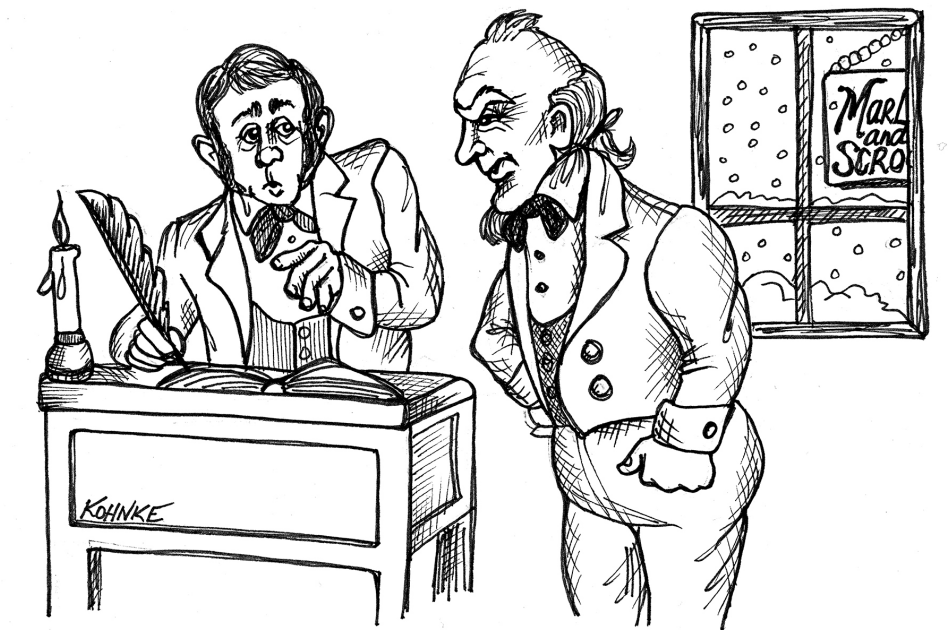
Teil I

Die Praktiken

In diesem Teil:

Kapitel 2

Testgetriebene Entwicklung



Unsere Diskussion der Testgetriebenen Entwicklung (TDD) erstreckt sich über zwei Kapitel. Zunächst behandeln wir die Grundlagen von TDD in einer sehr technischen und detaillierten Weise. In diesem Kapitel lernen Sie die Praktik Schritt für Schritt kennen. Das Kapitel bietet eine Menge Code zum Lesen und mehrere Videos, die Sie sich ansehen können[1].

In Kapitel 3, »Fortgeschrittenes TDD«, decken wir viele der Fallen und Probleme des TDD auf, denen Anfänger begegnen werden, wie z.B. bei Datenbanken und grafischen Benutzeroberflächen. Wir betrachten auch Designprinzipien, die ein gutes Testdesign fördern, und die Design Patterns des Testens. Schließlich betrachten wir einige interessante und tiefgreifende theoretische Möglichkeiten.

1 Anmerkung zur Übersetzung: Die Videos sind ausschließlich in englischer Sprache verfügbar. Sie finden den Downloadlink und den zugehörigen Downloadcode auf Seite 23.

2.1 Überblick

Null. Das ist eine wichtige Zahl. Die Null ist die Zahl des Gleichgewichts. Wenn die beiden Seiten einer Waage im Gleichgewicht sind, zeigt der Zeiger auf der Waage eine Null an. Ein neutrales Atom mit der gleichen Anzahl von Elektronen und Protonen hat eine Ladung von null. Die Summe der Kräfte auf einer Brücke ist gleich null. Null ist die Zahl des Gleichgewichts.

Haben Sie sich jemals gefragt, warum der Geldbetrag auf Ihrem Girokonto als »Saldo« bezeichnet wird? Das liegt daran, dass der Saldo Ihres Kontos die Summe aller Transaktionen ist, durch die Geld auf dieses Konto eingezahlt oder von ihm abgehoben wurde. Aber Transaktionen haben immer zwei Seiten, weil Geld *zwischen* Konten verschoben wird.

Die *nahe* Seite einer Transaktion betrifft Ihr eigenes Konto. Die *entfernte* Seite betrifft ein anderes Konto. Jede Transaktion, deren nahe Seite Geld auf *Ihr* Konto einzahlt, hat eine ferne Seite, die diesen Betrag von einem anderen Konto einzieht. Jedes Mal, wenn Sie einen Scheck ausstellen, zieht die nahe Seite der Transaktion Geld von Ihrem Konto ab, und die entfernte Seite zahlt dieses Geld auf ein anderes Konto ein. Der Saldo auf Ihrem Konto ist also die Summe der Transaktionen der nahen Seiten. Die Summe der entfernten Seiten sollte gleich dem Saldo Ihres Kontos sein. Die Summe der Transaktionen aller nahen und fernen Seiten sollte gleich null sein.

Vor zweitausend Jahren erkannte Gaius Plinius Secundus, bekannt als Plinius der Ältere, dieses Gesetz der Buchführung und erfand die Praktik der doppelten Buchführung. Und im Laufe der Jahrhunderte wurde diese Praktik von den Bankiers in Kairo und später von den Kaufleuten in Venedig verfeinert. Im Jahr 1494 verfasste Luca Pacioli, ein Franziskanermönch und Freund von Leonardo da Vinci, die erste vollständige Beschreibung der Praktik. Sie wurde in Buchform mit der neu erfundenen Druckerpresse veröffentlicht und verbreitete sich.

Im Jahr 1772, als die industrielle Revolution an Fahrt gewann, hatte Josiah Wedgwood mit seinem eigenen Erfolg zu kämpfen. Er war der Gründer einer Töpferfabrik, und sein Produkt war so gefragt, dass er sich bei dem Versuch, diese Nachfrage zu befriedigen, fast selbst in den Ruin trieb. Er führte die doppelte Buchführung ein und konnte dadurch die Geldströme in und aus seinem Unternehmen mit einer Genauigkeit verfolgen, die ihm zuvor nicht möglich gewesen war. Und indem er diese Ströme steuerte, konnte er den drohenden Bankrott abwenden und ein Unternehmen aufbauen, das bis heute existiert.

Wedgwood war nicht allein. Die Industrialisierung trieb das enorme Wachstum der Volkswirtschaften in Europa und Amerika voran. Um die aus diesem Wachstum resultierenden Geldströme zu bewältigen, übernahmen immer mehr Unternehmen diese Praktik.

Im Jahr 1795 schrieb Johann Wolfgang von Goethe in *Wilhelm Meisters Lehrjahre* das Folgende. Achten Sie genau auf den Inhalt, denn wir werden bald auf dieses Zitat zurückkommen.

> *»Leg es beiseite, wirf es ins Feuer!« versetzte Werner. »Die Erfindung ist nicht im geringsten lobenswürdig; schon vormals ärgerte mich diese Komposition genug und zog dir den Unwillen des Vaters zu. Es mögen ganz artige Verse sein; aber die Vorstellungsart ist grundfalsch. Ich erinnere mich noch deines personifizierten Gewerbes, deiner zusammengeschrumpften, erbärmlichen Sibylle. Du magst das Bild in irgendeinem elenden Kramladen aufgeschnappt haben. Von der Handlung hattest du damals keinen Begriff; ich wüßte nicht, wessen Geist ausgebreiteter wäre, ausgebreiteter sein müßte als der Geist eines echten Handelsmannes. Welchen Überblick verschafft uns nicht die Ordnung, in der wir unsere Geschäfte führen! Sie läßt uns jederzeit das Ganze überschauen, ohne daß wir nötig hätten, uns durch das Einzelne verwirren zu lassen. Welche Vorteile gewährt die doppelte Buchhaltung dem Kaufmanne! Es ist eine der schönsten Erfindungen des menschlichen Geistes, und ein jeder gute Haushalter sollte sie in seiner Wirtschaft einführen.«*[2]

Heute ist die doppelte Buchführung in fast allen Ländern der Welt gesetzlich verankert, und diese Praktik *definiert* zu einem großen Teil den Beruf des Buchhalters.

Aber kehren wir zu Goethes Zitat zurück. Beachten Sie die Worte, mit denen Goethe das von ihm so verabscheute Mittel des »Kommerz« beschreibt:

> *Ich erinnere mich noch deines personifizierten Gewerbes, deiner zusammengeschrumpften, erbärmlichen Sibylle. Du magst das Bild in irgendeinem elenden Kramladen aufgeschnappt haben.*

Haben Sie schon mal einen Code gesehen, der auf diese Beschreibung passt? Ich bin sicher, das haben Sie. Und ich habe das auch. Wenn es Ihnen so geht wie mir, dann haben Sie wahrscheinlich viel, viel zu viel davon gesehen. Wenn Sie sind wie ich, dann haben Sie sogar viel, viel zu viel davon selbst *geschrieben*.

Und nun ein letzter Blick auf Goethes Worte:

> *Welchen Überblick verschafft uns nicht die Ordnung, in der wir unsere Geschäfte führen! Sie läßt uns jederzeit das Ganze überschauen, ohne daß wir nötig hätten, uns durch das Einzelne verwirren zu lassen.*

Es ist bezeichnend, dass Goethe diesen mächtigen Nutzen der einfachen Praktik der doppelten Buchführung zuschreibt.

2 Quelle: »Wilhelm Meisters Lehrjahre« von Johann Wolfgang von Goethe auf `DigBib.Org`.

2.1.1 Software

Das Führen einer ordnungsgemäßen Buchhaltung ist für ein modernes Unternehmen unerlässlich, und die Praktik der doppelten Buchführung ist für die Führung einer ordnungsgemäßen Buchhaltung unerlässlich. Aber ist die ordnungsgemäße Wartung von Software für die Führung eines Unternehmens weniger wichtig? Ganz und gar nicht! Im 21. Jahrhundert ist Software das Herzstück eines jeden Unternehmens.

Welche Praktik könnten Softwareentwickler dazu verwenden, um eine solche Kontrolle und einen solchen Überblick über ihre Software zu erhalten, wie sie Buchhalter und Manager durch die doppelte Buchführung erhalten? Vielleicht denken Sie, dass Software und Buchhaltung so unterschiedliche Konzepte sind, dass keine Entsprechung erforderlich oder gar möglich ist. Ich bin da anderer Meinung.

Bedenken Sie, dass Buchhaltung so etwas wie die Kunst eines Magiers ist. Diejenigen von uns, die nicht mit den Ritualen und arkanen Künsten der Buchhalter vertraut sind, werden nur wenig von den Details oder Hintergründen dieses Berufsstands verstehen. Und was ist das Arbeitsergebnis dieses Berufs? Es ist eine Reihe von Dokumenten, die in einer komplexen und für den Laien verwirrenden Weise organisiert sind. Diese Dokumente sind mit einer Reihe von Symbolen versehen, die außer den Buchhaltern selbst nur wenige wirklich verstehen können. Und wenn auch nur eines dieser Symbole fehlerhaft ist, kann das schreckliche Folgen haben. Unternehmen könnten in Konkurs gehen, und Führungskräfte könnten ins Gefängnis kommen.

Überlegen Sie einmal, wie ähnlich die Buchhaltung der Softwareentwicklung ist. Software ist in der Tat die Kunst eines Magiers. Wer sich nicht mit den Ritualen und arkanen Künsten der Softwareentwicklung auskennt, hat keine wirkliche Vorstellung davon, was unter der Oberfläche vor sich geht. Und das Produkt? Wiederum eine Reihe von Dokumenten: der Quellcode – Dokumente, die auf äußerst komplexe und verwirrende Weise organisiert sind und die mit Symbolen übersät sind, die nur die Programmierer selbst entziffern können. Und wenn auch nur eines dieser Symbole fehlerhaft ist, kann das schreckliche Folgen haben.

Die beiden Berufe sind sich sehr ähnlich. Sie befassen sich beide mit der intensiven und anspruchsvollen Verwaltung komplizierter Details. Beide erfordern eine umfassende Ausbildung und Erfahrung, um gut zu sein. Beide sind mit der Erstellung komplexer Dokumente befasst, deren Genauigkeit auf der Ebene der einzelnen Symbole entscheidend ist.

Buchhalter und Programmierer wollen es vielleicht nicht zugeben, aber sie sind vom selben Schlag. Und die Praktik des älteren Berufsstands sollte von dem Jüngeren sorgfältig beobachtet werden.

Wie Sie im Folgenden sehen werden, ist TDD eine doppelte Buchführung. Es ist die gleiche Praktik, die für den gleichen Zweck ausgeführt wird und die gleichen

Ergebnisse liefert. Alles wird zweimal festgehalten, in komplementären Konten, die im Gleichgewicht gehalten werden müssen, indem Tests erfolgreich ausgeführt werden.

2.1.2 Die drei Gesetze des TDD

Bevor wir zu den drei Gesetzen kommen, möchte ich einige Vorbemerkungen machen.

Durch das Wesen der Praktik TDD wird es möglich, Folgendes zu erreichen:

1. Eine Testsuite zu erstellen, die ein Refactoring ermöglicht. Die Tests dieser Suite sind dermaßen vertrauenswürdig, dass das System ausgeliefert werden kann, wenn es die Tests besteht.
2. Produktionscode zu erstellen, der ausreichend entkoppelt ist, um testbar und refaktorierbar zu sein.
3. Schaffen einer extrem kurzen, zyklischen Feedbackschleife, die beim Schreiben von Programmen einen stabilen Rhythmus und eine konstante Produktivität aufrechterhält.
4. Erstellung von Tests und Produktionscode, die ausreichend voneinander entkoppelt sind, um eine bequeme Wartung beider zu ermöglichen, ohne dass Änderungen zwischen beiden repliziert werden müssen.

Die Praktik TDD ist in drei völlig willkürlichen Gesetzen verankert. Dass diese Gesetze willkürlich sind, wird auch dadurch bewiesen, dass der wesentliche Teil von TDD auch mit anderen Mitteln erreicht werden kann. Insbesondere durch die Praktik »test && commit || revert« (TCR) von Kent Beck. Obwohl sich TCR völlig von TDD unterscheidet, erreicht es genau dieselben wesentlichen Ziele.

Die drei Gesetze von TDD bilden die Grundlage der Praktik. Sie zu befolgen, ist sehr schwer, besonders am Anfang. Es erfordert auch einige Fähigkeiten und Kenntnisse, die nur schwer zu erlangen sind. Wenn Sie versuchen, die Gesetze ohne diese Fähigkeiten und Kenntnisse zu befolgen, werden Sie mit Sicherheit frustriert werden und die Praktik aufgeben. Wir werden diese Fähigkeiten und Kenntnisse in späteren Kapiteln behandeln. Für den Moment seien Sie gewarnt, dass das Befolgen dieser Gesetze ohne angemessene Vorbereitung sehr schwierig ist.

Das erste Gesetz

> *Schreiben Sie keinen produktiven Code, bevor Sie nicht einen Test geschrieben haben, der fehlschlägt, weil der produktive Code fehlt.*

Wenn Sie ein Programmierer mit einigen Jahren Erfahrung sind, mag Ihnen dieses Gesetz töricht vorkommen. Sie fragen sich vielleicht, welchen Test Sie schreiben sollen, wenn es keinen Code zu testen gibt. Diese Einstellung kommt von der allgemeinen Erwartung, dass Tests *nach* dem Code geschrieben werden. Aber

wenn Sie genauer darüber nachdenken, werden Sie feststellen, dass Sie den Test schreiben können, wenn Sie den produktiven Code schreiben können. Es mag unlogisch erscheinen, aber Ihnen fehlen keine Informationen, die Sie brauchen, um zuerst den Test zu schreiben.

Das zweite Gesetz

> *Schreiben Sie nicht mehr von einem Test, als nötig ist, damit der Test fehlschlägt oder nicht kompiliert. Beheben Sie dann den Fehler, indem Sie etwas produktiven Code schreiben.*

Wenn Sie ein erfahrener Programmierer sind, werden Sie wahrscheinlich wissen, dass die allererste Zeile des Tests nicht kompiliert werden kann, weil sie geschrieben wird, um mit Code zu interagieren, der noch gar nicht existiert. Und das bedeutet, dass Sie nicht in der Lage sein werden, mehr als eine Zeile eines Tests zu schreiben, bevor Sie zum Schreiben von produktivem Code übergehen müssen.

Das dritte Gesetz

> *Schreiben Sie nicht mehr produktiven Code, als zur Lösung des aktuell fehlgeschlagenen Tests nötig ist. Sobald der Test nicht mehr fehlschlägt, schreiben Sie mehr Testcode.*

Und damit ist der Kreislauf geschlossen. Ihnen sollte klar sein, dass diese drei Gesetze Sie in einen Zyklus zwingen, der nur ein paar Sekunden dauert und folgendermaßen aussieht:

- Sie schreiben eine Zeile Testcode, der sich (natürlich) nicht kompilieren lässt.
- Sie schreiben eine Zeile produktiven Code, damit der Test kompiliert.
- Sie schreiben eine weitere Zeile Testcode, die sich nicht kompilieren lässt.
- Sie schreiben ein oder zwei weitere Zeilen produktiven Code, damit der Test kompiliert.
- Sie schreiben ein oder zwei weitere Zeilen Testcode, der kompiliert, aber eine Testbedingung nicht erfüllt.
- Sie schreiben noch ein oder zwei Zeilen produktiven Code, damit die Testbedingung erfüllt wird.

Und das ist von nun an Ihr Leben.

Auch das werden die erfahrenen Programmierer unter Ihnen wahrscheinlich für unsinnig halten. Die drei Gesetze sperren Sie in einen nur wenige Sekunden andauernden Zyklus.

Jedes Mal, wenn Sie diesen Zyklus durchlaufen, wechseln Sie zwischen Testcode und produktivem Code. Sie werden nie einfach eine `if`-Anweisung oder eine

`while`-Schleife schreiben können. Sie werden nie einfach eine Funktion schreiben können. Sie werden für immer in dieser winzigen Schleife gefangen sein, in der Sie zwischen Testcode und produktivem Code hin und her wechseln.

Sie denken vielleicht, dass das mühsam, langweilig und langsam ist. Sie denken vielleicht, dass es Ihren Fortschritt behindert und Ihre Gedankengänge unterbricht. Vielleicht denken Sie sogar, dass es einfach nur töricht ist. Sie denken vielleicht, dass dieser Ansatz dazu führt, dass Sie Spaghetticode oder Code mit wenig oder gar keinem Design produzieren – ein zufälliges Sammelsurium von Tests und Code, der dafür sorgt, dass diese Tests erfolgreich durchlaufen.

Behalten Sie diese Gedanken im Hinterkopf und berücksichtigen Sie, was folgt.

Das Debug-Foo verlieren

Stellen Sie sich einen Raum voller Menschen vor, die diesen drei Gesetzen folgen – ein Team von Entwicklern, die alle an der Einführung eines großen Systems arbeiten. Wählen Sie zu einem beliebigen Zeitpunkt einen beliebigen Programmierer aus. Alles, woran dieser Programmierer gearbeitet hat, wurde innerhalb der letzten Minuten ausgeführt und hat alle Tests bestanden. Und das ist immer der Fall. Es spielt keine Rolle, wen Sie auswählen. Es spielt auch keine Rolle, wann Sie ihn auswählen. Alles hat vor wenigen Minuten noch funktioniert.

Wie würde Ihr Leben aussehen, wenn alles vor wenigen Minuten funktioniert hätte? Was meinen Sie, wie viel Debugging Sie dann noch brauchen? Tatsache ist, dass es wahrscheinlich nicht viel zu debuggen gibt, wenn alles vor wenigen Minuten noch funktioniert hat.

Kennen Sie sich mit einem Debugger aus? Haben Sie das Debug-Foo in Ihren Fingern? Haben Sie alle Tastenkombinationen im Kopf? Ist es für Sie eine Selbstverständlichkeit, Breakpoints und Watchpoints zu setzen und sich kopfüber in eine intensive Debugging-Sitzung zu stürzen?

Das ist keine besonders erstrebenswerte Fähigkeit!

Sie wollen nicht gut im Debuggen sein. Sie werden nur dann gut im Debuggen, wenn Sie viel Zeit mit dem Debuggen verbringen. Und ich will nicht, dass Sie viel Zeit mit dem Debuggen verbringen. Und Sie sollten das auch nicht. Ich möchte, dass Sie so viel Zeit wie möglich damit verbringen, Code zu schreiben, der funktioniert, und so wenig Zeit wie möglich damit, Code zu reparieren, der nicht funktioniert.

Ich möchte, dass Sie den Debugger so selten benutzen, dass Sie die Tastenkombinationen vergessen und den Debug-Foo in Ihren Fingern verlieren. Ich möchte, dass Sie über die obskuren Step-into- und Step-over-Symbole rätseln. Ich möchte, dass Sie im Umgang mit dem Debugger so ungeübt sind, dass sich der Debugger unbeholfen und langsam anfühlt. Und Sie sollten das auch wollen. Je wohler Sie

sich mit dem Debugger fühlen, desto sicherer können Sie sein, dass Sie etwas falsch machen.

Ich kann Ihnen nicht versprechen, dass diese drei Gesetze den Einsatz von Debuggern überflüssig machen. Sie werden trotzdem von Zeit zu Zeit debuggen müssen. Es handelt sich immer noch um Software, und es ist immer noch kompliziert. Aber die Häufigkeit und Dauer Ihrer Debugging-Sitzungen werden drastisch abnehmen. Sie werden viel mehr Zeit damit verbringen, Code zu schreiben, der funktioniert, und viel weniger Zeit damit, Code zu reparieren, der nicht funktioniert.

Dokumentation

Wenn Sie schon einmal ein Paket eines Drittanbieters integriert haben, wissen sie, dass in dem Softwarepaket, das sie erhalten, ein von einem technischen Redakteur verfasstes PDF-Dokument enthalten ist. In diesem Dokument wird angeblich beschrieben, wie man das Drittanbieterpaket integriert. Am Ende dieses Dokuments befindet sich fast immer ein unschöner Anhang, der alle *Codebeispiele* für die Integration des Pakets enthält.

Natürlich ist dieser Anhang der erste Ort, an dem Sie nachsehen. Sie wollen nicht lesen, was ein technischer Redakteur *über* den Code geschrieben hat; Sie wollen den Code selbst lesen. Und dieser Code wird Ihnen viel mehr verraten als die Worte, die der technische Redakteur geschrieben hat. Wenn sie Glück haben, können Sie den Code sogar per Copy&Paste in Ihre Anwendung übertragen, wo Sie ihn so hinfummeln können, dass er läuft.

Wenn Sie die drei Gesetze befolgen, schreiben Sie bereits die *Codebeispiele* für das gesamte System. Die Tests, die Sie schreiben, erklären jedes kleine Detail darüber, wie das System funktioniert. Wenn Sie wissen wollen, wie man ein bestimmtes Geschäftsobjekt erstellt, gibt es Tests, die Ihnen zeigen, wie man es auf alle möglichen Arten erstellt. Wenn Sie wissen wollen, wie man eine bestimmte API-Funktion aufruft, gibt es Tests, die diese API-Funktion und alle möglichen Fehlerbedingungen und Ausnahmen demonstrieren. Es gibt Tests in der Testsuite, die Ihnen alles sagen, was Sie über die Details des Systems wissen wollen.

Diese Tests sind Dokumente, die das gesamte System auf der untersten Ebene beschreiben. Diese Dokumente sind in einer Sprache geschrieben, die Sie sehr gut verstehen. Sie sind absolut eindeutig. Sie sind so formal, dass sie ausgeführt werden können. Und sie passen immer zum System.

Als Dokumentation sind sie nahezu perfekt.

Ich will das nicht überbewerten. Tests können nicht besonders gut die Motivation eines Systems zu beschreiben. Sie sind keine Abstraktion. Aber auf der Detail-Ebene sind sie besser als jede andere Art von Dokument. Sie sind Code. Und Sie wissen, dass Code Ihnen die Wahrheit sagt.

Vielleicht befürchten Sie, dass die Tests genauso schwer zu verstehen sind wie das System als Ganzes. Aber das ist nicht der Fall. Jeder Test ist ein kleiner Codeschnipsel, der sich auf einen sehr kleinen Teil des kompletten Systems konzentriert. Die Tests bilden für sich genommen kein System. Die Tests wissen nichts voneinander, und so gibt es keinen Rattenschwanz von Abhängigkeiten in den Tests. Jeder Test steht für sich allein. Jeder Test ist für sich selbst verständlich. Jeder Test zeigt Ihnen genau das, was Sie innerhalb eines sehr kleinen Teils des Systems verstehen müssen.

Auch diesen Punkt möchte ich nicht überbewerten. Es ist möglich, undurchsichtige und komplexe Tests zu schreiben, die schwer zu lesen und zu verstehen sind, aber es ist nicht notwendig. Tatsächlich ist es eines der Ziele dieses Buchs, Ihnen beizubringen, wie Sie Tests schreiben, die klare und saubere Dokumente sind, die das zugrundeliegende System beschreiben.

Löcher im Design

Haben Sie jemals Tests im Nachhinein geschrieben? Die meisten von uns haben das. Tests nach dem Code zu schreiben, ist die häufigste Art, wie Tests geschrieben werden. Aber das macht keinen Spaß, oder?

Es macht keinen Spaß, weil wir nachträglich Tests schreiben und zu dem Zeitpunkt bereits wissen, dass das System funktioniert. Wir haben es manuell getestet. Wir schreiben die Tests nur aus einem gewissen Pflicht- oder Schuldgefühl heraus oder vielleicht, weil unser Management ein bestimmtes Maß an Testabdeckung vorgeschrieben hat. Also schreiben wir widerwillig einen Test nach dem anderen, wohl wissend, dass jeder Test, den wir schreiben, funktionieren wird. Langweilig, langweilig, langweilig.

Wir werden unweigerlich zu dem Test kommen, der schwer zu schreiben ist. Er ist schwer zu schreiben, weil wir den Code nicht so entworfen haben, dass er testbar ist; wir haben uns stattdessen darauf konzentriert, ihn zum Laufen zu bringen. Um den Code zu testen, müssen wir jetzt das Design ändern.

Aber das ist mühsam. Es wird eine Menge Zeit in Anspruch nehmen. Dabei könnte etwas anderes kaputtgehen. Und wir wissen bereits, dass der Code funktioniert, weil wir ihn manuell getestet haben. Folglich lassen wir den Test weg und hinterlassen eine Lücke in der Testsuite. Sagen Sie mir nicht, dass Sie das noch nie gemacht haben. Sie wissen, dass Sie es getan haben.

Sie wissen auch, dass, wenn Sie eine Lücke in der Testsuite hinterlassen haben, alle anderen im Team es auch getan haben, also wissen Sie, dass die Testsuite voller Löcher ist.

Die Anzahl der Löcher in der Testsuite lässt sich ermitteln, indem man die Lautstärke und Dauer des Lachens der Programmierer misst, wenn die Testsuite erfolg-

reich durchläuft. Wenn die Programmierer viel lachen, dann hat die Testsuite eine Menge Löcher.

Eine Testsuite, die zum Lachen anregt, wenn sie erfolgreich durchläuft, ist keine besonders nützliche Testsuite. Sie sagt Ihnen vielleicht, wenn bestimmte Dinge nicht funktionieren, aber Sie können keine Entscheidungen treffen, wenn sie gelaufen ist. Wenn sie erfolgreich gelaufen ist, wissen Sie nur, dass einige Dinge funktionieren.

Eine gute Testsuite hat keine Löcher. Eine gute Testsuite erlaubt es Ihnen, Entscheidungen zu treffen, wenn sie erfolgreich durchgelaufen. Diese Entscheidung ist zu *deployen*.

Wenn die Testsuite erfolgreich ist, können Sie getrost empfehlen, das System zu deployen. Wenn Ihre Testsuite dieses Vertrauen nicht weckt, wozu ist sie dann gut?

Spaß

Wenn Sie die drei Gesetze befolgen, passiert etwas ganz anderes. Zunächst einmal macht es Spaß. Noch einmal, ich will das nicht überbewerten. TDD macht nicht so viel Spaß wie der Gewinn des Jackpots in Las Vegas. Es macht auch nicht so viel Spaß, wie auf eine Party zu gehen oder mit einem Vierjährigen das Leiterspiel[3] zu spielen. In der Tat ist *Spaß* vielleicht nicht das richtige Wort dafür.

Erinnern Sie sich noch daran, als Sie Ihr allererstes Programm zum Laufen gebracht haben? Erinnern Sie sich an dieses Gefühl? Vielleicht war es in einem örtlichen Kaufhaus, das einen TRS-80 oder einen Commodore 64 hatte. Vielleicht haben Sie eine dumme kleine Endlosschleife geschrieben, die Ihren Namen für immer und ewig auf dem Bildschirm ausgab. Vielleicht sind Sie mit einem kleinen Lächeln auf dem Gesicht von diesem Bildschirm weggegangen, weil Sie wussten, dass Sie der Herr des Universums sind und dass sich alle Computer für immer vor Ihnen verbeugen würden.

Ein winziges Echo dieses Gefühls erleben Sie jedes Mal, wenn Sie die TDD-Schleife durchlaufen. Jeder Test, der genauso scheitert, wie Sie es erwartet haben, lässt Sie nicken und ein kleines bisschen lächeln. Jedes Mal, wenn Sie den Code schreiben, der den fehlgeschlagenen Test erfolgreich laufen lässt, erinnern Sie sich daran, dass Sie einmal Herr des Universums waren und dass Sie immer noch *die Macht* haben.

Jedes Mal, wenn Sie die TDD-Schleife durchlaufen, wird ein winziger Schuss Endorphine in Ihr Reptiliengehirn ausgeschüttet, wodurch Sie sich ein wenig kompetenter und selbstbewusster fühlen und bereit sind, die nächste Herausforde-

3 Anmerkung des Übersetzers: Im engl. Sprachraum ist *Chutes and Ladders* ein beliebtes Brettspiel für Vorschulkinder

rung anzunehmen. Und obwohl dieses Gefühl nur winzig ist, macht es doch irgendwie Spaß.

Design

Aber vergessen Sie den Spaß. Etwas viel Wichtigeres passiert, wenn Sie zuerst die Tests schreiben. Es stellt sich heraus, dass man keinen schwer zu testenden Code schreiben kann, wenn man die Tests zuerst schreibt. Dadurch, dass Sie die Tests zuerst schreiben, sind Sie gezwungen, den Code so zu gestalten, dass er leicht zu testen ist. Es gibt kein Entrinnen. Wenn Sie die drei Gesetze befolgen, wird Ihr Code leicht zu testen sein.

Was macht Code schwer zu testen? Kopplungen und Abhängigkeiten. Leicht zu testender Code weist diese Kopplungen und Abhängigkeiten nicht auf. Leicht zu testender Code ist entkoppelt!

Die Befolgung der drei Gesetze zwingt Sie dazu, entkoppelten Code zu schreiben. Auch hier gibt es kein Entkommen. Wenn Sie zuerst die Tests schreiben, wird der Code, der diese Tests besteht, auf eine Weise entkoppelt sein, die Sie nie für möglich gehalten hätten.

Und das ist eine sehr gute Sache.

Das Beste zum Schluss

Es stellt sich heraus, dass die Anwendung der drei Gesetze von TDD die folgenden Vorteile mit sich bringt:

- Sie werden mehr Zeit damit verbringen, Code zu schreiben, der funktioniert, und weniger Zeit damit, Code zu debuggen, der nicht funktioniert.
- Sie werden eine Sammlung nahezu perfekter Low-Level-Dokumentation produzieren.
- Es macht Spaß – oder ist zumindest motivierend.
- Sie werden eine Testsuite erstellen, die Ihnen das nötige Vertrauen für den produktiven Einsatz gibt.
- Sie werden loser gekoppelte Designs erstellen.

Diese Gründe könnten Sie davon überzeugen, dass TDD eine gute Sache ist. Sie könnten ausreichen, um Sie dazu zu bringen, Ihre anfängliche Reaktion oder gar Abneigung zu ignorieren. Vielleicht. Aber es gibt einen viel wichtigeren Grund, warum die Praktik TDD so wichtig ist.

Furcht

Programmieren ist schwer. Es ist vielleicht das Schwierigste, was der Mensch je zu meistern versucht hat. Unsere Zivilisation hängt heute von Hunderttausenden miteinander verbundener Softwareanwendungen ab, von denen jede Hunderttau-

sende, wenn nicht gar 10 Millionen Codezeilen umfasst. Es gibt keinen anderen von Menschen konstruierten Apparat, der so viele bewegliche Teile hat.

Jede dieser Anwendungen wird durch Entwicklerteams gewartet, die sich zu Tode fürchten, etwas zu ändern. Das ist ironisch, denn Software gibt es, damit wir das Verhalten unserer Maschinen leicht ändern können.

Aber Softwareentwickler wissen, dass jede Änderung das Risiko von Fehlern mit sich bringt und dass diese schwer zu finden und zu beheben sind.

Stellen Sie sich vor, Sie blicken auf Ihren Bildschirm und sehen dort einen unschönen, verschachtelten Code. Sie müssen sich wahrscheinlich nicht sehr anstrengen, um sich dieses Bild vorzustellen, denn für die meisten von uns ist dies eine alltägliche Erfahrung.

Nehmen wir nun an, dass Ihnen beim Anblick dieses Codes für einen kurzen Moment der Gedanke kommt, dass Sie ihn ein wenig aufräumen sollten. Aber schon schlägt der nächste Gedanke zu wie Thors Hammer: »ICH FASSE IHN AUF KEINEN FALL AN!« Denn Sie wissen, wenn Sie ihn anfassen, machen Sie ihn kaputt; und wenn Sie ihn kaputt machen, gehört er *für immer Ihnen*.

Das ist eine Angstreaktion. Sie fürchten den Code, den Sie warten. Sie fürchten die Konsequenzen, wenn Sie ihn kaputt machen.

Das Ergebnis dieser Angst ist, dass der Code verrotten wird. Keiner wird ihn aufräumen. Keiner wird ihn verbessern. Wenn man gezwungen ist, Änderungen vorzunehmen, werden diese so vorgenommen, wie es für den Programmierer am sichersten ist und nicht, wie es für das System am besten wäre. Das Design wird sich verschlechtern, der Code wird verrotten, und die Produktivität des Teams wird sinken, und dieser Rückgang wird sich fortsetzen, bis die Produktivität auf nahezu null sinkt.

Fragen Sie sich selbst, ob Sie jemals durch schlechten Code in Ihrem System erheblich aufgehalten wurden. Natürlich wurden Sie das. Und jetzt wissen sie auch, warum es diesen schlechten Code überhaupt gibt. Er existiert, weil niemand den Mut hatte, die einzige Sache zu tun, die ihn verbessern könnte. Keiner wagt es, ihn aufzuräumen.

Mut

Was aber wäre, wenn Sie eine Testsuite hätten, der Sie so sehr vertrauen, dass Sie jedes Mal, wenn Sie diese Testsuite erfolgreich ausgeführt haben, das Deployment empfehlen könnten? Und was wäre, wenn diese Testsuite in Sekundenschnelle ausgeführt werden könnte? Wie sehr würden Sie sich dann davor fürchten, das System vorsichtig aufzuräumen?

Stellen Sie sich diesen Code noch einmal auf Ihrem Bildschirm vor. Stellen Sie sich den Gedanken vor, dass Sie Ihn ein wenig bereinigen könnten. Was würde Sie

davon abhalten? Sie haben doch die Tests. Diese Tests werden Ihnen sofort sagen, wenn Sie etwas kaputt gemacht haben.

Mit dieser Testsuite können Sie den Code sicher aufräumen. Mit dieser Testsuite können Sie den Code *sicher* aufräumen. *Mit dieser Testreihe können Sie den Code sicher aufräumen*.

Nein, das war kein Tippfehler. Ich wollte den Punkt sehr, sehr deutlich herausstellen. Mit dieser Testsuite können Sie den Code sicher aufräumen!

Und wenn Sie den Code sicher aufräumen können, *werden* Sie den Code auch aufräumen. Und auch jeder andere im Team wird es tun. Denn niemand mag Unordnung.

Die Pfadfinder-Regel

Wenn Sie eine Testsuite haben, der Sie Ihr berufliches Leben anvertrauen würden, dann können Sie diese einfache Richtlinie sicher befolgen:

> *Checken Sie Code sauberer ein, als Sie ihn ausgecheckt haben*.

Stellen Sie sich vor, jeder würde das tun. Bevor Sie den Code einchecken, vollführen Sie am Code einen kleinen Akt der Freundlichkeit. Sie räumen ihn ein kleines bisschen auf.

Stellen Sie sich vor, dass jeder Check-in den Code sauberer macht. Stellen sie sich vor, niemand würde den Code jemals schlechter einchecken, sondern immer etwas besser als er war.

Wie würde es sein, ein solches System zu pflegen? Was würde mit Schätzungen und Zeitplänen passieren, wenn das System mit der Zeit immer sauberer wird? Wie lang würden Ihre Fehlerlisten sein? Bräuchten Sie noch eine automatisierte Datenbank, um diese Fehlerlisten zu pflegen?

Das ist der Grund

Den Code sauber zu halten. Kontinuierliches Aufräumen des Codes. Das ist der Grund, warum wir TDD praktizieren. Wir praktizieren TDD, damit wir stolz auf die Arbeit sein können, die wir leisten. Damit wir uns den Code ansehen können und wissen, dass er sauber ist. Damit wir wissen, dass er jedes Mal, wenn wir ihn anfassen, besser wird als zuvor. Und damit wir abends nach Hause gehen, in den Spiegel schauen und lächeln, weil wir wissen, dass wir heute gute Arbeit geleistet haben.

2.1.3 Das vierte Gesetz

Ich werde in späteren Kapiteln noch viel mehr über Refactoring sagen. Für den Moment möchte ich lediglich anmerken, dass Refactoring das vierte Gesetz von TDD ist.