Kapitel 1

Einführung

Es gibt viele Arten von Tests, die man in einem Software-Projekt durchführen kann und soll. Einige Testarten erfordern eine besonders starke Einbindung der Anwender, andere verlangen ganze Teams von engagierten Mitarbeitern zur Qualitätssicherung oder andere kostspielige Ressourcen.

Aber darum soll es hier nicht gehen.

Stattdessen reden wir über *Unit-Tests*, einen sehr wesentlichen und leider oft missverstandenen Baustein zum Erfolg eines Projektes und der beteiligten Personen. Unit-Tests sind ein vergleichsweise kostengünstiger Weg zu besserem Quelltext in kürzerer Zeit.

Viele Unternehmen haben große Ziele, wenn es um das Testen geht, aber sie testen tendenziell erst gegen Projektende, wenn der Zeitplan besonders knapp ist. So wird das Testen immer wieder zusammengestrichen oder ganz weggelassen.

Programmierer empfinden das Testen oft als Belästigung. Sie sehen es als unerwünschte Nebenaufgabe, die vom eigentlichen Ziel ablenkt. Diese Programmierer wollen lieber produktiven Quelltext schreiben.

Jeder wird zustimmen, dass man mehr testen soll. Genauso wie man zustimmt, dass man mehr Gemüse essen, zu rauchen aufhören, Ruhepausen einlegen und auch regelmäßig Sport treiben sollte. Das bedeutet noch lange nicht, dass einer von uns das auch wirklich macht.

Unit-Tests können weit mehr. Manche betrachten sie als eine Art Gemüse beim Programmieren. Wir sehen in ihnen eher das geniale Gewürz, das einfach alles noch besser schmecken lässt. Unit-Tests sind nicht dazu da, um in groß angelegten Aktionen Qualität in die

Software zu testen. Sie sind kein Werkzeug für Endanwender, Manager oder Teamleiter. Unit-Tests sind von Programmierern für Programmierer. Sie sind nur für uns da, um uns die Arbeit zu erleichtern.

Einfach gesagt, Unit-Tests können den Unterschied zwischen Erfolg und Misserfolg ausmachen. Dazu eine kurze Geschichte.

1.1 Sicherheit beim Programmieren

Es war einmal vor langer, langer Zeit – vielleicht war es vergangenen Dienstag –, da saßen die beiden Programmierer Pat und Dale vor ihren Rechnern und kämpften gegen einen bedrohlich näher kommenden Termin. Pat hat den Quelltext nur so heruntergeschrieben, Klasse für Klasse, Methode für Methode. Er unterbrach nur, um den Quelltext gelegentlich zu compilieren.

Bis zu dem Abend direkt vor dem Termin hat Pat genau so weitergemacht. Dann wollte er das ganze Programm in Aktion sehen. Pat startete es, aber bekam überhaupt keine Ausgabe. Absolute Funkstille. Es war an der Zeit, mit dem Debugger schrittweise durch das Programm zu gehen. Hmm. Das kann doch gar nicht sein, dachte Pat. Es ist doch einfach nicht möglich, dass diese Variable jetzt `null` ist. Also ging Pat noch einen Schritt weiter zurück im Quelltext und versuchte, dieses schwer fassbare Problem einzukreisen.

Es wurde spät. Der Fehler war gefunden und beseitigt, aber Pat hat bei der Suche noch ein paar mehr gefunden. Und von seinem Programm hatte er noch immer keine Ausgabe gesehen. Pat konnte einfach nicht verstehen, wieso. Es war einfach unerklärlich.

Während der ganzen Zeit hat Dale nicht annähernd so schnell Quelltext produziert. Dale schrieb Methode für Methode und immer auch einen Test dazu. Keinen besonders ausgefallenen Test, nur so viel, dass er sehen konnte, ob die Methode so funktioniert wie angedacht. Über den Test nachzudenken und ihn dann zu programmieren, brauchte etwas Zeit, aber Dale wollte nicht voranstürmen, solange die Methode nicht nachweislich funktionierte. Erst dann ging Dale zur nächsten Methode, zum Beispiel jene, die seine gerade getestete Methode aufruft, und so weiter.

Dale nutzte den Debugger kaum, wenn überhaupt. Und er wunderte sich über Pat, weil er, wild fluchend, mit rot angelaufenen Augen auf den Debugger starrte.

Der Termin kam und verstrich, Pat hat ihn nicht geschafft. Der Quelltext von Dale wurde in das Projekt integriert und lief fast perfekt. Nur ein kleines Problem war aufgetaucht, aber es war einfach zu erkennen. Dale hatte es in wenigen Minuten behoben.

Die Pointe ist, dass Pat und Dale etwa gleichaltrig sind, die gleichen Programmiererfahrungen haben und auch intellektuell etwa gleichauf sind. Der einzige Unterschied ist das Vertrauen, das Dale durch die Arbeit mit Unit-Tests hat. Er testet jede neue Methode, bevor er darauf aufbaut und sie aus anderen Methoden aufruft.

Bei Pat ist das anders. Er „weiß“, dass Quelltext sich so verhalten soll, wie er es beim Tippen erwartet. Deshalb überprüft er das auch erst, wenn schon fast der gesamte Quelltext geschrieben ist. Aber dann ist es zu spät. Es wird sehr schwer, Fehlerursachen zu lokalisieren oder auch nur herauszufinden, was funktioniert und was nicht.

1.2 Was sind Unit-Tests?

Ein *Unit-Test* ist ein Stück Quelltext, das ein Entwickler nur zum Testen eines sehr kleinen und klar umrissenen Teiles der Funktionalität geschrieben hat. Üblicherweise führt jeder Unit-Test eine ganz bestimmte Methode in einem ganz bestimmten Kontext aus. Zum Beispiel können Sie einen hohen Wert an eine Liste anfügen und dann überprüfen, ob der Wert am Ende der Liste erscheint. Oder Sie löschen ein paar bestimmte Buchstaben aus einer Zeichenkette und stellen dann sicher, dass sie wirklich entfernt wurden.

Mit dem Ausführen der Unit-Tests beweist der Entwickler, dass der Quelltext das macht, was er von ihm erwartet.

Dabei ist es unerheblich, ob das die Funktionalität ist, die Kunden und Anwender wollen, denn für deren Wünsche gibt es Akzeptanz-Tests. Wir wollen an dieser Stelle keine formale Verifikation durchführen und auch nicht die Performance überprüfen. Wir wollen lediglich Quelltext, der das macht, was wir von ihm erwarten, nicht

mehr. Folglich möchten wir sehr kleine, abgeschlossene Teile der Funktionalität testen. Wenn wir Vertrauen in die vielen kleinen Teile haben, können wir aus ihnen anschließend Systeme zusammensetzen und die Systeme testen.

Wenn wir nicht sicher wären, dass der Quelltext das macht, was wir wollen, wäre jede andere Form von Tests reine Zeitverschwendung. Natürlich braucht man auch diese anderen Testformen und in manchen Anwendungsgebieten sogar ein sehr viel formaleres Testen. Aber Testen ist wie soziales Engagement, es beginnt zu Hause.

1.3 Warum soll ich mich um Unit-Tests kümmern?

Unit-Tests werden Ihre Arbeit vereinfachen. Mit ihnen werden Sie bessere Entwürfe machen und die Zeit, die Sie mit Debugging verbringen, erheblich reduzieren.

In unserer kleinen Geschichte bekam Pat Probleme, weil er angenommen hatte, dass seine Low-Level-Funktionen korrekt funktionieren würden. Er setzte diese Funktionen in darüber liegenden Schichten ein, die wiederum von noch höheren Schichten genutzt wurden, und so weiter. Ohne auch nur für irgendeinen Teil des Quelltextes Sicherheit zu haben, hat Pat ein Kartenhaus von Annahmen gebaut – nur ein kleiner Fehler ganz unten, und schon stürzt alles in sich zusammen.

Wenn die einfachen, grundlegenden Funktionen nicht zuverlässig sind, bleiben auch die notwendigen Fehlerkorrekturen nicht auf dieses einfache Niveau beschränkt. Diese Korrekturen beseitigen dann zwar die Probleme in den grundlegenden Funktionen, aber das hat wiederum Auswirkungen auf die Schichten darüber. Somit benötigen auch diese Schichten Korrekturen und so weiter. Die notwendigen Anpassungen weiten sich immer mehr aus, werden größer und komplizierter. Schließlich fällt das Kartenhaus – und damit das Projekt – in sich zusammen.

Pat sagt oft „Das ist unmöglich!“ und „Ich verstehe nicht, wie das passieren kann“. Wenn Sie derlei Gedanken haben, ist das ein Anzeichen für ungenügendes Vertrauen in den eigenen Quelltext – Sie wissen nicht wirklich, was funktioniert und was nicht.

Um Vertrauen in den Quelltext zu bekommen, so wie Dale es hat, müssen Sie den Quelltext selbst fragen, was er macht, und die Ergebnisse mit dem Erwarteten vergleichen.

Dieser einfache Ansatz beschreibt den Kern der Unit-Tests. Sie sind die effektivste Technik, um besser zu programmieren.

1.4 Was ist unser Ziel?

Es passiert leicht, dass man so viel Spass beim Schreiben der Unit-Tests hat, dass man darüber das Programmieren der eigentlichen Anwendung für die Kunden und Anwender vernachlässigt. Deshalb ist es wichtig, klar zu sagen, was wir mit den Unit-Tests erreichen wollen. Es geht uns vor allem darum, unsere Arbeit und die unseres Teams zu erleichtern.

Macht der Quelltext, was wir wollen?

Im Grunde geht es immer wieder um die eine Frage: „Macht der Quelltext das, was wir von ihm erwarten?“ Das bedeutet, er kann unter Umständen etwas anderes machen, als die Anforderungen verlangen würden. Das Testen gegen die Anforderungen ist aber ein ganz anderes Thema. Wir wollen nur überprüfen, ob der Quelltext so funktioniert, wie *wir* das wollen.

Funktioniert es immer, unter allen Umständen?

Viele Entwickler, die behaupten, zu testen, schreiben genau einen Test. Sie testen den Gut-Fall, den einen Ausführungspfad, auf dem alles perfekt funktioniert.

Natürlich ist die Realität selten so freundlich, oft arbeitet eben doch nicht alles perfekt: Ausnahmen (engl.: Exceptions) treten auf, Festplatten haben keinen Platz mehr, Netzwerkverbindungen brechen zusammen, Puffer laufen über und – um Gottes willen – wir machen Fehler. Das ist der ingenieurmäßige Anteil in der Softwaretechnik. Bauingenieure müssen die Last von Brücken berücksichtigen, die Einflüsse von starken Winden und Erdbeben, von Fluten und so weiter. Ingenieure der Elektrotechnik haben an Themen wie Frequenz-Drift, Spannungsspitzen, Rauschen, ja selbst an die Verfügbarkeit der eingesetzten Bauteile zu denken.

Eine Brücke testet man nicht, indem man an einem ruhigen, sonnigen Tag ein einzelnes Auto darüber fahren lässt. Das genügt einfach nicht. Genauso müssen wir bei unserem Quelltext nicht nur sicherstellen, dass er wie gewünscht funktioniert, sondern er muss *immer* korrekt arbeiten. Es darf keine Rolle spielen, welche Windstärke herrscht, ob die Parameter vielleicht falsch sind, ob die Festplatte voll ist oder das Netzwerk krankt.

Können wir uns darauf verlassen?

Quelltext, auf den man sich nicht verlassen kann, ist nutzlos. Schlimmer noch ist Quelltext, auf den man sich verlässt, der aber Fehler enthält. Das kann uns eine Menge Zeit bei der Fehlersuche kosten. Nur sehr wenige Projekte haben Zeit zu verschenken, also möchten wir diese „einen Schritt nach vorn, zwei Schritte zurück"-Technik gern vermeiden. Wir wollen immer vorwärts.

Niemand schreibt perfekten Quelltext, und das ist in Ordnung – solange man weiß, wo der Fehler liegt. Viele der spektakulärsten Softwarefehler, die ein Raumfahrzeug auf einem fernen Planeten stranden oder es während des Fluges explodieren lassen, hätten vermieden werden können. Man hätte nur die Grenzen der Software kennen und beachten müssen. In der Software der Ariane 5 wurde beispielsweise eine Bibliothek verwendet, die schon bei älteren Raketen zum Einsatz kam. Diese Bibliothek konnte aber die größeren Zahlenwerte der höher fliegenden, neuen Rakete nicht handhaben.[1] Nach 40 Sekunden explodierte die Rakete, übrig blieben 500 Millionen Dollar teure Trümmer.

Wir wollen uns auf unseren Quelltext verlassen können und sowohl seine Stärken als auch seine Schwachstellen und Einschränkungen kennen.

Nehmen wir beispielsweise an, wir hätten eine Methode geschrieben, um die Reihenfolge einer Liste von Zahlen umzukehren. Beim Testen übergeben wir auch eine leere Liste und der Test schlägt fehl. Die Anforderungen haben nichts davon gesagt, dass unsere Methode auch eine leere Liste handhaben muss. Wir können den

[1] Für Raumfahrt-Fans: Der numerische Überlauf passierte bei einer sehr viel größeren horizontalen Neigung. Der geänderte Abschusswinkel sollte der Rakete eine höhere horizontale Geschwindigkeit geben.

Fakt einfach im Kommentar zur Methode dokumentieren, und wenn die Methode mit einer leeren Liste aufgerufen wird, werfen wir eine Ausnahme (Exception). Jetzt haben wir die Grenzen der Software offensichtlich gemacht. So ist es angenehmer, als sie auf die harte Tour herausfinden zu müssen (was oft an einem ungemütlichen Ort passiert, wie zum Beispiel in den oberen Schichten der Atmosphäre).

Drückt es unsere Absicht aus?

Ein angenehmer Nebeneffekt von Unit-Tests ist, dass sie die beabsichtigte Verwendung des Quelltextes verdeutlichen. So gesehen sind Unit-Tests wie eine ausführbare Dokumentation, die zeigt, was man unter den verschiedenen, angedachten Randbedingungen vom Quelltext erwarten kann.

Team-Mitglieder können sich unsere Tests ansehen und so erkennen, wie man die Klassen/Methoden verwendet. Wenn jemandem ein Test einfällt, den wir nicht bedacht haben, werden sie sehr schnell aktiv werden.

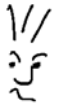
Nicht zuletzt hat eine ausführbare Dokumentation den Vorteil, immer korrekt zu sein. Anders als bei Text-Dokumenten kann sich einfach keine Abweichung zwischen Tests und Anwendungs-Quelltext entwickeln (außer wenn man aufhört, die Tests regelmäßig auszuführen).

1.5 Wie schreibt man Unit-Tests?

Die Arbeit mit Unit-Tests ist im Grunde ganz einfach zu lernen. Es gibt auch ein paar Richtlinien und Vorgehensweisen, mit denen man diese Arbeit einfacher und besonders effektiv gestalten kann.

Zuerst muss man sich entscheiden, wie man die fragliche Methode testen will – noch bevor man anfängt, den Quelltext dafür zu schreiben. Wenn man zumindest eine ungefähre Idee davon hat, wie es weitergehen soll, schreibt man den Test-Quelltext entweder vor dem Quelltext der Anwendung oder zeitgleich mit ihm.

Anschließend wird der Test ausgeführt. Wenn er schnell ausführbar ist, wird man wahrscheinlich alle Tests dieses Teilsystems oder sogar des ganzen Systems ausführen. Es ist wichtig, dass *alle* Tests bestanden werden, nicht nur der neue. Wir wollen Kollateralschäden genauso vermeiden wie neue Fehler an den gerade bearbeiteten Stellen.



Joe fragt:

„Was sind Kollateralschäden?“

Kollateralschäden sind Fehler, die in entfernten, eigentlich unabhängigen Teilen des Systems entstehen, wenn eine neue Eigenschaft hinzugefügt oder ein Fehler beseitigt wird. Wenn man diesen heimtückischen Fehlern nicht schnell entgegenwirkt, kann das ganze Softwaresystem mit der Zeit so weit gestört werden, dass es kaum mehr zu reparieren ist.

Wir nennen das manchmal den *Whac-a-Mole*-Effekt. Beim Kirmes-Spiel *Whac-a-Mole* muss der Spieler mit einer Art Hammer auf mechanische Maulwürfe schlagen, die auf dem Spielfeld auftauchen. Aber ihre Köpfe bleiben nicht lange oben. Sobald man einen Maulwurf anvisiert, verschwindet er wieder, und auf der gegenüber liegenden Seite des Feldes taucht ein anderer auf. Die Maulwürfe erscheinen und verschwinden sehr schnell. Es kann sehr frustrierend sein, wenn man versucht, einen zu erwischen. Oft endet es damit, dass die Spieler hilflos auf das Spielfeld schlagen, während die Maulwürfe genau dort auftauchen, wo man sie nicht erwartet.

Großflächige Kollateralschäden an einem Quelltext sind etwas Vergleichbares.

Jeder Test muss selbst entscheiden, ob er bestanden wurde oder nicht. Es genügt nicht, wenn Sie oder ein anderer Pechvogel alle Ausgaben lesen müssen, nur um zu entscheiden, ob der Quelltext funktioniert. Wir wollen dahin kommen, dass wir mit einem flüchtigen Blick auf das Testergebnis sagen können, ob alles in Ordnung ist. Wenn wir später über die Feinheiten des Einsatzes von Unit-Test-Frameworks reden, kommen wir auf dieses Thema zurück.

1.6 Ausreden gegen das Testen

Ungeachtet unserer vernünftigen und leidenschaftlichen Appelle werden einige Programmierer zwar mit dem Kopf nicken und auch der Wichtigkeit von Unit-Tests zustimmen, aber gleichzeitig felsenfest behaupten, dass *sie* diese Tests nicht machen können. Sie führen dabei eine Vielzahl von Gründen ins Feld. Hier folgen die verbreitetsten Ausreden, die wir gehört haben, und ihre Gegenbeweise.

Tests zu schreiben, kostet zu viel Zeit

Wer das erste Mal mit Unit-Tests in Berührung kommt, wird das mit großer Wahrscheinlichkeit so sehen. Natürlich stimmt es nicht. Um zu erkennen, warum es nicht stimmt, müssen wir mal genauer betrachten, womit wir unsere Zeit beim Programmieren eigentlich genau verbringen.

Viele Leute betrachten Testen als etwas, das gegen Ende eines Projektes gemacht wird. Ja, wenn man mit den Unit-Tests erst kurz vor Projektende anfängt, dann wird das definitiv zu zeitaufwändig. Es könnte sogar sein, dass man die Aufgabe nicht vor dem Ende des Universums lösen kann.

Jedenfalls wird es sich so anfühlen. Es ist so, als ob man ein paar Hektar mit einem einfachen Rasenmäher mähen wollte. Wenn man früh damit anfängt, solange nur Gras wächst, dann ist es ziemlich einfach. Aber wenn man abwartet bis dicke knochige Bäume mit einem dichten Unterholz entstanden sind, dann wird die Aufgabe unlösbar.

Wenn man den vollständigen Entwicklungszyklus betrachtet, ist es günstiger, schon begleitend in Tests zu investieren (engl. Pay-As-You-Go), als bis zum Projektende damit zu warten. Indem man kleine Tests gemeinsam mit dem Quelltext der Anwendung entwikkelt, erspart man sich viel Stress zum Projektende. Außerdem hat man generell weniger Fehler, weil man eben immer mit getestetem Quelltext arbeitet. Indem man begleitend immer ein wenig Zeit für die Tests investiert, kommt man ohne einen gewaltigen zeitlichen Aufwand am Ende aus.

Es geht nicht um die Entscheidung zwischen „jetzt testen“ oder „später testen“. Es geht um die Entscheidung zwischen einer gleichmäßigen Arbeitsweise oder der exponentiellen Arbeitslast und Komplexität, wenn man es am Ende mit Fehlerkorrekturen und Nacharbeiten versucht. Die ganze Mehrarbeit macht die Produktivität zunichte (siehe Abbildung 1.1).

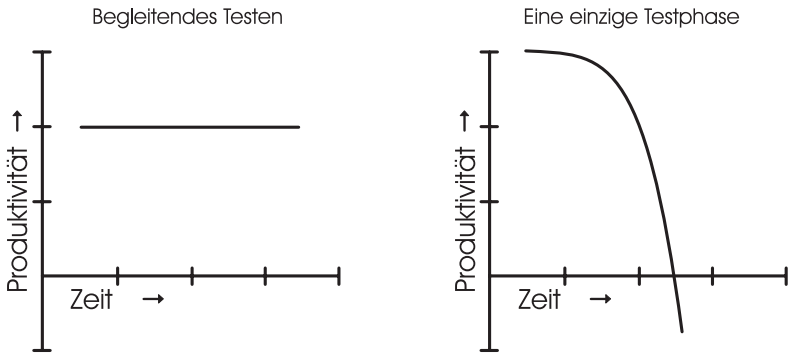


Abbildung 1.1: Vergleich von begleitendem Testen und einer abschließenden Testphase

Tests bekommt man aber nicht für umsonst. Auch beim begleitenden Testen ist der Aufwand nicht Null, es kostet durchaus etwas Aufwand (Zeit und Geld). Aber werfen Sie einen Blick auf den beängstigenden Verlauf der rechten Kurve – direkt nach unten. Ihre Produktivität kann sogar negativ werden! Derartige Einbrüche der Produktivität können einem Projekt sehr schnell zum Verhängnis werden.

Wenn Sie also glauben, Sie hätten keine Zeit, neben der Anwendung auch die Tests dazu zu entwickeln, stellen Sie sich bitte folgende Fragen:

1. Wie viel Zeit verbringen Sie mit Debugging von Quelltext, den Sie oder andere geschrieben haben?
2. Wie viel Zeit investieren Sie, um Quelltext zu überarbeiten, obwohl er eigentlich schon funktionieren sollte (allerdings zeigen sich noch erhebliche Fehler)?
3. Wie viel Zeit investieren Sie, die Ursache eines Fehlers zu suchen?

Bei den meisten Programmierern, die keine Unit-Tests verwenden, werden sich diese Zeiten sehr schnell aufsummieren. Sie werden sogar noch schneller wachsen, je länger das Projekt läuft. Wenn man Unit-Tests richtig einsetzt, lassen sich diese Zeiten erheblich reduzieren, was genug zeitlichen Freiraum für alle möglichen Unit-Tests schafft und vielleicht sogar noch etwas mehr.

Das Ausführen der Tests dauert zu lange

Das sollte es nicht. Die meisten Unit-Tests sollten ziemlich schnell sein, damit man Hunderte oder gar Tausende in wenigen Sekunden ausführen kann. Aber manchmal ist das einfach nicht möglich und man hat ein paar wenige Tests, die zu langsam sind, um bequem immer alle Tests ausführen zu können.

In diesen Fällen wird man die lang laufenden Tests von den kürzeren trennen. Führen Sie die Langläufer nur einmal am Tag aus, oder auch nur alle paar Tage, wenn das angemessen erscheint. Die schnell ausführbaren Unit-Tests sollten Sie aber immer alle zusammen ausführen.

Meinen Quelltext zu testen, ist nicht meine Aufgabe

Das ist eine wirklich interessante Ausrede. Was genau ist denn dann Ihre Aufgabe? Nehmen wir einmal an, zu Ihren Aufgaben gehört das Schreiben von funktionierendem Quelltext. Sie erfüllen Ihre Aufgabe nicht, wenn Sie Ihren Quelltext über den Zaun werfen, ohne sicher zu sein, dass er auch funktioniert. Es ist auch nicht höflich, wenn Sie von anderen erwarten, Ihre Fehler zu beseitigen. In einigen Fällen kann das Abliefern von großen Mengen fehlerträchigen Quelltextes zu einem Karrierebegrenzer werden.

Wenn es den Leuten der Qualitätssicherung dagegen schwerfällt, Fehler in Ihrem Quelltext zu finden, wird Ihr Ansehen rasch steigen, und auch Ihr Arbeitsplatz wird sicherer.

Ich weiß gar nicht genau, wie sich der Quelltext verhalten soll, wie kann ich ihn dann testen?

Wenn Sie nicht wissen, was Sie vom Programm erwarten, dann ist es vielleicht noch nicht an der Zeit, es zu schreiben. Vielleicht ist ein Prototyp an dieser Stelle angemessener, um in einem ersten Schritt die genauen Anforderungen herauszuarbeiten.

Wenn Sie nicht einmal die Aufgabe genau kennen, wie können Sie dann wissen, ob Ihr Quelltext diese Aufgabe erfüllt?

Es compiliert doch!

Ok, niemand wird das *wirklich* als Ausrede nehmen, zumindest wird es niemand aussprechen. Es ist jedoch sehr einfach, sich einzureden, ein erfolgreicher Compiler-Lauf wäre ein Siegel für das Erreichen einer bestimmten Qualität.

Aber der Segen des Compilers ist ein ziemlich leeres Kompliment. Ein Compiler kann nur bestätigen, dass die Syntax korrekt ist. Er kann aber nicht herausfinden, was Ihr Quelltext machen wird. Ein Java-Compiler kann beispielsweise sehr einfach herausfinden, dass diese Zeile falsch ist:

```
public statuc void main(String args[]) {
```

Es ist nur ein einfacher Tippfehler, korrekt wäre `static`. Das ist einfach, aber nehmen wir einmal an, Sie hätten Folgendes programmiert:

```
public void addit(Object anObject){
  ArrayList myList = new ArrayList();
  myList.add(anObject);
  myList.add(anObject);
  // und so weiter...
}
```

Wollten Sie das Objekt wirklich zweimal zur Liste hinzufügen? Vielleicht ja, vielleicht aber auch nicht. Für den Compiler ist das kein Unterschied, nur Sie wissen, was Sie eigentlich wollten.[2]

[2] Automatische Test-Generatoren, die vom Quelltext der Anwendung ausgehen, laufen in dieselbe Falle – sie können auch nur das verwenden, was im Quelltext steht, aber nicht, was gemeint war.

Ich werde fürs Programmieren bezahlt, nicht fürs Testen

Dieser Logik folgend, werden Sie aber auch nicht dafür bezahlt, den halben Tag mit dem Debugger zu verbringen. Wahrscheinlich werden Sie eher dafür bezahlt, ein *funktionierendes* Programm zu schreiben. Unit-Tests sind nur ein Werkzeug, um das zu erreichen, wie auch die IDE und der Compiler nur Werkzeuge sind.

Ich fühle mich schuldig, wenn ich den Testern die Arbeit wegnehme

Keine Bange, das machen Sie nicht. Schließlich reden wir hier nur von den *Unit-Tests*. Das ist das ganz elementare Testen auf unterster Ebene, das Testen für uns Entwickler. Es gibt noch mehr als genug zu tun, um funktionale Tests, Akzeptanz-, Performance- und System-Tests, Validierung und Verifikation, formale Analyse und so weiter auf den Weg zu bringen.

Meine Firma erlaubt keine Unit-Tests auf dem Produktiv-System

Halt! Wir sprechen hier von Unit-Tests, die der Entwickler durchführt. Es ist zwar technisch möglich, diese Tests auch in anderen Umgebungen auszuführen (beispielsweise auf dem Produktiv-System), aber dann sind es *keine Unit-Tests mehr*! Führen Sie die Unit-Tests auf Ihrer Entwickler-Maschine aus und verwenden Sie dabei Ihre eigene Datenbank und Mock-Objekte (siehe Kapitel 6).

Wenn die Kollegen von der Qualitätssicherung oder andere Tester diese Tests auch in einer Produktiv- oder einer Qualitätssicherungs-Umgebung laufen lassen wollen, können Sie natürlich bei den technischen Feinheiten helfen. Sie sollten sich jedoch darüber klar sein, dass es in diesen Umgebungen keine Unit-Tests mehr sind.

1.7 Was uns erwartet

Kapitel 2, *Ihre ersten Unit-Tests*, gibt einen Überblick über das Schreiben von Tests. Darauf aufbauend, werden in Kapitel 3 die Feinheiten der *Testprogrammierung mit JUnit* behandelt. Danach verwenden wir ein paar Kapitel darauf, wie man entscheidet, *was* getestet werden soll und wie.

Anschließend, im Kapitel 7, betrachten wir die wichtigen Eigenschaften von guten Tests. In Kapitel 8 beschäftigen wir uns damit, was man in einem Softwareprojekt benötigt, um effektiv testen zu können. In diesem Kapitel diskutieren wir außerdem, wie man mit großen Mengen an Quelltext von Altsystemen umgeht. In Kapitel 9 sehen wir uns den (positiven) Einfluss genauer an, den Unit-Tests auf den Entwurf der Anwendung haben.

Die Anhänge enthalten weitere, nützliche Informationen: einen Überblick zu den häufigsten Problemen mit Unit-Tests, einen Abschnitt zur Installation von JUnit, ein Beispiel für ein JUnit-Testgerüst und eine Liste weiterer Quellen einschließlich der Literaturliste. Den Abschluss bildet eine Überblicksseite mit den wichtigsten Tipps und Empfehlungen aus dem Buch.

Lehnen Sie sich zurück und entspannen Sie sich. Willkommen in der Welt des besseren Programmierens.