Roy
Osherove

The Art of
Unit Testing
Deutsche Ausgabe

Teil I

Erste Schritte

Dieser Teil des Buches behandelt die Grundlagen des Unit Testings.

In Kapitel 1 werden wir definieren, was »gutes« Unit Testing bedeutet und vergleichen es mit dem Integration Testing. Außerdem werfen wir einen kurzen Blick auf das Test-Driven Development und dessen Rolle in Bezug auf das Unit Testing.

Dann, in Kapitel 2, werden wir versuchen, unseren ersten Unit Test mithilfe von NUnit zu schreiben. Wir werden die grundlegende API von NUnit kennenlernen und wir 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 23
- **Kapitel 2**
  Ein erster Unit Test 41

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

Kapitel 1

Die Grundlagen des Unit Testings

Dieses Kapitel behandelt

- die Definition von Unit Testing und Integration Testing
- ein einfaches Unit-Testing-Beispiel
- die testgetriebene Entwicklung (Test-Driven Development)

Es beginnt immer mit dem ersten Schritt: das erste selbstgeschriebene 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 diese sogar als schlecht, misslich, langsam oder nicht wartbar (wie die meisten Leute). Andererseits 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 beider Ansätze 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 werden wir 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 – Die klassische Definition

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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

Unit Tests zu einer ausgesprochen nützlichen Praxis in der Softwareentwicklung wurde. Bevor wir weitermachen, sollten wir das Unit Testing zunächst genauer definieren. Hier die klassische Definition (basierend auf dem entsprechenden Wikipedia-Artikel[1]):

Definition

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 Unit Testing wird gegen ein »System Under Test« (SUT) durchgeführt.

Definition

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

Diese klassische Definition eines Unit Tests, obwohl technisch korrekt, ist kaum ausreichend, um uns als Entwickler weiterzubringen. Aller Wahrscheinlichkeit nach wissen Sie dies bereits und sind wenig begeistert, diese Definition schon wieder zu lesen, denn sie taucht in allen Büchern und auf jeder Webseite auf, die das Thema Unit Testing behandeln. Machen Sie sich nichts daraus: In diesem Buch gehen wir über die klassische Definition des Unit Testings hinaus und sprechen Wartbarkeit, Lesbarkeit, Richtigkeit und noch mehr an. Aber diese gewohnte Definition, eben gerade weil sie gewohnt ist, gibt uns eine gemeinsame Basis, von der aus wir die Idee des Unit Testings erweitern können.

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

Die meisten Leute, 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 selbstgeschriebener Testanwendungen oder indem sie den zu testenden Code von dem Produkt, das sie entwickeln, ausführen lassen.

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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

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 wir definieren, was ein »guter« Unit Test ist, können wir sicherstellen, dass wir nicht mit einer falschen Vorstellung davon starten, was wir zu schreiben versuchen.

Um in dieser heiklen Kunst des Unit Testings erfolgreich zu sein, bedarf es nicht nur einer *technischen Definition* dessen, was Unit Tests sind, sondern es ist auch von grundlegender Bedeutung, die *Eigenschaften* eines guten Unit Tests zu beschreiben. Um zu verstehen, was ein »guter« Unit Test ist, müssen wir 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 WinForm oder WebForm UI (User Interface) 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.

Diese Tests mögen nützlich sein und vielleicht auch der klassischen Definition eines Unit Tests nahekommen, aber sie sind weit davon entfernt, wie wir einen »guten« Unit Test in diesem Buch definieren werden. 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?

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

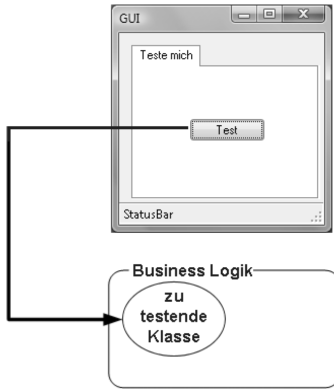

Abb. 1.1: Im klassischen Test verwenden Entwickler ein GUI (Graphical User Interface), um eine Aktion für die Klasse, die sie testen wollen, anzustoßen. Anschließend untersuchen sie das Resultat.

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 für die zukünftige Nutzung stehen bleiben.
- Jeder sollte in der Lage sein, den Test laufen zu lassen.
- Er sollte auf Knopfdruck ablaufen.
- Er sollte schnell ablaufen.

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 Unit Test innerhalb weniger Minuten schreiben?

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

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 *Integration Tests* durchgeführt.

1.3 Integration Tests

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 Teilen, die zusammenarbeiten. Jedes Teil 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 Teile liegen oder in mehr als nur einem. Es ist die *Integration* der Teile, die das Auto fahren lässt. Sie können sich die Bewegung des Autos als den ultimativen Integration Test vorstellen: 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 User Interfaces. Das Klicken auf einen Button löst eine Serie von Ereignissen aus – verschiedene 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).

The Complete Guide to Software Testing von Bill Hetzel 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 Leute die ganze Zeit über tun, nicht als Teil eines System Integration Tests, aber als Teil eines Entwicklungs- und Unit Tests.

Hier ist eine bessere Definition für das Integration Testing:

Definition

Integration Testing bedeutet, dass zwei oder mehr voneinander abhängige Softwaremodule als eine Gruppe getestet werden.

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

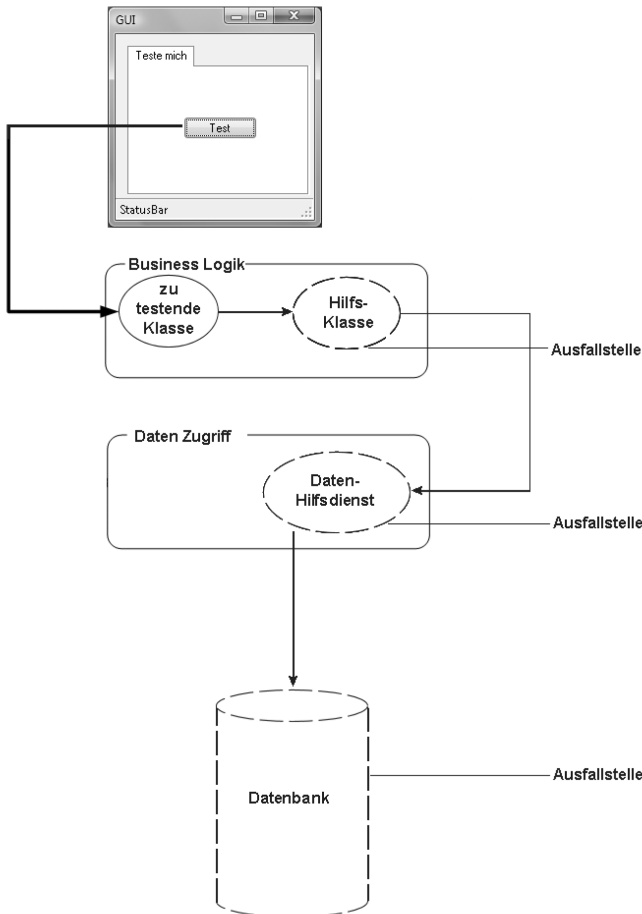

Abb. 1.2: Es gibt viele mögliche Schwachstellen in einem Integration Test. Alle Komponenten müssen zusammenarbeiten und jede von ihnen kann fehlerhaft sein, was es schwieriger macht, die Ursache eines Fehlers zu finden.

Noch mal zusammengefasst: Ein Integration Test behandelt viele sich ergänzende Einheiten von Code, um zu untersuchen, ob die Software ein oder mehrere erwartete Resultate liefert, wohingegen ein Unit Test nur eine einzelne Einheit isoliert betrachtet.

Die Fragen vom Anfang des Kapitels 1.2 können dabei helfen, einige der Nachteile des Integration Testings zu erkennen. Betrachten wir sie noch einmal und versuchen wir, die Eigenschaften zu definieren, nach denen wir bei einem »guten« Unit Test Ausschau halten müssen.

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

1.3.1 Nachteile von Integration Tests im Vergleich zu automatisierten Unit Tests

Wenden wir die Fragen aus Kapitel 1.2 auf Integration Tests an und überlegen wir, was wir 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 vor zwei Wochen 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 wir etwas später in diesem Buch sehen.

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

Definition

Eine *Regression* ist ein Feature, das mal funktioniert hat, 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 letzten Punkt zusammen, geht aber noch einen Schritt weiter. Sie möchten sicherstellen, dass Sie 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.

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

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 Leute definieren Legacy Code hingegen gern als »Code ohne Tests«. In dem Buch *Effektives Arbeiten mit Legacy Code* von Michael Feathers 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 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. Als Entwickler haben wir 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.

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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

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

  Eines der am einfachsten zu erkennenden Merkmale eines Integration Tests 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.

Ausgehend von dem, was wir bisher darüber gelernt haben, was ein Unit Test nicht ist und welche Eigenschaften für einen nützlichen Test vorhanden sein müssen, können wir nun damit beginnen, die primäre Frage dieses Kapitels zu beantworten: Was ist ein »guter« Unit Test?

1.4 »Gute« Unit Tests – Eine Definition

Lassen Sie uns nun, da wir die wichtigen Eigenschaften, die ein Unit Test haben sollte, erfasst haben, die Unit Tests ein für allemal definieren.

Definition

Ein *Unit Test* ist ein automatisiertes Stück Code, das eine zu testende Methode oder Klasse aufruft und dann einige Annahmen über das logische Verhalten dieser Methode oder Klasse prüft. Ein Unit Test wird fast immer mithilfe eines Unit Testing Frameworks erstellt. Er kann einfach geschrieben und schnell ausgeführt werden. Er ist vollständig automatisiert, vertrauenswürdig[3], lesbar und wartbar.

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.

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

Diese Definition scheint sicherlich ein bisschen viel verlangt, insbesondere wenn man bedenkt, wie viele Entwickler ich schon gesehen habe, die 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 7 diskutiert.)

Definition

Logischer Code ist jegliches Stück Code, das irgendeine Art von Logik enthält, so klein es auch sein mag. Es ist Logischer Code, wenn er irgendetwas von dem folgenden enthält: 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 getestet werden muss. Aber seien Sie auf der Hut: Sobald Sie irgendeine Überprüfung in die Property einbauen, sollten Sie sicherstellen, dass die Logik getestet wird.

Im nächsten Abschnitt werfen wir einen Blick auf einen einfachen Unit Test, der komplett im 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 eines solchen Tests ohne Framework aussehen kann. Sie können dies dann mit dem Einsatz eines Frameworks in Kapitel 2 vergleichen.

Angenommen, wir haben eine Klasse `SimpleParser` (siehe Listing 1.1), die wir gern 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).

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

```
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 den Zahlen 0 und 1 umgehen!");
      }
   }
}
```

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

Wir können das Projekt einer einfachen Konsolenanwendung anlegen, das eine Referenz auf das Assembly dieser Klasse hält, und wir können eine Methode `SimpleParserTests` wie in 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 sie 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:
-------
Parse and sum sollten für einen leeren String 0 zurückgeben");
         }
```


© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

```
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

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

Als nächstes können wir den gerade geschriebenen Test aufrufen, indem wir eine einfache `Main`-Methode wie in Listing 1.3 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. Wir können dann der `Main`-Funktion weitere Methodenaufrufe hinzufügen, wenn wir 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 Fehlermeldungen sorgt und von allen Tests verwendet werden kann. Sie könnten ebenso spezielle Hilfsmethoden hinzufügen, die auf verschiedene Dinge,

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

wie etwa Null-Objekte, leere Strings usw., prüfen würden, so dass Sie nicht immer wieder die gleichen langen Code-Zeilen in Ihren Tests schreiben müssen.

Listing 1.4 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,
   "Parse and sum sollten 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`

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

Unit Testing Frameworks können Sie dabei unterstützen, Hilfsfunktionen wie diese allgemeiner zu formulieren, so dass es einfacher wird, Tests zu schreiben. Wir werden 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 wir 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 Leute 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 meinen Blog »The various meanings of TDD« (`http://weblogs.asp.net/rosherove/archive/2007/10/08/the-various-meanings-of-tdd.aspx`) werfen. 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.

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.

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

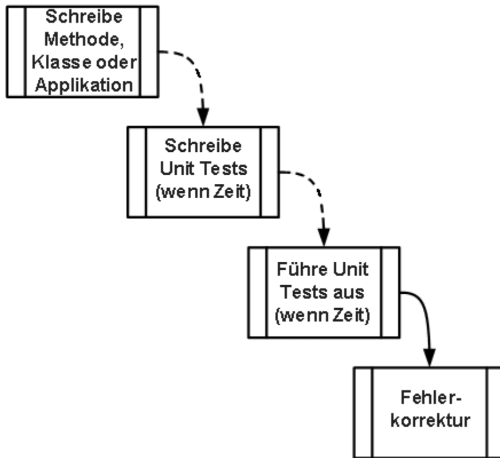

Abb. 1.3: Der traditionelle Weg, Unit Tests zu schreiben. Die gestrichelten Linien deuten optionale Aktionen an.

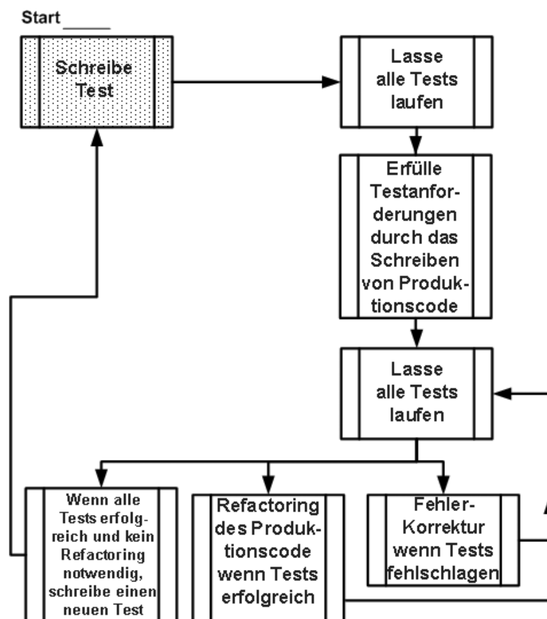

Abb. 1.4: Testgetriebene Entwicklung – ein Blick aus der Vogelperspektive. Beachten Sie den spiralförmigen Charakter des Prozesses: Schreibe einen Test, schreibe Code, passe ihn an (»Refactoring«), schreibe den nächsten Test. Die Abbildung zeigt die inkrementelle Natur von TDD: Kleine Schritte führen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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

Dieses Buch konzentriert sich auf die Technik des Schreibens »guter« Unit Tests und nicht auf die testgetriebene Entwicklung, obwohl ich ein großer Fan der testgetriebenen Entwicklung 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 der testgetriebenen Entwicklung 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, dass `LastSum` tatsächlich eine Zahl enthält. Der Test würde fehlschlagen, weil wir diese Funktionalität noch nicht eingebaut haben.

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

   Er sollte so einfach wie möglich geschrieben 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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

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 Leute finden dies auf die harte Tour heraus.

1.7 Zusammenfassung

In diesem Kapitel haben wir 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 wir uns anschauen, was für eine Art von Tests wir bisher durchgeführt haben. Wir haben sie als Integration Tests identifiziert, da ein Satz voneinander abhängiger Units getestet wird.

Es ist wichtig, den Unterschied zwischen Unit Tests und Integration Tests 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 Integration Tests ohne ein unterstützendes Framework geworfen: Diese Art von Test ist schwierig zu schreiben und zu automatisieren, langsam in der Anwendung und erfordert einen

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023

gewissen Konfigurationsaufwand. Auch wenn Sie Integration Tests 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 über die testgetriebene Entwicklung gesprochen, 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). Es unterstützt Sie bei der Erstellung von Tests, denen man vertrauen kann, indem es dafür sorgt, dass Ihre Tests fehlschlagen, wenn der Produktionscode noch nicht vorhanden ist, 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 verwenden, ohne zu wissen, wie man gute Tests schreibt.

Wenn Sie die Tests schreiben, *nachdem* Sie den Code geschrieben haben, dann nehmen Sie an, dass der Test gut ist, weil er nicht fehlschlägt, obwohl es sein könnte, dass Ihre Tests Bugs beinhalten. Glauben Sie mir – Fehler in seinen Tests zu finden, ist eines der frustrierendsten Dinge, die man sich vorstellen kann. Es ist wichtig, dass Sie Ihre Tests nicht in diesen Zustand geraten lassen, und TDD zählt zu den besten Methoden, die ich kenne, um diese Möglichkeit nahe bei null zu halten.

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

© des Titels »Unit Testing« (ISBN 978-3-8266-9023-5) 2010
by Verlagsgruppe Hüthig Jehle Rehm GmbH, Heidelberg
Nähere Informationen unter: http://www.it-fachportal.de/9023