1 Einleitung

Testen ist eigentlich nicht cool. Nie hätte ich erwartet, einmal ein Buch übers Testen zu schreiben.

JUnit hat einen Wandel eingeläutet. JUnits grüner Balken macht süchtig, sorgt er doch für ein lang vermisstes Gefühl: das Vertrauen, dass die produzierte Software tatsächlich wie gewünscht funktioniert. In gleicher Weise schickt sich derzeit FIT an, das Vertrauensverhältnis der Kunden zur gelieferten Software zu stärken. Auf einmal ist Testen doch cool. Warum dieser Wandel?

Testen war typischerweise

- die letzte Phase in wasserfallartigen Projekten;
- das letzte Kapitel in Büchern über die Softwareentwicklung;
- worauf kaum ein Programmierer wirklich Lust hat;
- Aufgabe der Abteilung für Qualitätssicherung;
- schmerzhaft, wenn wir Fehler lange nach ihrer Entstehung finden;
- das Erste, was über Bord geht, wenn es eng wird;
- nur ad hoc, selten systematisch.

Ich möchte in diesem Buch einen anderen Weg gehen. Ich möchte zeigen, wie ein bisschen Testen einen großen Unterschied machen kann. Deshalb sollten Sie weiterlesen.

Testen ist wirklich wichtig. Wissen wir.

Dass Software meist nicht ausreichend getestet wurde, ist zum Teil gut verständlich, da die Gründe dafür in erster Linie im *Menschlichen* liegen. Damit das Testen im Programmieralltag und selbst bei zunehmendem Stress nicht vernachlässigt wird, muss es mit den natürlichen Instinkten der verantwortlichen Menschen gehen und der Natur der Softwareentwicklung folgen.

Wie könnte eine effektive Teststrategie zur Entwicklung technisch und geschäftlich hochwertiger Software aussehen?

Effektives Testen muss

- **möglichst zeitnah zur Programmierung** erfolgen, damit es nicht zu einem nachgelagerten Prozess wird, der unter Zeitdruck ausfällt;
- **automatisiert wiederholbar** sein, weil manuelle Tests nicht durchgeführt werden, wenn der Stress im Programmieralltag zunimmt;
- **Spaß machen –** Softwareentwicklung soll Spaß machen;
- **so häufig wie das Kompilieren** ausgeführt werden, damit Fehler und Seiteneffekte gleich bei ihrer Entstehung entdeckt werden, nicht erst vom Testteam oder gar vom Kunden;
- **so einfach wie das Kompilieren** sein, um Tests in der Programmiersprache selbst zu schreiben, nicht mit proprietären Werkzeugen;
- **pragmatisch** sein, um an erster Stelle funktionierende Software zu produzieren, nicht um Punkte einer Checkliste abzuhaken;
- **mehr bringen als kosten,** ansonsten wird es zum Selbstzweck.

1.1 Was ist Testgetriebene Entwicklung?

Testgetriebene Entwicklung (engl. *Test-Driven Development*) ist eine qualitätsbewusste Programmiertechnik, mit der wir ein Programm in kleinen Schritten entwickeln können. Als Entwickler schreiben wir automatisierte *Unit Tests* für die Anforderungen an unseren Code. Diese Tests prüfen ein Programm in kleinen unabhängigen Einheiten. Sie helfen uns, das *Programm richtig* zu entwickeln. Zudem schreiben wir automatisierte *Akzeptanztests* für die Anforderungen der Kunden. Diese Tests prüfen ein Programm als große integrierte Systemeinheit. Sie helfen uns, das *richtige Programm* zu entwickeln.

Unit Tests

Akzeptanztests

Testgetriebene Programmierung

Die Idee testgetriebenen Arbeitens ist, erst Testcode zu schreiben, bevor wir den eigentlich zu testenden Programmcode schreiben. Das heißt, dass jede funktionale Programmänderung durch einen gezielten Test motiviert wird [be_{02a}]. Diesen Test entwerfen wir so, dass er zunächst fehlschlägt. Er muss fehlschlagen, weil das Programm die gewünschte Funktionalität noch nicht besitzt. Erst anschließend schreiben wir den Code, der diesen Test erfüllt. Auf diese Weise treiben wir die gesamte Entwicklung inkrementell durch das unmittelbare Feedback konkreter Tests an. Somit lernen wir, wann unser Programm zu funktionieren beginnt und wann es zu funktionieren aufhört.

1. Direktive der Testgetriebenen Entwicklung:
Motivieren Sie jede Änderung des Programmverhaltens durch einen automatisierten Test.

Sie fragen sich vielleicht, was wir denn testen wollen, wenn wir noch überhaupt keinen Code geschrieben haben? Doch diese Frage lässt sich umdrehen: Woher wissen wir denn, was wir programmieren sollen, wenn wir noch nicht wissen, was denn überhaupt erforderlich ist? Zuerst die Tests zu schreiben, ermöglicht uns herauszufinden, was wir programmieren müssen und was nicht. Zuerst die Tests zu schreiben, stellt außerdem sicher, dass wir tatsächlich programmieren, was wir programmieren wollten.

Spezifizieren – Programmieren – Verifizieren

Wahrscheinlich kennen Sie das typische Testproblem, dass sich Code nur schwer testen lässt, der nicht gleich testbar entworfen wurde. Die klassische Testliteratur weiß für solche Fälle schweres Geschütz aufzufahren, weil das Kind leider schon in den Brunnen gefallen ist. Testgetriebene Entwicklung umgeht dieses Problem, weil das frühe Testen noch Auswirkungen auf das resultierende Design nehmen kann. Wenn Sie Ihre Tests zuerst schreiben, wird Ihr Code auch testbar sein. Den Test haben Sie ja schließlich gerade schon geschrieben.

Testbarkeit von Anfang an

Iterativ-inkrementelle Entwicklung ist nur mit gut strukturiertem Code möglich. Ohne ständige Pflege nimmt die Codequalität mit zunehmender Entwicklungsdauer für gewöhnlich ab, was jede Weiterentwicklung behindert, wenn nicht sogar verhindert.

Die einzige existierende Gegenmaßnahme ist stetiges *Refactoring*. Durch kleine funktionserhaltende Schritte zur Designverbesserung können wir diesen Trend umkehren: Wir können Code so entwickeln, dass er nicht von Tag zu Tag mehr und mehr degeneriert, sondern dass seine Qualität unaufhörlich steigt. *Einfache Form* zu erhalten, ist das Ziel des Refactorings. Indem wir nach jedem einzelnen Refactoringschritt wieder und wieder alle gesammelten Tests ausführen, können wir sicherstellen, dass durch unsere Umformung nicht ungewollt das vorhandene Funktionsverhalten des Programms beeinträchtigt wurde.

Refactoring

2. Direktive der Testgetriebenen Entwicklung:
Bringen Sie Ihren Code immer in die Einfache Form.

Sobald wir Software im Team entwickeln, wollen wir alle Änderungen am Code regelmäßig miteinander abgleichen. Die *Häufige Integration* der einzelnen Entwicklungstätigkeiten stellt sicher, dass Konflikte, wenn sie auftreten, noch klein und somit sehr leicht zu beheben sind, und dass wir jederzeit eine lauffähige Software demonstrieren können. Zur erfolgreichen Integration müssen natürlich alle Tests laufen.

Häufige Integration

3. Direktive der Testgetriebenen Entwicklung:
Integrieren Sie Ihren Code so häufig wie nötig.

1.2 Warum Testgetriebene Entwicklung?

Testgetriebene Entwicklung ist eine qualitätsbewusste Programmiertechnik. Doch was heißt Softwarequalität in diesem Zusammenhang?

Jede Software hat einen bestimmten Wert, der sich definiert durch

- **funktionale Qualität** im Hinblick auf Funktionalität und Fehlerfreiheit für die einwandfreie Benutzbarkeit einer Software;
- **strukturelle Qualität** im Hinblick auf Design und Codestruktur für die nahtlose Weiterentwicklung einer Software.

Testgetriebene Entwicklung ist qualitätsbewusst, weil sie die zwei Qualitäten gleichzeitig über weite Zeiträume aufrechterhalten kann.

- Automatisierte Tests bewahren die funktionale Qualität.
- Fortlaufendes Refactoring bewahrt die strukturelle Qualität.

Software, die wie gewünscht funktioniert, aber nicht wie gewünscht weiterentwickelt werden kann, hat nach dieser Definition keinen Wert. Die eine Qualität kommt nicht ohne die andere aus. Der Gesamtwert der Software wird wirklich durch das Minimum der beiden bestimmt. Testgetriebene Entwicklung ermöglicht uns erst, dass wir entwickeln können, was wir brauchen, wenn wir es brauchen, ohne die Qualität unserer Software zunehmend zu kompromittieren. Just-in-Time.

Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken.

Tests machen Fortschritt greifbar. Programmierte Software ist an und für sich nicht greifbar. Erst durch sich selbst prüfenden Code wird der programmierte Code greifbar. Programmierer, die ihren Programmcode durch solche Selbsttests fixieren, vergegenständlichen ihn damit. Sie lernen, indem sie etwas Greifbares entwickeln. Wie ist das gemeint?

Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. Die Länge seiner Sicherheitsleine bestimmt dabei maßgeblich die Lücke von dem Moment, wo der Kletterer den ersten ungesicherten Schritt probiert, bis zu dem Moment, wo er seine Kletterkünste durch einen weiteren Haken belohnt. Der bis zum Haken erkletterte Weg gehört ihm in jedem Fall, selbst wenn ihm ein Fehler unterläuft. Jeder gesetzte Haken bedeutet Fortschritt für ihn, auch wenn es ihm Mühe kostet, den Haken in die Wand zu schlagen. Auch hat der Kletterer es selbst in der Hand, wie viel er in einer Situation aufs Spiel setzen möchte. An steilen Hängen wird er seine Leine sicher kürzer halten wollen als an zugänglichen Passagen.

Programmierte Tests sind wie diese Sicherheitshaken. Sie schenken uns ein dichtes Sicherheitsnetz, während wir bei stetig wachsender Softwarekomplexität höher und höher klettern. Beim Programmieren gilt es, die Länge der Leine bewusst zu kontrollieren. Wie viele Tests Sie schreiben müssen, damit Sie Vertrauen in den programmierten Code haben, und wie groß Sie Ihre Programmierschritte dabei wählen, lernen Sie auf Ihrem Weg nach oben.

Tests sichern den Erhalt der vorhandenen Funktionalität bei Erweiterung und Überarbeitung.

Software ist änderbar. Zu leicht änderbar, wie sich herausstellt. Es ist schließlich kinderleicht, ein Programm zu ändern. Auch irrtümlich. Unter Umständen muss dazu nur ein einziges Bit kippen. Die erste Hürde der Softwareentwicklung besteht darin, ein Programm so zu entwickeln, dass es genau das tut, was es tun soll. Mit zunehmender Komplexität einer Software gewinnt diese Hürde jedoch gewöhnlich an Höhe. Unser Interesse muss es sein, diese Barriere während der gesamten Entwicklung möglichst niedrig zu halten. Tests können uns dabei helfen, Software so gut wie möglich zu entwickeln. Denn ohne Tests überlassen wir die Softwareentwicklung zum Teil dem Zufall, weil selbst kleine und sorgfältige Änderungen auch Auswirkungen auf das übrige Programmverhalten nehmen können. Dafür ist Software zu komplex und deshalb ist häufiges Testen so wichtig.

Die erste Hürde: die Entwicklung selbst

Erfolgreiche Software wird weiterentwickelt. Software wird nie ganz fertig. Praktisch ergeben sich immer noch Änderungen, die teils notwendig oder wenigstens wünschenswert sind. Entweder können sich die wirklichen Benutzeranforderungen erst aufgrund von schon ausgelieferter Software herauskristallisieren oder schnell wandelnde Märkte entscheiden über den neuen Geschäftswert einer Softwareidee. Die zweite Hürde besteht in der Softwareentwicklung darin, einmal entwickelte Software auch problemlos weiterentwickeln zu können.

Die zweite Hürde: die Weiterentwicklung

Software leidet unter Entropie. Obwohl die Änderbarkeit von Software in der Natur noch ihresgleichen sucht, verhärtet der Code doch in vielen Softwareprojekten mit der Zeit. Der Grund dafür ist die einsetzende Entropie. Das heißt, der Grad an Unordnung innerhalb eines Systems nimmt zu. Die Ursache lässt sich so erklären: Mit jeder weiteren Zeile Code schlägt die Geschichte, die unser Programm erzählt, neue Bögen. Gleichzeitig schränken wir jedoch den Rahmen der Geschichte immer weiter ein. Wenn wir nicht bald wieder Raum für neuen Code schaffen, ist die Geschichte schnell zu Ende erzählt.

Refactoring verlängert die produktive Lebensdauer einer Software.

Software muss soft bleiben. Der Code von gestern darf den Code von heute nicht behindern. Wenn wir nicht ständig den Kompromiss neu eingehen zwischen wertvoller weiterer Programmfunktionalität und den Aufwänden, um den bereits bestehenden Code in Schuss zu halten, steht die Weiterentwicklung bald vor ihrem Aus. Unsere Software gibt durch unzählige Änderungen und unter gleichzeitiger Missachtung der Softwareentropie zuerst ihre strukturelle Qualität auf und oft ihre funktionale Qualität gleich hinterher.

Software verrottet. Wenn die ursprünglichen Entwurfsabsichten im geschriebenen Programmcode immer schwieriger zu entziffern sind, verringert sich unsere Entwicklungsgeschwindigkeit. Niemand traut sich mehr, den verrottenden Code anzufassen. Gerade wenn wir aus Unachtsamkeit oder Zeitdruck nicht regelmäßig den Code aufräumen, degeneriert er so rapide, dass er bald keine Änderungen mehr zulässt, ohne immerzu neue Fehler ins Programm einzuführen. Zusätzlich steigt damit natürlich unsere Angst vor unkontrolliert entstehenden Seiteneffekten und die Geschichte nimmt ihren weiteren Verlauf.

Code lässt sich im Nachhinein oft nur schlecht testen.

Hochwertige Software erfordert automatisierte Tests. Je weniger Zeit zwischen der Entstehung und Entdeckung von Fehlern vergeht, desto schneller können wir sie eingrenzen und beheben. Wenn wir unsere Software jederzeit ändern und binnen Sekunden ihre Fitness prüfen können, haben wir beweglichen Code und damit Wettbewerbsvorteile. Unser Vertrauen in die Software steigt und damit unser Mut und die Fähigkeit, Änderungen vorzunehmen, die wir uns unter anderen Umständen nicht zugetraut hätten. Eine vollständige Automatisierung des Testprozesses ist notwendig, weil wir unsere Tests extrem häufig ausführen. Die Wiederholung manueller Tests wäre zu langwierig.

Testen muss in die Softwareentwicklung integriert sein. Nur wenn wir unsere Tests zeitnah zur Programmierung durchführen, haben wir die Chance, über die Tests die Einsicht in ein insgesamt einfacher zu testendes Design zu finden. Der Moment, in dem wir das Verständnis gewonnen haben, wie das Programm arbeiten soll, ist zudem ein guter Zeitpunkt, unser Wissen in einem Test festzuhalten. Aus diesem Grund liegt der ideale Zeitpunkt, um einen Test zu schreiben, unmittelbar vor der eigentlichen Programmierung. Der Compiler prüft die Programmsyntax, unsere Tests prüfen die Semantik.

1.3 Über dieses Buch

Dieses Buch führt Sie in die Testgetriebene Entwicklung ein, und zwar im Kleinen (mit Unit Tests) wie auch im Großen (mit Akzeptanztests). Ich habe das Buch für angehende und professionelle Softwareentwickler geschrieben. Mein Ziel ist es, Ihnen die Techniken und Werkzeuge zu vermitteln, die ich für das Handwerk der Softwareentwicklung für wertvoll halte.

Aufbau des Buches

Kapitel 2 gibt Ihnen einen Überblick über die Philosophie und Basistechniken der Testgetriebenen Entwicklung. Die darauf folgenden vier Kapitel umfassen das Grundhandwerkszeug: Kapitel 3 macht Sie vorab mit JUnit vertraut, Kapitel 4 widmet sich dem Testgetriebenen Entwicklungszyklus, Kapitel 5 der Rolle des Refactorings darin und Kapitel 6 der Häufigen Integration im Team. Die zwei Kapitel 7 und 8 vermitteln Ihnen dann fortgeschrittenere Techniken zum Schreiben automatisierter Tests und zum Meistern schwieriger Testsituationen. Kapitel 9 schlägt eine alternative Route zu Kapitel 8 ein. Überspringen Sie dieses Kapitel fürs Erste, falls Sie sich vom Stoff abgehängt fühlen – hier ballen sich viele fortgeschrittene Techniken in einem Buchkapitel. In Kapitel 10 treffen Sie schließlich auf FIT und Kapitel 11 beendet das Buch mit einigen Schlussgedanken.

Ich arbeite in diesem Buch mit einer großen Menge Codebeispiele. Achten Sie jedoch nicht so sehr auf den Code, sondern auf meine Bewegungen. Der Code dient mir nur als Vehikel. Viel wichtiger sind die Schritte. Aus diesem Grund möchte ich Sie auch dazu motivieren, die Beispiele live am Rechner mitzuverfolgen. Sie werden ungleich mehr davon haben. Den Quellcode aller Kapitel und Ergänzungen zum Buch finden Sie auf meinen Webseiten:

```
http://www.frankwestphal.de/
TestgetriebeneEntwicklungmitJUnitundFIT.html
```

Alle Codebeispiele basieren auf Java, lassen sich jedoch leicht in andere Sprachen übersetzen. Ich habe das Buchbeispiel, ein kleines DVD-Verleihsystem, bewusst einfach und technologieunabhängig gehalten. Wenn Sie *konkrete Teststrategien* suchen, wie Sie dies und das testen können, konsultieren Sie das Buch [li$_{05}$] von Johannes Link. Ich habe dagegen versucht, mich auf die *allgemeinen Prinzipien* zu konzentrieren. In anderen Worten: Ich will Ihnen keine Testprobleme knacken; ich möchte Ihnen zeigen, wie Sie die Herausforderungen selber meistern. Die beiden Bücher sind sich gegenseitig also zwei gute Kompagnons.

1.4 Merci beaucoup

Insgesamt habe ich in der einen oder anderen Form über fünf Jahre an diesem Buch gearbeitet. Ich möchte an dieser Stelle den zahlreichen Menschen danken, die mir während dieser langen Zeit immer wieder Mut, Kraft und Inspiration geschenkt haben. Die Ehrenurkunden gehen an **Jutta Eckstein**, **Tammo Freese**, **Dierk König**, **Johannes Link** und **Christa Preisendanz**.

Für Review-Kommentare und Ideen danke ich (in chronologischer Reihenfolge): **Leah Striker**, **Michael Schürig**, **Meike Budweg**, **Tammo Freese**, **Ulrike Jürgens**, **Rolf F. Katzenberger**, **Hans Wegener**, **Marko Schulz**, **Antonín Andert**, **Manfred Lange**, **Julian Mack**, **Johannes Link**, **Karsten Menne**, **Martin Müller-Rohde**, **Stefan Roock**, **Andreas Schoolmann**, **Frankmartin Wiethüchter**, **Dierk König**, **Bastiaan Harmsen**, **Olaf Kock**, **Stefan Schmiedl**, **Martin Lippert**, **Etienne Studer**, **Ilja Preuß**, **Jens Uwe Pipka**, **Robert Wenner**, **Armin Röhrl**, **Eberhard Wolff**, **Peter Roßbach**, **Alexander Schmid**, **Mario Winter**, **Daniel Schweizer**, **Torsten Mumme**, **Eduard Bachner**, **Bernd Schiffer**, **Ali Natour**, **Jürgen Ahting**, **Ralf Stuckert**, **Markus Schramm**. Im Traum hätte ich nicht für möglich gehalten, wie viele unterschiedliche Fehler und Schwächen so viele Augen entdecken würden. Der Titel des Meisterrezensenten geht dabei an **Rolf F. Katzenberger** und **Robert Wenner**. Tausend Dank für eure Zeit und Gründlichkeit.

Besten Dank auch an meine Gastautoren: **Juan Altmayer Pizzorno**, **Ward Cunningham**, **Sabine Embacher**, **Michael Feathers**, **Steve Freeman**, **Tammo Freese**, **Bastiaan Harmsen**, **Michael Hill**, **Andy Hunt**, **Christian Junghans**, **Olaf Kock**, **Dierk König**, **Lasse Koskela**, **Steffen Künzel**, **Johannes Link**, **Tim Mackinnon**, **Ivan Moore**, **Moritz Petersen**, **Dave Thomas**, **Robert Wenner**.

Zu guter Letzt möchte ich meiner Lektorin meinen größten Dank aussprechen. Ohne **Christa Preisendanz** könnte sich dieses Buch heute sicherlich in die *Invisible Library* einreihen.

www.invisiblelibrary.com

Hamburg, Oktober 2005 *Frank Westphal*
http://www.frankwestphal.de