandelt Mock-Objekte ausführlicher.)

Indem Sie sagen, dass ein Objekt oder eine Variable ein Fake ist, verschieben Sie die Entscheidung, wie dieses so ähnlich aussehende Objekt benannt werden soll, und vermeiden jede Konfusion, die entstanden wäre, wenn Sie es *Mock* oder *Stub Extension Manager* genannt hätten.

Wenn Leute »Mock« oder »Stub« hören, dann erwarten sie ein bestimmtes Verhalten, was wir später noch diskutieren werden. Sie aber wollen nicht sagen, wie diese Klasse benannt ist, denn Sie werden diese Klasse in einer Art erzeugen, die es ihr erlaubt, als beides zu agieren, damit in Zukunft verschiedene Tests die Klasse wiederverwenden können.

Dieser Fake Extension Manager gibt immer `true` zurück, daher nennen Sie die Klasse `AlwaysValidFakeExtensionManager`. So wird der Leser Ihres zukünftigen Tests verstehen, wie das Verhalten des Fake-Objekts ist, ohne den Quellcode lesen zu müssen.

Dies ist nur eine Technik und sie kann zu einer Explosion von handgeschriebenen Fakes in Ihrem Code führen. Handgeschriebene Fakes sind Fakes, die Sie komplett in gewöhnlichem Code schreiben, ohne sie mithilfe eines Frameworks zu erzeugen. Eine andere Technik zur Konfiguration Ihrer Fakes sehen Sie etwas später in diesem Kapitel.

Sie können diesen Fake in Ihrem Test verwenden, damit kein Test jemals eine Abhängigkeit vom Dateisystem beinhaltet. Aber Sie können auch Code hinzufügen, der es erlaubt, das Werfen jeglicher Art von Ausnahmen zu simulieren. Auch dazu später mehr.

Nun haben Sie ein Interface und zwei Klassen, die es implementieren, aber Ihre zu testende Methode ruft die echte Implementierung immer noch direkt auf:

```
public bool IsValidFileName (string fileName)
{
    IExtensionManager mgr = new FileExtensionManager();
    return mgr.IsValid(fileName);
}
```

Sie müssen Ihrer Methode irgendwie beibringen, mit Ihrer Implementierung zu kommunizieren, statt mit der ursprünglichen Implementierung von `IExtensionManager`. Sie müssen ein Seam in Ihren Code einbauen, wo Sie Ihren Stub einstöpseln können.

3.4.2 Dependency Injection: Injiziere eine Fake-Implementierung in die zu testende Unit

Es gibt mehrere bewährte Wege, um Interface-basierte Seams in Ihren Code einzufügen – Stellen, an denen Sie die Implementierung eines Interface in eine Klasse einbauen können, die von den Methoden verwendet werden soll. Dies sind einige der relevanten Wege:

- Übergebe dem Konstruktor ein Interface und speichere es zur späteren Benutzung in einem Feld ab.
- Übergebe einer Property ein Interface und speichere es zur späteren Benutzung in einem Feld ab.
- Übergebe ein Interface unmittelbar vor dem Aufruf einer zu testenden Methode unter Verwendung einer der folgenden Möglichkeiten:
  - ein Parameter der Methode (*Parameter Injection*)
  - eine Fabrikklasse
  - eine lokale Fabrikmethode
  - Variationen der genannten Methoden

Die Parameter-Injection-Methode ist trivial: Sie übergeben die Instanz einer (vorgetäuschten) Abhängigkeit an die fragliche Methode, indem Sie der Signatur der Methode einen Parameter hinzufügen.

Lassen Sie uns Schritt für Schritt durch den Rest der möglichen Lösungen gehen und schauen, warum Sie sie jeweils benutzen sollten.

3.4.3 Injiziere einen Fake auf Konstruktor-Ebene (Construktor Injection)

In diesem Szenario fügen Sie einen neuen Konstruktor hinzu (oder einen neuen Parameter zu einem existierenden Konstruktor), der ein Objekt vom Typ des Interface akzeptiert, das Sie zuvor extrahiert haben (`IExtensionManager`). Der Konstruktor setzt dann ein lokales Feld in der Klasse, das vom Typ des Interface ist und das später von Ihrer Methode oder einer anderen verwendet werden kann. Abbildung 3.5 zeigt den Ablauf der Stub-Injektion.

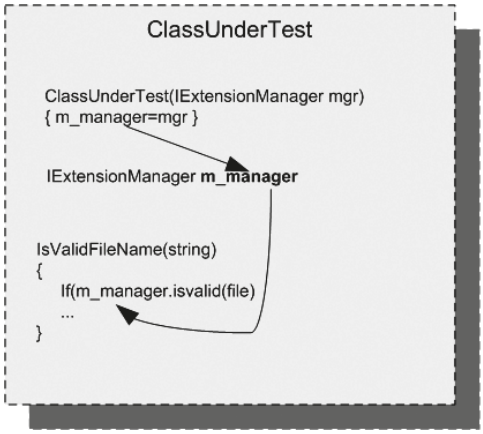


Abb. 3.5: Ablauf der Injektion via Konstruktor

Das folgende Listing zeigt, wie Sie den Test für Ihre Klasse `LogAnalyzer` mithilfe der Constructor-Injection-Technik schreiben können.

```
public class LogAnalyzer    //definiert den Produktionscode
{
    private IExtensionManager manager;

    //definiere Konstruktor, der von Tests aufgerufen werden kann:
    public LogAnalyzer(IExtensionManager mgr)
    {
        manager = mgr;
    }
    public bool IsValidLogFileName(string fileName)
    {
        return manager.IsValid(fileName);
    }
}

public interface IExtensionManager
{
    bool IsValid(string fileName);
}

//definiere den Testcode:
[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void IsValidFileName_NameSupportedExtension_ReturnsTrue()
```

```
    {
        //setze Stub und gebe true zurück:
        FakeExtensionManager myFakeManager =
                                        new FakeExtensionManager();
        myFakeManager.WillBeValid = true;

        LogAnalyzer log =
            new LogAnalyzer(myFakeManager);    //übergibt Stub

        bool result = log.IsValidLogFileName("short.ext");
        Assert.True(result);
    }
}

//definiere einen Stub, der den einfachsten möglichen
//Mechanismus benutzt:
internal class FakeExtensionManager : IExtensionManager
{
    public bool WillBeValid = false;

    public bool IsValid(string fileName)
    {
        return WillBeValid;
    }
}
```

Listing 3.4: Der Einbau Ihres Stubs über die Constructor Injection

Anmerkung

Der Fake Extension Manager befindet sich in derselben Datei wie der Testcode, denn derzeit wird der Fake nur innerhalb dieser Testklasse verwendet. Es ist wesentlich einfacher, einen handgeschriebenen Fake in derselben Datei statt in einer anderen zu finden, zu lesen und zu warten. Falls Sie später einmal eine weitere Klasse haben, die diesen Fake verwendet, so können Sie ihn einfach mithilfe eines Tools wie ReSharper (das ich ausdrücklich empfehle) in eine andere Datei verschieben. Lesen Sie dazu den Anhang.

Sie werden auch bemerken, dass das Fake-Objekt in Listing 3.4 anders aussieht als das, das Sie zuvor gesehen haben. Ihm kann über den Test mitgeteilt werden, welchen booleschen Wert es zurückgeben soll, wenn seine Methode aufgerufen wird. Die Konfigurierbarkeit des Stubs innerhalb des Tests bedeutet, dass der Quellcode der Stub-Klasse in mehr als einem Testfall wiederverwendet werden kann, wenn der Test die Werte für den Stub setzt, bevor er ihn für das zu testende Objekt verwendet. Dies hilft auch, die Lesbarkeit des Testcodes zu verbessern, denn der Leser des Codes kann den Test lesen und findet alles, was er wissen muss, an einer Stelle. Lesbarkeit ist ein wichtiger Aspekt beim Schreiben von Unit Tests und ich werde dies im weiteren Verlauf des Buches im Detail behandeln, insbesondere in Kapitel 8.

Eine andere bemerkenswerte Sache ist, dass Sie durch die Verwendung von Parametern im Konstruktor diese Parameter zu nicht-optionalen Abhängigkeiten gemacht haben (unter der Annahme, dass dies der einzige Konstruktor ist). Das ist eine Designentscheidung. Der Benutzer des Typs muss Argumente für alle spezifischen Abhängigkeiten angeben, die benötigt werden.

Probleme mit der Constructor Injection

Bei der Verwendung der Constructor Injection kann es zu Problemen kommen. Wenn Ihr zu testender Code mehr als einen Stub benötigt, um korrekt und ohne Abhängigkeiten zu arbeiten, dann führt das Hinzufügen von mehr und mehr Konstruktoren (und mehr und mehr Konstruktorparametern) zu Scherereien und kann sogar den Code weniger lesbar und weniger wartbar machen.

Angenommen, `LogAnalyzer` hätte neben der Abhängigkeit vom Dateisystem auch noch weitere Abhängigkeiten von einem Webservice und einem Logging-Dienst. Der Konstruktor könnte dann folgendermaßen aussehen:

```
public LogAnalyzer (IExtensionManager mgr, ILog logger,
                    IWebService service)
{
    // dieser Konstruktor kann von Tests aufgerufen warden
    manager = mgr;
    log = logger;
    svc = service;
}
```

Eine Lösung für diese Probleme ist es, eine spezielle Klasse anzulegen, die all die Werte enthält, die zur Initialisierung unserer Klasse benötigt werden, und dann nur einen Parameter an die Methode zu übergeben: den Typ dieser Klasse. Auf diese Weise gibt man nur ein Objekt mit allen relevanten Abhängigkeiten weiter. (Dies ist auch bekannt als *Parameter Object Refactoring*.) Zwar kann dies auch schnell – bei Dutzenden von Properties für ein Objekt – aus der Hand gleiten, aber es ist möglich.

Eine andere mögliche Lösung ist die Verwendung von *Inversion of Control*(IoC)-Containern. Sie können sich IoC-Container als »schlaue Fabriken« für Ihre Objekte vorstellen (obwohl sie viel mehr als nur das sind). Einige bekannte Container dieses Typs sind Microsoft Unity, StructureMap und Castle Windsor. Sie stellen spezielle Fabrikmethoden zur Verfügung, die den Objekttyp, den Sie erzeugen möchten, und alle benötigten Abhängigkeiten aufnehmen und dann das Objekt initialisieren, indem sie spezielle, konfigurierbare Regeln anwenden, wie etwa welcher Konstruktor aufgerufen werden soll, welche Properties in welcher Reihenfolge gesetzt werden usw. Sie sind leistungsstark, wenn sie auf eine kompliziert zusammengesetzte Objekthierarchie angewendet werden, wo das Erzeugen eines Objekts das Erzeugen und die Initialisierung von weiteren Objekten erfordert, die mehrere Ebenen unterhalb angeordnet sind. Wenn beispielsweise Ihre Klasse ein `ILogger`-Interface für ihren Konstruktor benötigt, dann können Sie solch ein Containerobjekt so konfigurieren, dass es immer das gleiche `ILogger`-Objekt zurückgibt, das Sie hineingegeben haben, um die Interface-Anforderung zu erfüllen. Das Endergebnis der Verwendung von Containern ist gewöhnlich ein einfacheres Handhaben und Abrufen von Objekten und weniger Ärger mit den Abhängigkeiten oder der Verwaltung der Konstruktoren.

Hinweis

Es gibt viele andere erfolgreiche Container-Implementierungen, wie etwa Autofac oder Ninject – werfen Sie einen Blick darauf, wenn Sie mehr zu diesem Thema lesen. Die Behandlung von Containern geht über den Rahmen dieses Buches hinaus, aber Sie können mit Scott Hanselmans Liste beginnen, mehr darüber zu erfahren: `http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx`. Um *wirklich* einen tieferen Einblick in das Thema zu erhalten, empfehle ich Dependency Injection in .NET (Manning, 2011) von Mark Seemann. Wenn Sie das gelesen haben, sollten Sie in der Lage sein, Ihre eigenen Container von Grund auf selbst zu bauen. Ich verwende Container selten in meinem realen Code. Ich glaube, dass sie das Design und die Lesbarkeit der Dinge meist komplizierter machen. Wenn Sie einen Container benötigen, könnte es sein, dass Ihr Design geändert werden sollte. Was meinen Sie dazu?

Wann Sie die Constructor Injection verwenden sollten

Meine Erfahrung ist, dass die Verwendung von Argumenten im Konstruktor zur Initialisierung von Objekten meinen Testcode unhandlicher machen kann, wenn ich keine Frameworks wie IoC-Container zur Erzeugung von Objekten verwende. Aber ich bevorzuge diesen Weg, denn es ist das kleinere Übel, wenn es darum geht, lesbare und verständliche APIs zu haben. Auf der anderen Seite ist die Verwendung von Parametern in Konstruktoren eine großartige Möglichkeit, dem Benutzer Ihrer API zu signalisieren, dass diese Parameter nicht optional sind. Sie müssen angegeben werden, wenn das Objekt erzeugt wird.

Wenn Sie möchten, dass diese Abhängigkeiten optional sind, dann schauen Sie in den Abschnitt 3.4.5. Er diskutiert die Verwendung von Properties, was ein wesentlich entspannterer Weg ist, um optionale Abhängigkeiten zu definieren, als, sagen wir, der Klasse verschiedene Konstruktoren für jede Abhängigkeit hinzuzufügen.

Dies ist kein Design-Buch, genauso wenig wie es ein Buch zu TDD ist. Ich empfehle, mal wieder, *Clean Code* von Rob Martin zu lesen, das Ihnen dabei hilft, zu entscheiden, wann Sie Konstruktorparameter verwenden wollen. Das kann sein, nachdem Sie sich sicher auf dem Gebiet des Unit Testings fühlen, oder auch, bevor Sie überhaupt damit anfangen, das Unit Testing zu lernen. Zwei oder mehr wesentliche Fähigkeiten auf einmal zu lernen, kann jedoch eine fast unüberwindliche Mauer errichten, die es sehr schwer und umständlich macht, die Dinge zu lernen. Indem Sie die Fähigkeiten einzeln angehen, werden Sie gut in jeder davon.

Hinweis

Sie werden entdecken, dass das Dilemma, welche Technik oder welches Design in welcher Situation verwendet werden sollte, in der Welt des Unit Testings weit verbreitet ist. Das ist eine wunderbare Sache. Hinterfragen Sie immer Ihre Annahmen; Sie könnten etwas Neues lernen.

Wenn Sie sich dafür entscheiden, die Constructor Injection zu nutzen, dann werden Sie wahrscheinlich IoC-Container verwenden wollen. Das wäre eine großartige Lösung, wenn der gesamte Code der Welt ebenfalls IoC-Container verwenden würde, aber die meisten kennen das Prinzip der Inversion der Kontrolle nicht, geschweige denn die Tools, die man verwenden kann, um das Prinzip Realität werden zu lassen. In der Zukunft des Unit Tes-

tings werden wir wahrscheinlich mehr und mehr den Gebrauch dieser Frameworks sehen. Während das geschieht, werden Sie immer klarer die Regeln erkennen, nach denen Klassen mit Abhängigkeiten entworfen werden sollten, und Sie werden Tools sehen, die das Problem der Dependency-Injection (DI) lösen, ohne dazu Konstruktoren verwenden zu müssen.

In jedem Fall sind Konstruktorparameter ein möglicher Weg. Properties werden ebenfalls häufig verwendet.

3.4.4 Simuliere Ausnahmen über Fakes

Nun folgt ein einfaches Beispiel, wie Sie Ihre Fake-Klasse so konfigurieren können, dass sie eine Ausnahme wirft. Damit können Sie jeden beliebigen Typ einer Exception simulieren, wenn eine Methode aufgerufen wird. Angenommen, Sie testen die folgende Anforderung: Wenn der File Extension Manager eine Ausnahme wirft, dann wollen Sie `false` zurückgeben, die Ausnahme aber nicht weiterreichen (ja, im wahren Leben wäre das ein schlechtes Vorgehen, aber haben Sie bitte um des Beispiels willen Nachsicht mit mir).

```
[Test]
public void
IsValidFileName_ExtManagerThrowsException_ReturnsFalse()
{
    FakeExtensionManager myFakeManager = new FakeExtensionManager();
    myFakeManager.WillThrow = new Exception("dies ist ein Fake");

    LogAnalyzer log = new LogAnalyzer(myFakeManager);
    bool result = log.IsValidLogFileName("irgendwas.irgendeineerweiterung");
    Assert.False(result);
}

internal class FakeExtensionManager : IExtensionManager
{
    public bool WillBeValid = false;
    public Exception WillThrow = null;

    public bool IsValid(string fileName)
    {
        if(WillThrow != null)
            { throw WillThrow; }
        return WillBeValid;
    }
}
```

Um diesen Test erfolgreich durchlaufen zu lassen, müssten Sie Code schreiben, der den File Extension Manager innerhalb einer `try-catch`-Anweisung aufruft und `false` zurückgibt, wenn die `catch`-Anweisung erreicht wird.

3.4.5 Injiziere ein Fake als Property Get oder Set

In diesem Szenario fügen Sie eine Property für jede Abhängigkeit, die Sie einschieben möchten, hinzu. Sie benutzen diese Abhängigkeit, wenn Sie sie in Ihrem zu testenden Code benötigen. Abbildung 3.6 zeigt den Ablauf einer Injektion mit Properties.

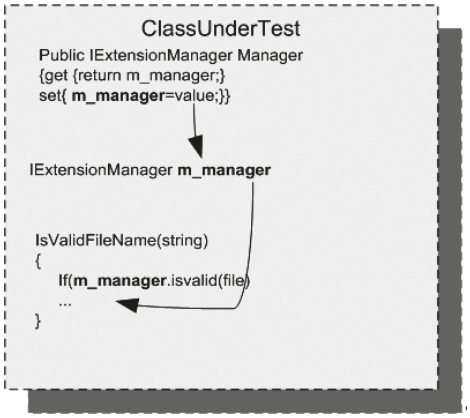


Abb. 3.6: Die Verwendung von Properties zum Einfügen der Abhängigkeiten. Das ist wesentlich einfacher als die Verwendung eines Konstruktors, denn jeder Test braucht nur die Properties zu setzen, die er für seine Ausführung benötigt.

Unter Verwendung dieser Technik (die auch *Dependency Injection* genannt wird, ein Ausdruck, der auch auf die anderen Techniken, die in diesem Kapitel beschrieben werden, angewandt werden kann) sieht Ihr Testcode dem in Abschnitt 3.4.3, der die Constructor Injection verwendete, recht ähnlich. Aber dieser Code, wie als Nächstes dargestellt, ist lesbarer und leichter zu schreiben.

```
public class LogAnalyzer
{
    private IExtensionManager manager;

    public LogAnalyzer ()
    {
        manager = new FileExtensionManager();
    }

    //erlaube das Setzen der Abhängigkeit via Property:
    public IExtensionManager ExtensionManager
    {
        get { return manager; }
        set { manager = value; }
    }

    public bool IsValidLogFileName(string fileName)
```

```
    {
        return manager.IsValid(fileName);
    }
}

[Test]
public void IsValidFileName_SupportedExtension_ReturnsTrue()
{
    //setze den Stub auf, stelle sicher, dass er true zurückgibt
    ...
    //erzeuge Analyzer und injiziere Stub
    LogAnalyzer log = new LogAnalyzer ();
    log.ExtensionManager =
        someFakeManagerCreatedEarlier;    //injiziert Stub

    //Überprüfe Logik via Assert unter der Annahme,
    //dass die Erweiterung unterstützt wird
    ...
}
```

Listing 3.5: Einfügen eines Fakes durch das Ergänzen der zu testenden Klasse um eine Property

Wie die Constructor Injection hat auch die Property Injection eine Auswirkung auf das API-Design im Hinblick auf die Definition, welche Abhängigkeiten erforderlich sind und welche nicht. Durch die Verwendung von Properties sagen Sie unter dem Strich: »Diese Abhängigkeit wird nicht benötigt, um mit diesem Typ zu arbeiten.«

Wann Sie die Property Injection verwenden sollten

Verwenden Sie diese Technik, wenn Sie signalisieren wollen, dass eine Abhängigkeit der zu testenden Klasse optional ist oder dass die Abhängigkeit eine Voreinstellung hat, die während des Tests keine Probleme verursachen wird.

3.4.6 Injiziere einen Fake unmittelbar vor einem Methodenaufruf

Dieser Abschnitt beschäftigt sich mit einem Szenario, bei dem Sie, unmittelbar bevor Sie die ersten Operationen damit ausführen, die Instanz eines Objekts erhalten, statt sie über den Konstruktor oder eine Property zu beziehen. Der Unterschied besteht darin, dass in dieser Situation das zu testende Objekt selbst die Nachfrage nach dem Stub initiiert; in den vorherigen Abschnitten war die Instanz des Fakes von externem Code an den zu testenden Code übergeben worden, bevor der Test gestartet wurde.

Verwende eine Fabrikklasse (Factory Class)

In diesem Szenario gehen Sie zurück zu den Grundlagen, wo eine Klasse den Manager im Konstruktor initialisiert, aber sie erhält die Instanz von einer Fabrikklasse. Das *Fabrikmuster* (*Factory Pattern*) ist ein Design, das es einer anderen Klasse erlaubt, die Verantwortung für das Erzeugen von Objekten zu übernehmen.

Ihre Tests werden die Fabrikklasse (die in diesem Fall eine statische Methode verwendet, die eine Instanz eines Objekts zurückgibt, das `IExtensionManager` implementiert) so konfigurieren, dass sie einen Stub statt der realen Implementierung zurückgibt. Abbildung 3.7 zeigt dies.

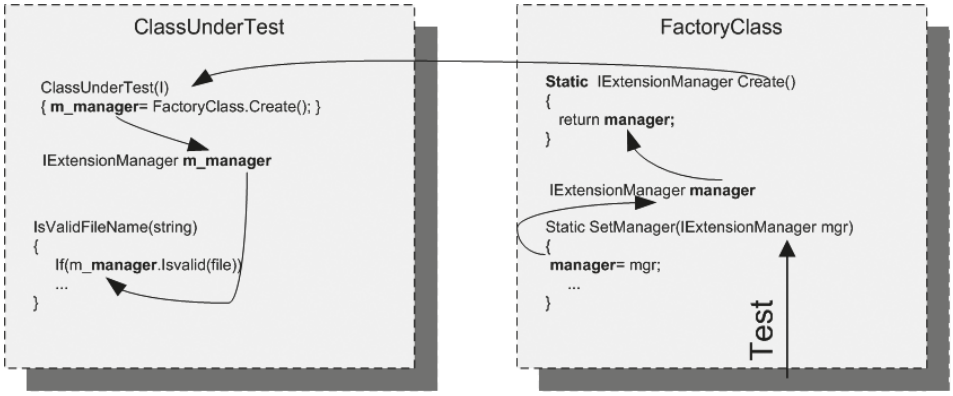


Abb. 3.7: Ein Test konfiguriert die Fabrikklasse so, dass sie einen Stub zurückgibt. Die zu testende Klasse benutzt die Fabrikklasse, um diese Instanz zu erhalten, während im Produktionscode ein Objekt zurückgegeben würde, das kein Stub ist.

Dies ist ein sauberes Design. Viele objektorientierte Systeme verwenden Fabrikklassen, um die Instanzen von Objekten zurückzugeben. Aber um das gekapselte Klassendesign zu wahren, erlauben die meisten Systeme niemandem außerhalb der Fabrikklasse, die Instanz, die zurückgegeben werden soll, zu ändern.

In diesem Fall habe ich eine neue Set-Methode zur Fabrikklasse (ein neues Seam) hinzugefügt, damit Ihre Tests eine größere Kontrolle darüber haben, welche Instanz zurückgegeben wird. Sobald Sie statische Variablen in Ihren Testcode einbauen, müssen Sie eventuell den Zustand der Fabrik vor oder nach jedem Testlauf zurücksetzen, damit andere Tests nicht von der Konfiguration betroffen sind.

Diese Technik bringt einen Testcode hervor, der einfach zu lesen ist, und es gibt eine klare Trennung der Belange zwischen den Klassen. Jede ist für eine andere Aktion verantwortlich.

Das nächste Listing zeigt Code, der die Fabrikklasse in `LogAnalyzer` verwendet (und ebenfalls die Tests einschließt).

```
public class LogAnalyzer
{
    private IExtensionManager manager;

    public LogAnalyzer ()
    {
        //verwende Fabrik im Produktionscode:
        manager = ExtensionManagerFactory.Create();
    }
```

```
    public bool IsValidLogFileName(string fileName)
    {
        return manager.IsValid(fileName)
        && Path.GetFileNameWithoutExtension(fileName).Length>5;
    }
}

[Test]
public void IsValidFileName_SupportedExtension_ReturnsTrue()
{
    //setze den Stub auf, stelle sicher, dass er true zurückgibt
    ...
    //setze für diesen Test Stub in die Fabrik ein:
    ExtensionManagerFactory.SetManager(myFakeManager);
    //erzeuge Analyzer und injiziere Stub
    LogAnalyzer log = new LogAnalyzer ();

    //Überprüfe Logik via Assert unter der Annahme,
    //dass die Erweiterung unterstützt wird
    ...
    }
}

class ExtensionManagerFactory
{
    private IExtensionManager customManager=null;
    public IExtensionManager Create()
    {
        //definiere eine Fabrik, die einen benutzerdefinierten
        //Manager verwenden und zurückgeben kann:
        if(customManager!=null) return customManager;
        return new FileExtensionManager();
    }
    public void SetManager(IExtensionManager mgr)
    {
        customManager = mgr;
    }
}
```

Listing 3.6: Eine Fabrikklasse wird so konfiguriert, dass sie einen Stub zurückgibt, wenn der Test durchgeführt wird.

Die Implementierung der Fabrikklasse kann deutlich variieren, und die gezeigten Beispiele verkörpern nur die einfachste Darstellung. Mehr zu Fabriken, der Fabrikmethode und dem

Entwurfsmuster der Abstrakten Fabrik (*Abstract Factory Design Pattern*) erfahren Sie im Klassiker *Design Patterns* (mitp-Verlag, 2015) der »Gang of Four« (Erich Gamma, Richard Helm, Ralph Johnson und John M. Vlissides).

Das Einzige, was Sie sicherstellen müssen, wenn Sie diese Entwurfsmuster benutzen, ist, den Fabriken, die Sie erzeugen, ein Seam hinzuzufügen, damit sie Ihre Stubs statt der Originalimplementierung zurückgeben können. Viele Systeme haben einen globalen `#debug`-Schalter, der, wenn eingeschaltet, dafür sorgt, dass die Seams automatisch die Fake- oder Testobjekte statt der Originalimplementierung zurückgeben. Das aufzusetzen, kann eine Menge Arbeit sein, aber es zahlt sich aus, wenn es an der Zeit ist, das System zu testen.

Das Verbergen der Seams im Release-Modus

Was, wenn Sie nicht wollen, dass die Seams im Release-Modus sichtbar sind? Es gibt mehrere Wege, das zu erreichen. In .NET können Sie beispielsweise die Seam Statements (den zusätzlichen Konstruktor, die Property oder die Set-Methode) in ein Argument für die bedingte Kompilierung einbetten. Mehr dazu wird in Abschnitt 3.6.2 folgen.

Verschiedene Level der Indirektion

Sie haben es hier mit einer anderen Schichttiefe zu tun als im vorigen Abschnitt. Bei jeder unterschiedlichen Tiefe können Sie sich dafür entscheiden, ein anderes Objekt durch einen Fake (oder Stub) zu ersetzen. Tabelle 3.1 zeigt drei Schichttiefen, die im Code verwendet werden und Stubs zurückgeben können.

Zu testender Code	Mögliche Aktion
Schichttiefe 1: die Variable `FileExtensionManager` innerhalb der Klasse	Fügen Sie ein Konstruktorargument hinzu, das als Abhängigkeit verwendet wird. Ein Member der zu testenden Klasse ist nun gefälscht; der gesamte restliche Code bleibt unverändert.
Schichttiefe 2: die Abhängigkeit, die von der Fabrikklasse an die zu testende Klasse zurückgegeben wird	Bringen Sie der Fabrikklasse bei, die Fake-Abhängigkeit durch das Setzen einer Property zurückzugeben. Der Member der Fabrikklasse ist nun gefälscht; die zu testende Klasse ist gar nicht geändert.
Schichttiefe 3: die Fabrikklasse, die die Abhängigkeit zurückgibt	Ersetzen Sie die Instanz der Fabrikklasse durch eine Fake-Fabrik, die Ihre Fake-Abhängigkeit zurückgibt. Die Fabrik ist ein Fake, die ebenso ein Fake zurückgibt; die zu testende Klasse ist nicht geändert.

Tabelle 3.1: Code-Schichten, die gefälscht werden können

Der springende Punkt bei den Indirektionsschichten ist, dass je tiefer Sie in den Kaninchenbau (den Call Stack der Code-Ausführung) »hineinklettern«, desto bessere Manipulationsmöglichkeiten haben Sie im Hinblick auf den zu testenden Code, denn Sie erzeugen Stubs, die im weiteren Verlauf für mehr Dinge verantwortlich sind. Aber es gibt auch eine Kehrseite: Je weiter Sie die Schichten herabsteigen, desto schwieriger wird der Test zu verstehen sein und desto schwieriger wird es sein, die richtige Stelle zu finden, um ein Seam einzufügen. Der Trick besteht darin, die richtige Balance zwischen Komplexität und Manipulationsmöglichkeiten zu finden, damit Ihre Tests zwar lesbar bleiben, Sie aber die volle Kontrolle über die Testsituation behalten.

Für das Szenario in Listing 3.6 (unter Verwendung einer Fabrik) würde das Hinzufügen eines Konstruktorarguments die Dinge komplizieren, da Sie doch schon eine gute, mögliche Zielschicht für Ihren Seam haben – die Fabrik in Tiefe 2. Schicht 2 ist die am leichtesten zu nutzende, denn die benötigten Code-Änderungen sind minimal:

- *Schicht 1 (wir fälschen einen Member der zu testenden Klasse)*

  Sie müssten einen Konstruktor hinzufügen, die Klasse im Konstruktor setzen, ihre Parameter aus dem Test setzen und Sie müssten sich über zukünftige Anwendungen dieser API im Produktionscode Gedanken machen. Diese Methode würde die Semantik bei der Verwendung der zu testenden Klasse ändern, was man besser vermeidet, solange man keinen guten Grund hat.

- *Schicht 2 (wir fälschen einen Member in der Fabrikklasse)*

  Diese Methode ist einfach. Fügen Sie der Fabrik eine Property hinzu und setzen Sie eine Fake-Abhängigkeit Ihrer Wahl ein. Es gibt keine Änderung der Semantik der Code-Basis, alles bleibt beim Alten und der Code ist äußerst einfach. Das einzige Gegenargument zu dieser Methode ist, dass Sie wissen müssen, wer die Fabrik aufruft und wann, was bedeutet, dass Sie ein wenig forschen müssen, bevor Sie den Code so einfach implementieren können. Eine Code-Basis zu verstehen, die man nie zuvor gesehen hat, ist eine beängstigende Aufgabe, aber das scheint dennoch vernünftiger zu sein als die anderen Optionen.

- *Schicht 3 (wir fälschen die Fabrikklasse)*

  Sie müssten Ihre eigene Version der Fabrikklasse entwerfen, die vielleicht ein Interface hat oder auch nicht. Das bedeutet, Sie müssen auch ein Interface entwerfen. Dann müssten Sie die Instanz Ihrer Fake-Fabrik erstellen, ihr mitteilen, dass sie Ihre gefälschte Abhängigkeitsklasse zurückgibt (beachten Sie: Ein Fake gibt ein Fake zurück!), und dann müssten Sie noch die Fake-Fabrikklasse in der zu testenden Klasse einsetzen. Ein Fake, das ein Fake zurückgibt, hat immer etwas von einem verrückten Szenario, was am besten vermieden wird, denn es macht Ihren Test weniger verständlich.

Fake-Methode – benutzen Sie eine lokale Fabrikmethode (Extract and Override)

Diese Methode liegt nicht in einer der in Tabelle 3.1 aufgeführten Schichten; sie erzeugt eine komplett neue Indirektionsschicht nahe an der Oberfläche des zu testenden Codes. Je näher Sie der Oberfläche des Codes kommen, desto weniger müssen Sie mit Änderungen von Abhängigkeiten herumspielen. In diesem Fall ist die zu testende Klasse auch eine Art von Abhängigkeit, die Sie manipulieren müssen.

In diesem Szenario benutzen Sie eine lokale, *virtuelle* Methode in der zu testenden Klasse als eine Fabrik, um die Instanz des Extension Managers zu erhalten. Weil diese Methode als *virtuell* markiert ist, kann sie in einer abgeleiteten Klasse überschrieben werden, womit Sie dann Ihren Seam erzeugt haben. Sie injizieren einen Stub in die Klasse durch *Vererbung* von unserer zu testenden Klasse an eine neue Klasse, dem *Überschreiben* der virtuellen Fabrikmethode und schließlich, aus der überschreibenden Methode heraus, der Rückgabe der Instanz, die in der neuen Klasse konfiguriert wurde. Die Tests werden dann mit der neuen, abgeleiteten Klasse durchgeführt. Die Fabrikmethode könnte auch als Stub-Methode, die ein Stub-Objekt zurückgibt, bezeichnet werden. Abbildung 3.8 zeigt den Fluss der Objektinstanzen.

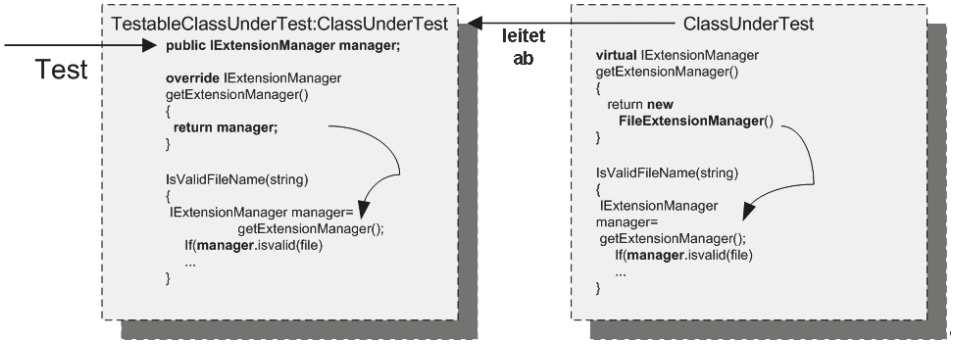


Abb. 3.8: Sie erben von der zu testenden Klasse, wodurch Sie ihre virtuelle Fabrikmethode überschreiben können, und geben eine beliebige Objektinstanz zurück, solange sie `IExtensionManager` implementiert. Dann führen Sie Ihre Tests mit der neuen, abgeleiteten Klasse durch.

Hier sind die Schritte, um eine Fabrikmethode in Ihren Tests zu verwenden:

- In der zu testenden Klasse
  - fügen Sie eine virtuelle Fabrikmethode hinzu, die die originale Instanz zurückgibt.
  - verwenden Sie die Fabrikmethode wie gewöhnlich in Ihrem Code.
- In Ihrem Testprojekt erzeugen Sie eine neue Klasse:
  - Leiten Sie die neue Klasse von der zu testenden ab.
  - Legen Sie ein öffentliches Feld (es ist nicht nötig, dafür eine Property zu verwenden) für den Interface-Typ an, den Sie ersetzen wollen (`IExtensionManager`).
  - Überschreiben Sie die virtuelle Fabrikmethode.
  - Geben Sie das öffentliche Feld zurück.
- In Ihrem Testcode
  - erzeugen Sie eine Instanz der Stub-Klasse, die das benötigte Interface implementiert (`IExtensionManager`).
  - erzeugen Sie eine Instanz der *neuen, abgeleiteten Klasse, nicht* der zu testenden Klasse.
  - konfigurieren Sie das öffentliche Feld (das Sie bereits erzeugt haben) der neuen Instanz und setzen darin den Stub ein, den Sie in Ihrem Test erzeugt haben.

Wenn Sie Ihre Klasse nun testen, wird Ihr Produktionscode über die überschriebene Fabrikmethode Ihren Fake verwenden.

Das folgende Listing zeigt, wie der Code aussehen könnte, wenn Sie diese Methode verwenden.

```
public class LogAnalyzerUsingFactoryMethod
{
    public bool IsValidLogFileName(string fileName)
    {
```

```
        //verwendet virtuelle GetManager()-Methode:
        return GetManager().IsValid(fileName);
    }

    protected virtual IExtensionManager GetManager()
    {
        //gibt einen festen Wert zurück:
        return new FileExtensionManager();
    }
}

[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void overrideTest()
    {
        FakeExtensionManager stub = new FakeExtensionManager();
        stub.WillBeValid = true;

        //erzeuge Instanz einer von der zu testenden Klasse
        //abgeleiteten Klasse:
        TestableLogAnalyzer logan = new TestableLogAnalyzer(stub);

        bool result = logan.IsValidLogFileName("file.ext");

        Assert.IsFalse(result);
    }
}

class TestableLogAnalyzer : LogAnalyzerUsingFactoryMethod
{
    public TestableLogAnalyzer(IExtensionManager mgr)
    {
        Manager = mgr;
    }

    public IExtensionManager Manager;

    //gibt zurück, was wir ihm sagen:
    protected override IExtensionManager GetManager()
    {
        return Manager;
    }
}
```

```
internal class FakeExtensionManager : IExtensionManager
{
    //keine Änderung zum vorhergehenden Beispiel
    ...
}
```

Listing 3.7: Das Fälschen einer Fabrikmethode

Die hier verwendete Methode heißt *Extract and Override* (»Extrahieren und Überschreiben«) und Sie werden merken, dass sie sehr einfach anzuwenden ist, sobald Sie es ein paar Mal ausprobiert haben. Es ist eine leistungsstarke Technik und eine, die ich in diesem Buch auch zu anderen Zwecken einsetzen werde.

Hinweis

Mehr über diese und weitere abhängigkeitsauflösende Techniken können Sie in dem Buch *Effektives Arbeiten mit Legacy Code* von Michael Feathers erfahren, von dem ich glaube, dass es sein Gewicht in Gold wert ist.

Extract and Override ist eine leistungsstarke Technik, weil sie Sie die Abhängigkeit direkt ersetzen lässt, ohne in den Kaninchenbau hinabsteigen zu müssen (also das Ändern der Abhängigkeiten tief unten im Call Stack). Das macht die Durchführung schnell und sauber und korrumpiert nahezu Ihren Sinn für die objektorientierte Ästhetik, indem es zu Code mit wahrscheinlich weniger Interfaces und mehr virtuellen Methoden führt. Ich nenne diese Methode auch gerne »Ex-Crack and Override«, denn es ist sehr schwer, die Finger davon zu lassen, sobald man sie einmal kennt.

Wann Sie diese Methode verwenden sollten

Extract and Override ist großartig, um die *Inputs* Ihres zu testenden Codes zu simulieren, aber es wird mühselig, wenn Sie damit Interaktionen verifizieren wollen, die *aus* dem zu testenden Code *heraus* auf Ihre Abhängigkeiten einwirken.

Beispielsweise ist es großartig, wenn Ihr Testcode einen Webservice aufruft und einen *Return-Wert* erhält und Sie diesen nun mit Ihrem eigenen Wert simulieren wollen. Aber es wird schnell schwierig, wenn Sie testen möchten, ob Ihr Code den Webservice korrekt *aufruft*. Das erfordert eine Menge manuelles Codieren, und Isolation-Frameworks sind für solche Aufgaben besser geeignet (wie Sie im nächsten Kapitel sehen werden). Extract and Override ist gut, wenn Sie Rückgabewerte oder ganze Interfaces als Rückgabewerte simulieren möchten, aber nicht, um Interaktionen zwischen den Objekten zu prüfen.

Ich benutze diese Technik häufig, wenn ich die Inputs meines zu testenden Codes simulieren möchte, denn sie hilft, die Änderungen an der Semantik der Code-Basis (neue Interfaces, Konstruktoren usw.) ein wenig besser handhabbar zu halten. Sie müssen nicht so viele Änderungen vornehmen, um den Code in einen testbaren Zustand zu bringen. Ich benutze diese Technik nur dann nicht, wenn die Code-Basis mir einen anderen Weg ganz klar vorzeichnet: Es gibt bereits ein Interface, das sich aufdrängt, gefälscht zu werden, oder es gibt bereits eine gute Stelle, an der ein Seam injiziert werden kann. Wenn diese Dinge nicht existieren und die Klasse selbst nicht versiegelt ist (oder die Versiegelung nicht ohne zu viel

Ärger mit Ihren Kollegen gelöst werden kann), dann überprüfe ich als Erstes diese Technik, bevor ich zu komplizierteren Verfahren übergehe.

3.5 Variationen der Refactoring-Technik

Es gibt viele Variationen der eben vorgestellten einfachen Techniken, um Seams in den Quellcode einzubauen. Beispielsweise können Sie, statt dem Konstruktor einen Parameter hinzuzufügen, ihn direkt in die zu testende Methode einbauen. Statt ein Interface zu übergeben, können Sie eine Basisklasse verwenden usw. Jede Variation hat ihre eigenen Stärken und Schwächen.

Einer der Gründe, warum Sie möglicherweise lieber keine Basisklasse statt eines Interface verwenden wollen, ist, dass eine Basisklasse durch den Produktionscode bereits eingebaute Abhängigkeiten aufweisen kann (und wahrscheinlich auch tut), von denen Sie wissen und die Sie überschreiben müssen. Das macht das Implementieren abgeleiteter Klassen für das Testen schwieriger als das Implementieren eines Interface, das Sie genau wissen lässt, was die darunter liegende Implementierung ist, und Ihnen die volle Kontrolle darüber gibt.

In Kapitel 4 schauen wir uns Techniken an, die Ihnen helfen, handgeschriebene Fakes, die Interfaces implementieren, nicht von Hand schreiben zu müssen, sondern stattdessen Frameworks verwenden, die Sie zur Laufzeit dabei unterstützen.

Aber für den Augenblick lassen Sie uns noch eine andere Möglichkeit anschauen, die uns die Kontrolle über den zu testenden Code *ohne* die Verwendung von Interfaces geben kann. Sie haben bereits auf den vorhergehenden Seiten einen möglichen Weg gesehen, diese Methode ist jedoch so effektiv, dass sie eine eigenständige Diskussion verdient.

3.5.1 Die Verwendung von Extract and Override, um Fake-Resultate zu erzeugen

Sie haben bereits in Abschnitt 3.4.6 ein Beispiel für Extract and Override gesehen. Sie leiten von der zu testenden Klasse ab, wodurch Sie eine virtuelle Methode überschreiben und sie zwingen können, Ihren Stub zurückzugeben.

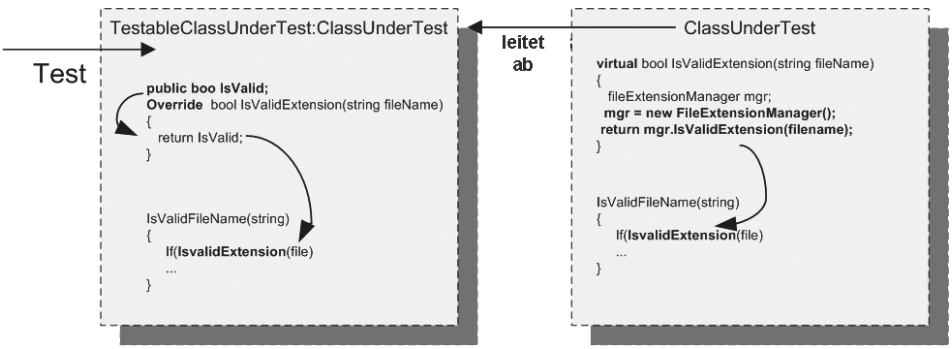


Abb. 3.9: Die Verwendung von Extract and Override, um ein logisches Resultat zurückzugeben, statt die eigentliche Abhängigkeit aufzurufen. Das Beispiel verwendet ein einfaches Fake-Resultat anstelle eines Stubs.

Aber warum hier aufhören? Was ist, wenn Sie nicht jedes Mal, wenn Sie die Kontrolle über ein Verhalten in Ihrem zu testenden Code benötigen, ein neues Interface hinzufügen wollen oder können? In diesen Fällen kann Extract and Override dabei helfen, die Dinge zu vereinfachen, denn es zwingt Sie nicht, neue Interfaces zu schreiben und einzubauen – Sie müssen nur von der zu testenden Klasse ableiten und ein Verhalten in der Klasse überschreiben.

Abbildung 3.9 zeigt einen anderen Weg, wie Sie den zu testenden Code zwingen können, immer `true` bei der Validierung der Dateierweiterung zurückzugeben.

In der zu testenden Klasse *virtualisieren* Sie das *Resultat der Berechnung*, statt eine *Fabrikmethode* zu virtualisieren. Das heißt, Sie überschreiben die Methode in Ihrer abgeleiteten Klasse und geben den Wert zurück, den Sie wollen, ohne ein Interface oder einen neuen Stub erzeugen zu müssen. Sie vererben und überschreiben einfach die Methode, um das gewünschte Resultat zurückzugeben.

Die folgenden Listings zeigen, wie Ihr Code bei Verwendung dieser Technik aussehen könnte.

```
public class LogAnalyzerUsingFactoryMethod
{
    public bool IsValidLogFileName(string fileName)
    {
        return this.IsValid(fileName);
    }

    protected virtual bool IsValid(string fileName)
    {
        FileExtensionManager mgr = new FileExtensionManager();
        //gibt Resultat der echten Abhängigkeit zurück:
        return mgr.IsValid(fileName);
    }
}

[Test]
public void overrideTestWithoutStub()
{
    TestableLogAnalyzer logan = new TestableLogAnalyzer();
    //setze gefälschten Ergebniswert:
    logan.IsSupported = true;

    bool result = logan.IsValidLogFileName("file.ext");
    Assert.True(result,"...");
}

class TestableLogAnalyzer: LogAnalyzerUsingFactoryMethod
{
```

```
    public bool IsSupported;

    protected override bool IsValid(string fileName)
    {
        //gibt vom Test gefälschten Wert zurück:
        return IsSupported;
    }
}
```

Listing 3.8: Aus der extrahierten Methode wird ein Resultat statt eines Stub-Objekts zurückgegeben.

Wann Sie Extract and Override verwenden sollten

Die grundlegende Motivation für die Verwendung dieser Technik ist die gleiche wie die für die in Abschnitt 3.4.6 vorgestellte Methode. Diese Technik ist sogar einfacher als die vorangegangene. Wenn ich kann, bevorzuge ich diese Technik gegenüber der Letzteren, denn sie ist viel einfacher.

Inzwischen mögen Sie zu sich selber sagen, dass das Hinzufügen all dieser Konstruktoren, Properties, Methoden und Fabriken um des Testens willen problematisch ist. Es durchbricht einige schwerwiegende objektorientierte Prinzipien, insbesondere die Idee der Kapselung (*Encapsulation*), die besagt: »Verbirg alles, was der Benutzer deiner Klasse nicht sehen muss.« Das ist unser nächstes Thema. (Kapitel 11 beschäftigt sich auch mit Aspekten der Testbarkeit und des Designs.)

3.6 Die Überwindung des Kapselungsproblems

Einige glauben, es sei eine schlechte Idee, das Design zu öffnen, um es besser testen zu können, weil dies die objektorientierten Prinzipien, auf denen das Design basiert, verletzt. Diesen Leuten kann ich aufrichtig sagen: »Seid nicht dumm!« Objektorientierte Techniken sind dazu da, dem Endbenutzer der API (der Endbenutzer ist der Programmierer, der Ihr Objektmodell benutzen wird) einige Beschränkungen aufzuerlegen, damit das Objektmodell fachgerecht verwendet wird und vor einer nicht beabsichtigten Art der Verwendung geschützt ist. Objektorientierung hat auch eine Menge mit der Wiederverwendung von Code und dem Single Responsibility Principle (Eine-Verantwortlichkeit-Prinzip) zu tun (das erfordert, dass jede Klasse nur eine Verantwortung besitzt).

Wenn Sie Unit Tests für Ihren Code schreiben, fügen Sie dem Objektmodell einen weiteren Benutzer (den Test) hinzu. Dieser ist genauso wichtig wie der eigentliche Endbenutzer, aber er hat andere Ziele bei der Benutzung des Modells. Der Test hat spezifische Anforderungen an das Objektmodell, die der Basislogik hinter einigen objektorientierten Prinzipien zu widersprechen scheinen, vor allem der *Kapselung* (*Encapsulation*). Irgendwo die externen Abhängigkeiten zu kapseln, ohne irgendwem zu erlauben, sie zu ändern, private Konstruktoren oder versiegelte Klassen, nicht virtuelle Methoden, die nicht überschrieben werden können: All das sind klassische Anzeichen eines überängstlichen Designs. (Ein sicherheitsbezogenes Design ist die Ausnahme, der ich es nachsehe.) Das Problem ist, dass der zweite Benutzer der API, der Test, diese externen Abhängigkeiten als Features im Code benötigt. Ich bezeichne ein Design, das unter Berücksichtigung der Testbarkeit entworfen wurde, als ein »Testbares objektorientiertes Design« (»Testable Object-Oriented Design«, TOOD). In Kapitel 11 werden Sie mehr über TOOD erfahren.

Das Konzept des *testbaren Designs* gerät, nach Ansicht einiger Leute, in Konflikt mit dem Konzept des objektorientierten Designs. Wenn Sie wirklich diese beiden Welten vereinigen wollen (Sie essen Ihren Kuchen und behalten ihn trotzdem), dann sind hier ein paar Tipps und Tricks, mit deren Hilfe Sie sicherstellen können, dass die extra Konstruktoren und Setters nicht im Release-Modus erscheinen oder zumindest keine Rolle im Release-Modus spielen.

Hinweis

Eine gute Stelle, um Ausschau nach Designzielen zu halten, die die Idee des testbaren Designs besser wahren, ist Bob Martins *Clean Code*.

3.6.1 Die Verwendung von internal und [InternalsVisibleTo]

Wenn Sie Ihrer Klasse keinen öffentlichen Konstruktor hinzufügen möchten, den jeder sehen kann, dann können Sie das Attribut `internal` statt `public` verwenden. Sie können dann alle mit `internal` attributierten Member und Methoden für Ihr Test-Assembly sichtbar machen, indem Sie das Assembly-Level-Attribut `[InternalsVisibleTo]` setzen. Das nächste Listing zeigt dies klarer.

```
public class LogAnalyzer
{
    ...
    internal LogAnalyzer (IExtensionManager extentionMgr)
    {
        manager = extentionMgr;
    }
    ...
}

using System.Runtime.CompilerServices;
[assembly: InternalsVisibleTo("AOUT.CH3.Logan.Tests")]
```

Listing 3.9: Die `internal`-Member werden dem Test-Assembly sichtbar gemacht.

Ein solcher Code findet sich gewöhnlich in `AssemblyInfo.cs`-Dateien. Die Verwendung von `internal` ist eine gute Lösung, wenn es keine andere Möglichkeit gibt, die Dinge für den Testcode öffentlich zu machen.

3.6.2 Die Verwendung des Attributs [Conditional]

Das Attribut `System.Diagnostics.ConditionalAttribute` ist im Hinblick auf seine nicht intuitive Wirkung bemerkenswert. Wenn Sie dieses Attribut auf eine Methode anwenden, dann initialisieren Sie es mit einem String, der für den Parameter eines bedingten Builds (Conditional Build) steht, der als Teil des Build-Prozesses angegeben wird. (DEBUG und RELEASE sind die beiden gebräuchlichsten. Visual Studio verwendet sie automatisch in Abhängigkeit von Ihrem Build-Typ.)

Wenn das Build-Flag während des Build-Vorgangs *nicht* gesetzt ist, werden die *Aufrufe* der mit dem Attribut versehenen Methoden nicht in das Build eingeschlossen. Beispielsweise

werden für ein Release-Build alle Aufrufe zu dieser Methode entfernt, aber die Methode selber bleibt:

```
[Conditional("DEBUG")]
public void DoSomething()
{
}
```

Sie können dieses Attribut auf Methoden (aber nicht auf Konstruktoren) anwenden, die nur in bestimmten Debug-Modi aufgerufen werden sollen.

Anmerkung

Die mit diesem Attribut versehenen Methoden werden nicht vor dem Produktionscode versteckt, was sich vom Verhalten der Technik, die ich als Nächstes vorstellen werde, unterscheidet.

Es ist wichtig zu erkennen, dass die Verwendung von Konstrukten zur bedingten Kompilierung in Ihrem Produktionscode dessen Lesbarkeit verringern und zu einem »spaghettiartigeren« Code führen kann. Also Vorsicht!

3.6.3 Die Verwendung von #if und #endif zur bedingten Kompilierung

Das Setzen Ihrer Methoden oder spezieller Test-Only-Konstruktoren zwischen `#if`- und `#endif`-Konstrukten stellt sicher, dass sie nur dann kompiliert werden, wenn das Build-Flag gesetzt ist, wie das nächste Listing zeigt.

```
#if DEBUG
    public LogAnalyzer (IExtensionManager extensionMgr)
    {
        manager = extensionMgr;
    }
#endif
...
#if DEBUG
    [Test]
    public void
        IsValidFileName_SupportedExtension_True()
    {
        ...
        //erzeuge Analyzer und injiziere Stub
        LogAnalyzer log = new LogAnalyzer (myFakeManager);
        ...
    }
#endif
```

Listing 3.10: Die Verwendung spezieller Build-Flags

Diese Methode ist weit verbreitet, aber sie kann zu chaotisch aussehendem Code führen. Erwägen Sie, um der Klarheit willen, die Verwendung des Attributs `[InternalVisibleTo]` dort, wo es möglich ist.

3.7 Zusammenfassung

Sie sind in den ersten Kapiteln mit dem Schreiben simpler Tests gestartet, aber Sie hatten Abhängigkeiten in Ihren Tests, für die Sie einen Weg finden mussten, sie zu überschreiben. In diesem Kapitel haben Sie gelernt, wie Sie mithilfe von Interfaces und Vererbung Stubs bauen und so die Abhängigkeiten eliminieren können.

Ein Stub kann auf unterschiedliche Weise in den Code injiziert werden. Der wahre Trick besteht darin, die richtige Indirektionsschicht zu finden oder zu erzeugen und sie dann als *Seam* zu benutzen, von wo aus der Stub in den laufenden Code injiziert werden kann.

Wir nennen diese Klassen *Fake*, denn wir wollen sie nicht daran binden, nur als Stubs oder nur als Mocks verwendet zu werden.

Je tiefer Sie in die Indirektionsschichten eintauchen, desto schwieriger wird es, den Test und den zu testenden Code sowie dessen tief gehende Interaktion mit anderen Objekten zu verstehen. Je näher Sie an der Oberfläche des zu testenden Objekts bleiben, desto einfacher wird Ihr Test zu verstehen und handzuhaben sein, aber Sie geben damit möglicherweise auch einigen Spielraum ab, die Umgebung des zu testenden Objekts zu manipulieren.

Lernen Sie die verschiedenen Wege kennen, einen Stub in Ihren Code zu injizieren. Wenn Sie sie beherrschen, werden Sie in einer wesentlich besseren Position sein, zu entscheiden, welche Methode Sie wann anwenden.

Die Extract-and-Override-Methode ist hervorragend für die Simulation von Inputs in den zu testenden Code geeignet, aber wenn Sie auch Interaktionen zwischen Objekten testen (das Thema des nächsten Kapitels), dann sollten Sie sicherstellen, dass sie ein Interface statt eines beliebigen Rückgabewerts zurückgibt. Das macht Ihr Testleben einfacher.

TOOD kann einige interessante Vorteile gegenüber dem klassischen objektorientierten Design bieten, wie etwa die Unterstützung der Wartbarkeit während des Schreibens von Tests für die Code-Basis.

In Kapitel 4 werden wir einen Blick auf einige andere Abhängigkeitsaspekte werfen und Wege finden, sie aufzulösen. Sie werden erfahren, wie man es vermeidet, von Hand Fakes für Interfaces zu schreiben, und wie man die Interaktion zwischen Objekten als Teil der Unit Tests überprüft.

Kapitel 4

Interaction Testing mit Mock-Objekten

Dieses Kapitel behandelt

- die Definition von Interaction Testing
- Mock-Objekte
- den Unterschied zwischen Fakes, Mocks und Stubs
- eine Diskussion der besten Methoden beim Einsatz von Mock-Objekten

Im vorangegangenen Kapitel haben Sie das Problem gelöst, wie Code getestet werden kann, der, um korrekt zu laufen, von anderen Objekten abhängt. Sie haben Stubs verwendet, damit der zu testende Code allen benötigten Input erhält und Sie seine Logik unabhängig testen können.

Bisher haben Sie nur Tests geschrieben, die mit den ersten beiden der drei Typen von Endresultaten, die eine Unit of Work haben kann, umgehen können: die Rückgabe eines Wertes und die Änderung des Systemzustands.

In diesem Kapitel werden wir uns anschauen, wie Sie die dritte Art von Endresultaten testen können – den Aufruf eines Third-Party-Objekts. Sie werden prüfen, ob ein Objekt ein anderes korrekt aufruft. Das aufgerufene Objekt hat vielleicht keinen Rückgabewert oder speichert keinen Zustand, aber es hat eine komplexe Logik, die in korrekten Aufrufen von anderen Objekten, die nicht unter Ihrer Kontrolle oder nicht Teil der zu testenden Unit of Work sind, resultieren muss. Der Ansatz, den Sie bisher verfolgt haben, wird es hier nicht tun, denn es gibt keine externalisierte API, die Sie heranziehen können, um zu überprüfen, ob sich irgendetwas im zu testenden Objekt geändert hat. Wie testet man, dass ein Objekt mit anderen Objekten korrekt interagiert? Sie werden Mock-Objekte verwenden.

Als Erstes müssen wir das Interaction Testing definieren und wie es sich von Ihrer bisherigen Art zu testen – dem wertbasierten und dem zustandsbasierten Testen – unterscheidet.

4.1 Wertbasiertes Testen versus zustandsbasiertes Testen versus Testen versus Interaction Testing

Ich habe die drei Arten von Endresultaten, die Units of Work erzeugen können, in Kapitel 1 definiert. Ich werde nun das Interaction Testing definieren, das sich mit der dritten Art von Resultaten beschäftigt: dem Aufruf eines Third-Party-Objekts. Wertbasiertes Testen prüft den Rückgabewert einer Funktion. Zustandsbasiertes Testen beschäftigt sich mit der Prüfung von wahrnehmbaren Verhaltensänderungen des zu testenden Systems, nachdem sich sein Zustand geändert hat.

Definition

Interaction Testing ist das Überprüfen, wie ein Objekt Nachrichten an andere Objekte versendet (Methoden aufruft). Sie verwenden Interaction Testing, wenn der Aufruf eines anderen Objekts das Endresultat einer spezifischen Unit of Work ist.

Sie können sich das Interaction Testing als »aktionsgetriebenes Testen« vorstellen. *Aktionsgetriebenes* Testen meint, dass Sie eine bestimmte Aktion testen, die ein Objekt unternimmt (wie das Versenden einer Nachricht an ein anderes Objekt).

Verwenden Sie das Interaction Testing nur als *letzte* Möglichkeit. Dies ist sehr wichtig. Es ist besser, zu versuchen, die ersten beiden Arten (Wert oder Zustand) der Endresultate einer Unit of Work zu verwenden, denn sehr vieles wird erheblich komplizierter mit Interaction Tests, wie Sie in diesem Kapitel sehen werden. Aber manchmal, wie im Fall eines Third-Party-Aufrufs in einen Logger, sind die Interaktionen zwischen Objekten das Endresultat. Dann müssen Sie die Interaktion selbst testen.

Beachten Sie bitte, dass nicht jeder den Standpunkt teilt, dass Mocks nur dann verwendet werden sollten, wenn es keine anderen Möglichkeiten gibt, die Software zu testen. In *Growing Object-Oriented Software, Guided by Tests* befürworten Steve Freeman und Nat Pryce, was viele »die Londoner Schule des TDD« nennen, die zu Design-Zwecken Mocks und Stubs als eine Möglichkeit verwenden, das Design der Software zu verschmelzen. Ich widerspreche ihnen nicht in Gänze, dass dies ein gangbarer Weg ist, den Code zu designen. Aber dieses Buch handelt *nicht* von Design, und unter reinen Wartbarkeits-Aspekten führt die Verwendung von Mocks zu mehr Ärger, als wenn ich sie nicht benutze. Das ist meine Erfahrung, aber ich lerne immer Neues hinzu. Es kann sein, dass ich in der nächsten Auflage dieses Buches meine Meinung zu diesem Thema um 180 Grad geändert haben werde.

Interaction Testing existiert auf die eine oder andere Art schon seit den ersten Tagen des Unit Testings. Damals gab es noch keine Namen oder Muster dafür, aber die Leute mussten trotzdem wissen, ob ein Objekt das andere korrekt aufgerufen hatte. Die meiste Zeit wurde es entweder übertrieben oder schlecht durchgeführt und resultierte in unwartbarem und unlesbarem Code. Daher empfehle ich immer Tests zu den zwei anderen Arten von Endresultaten.

Lassen Sie uns ein Beispiel betrachten, um die Vor- und Nachteile des Interaction Testings zu verstehen. Nehmen wir an, Sie besitzen ein Bewässerungssystem und Sie haben Ihr System konfiguriert, wann es den Baum in Ihrem Vorgarten bewässern soll: wie oft am Tag und wie viel Wasser jeweils. Hier sind zwei Arten, um zu testen, ob es korrekt arbeitet:

- *Zustandsbasierter Integrationstest* (jawohl, Integrations- und nicht Unit Test) – Lassen Sie das System für 12 Stunden laufen und während dieser Zeit sollte es den Baum mehrfach wässern. Am Ende dieser Zeit prüfen Sie den Zustand des bewässerten Baums. Ist der Boden feucht genug, geht es dem Baum gut, sind seine Blätter grün usw. Es mag ein recht schwierig durchzuführender Test sein, aber angenommen, Sie bekommen das hin, dann können Sie herausfinden, ob Ihr Bewässerungssystem funktioniert. Ich nenne das einen Integrationstest, denn er ist sehr, sehr laaangsam, und seine Durchführung bindet das ganze Umfeld des Bewässerungssystems ein.
- *Interaction Testing* – Bringen Sie am Ende des Bewässerungsschlauchs ein Gerät an, das aufzeichnet, wann die Bewässerung beginnt und endet und wie viel Wasser durch das Gerät fließt und wie oft. Am Ende des Tages überprüfen Sie, ob das Gerät die richtige

Anzahl von Aufzeichnungen gemacht hat, jedes Mal mit der richtigen Wassermenge und ohne dass Sie sich Gedanken um die Überprüfung des Baums machen müssen. Tatsächlich brauchen Sie nicht einmal einen Baum, um zu prüfen, ob das System funktioniert. Sie können noch weiter gehen und die Systemuhr der Bewässerungseinheit manipulieren (sie wird zu einem Stub), damit das System glaubt, der nächste Zeitpunkt zur Bewässerung sei gekommen, und es wird den Baum wässern, wann immer Sie möchten. Auf diese Weise müssen Sie nicht warten (in diesem Beispiel waren es 12 Stunden) und können direkt herausfinden, ob es funktioniert.

Wie Sie an diesem Beispiel sehen, kann der Einsatz eines Interaction Tests Ihr Leben deutlich vereinfachen.

Aber manchmal ist das zustandsbasierte Testen der beste Weg, wenn das Interaction Testing zu schwierig hinzubekommen ist. Das ist bei Crash Test Dummies der Fall: Ein Auto kracht mit einer bestimmten Geschwindigkeit in ein stehendes Ziel und nach dem Zusammenstoß werden Auto und Dummy überprüft, um das Ergebnis zu ermitteln. Das Durchführen dieser Art von Test als Interaction Test im Labor kann zu kompliziert sein und ein praktischer zustandsbasierter Test ist notwendig. (Die Leute arbeiten am Computer an der Simulation von Zusammenstößen, aber es kommt den Tests eines echten Crashs noch nicht nahe.)

Nun zurück zum Bewässerungssystem. Was ist das für ein Gerät, das die Bewässerungsinformationen aufzeichnet? Es ist ein nachgeahmter (Fake) Wasserschlauch oder Sie könnten es auch einen »Stub« nennen. Aber es ist eine schlaue Art von Stub, ein Stub, der alle Aufrufe an sich aufzeichnet, und Sie verwenden ihn, um festzulegen, ob Ihr Test erfolgreich ist oder nicht. Das ist halbwegs das, was ein Mock-Objekt ist. Und was ist die Uhr, die Sie durch die manipulierte Uhr ersetzen? Sie ist ein Stub, denn sie dreht nur Däumchen und simuliert die Zeit, weshalb Sie einen anderen Teil des Systems bequemer testen können.

Definition

Ein *Mock-Objekt* ist ein nachgeahmtes Objekt im System, das entscheidet, ob ein Unit Test funktioniert hat oder fehlgeschlagen ist. Es macht dies, indem es verifiziert, ob das zu testende Objekt das Fake-Objekt wie erwartet aufgerufen hat. Gewöhnlich gibt es nicht mehr als einen Mock pro Test.

Ein Mock-Objekt mag sich scheinbar nicht großartig von einem Stub unterscheiden, aber die Unterschiede sind groß genug, um für Diskussionen und eine spezielle Syntax in verschiedenen Frameworks zu sorgen, wie Sie in Kapitel 5 sehen werden. Lassen Sie uns anschauen, was genau der Unterschied ist.

Nachdem ich jetzt die Idee der Fakes, Mocks und Stubs, behandelt habe, ist es Zeit für eine formale Definition des Konzepts der Fakes.

Definition

Ein *Fake* ist ein allgemeiner Begriff, der benutzt werden kann, um entweder ein Stub- oder Mock-Objekt (handgeschrieben oder nicht) zu bezeichnen, denn beide sehen aus wie das echte Objekt. Ob es sich bei dem Fake um einen Stub oder einen Mock handelt, hängt davon ab, wie er im aktuellen Test verwendet wird. Wenn er dazu benutzt wird, eine Interaktion zu überprüfen, ist er ein *Mock-Objekt*. Anderenfalls handelt es sich um einen *Stub*.

Lassen Sie uns tiefer einsteigen, um die Abgrenzung zwischen den beiden Typen eines Fakes zu untersuchen.

4.2 Der Unterschied zwischen Mocks und Stubs

Ein Stub ersetzt ein Objekt, wodurch Sie ein anderes Objekt ohne Probleme testen können. Abbildung 4.1 zeigt die Interaktion zwischen dem Stub und der zu testenden Klasse.

Die Unterscheidung zwischen Mocks und Stubs ist wichtig, denn viele der heutigen Tools und Frameworks (wie auch Artikel) benutzen diese Begriffe, um verschiedene Dinge zu beschreiben. Sie ist auch wichtig, denn zu verstehen, dass Sie es mit mehr als einem Mock-Objekt zu tun haben, ist eine wichtige Fähigkeit, die Sie lernen müssen, wenn Sie die Tests anderer begutachten. Es gibt eine Menge Verwirrung darüber, was die einzelnen Begriffe bedeuten und viele scheinen sie synonym zu verwenden. Sobald Sie aber die Unterschiede kennen, können Sie die Welt der Tools, Frameworks und APIs sorgfältiger einschätzen und klarer verstehen, was jedes einzelne tut.

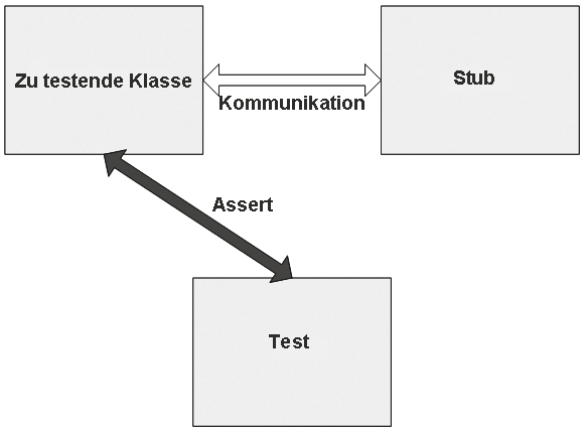


Abb. 4.1: Wenn ein Stub verwendet wird, dann wird das Assert mit der zu testenden Klasse ausgeführt. Der Stub hilft dabei, sicherzustellen, dass der Test reibungslos abläuft.

Auf den ersten Blick mögen die Unterschiede zwischen Mocks und Stubs klein oder gar nicht vorhanden sein. Die Unterscheidung ist subtil, aber wichtig, denn viele der Mock-Objekt-Frameworks, mit denen Sie in den nächsten Kapiteln zu tun haben werden, benutzen diese Begriffe zur Beschreibung von unterschiedlichem Verhalten im Framework. Der grundlegende Unterschied ist, dass Stubs bei Tests nicht durchfallen können, Mocks aber schon.

Der einfachste Weg, um zu erkennen, dass Sie es mit einem Stub zu tun haben, ist, zur Kenntnis zu nehmen, dass ein Stub niemals durch den Test fallen kann. Der Test benutzt die Asserts immer gegen die zu testende Klasse.

Auf der anderen Seite wird der Test ein Mock-Objekt benutzen, um zu verifizieren, ob der Test fehlgeschlagen ist oder nicht. Abbildung 4.2 zeigt die Interaktion zwischen einem Test und einem Mock-Objekt. Beachten Sie, dass das Assert an dem Mock vorgenommen wird.

Noch einmal: Das Mock-Objekt ist das Objekt, das Sie verwenden, um zu sehen, ob der Test schiefgeht oder nicht.

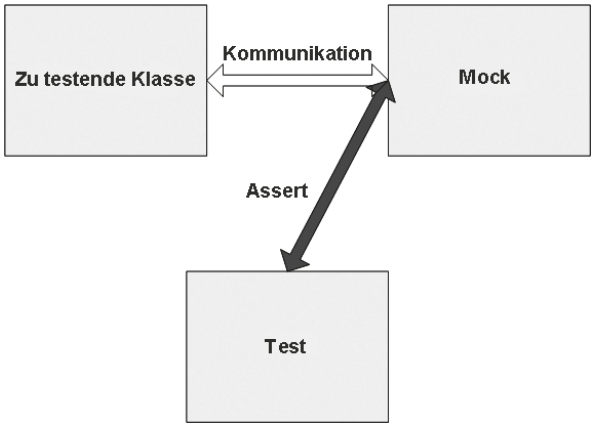


Abb. 4.2: Die zu testende Klasse kommuniziert mit dem Mock-Objekt und die gesamte Kommunikation wird vom Mock aufgezeichnet. Der Test benutzt das Mock-Objekt, um zu verifizieren, dass er erfolgreich ist.

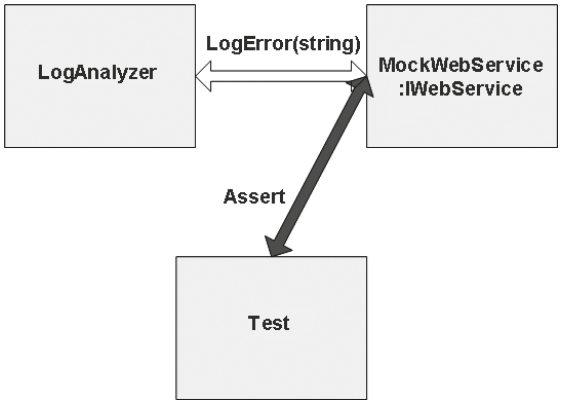


Abb. 4.3: Ihr Test erzeugt einen `MockWebService`, um die Nachrichten, die `LogAnalyzer` sendet, aufzuzeichnen. Das Assert wird dann gegen den `MockWebService` durchgeführt.

Lassen Sie uns diese Ideen in Aktion betrachten, indem Sie Ihr eigenes Mock-Objekt bauen.

4.3 Ein einfaches manuelles Mock-Beispiel

Das Erzeugen und Verwenden eines Mock-Objekts ist im Wesentlichen wie das Verwenden eines Stubs, außer dass ein Mock etwas mehr tut als ein Stub: Es zeichnet den Verlauf der Kommunikation auf, der später in Form von *Erwartungen* verifiziert werden wird.

Lassen Sie uns eine neue Anforderung an Ihre Klasse `LogAnalyzer` stellen. Dieses Mal soll sie mit einem externen Webservice interagieren, der eine Fehlernachricht erhält, wann immer `LogAnalyzer` auf einen zu kurzen Dateinamen trifft.

Unglücklicherweise funktioniert der Webservice, mit dem Sie testen wollen, noch nicht vollständig, aber selbst wenn er das täte, würde es zu lange dauern, ihn als Teil Ihres Tests zu verwenden. Deshalb führen Sie ein Refactoring Ihres Designs durch und erzeugen ein neues Interface, für das Sie später ein Mock-Objekt entwerfen können. Das Interface besitzt nur die Methoden, die Sie für unseren Webservice benötigen und sonst nichts.

Abbildung 4.3 zeigt, wie Ihr Mock, implementiert als `FakeWebService`, in den Test passt.

Als Erstes extrahieren Sie ein einfaches Interface, das Sie in Ihrem zu testenden Code verwenden können, statt direkt mit dem Webservice zu kommunizieren:

```
public interface IWebService
{
    void LogError(string message);
}
```

Dieses Interface wird Ihnen von Nutzen sein, wenn Sie sowohl Stubs als auch Mocks erzeugen wollen. Mit ihm können Sie eine externe Abhängigkeit, über die Sie keine Kontrolle haben, vermeiden.

Als Nächstes erzeugen Sie das Mock-Objekt selbst. Es mag wie ein Stub aussehen, aber es enthält ein extra Stück Code, das es zum Mock-Objekt macht:

```
public class FakeWebService:IWebService
{
    public string LastError;

    public void LogError(string message)
    {
        LastError = message;
    }
}
```

Diese handgeschriebene Klasse implementiert ein Interface, wie es auch ein Stub macht, aber es speichert einen Zustand für später, wodurch Ihr Test ein Assert ausführen und verifizieren kann, ob Ihr Mock korrekt aufgerufen wurde. Es ist immer noch kein Mock-Objekt. Es wird erst zu einem werden, wenn Sie es *wie eines* in Ihrem Test verwenden.

Anmerkung

Laut *xUnit Test Patterns: Refactoring Test Code* von Gerard Meszaros würde dies als »Test Spy« bezeichnet werden.

Das folgende Listing zeigt, wie der Test aussehen könnte.

```
[Test]
public void Analyze_TooShortFileName_CallsWebService()
{
    FakeWebService mockService = new FakeWebService();
```

```
    LogAnalyzer log = new LogAnalyzer(mockService);
    string tooShortFileName="abc.ext";

    log.Analyze(tooShortFileName);

    //überprüfe das Mock-Objekt:
    StringAssert.Contains("Dateiname zu kurz:abc.ext",
                                    mockService.LastError);
}

public class LogAnalyzer
{
    private IWebService service;

    public LogAnalyzer(IWebService service)
    {
        this.service = service;
    }

    public void Analyze(string fileName)
    {
        if(fileName.Length<8)
        {
            //protokolliert Fehler im Produktionscode:
            service.LogError("Dateiname zu kurz:" + fileName);
        }
    }
}
```

Listing 4.1: Das Testen von `LogAnalyzer` mit einem Mock-Objekt

Beachten Sie, wie das Assert mit dem Mock-Objekt und nicht mit der `LogAnalyzer`-Klasse durchgeführt wird. Das liegt daran, dass Sie die Interaktion zwischen `LogAnalyzer` und dem Webservice testen. Sie benutzen noch die gleiche DI-Techniken aus Kapitel 3, aber diesmal schafft oder unterbricht das Mock-Objekt (das statt des Stubs verwendet wird) auch den Test.

Beachten Sie auch, dass Sie die Tests nicht direkt innerhalb des Mock-Objekt-Codes schreiben. Es gibt mehrere Gründe dafür:

- Sie möchten in der Lage sein, das Mock-Objekt in anderen Testfällen wieder zu verwenden, mit anderen Asserts für die Benachrichtigung.
- Wenn die Asserts innerhalb der handgeschriebenen Fake-Klasse liegen würden, dann hätte jemand, der den Test liest, keine Ahnung, worauf Sie das Assert anwenden. Sie würden eine wesentliche Information vor dem Testcode verstecken, was die Lesbarkeit und die Wartbarkeit des Tests behindern würde.

Es kann sein, dass Sie in Ihren Tests feststellen, dass Sie mehr als ein Objekt ersetzen müssen. Die Kombination von Mocks und Stubs schauen wir uns als Nächstes an. Wie Sie sehen werden, ist es vollkommen in Ordnung, mehrere Stubs in einem einzigen Test zu haben, aber mehr als ein einzelner Mock kann Ärger bedeuten, denn dann testen Sie mehr als eine Sache.

4.4 Die gemeinsame Verwendung von Mock und Stub

Lassen Sie uns ein etwas komplizierteres Problem betrachten. Diesmal muss `LogAnalyzer` nicht nur mit einem Webservice »reden«, sondern wenn der Webservice eine Ausnahme wirft, muss `LogAnalyzer` den Fehler in eine externe Abhängigkeit schreiben, die ihn per E-Mail an den Webservice-Administrator schickt, wie in Abbildung 4.4 gezeigt wird.

Hier ist die Logik, die Sie in `LogAnalyzer` testen möchten:

```
if(fileName.Length<8)
{
    try
    {
        service.LogError("Dateiname zu kurz:" + fileName);
    }
    catch (Exception e)
    {
        email.SendEmail("a", "Betreff", e.Message);
    }
}
```

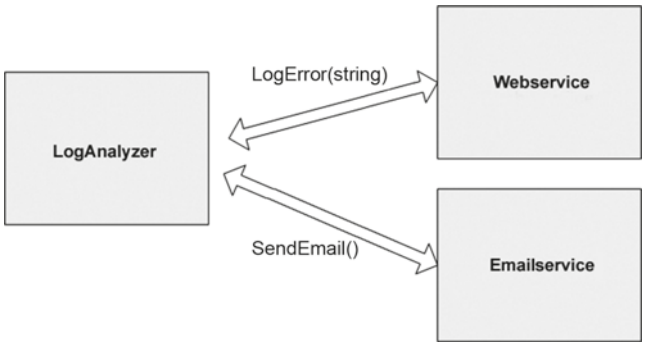


Abb. 4.4: `LogAnalyzer` hat zwei externe Abhängigkeiten: den Webservice und den E-Mail-Service. Sie müssen die Logik von `LogAnalyzer` beim Aufruf beider testen.

Beachten Sie hier, dass sich ein Teil der Logik nur auf die Interaktion mit externen Objekten bezieht; kein Wert wird zurückgegeben und kein Systemzustand ändert sich. Wie testen Sie, dass `LogAnalyzer` den E-Mail-Service korrekt aufruft, wenn der Webservice eine Ausnahme wirft?

Dies sind die Fragen, mit denen Sie es zu tun haben:

- Wie können Sie den Webservice ersetzen?
- Wie können Sie eine Ausnahme aus dem Webservice simulieren, damit Sie den Aufruf des E-Mail-Services testen können?
- Wie können Sie wissen, dass der E-Mail-Service korrekt oder überhaupt aufgerufen wird?

Sie können die ersten beiden Fragen erledigen, indem Sie einen Stub für den Webservice verwenden. Um das dritte Problem zu lösen, können Sie ein Mock-Objekt für den E-Mail-Service verwenden.

In Ihrem Test haben Sie zwei Fakes. Einer ist der E-Mail-Service-Mock, den Sie benutzen, um zu verifizieren, dass die richtigen Parameter an den E-Mail-Service übergeben werden. Der andere ist der Stub, den Sie verwenden, um eine Ausnahme, die der Webservice wirft, zu simulieren. Es ist ein Stub, denn Sie werden den Webservice-Fake nicht benutzen, um das Testresultat zu verifizieren, sondern nur, um den Test korrekt ablaufen zu lassen. Der E-Mail-Service ist ein Mock, weil Sie darauf ein Assert anwenden, um zu prüfen, dass er korrekt aufgerufen wird. Abbildung 4.5 zeigt dies.

Das nächste Listing zeigt den Code, der Abbildung 4.5 implementiert.

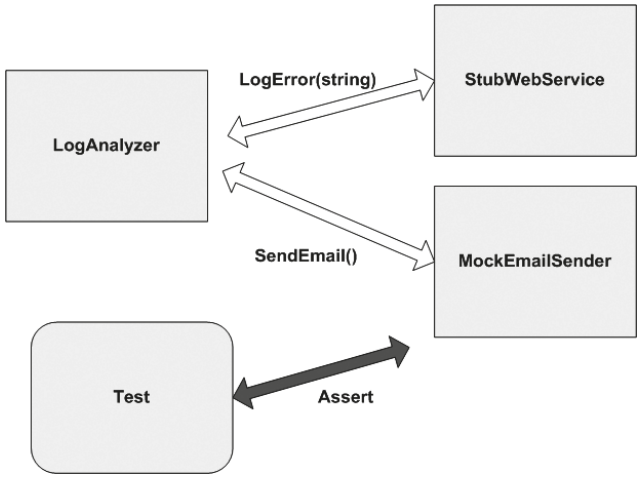


Abb. 4.5: Der Webservice wird durch einen Stub ersetzt, um eine Ausnahme zu simulieren; dann wird der E-Mail-Sender durch einen Mock ersetzt, um zu sehen, ob er korrekt aufgerufen wird. Der ganze Test handelt davon, wie `LogAnalyzer` mit anderen Objekten zusammenarbeitet.

```
public interface IEmailService
{
    void SendEmail(string to, string subject, string body);
}
public class LogAnalyzer2
{
    public LogAnalyzer2(IWebService service, IEmailService email)
```

```
    {
        Email = email;
        Service = service;
    }
    public IWebService Service
    {
        get;
        set;
    }
    public IEmailService Email
    {
        get;
        set;
    }

    public void Analyze(string fileName)
    {
        if(fileName.Length<8)
        {
            try
            {
                Service.LogError("Dateiname zu kurz:" + fileName);
            }
            catch (Exception e)
            {
                Email.SendEmail("jemand@irgendwo.com",
                                "kann nicht loggen",e.Message);
            }
        }
    }
}

[TestFixture]
public class LogAnalyzer2Tests
{
    [Test]
    public void Analyze_WebServiceThrows_SendsEmail()
    {
        FakeWebService stubService = new FakeWebService();
        stubService.ToThrow = new Exception("Fake Exception");

        FakeEmailService mockEmail = new FakeEmailService();

        LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);
```

```
        string tooShortFileName="abc.ext";
        log.Analyze(tooShortFileName);

        StringAssert.Contains("jemand@irgendwo.com",mockEmail.To);
        StringAssert.Contains ("Fake Exception",mockEmail.Body);
        StringAssert.Contains ("kann nicht loggen",mockEmail.Subject);
    }
}

public class StubService:IWebService
{
    public Exception ToThrow;
    public void LogError(string message)
    {
        if(ToThrow!=null)
        {
            throw ToThrow;
        }
    }
}

public class FakeEmailService:IEmailService
{
    public string To;
    public string Subject;
    public string Body;

    public void SendEmail(string to,
                          string subject,
                          string body)
    {
        To = to;
        Subject = subject;
        Body = body;
    }
}
```

Listing 4.2: Das Testen von `LogAnalyzer` mit einem Mock und einem Stub

Dieser Code beleuchtet einige interessante Punkte:

- Manchmal kann es ein Problem sein, mehrere Asserts zu haben, denn sobald ein Assert in Ihrem Test fehlschlägt, wirft es einen speziellen Typ von Ausnahme, der vom Test-Runner aufgefangen wird. Das bedeutet auch, dass keine Zeile unterhalb derer, die gerade zum Fehler geführt hat, ausgeführt wird. Im aktuellen Fall ist das in Ordnung, denn wenn ein Assert fehlschlägt, interessieren Sie die anderen nicht mehr, denn sie beziehen sich alle auf das gleiche Objekt und sind Teil des gleichen »Features«.

- Falls es Ihnen aber lieber wäre, wenn die anderen Asserts ebenfalls ausgeführt würden, auch wenn das erste fehlschlägt, so ist das ein guter Hinweis für Sie, diesen Test auf mehrere aufzuteilen. Alternativ könnten Sie vielleicht ein neues `EmailInfo`-Objekt anlegen, dem Sie alle drei Attribute geben. In Ihrem Test erzeugen Sie dann eine erwartete Version dieses Objekts mit allen korrekten Properties. Das wäre dann nur ein Assert.

So würde das dann aussehen:

```
Class EmailInfo
{
    public string Body;
    public string To;
    public string Subject;
}

[Test]
public void Analyze_WebServiceThrows_SendsEmail()
{
    FakeWebService stubService = new FakeWebService();
    stubService.ToThrow = new Exception("Fake Exception");

    FakeEmailService mockEmail = new FakeEmailService();

    LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);

    string tooShortFileName="abc.ext";
    log.Analyze(tooShortFileName);

    //erzeuge ein erwartetes Objekt:
    EmailInfo expectedEmail = new EmailInfo {
                                    Body = "Fake Exception",
                                    To = "jemand@irgendwo.com",
                                    Subject = ("kann nicht loggen" }

    //das Assert wird auf alle Properties gemeinsam angewendet:
    Assert.AreEqual(expectedEmail, mockEmail.email);
}

public class FakeEmailService:IEmailService
{
    public EmailInfo email = null;
    public void SendEmail(EmailInfo emailInfo)
    {
        email = emailInfo;
    }
}
```

Wenn Sie eine Referenz auf das *eigentliche* Objekt, das Sie als Endresultat erwarten, haben, dann könnten Sie auch das folgende verwenden:

```
Assert.AreSame(expectedEmail, mockEmail.email);
```

Eine wichtige Sache, die beachtet werden muss, ist, wie viele Mocks und Stubs in einem Test verwendet werden können.

4.5 Ein Mock pro Test

In einem Test, in dem man nur eine Sache testet (was ich Ihnen für das Schreiben von Tests empfehle), sollte nicht mehr als ein Mock-Objekt vorkommen. Alle anderen Fake-Objekte arbeiten als Stubs. Hat man mehr als einen Mock pro Test, so bedeutet das gewöhnlich, dass man mehr als eine Sache testet und das kann zu komplizierten oder brüchigen Tests führen. (Mehr hierzu finden Sie in Kapitel 8.)

Wenn Sie dieser Leitlinie folgen, sobald Sie es mit komplizierteren Tests zu tun haben, können Sie sich immer fragen: »Welches ist mein Mock-Objekt?« Sobald Sie es identifiziert haben, können Sie die anderen als Stubs belassen und keine Asserts darauf anwenden. (Wenn Sie Asserts auf Fakes finden, die eindeutig als Stubs verwendet werden, dann seien Sie auf der Hut. Das ist ein Anzeichen für eine Überspezifizierung.)

Überspezifizierung ist das Festlegen zu vieler Dinge, die passieren sollen, um die sich Ihr Test aber nicht kümmern sollte; beispielsweise, dass Stubs aufgerufen werden.

Diese Extra-Anforderungen können dazu führen, dass Ihr Test aus den falschen Gründen fehlschlägt: Sie ändern Ihren Produktions-Code, damit er anders arbeitet, aber obwohl das Endresultat Ihres Codes immer noch gut ist, wird Ihr Test Sie anschreien: »Ich bin fehlgeschlagen! Du sagtest mir, diese Methode würde aufgerufen werden und sie wurde es nicht! Wäääh!«

Dann werden Sie dauernd Ihre Tests ändern müssen, um sie für die interne Implementierung Ihres Codes passend zu machen – und schließlich werden Sie die Lust verlieren und sich fragen: »Warum mache ich das schon wieder?« Und dann werden Sie anfangen, diese Tests zu löschen.

Game over.

Spezifizieren Sie nur eine der drei Endresultate einer Unit of Work, oder der Himmel wird Ihnen auf den Kopf fallen und Sie enden in der Hölle. Ich habe Sie gewarnt.

Als Nächstes beschäftigen wir uns mit einem etwas komplexeren Szenario: die Verwendung eines Stubs, um einen Fake (Mock oder Stub) zurückzugeben, der von der Applikation benutzt wird.

4.6 Fake-Ketten: Stubs, die Mocks oder andere Stubs erzeugen

Eine der heutzutage am weitesten verbreiteten Online-Betrugstechniken folgt einem einfachen Muster: Eine gefälschte E-Mail wird an eine sehr große Zahl von Empfängern geschickt. Die gefälschte E-Mail ist von einer vorgetäuschten Bank oder einem vorgetäusch-

ten Onlinedienst und behauptet, der potenzielle Kunde müsse seinen Kontostand überprüfen oder irgendwelche Details seines Kontos online ändern.

Alle Links in der E-Mail verweisen auf eine gefälschte Seite. Sie sieht genauso aus wie die echte, aber ihre einzige Aufgabe besteht darin, Daten vom ahnungslosen Kunden der Firma zu sammeln. Im Kern haben Sie eine gefälschte E-Mail, die Sie auf eine gefälschte Webseite bringt. Diese einfache »Kette von Lügen« ist bekannt als »Phishing«-Angriff und sie ist lukrativer, als Sie sich vorstellen mögen. Warum betrifft Sie diese »Kette von Lügen«? Manchmal wollen Sie ein gefälschtes Objekt haben, das eine andere gefälschte Komponente zurückgibt (aus einer Methode oder Property) und damit Ihre eigene kleine Kette von Stubs produziert, die mit einem Mock-Objekt tief in den Eingeweiden des Systems endet, sodass Sie letztlich einige Daten während Ihres Tests sammeln können. Sie erzeugen einen Stub, der zu einem Mock-Objekt führt, das die Daten sammelt.

Das Design vieler zu testender Systeme ermöglicht es, komplexe Objektketten zu erzeugen. Es ist nicht ungewöhnlich, Code vorzufinden, der so aussieht:

```
IServiceFactory factory = GetServiceFactory();
IService service = factory.GetService();
```

Oder so:

```
String connstring =
GlobalUtil.Configuration.DBConfiguration.ConnectionString;
```

Nehmen wir an, Sie wollen während eines Tests den Connection String durch einen Ihrer eigenen ersetzen. Sie könnten die Property `Configuration` des Objekts `GlobalUtil` mit einem Stub-Objekt belegen. Danach könnten Sie die Property `DBConfiguration` in diesem Objekt mit einem anderen Stub-Objekt belegen und so weiter, bis Sie schließlich ein Fake-Objekt zurückgeben, das Sie als Mock verwenden, oder einen Stub des Connection-Strings.

Das ist eine leistungsstarke Technik, aber Sie müssen sich selber fragen, ob es nicht besser wäre, den Code zu überarbeiten, um etwa das Folgende zu tun:

```
String connstring = GetConnectionString();
protected virtual String GetConnectionString()
{
    return GlobalUtil.Configuration.DBConfiguration.ConnectionString;
}
```

Sie können dann die virtuelle Methode wie in Kapitel 3 (Abschnitt 3.4.5) gezeigt überschreiben. Das kann den Code lesbarer und leichter wartbar machen und es verlangt nicht das Hinzufügen neuer Interfaces, um mehr Stubs in das System einzufügen.

Hinweis

Ein anderer guter Weg, um Aufrufketten zu vermeiden, ist es, spezielle Wrapper-Klassen um die API zu entwerfen, die es erleichtern, diese zu benutzen und zu testen. Mehr zu dieser Methode erfahren Sie im Buch *Arbeiten mit Legacy Code* von Michael Feathers. Dieses Muster wird im Buch als »Adapt Parameter Pattern« bezeichnet.

Handgeschriebene Mocks und Stubs haben Vorteile, aber sie bringen auch Probleme mit sich. Lassen Sie uns einen Blick darauf werfen.

4.7 Die Probleme mit handgeschriebenen Mocks und Stubs

Es gibt mehrere Aspekte, die auftauchen, wenn Sie manuelle Mocks und Stubs verwenden:

- Es braucht Zeit, um die Mocks und Stubs zu schreiben.
- Es ist schwierig, Stubs und Mocks für Klassen und Interfaces zu schreiben, die viele Methoden, Properties und Events haben.
- Um den Zustand für mehrfache Aufrufe einer Mock-Methode zu speichern, müssen Sie eine Menge Standardcode innerhalb der manuellen Fakes schreiben.
- Wenn Sie verifizieren wollen, dass alle Parameter vom Aufrufer korrekt an eine Methode übergeben wurden, müssen Sie zahlreiche Asserts schreiben. Das ist ein Langweiler.
- Es ist schwierig, den Code von Mocks und Stubs für andere Tests wiederzuverwenden. Der grundlegende Krempel funktioniert, aber sobald Sie es mit mehr als zwei oder drei Funktionen im Interface zu tun haben, fängt es an, mühsam zu werden.
- Kommt es vor, dass ein Fake beides ist, ein Mock und ein Stub? Sehr selten. Damit meine ich vielleicht ein- oder zweimal in einem Projekt. Ich selbst habe das erst ein paar Mal in den letzten Jahren gesehen.

Ein Mock kann auch als Stub dienen, wenn Sie ein Mock-Objekt benötigen, das eine Funktion mit einem Rückgabewert hat. Um den Compiler zufriedenzustellen, müssen Sie auch einen gefälschten Wert vom Fake-Objekt zurückgeben, oder der Code wird nicht laufen (oder nicht einmal kompiliert werden können). In diesem Fall dient Ihr Mock als Stub, und ich vermute, dass Sie von Anfang an das falsche Endresultat testen, wenn Sie von Ihrem Mock-Objekt einen Wert zurück an das System geben. Gewöhnlich schaue ich nach Endresultaten, die Aufrufe in ein Third-Party-System sind und nichts zurückgeben. Ich schaue so oft es geht nach Methoden vom Typ `void`. Manchmal erfordert es das Design des Systems, dass die Methode, die das Drittsystem aufruft, ebenfalls eine Funktion ist (das geschieht häufig in C++, um Fehler anzuzeigen). Das ist der einzige Fall, wo ich irgendeinem Objekt erlaube, gleichzeitig ein Mock und ein Stub zu sein. Hier kommt ein Beispiel:

```
public interface IComNotificationService
{
    int SendNotification(string info);
}
```

Wenn das Endresultat der Unit of Work der Aufruf von `SendNotification` wäre, würden Sie in diesem Code einen Mock verwenden, um zu überprüfen, dass die Methode aufgerufen wird. Aber um den Compiler zufriedenzustellen, müssten Sie ihm auch sagen, dass er einen Wert zurückgeben soll, der Sie aber in diesem Test gar nicht interessiert.

Dies sind inhärente Probleme bei handgeschriebenen Mocks und Stubs. Glücklicherweise gibt es noch andere Wege, um Mocks und Stubs zu erzeugen, wie Sie im nächsten Kapitel sehen werden.

4.8 Zusammenfassung

Dieses Kapitel beschäftigte sich mit der Unterscheidung zwischen Stub- und Mock-Objekten. Ein Mock-Objekt ist wie ein Stub, aber es hilft Ihnen, Asserts auf irgendetwas in Ihrem Test anzuwenden. Ein Stub kann niemals dazu führen, dass Ihr Test fehlschlägt, er ist nur dazu da, unterschiedliche Situationen zu simulieren. Diese Unterscheidung ist wichtig, denn vielen der Mock-Objekt-Frameworks, die Sie im nächsten Kapitel kennenlernen, sind diese Definitionen eingebrannt und Sie müssen wissen, wann Sie was benutzen.

Die Kombination von Stubs und Mocks im gleichen Test ist eine leistungsstarke Technik, aber Sie müssen aufpassen, dass Sie nicht mehr als einen Mock pro Test haben. Die restlichen Fake-Objekte sollten Stubs sein, die zu keinem Fehlschlag Ihres Tests führen können. Das Befolgen dieser Praxis kann zu besser wartbaren Tests führen, die seltener ruiniert werden, wenn sich der interne Code ändert.

Stubs, die andere Stubs oder Mocks produzieren, können eine leistungsstarke Möglichkeit sein, um gefälschte Abhängigkeiten in Ihren Code zu injizieren, die andere Objekte benutzen, um an ihre Daten zu gelangen. Das ist eine großartige Technik in Kombination mit Fabrikklassen und -methoden. Sie können sogar Stubs haben, die andere Stubs zurückgeben, die wiederum andere Stubs zurückgeben usw., aber an irgendeinem Punkt werden Sie sich fragen, ob es all das wert ist. In diesem Fall sollten Sie einen Blick auf die in Kapitel 3 beschriebenen Techniken zur Injektion von Stubs in Ihr Design werfen. (Das nächste Kapitel diskutiert, wie es Ihnen einige Isolation-Frameworks ermöglichen, ganze Ketten von Fake-Aufrufen mit einer Zeile Code zu erzeugen – rekursive Fakes zur Rettung!)

Eines der häufigsten Probleme, auf das die Leute beim Schreiben von Tests treffen, ist, dass sie zu häufig Mocks in ihren Tests verwenden (Überspezifizierung). Sie sollten nur selten Aufrufe von Fake-Objekten verifizieren, die sowohl als Stubs als auch als Mocks im gleichen Test verwendet werden. (Das ist wohl ein seltener Grenzfall. Sie verifizieren, dass eine Funktion aufgerufen wurde. Weil es eine Funktion ist, muss sie irgendeinen Wert zurückgeben, und weil Sie diese Methode fälschen, müssen Sie dem Test sagen, was der Wert sein soll. Dieser Wert ist der Teil im Test, der ein Stub ist, denn er hat nichts mit der Entscheidung – also dem Assert – zu tun, ob der Test fehlschlägt oder erfolgreich durchläuft.) Wenn Sie die Begriffe »verifizieren« und »Stub« für die gleiche Variable im gleichen Test verwenden, dann überspezifizieren Sie ihn wahrscheinlich, was ihn eher spröde macht.

Sie können mehrere Stubs in einem Test haben, weil eine Klasse mehrere Abhängigkeiten haben kann. Sorgen Sie aber dafür, dass Ihr Test lesbar bleibt. Strukturieren Sie Ihren Code so gut, dass der Leser des Tests versteht, was da vor sich geht.

Sie mögen es unbequem finden, für große Interfaces oder komplizierte, interaktionstestende Szenarien Mocks und Stubs von Hand zu schreiben. Das ist es wirklich und es gibt bessere Wege, dies zu tun, wie Sie im nächsten Kapitel sehen werden. Aber es wird Ihnen auch häufig begegnen, dass handgeschriebene Mocks und Stubs Frameworks im Hinblick auf Einfachheit und Lesbarkeit schlagen. Die Kunst liegt darin, wann Sie welches Tool benutzen.

Das nächste Kapitel beschäftigt sich mit Isolation-(Mocking-)Frameworks, die Ihnen erlauben, Stub- oder Mock-Objekte automatisch zur Laufzeit zu erzeugen und mit der gleichen Leistungsfähigkeit – wenn nicht mit einer viel größeren – zu benutzen wie handgeschriebene Mocks und Stubs.

Kapitel 5

Isolation-(Mock-Objekt-)Frameworks

Dieses Kapitel behandelt

- eine Einführung in die Isolation-Frameworks
- die Verwendung von NSubstitute, um Stubs und Mocks zu erzeugen
- einen Überblick über fortgeschrittene Use Cases für Mock und Stubs
- die Vermeidung von verbreiteten Zweckentfremdungen der Isolation-Frameworks

Im vorangegangenen Kapitel haben wir uns das manuelle Schreiben von Mocks und Stubs angeschaut und wir haben die damit verbundenen Herausforderungen gesehen. In diesem Kapitel schauen wir uns einige elegante Lösungen für diese Probleme in Form eines *Isolation-Frameworks* an – eine wiederverwendbare Bibliothek, die Fake-Objekte *zur Laufzeit* erzeugen und konfigurieren kann. Diese Objekte werden als »dynamische Stubs« und »dynamische Mocks« bezeichnet.

Wir beginnen mit einem Überblick über Isolation-Frameworks (oder auch Mocking-Frameworks – das Wort »Mock« ist schon zu überladen) und dem, was sie tun können. Ich nenne sie Isolation-Frameworks, weil Sie Ihnen erlauben, die Unit of Work von ihren Abhängigkeiten zu isolieren. Wir werden uns ein spezifisches Framework genauer anschauen: NSubstitute. Sie werden sehen, wie Sie es verwenden können, um unterschiedliche Dinge zu testen und um Stubs, Mocks und andere spannende Sachen zu erzeugen.

Aber NSubstitute (oder einfach NSub) ist hier nicht der springende Punkt. Während Sie NSub verwenden, werden Sie sehen, welche spezifischen Werte seine API in Ihren Tests unterstützt (Lesbarkeit, Wartbarkeit, robuste und langlebige Tests und Weiteres) und Sie werden herausfinden, was ein Isolation-Framework gut macht oder was im Gegenteil seine Nachteile für Ihre Tests sein können.

Daher werde ich später in diesem Kapitel NSub andere Frameworks für .NET-Entwickler gegenüberstellen, ihr API-Design vergleichen und wie sie die Test-Lesbarkeit, -Wartbarkeit und -Robustheit berühren. Ich werde schließlich mit einer Liste von Dingen enden, auf die Sie achten müssen, wenn Sie solche Frameworks in Ihren Tests einsetzen.

Lassen Sie uns vorne anfangen: Was sind Isolation-Frameworks?

5.1 Warum überhaupt Isolation-Frameworks?

Ich werde mit einer grundlegenden Definition starten, die vielleicht ein wenig nichtssagend klingt, aber sie soll recht allgemein bleiben, um den verschiedenen Isolation-Frameworks, die es auf dem Markt gibt, gerecht zu werden.

Definition

Ein *Isolation-Framework* ist ein Satz von programmierbaren APIs, die die Erzeugung von Fake-Objekten wesentlich einfacher, schneller und kürzer machen, als dies von Hand möglich ist.

Gut konstruierte Isolation-Frameworks können den Entwickler von der Notwendigkeit befreien, sich wiederholenden Code zu schreiben, der Objekt-Interaktionen prüft oder simuliert. Und wenn sie *sehr* gut konstruiert sind, können sie dazu beitragen, dass Tests viele Jahre überdauern, ohne dass der Entwickler zurückkommen und sie bei jeder kleinen Änderung des Produktionscodes anpassen muss.

Isolation-Frameworks gibt es für die meisten Sprachen, für die auch ein Unit-Testing-Framework existiert. Beispielsweise hat C++ »mockpp« und andere Frameworks und Java hat unter anderem »jMock« und »PowerMock«. Mehrere bekannte Frameworks wie »Moq«, »FakeItEasy«, »NSubstitute«, »Typemock Isolator« und »Just Mock« unterstützen .NET. Es gibt noch weitere Isolation-Frameworks, die ich nicht mehr verwende oder unterrichte, weil sie entweder zu alt oder zu mühsam sind, oder ihnen fehlen zahlreiche Features, die die neueren Frameworks eingeführt haben. Dazu gehören »RhinoMocks«, »NMock«, »EasyMock«, »NUnit.Mocks« und »Moles«. In Visual Studio 2012 wurde Moles integriert und in Microsoft Fakes umbenannt – ich lasse aber trotzdem die Finger davon. Mehr dazu und den anderen Tools finden Sie im Anhang.

Die Verwendung von Isolation-Frameworks statt handgeschriebener Mocks und Stubs wie in den vorangegangenen Kapiteln hat mehrere Vorteile, die die Entwicklung von eleganteren und komplexeren Tests einfacher, schneller und weniger fehleranfällig machen.

Der beste Weg, den Wert eines Isolation-Frameworks zu verstehen, ist, ein Problem und seine Lösung zu sehen. Ein Problem, das bei der Verwendung von handgeschriebenen Mocks und Stubs auftreten kann, ist sich wiederholender Code.

Nehmen Sie an, Sie haben ein Interface, das ein wenig komplizierter als die bisherigen ist:

```
public interface IComplicatedInterface
{
    void Method1 (string a, string b, bool c, int x, object o);
    void Method2 (string b, bool c, int x, object o);
    void Method3 (bool c, int x, object o);
}
```

Das Erzeugen eines handgeschriebenen Stubs oder Mocks für dieses Interface mag zeitaufwendig sein, denn Sie müssen sich die Parameter für jede Methode merken, wie dieses Listing zeigt.

```
class MytestableComplicatedInterface:IComplicatedInterface
{
    //mühsam handgeschriebene Statements:
    public string meth1_a;
    public string meth1_b,meth2_b;
    public bool meth1_c,meth2_c,meth3_c;
```

```
    public int meth1_x,meth2_x,meth3_x;
    public int meth1_0,meth2_0,meth3_0;

    public void Method1(string a, string b, bool c, int x, object o)
    {
        meth1_a = a;
        meth1_b = b;
        meth1_c = c;
        meth1_x = x;
        meth1_0 = 0;
    }

    public void Method2(string b, bool c, int x, object o)
    {
        meth2_b = b;
        meth2_c = c;
        meth2_x = x;
        meth2_0 = 0;
    }

    public void Method3(bool c, int x, object o)
    {
        meth3_c = c;
        meth3_x = x;
        meth3_0 = 0;
    }
}
```

Listing 5.1: Das Implementieren komplizierter Interfaces mit handgeschriebenen Stubs

Das ist nicht nur zeitaufwendig und mühselig zu schreiben – was passiert eigentlich, wenn Sie eine Methode testen wollen, die mehrfach aufgerufen wird? (Erinnern Sie sich an Kapitel 4, wo ich den Begriff *Fake* für etwas eingeführt habe, das nur so aussieht wie die echte Sache, ohne es zu sein. Je nachdem wie es verwendet wird, handelt es sich um einen Mock oder Stub.) Oder wenn Sie einen bestimmten Wert in Abhängigkeit von den angegebenen Parametern zurückgeben oder alle Werte für alle Aufrufe der gleichen Methode (die Parameterhistorie) aufzeichnen wollen? Der Code wird schnell hässlich.

Durch die Verwendung eines Isolation-Frameworks wird der Code hierfür trivial, lesbar und viel kürzer, wie Sie sehen werden, wenn Sie Ihr erstes dynamisches Mock-Objekt anlegen.

5.2 Das dynamische Erzeugen eines Fake-Objekts

Lassen Sie uns *dynamische Fake-Objekte* definieren und wie sie sich von regulären, handgeschriebenen Fakes unterscheiden.

Definition

Ein *dynamisches Fake-Objekt* ist jeder Stub oder Mock, der zur Laufzeit erzeugt wird, ohne dass die Notwendigkeit besteht, eine handgeschriebene (hardcoded) Implementierung dieses Objekts zu verwenden.

Die Verwendung von dynamischen Fakes macht es überflüssig, handcodierte Klassen, die Interfaces implementieren oder von anderen Klassen ableiten, zu verwenden, denn die benötigten Klassen können vom Entwickler zur Laufzeit im Arbeitsspeicher und mit ein paar einfachen Code-Zeilen erzeugt werden.

Als Nächstes werden wir uns NSubstitute anschauen und sehen, wie es Ihnen dabei helfen kann, einige der gerade diskutierten Probleme zu überwinden.

5.2.1 Die Einführung von NSubstitute in Ihre Tests

In diesem Kapitel werde ich NSubstitute (`http://nsubstitute.github.com`) benutzen, ein Open-Source-Isolation-Framework, das frei heruntergeladen und über NuGet (`http://nuget.org`) installiert werden kann. Es war schwierig für mich, zu entscheiden, ob ich NSubstitute oder FakeItEasy benutzen wollte. Beide sind großartig und Sie sollten sich beide anschauen, bevor Sie sich für eines entscheiden. Ein Vergleich von Frameworks folgt im nächsten Kapitel und im Anhang, aber ich wähle NSubstitute, weil es die bessere Dokumentation hat und die meisten Werte unterstützt, die ein gutes Framework unterstützen sollte. Diese Werte werden im nächsten Kapitel aufgelistet.

Um der Kürze willen (und um mir das Schreiben zu erleichtern) werde ich von nun an NSubstitute nur noch als NSub bezeichnen. Es ist einfach und schnell zu verwenden, mit wenig Overhead beim Kennenlernen der API. Ich werde Sie durch ein paar Beispiele führen und Sie können sehen, wie die Verwendung eines Frameworks Ihr Leben als Entwickler vereinfacht (manchmal). Im nächsten Kapitel werde ich noch tiefer in einige »Meta«-Themen zu Isolation-Frameworks einsteigen, damit Sie verstehen, wie sie arbeiten und warum einige Frameworks Dinge können, die andere nicht können. Aber zunächst zurück an die Arbeit.

Um mit dem Experimentieren zu beginnen, erzeugen Sie eine Klassen-Bibliothek, die als Ihr Unit-Test-Projekt dienen wird, und fügen eine Referenz auf NSub hinzu, indem Sie es über NuGet installieren (wählen Sie Extras|NuGet-Paket-Manager| Paket-Manager-Console|Install-Package NSubstitute aus).

NSub unterstützt das *Arrange-Act-Assert-Modell*, das mit der Art, wie Sie bisher Tests geschrieben und ausgeführt haben, konsistent ist. Die Idee ist, Fakes zu erzeugen und sie im *Arrange*-Teil des Tests zu konfigurieren, im *Act*-Teil auf das zu testende Produkt anzuwenden und im *Assert*-Teil am Ende zu verifizieren, dass ein Fake aufgerufen wurde.

NSub enthält eine Klasse namens `Substitute`, die Sie für das Erzeugen von Fakes zur Laufzeit verwenden. Diese Klasse hat eine Methode mit einer generischen und einer nichtgenerischen Geschmacksrichtung namens `For(type)`. Dies ist der bevorzugte Weg, um ein Fake-Objekt in Ihre Anwendung einzubringen, wenn Sie mit NSub arbeiten. Sie rufen diese Methode mit dem Typ auf, von dem Sie eine gefälschte Instanz erzeugen wollen. Diese Methode erzeugt dann *dynamisch* ein Fake-Objekt, das zur Laufzeit an diesem Typ oder Interface anhaftet, und gibt es zurück. Sie brauchen dieses neue Objekt nicht mit echtem Code zu implementieren.

Weil NSub ein eingeschränktes (constrained) Framework ist, arbeitet es am besten mit Interfaces. Bei echten Klassen arbeitet es nur mit solchen, die nicht sealed sind, und für diese kann es auch nur virtuelle Methoden fälschen.

5.2.2 Das Ersetzen eines handgeschriebenen Fake-Objekts durch ein dynamisches

Lassen Sie uns einen Blick auf ein handgeschriebenes Fake-Objekt werfen, das verwendet wird, um zu überprüfen, dass der Protokollaufruf korrekt durchgeführt wurde. Das folgende Listing zeigt die Testklasse und den handgeschriebenen Fake, den Sie erzeugen würden, wenn Sie kein Isolation-Framework verwenden würden.

```
[TestFixture]
class LogAnalyzerTests
{
    [Test]
    public void Analyze_TooShortFileName_CallLogger()
    {
        //das Erzeugen des Fakes:
        FakeLogger logger = new FakeLogger ();

        LogAnalyzer analyzer = new LogAnalyzer(logger);

        analyzer.MinNameLength = 6;
        analyzer.Analyze("a.txt";);

        //die Verwendung des Fakes als Mock-Objekt durch
        //Anwendung eines Assert:
        stringAssert.Contains("zu kurz",logger.LastError);
    }
}

class FakeLogger: ILogger
{
    public string LastError;

    public void LogError(string message)
    {
        LastError = message;
    }
}
```

Listing 5.2: Das Assert für ein handgeschriebenes Fake-Objekt

Die fett gedruckten Teile des Codes sind die Teile, die sich ändern werden, wenn Sie beginnen, dynamische Mocks und Stubs zu verwenden.

Sie legen nun ein dynamisches Mock-Objekt an und ersetzen schließlich den früheren Test. Das nächste Listing zeigt, wie einfach es ist, ILogger zu fälschen und zu verifizieren, dass es mit einem String aufgerufen wurde.

```
[Test]
public void Analyze_TooShortFileName_CallLogger()
{
    //erzeuge dynamisches Mock-Objekt, auf das Sie am Ende des Tests
    //ein Assert anwenden (1):
    ILogger logger = substitute.For<ILogger>;
    LogAnalyzer analyzer = new LogAnalyzer(logger);

    analyzer.MinNameLength = 6;
    analyzer.Analyze("a.txt";);

    //setze Erwartungen unter Verwendung der API vonNSub (2):
    logger.Received().LogError("Dateiname zu kurz: a.txt");
    }
}
```

Listing 5.3: Das Fälschen eines Objekts mit NSub

Ein paar Zeilen Code befreien Sie von der Notwendigkeit, einen handgeschriebenen Stub oder Mock zu verwenden, denn sie erzeugen ihn dynamisch **(1)**. Die gefälschte `ILogger`-Objektinstanz ist ein dynamisch generiertes Objekt, das das Interface `ILogger` implementiert, aber es gibt keine Implementation innerhalb der `ILogger`-Methoden.

Von diesem Moment an bis zur letzten Zeile des Tests werden alle Aufrufe für das gefälschte Objekt automatisch aufgezeichnet oder für die spätere Verwendung gespeichert, so wie in der letzten Zeile des Tests **(2)**.

Statt des traditionellen Assert-Aufrufs verwenden Sie in der letzten Zeile eine spezielle API – eine Erweiterungs-Methode, die vom Namespace von NSub bereitgestellt wird. `ILogger` hat keine Methode mit Namen `Received()` in seinem Interface. Diese Methode ist Ihre Art, zu überprüfen, dass ein Methodenaufruf auf Ihr Fake-Objekt ausgeführt wurde (und es damit dem Begriff nach zu einem Mock-Objekt wird).

Die Art, wie die Methode `Received()` funktioniert, wirkt nahezu magisch. Sie gibt den gleichen Typ des Objekts zurück, für das sie aufgerufen wurde, aber tatsächlich wird sie verwendet, um festzulegen, worauf geprüft wird.

Wenn Sie in der letzten Zeile des Tests einfach geschrieben hätten

```
logger.LogError("Dateiname zu kurz: a.txt");
```

dann würde Ihr gefälschtes Objekt diesen Methodenaufruf so behandeln, als würde er während des Produktionslaufs ausgeführt werden, und einfach gar nichts machen, falls nicht eine bestimmte Aktion für die Methode `LogError()` konfiguriert wurde.

Indem `Received()` unmittelbar vor `LogError()` aufgerufen wird, lassen Sie NSub wissen, dass Sie in Wirklichkeit das gefälschte Objekt danach fragen, ob die Methode aufgerufen wird oder nicht. Wenn sie nicht aufgerufen wurde, dann erwarten Sie, dass eine Ausnahme

von der letzten Zeile dieses Tests geworfen wird. Für den Leser verbessern sie die Lesbarkeit und geben ihm den Hinweis: »Irgendetwas *erhielt* (*received*) einen Methodenaufruf, oder dieser Test wäre fehlgeschlagen«.

Wenn die Methode `LogError` nicht aufgerufen wird, dann können Sie einen Fehler mit einer Nachricht, die der folgenden in Ihrem Test-Log nahe kommt, erwarten:

```
NSubstitute.Exceptions.ReceivedCallsException : Expected to receive
a call matching:
    LogError("Dateiname zu kurz: a.txt");
Actually received no matching calls.
```

Arrange-Act-Assert

Beachten Sie bitte, wie gut die Art und Weise, in der Sie das Isolation-Framework benutzen, zur Struktur von Arrange-Act-Assert passt. Sie beginnen mit dem Arrangieren eines Fake-Objekts, Sie agieren mit der Sache, die Sie testen, und dann überprüfen (assert) Sie auf etwas am Ende des Tests.

Allerdings war das nicht immer so einfach.

In den frühen Tagen (so um 2006) haben die meisten Open-Source-Isolation-Frameworks die Idee von Arrange-Act-Assert nicht unterstützt und stattdessen ein Konzept namens Record-Replay unterstützt.

Record-Replay war ein hässlicher Mechanismus, bei dem Sie der Isolation-API zunächst mitteilen mussten, dass das Fake-Objekt im Aufnahme-Modus (Record Mode) war und anschließend mussten Sie die Methoden des Objekts so aufrufen, wie Sie erwarteten, dass sie vom Produktionscode aufgerufen werden.

Danach mussten Sie die Isolation-API anweisen, in den Wiedergabe-Modus (Replay Mode) zu wechseln und erst *dann* konnten Sie Ihr Fake-Objekt auf Ihren Produktions-Code loslassen.

Ein Beispiel können Sie im Google Testing Blog sehen:
`http://googletesting.blogspot.no/2009/01/tott-use-easymock.html`.

Asserts involvierten bei der Verwendung dieser Tests gewöhnlich einen einfachen Aufruf der Methoden `verify()` oder `verifyAll()` der Isolation-API, wobei der arme Leser des Tests zurückgehen und herausfinden musste, was tatsächlich erwartet wurde.

Verglichen mit den heutigen Möglichkeiten, Tests zu schreiben, die das bei Weitem lesbarere Arrange-Act-Assert-Modell unterstützen, hat diese Tragödie viele Entwickler in der Summe Millionen von Stunden mühevolles Testlesen gekostet, nur um herauszubekommen, wo genau der Test fehlgeschlagen ist.

Wenn Sie die erste Auflage dieses Buches besitzen, können Sie ein Beispiel von Record-Replay in diesem Kapitel bei der Vorstellung von Rhino Mocks sehen. Ah, gute alte Zeiten! Nun lasse ich die Hände von Rhino Mocks, teilweise, weil seine API nicht so gut wie die neuerer Frameworks ist, und teilweise, weil die Wartung durch Oren Eini (`http://Ayende.com`) fraglich ist. Es scheint so, dass Oren, der in vieler Hinsicht als super Programmierer bekannt ist, doch noch ein anderes Leben und geheiratet hat, weshalb er schließlich doch anfangen musste, unter den Schlachten, die er schlagen will, auszuwählen. Rhino Mocks scheint eine der Schlachten zu sein, die er nicht mehr schlagen will.

Nachdem Sie gesehen haben, wie man Fakes als Mocks verwendet, lassen Sie uns nun sehen, wie man sie als Stubs, die Werte im zu testenden System simulieren, verwendet.

5.3 Die Simulation von Fake-Werten

Das nächste Listing zeigt, wie man den Wert eines Fake-Objekts zurückgeben kann, wenn die Methode des Interface einen Rückgabewert hat, der nicht `void` ist. Für dieses Beispiel fügen Sie ein IFileNameRules Interface in das System ein (siehe `NSubBasics.cs` im Quellcode zum Buch).

```
[Test]
public void Returns_ByDefault_WorksForHardCodeArgument()
{
    IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

    //zwingt den Methodenaufruf einen Fake-Wert zurückzugeben:
    fakeRules.IsValidLogFileName("strict.txt").Returns(true);

    Assert.IsTrue(fakeRules.IsValidLogFileName("strict.txt"));
}
```

Listing 5.4: Die Rückgabe eines Wertes von einem Fake-Objekt

Was ist, wenn Sie das Argument überhaupt nicht interessiert? Es wäre im Sinne der Wartbarkeit sicherlich besser, wenn Sie *immer* einen Fake-Wert – egal welchen – zurückgeben würden, denn dann müssen Sie sich keine Gedanken um interne Produktionscode-Änderungen machen und Ihr Test würde immer noch erfolgreich durchlaufen, auch wenn der Produktionscode die Methode mehrfach aufruft. Es würde auch der Lesbarkeit dienen, denn momentan weiß der Leser des Tests nicht, ob der Dateiname wichtig ist. Sie können seinen Tag verbessern, indem Sie aus seiner Lektüre nicht benötigte Informationen entfernen, damit er es mit Ihrem Code leichter hat.

Lassen Sie uns nun einen »Argument Matcher« verwenden:

```
[Test]
public void Returns_ByDefault_WorksForHardCodeArgument()
{
    IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

    //ignoriere den Wert des Arguments:
    fakeRules.IsValidLogFileName(Arg.Any<String>()).Returns(true);

    Assert.IsTrue(fakeRules.IsValidLogFileName("anything.txt"));
}
```

Beachten Sie, wie Sie die `Arg`-Klasse verwenden, um deutlich zu machen, dass Sie sich nicht für den Input interessieren, der notwendig ist, damit der gefälschte Wert zurückgegeben wird. Dies wird »Argument Matcher« genannt und in Isolation-Frameworks häufig verwendet, um zu steuern, wie Argumente behandelt werden, eines nach dem anderen.

Und wie würden Sie eine Ausnahme simulieren? Hier kommt, wie Sie das mit NSub machen würden:

```
[Test]
public void Returns_ArgAny_Throws()
{
    IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

    //hier wird ein Lambda-Ausdruck benötigt:
    fakeRules.When(x => x.IsValidLogFileName(Arg.Any<string>()))
             .Do(context =>
              {throw new Exception("fake exception");});

    Assert.Throws<Exception>(() =>
        fakeRules.IsValidLogFileName("anything"));
}
```

Beachten Sie, wie Sie `Assert.Throws` verwenden, um zu überprüfen, dass eine Ausnahme tatsächlich geworfen wird.

Ich finde die Syntax-Bänder, die Ihnen NSub hier anlegt, nicht besonders toll. (Das Ganze hier wäre mit FakeItEasy einfacher, aber NSub hat die ausführlichere Dokumentation, weshalb ich es hier verwende.)

Beachten Sie, dass Sie einen Lambda-Ausdruck verwenden müssen. Im Aufruf der `When`-Methode zeigt das `x`-Argument das Fake-Objekt an, dessen Verhalten Sie ändern. Im `Do`-Aufruf beachten Sie das CallInfo-`context`-Argument. Zur Laufzeit enthält das `context`-Argument Werte und erlaubt Ihnen, wundervolle Dinge zu tun, aber für dieses Beispiel brauchen Sie es nicht.

Da Sie nun wissen, wie Dinge simuliert werden, lassen Sie uns die Sache etwas realistischer machen und sehen, wohin uns das führt.

5.3.1 Ein Mock, ein Stub und ein Ausflug in einen Test

Lassen Sie uns im gleichen Szenario zwei Arten von Fake-Objekten kombinieren. Eines wird als Stub und das andere als Mock verwendet.

Sie verwenden Analyzer2 im Quellcode des Buches unter Kapitel 5. Es ähnelt dem Beispiel aus Listing 4.2 in Kapitel 4, wo ich über `LogAnalyzer` und die Verwendung der Klassen `MailSender` und `WebService` geschrieben habe, aber diesmal ist die Anforderung so, dass der Webservice benachrichtigt wird, wenn der Logger eine Ausnahme wirft. Dies wird in Abbildung 5.1 dargestellt.

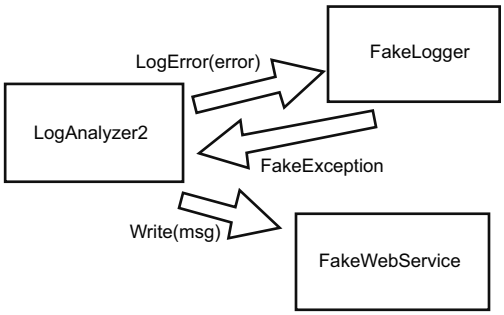


Abb. 5.1: Der Logger wird durch einen Stub ersetzt, um eine Ausnahme zu simulieren, und ein gefälschter Webservice wird als Mock verwendet, um zu überprüfen, ob er korrekt aufgerufen wurde. Der ganze Test dreht sich darum, wie `LogAnalyzer2` mit anderen Objekten interagiert.

Sie wollen, dass `LogAnalyzer2` `WebService` über das Problem informiert, wenn der Logger eine Ausnahme wirft.

Das nächste Listing zeigt, wie die Logik mit den Tests aussieht.

```
    [Test]
    //der Test:
    public void Analyze_LoggerThrows_CallsWebService()
    {
        FakeWebService mockWebService = new FakeWebService();

        FakeLogger2 stubLogger = new FakeLogger2();
        stubLogger.WillThrow = new Exception("fake exception");

        var analyzer2 = new LogAnalyzer2(stubLogger, mockWebService);
        analyzer2.MinNameLength = 8;

        string tooShortFileName="abc.ext";
        analyzer2.Analyze(tooShortFileName);

        Assert.That(mockWebService.MessageToWebService,
                        Is.StringContaining("fake exception"));
    }
}

//dieser gefälschte Web Service wird als Mock verwendet werden:
public class FakeWebService:IWebService
{
    public string MessageToWebService;

    public void Write(string message)
```

```
    {
        MessageToWebService = message;
    }
}

//dieser gefälschte Logger wird als Stub verwendet werden:
public class FakeLogger2:ILogger
{
    public Exception WillThrow = null;
    public string LoggerGotMessage = null;

    public void LogError(string message)
    {
        LoggerGotMessage = message;
        if (WillThrow != null)
        {
            throw WillThrow;
        }
    }
}

//---------- PRODUCTION CODE
//die zu testende Klasse:
public class LogAnalyzer2
{
    private ILogger _logger;
    private IWebService _webService;

    public LogAnalyzer2(ILogger logger,IWebService webService)
    {
        _logger = logger;
         _webService = webService;
    }
}

public int MinNameLength { get; set; }

public void Analyze(string filename);
{
    if (filename.Length<MinNameLength)
    try
    {
        _logger.LogError(
        string.Format("Dateiname zu kurz: {0}",filename));
    }
```

```
            catch (Exception e)
            {
                _webService.Write("Fehler vom Logger: " + e);
            }
        }
    }
}

public interface IWebService
{
    void Write(string message);
}
```

Listing 5.5: Die zu testende Methode und ein Test mit handgeschriebenen Mocks und Stubs

Das nächste Listing zeigt, wie der Test aussehen könnte, wenn Sie NSubstitute verwenden würden.

```
[Test]
public void Analyze_LoggerThrows_CallsWebService()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    //simuliert bei jedem Input eine Ausnahme:
    stubLogger.When(
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});

    var analyzer = new LogAnalyzer2(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

    //überprüft, ob der Mock Web Service mit einem String,
    //der "fake exception" enthält, aufgerufen wird:
    mockWebService.Received()
      .Write(Arg.Is<string>(s => s.Contains("fake exception")));
}
```

Listing 5.6: Der vorangegangene Test wurde so abgeändert, dass er NSubstitute verwendet.

Das Schöne an diesem Test ist, dass er keine handgeschriebenen Fakes benötigt, aber beachten Sie auch, wie er bereits seinen Tribut im Hinblick auf die Lesbarkeit fordert. Diese Lambdas sind für meinen Geschmack nicht sehr lesefreundlich, aber sie gehören zu den kleinen Übeln, mit denen Sie in C# zu leben lernen müssen, denn diese sind es, die es Ihnen erlauben, auf Strings für Methodennamen zu verzichten. Das erleichtert ein späteres Refactoring, wenn einmal ein Methodenname geändert wird.

Beachten Sie auch, dass Argument-Matching-Beschränkungen sowohl im Simulations-Teil, wo Sie den Stub konfigurieren, als auch im Assert-Teil, wo Sie überprüfen, ob der Mock aufgerufen wurde, verwendet werden können.

Es gibt in NSubstitute eine ganze Reihe von möglichen Argument-Matching-Beschränkungen und die Webseite bietet einen netten Überblick. Weil dieses Buch nicht als Anleitung für NSub gedacht ist (daher hat Gott schließlich die Online-Dokumentation erschaffen), sollten Sie auf `http://nsubstitute.github.com/help/argument-matchers` gehen, wenn Sie daran interessiert sind, mehr über diese API herauszufinden.

Der Vergleich zwischen Objekten und Properties

Was geschieht, wenn Sie ein Objekt mit bestimmten Properties als Argument erwarten? Wenn Sie beispielsweise ein `ErrorInfo`-Objekt mit den Properties `Severity` und `Message` im Aufruf von `webservice.Write` angeben?

```
[Test]
public void Analyze_LoggerThrows_CallsWebServiceWithNSubObject()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    stubLogger.When(
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});

    var analyzer = new LogAnalyzer3(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

    mockWebService.Received()
    //stark typisierter argument matcher zum von Ihnen erwarteten
    //Objekttypen:
        .Write(Arg.Is<ErrorInfo>(info => info.Severity == 1000
    //einfaches C# "and" für etwas komplexere Erwartung:
            && info.Message.Contains("fake exception")));
}
```

Beachten Sie, wie Sie ganz einfach 08/15-C# verwenden können, um verbundene Matcher für das gleiche Argument zu erzeugen. Sie wollen, dass `info`, das Sie als Argument angeben, eine bestimmte `Severity` und eine bestimmte `Message` besitzt.

Beachten Sie auch, wie das die Lesbarkeit beeinflusst. Ich denke, als Faustformel kann ich sagen, je mehr ich ein Isolation-Framework benutze, umso weniger lesbar wird der Code, der dabei herauskommt. Aber dennoch ist manchmal die Verwendung akzeptabel. Dies hier wäre wohl ein Grenzfall. Wenn ich beispielsweise einen Fall mit mehr als einem Lambda-Ausdruck in einem `Assert` habe, dann frage ich mich, ob nicht ein handgeschriebener Fake besser lesbar wäre.

Aber wenn Sie dabei sind, die Dinge auf die einfachste Art zu testen, dann könnten Sie zwei Objekte vergleichen und einfach die Lesbarkeit testen. Sie könnten ein erwartetes Objekt mit all seinen erwarteten Properties erzeugen und mit dem tatsächlich angegebenen Objekt vergleichen, wie im Folgenden gezeigt wird.

```
[Test]
public void
Analyze_LoggerThrows_CallsWebServiceWithNSubObjectCompare()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    stubLogger.When(
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});
    var analyzer = new LogAnalyzer3(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

    //erzeuge das Objekt, das erwartet wird:
    var expected = new ErrorInfo(1000, "fake exception");
    //überprüfe, dass exakt das gleiche Objekt erhalten wurde
    //(im Wesentlichen assert.equals()):
    mockWebService.Received().Write(expected);
}
```

Listing 5.7: Der Vergleich ganzer Objekte

Das Testen ganzer Objekte funktioniert nur, wenn das Folgende wahr ist:

- Es ist einfach, das Objekt mit den erwarteten Properties zu erzeugen.
- Sie wollen *alle* Properties des fraglichen Objekts testen.
- Sie kennen vollständig die exakten Werte jeder einzelnen Property.
- Die Methode `Equals()` ist für die beiden zu vergleichenden Objekte korrekt implementiert. (Gewöhnlich ist es eine schlechte Gepflogenheit, sich auf die Standardimplementierung von `object.Equals()` zu verlassen. Wenn `Equals()` nicht implementiert ist, wird dieser Test immer fehlschlagen, denn der voreingestellte Rückgabewert von `Equals()` ist `false`.)

Noch eine Bemerkung zur Robustheit des Tests: Bei Verwendung dieser Technik können Sie keine Argument-Matcher verwenden, um zu fragen, ob ein String einen bestimmten Wert in einer der Properties enthält, wodurch Ihre Tests ein klein wenig weniger robust gegen Änderungen in der Zukunft sein werden.

Auch wird Ihr Test jedes Mal fehlschlagen, wenn sich ein String in einer erwarteten Property in der Zukunft ändert, und sei es nur durch ein extra Leerzeichen am Anfang oder Ende. Sie müssen dann jedes Mal Ihren Test auf diesen neuen String anpassen. Die Kunst besteht hier darin zu entscheiden, wie viel Lesbarkeit sie zugunsten der Robustheit auf

lange Sicht opfern wollen. Für mich ist es grenzwertig akzeptabel, vielleicht nicht das ganze Objekt, sondern nur einige Properties mit Argument Matchern zu testen, um langfristig mehr Robustheit zu bekommen. Ich *hasse* es, Tests aus den falschen Gründen zu ändern.

5.4 Das Testen auf ereignisbezogene Aktivitäten

Events sind eine Straße mit zwei Fahrbahnen und Sie können sie in zwei verschiedene Richtungen testen:

- Der Test, dass jemand auf das Event hört
- Der Test, dass jemand das Event triggert

5.4.1 Das Testen eines Event Listeners

Das erste Szenario für einen Test, das wir angehen werden, habe ich schon viele Entwickler schlecht implementieren sehen: nämlich zu überprüfen, ob ein Objekt für ein Event eines anderen Objekts registriert ist.

Viele Entwickler wählen die weniger wartbare und überspezifizierte Art der Überprüfung, ob der interne Zustand eines Objekts registriert wurde, um das Event eines anderen Objekts zu erhalten.

Diese Implementierung ist etwas, das ich für echte Tests nicht empfehlen würde. Das Registrieren für ein Event ist das interne, private Verhalten des Codes. Es tut nichts als Endresultat, außer den Zustand im System zu wechseln, wodurch sich dieses anders verhält.

Es ist besser, diesen Test so zu implementieren, dass auf das Listener-Objekt geachtet wird, das als Antwort auf das ausgelöste Event irgendetwas tun muss. Wenn der Listener nicht für das Event registriert wurde, dann wird es kein öffentlich sichtbares Verhalten geben, wie im folgenden Listing dargestellt.

```
class Presenter
{
    private readonly IView _view;

    public Presenter(IView view)
    {
        _view = view;
        this._view.Loaded += OnLoaded;
    }

    private void OnLoaded()
    {
        _view.Render("Hello World");
    }
}

public interface IView
{
```

```
    event Action Loaded;
    void Render(string text);
}

//------ TESTS
[TestFixture]
public class EventRelatedTests
{
    [Test]
    public void ctor_WhenViewIsLoaded_CallsViewRender()
    {
        var mockView = Substitute.For<IView>();

        Presenter p = new Presenter(mockView);
        //trigger das Event mit NSubstitute:
        mockView.Loaded += Raise.Event<Action>();

        mockView.Received()
        //überprüfe, dass view aufgerufen wurde:
          .Render(Arg.Is<string>(s => s.Contains("Hello World")));
    }
}
```

Listing 5.8: Event-bezogener Code und wie er getriggert wird

Beachten Sie das Folgende:

- Der Mock ist auch ein Stub (Sie simulieren ein Event).
- Um ein Event zu triggern, müssen Sie es im Test ungelenk registrieren. Das dient nur dazu, den Compiler zufriedenzustellen, denn Event-bezogene Properties werden anders behandelt und vom Compiler streng überwacht. Events können nur direkt von der Klasse/Struktur, in der sie deklariert wurden, aufgerufen werden.

Hier kommt ein anderes Szenario, bei dem Sie zwei Abhängigkeiten haben: einen Logger und einen View. Das folgende Listing zeigt einen Test, der überprüft, ob Presenter in ein Log schreibt, wenn es ein Fehler-Event von Ihrem Stub erhält.

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var stubView = Substitute.For<IView>();
    var mockLogger = Substitute.For<ILogger>();

    Presenter p = new Presenter(stubView, mockLogger);
    stubView.ErrorOccured +=
    //simuliere den Fehler (1):
      Raise.Event<Action<string>>("fake error");
```

```
    mockLogger.Received()
    //verwende Mock um Log-Aufruf zu prüfen (2):
      .LogError(Arg.Is<string>(s => s.Contains("fake error")));
}
```

Listing 5.9: Die Simulation eines Events mitsamt einem separaten Mock

Beachten Sie, dass Sie einen Stub verwenden, um das Event zu triggern, und einen Mock, um zu prüfen, dass in den Dienst geschrieben wurde.

Lassen Sie uns nun einen Blick auf das andere Ende des Test-Szenarios werfen. Statt den Listener zu testen, möchten Sie, dass die Event-Quelle das Ereignis zum richtigen Zeitpunkt auslöst. Der nächste Abschnitt zeigt, wie das bewerkstelligt werden kann.

5.4.2 Der Test, ob ein Event getriggert wurde

Eine einfache Möglichkeit, das Event zu testen, besteht darin, sich innerhalb der Test-Methode über einen anonymen Delegaten für das Event zu registrieren. Das nächste Listing zeigt ein einfaches Beispiel.

```
[Test]
{
    bool loadFired = false;
    SomeView view = new SomeView();
    view.Load+=delegate
              {
                  loadFired = true;
              };
    view.DoSomethingThatEventuallyFiresThisEvent();

    Assert.IsTrue(loadFired);
}
```

Listing 5.10: Die Verwendung eines anonymen Delegaten, um sich für ein Event zu registrieren

Der Delegat zeichnet einfach auf, ob das Event ausgelöst wurde oder nicht. Ich wähle lieber einen Delegaten als ein Lambda, denn ich denke, so ist es leichter zu lesen. Sie können auch Parameter im Delegaten verwenden, um die Werte aufzuzeichnen und sie später auch zu überprüfen.

Als Nächstes werfen wir einen Blick auf die Isolation-Frameworks für .NET.

5.5 Die aktuellen Isolation-Frameworks für .NET

NSub ist sicherlich nicht das einzige Isolation-Framework, das es gibt. Aber in einer informellen Umfrage vom August 2012 fragte ich die Leser meines Blogs: »Welches Isolation-Framework verwenden Sie?« Abbildung 5.2 zeigt die Ergebnisse.

Moq, das in der vorhergehenden Ausgabe dieses Buches und der dortigen Umfrage ein Newcomer war, hat nun die Führung übernommen, während Rhino Mocks zurückgefallen ist und weiter an Boden verliert (hauptsächlich weil es nicht mehr aktiv weiterentwickelt wird). Bemerkenswert ist auch, dass sich seit der ersten Ausgabe die Zahl der Mitbewerber verdoppelt hat. Das sagt etwas darüber aus, wie sehr die Community gereift ist und dass sie die Notwendigkeit anerkennt, zu testen und zu isolieren. Das zu sehen, finde ich großartig.

FakeItEasy, das beim Erscheinen der ersten Auflage dieses Buches wohl niemanden vom Hocker gerissen hat, ist inzwischen zu einem starken Mitbewerber geworden. Ich mag es aus den gleichen Gründen wie NSubstitute und empfehle Ihnen dringend, einen Blick darauf zu werfen. Diese Bereiche (Werte, tatsächlich) werden im nächsten Kapitel aufgelistet, wenn wir noch tiefer in die Zutaten der Isolation-Frameworks eintauchen.

Ich persönlich verwende Moq nicht – wegen der schlechten Fehler-Benachrichtigung und weil »Mock« zu häufig in der API benutzt wird. Das ist verwirrend, da Sie Mocks auch verwenden, um Stubs zu erzeugen.

Gewöhnlich ist es eine gute Idee, eines auszuwählen und im Sinne der Lesbarkeit und der flacheren Lernkurve für die Team-Mitglieder so weit wie möglich dabeizubleiben.

Im Anhang des Buches behandle ich jedes dieser Frameworks eingehender und erkläre, warum ich es mag oder auch nicht mag. Dort finden Sie eine Referenz-Liste dieser Tools.

Lassen Sie uns die Vorteile der Verwendung von Isolation-Frameworks statt handgeschriebener Mocks rekapitulieren. Dann diskutieren wir die Dinge, auf die man bei der Verwendung von Isolation-Frameworks achten sollte.

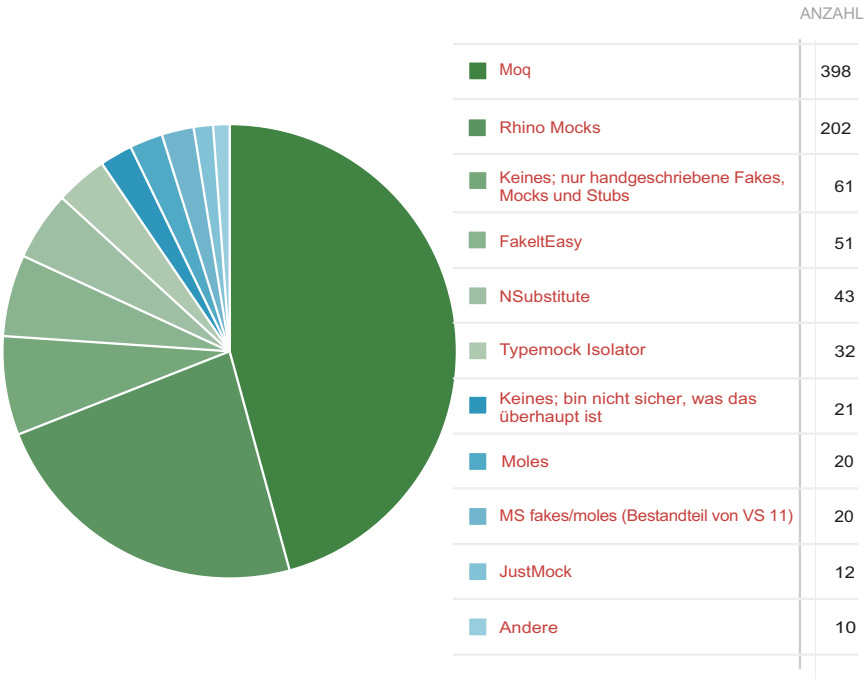


Abb. 5.2: Die Verbreitung von Isolation-Frameworks unter den Lesern meines Blogs

Warum Methoden-Strings innerhalb von Tests schlecht sind

In vielen Frameworks außerhalb der .NET-Welt ist es üblich, mit Strings zu beschreiben, für welche Methoden man gerade dabei ist, das Verhalten zu ändern. Warum ist das nicht so toll?

Wenn Sie den Namen einer Methode im Produktionscode ändern, dann könnten Sie alle Tests, die den Methodennamen als String verwenden, immer noch erfolgreich kompilieren und erst zur Laufzeit würden sie mit einer Ausnahme, die sagt, dass eine Methode nicht gefunden werden konnte, fehlschlagen.

Mit stark typisierten Methodennamen (dank der Lambda-Ausdrücke und der Delegaten) wäre das Ändern eines Methodennamens kein Problem, denn die Methode wird direkt im Test verwendet. Jegliche Änderungen an einer Methode würden zu einem Kompilier-Fehler führen und Sie würden sofort wissen, dass es ein Problem mit Ihrem Test gibt.

Mit automatischen Refactoring-Werkzeugen, wie solchen in Visual Studio, ist die Namensänderung einer Methode einfacher, aber meist werden Strings innerhalb des Quellcodes ignoriert. (ReSharper für .NET ist eine Ausnahme. Es korrigiert ebenso Strings, das ist aber nur teilweise eine Lösung, die sich in einigen Szenarien als problematisch erweisen kann.)

5.6 Die Vorteile und Fallstricke von Isolation-Frameworks

Ausgehend von dem, was ich in diesem Kapitel bisher beschrieben habe, können Sie eindeutige Vorteile bei der Verwendung von Isolation-Frameworks erkennen:

- *Einfachere Parameterverifizierung* – Die Verwendung von handgeschriebenen Mocks, um zu testen, dass einer Methode die korrekten Parameterwerte übergeben wurden, kann ein mühsamer Prozess sein, der Zeit und Geduld verlangt. Die meisten Isolation-Frameworks machen die Überprüfung von Parameterwerten, die an Methoden übergeben werden, zu einem trivialen Prozess, auch wenn es sich um viele Parameter handelt.
- *Einfachere Verifizierung bei einer Vielzahl von Methodenaufrufen* – Mit handgeschriebenen Mocks kann es schwierig sein, zu überprüfen, ob mehrfache Aufrufe der gleichen Methode korrekt ausgeführt wurden, wobei jeder Aufruf die jeweils richtigen, aber teilweise unterschiedlichen Parameter enthalten muss. Wie Sie später sehen werden, ist das mit Isolation-Frameworks ein trivialer Prozess.
- *Einfachere Erzeugung von Fakes* – Isolation-Frameworks können verwendet werden, um das Erzeugen von Mocks und Stubs zu vereinfachen.

5.6.1 Fallstricke, die man bei der Verwendung von Isolation-Frameworks besser vermeidet

Obwohl die Verwendung von Isolation-Frameworks viele Vorteile mit sich bringt, gibt es auch einige mögliche Gefahren. Beispiele sind etwa die Überbeanspruchung eines Isolation-Frameworks, wenn ein manuelles Mock-Objekt ausreichen würde, die Unlesbarkeit eines Tests durch die übertriebene Verwendung von Mocks oder eine unzureichende Trennung der Tests voneinander.

Hier ist eine Liste der Dinge, die zu beachten sind:

- Unlesbarer Testcode
- Die Verifizierung der falschen Dinge
- Die Verwendung von mehr als einem Mock pro Test
- Die Überspezifizierung von Tests

Lassen Sie uns jeden dieser Punkte etwas genauer betrachten.

5.6.2 Unlesbarer Testcode

Die Verwendung eines Mocks in einem Test macht den Test schon ein wenig unleserlicher, aber er ist immer noch so gut lesbar, dass ein Außenstehender ihn anschauen und verstehen kann, was da vor sich geht. Hat man aber viele Mocks oder viele Erwartungen innerhalb eines Tests, so können diese die Lesbarkeit eines Tests ruinieren, womit es schwierig wird, den Test zu warten oder auch nur zu verstehen, was da eigentlich getestet wird.

Wenn Sie meinen, dass Ihr Test unlesbar oder unverständlich wird, dann überlegen Sie, ob Sie nicht einige Mocks oder Mock-Erwartungen entfernen können oder ob Sie den Test vielleicht in mehrere kleinere Tests aufspalten können, die besser lesbar sind.

5.6.3 Die Verifizierung der falschen Dinge

Mock-Objekte erlauben es Ihnen, zu verifizieren, dass Methoden für Ihre Interfaces aufgerufen wurden, aber das heißt nicht notwendigerweise, dass Sie die richtigen Dinge testen. Zu testen, dass ein Objekt ein Event abonniert hat, sagt Ihnen nichts über die Funktionalität dieses Objekts. Zu überprüfen, dass etwas Sinnvolles passiert, wenn das Event ausgelöst wird, ist ein besserer Weg, das Objekt zu testen.

5.6.4 Die Verwendung von mehr als einem Mock pro Test

Es wird als eine gute Praxis betrachtet, nur eine Sache pro Test zu überprüfen. Mehr als ein Kriterium zu testen, kann zu Konfusion und Problemen bei der Wartung des Tests führen. Die Verwendung von zwei Mocks ist das Gleiche wie das Testen mehrerer Endresultate der gleichen Unit of Work. Wenn Sie Ihren Test nicht benennen können, weil er zu viele Aufgaben durchführt, ist es an der Zeit, ihn in mehr als einen Test aufzuteilen.

5.6.5 Die Überspezifizierung von Tests

Vermeiden Sie Mock-Objekte, wenn Sie können. Die Lesbarkeit und Wartbarkeit von Tests ist immer besser, wenn Sie nicht überprüfen, ob ein Objekt aufgerufen wurde. Sicherlich gibt es Situationen, in denen man nur mit einem Mock-Objekt weiterkommt, aber das sollte nicht allzu oft der Fall sein.

Wenn mehr als 5% Ihrer Tests Mock-Objekte (keine Stubs) beinhalten, dann überspezifizieren Sie möglicherweise die Dinge, statt Zustandsänderungen oder Rückgabewerte zu testen. Mit den 5%, die Mock-Objekte verwenden, können Sie es immer noch übertreiben.

Wenn Ihr Test zu viele Erwartungen (`x.received().X()` und `X.received().Y()` usw.) hat, dann kann er sehr zerbrechlich werden und bei der kleinsten Änderung des Produktions-Codes kaputtgehen, obwohl die Funktionalität insgesamt noch in Ordnung ist.

Das Testen von Interaktionen ist ein zweischneidiges Schwert: Testet man zu viel, verliert man das große Ganze aus dem Blick – die übergeordnete Funktionalität; testet man zu wenig, verpasst man die wichtigen Interaktionen zwischen den Objekten.

Nachfolgend einige Wege, um diesen Effekt auszubalancieren:

- *Verwenden Sie, wenn möglich, nicht-strikte Mocks (strikte und nicht-strikte Mocks werden im nächsten Kapitel erklärt).*

  Der Test wird weniger häufig wegen nicht erwarteter Methodenaufrufe zusammenbrechen. Das hilft, wenn sich die privaten Methoden im Produktionscode häufig ändern.

- *Verwenden Sie, wenn möglich, Stubs anstelle von Mocks.*

  Wenn mehr als 5% Ihrer Tests Mock-Objekte beinhalten, dann übertreiben Sie es vielleicht. Stubs können überall sein. Mocks nicht so sehr. Sie müssen nur ein Szenario zu einem Zeitpunkt testen. Je mehr Mocks Sie haben, desto mehr Überprüfungen finden am Ende des Tests statt, aber nur eine von ihnen wird gewöhnlich von Bedeutung sein. Der Rest wird Hintergrundrauschen des aktuellen Testszenarios sein.

- *Vermeiden Sie, so weit menschenmöglich, den Gebrauch von Stubs als Mocks.*

  Verwenden Sie einen Stub nur, um Rückgabewerte in das zu testende Programm zu fälschen oder um Ausnahmen zu werfen. Verifizieren Sie nicht, dass Methoden von Stubs aufgerufen werden. Verwenden Sie einen Mock nur, um zu verifizieren, dass eine Methode aufgerufen wurde, aber verwenden Sie ihn nicht, um Werte an Ihr zu testendes Programm zurückzugeben. Meist kann man vermeiden, einen Mock zu verwenden, der ebenfalls ein Stub ist, aber nicht immer (wie Sie bereits in diesem Kapitel in Bezug auf Events gesehen haben).

5.7 Zusammenfassung

Isolation-Frameworks sind ziemlich cool und Sie sollten lernen, sie nach Belieben einsetzen zu können. Aber es ist wichtig, wann immer möglich, in Richtung auf das Testen von Rückgabewerten und Zuständen zu tendieren (im Gegensatz zum Interaction Testing), damit Ihre Tests so wenig Annahmen wie möglich zu den internen Implementierungsdetails machen. Mocks sollten nur benutzt werden, wenn es keinen anderen Weg gibt, die Implementierung zu testen, denn sie führen letztendlich zu Tests, die, wenn man nicht sorgfältig ist, schwerer zu warten sind.

Wenn mehr als 5% Ihrer Tests Mock-Objekte (nicht Stubs) beinhalten, überspezifizieren Sie die Dinge vielleicht.

Lernen Sie, wie die fortgeschrittenen Features eines Isolation-Frameworks wie NSub angewendet werden. So können Sie weitgehend erreichen, dass alles Mögliche in Ihren Tests geschieht oder eben nicht geschieht. Alles, was Sie brauchen, ist ein testbarer Code.

Sie können sich auch selber ein Eigentor schießen, indem Sie überspezifizierte Tests anlegen, die nicht lesbar sind oder schnell zusammenbrechen. Die Kunst liegt darin, zu wissen, wann dynamische und wann handgeschriebene Mocks verwendet werden sollten. Sobald der Code, der das Isolation-Framework benutzt, beginnt, unansehnlich zu werden, ist meine Empfehlung, dies als Hinweis zu verstehen und die Dinge zu vereinfachen. Verwenden Sie einen handgeschriebenen Mock oder testen Sie ein anderes Resultat, um auf eine andere, aber einfachere Weise das zu prüfen, auf das Sie hinauswollen.

Wenn alles andere fehlschlägt und Ihr Code schwer zu testen ist, dann haben Sie drei Möglichkeiten: Sie können ein »Super-Framework« wie Typemock Isolator (wird im nächsten Kapitel beschrieben) verwenden, das Design ändern oder ... Ihren Job kündigen.

Isolation-Frameworks können Ihnen dabei helfen, Ihr Testleben zu erleichtern und Ihre Tests lesbarer und wartbarer zu machen. Aber es ist auch wichtig zu wissen, wann sie Ihren Entwicklungszyklus mehr behindern als unterstützen. In Legacy-Situationen beispielsweise könnten Sie überlegen, ein anderes Framework in Abhängigkeit von dessen Fähigkeiten einzusetzen. Es geht darum, das richtige Werkzeug für die Aufgabe auszuwählen. Daher sollten Sie sich das große Ganze anschauen, wenn Sie sich überlegen, wie Sie ein bestimmtes Testproblem angehen wollen.

Im nächsten Kapitel werden wir tiefer in die Isolation-Frameworks eintauchen und sehen, wie ihr Design und die zugrunde liegende Implementation ihre Fähigkeiten beeinflussen.

Kapitel 6

Wir tauchen tiefer ein in die Isolation-Frameworks

Dieses Kapitel behandelt

- das Arbeiten mit eingeschränkten im Gegensatz zu uneingeschränkten Frameworks
- eine Einführung, wie uneingeschränkte Profiler-basierte Frameworks funktionieren
- die Definition der Werte eines guten Isolation-Frameworks

Im vorangegangenen Kapitel haben wir NSubstitute verwendet, um Fakes zu erzeugen. In diesem Kapitel werden wir einen Schritt zurücktreten, um uns das größere Bild der Isolation-Frameworks sowohl in .NET als auch außerhalb davon anzuschauen. Die Welt der Isolation-Frameworks ist riesig und bei der Auswahl gilt es, viele verschiedene Dinge zu beachten.

Lassen Sie uns mit einer einfachen Frage beginnen: Warum haben manche Frameworks mehr Fähigkeiten als andere? Beispielsweise sind manche Frameworks in der Lage, statische Methoden zu fälschen, und andere nicht. Manche sind sogar in der Lage, Objekte zu fälschen, die noch gar nicht erzeugt wurden, und andere sind in seliger Unwissenheit von solchen Möglichkeiten. Was soll das?

6.1 Eingeschränkte und uneingeschränkte Frameworks

Isolation-Frameworks in .NET (und in Java, C++ und anderen statischen Sprachen) fallen in Bezug auf ihre Fähigkeiten, bestimmte Dinge in der Programmiersprache zu tun, in eine von zwei grundlegenden Kategorien. Ich nenne diese beiden Archetypen *eingeschränkt (constrained)* und *uneingeschränkt (unconstrained)*.

6.1.1 Eingeschränkte Frameworks

Zu den eingeschränkten Frameworks in .NET gehören Rhino Mocks, Moq, NMock, EasyMock, NSubstitute und FakeItEasy. In Java sind jMock und EasyMock Beispiele eingeschränkter Frameworks.

Ich nenne sie *eingeschränkt*, denn es gibt einige Dinge, die diese Frameworks nicht fälschen können. Was sie fälschen oder nicht fälschen können, ändert sich in Abhängigkeit von der Plattform, auf der sie laufen und wie sie diese Plattform nutzen.

In .NET sind eingeschränkte Frameworks nicht in der Lage, statische Methoden, nicht virtuelle Methoden, nicht öffentliche Methoden und Weiteres zu fälschen.

Was ist der Grund dafür? Eingeschränkte Isolation-Frameworks arbeiten in der gleichen Weise, wie Sie handgeschriebene Fakes verwenden: Sie generieren Code und kompilieren

ihn zur Laufzeit, womit sie durch die Fähigkeiten des Compilers und der Intermediate Language (IL) beschränkt sind. Dem entsprechen in Java der Compiler und der resultierende Bytecode. In C++ werden die eingeschränkten Frameworks durch die Sprache C++ und ihre Fähigkeiten beschränkt.

Gewöhnlich arbeiten eingeschränkte Frameworks, indem sie Code zur Laufzeit erzeugen, der von Interfaces und Basisklassen erbt und sie überschreibt. Genau das Gleiche haben Sie im vorherigen Kapitel gemacht, außer dass Sie es taten, bevor der Code ausgeführt wurde. Das bedeutet, dass diese Isolation-Frameworks auch die gleichen Anforderungen für die Kompilierung stellen: Der Code, den Sie fälschen wollen, muss öffentlich und vererbbar (nicht `sealed`) sein, muss einen öffentlichen Konstruktor haben oder sollte ein Interface sein. Für Basisklassen gilt, dass die Methoden, die Sie überschreiben möchten, virtuell sein müssen.

All das bedeutet, dass Sie bei der Benutzung von eingeschränkten Frameworks an die gleichen Compiler-Regeln gebunden sind wie regulärer Code. Statische Methoden, private Methoden, versiegelte Klassen, Klassen mit privaten Konstruktoren und so weiter sind raus aus dem Spiel, wenn Sie ein solches Framework benutzen.

6.1.2 Uneingeschränkte Frameworks

Zu den uneingeschränkten Frameworks in .NET gehören Typemock Isolator, JustMock und Moles (auch als MS Fakes bekannt). In Java sind PowerMock und JMockit Beispiele für uneingeschränkte Frameworks. In C++ sind Isolator und Hippo Mocks Beispiele solcher Frameworks. Uneingeschränkte Frameworks generieren und kompilieren Code, der von anderem Code erbt, nicht zur Laufzeit. Sie wenden gewöhnlich andere Mittel an, um zu bekommen, was sie brauchen, und diese Mittel ändern sich in Abhängigkeit von der Plattform.

Bevor wir darin eintauchen, wie sie in .NET arbeiten, sollte ich noch erwähnen, dass dieses Kapitel doch ein wenig tiefer geht. Es geht nicht wirklich um die Kunst des Unit Testings, sondern es vermittelt Ihnen Wissen, warum die Dinge sind, wie sie sind, und gibt Ihnen damit die Möglichkeit, Ihre Entscheidungen zu Ihrem Unit-Test-Design auf Basis besserer Informationen zu treffen und daran Ihre Handlungen auszurichten.

In .NET sind alle uneingeschränkten Frameworks Profiler-basiert. Das bedeutet, dass sie einen Satz von nicht gemanagten APIs aufrufen, die sogenannten *Profiling-APIs*, die in .NET um die laufende Instanz der CLR – die Common Language Runtime – gewickelt sind. Mehr dazu können Sie hier lesen: `http://msdn.microsoft.com/en-us/library/bb384493.aspx#profiling_api`. Diese APIs bieten Events für alles an, das während der CLR-Codeausführung geschieht, sogar auf Ereignisse, die vor der In-Memory Übersetzung des .NET IL Codes in Binärcode liegen. Einige dieser Events erlauben es Ihnen auch, IL-basierten Code zu ändern und neuen Code zu injizieren, bevor er In-Memory übersetzt wird und damit neue Funktionalität in den existierenden Code einzufügen. Eine Reihe von Werkzeugen da draußen, vom ANTS Profiler bis zu Memory Profilern benutzt bereits die Profiling-APIs. Typemock Isolator war vor mehr als 7 Jahren das erste Framework, das das Potenzial der Profiling-APIs und ihre Verwendung verstand und damit das Verhalten von »Fake«-Objekten ändern konnte.

Da die Profiling-Events den gesamten Code betreffen, einschließlich der statischen Methoden, der privaten Konstruktoren und des Third-Party-Codes, der Ihnen gar nicht gehört, wie etwa SharePoint, können diese uneingeschränkten Frameworks in .NET jeden Code injizie-

ren und sein Verhalten ändern, in jeder Klasse, in jeder Library, auch wenn sie nicht von Ihnen kompiliert wurden. Die Möglichkeiten sind unbegrenzt. Im Anhang werde ich die Unterschiede zwischen den Profiler-basierten Frameworks im Detail vorstellen.

In .NET müssen Sie die Umgebungsvariablen für den ausführbaren Prozess, der die Tests laufen lässt, aktivieren, um das Profiling zu ermöglichen. Das erlaubt Ihrem Code, die Tests auszuführen, die Sie mithilfe eines Frameworks auf Basis der Profiling-APIs geschrieben haben. In der Voreinstellung sind sie nicht aktiviert, weshalb kein Profiling auf .NET-Code möglich ist, bis Sie sie umstellen. Setzen Sie `Cor_Enable_Profiling=0x1` und `COR_PROFILER=SOME_GUID` für den Profiler, den Sie an den Prozess, der die Tests durchführt, anhängen wollen. (Ja, zu einem Zeitpunkt kann nur ein Profiler angehängt sein.)

Frameworks wie Moles, Typemock und JustMock haben alle spezielle Add-Ins für Visual Studio, die diese Umgebungsvariablen einschalten und es Ihren Tests ermöglichen, zu laufen. Diese Tools haben meist ein spezielles Kommandozeilen-Programm, das Ihre anderen Kommandozeilen-Aufgaben mit den angesprochenen und eingeschalteten Umgebungsvariablen laufen lässt.

Wenn Sie versuchen, Ihre Tests laufen zu lassen, ohne sie einzuschalten, können Sie bizarre Fehlermeldungen im Ausgabefenster des Test-Runners sehen. Seien Sie also gewarnt. Das von Ihnen verwendete Isolation-Framework könnte beispielsweise melden, dass nichts aufgezeichnet wurde oder dass keine Tests ausgeführt wurden.

Die Verwendung von uneingeschränkten Isolation-Frameworks hat einige Vorteile:

- Sie können Tests für zuvor untestbaren Code schreiben, denn sie können Dinge um die Unit of Work herum fälschen und sie isolieren, ohne den Code anzurühren und anzupassen. Später, wenn Sie Tests haben, können Sie mit dem Refactoring beginnen.
- Sie können Third-Party-Systeme fälschen, die Sie nicht kontrollieren können und die möglicherweise sehr schwer in einen Test einzubeziehen sind. Wenn Sie etwa Ihre Objekte von der Basisklasse eines Third-Party-Produkts ableiten müssen, das viele Abhängigkeiten auf einem niedrigeren Level enthält (SharePoint, CRM, Entity Framework oder Silverlight, um nur ein paar zu nennen).
- Sie können Ihr eigenes Design-Level wählen, statt bestimmte Muster aufgezwungen zu bekommen. Das Design wird nicht vom Werkzeug erzeugt; das ist eine Sache der Leute. Wenn Sie nicht wissen, was Sie da tun, wird Ihnen ein Werkzeug sowieso nicht helfen. Ich werde mehr dazu in Kapitel 11 erwähnen.

Die Verwendung von uneingeschränkten Frameworks hat auch ein paar Nachteile:

- Wenn Sie nicht achtsam sind, dann können Sie sich selber in eine Ecke fälschen, indem Sie die Dinge fälschen, die gar nicht notwendig sind, statt auf die Unit of Work auf einem höheren Level zu achten.
- Wenn Sie nicht achtsam sind, können einige Tests unwartbar werden, denn Sie fälschen APIs, die Ihnen nicht gehören. Das kann passieren, aber nicht so häufig, wie Sie vielleicht meinen. Wenn Sie eine API in einem Framework auf einem Level fälschen, der tief genug ist, dann ist es meiner Erfahrung nach unwahrscheinlich, dass sie sich in Zukunft ändern wird. Je tiefer eine API ist, desto mehr Dinge bauen wahrscheinlich auf ihr auf und desto unwahrscheinlicher ist eine Änderung.

Als Nächstes schauen wir uns an, was den uneingeschränkten Frameworks solche Kunststücke erlaubt.

6.1.3 Wie Profiler-basierte uneingeschränkte Frameworks arbeiten

Dieser Abschnitt trifft nur auf die .NET-Plattform und die CLR zu, denn dort ist es, wo die Profiling-APIs leben, und er sollte nur für die Leser eine Rolle spielen, die sich für die genauen und ganz kleinen Details interessieren. Diese zu kennen ist nicht wichtig, um gute Unit Tests zu schreiben, aber es ist gut für extra Bonuspunkte, falls Sie jemals einen Mitbewerber dieser Frameworks ins Rennen schicken wollen. Für diese Angelegenheit werden in Java oder C++ andere Techniken eingesetzt.

In .NET schreiben Tools wie Typemock Isolator nativen Code in C++, der sich an das COM-Interface der CLR-Profiler-API anhängt und sich für eine Handvoll Event-Hook-Callbacks registriert. Typemock besitzt tatsächlich ein Patent darauf (Sie finden es auf `http://bit.ly/typemockpatent`), das sie aber scheinbar nicht geltend machen, denn sonst würden wir wohl nicht Mitbewerber wie JustMock oder Moles den Ring besteigen sehen.

`JitCompilationStarted` in Kombination mit `SetILFunctionBody`, beides Mitglieder des `ICorProfilerCallback2`-COM-Interface, erlauben es Ihnen, zur Laufzeit den IL-Code zu ermitteln und zu ändern, der kurz *davor* steht, in Binärcode umgewandelt zu werden. Sie ändern diesen IL-Code so, dass er Ihren eigenen Code einschließt. Werkzeuge wie Typemock fügen IL-Header vor und nach jeder Methode ein, die sie in die Finger bekommen. Diese Header sind im Wesentlichen Logik-Code, der gemanagten C#-Code aufruft und überprüft, ob irgendjemand ein spezielles Verhalten auf diese Methoden gesetzt hat. Stellen Sie sich diesen Prozess so vor, dass globale, aspektorientierte und quer verlaufende Checks für das Verhalten aller Methoden in Ihrem Code erzeugt werden. Die injizierten IL-Header werden auch Aufrufe für gemanagte Code-Hooks (gewöhnlich in C# geschrieben, wo das Herz der Isolation-Framework-Logik liegt) beinhalten, je nachdem, welches Verhalten vom Benutzer der Framework-API gesetzt wurde (so wie »werfe eine Ausnahme« oder »gebe einen gefälschten Wert zurück«).

Die Just-In-Time-(JIT-)Kompilierung wird in .NET für so ziemlich alles verwendet (außer es ist mit `NGen.exe` pre-JITed). Das schließt allen Code ein, nicht nur Ihren eigenen, und selbst das .NET-Framework, SharePoint oder andere Bibliotheken.

Das bedeutet, dass ein Framework wie Typemock jeden IL-Code, der das Verhalten ändert, in jeden Code nach Belieben injizieren kann, auch wenn er nicht Teil des .NET-Frameworks ist. Sie können diese Header vor und nach jeder Methode einführen, auch wenn Sie den Code der Methode nicht geschrieben haben. Bei Legacy-Code, für den Sie keine Möglichkeit eines Refactorings haben, können diese Frameworks ein Geschenk des Himmels sein.

Hinweis

Die Profiling-APIs sind nicht besonders gut dokumentiert (mit Absicht?). Aber wenn Sie eine Internet-Suche nach `JitCompilationStarted` und `SetILFunctionBody` durchführen, sollten Sie viele Referenzen und Anekdoten finden, die Ihnen bei Ihrer Untersuchung, wie man ein uneingeschränktes Isolation-Framework in .NET selber baut, weiterhelfen. Bereiten Sie sich auf eine lange und anstrengende Reise vor und lernen Sie C++. Und nehmen Sie eine Flasche Whiskey mit.

Frameworks zeigen verschiedene Profiler-Fähigkeiten

Alle Profiler-basierten Isolation-Frameworks haben potenziell die gleichen, zugrunde liegenden Fähigkeiten. Aber im wahren Leben sind die wichtigsten Frameworks in .NET in

Bezug auf ihre Fähigkeiten nicht gleich. Jedes der drei großen Profiler-basierten Frameworks – JustMock, Typemock und MS Fakes (Moles) – implementiert eine Teilmenge der kompletten Möglichkeiten.

Hinweis

Ich benutze die Namen Typemock und Typemock Isolator synonym, denn derzeit verweisen beide Begriffe auf das gleiche Isolator-Produkt.

Typemock, das von allen am längsten existiert, unterstützt nahezu jeden Code, der heute beim Umgang mit Legacy-Code untestbar erscheint. Dazu gehören auch zukünftige Objekte, statische Konstruktoren und andere schräge Kreaturen. Schwächen hat es nur im Bereich des Fälschens der APIs aus `mscorlib.dll`; das ist die Bibliothek, die grundlegende APIs wie DateTime, System.String und System.IO Namespaces enthält. In dieser spezifischen DLL (und nur in dieser), implementiert Typemock nur eine Handvoll der APIs statt aller.

Technisch gesehen könnte Typemock das Fälschen aller Typen der kompletten Library erlauben, aber aus Performance-Gründen ist das unrealistisch. Stellen Sie sich vor, alle Strings in Ihrem System wären gefälscht, um ein paar gefälschte Werte zurückzugeben. Multiplizieren Sie die Anzahl der String-Aufrufe in die zugrunde liegende API des .NET-Frameworks mit einem oder zwei Checks innerhalb der Typemock-API, um zu prüfen, ob diese Aktion gefälscht werden soll oder nicht, und Sie stecken mitten in einem Performance-Albtraum.

Abgesehen von ein paar Kerntypen des .NET-Frameworks unterstützt Typemock so ziemlich alles, wonach Sie werfen können.

MS Fakes hat einen Vorteil gegenüber Typemock Isolator. Es wurde innerhalb von Microsoft geschrieben und entwickelt. Ursprünglich war es eine Erweiterung zu einem anderen Tool namens Pex (wird im Anhang beschrieben). Weil es inhouse entwickelt wurde, hatten die Microsoft-Entwickler einen besseren Einblick in die weitgehend undokumentierten Profiling-APIs und konnten so eine Unterstützung für einige Typen einbauen, die selbst Typemock Isolator nicht zu fälschen gestattet. Auf der anderen Seite enthält die API von MS Fakes nur wenig Unterstützung für die Legacy-Code-bezogene Funktionalität, die Sie in Isolator oder JustMock finden und die Sie von einem Framework mit solchen Fähigkeiten erwarten können. Hauptsächlich erlaubt es Ihnen, die API, öffentliche Methoden (statisch oder nicht statisch) mit Ihren eigenen Delegaten zu ersetzen, aber es erlaubt Ihnen nicht das Fälschen nicht öffentlicher Methoden mit Bordmitteln.

Im Hinblick auf seine API und darauf, was es fälschen kann, kommt JustMock den Fähigkeiten von Typemock Isolator recht nahe, aber es fehlen noch ein paar Dinge in Bezug auf Legacy-Code, wie etwa das Fälschen statischer Konstruktoren und privater Methoden. Das liegt hauptsächlich daran, wie lange das Produkt schon existiert. MS Fakes und JustMock sind inzwischen vielleicht drei Jahre alt. Typemock hat auf sie einen Vorsprung von drei bis vier Jahren.

Sie müssen sich darüber im Klaren sein, dass Sie mit der Wahl eines Isolation-Frameworks auch eine Wahl in Bezug auf die grundlegenden Fähigkeiten und Einschränkungen treffen.

Hinweis

Profiler-basierte Frameworks bringen einige Performance-Nachteile mit sich. Bei jedem Schritt entlang des Weges fügen sie Ihrem Code Aufrufe hinzu, wodurch er langsamer läuft. Möglicherweise bemerken Sie das erst, nachdem Sie einige Hundert Tests hinzugefügt haben, aber es fällt auf und es ist da. Ich denke, das ist ein kleiner Preis dafür, dass man einen großen Vorteil erhält und in der Lage ist, Legacy-Code zu fälschen und zu testen.

6.2 Werte guter Isolation-Frameworks

In .NET (und ein wenig auch in Java) ist in den letzten paar Jahren eine neue Generation von Isolation-Frameworks entstanden. Sie haben etwas von dem Gewicht verloren, das die älteren, etablierteren Frameworks mit sich herumgetragen haben, und machten große Fortschritte auf den Gebieten der Lesbarkeit, Benutzbarkeit und Einfachheit. Aber am wichtigsten ist, dass sie die Robustheit der Tests langfristig mit Fähigkeiten unterstützen, die ich gleich auflisten werde.

Zu diesen neuen Isolation-Frameworks gehören Typemock Isolator (obwohl es schon in die Jahre gekommen ist), NSubstitute und FakeItEasy. Das erste ist ein uneingeschränktes Framework, während die anderen beiden eingeschränkte Frameworks sind, aber sie alle bringen unabhängig von ihren innewohnenden Einschränkungen interessante Dinge mit sich.

Unglücklicherweise unterstützen die meisten Isolation-Frameworks diese Werte der Lesbarkeit und Benutzerfreundlichkeit immer noch nicht in Sprachen wie Ruby, Python, JavaScript und anderen. Das mag an der mangelnden Reife der Frameworks selbst liegen, aber es könnte auch daran liegen, dass die Unit-Testing-Kultur in diesen Sprachen noch nicht zu den gleichen Schlüssen gelangt ist, die die .NET-Unit-Testing-Freaks bereits gezogen haben. Andererseits könnten wir alle es auch falsch machen und die Art, wie die Dinge in Ruby isoliert werden, ist der richtige Weg, dem man folgen sollte. Na ja, wo war ich jetzt?

Gute Isolation-Frameworks besitzen, was ich *die zwei großen Werte* nenne:

- Zukunftssicherheit
- Benutzerfreundlichkeit

Hier kommen einige Fähigkeiten, die in den neueren Frameworks diese Werte unterstützen:

- rekursive Fakes
- per Voreinstellung ignorierte Argumente
- umfangreiches Fälschen
- nicht striktes Verhalten von Fakes
- nicht strikte Mocks

6.3 Eigenschaften, die Zukunftssicherheit und Benutzerfreundlichkeit unterstützen

Ein zukunftssicherer Test wird angesichts größerer, zukünftiger Änderungen am Produktionscode nur aus den richtigen Gründen fehlschlagen. Benutzerfreundlichkeit ist die Quali-

tät, die es Ihnen erlaubt, das Framework leicht zu verstehen und zu benutzen. Isolation-Frameworks können sehr leicht sehr schlecht benutzt werden und damit die Ursache für sehr zerbrechliche und wenig zukunftssichere Tests sein.

Dies sind einige Fähigkeiten, die die Robustheit fördern:

- rekursive Fakes
- die Voreinstellung, Argumente bei Verhalten und Überprüfung zu ignorieren
- nicht striktes Überprüfen und Verhalten
- umfangreiches Fälschen

6.3.1 Rekursive Fakes

Rekursives Fälschen ist ein besonderes Verhalten von Fake-Objekten, wenn Funktionen andere Objekte zurückgeben. Diese Objekte sind immer und automatisch gefälscht. Alle Objekte, die von Funktionen innerhalb dieser automatisch gefälschten Objekte zurückgegeben werden, sind ebenfalls gefälscht. Das Ganze ist rekursiv.

Hier ist ein Beispiel:

```
public Interface IPerson
{
}
[Test]
public void RecursiveFakes_work()
{
    IPerson p = Substitute.For<IPerson>();

    Assert.IsNotNull(p.GetManager());
    Assert.IsNotNull(p.GetManager().GetManager());
    Assert.IsNotNull(p.GetManager().GetManager().GetManager());
}
```

Um das ans Laufen zu bekommen, müssen Sie nichts anderes tun, als eine einzige Zeile Code zu schreiben. Aber warum ist diese Fähigkeit wichtig? Je weniger Sie dem Test-Setup über jede spezifische API, die gefälscht werden soll, mitteilen müssen, umso besser ist Ihr Test von der aktuellen Implementierung des Produktionscodes entkoppelt und umso weniger müssen Sie den Test ändern, wenn sich der Produktionscode in Zukunft ändert.

Nicht alle Isolation-Frameworks erlauben rekursive Fakes, weshalb Sie überprüfen sollten, ob sie von Ihrem Lieblings-Framework unterstützt wird. Nach meinem Kenntnisstand wird diese Fähigkeit aktuell nur von .NET-Frameworks auch nur in Erwägung gezogen. Ich wünschte mir, sie würde auch für andere Sprachen existieren.

Beachten Sie auch, dass beschränkte Frameworks in .NET rekursive Fakes nur für solche Funktionen unterstützen, die von generiertem Code überschrieben werden können: öffentliche Methoden, die virtuell oder Teil eines Interface sind.

Manche fürchten, dass es durch eine solche Fähigkeit leichter wird, das Gesetz von Demeter (`http://de.wikipedia.org/wiki/Gesetz_von_Demeter`) zu brechen. Ich stimme

dem nicht zu, denn ein gutes Design wird nicht durch ein Werkzeug erzwungen, sondern wird von Menschen erschaffen, die miteinander reden und voneinander lernen und zu zweit Code-Reviews durchführen. Aber Sie werden mehr zum Thema Design in Kapitel 11 erfahren.

6.3.2 Ignoriere Argumente als Voreinstellung

Derzeit werden von allen Frameworks, außer Typemock Isolator, Argument-Werte, die Sie an verhaltensändernde oder verifizierende APIs übergeben, per Voreinstellung als erwartete Werte verwendet. Isolator ignoriert diese übergebenen Werte, außer Sie geben in den API-Aufrufen explizit an, dass Sie sich für die Werte der Argumente interessieren. Es ist nicht notwendig, immer `Arg.IsAny<Type>` in alle Methoden einzufügen, was Tipparbeit spart und Allgemeinplätze vermeidet, die die Lesbarkeit erschweren. Um Typemock Isolator (`typemock.com`) dazu zu bringen, eine Ausnahme zu werfen, egal was die Argumente sind, brauchen Sie nur dies zu schreiben:

```
Isolate.WhenCalled(() => stubLogger.Write(""))
                         .WillThrow(new Exception("Fake"));
```

6.3.3 Umfangreiches Fälschen

Umfangreiches Fälschen (Wide Faking) ist die Möglichkeit, mehrere Methoden auf einmal zu fälschen. In gewisser Hinsicht sind rekursive Fakes eine Teilmenge dieser Idee, aber es gibt auch andere Implementierungen.

Mit einem Tool wie FakeItEasy können Sie beispielsweise anzeigen, dass alle Methoden eines bestimmten Objekts den gleichen Wert zurückgeben oder auch nur solche Methoden mit einem bestimmten Typ:

```
A.CallTo(foo).Throws(new Exception());
A.CallTo(foo).WithReturnType<string>().Returns("hello world");
```

Mit Typemock können Sie kennzeichnen, dass alle statischen Methoden eines bestimmten Typs per Voreinstellung einen gefälschten Wert zurückgeben:

```
Isolate.Fake.StaticMethods(typeof(HttpRuntime));
```

Von diesem Moment an gibt jede statische Methode für dieses Objekt einen gefälschten Wert in Abhängigkeit von seinem Typ zurück oder ein rekursives Fake-Objekt, wenn der Rückgabetyp ein Objekt ist.

Ich glaube, dass das eine sehr gute Sache für die zukünftige Tragfähigkeit der Tests ist, während der Produktionscode sich entwickelt. Eine Methode, die nach sechs Monaten hinzugefügt und vom Produktionscode verwendet wird, die wird automatisch von allen existierenden Tests gefälscht, weshalb sich diese Tests nicht um die neue Methode kümmern.

6.3.4 Nicht striktes Verhalten von Fakes

Die Welt der Isolation-Frameworks war für gewöhnlich eine sehr strikte und ist es meist immer noch. Viele der Frameworks in anderen Sprachen als .NET (wie Java und Ruby) sind

von vornherein strikt, während viele .NET-Frameworks inzwischen aus diesem Stadium herausgewachsen sind.

Die Methoden eines strikten Fake-Objekts können nur dann erfolgreich aufgerufen werden, wenn Sie sie in der Isolation-API als »erwartet« angeben. Die Fähigkeit, zu erwarten, dass eine Methode für ein Fake-Objekt aufgerufen wird, existiert in NSubstitute (oder FakeItEasy) nicht, aber sie existiert in vielen der anderen Frameworks in .NET und in anderen Sprachen (siehe Moq, Rhino Mocks und die alte API von Typemock Isolator).

Wenn eine Methode als erwartet konfiguriert wird, dann führt gewöhnlich jeder Aufruf, der sich von der Erwartung unterscheidet (beispielsweise erwarte ich, dass die Methode `LogError` zu Beginn des Tests mit Parameter `a` aufgerufen wird), sei es durch die Parameter-Werte oder durch den Methodennamen, zum Werfen einer Ausnahme.

In der Regel wird der Test mit dem ersten unerwarteten Methodenaufruf für ein striktes Mock-Objekt fehlschlagen. Ich sage *in der Regel*, denn ob der Mock eine Ausnahme wirft, hängt von der Implementation des Isolation-Frameworks ab. Einige Frameworks erlauben Ihnen, alle Ausnahmen bis zum Aufruf von `verify()` am Ende des Tests zu verschieben.

Die wichtigsten Gründe, warum das Design der meisten Frameworks so ist, können Sie im Buch *Growing Object-Oriented Software, Guided by Tests* von Freeman und Pryce (Addison-Wesley, 2009) nachlesen. In diesem Buch verwenden sie Mock-Assertions, um das »Protokoll« der Kommunikation zwischen Objekten zu beschreiben. Weil ein Protokoll etwas ist, das recht strikt sein muss, sollte Ihnen das Lesen des Protokolls das Verständnis erleichtern, welche Erwartungen an die Interaktion mit einem Objekt gestellt werden.

Was ist daran problematisch? Die Idee selbst ist nicht das Problem; es ist die Leichtigkeit, mit der man diese Fähigkeit missbrauchen und überstrapazieren kann.

Ein strikter Mock kann auf zwei Arten fehlschlagen: wenn eine unerwartete Methode für ihn aufgerufen wird oder wenn erwartete Methoden nicht für ihn aufgerufen werden (was durch den Aufruf von `Received()` entschieden wird).

Es ist das Erstere, was mich beunruhigt. Angenommen, ich interessiere mich nicht für die internen Protokolle zwischen Objekten, die Interna meiner Unit of Work sind, dann sollte ich auch keine Asserts auf ihre Interaktionen setzen, oder ich gerate in ein Minenfeld. Ein Test kann fehlschlagen, wenn ich mich entscheide, eine Methode für irgendein internes Objekt innerhalb der Unit of Work aufzurufen, die keinerlei Einfluss auf das Endresultat der Unit of Work hat. Trotzdem wird mein Test fehlschlagen und jammern: »Du hast mir nicht gesagt, dass jemand diese Methode aufrufen wird!«

6.3.5 Nicht strikte Mocks

Meist sorgen nicht strikte Mocks für weniger zerbrechliche Tests. Ein nicht striktes Mock-Objekt erlaubt jeden Aufruf, auch wenn er nicht erwartet wurde. Für Methoden mit Rückgabewerten gibt es den voreingestellten Wert zurück, wenn es sich um ein Wert-Objekt handelt, oder `null` für ein Objekt. In fortschrittlicheren Frameworks gibt es auch den Begriff des rekursiven Fakes, wobei ein Fake-Objekt, das eine Methode hat, die ein Objekt zurückgibt, per Voreinstellung aus dieser Methode ein Fake-Objekt zurückgibt. Und dieses Fake-Objekt wird rekursiv auch Fake-Objekte für seine Methoden zurückgeben, die Objekte zurückgeben. (Dieses Verhalten existiert in Typmock Isolator sowie in NSub und Moq und teilweise auch in Rhino Mocks.)

Listing 5.3 in Kapitel 5 ist ein reines Beispiel für nicht strikte Mocks. Sie interessiert nicht, welche anderen Aufrufe geschehen. Listing 5.4 und der Code-Block danach zeigen, wie Sie den Test robuster und zukunftssicherer machen, indem Sie Argument-Matcher verwenden, statt ganze Strings zu erwarten. Argument-Matcher ermöglichen es Ihnen, Regeln dafür festzulegen, wie Parameter an den Fake übergeben werden sollten, damit sie als in Ordnung akzeptiert werden. Beachten Sie, wie das den Test recht schnell unansehnlich macht.

6.4 Isolation-Framework-Design-Antimuster

Hier sind einige der Antimuster, die wir heute in den Frameworks vorfinden und die wir leicht entschärfen können:

- Konzept-Konfusion (Concept Confusion)
- Aufnahme und Wiedergabe (Record and Replay)
- Klebriges Verhalten (Sticky Behaviour)
- Komplexe Syntax (Complex Syntax)

In diesem Abschnitt werfen wir einen Blick auf jedes davon.

6.4.1 Konzept-Konfusion

Die Konzept-Konfusion ist etwas, das ich auch gerne als *Mock Überdosis* bezeichne. Ich bevorzuge ein Framework, das nicht für alles das Wort *Mock* verwendet.

Sie müssen wissen, wie viele Mocks und Stubs in einem Test sind, denn gewöhnlich ist mehr als ein Mock pro Test ein Problem. Wenn es nicht zwischen diesen beiden unterscheidet, könnte Ihnen das Framework vormachen, dass etwas ein Mock sei, das aber tatsächlich als Stub verwendet wird. Sie brauchen länger, um zu verstehen, ob es sich um ein echtes Problem handelt oder nicht, womit die Lesbarkeit des Tests verletzt ist.

Hier kommt ein Beispiel aus Moq:

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var view = new Mock<IView>();
    var logger = new Mock<ILogger>();

    Presenter p = new Presenter(view.Object, logger.Object);
    view.Raise(v => v.ErrorOccured += null, "fake error");

    logger.Verify(log =>
        log.LogError(It.Is<string>(s=> s.Contains("fake error"))));
}
```

Auf die folgende Weise können Sie die Konzept-Konfusion vermeiden:

- Verwenden Sie spezifische Wörter für *Mock* und *Stub* in der API. Rhino Mocks macht das beispielsweise so.

- Verwenden Sie die Begriffe *Mock* und *Stub* überhaupt nicht in der API. Verwenden Sie stattdessen einen allgemeinen Begriff, für was auch immer ein Fake ist. Beispielsweise ist in FakeItEasy alles ein *Fake*<Something>. Es gibt überhaupt keinen Mock oder Stub in der API. Wie Sie sich erinnern, ist in NSubstitute alles ein *Substitute*<Something>. In Typemock Isolator rufen Sie nur `Isolate.Fake.Instance<Something>` auf. Ein Mock oder Stub wird nicht erwähnt.
- Wenn Sie ein Isolation-Framework verwenden, das nicht zwischen Mocks und Stubs unterscheidet, dann benennen Sie zumindest Ihre Variablen `mockXXX` und `stubXXX`, um einige der Lesbarkeitsprobleme abzuschwächen.

Indem Sie den überladenen Term komplett entfernen oder dem Benutzer die Möglichkeiten geben, zu spezifizieren, was sie erzeugen, wird die Lesbarkeit der Tests verbessert oder zumindest ist die Terminologie weniger verwirrend.

Hier kommt der vorherige Test mit den geänderten Variablen-Namen, um anzudeuten, wie sie verwendet werden. Liest sich das für Sie besser?

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var stubView = new Mock<IView>();
    var mockLogger = new Mock<ILogger>();

    Presenter p = new Presenter(stubView.Object, mockLogger.Object);
    stubView.Raise(view => view.ErrorOccured += null, "fake error");

    mockLogger.Verify(logger =>
        logger.LogError(It.Is<string>(s=> s.Contains("fake error"))));
}
```

6.4.2 Aufnahme und Wiedergabe

Der Stil der Aufnahme und Wiedergabe (Record and Replay) für die Isolation-Frameworks erzeugte eine schlechte Lesbarkeit. Ein sicheres Anzeichen für schlechte Lesbarkeit ist es, wenn der Leser eines Tests immer wieder im gleichen Test oben und unten nachlesen muss, um zu verstehen, was da vor sich geht. Meist können Sie das in Code sehen, der mithilfe eines Isolation-Frameworks geschrieben wurde, das Record-and-Replay-APIs unterstützt.

Werfen Sie einen Blick auf dieses Beispiel aus dem Blog von Rasmus Kromann-Larsen, `http://rasmuskl.dk/post/Why-AAA-style-mocking-is-better-than-Record-Playback.aspx`, das Rhino Mocks verwendet (das Record and Replay unterstützt). Versuchen Sie nicht, es zu kompilieren. Es ist nur ein Beispiel.

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExistRecordPlayback()
{
    // Arrange
    var guid = Guid.NewGuid();
```

```
    // Part of Act
    IEventRaiser executeRaiser;

    using(_mocks.Record())
    {
        // Arrange (or Assert?)
        Expect.Call(_view.Respondents).Return(new[] {guid.ToString()});
        Expect.Call(_repository.GetById(guid)).Return(null);

        // Part of Act
        _view.ExecuteOperation += null;
        executeRaiser = LastCall.IgnoreArguments()
            .Repeat.Any()
            .GetEventRaiser();

        // Assert
        Expect.Call(_view.OperationErrors = null)
          .IgnoreArguments()
          .Constraints(List.IsIn("Non-existant respondent: " + guid));
    }

    using(_mocks.Playback())
    {
        // Arrange
        new BulkRespondentPresenter(_view, _repository);
        // Act
        executeRaiser.Raise(null, EventArgs.Empty);
    }
}
```

Und hier kommt der gleiche Code mit Moq (das den Arrange-Act-Assert-(AAA-)Stil unterstützt):

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExistRecordPlayback()
{
    // Arrange
    var guid = Guid.NewGuid();
    _viewMock.Setup(x => x.Respondents).Returns(new[]
                                        {guid.ToString() });
    _repositoryMock.Setup(x => x.GetById(guid)).Returns(() => null);

    // Act
    _viewMock.Raise(x => x.ExecuteOperation += null, EventArgs.Empty);
```

```
    // Assert
    _viewMock.VerifySet(x => x.OperationErrors =
                        It.Is<IList<string>>(l=>
                        l.Contains("Non-existant respondent: "+guid));
}
```

Sehen Sie, welchen großen Unterschied die Verwendung des AAA-Stils im Verhältnis zu Record and Replay macht?

6.4.3 Klebriges Verhalten

Sobald Sie einer gefälschten Methode sagen, dass sie sich auf eine bestimmte Weise verhalten soll, wenn sie aufgerufen wird, was geschieht dann das nächste Mal, wenn sie im Produktionscode aufgerufen wird? Oder die nächsten 100 Mal? Sollte sich Ihr Test darum kümmern? Wenn das gefälschte Verhalten von Methoden darauf ausgelegt ist, nur einmalig benutzt zu werden, dann muss Ihr Test eine Antwort auf die Frage bereithalten »Was mache ich nun«, wann immer sich der Produktionscode ändert und die gefälschte Methode aufruft – auch dann, wenn sich Ihr Test gar nicht für die Extra-Aufrufe interessiert. Er ist nun stärker an interne Implementierungsdetails gekoppelt.

Um das zu lösen, kann das Isolation-Framework dem Verhalten eine voreingestellte »Klebrigkeit« (»Stickiness«) hinzufügen. Sobald Sie einer Methode sagen, sich auf eine bestimmte Weise zu verhalten (etwa `false` zurückzugeben), wird sie sich *immer* so verhalten, bis Sie ihr etwas anderes mitteilen (alle zukünftigen Aufrufe werden `false` zurückgeben, auch wenn Sie sie 100 Mal aufrufen). Das entbindet den Test vom Wissen, wie die Methode sich später verhalten soll, wenn das für den Zweck des aktuellen Tests nicht mehr von Bedeutung ist.

6.4.4 Komplexe Syntax

Bei manchen Frameworks ist es schwierig, sich die Standard-Operationen zu merken, auch wenn Sie es schon seit einer Weile benutzen. Im Hinblick auf die Programmierung streut das Sand ins Getriebe. Das kann man vereinfachen durch eine entsprechende Gestaltung der API. Beispielsweise beginnen in FakeItEasy alle möglichen Operationen *immer* mit einem großen `A`. Hier kommt ein Beispiel aus der FakeItEasy-Wiki, `https://github.com/FakeItEasy/FakeItEasy/wiki`:

```
//das Erzeugen eines Fakes beginnt mit einem A:
var lollipop = A.Fake<ICandy>
var shop = A.Fake<ICandyShop>

// To set up a call to return a value is also simple:
//das Setzen des Verhaltens einer Methode beginnt mit einem A:
A.CallTo(() => shop.GetTopSellingCandy()).Returns(lollipop);

//die Verwendung eines Argument-Matchers beginnt mit einem A:
    A.CallTo(() => foo.Bar(A<string>.Ignored,
        "second argument")).Throws(new Exception));
```

```
// Use your fake as you would an actual instance of the faked type.
var developer = new SweetTooth();
developer.BuyTastiestCandy(shop);

// Asserting uses the exact same syntax as when configuring calls,
// no need to teach yourself another syntax.
//die Überprüfung, ob Methode aufgerufen wurde, beginnt mit einem A:
A.CallTo(() => shop.BuyCandy(lollipop)).MustHaveHappened();
```

Das gleiche Konzept existiert in Typemock Isolator, wo alle API-Aufrufe mit dem Wort `Isolate` beginnen.

Dieser einzige Zugangspunkt erleichtert es, mit dem richtigen Wort zu beginnen und dann die in die IDE eingebauten Features von Intellisense zu nutzen und den nächsten Schritt herauszubekommen.

In NSubstitute müssen Sie daran denken, `Substitute` zum Erzeugen von Fakes, erweiterte Methoden realer Objekte zum Verifizieren oder Ändern von Verhalten und `Arg<T>` zum Einsatz von Argument-Matchern zu nutzen.

6.5 Zusammenfassung

Die Isolation-Frameworks zerfallen in zwei Kategorien: eingeschränkte und uneingeschränkte Frameworks. Abhängig von der Plattform, auf der sie laufen, kann ein Framework mehr oder weniger Fähigkeiten haben und bei der Auswahl des Frameworks ist es wichtig zu verstehen, was es kann oder nicht kann.

In .NET verwenden uneingeschränkte Frameworks die Profiling-APIs, während die meisten eingeschränkten Frameworks den Code zur Laufzeit erzeugen und kompilieren, genauso wie Sie es manuell mit handgeschriebenen Mocks und Stubs machen.

Isolation-Frameworks, die die Werte der Zukunftssicherheit und der Benutzerfreundlichkeit unterstützen, können Ihr Leben im Land der Unit Tests einfacher machen, während es diejenigen erschweren, die das nicht tun.

Das war's! Wir haben die Kerntechniken zum Schreiben von Unit Tests untersucht. Der nächste Teil des Buches beschäftigt sich mit der Verwaltung von Testcode, dem Arrangieren der Tests und dem Gestalten von Mustern für Tests, auf die Sie sich verlassen können, die einfach zu warten und die klar zu verstehen sind.

Teil III

Der Testcode

Dieser Teil des Buches behandelt die Techniken zum Verwalten und Organisieren von Unit Tests, um sicherzustellen, dass die Qualität von Unit Tests in den alltäglichen Projekten hoch ist.

Kapitel 7 beschäftigt sich zunächst mit der Rolle des Unit Testings als Teil eines automatisierten Build-Prozesses. Dann folgen verschiedene Techniken zur Organisation der unterschiedlichen Arten der Tests in Bezug auf Kategorien (Geschwindigkeit, Typ), mit dem Ziel, das zu erreichen, was ich die »sichere grüne Zone« nenne. Es erklärt auch, wie man eine Test-API oder die Test-Infrastruktur für die eigene Anwendung »wachsen« lässt.

In Kapitel 8 werden wir einen Blick auf die drei wesentlichen Säulen von guten Unit Tests werfen – Lesbarkeit, Wartbarkeit und Vertrauenswürdigkeit – und erkunden die Techniken, die das unterstützen. Wenn Sie nur ein Kapitel dieses Buches lesen würden, dann sollte es Kapitel 8 sein.

In diesem Teil:

Kapitel 7

Testhierarchie und Organisation

Dieses Kapitel behandelt

- Testläufe als Teil automatisierter, nächtlicher Builds
- die Verwendung kontinuierlicher Integration für automatisierte Builds
- die Organisation von Tests als Teil einer Solution
- die Diskussion von Testklassen-Vererbungsmustern

Unit Tests sind für eine Applikation genauso wichtig wie der Produktions-Quellcode. Wie bei regulärem Code auch müssen Sie sorgfältig überlegen, wo die Tests in Bezug auf den zu testenden Code liegen sollen, sowohl in physischer als auch in logischer Hinsicht. Wenn Sie Unit Tests, die Sie so sorgfältig geschrieben haben, an die falsche Stelle legen, dann kann es sein, dass sie nicht laufen.

Ganz ähnlich kann es sein, dass Sie mit einem Testcode enden, der entweder nicht wartbar oder schwierig zu verstehen ist, wenn Sie sich keine Möglichkeiten einfallen lassen, um Teile Ihrer Tests wiederzuverwenden, Hilfsmethoden für das Testen entwickeln oder Testhierarchien verwenden.

Dieses Kapitel geht diese Themen mit Mustern und Richtlinien an, die Ihnen dabei helfen werden, die Art und Weise zu gestalten, wie Ihre Tests aussehen, sich anfühlen und laufen werden, und das wird sich darauf auswirken, wie gut sie mit dem Rest Ihres Codes und mit anderen Tests zusammenspielen werden.

Wo die Tests platziert sind, hängt davon ab, wo sie gebraucht werden und wer sie ausführen wird. Es gibt zwei verbreitete Szenarien: Die Tests laufen als Teil des automatisierten Build-Prozesses und die Tests werden lokal von den Entwicklern auf ihren Maschinen laufen gelassen. Der automatisierte Build-Prozess ist sehr wichtig, deshalb ist er es, auf den wir uns konzentrieren werden.

7.1 Automatisierte Builds, die automatisierte Tests laufen lassen

Die Leistungsfähigkeit des automatisierten Build-Prozesses sollte nicht ignoriert werden. Ich automatisiere meinen Build- und Auslieferungs-Prozess seit mehr als einem Jahrzehnt, und das ist eine der besten Dinge, die Sie unternehmen können, um Ihr Team produktiver zu machen und schneller ein Feedback zu erhalten. Wenn Sie planen, Ihr Team agiler zu machen, und es so ausstatten wollen, dass es Anforderungsänderungen handhaben kann, sobald diese bei Ihnen aufschlagen, dann müssen Sie in der Lage sein, das Folgende zu tun:

- Nehmen Sie eine kleine Änderung an Ihrem Code vor.
- Lassen Sie alle Tests laufen, um sich zu vergewissern, dass keine existierende Funktionalität kaputtgegangen ist.
- Stellen Sie sicher, dass sich Ihr Code immer noch gut integrieren lässt und dass er keine anderen Projekte stört, von denen Sie abhängig sind.
- Erstellen Sie ein auslieferbares Paket Ihres Codes und installieren Sie es automatisch auf Knopfdruck.

Wahrscheinlich werden Sie verschiedene Typen von Build-Konfigurationen und Build-Skripten benötigen, um diese Aufgaben zu bewältigen. Build-Skripte sind kleine Teile von Skripten, die neben Ihrem Code im Source Control hausen und in voller Kenntnis der Version sind, da sie im Source Control mit dem Quellcode Ihres Produkts leben. Sie werden eingebettet in die kontinuierliche Build-Konfiguration eines Integrations-Servers.

Einige dieser Build-Skripte werden Ihre Tests laufen lassen, insbesondere diejenigen, die unmittelbar, nachdem Sie Ihren Code ins Source-Control-System eingecheckt haben, ausgeführt werden. Die Ausführung der Tests lässt Sie oder irgendwen anderen im Projekt wissen, ob irgendeine existierende oder neue Funktionalität zerstört wurde. Sie integrieren Ihren Code in andere Projekte. Ihre Tests zeigen Ihnen, ob Sie die Kompilierung des Codes oder der Dinge, die logisch von Ihrem Code abhängen, stören. Indem Sie das automatisch beim Check-In durchführen, starten Sie einen Prozess, der allgemein als kontinuierliche Integration bekannt ist. Was das bedeutet, werde ich in Abschnitt 7.1.2 erläutern.

Gewöhnlich würde es das Folgende bedeuten, wenn Sie Ihren Code persönlich integrieren würden:

1. Sie kopieren die neueste Version des Quellcodes aller Entwickler aus dem Source Control Repository.
2. Sie versuchen, alles lokal zu kompilieren.
3. Sie lassen alle Tests lokal laufen.
4. Sie bessern alles aus, was nicht funktioniert.
5. Sie checken Ihren Quellcode ein.

Sie können Werkzeuge in Form von automatisierten Build-Skripten und kontinuierlichen Integrations-Servern verwenden, um diese Arbeit zu automatisieren.

Ein automatisierter Build-Prozess kombiniert alle diese Schritte unter einem einzelnen logischen Schirm, den man sich vorstellen kann als »wie wir hier ein Code-Release durchführen«. Dieser Build-Prozess ist eine Sammlung von Build-Skripten, automatischen Triggern, einem Server, möglicherweise ein paar Build-Beauftragten (die die Arbeit erledigen) und der Übereinkunft im Team, auf diese Weise zu arbeiten.

Die Übereinkunft schließt ein, dass darauf geachtet wird, dass alle die Warnungen und die benötigten Schritte beachten und einhalten, damit all das kontinuierlich und weitgehend automatisch abläuft. Jedenfalls so automatisch, wie es sachdienlich ist (es mag nicht sachdienlich sein, automatisch in die Produktion auszuliefern, ohne dass darüber ein Mensch wacht). Wenn irgendetwas in diesem Prozess scheitert, dann kann der Build Server alle relevanten Parteien über ein *Build Break* informieren.

Um es klar auszudrücken: Ein Build-Prozess ist ein logisches Konzept, das Build-Skripte, Build Integration Server, Build Trigger und ein gemeinsames Team-Verständnis, wie der Code angewendet und integriert wird, umfasst.

7.1.1 Die Anatomie eines Build-Skripts

Letztendlich habe ich eigentlich immer mehrere Build-Skripte für jeweils spezielle Zwecke. Diese Art des Aufbaus gestattet eine bessere Wartbarkeit und Kohärenz des Build-Prozesses und umfasst gewöhnlich diese Skripte:

- Ein Build-Skript zur kontinuierlichen Integration (Continuous Integration, CI)
- Ein Skript für das nächtliche Build
- Ein Skript für die Verteilung (Deployment)

Ich mag es, sie zu trennen, denn ich behandle Build-Skripte wie kleine Code-Funktionen, die mit Parametern und der aktuellen Version des Sourcecodes aufgerufen werden können. Der Aufrufer dieser Funktionen (Skripte) ist der CI-Server.

Üblicherweise wird ein CI-Skript zumindest den aktuellen Quellcode im Debug Mode kompilieren und alle Unit-Tests ausführen. Möglicherweise wird es auch andere Tests, soweit sie schnell sind, laufen lassen. Ein CI-Build-Skript ist dazu da, in der kleinstmöglichen Zeit ein Maximum an Informationen auszugeben. Je schneller es ist, desto eher wissen Sie, dass Sie wahrscheinlich nichts beschädigt haben, und können wieder an die Arbeit zurückkehren.

Ein nächtliches Build braucht normalerweise länger. Ich stoße es gerne nach einem CI-Build an, um noch mehr Feedback zu erhalten, aber ich warte darauf nicht zu ungeduldig und kann mit dem Programmieren weitermachen, während es läuft. Es braucht länger, weil es auch all die Aufgaben erledigen soll, die für das CI-Build irrelevant oder zu unbedeutend waren, um sie in einen kurzen CI-Feedback-Zyklus einzuschließen. Diese Aufgaben können so ziemlich alles umfassen, meist jedoch gehören das Kompilieren im Release-Modus, das Ausführen aller langsamen Tests und möglicherweise auch die Verteilung auf die Testumgebung für den nächsten Tag dazu.

Ich nenne sie nächtliche Builds, aber sie können vielfach im Laufe des Tages ausgeführt werden. Mindestens aber laufen sie einmal in der Nacht. Sie geben mehr Feedback, aber sie brauchen mehr Zeit dazu.

Ein Verteilungs(Deployment)-Build-Skript ist gewöhnlich ein Auslieferungsmechanismus. Es wird vom CI-Server angestoßen und kann etwas so Einfaches wie ein `xcopy` zu einem Remote-Server sein oder auch etwas so Kompliziertes wie die Verteilung auf Hunderte von Servern, die Reinitialisierung von Azure oder von Amazon-Elastic-Compute-Cloud(EC2)-Instanzen und das Zusammenführen von Datenbanken.

Alle Builds informieren den Benutzer meist via E-Mail, wenn sie scheitern, aber das ultimative Ziel der Benachrichtigung ist der Aufrufer der Build-Skripte: der CI-Server.

Es gibt viele Tools, die Ihnen helfen können, ein automatisiertes Build-System zu erzeugen. Einige sind frei oder Open Source, andere sind kommerziell. Hier sind ein paar Tools, die Sie in Betracht ziehen können.

Für Build-Skripte:

- NAnt (`nant.sourceforge.net`)
- MSBuild (`www.infoq.com/articles/MSBuild-1`)
- FinalBuilder (`www.FinalBuilder.com`)
- Visual Build Pro (`www.kinook.com`)
- Rake (`http://rake.rubyforge.org`)

Für CI-Server:

- CruiseControl.NET (`cruisecontrol.sourceforge.net`)
- Jenkins (`http://jenkins-ci.org`)
- Travis CI (`http://about.travis-ci.org/docs/user/getting-started`)
- TeamCity (`JetBrains.com`)
- Hudson (`http://hudson-ci.org`)
- Visual Studio Team Foundation Service (`http://tfs.visualstudio.com`)
- ThoughtWorksGo (`www.thoughtworks-studios.com/go-agile-release-management`)
- CircleCI (`https://circleci.com`), wenn Sie ausschließlich über `github.com` arbeiten
- Bamboo (`www.atlassian.com/software/bamboo/overview`)

Einige CI-Server haben ein eingebautes Feature, das die Erzeugung Build-Skript-bezogener Aufgaben ermöglicht. Ich versuche, die Finger davon zu lassen, denn ich möchte, dass die Aktionen meiner Build-Skripte versionsbezogen (oder versionskontrolliert) sind, damit ich immer zu jeder beliebigen Version des Quellcodes zurückgehen kann, und die Build-Aktionen auf diese Version zutreffen.

Von diesen Tools sind FinalBuilder für die Build-Skripte und TeamCity für die CI-Server meine Favoriten. Wenn ich nicht FinalBuilder (läuft nur unter Windows) benutzen kann, setze ich Rake ein, denn ich hasse es, XML für das Build-Management zu verwenden. Das macht es sehr schwierig, die Build-Skripte zu warten. Rake ist XML-frei, wohingegen MSBuild oder NAnt Ihnen so viel XML einschenken, dass Sie monatelang im Schlaf von XML-Tags träumen werden. Jedes Tool in diesen Listen tut sich dabei hervor, eine Sache ganz besonders gut zu machen, wobei TeamCity versucht, immer mehr eingebaute Aufgaben hinzuzufügen, was meiner Meinung nach dazu führt, dass weniger wartbare Builds erzeugt werden.

7.1.2 Das Anstoßen von Builds und Integration

Ich habe die CI bereits kurz erläutert, aber lassen Sie uns das ein wenig offizieller tun. Der Begriff *kontinuierliche Integration* (*Continuous Integration*) meint wörtlich, das automatisierte Build und den Integrationsprozess kontinuierlich laufen zu lassen. Beispielsweise könnten Sie ein bestimmtes Build-Skript jedes Mal laufen lassen, wenn jemand Quellcode eincheckt, oder alle 45 Minuten, oder wenn ein anderes Build-Skript durchgelaufen ist.

Die wichtigsten Aufgaben eines CI-Servers sind diese:

- ein Build-Skript aufgrund bestimmter Ereignisse anzustoßen
- einen Build-Skript-Kontext und Daten wie die Version, den Quellcode, Artefakte aus anderen Builds, Build-Skript-Parameter und Ähnliches bereitzustellen
- einen Überblick über die Build-Vergangenheit und -Metriken zu geben
- den aktuellen Status aller aktiven und inaktiven Builds bereitzustellen

Lassen Sie uns als Erstes die Trigger untersuchen. Ein Trigger kann ein Build-Skript automatisch starten, wenn bestimmte Ereignisse eintreten, wie etwa Source-Control-Updates, das Verstreichen von Zeit oder der Fehlschlag bzw. Erfolg einer anderen Build-Konfigura-

tion. Sie können mehrere Trigger so konfigurieren, dass sie eine bestimmte Unit of Work im CI-Server starten. Diese Units of Work werden häufig Build-Konfigurationen genannt.

Eine Build-Konfiguration hat Anweisungen, wie etwa das Ausführen einer Kommandozeile, das Kompilieren etc., die sie ausführt. Ich rate dazu, dies auf ein Executable zu begrenzen, das das Build-Skript ausführt, und unter die Quellcodeverwaltung zu stellen, um die Kompatibilität der Aktionen zur aktuellen Version des Quellcodes zu maximieren. In TeamCity können Sie beispielsweise einer Build-Konfiguration, die Sie gerade erzeugen, Build-Schritte hinzufügen. Ein Build-Schritt kann von unterschiedlicher Art sein. Das Ausführen einer DOS-Kommandozeile ist eine solche Art. Eine andere könnte das Kompilieren einer .NET-`.sln`-Datei sein. Ich persönlich bleibe bei einem einfachen Kommandozeilen-Build-Schritt. In dieser Kommandozeile führe ich eine Batch-Datei oder ein Build-Skript aus, das unter Quellcodeverwaltung des Build-Agenten steht.

Eine Build-Konfiguration kann einen Kontext haben. Das kann viele Dinge einschließen, aber meist gehört eine aktuelle Momentaufnahme (Snapshot) des Quellcodes aus der Quellcodeverwaltung dazu. Es kann auch das Aufsetzen von Umgebungsvariablen beinhalten, die das Build-Skript verwendet, oder direkte Parameter via Kommandozeile. Das Endresultat der Ausführung eines Build-Skripts sind Artefakte. Dabei kann es sich um Binärdateien, Konfigurationsdateien oder jede andere Art von Dateien handeln.

Ein Build-Kontext kann eine Historie haben. Sie können erkennen, wann es lief, wie lange es brauchte und wann es zum letzen Mal erfolgreich durchlief. Vielleicht können Sie auch sehen, wie viele Tests liefen und welche Tests fehlschlugen. Die Details der Historie hängen vom CI-Server ab.

Ein CI-Server hat gewöhnlich ein Dashboard, das den aktuellen Status der Builds anzeigt. Einige Server bieten auch selbst erstelltes HTML und JavaScript an, das Sie in die internen Intranet-Seiten Ihrer Firma einbetten können, um den Status benutzerspezifisch anzuzeigen. Einige CI-Server stellen Integration-Tools oder maßgeschneiderte Tools zur Verfügung, die auf dem Desktop laufen, kontinuierlich den Status überwachen und Sie benachrichtigen, wenn Builds, für die Sie sich interessieren, schiefgehen.

Mehr Informationen zur Build-Automatisierung

Es gibt eine Reihe weiterer guter Vorgehensweisen, über die Sie vielleicht mehr erfahren möchten, aber das steht nicht im Fokus dieses Buches. Wenn Sie mehr über die kontinuierliche Auslieferung lesen möchten, empfehle ich *Continuous Delivery* von Jez Humble und David Farley (Addison-Wesley, 2010) und *Continuous Integration* von Paul Duvall, Steve Matyas und Andrew Glover (Addison-Wesley, 2007). Vielleicht sind Sie auch an meinem eigenen Buch namens *Beautiful Builds* zu diesem Thema interessiert. *Beautiful Builds* ist mein Versuch, eine Mustersprache von verbreiteten Build-Prozess-Lösungen und -Problemen zu erschaffen. Es wohnt auf der Seite `www.BeautifulBuilds.com`.

7.2 Testentwürfe, die auf Geschwindigkeit und Typ basieren

Es ist einfach, die Tests laufen zu lassen, die Laufzeiten zu prüfen und anhand dessen zu bestimmen, welche Integrationstests und welche Unit Tests sind. Wenn Sie das tun, legen Sie sie an verschiedene Stellen. Sie brauchen sich nicht in separaten Testprojekten zu befinden; unterschiedliche Ordner und Namespaces sollten ausreichen.

Abbildung 7.1 zeigt eine einfache Ordnerstruktur, die Sie innerhalb Ihrer Visual-Studio-Projekte nutzen können.

Je nachdem, welche Build-Software und welches Unit-Testing-Framework sie benutzen, finden es manche Firmen leichter, die Testprojekte nach Unit Tests und Integrationstests zu unterscheiden. Das vereinfacht die Verwendung von Kommandozeilen-Tools, die eine ganze Testgruppe, die nur eine bestimmte Art von Tests enthält, akzeptieren und ausführen. Abbildung 7.2 zeigt, wie Sie die zwei unterschiedlichen Arten von Testprojekten innerhalb einer einzigen Solution aufsetzen.

Auch wenn Sie bisher noch kein automatisiertes Build-System implementiert haben, ist das Separieren der Unit Tests von den Integrationstests eine gute Idee. Das Vermischen der zwei Testarten kann schwerwiegende Konsequenzen haben, wie etwa, dass man Ihre Tests gar nicht laufen lässt, was Sie als Nächstes sehen werden.

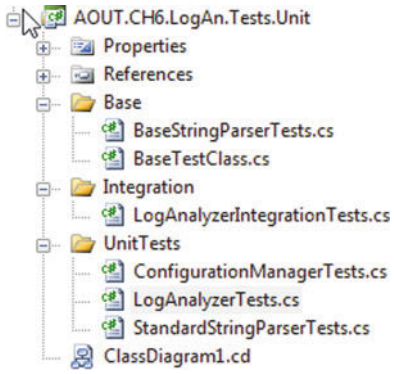


Abb. 7.1: Integrationstests und Unit Tests können in verschiedenen Ordnern und Namespaces liegen, aber im gleichen Projekt bleiben. Basisklassen haben ihre eigenen Ordner.

Abb. 7.2: Die Unit-Testing- und Integration-Projekte sind für das LogAn-Projekt spezifisch und haben unterschiedliche Namespaces.

7.2.1 Der menschliche Faktor beim Trennen von Unit und Integrationstests

Ich empfehle die Trennung der Unit Tests von den Integrationstests. Falls Sie das nicht tun, ist das Risiko groß, dass man die Tests nicht oft genug laufen lässt. Wenn aber die Tests existieren, warum sollte man die Tests dann nicht hinreichend oft laufen lassen? Ein Grund ist, dass Entwickler manchmal faul sind oder unter enormem Zeitdruck stehen können.

Wenn ein Entwickler die neueste Version des Quellcodes kopiert und bemerkt, dass einige Unit Tests fehlschlagen, gibt es dafür mehrere verschiedene Gründe:

- Im getesteten Code ist ein Bug.
- Der Test hat ein Problem in der Art und Weise, wie er geschrieben ist.

- Der Test ist nicht mehr relevant.
- Der Test erfordert, dass bestimmte Konfigurationen ausgeführt werden.

Alle, außer dem letzten Punkt, sind gute Gründe für den Entwickler, anzuhalten und den Code zu untersuchen. Der letzte ist keine Angelegenheit der Entwicklung; es ist ein Konfigurationsproblem, was oft als weniger wichtig eingeschätzt wird, weil es nur das Ausführen der Tests behindert. Wenn deshalb der Test fehlschlägt, wird der Entwickler das Ergebnis häufig ignorieren und mit anderen Dingen weitermachen. (Sie haben »wichtigere« Dinge zu tun.)

Es ist in vieler Hinsicht ungünstig, solche »versteckten« Integrationstests mit Unit Tests zu vermengen und über Ihr Testprojekt zu verteilen, das dann unbekannte oder nicht erwartete Konfigurationsanforderungen enthält (wie etwa eine Datenbankanbindung). Diese Tests sind nicht so leicht zugänglich, sie verschwenden Zeit und Geld beim Suchen von Problemen, die gar nicht da sind, und sie entmutigen den Entwickler ganz allgemein, dem Satz von Tests noch mal zu vertrauen. So wie ein paar faule Äpfel dafür sorgen, dass alle anderen im Korb auch schlecht aussehen. Das nächste Mal, wenn etwas Ähnliches passiert, schaut der Entwickler vielleicht nicht einmal nach der Ursache für den Fehler, sondern sagt gleich: »Oh, dieser Test schlägt manchmal eben fehl, das ist okay.«

Damit das nicht passiert, können Sie eine sichere grüne Zone (Safe Green Zone) erschaffen.

7.2.2 Die sichere grüne Zone

Verteilen Sie Ihre Integrationstests und Unit Tests an verschiedene Stellen. Dadurch geben Sie den Entwicklern Ihres Teams ein *sicheres grünes Testgebiet* (*Safe Green Test Area*), das nur Unit Tests enthält, woher sie ihre aktuelle Code-Version kopieren können, wo sie alle Tests in diesem Namespace oder Ordner laufen lassen können, und die Tests sollten alle *grün* sein. Wenn einige Tests in der sicheren grünen Zone nicht erfolgreich sind, dann liegt ein echtes Problem zugrunde und kein (fälschlicherweise positives) Konfigurationsproblem in diesem Test.

Das heißt nicht, dass die Integrationstests nicht durchgeführt werden sollten. Aber weil die Ausführung der Integrationstests inhärent länger braucht, ist es wahrscheinlicher, dass die Entwickler die Unit Tests mehrfach am Tag ausführen lassen und die Integrationstests seltener, doch hoffentlich immer noch während des nächtlichen Builds. Die Entwickler können sich darauf konzentrieren, produktiv zu sein und zumindest ein Grundvertrauen zu haben, wenn alle ihre Unit Tests erfolgreich ausgeführt werden. Das nächtliche Build sollte all die automatisierten Aufgaben ans Laufen bringen, damit die Integrationstests störungsfrei ablaufen können.

Zusätzlich bietet Ihnen die Errichtung einer getrennten *Integration Zone* (das Gegenstück zur sicheren grünen Zone) für die Integrationstests nicht nur einen Platz für die Quarantäne solcher Tests, die besonders langsam ablaufen können, sondern Sie erhalten auch einen Ort, an dem Sie Dokumente zu den für die Tests benötigten Konfigurationsarbeiten ablegen können.

Ein automatisiertes Build-System übernimmt all die Konfigurationsarbeit für Sie. Wenn Sie die Tests jedoch lokal ausführen wollen, dann sollten Sie in Ihrer Solution oder in Ihrem Projekt eine Integration Zone haben, die alle benötigten Informationen enthält, um das

Ganze laufen zu lassen, die Sie aber auch überspringen können, wenn Sie nur die schnellen Tests ausführen lassen wollen (in der sicheren grünen Zone).

Nichts von dem spielt jedoch eine Rolle, wenn Sie Ihre Tests nicht im Baum der Quellcodeverwaltung haben, wie Sie als Nächstes sehen werden.

7.3 Stellen Sie sicher, dass die Tests zu Ihrer Quellcodekontrolle gehören

Tests *müssen* ein Teil Ihrer Quellcodekontrolle sein. Der Testcode, den Sie schreiben, muss von Ihrem Quellcode-Kontrollsystem verwaltet werden, genauso wie Ihr Produktionscode. Tatsächlich sollten Sie Ihren Testcode genauso sorgfältig behandeln wie Ihren Produktionscode. Er sollte ein Teil des Zweigs für jede Produktversion sein und er sollte ein Teil des Codes sein, den die Entwickler automatisch erhalten, wenn sie die neueste Version kopieren.

Weil Unit Tests so eng mit dem Code und der API verknüpft sind, sollten sie immer an die Version des Codes, den sie testen, gebunden sein. Das Kopieren der Version 1.0.1 Ihres Produkts kopiert auch die Version 1.0.1 der Tests für Ihr Produkt; Version 1.0.2 Ihres Produkts und dessen Tests werden sich davon unterscheiden.

Die Tatsache, dass Ihre Tests ein Teil des Quellcode-Kontrollbaums sind, sorgt auch dafür, dass Ihr automatisierter Build-Prozess sicherstellen kann, die korrekte Version Ihrer Tests auf die Software anzuwenden.

Da die Tests nun Teil der Quellcodeverwaltung sind, stellt sich die Frage, wo sie abgelegt werden sollen.

7.4 Das Abbilden der Testklassen auf den zu testenden Code

Wenn Sie Testklassen anlegen, dann sollte es Ihnen die Art, wie sie strukturiert und abgelegt werden, erleichtern, das Folgende zu tun:

- Schauen Sie auf ein Projekt und finden Sie alle Tests, die dazugehören.
- Schauen Sie auf eine Klasse und finden Sie alle Tests, die dazugehören.
- Schauen Sie auf eine Methode und finden Sie alle Tests, die dazugehören.

Es gibt mehrere Muster, die Ihnen dabei helfen können. Wir werden nacheinander diese Ziele untersuchen.

7.4.1 Das Abbilden von Tests auf Projekte

Ich erzeuge gerne ein Projekt, das die Tests enthält, gebe ihm den gleichen Namen wie das zu testende Projekt und füge `.UnitTests` an das Ende des Namens an. Wenn ich beispielsweise ein Projekt namens `Osherove.MyLibrary` hätte, dann hätte ich auch ein Testprojekt namens `Osherove.MyLibrary.UnitTests` und eines namens `Osherove.MyLibrary.IntegrationTests` oder irgendeine Variante dieser Idee (siehe als Beispiel dazu Abbildung 7.2). Das mag schwerfällig klingen, aber es ist intuitiv und es erlaubt einem Entwickler, alle Tests für ein bestimmtes Projekt zu finden.

Sie möchten vielleicht die Fähigkeit von Visual Studio nutzen, Ordner unter der Solution anzulegen und diese drei in ihre eigenen Ordner zu gruppieren, aber das ist Geschmackssache.

7.4.2 Das Abbilden von Tests auf Klassen

Es gibt mehrere Wege, die Sie beschreiten können, um die Tests auf eine zu testende Klasse abzubilden. Wir werden uns zwei wichtige Szenarios anschauen: eine eigene Testklasse für jede zu testende Klasse und einzelne Testklassen für komplexe zu testende Methoden.

Hinweis

Das sind die zwei Testklassenmuster, die ich meistens verwende, aber es gibt auch andere. Ich empfehle Ihnen, sich weitere Beispiele dazu im Buch *xUnit Test Patterns: Refactoring Test Code* von Gerard Meszaros anzuschauen.

Eine Testklasse pro Klasse oder Unit of Work im Test

Sie wollen in der Lage sein, alle Tests für eine bestimmte Klasse schnell zu lokalisieren, und die Lösung ist dem vorigen Muster für Projekte sehr ähnlich: Nehmen Sie den Namen der Klasse, für die Sie Tests schreiben wollen, und erzeugen Sie im Testprojekt eine Testklasse mit dem gleichen Namen und einem angehängten `UnitTests`. Für eine Klasse `LogAnalyzer` würden Sie eine Testklasse namens `LogAnalyzer.UnitTests` in Ihrem Testprojekt anlegen.

Beachten Sie den Plural: Dies ist eine Klasse, die mehrere Tests für die zu testende Klasse enthält und nicht nur einen Test. Es ist wichtig, hierbei genau zu sein. Lesbarkeit und Sprache machen eine Menge aus, wenn es um Testcode geht, und falls Sie anfangen, an einer Stelle an allen Ecken und Kanten zu sparen, so werden Sie das auch an anderen Stellen tun, was zu Problemen führen kann.

Das *»Eine Testklasse pro Klasse«-Muster* (ebenfalls in Meszaros Buch *xUnit Test Patterns:Refactoring Test Code* erwähnt) ist das einfachste und am weitesten verbreitete Muster zur Organisation von Tests. Man setzt alle Tests für alle Methoden der zu testenden Klasse in eine große Testklasse. Wenn dieses Muster verwendet wird, können einige Methoden in der zu testenden Klasse so viele Tests haben, dass man die Testklasse schlecht lesen und durchsuchen kann. Manchmal übertönen die Tests für eine Methode die Tests für andere Methoden. Das an sich kann schon ein Zeichen dafür sein, dass der Methoden-Test vielleicht zu viel macht.

Hinweis

Die Lesbarkeit des Tests ist wichtig. Sie schreiben die Tests genauso für die Person, die sie lesen wird, wie für den Computer, der sie ausführen wird. Auf die Aspekte der Lesbarkeit werde ich im nächsten Kapitel eingehen.

Wenn derjenige, der den Test liest, mehr Zeit benötigt, den Testcode zu durchsuchen, als ihn zu verstehen, dann wird der Test »Wartungskopfschmerzen« verursachen, wenn der Code länger und länger wird. Daher denken Sie vielleicht darüber nach, es anders zu machen.

Eine Testklasse pro Feature

Eine Alternative ist das Anlegen einer separaten Testklasse für ein bestimmtes Feature (das könnte so klein wie eine Methode sein). Das *»Eine Testklasse pro Feature«*-Muster wird ebenfalls in Meszaros Buch erwähnt. Wenn es scheint, dass Sie viele Testmethoden haben, die es schwierig machen, Ihre Testklasse zu lesen, dann finden Sie die Methode oder Gruppe von Methoden, deren Tests die anderen Tests für diese Klasse übertönen, und erzeugen Sie dafür eine eigene Testklasse mit einem Namen, der sich auf das Feature bezieht.

Angenommen, eine Klasse namens `LoginManager` hat eine Methode `ChangePassword`, die Sie testen möchten, aber sie hat so viele Testfälle, dass Sie diese in eine eigene Testklasse ausgliedern möchten. Dann würden Sie letztlich zu zwei Testklassen kommen: `LoginManagersTests`, die alle anderen Tests enthält; und `LoginManagerTestsChangePassword`, die nur die Tests für die Methode `ChangePassword` enthält.

7.4.3 Das Abbilden von Tests auf bestimmte Methoden

Über die Vergabe von lesbaren und verständlichen Testnamen hinaus ist es Ihr Hauptziel, alle Testmethoden für eine bestimmte zu testende Unit of Work leicht finden zu können, weshalb Sie Ihren Testmethoden aussagekräftige Namen geben sollten. Sie können den Beginn der öffentlichen Methodennamen als Teil des Testnamens verwenden.

Sie könnten einen Test `ChangePassword_scenario_expectedbehaviour` nennen. Diese Namenskonvention wurde in Kapitel 2 vorgestellt (Abschnitt 2.3.2). Es gibt Fälle, in denen Sie die in den vorangegangenen Kapiteln näher beschriebenen Techniken der Injektion, wie das Extrahieren der Interfaces oder das Überschreiben von virtuellen Methoden, nicht in Ihrem Produktionscode anwenden wollen. Das geschieht, wenn Sie es mit Querschnittsbelangen zu tun haben.

7.5 Querschnittsbelang-Injektion

Wenn Sie diese Techniken anwenden und es mit Querschnittsbelangen (Cross-Cutting Concerns) zu tun haben wie dem Zeitmanagement, Exceptions oder dem Logging, dann können Sie am Ende mit Code dastehen, der weniger lesbar und weniger wartbar ist.

Das Problem bei Querschnittsbelangen wie `DateTime` ist, dass, wenn sie in Ihrer Anwendung vorkommen, sie an sehr vielen Stellen verwendet werden. Sie dann als injizierbare Legosteine aufzubauen, kann dazu führen, dass man Ihren Code zwar sehr leicht testen, ihn aber nur sehr schwer lesen und ihm schlecht folgen kann.

Lassen Sie uns annehmen, dass Ihre Anwendung die aktuelle Zeit zur Terminplanung oder zur Protokollierung benötigt und Sie ebenfalls testen möchten, dass Ihre Anwendung die aktuelle Zeit in ihren Protokollen verwendet.

Dann haben Sie vielleicht diese Art von Code in Ihrem System:

```
public static class TimeLogger
{
    public static string CreateMessage(string info)
    {
```

```
        return DateTime.Now.ToShortDateString() + " " + info;
    }
}
```

Wenn Sie die Testbarkeit verbessern wollen, indem Sie ein Interface `ITimeProvider` einführen, dann müssen Sie dieses Interface überall verwenden, wo `DateTime` benutzt wird. Das ist sehr zeitaufwendig in Anbetracht der Tatsache, dass es schnörkellosere Ansätze gibt.

Der Ansatz, den ich gerne für zeitbasierte Systeme verfolge, besteht darin, eine benutzerdefinierte Klasse namens `SystemTime` anzulegen, und dafür zu sorgen, dass mein ganzer Produktionscode diese Klasse statt der Standardklasse `DateTime` verwendet.

Diese Klasse und der überarbeitete Produktionscode, der sie verwendet, könnte wie das folgende Listing aussehen.

```
public static class TimeLogger
{
    public static string CreateMessage(string info)
    {
    //Produktionscode, der SystemTime verwendet:
        return SystemTime.Now.ToShortDateString() + " " + info;
    }
}

public class SystemTime
{
    private static DateTime _date;

    //SystemTime ermöglicht das Ändern der aktuellen Zeit ...
    public static void Set(DateTime custom)
    {    _date = custom;    }

    //...und das Zurücksetzen der aktuellen Zeit:
    public static void Reset()
    {    _date=DateTime.MinValue;    }

    //SystemTime gibt die wahre Zeit zurück oder die gefälschte,
    //falls eine gesetzt wurde:
    public static DateTime Now
    {
        get
        {
            if (_date != DateTime.MinValue)
            {
                return _date;
```

```
            }
            return DateTime.Now;
        }
    }
}
```

Listing 7.1: Die Verwendung der Klasse `SystemTime`

Der einfache Trick ist hier, dass die Klasse `SystemTime` spezielle Funktionen beinhaltet, die es Ihnen ermöglichen, die aktuelle Zeit systemweit zu ändern. Das bedeutet, dass jeder, der die Klasse `SystemTime` benutzt, genau die Zeit und das Datum sehen wird, die Sie vorgeben.

Das gibt Ihnen eine perfekte Möglichkeit, zu testen, ob die aktuelle Zeit in Ihrem Produktionscode verwendet wird. Sie benötigen nur einen einfachen Test wie den im nächsten Listing.

```
[TestFixture]
public class TimeLoggerTests
{
    [Test]
    public void SettingSystemTime_Always_ChangesTime()
    {
        //setze gefälschtes Datum:
        SystemTime.Set(new DateTime(2000,1,1));

        string output = TimeLogger.CreateMessage("a");

        StringAssert.Contains("01.01.2000", output);
    }

    [TearDown]
    public void afterEachTest()
    {
        //setze Datum am Ende jedes Tests zurück:
        SystemTime.Reset();
    }
}
```

Listing 7.2: Ein Test unter Verwendung von `SystemTime`

Als Bonus brauchen Sie nicht Millionen von Interfaces in Ihre Anwendung zu injizieren. Der Preis, den Sie bezahlen, ist eine einfache `[TearDown]`-Methode in Ihrer Testklasse, die dafür sorgt, dass kein Test die Zeit für andere Tests ändert.

Sie müssen allerdings berücksichtigen, dass die aktuell eingestellte Kultur des Systems (beispielsweise en-US versus en-GB) den Ausgabestring beeinflussen kann. In diesem Fall

können Sie in NUnit auch ein Attribut `CultureInfoAttribute` hinzufügen, um den Test zu zwingen, unter einer bestimmten Kultur zu laufen.

Diese Art der externen Abstraktion eines Querschnittbelangs erlaubt es Ihnen, in Ihrem Produktionscode einen einzelnen gefälschten Brennpunkt anzulegen, statt vieler kleiner. Aber das macht nur Sinn für die Dinge, die quer durch das ganze System verwendet werden. Wenn Sie das für alles benutzen, dann bekommen Sie schließlich ein System, das genauso schwer zu lesen ist wie jenes, das Sie zu vermeiden suchen.

Wenn ich auf dieses Beispiel hinweise, fragen mich viele Entwickler:»Wie sorgen wir dafür, dass jeder diese Klasse verwendet?« Meine Antwort darauf ist, dass ich Code-Reviews durchführe und damit sicherstelle, dass niemand `DateTime` direkt verwendet. Ich versuche, mich nicht zu sehr auf Tools zu verlassen, denn ich glaube, dass man wirklich am meisten lernt, wenn zwei Menschen (oder mehr) eng genug zusammensitzen, sich gegenseitig hören und sehen und zusammenarbeiten und abwechselnd mit derselben Tastatur schreiben und über den Code sprechen. Aber wenn es sich um ein bereits existierendes Projekt handelt, das wir konvertieren, um `SystemTime` zu verwenden, dann mache ich nur eine einfache Textsuche (»Find in Files«) nach Code, der `DateTime` verwendet, und wenn möglich, wende ich ein einfaches »Replace« auf alles an, was gefunden wird. `DateTime` ist so benannt, dass es einfach zu finden und zu ersetzen ist.

Als Nächstes diskutieren wir den Bau einer Test-API für Ihre Anwendung.

7.6 Der Bau einer Test-API für Ihre Applikation

Früher oder später, wenn Sie beginnen, Tests für Ihre Applikationen zu schreiben, sind Sie gezwungen, ein Refactoring durchzuführen, und Sie erzeugen Hilfsmethoden, Hilfsklassen und viele andere Konstrukte (entweder in den Testprojekten oder im zu testenden Code) nur mit dem Ziel der Testbarkeit oder der Lesbarkeit und der Wartbarkeit der Tests.

Hier sind einige Dinge, die Sie vielleicht tun wollen:

- Sie verwenden Vererbung in Ihren Testklassen, um den Code wiederzuverwenden, zur Anleitung und für anderes.
- Sie erzeugen Test-Hilfsklassen und -Hilfsmethoden.
- Sie machen Ihre API den Entwicklern bekannt.

Lassen Sie uns diese drei Punkte nacheinander anschauen.

7.6.1 Die Verwendung von Testklassen-Vererbungsmustern

Eines der stärksten Argumente für objektorientierten Code ist, dass man eine existierende Funktionalität wiederverwenden kann, statt sie wieder und wieder in anderen Klassen nachzubauen – was Andy Hunt und Dave Thomas in *The Pragmatic Programmer* (Addison-Wesley, 1999) das »DRY-Prinzip« (»Don't Repeat Yourself«) genannt haben. Weil die Unit Tests, die Sie in .NET und den meisten objektorientierten Sprachen schreiben, in einem objektorientierten Paradigma liegen, ist es kein Verbrechen, Vererbung in den Testklassen selber zu verwenden. Tatsächlich halte ich Sie dazu an, dies zu tun, wenn Sie einen guten Grund haben. Die Implementierung einer Basisklasse kann helfen, Standardprobleme im Testcode zu entschärfen, indem

- Hilfs- und Fabrikmethoden wiederverwendet werden
- der gleiche Satz von Tests für unterschiedliche Klassen ausgeführt wird (das werden wir genauer betrachten)
- ein gemeinsamer Aufbau- (Setup) oder Abbaucode verwendet wird (das ist ebenfalls für das Integration Testing nützlich)
- Testleitlinien für Entwickler, die von der Basisklasse ableiten wollen, festgelegt werden

Ich werde Ihnen drei Muster vorstellen, die auf der Testklassenvererbung basieren, wobei jedes Muster auf dem vorherigen aufbaut. Ich werde auch erklären, wann Sie welches Muster ggf. verwenden mögen und was die Vor- und Nachteile von jedem sind.

Dies sind die grundlegenden drei Muster:

- Abstrakte Testinfrastruktur-Klasse
- Template-Testklasse
- Abstrakte Testtreiber-Klasse

Wir werden auch einen Blick auf die folgenden Refactoring-Techniken werfen, die Sie anwenden können, wenn Sie die genannten Muster verwenden:

- Das Refactoring in eine Klassenhierarchie
- Die Verwendung von Generics

Abstrakte Testinfrastruktur-Klassenmuster

Das *Abstrakte Testinfrastruktur-Klassenmuster* erzeugt eine abstrakte Testklasse, die die wesentliche, gemeinsame Infrastruktur für die Testklassen, die von ihr ableiten, enthält. Die Szenarien, für die Sie eine solche Basisklasse erzeugen wollen, können sich vom gemeinsamen Aufbau- und Abbaucode bis zu speziellen, benutzerdefinierten Asserts, die in mehreren Testfällen angewendet werden, erstrecken.

Wir schauen uns ein Beispiel an, das es Ihnen erlauben wird, eine Setup-Methode in zwei Testklassen wiederzuverwenden. Hier ist das Szenario: Alle Tests müssen die Standardimplementierung des Loggers in der Applikation so überschreiben, dass statt in eine Datei in den Arbeitsspeicher geschrieben wird (das heißt, dass alle Tests die Logger-Abhängigkeit aufbrechen müssen, um korrekt ablaufen zu können).

Listing 7.3 zeigt diese Klassen:

- *Die* `LogAnalyzer`*-Klasse und -Methode* – Die Klasse und Methode, die Sie testen möchten.
- *Die* `LoggingFacility`*-Klasse* – Die Klasse, die die Logger-Implementierung enthält, die Sie in Ihren Tests überschreiben möchten.
- *Die* `ConfigurationManager`*-Klasse* – Ein weiterer Benutzer von `LoggingFacility`, den Sie später testen werden.
- *Die* `LogAnalyzerTests`*-Klasse und -Methode* – Die ursprüngliche Testklasse und Methode, die Sie schreiben.
- *Die* `ConfigurationManagerTests`*-Klasse* – Eine Klasse, die Tests für `ConfigurationManager` enthält.

```
//Diese Klasse verwendet intern LoggingFacility
public class LogAnalyzer
{
    public void Analyze(string fileName)
    {
        if (fileName.Length < 8)
        {
            LoggingFacility.Log("Dateiname zu kurz:" + fileName);
        }
        //hier Rest der Methode
    }
}

//Eine andere Klasse, die intern LoggingFacility verwendet
public class ConfigurationManager
{
    public bool IsConfigured(string configName)
    {
        LoggingFacility.Log("überprüfe " + configName);
        return result;
    }
}

public static class LoggingFacility
{
    public static void Log(string text)
    {
        logger.Log(text);
    }
    private static ILogger logger;

    public static ILogger Logger
    {
        get { return logger; }
        set { logger = value; }
    }
}

[TestFixture]
public class LogAnalyzerTests
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
```

```
    {
        LogAnalyzer la = new LogAnalyzer();
        la.Analyze("meineleeredatei.txt");
        //Rest vom Test
    }

    [TearDown]
    public void teardown()
    {
        //statische Ressource muss zwischen den Tests
        //zurückgesetzt werden
        LoggingFacility.Logger = null;
    }
}

[TestFixture]
public class ConfigurationManagerTests
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        ConfigurationManager cm = new ConfigurationManager();
        bool configured = cm.IsConfigured("irgendwas");
        //Rest vom Test
    }

    [TearDown]
    public void teardown()
    {
        //statische Ressource muss zwischen den Tests
        //zurückgesetzt werden
        LoggingFacility.Logger = null;
    }
}
```

Listing 7.3: Ein Beispiel, das nicht dem DRY-Prinzip in den Testklassen folgt

Die Klasse `LoggingFacility` wird wahrscheinlich von vielen Klassen verwendet werden. Sie ist so entworfen, dass der Code, der sie verwendet, getestet werden kann, indem sie erlaubt, die Implementation des Loggers über den Property-Setter (die statisch ist) durch eine andere Implementation zu ersetzen.

Es gibt zwei Klassen, die die Klasse `LoggingFacility` intern benutzen, und Sie möchten beide von ihnen testen: die Klasse `LogAnalyzer` und die Klasse `ConfigurationManager`.

Eine Möglichkeit, diesen Code in einen besseren Zustand zu refaktorieren, ist es, eine neue Hilfsmethode zu extrahieren und wiederzuverwenden, um einige Wiederholungen in beiden Testklassen zu entfernen. Beide fälschen die Standardimplementation des Loggers.

Sie könnten eine Test-Basisklasse erzeugen, die die Hilfsmethode enthält, und dann die Methode von jedem Test in den abgeleiteten Klassen aufrufen.

Sie verwenden keine gemeinsame Basis-`[SetUp]`-Methode, da dies die Lesbarkeit der abgeleiteten Klasse beeinträchtigen würde. Stattdessen benutzen Sie eine Hilfsmethode namens `FakeTheLogger()`.

Der ganze Code für die Testklassen wird hier gezeigt.

```
[TestFixture]
public class BaseTestsClass
{
    //in eine gemeinsame, lesbare Hilfsmethode umgebaut
    //die von abgeleiteten Klassen benutzt wird:
    public ILogger FakeTheLogger()
    {
        LoggingFacility.Logger = Substitute.For<ILogger>;
        return LoggingFacility.Logger;
    }

    [TearDown]
    //automatisches Aufräumen für abgeleitete Klassen:
    public void teardown()
    {
        //statische Ressource muss zwischen den Tests
        //zurückgesetzt werden
        LoggingFacility.Logger = null;
    }
}

[TestFixture]
public class ConfigurationManagerTests:BaseTestsClass
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        //rufe Hilfsmethode der Basisklasse auf:
        FakeTheLogger();

        ConfigurationManager cm = new ConfigurationManager();
        bool configured = cm.IsConfigured("irgendwas");
```

```
        //Rest vom Test
    }
}

[TestFixture]
public class LogAnalyzerTests : BaseTestsClass
{
    [Test]
    public void Analyze_EmptyFile_ThrowsException()
    {
        //rufe Hilfsmethode der Basisklasse auf:
        FakeTheLogger();

        LogAnalyzer la = new LogAnalyzer();
        la.Analyze("meineleeredatei.txt");
        //Rest vom Test
    }
}
```

Listing 7.4: Eine refaktorierte Lösung

Wenn Sie in der Basisklasse eine Methode mit `Setup`-Attribut verwendet hätten, würde sie nun automatisch vor jedem Test in allen abgeleiteten Klassen ausgeführt werden. Das Hauptproblem, das Sie nun in die abgeleiteten Testklassen eingeschleppt hätten, besteht darin, dass jemand, der den Code liest, nicht mehr so einfach verstehen kann, was passiert, wenn `Setup` aufgerufen wird. Derjenige muss in der Setup-Methode der Basisklasse nachschauen, um zu sehen, was die abgeleiteten Klassen standardmäßig erhalten. Dies führt zu weniger gut lesbaren Tests, weshalb Sie eine Hilfsmethode benutzen, die aussagekräftiger ist.

Auch das verringert die Lesbarkeit in gewisser Weise, denn Entwickler, die Ihre Basisklasse benutzen, haben nur eine geringe Dokumentation oder Idee davon, welche API sie von Ihrer Basisklasse benutzen sollen. Daher empfehle ich, diese Technik so selten zu nutzen wie möglich – aber auch nicht seltener. Genauer ausgedrückt hatte ich nie einen Grund, der gut genug war, um mehrere Basisklassen zu benutzen. Mit einer einzigen Basisklasse war die Lesbarkeit immer besser, wenn auch die Wartbarkeit geringfügig schlechter war. Sie sollten auch *nicht* mehr als eine einzige Vererbungsschicht in Ihren Tests haben. Dieser Schlamassel wird schneller unlesbar, als Sie sagen können: »Warum schlägt mein Build fehl?«

Lassen Sie uns eine interessantere Anwendung der Vererbung anschauen, um ein verbreitetes Problem zu lösen.

Template-Testklassenmuster

Nehmen wir an, Sie wollen dafür sorgen, dass diejenigen, die bestimmte Arten von Klassen im Code testen, nicht vergessen, einen bestimmten Satz von Unit Tests dafür durchzuspielen, während sie die Klassen entwickeln. Dabei könnte es sich etwa um Netzwerk-Code mit Paketen, Sicherheits-Code, datenbankbezogenen Code oder einfach nur herkömmlichen

Parser-Code handeln. Der Punkt ist, dass Sie wissen, dass, wenn im Code an dieser Art von Klassen gearbeitet wird, einige Tests existieren müssen, denn diese Art von Klassen müssen einen bekannten Satz von Diensten mit ihrer API bereitstellen.

Das *Template-Testklassenmuster* ist eine abstrakte Klasse, die abstrakte Testmethoden enthält, die die abgeleiteten Klassen implementieren müssen. Die treibende Kraft hinter diesem Muster ist die Notwendigkeit, den ableitenden Klassen vorschreiben zu können, welche Tests sie immer implementieren müssen. Wenn Sie in Ihrem System Klassen mit Interfaces haben, dann können sie gute Kandidaten für dieses Muster sein. Ich verwende es, wenn ich eine Hierarchie von Klassen habe, die wächst, und jedes neue Mitglied einer abgeleiteten Klasse ungefähr die gleichen Ideen implementiert.

Stellen Sie sich ein Interface als einen »Leistungsvertrag« vor, bei dem die gleiche Endleistung von allen abgeleiteten Klassen erwartet wird, das Resultat aber auf verschiedenen Wegen erreicht werden kann. Ein Beispiel für solch einen Leistungsvertrag könnte ein Satz von Parsern sein, die alle Textanalysemethoden implementieren, die sich gleich verhalten, aber mit verschiedenen Eingabetypen umgehen.

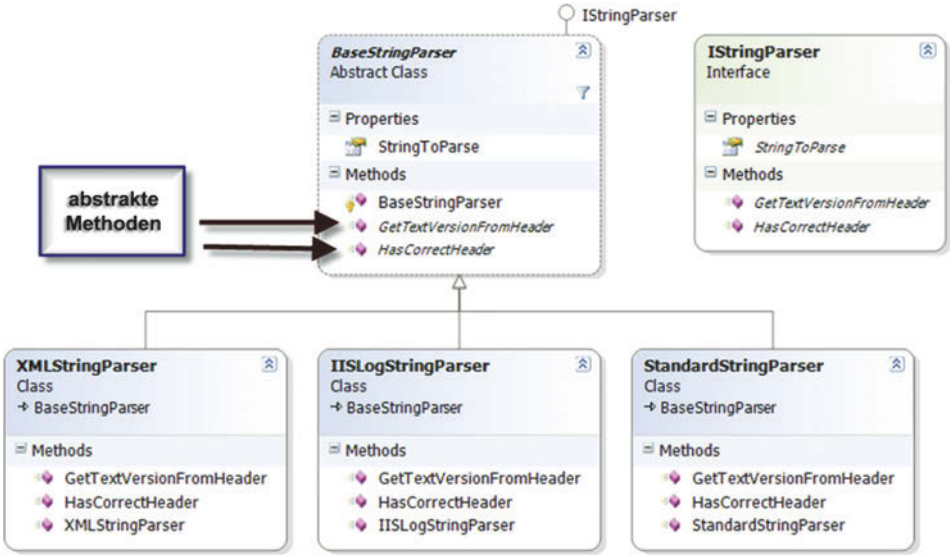


Abb. 7.3: Eine typische Vererbungshierarchie, die Sie testen wollen, umfasst eine abstrakte Klasse und Klassen, die davon ableiten.

Entwickler vernachlässigen oder vergessen häufig, alle erforderlichen Tests für einen bestimmten Fall zu schreiben. Das Vorhandensein einer Basisklasse für jeden Satz von Klassen mit identischen Interfaces kann dabei helfen, einen grundsätzlichen Testvertrag zu entwerfen, den alle Entwickler in ihren abgeleiteten Klassen umsetzen müssen.

Nun zu einem realen Szenario. Nehmen Sie an, Sie wollten das Objektmodell in Abbildung 7.3 testen.

`BaseStringParser` ist eine abstrakte Basisklasse, von der andere Klassen ableiten, um eine Funktionalität für verschiedene Arten von Zeichenketteninhalten zu implementieren. Von

jedem Zeichenkettentyp (XML-Strings, IIS-Log-Strings, Standard-Strings) können Sie eine Art von Versionsinfo erhalten (Metadaten für den String, die vorher eingefügt wurden). Sie können die Versionsinfo aus einem benutzerdefinierten Header (die ersten paar Zeilen des Strings) erhalten und prüfen, ob dieser Header für den Zweck Ihrer Applikation gültig ist. Die Klassen `XMLStringParser`, `IISLogStringParser` und `StandardStringParser` leiten von dieser Basisklasse ab und implementieren die Methoden mit der jeweiligen Logik für ihre spezifischen String-Typen.

Der erste Schritt beim Testen solch einer Hierarchie ist das Schreiben eines Testsatzes für eine der abgeleiteten Klassen (wir nehmen an, die abstrakte Klasse enthält keine Logik, die getestet werden muss). Danach müssten Sie die gleiche Art von Tests für die anderen Klassen schreiben, die die gleiche Funktionalität haben.

Das nächste Listing zeigt die Tests für die Klasse `StandardStringParser`, mit denen Sie zunächst beginnen können, bevor Sie ein Refactoring für Ihre Testklassen durchführen, um das Template-Testklassenmuster zu nutzen.

```
[TestFixture]
public class StandardStringParserTests
{
    //definiert die Parser-Fabrikmethode (1)
    private StandardStringParser GetParser(string input)
    {
        return new StandardStringParser(input);
    }

    [Test]
    public void GetStringVersionFromHeader_SingleDigit_Found()
    {
        string input = "header;version=1;\n";
        //verwendet die Fabrikmethode (2)
        StandardStringParser parser = GetParser(input);

        string versionFromHeader = parser.GetTextVersionFromHeader();
        Assert.AreEqual("1",versionFromHeader);
    }

    [Test]
    public void GetStringVersionFromHeader_WithMinorVersion_Found()
    {
        string input = "header;version=1.1;\n";
        //verwendet die Fabrikmethode (2)
        StandardStringParser parser = GetParser(input);

        //Rest vom Test
    }
```

```
    [Test]
    public void GetStringVersionFromHeader_WithRevision_Found()
    {
        string input = "header;version=1.1.1;\n";
        StandardStringParser parser = GetParser(input);

        //Rest vom Test
    }
}
```

Listing 7.5: Ein Entwurf einer Testklasse für `StandardStringParser`

Beachten Sie, wie Sie die Hilfsmethode `GetParser()` **(1)** einsetzen, um die Erzeugung des Parser-Objekts wegzurefaktorieren **(2)**, was Sie in allen Tests verwenden. Sie benutzen die Hilfsmethode und nicht eine Setup-Methode, weil der Konstruktor den zu parsenden String als Input übernimmt, weshalb jeder Test in der Lage sein muss, eine Version des Parsers zu erzeugen, die er mit seinem spezifischen Input testen kann.

Wenn Sie beginnen, Tests für die anderen Klassen in der Hierarchie zu schreiben, dann werden Sie die gleichen Tests wiederholen wollen. Alle anderen Parser sollten das gleiche nach außen gerichtete Verhalten zeigen: das Holen der Header-Version und die Validierung des Headers. Wie sie das tun, unterscheidet sich, aber die Semantik des Verhaltens bleibt gleich. Das bedeutet, dass Sie für jede Klasse, die von `BasicStringParser` ableitet, die gleichen grundlegenden Tests schreiben würden, und nur der Typ der zu testenden Klasse würde sich ändern.

Lassen Sie uns vorne anfangen und sehen, wie Sie abgeleiteten Testklassen leicht vorschreiben können, für welche Tests die Ausführung entscheidend ist. Das folgende Listing zeigt ein einfaches Beispiel dazu (`IStringParser` finden Sie im Code des Buches zu GitHub).

```
[TestFixture]
//die Test-Template-Klasse:
public abstract class TemplateStringParserTests
{
    public abstract
        void TestGetStringVersionFromHeader_SingleDigit_Found();

    public abstract
        void TestGetStringVersionFromHeader_WithMinorVersion_Found();

    public abstract
        void TestGetStringVersionFromHeader_WithRevision_Found();
}

[TestFixture]
```

```
//die abgeleitete Klasse:
public class XmlStringParserTests : TemplateStringParserTests
{
    protected IStringParser GetParser(string input)
    {
        return new XMLStringParser(input);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_SingleDigit_Found()
    {
        IStringParser parser = GetParser("<Header>1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1",versionFromHeader);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_WithMinorVersion_Found()
    {
        IStringParser parser = GetParser("<Header>1.1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1.1",versionFromHeader);
    }

    [Test]
    public override
        void TestGetStringVersionFromHeader_WithRevision_Found()
    {
        IStringParser parser = GetParser("<Header>1.1.1</Header>");

        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual("1.1.1",versionFromHeader);
    }
}
```

Listing 7.6: Eine Template-Testklasse zum Testen von String-Parsern

Abbildung 7.4 zeigt die Visualisierung dieses Codes, wenn Sie zwei abgeleitete Klassen haben. Beachten Sie, dass `GetParser()` nur eine Standardmethode ist, die in den abgeleiteten Klasse beliebig benannt werden kann.

Ich fand diese Technik in vielen Situationen hilfreich, nicht nur als Entwickler, sondern auch als Architekt. Als Architekt war ich in der Lage, die Entwickler mit einer Liste essenzieller Testklassen zu versorgen, von denen abgeleitet werden musste, und Ratschläge zu geben, welche Art von Tests sie als Nächstes schreiben sollten. In dieser Situation ist es entscheidend, dass die Testnamen verständlich sind. Ich nutze das Wort `Test` als Präfix für abstrakte Methoden in der Basisklasse, damit es für diejenigen, die sie in abgeleiteten Klassen überschreiben, leichter ist herauszufinden, was zu überschreiben wichtig ist.

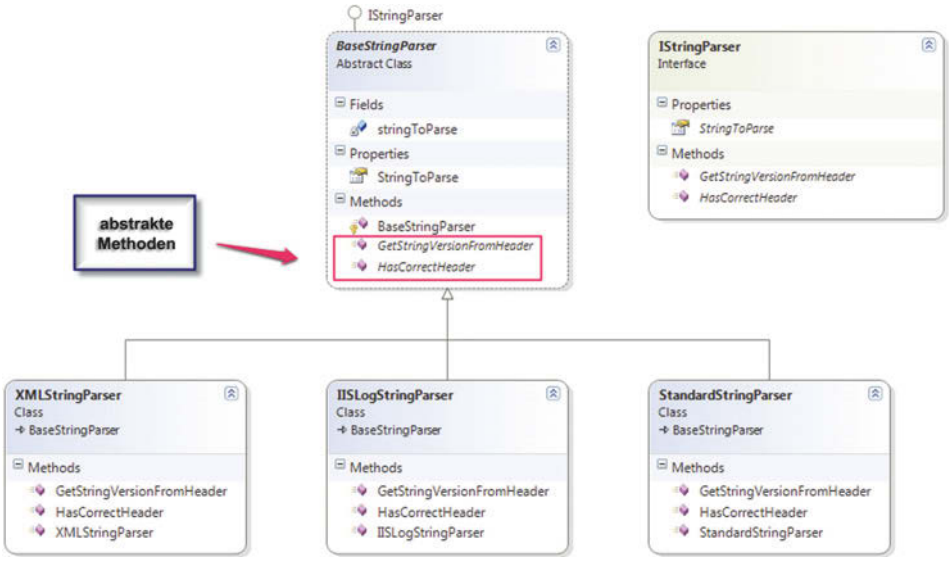


Abb. 7.4: Ein Template-Test-Muster gewährleistet, dass die Entwickler keine wichtigen Tests vergessen. Die Basisklasse enthält abstrakte Tests, die die abgeleiteten Klassen implementieren müssen.

Aber wie wäre es, wenn Sie der Basisklasse beibringen könnten, noch mehr zu tun?

Das abstrakte »Fill In The Blanks«-Test-Driver-Klassenmuster

Das abstrakte Test-Driver-Klassenmuster – ich nenne es gerne »fill in the blanks« (»fülle die Lücken«) – führt die vorherige Idee weiter, indem die Tests in der Basisklasse selbst implementiert und mit Methoden-Hooks versehen werden, die die abgeleiteten Klassen implementieren müssen.

Es ist entscheidend, dass Ihre Tests keinen Klassen-Typ explizit testen, sondern dass sie gegen ein Interface oder eine Basisklasse in Ihrem Produktionscode testen.

Hier kommt ein Beispiel einer solchen Basisklasse.

```
public abstract class FillInTheBlanksStringParserTests
{
    //abstrakte Fabrikmethode, die die Rückgabe eines Interface
    //erfordert:
    protected abstract IStringParser GetParser(string input);
```

```
//drei abstrakte Eingabe-Methoden, um abgeleiteten Klassen Daten
//in einem bestimmten Format anzubieten:
protected abstract string HeaderVersion_SingleDigit { get; }
protected abstract string HeaderVersion_WithMinorVersion { get; }
protected abstract string HeaderVersion_WithRevision { get; }

//vordefinierte, erwartete Ausgabe für die abgeleiteten
//Klassen (falls benötigt):
public const string EXPECTED_SINGLE_DIGIT = "1";
public const string EXPECTED_WITH_REVISION = "1.1.1";
public const string EXPECTED_WITH_MINORVERSION = "1.1";

[Test]
public void TestGetStringVersionFromHeader_SingleDigit_Found()
{
    string input = HeaderVersion_SingleDigit;
    IStringParser parser = GetParser(input);

    //vordefinierte Testlogik, die abgeleiteten Input verwendet:
    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_SINGLE_DIGIT,versionFromHeader);
}

[Test]
public void TestGetStringVersionFromHeader_WithMinorVersion_Found()
{
    string input = HeaderVersion_WithMinorVersion;
    IStringParser parser = GetParser(input);

    //vordefinierte Testlogik, die abgeleiteten Input verwendet:
    string versionFromHeader = parser.GetStringVersionFromHeader();
    Assert.AreEqual(EXPECTED_WITH_MINORVERSION,versionFromHeader);
}

[Test]
public void TestGetStringVersionFromHeader_WithRevision_Found()
{
    string input = HeaderVersion_WithRevision;
    IStringParser parser = GetParser(input);
```

```
        //vordefinierte Testlogik, die abgeleiteten Input verwendet:
        string versionFromHeader = parser.GetStringVersionFromHeader();
        Assert.AreEqual(EXPECTED_WITH_REVISION,versionFromHeader);
    }
}

[TestFixture]
//abgeleitete Klasse, die die Lücken füllt:
public class
    StandardStringParserTests : FillInTheBlanksStringParserTests
{
    //füllt das richtige Format für diese Anforderung:
    protected override string HeaderVersion_SingleDigit
    {get  {
        return string.Format("header\tversion={0}\t\n",
                             EXPECTED_SINGLE_DIGIT);
    }}

    protected override string HeaderVersion_WithMinorVersion
    {get  {
        return string.Format("header\tversion={0}\t\n",
                             EXPECTED_WITH_MINORVERSION);
    }}

    protected override string HeaderVersion_WithRevision
    {get  {
        return string.Format("header\tversion={0}\t\n",
                             EXPECTED_WITH_REVISION);
    }}

    protected override IStringParser GetParser(string input)
    {
        //trägt den richtigen Typ der zu testenden Klasse ein:
        return new StandardStringParser(input);
    }
}
```

Listing 7.7: Eine »fill in the blanks«-Basistestklasse

In diesem Listing haben Sie keine Tests in der abgeleiteten Klasse. Sie werden alle geerbt. In der abgeleiteten Klasse könnten Sie aber weitere Tests hinzufügen, wenn das Sinn macht. Abbildung 7.5 zeigt die Vererbungsreihe, die Sie gerade erzeugt haben.

Wie modifizieren Sie existierenden Code mithilfe dieses Musters? Das ist unser nächstes Thema.

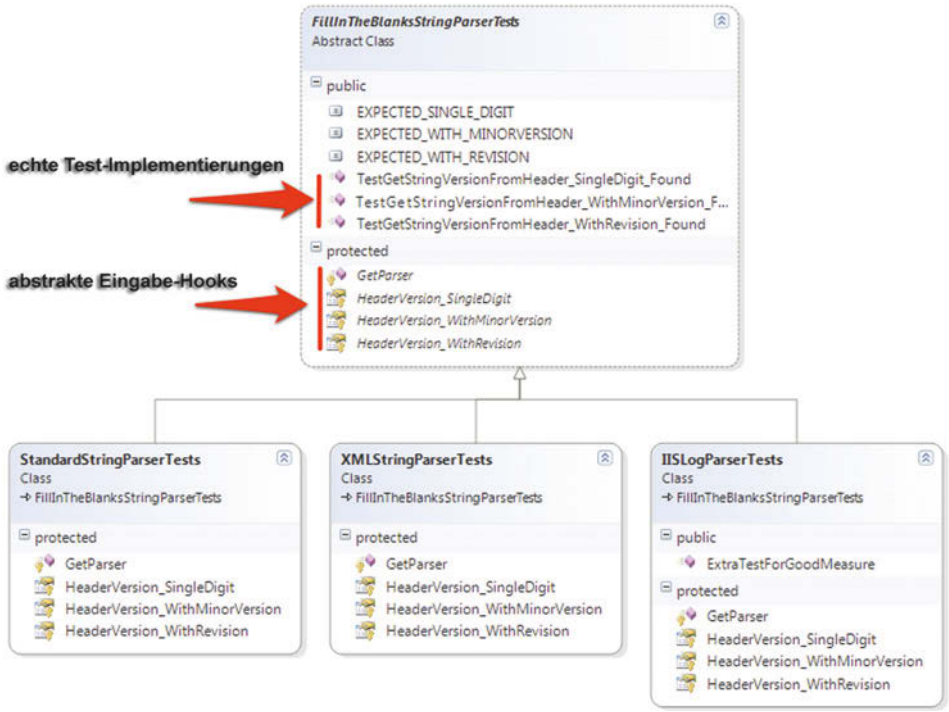


Abb. 7.5: Eine Standard-Testklassenhierarchie-Implementierung. Die meisten der Tests liegen in der Basisklasse, aber können ihre eigenen spezifischen Tests hinzufügen.

Das Refactoring Ihrer Testklasse in eine Testklassenhierarchie

Den meisten Entwicklern schweben nicht diese Vererbungsmuster vor, wenn sie mit dem Schreiben ihrer Tests beginnen. Stattdessen schreiben sie die Tests ganz normal wie in Listing 7.7. Die Schritte, um die Tests in eine Basisklasse zu verlegen, sind recht einfach, insbesondere wenn Sie IDE-Refactoring-Tools verwenden können, wie die in Eclipse, IntelliJ IDEA oder Visual Studio (JetBrains ReSharper, Teleriks JustCode oder Refactor! von DevExpress).

Hier ist eine Liste von möglichen Schritten für das Refactoring Ihrer Testklasse:

1. Refaktorieren: Extrahieren Sie die Superklasse.
   - Erzeugen Sie eine Basisklasse (`BaseXXXTests`).
   - Verschieben Sie die Fabrikmethoden (wie `GetParser`) in die Basisklasse.
   - Verschieben Sie alle Tests zur Basisklasse.
   - Extrahieren Sie die erwarteten Ausgaben in öffentliche Felder der Basisklasse.
   - Extrahieren Sie die Testeingaben in abstrakte Methoden oder Properties, die die abgeleiteten Klassen anlegen werden.
2. Refaktorieren: Machen Sie die Fabrikmethoden abstrakt und lassen Sie sie Interfaces zurückgeben.

3. Refaktorieren: Finden Sie alle Stellen in den Testmethoden, wo explizit Klassentypen verwendet werden, und ändern Sie das, damit statt dieser Typen Interfaces benutzt werden.
4. In der abgeleiteten Klasse implementieren Sie die abstrakten Fabrikmethoden und geben die expliziten Typen zurück.

Sie können auch die .NET Generics verwenden, um die Vererbungsmuster zu erzeugen.

Eine Variante, um mithilfe der .NET Generics die Testhierarchie zu implementieren

Sie können Generics als Teil der Test-Basisklasse verwenden. Auf diese Weise brauchen Sie nicht einmal eine einzige Methode in den abgeleiteten Klassen zu überschreiben; deklarieren Sie nur den Typ, den Sie testen wollen. Das nächste Listing zeigt sowohl die Generic-Version der Test-Basisklasse als auch eine von ihr abgeleitete Klasse.

```
//ein Beispiel der gleichen Idee unter Verwendung von Generics
public abstract class GenericParserTests<T>
    where T:IStringParser    //definiert generischen Constraint
                             //für Parameter (1)
{
    protected abstract string GetInputHeaderSingleDigit();

    //gibt Variable von generischem Typ zurück statt Interface(2):
    protected T GetParser(string input)
    {
        //gibt generischen Typ zurück (3):
        return (T) Activator.CreateInstance(typeof (T), input);
    }

    [Test]
    public void GetStringVersionFromHeader_SingleDigit_Found()
    {
        string input = GetInputHeaderSingleDigit();
        T parser = GetParser(input);

        bool result = parser.HasCorrectHeader();
        Assert.IsFalse(result);
    }

    //weitere Tests
    //...
}

//ein Beispiel eines Tests, der von einer
//generischen Basisklasse erbt
[TestFixture]
```

```
// erbt von einer generischen Basisklasse (4):
public class StandardParserGenericTests
    :GenericParserTests<StandardStringParser>
{
    protected override string GetInputHeaderSingleDigit()
    {
        //gibt benutzerdefinierten Input für den aktuell zu
        //testenden Typ zurück (5):
        return "Header;1";
    }
}
```

Listing 7.8: Die Implementierung der Testfallvererbung mit .NET Generics

Mehrere Dinge ändern sich in der Generic-Implementierung der Hierarchie:

- Die `GetParser`-Fabrikmethode **(2)** braucht nicht länger überschrieben zu werden. Erzeugen Sie das Objekt mithilfe von `Activator.CreateInstance` (was das Anlegen von Objekten erlaubt, ohne ihren Typ zu kennen) und übergeben Sie die String-Eingabeargumente an den Konstruktor als Typ T **(3)**.
- Die Tests selber verwenden das `IStringParser`-Interface nicht, sondern stattdessen den Generic-Typ `T` **(4)**.
- Die Generic-Klassendeklaration enthält die `where`-Klausel, die festlegt, dass der Typ `T` der Klasse das Interface `IStringParser` implementieren muss **(1)**.
- Die abgeleitete Klasse gibt einen benutzerdefinierten Input an den Basistest zurück **(5)**.

Insgesamt sehe ich keinen größeren Nutzen in der Verwendung von Generic-Basisklassen. Jeder mögliche Performancegewinn ist für diese Tests bedeutungslos, aber ich überlasse es Ihnen, zu beurteilen, was für Ihre Projekte Sinn macht. Es ist mehr eine Frage des Geschmacks als alles andere.

Lassen Sie uns mit etwas völlig anderem weitermachen: der Infrastruktur-API in Ihren Testprojekten.

7.6.2 Der Entwurf von Test-Hilfsklassen und -Hilfsmethoden

Während Sie Ihre Tests schreiben, werden Sie viele einfache Hilfsmethoden entwerfen, die letztlich in Ihren Testklassen enden oder auch nicht. Diese Hilfsklassen werden einen großen Teil Ihrer Test-API ausmachen und sie können sich als ein einfaches Objektmodell herausstellen, das Sie verwenden können, wenn Sie Ihre Tests entwickeln.

Das kann auf die folgenden Typen von Hilfsmethoden hinauslaufen:

- Fabrikmethoden für Objekte, deren Erzeugung komplex ist oder die routinemäßig von Ihren Tests erzeugt werden
- Systeminitialisierungsmethoden (etwa solche Methoden, die den Systemzustand vor dem Testen aufsetzen oder die die Logging-Einrichtungen ändern, um Stub-Logger einzusetzen)
- Objektkonfigurationsmethoden (beispielsweise Methoden, die den internen Zustand eines Objekts setzen, etwa indem der Klient einer Transaktion auf ungültig gesetzt wird)

- Methoden, die externe Ressourcen wie Datenbanken, Konfigurationsdateien oder Test-Eingabedateien aufsetzen oder von ihnen lesen (beispielsweise eine Methode, die eine Textdatei mit allen von Ihnen gewünschten Permutationen und den erwarteten Resultaten einliest, wenn Sie den Input an eine bestimmte Methode übergeben). Dies wird häufiger in Integrations- oder Systemtests verwendet.
- Spezielle Assert-Hilfsmethoden, die ein Assert auf eine komplexe Sache anwenden oder auf eine, die innerhalb des Systemzustands wiederholt getestet wird. (Wenn etwas ins System-Log geschrieben wurde, könnte die Methode darüber informieren, dass X, Y und Z `true` sind, nicht aber G.)

Schließlich könnten Sie Ihre Hilfsmethoden in diese drei Arten von Hilfsklassen refaktorieren:

- Spezielle Assert-Hilfsklassen, die alle benutzerdefinierten Assert-Methoden enthalten
- Spezielle Fabrikklassen, die alle Fabrikmethoden enthalten
- Spezielle Konfigurationsklassen oder Datenbank-Konfigurationsklassen, die integrationsartige Aktionen enthalten

In der Open-Source-Welt von .NET gibt es ein paar hilfreiche Utility-Frameworks, die gute Beispiele geben, wie man etwas richtig schön machen kann. Ein Beispiel ist das *Fluent Assertions Framework*, das unter `https://github.com/dennisdoomen/FluentAssertions` gefunden werden kann.

Allein, dass Sie solche Hilfsmethoden haben, garantiert noch nicht, dass sie irgendwer auch benutzt. Ich war bei vielen Projekten dabei, wo die Entwickler das Rad dauernd neu erfanden und Hilfsmethoden entwickelten, von denen sie nicht wussten, dass sie bereits vorhanden waren.

Als Nächstes werden Sie erfahren, wie Sie Ihre API bekannt machen.

7.6.3 Machen Sie Ihre API den Entwicklern bekannt

Es ist unbedingt notwendig, dass diejenigen, die Tests schreiben, von den verschiedenen APIs wissen, die bereits beim Schreiben der Applikation und ihrer Tests entwickelt wurden. Sie können auf mehrere Arten erreichen, dass Ihre API benutzt wird:

- Lassen Sie Teams von zwei Leuten die Tests gemeinsam schreiben (zumindest hin und wieder), wobei einer die existierenden APIs kennt und dem anderen beim Schreiben neuer Tests ihre Vorteile und den Code, der verwendet werden kann, erklärt.
- Legen Sie ein kurzes Dokument (nicht länger als ein paar Seiten) oder einen Spickzettel an, der die Typen von vorhandenen APIs auflistet – und wo sie zu finden sind. Sie können kurze Dokumente für bestimmte Teile Ihres Testing-Frameworks schreiben (beispielsweise zu den APIs, die sich auf die Datenschicht beziehen) oder ein globales für die gesamte Anwendung. Wenn es nicht kurz ist, wird es jedoch keiner pflegen.

  Ein möglicher Weg, um zu erreichen, dass es auf dem Laufenden bleibt, ist die Automatisierung der Herstellung:

  - Verwenden Sie bei der Benennung der API-Helfer einen definierten Satz von Präfixen oder Postfixen (z. B. helper`[irgendwas]`).
  - Verwenden Sie ein spezielles Tool, das die API-Namen und -Standorte durchsucht und ein Dokument erzeugt, das sie einschließlich der Stellen, wo sie zu finden sind,

auflistet. Oder geben Sie einige einfache Anweisungen, die das Tool durchsuchen kann, indem Sie entsprechende Kommentare einfügen.

- Automatisieren Sie die Erzeugung dieses Dokuments als Teil des automatischen Build-Prozesses.

- Diskutieren Sie Änderungen der API während der Team-Meetings – ein oder zwei Sätze zu den Hauptänderungen und wo die signifikanten Stellen zu finden sind. Auf diese Art weiß das Team um die Bedeutung und die Leute haben es im Hinterkopf.
- Besprechen Sie das Dokument mit neuen Mitarbeitern während deren Einarbeitung.
- Führen Sie Test-Reviews durch (zusätzlich zu Code-Reviews), damit die Tests gewissen Standards hinsichtlich der Lesbarkeit, Wartbarkeit und Korrektheit genügen und dass die richtigen APIs verwendet werden, wenn man sie benötigt. Mehr zu diesem Vorgehen finden Sie in meinem Blog für Software-Leader: `http://5whys.com/blog/step-4-start-doing-code-reviews-seriously.html`.

Wenn Sie einer oder mehreren dieser Empfehlungen folgen, kann das Ihrem Team helfen, produktiv zu sein und eine gemeinsame Sprache zu finden, die es beim Schreiben der Tests benutzen kann.

7.7 Zusammenfassung

Lassen Sie uns zurückblicken und schauen, was Sie aus diesem Kapitel mitnehmen können.

- Was auch immer Ihre Tests sind – und wie auch immer Sie sie durchführen –, automatisieren Sie sie und verwenden Sie eine automatisierte Build-Prozedur, um sie so häufig wie möglich, bei Tag oder in der Nacht, laufen zu lassen. Verteilen Sie das Resultat so kontinuierlich und so häufig wie möglich.
- Trennen Sie die Integrationstests von den Unit Tests (die langsamen Tests von den schnellen), damit Ihr Team eine sichere grüne Zone besitzt, in der alle Tests erfolgreich ablaufen müssen.
- Entwerfen Sie die Tests nach Projekt und nach Typ (Unit Tests gegenüber Integratiost-Tests, langsame gegenüber schnellen Tests) und separieren Sie sie in verschiedene Kategorien, Ordner oder Namespaces (oder in alle davon). Ich verwende gewöhnlich alle drei Arten der Unterteilung.
- Verwenden Sie eine Testklassenhierarchie, um den gleichen Satz von Tests auf eine Vielzahl von verwandten, zu testenden Typen in einer Hierarchie anzuwenden oder auf Typen, die ein gemeinsames Interface oder eine gemeinsame Basisklasse besitzen.
- Verwenden Sie Hilfs- und Dienstklassen statt Hierarchien, wenn die Testklassenhierarchie die Tests weniger lesbar macht, insbesondere, wenn es eine gemeinsame Setup-Methode in der Basisklasse gibt. Verschiedene Leute haben verschiedene Meinungen dazu, wann man was verwenden sollte, aber Lesbarkeit ist gewöhnlich das Hauptargument, Hierarchien nicht zu verwenden.
- Machen Sie Ihre API Ihrem Team bekannt. Wenn Sie das nicht tun, werden Sie Zeit und Geld verlieren, weil die Teammitglieder unwissentlich viele der APIs immer wieder neu erfinden werden.

Kapitel 8

Die Säulen guter Unit Tests

Dieses Kapitel behandelt

- das Schreiben vertrauenswürdiger Tests
- das Schreiben wartbarer Tests
- das Schreiben lesbarer Tests
- die Diskussion von Namenskonventionen für Unit Tests

Egal, wie Sie Ihre Tests organisieren oder wie viele Sie haben, sie sind wenig wert, wenn Sie ihnen nicht vertrauen, sie nicht warten oder lesen können. Die Tests, die Sie schreiben, sollten drei Eigenschaften besitzen, die zusammengenommen Ihre Tests zu »guten« Tests machen:

- *Vertrauenswürdigkeit* – Entwickler *möchten* vertrauenswürdige Tests ausführen und sie werden die Ergebnisse mit Zutrauen akzeptieren. Vertrauenswürdige Tests enthalten keine Bugs und testen die richtigen Dinge.
- *Wartbarkeit* – Unwartbare Tests sind ein Albtraum, denn sie können Projektpläne ruinieren oder Sie riskieren, die Tests zu verlieren, wenn das Projekt auf einen aggressiveren Zeitplan gesetzt wird. Entwickler werden einfach aufhören, Tests, deren Änderungen zu lange dauern oder zu häufig bei jeder kleinen Änderung des Produktionscodes sind, zu warten und zu reparieren.
- *Lesbarkeit* – Das bedeutet nicht nur, in der Lage zu sein, einen Test zu lesen, sondern auch herauskriegen zu können, wo ein Problem liegt, wenn der Test falsch zu sein scheint. Ohne Lesbarkeit stürzen die zwei anderen Säulen recht schnell ein. Die Pflege von Tests wird schwieriger und Sie können ihnen nicht mehr trauen, weil Sie sie nicht verstehen.

Dieses Kapitel präsentiert eine Reihe von Verfahren, die in Zusammenhang mit einer dieser drei Säulen stehen und die Sie bei Test-Reviews anwenden können. Gemeinsam stellen diese drei Säulen sicher, dass Ihre Zeit gut genutzt wird. Lassen Sie eine davon umstürzen, laufen Sie Gefahr, die Zeit aller zu verschwenden.

8.1 Das Schreiben vertrauenswürdiger Tests

Es gibt mehrere Indizien dafür, dass ein Test vertrauenswürdig ist. Wenn ein Test erfolgreich abläuft, sagen Sie nicht: »Ich gehe mit dem Debugger durch den Code, um sicher zu sein.« Sie vertrauen darauf, dass er erfolgreich ist und dass der Code, den er testet, im entsprechenden Szenario funktioniert. Wenn der Test fehlschlägt, dann sagen Sie nicht zu sich selber: »Oh, er sollte ja eigentlich fehlschlagen«, oder »Das heißt nicht, dass der Code nicht funktioniert.« Sie glauben, dass da ein Problem in Ihrem Code ist und nicht in Ihrem Test. Kurz gefasst ist ein vertrauenswürdiger Test einer, der Ihnen das Gefühl gibt, zu wissen, was vor sich geht, und etwas daran ändern zu können.

In diesem Kapitel werde ich Leitlinien und Techniken einführen, die Ihnen dabei helfen, das Folgende zu tun:

- Zu entscheiden, wann Tests entfernt oder geändert werden
- Das Testen der Logik zu vermeiden
- Nur eine einzige Angelegenheit zu testen
- Unit Tests von Integrationstests zu trennen
- So sehr auf Code-Reviews zu drängen wie auf die Code-Abdeckung

Nach meiner Erfahrung kann ich Tests, die diesen Richtlinien folgen, tendenziell eher vertrauen als anderen und dieses Vertrauen wird dabei helfen, die echten Fehler in meinem Code zu finden.

8.1.1 Die Entscheidung, wann Tests entfernt oder geändert werden

Sobald Sie die Tests einmal an Ort und Stelle haben und sie erfolgreich sind, sollten Sie sie im Allgemeinen nicht ändern oder entfernen. Sie dienen als Ihr Sicherheitsnetz und lassen es Sie wissen, sobald irgendetwas kaputtgeht, wenn Sie Ihren Code ändern. Nichtsdestotrotz mag es Augenblicke geben, in denen Sie sich genötigt sehen, vorhandene Tests zu ändern oder zu entfernen. Um zu verstehen, wann die Tests ein Problem verursachen könnten und wann es vernünftig ist, sie zu ändern oder zu entfernen, lassen Sie uns die Gründe für beides betrachten.

Der Hauptgrund, einen Test zu entfernen, ist der, dass er fehlschlägt. Ein Test kann aus mehreren Gründen »plötzlich« fehlschlagen:

- *Produktions-Bugs* – Es gibt einen Bug im zu testenden Produktionscode.
- *Test-Bugs* – Es gibt einen Bug im Test.
- *Semantik- oder API-Änderungen* – Die Semantik des zu testenden Codes hat sich geändert, nicht aber die Funktionalität.
- *Widersprüchliche oder ungültige Tests* – Der Produktionscode wurde geändert, um eine widersprüchliche Anforderung zu reflektieren.

Auch wenn nichts falsch an den Tests oder dem Code ist, gibt es Gründe, die Tests zu ändern oder zu entfernen:

- Um den Test umzubenennen oder zu refaktorieren
- Um doppelte Tests zu eliminieren

Lassen Sie uns schauen, wie Sie mit jedem dieser Fälle umgehen können.

Produktions-Bugs

Ein Produktions-Bug tritt auf, wenn Sie den Produktionscode ändern und ein existierender Test fehlschlägt. Wenn dies tatsächlich an einem Bug im zu testenden Code liegt, dann ist Ihr Test gut und Sie sollten ihn nicht anpacken müssen. Dies ist das beste und am meisten gewünschte Ergebnis des Testens.

Weil das Auftreten von Produktions-Bugs einer der Hauptgründe dafür ist, dass Sie an erster Stelle Unit Tests haben, ist das Einzige, was uns bleibt, den Fehler im Produktionscode zu beheben. Fassen Sie den Test nicht an.

Test-Bugs

Wenn es einen Bug im Test gibt, müssen Sie den Test ändern. Bugs in Tests sind bekanntermaßen schwierig zu finden, weil vorausgesetzt wird, dass die Tests korrekt sind. (Daher mag ich TDD so sehr. Es ist ein zusätzlicher Weg, den Test zu testen und zu sehen, ob er dann fehlschlägt und dann erfolgreich ist, wenn er es sollte.) Ich habe bemerkt, dass Entwickler verschiedene Etappen durchlaufen, wenn sie auf einen Fehler im Test stoßen:

1. *Leugnung* – Der Entwickler sucht weiterhin nach einem Problem im Code selber, ändert ihn und verursacht somit, dass all die anderen Tests auch anfangen, fehlzuschlagen. Der Entwickler baut neue Bugs in den Produktionscode ein, während er nach dem Bug fahndet, der tatsächlich im Test vorhanden ist.
2. *Unterhaltung* – Der Entwickler ruft, wenn möglich, einen anderen Entwickler und sie fahnden gemeinsam nach dem nicht vorhandenen Bug.
3. *Debuggerei* – Der Entwickler debuggt geduldig den Test und entdeckt, dass es da ein Problem im Test gibt. Das kann irgendetwas zwischen einer Stunde und ein paar Tagen dauern.
4. *Akzeptanz und Erleuchtung* – Der Entwickler entdeckt schließlich den Bug und schlägt sich auf die Stirn.

Wenn Sie schließlich den Bug finden und mit dem Reparieren beginnen, ist es wichtig, dafür zu sorgen, dass der Fehler behoben wird und der Test nicht auf magische Weise erfolgreich durchläuft, weil Sie das Falsche testen. Sie müssen Folgendes machen:

1. Fixen Sie den Bug in Ihrem Test.
2. Stellen Sie sicher, dass der Test fehlschlägt, wenn er sollte.
3. Stellen Sie sicher, dass der Test erfolgreich ist, wenn er sollte.

Der erste Schritt, das Reparieren des Tests, ist recht geradlinig. Die beiden nächsten sorgen dafür, dass Sie immer noch die richtige Sache testen und dass Ihren Tests immer noch vertraut werden kann.

Sobald Sie Ihren Test repariert haben, wechseln Sie zum zu testenden Produktionscode und ändern ihn so, dass er den Bug aufweist, den ihr Test finden soll. Das könnte etwa bedeuten, dass Sie eine Zeile auskommentieren oder dass Sie irgendwo einen booleschen Wert ändern. Dann lassen Sie den Test laufen. Wenn der Test fehlschlägt, dann funktioniert er schon mal zur Hälfte. Die andere Hälfte kommt dann in Schritt 3 hinzu. Wenn der Test nicht fehlschlägt, dann testen Sie höchstwahrscheinlich die falsche Sache. (Ich habe schon Entwickler gesehen, die versehentlich Asserts beim Bugfixing aus ihren Tests gelöscht haben. Sie wären überrascht, wie oft das passiert und wie effektiv Schritt 2 darin ist, dies zu verhindern.)

Sobald Sie den Test fehlschlagen sehen, ändern Sie den Produktionscode, damit der Bug nicht mehr existiert. Der Test sollte nun erfolgreich ablaufen. Wenn er das nicht tut, haben Sie entweder immer noch einen Bug in Ihrem Test oder Sie testen die falsche Sache. Sie wollen sehen, wie der Test fehlschlägt und anschließend nach der Korrektur erfolgreich abläuft, damit Sie Gewissheit haben können, dass er dann fehlschlägt, wenn er fehlschlagen soll, bzw. dann erfolgreich ist, wenn er erfolgreich sein soll.

Semantik- oder API-Änderungen

Ein Test kann fehlschlagen, wenn sich der zu testende Produktionscode so ändert, dass ein getestetes Objekt nun anders *verwendet* werden muss, auch wenn es immer noch die gleiche Endfunktionalität besitzt.

Betrachten Sie den einfachen Test im folgenden Listing.

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = new LogAnalyzer();
    Assert.IsFalse(logan.IsValid("abc"));
}
```

Listing 8.1: Ein einfacher Test der Klasse `LogAnalyzer`

Sagen wir, es hat eine Semantikänderung in der Klasse `LogAnalyzer` in Form einer Methode `Initialize` gegeben. Sie müssen nun `Initialize` vor jeder anderen Methode von `LogAnalyzer` aufrufen.

Wenn Sie diese Änderung in den Produktionscode einbauen, dann wird die `Assert`-Zeile des Tests in Listing 8.1 eine Ausnahme werfen, weil `Initialize` nicht aufgerufen wurde. Der Test ist kaputt, aber es ist immer noch ein gültiger Test. Die Funktionalität, die er testet, arbeitet nach wie vor, die Semantik bei der Verwendung des zu testenden Objekts hat sich jedoch geändert.

In diesem Fall müssen Sie den Test ändern und ihn an die neue Semantik anpassen, wie im folgenden Listing gezeigt wird.

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = new LogAnalyzer();
    logan.Initialize();

    Assert.IsFalse(logan.IsValid("abc"));
}
```

Listing 8.2: Der geänderte Test mit der neuen Semantik von `LogAnalyzer`

Die Änderung der Semantik ist für die meisten der schlechten Erfahrungen verantwortlich, die Entwickler beim Schreiben und Pflegen von Unit Tests haben, denn die Last der Teständerungen wird größer und größer, wenn sich die API des zu testenden Codes immer wieder ändert. Das folgende Listing zeigt eine besser wartbare Version des Tests aus Listing 8.2.

```
[Test]
public void SemanticsChange()
{
    LogAnalyzer logan = MakeDefaultAnalyzer();  //verwendet die
                                                //Fabrikmethode (1)
    Assert.IsFalse(logan.IsValid("abc"));
}

public static LogAnalyzer MakeDefaultAnalyzer()
```

```
{
    LogAnalyzer analyzer = new LogAnalyzer();
    analyzer.Initialize();
    return analyzer;
}
```

Listing 8.3: Ein refaktorierter Test, der eine Fabrikmethode verwendet

In diesem Fall verwendet der refaktorierte Test eine Dienst-Fabrikmethode **(1)**. Sie können das Gleiche für andere Tests machen und sie auch die gleiche Dienstmethode verwenden lassen. Dann brauchen Sie nicht all die Tests zu ändern, die dieses Objekt anlegen, sobald sich die Semantik des Erzeugens und Initialisierens des Objekts erneut ändern sollte – Sie müssen nur eine kleine Dienstmethode anpassen. Falls es Sie ermüdet, all diese Fabrikmethoden zu erzeugen, dann empfehle ich Ihnen, einen Blick auf ein Test-Helper-Framework namens AutoFixture zu werfen.

Ich werde darauf im Anhang ein bisschen näher eingehen, aber kurz gesagt kann AutoFixture neben anderen Dingen als schlaue Objektfabrik benutzt werden, um das zu testende Objekt anzulegen, ohne sich allzu sehr mit der Struktur des Konstruktors befassen zu müssen. Um mehr über dieses Framework herauszufinden, suchen Sie im Internet nach »String Calculator Kata with AutoFixture« oder Sie gehen auf die GitHub-Seite von AutoFixture `https://github.com/AutoFixture/AutoFixture`. Ich bin mir noch nicht ganz sicher, ob ich ein eifriger Benutzer von AutoFixture bin (denn das Erzeugen einer Fabrikmethode ist nicht wirklich eine große Sache), aber es ist in jedem Fall einen Blick wert und Sie können selber entscheiden, ob Sie es mögen. So lange Ihre Tests lesbar und wartbar bleiben, spricht nichts dagegen.

Sie werden andere Techniken zur Wartbarkeit später in diesem Kapitel kennenlernen.

Widersprüchliche oder ungültige Tests

Ein Konfliktproblem tritt auf, wenn in den Produktionscode ein neues Feature eingebaut wird, das im direkten Widerspruch zu einem Test steht. Das bedeutet, dass der Test statt eines Bugs eine widersprüchliche Anforderung entdeckt.

Lassen Sie uns ein kurzes Beispiel ansehen. Nehmen wir an, der Kunde fordert von `LogAnalyzer`, keine Dateinamen mit weniger als vier Buchstaben zu erlauben. In diesem Fall soll eine Ausnahme geworfen werden. Dieses Feature wird implementiert und Tests werden geschrieben.

Viel später realisiert der Kunde dann, dass Dateinamen mit drei Buchstaben doch einen Sinn haben, und fordert, dass sie in einer besonderen Weise behandelt werden. Das Feature wird hinzugefügt und der Produktionscode geändert. Dann schreiben Sie neue Tests, damit der Produktionscode keine Ausnahme mehr wirft und die Tests erfolgreich durchlaufen. Plötzlich schlägt ein alter Test (einer mit einem Dateinamen aus drei Buchstaben) fehl, denn er erwartet eine Ausnahme. Ändert man nun den Produktionscode so, dass der Test erfolgreich ausgeführt werden kann, dann führt das dazu, dass jetzt der Test fehlschlägt, der eine besondere Behandlung der Dateinamen mit drei Buchstaben erwartet.

Dieses Entweder-Oder-Szenario, bei dem nur einer von zwei Tests erfolgreich ablaufen kann, dient als Warnung, dass es sich hier um widersprüchliche Tests handeln kann. In diesem Fall müssen Sie als Erstes überprüfen, dass die Tests wirklich im Widerspruch zueinan-

der stehen. Sobald Sie das bestätigen können, müssen Sie entscheiden, welche Anforderung Sie aufrechterhalten wollen. Sie sollten dann die ungültige Anforderung und die dazugehörigen Tests entfernen (und nicht nur auskommentieren). (Ernsthaft, wenn ich jemanden ertappe, der etwas auskommentiert, statt es zu löschen, dann werde ich ein Buch schreiben *Warum Gott die Quellcodekontrolle erfand*.)

Widersprüchliche Tests können manchmal Probleme in den Kundenanforderungen aufzeigen und der Kunde muss möglicherweise die Gültigkeit jeder Anforderung beurteilen.

Das Umbenennen oder Refactoring von Tests

Ein unlesbarer Test ist eher ein Problem als eine Lösung. Er kann sowohl die Lesbarkeit Ihres Produktionscodes behindern als auch Ihr Verständnis für ein Problem, das er findet.

Wenn Sie einem Test begegnen, der einen unklaren oder irreführenden Namen hat oder dessen Wartbarkeit verbessert werden kann, dann ändern Sie den Testcode (aber nicht die Basisfunktionalität des Tests). Listing 8.3 zeigte ein solches Beispiel für das Refactoring eines Tests zur besseren Wartbarkeit, was ihn auch ein gutes Stück lesbarer macht.

Das Entfernen doppelter Tests

Wenn Sie es mit einem Team von Entwicklern zu tun haben, dann ist es üblich, dass Sie auf mehrere Tests für die gleiche Funktionalität stoßen, die von verschiedenen Entwicklern geschrieben wurden. Ich bin aus mehreren Gründen nicht darauf versessen, doppelte Tests zu entfernen:

- Je mehr (gute) Tests Sie haben, desto sicherer können Sie sein, Bugs zu erwischen.
- Sie können die Tests lesen und die Unterschiede beim Testen der gleichen Sache sehen.

Hier sind einige Nachteile von doppelten Tests:

- Es kann schwieriger sein, mehrere unterschiedliche Tests zu pflegen, die die gleiche Funktionalität bieten.
- Manche Tests können von besserer Qualität sein als andere, aber Sie müssen sie alle auf ihre Korrektheit prüfen.
- Mehrere Tests können fehlschlagen, wenn eine einzige Sache nicht funktioniert (das mag vielleicht nicht wirklich unerwünscht sein).
- Ähnliche Tests müssen unterschiedlich benannt werden, denn sonst können die Tests in verschiedene Klassen auseinanderlaufen.
- Eine Vielzahl von Tests kann zu mehr Wartbarkeitsproblemen führen.

Hier sind einige Vorteile:

- Die Tests mögen kleine Unterschiede haben und können als etwas betrachtet werden, das die gleiche Sache leicht unterschiedlich testet. Sie können aber für ein größeres und besseres Bild des zu testenden Objekts sorgen.
- Einige Tests können aussagekräftiger als andere sein, weshalb mehr Tests die Chancen auf eine gute Lesbarkeit verbessern können.

Obwohl ich, wie erwähnt, nicht darauf versessen bin, doppelte Tests zu entfernen, mache ich es gewöhnlich doch; die Nachteile überwiegen meist die Vorteile.

8.1.2 Vermeiden Sie Logik in Tests

Die Chancen auf Bugs in Ihren Tests wachsen fast exponentiell, wenn Sie mehr und mehr Logik darin einbauen. Ich habe schon eine Menge Tests gesehen, die einfach hätten sein sollen und dynamisch zu Logik ändernden, Zufallszahlen ziehenden, Threads anlegenden, Datei schreibenden Monstern wurden, die eigenständige kleine Test-Engines sind. Leider hat der Autor, weil sie ein `[Test]`-Attribut besitzen, nicht in Betracht gezogen, dass sie Bugs enthalten können, oder er hat sie nicht in einer wartbaren Weise geschrieben. Solche Testmonster verschwenden mehr Zeit zum Debuggen und Verifizieren, als sie einsparen.

Aber alle Monster fangen klein an. Häufig schaut ein Guru in der Firma auf einen Test und beginnt nachzudenken: »Wie wäre es, wenn wir die Methode in eine Schleife legen und Zufallszahlen als Input erzeugen? Auf diese Weise finden wir bestimmt viel mehr Bugs!« Und das werden Sie, ganz besonders in Ihren Tests. Test-Bugs gehören zu den ärgerlichsten Dingen für Sie als Entwickler, weil Sie die Ursache für einen fehlschlagenden Test so gut wie nie im Test selber suchen. Ich sage damit nicht, dass solche Tests überhaupt keinen Wert haben. In der Tat ist es wahrscheinlich, dass ich solche Tests selber schreibe. Aber ich würde sie nicht *Unit Tests* nennen. Ich nenne sie *Integrationstests*, denn sie haben wenig Kontrolle über das, was sie testen, und vermutlich kann nicht darauf vertraut werden, dass ihre Ergebnisse wahrheitsgemäß sind (mehr dazu im Abschnitt zur Trennung zwischen Integrationstests und Unit Tests später in diesem Kapitel).

Wenn sich etwas aus der folgenden Auflistung in Ihrem Unit Test befindet, dann enthält Ihr Test Logik, die nicht dort sein sollte:

- `switch`-, `if`- oder `else`-Anweisungen
- `foreach`-, `for`- oder `while`-Schleifen

Ein Test, der Logik enthält, prüft meist mehr als eine Sache auf einmal, was nicht empfehlenswert ist, denn der Test ist dann weniger lesbar und eher brüchig. Aber die Testlogik fügt auch Komplexität hinzu, die einen versteckten Bug enthalten kann.

Als allgemeine Regel sollten Unit Tests eine Reihe von Methodenaufrufen mit zusätzlichen Assert-Aufrufen sein, aber ohne Ablaufsteuerung, nicht einmal mit `try-catch`-Anweisungen. Alles, was komplizierter ist, verursacht die folgenden Probleme:

- Der Test ist schwieriger zu lesen und zu verstehen.
- Der Test ist schwierig zu wiederholen. (Denken Sie an einen Multithread-Test oder an einen Test mit Zufallszahlen, der plötzlich fehlschlägt.)
- Die Wahrscheinlichkeit steigt, dass der Test einen Bug enthält oder die falsche Sache testet.
- Die Benennung des Tests kann schwieriger sein, weil er mehrere Dinge tut.

Im Allgemeinen ersetzen Monster-Tests die ursprünglichen, einfacheren Tests und das macht es schwieriger, die Bugs im Produktionscode zu finden. Wenn Sie gezwungen sind, einen Monster-Test zu entwerfen, dann sollte er *hinzugefügt* werden und keine existierenden Tests *ersetzen*. Außerdem sollte er in einem Projekt liegen, das explizit für Integrationstests ist und nicht für Unit Tests.

Im Folgenden sehen Sie noch eine andere Art von Logik, die in Unit Tests unbedingt vermieden werden sollte.

```
[Test]
public void ProductionLogicProblem()
{
    string user = "USER";
    string greeting = "GREETING";
    string actual = MessageBuilder.Build(user,greeting);

    Assert.AreEqual(user + greeting, actual);
}
```

Das Problem hier ist, dass der Test das erwartete Resultat im `Assert` dynamisch über eine wenn auch simple Logik erzeugt. Es ist sehr wahrscheinlich, dass der Test sowohl die Logik des Produktionscodes wiederholt als auch die Logik darin (denn die Person, die die Logik schrieb, und die Person, die den Test schreibt, könnten die gleiche Person sein oder die gleichen irrigen Annahmen zum Code machen).

Das heißt, dass irgendwelche Fehler im Produktionscode im Test wiederholt werden könnten und so der Test erfolgreich ist, auch wenn der Bug existiert. Im Beispielcode fehlt ein Leerzeichen im erwarteten Wert des `Assert` und wenn es auch im Produktionscode fehlt, wird der Test erfolgreich durchlaufen.

Es wäre stattdessen besser, den Test mit fest codierten Werten zu schreiben, etwa so:

```
[Test]
public void ProductionLogicProblem()
{
    string actual = MessageBuilder.Build("user","greeting");

    Assert.AreEqual("user greeting", actual);
}
```

Da Sie bereits wissen, wie das Endresultat aussehen sollte, hält nichts Sie davon ab, es auf diese Weise fest in den Test zu codieren. Jetzt interessiert es Sie nicht, wie das Endresultat zustande kam, aber Sie finden heraus, wenn es nicht stimmt. Und Sie haben keine Logik in Ihrem Test, die einen Fehler haben könnte.

Logik kann nicht nur in den Tests gefunden werden, sondern auch in den Hilfsmethoden der Tests, in handgeschriebenen Fakes und Test-Utility-Klassen. Denken Sie daran, jedes Stückchen Logik, das Sie in diesen Stellen einfügen, macht den Code sehr viel schwerer zu lesen und erhöht die Chancen, dass Sie einen Bug in einer Utility-Methode haben, die von Ihren Tests verwendet wird.

Wenn Sie meinen, dass Sie aus bestimmten Gründen eine komplizierte Logik in Ihrer Test-Suite brauchen (obwohl das im Allgemeinen etwas ist, das ich mit Integrationstests mache und nicht mit Unit Tests), dann sollten Sie zumindest dafür sorgen, dass Sie ein paar Tests haben, mit denen Sie die Logik Ihrer Utility-Methoden im Testprojekt testen können. Das wird Ihnen langfristig eine Menge Kopfschmerzen ersparen.

8.1.3 Testen Sie nur einen Belang

Ein Belang (»Concern«) ist, wie bereits zuvor erklärt, ein einzelnes Endresultat einer Unit of Work: ein Rückgabewert, eine Änderung des Systemzustands oder der Aufruf eines Third-Party-Objekts. Wenn Ihr Unit Test beispielsweise Asserts auf mehr als ein einzelnes Objekt setzt, dann testet er vielleicht mehr als einen Belang. Oder wenn er sowohl testet, dass das gleiche Objekt den richtigen Wert zurückgibt, als auch, dass sich der Systemzustand ändert, sodass sich das System jetzt anders verhält, dann testet er wahrscheinlich mehr als einen Belang.

Mehr als einen Belang zu testen, klingt nicht so schlimm, bis Sie beschließen, Ihren Test zu benennen, oder überlegen, was passiert, wenn die Asserts für das erste Objekt fehlschlagen.

Das Benennen eines Tests mag wie eine leichte Aufgabe erscheinen, aber wenn Sie mehr als eine Sache testen, wird es nahezu unmöglich, dem Test einen guten Namen zu geben, der anzeigt, was getestet wird. Sie landen dann bei einem sehr allgemeinen Testnamen, der den Leser dazu zwingt, auch den Testcode zu lesen (mehr dazu im Abschnitt zur Lesbarkeit in diesem Kapitel). Wenn Sie nur einen Belang testen, ist das Benennen des Tests einfach.

Störender ist aber, dass ein fehlgeschlagenes Assert in den meisten Test-Frameworks (NUnit eingeschlossen) einen besonderen Typ von Ausnahme wirft, der vom Test Framework Runner aufgefangen wird. Wenn das Test-Framework diese Ausnahme auffängt, bedeutet das, dass der Test fehlgeschlagen ist. Unglücklicherweise stoppen Ausnahmen mit Absicht die weitere Ausführung des Codes. Die Methode wird mit der Zeile verlassen, in der die Ausnahme geworfen wird. Listing 8.4 zeigt ein Beispiel. Wenn das erste Assert fehlschlägt, wirft es eine Ausnahme, was bedeutet, dass das zweite Assert niemals ausgeführt wird und Sie nie erfahren werden, ob sich das Verhalten des Objekts in Abhängigkeit von seinem Zustand geändert hat. Jedes davon kann und sollte als unterschiedliche Anforderung betrachtet werden. Und jede kann und sollte getrennt und inkrementell Schritt für Schritt implementiert werden.

```
[Test]
public void IsValid_WhenValid_ReturnsTrueAndRemembersItLater()
{
    LogAnalyzer logan = MakeDefaultAnalyzer();

    Assert.IsTrue(logan.IsValid("abc"));
    Assert.IsTrue(logan.WasLastCallValid);
}
```

Listing 8.4: Ein Test mit mehreren Asserts

Betrachten Sie Assert-Fehlschläge als Symptom einer Krankheit. Je mehr Symptome Sie finden können, desto einfacher wird die Diagnose der Krankheit sein. Nach einem Fehlschlag werden nachfolgende Asserts nicht mehr ausgeführt und Sie verpassen es, andere mögliche Symptome zu sehen, die Ihnen wertvolle Daten (Symptome) liefern könnten, die Ihnen helfen würden, den Fokus einzuschränken und das zugrunde liegende Problem zu entdecken.

Der Test in Listing 8.4 müsste eigentlich in zwei getrennte Tests mit guten Namen aufgeteilt werden.

Hier ist noch eine andere Art, die Sache zu betrachten: Wenn das erste Assert fehlschlägt, interessiert Sie dann noch, was in den nächsten passiert? Falls ja, sollten Sie den Test wahrscheinlich auf zwei Unit Tests aufteilen.

Mehrere Belange in einem einzelnen Unit Test zu überprüfen, erhöht die Komplexität, ohne den Nutzen nennenswert zu steigern. Sie sollten zusätzliche Überprüfungen von Belangen in getrennten, unabhängigen Unit Tests laufen lassen, damit Sie sehen können, was wirklich schiefläuft.

8.1.4 Trennen Sie Unit Tests von Integrationstests

In Kapitel 7 habe ich die sichere grüne Zone für Tests vorgestellt. Ich spreche das wieder an, denn es ist sehr wichtig. Wenn Entwickler nicht davon überzeugt sind, dass sie Ihre Tests ohne Konfigurationsaufwand einfach und durchgängig ausführen können, dann werden sie sie auch nicht laufen lassen. Das Refactoring Ihrer Tests, damit sie leicht ausgeführt werden können und konsistente Resultate liefern, wird ihre Vertrauenswürdigkeit erhöhen. Eine sichere grüne Zone in Ihren Tests kann bei den Entwicklern zu mehr Vertrauen in Ihre Tests führen. Diese grüne Zone erhalten Sie leicht, wenn Sie ein eigenes Projekt für die Unit Tests haben, in dem nur solche Tests liegen, die im Arbeitsspeicher laufen und die konsistent und wiederholbar sind.

8.1.5 Stellen Sie Code-Reviews mit Codeabdeckung sicher

Was bedeutet eine Codeabdeckung von 100%? Nichts, wenn Sie kein Code-Review durchführen. Ihr CEO mag alle Mitarbeiter aufgefordert haben, »eine Codeabdeckung von über 95%« zu erzielen, und Sie haben genau das getan, was von Ihnen verlangt wurde. Vielleicht haben diese Tests nicht einmal Asserts. Die Leute neigen dazu, das zu tun, was notwendig ist, um eine vorgegebene Zielmetrik zu erreichen.

Was bedeuten 100% Codeabdeckung zusammen mit Tests und Code-Reviews? Es bedeutet, dass Ihnen die Welt gehört. Wenn Sie Code-Reviews und Test-Reviews durchgeführt und dafür gesorgt haben, dass die Tests gut sind und den ganzen Code abdecken, dann sind Sie in einer goldenen Position und haben ein Sicherheitsnetz, das Sie vor dummen Fehlern bewahrt, während auf der anderen Seite das Team vom Teilen des Wissens und von kontinuierlichem Lernen profitiert.

Wenn ich »Code-Review« sage, dann meine ich nicht die halbherzige Verwendung eines Tools von der anderen Seite der Welt, um mit einer Textzeile den Code von jemand anderem zu kommentieren, den dieser Jemand drei Stunden später sehen wird, wenn Sie nicht mehr bei der Arbeit sind.

Nein, wenn ich »Code-Review« sage, dann meine ich wirklich zwei Menschen, die zusammenhocken und sprechen, die auf das gleiche Stück Code schauen und es ändern – und alles live. (Hoffentlich sitzen sie direkt nebeneinander, aber Telekommunikations-Anwendungen wie Skype und TeamViewer tun es auch, danke sehr.) Im nächsten Kapitel werde ich mehr darüber erzählen, wie sich hammermäßige Code-Reviews anfühlen, aber für den Augenblick reicht es zu betonen, dass Sie und Ihre Kollegen ohne kontinuierliche Reviews und Paarbildung beim Programmieren und Testen eine tolle, gewinnbringende Dimension des Lernens und der Produktivität verpassen werden. Wenn das der Fall ist, dann sollten Sie

alles unternehmen, was Sie können, um nicht länger auf diese notwendige, essenzielle Fähigkeit zu verzichten. Das Durchführen von Code-Reviews ist auch eine Technik, um lesbaren und hochwertigen Code zu entwickeln, der viele Jahre überdauern wird und der Sie jeden Morgen zufrieden in den Spiegel schauen lässt.

Hören Sie auf, mich so anzuschauen. Ihre Skepsis hält Sie davon ab, aus Ihrem jetzigen Job Ihren Traumjob zu machen.

Wie auch immer, lassen Sie uns über Codeabdeckung reden.

Um eine gute Abdeckung Ihres neuen Codes sicherzustellen, sollten Sie eines der automatisierten Tools verwenden (z. B. dotCover von JetBrains, OpenCover, NCover oder Visual Studio Pro). Meine persönliche Empfehlung ist zurzeit NCrunch, das Ihnen eine Echtzeitabdeckung mit Rot/Grün-Ansicht Ihres Codes zeigt, die sich beim Programmieren ändert. Es kostet Geld, aber es spart auch Geld. Die Hauptsache ist, dass Sie ein gutes Tool finden, es zu beherrschen lernen, sein Potenzial ausschöpfen und Ihre Vorteile daraus ziehen, damit Sie nie eine geringe Abdeckung haben.

Weniger als 20 % bedeutet, dass ein ganzer Haufen von Tests fehlt. Sie wissen nie, ob der nächste Entwickler nicht versuchen wird, mit Ihrem Code zu spielen. Er mag vielleicht versuchen, ihn zu optimieren, oder er löscht fälschlicherweise eine wichtige Zeile und wenn Sie dann keinen Test haben, der fehlschlägt, dann kann der Fehler unentdeckt bleiben.

Wenn Sie Code- und Test-Reviews durchführen, können Sie auch eine manuelle Prüfung vornehmen, die großartig für das Ad-hoc-Testen eines Tests geeignet ist: Versuchen Sie, eine Zeile oder eine boolesche Überprüfung auszukommentieren. Wenn alle Tests immer noch erfolgreich sind, dann fehlen vielleicht einige Tests oder die aktuellen Tests prüfen eventuell nicht die richtige Sache.

Wenn Sie einen neuen, zuvor fehlenden Test hinzufügen, dann prüfen Sie mit den folgenden Schritten, ob es sich um den richtigen Test handelt:

1. Kommentieren Sie den Produktionscode aus, von dem Sie annehmen, er sei nicht abgedeckt.
2. Lassen Sie alle Tests laufen.
3. Wenn alle Tests erfolgreich sind, dann fehlt ein Test oder sie testen die falsche Sache. Andernfalls hätte es irgendwo einen Test gegeben, der den Aufruf dieser Zeile oder irgendeine aus dieser Zeile resultierende Konsequenz erwartet hätte, und dieser fehlende Test würde nun fehlschlagen.
4. Sobald Sie den fehlenden Test gefunden haben, müssen Sie ihn hinzufügen. Lassen Sie den Code auskommentiert und schreiben Sie einen neuen Test, der unter der Voraussetzung fehlschlägt, dass der auskommentierte Code fehlt.
5. Machen Sie die Auskommentierung des Codes rückgängig.
6. Der Test, den Sie hinzugefügt haben, sollte nun erfolgreich ablaufen.
7. Wenn der Test immer noch fehlschlägt, dann heißt das, der Test hat einen Bug oder er prüft die falsche Sache. Modifizieren Sie den Test, bis er erfolgreich ist. Sie wollen nun sehen, ob der Test in Ordnung ist und nicht nur erfolgreich ist, wenn er fehlerfrei ablaufen soll, sondern auch fehlschlägt, wenn er soll. Um sicher zu sein, dass der Test fehlschlägt, wenn er soll, fügen Sie den Bug wieder in Ihren Code ein (kommentieren Sie die Zeile des Produktionscodes wieder aus) und schauen Sie, ob der Test tatsächlich fehlschlägt.

Zur zusätzlichen Vertrauensstärkung können Sie auch versuchen, die verschiedenen Parameter oder internen Variablen in Ihrer zu testenden Methode durch Konstanten zu ersetzen (beispielsweise indem Sie ein `bool` immer auf `true` setzen und sehen, was passiert).

Der Trick bei all diesem Testen ist es, dafür zu sorgen, dass es nicht zu viel Zeit benötigt und die Mühe wert ist. Darum geht es im nächsten Abschnitt: die Wartbarkeit.

8.2 Das Schreiben wartbarer Tests

Wartbarkeit ist eines der Kernthemen, mit dem Entwickler beim Schreiben von Unit Tests konfrontiert sind. Schließlich scheint es, dass es immer schwieriger wird, die Tests zu pflegen und zu verstehen, und jede kleine Änderung am System scheint den einen oder anderen Test zu zerbrechen, auch wenn Bugs gar nicht vorhanden sind. Zu all den Teilen von Code scheint die Zeit eine »Indirektionsschicht« zwischen dem, was Sie glauben, dass der Code tut, und dem, was er wirklich macht, hinzuzufügen.

Dieser Abschnitt wird einige Techniken behandeln, die ich auf die harte Tour beim Schreiben von Unit Tests mit verschiedenen Teams gelernt habe. Dazu gehört unter anderem das Testen nur mit öffentlichen Kontrakten, das Entfernen von Duplizitäten in Tests und das Erzwingen der Test-Entkopplung.

8.2.1 Das Testen privater oder geschützter Methoden

Aus Sicht des Entwicklers sind private oder geschützte Methoden gewöhnlich aus gutem Grund privat. Manchmal sind sie es, um Details der Implementierung zu verstecken, damit sich die Implementierung später ändern kann, ohne dass sich auch die Endfunktionalität ändert. Es könnten aber auch Sicherheits- oder IP-bezogene Gründe dafür vorliegen (beispielsweise zur Verschleierung).

Wenn Sie eine private Methode testen, dann testen Sie einen internen Kontrakt des Systems, der sich durchaus ändern kann. Interne Kontrakte sind dynamisch und sie ändern sich, wenn Sie das System überarbeiten. Wenn sie sich ändern, könnte Ihr Test fehlschlagen, denn irgendeine interne Arbeit wird anders erbracht, auch wenn die Gesamtfunktionalität des Systems die gleiche bleibt.

Zu Testzwecken ist der öffentliche Kontrakt (die Gesamtfunktionalität) alles, worum Sie sich kümmern müssen. Das Testen der Funktionalität von privaten Methoden kann zu fehlschlagenden Tests führen, auch wenn die Gesamtfunktionalität korrekt ist.

Betrachten Sie es so: Keine private Methode existiert ohne Grund. Irgendwo in der ganzen Kette gibt es eine öffentliche Methode, die zu guter Letzt diese Methode aufrufen wird oder die eine andere private Methode aufruft, die ihrerseits letztlich diese Methode, an der Sie interessiert sind, aufrufen wird. Das heißt, dass jede private Methode gewöhnlich Teil einer größeren Unit of Work oder eines Use Case in dem System ist, das mit einer öffentlichen API beginnt und mit einem der drei Endresultate endet: einem Rückgabewert, einer Zustandsänderung oder einem Third-Party-Aufruf (oder allen drei).

Von diesem Standpunkt aus betrachtet sollten Sie, wenn Sie eine private Methode sehen, den öffentlichen Use Case im System finden, der diese ausführt. Wenn Sie nur die private Methode testen und diese funktioniert, bedeutet das nicht, dass der Rest des Systems diese private Methode korrekt benutzt oder das zurückgegebene Resultat richtig verwendet. Sie

könnten ein System haben, das im Inneren richtig arbeitet, aber dessen hübscher innerer Kram schrecklich falsch von den öffentlichen APIs verwendet wird.

Manchmal, wenn eine Methode das Testen wert ist, dann ist sie es auch wert, `public`, `static` oder zumindest `internal` zu sein und einen öffentlichen Kontrakt für jeden Code, der sie benutzt, zu definieren. In einigen Fällen mag das Design klarer sein, wenn Sie die Methode ganz in eine andere Klasse verschieben. Wir werden uns diese Ansätze gleich anschauen.

Heißt das, es sollten schließlich gar keine privaten Methoden in der Code-Basis vorhanden sein? Nein. Mit TDD schreiben Sie Tests gewöhnlich für Methoden, die öffentlich sind, und diese öffentlichen Methoden werden später so überarbeitet, dass sie kleinere, private Methoden aufrufen. Die ganze Zeit über werden die Tests der öffentlichen Methoden weiterhin erfolgreich ausgeführt.

Sie machen Methoden öffentlich

Es ist nicht unbedingt eine schlechte Sache, eine Methode öffentlich zu machen. Das mag den objektorientierten Prinzipien widersprechen, die Ihnen beigebracht wurden, aber eine Methode testen zu wollen, kann heißen, dass die Methode ein *bekanntes Verhalten* oder einen *bekannten Kontrakt* für den aufrufenden Code aufweist. Indem Sie sie öffentlich machen, machen Sie dies offiziell. Bleibt die Methode privat, sagen Sie damit allen Entwicklern, die nach Ihnen kommen, dass sie die Implementierung der Methode ändern können, ohne sich um unbekannten Code, der sie benutzt, kümmern zu müssen, denn sie dient nur als Teil einer größeren Gruppe von Dingen, die zusammen einen Kontrakt für den aufrufenden Code bilden.

Sie extrahieren Methoden in neue Klassen

Wenn Ihre Methode eine Menge Logik enthält, die für sich alleine stehen kann, oder sie benutzt einen Zustand in der Klasse, der nur für die fragliche Methode von Bedeutung ist, dann kann es eine gute Idee sein, die Methode in eine neue Klasse mit einer speziellen Rolle innerhalb des Systems auszulagern. Sie können die Klasse dann separat testen. Das Buch *Arbeiten mit Legacy Code* von Michael Feathers enthält einige gute Beispiele zu dieser Technik und *Clean Code* von Robert C. Martin kann dabei helfen, herauszufinden, wann dies eine gute Idee ist.

Sie machen Methoden statisch

Wenn Ihre Methode keine der Variablen ihrer Klasse verwendet, dann können Sie die Methode überarbeiten und sie statisch machen. Das macht sie viel leichter zu testen, sagt aber auch aus, dass es sich nun um eine Art Dienstmethode handelt, die einen bekannten, öffentlichen Kontrakt aufweist, der durch ihren Namen festgelegt ist.

Sie machen Methoden intern

Wenn nichts anderes hilft und Sie es sich nicht leisten können, die Methode auf eine »offizielle« Weise preiszugeben, dann möchten Sie sie vielleicht zu einer internen Methode machen und das Attribut `[InternalsVisibleTo("TestAssembly")]` für das Produktionscode-Assembly setzen, damit Ihre Tests die Methode immer noch aufrufen können. Dies ist mein am wenigsten geliebter Ansatz, aber manchmal hat man keine Wahl (vielleicht aus Sicherheitsgründen, aus Mangel an Kontrolle über das Code-Design und so weiter).

Die Methode auf `internal` zu setzen, ist kein großartiger Weg, damit Ihre Tests besser wartbar sind, denn ein Programmierer kann trotzdem glauben, es sei einfacher, die Methode zu ändern. Doch durch die Enthüllung einer Methode als expliziter, öffentlicher Kontrakt sorgen Sie dafür, dass der Programmierer, der sie womöglich ändert, weiß, dass die Methode einen wirklich verwendeten Kontrakt besitzt, den er nicht brechen kann.

Wenn Sie eine ältere Version als Visual Studio 2012 benutzen, haben Sie die Möglichkeit, private Accessoren zu erzeugen. Das sind Methoden, die Visual Studio erzeugt, um via Reflection Ihre privaten Methoden aufzurufen. Bitte verwenden Sie dieses Werkzeug nicht. Es erzeugt ein problematisches Stück Code, das mit der Zeit schwierig zu warten und zu lesen ist. In der Tat sollten Sie alles vermeiden, was Ihnen erzählt, es würde für Sie Unit Tests oder mit Tests zusammenhängendes Zeug erzeugen, es sei denn, Sie haben absolut keine Wahl.

Das Entfernen der Methode ist keine gute Option, weil der Produktionscode diese Methode auch benutzt. Andernfalls gäbe es keinen Grund, die Tests überhaupt zu schreiben.

Eine andere Möglichkeit, die Wartbarkeit des Codes zu verbessern, ist die Entfernung der Duplizitäten in den Tests.

8.2.2 Das Entfernen von Duplizitäten

Duplizitäten in Ihren Unit Tests können Ihnen als Entwickler genauso weh tun wie Duplizitäten im Produktionscode. Das DRY-Prinzip sollte im Testcode genauso gelten wie im Produktionscode. Duplizierter Code bedeutet, dass mehr Code geändert werden muss, wenn sich ein Aspekt, den Sie testen, ändert. Die Änderung eines Konstruktors oder der Semantik bei der Verwendung einer Klasse kann einen großen Effekt auf Tests mit einer Menge dupliziertem Code haben.

Um zu verstehen, wieso, lassen Sie uns mit einem einfachen Beispiel eines Tests beginnen.

```
public class LogAnalyzer
{
    public bool IsValid(string fileName)
    {
        if (fileName.Length < 8)
        {
            return true;
        }
        return false;
    }
}

[TestFixture]
public class LogAnalyzerTestsMaintainable
{
    [Test]
    public void IsValid_LengthBiggerThan8_IsFalse()
    {
```

```
        LogAnalyzer logan = new LogAnalyzer();

        bool valid = logan.IsValid("123456789");

        Assert.IsFalse(valid);
    }
}
```

Listing 8.5: Eine zu testende Klasse und ein Test, der sie verwendet

Der Test am Ende von Listing 8.5 scheint vernünftig, bis Sie einen weiteren Test für die gleiche Klasse einbauen und mit zwei Tests enden, wie im nächsten Listing.

```
[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
    LogAnalyzer logan = new LogAnalyzer();

    bool valid = logan.IsValid("123456789");

    Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
    LogAnalyzer logan = new LogAnalyzer();

    bool valid = logan.IsValid("1234567");

    Assert.IsTrue(valid);
}
```

Listing 8.6: Zwei Tests mit Duplizitäten

Was ist falsch an den Tests im vorherigen Listing? Wenn sich die Art ändert, wie Sie `LogAnalyzer` verwenden (seine Semantik), so ist das Hauptproblem, dass die Tests unabhängig voneinander gepflegt werden müssen, was zu mehr Wartungsaufwand führt. Das folgende Listing zeigt ein Beispiel für solch eine Änderung.

```
public class LogAnalyzer
{
    private bool initialized=false;

    public bool IsValid(string fileName)
    {
        if(!initialized)
```

```
        {
            throw new NotInitializedException(
            "Die Methode analyzer.Initialize() sollte vor" +
            " jeder anderen Operation aufgerufen werden!");
        }
        if (fileName.Length < 8)
        {
            return true;
        }
        return false;
    }
    public void Initialize()
    {
        //initialisiere Logik hier
        ...
        initialized=true;
    }
}
```

Listing 8.7: `LogAnalyzer` mit geänderter Semantik benötigt nun eine Initialisierung.

Jetzt werden die beiden Tests in Listing 8.6 schiefgehen, weil sie beide den Aufruf von `Initialize()` der Klasse `LogAnalyzer` vernachlässigen. Da Sie den doppelten Code haben (beide Tests legen die Klasse innerhalb des Tests an), müssen Sie beide anpacken und den Aufruf von `Initialize()` einbauen.

Sie können beide Tests überarbeiten und die Duplizität entfernen, indem Sie `LogAnalyzer` in einer Methode `CreateDefaultAnalyzer()` anlegen, die beide Tests aufrufen können. Sie könnten auch die Erzeugung und Initialisierung in eine neue Setup-Methode Ihrer Testklasse verlegen.

Die Entfernung der Duplizität mit einer Hilfsmethode

Listing 8.8 zeigt, wie Sie die Tests in einen besser wartbaren Zustand bringen können, indem Sie eine gemeinsame Fabrikmethode einführen, die eine Standardinstanz von `LogAnalyzer` anlegt. Angenommen, alle Tests würden diese Fabrikmethode verwenden, dann könnten Sie einen Aufruf von `Initialize()` in die Fabrikmethode einbauen, statt alle Tests so zu ändern, dass sie `Initialize()` aufrufen.

```
[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
    LogAnalyzer logan = GetNewAnalyzer();

    bool valid = logan.IsValid("123456789");

    Assert.IsFalse(valid);
}
```

```
[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
    LogAnalyzer logan = GetNewAnalyzer();

    bool valid = logan.IsValid("1234567");

    Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
    LogAnalyzer analyzer = new LogAnalyzer();
    analyzer.Initialize();
    return analyzer;
}
```

Listing 8.8: Der Aufruf von `Initialize()` wird in der Fabrikmethode hinzugefügt.

Fabrikmethoden sind nicht der einzige Weg, um Duplizitäten in Tests zu entfernen, wie der nächste Abschnitt zeigen wird.

Die Entfernung der Duplizität mit [SetUp]

Sie könnten auch einfach `LogAnalyzer` innerhalb der `Setup`-Methode initialisieren, wie hier dargestellt.

```
[SetUp]
public void Setup()
{
    logan=new LogAnalyzer();
    logan.Initialize();
}

private LogAnalyzer logan= null;

[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
    bool valid = logan.IsValid("123456789");
    Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
```

```
{
    bool valid = logan.IsValid("1234567");
    Assert.IsTrue(valid);
}
```

Listing 8.9: Die Verwendung einer Setup-Methode, um die Duplizität zu entfernen

In diesem Fall benötigen Sie nicht einmal eine Zeile, die das Analyzer-Objekt in jedem Test anlegt: Eine gemeinsame Klasseninstanz wird vor jedem Test mit einer neuen Instanz von `LogAnalyzer` initialisiert und dann wird `Initialize()` für diese Instanz aufgerufen. Aber seien Sie vorsichtig: Die Verwendung einer Setup-Methode zum Entfernen von Duplizitäten ist nicht immer eine gute Idee, wie ich im nächsten Abschnitt erklären werde.

8.2.3 Die Verwendung von Setup-Methoden in einer wartbaren Art und Weise

Die `Setup()`-Methode ist einfach zu benutzen. Tatsächlich ist das beinahe zu einfach – die Entwickler neigen dazu, sie für Dinge zu verwenden, für die sie nicht gedacht ist, und die Tests werden im Ergebnis weniger gut lesbar und wartbar.

Nichtsdestotrotz weisen Setup-Methoden mehrere Einschränkungen auf, die Sie mit einfachen Hilfsmethoden umgehen können:

- Setup-Methoden können nur bei der Initialisierung helfen.
- Setup-Methoden sind nicht immer die besten Kandidaten für die Duplizitätsentfernung. Bei der Entfernung von Duplizitäten geht es nicht immer um das Erzeugen und die Initialisierung von neuen Instanzen von Objekten. Manchmal geht es um die Entfernung von Duplizitäten in der Assert-Logik.
- Setup-Methoden können keine Parameter oder Rückgabewerte haben.
- Setup-Methoden können nicht als Fabrikmethoden verwendet werden, die Werte zurückgeben. Sie werden vor dem Test ausgeführt, also müssen sie in der Art, wie sie arbeiten, allgemeiner sein. Tests müssen manchmal bestimmte Dinge abfragen oder gemeinsamen Code mit einem Parameter für den spezifischen Test aufrufen (beispielsweise ein Objekt abfragen und seine Property auf einen bestimmten Wert setzen).
- Setup-Methoden sollten nur solchen Code enthalten, der für alle Tests in der aktuellen Testklasse gilt, denn andernfalls wird die Methode schwieriger zu lesen und zu verstehen sein.

Da Sie nun die grundlegenden Einschränkungen der Setup-Methoden kennen, lassen Sie uns sehen, wie Entwickler versuchen, damit klarzukommen. Entwickler neigen dazu, in ihrem Bestreben, Setup-Methoden auf Teufel komm raus anzuwenden, sie der Verwendung von Hilfsmethoden vorzuziehen. Entwickler missbrauchen Setup-Methoden auf verschiedene Arten:

- Sie initialisieren Objekte in der Setup-Methode, die nur in einigen Tests der Klasse verwendet werden.
- Sie haben einen Setup-Code, der lang und schwer verständlich ist.
- Sie setzen Mocks und Fake-Objekte in der Setup-Methode auf.

Lassen Sie uns das näher betrachten.

Die Initialisierung von Objekten, die nur von einigen Tests verwendet werden

Diese Sünde ist tödlich. Sobald Sie sie begehen, wird es schwierig, die Tests zu pflegen oder auch nur zu lesen, weil die Setup-Methode schnell mit Objekten überladen wird, die nur für einige der Tests bestimmt sind. Das folgende Listing zeigt, wie Ihre Testklasse aussehen würde, wenn Sie ein `FileInfo`-Objekt in der Setup-Methode initialisieren, aber nur in einem einzigen Test verwenden würden.

```
[SetUp]
public void Setup()
{
    logan=new LogAnalyzer();
    logan.Initialize();

    fileInfo=new FileInfo("c:\\someFile.txt");
}

private FileInfo fileInfo = null;
private LogAnalyzer logan= null;

[Test]
public void IsValid_LengthBiggerThan8_IsFalse()
{
    bool valid = logan.IsValid("123456789");
    Assert.IsFalse(valid);
}

[Test]
public void IsValid_BadFileInfoInput_returnsFalse()
{
    bool valid = logan.IsValid(fileInfo);
    Assert.IsFalse(valid);
}

[Test]
public void IsValid_LengthSmallerThan8_IsTrue()
{
    bool valid = logan.IsValid("1234567");
    Assert.IsTrue(valid);
}

private LogAnalyzer GetNewAnalyzer()
{
...
}
```

Listing 8.10: Eine schlecht implementierte `Setup()`-Methode

Warum ist die Setup-Methode in diesem Listing weniger gut zu warten? Weil Sie Folgendes machen müssen, um die Tests das erste Mal zu lesen und zu verstehen, weshalb sie fehlschlagen:

1. Sie gehen durch die Setup-Methode, um zu verstehen, was initialisiert wird.
2. Sie nehmen an, dass die Objekte in der Setup-Methode in allen Tests verwendet werden.
3. Später finden Sie heraus, dass Sie falsch lagen, und lesen die Tests noch mal sorgfältiger, um zu sehen, welcher Test die Objekte verwendet, die die Probleme möglicherweise verursachen.
4. Sie tauchen wegen nichts tiefer in den Testcode ein und brauchen mehr Zeit und Aufwand, um zu verstehen, was der Code tut.

Denken Sie beim Schreiben der Tests immer an die Leser Ihrer Tests. Stellen Sie sich vor, sie würden sie zum allerersten Mal lesen. Verärgern Sie sie nicht.

Setup-Code, der lang und schwierig zu verstehen ist

Weil die Setup-Methode nur eine Stelle im Test bereitstellt, um die Dinge zu initialisieren, tendieren die Entwickler dazu, vieles zu initialisieren, was notwendigerweise mühsam zu lesen und zu verstehen ist. Eine Lösung ist, die Aufrufe umzubauen und bestimmte Dinge innerhalb von Hilfsmethoden zu initialisieren, die dann von der Setup-Methode aufgerufen werden. Gewöhnlich ist das Refaktorieren der Setup-Methode eine gute Idee. Je lesbarer sie ist, desto lesbarer wird Ihre Testklasse sein.

Aber es ist ein schmaler Grat zwischen Über-Refaktorierung und Lesbarkeit. Über-Refaktorierung kann zu weniger gut lesbarem Code führen. Das ist eine Frage der persönlichen Vorliebe. Sie müssen darauf achten, wenn Ihr Code immer unlesbarer wird. Ich empfehle, während des Refactorings die Rückmeldung eines Kollegen einzuholen. Wir alle können von dem Code, den wir geschrieben haben, zu sehr fasziniert sein und das Involvieren eines zweiten Augenpaares in das Refactoring kann zu guten und unbefangenen Resultaten führen. Einen Kollegen im Nachhinein ein Code-Review (Test-Review) durchführen zu lassen, ist auch gut, aber nicht so produktiv, wie es gleich zu tun.

Das Aufsetzen von Fakes in der Setup-Methode

Bitte bauen Sie keine Fakes in Setup-Methoden ein. Dies würde es erschweren, die Tests zu lesen und zu warten.

Ich bevorzuge es, jeden Test seine eigenen Mocks und Stubs durch das Aufrufen von Hilfsmethoden innerhalb des Tests erzeugen zu lassen, damit der Leser des Tests genau weiß, was vor sich geht, ohne dass er vom Test zum Setup springen muss, um das große Ganze zu verstehen.

Hören Sie auf, Setup-Methoden zu verwenden

Ich habe damit aufgehört, für die Tests, die ich schreibe, Setup-Methoden zu verwenden. Sie sind ein Relikt aus einer Zeit, als es in Ordnung war, miese, unleserliche Tests zu schreiben, aber diese Zeit ist vorüber. Testcode sollte hübsch und sauber sein, genauso wie Produktionscode. Aber wenn Ihr Produktionscode schrecklich aussieht, dann verwenden Sie das bitte nicht als Ausrede für unleserliche Tests. Verwenden Sie einfach Fabrik- und Hilfsmethoden, und die Dinge werden für jeden Beteiligten besser.

Wenn Ihre Tests immer gleich aussehen, ist es eine andere großartige Möglichkeit, die Setup-Methoden zu ersetzen, parametrisierte Tests zu verwenden (`[TestCase]` in NUnit, `[Theory]` in XUnit.net oder `[OopsWeStillDontHaveThatFeatureAfterFiveYears]` in MSTest). Okay, ein schlechter Scherz, aber MSTest hat dafür immer noch keine einfache Unterstützung.

8.2.4 Das Erzwingen der Test-Isolierung

Ein Mangel an Test-Entkopplung ist der wichtigste einzelne Grund der Testblockade, den ich während meiner Arbeit an Unit Tests und meiner Beratungstätigkeit gesehen habe. Das Grundkonzept ist, dass ein Test immer in seiner eigenen kleinen Welt ablaufen sollte, isoliert selbst vom Wissen, dass da draußen andere Tests ähnliche oder andere Dinge machen.

Der Test, der »Fehlschlag« rief

In einem Projekt, an dem ich beteiligt war, verhielten sich die Tests merkwürdig, und sie wurden mit der Zeit immer merkwürdiger. Ein Test schlug fehl, lief anschließend aber für mehrere Tage ohne Fehler durch. Einen Tag später schlug er wieder fehl, anscheinend zufällig, und bei anderen Gelegenheiten lief er erfolgreich durch, obwohl Code geändert wurde, um sein Verhalten zu ändern oder ganz zu entfernen. Es kam zu dem Punkt, an dem die Entwickler zueinander sagten: »Ach, das ist okay. Wenn er manchmal erfolgreich ist, dann ist er erfolgreich.«

Es stellte sich heraus, dass der Test als Teil seines Codes einen anderen Test aufrief und wenn dieser Test fehlschlug, führte das auch zum Fehlschlag des ersten Tests.

Es kostete uns drei Tage, das herauszufinden, nachdem wir schon einen Monat mit dieser Situation gelebt hatten. Nachdem wir den Test schließlich korrekt ans Laufen gebracht hatten, fanden wir heraus, dass wir tatsächlich einen ganzen Haufen echter Bugs in unserem Code hatten, die wir ignorierten, weil wir dachten, wir würden fälschlicherweise Fehlermeldungen vom Test erhalten. Die Geschichte vom Jungen, der Wolf schrie, ist auch in der Softwareentwicklung gültig.

Wenn die Tests nicht gut isoliert sind, dann können sie einander genug auf die Füße treten, um Ihnen ein schlechtes Gefühl zu geben und Sie bedauern zu lassen, sich für das Unit Testing in Ihrem Projekt entschieden zu haben, weshalb Sie sich selbst versprechen, es nie wieder zu tun. Ich habe gesehen, wie dies geschehen ist. Entwickler plagen sich nicht damit ab, nach Problemen in den Tests zu schauen. Wenn es dann aber ein Problem in den Tests gibt, kann es eine Menge Zeit kosten, herauszufinden, was schiefläuft.

Es gibt einige Test-»Gerüche«, die auf eine kaputte Test-Isolierung hindeuten können:

- *Eingeschränkte Testreihenfolge* – Tests, die erwarten, in einer bestimmten Reihenfolge ausgeführt zu werden oder Informationen von anderen Testresultaten zu erhalten.
- *Versteckte Testaufrufe* – Tests, die andere Tests aufrufen.
- *Shared-State Corruption* – Tests, die einen In-Memory-Zustand teilen, ohne diesen zurückzusetzen.
- *External-Shared-State Corruption* – Integrationstests mit gemeinsamen Ressourcen ohne Rollback.

Lassen Sie uns diese einfachen *Antimuster* anschauen.

Antimuster: Eingeschränkte Testreihenfolge

Dieses Problem tritt auf, wenn Tests so geschrieben sind, dass sie einen bestimmten Zustand im Arbeitsspeicher, in einer externen Ressource oder in der aktuellen Testklasse erwarten – ein Zustand, der in der gleichen Klasse von anderen Testläufen vor dem aktuellen Test erzeugt wurde. Das Problem ist, dass die meisten Testplattformen (einschließlich NUnit, JUnit und MbUnit) nicht garantieren, dass die Tests in einer bestimmten Reihenfolge ausgeführt werden. Damit kann morgen fehlschlagen, was heute noch erfolgreich abläuft.

Das folgende Listing zeigt einen Test für `LogAnalyzer`, der erwartet, dass ein vorangegangener Test die Funktion `Initialize()` bereits aufgerufen hat.

```
[TestFixture]
public class IsolationsAntiPatterns
{
    private LogAnalyzer logan;
    [Test]
    public void CreateAnalyzer_BadFileName_ReturnsFalse()
    {
        logan = new LogAnalyzer();
        logan.Initialize();

        bool valid = logan.IsValid("abc");

        Assert.That(valid, Is.False);
    }

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        bool valid = logan.IsValid("abcdefg");

        Assert.That(valid, Is.True);
    }
}
```

Listing 8.11: Eingeschränkte Testreihenfolge: Der zweite Test wird fehlschlagen, wenn er zuerst ausgeführt wird.

Wenn Tests die Isolierung nicht strikt einhalten, kann das zu einer Unzahl von Problemen führen:

- Ein Test kann plötzlich schiefgehen, wenn eine neue Version des Test-Frameworks eingeführt wird, die die Tests in einer anderen Reihenfolge ausführt.
- Das Ausführen eines Teils der Tests kann andere Resultate liefern als das Ausführen aller Tests oder einer anderen Teilmenge der Tests.
- Die Pflege der Tests ist mühsamer, denn Sie müssen sich darüber Gedanken machen, wie sich andere Tests auf Ihren jeweiligen Test und auf den Zustand auswirken.

- Ihre Tests können aus den falschen Gründen fehlschlagen oder erfolgreich sein, beispielsweise weil ein anderer Test zuvor fehlgeschlagen ist oder erfolgreich war und die Ressourcen in einem unbekannten Zustand hinterlassen hat.
- Das Entfernen oder Ändern einiger Tests kann die Ergebnisse anderer Tests beeinflussen.
- Es ist schwierig, Ihre Tests angemessen zu benennen, denn sie testen mehr als nur eine einzige Sache.

Es gibt ein paar verbreitete Muster, die zu einer schlechten Test-Isolation führen:

- *Ablaufprüfung* – Ein Entwickler schreibt Tests, die in einer bestimmten Reihenfolge ablaufen müssen, damit sie den Ausführungsablauf oder einen großen Use Case mit vielen Aktionen testen können, oder es handelt sich um einen kompletten Integrationstest, bei dem jeder Test ein Schritt des Gesamttests ist.
- *Nachlässigkeit beim Aufräumen* – Ein Entwickler ist nachlässig und setzt einen Zustand, den sein Test möglicherweise geändert hat, nicht auf den ursprünglichen Wert zurück, während andere Entwickler Tests schreiben, die wissentlich oder unwissentlich von dieser Unzulänglichkeit abhängen.

Diese Probleme können auf verschiedene Arten gelöst werden:

- *Ablaufprüfung* – Statt innerhalb der Unit Tests ablaufbezogene Tests zu schreiben (beispielsweise Use Cases mit langer Dauer), ziehen Sie die Verwendung irgendeiner Art von Integration-Testing-Framework wie FIT oder FitNesse oder von Produkten aus dem QS-Bereich wie AutomatedQA oder WinRunner in Betracht.
- *Nachlässigkeit beim Aufräumen* – Wenn Sie zu faul sind, Ihre Datenbank oder Ihr Dateisystem oder Ihre Objekte im Speicher nach dem Testen aufzuräumen, dann sollten Sie ernsthaft in Erwägung ziehen, den Beruf zu wechseln. Dies ist nicht der richtige Job für Sie.

Antimuster: Versteckter Testaufruf

In diesem Antimuster enthalten die Tests einen oder mehrere direkte Aufrufe von anderen Tests in der gleichen oder in anderen Testklassen, was dazu führt, dass die Tests voneinander abhängig sind. Das folgende Listing zeigt den Test `CreateAnalyzer_GoodNameAndBadNameUsage`, der am Ende einen anderen Test aufruft und damit eine Abhängigkeit zwischen den Tests schafft, der die Isolierung beider zerstört.

```
[TestFixture]
public class HiddenTestCall
{
    private LogAnalyzer logan;

    [Test]
    public void CreateAnalyzer_GoodNameAndBadNameUsage()
    {
        logan = new LogAnalyzer();
        logan.Initialize();
```

```
        bool valid = logan.IsValid("abc");

        Assert.That(valid, Is.False);

        //versteckter Testaufruf (1):
        CreateAnalyzer_GoodFileName_ReturnsTrue();
    }

    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        bool valid = logan.IsValid("abcdefg");

        Assert.That(valid, Is.True);
    }
}
```

Listing 8.12: Ein Test, der einen anderen aufruft, zerstört die Entkopplung und führt eine Abhängigkeit ein.

Diese Art der Abhängigkeit **(1)** kann mehrere Probleme verursachen:

- Das Ausführen eines Teils der Tests kann andere Resultate liefern als das Ausführen aller Tests oder einer anderen Teilmenge der Tests.
- Die Pflege der Tests ist mühsamer, denn Sie müssen sich darüber Gedanken machen, wie andere Tests mit bestimmten Tests verknüpft sind und wie und wann sie einander aufrufen.
- Tests können aus den falschen Gründen fehlschlagen oder erfolgreich sein. Beispielsweise kann ein anderer Test fehlgeschlagen sein, wodurch Ihr Test auch fehlschlägt oder gar nicht erst aufgerufen wird. Oder ein anderer Test hat vielleicht ein paar gemeinsame Variablen in einem unbekannten Zustand hinterlassen.
- Änderungen an einem Test können die Ergebnisse anderer Tests beeinflussen.
- Es ist schwierig, Tests, die andere Tests aufrufen, eindeutig zu benennen.

Wie wir dahin kamen:

- *Ablaufprüfung* – Ein Entwickler schreibt Tests, die in einer bestimmten Reihenfolge ablaufen müssen, damit sie den Ausführungsablauf oder einen großen Use Case mit vielen Aktionen testen können, oder es handelt sich um einen kompletten Integrationstest, bei dem jeder Test ein Schritt des Gesamttests ist.
- *Der Versuch, Duplizitäten zu entfernen* – Ein Entwickler versucht, eine Duplizität in den Tests zu entfernen, indem er andere Tests aufruft (die einen Code haben, von dem er nicht will, dass ihn der aktuelle Test wiederholt).
- *Nachlässigkeit beim Trennen der Tests* – Ein Entwickler ist bequem und nimmt sich nicht die Zeit, einen eigenständigen Test anzulegen und den Code entsprechend umzubauen. Stattdessen nimmt er eine Abkürzung und ruft einen anderen Test auf.

Hier sind ein paar Lösungen:

- *Ablaufprüfung* – Statt innerhalb der Unit Tests ablaufbezogene Tests zu schreiben (beispielsweise Use Cases mit langer Dauer), ziehen Sie die Verwendung eines Integration-Testing-Frameworks wie FIT oder FitNesse oder von Produkten aus dem QS-Bereich wie AutomatedQA oder WinRunner in Betracht.
- *Der Versuch, Duplizitäten zu entfernen* – Entfernen Sie niemals eine Duplizität durch den Aufruf eines anderen Tests von Ihrem Test aus. Sie verhindern damit, dass sich der Test auf die Setup- und Abräummethoden in der Klasse verlassen kann, und lassen im Wesentlichen zwei Tests in einem laufen (denn sowohl der aufrufende Test als auch der aufgerufene besitzen eine Assertion). Stattdessen verlagern Sie den Code, den Sie nicht zweimal schreiben wollen, in eine dritte Methode, die sowohl Ihr Test als auch der andere aufruft.
- *Nachlässigkeit beim Trennen der Tests* – Wenn Sie zu faul sind, Ihre Tests zu trennen, dann denken Sie an all die zusätzliche Arbeit, die auf Sie wartet, wenn Sie die Tests nicht trennen. Versuchen Sie, sich eine Welt vorzustellen, in der der aktuelle Test, an dem Sie gerade schreiben, der einzige Test im ganzen System ist, weshalb er sich nicht auf irgendeinen anderen Test verlassen kann.

Antimuster: Shared-State Corruption

Dieses Antimuster macht sich auf zweierlei Arten bemerkbar, die unabhängig voneinander sind:

- Die Tests packen gemeinsame Ressourcen (entweder im Arbeitsspeicher oder in externen Ressourcen wie Datenbanken, Dateisystemen usw.) an, ohne die Änderungen, die sie an diesen Ressourcen vorgenommen haben, aufzuräumen oder ein Rollback durchzuführen.
- Die Tests setzen den Anfangszustand, den sie zu Beginn benötigen, nicht auf und verlassen sich darauf, dass dieser Zustand schon vorliegt.

Jede dieser Situationen verursacht Probleme, die wir uns gleich näher anschauen werden.

Das Problem besteht darin, dass sich die Tests auf einen bestimmten Zustand verlassen, damit sie ein konsistentes Erfolg/Misserfolg-Verhalten zeigen. Wenn ein Test nicht die Kontrolle über den erwarteten Zustand ausübt oder andere Tests diesen Zustand aus welchem Grund auch immer beschädigen, dann kann der Test nicht ordnungsgemäß ablaufen und konsistent die richtigen Ergebnisse melden.

Angenommen, Sie haben eine Klasse `Person` mit einfachen Features: Sie enthält eine Liste mit Telefonnummern und die Fähigkeit, diese Liste unter Angabe der ersten Ziffern einer Nummer zu durchsuchen. Das nächste Listing zeigt ein paar Tests, die die Instanz eines `Person`-Objekts nicht richtig aufsetzen oder aufräumen.

```
[TestFixture]
public class SharedStateCorruption
{
    Person person = new Person();  //definiert gemeinsamen Zustand
```

```
    [Test]
    public void CreateAnalyzer_GoodFileName_ReturnsTrue()
    {
        person.AddNumber("055-4556684(34)"); //ändert den gemeinsamen
                                             //Zustand (1)
        string found = person.FindPhoneStartingWith("055");
        Assert.AreEqual("055-4556684(34)", found);
    }

    [Test]
    public void FindPhoneStartingWith_NoNumbers_ReturnsNull()
    {
        string found =
            person.FindPhoneStartingWith("0");  //liest den
                                                //gemeinsamen Zustand
        Assert.IsNull(found);
    }
}
```

Listing 8.13: Shared-State Corruption durch einen Test

In diesem Beispiel wird der zweite Test (der einen Rückgabewert `null` erwartet) fehlschlagen, weil der vorangegangene Test bereits eine Nummer zur Instanz von `Person` hinzugefügt hat **(1)**.

Diese Art von Problem verursacht eine Reihe von Symptomen:

- Das Ausführen eines Teils der Tests kann andere Resultate liefern als das Ausführen aller Tests oder einer anderen Teilmenge der Tests.
- Die Pflege der Tests ist mühsamer, denn Sie können den Zustand für andere Tests beschädigen und dies auch, ohne dass Sie es bemerken.
- Tests können aus den falschen Gründen fehlschlagen oder erfolgreich sein. Beispielsweise kann ein anderer Test fehlgeschlagen oder erfolgreich gewesen sein und hat einen problematischen Zustand hinterlassen oder er hat am Ende seiner Ausführung gar nicht aufgeräumt.
- Änderungen an einem Test können die Ergebnisse anderer Tests scheinbar zufällig beeinflussen.

Hier ist, wie wir dahin kamen:

- *Der Zustand wird nicht vor jedem Test initialisiert* – Ein Entwickler setzt den für den Test benötigten Zustand nicht auf oder nimmt an, er sei bereits korrekt gesetzt.
- *Ein gemeinsamer Zustand wird verwendet* – Ein Entwickler verwendet ohne die notwendigen Vorsichtsmaßnahmen gemeinsamen Speicher oder externe Ressourcen für mehr als einen Test.
- *Die Verwendung von statischen Instanzen in Tests* – Ein Entwickler setzt einen statischen Zustand, der in anderen Tests verwendet wird.

Hier sind einige Lösungen:

- *Der Zustand wird nicht vor jedem Test initialisiert* – Dies ist eine zwingend erforderliche Praxis beim Schreiben von Unit Tests. Verwenden Sie entweder eine Setup-Methode oder rufen Sie am Anfang jedes Tests eine bestimmte Hilfsmethode auf, damit der Zustand so ist, wie Sie ihn erwarten.
- *Ein gemeinsamer Zustand wird verwendet* – In vielen Fällen brauchen Sie einen Zustand überhaupt nicht zu teilen. Der sicherste Weg ist es, für jeden Test eigene Objektinstanzen zu verwenden.
- *Die Verwendung von statischen Instanzen in Tests* – Sie müssen sorgfältig damit umgehen, wie Ihre Tests einen statischen Zustand verwalten. Sorgen Sie dafür, den statischen Zustand über Setup- oder Abbruchmethoden sauber zu halten. Manchmal ist es effektiv, direkte Hilfsmethodenaufrufe zu verwenden, um den statischen Zustand innerhalb des Tests eindeutig zurückzusetzen. Wenn Sie Singletons testen, ist es den Aufwand wert, öffentliche oder interne Setter einzubauen, damit Ihre Tests sie auf eine saubere Objektinstanz zurücksetzen können.

Antimuster: External-Shared-State Corruption

Dieses Antimuster ist dem In-Memory-, Shared-State-Corruption-Muster ähnlich, aber es tritt beim Integration Testing auf:

- Die Tests packen gemeinsame Ressourcen (entweder im Arbeitsspeicher oder in externen Ressourcen wie Datenbanken und Dateisysteme) an, ohne die Änderungen, die sie an diesen Ressourcen vorgenommen haben, aufzuräumen oder ein Rollback durchzuführen.
- Die Tests setzen den Anfangszustand, den sie zu Beginn benötigen, nicht auf und verlassen sich darauf, dass dieser Zustand schon vorliegt.

Nachdem wir uns jetzt die Entkopplung der Tests angeschaut haben, lassen Sie uns nun anschauen, wie Sie Ihre *Asserts* in den Griff bekommen, damit Sie die ganze Sache auch mitbekommen, wenn ein Test fehlschlägt.

8.2.5 Vermeiden Sie mehrfache Asserts für unterschiedliche Belange

Um das Problem der mehrfachen Belange zu verstehen, werfen Sie einen Blick auf das folgende Beispiel.

```
[Test]
public void CheckVariousSumResultsIgnoringHigherThan1001()
{
    Assert.AreEqual(3, Sum(1001, 1, 2));
    Assert.AreEqual(3, Sum(1, 1001, 2));
    Assert.AreEqual(3, Sum(1, 2, 1001));
}
```

Listing 8.14: Ein Test mit mehreren Asserts

Hier existiert mehr als ein Test in der Testmethode. Sie könnten sagen, dass hier drei verschiedene Teileigenschaften getestet werden.

Der Autor der Testmethode hat versucht, Zeit zu sparen, indem er drei Tests als drei einfache Asserts eingebaut hat. Was ist dabei das Problem? Wenn Asserts fehlschlagen, dann werfen sie Ausnahmen. (Im Fall von NUnit werfen sie eine spezielle `AssertException`, die vom NUnit-Test-Runner aufgefangen wird, der diese Ausnahme als Signal versteht, dass die aktuelle Testmethode fehlgeschlagen ist.) Sobald eine Assert-Anweisung eine Ausnahme wirft, wird keine andere Zeile der Testmethode mehr ausgeführt. Das bedeutet, dass die beiden anderen Assert-Anweisungen niemals ausgeführt werden, sobald das erste Assert in Listing 8.14 fehlschlägt. Na und? Vielleicht interessieren Sie die anderen nicht, sobald eines fehlschlägt? Manchmal. In diesem Fall testet jedes Assert eine andere Eigenschaft des Endresultats der Anwendung und es interessiert Sie sehr wohl, was mit den anderen geschieht, wenn eines fehlschlägt.

Es gibt mehrere Möglichkeiten, das gleiche Ziel zu erreichen:

- Erzeugen Sie einen eigenen Test für jedes Assert.
- Verwenden Sie parametrisierte Tests.

Packen Sie den Assert-Aufruf in einen `try-catch`-Block.

Warum spielt es eine Rolle, wenn einige Asserts nicht ausgeführt werden?

Wenn nur ein Assert fehlschlägt, dann können Sie nicht wissen, ob nicht die anderen Asserts in der gleichen Testmethode auch fehlgeschlagen wären oder nicht. Sie *glauben* vielleicht, es zu wissen, aber das ist nur eine Vermutung, bis Sie es mit einem fehlschlagenden oder erfolgreichen Assert beweisen können. Wenn die Leute nur einen Teil des Bildes sehen, dann tendieren sie dazu, den Zustand des Systems zu beurteilen, was sich als voreilig erweisen kann. Je mehr Informationen Sie über alle fehlgeschlagenen oder erfolgreichen Asserts haben, desto besser sind Sie ausgerüstet, um zu verstehen, wo der Fehler im System liegen mag und wo nicht.

Dies trifft nur zu, wenn Sie Asserts auf mehrere Belange anwenden. Es wäre nicht so, wenn Sie eine Person auf ihren Namen X, ihr Alter Y usw. testen würden, denn sobald ein Assert fehlschlägt, interessieren Sie die anderen nicht mehr. Aber es wäre ein Belang, wenn Sie von einer Aktion mehrere Endresultate erwarten. Beispielsweise, wenn sie 3 zurückgeben *und* den Systemzustand ändern sollte. Jedes davon ist eine Fähigkeit und sollte unabhängig von anderen Fähigkeiten funktionieren.

Weil nur ein Belang von mehreren fehlschlug, habe ich mich schon bei der Suche nach Bugs, die gar nicht da waren, auf die Suche nach der Nadel im Heuhaufen begeben. Hätte ich mich darum gekümmert, zu überprüfen, ob die anderen Asserts fehlschlugen oder erfolgreich waren, dann hätte ich vielleicht erkannt, dass der Bug an einer ganz anderen Stelle lag.

Manchmal geht jemand hin und findet Bugs, von denen er glaubt, dass sie echt wären. Wenn er sie dann »reparieret«, ist das zuvor fehlgeschlagene Assert zwar erfolgreich, aber die *anderen* Asserts in diesem Test schlagen fehl (oder schlagen weiterhin fehl). Manchmal kann man das komplette Problem nicht sehen, weshalb das Reparieren von Teilen neue Bugs in das System einführen kann, die erst entdeckt werden, nachdem Sie das Resultat jedes einzelnen Asserts enthüllt haben.

Darum ist es im Falle von mehreren Belangen wichtig, dass alle Asserts eine Chance haben, ausgeführt zu werden, auch wenn andere Asserts zuvor schiefgegangen sind. Für die meisten Fälle heißt das, dass nur ein Assert in einen Test eingebaut werden sollte.

Die Verwendung parametrisierter Tests

Sowohl xUnit.net als auch NUnit unterstützen das Konzept des parametrisierten Tests über ein besonderes Attribut namens `[TestCase]`. Das folgende Listing zeigt, wie Sie `[TestCase]` und Attribute verwenden können, um die gleichen Tests mit verschiedenen Parametern in einer einzigen Testmethode auszuführen. Beachten Sie, dass das Attribut `[TestCase]` das Attribut `[Test]` in NUnit ersetzt.

```
[TestCase(1001,1,2,3)]
[TestCase (1,1001,2,3)]
[TestCase (1,2,1001,3)]
public void Sum_HigherThan1000_Ignored(int x,int y, int z,int expected)
{
    Assert.AreEqual(expected, Sum(x, y, z));
}
```

Listing 8.15: Eine refaktorierte Testklasse, die parametrisierte Tests verwendet

Parametrisierte Testmethoden in NUnit und xUnit.net unterscheiden sich von regulären Tests dadurch, dass sie Parameter übernehmen können. In NUnit erwarten sie auch mindestens ein `[TestCase]`-Attribut, das der aktuellen Methode statt des regulären `[Test]`-Attributs vorangestellt werden muss. Das Attribut nimmt eine beliebige Anzahl von Parametern entgegen, die dann zur Laufzeit auf die Parameter, die die Testmethode in ihrer Signatur erwartet, abgebildet werden.

Das Beispiel in Listing 8.15 erwartet vier Argumente. Sie rufen eine Assert-Methode mit den ersten drei Parametern auf und benutzen den letzten als den erwarteten Wert. Dies gibt Ihnen eine deklarative Möglichkeit, einen einzelnen Test mit verschiedenen Inputs zu erzeugen.

Das Beste daran ist, dass selbst dann, wenn eines der `[TestCase]`-Attribute fehlschlägt, die anderen Attribute immer noch vom Test-Runner ausgeführt werden, wodurch Sie das ganze Bild der erfolgreichen/erfolglosen Zustände in allen Tests sehen können.

Wrapper mit try-catch

Manche halten es für eine gute Idee, einen `try-catch`-Block für jedes Assert zu verwenden, um dessen Ausnahme aufzufangen, dann in die Konsole zu schreiben und anschließend mit der nächsten Anweisung weiterzumachen, womit sie die problematische Natur von Ausnahmen in Tests umgehen. Ich denke, die Verwendung von parametrisierten Tests ist eine bei Weitem bessere Möglichkeit, das Gleiche zu erreichen. Verwenden Sie parametrisierte Tests statt `try-catch`-Blöcken um mehrfache Asserts.

Da Sie nun wissen, wie Sie mehrfache Asserts vermeiden, die als mehrfache Tests agieren, werfen wir als Nächstes einen Blick auf das Testen mehrerer Aspekte eines einzelnen Objekts.

8.2.6 Der Vergleich von Objekten

Hier kommt ein anderes Beispiel für einen Test mit mehreren Asserts, aber dieses Mal wird nicht versucht, mehrere Tests in einem einzigen Test agieren zu lassen, sondern mehrere

Aspekte des gleichen Zustands zu prüfen. Wenn auch nur ein Aspekt fehlschlägt, müssen Sie davon wissen.

```
[Test]
public void
    Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields()
{
    LogAnalyzer log = new LogAnalyzer();
    AnalyzedOutput output =
    log.Analyze("10:05\tOpen\tRoy");
    Assert.AreEqual(1,output.LineCount);
    Assert.AreEqual("10:05",output.GetLine(1)[0]);
    Assert.AreEqual("Open",output.GetLine(1)[1]);
    Assert.AreEqual("Roy",output.GetLine(1)[2]);
}
```

Listing 8.16: Das Testen mehrerer Aspekte des gleichen Objekts in einem Test

Dieses Beispiel prüft, ob die Analyseausgabe von `LogAnalyzer` funktioniert hat, indem jedes Feld im resultierenden Objekt separat getestet wird. Sie sollten alle in Ordnung sein oder der Test sollte fehlschlagen.

Machen Sie Tests besser wartbar

Das nächste Listing zeigt eine Möglichkeit, den Test aus Listing 8.16 so umzubauen, dass er leichter zu lesen und zu warten ist.

```
[Test]
public void
    Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
{
    LogAnalyzer log = new LogAnalyzer();
    //setze ein erwartetes Objekt auf:
    AnalyzedOutput expected = new AnalyzedOutput();
    expected.AddLine("10:05", "Open", "Roy");

    AnalyzedOutput output = log.Analyze("10:05\tOpen\tRoy");
    //vergleiche erwartetes und tatsächliches Objekt:
    Assert.AreEqual(expected,output);
}
```

Listing 8.17: Der Vergleich von Objekten statt der Verwendung mehrerer Asserts

Anstatt mehrere Asserts hinzuzufügen, können Sie zum Vergleich ein komplettes Objekt anlegen, alle benötigten Properties setzen und das Resultat mit dem erwarteten Objekt in einem Assert vergleichen. Der Vorteil dieses Ansatzes ist, dass viel leichter verstanden werden kann, was Sie da testen, und dass direkt erkannt wird, dass es sich um einen logischen Block handelt und nicht um mehrere separate Tests.

Beachten Sie, dass Sie für diese Art des Testens die Objekte, die verglichen werden, die `Equals()`-Methode überschreiben müssen, denn sonst wird der Vergleich zwischen den Objekten fehlschlagen. Manche halten das für einen inakzeptablen Kompromiss. Ich benutze ihn von Zeit zu Zeit, aber ich bin mit beidem einverstanden. Entscheiden Sie selbst. Da ich ReSharper verwende, benutze ich Alt-Einfg, dann wähle ich aus dem Menü Generate Equality Members, und BAM! Der ganze Code zur Überprüfung der Gleichheit wird für mich erzeugt. Das ist ziemlich cool.

Das Überschreiben von ToString()

Ein anderer Ansatz ist das Überschreiben der `ToString()`-Methode von zu vergleichenden Objekten, wodurch Sie aussagekräftigere Fehlermeldungen erhalten, wenn Tests schiefgehen. Hier ist zum Beispiel die Ausgabe des Tests in Listing 8.17, wenn er fehlschlägt.

```
TestCase 'AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2'
failed:
  Expected: <AOUT.CH789.LogAn.AnalyzedOutput>
  But was: <AOUT.CH789.LogAn.AnalyzedOutput>
    C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting
              \LogAn.Tests\MultipleAsserts.cs(41,0):
at AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
```

Nicht sehr hilfreich, oder?

Indem wir `ToString()` sowohl in der Klasse `AnalyzedOutput` als auch in der Klasse `LineInfo` implementieren (die Teil des zu vergleichenden Objektmodells sind), können Sie von den Tests eine besser lesbare Ausgabe erhalten. Das nächste Listing zeigt die beiden Implementierungen der Methode `ToString()` in den zu testenden Klassen, gefolgt von der resultierenden Testausgabe.

```
//Überschreibe ToString innerhalb des AnalyzedOutput-Objekts////////
public override string ToString()
{
    StringBuilder sb = new StringBuilder();
    foreach (LineInfo line in lines)
    {
        sb.Append(line.ToString());
    }
    return sb.ToString();
}

//Überschreibe ToString innerhalb jedes LineInfo-Objekts////////////
public override string ToString()
```

```
{
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < this.fields.Length; i++)
    {
        sb.Append(this[i]);
        sb.Append(",");
    }
    return sb.ToString();
}

///TEST OUTPUT///////////////
------ Test started: Assembly: er.dll ------
TestCase 'AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2'
failed:
  Expected: <10:05,Open,Roy,>
  But was: <>
    C:\GlobalShare\InSync\Book\Code\ARtOfUniTesting
\LogAn.Tests\MultipleAsserts.cs(41,0):
at AOUT.CH8.LogAn.Tests.MultipleAsserts
.Analyze_SimpleStringLine_UsesDefaulTabDelimiterToParseFields2()
```

Listing 8.18: Um eine saubere Ausgabe zu erhalten, wird `ToString()` in den Klassen implementiert, die verglichen werden.

Nun ist die Ausgabe des Tests viel klarer und Sie können erkennen, dass Sie zwei sehr unterschiedliche Objekte haben. Eine klare Ausgabe erleichtert das Verständnis dafür, warum ein Test fehlschlägt, und führt zu einer besseren Wartbarkeit.

Ein anderer Grund, warum die Pflege von Tests schwierig werden kann, liegt dann vor, wenn Sie die Tests durch eine Überspezifizierung zu anfällig machen.

8.2.7 Vermeiden Sie eine Überspezifizierung der Tests

Ein überspezifizierter Test enthält Annahmen darüber, wie eine bestimmte zu testende Unit (Produktionscode) ihr internes Verhalten implementieren sollte, anstatt nur das Endverhalten auf Korrektheit zu prüfen.

Auf diese Arten sind Unit Tests häufig überspezifiziert:

- Ein Test überprüft den rein internen Zustand eines zu testendes Objekts.
- Ein Test verwendet mehrere Mocks.
- Ein Test benutzt sowohl Stubs auch als Mocks.
- Ein Test setzt eine bestimmte Reihenfolge oder genau passende Zeichenketten voraus, wenn das gar nicht erforderlich ist.

Lassen Sie uns einige Beispiele für überspezifizierte Tests betrachten.

Die Spezifizierung rein internen Verhaltens

Das folgende Listing zeigt einen Test der Methode `Initialize()` von `LogAnalyzer`, der den internen Zustand und keine äußere Funktionalität testet.

```
[Test]
public void
    Initialize_WhenCalled_SetsDefaultDelimiterIsTabDelimiter()
{
    LogAnalyzer log = new LogAnalyzer();

    Assert.AreEqual(null,log.GetInternalDefaultDelimiter());
    log.Initialize();
    Assert.AreEqual('\t', log.GetInternalDefaultDelimiter());
}
```

Listing 8.19: Ein überspezifizierter Test, der rein internes Verhalten testet

Dieser Test ist überspezifiziert, denn er testet nur den internen Zustand des `LogAnalyzer`-Objekts. Weil dieser Zustand intern ist, könnte er sich später noch ändern.

Unit Tests sollten den öffentlichen Kontrakt und die öffentliche Funktionalität eines Objekts testen. In diesem Beispiel ist der getestete Code weder Teil eines öffentlichen Kontrakts noch eines öffentlichen Interface.

Die Verwendung von Stubs als Mocks

Die Verwendung von Mocks statt Stubs ist eine verbreitete Überspezifizierung. Lassen Sie uns ein Beispiel betrachten.

Stellen Sie sich vor, Sie haben ein Daten-Repository, auf das Sie sich verlassen, um gefälschte Daten zurückzugeben. Was geschieht nun, wenn Sie den Stub, der gefälschte Daten zurückgibt, auch benutzen, um zu überprüfen, dass er aufgerufen wurde? Das folgende Listing zeigt dies.

```
[Test]
public void IsLoginOK_UserDoesNotExist_ReturnsFalse()
{
    IDataRepository fakeData = A.Fake<IDataRepository>();

    A.CallTo(()=> fakeData.GetUserByName(A<string>.Ignored))
        .Returns(null);

    LoginManager login = new LoginManager(fakeData);

    bool result =
        login.IsLoginOK("UserNameThatDoesNotExist","anypassword");
```

```
    Assert.IsFalse(result);
    //Sie müssen nicht überprüfen, dass der Stub aufgerufen wird.
    //Das ist Überspezifizierung:
    A.CallTo(()=>fakeData.GetUserByName("UserNameThatDoesNotExist"))
        .MustHaveHappened();
}
```

Listing 8.20: Ein überspezifizierter Test, der rein internes Verhalten testet

Der Test ist überspezifiziert, denn er testet die Interaktion zwischen dem Repository-Stub und `LoginManager` (unter Verwendung von FakeItEasy). Der Test sollte die zu testende Methode ihren eigenen internen Algorithmus durchlaufen lassen und die Rückgabewerte überprüfen. Dadurch würde Ihr Test weniger zerbrechlich werden. So, wie er ist, wird er beschädigt, wenn Sie entscheiden, dass Sie einen internen Aufruf hinzufügen wollen, oder ihn optimieren möchten, indem Sie die Aufrufparameter ändern. Solange der Wert des Endergebnisses richtig ist, sollte Ihr Test sich nicht darum kümmern, ob irgendetwas intern aufgerufen oder nicht aufgerufen wurde.

In einem besser definierten Test würde die letzte Codezeile nicht existieren.

Schließlich tendieren Entwickler auch zur Überspezifikation ihrer Tests, indem sie sie zu häufig mit Annahmen versehen.

Die Annahme einer Reihenfolge oder einer genauen Übereinstimmung, wenn das gar nicht erforderlich ist

Ein anderes, verbreitetes und gern wiederholtes Muster ist es, Asserts auf feste Strings im Rückgabewert oder eine Property der Unit anzuwenden, wenn nur ein bestimmter Teil des Strings erforderlich ist. Fragen Sie sich selbst: »Kann ich `string.Contains()` statt `string.Equals()` verwenden?«

Das Gleiche gilt für Listen und Collections. Besser ist es, dafür zu sorgen, dass eine Collection ein erwartetes Element enthält, als darauf zu prüfen, dass sich ein Element an einer bestimmten Stelle in der Collection befindet (außer wenn genau dies erwartet wird).

Durch diese Art von kleinen Änderungen können Sie garantieren, dass der Test erfolgreich sein wird, solange der String oder die Collection das enthält, was erwartet wird. Auch wenn sich die Implementierung oder die Reihenfolge innerhalb des Strings oder der Collection ändert, müssen Sie nicht zurückgehen und jedes kleine Zeichen, das Sie dem String hinzugefügt haben, wieder ändern.

Und nun lassen Sie uns zur dritten und letzten Säule von »guten« Unit Tests kommen: der Lesbarkeit.

8.3 Das Schreiben lesbarer Tests

Ohne Lesbarkeit sind die Tests, die Sie schreiben, nahezu bedeutungslos. Lesbarkeit ist das Bindeglied zwischen demjenigen, der den Test geschrieben hat, und der armen Seele, die ihn ein paar Monate später lesen muss. Tests sind Geschichten, die Sie der nächsten Generation von Entwicklern in einem Projekt erzählen. Sie erlauben es einem Entwickler, genau zu erkennen, woraus eine Applikation besteht und wo sie angefangen hat.

Dieser Abschnitt handelt davon, wie Sie dafür sorgen können, dass die Entwickler, die nach Ihnen kommen, in der Lage sein werden, den von Ihnen geschriebenen Produktionscode und seine Tests zu pflegen und gleichzeitig zu verstehen, was sie da tun und wo sie es tun sollten.

Es gibt mehrere Facetten der Lesbarkeit:

- Die Benennung von Unit Tests
- Die Benennung von Variablen
- Das Anlegen guter Assert-Nachrichten
- Das Trennen der Asserts von Aktionen

Lassen Sie uns diese Punkte Schritt für Schritt durchgehen.

8.3.1 Die Benennung der Unit Tests

Namensstandards sind wichtig, denn sie geben Ihnen bequeme Regeln und Schablonen, die umreißen, was Sie zu den Tests erklären sollten. Der Name des Tests besteht aus drei Teilen:

- *Der Name der zu testenden Methode* – Dies ist essenziell, damit Sie leicht sehen können, wo sich die getestete Logik befindet. Die Tatsache, dass dies der erste Teil des Testnamens ist, erlaubt eine einfache Navigation und ein »as-you-type«-IntelliSense (wenn Ihre IDE das unterstützt) in der Testklasse.
- *Das Szenario, in dem getestet wird* – Dieser Teil gibt Ihnen den »mit«-Teil des Namens: »Wenn ich Methode X *mit einem Null-Wert* aufrufe, dann sollte sie Y tun.«
- *Das erwartete Verhalten, wenn das Szenario aufgerufen wird* – Dieser Teil gibt im Klartext an, was die Methode tun oder zurückgeben sollte oder wie sie sich in Abhängigkeit vom aktuellen Szenario verhalten sollte: »Wenn ich Methode X mit einem Null-Wert aufrufe, *dann sollte sie Y tun*.«

Das Entfernen auch nur eines dieser Teile aus dem Namen des Tests kann den Leser des Tests veranlassen, sich darüber zu wundern, was da vor sich geht, und ihn zum Lesen des Testcodes bringen. Ihr wichtigstes Ziel ist es, den nächsten Entwickler von der Last zu befreien, den Testcode lesen zu müssen, um verstehen zu können, was der Test überhaupt testet.

Ein üblicher Weg, die drei Teile des Testnamens zu schreiben, ist es, sie mit Unterstrichen zu trennen, wie etwa: `ZuTestendeMethode_Szenario_Verhalten()`. Hier kommt ein Test, der dieser Namenskonvention folgt.

```
[Test]
public void
  AnalyzeFile_FileWith3LinesAndFileProvider_ReadsFileUsingProvider()
{
//...
}
```

Listing 8.21: Ein Test mit einem dreiteiligen Namen

Die Methode im Listing testet die Methode `AnalyzeFile`, übergibt ihr eine Datei mit drei Zeilen sowie einen dateilesenden Provider und erwartet von ihr, dass sie den Provider zum Lesen der Datei benutzt.

Wenn die Entwickler sich an diese Namenskonvention halten, wird es für andere Entwickler leicht sein, neu einzusteigen und die Tests zu verstehen.

8.3.2 Die Benennung der Variablen

Wie Sie die Variablen in Unit Tests benennen, ist mindestens genauso wichtig wie die Konventionen der Variablenbenennung im Produktionscode. Abgesehen von ihrer Hauptfunktion für das Testen dienen Tests auch als eine Form der Dokumentation für eine API. Indem Sie den Variablen gute Namen geben, können Sie dafür sorgen, dass diejenigen, die Ihre Tests lesen, so schnell wie möglich verstehen, was Sie zu *beweisen* versuchen (im Gegensatz zum Verstehen dessen, was Sie mit dem Produktionscode zu *erreichen* versuchen). Das nächste Listing zeigt das Beispiel eines schlecht benannten und schlecht geschriebenen Tests. Ich nenne ihn in dem Sinne »unleserlich«, als dass ich nicht herausbekommen kann, worum es in diesem Test eigentlich geht.

```
[Test]
public void BadlyNamedTest()
{
    LogAnalyzer log = new LogAnalyzer();

    int result= log.GetLineCount("abc.txt");

    Assert.AreEqual(-100,result);
}
```

Listing 8.22: Ein unleserlicher Testname

In diesem Fall benutzt das Assert eine »magische« Zahl (-100) (eine Zahl, deren Bedeutung der Entwickler auf irgendeine Art und Weise kennen muss). Weil Sie keinen beschreibenden Namen für die Bedeutung der Zahl haben, können Sie nur *vermuten*, was sie wohl bedeuten könnte. Der Testname sollte Ihnen hier ein wenig helfen, aber er müsste auch überarbeitet werden, um es vorsichtig auszudrücken.

Ist -100 eine Art von Ausnahme? Ist es ein gültiger Rückgabewert? Hier haben Sie die Wahl:

- Sie können das Design der API ändern und sie eine Ausnahme statt der Rückgabe von -100 werfen lassen (wenn wir annehmen, dass -100 irgendeine Art von verbotenem Resultat ist).
- Sie können das Resultat mit irgendeiner Art von passend benannter Konstanten oder Variablen vergleichen, wie im nächsten Listing gezeigt.

```
[Test]
public void BadlyNamedTest()
{
    LogAnalyzer log = new LogAnalyzer();
```

```
    int result= log.GetLineCount("abc.txt");

    const int COULD_NOT_READ_FILE = -100;

    Assert.AreEqual(COULD_NOT_READ_FILE,result);
}
```

Listing 8.23: Eine besser lesbare Version des Tests

Der Code in Listing 8.23 ist deutlich besser, denn Sie können die Intention des Rückgabewerts direkt verstehen.

Der letzte Teil eines Tests ist gewöhnlich das Assert und Sie müssen aus der Assert-Nachricht möglichst viel herausschlagen. Wenn die Prüfung des Asserts fehlschlägt, dann ist das Erste, was der Benutzer sehen wird, diese Nachricht.

8.3.3 Benachrichtigen Sie sinnvoll

Das Schreiben einer guten Assert-Nachricht ist dem Schreiben einer guten Ausnahmenachricht sehr ähnlich. Es ist einfach, es schlecht zu machen, ohne es zu bemerken, und es macht für diejenigen, die sie lesen müssen, einen großen Unterschied (auch hinsichtlich der benötigten Zeit).

Es gibt einige Kernpunkte, die beim Schreiben einer Nachricht in einer Assert-Anweisung zu beachten sind:

- Wiederholen Sie nicht, was das eingebaute Test-Framework sowieso in die Konsole schreibt.
- Wiederholen Sie nicht, was der Testname schon erklärt.
- Wenn Sie nichts Sinnvolles zu sagen haben, dann sagen Sie gar nichts.
- Schreiben Sie, was passieren sollte oder was fehlschlug und daher nicht passierte, und erwähnen Sie möglichst auch, wann es passieren sollte.

Das Folgende zeigt das Beispiel einer schlechten Assert-Nachricht und die resultierende Ausgabe.

```
[Test]
public void BadAssertMessage()
{
    LogAnalyzer log = new LogAnalyzer();
    int result= log.GetLineCount("abc.txt");
    const int COULD_NOT_READ_FILE = -100;
    Assert.AreEqual(COULD_NOT_READ_FILE,result,
                    "result was {0} instead of {1}",
                    result,COULD_NOT_READ_FILE);
}

//Die Ausführung dieses Tests würde Folgendes produzieren:
TestCase 'AOUT.CH8.LogAn.Tests.Readable.BadAssertMessage'
```

```
failed:
  result was -1 instead of -100
  Expected: -100
  But was: -1
    C:\GlobalShare\InSync\Book\Code
\ARtOfUniTesting\LogAn.Tests\Readable.cs(23,0)
: at AOUT.CH8.LogAn.Tests.Readable.BadAssertMessage()
```

Listing 8.24: Eine schlechte Assert-Nachricht, die wiederholt, was das Test-Framework bereits ausgibt

Wie Sie sehen können, wird die Nachricht wiederholt. Die Assert-Nachricht hat nichts hinzugefügt außer ein paar überflüssigen Wörtern. Es wäre besser gewesen, nichts auszugeben und stattdessen den Test sinnvoller zu benennen. Eine klarere Assert-Nachricht würde etwa so aussehen:

```
Der Aufruf von GetLineCount() für eine nicht existierende Datei sollte
COULD_NOT_READ_FILE zurückgeben.
```

Da Ihre Assert-Nachrichten nun verständlich sind, ist es an der Zeit, dafür zu sorgen, dass das Assert in einer anderen Zeile geschieht als der Methodenaufruf.

8.3.4 Das Trennen der Asserts von den Aktionen

Dies ist ein kurzer Abschnitt, aber nichtsdestoweniger ein wichtiger. Um der Lesbarkeit willen sollten Sie es vermeiden, das Assert und den Methodenaufruf in der gleichen Anweisung unterzubringen.

Das folgende Listing zeigt ein gutes Beispiel, Listing 8.26 ein schlechtes.

```
[Test]
public void BadAssertMessage()
{
    //etwas Code hier
    int result= log.GetLineCount("abc.txt");
    Assert.AreEqual(COULD_NOT_READ_FILE,result);
}
```

Listing 8.25: Das Trennen der Assert-Anweisung von der Sache, auf die das Assert angewendet wird, erhöht die Lesbarkeit.

```
[Test]
public void BadAssertMessage()
{
    // etwas Code hier
    Assert.AreEqual(COULD_NOT_READ_FILE,log.GetLineCount("abc.txt"));
}
```

Listing 8.26: Die Assert-Anweisung nicht von der Sache zu trennen, auf die das Assert angewendet wird, macht das Lesen schwierig.

Sehen Sie den Unterschied zwischen den beiden Beispielen? Listing 8.26 ist im Kontext eines realen Tests viel schwieriger zu lesen und zu verstehen, denn der Aufruf der Methode `GetLineCount()` liegt innerhalb des Aufrufs der Assert-Benachrichtigung.

8.3.5 Aufbauen und Abreißen

Aufbau- und Abbaumethoden in Unit Tests können bis zu dem Punkt missbraucht werden, an dem die Tests oder die Methoden unleserlich werden. Meist ist die Situation in den Setup-Methoden schlechter als in den Abbaumethoden.

Lassen Sie uns einen möglichen Missbrauch genauer betrachten. Wenn Sie Stubs und Mocks haben, die in einer Setup-Methode aufgesetzt werden, dann heißt das, sie werden nicht im eigentlichen Test aufgesetzt. Das wiederum bedeutet, dass wer auch immer Ihren Test liest, vielleicht nicht einmal die Verwendung der Mock-Objekte bemerkt oder was von ihnen im Test erwartet wird.

Die Lesbarkeit ist wesentlich besser, wenn die Mock-Objekte direkt im Test selbst initialisiert werden, zusammen mit all ihren Anforderungen. Wenn Sie wegen der Lesbarkeit besorgt sind, dann können Sie die Erzeugung der Mocks in eine Hilfsmethode auslagern, die von jedem Test aufgerufen wird. Auf diese Weise wird jeder, der den Test liest, genau wissen, was angelegt wird, statt an mehreren Stellen suchen zu müssen.

Hinweis

Ich habe der besseren Wartbarkeit wegen einige Male komplette Testklassen geschrieben, die keine Setup-Methode besaßen, sondern nur Hilfsmethoden, die von jedem Test aufgerufen wurden. Diese Klassen sind immer noch lesbar und wartbar.

8.4 Zusammenfassung

Wenige Entwickler schreiben Tests, denen sie vertrauen können, wenn sie zum ersten Mal mit dem Schreiben von Unit Tests beginnen. Es braucht Disziplin und Vorstellungskraft, die richtigen Dinge zu tun. Ein Test, dem man vertrauen kann, ist zunächst eine trügerische Bestie, aber wenn man es richtig anstellt, spürt man den Unterschied sofort.

Einige Wege, um diese Art der Vertrauenswürdigkeit zu erreichen, schließen das Am-Leben-Erhalten guter Tests und das Entfernen oder Umbauen schlechter Tests mit ein. Wir haben mehrere solcher Methoden in diesem Kapitel diskutiert. Im Rest des Kapitels ging es um Probleme, die innerhalb der Tests auftauchen können, wie die Logik, das Testen mehrerer Dinge, die Einfachheit der Ausführung usw. All diese Dinge zusammenzusetzen, kann schon eine Form von Kunst sein.

Wenn es eine einzige Erkenntnis gibt, die Sie aus diesem Kapitel mitnehmen sollten, dann ist es diese: Tests wachsen und ändern sich mit dem zu testenden System.

Das Thema vom Schreiben *wartbarer* Tests hat in den letzten paar Jahren Fahrt aufgenommen, aber während ich dies schreibe, ist dazu noch nicht viel in der Unit-Testing- und TDD-Literatur erschienen. Und das aus gutem Grund. Ich glaube, dass dies der nächste Schritt in der Lern-Entwicklung der Techniken des Unit Testings sein wird. Der erste Schritt, sich das anfängliche Wissen anzueignen (was ein Unit Test ist und wie man einen schreibt), wird an vielen Stellen behandelt. Der zweite Schritt bezieht die Verfeinerungen

der Techniken zur Verbesserung aller Aspekte des von uns geschriebenen Codes mit ein und schaut auch auf andere Faktoren wie die Wartbarkeit und die Lesbarkeit. Es ist dieser kritische Schritt, auf den dieses Kapitel (und der größte Teil des Buches) den Blick richtet.

Letztlich ist es einfach: Lesbarkeit geht Hand in Hand mit Wartbarkeit und Vertrauenswürdigkeit. Jemand, der Ihre Tests lesen kann, kann sie verstehen und warten und wird ihnen auch vertrauen, wenn sie erfolgreich ablaufen. Wenn dieser Punkt erreicht ist, dann sind Sie bereit, mit Änderungen umzugehen und den Code zu ändern, sobald eine Änderung notwendig ist, denn Sie werden erkennen, wenn Dinge kaputtgehen.

In den nächsten Kapiteln werden wir einen umfassenderen Blick auf Unit Tests als Teil eines größeren Systems werfen: wie sie in eine Organisation integriert werden können und wie sie in existierende Systeme und Legacy-Code passen. Sie werden lernen, was den Code testbar macht, wie Sie mit Blick auf die Testbarkeit designen und wie Sie existierenden Code in einen testbaren Zustand bringen können.

Teil IV

Design und Durchführung

Diese abschließenden Kapitel behandeln die Probleme, denen Sie begegnen, sowie die Methoden, die Sie benötigen werden, wenn Sie das Unit Testing in eine bestehende Organisation oder in vorhandenen Code einführen wollen.

In Kapitel 9 werden wir uns mit dem schwierigen Thema der Implementierung von Unit Tests in einer Organisation beschäftigen und Methoden diskutieren, die Ihnen den Job erleichtern können. Dieses Kapitel gibt Antworten auf einige schwierige Fragen, die sich häufig bei der ersten Implementierung des Unit Testings stellen.

In Kapitel 10 werden wir uns verbreitete Probleme im Zusammenhang mit Legacy-Code und einige Tools für den Umgang damit ansehen.

Kapitel 11 beschäftigt sich mit einer häufigen Diskussion um das Unit Testing. Sollte die Testbarkeit ein Design-Ziel sein? Was ist testbares Design überhaupt?

In diesem Teil:

Kapitel 9

Die Integration von Unit Tests in die Organisation

Dieses Kapitel behandelt

- wie Sie zu einem Agenten des Wandels werden
- wie Sie den Wandel »Top-down« oder »Bottom-up« implementieren
- wie Sie sich darauf vorbereiten, die schwierigen Fragen zum Unit Testing zu beantworten

Als Berater habe ich verschiedenen Unternehmen, großen wie kleinen, geholfen, die testgetriebene Entwicklung und das Unit Testing in ihre Organisationskultur einzuführen. Manchmal ist das schiefgegangen, aber die Firmen, bei denen es erfolgreich war, haben mehrere Dinge gemeinsam. Dieses Kapitel greift auf Geschichten aus beiden Lagern zurück, während es sich mit den folgenden Themen beschäftigt:

- *Werden Sie ein Agent des Wandels* – Die ersten Schritte, die Sie unternehmen sollten, bevor Sie irgendeine Änderung einführen.
- *Wege zum Erfolg* – Dinge, die zu erfolgreichen Änderungen innerhalb eines Prozesses beitragen.
- *Wege zum Misserfolg* – Dinge, die das, was Sie umzusetzen versuchen, zerstören können.
- *Schwierige Fragen und Antworten* – Die häufigsten Fragen, wenn ein Team das Unit Testing einführt.

In jeder Art von Organisation ist der Versuch, die Gewohnheiten zu ändern, eher psychologischer als technischer Natur. Sie mögen den Wechsel nicht und Änderungen sind gewöhnlich von einer Menge Furcht, Ungewissheit und Zweifel begleitet. Für die meisten wird es kein Spaziergang sein, wie Sie in diesem Kapitel sehen werden.

9.1 Schritte, um ein Agent des Wandels zu werden

Wenn Sie der Agent des Wandels in Ihrer Organisation werden, sollten Sie zunächst diese Rolle akzeptieren. Ob Sie das wollen oder nicht, man wird in Ihnen die Person sehen, die verantwortlich ist für das, was geschieht, und Verstecken gilt nicht. Tatsächlich kann ein Verstecken die Dinge schrecklich schiefgehen lassen.

Wenn Sie beginnen, die Änderungen umzusetzen, werden Ihnen die Leute die schwierigen Fragen stellen, die sie umtreiben. Wie viel Zeit wird dies »verschwenden«? Was bedeutet das für mich als QS-Mitarbeiter? Woher wissen wir, dass es funktioniert? Seien Sie aufdiese Fragen vorbereitet. Die Antworten auf die häufigsten Fragen werden in Abschnitt 9.4 diskutiert. Sie werden merken, dass es Ihnen ungemein dabei hilft, schwierige Entscheidun-

gen zu treffen und diese Fragen zu beantworten, wenn Sie andere in Ihrer Organisation überzeugen, bevor Sie damit anfangen, Änderungen umzusetzen.

Letztlich muss jemand am Steuerruder stehen und dafür sorgen, dass die Änderungen nicht aus Mangel an Schwungkraft eingehen. Derjenige sind Sie. Es gibt Möglichkeiten, die Dinge am Leben zu erhalten, wie Sie in den nächsten Abschnitten erfahren werden.

9.1.1 Seien Sie auf die schweren Fragen vorbereitet

Machen Sie Ihre Hausaufgaben und bereiten Sie sich vor. Lesen Sie die Antworten am Ende dieses Kapitels und schauen Sie sich die entsprechenden Quellen an. Lesen Sie Foren, Mailing-Listen und Blogs und beraten Sie sich mit Ihren Kollegen. Wenn Sie Ihre eigenen schweren Fragen beantworten können, sind die Chancen gut, dass Sie auch die Fragen von jemand anderem beantworten können.

9.1.2 Überzeugen Sie Insider: Champions und Blockierer

Einsamkeit ist eine schreckliche Sache und nicht viele Dinge lassen Sie die Einsamkeit mehr spüren, als in einer Organisation gegen den Strom zu schwimmen. Wenn Sie der Einzige sind, der glaubt, was Sie tun, sei eine gute Idee, dann gibt es wenige Gründe für jemand anderen, sich anzustrengen und Ihre Vorschläge umzusetzen. Überlegen Sie, wer Ihren Anstrengungen nützen oder schaden kann: die Champions und die Blockierer.

Champions

Wenn Sie anfangen, auf einen Wandel zu drängen, dann identifizieren Sie am besten diejenigen, von denen Sie am wahrscheinlichsten Unterstützung für Ihr Bestreben erwarten können. Sie werden Ihre *Champions* sein. Gewöhnlich handelt es sich dabei um Leute, die Dinge frühzeitig aufgreifen oder offen für Neues sind und die Dinge, die Sie vorschlagen, zumindest ausprobieren. Sie mögen vielleicht schon teilweise überzeugt sein, warten aber noch auf einen Impuls, um den Wandel zu starten. Sie mögen es vielleicht schon selbst versucht haben und sind daran gescheitert.

Treten Sie als Erstes an diese Leute heran und fragen Sie sie nach ihrer Meinung zu dem, was Sie zu tun gedenken. Vielleicht erzählen sie Ihnen Dinge, die Sie noch nicht in Betracht gezogen haben: Teams, mit denen der Anfang leichter fällt, oder Stellen, an denen man offener für Änderungen ist. Vielleicht erzählen sie Ihnen auch aus eigener Erfahrung, worauf Sie achten sollten.

Indem Sie sie ansprechen, sorgen Sie dafür, dass sie Teil des Prozesses werden. Wer sich als Teil eines Prozesses begreift, versucht meist, ihn zu unterstützen. Machen Sie sie zu Ihren Champions: Fragen Sie sie, ob sie Ihnen helfen können und ob die anderen sich mit Fragen an sie wenden können. Bereiten Sie sie darauf vor.

Blockierer

Als Nächstes identifizieren Sie die *Blockierer*. Das sind diejenigen innerhalb der Organisation, die sich am wahrscheinlichsten gegen die Änderungen, die Sie durchsetzen wollen, wehren werden. Beispielsweise könnte ein Manager dem Hinzufügen von Unit Tests widersprechen und behaupten, dass sie die Entwicklungszeit zu sehr verlängern und die

Menge des Codes, der gewartet werden muss, vergrößern würden. Machen Sie sie zu einem Teil des Prozesses statt des Widerstands gegen ihn, indem Sie ihnen eine aktive Rolle im Prozess geben (zumindest denen, die willens und fähig sind).

Die Gründe, aus denen die Leute bestimmten Änderungen ablehnend gegenüberstehen, können variieren und die Antworten auf einige der möglichen Einwände sind in Abschnitt 9.5 enthalten. Einige werden sich Sorgen um ihre Jobs machen und andere sind zufrieden damit, wie die Dinge sind, und lehnen deshalb jede Änderung ab.

Zu diesen Leuten zu gehen und all die Dinge aufzuzählen, die sie hätten besser machen können, ist meist wenig hilfreich, wie ich auf die harte Tour lernen musste. Keiner mag es, wenn ihm gesagt wird, was er nicht gut macht. Stattdessen ist es besser, diese Leute um Hilfe bei der Umsetzung zu bitten, beispielsweise indem sie für die Definition von Coding-Standards der Unit Tests verantwortlich sind oder indem sie jeden zweiten Tag mit Kollegen Code- und Test-Reviews durchführen. Oder nehmen Sie sie in das Team auf, das Kursunterlagen oder externe Berater auswählt. Sie werden ihnen damit eine neue Verantwortung geben, die ihnen das Gefühl vermittelt, dass ihnen vertraut wird und dass sie innerhalb der Organisation wichtig sind. Sie müssen ein Teil des Wandels werden oder sie werden ihn so gut wie sicher untergraben.

9.1.3 Identifizieren Sie mögliche Einstiegspunkte

Finden Sie heraus, wo in der Organisation Sie mit der Umsetzung von Änderungen beginnen können. Die meisten erfolgreichen Umsetzungen nehmen einen beständigen Verlauf. Starten Sie in einem kleinen Team mit einem Pilotprojekt und schauen Sie, was passiert. Wenn alles gut läuft, dann gehen Sie weiter zu anderen Teams und anderen Projekten.

Hier sind ein paar Tipps, die Ihnen auf dem Weg helfen werden:

- Wählen Sie kleinere Teams.
- Teilen Sie das Team in kleinere Einheiten auf.
- Beachten Sie die Projekt-Realisierbarkeit.
- Setzen Sie Code-Reviews und Test-Reviews als Lehrwerkzeuge ein.

Diese Tipps können Sie auch in einer sehr unfreundlichen Umgebung weit bringen.

Wählen Sie kleinere Teams

Meist ist es einfach, mögliche Teams, die für den Anfang geeignet sind, zu erkennen. Sie werden es gewöhnlich vorziehen, mit einem kleinen Team an einem weniger wichtigen Projekt mit geringen Risiken zu arbeiten. Wenn das Risiko minimal ist, wird es leichter sein, die anderen zu überzeugen, Ihre vorgeschlagenen Änderungen auszuprobieren.

Ein Vorbehalt ist, dass das Team aus Mitgliedern bestehen muss, die offen dafür sind, die Art und Weise, wie sie arbeiten, zu ändern und neue Fähigkeiten zu erlernen. Ironischerweise sind diejenigen mit der geringsten Erfahrung im Team am ehesten bereit für Änderungen, während diejenigen mit der meisten Erfahrung dazu tendieren, an ihrer Art, die Dinge anzupacken, festzuhalten. Wenn Sie ein Team mit einem erfahrenen Leiter finden können, der offen für Änderungen ist, das aber auch weniger erfahrene Entwickler umfasst, dann ist es wahrscheinlich, dass Ihnen das Team wenig Widerstand entgegensetzen wird. Gehen Sie zum Team und fragen Sie es nach seiner Meinung, ob es eine sol-

che Vorreiterrolle übernehmen will. Es wird Ihnen sagen, ob dies der richtige Platz zum Starten ist.

Teilen Sie das Team in kleinere Einheiten auf

Eine andere mögliche Herangehensweise für einen Pilottest ist es, ein Unterteam innerhalb eines existierenden Teams zu bilden. Nahezu jedes Team wird eine Komponente haben, die einem »Schwarzen Loch« ähnelt und die gepflegt werden muss, und obwohl vieles funktioniert, wird es auch eine Reihe von Bugs geben. So einer Komponente Features hinzuzufügen, ist eine harte Aufgabe und diese Art von Mühe kann die Leute dazu veranlassen, mit einem Pilotprojekt zu experimentieren.

Beachten Sie die Projekt-Realisierbarkeit

Sorgen Sie bei einem Pilotprojekt dafür, dass Sie nicht mehr abbeißen, als Sie auch verdauen können. Es braucht mehr Erfahrung, um schwierigere Projekte durchzuführen, weshalb es am besten ist, wenn Sie wenigstens zwei Optionen haben – ein kompliziertes Projekt und ein einfacheres – und zwischen beiden wählen können.

Jetzt sind Sie mental auf die bevorstehende Aufgabe vorbereitet. Nun ist es an der Zeit, den Blick auf Dinge zu werfen, die Sie unternehmen können, damit alles glattgeht (oder überhaupt geht).

Setzen Sie Code-Reviews und Test-Reviews als Lehrwerkzeuge ein

Wenn Sie der fachliche Leiter in einem kleinen Team (bis zu 8 Personen) sind, besteht eine der besten Möglichkeiten des Unterrichtens darin, Code-Reviews, die auch Test-Reviews einschließen, zu institutionalisieren. Die Idee dahinter ist, dass Sie den anderen beibringen, wonach Sie in den Tests Ausschau halten und wie Sie über das Schreiben von Tests und die Anwendung von TDD denken, während Sie ihren Code und ihre Tests begutachten. Hier sind ein paar Tipps:

- Führen Sie die Reviews persönlich durch und nicht über eine Remote-Software. Die persönliche Beziehung sorgt für einen besseren nichtverbalen Informationsaustausch, wodurch das Lernen besser und schneller vonstattengeht.
- In den ersten paar Wochen begutachten Sie jede einzelne Codezeile, die eingecheckt wird. Das wird Ihnen helfen, das Problem des »wir haben nicht geglaubt, dass dieser Code begutachtet werden muss« zu vermeiden. Wenn es einfach keine rote Linie gibt (jeder Code wird begutachtet), so kann sie auch nicht überschritten werden und kein Code wird an ihr entlangbewegt.
- Nehmen Sie eine dritte Person zu Ihren Code-Reviews hinzu, die danebensitzt und lernt, wie Sie den Code begutachten. Das wird es ihr ermöglichen, später selber Code-Reviews durchzuführen und es anderen beizubringen, damit Sie kein Engpass für das Team werden, weil Sie der Einzige sind, der in der Lage ist, ein Code-Review durchzuführen. Die Idee ist, bei den anderen die Fähigkeit zum Code-Review zu entwickeln und sie mehr Verantwortung übernehmen zu lassen.

Wenn Sie mehr über diese Technik erfahren wollen, dann schauen Sie auf die Seite `http://5whys.com/blog/what-should-a-good-code-review-look-and-feel-like.html`, wo ich in meinem Blog für fachliche Leiter darüber geschrieben habe.

9.2 Wege zum Erfolg

Es gibt im Wesentlichen zwei Arten, auf die eine Organisation oder ein Team den Änderungsprozess beginnen kann: *Bottom-up* oder *Top-down* (und manchmal beide). Wie Sie sehen werden, unterscheiden sich die beiden Wege grundlegend voneinander und jeder könnte der richtige Ansatz für Ihr Team oder Ihre Firma sein. Den einen richtigen Weg gibt es nicht.

Während Sie Fortschritte machen, müssen Sie lernen, wie Sie das Management davon überzeugen, dass Ihre Anstrengungen auch deren Anstrengungen sein sollten, oder wann es klug wäre, externe Hilfe zu holen. Es ist wichtig, Fortschritte sichtbar zu machen, genauso wie es wichtig ist, klare und messbare Ziele zu setzen. Die Identifizierung und das Umschiffen von Hindernissen sollten ebenfalls weit oben auf Ihrer Liste stehen. Es gibt viele Kämpfe, die Sie ausfechten können, aber Sie sollten die richtigen auswählen.

9.2.1 Guerilla-Implementierung (Bottom-up)

Bei der Implementierung im Guerilla-Stil beginnen Sie mit einem Team, erzielen Resultate und erst dann überzeugen Sie andere, dass es die Sache wert ist. Gewöhnlich sind die Vorkämpfer der Guerilla-Implementierung die Mitglieder eines Teams, das es leid ist, die Dinge auf eine vorgeschriebene Weise zu tun. Sie brechen auf, um die Dinge anders zu machen; sie lernen die Dinge auf eigene Faust und lassen den Wandel geschehen. Wenn das Team Resultate vorweist, dann entscheiden vielleicht andere in der Organisation, in ihren Teams ähnliche Änderungen umzusetzen.

Manchmal ist die Implementierung im Guerilla-Stil ein Prozess, der zuerst von den Entwicklern *angewendet* wird und dann vom Management. Manchmal ist es ein Prozess, der zuerst von den Entwicklern *befürwortet* wird und dann vom Management. Der Unterschied besteht darin, dass Sie das Erstere verdeckt erreichen können, ohne dass die »höheren Mächte« davon erfahren. Das Letztere wird gemeinsam mit dem Management umgesetzt.

Es liegt an Ihnen, herauszufinden, welcher Ansatz besser funktionieren wird. Manchmal sind verdeckte Operationen der einzige Weg, um die Dinge zu ändern. Vermeiden Sie es, wenn Sie können, aber wenn es keinen anderen Weg gibt und Sie sicher sind, dass der Wandel notwendig ist, dann können Sie nur das tun.

Nehmen Sie dies nicht als Empfehlung, mit einem solchen Schritt Ihre Karriere zu beenden. Entwickler machen die ganze Zeit Dinge, ohne um Erlaubnis zu fragen: das Debuggen von Code, das Lesen von E-Mails, das Schreiben von Code-Kommentaren, das Entwerfen von Flussdiagrammen usw. Dies alles sind Aufgaben, die Entwickler als regulären Teil ihres Jobs durchführen. Das Gleiche gilt für das Unit Testing. Die meisten Entwickler schreiben bereits Tests auf die eine oder andere Weise (automatisiert oder nicht). Die Idee ist, diese Zeit, die schon in die Tests investiert wird, in etwas umzuleiten, das langfristig Vorteile bringen wird.

9.2.2 Überzeugen Sie das Management (Top-down)

Der Top-down-Ansatz beginnt gewöhnlich auf eine von zwei Arten. Ein Manager oder ein Entwickler initiiert den Prozess und der Rest der Organisation beginnt, sich ebenfalls Stück für Stück in diese Richtung zu bewegen. Oder jemand im mittleren Management sieht vielleicht eine Präsentation, liest ein Buch (wie dieses) oder spricht mit einem Kollegen über

die Vorteile von bestimmten Änderungen und die Art, wie sie funktionieren. Ein solcher Manager wird gewöhnlich den Prozess initiieren und einen Vortrag vor den Mitgliedern anderer Teams halten oder schlicht über seine Autorität den Wandel geschehen lassen.

9.2.3 Holen Sie einen externen Champion

Ich empfehle dringend, einen Außenstehenden hinzuzuziehen, um den Wandel zu unterstützen. Ein externer Berater, der hereinkommt, um beim Unit Testing und den damit zusammenhängenden Fragestellungen zu helfen, hat gegenüber firmeninternen Mitarbeitern Vorteile:

- *Redefreiheit* – Ein Berater kann Dinge sagen, die man innerhalb der Firma nicht von jemandem hören möchte, der dort arbeitet: »Die Code-Integrität ist schlecht«, »Die Tests sind unleserlich« usw.
- *Erfahrung* – Ein Berater hat mehr Erfahrung im Umgang mit firmeninternen Widerständen, er hat gute Antworten zu schwierigen Fragen auf Lager und er weiß, auf welche Knöpfe er drücken muss, um die Dinge ans Laufen zu bringen.
- *Fest zugeordnete Zeit* – Für einen Berater ist das sein Job. Im Gegensatz zu den anderen Angestellten der Firma, die bessere Dinge zu tun haben, als auf Änderungen zu drängen (wie etwa das Entwickeln von Software), ist es für den Berater ein Fulltime-Job und er widmet sich diesem Zweck ganz.

Code-Integrität

Code-Integrität ist eine Bezeichnung, die ich verwende, um den Zweck hinter den Entwicklungsaktivitäten eines Teams zu beschreiben, und zwar in puncto Code-Stabilität, Wartbarkeit und Feedback. Hauptsächlich bedeutet das, dass der Code das tut, was von ihm erwartet wird, und das Team weiß, wann er es nicht tut.

Diese Verfahren sind alle Teil der Code-Integrität:

- Automatische Builds
- Kontinuierliche Integration
- Unit Testing und testgetriebene Entwicklung
- Code-Konsistenz und vereinbarte Qualitätsstandards
- Die Möglichkeit, in kürzester Zeit Bugs zu fixen (oder fehlschlagende Tests ans Laufen zu bringen)

Manche betrachten dies als »Werte« der Entwicklung und Sie können sie in einer Methodologie wie dem Extreme Programming finden, aber ich sage gerne: »Wir haben eine gute Code-Integrität«, statt zu sagen, dass wir all diese Dinge gut machen.

Ich habe schon häufig einen Wandel zusammenbrechen sehen, weil ein überarbeiteter Champion nicht die Zeit hatte, sich dem Prozess voll und ganz zu widmen.

9.2.4 Machen Sie Fortschritte sichtbar

Es ist wichtig, den Fortschritt aufrecht und den Zustand des Wandels sichtbar zu halten. Hängen Sie Whiteboards oder Poster an die Wände in den Fluren oder in Pausenecken, wo

die Leute sich zusammenfinden. Die dargestellten Daten sollten in Beziehung zu den angestrebten Zielen stehen.

Beispielsweise können Sie die Anzahl der erfolgreichen und fehlgeschlagenen Tests des letzten nächtlichen Builds darstellen. Stellen Sie grafisch dar, welche Teams bereits einen automatisierten Build-Prozess haben. Hängen Sie ein Scrum-Burndown-Chart zum Iterationsfortschritt oder einen Report zur Testcode-Abdeckung (siehe Abbildung 9.1) auf, wenn dies Ihren Zielen entspricht. (Auf `www.controlchaos.com` können Sie mehr zum Scrum erfahren.) Schlagen Sie Kontaktinfos zu sich und den Champions an, damit jemand aufkommende Fragen beantworten kann. Stellen Sie einen großen LCD-Bildschirm auf, der in plakativen Grafiken laufend den Status der Builds anzeigt, was aktuell funktioniert und was fehlschlägt. Stellen Sie ihn an einem gut sichtbaren Platz auf, wo ihn alle Entwickler sehen können, etwa auf einem stark frequentierten Korridor oder an der Wand des Team-Besprechungsraums.

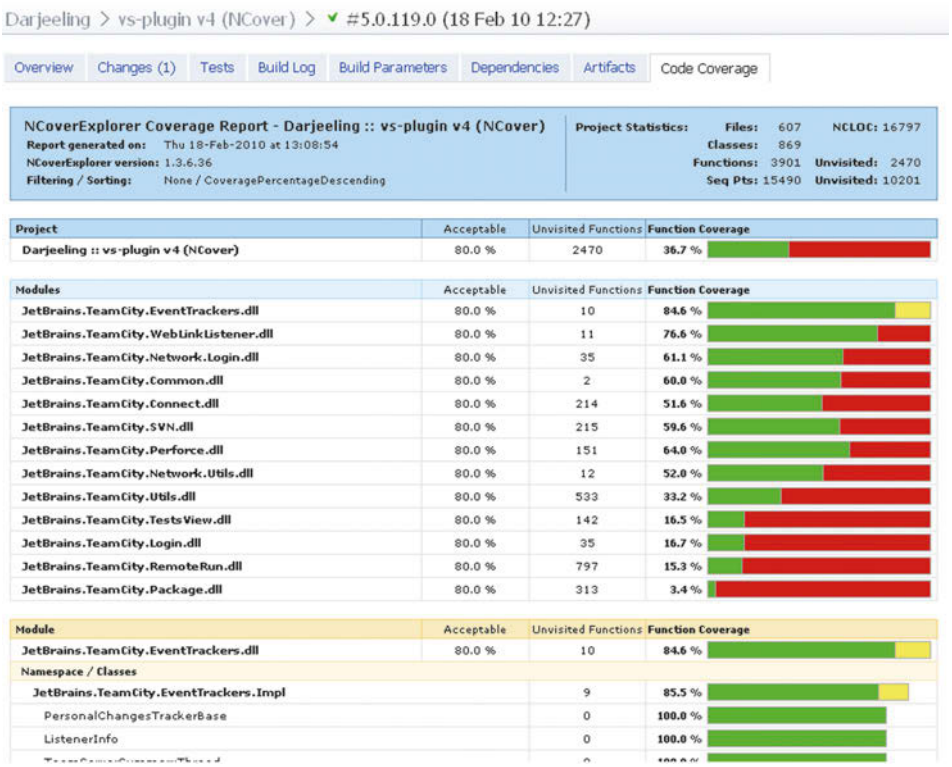


Abb. 9.1: Ein Beispiel für einen Report zur Testcode-Abdeckung in TeamCity mit NCover

Sie zielen darauf ab, über diese Charts mit zwei Gruppen ins Gespräch zu kommen:

- *Die Gruppe, die den Wandel durchlebt* – Die Leute in dieser Gruppe werden ein stärkeres Gefühl der Leistung und des Stolzes bekommen, wenn die Charts (die für alle zugänglich sind) regelmäßig aktualisiert werden, und sie fühlen sich eher verpflichtet, den Prozess fortzuführen, weil das für alle sichtbar ist. Sie können dann verfolgen, wie sie im

Verhältnis zu anderen Gruppen abschneiden. Möglicherweise strengen sie sich mehr an, wenn sie wissen, dass eine andere Gruppe bestimmte Verfahren schneller umgesetzt hat.

- *Diejenigen in der Organisation, die nicht Teil des Prozesses sind* – Sie steigern das Interesse und die Neugier unter diesen Leuten, Sie lösen Gespräche und Klatsch aus und Sie erschaffen eine Strömung, an der sie teilnehmen können, wenn sie das möchten.

9.2.5 Streben Sie bestimmte Ziele an

Ohne Ziele wird es schwierig sein, den Wandel zu messen und anderen mitzuteilen. Es wird ein vages »Etwas« sein, das leicht beim ersten Anzeichen von Ärger beendet werden kann.

Hier sind einige Ziele, die Sie in Betracht ziehen sollten:

- *Verbessern Sie die Testcode-Abdeckung parallel zu Code-Reviews und Test-Reviews*

  Eine Studie von Boris Beizer zeigt, dass Entwickler, die Tests schreiben, aber keine Tools oder andere Techniken zum Testen der Code-Abdeckung einsetzen, naiv überoptimistisch sind, was die Abdeckung angeht, die sie mit ihren Tests erreichen. Eine andere Studie aus dem Buch von Karl Wiegers *Peer Reviews in Software: A Practical Guide* (Addison-Wesley, 2001) deutet darauf hin, dass das Testen ohne Code-Abdeckungs-Tools nur zu einer Abdeckung von 50 bis 60 Prozent des Codes führt. (Einiges an anekdotischer Evidenz spricht dafür, dass mit TDD eine Abdeckung von 95 bis 100 Prozent des logischen Codes erreicht werden kann.)

  Ein einfach zu messendes Ziel ist der Anteil des Codes, der durch Tests abgedeckt wird. Je mehr Abdeckung, desto besser sind die Chancen, Bugs zu finden. Aber das ist trotzdem kein Patentrezept. Sie können leicht eine nahezu 100-prozentige Code-Abdeckung mit schlechten Tests erreichen, die keine Aussagekraft haben. Niedrige Abdeckung ist ein schlechtes Zeichen; hohe Abdeckung ist ein mögliches Zeichen, dass die Dinge besser laufen.

  Was Sie wirklich erreichen wollen, ist eine hohe Code-Abdeckung zusammen mit kontinuierlichen Code-Reviews und Test-Reviews (wie ich zuvor in diesem Kapitel erläutert habe). Auf diese Weise stellen Sie sicher, dass die Tests nicht nur geschrieben werden, um die Abdeckungs-Anforderungen zu erfüllen (z. B. ohne Asserts), sondern dass sie auch tatsächlich aussagekräftig sind.

Anmerkung

Die Studie von Boris Beizer wird im Artikel »Dr. Boris Beizer on Software Testing: An Interview, Part 1« von Mark Johnson in *The Software QA Quarterly* (Sommer 1994) diskutiert. Die andere Studie wird in *Peer Reviews in Software: A Practical Guide* diskutiert.

- *Verbessern Sie die Testcode-Abdeckung relativ zum Code Churn (Code-Änderungsrate).*

  Einige Produktionssysteme erlauben es Ihnen, den Code Churn zu messen – also wie viele Zeilen des Codes zwischen den einzelnen Builds geändert wurden. Je weniger Code-Zeilen sich geändert haben, desto weniger Fehler haben Sie wahrscheinlich neu in das System eingebaut. Diese Berechnung ist nicht immer praktikabel, insbesondere in Systemen, wo ein großer Teil des Codes automatisch als Teil des Build-Prozesses

erzeugt wird. Aber dieses Problem kann gelöst werden, indem der generierte Code ignoriert wird. Ein System, das es Ihnen erlaubt, den Code Churn[1] zu messen, ist Microsofts Team System (siehe hierzu Microsofts Artikel »Analyze and Report on Code Churn and Code Coverage Using the Code Churn and Run Coverage Perspectives « auf `http://msdn.microsoft.com/en-us/library/ms244661.aspx`).

- *Reduzieren Sie das Wiedereröffnen von Bugs.*

  Es ist einfach, eine Sache zu reparieren und versehentlich etwas anderes kaputt zu machen. Wenn das nicht häufig passiert, ist das ein Zeichen dafür, dass Sie in der Lage sind, die Dinge schnell zu reparieren und das System zu warten, ohne vorherige Annahmen zu ruinieren.

- *Reduzieren Sie die durchschnittliche Bugfixing-Zeit (die Zeit vom Öffnen des Bugs bis zum Schließen des Bugs).*

  Ein System mit guten Tests und einer guten Abdeckung gestattet es Ihnen meistens, Dinge schneller zu reparieren (vorausgesetzt, die Tests sind auf eine wartbare Weise geschrieben). Das wiederum bedeutet kürzere Durchlaufzeiten und die Release-Zyklen sind weniger aufreibend.

In seinem Buch *Code Complete* (Microsoft Press, 2. Auflage, 2004) erläutert Steve McConnell mehrere Kriterien, die Sie verwenden können, um den Fortschritt zu testen. Dazu gehören unter anderem die folgenden:

- Die Anzahl der Defekte, die pro Klasse gefunden wird, nach Priorität
- Die Anzahl der Defekte pro Routine und die Anzahl von Teststunden pro gefundenem Bug
- Die durchschnittliche Anzahl von Defekten pro Testfall

Ich lege Ihnen sehr ans Herz, Kapitel 22 in McConnells Buch zu lesen. Es handelt vom Testen durch die Entwickler.

9.2.6 Machen Sie sich klar, dass es Hürden geben wird

Es gibt immer Hürden. Einige entstehen durch die Organisationsstruktur, andere werden technischer Natur sein. Die technischen sind leichter zu nehmen, denn hier kommt es nur darauf an, die richtige Lösung zu finden. Die organisatorischen brauchen Umsicht und Aufmerksamkeit und einen psychologischen Ansatz.

Es ist wichtig, sich nicht einem Gefühl des zwischenzeitlichen Misserfolges hinzugeben, wenn einmal eine Iteration schlecht läuft, Tests langsamer als erwartet vorankommen usw. Manchmal ist es schwierig, am Ball zu bleiben. Sie müssen zumindest für ein paar Monate beharrlich bleiben, bevor Sie sich mit dem neuen Prozess wohlfühlen und alle Knoten ausgebügelt haben. Schwören Sie das Management darauf ein, mindestens drei Monate weiterzumachen, auch wenn die Dinge nicht wie geplant laufen. Es ist wichtig, deren Zustimmung im Vorfeld zu erhalten, denn Sie wollen nicht mitten in einem stressigen ersten Monat herumlaufen müssen, um die Leute zu überzeugen.

1 Anmerkung des Übersetzers: »Code Churn« bedeutet so viel wie »Code-Durchwirbelung«. Damit ist die Zahl der modifizierten, hinzugefügten und gelöschten Zeilen von einer Dateiversion zur nächsten gemeint.

Und verdauen Sie auch diese kurze Erkenntnis, die Tim Ottinger über Twitter (@Tottinge) mitteilte: »Wenn Ihre Tests nicht alle Fehler finden, so machen sie es doch leichter, die nicht gefundenen Fehler zu reparieren. Dies ist eine tiefgründige Wahrheit.«

Nachdem wir uns Wege angeschaut haben, um die Dinge richtig zu machen, betrachten wir jetzt ein paar Sachen, die zum Misserfolg führen können.

9.3 Wege zum Misserfolg

In der Einleitung zu diesem Buch erwähnte ich ein Projekt, an dem ich beteiligt war und das fehlschlug, zum Teil, weil das Unit Testing nicht korrekt implementiert wurde. Das ist eine Möglichkeit, wie Sie ein Projekt versieben können. Im Folgenden habe ich noch weitere mögliche Ursachen aufgelistet, ebenso wie die, die mich jenes Projekt gekostet haben, sowie einige Dinge, die dagegen unternommen werden können.

9.3.1 Mangelnde Triebkraft

An allen Stellen, wo ich den Wandel habe scheitern sehen, war mangelnde Antriebskraft der wichtigste Faktor im Spiel. Es hat seinen Preis, dauernd der Motor des Wandels zu sein. Es nimmt Zeit von Ihrem normalen Job weg, andere anzuleiten, ihnen zu helfen und interne politische Kämpfe um den Wandel zu führen. Sie müssen bereit sein, die Zeit für diese Aufgaben aufzubringen, oder der Wandel wird nicht geschehen. Die Hereinnahme einer externen Person, wie in Abschnitt 9.2.3 erwähnt, hilft Ihnen bei Ihrer Suche nach einer andauernden Antriebskraft.

9.3.2 Mangelnde politische Unterstützung

Wenn Sie Ihr Chef ausdrücklich anweist, keine Änderung durchzuführen, gibt es nicht viel, was Sie dagegen tun können, außer zu versuchen, das Management zu überzeugen, das zu sehen, was Sie sehen. Aber manchmal ist der Mangel an Unterstützung viel subtiler als das und der Trick besteht darin, zu bemerken, dass Sie Widerständen gegenüberstehen.

Beispielsweise wird Ihnen gesagt: »Sicherlich, legen Sie los und implementieren Sie diese Tests. Wir geben Ihnen 10% Ihrer Zeit, um das zu machen.« Alles unter 30% ist nicht realistisch für den Anfang einer Unit-Testing-Anstrengung. Das ist eine Art, wie ein Manager versuchen kann, eine Entwicklung zu stoppen – indem er sie bis zum Verschwinden drosselt.

Als Erstes müssen Sie erkennen, dass Sie einem Widerstand gegenüberstehen, aber sobald Sie das tun, ist er einfach zu identifizieren. Wenn Sie ihnen sagen, dass ihre Einschränkungen nicht realistisch sind, wird Ihnen gesagt: »Dann lassen Sie es bleiben.«

9.3.3 Schlechte Implementierungen und erste Eindrücke

Wenn Sie planen, das Unit Testing ohne Vorwissen darüber, wie »gute« Unit Tests geschrieben werden, einzuführen, dann sollten Sie sich selber einen großen Gefallen tun: Binden Sie jemanden mit Erfahrung ein und folgen Sie bewährten Vorgehensweisen (wie den in diesem Buch beschriebenen).

Ich habe Entwickler ins kalte Wasser springen sehen, ohne dass sie ein ausreichendes Verständnis davon hatten, was zu tun ist oder wo man anfangen sollte. Das ist kein idealer

Start. Es wird Sie nicht nur eine Menge Zeit kosten, zu lernen, wie man Änderungen vornimmt, die der Situation angemessen sind, sondern Sie werden unterwegs auch eine Menge an Glaubwürdigkeit verlieren, wenn Sie mit einer schlechten Implementierung beginnen. Das kann dazu führen, dass das Pilotprojekt einfach dichtgemacht wird.

Wenn Sie die Einleitung dieses Buches gelesen haben, dann werden Sie wissen, dass mir genau dies widerfahren ist. Sie haben nur ein paar Monate Zeit, um die Dinge in Gang zu setzen und die Vorgesetzten davon zu überzeugen, dass Sie Resultate erzielen. Nutzen Sie diese Zeit und vermeiden Sie, so weit möglich, Risiken. Wenn Sie nicht wissen, wie man gute Tests schreibt, dann lesen Sie ein Buch oder holen Sie einen Berater ins Boot. Wenn Sie nicht wissen, wie Sie Ihren Code testbar machen, dann tun Sie das Gleiche. Verschwenden Sie keine Zeit damit, Testverfahren erneut zu erfinden.

9.3.4 Mangelnde Teamunterstützung

Wenn Ihr Team Ihre Anstrengungen nicht unterstützt, dann wird der Erfolg mit Sicherheit ausbleiben, denn Sie werden eine Durststrecke zurücklegen müssen, um die zusätzliche Arbeit an dem neuen Prozess mit Ihren normalen Aufgaben in Einklang zu bringen. Sie sollten bestrebt sein, Ihr Team zu einem Teil des neuen Prozesses zu machen oder zumindest nicht zu einem Teil des Problems.

Reden Sie mit den Teammitgliedern über die Änderungen. Manchmal ist es ein guter Anfang, wenn Ihnen einer nach dem anderen seine Unterstützung gibt, aber es kann auch sinnvoll sein, mit ihnen als Gruppe über Ihre Anstrengungen zu reden und ihre schwierigen Fragen zu beantworten. Was immer Sie tun, gehen Sie nicht von einer selbstverständlichen Unterstützung durch das Team aus. Sorgen Sie dafür, dass Sie wissen, worauf Sie sich einlassen; das sind diejenigen, mit denen Sie jeden Tag arbeiten müssen.

Egal, wie Sie vorgehen, Ihnen werden in jedem Fall schwierige Fragen zum Unit Testing gestellt werden. Die folgenden Fragen und Antworten werden Ihnen helfen, sich auf die Diskussionen mit denjenigen vorzubereiten, die Ihre Agenda des Wandels unterstützen oder verhindern können.

9.4 Einflussfaktoren

Zu den Dingen, die ich noch interessanter finde als Unit Tests, gehört, wie sich Menschen verhalten und warum sie sich so verhalten. Es kann sehr frustrierend sein, jemanden dazu zu bringen, mit etwas anzufangen (beispielsweise TDD), aber egal, wie sehr Sie sich auch anstrengen, er macht es einfach nicht. Sie haben vielleicht schon mit ihm diskutiert, aber trotz Ihres Gespräches geschieht nichts.

Ein großartiges Buch zum Thema Einflussnahme ist *Influence: The Power to Change Anything* von Kerry Patterson, Joseph Grenny, David Maxfield, Ron McMillan und Al Switzler (McGraw-Hill, 2007). Sie finden hier einen Link darauf: `http://5whys.com/recommended-books/`. Das Mantra des Buches ist ein tiefgründiges: Egal welches Verhalten Sie auch sehen – die Welt ist dafür gemacht, dass sich irgendjemand auch so verhält. Das bedeutet, dass es andere Faktoren gibt, die das Verhalten beeinflussen, neben der Person, die etwas möchte oder die in der Lage ist, etwas zu tun. Allerdings schauen wir selten über diese zwei Faktoren hinaus.

Das Buch zeigt uns sechs Einflussfaktoren auf:

Persönliche Fähigkeit	Hat die Person all die Fähigkeiten und Kenntnisse, um durchzuführen, was erforderlich ist?
Persönliche Motivation	Zieht die Person Befriedigung aus dem richtigen Verhalten oder lehnt sie das falsche Verhalten ab? Hat sie die Selbstkontrolle, sich für das Verhalten zu engagieren, wenn es am schwierigsten ist?
Soziale Fähigkeit	Stellen Sie oder andere die Hilfe, Informationen und Ressourcen zur Verfügung, die diese Person insbesondere zu kritischen Zeiten benötigt?
Soziale Motivation	Unterstützen die anderen in der Umgebung aktiv das richtige Verhalten und lehnen das falsche ab? Führen Sie oder andere das richtige Verhalten auf effektive Weise vor?
Strukturelle (Umwelt-)Fähigkeit	Gibt es Aspekte in der Umgebung (Gebäude, Budget etc.), die das Verhalten bequem, einfach und sicher machen? Gibt es genug Hinweise und Mahnungen, um den Kurs zu halten?
Strukturelle Motivation	Gibt es klare und bedeutsame Belohnungen (wie Bezahlung, Boni oder Anreize), wenn Sie oder andere sich auf die richtige oder falsche Weise verhalten? Passen die kurzfristigen Belohnungen zu den beabsichtigten langfristigen Resultaten und Verhaltensweisen, die Sie stärken oder vermeiden wollen?

Betrachten Sie dies als kurze Checkliste, um zu verstehen, warum die Dinge nicht so laufen, wie Sie möchten. Und dann bedenken Sie eine wichtige andere Tatsache: Es könnte mehr als einen Faktor im Spiel sein. Um das Verhalten zu ändern, sollten Sie alle Faktoren im Spiel ändern. Wenn Sie nur einen ändern, wird das nicht das Verhalten ändern.

Hier kommt ein Beispiel für eine imaginäre Checkliste, die ich mal angelegt habe, weil jemand TDD nicht anwenden wollte. (Vergessen Sie nicht, dass sich dies für jede Person in jeder Organisation unterscheidet.)

Persönliche Fähigkeit	Hat die Person all die Fähigkeiten und Kenntnisse, um durchzuführen, was erforderlich ist?	Ja. Sie belegte einen dreitägigen TDD-Kurs mit Roy Osherove.
Persönliche Motivation	Zieht die Person Befriedigung aus dem richtigen Verhalten oder lehnt sie das falsche Verhalten ab? Hat sie die Selbstkontrolle, sich für das Verhalten zu engagieren, wenn es am schwierigsten ist?	Ich sprach mit ihr und sie möchte TDD einsetzen.
Soziale Fähigkeit	Stellen Sie oder andere die Hilfe, Informationen und Ressourcen zur Verfügung, die diese Person insbesondere zu kritischen Zeiten benötigt?	Ja.

Soziale Motivation	Unterstützen die anderen in der Umgebung aktiv das richtige Verhalten und lehnen das falsche ab? Führen Sie oder andere das richtige Verhalten auf effektive Weise vor?	So viel wie möglich.
Strukturelle (Umwelt-)Fähigkeit	Gibt es Aspekte in der Umgebung (Gebäude, Budget etc.), die das Verhalten bequem, einfach und sicher machen? Gibt es genug Hinweise und Mahnungen, um den Kurs zu halten?	*Sie haben kein Budget für eine Build-Maschine.
Strukturelle Motivation	Gibt es klare und bedeutsame Belohnungen (wie Bezahlung, Boni oder Anreize), wenn Sie oder andere sich auf die richtige oder falsche Weise verhalten? Passen die kurzfristigen Belohnungen zu den beabsichtigten langfristigen Resultaten und Verhaltensweisen, die Sie stärken oder vermeiden wollen?	*Wenn sie versuchen, Zeit für das Unit Testing aufzuwenden, sagen ihre Manager, dass sie Zeit verschwenden. Wenn sie früh und fehlerhaft ausliefern, bekommen sie einen Bonus.

Ich habe an die Dinge in der rechten Spalte Sternchen gesetzt, die verbessert werden müssen. Hier habe ich zwei Probleme identifiziert, die gelöst werden müssen. Nur das Problem mit dem Budget der Build-Maschine zu lösen, wird nicht reichen, um das Verhalten zu ändern. Sie brauchen eine Build-Maschine *und* müssen ihre Manager davon abhalten, ihnen für das schnelle Ausliefern von fehlerhaftem Zeug einen Bonus zu geben.

Ich schreibe viel mehr dazu in *Notes to a Software Team Leader*, einem Buch über die Führung eines technischen Teams. Sie finden es unter `5whys.com`.

9.5 Schwierige Fragen und Antworten

Dieser Abschnitt beschäftigt sich mit einigen Fragen, denen ich an verschiedenen Stellen begegnet bin. Sie entstehen gewöhnlich aus der Annahme, dass die Einführung des Unit Testings jemandem persönlich wehtun kann – einem Manager, der sich um seine Deadlines sorgt, oder einem QS-Mitarbeiter, der um seine Bedeutung fürchtet. Sobald Sie verstanden haben, woher eine Frage stammt, ist es wichtig, das Thema aufzugreifen, sei es direkt oder indirekt. Andernfalls wird es immer einen subtilen Widerstand geben.

9.5.1 Wie viel zusätzliche Zeit wird der aktuelle Prozess für das Unit Testing benötigen?

Teamleiter, Projektmanager und Kunden sind gewöhnlich diejenigen, die fragen, wie viel zusätzliche Zeit der Prozess wegen des Unit Testings benötigt. Das sind diejenigen, die in Bezug auf die Zeitabschätzung an der Front stehen.

Lassen Sie uns mit ein paar Fakten beginnen. Studien haben gezeigt, dass eine Verbesserung der gesamten Code-Qualität innerhalb eines Projekts die Produktivität erhöhen und

die Zeitpläne verkürzen kann. Wie passt das zu der Tatsache, dass das Schreiben von Tests das Schreiben von Code langsamer macht? Hauptsächlich durch die Wartbarkeit und die Einfachheit, mit der Bugs behoben werden können.

Anmerkung

Für Studien zur Code-Qualität und Produktivität beachten Sie bitte , *Programming Productivity* (McGraw-Hill College, 1986) und *Software Assessments, Benchmarks, and Best Practices* (Addison-Wesley, 2000). Beide sind von Casper Jones.

Wenn sie nach der Zeit fragen, dann fragen Teamleiter vielleicht tatsächlich: »Was soll ich meinem Projektleiter sagen, wenn wir unseren Abgabetermin überschreiten?« Sie mögen wirklich von der Nützlichkeit des Prozesses überzeugt sein, aber sie brauchen Munition für das bevorstehende Gefecht. Möglicherweise stellen sie die Frage auch nicht in Bezug auf das komplette Produkt, sondern in Bezug auf bestimmte Features oder die Funktionalität.

Auf der anderen Seite redet ein Projektmanager oder ein Kunde, der nach Terminen fragt, meist über komplette Produkt-Releases.

Weil sich die verschiedenen Leute Sorgen über verschiedene Bereiche machen, können die Antworten, die Sie ihnen geben, variieren. Beispielsweise kann das Unit Testing die Zeit, die für die Implementierung eines bestimmten Features benötigt wird, verdoppeln, aber das Release-Datum des ganzen Produkts kann möglicherweise vorverlegt werden. Um das zu verstehen, werfen wir einen Blick auf ein reales Beispiel, mit dem ich zu tun hatte.

Eine Geschichte von zwei Funktionalitäten

Eine große Firma, die ich beraten habe, wollte das Unit Testing in ihren Prozess integrieren und mit einem Pilotprojekt beginnen. Dieses bestand darin, dass eine Gruppe von Entwicklern einer großen, existierenden Anwendung eine neue Funktionalität hinzufügen sollte. Die Firma verdiente ihren Lebensunterhalt hauptsächlich mit der Entwicklung dieser großen Abrechnungssoftware und der Anpassung der Teile für verschiedene Kunden. Die Firma hat Tausende von Entwicklern überall in der Welt.

Die folgenden Kennziffern wurden ermittelt, um den Erfolg des Piloten zu testen:

- Die Zeit, die das Team für jeden Entwicklungsschritt benötigte
- Die Gesamtzeit bis zur Übergabe an den Kunden
- Die Zahl der Bugs, die der Kunde nach der Übergabe gefunden hat

Die gleiche Statistik wurde für eine ähnliche Funktionalität ermittelt, die von einem anderen Team für einen anderen Kunden eingebaut wurde. Die beiden Funktionalitäten waren nahezu gleich umfangreich und die beiden Teams waren ungefähr auf dem gleichen Level an Fähigkeiten und Erfahrung. Beide Aufgaben waren Kundenanpassungen – eine mit Unit Tests und die andere ohne. Tabelle 9.1 zeigt den zeitlichen Unterschied.

Phase	Team ohne Tests	Team mit Tests
Implementierung (Codierung)	7 Tage	14 Tage
Integration	7 Tage	2 Tage

Tabelle 9.1: Teamfortschritt und Arbeitsleistung mit und ohne Tests

Phase	Team ohne Tests	Team mit Tests
Testen und Bugfixing	Testen, 3 Tage Fixing, 3 Tage Testen, 3 Tage Fixing, 2 Tage Testen, 1 Tag Gesamt: 12 Tage	Testen, 3 Tage Fixing, 1 Tag Testen, 1 Tag Fixing, 1 Tag Testen, 1 Tag Gesamt: 8 Tage
Gesamte Release-Zeit	26 Tage	24 Tage
Anzahl der Bugs, die im Produktionsbetrieb gefunden wurden	71	11

Tabelle 9.1: Teamfortschritt und Arbeitsleistung mit und ohne Tests (Forts.)

Insgesamt war die Zeit bis zur Übergabe mit Tests kürzer als ohne Tests. Trotzdem haben die Manager des Teams mit den Tests zunächst nicht geglaubt, dass der Pilot ein Erfolg sei, denn sie schauten statt auf das Gesamtergebnis nur auf die Implementierungs(Codierungs)-Statistik (die erste Zeile in Tabelle 9.1) als Erfolgskriterium. Es dauerte doppelt so lange, die Funktionalität zu codieren (denn Unit Tests erfordern das Schreiben von mehr Code). Trotzdem wurde die extra benötigte Zeit dadurch mehr als ausgeglichen, dass das QS-Team weniger Bugs fand, auf die reagiert werden musste.

Deshalb ist es so wichtig, zu betonen, dass Unit Tests zwar den Zeitbedarf für die Implementierung einer Funktionalität erhöhen können, der übergeordnete Zeitbedarf aber ausgeglichen wird, wenn man den ganzen Produktzyklus betrachtet, da Qualität und Wartbarkeit verbessert werden.

9.5.2 Ist mein Job bei der QS in Gefahr wegen des Unit Testing?

Unit Testing macht die Jobs in der QS nicht überflüssig. Die QS-Mitarbeiter erhalten die Anwendung mit einem kompletten Unit-Test-Paket, was heißt, dass sie sicherstellen können, dass alle Unit Tests erfolgreich sind, bevor sie mit ihrem eigenen Testprozess beginnen. Das Vorhandensein der Unit Tests macht ihren Job eigentlich interessanter. Statt ein UI-Debugging durchzuführen (wo jeder zweite Buttonklick zu irgendeiner Art von Ausnahme führt), sind sie in der Lage, sich auf das Finden von eher logischen Fehlern in Real-World-Szenarien zu konzentrieren. Unit Tests bilden die erste Verteidigungsschicht gegen Bugs und die Arbeit der QS bildet die zweite Schicht – die Schicht der Benutzerakzeptanz. So wie eine Anwendung im Hinblick auf die Sicherheit auch immer mehr als nur eine Schicht des Schutzes benötigt. Wenn der QS erlaubt wird, sich auf die größeren Themen zu konzentrieren, kann das bessere Anwendungen ermöglichen.

Manchmal schreiben QS-Mitarbeiter Code und können beim Schreiben von Unit Tests für die Anwendung helfen. Das geschieht im Zusammenspiel mit den Anwendungsentwicklern und nicht stattdessen. Sowohl Entwickler als auch QS-Mitarbeiter können Unit Tests schreiben.

9.5.3 Woher wissen wir, dass Unit Tests wirklich funktionieren?

Um festzustellen, ob Ihr Unit Testing funktioniert, legen Sie eine Art von Metrik an, wie in Abschnitt 9.2.5 vorgestellt. Wenn Sie es messen können, haben Sie eine Möglichkeit, es zu wissen, und Sie werden es spüren.

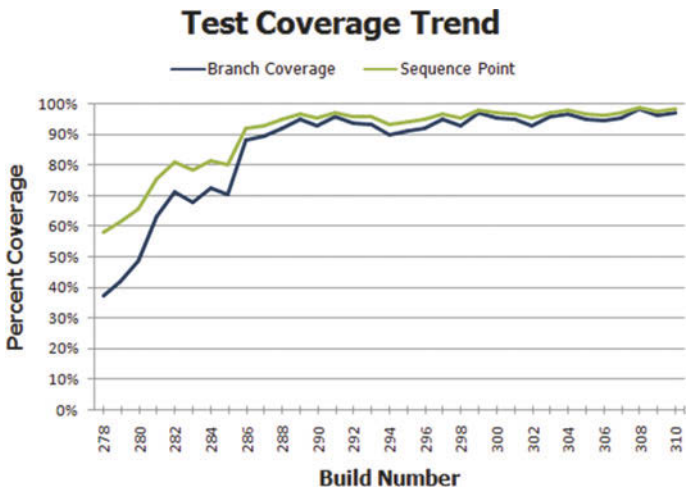


Abb. 9.2: Ein Beispiel eines Test-Code-Coverage-Trend-Reports

Abbildung 9.2 zeigt einen beispielhaften Test-Code-Coverage-Report (Code-Abdeckung pro Build). Indem Sie ein Tool wie NCover for .NET in den automatischen Build-Prozess einbinden und einen Report wie diesen erzeugen lassen, können Sie den Fortschritt in einem Aspekt der Entwicklung aufzeigen.

Die Code-Abdeckung ist ein guter Ausgangspunkt, wenn Sie sich fragen, ob Unit Tests fehlen.

9.5.4 Gibt es denn einen Beweis, dass Unit Testing hilft?

Es gibt keine speziellen Studien zum Thema, ob Unit Tests dabei helfen, eine bessere Qualität des Codes zu erreichen, auf die ich verweisen könnte. Die meisten derartigen Studien beschäftigen sich mit der Übernahme bestimmter agiler Methoden, wobei das Unit Testing nur eine davon ist. Im Web können einige empirische Hinweise gesammelt werden, ebenso bei Firmen und Kollegen, die großartige Resultate erzielen und nicht mehr zu einer Code-Basis ohne Tests zurückkehren wollen.

Ein paar Studien zu TDD finden Sie unter `http://biblio.gdinwiddie.com/biblio/StudiesOfTestDrivenDevelopment`.

9.5.5 Warum findet die QS immer noch Bugs?

Es ist der Job eines QS-Mitarbeiters, Bugs auf vielen unterschiedlichen Ebenen zu finden und die Anwendung mit vielen unterschiedlichen Ansätzen in Angriff zu nehmen. Gewöhnlich führt ein QS-Mitarbeiter Tests im Integrationsstil durch, wodurch er Probleme finden kann, die er mit Unit Tests nicht entdeckt. Beispielsweise kann die Art, wie verschiedene Komponenten im Betrieb zusammenarbeiten, auf Bugs hinweisen, auch wenn die einzelnen Komponenten die Unit Tests erfolgreich durchlaufen haben (und isoliert gut funktionieren). Zusätzlich kann ein QS-Mitarbeiter Dinge im Hinblick auf Use Cases oder komplette Szenarien testen, die Unit Tests gewöhnlich nicht abdecken. Dieser Ansatz kann logische oder akzeptanzbezogene Fehler abdecken und ist eine großartige Hilfe, um eine bessere Projektqualität zu gewährleisten.

Eine Untersuchung von Glenford Myre zeigt, dass Entwickler, die Tests schreiben, nicht wirklich nach Fehlern im Code suchen und deshalb nur die Hälfte bis zwei Drittel der Bugs in einer Applikation finden. Grob ausgedrückt bedeutet das, dass es immer Aufgaben für QS-Mitarbeiter geben wird, komme da, was wolle. Obwohl die Studie mehr als 34 Jahre alt ist, glaube ich, dass sich diese Mentalität nicht geändert hat und das Resultat somit auch heute noch von Bedeutung ist, zumindest für mich.

Anmerkung

Die Studie von Glenford Myre wird in »A controlled experiment in program testing and code walkthroughs/inspections« in *Communications of the ACM* 21, Nr. 9 (September 1978), 760–768, diskutiert.

9.5.6 Wir haben eine Menge Code ohne Tests: Wo fangen wir an?

Untersuchungen, die in den 1970er- und 1980er-Jahren durchgeführt wurden, zeigen, dass typischerweise 80% der Bugs in 20% des Codes gefunden werden. Der Trick besteht darin, den Code zu finden, der die meisten Probleme hat. In den meisten Fällen kann Ihnen jedes Team sagen, welche Komponenten die problematischsten sind. Beginnen Sie dort. Sie können immer eine Metrik, wie in Abschnitt 9.2.5 erläutert, hinzufügen, die sich auf die Bugs pro Klasse bezieht.

Anmerkung

Zu den Studien, die zeigen, dass 80% der Bugs in 20% des Codes zu finden sind, gehören die folgenden: Albert Endres, »An analysis of errors and their causes in system programs«, *IEEE Transactions on Software Engineering* 2 (Juni 1975), 140–149; Lee L. Gremillion, »Determinants of program repair maintenance requirements«, *Communications of the ACM* 27, Nr.8 (August 1984), 826–832; Barry W. Boehm, »Industrial software metrics top 10 list«, *IEEE Software* 4, Nr. 9 (September 1987), 84–85; Shull et al., »What we have learned about fighting defects«, *Proceedings of the 8th International Symposium on Software Metrics* (2002), 249–258.

Das Testen von Legacy-Code erfordert eine andere Herangehensweise als das Schreiben von neuem Code mit Tests. Siehe hierzu Kapitel 10 für mehr Details.

9.5.7 Wir arbeiten mit mehreren Sprachen: Ist Unit Testing da praktikabel?

Manchmal können Tests auf einen Code angewendet werden, der in einer anderen Sprache geschrieben ist, insbesondere wenn es sich um einen Mix von .NET-Sprachen handelt. Sie können beispielsweise Tests in C# schreiben, um in VB.NET geschriebenen Code zu testen. Manchmal schreibt jedes Team Tests in der Sprache, in der es auch entwickelt: C#-Entwickler können Tests in C# schreiben und dabei eines der vielen Frameworks verwenden (MSTest oder NUnit, um nur zwei Beispiele zu nennen) und C++-Entwickler können die Tests mithilfe eines der an C++ angelehnten Frameworks schreiben, etwa mit CppUnit. Ich habe auch schon Projekte gesehen, wo Leute den Code in C++ entwickelt, diesen in gemanagte C++-Wrapper gepackt und schließlich die Tests dafür in C# geschrieben haben, was das Schreiben und das Warten erleichtert hat.

9.5.8 Was ist, wenn wir eine Kombination aus Soft- und Hardware entwickeln?

Wenn Ihre Anwendung aus einer Kombination aus Software und Hardware besteht, dann müssen Sie Tests für die Software schreiben. Wahrscheinlich haben Sie bereits eine Art von Hardwaresimulator und die von Ihnen geschriebenen Tests können daraus Nutzen ziehen. Es mag ein bisschen aufwendiger sein, aber es ist definitiv möglich und Firmen tun dies die ganze Zeit.

9.5.9 Wie können wir wissen, dass wir keine Bugs in unseren Tests haben?

Sie müssen dafür sorgen, dass Ihre Tests fehlschlagen, wenn sie fehlschlagen sollen, und dass sie erfolgreich sind, wenn sie erfolgreich sein sollen. TDD ist eine großartige Möglichkeit, dass die Überprüfung dieser Dinge nicht vergessen wird. Beachten Sie Kapitel 1 für eine kurze Einführung in TDD.

9.5.10 Mein Debugger zeigt mir, dass mein Code funktioniert: Wozu brauche ich Tests?

Debugger helfen nicht viel bei Multithread-Code. Vielleicht sind Sie sicher, dass Ihr Code sauber funktioniert, aber was ist mit dem Code anderer? Woher wissen Sie, dass der funktioniert? Woher wissen die anderen, dass Ihr Code funktioniert und dass sie durch ihre Änderungen nichts kaputt gemacht haben? Bedenken Sie, dass die Codierung der erste Schritt im Leben des Codes ist. Den überwiegenden Teil seines Lebens wird der Code im Zustand der Wartung sein. Sie müssen mithilfe der Unit Tests dafür sorgen, dass er Bescheid sagt, wenn irgendetwas kaputt geht.

Eine Untersuchung von Curtis, Krasner und Iscoe hat gezeigt, dass die meisten Defekte nicht aus dem Code selber stammen, sondern dass sie das Resultat einer schlechten Kommunikation der Leute untereinander, von sich ändernden Anforderungen und einem Mangel an Wissen über die Applikationsdomäne sind. Auch wenn Sie der weltbeste Programmierer sind, besteht die Möglichkeit, dass Sie die falsche Sache codieren, weil es Ihnen jemand so gesagt hat. Und wenn Sie etwas ändern müssen, werden Sie froh sein, Tests für alles andere zu haben, damit Sie sicher sein können, dass Sie nichts beschädigen.

Anmerkung

Die Studie von Bill Curtis, H. Krasner und N. Iscoe ist »A field study of the software design process for large systems«, *Communications of the ACM* 31, Nr. 11 (November 1988), 1268–1287.

9.5.11 Müssen wir Code im TDD-Stil schreiben?

TDD ist eine Stilfrage. Ich persönlich sehe eine Reihe von Vorteilen in TDD und viele finden sie produktiv und nützlich, aber andere sind der Überzeugung, dass es für sie ausreichend ist, die Tests nach dem Code zu schreiben. Sie können Ihre eigene Wahl treffen.

Wenn diese Frage aus Furcht vor zu vielen gleichzeitigen Änderungen entsteht, dann kann der Lernprozess in mehrere Zwischenschritte aufgebrochen werden:

- Lernen Sie das Unit Testing aus Büchern wie diesem und verwenden Sie Tools wie Typemock Isolator oder JMockIt, damit Sie sich beim Testen keine Sorgen um Designaspekte machen müssen.
- Lernen Sie gute Designmethoden wie etwa SOLID (wird in Kapitel 11 vorgestellt).
- Lernen Sie, die testgetriebene Entwicklung anzuwenden. (Ein gutes Buch dazu ist *Test-Driven Development: By Example* von Kent Beck.)

Auf diese Weise wird das Lernen einfacher, Sie können schneller starten und verlieren weniger Zeit für Ihr Projekt.

9.6 Zusammenfassung

Die Einführung von Unit Tests in der Organisation ist etwas, mit dem sich viele Leser dieses Buches früher oder später beschäftigen müssen. Seien Sie vorbereitet. Sorgen Sie dafür, gute Antworten auf die Fragen zu haben, die Ihnen wahrscheinlich gestellt werden. Sorgen Sie dafür, dass Sie diejenigen, die Ihnen helfen können, nicht verschrecken. Seien Sie auf einen mühsamen Kampf vorbereitet. Verstehen Sie die Kräfte der Einflussnahme.

Im nächsten Kapitel werden wir uns Legacy-Code ansehen und einige Tools und Techniken untersuchen, die damit umgehen können.

Kapitel 10

Der Umgang mit Legacy-Code

Dieses Kapitel behandelt

- verbreitete Probleme im Zusammenhang mit Legacy-Code
- die Überlegung, wo man mit dem Schreiben von Tests beginnt
- eine Übersicht zu hilfreichen Tools, um mit Legacy-Code umzugehen

Ich habe mal einen großen Entwicklungs-Laden beraten, der Abrechnungssoftware herstellte. Sie hatten mehr als 10.000 Entwickler und setzten eine Mischung aus .NET, Java und C++ in den Produkten, Unterprodukten und den damit verknüpften Projekten ein. Die Software existierte in der einen oder anderen Form seit fünf Jahren und die meisten Entwickler waren mit der Wartung und der Weiterentwicklung der bestehenden Funktionalität beschäftigt.

Es war meine Aufgabe, verschiedenen Abteilungen (die alle Sprachen verwendeten) dabei zu helfen, die Methoden von TDD zu erlernen. Aber für etwa 90% der Entwickler, mit denen ich zusammenarbeitete, wurde dies aus unterschiedlichen Gründen niemals Realität. Einige der Ursachen waren im Legacy-Code begründet:

- Es war schwierig, Tests für den existierenden Code zu schreiben.
- Es war nahezu unmöglich, ein Refactoring für den existierenden Code durchzuführen (oder es stand nicht genug Zeit zur Verfügung).
- Einige wollten ihr Design nicht ändern.
- Werkzeuge (oder ein Mangel daran) standen im Weg.
- Es war schwierig, herauszufinden, wo begonnen werden sollte.

Jeder, der jemals versucht hat, einem existierenden System Tests hinzuzufügen, weiß, dass es nahezu unmöglich ist, Tests für solche Systeme zu schreiben. Gewöhnlich sind sie so geschrieben, dass keine vernünftigen Stellen (Seams) in der Software existieren, um dort Erweiterungen einfügen oder existierende Komponenten austauschen zu können.

Bei der Behandlung von Legacy-Code müssen mehrere Problembereiche angegangen werden:

- Es gibt so viel Arbeit, wo sollen Sie mit dem Hinzufügen von Tests anfangen? Worauf sollen Sie Ihre Anstrengungen konzentrieren?
- Wie können Sie Ihren Code sicher umbauen, wenn es keine Tests gibt, mit denen Sie anfangen können?
- Welche Tools können Sie für den Legacy-Code einsetzen?

Dieses Kapitel wird diese schwierigen Fragen im Zusammenhang mit der Behandlung einer Legacy-Code-Basis angehen, indem ich Methoden, Referenzen und Tools vorstelle, die dabei helfen können.

10.1 Wo soll man mit dem Einbauen der Tests beginnen?

Vorausgesetzt, Ihr existierender Code befindet sich in Komponenten, so müssen Sie eine Prioritätsliste der Komponenten erstellen, für die das Testen den meisten Sinn macht. Mehrere Faktoren, die die Priorität jeder einzelnen Komponente beeinflussen können, müssen berücksichtigt werden:

- *Logische Komplexität* – Dies bezieht sich auf den Umfang der Logik innerhalb der Komponente, wie etwa verschachtelte `if`-Anweisungen, `switch`-Anweisungen oder Rekursionen. Um dies zu ermitteln, können auch Tools zur Prüfung der zyklischen Komplexität eingesetzt werden.
- *Abhängigkeitsgrad* – Dies bezieht sich auf die Anzahl der Abhängigkeiten innerhalb der Komponente. Wie viele Abhängigkeiten müssen Sie aufbrechen, um diese Klasse testen zu können? Kommuniziert sie vielleicht mit einer externen E-Mail-Komponente oder ruft sie irgendwo eine statische Log-Methode auf?
- *Priorität* – Dies ist die allgemeine Priorität der Komponente innerhalb des Projekts.

Für diese Faktoren können Sie jeder Komponente eine Bewertung von 1 (niedrigste Priorität) bis 10 (höchste Priorität) geben.

Tabelle 10.1 zeigt Klassen mit Bewertungen für diese Faktoren. Ich nenne dies eine *Test-Realisierbarkeits-Tabelle*.

Komponente	Logische Komplexität	Abhängigkeitsgrad	Priorität	Anmerkungen
`Utils`	6	1	5	Diese Utility-Klasse enthält wenige Abhängigkeiten, aber eine Menge Logik. Sie ist einfach zu testen und hat große Bedeutung.
`Person`	2	1	1	Dies ist eine Datenhaltungsklasse mit wenig Logik und keinen Abhängigkeiten. Sie zu testen, hat einen geringen Wert.
`TextParser`	8	4	6	Diese Klasse enthält eine Menge Logik und eine Menge Abhängigkeiten. Um noch einen draufzusetzen, gehört sie zu einem sehr wichtigen Teilprojekt. Sie zu testen ist wichtig, wird aber schwierig und zeitaufwendig sein.
`ConfigManager`	1	6	1	Diese Klasse hält die Konfigurationsdaten und liest die Dateien von der Festplatte. Sie enthält wenig Logik, aber viele Abhängigkeiten. Sie zu testen, bringt wenig Nutzen für das Projekt und es wird auch schwierig und zeitaufwendig sein.

Tabelle 10.1: Eine einfache Test-Realisierbarkeits-Tabelle

Ausgehend von den Daten in Tabelle 10.1 können wir das Diagramm in Abbildung 10.1 anlegen, das Ihre Komponenten in Abhängigkeit vom Wert für das Projekt und der Anzahl der Abhängigkeiten darstellt.

Die Posten, die unterhalb unserer festgelegten Logik-Schwelle liegen (und die ich gewöhnlich auf 2 oder 3 lege), können Sie gefahrlos ignorieren, also fallen `Person` und `ConfigManager` weg. Somit bleiben nur die beiden oberen Komponenten aus Abbildung 10.1 übrig. Im Grunde gibt es zwei Möglichkeiten, das Diagramm zu betrachten und zu entscheiden, was man als Erstes testen möchte (siehe Abbildung 10.2):

- Wähle die komplexere und einfacher zu testende Komponente (oben links).
- Wähle die komplexere und schwieriger zu testende Komponente (oben rechts).

Die Frage ist nun, welchen Weg man einschlagen sollte. Soll man mit dem einfachen Zeug starten oder mit dem schwierigen?

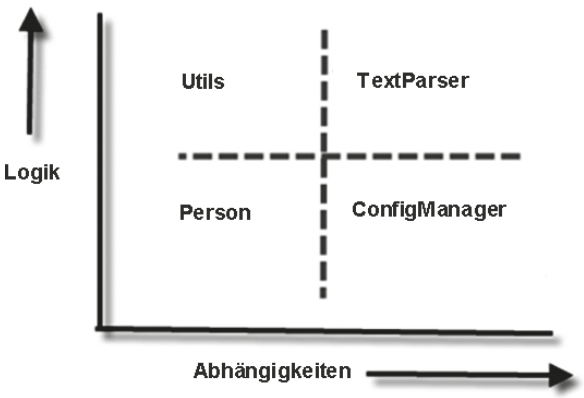


Abb. 10.1: Die Abbildung der Komponenten für die Test-Realisierbarkeit

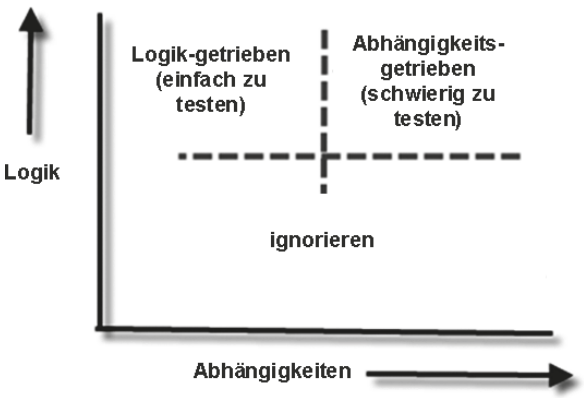


Abb. 10.2: Einfache, schwierige und unwichtige Komponentenabbildung in Bezug auf Logik und Abhängigkeiten

10.2 Bestimmen Sie eine Auswahlstrategie

Wie im vorangegangenen Abschnitt erläutert, können Sie entweder mit den Komponenten starten, die am einfachsten zu testen sind, oder mit denen, die am schwierigsten zu testen sind (weil sie viele Abhängigkeiten enthalten). Jede Strategie stellt andere Anforderungen.

10.2.1 Vor- und Nachteile der Strategie »Einfaches zuerst«

Wenn man mit den Komponenten beginnt, die weniger Abhängigkeiten aufweisen, dann ist das Schreiben der Tests zunächst wesentlich schneller und einfacher. Aber die Sache hat einen Haken, wie Abbildung 10.3 demonstriert.

Abbildung 10.3 zeigt, wie lange es dauert, um während der Laufzeit des Projekts Komponenten in die Tests einzubinden. Anfangs ist das Schreiben der Tests einfach, aber mit der Zeit bleiben Sie auf den Komponenten sitzen, die zunehmend schwieriger zu testen sind, und die besonders harten Nüsse warten am Ende des Projektzyklus auf Sie, wenn sowieso jeder genervt ist und das Produkt nur noch fertigstellen möchte.

Abb. 10.3: Wenn man mit den einfachen Komponenten beginnt, dann wird die pro Test benötigte Zeit länger und länger, bis die schwierigsten Komponenten erledigt sind.

Wenn für Ihr Team die Methoden des Unit Testings relativ neu sind, ist es sinnvoll, mit den einfachen Komponenten zu starten. Im Laufe der Zeit wird das Team die notwendigen Techniken erlernen und mit den komplexeren Komponenten und Abhängigkeiten umgehen können.

Für so ein Team ist es wahrscheinlich klug, zunächst alle Komponenten mit mehr als einer bestimmten Anzahl von Abhängigkeiten zu vermeiden (vier ist ein vernünftiges Limit).

10.2.2 Vor- und Nachteile der Strategie »Schwieriges zuerst«

Mit den schwierigeren Komponenten zu beginnen, mag nicht wie ein besonders kluges Vorhaben erscheinen, aber es hat einen Vorteil, wenn Ihr Team genügend Erfahrung im Umgang mit den Methoden des Unit Testings besitzt.

Abbildung 10.4 zeigt die durchschnittliche Zeit, die für das Schreiben eines Tests für eine einzelne Komponente benötigt wird, über die Laufzeit des Projekts, wenn Sie die Komponenten mit den meisten Abhängigkeiten als Erste testen.

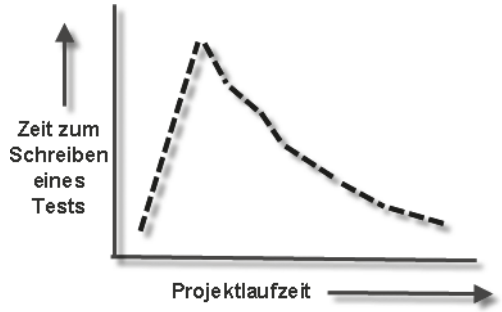


Abb. 10.4: Wenn Sie die Strategie »Schwieriges zuerst« verfolgen, dann ist die für die Tests der ersten Komponenten benötigte Zeit lang, wird aber immer kürzer, je mehr Abhängigkeiten durch das Refactoring entfernt werden.

Bei dieser Strategie können Sie gut und gerne mehr als einen Tag damit verbringen, auch nur die einfachsten Tests für die komplexeren Komponenten ans Laufen zu bringen. Aber beachten Sie die schnelle Abnahme in der zum Schreiben der Tests benötigten Zeit im Verhältnis zur langsamen Zunahme in Abbildung 10.3. Jedes Mal, wenn Sie eine Komponente in den Test einbinden und sie umbauen, um sie leichter testen zu können, dann lösen Sie vielleicht schon Testprobleme für die eine oder andere verknüpfte Komponente mit. Gerade weil diese Komponente eine Menge Abhängigkeiten besitzt, kann ihr Umbau die Dinge für andere Teile des Systems verbessern. Das ist der Grund für den steilen Abfall im Diagramm.

Die Strategie »Schwieriges zuerst« ist nur möglich, wenn Ihr Team Erfahrung im Umgang mit dem Unit Testing hat, denn sie ist schwieriger umzusetzen. Wenn Ihr Team die Erfahrung hat, dann nutzen Sie den Prioritätsaspekt der Komponenten, um zu entscheiden, ob Sie mit den schwierigen oder den einfachen Komponenten starten wollen. Vielleicht wollen Sie auch mit einer Mischung beginnen, aber es ist wichtig, dass Sie im Voraus wissen, welcher Aufwand auf Sie zukommen wird und was die möglichen Konsequenzen sind.

10.3 Schreiben Sie Integrationstests, bevor Sie mit dem Refactoring beginnen

Wenn Sie beabsichtigen, Ihren Code aus Gründen der Testbarkeit umzubauen (also damit Sie Unit Tests schreiben können), dann ist es ein praktikabler Weg, Tests im Integrationsstil für das Produktionssystem zu schreiben, damit Sie während der Refactoring-Phase nichts kaputt machen.

Als ich einmal ein großes Legacy-Projekt beraten habe, arbeitete ich mit einem Entwickler zusammen, der an einem XML-Konfigurationsmanager arbeiten musste. Das Projekt hatte keine Tests und war kaum testbar. Es war auch ein C++-Projekt, weshalb wir ein Tool wie Typemock Isolator nicht einsetzen konnten, um Komponenten zu isolieren, ohne den Code umzubauen.

Der Entwickler musste ein weiteres Werteattribut in die XML-Datei einbauen und in der Lage sein, es über die existierende Konfigurationskomponente zu lesen und zu ändern. Schließlich schrieben wir eine Reihe von Integrationstests, die das reale System verwendeten, um die Konfigurationsdaten zu speichern und zu laden. Wir bauten Asserts für die Werte ein, die die Konfigurationskomponente erhielt und in die Datei schrieb. Diese Tests

machten das ursprüngliche »Arbeitsverhalten« des Konfigurationsmanagers zur Basis unserer Arbeit.

Als Nächstes schrieben wir einen Integrationstest, der die Komponente überprüfte, sobald sie anfing, aus der Datei zu lesen, und der zeigte, dass sie keine Attribute mit einem Namen im Speicher hielt, den wir hinzuzufügen versuchten. Wir bewiesen damit, dass diese Fähigkeit fehlte, und hatten nun einen Test, der erfolgreich sein würde, sobald wir der XML-Datei das neue Attribut hinzufügen und aus der Komponente korrekt hineinschreiben würden.

Sobald wir den Code geschrieben hatten, der das extra Attribut laden und speichern konnte, ließen wir die drei Integrationstests laufen (zwei Tests für die ursprüngliche Basisimplementierung und ein neuer, der das neue Attribut zu lesen versuchte). Alle drei waren erfolgreich, womit wir wussten, dass wir beim Hinzufügen der neuen Funktionalität keine existierende beschädigt hatten.

Wie Sie sehen können, ist der Prozess relativ einfach:

- Fügen Sie dem System einen oder mehrere Integrationstests (keine Mocks oder Stubs) hinzu, um zu überprüfen, dass das ursprüngliche System wie erwartet funktioniert.
- Bauen Sie einen Test um oder fügen Sie einen hinzu, der für das Feature, das Sie dem System hinzufügen wollen, fehlschlägt.
- Führen Sie ein Refactoring durch und bauen Sie das System in kleinen Stücken um. Lassen Sie die Integrationstests so oft wie möglich laufen und überprüfen Sie, ob irgendetwas beschädigt wurde.

Manchmal mag es einfacher scheinen, Integrationstests zu schreiben statt Unit Tests, denn Sie müssen sich nicht mit DI (Abhängigkeitsinjektion) herumschlagen. Aber solche Tests auf Ihrem lokalen System ans Laufen zu bringen, mag sich als ärgerlich oder zeitaufwendig herausstellen, weil Sie dafür sorgen müssen, dass jede Kleinigkeit, die das System benötigt, am richtigen Platz ist.

Der Trick besteht darin, an den Teilen des Systems zu arbeiten, die Sie reparieren oder denen Sie Features hinzufügen müssen. Konzentrieren Sie sich nicht auf die anderen Teile. Auf diese Weise wächst das System an den richtigen Stellen und Sie müssen sich nicht um ungelegte Eier kümmern.

Während Sie mehr und mehr Tests hinzufügen, können Sie das System umbauen und mehr Unit Tests einbauen, wodurch es zu einem immer besser wartbaren und testbaren System wächst. Das dauert zwar seine Zeit (manchmal Monate und Monate), aber das ist es wert.

Habe ich erwähnt, dass Sie gute Werkzeuge benötigen? Lassen Sie uns einen Blick auf einige meiner Lieblings-Tools werfen.

10.4 Wichtige Tools für das Unit Testing von Legacy-Code

Hier kommen ein paar Tipps zu Tools, die Ihnen einen Vorsprung geben können, wenn Sie existierenden .NET-Code testen wollen:

- Abhängigkeiten isolieren Sie leicht mit JustMock oder Typemock Isolator.
- Verwenden Sie JMockit für Java-Legacy-Code.

- Verwenden Sie Vise beim Refactoring Ihres Java-Codes.
- Verwenden Sie FitNesse für Akzeptanztests, bevor Sie mit dem Refactoring beginnen.
- Lesen Sie das Buch von Michael Feathers zu Legacy-Code.
- Verwenden Sie NDepend, um Ihren Produktionscode zu untersuchen.
- Verwenden Sie ReSharper für die einfachere Navigation und das einfachere Refactoring Ihres Produktionscodes.
- Spüren Sie Code-Duplikate (und Bugs) mit Simian und TeamCity auf.

Lassen Sie uns das im Einzelnen durchgehen.

10.4.1 Abhängigkeiten isolieren Sie leicht mit uneingeschränkten Isolation-Frameworks

Uneingeschränkte Frameworks wie Typemock Isolator wurden in Kapitel 6 vorgestellt. Es ist ihre Fähigkeit, Abhängigkeiten im Produktionscode zu fälschen, ohne dass Sie irgendwelche Umbauten vornehmen müssen, die sie so einzigartig geeignet für diese Herausforderung machen. Das erspart Ihnen zu Beginn wertvolle Zeit, wenn Sie eine Komponente in die Tests aufnehmen wollen.

Anmerkung

Offenlegung: Während ich an der ersten Auflage dieses Buch schrieb, habe ich auch für Typemock an einem anderen Produkt gearbeitet. Ich habe ebenfalls am Design der API von Isolator 5.0 mitgearbeitet. Im Dezember 2010 habe ich aufgehört, für Typemock zu arbeiten.

Warum Typemock und nicht Microsoft Fakes?

Obwohl Microsoft Fakes im Gegensatz zu Isolator und JustMock frei verfügbar ist, glaube ich, dass Microsoft Fakes ein ziemlich großes Bündel von unwartbarem Testcode in Ihrem Projekt erzeugen wird. Sein Design und seine Verwendung (Code-Generierung und überall Delegaten) führen zu einer sehr fragilen API, die schwer zu warten ist. Dieses Problem wird sogar in einem ALM-Rangers-Dokument zu Microsoft Fakes erwähnt, das Sie hier finden: `http://vsartesttoolingguide.codeplex.com/releases/view/102290`. Dort heißt es: »Wenn Sie Ihren zu testenden Code refaktorieren, dann lassen sich die Unit Tests, die Sie unter Verwendung von Shims und Stubs aus zuvor generierten Fake-Assemblies geschrieben haben, nicht länger kompilieren. Zu diesem Zeitpunkt gibt es für dieses Problem keine einfache Lösung, außer möglicherweise einen Satz maßgefertigter regulärer Ausdrücke zu verwenden, um Ihre Tests zu aktualisieren. Beachten Sie dies, wenn Sie eine Abschätzung für ein Refactoring von Code machen, für den zahlreiche Unit Tests existieren. Dies kann einen erheblichen Aufwand bedeuten.«

Ich werde Typemock Isolator für die nächsten Beispiele verwenden, denn dies ist das Framework, mit dem ich mich am wohlsten fühle. Isolator (während ich dies schreibe, ist Version 7.0 aktuell) verwendet den Begriff »Fake« und entfernt die Wörter »Mock« und »Stub« aus der API. Wenn Sie dieses Framework verwenden, können Sie ein »Fake« für Interfaces, versiegelte und statische Typen, nicht virtuelle Methoden und statische Methoden verwenden. Das heißt, Sie müssen sich keine Sorgen über Designänderungen machen (wozu Sie vielleicht nicht die Zeit haben oder was Sie aus Sicherheitsgründen eventuell nicht kön-

nen). Sie können mit dem Testen fast sofort beginnen. Es gibt auch eine freie, eingeschränkte Version von Typemock, sodass Sie dieses Produkt herunterladen und selber ausprobieren können. Sie müssen nur daran denken, dass es per Voreinstellung so eingeschränkt ist, dass es nur mit testbarem Standardcode funktioniert.

Das folgende Listing zeigt ein paar Beispiele für die Verwendung der Isolator-API, um Instanzen von Klassen zu fälschen.

```
[Test]
public void FakeAStaticMethod()
{
    Isolate
    .WhenCalled(()=>MyClass.SomeStaticMethod())
    .WillThrowException(new Exception());
}

[Test]
public void FakeAPrivateMethodOnAClassWithAPrivateConstructor()
{
    ClassWithPrivateConstructor c =
                Isolate.Fake.Instance<ClassWithPrivateConstructor>();
    Isolate.NonPublic.WhenCalled(c,"SomePrivateMethod").WillReturn(3);
}
```

Listing 10.1: »Fakes« für statische Methoden und Klassen mithilfe von Isolator

Wie Sie sehen können, ist die API einfach und klar und verwendet Generics und Delegates zur Rückgabe gefälschter Werte. Es gibt auch eine eigene API für VB.NET mit einer eher VB-typischen Syntax. Für beide APIs brauchen Sie nichts am Design Ihrer zu testenden Klassen zu ändern, damit die Tests funktionieren.

10.4.2 Verwenden Sie JMockit für Java-Legacy-Code

JMockit oder PowerMock ist ein Open-Source-Projekt, das die Java-Instrumentation-API verwendet, um die gleichen Sachen wie Typemock Isolator in .NET zu machen. Sie müssen das Design Ihres existierenden Projekts nicht verändern, um Ihre Komponenten von ihren Abhängigkeiten zu isolieren.

JMockit verfolgt einen *Tauschansatz*. Als Erstes erzeugen Sie eine manuell codierte Klasse, die die Klasse ersetzen wird, die eine Abhängigkeit für Ihre zu testende Komponente darstellt (sagen wir, Sie codieren eine Klasse `FakeDatabase`, um eine Klasse `Database` zu ersetzen). Dann verwenden Sie JMockit, um die Aufrufe der ursprünglichen Klasse gegen Aufrufe in Ihre eigene gefälschte Klasse auszutauschen. Sie können auch die Methoden einer Klasse umbenennen und sie als anonyme Methoden innerhalb des Tests wieder neu definieren.

Das nächste Listing zeigt das Beispiel eines Tests, der JMockit verwendet.

```
public class ServiceATest extends TestCase
{
```

```
private boolean serviceMethodCalled;

public static class MockDatabase
{
    static int findMethodCallCount;
    static int saveMethodCallCount;

    public static void save(Object o)
    {
        assertNotNull(o);
        saveMethodCallCount++;
    }

    public static List find(String ql, Object arg1)
    {
        assertNotNull(ql);
        assertNotNull(arg1);
        findMethodCallCount++;
        return Collections.EMPTY_LIST;
    }
}

protected void setUp() throws Exception
{
    super.setUp();
    MockDatabase.findMethodCallCount = 0;
    MockDatabase.saveMethodCallCount = 0;
    Mockit.redefineMethods(Database.class,
            MockDatabase.class);    //die Magie geschieht hier
}

public void testDoBusinessOperationXyz() throws Exception
{
    final BigDecimal total = new BigDecimal("125.40");
    Mockit.redefineMethods(ServiceB.class,
            new Object()            //die Magie geschieht hier
    {
        public BigDecimal computeTotal(List items)
        {
            assertNotNull(items);
            serviceMethodCalled = true;
            return total;
        }
```

```
        });

        EntityX data = new EntityX(5, "abc", "5453-1");
        new ServiceA().doBusinessOperationXyz(data);

        assertEquals(total, data.getTotal());
        assertTrue(serviceMethodCalled);
        assertEquals(1, MockDatabase.findMethodCallCount);
        assertEquals(1, MockDatabase.saveMethodCallCount);
    }
}
```

Listing 10.2: Die Verwendung von JMockit zum Austausch der Klassenimplementierung

JMockit eignet sich sehr gut, um mit dem Testen von Java-Legacy-Code zu beginnen.

10.4.3 Verwenden Sie Vise beim Refactoring Ihres Java-Codes

Michael Feathers hat ein interessantes Tool für Java geschrieben. Es ermöglicht Ihnen, zu prüfen, ob Sie die Werte in Ihrer Methode, die sich während des Refactorings ändern können, durcheinanderbringen. Wenn Ihre Methode beispielsweise ein Array von Werten ändert, dann wollen Sie sicher sein, dass Sie beim Umbau keinen Wert des Arrays verpfuschen.

Zu diesem Zweck kann die Methode `Vise.grip()` verwendet werden, wie das Beispiel im folgenden Listing zeigt.

```
import vise.tool.*;

public class RPRequest
{
    ...
    public int process(int level, RPPacket packet)
    {
        if (...)
        {
            if (...)
            {
                ...
            }
            else
            {
                ...
                bar_args[1] += list.size();
                Vise.grip(bar_args[1]);        //greift ein Objekt
                packet.add(new Subpacket(list, arrivalTime));
```

```
                if (packet.calcSize() > 2)
                    bar_args[1] += 2;
                Vise.grip(bar_args[1]);        //greift ein Objekt
            }
        }
        else
        {
            int reqLine = -1;
            bar_args[0] = packet.calcSize(reqLine);
            Vise.grip(bar_args[0]);
            ...
        }
    }
}
```

Listing 10.3: Die Verwendung von Vise in Java-Code, um zu verifizieren, dass die Werte beim Refactoring nicht geändert wurden

Anmerkung

Der Code in Listing 10.3 wurde mit freundlicher Genehmigung von der Seite `www.artima.com/weblogs/viewpost.jsp?thread=171323` kopiert.

Vise zwingt Sie dazu, Code-Zeilen in Ihren Produktionscode einzufügen, und es ist dazu da, das Refactoring des Codes zu unterstützen. Es gibt kein solches Tool für .NET, aber es sollte leicht sein, etwas Ähnliches zu schreiben. Jedes Mal, wenn Sie die Methode `Vise.grip()` aufrufen, wird überprüft, ob der Wert der Eingabevariablen noch so ist, wie er sein sollte. Es ist so, als ob Sie ein internes Assert mit einer einfachen Syntax in Ihren Code einbauen. Vise kann Sie auch über alle erfassten Variablen und ihre Werte benachrichtigen.

In Michael Feathers Blog auf `www.artima.com/weblogs/viewpost.jsp?thread=171323` können Sie mehr dazu nachlesen und Vise auch frei herunterladen.

10.4.4 Verwenden Sie Akzeptanztests, bevor Sie mit dem Refactoring beginnen

Bevor Sie mit dem Refactoring beginnen, ist es eine gute Idee, Ihrem Code Integrationstests hinzuzufügen. *FitNesse* ist ein Tool, das Ihnen dabei hilft, ein ganzes Paket von Integrations- und Akzeptanztests zu erzeugen. Vielleicht möchten Sie auch einen Blick auf Cucumber oder SpecFlow werfen. (Um mit Cucumber zu arbeiten, müssen Sie ein wenig über Ruby wissen. SpecFlow ist in .NET heimisch und dazu da, Cucumber-Szenarios zu parsen.) FitNesse erlaubt es Ihnen, Integrationstests für Ihre Anwendung zu schreiben (in Java oder .NET) und sie dann einfach zu ändern oder hinzuzufügen, ohne dafür Code schreiben zu müssen.

Der Einsatz des FitNesse-Frameworks umfasst drei Schritte:

1. Erzeugen Sie Code-Adapter-Klassen (*Fixtures* genannt), die Ihren Produktionscode einpacken können und die Aktionen repräsentieren, die ein Benutzer mit ihnen ausführen

möchte. Wenn es sich beispielsweise um eine Bankanwendung handeln würde, dann hätten Sie vielleicht eine Klasse `bankingAdapter` mit Methoden zum Abheben und Einzahlen.

2. Erzeugen Sie mithilfe einer speziellen Syntax HTML-Tabellen, die die FitNesse-Engine erkennt und parst. Diese Tabelle enthält die Werte, mit denen die Tests ausgeführt werden. Sie schreiben diese Tabellen auf die Seiten einer spezialisierten Wiki-Website, die die FitNesse-Engine im Hintergrund ausführt, sodass Ihr Testpaket der Welt da draußen über eine spezialisierte Website präsentiert wird. Jede Seite mit einer Tabelle (die Sie mit jedem Webbrowser darstellen können) ist wie eine reguläre Wiki-Seite editierbar und hat einen besonderen EXECUTE TESTS-Button. Diese Tabellen werden von der Test-Runtime geparst und anschließend in Testläufe übersetzt.
3. Betätigen Sie den EXECUTE TESTS-Button auf einer der Wiki-Seiten. Dieser Button bindet die FitNesse-Engine mit den Parametern in der Tabelle ein. Schließlich ruft die Engine Ihre spezialisierten Wrapper-Klassen auf, die wiederum die Zielanwendung einbinden. Dann überprüft sie die Rückgabewerte Ihrer Wrapper-Klassen.

Das Beispiel in Abbildung 10.5 zeigt eine FitNesse-Tabelle im Browser. Mehr zu FitNesse können Sie auf der Seite `http://fitnesse.org/` erfahren. Für die .NET-Integration gehen Sie auf die Seite `http://fitnesse.org/FitNesse.DotNet`.

Ich persönlich habe mich fast immer geärgert, wenn ich mit FitNesse gearbeitet habe. Es krankt sehr an der Benutzerfreundlichkeit und die Hälfte der Zeit funktioniert es nicht, insbesondere mit .NET-Krempel. Stattdessen ist Cucumber einen Blick wert. Sie finden es auf der Seite `http://cukes.info/`.

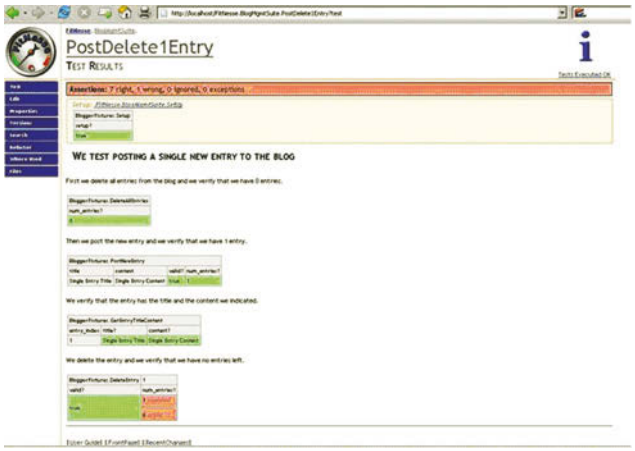


Abb. 10.5: Die Verwendung von FitNesse für das Integration Testing

10.4.5 Lesen Sie das Buch von Michael Feathers zu Legacy-Code

Effektives Arbeiten mit Legacy Code von Michael Feathers ist die einzige Quelle, die ich kenne und die sich mit den Themen beschäftigt, auf die Sie bei der Arbeit mit Legacy-Code treffen werden (mit Ausnahme dieses Kapitels). Es stellt viele Methoden des Refactorings und häufige Fehler dar, die dieses Buch gar nicht erst zu diskutieren versucht. Es ist sein Gewicht in Gold wert. Lesen Sie es!

10.4.6 Verwenden Sie NDepend, um Ihren Produktionscode zu untersuchen

NDepend ist ein relativ neues, kommerzielles Analyse-Tool für .NET, das viele Aspekte Ihrer übersetzten Assemblies grafisch darstellen kann, wie etwa Abhängigkeitsbäume, Code-Komplexität, Änderungen zwischen den Versionen des gleichen Assemblys und anderes mehr. Das Potenzial dieses Tools ist enorm und ich empfehle, sich mit ihm zu beschäftigen.

Das mächtigste Feature von NDepend ist seine spezielle Abfragesprache (namens CQL), die Sie auf die Struktur Ihres Codes anwenden können, um verschiedene Metriken der Komponente zu ermitteln. Beispielsweise können Sie leicht eine Abfrage erstellen, die alle Komponenten auflistet, die einen privaten Konstruktor besitzen.

Sie können NDepend auf der Seite `http://www.ndepend.com/` erhalten.

10.4.7 Verwenden Sie ReSharper für die Navigation und das Refactoring des Produktionscodes

ReSharper ist im Hinblick auf die Produktivität eines der besten Plug-ins für VS.NET. Zusätzlich zu den umfangreichen automatischen Refactoring-Fähigkeiten (sie sind wesentlich mächtiger als die in Visual Studio 2008 eingebauten) ist es für seine Navigations-Features bekannt. Wenn Sie in ein existierendes Projekt springen, kann ReSharper ganz einfach durch die Code-Basis navigieren. Mithilfe von Shortcuts können Sie von einem Punkt in der Solution zu jedem beliebigen anderen springen, der dazu in Beziehung stehen könnte.

Hier sind ein paar Beispiele für eine mögliche Navigation:

- Wenn Sie sich gerade in einer Klassen- oder Methodendeklaration befinden, können Sie von dort zu irgendeiner der abgeleiteten Klassen oder Methoden oder zur Basisimplementierung der aktuellen Methode oder Klasse springen, sofern sie existiert.
- Sie können alle Anwendungen einer bestimmten Variablen finden (sie wird im aktuellen Editor hervorgehoben).
- Sie können alle Anwendungen eines gemeinsamen Interface oder einer Klasse finden, die das Interface implementiert.

Diese und viele andere Shortcuts sorgen für eine wesentlich schmerzfreiere Navigation und erleichtern das Verständnis für die Struktur von bestehendem Code.

ReSharper funktioniert sowohl mit VB.NET als auch mit C#-Code. Sie können eine Testversion auf `http://www.jetbrains.com/` herunterladen.

10.4.8 Spüren Sie Code-Duplikate (und Bugs) mit Simian und TeamCity auf

Nehmen wir an, Sie haben einen Bug in Ihrem Code gefunden und Sie wollen sicherstellen, dass er nicht irgendwo anders dupliziert wurde.

TeamCity enthält einen eingebauten Duplikat-Finder für .NET. Mehr Informationen zu den TeamCity-Duplikaten finden Sie unter `http://confluence.jetbrains.com/display/TCD6/Duplicates+Finder+(.NET)`.

Mit *Simian* ist es einfach, sowohl eine Code-Duplizität aufzuspüren und herauszufinden, wie viel Arbeit vor Ihnen liegt, als auch den Code umzubauen, um die Duplizitäten zu entfernen. Simian ist ein kommerzielles Produkt, das für .NET, Java, C++ und andere Sprachen funktioniert.

Sie können Simian hier erhalten: `http://www.harukizaemon.com/simian/`.

10.5 Zusammenfassung

In diesem Kapitel ging es darum, wie man sich Legacy-Code zunächst annähern kann. Es ist wichtig, die verschiedenen Komponenten in Bezug auf die Zahl ihrer Abhängigkeiten, den Umfang an Logik und die Projektpriorität herauszuarbeiten. Sobald Sie diese Informationen ermittelt haben, können Sie in Abhängigkeit davon, wie schwierig es sein wird, eine Komponente in die Tests aufzunehmen, bestimmen, an welchen Komponenten Sie arbeiten wollen.

Wenn Ihr Team wenig oder keine Erfahrung im Umgang mit Unit Tests hat, dann ist es eine gute Idee, mit den einfachen Komponenten zu beginnen und das Vertrauen des Teams wachsen zu lassen, während es dem System mehr und mehr Tests hinzufügt. Ist Ihr Team erfahren, kann es helfen, die schwierigen Komponenten zuerst zu testen, um dann schneller durch den Rest des Systems zu kommen.

Wenn Ihr Team nicht mit dem Refactoring anfangen möchte, um die Testbarkeit zu verbessern, sondern direkt mit dem Unit Testing beginnen will, dann erweisen sich uneingeschränkte Isolation-Frameworks als hilfreich, denn sie erlauben Ihnen die Isolierung von Abhängigkeiten, ohne dass Sie das Design des existierenden Codes ändern müssen. Ziehen Sie ein solches Tool in Erwägung, wenn Sie mit Legacy-Code in .NET arbeiten. Wenn Sie mit Java arbeiten, dann kommt JMockit oder PowerMock aus den gleichen Gründen in Betracht.

Ich habe auch eine Reihe von Tools vorgestellt, die sich auf Ihrer Reise zu einer besseren Qualität des existierenden Codes als hilfreich erweisen können. Jedes dieser Tools kann in unterschiedlichen Phasen des Projekts eingesetzt werden, aber es liegt an Ihrem Team, wann es welches Tool benutzen will (wenn überhaupt).

Und schließlich kann, wie ein Freund einmal sagte, bei der Arbeit an Legacy-Code eine gute Flasche Wodka niemals schaden.

Kapitel 11

Design und Testbarkeit

Dieses Kapitel behandelt

- das Profitieren von den Design-Zielen der Testbarkeit
- das Abwägen der Vor- und Nachteile des Designs zum Zwecke der Testbarkeit
- das Angehen eines schwer zu testenden Designs

Für manche Entwickler ist es ein brisantes Thema, das Design des Codes zu ändern, um ihn besser testen zu können. Dieses Kapitel beschäftigt sich mit den grundlegenden Konzepten und Methoden des Designs zum Zwecke der Testbarkeit. Wir schauen auch auf die Vor- und Nachteile und darauf, wann es angemessen ist.

Als Erstes jedoch lassen Sie uns überlegen, warum Sie vor allem die Testbarkeit beim Design berücksichtigen sollten.

11.1 Warum sollte ich mir Gedanken um die Testbarkeit in meinem Design machen?

Diese Frage ist legitim. Beim Entwurf von Software lernen Sie, darüber nachzudenken, was die Software erreichen soll und was die Ergebnisse für den Endbenutzer des Systems sein werden. Aber die Tests für Ihre Software sind eine ganz andere Art von Benutzer. Dieser Benutzer stellt strikte Anforderungen an Ihre Software, sie alle stammen jedoch von einer mechanischen Anforderung: Testbarkeit. Diese Anforderung kann das Design Ihrer Software auf verschiedene Weise beeinflussen, meist zu ihrem Vorteil.

In einem testbaren Design sollte für jedes logische Code-Stück (Schleifen, `if`-Anweisungen, `switch`-Anweisungen usw.) leicht und schnell ein Unit Test geschrieben werden können, der diese Eigenschaften aufweist:

- Er läuft schnell ab.
- Er ist isoliert, was bedeutet, er kann unabhängig oder als Teil einer Gruppe von Tests und vor oder nach anderen Tests ausgeführt werden.
- Er erfordert keine externe Konfiguration.
- Er liefert ein konsistentes Erfolgs-/Misserfolgsergebnis.

Dies sind die FICC-Eigenschaften: schnell (fast), isoliert (isolated), konfigurationsfrei (configuration-free) und konsistent (consistent). Wenn es schwierig ist oder lange dauert, einen solchen Test zu schreiben, dann ist das System nicht testbar.

Wenn Sie sich die Tests als Benutzer Ihres Systems vorstellen, dann wird aus dem Designvorgang zum Zwecke der Testbarkeit eine Art des Denkens. Wenn Sie der testgetriebenen Entwicklung folgen, dann haben Sie keine andere Wahl, als testbare Systeme zu schreiben,

denn in TDD kommt der Test zuerst, bestimmt größtenteils das Design der API des Systems und zwingt sie in etwas, mit dem die Tests arbeiten können.

Da Sie nun wissen, was ein testbares Design ist, lassen Sie uns schauen, was das nach sich zieht. Dann gehen wir durch die Vor- und Nachteile einer solchen Designentscheidung, diskutieren Alternativen zum Ansatz des testbaren Designs und schauen uns ein Beispiel eines schwer zu testenden Designs an.

11.2 Designziele für die Testbarkeit

Es gibt mehrere Designpunkte, die die Testbarkeit des Codes wesentlich verbessern können. Robert C. Martin hat eine hübsche Liste von Designzielen für objektorientierte Systeme veröffentlicht, die zum großen Teil die Grundlage für die in diesem Kapitel gezeigten Designs bilden. Beachten Sie dazu seinen Artikel »Principles of OOD« auf der Seite `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`.

Ein Großteil der Ratschläge, die ich an dieser Stelle übernehme, handelt davon, Ihrem Code *Seams* zu geben – Stellen, an denen Sie anderen Code einfügen oder das Verhalten ersetzen können, ohne die ursprüngliche Klasse zu ändern. (Über Seams wird häufig im Zusammenhang mit dem *Open-Closed Principle* gesprochen, das in Martins Artikel »Principles of OOD« erwähnt wird.) Nehmen wir beispielsweise eine Methode an, die einen Webservice aufruft. Dann kann die API des Webservices hinter einem Interface versteckt werden, was es Ihnen erlaubt, den echten Webservice durch einen Stub, der einen beliebigen Wert zurückgeben kann, oder durch ein Mock-Objekt zu ersetzen. Kapitel 3 bis 5 erläutern Fakes, Mocks und Stubs im Detail.

Tabelle 11.1 listet einige grundlegende Testrichtlinien und ihre Vorteile auf. Die folgenden Abschnitte diskutieren sie dann ausführlicher.

Design Richtlinie	Vorteile
Deklarieren Sie Methoden standardmäßig als virtuell.	Das ermöglicht es Ihnen, die Methode zu Testzwecken in einer abgeleiteten Klasse zu überschreiben. Das Überschreiben ermöglicht eine Änderung des Verhaltens oder das Aufbrechen einer externen Abhängigkeit.
Benutzen Sie ein Interface-basiertes Design.	Das ermöglicht Ihnen die Verwendung von Polymorphismus, um Abhängigkeiten im System durch Ihre eigenen Stubs oder Mocks zu ersetzen.
Deklarieren Sie Klassen standardmäßig als nicht versiegelt.	Sie können nichts Virtuelles überschreiben, wenn die Klasse versiegelt ist (`final` in Java).
Vermeiden Sie es, konkrete Klassen innerhalb von Methoden mit Logik zu instanziieren. Holen Sie Klasseninstanzen aus Hilfsmethoden, Fabriken, »Inversion of Control«-Containern wie Unity oder von anderen Stellen, aber erzeugen Sie sie nicht direkt.	Das erlaubt Ihnen, Ihre eigenen, gefälschten Klasseninstanzen für die Methoden, die sie benötigen, aufzufahren, anstatt an die interne Produktionsinstanz einer Klasse gebunden zu sein.

Tabelle 11.1: Testdesignrichtlinien und -vorteile

Design Richtlinie	Vorteile
Vermeiden Sie direkte Aufrufe von statischen Methoden. Bevorzugen Sie Aufrufe von Instanzmethoden, die ihrerseits statische Methoden aufrufen.	Dies erlaubt Ihnen, den Aufruf statischer Methoden zu unterbrechen, indem Sie die Instanzmethoden überschreiben. (Sie können keine statischen Methoden überschreiben.)
Vermeiden Sie Konstruktoren und statische Konstruktoren, die Logik enthalten.	Es ist schwierig, Konstruktoren zu überschreiben. Wenn Sie die Konstruktoren einfach halten, vereinfacht das den Job, in Ihren Tests von der Klasse abzuleiten.
Trennen Sie die Singleton-Logik vom Singleton-Halter.	Wenn Sie ein Singleton haben, dann haben Sie damit eine Möglichkeit, dessen Instanz zu ersetzen.

Tabelle 11.1: Testdesignrichtlinien und -vorteile (Forts.)

11.2.1 Deklarieren Sie Methoden standardmäßig als virtuell

Java deklariert Methoden standardmäßig als virtuell, aber .NET-Entwickler haben dieses Glück nicht. In .NET müssen Sie, um das Verhalten einer Methode ändern zu können, diese explizit als virtuell deklarieren, damit Sie sie in einer abgeleiteten Klasse überschreiben können. Wenn Sie das tun, können Sie die »Extract and Override«-Methode benutzen, die ich in Kapitel 3 vorgestellt habe.

Eine Alternative zu dieser Methode ist es, die Klasse einen benutzerdefinierten Delegaten aufrufen zu lassen. Sie können diesen Delegaten von außen ersetzen, indem Sie eine Property setzen oder einen Parameter im Konstruktor oder in einer Methode übergeben. Dies ist kein typischer Ansatz, aber manche Systemdesigner finden ihn angemessen. Das folgende Listing zeigt ein Beispiel einer Klasse mit einem Delegaten, der von einem Test ersetzt werden kann.

```
public class MyOverridableClass
{
    public Func<int,int> calculateMethod=delegate(int i)
    {
        return i*2;
    };

    public void DoSomeAction(int input)
    {
        int result = calculateMethod(input);
        if (result==-1)
        {
            throw new Exception("Eingabe war ungültig");
        }
        //erledige andere Aufgaben
    }
}
```

```
[Test]
[ExpectedException(typeof(Exception))]
public void DoSomething_GivenInvalidInput_ThrowsException()
{
    MyOverridableClass c = new MyOverridableClass();
    int SOME_NUMBER=1;

    //ersetze Berechnungsmethode durch Stub,
    //um "ungültig" zurückzugeben
    c.calculateMethod = delegate(int i) { return -1; };

    c.DoSomeAction(SOME_NUMBER);
}
```

Listing 11.1: Eine Klasse, die einen Delegaten aufruft, der vom Test ersetzt werden kann

Die Verwendung virtueller Methoden ist bequem, aber Interface-basierte Designs sind ebenfalls eine gute Wahl, wie der nächste Abschnitt erläutert.

11.2.2 Benutzen Sie ein Interface-basiertes Design

Die Identifizierung von »Rollen« in der Applikation und ihre Abstraktion in Interfaces ist ein wichtiger Teil des Designprozesses. Eine abstrakte Klasse sollte keine konkreten Klassen aufrufen und konkrete Klassen sollten auch keine konkreten Klassen aufrufen, es sei denn, es würde sich um Datenobjekte handeln (Objekte, die Daten halten, aber kein eigenes Verhalten aufweisen). Dies erlaubt es Ihnen, in der Applikation vielfältige Seams zu haben, wo Sie einschreiten und Ihre eigene Implementierung einbauen können.

Beispiele zum Interface-basierten Ersatz finden Sie in den Kapiteln 3 bis 5.

11.2.3 Deklarieren Sie Klassen standardmäßig als nicht versiegelt

Manchen fällt es schwer, Klassen nicht standardmäßig zu versiegeln, denn sie haben gern die Kontrolle darüber, wer von wem in der Applikation ableitet. Das Problem ist, wenn Sie von einer Klasse nicht ableiten können, dann können Sie auch keine ihrer virtuellen Methoden überschreiben.

Manchmal kann man dieser Regel aus Sicherheitsgründen nicht folgen, dennoch sollte ihr zu folgen der Standard sein, nicht die Ausnahme.

11.2.4 Vermeiden Sie es, konkrete Klassen innerhalb von Methoden mit Logik zu instanziieren

Es kann knifflig sein, die Instanziierung konkreter Klassen innerhalb von Methoden mit Logik zu vermeiden, weil Sie es so gewohnt sind. Der Grund dafür, dies zu vermeiden, liegt darin, dass Ihre Tests später vielleicht die Kontrolle darüber benötigen, welche Instanz in der zu testenden Klasse verwendet wird. Wenn es kein Seam gibt, das diese Instanz zurückgibt, dann ist diese Aufgabe erheblich schwerer, es sei denn, Sie verwenden ein uneingeschränk-

tes Isolation-Framework wie z. B. Typemock Isolator. Wenn Ihre Methode beispielsweise einen Logger benötigt, dann instanziieren Sie den Logger nicht innerhalb der Methode. Holen Sie ihn von einer simplen Fabrikmethode und machen Sie diese Fabrikmethode virtuell, damit Sie sie später überschreiben und kontrollieren können, mit welchem Logger Ihre Methode arbeitet. Oder verwenden Sie DI über einen Konstruktor statt einer virtuellen Methode. Diese und andere Injektionsmethoden werden in Kapitel 3 diskutiert.

11.2.5 Vermeiden Sie direkte Aufrufe von statischen Methoden

Versuchen Sie, alle direkten Abhängigkeiten, die zur Laufzeit schwierig zu ersetzen wären, zu abstrahieren. In den meisten Fällen ist es schwierig oder mühsam, in einer statischen Sprache wie VB.NET oder C# das Verhalten einer statischen Methode zu ersetzen. Es ist eine Möglichkeit, mit dieser Situation klarzukommen, indem man mit dem »Extract and Override«-Refactoring (wie in Abschnitt 3.4 in Kapitel 3 gezeigt) die statische Methode wegabstrahiert.

Ein etwas extremerer Ansatz wäre es, statische Methoden überhaupt zu vermeiden. Auf diese Weise ist jedes Stück Logik Teil der Instanz einer Klasse, was es einfacher macht, dieses Stück Logik zu ersetzen. Der Mangel an Ersetzbarkeit ist einer der Gründe, warum manche, die Unit Testing oder TDD durchführen, Singletons nicht mögen; sie agieren als eine statische, öffentliche und gemeinsame Ressource, die schwierig zu überschreiben ist.

Es mag zu schwierig sein, statische Methoden komplett zu vermeiden, aber wenn Sie versuchen, die Zahl der Singletons oder der statischen Methoden in Ihrer Applikation zu minimieren, macht das die Dinge für Sie beim Testen einfacher.

11.2.6 Vermeiden Sie Konstruktoren und statische Konstruktoren, die Logik enthalten

Dinge wie konfigurationsbasierte Klassen werden oft als statische Klassen oder Singletons entworfen, weil so viele Teile der Applikation sie verwenden. Das macht es schwierig, sie während eines Tests zu ersetzen. Ein Weg, dieses Problem zu lösen, ist es, irgendeine Art von Kontroll-Inversions-Container (IoC-Container) zu verwenden (wie etwa Microsoft Unity, Autofac, Ninject, StructureMap, Spring.Net oder Castle Windsor – alles Open-Source-Frameworks für .NET).

Diese Container können vieles, aber sie alle bieten so etwas wie eine schlaue Fabrik, die es Ihnen erlaubt, Instanzen von Objekten zu erhalten, ohne zu wissen, ob es sich bei der Instanz um ein Singleton handelt oder was die der Instanz zugrunde liegende Implementierung ist. Sie fragen nach einem Interface (meist im Konstruktor) und ein Objekt, das zu diesem Typ passt, wird Ihnen automatisch bereitgestellt, während Ihre Klasse erzeugt wird.

Wenn Sie einen IoC-Container (wird auch als DI-Container bezeichnet) verwenden, dann abstrahieren Sie das Lebensdauermanagement eines Objekttypen weg und vereinfachen den Entwurf eines Objektmodells, das zum größten Teil auf Interfaces basiert, denn alle Abhängigkeiten in einer Klasse werden automatisch für Sie aufgefüllt.

Die Diskussion von Containern liegt außerhalb der Aufgabenstellung dieses Buches, aber Sie können eine umfassende Liste und einige Startpunkte im Artikel »List of .NET Dependency Injection Containers (IOC)« im Blog von Scott Hanselman finden: `http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx`.

11.2.7 Trennen Sie die Singleton-Logik und Singleton-Halter

Wenn Sie vorhaben, ein Singleton in Ihrem Design zu verwenden, dann teilen Sie die Logik der Singleton-Klasse und die Logik, die sie zu einem Singleton macht (beispielsweise der Teil, der eine statische Variable initialisiert), in zwei eigene Klassen auf. Somit können Sie das Prinzip der »Single Responsibility« (SRP) aufrechterhalten und haben außerdem eine Möglichkeit, die Logik des Singletons zu überschreiben.

Als Beispiel zeigt das nächste Listing eine Singleton-Klasse und Listing 11.3 zeigt sie nach dem Umbau in ein besser testbares Design.

```
public class MySingleton
{
    private static MySingleton _instance;
    public static MySingleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new MySingleton();
            }

            return _instance;
        }
    }
}
```

Listing 11.2: Ein untestbares Singleton-Design

```
public class RealSingletonLogic                //jetzt testbare Logik
{
    public void Foo()
    {
        //hier ist eine Menge Logik
    }
}

public class MySingletonHolder                  //Singleton Container
{
    private static RealSingletonLogic _instance;
    public static RealSingletonLogic Instance
    {
        get
        {
            if (_instance == null)
```

```
            {
                _instance = new RealSingletonLogic();
            }
            return _instance;
        }
    }
}
```

Listing 11.3: Die Singleton-Klasse nach dem Umbau in ein testbares Design

Lassen Sie uns nun, da ich einige mögliche Techniken für ein besser testbares Design vorgestellt habe, zum großen Ganzen zurückkehren. Sollen Sie es überhaupt tun und gibt es negative Konsequenzen eines solchen Handelns?

11.3 Vor- und Nachteile des Designs zum Zwecke der Testbarkeit

Design zum Zwecke der Testbarkeit ist für viele ein vorbelastetes Thema. Einige glauben, dass Testbarkeit eines der Standardmerkmale des Designs sein sollte, und andere glauben, dass das Design nicht leiden sollte, bloß weil es irgendjemand testen muss.

Man muss sich klarmachen, dass Testbarkeit kein Endziel an sich ist, sondern lediglich ein Nebenprodukt einer bestimmten Designschule, die die besser testbaren objektorientierten Prinzipien benutzt, wie sie von Robert C. Martin dargelegt wurden (siehe den Anfang von Abschnitt 11.2). In einem Design, das Klassenerweiterbarkeit und Abstraktion favorisiert, können Seams für testbezogene Aktionen leicht gefunden werden. Alle bisher in diesem Kapitel gezeigten Methoden stimmen weitgehend mit Robert Martins Prinzipien überein: Klassen, deren Verhalten durch Vererbung und Überschreiben geändert werden kann oder durch Injizierung eines Interface, sind »offen für eine Erweiterung, aber abgeschlossen gegen eine Veränderung« – das Open-Closed-Prinzip. Solche Klassen weisen gewöhnlich auch eine Kombination des DI-Prinzips und des IoC-Prinzips auf, was die Konstruktor-Injektion erlaubt. Durch Anwendung des Prinzips der Einzel-Verantwortlichkeit können Sie beispielsweise ein Singleton von seiner Halter-Logik trennen und in eine eigene Singelton-Halter-Klasse auslagern. Nur das Liskov'sche Substitutionsprinzip bleibt alleine in der Ecke stehen, denn mir fällt kein einziges Beispiel ein, in dem ein Verstoß gegen das Prinzip auch die Testbarkeit zerstört. Aber die Tatsache, dass Ihre testbaren Designs irgendwie mit den SOLID-Prinzipien zu korrelieren scheinen, bedeutet *nicht* notwendigerweise, dass Ihr Design gut ist oder dass Sie ein Design-Talent haben. Oh nein. Sehr wahrscheinlich könnte Ihr Design, genauso wie meines, besser sein. Schnappen Sie sich ein gutes Buch zu diesem Thema wie etwa *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003) von Eric Evans oder *Refactoring to Patterns* (Addison-Wesley, 2004) von Joshua Kerievsky. Oder wie wäre es mit *Clean Code* von Robert Martin (mitp-Verlag, 2009). Geht auch!

Ich sehe eine ganze Menge schlecht entworfenen, sehr gut testbaren Code da draußen. Was beweist, dass TDD ohne ordentliches Wissen um Design nicht unbedingt einen guten Einfluss auf das Design hat.

Aber die Frage bleibt, ob dies der beste Weg ist, die Dinge anzupacken. Was sind die Nachteile des Testbarkeits-getriebenen Designs? Was passiert, wenn man es mit Legacy-Code zu tun hat? Und so weiter.

11.3.1 Arbeitsumfang

In den meisten Fällen ist der Arbeitsaufwand für das Design höher, wenn Sie die Testbarkeit berücksichtigen, denn es bedeutet gewöhnlich, dass Sie mehr Code schreiben müssen. Selbst Uncle Bob sagt gerne (mit einer Sherlock-Holmes-Stimme und mit Pfeife) in seinen ausführlichen und mitunter lustigen Videos auf `http://cleancoders.com`, dass er mit einem grob vereinfachenden Design beginnt, das nur die einfachsten Dinge tut, und dann auch nur dort umbaut, wo er die Notwendigkeit sieht.

Sie könnten argumentieren, dass die für die Testbarkeit zusätzlich benötigte Arbeit am Design Themen sichtbar macht, die Sie bisher noch nicht in Betracht gezogen haben und die Sie in jedem Fall noch in Ihr Design mit hätten aufnehmen müssen (Trennung der Belange, Single-Responsibility-Prinzip usw.).

Wenn Sie aber mit Ihrem Design, so wie es ist, zufrieden sind, dann kann es problematisch sein, Änderungen zum Zwecke der Testbarkeit, die ja nicht Teil der Produktion ist, durchzuführen. Wieder könnten Sie argumentieren, dass Testcode genauso wichtig ist wie Produktionscode, denn er stellt die Charakteristika der API-Verwendung in Ihrem Domänenmodell heraus und zwingt Sie, darauf zu achten, wie jemand anderes Ihren Code verwenden wird.

Von diesem Punkt an sind Diskussionen zu diesem Thema selten noch produktiv. Lassen Sie uns einfach sagen, dass mehr Code und mehr Arbeit erforderlich sind, wenn Testbarkeit ein Thema ist, aber dass das Designen zum Zwecke der Testbarkeit Sie mehr über Ihre API nachdenken lässt, was wiederum eine gute Sache ist.

11.3.2 Komplexität

Das Designen zum Zwecke der Testbarkeit hat manchmal ein wenig (oder sehr stark) den Beigeschmack, als ob man die Dinge zu kompliziert macht. Sie könnten sich dabei ertappen, Interfaces hinzuzufügen, wo es nicht natürlich erscheint, Interfaces zu verwenden oder die Semantik des Klassenverhaltens preiszugeben, ohne dies vorher in Betracht gezogen zu haben. Insbesondere kann es schwierig und nervenaufreibend werden, in der Code-Basis zu navigieren und die echte Implementierung einer Methode zu finden, wenn viele Dinge Interfaces haben und wegabstrahiert sind.

Sie könnten argumentieren, dass die Verwendung eines Tools wie ReSharper dieses Argument widerlegt, denn die Navigation mit ReSharper ist viel einfacher. Ich stimme zu, dass es die meisten Navigationsmühen erleichtert. Das richtige Tool für den richtigen Job kann sehr helfen.

11.3.3 Das Preisgeben von sensiblem IP

Viele Projekte enthalten sensibles geistiges Eigentum (Intellectual Property), das nicht preisgegeben werden sollte, das aber das Design zum Zwecke der Testbarkeit gezwungenermaßen offenlegen würde: Sicherheits- oder Lizenzinformationen oder vielleicht patentgeschützte Algorithmen. Es gibt dafür Workarounds – die Dinge intern zu halten und das

Attribut `[InternalsVisibleTo]` zu verwenden –, aber im Wesentlichen widersetzen sie sich der ganzen Idee des testbaren Designs. Sie ändern das Design, halten die Logik aber trotzdem verborgen. Tolle Sache.

Das Designen zum Zwecke der Testbarkeit verliert an diesem Punkt ein wenig von seinem Glanz. Manchmal kann man nicht um Sicherheits- oder Patentangelegenheiten herumarbeiten. Man muss manchmal ändern, was man tut, oder auf dem Weg Kompromisse machen.

11.3.4 Manchmal geht's nicht

Manchmal gibt es politische oder andere Gründe für eine bestimmte Art von Design und Sie können es nicht ändern (Soul-Crushing-Enterprise-Software-Projekte, jemand da?). Manchmal haben Sie nicht die Zeit, Ihr Design umzubauen, oder das Design ist zu zerbrechlich, um es umzubauen. Dies ist ein anderer Fall, wo das Designen zum Zwecke der Testbarkeit zusammenbricht – wenn das Umfeld Sie davon abhält. Das ist ein Beispiel der Einflussfaktoren, die ich in Kapitel 9 erläutert habe.

Ich habe jetzt die Vor- und Nachteile aufgezeigt, lassen Sie uns nun Alternativen betrachten.

11.4 Alternativen des Designs zum Zwecke der Testbarkeit

Es ist interessant, über den Tellerrand auf andere Sprachen zu schauen und andere Arten des Umgangs zu sehen.

In dynamischen Sprachen wie Ruby oder Smalltalk ist der Code inherent testbar, weil man alles und jedes dynamisch zur Laufzeit austauschen kann. In so einer Sprache können Sie designen, wie Sie wollen, ohne sich Sorgen um die Testbarkeit machen zu müssen. Sie brauchen kein Interface, um etwas zu ersetzen, und Sie müssen nichts öffentlich machen, um es zu überschreiben. Sie können sogar das Verhalten von Kerntypen dynamisch ändern und niemand wird Sie anschreien oder Ihnen sagen, dass Sie nicht kompilieren können.

In einer Welt, wo alles testbar ist, müssen Sie da noch die Testbarkeit im Design berücksichtigen? Die erwartete Antwort ist natürlich »Nein«. In dieser Art von Welt sollten Sie frei sein, Ihr eigenes Design zu wählen.

11.4.1 Design-Argumente und Sprachen mit dynamischen Typen

Interessanterweise gibt es seit 2010 zum SOLID-Design eine wachsende Diskussion in der Ruby-Community, woran ich auch teilgenommen habe. SOLID steht für *Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation und Dependency Inversion*. »Nur dass du es kannst, heißt nicht, dass du es auch sollst«, sagen einige Rubyisten, beispielsweise Avdi Grimm, der Autor von Objects on Rails, abrufbar auf `http://objectsonrails.com`. Sie können viele Blog-Einträge finden, die den Zustand des Designs in der Ruby-Community wiederkäuen, wie etwa hier: `http://jamesgolick.com/2012/5/22/objectify-a-better-way-to-build-rails-applications.html`. Andere Rubyisten antworten darauf mit: »Nerv uns nicht mit diesem Übertechnisierungskram.« Vor allem David Heinemeier Hansson, auch bekannt als DHH und der ursprüngliche Schöpfer des Ruby-on-Rails-Frameworks, antwortet im Blog-Beitrag »Dependency Injection is not a Virtue« auf: `http://david.heinemeierhansson.com/2012/dependency-injection-is-not-a-virtue.html`.

Der Spaß geht dann auf Twitter weiter, wie Sie sich denken können.

Das Lustige an dieser Art von Diskussionen ist, wie sehr sie mich an die gleiche Art von Diskussionen erinnern, die um 2008/2009 in der .NET-Community liefen, insbesondere in der kürzlich geschlossenen ALT.NET-Community. (Die meisten der ALT.NET-Leute entdeckten Ruby oder Node.js und wanderten ab von .NET, nur um ein Jahr später wiederzukommen und den .NET-Kram nebenbei »für Geld« zu machen. Schuldig!) Der große Unterschied hier ist, dass es um Ruby geht. In der .NET-Community gab es zumindest den Schnipsel eines halbgaren Belegs, der die Seite der »Lasst uns SOLID anwenden«-Leute zu unterstützen schien: Beispielsweise können Sie Ihre Designs nicht testen ohne offene/geschlossene Klassen, da Ihnen der Compiler auf den Kopf hauen würde, wenn Sie es auch nur versuchten. Die ganzen Design-Leute sagten also: »Seht ihr? Der Compiler will euch sagen, dass euer Design Mist ist«, was im Rückblick ziemlich dumm ist, denn viele testbare Entwürfe scheinen dennoch besonders großer Mist zu sein, auch wenn sie testbar sind. Und jetzt kommen da so ein paar Ruby-Leute an und sagen, dass sie die SOLID-Prinzipien anwenden wollen? Warum um Himmels willen sollten sie das wollen?

Es scheint, dass die Anwendung von SOLID einige Vorteile mit sich bringt: Der Code kann leichter gewartet und verstanden werden, was in der Ruby-Welt ein ziemlich großes Problem sein kann. Manchmal ist das für Ruby ein größeres Problem als für Sprachen mit statischen Typen, denn in Ruby kann man es mit dynamischem Code zu tun haben, der alle Arten von darunterliegendem, scheußlichem, verstecktem und umgeleitetem Code aufrufen kann, und Sie können in einem Meer aus Tränen enden, wenn dies geschieht. Tests helfen, aber nur bis zu einem gewissen Grad.

Wie auch immer, was wollte ich sagen? Ursprünglich haben die Leute nicht einmal versucht, das Design in Ruby-Software testbar zu machen, denn der Code war sowieso schon testbar. Alles war gut, aber dann entdeckten sie die Ideen zum *Design* von Code. Das impliziert, dass *Design* eine eigenständige Tätigkeit ist, mit anderen Konsequenzen als einem einfachen Code-Umbau zum Zwecke der Testbarkeit.

Zurück zu .NET und den Sprachen mit statischen Typen: Betrachten Sie eine auf .NET bezogene Analogie, die zeigt, wie die Verwendung von Tools die Art, über Probleme zu denken, ändern kann und sich manchmal große Probleme in Luft auflösen. In einer Welt, wo der Speicher für Sie verwaltet wird, müssen Sie da noch die Speicherverwaltung im Design berücksichtigen? Meist wird die Antwort »Nein« lauten. Wenn Sie mit Sprachen arbeiten, in denen der Speicher nicht für Sie verwaltet wird (beispielsweise C++), müssen Sie sich über die Speicheroptimierung und -freigabe Gedanken machen und dies im Design berücksichtigen, andernfalls wird die Anwendung darunter leiden. Das hält Sie nicht davon ab, gut entworfenen Code zu haben, aber die Speicherverwaltung ist nicht der Grund dafür. Die Lesbarkeit des Codes, Benutzerfreundlichkeit und andere Werte treiben dies voran. Sie sollten keine Scheinargumente verwenden, um das Design Ihres Codes festzulegen.

Auf die gleiche Weise, indem Sie den testbaren, objektorientierten Design-Prinzipien folgen, können Sie testbare Designs als Nebenprodukt erhalten, aber Testbarkeit sollte kein Ziel in Ihrem Design sein. Es ist dazu da, ein bestimmtes Problem zu lösen. Wenn ein Tool daherkommt, das das Testbarkeitsproblem für Sie löst, dann gibt es keine Notwendigkeit, die Testbarkeit ausdrücklich im Design zu berücksichtigen. Solche Designs haben noch andere Vorzüge, aber ihre Verwendung sollte eine Option sein und keine schlichte Tatsache.

Das Hauptproblem mit nicht testbaren Designs ist die Unfähigkeit, Abhängigkeiten zur Laufzeit ersetzen zu können. Deshalb müssen Sie Interfaces anlegen, Methoden virtuell

machen und andere ähnliche Dinge tun. Es gibt Tools, die dabei helfen können, Abhängigkeiten im .NET-Code zu ersetzen, ohne ein Refactoring zum Zwecke der Testbarkeit durchführen zu müssen. Das ist eine Stelle, an der uneingeschränkte Isolation-Frameworks ins Spiel kommen.

Bedeutet die Existenz von uneingeschränkten Frameworks, dass das Design zum Zwecke der Testbarkeit überflüssig ist? In einer gewissen Weise ja. Sie werden von der Notwendigkeit befreit, Testbarkeit als ein Designziel anzusehen. Die objektorientierten Muster, die Bob Martin präsentiert, haben großartige Vorteile und sie sollten nicht wegen der Testbarkeit verwendet werden, sondern weil sie im Bezug auf das Design Sinn machen. Sie können dazu beitragen, den Code leichter zu warten, leichter zu lesen und leichter zu entwickeln, auch wenn Testbarkeit kein Thema mehr ist.

Ich runde diese Ausführungen mit einem Beispiel eines schwer zu testenden Designs ab.

11.5 Beispiel eines schwer zu testenden Designs

Es ist einfach, interessante Projekte zu finden, in die man sich eingraben kann. Ein solches Projekt ist die Open Source BlogEngine.NET, deren Quellcode Sie hier finden können: `http://blogengine.codeplex.com/SourceControl/latest`. Sie können es erkennen, wenn ein Projekt ohne testgetriebenen Ansatz gebaut wurde oder ohne einen Gedanken an die Testbarkeit. In diesem Fall gibt es überall Statisches: statische Klassen, statische Methoden, statische Konstruktoren. Das ist nicht schlecht im Sinne des Designs. Vergessen Sie nicht, dies ist kein Buch über Design. Aber es *ist* schlecht im Sinne der Testbarkeit.

Werfen wir einen Blick auf eine einzelne Klasse aus der Solution: die Manager-Klasse im Ping-Namespace (zu finden unter: `http://blogengine.codeplex.com/SourceControl/latest#BlogEngine/BlogEngine.Core/Ping/Manager.cs`).

```
namespace BlogEngine.Core.Ping
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text.RegularExpressions;
    public static class Manager
    {
        private static readonly Regex TrackbackLinkRegex = new Regex(
            "trackback:ping=\"([^\"]+)\"", RegexOptions.IgnoreCase |
            RegexOptions.Compiled);

        private static readonly Regex UrlsRegex = new Regex(
        @"<a.*?href=[""']?(?<url>.*?)[""'].*?>(?<name>.*?)</a>",
        RegexOptions.IgnoreCase | RegexOptions.Compiled);

        public static void Send(IPublishable item, Uri itemUrl)
        {
```

```
            foreach (var url in GetUrlsFromContent(item.Content))
            {
                var trackbackSent = false;

                if (BlogSettings.Instance.EnableTrackBackSend)
                {
                    // ignoreRemoteDownloadSettings should be set to
                    // true  for backwards compatibilty with
                    // Utils.DownloadWebPage.
                    var remoteFile = new RemoteFile(url, true);
                    var pageContent = remoteFile.GetFileAsString();

                    var trackbackUrl =
                       GetTrackBackUrlFromPage(pageContent);

                    if (trackbackUrl != null)
                    {
                        var message =
                        new TrackbackMessage(item,
                                              trackbackUrl, itemUrl);
                        trackbackSent = Trackback.Send(message);
                    }
                }
                if (!trackbackSent &&
                        BlogSettings.Instance.EnablePingBackSend)
                {
                    Pingback.Send(itemUrl, url);
                }
            }
        }
        private static Uri GetTrackBackUrlFromPage(string input)
        {
            var url = TrackbackLinkRegex.Match(input).
                                  Groups[1].ToString().Trim();
            Uri uri;

            return
            Uri.TryCreate(url, UriKind.Absolute,
                                  out uri) ? uri : null;
        }
        private static IEnumerable<Uri>
                            GetUrlsFromContent(string content)
```

```
        {
            var urlsList = new List<Uri>();
            foreach (var url in UrlsRegex.Matches(content).
                Cast<Match>().Select(myMatch =>
                myMatch.Groups["url"].ToString().Trim()))
            {
                Uri uri;
                if (Uri.TryCreate(url, UriKind.Absolute, out uri))
                {
                    urlsList.Add(uri);
                }
            }
            return urlsList;
        }
    }
}
```

Wir konzentrieren uns auf die Methode `send` der Klasse `Manager`. Diese Methode ist dazu da, irgendeine Art von Ping oder Trackback (für diese Diskussion kümmern wir uns nicht darum, was das eigentlich bedeuten soll) zu senden, wenn sie die Erwähnung einer URL in einem Blog-Eintrag eines Benutzers findet. Eine Reihe von Anforderungen ist hier bereits implementiert:

- Sende den Ping oder Trackback nur, wenn ein globales Konfigurations-Objekt auf den Wert `true` gesetzt ist.
- Wenn kein Ping gesendet wurde, versuche einen Trackback zu senden.
- Sende einen Ping oder Trackback für jede URL, die im Inhalt des Eintrags gefunden werden kann.

Warum glaube ich, dass diese Methode wirklich schwer zu testen ist? Es gibt mehrere Gründe:

- Die Abhängigkeiten (wie etwa die Konfiguration) sind alle statische Methoden, weshalb man sie ohne ein uneingeschränktes Framework nicht einfach fälschen und ersetzen kann.
- Auch wenn Sie in der Lage wären, die Abhängigkeiten zu fälschen, so gäbe es keine Möglichkeit, sie als Parameter oder Properties zu injizieren. Sie werden direkt verwendet.
- Sie könnten versuchen, »Extract and Override« zu verwenden (wie in Kapitel 3 erläutert), um die Abhängigkeiten über virtuelle Methoden, die Sie in einer abgeleiteten Klasse überschreiben können, aufzurufen – allerdings ist die Klasse `Manager` statisch und kann somit keine nichtstatischen Methoden enthalten und damit eindeutig auch keine virtuellen. Also ist es auch nichts mit »Extract and Override«.
- Selbst wenn die Klasse nicht statisch wäre, so ist es die Methode, die Sie testen wollen, doch. Somit kann sie keine virtuellen Methoden direkt aufrufen. Die Methode müsste eine Instanzmethode sein, um in »Extract and Override« umgebaut zu werden. Aber sie ist es nicht.

So würde ich das Refactoring dieser Klasse angehen (vorausgesetzt ich hätte Integrationstests):

1. Entferne `static` von der Klasse.
2. Lege eine nicht statische Kopie der Methode `send()` mit den gleichen Parametern an. Ich würde ein Präfix `Instance` verwenden, damit sie `InstanceSend()` heißt und sich kompilieren ließe, ohne mit der originalen statischen Methode zu kollidieren.
3. Entferne den ganzen Code aus der originalen statischen Methode und ersetze ihn durch `Manager().InstanceSend(item, itemUrl);` damit die statische Methode jetzt nur noch ein Weiterleitungs-Mechanismus ist. Das stellt sicher, dass der existierende Code, der diese Methode aufruft, weiterhin funktioniert (auch bekannt als Refactoring!).
4. Nun, da ich eine Instanz-Klasse und eine Instanz-Methode habe, kann ich weitermachen und »Extract and Override« auf Teile der Methode `InstanceSend()` anwenden und Abhängigkeiten aufbrechen, etwa durch Herausziehen des Aufrufs von `BlogSettings.Instance.EnableTrackBackSend` in eine eigene virtuelle Methode. Die kann ich später überschreiben, wenn ich in meinen Tests von `Manager` ableite.
5. Ich bin zwar noch nicht fertig, aber ich habe einen Anfang. Nach Bedarf kann ich mit dem Refactoring und »Extract and Override« weitermachen.

So sähe die Klasse schließlich aus, bevor ich mit »Extract and Override« beginnen kann:

```
public static class Manager
{
    ...
    public static void Send(IPublishable item, Uri itemUrl)
    {
        new Manager().InstanceSend (item, itemUrl);
    }

    public void InstanceSend(IPublishable item, Uri itemUrl)
    {
        foreach (var url in GetUrlsFromContent(item.Content))
        {
            var trackbackSent = false;

            if (BlogSettings.Instance.EnableTrackBackSend)
            {
                // ignoreRemoteDownloadSettings should be set to
                // true  for backwards compatibilty with
                // Utils.DownloadWebPage.
                var remoteFile = new RemoteFile(url, true);
                var pageContent = remoteFile.GetFileAsString();

                var trackbackUrl =
                       GetTrackBackUrlFromPage(pageContent);
```

```
                if (trackbackUrl != null)
                {
                    var message =
                    new TrackbackMessage(item,
                                         trackbackUrl, itemUrl);
                    trackbackSent = Trackback.Send(message);
                }
            }
            if (!trackbackSent &&
                        BlogSettings.Instance.EnablePingBackSend)
            {
                Pingback.Send(itemUrl, url);
            }
        }
    }
    private static Uri GetTrackBackUrlFromPage(string input)
    {
        ...
    }
    private static IEnumerable<Uri>
                            GetUrlsFromContent(string content)
    {
        ...
    }
}
```

Hier sind ein paar Dinge, die ich hätte machen können, damit diese Methode besser testbar wäre:

- Standardmäßig sollten Klassen nicht statisch sein. Es gibt nur ganz selten einen guten Grund für eine rein statische Klasse in C#.
- Verwende Instanzmethoden statt statischer Methoden.

Es gibt eine Demo, wie ich dieses Refactoring durchführe, in einem Video eines Online-TDD-Kurses auf `http://courses.osherove.com`.

11.6 Zusammenfassung

In diesem Kapitel haben wir uns die Idee des Designs zum Zwecke der Testbarkeit angeschaut: was das im Hinblick auf die Designmethoden bedeutet, ihre Vor- und Nachteile und mögliche Alternativen. Es gibt keine einfachen Antworten, aber die Fragen sind interessant. Die Zukunft des Unit Testings wird davon abhängen, wie die Leute solche Themen angehen und welche Tools als Alternativen vorhanden sind.

Testbare Designs spielen nur in statischen Sprachen wie C# oder VB.NET eine Rolle, wo die Testbarkeit von proaktiven Designentscheidungen abhängt, die es erlauben, Dinge zu

ersetzen. Das Designen zum Zwecke der Testbarkeit spielt in dynamischen Sprachen, wo die Dinge *von vornherein leichter testbar* sind, weniger eine Rolle. In solchen Sprachen können die meisten Dinge leicht ausgetauscht werden, ganz unabhängig vom Projektdesign. Das befreit die Nutzer-Community solcher Sprachen vom Scheinargument, dass ein Testbarkeitsmangel des Codes gleichbedeutend mit einem schlechten Design ist, und lässt sie sich darauf konzentrieren, was gutes Design auf einer tieferen Ebene erreichen sollte.

Testbare Designs enthalten virtuelle Methoden, nicht versiegelte Klassen, Interfaces und eine klare Trennung der Belange. Sie haben weniger statische Klassen und Methoden und wesentlich mehr Instanzen von logischen Klassen. Tatsächlich korrelieren testbare Designs mit den SOLID-Design-Prinzipien, was aber nicht unbedingt bedeutet, dass Sie automatisch ein gutes Design haben. Vielleicht ist es an der Zeit, dass das Endziel nicht die Testbarkeit sein sollte, sondern stattdessen einfach nur gutes Design.

Wir haben uns ein kurzes Beispiel angeschaut, das so ziemlich untestbar ist, und all die Schritte, die notwendig wären, um es in ein testbares Beispiel umzubauen. Bedenken Sie doch, wie einfach es zu testen wäre, wenn TDD schon beim Schreiben des Codes angewendet worden wäre! Von der ersten Zeile an wäre es testbar gewesen und wir hätten nicht durch all die Schleifen gehen müssen.

Das ist genug für den Augenblick, junger Hüpfer. Die Welt da draußen ist fantastisch und voller Dinge, in die Sie sich, wie ich glaube, gerne vertiefen würden.

11.7 Zusätzliche Ressourcen

Mir scheint, dass viele derjenigen, die dieses Buch lesen, die folgende Verwandlung durchlaufen:

- Nachdem sie sich an die Namenskonventionen gewöhnt haben, fangen sie an, andere anzuwenden oder neue zu kreieren. Das ist großartig. Meine Namenskonventionen sind für einen Anfänger gut und ich benutze sie immer noch selbst, aber sie sind nicht die einzige Variante. Sie sollen sich mit Ihren Testnamen wohlfühlen.
- Sie fangen an, nach anderen Formen für das Schreiben von Tests Ausschau zu halten, wie etwa nach Frameworks im Stil der verhaltensgetriebenen Entwicklung (Behaviour-Driven Development, BDD), beispielsweise MSpec oder NSpec. Das ist großartig, weil die Lesbarkeit weiterhin gut ist, solange Sie die drei wichtigen Teile von Informationen bewahren (was Sie testen, unter welchen Bedingungen Sie testen und wie das erwartete Resultat ist). In den APIs im BDD-Stil ist es einfacher, einen einzelnen Einstiegspunkt zu setzen und mehrere Endresultate mit unterschiedlichen Anforderungen zu testen, und das in einer sehr lesbaren Weise. Das liegt daran, dass die meisten BDD-artigen APIs eine hierarchische Art der Entwicklung ermöglichen.
- Sie automatisieren mehr Integrationstests und mehr Systemtests, denn sie sehen das Unit Testing auf einer zu tiefen Ebene. Das ist ebenfalls großartig, denn Sie tun, was Sie tun müssen, um das Vertrauen zu bekommen, das Sie brauchen, um den Code zu ändern. Wenn Sie schließlich keine Unit Tests in Ihrem Projekt haben, aber immer noch mit hoher Geschwindigkeit, mit Vertrauen und Qualität entwickeln können, dann ist das fantastisch, und könnte ich dann bitte ein wenig von dem bekommen, was Sie da haben? (Es ist möglich, aber an einem bestimmten Punkt werden die Tests sehr langsam. Wir haben noch nicht den magischen Weg gefunden, damit das vollständig geschieht.)

Wie ist es mit Büchern?

Eines, das die Themen dieses Buches im Hinblick auf das Design ergänzt, ist *Growing Object-Oriented Software, Guided by Tests* von Steve Freeman und Nat Pryce.

Ein gutes Referenz-Buch zu Mustern und Antimustern im Unit Testing ist *xUnit Test Patterns: Refactoring Test Code*, von Gerard Meszaros.

Effektives Arbeiten mit Legacy Code von Michael Feathers ist Pflichtlektüre, wenn Sie sich mit Themen zu Legacy Code auseinandersetzen.

Es gibt auch eine umfangreichere und kontinuierlich aktualisierte (zweimal im Jahr, wirklich) Liste interessanter Bücher auf `ArtOfUnitTesting.com`.

Wenn Sie ein paar Test-Reviews sehen wollen, können Sie die Videos, die ich gemacht habe, ausprobieren. Ich nehme die Tests von Open-Source-Projekten und zerlege sie, um zu sehen, was man besser machen könnte: `http://artofunittesting.com/test-reviews/`.

Ich habe auch eine ganze Menge freier Videos, Test-Reviews, Pair-Programming-Sessions-Konferenzvorträge zur testgetriebenen Entwicklung ins Netz gestellt: `http://artofunittesting.com` und `http://osherove.com/Videos`. Ich hoffe, dass Ihnen das zusätzliche Informationen über dieses Buch hinaus geben wird.

Vielleicht sind Sie auch daran interessiert, an meinem TDD-Master-Kurs (steht als Online-Streaming-Videos bereit) teilzunehmen auf `http://TDDCourse.Osherove.com`.

Sie erwischen mich immer mit `@RoyOsherove` auf Twitter oder kontaktieren Sie mich direkt über `http//Contact.Osherove.com`.

Ich freue mich, von Ihnen zu hören!

Anhang A

Tools und Frameworks

Dieses Buch wäre nicht komplett ohne einen Überblick über einige Tools und Basismethoden, die Sie beim Schreiben von Unit Tests anwenden können. Vom Datenbanktest über den UI-Test zum Webtest listet dieser Anhang Tools auf, deren Verwendung Sie in Betracht ziehen sollten. Einige werden für das Integration Testing benutzt, andere erlauben das Unit Testing. Ich werde auch einige erwähnen, von denen ich glaube, dass sie sich besonders für Anfänger eignen.

Die hier aufgelisteten Tools und Methoden sind in die folgenden Kategorien unterteilt:

- Isolation-Frameworks
- Test-Frameworks
  - Test-Runner
  - Test-APIs
  - Test Helper APIs
- DI- und IoC-Container
- Datenbank-Tests
- Webtests
- UI-Tests
- Thread-bezogene Tests
- Akzeptanztests

Hinweis

Eine aktualisierte Version der Liste der Tools und Techniken finden Sie auf der Webseite des Buches: `http://ArtOfUnitTesting.com`.

Fangen wir an.

A.1 Isolation-Frameworks

Mock- oder *Isolation-Frameworks* sind alltäglich in den fortgeschrittenen Unit-Testing-Szenarios. Es gibt viele, unter denen Sie wählen können, und das ist eine großartige Sache:

- Moq
- Rhino Mocks
- Typemock Isolator
- JustMock

- Moles/Microsoft Fakes
- NSubstitute
- FakeItEasy
- Foq

Die vorherige Ausgabe dieses Buches enthielt die folgenden Tools, die ich entfernt habe, weil sie nicht mehr aktuell oder inzwischen unbedeutend sind:

- NMock
- NUnit.Mocks

Hier ist eine kurze Beschreibung zu jedem Framework:

A.1.1 Moq

Moq ist ein Open-Source-Isolation-Framework und hat eine API, die versucht, sowohl einfach erlernbar als auch einfach benutzbar zu sein. Die API war eine der ersten, die dem Arrange-Act-Assert-Stil (im Gegensatz zum Record-and-Replay-Modell in älteren Frameworks) folgt, und sie benutzt sehr stark die Features von .NET 3.5 und 4, wie Lambdas und Extension Methods. Sie müssen sich wohl dabei fühlen, Lambdas zu verwenden, was auch für den Rest der Frameworks auf dieser Liste gilt.

Es ist recht einfach zu erlernen. Das Einzige, über das ich meckern kann, ist, dass das Wort *Mock* überall in der API verstreut ist, was sie etwas konfus macht. Ich würde gerne zumindest eine Unterscheidung zwischen dem Erzeugen von Stubs und Mocks sehen oder stattdessen das Wort *Fake* bevorzugen, um diese Verwirrung zu vermeiden.

Mehr über Moq erfahren Sie auf der Seite `http://code.google.com/p/moq/`, und installieren können Sie es als NuGet-Paket.

A.1.2 Rhino Mocks

Rhino Mocks ist ein weitverbreitetes Open-Source-Framework für Mocks und Stubs. Obwohl ich es in der vorherigen Ausgabe dieses Buches empfohlen habe, mache ich das nun nicht mehr. Seine Entwicklung wurde eingestellt und es gibt bessere, leichtere, einfachere und besser entworfene Frameworks auf dem Markt. Wenn Sie eine Wahl haben, dann benutzen Sie es nicht. Ayende, sein Schöpfer, erwähnte auf Twitter, dass er nicht mehr daran arbeitet.

Sie finden Rhino Mocks auf der Seite `http://ayende.com/projects/rhino-mocks.aspx`.

A.1.3 Typemock Isolator

Typemock Isolator ist ein kommerzielles *uneingeschränktes* (es kann alles fälschen, siehe Kapitel 6) Isolation-Framework, das versucht, die Begriffe *Mock* und *Stub* aus seinem Vokabular zu entfernen und durch eine einfachere und prägnantere API zu ersetzen.

Isolator unterscheidet sich von den meisten anderen Frameworks darin, dass es Ihnen erlaubt, Komponenten von ihren Abhängigkeiten zu isolieren, und zwar unabhängig davon, wie das Design des Systems ist (gleichwohl unterstützt es all die Features, die auch die anderen Frameworks haben). Dies macht es zu einem idealen Tool für diejenigen, die neu in das Unit Testing einsteigen und sich zum Lernen von Design und Testbarkeit einen inkrementellen Ansatz wünschen. Weil es Sie nicht dazu zwingt, für die Testbarkeit zu planen, können Sie zunächst lernen, die Tests korrekt zu schreiben, und dann damit weitermachen, Ihre Designfähigkeiten zu verbessern, ohne diese zwei Dinge zu vermischen. Es ist auch das teuerste aus dem uneingeschränkten Haufen, was es aber ausgleicht durch seine Benutzerfreundlichkeit und seine Fähigkeiten im Hinblick auf den Legacy-Code.

Typemock Isolator gibt es in zwei Geschmacksrichtungen: eine eingeschränkte Basic-Edition, die kostenlos ist und alle Begrenzungen eines eingeschränkten Frameworks mit sich bringt (nichts Statisches, nur Virtuelles usw.), und eine uneingeschränkte, kostenpflichtige Edition, die fast alles fälschen kann.

Anmerkung

Offenlegung: Ich habe zwischen 2008 und 2010 für Typemock gearbeitet.

Sie finden Typemock Isolator auf der Seite `www.typemock.com`.

A.1.4 JustMock

JustMock von Telerik ist ein relativ neues Isolation-Framework und eine offensichtliche Konkurrenz zu Typemock Isolator. Das API-Design der beiden Frameworks ist so ähnlich, dass es zumindest bei den grundlegenden Dingen recht einfach sein sollte, zwischen ihnen zu wechseln. Ähnlich wie Typemock hat JustMock zwei Geschmacksrichtungen: eine eingeschränkte Edition und eine uneingeschränkte, kostenpflichtige Edition, die fast alles fälschen kann.

Die APIs sind ein wenig spröde und derzeit unterstützt es, soweit ich es ausprobieren konnte, keine rekursiven Fakes – also die Fähigkeit, von einem Fake ein gefälschtes Objekt zurückgeben zu lassen, das wiederum ein gefälschtes Objekt zurückgibt, ohne die Sache explizit spezifizieren zu müssen. Greifen Sie zu auf `http://www.telerik.com/products/mocking.aspx`.

A.1.5 Microsoft Fakes (Moles)

Microsoft Fakes hat als Projekt in der Microsoft Forschung als Antwort auf die Frage begonnen: »Wie können wir das Dateisystem und andere Dinge wie SharePoint fälschen, ohne eine Firma wie Typemock aufkaufen zu müssen?« Dabei kam ein Framework namens Moles heraus. Später entwickelte sich Moles zu Microsoft Fakes und ist Bestandteil einiger Versionen von Visual Studio.

MS Fakes ist ein weiteres uneingeschränktes Isolation-Framework, ohne API zur Überprüfung, dass irgendetwas aufgerufen wurde. Im Wesentlichen stellt es Hilfsmittel zur Erzeugung von Stubs bereit. Wenn Sie überprüfen möchten, dass ein bestimmtes Objekt aufgerufen wurde, dann *können* Sie das tun, aber der Code würde ziemlich chaotisch aussehen.

Wie die beiden vorher genannten uneingeschränkten Frameworks erlaubt Ihnen MS Fakes die Erzeugung von zwei Arten von gefälschten Objekten: Entweder erzeugen Sie uneingeschränkte Klassen, die von bereits testbarem Code erben und ihn überschreiben, oder Sie verwenden Shims. *Shims* sind uneingeschränkt und *Stubs*, die erzeugten Klassen, sind eingeschränkt. Verwirrt? Ja, ich auch. Einer der Gründe, warum ich außer den besonders Tapferen, die nichts zu verlieren haben, niemandem empfehle, MS Fakes zu benutzen, ist die schreckliche Benutzerfreundlichkeit. Die Anwendung ist einfach verwirrend. Außerdem ist die Wartbarkeit der Tests, die entweder Shims oder Stubs verwenden, fraglich. Die erzeugten Stubs müssen immer wieder neu erzeugt werden, wenn Sie den zu testenden Code ändern, was die Änderung der Tests nach sich zieht, und Code, der Shims verwendet, ist ziemlich lang und schwer zu lesen und daher schwer zu warten. MS Fakes mag ja kostenlos und Bestandteil von Visual Studio sein, aber über die ganze Strecke gesehen wird es Sie eine ganze Menge Geld in Form von Entwicklungsstunden und dem Reparieren und Verstehen Ihrer Tests kosten.

Ein anderer wichtiger Punkt: Die Verwendung von MS Fakes zwingt Sie auch dazu, MSTest als Ihr Test-Framework zu verwenden. Leider haben Sie Pech, wenn Sie ein anderes benutzen möchten.

Wenn Sie ein uneingeschränktes Framework benötigen, um Tests zu schreiben, die länger als ein oder zwei Wochen halten, dann sollten Sie JustMock oder Typemock Isolator wählen.

Mehr zu MS Fakes erfahren Sie auf `http://msdn.microsoft.com/en-us/library/hh549175.aspx`.

A.1.6 NSubstitute

NSubstitute ist ein eingeschränktes Isolation-Framework und Open Source. Seine API ist sehr einfach zu lernen und zu behalten und es hat eine gute Dokumentation. Auch gut: Fehlermeldungen sind sehr detailliert. Bei einem neuen Projekt ist es für mich zusammen mit FakeItEasy die erste Wahl eines eingeschränkten Frameworks.

Mehr zu NSubstitute erfahren Sie auf `http://nsubstitute.github.com/` und installieren können Sie es als NuGet-Paket.

A.1.7 FakeItEasy

FakeItEasy hat nicht nur einen großen Namen, sondern auch eine sehr nette API. Es ist zusammen mit NSubstitute mein aktueller Favorit für eingeschränkte Frameworks, aber seine Dokumentation ist nicht so gut wie die von NSub. Was mir an seiner API am meisten gefällt, ist, dass alles, was Sie durchführen wollen, mit dem Buchstaben `A` beginnt, wie zum Beispiel:

```
var foo = A.Fake<IFoo>();
A.CallTo(() => foo.Bar()).MustHaveHappened();
```

Mehr zu FakeItEasy erfahren Sie unter `https://github.com/FakeItEasy/FakeItEasy/wiki` und Sie können es als NuGet-Paket installieren.

A.1.8 Foq

Foq wurde von F#-Entwicklern aus der Notwendigkeit heraus erschaffen, Fakes in einer Weise zu erzeugen, die in F# lesbar und nutzbar sind. Es ist ein eingeschränktes Isolation-Framework, das in der Lage ist, Fakes für abstrakte Klassen und Interfaces zu erzeugen. Ich selbst habe es nie verwendet, da ich nie mit F# gearbeitet habe, aber es scheint auf seinem Gebiet die einzige vernünftige Lösung zu sein. Mehr zu Foq erfahren SIe auf `https://foq.codeplex.com/` und Sie können es als NuGet-Paket installieren.

A.1.9 Isolator++

Isolator++ wurde von Typemock als uneingeschränktes Isolation-Framework für C++ entworfen. Es kann statische Methoden, private Methoden und Weiteres in Legacy-C++-Code fälschen. Es ist ein weiteres kommerzielles Produkt und es scheint das Einzige mit diesen Fähigkeiten auf seinem Gebiet zu sein. Mehr dazu erfahren Sie auf `www.typemock.com/what-is-isolator-pp`.

A.2 Test-Frameworks

Test-Frameworks sind aus zwei Arten von Funktionalität zusammengesetzt:

- Test-Runner führen die Tests, die Sie schreiben, aus, geben Resultate zurück und erlauben Ihnen, zu sehen, was schiefgelaufen ist.
- Test-APIs umfassen die Attribute oder Klassen, die Sie zum Ableiten und Überprüfen benötigen.

Lassen Sie sie uns der Reihe nach anschauen.

Visual-Studio-Test-Runner:

- MS Test Runner von Visual Studio
- TestDriven.NET
- ReSharper
- NUnit
- DevExpress
- TypemockIsolator
- NCrunch
- ContinuousTests (Mighty Moose)

Test-und Assertion-APIs:

- NUnit.Framework
- Microsoft.VisualStudio.TestPlatform.UnitTestFramework
- Microsoft.VisualStudio.TestTools.UnitTesting
- FluentAssertions
- Shouldly
- SharpTestEx
- AutoFixture

A.2.1 Mighty Moose (auch bekannt als ContinuousTests) Continuous Runner

Ein zuvor kommerzielles Produkt verwandelte sich in ein kostenloses. Ähnlich wie NCrunch hat sich *Mighty Moose* dem kontinuierlichen Feedback von Tests und der Testabdeckung verschrieben.

- Es lässt die Tests in einem Hintergrund-Thread laufen.
- Die Tests werden automatisch ausgeführt, während Sie den Code ändern, speichern und kompilieren.
- Es hat einen schlauen Algorithmus, um herauszufinden, welche Tests in Abhängigkeit von welchen Code-Änderungen ausgeführt werden müssen.

Unglücklicherweise sieht es so aus, als wäre die Entwicklung für dieses Tool eingestellt worden. Mehr dazu erfahren Sie auf `http://continuoustests.com/`.

A.2.2 NCrunch Continuous Runner

NCrunch Continuous Runner ist ein kommerzielles Tool und hat sich dem kontinuierlichen Feedback von Tests und der Testabdeckung verschrieben. Als es relativ neu war, hat es mich mit einer Reihe netter Eigenschaften überzeugt (ich habe eine Lizenz gekauft):

- Es lässt die Tests in einem Hintergrund-Thread laufen.
- Die Tests werden automatisch ausgeführt, während Sie den Code ändern, und Sie müssen ihn nicht einmal speichern.
- Grün-rote Punkte neben den Tests und dem Produktionscode zeigen Ihnen an, ob die Zeile im Produktionscode, an der Sie aktuell arbeiten, von einem Test abgedeckt ist und ob er derzeit fehlschlägt. Es ist bis zum Punkt der Schikane sehr gut konfigurierbar. Denken Sie daran, einfach Esc zu drücken, um zu den Voreinstellungen für alle Tests zu gelangen, wenn der Wizard für ein einfaches Projekt aufgeht.

Mehr dazu erfahren Sie auf `www.ncrunch.net/`.

A.2.3 Typemock Isolator Test Runner

Dieser Test-Runner ist Bestandteil des kommerziellen Isolation-Frameworks namens *Typemock Isolator*.

Diese Erweiterung versucht, bei jeder Kompilierung gleichzeitig die Tests auszuführen und die Abdeckung zu zeigen. Es ist noch im Beta-Stadium und hat ein inkonsistentes Verhalten. Vielleicht wird es eines Tages hilfreicher sein, aber derzeit tendiere ich dazu, es auszuschalten und nur die APIs des Isolation-Frameworks zu benutzen.

Mehr dazu finden Sie auf `http://Typemock.com`.

A.2.4 CodeRush Test Runner

Dieser Test-Runner-Teil eines kommerziellen Tools namens *CodeRush* ist ein bekanntes Plug-in für Visual Studio.

Ähnlich wie bei ReSharper gibt es ein paar nette Argumente für diesen Runner:

- Er ist sehr schön in den Visual-Studio-Code-Editor integriert und zeigt in der Nähe von Tests Symbole an, auf die Sie klicken können, um einzelne Tests auszuführen.
- Er unterstützt die meisten der Test-APIs für .NET.
- Wenn Sie bereits CodeRush benutzen, ist er gut genug.

Ähnlich wie bei ReSharper kann die visuelle Natur der Testresultate für erfahrene TDDler eher hinderlich sein. Die Baumstruktur der laufenden Tests und das standardmäßige Anzeigen aller Resultate, auch der erfolgreichen Tests, verschwendet Zeit, wenn Sie mitten im Fluss von TDD sind. Aber einige mögen es. Ihre Fahrleistung kann unterschiedlich sein.

Sie erfahren mehr auf `https://www.devexpress.com/Products/Viusal_Studio_Add-in/Coding_Assistance/unit_test_runner.xml`.

A.2.5 ReSharper Test Runner

Dieser Test-Runner ist Teil eines kommerziellen Tools namens *ReSharper*, ein bekanntes Plug-in für Visual Studio.

Es gibt ein paar nette Argumente für diesen Runner:

- Er ist sehr schön in den Visual-Studio-Code-Editor integriert und zeigt in der Nähe von Tests Symbole an, auf die Sie klicken können, um einzelne Tests auszuführen.
- Er unterstützt die meisten der Test-APIs für .NET.
- Wenn Sie bereits ReSharper benutzen, ist er gut genug.

Ich betrachte die übermäßig visuelle Natur der Testresultate als Nachteil. Die Baumstruktur der laufenden Tests ist sehr nett und bunt. Aber sie bunt anzumalen und standardmäßig alle Resultate, auch die erfolgreichen Tests, anzuzeigen, verschwendet Zeit, wenn Sie mitten im Fluss von TDD sind. Aber einige mögen es. Ihre Fahrleistung kann unterschiedlich sein.

Sie erfahren mehr auf `http://www.jetbrains.com/resharper/features/unit_testing.html`.

A.2.6 TestDriven.NET Runner

Dies ist ein kommerzieller Test-Runner (für den persönlichen Gebrauch ist er frei). Bis ich angefangen habe, NCrunch zu verwenden, war dies mein persönlicher Lieblings-Test-Runner. Es gibt eine Menge, das man mögen muss:

- Er kann die Tests für die meisten, wenn nicht sogar für alle Test-API-Frameworks für .NET ausführen. Dazu gehören NUnit, MSTest und xUnit.net ebenso wie einige der BDD-API-Frameworks.
- Es ist ein kleines Paket. Es ist eine sehr kleine Installation und ein minimalistisches Interface. Die Ausgabe ist einfach: Sie erscheint im Ausgabefenster von Visual Studio und ein wenig Text steht auf den unteren Seitenleisten von Visual Studio.
- Er ist sehr schnell, einer der schnellsten Test-Runner.
- Er hat eine einzigartige Fähigkeit. Sie können auf *irgendein* Stück Code rechtsklicken und wählen `Test with > Debugger` aus. Sie können dann in jeden Code hineindebuggen (auch in Produktionscode und auch, wenn der gar keinen Test hat). Unter der Haube ruft TD.NET die Methode, in der Sie sich gerade befinden, via Reflection auf und gibt ihr Standardwerte vor, falls Parameter benötigt werden. Das spart in Legacy-Code eine Menge Zeit.

Es wird empfohlen, einen Shortcut an das TD.NET-`ReRunTests`-Kommando in Visual Studio anzuhängen, damit der TDD-Ablauf so geschmeidig wie möglich wird.

A.2.7 NUnit GUI Runner

Der *NUnit GUI Runner* ist Open Source und kostenlos. Dieser Runner ist nicht in Visual Studio integriert, weshalb Sie ihn von Ihrem Desktop ausführen müssen. Daher benutzt ihn fast niemand, wenn die Möglichkeit besteht, eines der hier aufgeführten und in Visual Studio integrierten Tools zu verwenden. Er ist unausgegoren, ungeschliffen und nicht zu empfehlen.

A.2.8 MSTest Runner

Der *MSTest Runner* kommt im Paket mit jeder Version von Visual Studio. In der kostenpflichtigen Version enthält er auch einen Plug-in-Mechanismus, der es Ihnen erlaubt, spezielle Adapter als Visual Studio Extensions zu installieren. Damit erhalten Sie eine Unterstützung zur Ausführung von Tests, die mit anderen APIs, wie etwa NUnit oder xUnit.net, geschrieben wurden.

Ein Punkt zugunsten dieses Frameworks besteht darin, dass es in die Visual-Studio-Team-System-Tool-Suite integriert ist und standardmäßig ein gutes Reporting, eine Abdeckungsanalyse und eine Build-Automatisierung anbietet. Wenn Ihre Firma Team System für automatisierte Builds einsetzt, dann sollten Sie MSTest als Ihren Test-Runner in den nächtlichen und den CI Builds wegen der guten Integrationsmöglichkeiten (z. B. Reporting) ausprobieren.

Zwei Gebiete, auf denen MSTest nicht gut ist, sind Performance und Abhängigkeiten:

- *Abhängigkeiten* – Um Ihre Tests unter Verwendung von `mstest.exe` laufen zu lassen, müssen Sie Visual Studio auf Ihrem Build Server installiert haben. Für manche mag das in Ordnung sein, insbesondere wenn der Ort, an dem Sie kompilieren, auch der ist, an dem Sie Ihre Tests laufen lassen. Aber wenn Sie Ihre Tests in einer relativ sauberen Umgebung laufen lassen wollen, und in bereits kompilierter Form, kann das übertrieben sein und auch problematisch, wenn Sie in der Umgebung eben gerade kein Visual Studio installieren wollen.
- *Langsam* – Unter der Motorhaube machen die Tests in MSTest vor und nach jedem Lauf eine ganze Menge, das Kopieren von Dateien, die Ausführung externer Prozesse, Profiling usw., was MSTest zum gefühlt langsamsten Runner aller, die ich je benutzt habe, macht.

A.2.9 Pex

Pex (Abkürzung für »Program Exploration«) ist ein intelligenter Assistent für den Programmierer. Aus einem parametrisierten Unit Test produziert es automatisch ein traditionelles Unit-Test-Paket mit einer hohen Code-Abdeckung. Zusätzlich macht es dem Programmierer Vorschläge zum Bugfixing.

Mit Pex können Sie spezielle Tests erzeugen, die Parameter beinhalten, und Sie können spezielle Attribute für diese Tests setzen. Die Pex-Engine generiert dann neue Tests, die Sie später als Teil Ihres Testpakets laufen lassen können. Das ist großartig, um Grenzfälle und Randbedingungen zu finden, die von Ihrem Code nicht sauber gehandhabt werden. Sie sollten Pex zusätzlich zu einem regulären Test-Framework wie NUnit oder MbUnit einsetzen.

Sie finden Pex auf der Seite `http://research.microsoft.com/projects/pex/`.

A.3 Test-APIs

Das nächste Bündel von Tools bietet Abstraktionen auf höherem Level und Wrapper für die Basis-Unit-Testing-Frameworks.

A.3.1 MSTest-API – Microsofts Unit-Testing-Framework

Dies erhalten Sie zusammen mit jeder Version von Visual Studio .NET Professional oder höher. Es enthält grundlegende Fähigkeiten, die denen von NUnit ähneln.

Aber verschiedene Probleme machen es zu einem schwächeren Produkt für das Unit Testing als NUnit oder xUnit.net:

- Erweiterbarkeit
- Das Fehlen von `Assert.Throws`

Erweiterbarkeit

Ein großes Problem dieses Frameworks ist, dass es nicht so einfach wie die anderen erweitert werden kann. Obwohl es in der Vergangenheit schon mehrere Online-Diskussionen gab, MSTest über benutzerdefinierte Test-Attribute besser erweiterbar zu machen, sieht es so aus, als ob das Visual-Studio-Team aufgegeben hätte, aus MSTest eine echte Alternative zu NUnit und anderen zu machen.

Stattdessen weist VS 2012 (und VS 2013) einen Mechanismus auf, der es Ihnen erlaubt, NUnit oder jedes andere Test-Framework als Ihr Standard-Test-Framework zu verwenden, wobei der MSTest Runner Ihre NUnit-Tests ausführt. Es gibt bereits Adapter für NUnit und xUnit.net (NUnit Test Adapter oder xUnit.net Runner für Visual Studio 2012 und 2013), wenn Sie nur den MSTest Runner zusammen mit anderen Frameworks verwenden wollen. Unglücklicherweise enthält die freie Express-Version von Visual Studio diesen Mechanismus nicht, was Sie zwingt, das unterlegene MSTest zu verwenden. (Nebenbei bemerkt: Warum zwingt Microsoft Sie, Visual Studio zu kaufen, damit Sie Code entwickeln können, der die MS-Plattform noch dominanter macht?)

Das Fehlen von »Assert.Throws«

Dies ist eine einfache Sache. In MSTest haben Sie ein Attribut `ExpectedException`, aber Sie haben kein `Assert.Throws`, das es Ihnen zu überprüfen erlaubt, ob eine bestimmte Zeile eine Ausnahme geworfen hat. Mehr als sechs Jahre nach dem Anbeginn und vier Jahre nach den meisten Frameworks haben sich die Entwickler dieses Frameworks noch immer nicht darum bemüht, diese wirklich nur zehn Zeilen Code hinzuzufügen, was mich daran zweifeln lässt, ob sie Unit Tests überhaupt ernst nehmen.

A.3.2 MSTest für Metro Apps (Windows Store)

MSTest für Metro Apps ist eine API zum Schreiben von Windows-Store-Apps, die wie MSTest aussieht, aber anscheinend die richtige Idee in Bezug auf Unit Tests umsetzt. Beispielsweise trägt sie ihre eigene Version von `Assert.ThrowsException` zur Schau.

Es sieht so aus, dass Sie gezwungen sind, dieses Framework zum Schreiben von Windows-Store-Apps mit Unit Tests zu verwenden, aber es gibt eine Lösung, wenn Sie verlinkte Projekte verwenden. Weitere Informationen finden Sie hier: `http://stackoverflow.com/questions/12924579/testing-a-windows-8-store-app-with-nunit`.

A.3.3 NUnit API

NUnit ist derzeit der Standard unter den Test-API-Frameworks für Unit-Test-Entwickler in .NET. Es ist Open Source und nahezu allgegenwärtig bei denen, die sich mit dem Unit Testing beschäftigen. In Kapitel 2 diskutiere ich NUnit im Detail. NUnit lässt sich einfach erweitern und hat eine große Benutzerbasis mit zahlreichen Foren. Ich empfehle es jedem, der mit dem Unit Testing in .NET beginnt. Ich nutze es noch heute.

Sie finden NUnit auf der Seite `www.Nunit.org`.

A.3.4 xUnit.net

xUnit.net ist ein Open-Source-Test-API-Framework, das in Zusammenarbeit mit Jim Newkirk, einem der ursprünglichen Autoren von NUnit, entstand. Es ist ein minimalistisches und elegantes Test-Framework, das versucht, zu den Grundlagen zurückzukehren, indem es weniger und nicht mehr Features als andere Frameworks besitzt und indem es verschiedene Namen für seine Attribute unterstützt.

Was ist daran so radikal anders? Es hat keine Aufbau- oder Abbaumethoden, um nur eines zu nennen. Sie müssen den Konstruktor und eine Dispose-Methode der Testklasse verwenden. Ein anderer, großer Unterschied ist, wie einfach es erweitert werden kann.

Weil sich xUnit.net so deutlich von den anderen Frameworks unterscheidet, braucht es eine gewisse Eingewöhnungszeit, wenn man von einem anderen Framework wie NUnit oder MbUnit kommt. Wenn Sie nie zuvor irgendein Test-Framework benutzt haben, dann ist xUnit.net einfach zu begreifen und zu verwenden und es ist robust genug, um es in echten Projekten einzusetzen.

Mehr Informationen und den Download finden Sie auf der Seite `www.codeplex.com/xunit`.

A.3.5 Fluent Assertions Helper API

Die *Fluent Assertions Helper API* ist eine neue Züchtung einer Test-API. Sie ist eine niedliche Bibliothek, die nur zu einem einzigen Zweck entworfen wurde: damit Sie auf alles ein Assert setzen können, egal welche Test-API Sie verwenden. Sie können mit ihr beispielsweise eine `Assert.Thorws()`-ähnliche Funktionalität in MSTest erreichen.

Mehr Informationen finden Sie auf `http://www.fluentassertions.com/`.

A.3.6 Shouldly Helper API

Die *Shouldly Helper API* ist Fluent Assertions sehr ähnlich, nur kleiner. Sie ist ebenfalls nur zu einem einzigen Zweck entworfen wurde: damit Sie auf alles ein Assert setzen können, egal welche Test-API Sie verwenden. Mehr Informationen finden Sie auf `http://shouldly.github.com`.

A.3.7 SharpTestsEx Helper API

Ähnlich wie Fluent Assertions ist die *SharpTestsEx Helper API* nur zu einem einzigen Zweck entworfen wurde: damit Sie auf alles ein Assert setzen können, egal welche Test-API Sie verwenden. Mehr Informationen finden Sie auf `http://sharptestex.codeplex.com`.

A.3.8 AutoFixture Helper API

Die *AutoFixture Helper API* ist keine Assertion API. AutoFixture wurde entworfen, um in einem Test das Erzeugen von solchen Objekten zu erleichtern, um die Sie sich nicht kümmern wollen. Etwa, wenn Sie irgendeine Zahl oder irgendeinen String benötigen. Stellen Sie es sich als eine clevere Fabrik vor, die Objekte und Eingabewerte in Ihre Tests injizieren kann.

Was ich daran am interessantesten finde, ist die Möglichkeit, die Instanz einer zu testenden Klasse zu erzeugen, ohne zu wissen, wie die Signatur des Konstruktors aussieht. Das kann langfristig meine Tests besser wartbar machen. Allerdings ist dieser Grund alleine noch nicht ausreichend, damit ich es benutze, denn das Gleiche kann ich mit einer kleinen Fabrikmethode in meinen Tests erreichen.

Auch erschreckt es mich ein wenig, Zufallswerte in meine Tests injizieren zu lassen, denn das bedeutet, dass ich jedes Mal einen anderen Test laufen lasse. Es macht auch meine Asserts komplizierter, denn ich muss bedenken, dass meine erwartete Ausgabe auf zufällig injizierten Parametern basiert. Das kann dazu führen, die Logik des Produktionscodes in meinen Tests zu wiederholen.

Weitere Informationen können Sie hier finden: `https://github.com/AutoFixture/AutoFixture`.

A.4 IoC-Container

IoC-Container können verwendet werden, um die Architekturqualitäten eines objektorientierten Systems zu verbessern, indem die mechanischen Kosten der guten Designmethoden verringert werden (wie die Verwendung von Konstruktorparametern, die Verwaltung der Objektlebensdauer usw.).

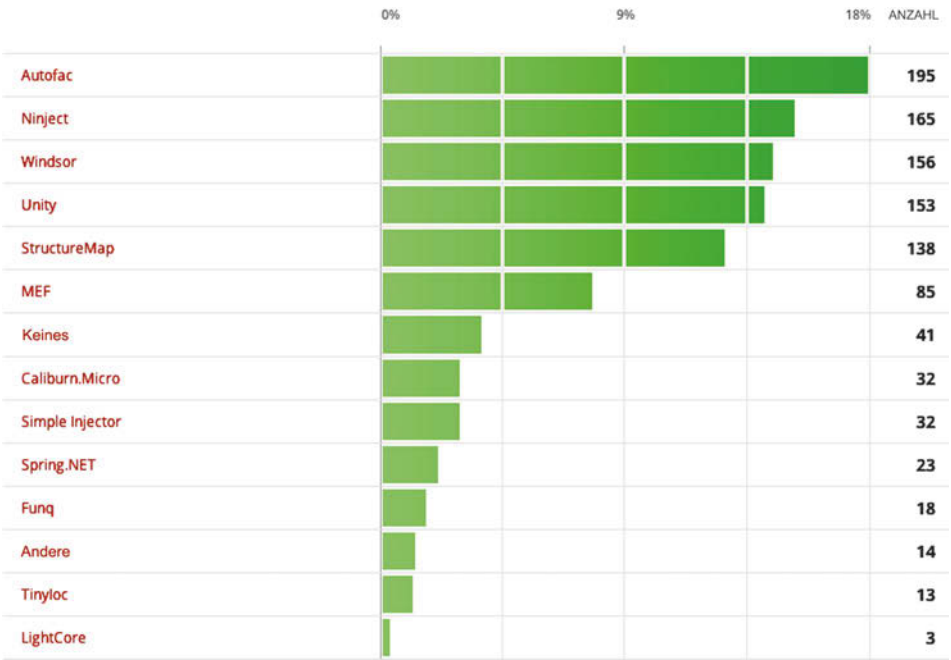
Container können eine schwächere Kopplung zwischen Klassen und ihren Abhängigkeiten ermöglichen, die Testbarkeit einer Klassenstruktur verbessern und einen generischen Flexibilitätsmechanismus bereitstellen. Wenn sie mit Umsicht eingesetzt werden, können Container die Möglichkeiten zur Wiederverwendung von Code ganz erheblich verbessern, indem sie die direkte Kopplung zwischen Klassen und Konfigurationsmechanismen verringern (etwa durch die Verwendung von Interfaces).

Es gibt *eine Menge* davon in der .NET-Welt. Sie sind mannigfaltig und interessant anzuschauen. In puncto Performance, falls Sie das sehr interessiert, gibt es einen guten Vergleich auf `http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison`. Ich selbst hatte allerdings nie den Eindruck, dass IoC-Container der eigentliche Grund für Performance-Probleme waren, und falls das je der Fall sein sollte, wäre ich gerne dabei.

Jedenfalls gibt es viele davon, aber wir schauen uns die folgenden an, die häufig in der Community genutzt werden.

Ich habe die Tools, die ich beschreibe, anhand einer Umfrage zur Nutzung unter meinen Blog-Lesern im März 2013 durchgeführt. Hier ist die Spitzengruppe der am häufigsten genutzten Tools:

- Autofac (Auto Factory)
- Ninject
- Castle Windsor
- Microsoft Unity
- Structure Map
- Microsoft Managed Extensibility Framework



Lassen Sie uns kurz jedes dieser Frameworks betrachten.

A.4.1 Autofac

Autofac war eines der ersten Tools, das einen neuen Zugang zu IoC in .NET angeboten hat, der gut zur C#-3- und -4-Syntax passt. In Bezug auf die APIs ist der Ansatz ziemlich minimalistisch. Die API unterscheidet sich drastisch von der anderer Frameworks und erfordert eine gewisse Eingewöhnungszeit. Es setzt auch .NET 3.5 voraus und Sie benötigen gute Kenntnisse der Lambda-Syntax. Es ist schwierig, Autofac zu erklären, daher müssen Sie es ausprobieren, um die Unterschiede zu sehen. Ich empfehle es denjenigen, die bereits Erfahrung mit anderen DI-Frameworks besitzen.

Sie finden es auf der Seite `https://github.com/autofac/Autofac`.

A.4.2 Ninject

Ninject hat ebenfalls eine einfache Syntax und eine gute Benutzerfreundlichkeit. Es gibt nichts weiter dazu zu sagen, außer dass ich Ihnen sehr ans Herz lege, einen Blick darauf zu werfen.

Mehr zu Ninject finden Sie auf `http://ninject.org/`.

A.4.3 Castle Windsor

Castle ist ein großes Open-Source-Projekt, das eine Menge Gebiete abdeckt. *Windsor* ist eines von diesen Gebieten und bietet eine reife und mächtige Implementierung eines DI-Containers an.

Castle Windsor enthält die meisten der Features, die Sie sich jemals wünschen werden – und mehr –, aber es hat eine relativ steile Lernkurve, eben wegen all der Features.

Mehr zu Castle Windsor können Sie auf der Seite `http://docs.castleproject.org/Windsor.MainPage.ashx` erfahren.

A.4.4 Microsoft Unity

Unity ist ein Nachzügler auf dem Gebiet der DI-Container, aber es bietet einen einfachen und minimalen Zugang, der von Anfängern leicht erlernt und verwendet werden kann. Fortgeschrittene Anwender mögen es für mangelhaft halten, aber es erfüllt meine 80-20-Regel: Es bietet 80 Prozent der Features an, die Sie die meiste Zeit benötigen.

Unity ist ein Open-Source-Projekt von Microsoft mit einer guten Dokumentation. Ich empfehle es als einen Ausgangspunkt für die Arbeit mit Containern.

Sie finden Unity auf der Seite `www.codeplex.com/unity`.

A.4.5 StructureMap

StructureMap ist ein Open-Source-Container-Framework, das von Jeremy D. Miller geschrieben wurde. Seine API ist sehr flüssig und versucht, so gut wie möglich die natürliche Sprache und generische Konstrukte nachzubilden.

Es hapert an der aktuellen Dokumentation, aber es enthält einige mächtige Features, wie etwa einen eingebauten Automocking-Container (ein Container, der auf Anfrage vom Test automatisch Stubs erzeugen kann), ein mächtiges Lebenszeitmanagement, eine XML-freie Konfiguration, die Integration von ASP.NET und anderes mehr.

Sie finden StructureMap auf der Seite `http://structuremap.net`.

A.4.6 Microsoft Managed Extensibility Framework

Das *Managed Extensibility Framework* (MEF) ist nicht wirklich ein Container, aber es fällt in die gleiche allgemeine Kategorie der Serviceanbieter, um Klassen in Ihrem Code zu instanziieren. Es wurde entworfen, um viel mehr als nur ein Container zu sein: Es ist ein vollständiges Plug-in-Modell für kleine und große Applikationen. MEF enthält ein leichtgewichtiges IoC-Container-Framework, damit Sie Abhängigkeiten einfach an verschiedenen Stellen in Ihrem Code über die Verwendung spezieller Attribute injizieren können.

MEF bringt eine gewisse Lernkurve mit sich und ich würde nicht empfehlen, es strikt als einen IoC-Container einzusetzen. Wenn Sie es in Ihrer Applikation aus Gründen der Erweiterbarkeit einsetzen, dann können Sie es auch als einen DI-Container verwenden.

Sie finden das MEF auf der Seite `http://mef.codeplex.com/`.

A.5 Datenbanktests

Für viele Anfänger ist es eine brennende Frage, wie eine Datenbank getestet werden sollte. Viele Fragen tauchen da auf, wie etwa: »Soll ich die Datenbank in meinen Tests durch einen Stub ersetzen?« Dieser Abschnitt gibt Ihnen ein paar Leitlinien an die Hand.

Lassen Sie uns als Erstes Integrationstests für die Datenbank untersuchen.

A.5.1 Verwenden Sie Integrationstests für Ihre Datenschicht

Wie sollen Sie Ihre Datenschicht testen? Sollten Sie das Interface der Datenbank wegabstrahieren? Sollten Sie die echte Datenbank benutzen?

Ich schreibe gewöhnlich Integrationstests für die Datenschicht (der Teil der Applikationsstruktur, der direkt mit der Datenbank kommuniziert) in meinen Anwendungen, denn die Datenlogik ist fast immer auf die Applikationslogik und die Datenbank selbst verteilt (Trigger, Sicherheitsrollen, referenzielle Integrität usw.). Außer wenn man die Datenbanklogik komplett isoliert testen kann (und ich habe zu diesem Zweck kein wirklich gutes Framework gefunden), besteht der einzige Weg, um in den Tests sicherzustellen, dass sie funktioniert, darin, das Testen der Datenschichtlogik an die echte Datenbank zu koppeln.

Das gemeinsame Testen der Datenschicht und der Datenbank verringert die Überraschungen, die es später im Projekt geben wird. Aber das Testen mit der Datenbank birgt seine Probleme, vor allem, dass Sie gegen einen Zustand testen, der von vielen Tests geteilt wird. Wenn Sie in einem Test eine Zeile in die Datenbank einfügen, kann der nächste Test diese Zeile ebenfalls sehen.

Was Sie brauchen, ist eine Möglichkeit, die Änderungen, die Sie in der Datenbank vornehmen, anschließend wieder zurückzufahren, und glücklicherweise gibt es eine einfache Möglichkeit dafür im .NET-Framework.

A.5.2 Verwenden Sie TransactionScope für ein Rollback der Daten

Die Klasse `TransactionScope` ist clever genug, um sowohl mit sehr komplizierten als auch mit verschachtelten Transaktionen umzugehen, bei denen der zu testende Code Commits auf seine eigenen lokalen Transaktionen aufruft.

Hier kommt ein einfaches Stück Code, das zeigt, wie einfach es ist, Ihren Tests eine Rollback-Fähigkeit hinzuzufügen:

```
[TestFixture]
public class TransactionScopeTests
{
    private TransactionScope trans = null;
    [SetUp]
    public void SetUp()
```

```
    {
        trans = new TransactionScope(TransactionScopeOption.Required);
    }
    [TearDown]
    public void TearDown()
    {
        trans.Dispose();
    }

    [Test]
    public void TestServicedSameTransaction()
    {
        MySimpleClass c = new MySimpleClass();

        long id = c.InsertCategoryStandard("whatever");
        long id2 = c.InsertCategoryStandard("whatever");
        Console.WriteLine("Got id of " + id);
        Console.WriteLine("Got id of " + id2);
        Assert.AreNotEqual(id, id2);
    }
}
```

Zunächst setzen Sie in der Methode `SetUp()` einen Transaktionsbereich auf und beenden ihn in `TearDown()` wieder.

Wenn Sie kein `Commit` auf der Ebene der Testklasse ausführen, machen Sie im Wesentlichen alle Änderungen an der Datenbank über ein Rollback rückgängig, denn `Dispose()` initiiert ein Datenbank-Rollback, wenn zuvor kein `Commit` aufgerufen wurde.

Manche halten es auch für eine gute Option, die Tests mit einer In-Memory-Datenbank durchzuführen. Meine Ansichten dazu sind gemischt. Auf der einen Seite ist es näher an der Realität, insofern Sie dann auch die Datenbanklogik testen. Wenn Sie auf der anderen Seite in Ihrer Applikation eine andere Datenbank-Engine mit anderen Features einsetzen, ist die Wahrscheinlichkeit hoch, dass während der Tests mit der In-Memory-Datenbank einige Dinge erfolgreich sind oder fehlschlagen und sich dann aber in der Produktionsumgebung anders verhalten. Ich arbeite lieber mit dem, was so nahe wie möglich an der echten Sache ist. Gewöhnlich heißt das, die gleiche Datenbank-Engine zu verwenden.

Wenn in die In-Memory-Datenbank-Engine die gleichen Fähigkeiten und die gleiche Logik eingebettet sind, dann kann das eine sehr gute Idee sein.

A.6 Webtests

»Wie teste ich meine Webseiten?« ist eine andere häufig gestellte Frage. Dies sind einige Tools, die Ihnen dabei helfen können:

- Ivonna
- Team System Web Test

- Watir
- Selenium

Nachfolgend gebe ich Ihnen eine kurze Beschreibung dieser Tools.

A.6.1 Ivonna

Ivonna ist ein Unit-Testing-Hilfs-Framework, das die Notwendigkeit wegabstrahiert, ASP.NET-bezogene Tests mit echten HTTP-Sessions und -Seiten ablaufen zu lassen. Hinter den Kulissen führt es einige mächtige Dinge aus, indem es etwa die Seiten, die Sie testen wollen, so kompiliert, dass Sie anschließend die beinhalteten Controls testen können, ohne eine Browsersession zu benötigen, oder indem es das komplette HTTP-Runtime-Modell imitiert.

Sie schreiben den Code in Ihren Unit Tests genauso, wie Sie auch andere In-Memory-Objekte testen würden. Sie brauchen keinen Webserver und anderen Schnickschnack.

Ivonna wird in Partnerschaft mit Typemock entwickelt und läuft als ein Addon des Typemock Isolator Frameworks. Sie erhalten Ivonna unter `http://ivonna.biz/`.

A.6.2 Team System Web Test

Visual Studio *Team* bzw. *Ultimate* beinhalten die mächtige Fähigkeit, Webanfragen für Seiten aufzunehmen und abzuspielen und verschiedene Dinge während der Ausführung zu verifizieren. Das ist genau genommen Integration Testing, aber es ist wirklich mächtig. Die neuesten Versionen unterstützen auch die Aufnahme von Ajax-Aktionen auf der Seite und machen die Handhabung in vielerlei Hinsicht einfacher.

Mehr dazu erfahren Sie unter `http://msdn.microsoft.com/en-us/teamsystem/default.aspx`.

A.6.3 Watir

Watir (wird wie das englische Wort »water« ausgesprochen) steht für »Web Application Testing in Ruby«. Es ist Open Source und erlaubt Ihnen, Skripts für die Browseraktionen in der Programmiersprache Ruby zu schreiben. Viele Anhänger von Ruby schwören darauf, aber es verlangt von Ihnen, eine komplett neue Sprache zu lernen. Eine Menge .NET-Projekte nutzen es erfolgreich, es ist also keine große Sache.

Sie erhalten Watir unter `http://watir.com/`.

A.6.4 Selenium WebDriver

Selenium ist ein Paket von Tools, die entworfen wurden, um das Testen von Webapplikationen über viele Plattformen hinweg zu automatisieren. Es ist älter als all die anderen Frameworks in dieser Liste und besitzt auch API-Wrapper für .NET. WebDriver ist eine Erweiterung, die zu vielen verschiedenen Arten von Browsern passt, einschließlich der mobilen. Es ist sehr mächtig.

Selenium ist ein weitverbreitetes Integration-Testing-Framework. Es eignet sich gut, um damit zu beginnen. Aber seien Sie gewarnt: Es hat viele Features und seine Lernkurve ist steil.

Sie finden es unter `http://docs.seleniumhq.org/projects/webdriver/`.

A.6.5 Coypu

Coypu ist ein .NET Wrapper, der auf Selenium und anderen Web-bezogenen Test-Tools aufsetzt. Es ist im Augenblick noch recht neu, hat aber womöglich ein großes Potenzial. Sie sollten vielleicht einen Blick darauf werfen.

Mehr dazu finden Sie hier: `https://github.com/featurist/coypu`.

A.6.6 Capybara

Capybara ist ein Tool, das auf Ruby basiert und den Browser automatisiert. Zur Automatisierung des Browsers können Sie damit die RSpec-(BDD-Stil-)API nutzen, die viele sehr angenehm zu lesen finden.

Selenium ist reifer, aber Capybara ist einladender und entwickelt sich schnell.

Ich nutze es immer für mein Ruby-Zeug.

Mehr dazu finden Sie hier: `https://github.com/jnicklas/capybara`.

A.6.7 JavaScript-Tests

Es gibt eine Reihe von Tools, die einen Blick wert sind, wenn Sie vorhaben, Unit Tests oder Akzeptanztests für JavaScript-Code zu schreiben. Beachten Sie, dass viele davon voraussetzen, dass Sie Node.JS auf Ihrer Maschine installieren, was heutzutage ein Kinderspiel ist. Gehen Sie einfach auf diese Seite: `http://nodejs.org/download/`.

Hier ist eine unvollständige Liste von Tools, die Sie sich anschauen können:

- *JSCover* – Verwenden Sie es, um die Codeabdeckung von JavaScript durch Tests zu überprüfen: `http://tntim96.github.io/JSCover/`.
- *Jasmin* – Ein sehr bekanntes Framework im BDD-Stil, das ich verwendet habe. Ich kann es empfehlen: `http://jasmine.github.io/`.
- *Sinon.JS* – Erzeugt Fakes in JS: `http://sinonjs.org/`.
- *CasperJS + PhantomJS* – Verwenden Sie es zum kopflosen Testen Ihres Browser-JavaScripts. Richtig – kein echter Browser muss vorhanden sein (verwendet intern `node.js`): `http://casperjs.org/`.
- *Mocha* – Auch sehr bekannt und wird in vielen Projekten verwendet: `http://visionmedia.github.io/mocha/`.
- *QUnit* – Ein bisschen in die Jahre gekommen, aber immer noch ein gutes Test-Framework: `http://qunitjs.com/`.
- *Buster.JS* – Ein sehr neues Framework: `http://docs.busterjs.org/en/latest/`.
- *Vow.js* – Ein Nachwuchs-Framework: `https://github.com/flatiron/vows`.

A.7 UI-Tests (Desktop)

UI-Testing ist immer eine schwierige Aufgabe. Ich halte nicht sehr viel davon, Unit Tests oder Integrationstests für UIs zu schreiben, weil der Ertrag im Vergleich zu der Menge an Zeit, die Sie für das Schreiben investieren, gering ist. Für meinen Geschmack ändern sich UIs zu sehr, um in einer konsistenten Weise getestet werden zu können. Darum versuche

ich gewöhnlich, all die Logik aus dem UI in eine tiefere Schicht zu separieren, die ich dann getrennt mit Standard-Unit-Testing-Methoden behandeln kann.

In diesem Bereich gibt es keine Tools, die ich von Herzen empfehlen kann (derentwegen Sie Ihre Tastatur nicht nach drei Monaten zertrümmern werden).

A.8 Thread-bezogene Tests

Threads waren immer schon der Fluch des Unit Testings. Sie sind einfach untestbar. Daher tauchen neue Frameworks auf, die Sie die Thread-bezogene Logik testen lassen (Deadlocks, Race Conditions usw.), so wie diese:

- Microsoft CHESS
- Osherove.ThreadTester

Ich werde zu jedem Tool eine kurze Zusammenfassung geben.

A.8.1 Microsoft CHESS

Microsoft *CHESS* war ein aufstrebendes Tool von Microsoft, das nun irgendwie Open Source auf Codeplex.com ist. CHESS versucht Thread-bezogene Probleme (Deadlocks, Hänger, Livelocks und anderes) in Ihrem Code zu finden, indem es alle relevanten Permutationen von Threads im Code laufen lässt. Diese Tests werden als einfache Unit Tests geschrieben.

Sie finden CHESS unter `http://chesstool.codeplex.com/`.

A.8.2 Osherove.ThreadTester

Dies ist ein kleines Open-Source-Framework, das ich vor einiger Zeit entwickelt habe. Es erlaubt Ihnen, während eines Tests mehrere Threads auszuführen, um zu sehen, ob irgendetwas Merkwürdiges mit Ihrem Code geschieht (beispielsweise Deadlocks). Seine Features sind nicht komplett, aber es ist ein guter Anlauf zu einem multithreaded Test (statt eines Tests für multithreaded Code).

Sie finden es in meinem Blog auf der Seite `http://osherove.com/blog/2007/6/22/multi-threaded-unit-tests-with-osherovethreadtester.html`.

A.9 Akzeptanztests

Akzeptanztests verbessern die Zusammenarbeit zwischen den Kunden und den Entwicklern in der Softwareentwicklung. Sie ermöglichen es den Kunden, den Testern und den Programmierern, zu lernen, was die Software tun soll und sie vergleichen das automatisch mit dem, was die Software tatsächlich tut. Sie vergleichen die Kundenerwartungen mit den tatsächlichen Resultaten. Es ist eine großartige Möglichkeit, früh in der Entwicklungsphase bei komplizierten Problemen zusammenzuarbeiten (und sie zu lösen).

Unglücklicherweise gibt es nur wenige Frameworks für automatische Akzeptanztests und nur eines, das derzeit wirklich funktioniert! Ich hoffe, dass sich dies bald ändern wird. Hier sind die Tools, auf die wir einen Blick werfen werden:

- FitNesse
- SpecFlow
- Cucumber
- TickSpec

Gehen wir sie im Einzelnen durch.

A.9.1 FitNesse

FitNesse ist ein leichtgewichtiges Open-Source-Framework, das es Softwareteams angeblich leicht macht, Akzeptanztests – Webseiten, die einfache Tabellen mit Eingaben und erwarteten Ausgaben enthalten – zu definieren, die Tests auszuführen und die Resultate zu betrachten.

FitNesse ist recht fehlerhaft, aber es wurde schon häufig und mit unterschiedlichem Erfolg eingesetzt. Ich selbst habe es nicht so richtig gut ans Laufen gebracht.

Mehr zu FitNesse finden Sie auf der Seite `http://www.fitnesse.org/`.

A.9.2 SpecFlow

SpecFlow versucht, der .NET-Welt zu geben, was Cucumber der Ruby-Welt gab: ein Tool, das es Ihnen erlaubt, die Spezifikations-Sprache in eine einfache Textdatei zu schreiben, mit deren Hilfe Sie dann mit Ihren Kunden und der QS-Abteilung zusammenarbeiten können. Es leistet dabei ziemlich gute Arbeit.

Mehr finden Sie auf: `http://www.specflow.org/`.

A.9.3 Cucumber

Cucumber ist ein auf Ruby basierendes Tool, das es Ihnen ermöglicht, Ihre Spezifikationen in einer speziellen Sprache namens Gherkin (ja, ich stimme zu) zu schreiben. Das sind einfache Textdateien, und dann müssen Sie noch speziellen Verbindungs-Code schreiben, um wirklichen Code ausführen zu lassen, der mit Ihrem Anwendungs-Code arbeitet. Klingt kompliziert, ist es aber nicht.

Aber warum taucht es hier auf, wenn es doch ein Ruby-Tool ist? Ich führe es hier auf, weil es eine ganze Reihe von Tools in der .NET-Welt inspiriert hat, von denen aktuell wohl nur ein einziges überlebt – SpecFlow.

Aber es gibt eine Möglichkeit, Cucumber unter .NET laufen zu lassen, wenn Sie IronRuby verwenden – eine Sprache, die von Microsoft aufgegeben und über die Mauer in die Open-Source-Welt geworfen wurde, um nie mehr wieder aufzutauchen. (Großartige Leistung!)

Jedenfalls ist Cucumber wichtig genug, um es auf dem Radarschirm zu haben, egal ob Sie planen, es einzusetzen oder nicht. Es wird Ihnen helfen zu verstehen, warum manche Dinge in .NET versuchen, das Gleiche zu machen.

Auch ist es die Basis für die Sprache Gherkin, die andere Tools jetzt und in Zukunft zu implementieren versuchen werden. Mehr dazu auf: `http://cukes.info/`.

A.9.4 TickSpec

Wenn Sie F# verwenden, ist *TickSpec* das Richtige für Sie. Ich selbst habe es noch nicht verwendet, da ich F# noch nicht eingesetzt habe, aber in Bezug auf Akzeptanztests und BDD-Stil ist es als ein ähnliches Framework gedacht wie die bereits erwähnten. Ich habe auch noch nicht von anderen gehört, die es verwenden, aber das mag daran liegen, dass ich mich nicht so sehr in F#-Kreisen bewege. Mehr dazu finden Sie auf: `https://tickspec.codeplex.com/`.

A.10 API-Frameworks im BDD-Stil

In den letzten paar Jahren sind eine ganze Reihe von Frameworks entstanden, die ein anderes Tool aus der Ruby-Welt namens *RSpec* imitieren. Dieses Tool hat die Idee eingeführt, dass Unit Testing vielleicht keine so großartige Namenskonvention ist, und indem man zu BDD wechselt, kann man die Dinge besser lesbar machen und vielleicht sogar eher mit seinen Kunden darüber sprechen.

Meiner Meinung nach negiert bereits die Idee, einfach diese Frameworks als unterschiedliche APIs zu implementieren, mit denen man Unit Tests oder Integrationstests schreibt, fast alle Möglichkeiten, mit den Kunden mehr (als zuvor) darüber zu reden. Denn es ist unwahrscheinlich, dass sie Ihren Code wirklich lesen oder ändern wollen. Ich glaube, dass die Akzeptanztest-Frameworks aus dem vorhergehenden Abschnitt besser zu dieser Haltung passen.

Damit bleiben nur noch Programmierer übrig, die versuchen, diese APIs zu verwenden.

Da sich diese APIs von der Sprache Cucumber im BDD-Stil inspirieren lassen, scheinen sie in manchen Fällen besser lesbar zu sein, aber meiner Meinung nach nicht in den einfachen Fällen, die mehr von den simplen Tests im Assert-Stil profitieren. Ihre Kilometerleistung mag da unterschiedlich sein.

Hier kommen einige der besser bekannten Frameworks im BDD-Stil. Ich gehe nicht näher auf sie ein, da ich selbst keines von ihnen in einem echten Projekt über einen längeren Zeitraum benutzt habe:

- *NSpec* ist das älteste und scheint recht gut in Form zu sein. Mehr dazu auf `http://nspec.org/`.
- *StoryQ* ist ein anderer Oldie, aber immer noch toll. Es produziert eine sehr gut lesbare Ausgabe und es besitzt auch ein Tool, das Gherkin Stories in kompilierbaren Test-Code übersetzt. Mehr dazu auf `http://storyq.codeplex.com/`.
- *MSpec* oder Machine.Specifications versucht, mit vielen Lambda-Tricks so nah wie möglich an der Quelle (RSpec) zu bleiben. Man gewöhnt sich daran. Mehr dazu auf `https://github.com/machine/machine.specifications`.
- *TickSpec* folgt der gleichen Idee in einer Implementierung für F#. Mehr dazu auf `http://tickspec.codeplex.com/`.

Stichwortverzeichnis

S

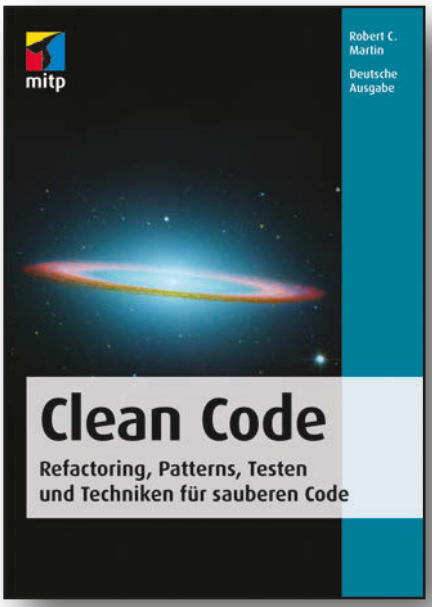
T

U

Robert C. Martin

Clean Code

Refactoring, Patterns, Testen und Techniken für sauberen Code



- **Kommentare, Formatierung, Strukturierung**
- **Fehler-Handling und Unit-Tests**
- **Zahlreiche Fallstudien, Best Practices, Heuristiken und Code Smells**

Selbst schlechter Code kann funktionieren. Aber wenn der Code nicht sauber ist, kann er ein Entwicklungsunternehmen in die Knie zwingen. Jedes Jahr gehen unzählige Stunden und beträchtliche Ressourcen verloren, weil Code schlecht geschrieben ist. Aber das muss nicht sein.

Mit Clean Code präsentiert Ihnen der bekannte Software-Experte Robert C. Martin ein revolutionäres Paradigma, mit dem er Ihnen aufzeigt, wie Sie guten Code schreiben und schlechten Code überarbeiten. Zusammen mit seinen Kollegen von Object Mentor destilliert er die besten Praktiken der agilen Entwicklung von sauberem Code zu einem einzigartigen Buch. So können Sie sich die Erfahrungswerte der Meister der Software-Entwicklung aneignen, die aus Ihnen einen besseren Programmierer machen werden – anhand konkreter Fallstudien, die im Buch detailliert durchgearbeitet werden.

Sie werden in diesem Buch sehr viel Code lesen. Und Sie werden aufgefordert, darüber nachzudenken, was an diesem Code richtig und falsch ist. Noch wichtiger: Sie werden herausgefordert, Ihre professionellen Werte und Ihre Einstellung zu Ihrem Beruf zu überprüfen.

Clean Code besteht aus drei Teilen: Der erste Teil beschreibt die Prinzipien, Patterns und Techniken, die zum Schreiben von sauberem Code benötigt werden. Der zweite Teil besteht aus mehreren, zunehmend komplexeren Fallstudien. An jeder Fallstudie wird aufgezeigt, wie Code gesäubert wird – wie eine mit Problemen behaftete Code-Basis in eine solide und effiziente Form umgewandelt wird. Der dritte Teil enthält den Ertrag und den Lohn der praktischen Arbeit: ein umfangreiches Kapitel mit Best Practices, Heuristiken und Code Smells, die bei der Erstellung der Fallstudien zusammengetragen wurden. Das Ergebnis ist eine Wissensbasis, die beschreibt, wie wir denken, wenn wir Code schreiben, lesen und säubern.

Dieses Buch ist ein Muss für alle Entwickler, Software-Ingenieure, Projektmanager, Team-Leiter oder Systemanalytiker, die daran interessiert sind, besseren Code zu produzieren.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5548

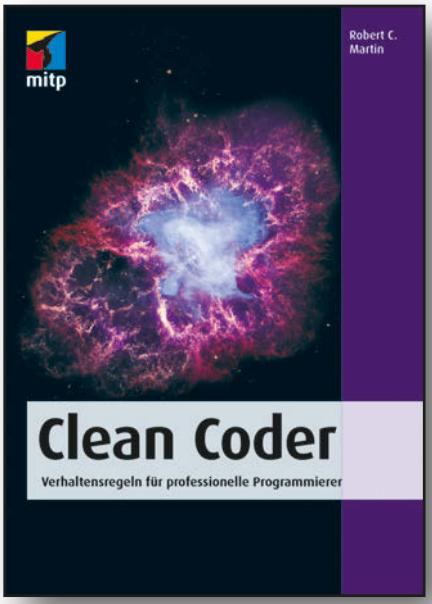
ISBN 978-3-8266-5548-7

Robert C. Martin

Clean Coder

Verhaltensregeln für professionelle Programmierer

Der Nachfolger von „Uncle Bobs“ erfolgreichem Bestseller *Clean Code*



Erfolgreiche Programmierer haben eines gemeinsam: Die Praxis der Software-Entwicklung ist ihnen eine Herzensangelegenheit. Auch wenn sie unter einem nicht nachlassenden Druck arbeiten, setzen sie sich engagiert ein. Software-Entwicklung ist für sie eine Handwerkskunst.

In Clean Coder stellt der legendäre Software-Experte Robert C. Martin die Disziplinen, Techniken, Tools und Methoden vor, die Programmierer zu Profis machen.

Dieses Buch steckt voller praktischer Ratschläge und behandelt alle wichtigen Themen vom professionellen Verhalten und Zeitmanagement über die Aufwandsschätzung bis zum Refactoring und Testen. Hier geht es um mehr als nur um Technik: Es geht um die innere Haltung. Martin zeigt, wie Sie sich als Software-Entwickler professionell verhalten, gut und sauber arbeiten und verlässlich kommunizieren und planen. Er beschreibt, wie Sie sich schwierigen Entscheidungen stellen und zeigt, dass das eigene Wissen zu verantwortungsvollem Handeln verpflichtet.

In diesem Buch lernen Sie:

- Was es bedeutet, sich als echter Profi zu verhalten
- Wie Sie mit Konflikten, knappen Zeitplänen und unvernünftigen Managern umgehen
- Wie Sie beim Programmieren im Fluss bleiben und Schreibblockaden überwinden
- Wie Sie mit unerbittlichem Druck umgehen und Burnout vermeiden
- Wie Sie Ihr Zeitmanagement optimieren
- Wie Sie für Umgebungen sorgen, in denen Programmierer und Teams wachsen und sich wohlfühlen
- Wann Sie „Nein“ sagen sollten – und wie Sie das anstellen
- Wann Sie „Ja“ sagen sollten – und was ein Ja wirklich bedeutet

Großartige Software ist etwas Bewundernswertes: Sie ist leistungsfähig, elegant, funktional und erfreut bei der Arbeit sowohl den Entwickler als auch den Anwender. Hervorragende Software wird nicht von Maschinen geschrieben, sondern von Profis, die sich dieser Handwerkskunst unerschütterlich verschrieben haben. Clean Coder hilft Ihnen, zu diesem Kreis zu gehören.

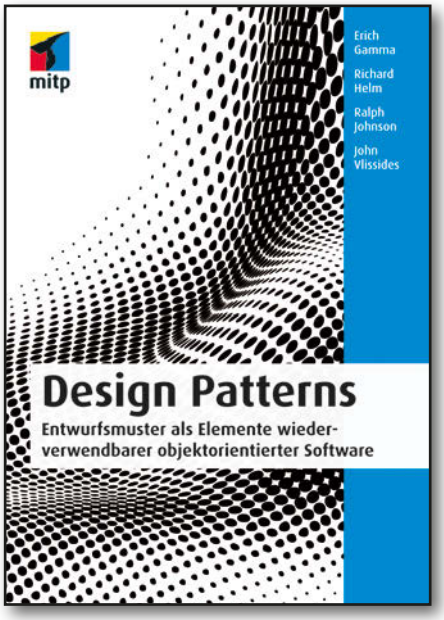
Probekapitel und Infos erhalten Sie unter:
www.mitp.de/9695

ISBN 978-3-8266-9695-4

**Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides**

Design Patterns

Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software



- **Der Bestseller von Gamma und Co. in komplett neuer Übersetzung**
- **Das Standardwerk für die objektorientierte Softwareentwicklung**
- **Zeitlose und effektive Lösungen für wiederkehrende Aufgaben im Softwaredesign**

Mit Design Patterns lassen sich wiederkehrende Aufgaben in der objektorientierten Softwareentwicklung effektiv lösen. Die Autoren stellen einen Katalog einfacher und prägnanter Lösungen für häufig auftretende Aufgabenstellungen vor. Mit diesen 23 Patterns können Softwareentwickler flexiblere, elegantere und vor allem auch wiederverwendbare Designs erstellen, ohne die Lösungen jedes Mal aufs Neue selbst entwickeln zu müssen.

Die Autoren beschreiben zunächst, was Patterns eigentlich sind und wie sie sich beim Design objektorientierter Software einsetzen lassen. Danach werden die stets wiederkehrenden Designs systematisch benannt, erläutert, beurteilt und katalogisiert. Mit diesem Leitfaden lernen Sie, wie sich diese wichtigen Patterns in den Softwareentwicklungsprozess einfügen und wie sie zur Lösung Ihrer eigenen Designprobleme am besten eingesetzt werden.

Bei jedem Pattern ist angegeben, in welchem Kontext es besonders geeignet ist und welche Konsequenzen und Kompromisse sich aus der Verwendung des Patterns im Rahmen des Gesamtdesigns ergeben. Sämtliche Patterns entstammen echten Anwendungen und beruhen auf tatsächlich existierenden Vorbildern. Außerdem ist jedes Pattern mit Codebeispielen versehen, die demonstrieren, wie es in objektorientierten Programmiersprachen wie C++ oder Smalltalk implementiert werden kann.

Das Buch eignet sich nicht nur als Lehrbuch, sondern auch hervorragend als Nachschlagewerk und Referenz und erleichtert so auch besonders die Zusammenarbeit im Team.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/9700

ISBN 978-3-8266-9700-5