binaris informatik GmbH

Test Driven Development in 6 einfachen Schritten

Wenn Software produziert wird, dann existiert in der Regel die Absicht, ein tolles Produkt mit einer hervorragenden technischen Qualität zu erzeugen. Und diese hervorragende technische Qualität mit einer möglichst umfassenden Testabdeckung zu beweisen und abzusichern. Doch in erschreckend vielen Fällen werden die Punkte *hervorragende technische Qualität* und *umfassende Testabdeckung* deutlich verfehlt.

Auf der Suche nach einer Lösung, mit der genau das in Zukunft nicht mehr passieren soll, steht oft ein Begriff im Raum:

Test Driven Development

Dieses bestechend einfache Konzept, das die Reihenfolge von Programmierung und Testen umdreht, verspricht, die klassischen Probleme von mangelnder Testabdeckung und unzureichender technischer Qualität zu lösen. Ganz einfach.

„Test Driven Development ist nicht nur ‚erst Test, dann Programmierung', es ist ein Gesamtpaket"

Um es hier gleich vorweg zu verraten: Ja, Test Driven Development kann tatsächlich dieses scheinbare Wunder wirken. Allerdings nur, wenn wir Test Driven Development, oder auch TDD, wirklich verstehen. Inklusive allem, was rund um das Grundkonzept mit zum Gesamtpaket Test Driven Development gehört.

Wenn Unternehmen TDD einführen wollen und kurz danach wieder verwerfen, weil es bei ihnen nicht funktioniert oder nicht passt, dann liegt das oft daran, dass sie nicht den vollen Umfang von Test Driven Development verstanden und nur den offensichtlichen Teil umgesetzt haben.

Wenn wir uns TDD genauer ansehen, dann stellen wir fest, dass bevor wir zum Kernelement kommen, also die Reihenfolge Implementierung/Test umzudrehen, wir erst ein paar Hausaufgaben zu machen haben. Und dass wir auch nach der Umsetzung des Kernelements noch einen Schritt weitergehen müssen, um wirklichen Erfolg haben zu können.

Aber keine Angst, in diesem einfachen 6-Schritte-Programm zeigen wir Ihnen, wie Sie Test Driven Development erfolgreich in Ihrem Unternehmen oder Projekt einführen und damit in Zukunft Ihre Erwartungen an technische Qualität und Testabdeckung erfüllen können.

1. Schritt Die Motivation zum Testen klären

Der erste Schritt besteht darin, zu klären was Tests eigentlich sind oder besser, warum wir eigentlich testen wollen.

„Was müssen wir denn über den Grund für Testen reden, der ist doch klar. Ich will wissen, ob mein Programm auch funktioniert!" mag sich da so mancher nun denken. Und so haben es viele von uns auch ursprünglich mal gelernt. Wir schreiben ein Stück Software und dann überprüfen wir, ob es auch funktioniert. Fertig!

„Es gibt drei Gründe zum Testen"

Doch die Realität ist dann doch ein bisschen komplexer. In unserer Wahrnehmung gibt es drei Gründe zum Testen bzw. zum Schreiben von Tests:

1. Wir suchen die Ursache für einen Fehler und testen deshalb das Verhalten eines gegebenen Systems
2. Wir haben ein System oder einen Teil eines Systems erstellt und wollen nun die einwandfreie Funktionsweise unserer Schöpfung beweisen und sicherstellen
3. Wir haben vor, ein System oder einen Teil eines Systems zu erstellen und definieren über Tests bereits im Vorfeld unsere Erwartungshaltung, damit wir genau das Richtige implementieren

Für jede dieser Intentionen gibt es verschiedene Arten von Test und Vorgehensweisen, die den angestrebten Zweck befrieden können. Und abhängig davon, was wir eigentlich mit unserem Testen erreichen wollen, müssen oder können wir die dazu passenden Formen von Tests und Vorgehensweisen wählen.

Klingt sehr trivial und ist es auch. Und vielleicht ist es gerade deshalb so wichtig, diesen Schritt bewusst zu gehen. Wenn wir ein neues Produkt erstellen oder ein bereits bestehendes um neue Funktionalitäten erweitern wollen, dann ist vor allem der dritte Grund unsere Hauptmotivation kombiniert mit dem zweiten.

Wer jetzt allerdings argumentiert, dass eindeutig der zweite Grund die Hauptmotivation zum Testen bei einer Neuimplementierung ist, der ist noch viel zu stark in der Trennung vom Implementierung und Testen verhaftet. Erst hinterher durch Tests die korrekte Funktionsweise einer Software sicherstellen zu wollen ist der Grund, warum in vielen Unternehmen die Qualitätssicherung nichts mit der Produktentwicklung zu tun hat, warum Testen als ein separater, von der Programmierung getrennter Schritt verstanden wird. Das ist die Absicht, erst hinterher festzustellen, dass „es“ nicht funktioniert und dann noch zu versuchen, das Produkt zu retten.

Wir dahingehen wollen Qualitätssicherung oder Qualitätsmanagement als Bestandteil der Produktentwicklung sehen. Wir wollen direkt das funktionierende Produkt bauen.

2. Schritt Die Eigenschaften guter Software erkennen

Das führt uns direkt zum zweiten Schritt in unserem Programm, die technischen Eigenschaften guter Software. Gute, wir sagen manchmal robuste, Softwarekomponenten zeichnen sich durch folgende vier Eigenschaften aus:

- Lauffähigkeit
- Einfachheit
- Relevanz
- Redundanzfreiheit

Was bedeutet das? In erster Linie wollen wir, dass unsere Software funktioniert. Und zwar fehlerfrei. Das ist schon mal klar verständlich.
Dann möchten wir aus Gründen der Wartbarkeit, dass sie möglichst einfach ist bzw. das alle einzelnen Komponenten möglichst simple und leicht verständlich sind. Jeder Softwareentwickler, der schon mal in eher schwer verständlichen Sourcecode Änderungen einbauen musste, ist von diesem Punkt direkt ein großer Fan. Und ebenso jeder Projektverantwortliche, der für eine Weiterentwicklung schon mal astronomische Aufwandsschätzungen zu hören bekam mit der Begründung, der aktuelle Softwarestand wäre schwer zu verstehen und deshalb schwer zu erweitern.
Und natürlich ist es eine gute Idee, wenn jedes Stück Sourcecode auch wirklich einen relevanten Teil zur Lösung des Problems beiträgt. Unnützer oder toter Code reduziert nur die Wartbarkeit.
Das gleiche gilt auch für die Freiheit von Redundanzen, also doppeltem Code, auch diese erhöht die Wartbarkeit und reduziert das Fehlerrisiko.

Relevanz und Redundanzfreiheit sind in dieser Liste Begriffe, die in sich schon sehr gut erklären, wann sie tatsächlich gegeben sind. Wenn jedes Stück Sourcecode auch wirklich benutzt wird, dann ist alles relevant und wenn es nirgendwo identische Codestellen gibt, dann ist die Software auch redundanzfrei.

Spannender wird es schon der Lauffähigkeit und der Einfachheit, denn für beides ist eine genauere Definition nötig. Die Lauffähigkeit lässt sich gut durch Akzeptanzkriterien definieren oder auch eine sogenannte *Definition of Done*, in der die Beteiligten gemeinsam festhalten, was es bedeutet, dass eine Funktionalität oder ein Feature

fertig ist.
Ähnliches gilt für Einfachheit, auch diese Worthülse darf mit einer gemeinsamen Definition mit echtem Leben gefüllt werden, z.B. mit Codemetriken, Style Guides oder gemeinsamen Architekturkonzepten.

3. Schritt Das Schreiben echter Unit Tests lernen

Im nächsten Schritt dürfen wir lernen, was echte Unit Tests sind und wie sie geschrieben werden.
Wie wir jetzt auf Unit Tests kommen? Wie wir gleich sehen werden, sind Unit Tests die unterste und elementarste Ebene der technischen Tests und eine Grundlage beim Test Driven Development. Deshalb beschäftigen wir uns nun in diesem Schritt mit diesen Unit Tests.

„Wir verstehen nicht alle dasselbe unter diesen Begriffen!“

Bei dem Begriff *Unit Test* haben wir ähnliche Herausforderungen wie bei den Begriffen Lauffähigkeit und Einfachheit: Jeder hat den Begriff schon gehört und auch eine Vorstellung davon, was diese bedeuten. Nur bedeutet das noch lange nicht, dass wir auch alle das Gleiche darunter verstehen.
Definitionen, die wir oft von unseren Kunden zu hören bekommen sind z.B. diese:

„Unit Tests sind die Entwicklertests, also alles, was die Entwickler testen“
„Mit Unit Tests testen wir einzelne Einheiten“
„Unit Tests sind Klassentests“
„Unit Tests testen die Funktionen von Klassen“
„Unit Tests sind die automatisierten Tests“
„Unit Tests sind Tests, die mit JUnit geschrieben wurden“

Jede dieser Definitionen ist irgendwie nachvollziehbar und doch ist keine davon so richtig hilfreich. Gehen wir also in die genaue Klärung des Begriffs *Unit Test*.

In der Liste der Motivationen zum Testen sind Unit Tests ein geeignetes Mittel für die Punkte 2 und 3. Um diese Behauptung überprüfen zu können, betrachten wir, was denn ein Unit Test eigentlich sein soll. Unser erster Versuch einer Definition lautet wie folgt:

„Unit Tests klammern Abhängigkeiten zu anderen Komponenten aus“

Unit Tests sind ein Verfahren, um das Verhalten einer Komponente isoliert, d.h. ohne ihre Abhängigkeiten zu anderen Komponenten, zu überprüfen.

Das bedeutet, wir wollen ein bestimmtes Verhalten in einem klar definierten Kontext überprüfen und dabei jede Form von Seiteneffekten vermeiden. Deshalb klammern wir die Abhängigkeiten zu anderen Komponenten so explizit aus. Jede weitere Komponente, von der unser zu testendes Verhalten abhängig ist, bedeutet einen weiteren Faktor, den wir kontrollieren müssen bzw. der das Ergebnis unseres Tests beeinflussen kann. Wie können wir nun diesen Anspruch auf isoliertes Testen in unseren Tests umsetzen?

Sehen wir uns dazu ein Beispiel in Java an. In einem Baseballszenario wirft ein Pitcher einen Ball, den ein Spieler zu treffen versucht. Wir möchten einen Unit Test für die Methode hitBaseball() schreiben, bei der der Spieler einen durchschnittlich geworfenen Ball erfolgreich treffen soll. Spontan sähe unser Test so aus:

Codebeispiel erster Versuch

```
@Test
Public void testPlayerHitBallSuccessfully() {
       BaseballPlayer player = new BaseballPlayer(„Mike“);

       Baseball baseball = new Pitcher(„Tom“).throw(„average“);

       assertTrue(„Ball verfehlt“, player.hitBaseball(baseball));
}
```

Der Pitcher erzeugt beim Werfen einen Ball, der alle Attribute eines Balles hat, der durchschnittlich geworfen wurden. Der Spieler versucht den Ball zu treffen und wir gehen eigentlich davon aus, dass er das müsste. Das ist unsere Erwartungshaltung.

Ist das nun ein Unit Test gemäß unserer Definition oben? Ganz klar nein! Neben der Methode hitBaseball, dem Verhalten, dass wir eigentlich testen wollen, testen wir hier auch direkt noch throw(x) von Pitcher mit und die interne Implementierung von Baseball. Wenn dieser Test fehlschlägt, dann ist nicht deutlich ersichtlich, woran das liegen könnte, die Anzahl der Codestellen, die sich unerwartet verhalten ist einfach zu groß.

Probieren wir es mal anders:

Codebeispiel überarbeiteter Versuch

```
@Test
Public void testPlayerHitBallSuccessfully() {
        BaseballPlayer player = new BaseballPlayer(„Mike“);

        Baseball baseball = Mockito.mock(Baseball.class);
        Baseball.when(baseball.getSpeed()).thenReturn(90);

        assertTrue(„Ball verfehlt“, player.hitBaseball(baseball));
}
```

Was ist jetzt anders?

Um die Methode hitBaseball testen zu können, benötigen wir einen Baseball. Nur lassen wir uns den diesmal nicht von einem Pitcher, den wir ja hier überhaupt nicht testen wollen, erzeugen. Stattdessen erzeugen wir uns mit Hilfe eines Mockframeworks, hier Mockito, einen Mock, der so aussieht, wie ein Baseball.

Erzeugung des Baseball Mocks

```
        Baseball baseball = Mockito.mock(Baseball.class);
```

Wir brauchen jetzt keinen Pitcher mehr und sind damit in unserem Test auch nicht mehr davon abhängig, dass der Pitcher sich wirklich wie erwartet verhält. Durch den Mock haben wir uns selber etwas erzeugt, dass wie ein Baseball aussieht. Allerdings müssen wir nun noch sicherstellen, dass sich dieser Mock auch wie ein durchschnittlich geworfener Baseball verhält.
In diesem Beispiel wissen wir aus der Implementierung vom BaseballPlayer, dass wir den Ball nach seiner Geschwindigkeit fragen werden. Also erklären wir dem Mock, dass er 90 (mph) zu antworten hat, wenn er nach seiner Geschwindigkeit gefragt wird.

Programmierung des Verhaltens des Baseball Mocks

```
        Baseball.when(baseball.getSpeed()).thenReturn(90);
```

Und schon testen wir exakt nur noch das Verhalten von hitBaseball(x).

„Ein Unit Test testet einen einzelnen Aspekt einer Komponente“

Wenn wir Tests auf diese Art und Weise schreiben, dann bekommen wir als Ergebnis eine ganze Menge an Tests, die jeweils nur ein ganz bestimmtes Verhalten einer Komponente testen. Wir sagen manchmal auch, ein Unit Test testet nur einen einzelnen Aspekt einer Komponente. Das verfeinert unsere Definition von Unit Test.

Aber warum tun wir uns das an? Es ist ja schon eine ganze Menge Arbeit, so viele so kleine Tests zu schreiben. Und es ist ja nicht nur die Menge der Tests, die nun zu schreiben sind. Auch die Implementierung selber ist so zu halten, dass diese einzelnen Aspekte überhaupt isoliert testbar sind.
Wir könnten doch auch einfach eine kleinere Anzahl an großen Tests schreiben, die direkt mehrere, oder sogar alle, Aspekte einer Komponente und das Zusammenspiel mit ihren abhängigen Komponenten testen, oder? Ja, könnten wir. Und genau das wollen wir nicht mehr.

Lassen Sie uns drei Motivationen anführen, warum wir statt dessen echte Unit Tests verwenden möchten:

„Irgendetwas stimmt nicht“

a. Wenn meine „großen“, komplexen Tests grün laufen, also kein Fehler auftritt, dann ist alles in Ordnung. Easy going. Aber wenn jetzt doch was nicht in Ordnung ist, was genau stimmt dann nicht. Mein „großer“ Test verrät es im schlimmsten Fall überhaupt nicht, die einzige Aussage wäre: „Irgendetwas

stimmt nicht!“ Und selbst bei einem besser strukturierten Test bleiben noch eine Reihe von potentiellen Fehlergründen offen, die wir nun manuell durchtesten müssten. Bis wir wüssten, was tatsächlich das Problem ist und es beheben könnten, müssten wir viel weitere Arbeit investieren. Haben wir viele einzelne Tests, die unabhängig voneinander einzelne Aspekte testen, zeigt uns der fehlschlagende Test direkt die Fehlerursache an. Fehlerfinden und -beheben wird damit zum Kinderspiel.

b. Ein größerer Testkontext besteht meistens aus einer größeren Anzahl an Parametern, Ausgangsbedingungen oder Ablaufmöglichkeiten, die unterschiedliche Ergebnisse erzielen können. Selbst wenn wir nur eine repräsentative Auswahl an realistischen Szenarien mit Tests abbilden möchten, wird durch die Komplexität der Abhängigkeiten die Anzahl der notwendigen Tests schnell groß, oft sogar sehr groß. Und wir reden hier von „größeren“ Tests, also höherem Aufwand pro Test. Haben Sie dadurch schon mal sehenden Auges eine geringere Testabdeckung in Kauf genommen, obwohl Sie ein schlechtes Gefühl dabei hatten? Seien Sie ehrlich! Die Reduzierung des Testkontextes auf den einzelnen Aspekt einer Komponente, reduziert damit auch die Komplexität der Tests. Und weil diese Test unabhängig voneinander sind, entsteht für die Testabdeckung nun auch nicht mehr das Kreuzprodukt aller möglichen Kombinationen. Für die Auswahl der übergeordneten Tests, die das Zusammenspiel der Aspekte testen, können wir uns nun auf eine wirklich repräsentative Auswahl realistischer Szenarien beschränken. Wir werden später noch über diese Art von Tests reden.

c. Was schrieben wir oben? Für Unit Tests muss die Implementierung so geschrieben sein, dass die einzelnen Aspekte überhaupt einzeln testbar sind. Das bedeutet nichts anderes, als ein wirklich modulares Design bis in den letzten Winkel und ein hohes Maß an loser Kopplung. Müssen wir hier wirklich noch über die Vorteile einer derartigen Architektur reden? Wohl kaum. Unit Tests zwingen uns also zum besseren technischen Design.

AAA-Regel

Um jetzt wirklich gute Unit Tests zu schreiben, gibt es ein paar Faustformeln, wie ein Unit Test aussehen sollte. Eine davon ist die AAA-Regel. AAA steht dabei für

- Arrange
- Act
- Assert

und beschreibt den Aufbau eines Unit Tests. Im **Arrange** wird die Startsituation des Tests definiert. Also die Komponente wird erzeugt und initialisiert, Mocks werden erzeugt, gesetzt und evtl. das Verhalten der Mocks definiert, Variablen werden gesetzt.

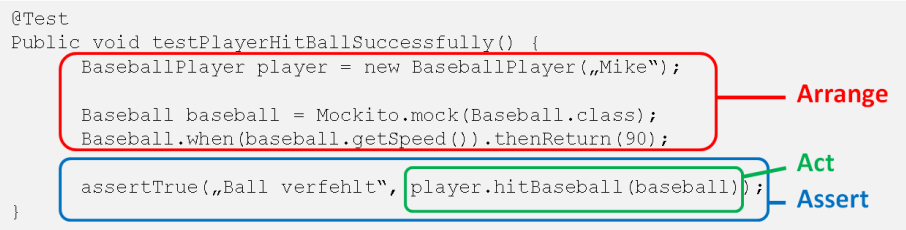
„Ein guter Unit Test besteht nur aus einem Aufruf“

Im **Act** wird die eigentliche Aktion, die das zu verifizierende Ergebnis erzeugen soll, durchgeführt. In der Regel ist das der Aufruf der Methode, deren Verhalten getestet werden soll. Ein guter Unit-Test besteht hier nur aus einem Aufruf. Ist mehr als ein Aufruf notwendig, dann ist das ein Hinweis darauf, dass mehr als nur ein einfacher Aspekt getestet wird!

Und schlussendlich wird im **Assert** nun überprüft, ob auch das erwartete Ergebnis durch das Act erzielt wurde. Hier gilt, dass nur ein einzelnes fachliches Ergebnis geprüft werden soll. Und dabei liegt die Betonung auf fachliches Ergebnis, nicht technisches. Das bedeutet also nicht zwangsläufig, dass dafür nur ein einzelnes Assert-Statement verwendet werden darf. Wenn über mehrere Vergleiche verschiedene Teilaspekte des fachlichen Ergebnisses überprüft werden müssen, dann ist das schon in Ordnung. Wenn allerdings z.B. ein Unit-Test TestInsertKunde im Assert prüft, ob der Kunde wirklich in der die Datenbank geschrieben UND ein entspre-

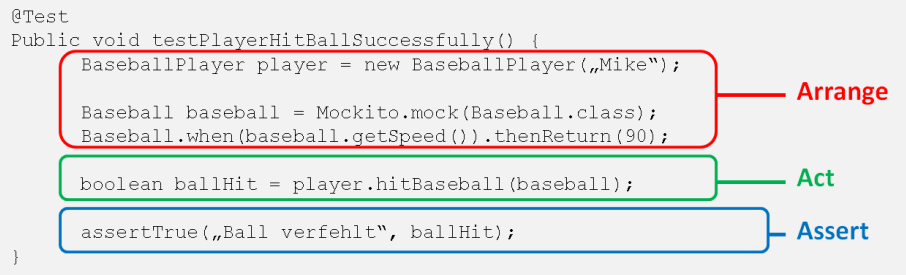
chender Logeintrag im Logfile erzeugt wurde, dann ist das ein Hinweis darauf, dass hier mehr als nur ein einzelner Aspekt getestet wurde. So simple und vielleicht auch nervig, wie die Einhaltung der AAA-Regel auch scheinen mag, sie hilft, echte Unit Tests zu identifizieren.

AAA-Regel in Codebeispiel



Wir sehen in dem Beispiel, dass hier Act und Assert in einer Zeile, quasi ineinander verschachtelt stehen. Hier könnte man nun argumentieren, dass für die bessere Lesbarkeit dieses zu trennen wäre, also den Schlag des Spielers in einer einzelnen Zeile und das Ergebnis daraus in einer lokalen Variable fangen und dann diese im Assert auswerten.

AAA-Regel in überarbeitetem Codebeispiel



Ob das nun der besseren Lesbarkeit und Wartbarkeit wegen wirklich notwendig ist, ist schwer objektiv zu beurteilen. Deshalb überlassen wir diese Entscheidung denen, die mit dem Code und den Tests arbeiten müssen.

FIRST

Ein anderes Akronym zur Identifizierung von guten Unit Tests ist FIRST, auch als FIRST Properties von Unit Tests bekannt. FIRST steht dabei für:

- Fast
- Isolated
- Repeatable
- Self-Validating
- Timely

Fast: Ein Unit Test soll schnell in der Ausführung sein. Warum? Nun, so granular wie Unit Tests sind, reden wir bei der Durchführung der Tests nicht von einem oder zwei Unit Tests, sondern im Laufe der Zeit wird die Anzahl eher auf ein paar Hundert Unit Tests ansteigen, vielleicht auch ein paar Tausend. Und der große Vorteil einer guten Codeabdeckung durch automatisierte Tests ist, dass wir bei Änderungen quasi durch einen Mausklick überprüfen können, ob noch alles funktioniert. Wenn der Durchlauf aller Tests aber ein oder zwei Stunden dauert, oder vielleicht noch länger, dann wird die Motivation abnehmen, diese Testläufe in kurzen Intervallen durchzuführen. Das geht wieder auf Kosten der Qualität, weshalb jeder einzelne Unit Test schnell sein muss, zwei bis drei Sekunden maximal. Und noch schneller wäre noch besser.

Isolated: Unit Tests sind unabhängig voneinander. Das bedeutet, die Durchführbarkeit eines Unit Tests hängt weder davon ab, dass ein anderer Test bereits gelaufen ist, noch dass ein anderer Test vorher ein bestimmtes Ergebnis erzielt hat. Jeder Unit Test erzeugt im Arrange selber die für ihn notwendige Startkonstellation.

Repeatable: Ein Test muss beliebig oft wiederholbar sein. Und zwar ohne dass zwischen zwei Testläufen manuell oder auch durch einen anderen Prozess (z.B. ein anderer Test) die im ersten erzeugten Zustände zurückgesetzt werden müssen. Oder anderes formuliert: Ein Unit Test hinterlässt keine persistenten Zustandsänderungen im System. Entweder erzeugt er erst gar keine solchen Zustandsänderungen oder er räumt sie selber zum Abschluss des Tests wieder auf, indem er z.B. Werte aus der Datenbank wieder löscht oder zurücksetzt. Das steht in einem engen Zusammenhang mit den Punkt Isolated.

Self-Validating: Ein Unit Test kann immer selber eineindeutig bestimmen, ob er erfolgreich war oder fehlgeschlagen ist. Für diese Beurteilung ist definitiv keine manuelle Bewertung durch einen Menschen erforderlich. Klingt blöd und eigentlich selbstverständlich. Und doch finden wir das immer mal wieder vor, dass erst einer der Entwickler oder Tester nochmal darauf schauen muss, um zu beurteilen, ob der Testlauf jetzt ok war oder eher nicht. Wenn das der Fall ist, dann stimmt mit der Grundidee der entsprechenden Test etwas nicht.

Timely: Unit Tests sind nicht besonders gut geeignet, um sie irgendwann mal später zu schreiben, Wochen oder Monate nachdem der Sourcecode geschrieben wurde. Quasi nur, um nachträglich noch eine gute Testabdeckung zu erreichen. Wie wir oben schon gesehen haben, hat alleine das Schreiben der Unit Tests einen großen Einfluss darauf, wie der Sourcecode strukturiert wird. Deshalb gehören die Erstellung von Sourcecode und Unit Tests unmittelbar zusammen. Das werden wir uns gleich noch genauer ansehen, wenn wir über die testgetriebene Entwicklung reden.

„Die Erstellung von Sourcecode und Unit Tests gehört unmittelbar zusammen“

Das sind eine Menge Faktoren, die für die Erstellung guter Unit Tests zu berücksichtigen sind. Und deshalb nochmal die Frage. Warum tun wir uns das an?

Nun, der konsequente Einsatz von Unit Tests hilft uns, Software zu schreiben, die die Kriterien guter, robuster Software erfüllen (siehe Schritt 2).

Also schreiben wir doch einfach konsequent Unit Tests. Doch jetzt gibt es beim klassischen Test-After Ansatz, das heißt ich schreibe erst meinen Sourcecode und erst danach die entsprechenden Tests, ein kleines Problem mit den Unit Tests. Wie wir oben gesehen haben, hat die Erstellung eines Unit Tests großen Einfluss auf die Struktur und das Design des eigentlichen Sourcecodes. Ich werde also vermutlich oft beim Schreiben eines Unit Tests feststellen, dass ich diesen Test so gar nicht schreiben kann, weil die Implementierung des Sourcecodes es nicht zulässt. Also zurück in den Sourcecode und diesen ändern. Und beim nächsten Unit Test wieder und wieder und wieder. Das kann schnell nervig werden und die Motivation reduzieren, weitere Unit Tests zu schreiben.

Wenn jedoch meine Unit Tests mir sowieso schon vorgeben wollen, wie ich den Sourcecode zu schreiben habe, warum dann nicht direkt die ganze Idee einfach umdrehen? Also erst den Test schreiben und dann dazu passend implementieren? Und schon sind wir mitten in der testgetriebenen Entwicklung gelandet, auf Neudeutsch auch Test Driven Development oder TDD genannt.

4. Schritt Test Driven Development Grundprinzip verstehen

Also ist TDD einfach nur zuerst den Test zu schreiben und dann den Sourcecode?

Ja … und nein. Tatsächlich ist die einfache Umkehrung der Reihenfolge die Grundidee des TDDs. Und schon alleine das wäre oft eine große Verbesserung im Hinblick auf Testqualität und -abdeckung. Doch die TDD Gurus sagen immer so schön: „TDD is a design process, not a testing process".
Es geht, genau wie bei den Unit Tests also nicht nur darum, eine gute Testqualität und -abdeckung zu erzielen, sondern wir wollen besseren Sourcecode schreiben.

„TDD is a design process, not a testing process"

Damit wir das erreichen können, gibt es da noch ein paar weiterführende Ideen im TDD, die wir uns jetzt mal ansehen wollen. Die erste grundlegende Idee ist, dass der TDD Zyklus nicht aus zwei, sondern aus drei Phasen besteht:

1. Schreibe einen Test
2. Implementiere gegen den Test
3. Räume auf

Zum ersten Schritt müssen wir eigentlich nicht mehr viel sagen. Obwohl … ja doch, ein paar Anmerkungen gibt es auch dazu. Die Betonung liegt auf einen Test. Also als erstes schreiben wir nur einen Test und nicht alle Tests, die uns gerade einfallen. Und dann starten wir den Test und dieser Test muss fehlschlagen! Das ist wichtig! Wenn dieser Test jetzt schon erfolgreich durchlaufen würde, dann hätten wir in Wirklichkeit einen Test für eine bereits bestehende Lösung geschrieben. Und das wäre wieder Test-After.

Jetzt implementieren wir gegen den Test, das heißt, wir schreiben genau den Sourcecode, der nötig ist, damit der Test erfolgreich durchläuft. Und zwar auch wirklich nur so viel, wie nötig und keine Zeile mehr. Alles was wir mehr programmieren würden, wäre nicht durch einen Test abgedeckt. In dieser Phase geht es ausschließlich darum, eine lauffähige Lösung zu finden. Und noch nicht um Schönheit.

Refactoring
Refactoring ist die Verbesserung der Struktur von Sourcecode, ohne dabei sein Verhalten zu verändern. Das Ziel des Refactorings ist die Wartbarkeit und Erweiterbarkeit von Programmcode zu verbessern. Mehr im Detail verbessern wir die Lesbarkeit und Verständlichkeit des Codes, wir reduzieren Fehlerpotential durch die Beseitigung von redundanten Codestellen, erhöhen die Erweiterbarkeit durch eine bessere Aufteilung in Module und mehr Berücksichtigung von loser Kopplung.
Und auch die Verbesserung der Performance kann ein Ziel von Refactoring sein.

Wenn der Test erfolgreich durchgelaufen ist, räumen wir auf, inzwischen auch in Deutschland gerne als Refactoring bezeichnet. Nachdem wir bewiesen haben, dass wir die eigentliche Lösung gefunden haben, dürfen wir das Ganze auch hübsch machen, das heißt, wir entfernen jetzt eventuell redundanten Code, machen Variablen oder Methodennamen sprechender, überprüfen ob wir uns an alle Programmiervorgaben gehalten haben, ob alle Codemetriken erfüllt sind und was sonst noch so ansteht, damit wir das Ergebnis für richtig guten Sourcecode halten.
So mancher Profi behauptet hier gerne, er könne sofort richtig und sauber programmieren. Und auf manche trifft das auch zu. Und der eine oder andere ist dann in Wirklichkeit doch noch nicht ganz so weit. Und das ist keine Schande, auch wir halten uns an die Trennung von zuerst die Lösung finden und erst dann aufräumen. Wir stehen dazu!

Was hier auch noch einen besonderen Hinweis verdient ist, dass das Aufräumen sich nicht nur auf unsere Lösungsimplementierung bezieht, sondern natürlich auch auf unseren Testcode. Sie erinnern Sich an das F in FIRST? F wie Fast. Jeder einzelne Test verdient es, dass Sie in nochmal unter die Lupe nehmen. Kann der nicht noch schlanker und schnell werden? Und auch noch lesbarer und wartbarer? Bedenken Sie, im Laufe der Zeit wird die Anzahl der Tests auf eine drei- bis vierstellige Anzahl anwachsen. Und auch dann müssen die alle zusammen immer noch performant und wartbar sein. Kümmern wir uns also doch einfach direkt darum, bevor es zu spät ist.

5. Schritt In kurze Zyklen zu arbeiten lernen

Wo wir gerade bei Geschwindigkeit sind, sehen wir uns doch mal die zweite grundlegende Idee an, der schnelle Durchlauf des kompletten TDD Zyklus.
Eine sehr spannende Idee ist, dass ein vollständiger TDD Zyklus (Testscheiben, Implementierung, Refactoring) innerhalb von 90 Sekunden durchlaufen werden soll.

90 Sekunden? Warum diese Eile?

Weil es uns zwingt, in kleinen Lösungsschritten zu arbeiten. Wenn ich in 90 Sekunden den ganzen Zyklus durchlaufen möchte, wie groß darf der Test dann maximal sein, dass ich ihn einem Teil der Zeit schreiben kann. Und wie groß darf das Problem maximal sein, dass ich dann in einem Teil der verbleibenden Zeit lösen muss. Diese selbstauferlegte Herausforderung erzwingt, dass wir wirklich in sehr kleinen Einheiten unser Softwareproblem lösen. Wir müssen große Probleme in kleinere und noch kleinere Probleme zerlegen, für die wir dann auch dementsprechend kleine Lösungen finden müssen. Das nimmt die Komplexität aus dem Thema und reduzierte Komplexität führt in den meisten Fällen zu besseren und zuverlässigeren Lösungen. Das erinnert stark an das, was wir oben über Unit-Tests besprochen haben.

„Reduzierte Komplexität führt in den meisten Fällen zu besseren und zuverlässigeren Lösungen“

Jetzt sind 90 Sekunden allerdings tatsächlich ziemlich knapp für die Vorstellung, in dieser Zeit aus dem Nichts einen vollständigen lauffähigen Unit Test und die dazu passende Implementierung zu schreiben und dann noch beides hübsch zu machen. Auch wenn wir uns nur ein sehr, sehr kleines Problem vornehmen wollen.
Deswegen dürfen wir verstehen, dass ein Durchlauf des Zyklus nicht bedeutet, dass danach schon alles fertig läuft. Ein Stück Software zu schreiben, auch wenn es nur ein sehr kleines Stück ist, erfordert viele Durchläufe des TDD Zyklus.

„Der Compiler ist das erste Testtool“

Wie kann ich mir das jetzt vorstellen? Nun, das erste Testtool, das uns zur Verfügung steht ist der Compiler. In den meisten modernen IDEs ist der bereits fest eingebunden und läuft im Hintergrund los, sobald wir Code eingeben. Und gibt uns umgehend Feedback darüber, ob wir kompilerkonformen Sourcecode erzeugt haben oder nicht.

Nehmen wir an, wir wollen in Java mit z.B. Eclipse eine Klasse *Rechner* und in dieser eine Methode *addieren* erstellen.
Als erstes generieren wir z.B. eine JUnit Testklasse *TestRechner* und in dieser den Testfall *TestAddierenEinfach*. Und in diesem Testfall erzeugen wir uns eine Instanz von Rechner mit

Testklasse mit Initialisierung Objekt

```
public class TestRechner {

    @Test
    public void testAddierenEinfach() {
        Rechner rechner = new Rechner();
    }
}
```

Dumm nur, dass es die Klasse Rechner noch nicht gibt und der Compiler und dementsprechend Feedback gibt. Das ist quasi unser erster Testlauf und er schlägt fehl. Also implementieren wir jetzt so lange, bis der Test durchläuft, also der Compiler zufrieden ist. Bedeutet in diesem Fall, wir implementieren eine leere Klasse Rechner. Das können wir bequem von der IDE erledigen lassen und bekommen dann z.B. das hier als Ergebnis:

Leerimplementierung der Objektklasse

```
public class Rechner {

}
```

Zu refakturieren gibt es hier noch nicht viel, also ist der erste Durchlauf durch den Zyklus erledigt. Problemlos innerhalb von 90 Sekunden. Zugegeben, viel bekommen haben wir noch nicht. Aber jetzt geht es ja weiter.

Jetzt wollen im Test die Methode addieren mit den ganzen Zahlen 3 und 5 aufrufen und das Ergebnis in einer Variablen aufnehmen.

Methodenaufruf in der Testklasse

```
public class TestRechner {

    @Test
    public void testAddierenEinfach() {
        Rechner rechner = new Rechner();

        int summe = rechner.addieren(3,5);
    }
}
```

Auch hier bekommen wir unverzüglich Feedback vom Compiler, unserem ersten Testtool, er kennt die Methode *addieren* nicht. Also tun wir ihm den Gefallen und erzeugen sie, und am besten überlassen wir auch das wieder der IDE.

Defaultimplementierung der Methode

```
public class Rechner {
    public int addieren(final int i, final int j) {
        return 0;
    }
}
```

Der Compiler ist glücklich und der Testlauf damit bestanden, allerdings hätten wir diesmal durchaus ein paar Punkte, die wir bezüglich Lesbarkeit und Wartbarkeit verbessern können. Zum Beispiel ist die Benennung der Parameter in der Implementierung nicht so toll und wir verändern sie entsprechend:

Refactoring der Defaultimplementierung

```
public class Rechner {
    public int addieren(final int summand1, final int summand2) {
        return 0;
    }
}
```

Und auch im Testfall wollen wir vielleicht die konkrete Festlegung der Testwerte aus dem eigentlichen Aufruf herausziehen:

Refactoring der Testklasse

```
public class TestRechner {

    @Test
    public void testAddierenEinfach() {
        Rechner rechner = new Rechner();

        int summand1 = 3;
        int summand2 = 5;

        int summe = rechner.addieren(summand1, summand2);
    }
}
```

Schon sieht das Ganze deutlich zukunftsträchtiger aus. Wenn wir uns an die AAA-Regel erinnern, wir haben nun Arrange und Act, fehlt noch Assert:

Assert in der Testklasse

```
public class TestRechner {

    @Test
    public void testAddierenEinfach() {
        Rechner rechner = new Rechner();

        int summand1 = 3;
        int summand2 = 5;

        int summe = rechner.addieren(summand1, summand2);

        Assert.assertEquals(8, summe);
    }
}
```

Da der Compiler in diesem Fall nichts zu bemängeln hat, besteht der Test ab jetzt auch darin, den JUnit Test laufen zu lassen. Und der schlägt nun natürlich fehl, weil unsere aktuelle Implementierung stur 0 zurückgibt. Also implementieren wir nun

soweit, dass unser Test grün läuft. In diesem Beispiel ist das ziemlich einfach:

Lösungsimplementierung in der Objektklasse

```
public class Rechner {
    public int addieren(final int summand1, final int summand2) {
        return summand1+summand2;
    }
}
```

Test neu laufen lassen und nun läuft er erfolgreich durch. Jetzt noch wieder den letzten Schritt des TDD Zyklus, das Aufräumen. Die Implementierung sieht ok für uns aus, allerdings am Testfall ist noch was zu verbessern. Ähnlich wie bei den Summanden kann auch der Vergleichswert der Summe, der im Assert verwendet wird, in eine eigene Variable ausgelagert werden:

Erneutes Refactoring der Testklasse

```
public class TestRechner {

    @Test
    public void testAddierenEinfach() {
        Rechner rechner = new Rechner();

        int summand1 = 3;
        int summand2 = 5;
        int vergleichsErgebnis = 8;

        int summe = rechner.addieren(summand1, summand2);

        Assert.assertEquals(vergleichsErgebnis, summe);
    }
}
```

Wir haben jetzt in kurzen Zyklen eine Methode geschrieben, bei der wir jederzeit wiederholbar beweisen können, dass sie korrekt 3 und 5 addieren kann. Jetzt können wir in weiteren Zyklen erst mal sicherstellen, dass andere gültige Konstellationen funktionieren. Oder prüfen, wie die aktuelle Methode mit extremen Eingabewerten umgeht und das Verhalten der Methode dann entsprechend erweitern. Was passiert zum Beispiel, wenn wir 2147483647 und 1 addieren wollen. Das übersteigt den Wertebereich von int und wird zu unerwünschten Ergebnissen führen. Was uns der Test schnell zeigt und uns die Möglichkeit gibt, dass zu lösen.

Das war jetzt kein besonders schwieriges Beispiel, es zeigt jedoch sehr gut, wie in sehr kurzen Zyklen testgetrieben eine Lösung entwickelt werden kann. Und jeder dieser Schritte ist mit etwas Übung gut innerhalb von 90 Sekunden realisierbar.

„Lauffähiger Test innerhalb von maximal 10 Minuten“

Jetzt könnten wir allerdings mit diesem Verfahren fast unbegrenzt lange fortfahren und erst nach Stunden den ersten wirklich lauffähigen Test vorweisen. Was aber auch nicht der Sinn der Sache ist, denn erst wenn wir erste wirklich lauffähige Tests haben, sehen wir, dass wir auch etwas mit Wert schaffen. Also ergänzen wir unsere 90 Sekundenregel noch um eine 10 Minutenregel. Ziel ist, innerhalb von maximal 10 Minuten einen ersten/weiteren lauffähigen Test zu haben. Oder anders formuliert, wir wollen innerhalb von 10 Minuten einen durch Tests abgesicherten Mehrwert schaffen.
Und damit sind wir wieder beim Fokus auf kleine Einheiten mit simplen Lösungen.

Wir sind jetzt soweit, dass wir durch Unit Test getrieben unsere Software schreiben. Oder zumindest die vielen kleinen technischen Komponenten. Das ist schon super. Aber ein Softwareprodukt besteht aus mehr nur kleinen technischen Komponenten, es besteht auch aus mehr oder weniger großen Zusammenhängen. Und genauso besteht eine gute Testabdeckung aus mehr als nur Unit Tests.

6. Schritt Die Testpyramide aufbauen

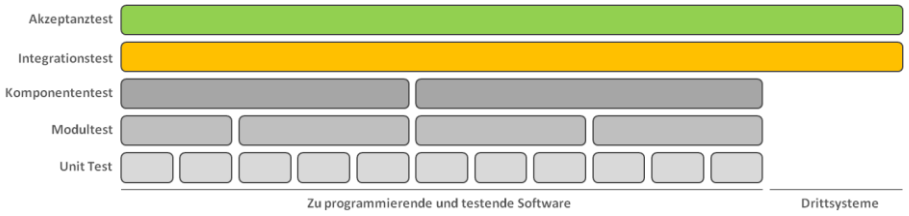
Die babylonische Sprachverwirrung
Modul-, Komponenten-, Integrations-, System-, Akzeptanz-, End-to-End- oder Abnahmetest, Namen für unterschiedlichen Testlevel gibt es viele. Allerdings sind auch hier die Bedeutungen nicht wirklich standardisiert. Sie sind nur Vereinbarungen zwischen den betroffenen Personen. Stellen Sie also sicher, dass Sie wirklich alle da gleiche unter diesen Begriffen verstehen und gegen Sie nicht einfach blind davon aus. Den Begriffen oben ist übrigens gemeinsam, dass sie alle die Reichweite, den Scope, der Teststufe beschreiben und nicht das Verfahren. Im Gegensatz zu Begriffen wie Performance-, Last-, Langzeit-, Monkey- oder Smoketests. Und auch hier gilt, klären Sie das gemeinsame Verständnis für die Begriffe.

Die drängende Frage lautet nun also: Kann ich mit Test Driven Development jetzt also nur Unit Tests schreiben? Das wäre ja nicht so günstig, weil wir wollen ja nicht nur die einzelnen, voneinander unabhängigen Aspekte unserer Komponenten testen, wir wollen auch sicherstellen, dass alles korrekt zusammen funktioniert.

Und das kann ich natürlich auch testgetrieben machen. Nachdem ich z.B. durch Unit Tests getrieben zwei kleine Aspekte implementiert habe, schreibe ich einen Test, der das Zusammenspiel der beiden Aspekte testet. Und dann implementiere ich dieses Zusammenspiel. Zu beachten ist dabei, dass ich in diesem ‚Integrationstest' jetzt nicht mehr die grundsätzliche Funktion der jeweiligen einzelnen Units testen muss, sondern wirklich nur noch das Zusammenspiel.

Auf diese Art und Weise können wir uns nun die Testpyramide hocharbeiten. Wie bei einer echten Pyramide legen wir immer erst unten Steine hin, bevor wir die Steine der nächsten Ebene setzen. Bedeutet im Bereich Softwaretesting: Erst ein paar Unit Tests, dann die Modultests, die die getesteten Units verbinden, dann die Komponententests, die z.B. die bereits getesteten Module verbinden. Dann evtl. schon der Integrationstest, bei dem unser bereits erstelltes und bis dorthin getestetes Stück Software jetzt mit Drittsystemkomponenten zusammen getestet wird und weiter bis hin zum End-to-End-Test oder Akzeptanztest oder wie auch immer das bei Ihnen heißt. Dabei haben wir jetzt mit Ausnahme des Unit Tests die Begriffe der Teststufen nach unserem Ermessen gewählt.

Wie die jeweiligen Teststufen in Ihrem Unternehmen benannt werden, ist auch weiterhin Ihre Entscheidung.



Testebenen und deren Scope

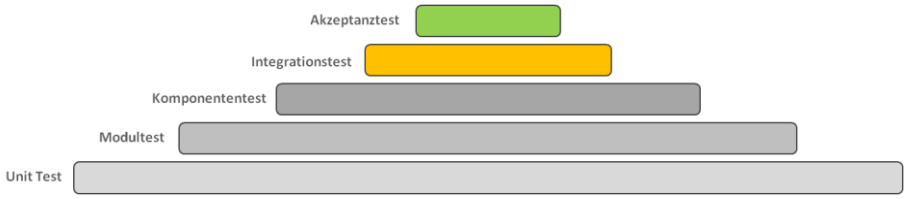
Nur zur Wiederholung nochmal: Auch bei den Teststufen oberhalb der Unit Tests schreiben wir immer zuerst den Test und erst dann Implementieren wir den Sourcecode, der bereits getestete Einheiten zu einem größeren Ganzen zusammenfügt.

Ein einfaches Beispiel: Wir haben testgetrieben bereits eine *sum* Methode geschrieben, die zwei ganze Zahlen entgegennimmt und diese aufaddiert zurückgibt. Und wir haben durch Unit Tests ausgiebig sichergestellt, dass das auch funktioniert. Und wir haben bereits eine *checksum* Methode geschrieben, der wir eine ganze Zahl übergeben können und von dieser die Quersumme zurückbekommen. Und auch das ist hinreichend durch Unit Tests sichergestellt. Nun wollen wir einen Handler schreiben, der zwei ganze Zahlen übergeben bekommt, sich diese durch die bereits vorhandenen Methode *sum* aufaddieren und das Ergebnis davon von *checksum* bearbeiten lässt.
Alles was der Handler braucht, um sicherzustellen, dass er auch wirklich zwei ganze Zahlen bekommen hat, stellen wir durch Unit Tests sicher, denn das ist vermutlich eine neu zu erstellende Funktionalität. Wenn wir jetzt jedoch die Tests für den Ablauf „zwei ganze Zahlen durch *sum* aufaddieren und dann durch *checksum* bearbeiten lassen" schreiben, dann müssen wir nun nicht mehr in diesem Test sicherstellen, dass *sum* oder *checksum* wirklich mit allen möglichen Eingaben klarkommen und die Ergebnisse wirklich stimmen. Das haben wir bereits in den entsprechenden Unit Tests sichergestellt.

Der Prozess des Test Driven Development führt uns auf diese Art von unten nach oben durch die verschiedenen Testebenen und wir wissen das wir mit dem Entwickeln fertig sind, wenn wir die Tests auf der obersten Ebene geschrieben haben und diese erfolgreich durchlaufen.

Warum nennen wir das Testpyramide, das sieht doch in der Abbildung oben gar nicht aus wie eine Pyramide? Dem können wir abhelfen.

Testfallverteilung als Testpyramide



In der ersten Abbildung haben wir den Scope der einzelnen Tests auf den verschiedenen Testebenen abgebildet. In der zweiten Graphik zeigen wir die typische Verteilung der Menge an Tests, die bei einer sauber umgesetzten Teststruktur entsteht.

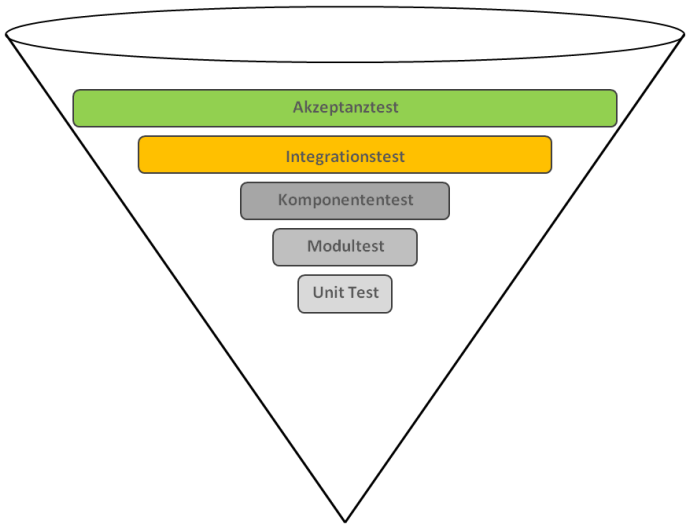
Was auffällt ist, dass die Menge an Tests/Testfällen von unten nach oben hin abnimmt. Das ist das Ergebnis daraus, dass wir schon auf der Ebene der Unit Tests einen Großteil der Funktionalität mit Tests abdecken und dann von Ebene zu Ebene quasi nur noch den Mehrwert testen, der aus der Kombination kleinerer Einheiten entsteht.
Das hat direkt mehrere Vorteile:

a. Da jeder Test nur noch einen begrenzten Scope hat, den es an Funktionalität wirklich sicherzustellen hat, sind alles Tests relativ klein, leicht verständlich und gut wartbar

b. Je weiter oben in der Testhierarchie ein Test ist, umso schwieriger und aufwändiger ist es, Funktionalität weiter unten im System zu testen. Das ist das typische Problem von Black Box Tests. Je weiter unten in der Testhierarchie wir einen Test schreiben, umso einfacher ist es eine bestimmte technische Funktionalität testen zu können. Wir bewegen uns im Bereich der White Box Tests. Durch den Aufbau einer sauberen Testpyramide reduzieren oder vermeiden wir es sogar komplett die Notwendigkeit, von höheren Testebenen aus technische Detailfunktionen testen zu müssen, Tests auf höheren Ebenen sind damit einfacher (siehe Punkt a.)

c. Wenn zu einem späteren Zeitpunkt sich Verhalten im System unerwünscht ändert, also ein Fehler auftritt, dann zeigen uns die fehlschlagenden Tests in der Testpyramide konkret, an welcher Stelle das Verhalten des Systems nicht mehr stimmt. Bei der Fehleranalyse und Fehlerbehebung fangen wir dann immer mit den fehlschlagenden Tests auf der niedrigsten Ebene an. Die Fehleranalyse und Fehlerbehebung werden dadurch deutlich beschleunigt.

Diese Verteilung der Testfälle wird einige etwas überraschen, weil es sich nicht mit ihrer Erfahrung deckt. In vielen Unternehmen besteht die große Menge an Tests aus GUI-, Integrations- und Akzeptanztests. Und je weiter nach unten man in der Testhierarchie schaut, umso weniger Testfälle werden es. Wir reden hier auch von der sogenannten Eistüte.



Testfallverteilung als Eistüte

Was in diesem Modell viele Projekte an die Grenzen oder sogar darüber hinausbringt ist, dass sehr viel auch der fundamentalsten technischen Basisfunktionalitäten über die oberste der Testebenen abgesichert werden muss. Wie wir gerade besprochen haben, ist genau das der ungünstigste weil schwierigste und damit zeitaufwändigste Ort dafür. Die Konsequenz ist oft eine unzureichende Testabdeckung, weil zu langwierig.

Nutzen Sie also lieber Test Driven Development und formen Sie Ihre Eistüte in eine Pyramide um.

Fazit

Richtig verstanden und angewendet ist Test Driven Development ein sehr mächtiges Vorgehensmodell um funktionierende Software auf einem sehr hohen Qualitätsniveau zu erstellen. Allerdings stellt Test Driven Development keine Abkürzung dar, mit der man mal so auf die Schnelle ein Stück Software aus dem Boden stanzen kann und dann läuft es schon. TDD setzt voraus, das von Anfang an kompromisslos auf Qualität gesetzt und damit dafür notwendige Zeit investiert wird. Nachhinein kann auch TDD die Qualität einer Software nicht mehr retten.

„Qualität ist nicht verhandelbar“

Den einen oder anderen Entscheider mag diese konsequente Investition in Qualität erst mal erschrecken, weil damit Quick-n-Dirty Schnellschüsse nicht mehr möglich sind. Aber sobald Sie TDD erst mal wirklich in Ihrem Unternehmen eingesetzt haben, werden Sie schnell merken, dass sich diese Investition doppelt und dreifach rechnet.

Was wir bekommen, ist eine Software, bei der von Anfang an die Funktionsfähigkeit bewiesen ist und die aufgrund des ständigen Refactorings während des Entwicklungsprozess und der hohen Abdeckung durch automatisierte Tests jederzeit problemlos gewartet und weiterentwickelt werden kann.

Damit Sie auch wirklich das Beste aus Test Driven Development herausholen können, ist neben der guten Absicht auch eine gute Ausbildung Ihrer Softwareentwickler nicht nur im Umgang mit Test Driven Development, sondern auch in den Prinzipien guten Softwaredesigns sinnvoll.

„Investieren Sie in die Ausbildung Ihrer Softwareentwickler“

Investieren Sie hier um auch in Zukunft hervorragende Produkte zu bauen.

Lernen Sie mehr unter

testdrivendevelopment.de

oder

binaris-education.com

Autoren: Carsten Czeczine
Thorsten Werle

Binaris informatik GmbH
Elisabethstrasse 5
40764 Langenfeld

Mehr Informationen finden Sie unter:

www.binaris-informatik.de
www.binaris-education.de
www.testdrivendevelopment.de

Wenn Sie Fragen oder Anregungen haben,
bitte kontaktieren Sie uns:

carsten.czeczine@binaris.de
thorsten.werle@binaris.de

☎ +49 2103 253 64 47
+49 151 240 39 115