Management im Software Engineering
Universität Würzburg
WS 2013/2014

Ausarbeitung

Test-Driven Development

Philipp Jeske Benjamin Morgan

4. Januar 2014

Inhaltsverzeichnis

1 Einführung **4**

2 Definition **5**

3 Umsetzung **7**
3.1 Unit-Tests 7
3.2 Integrationtests 7
3.3 Systemtest 8

4 Bewertung **9**
4.1 Vorraussetzungen 9
4.2 Vorteile 10
4.3 Nachteile 10

5 Fazit **13**

6 Erfahrungsbericht **15**
6.1 Benjamin Morgan 15
6.2 Philipp Jeske 16

Glossary **17**

Diese Ausarbeitung wurde im Rahmen der Vorlesung „Management im Software Engineering“ an der Universität Würzburg im Wintersemester 2013/2014 bei Dr. Jürgen Schmied geschrieben. Im Folgenden wird ein kurzer Überblick über das Entwicklungsparadigma „Testgetriebene Entwicklung“ gegeben und anhand eines Beispieles die wichtigsten Faktor aufgezeigt. Anschließend werden Vor- und Nachteile aufgezeigt und ein Fazit gezogen. Abschließend findet sich noch ein persönlicher Erfahrungsbericht der beiden Autoren dieser Arbeit.

1 Einführung

Philipp Jeske

Bei den klassischen Entwicklungsmodellen werden Tests erst nach Fertigstellung einzelner Module ausgeführt, dies führt dazu, dass unter Umstände viele Nacharbeiten nötig sind bevor mit dem nächsten Modul angefangen werden kann. Auch ist es bei diesem Vorgehen möglich, dass Funktionen vergessen werden zu implementieren. Dieses Problem ist gerade heutzutage immer schwerwiegender, da die Software immer komplexer wird und somit die Pflichtenhefte immer länger werden. Im Zusammenspiel mit Zeitdruck im Projekt führt dies dazu, dass manches überlesen wird. Eine weitere Problematik die bei klassischen Methoden auftreten kann, ist dass Tests vernachlässigt werden, sollte es am Projektende eng werden oder der Kostenrahmen gesprengt werden.

Diese Probleme versucht die agile Softwareentwicklungs-Methode Extreme Programming (XP) mit häufigen Tests und *Pair Programming* zu umgehen. Mittlerweile hat sich der Aspekt des häufigen Testens zu einem eigenen Paradigma entwickelt und findet in vielen Bereichen Anwendung in denen hohe Codequalität gefordert wird. Bei Testgetriebene Entwicklung (TDD) steht das regelmäßige Testen des Codes bereits während der Implementierung im Mittelpunkt. Da dazu die Test bereits vor dem Production Code geschrieben werden, wird auch teilweise bereits auf Vollständigkeit geprüft. Im folgenden Kapitel wird auf die genaue Definition von Testgetriebener Entwicklung genauer eingegangen.

2 Definition

Benjamin Morgan

Testgetriebene Entwickelung (engl. *test-driven development*, auch TDD) ist ein Teil der agile Softwareentwickelung, bestehend aus *test-first development* (TFD) und Refaktorisierung. Die Tests werden zuerst geschrieben und steuern die gesamte Softwareentwickelung. [**AgileData**, **itAgile**]

Diese Kombination stellt den traditionellen Softwareentwicklungsprozess oft auf den Kopf, wo Code der funktioniert nicht mehr angefasst wird (da etwas kaputt gehen könnte) und Testing in der Regel zuletzt gemacht wird oder teilweise weggelassen wird.

TDD besteht aus einem kurzen iterativen Zyklus:

- Schreibe einen Test, der erstmal fehlschlägt.
- Implementiere gerade so viel Produktivcode, dass der Test erfolgreich durchläuft wird.
- Refaktorisiere den Test- und Produktivcode. [**itAgile**]

Die ersten zwei Schritte machen *test-first development* (TFD) aus, und der dritte Schritt ist das Refaktorieren. Man kann das TFD-Paradigma auch zusammenfassen als eine Reihe von Gesetzen die der Entwickeler befolgen muss, Diese Sicht hat den Vorteil, sehr präzise und prägnant zu sein. Die folgenden Punkte bilden eine Sammlung solcher Gesetze.

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test. [**UncleBob**]

Die Voraussetzung ist natürlich, dass man Produktivcode schreibt, der von Unit-Tests getestet werden kann. In einem gewissen Sinne reagiert der Entwickler nur auf fehlschlagende Tests. Allerdings fehlt hier das kritische Refaktorisieren.

Aus diesen drei Gesetzen folgt, dass man nicht sehr viel Zeit nur mit den Tests oder nur mit dem Produktivcode verbringen kann. Man wechselt häufig zwischen Tests schreiben und Produktivcode schreiben ab, sogar alle paar Minuten [**itAgile**, **UncleBob**]. Das System kompiliert dabei ständig, um den Entwickler dabei zu unterstützen. Das Ergebnis ist, dass man fast immer Produktivcode hat der funktioniert,

und wenn er nicht funktioniert, dann wird der Code der vor ein paar Minuten existiert hat funktionieren.

Zu diesem Zweck gebraucht TDD Konfigurationsmanagement in großem Maß, sodass man beim Refaktorisieren leicht zu ältere Versionen vom Code zurückfallen kann, falls das Refaktorisieren komplett fehlschlägt.

Die Entwicklungsumgebung muss diese Arbeitsweise also unterstützen, um TFD anwenden zu können; lange Kompilierzeiten, schlechte Testframeworks oder ein Konfigurationsmanagement das nicht schnell und unkompliziert in den Workflow integriert werden kann, können TDD sehr unprofitabel machen.

Ein signifikanter Nebeneffekt von TDD oder TFD im Allgemeinen ist, dass man sehr viel Testcode generiert. In der Regel rechnet man mit mindestens so viel Testcode wie man Produktivcode hat. [**SQlite3**, **C2**]

3 Umsetzung

Philipp Jeske

Die Umsetzung der TDD erfolgt in mehren Stufen und findet auf unterschiedlichen Integrationsebenen statt. Je nach Ebene werden unterschiedliche Praktiken und Tools angewendet, auf die im Folgenden näher eingegangen wird.

3.1 Unit-Tests

Die bekannteste Testtechnik in der Softwareentwicklung sind wahrscheinlich die Unit- bzw. Modultests. Diese werden häufig mit Hilfe von in die Entwicklungssprache und die Integrated Development Environment (IDE) integrierte Testframeworks durchgeführt. Einer der bekanntesten und ersten Repräsentanten dieser Testframeworks ist jUnit. Mittlerweile gibt es viele Ports auf andere Sprachen und Platformen, zum Beispiel nUnit für das .NET-Framwork von Microsoft oder cUnit für C und C++, das nicht nur auf x86- und x64-Platformen portiert wurde, sondern unter anderem auch auf MIPS und MSP430.

Diese Test finden auf der untersten Ebene statt und sichern die Software gegen Implementierungsfehler von Teilaufgaben ab, so werde einzelne Funktionen auf die korrekte Ausgabe bei einer genau definierten Eingabe getestet. Auf dieser Integrationsebene werden alle Objekte, die mit dem zu testenden Objekt interagieren durch so genannte Mock-Objekte abstrahiert.

Ein Mock-Objekt bietet die gleiche Schnittstellen wie das zu simulierende Objekt, allerdings sind die Ausgaben fest definiert, um ein deterministischen Verhalten ohne Interferenzen mit unter Umständen externen Quellen zu vermeiden. Zum Beispiel wird bei datenverarbeitenden Programmen der Datenbankzugriff auf ein Mock-Objekt zum Testen abgebildet, da dadurch sich die Eingabe genau definieren lässt.

Die Unit-Tests werden bei Anwendung des Paradigmas der TDD bei jeder kleinsten Änderung ausgeführt und die Entwicklung wird erst durchgeführt, sobald der entsprechende Test fehlerfrei durchläuft.

3.2 Integrationtests

Bei den Integrationstests, die häufig auch noch mit den Testframeworks durchgeführt werden, werden in einzelnen Iterationen die Mock-Objekte durch ihr reales Pendant ersetzt und dadurch kontrolliert, dass die Zusammenarbeit der Komponenten untereinander fehlerfrei funktioniert.

Die Integrationstests werden bei der TDD ab der ersten Iteration ständig durchgeführt ab der eine Integration möglich ist. Beim Beispiel der datenverarbeitenden

Anwendung, wird neben den Modultests, die z.B. die Berechnung eines Indexes usw. beinhalten, ab der vollständigen Implementierung des Datenbankzugriffes neben dem Mock-Objekt-Test auch direkt der Datenbankzugriff getestet, um frühzeitig Probleme bzw. Fehler zu erkennen und diese in einem frühen Stadium beheben zu können.

3.3 Systemtest

Sind alle Teile einer Anwendung erfolreich mit Modultests und Integrationstests getestet, kann die letzte Iteration der Integrationstests durchgeführt werden. Diese wird auch häufig als Systemtest bezeichnet und ist der erste Test, bei dem alle Komponenten zusammenspielen und ihr Verhalten miteinander getestet wird.

Ab der ersten vollständigen Integration aller Komponenten wird bei der jeder Iteration des Refactor/Debug-Zyklus neben Unit-Tests und den Integrationstests ausgeführt auch bei diesem Test gilt, die Entwicklung wird erst fortgesetzt, wenn alle Tests erneut erfolreich bestanden sind.

4 Bewertung

Benjamin Morgan

Es ist nicht immer offensichtlich, was die Vor- und Nachteile von TDD sind. Manchmal kommt es drauf an, wie genau man TDD anwendet, ob eine Eigenschaft von TDD ein Nachteil oder ein Vorteil ist. Es gibt auch Vorraussetzungen, die erfüllt sein sollten, um TDD sinnvoll anzuwenden.

4.1 Vorraussetzungen

Bill Landers auf *StackExchange* [**StackExchange**] hat prägnant zusammengefasst, welche Voraussetzungen erfüllt werden müssen, dass man von TDD richtig profitieren kann:

- Der Produktivcode ist testbar auf Basis von Unit-Tests und es macht auch Sinn auf dieser Ebene zu testen.
- Der Entwicklungsgegenstand hat einen klaren Ansatz (d.h. einen klaren Weg dahin) und erfordert kein aufwendiges Experimentieren oder Prototyping.
- Es wird nicht zu viel refaktoriert/überarbeitet und die Spezifikation wird nicht signifikant geändert; es sei denn es ist akzeptabel hunderte bis tausende von Testfälle immer wieder neu zu schreiben.
- Nichts ist geschlossen (engl. *sealed*).
- Man kann den Entwicklungsgegenstand logisch in Module zerteilen.
- Es ist möglich, überall Dependency Injection anzuwenden oder die Objekte durch Mockobjekte zu ersetzen.
- Das Management und das Entwicklungsteam akzeptieren TDD und wenden es konsistent an.
- Der Auftraggeber legt genug Wert auf insignifikante[1] Fehlern um den Zusatzaufwand zu rechtfertigen.

Auch wenn nicht alle Punkte zutreffen, ist es dennoch möglich eingeschränkt TDD anzuwenden, allerdings sollte man besonders Acht geben, dass man keine Teile des Codebases benachteiligt, weil sie nicht mittels dem TDD Paradigma entwickelt werden.

[1] Eigentlich sind fast keine Fehler insignifikant; hier geht es um Fehler die nicht so gravierend sind, oder die nur sehr selten auftreten.

4.2 Vorteile

Für Softwareprojekte die mit TDD entwickelt werden, gibt es verschiedene Vorteile.

- Hunderte und Tausende von Tests werden erzeugt, die jederzeit durchgeführt werden können; und sie sollten immer erfolgreich verlaufen.
- Durch die umfassenden Tests, gibt es mehr Transparenz im Produktivcode; die Tests spezifizieren die Funktionalitäten.
- Die Menge an Tests dient als Dokumentation und Beispielanwendungen zugleich, zusätzlich sind sie immer aktuell.
- Die Tests bilden eindeutig die Spezifikation und zwingen den Entwickler die Spezifikation gut zu durchdenken bevor er mit dem programmieren anfängt.
- Man kann nach Belieben refaktorisieren, ohne zu riskieren, dass man aus Versehen eine Regression einführt, da die Tests eine Art Sicherheitsnetz bilden.
- Es gibt kein Grund, Angst zu haben, Code „anzufassen“, also zu ändern oder löschen. Die Software wird wirklich *soft*. [**UncleBob**]
- Man hat ein testbares Design und Codebase, weil TDD es erzwingt.
- Es gibt keine Ausreden oder Abkürzungen mehr um das Testen; weder kann der Entwickler faul sein noch das Management das Überspringen der Tests erzwingen.

In der Tat ist TDD ein effektives Werkzeug gegen die menschliche Natur; wenn ein Softwareprodukt fertig ist, neigt man oft dazu, nicht mehr zu testen wenn alles im grünen Bereich zu sein *scheint*. TDD beugt diesem vor.

4.3 Nachteile

Die Vorraussetzungen von TDD können auch als Nachteile gesehen werden. Obwohl TDD Vorteile bringt, ist der Zusatzaufwand nicht Umsonst. Nicht nur müssen Entwickler TDD gut beherrschen lernen, sondern die konsiste Anwendung erfordert auch mehr Disziplin und Zeit wie ohne TDD.

- Gewisse Softwaregegenstände lassen sich weniger effektiv mit TDD entwickeln als andere. So sind beispielsweise Projekte die sehr User-Interface oder Netzwerk lastig sind, schwer zu testen.
- Auch trivialle Änderungen an der Spezifikation können dazu führen, dass man hunderte oder tausende von Tests neuschreiben muss. Sogar ein signifikantes Refaktorisieren kann dazu führen, dass viele Tests fehlschlagen, obwohl die Spezifikation eigentlich eingehalten wird. [**StackExchange**]

- Das Management vom gesamten TDD Paket zu überzeugen kann sehr schwer sein. Zum Beispiel eignet sich TDD am besten wenn man Paarprogrammiert, welches aber viele Manager nicht akzeptieren (zwei tun die Arbeit von einem, das ist uneffizient). [**StackOverflow**]
- TDD erfordert eine große Investition von Zeit und Aufwand um überhaupt reinzukommen; viele Entwickler haben nicht die nötige Geduld, und somit kann es schwierig sein, das gesamte Entwicklungsteam von TDD zu überzeugen. [**StackOverflow**]
- Die gesamte Menge an Tests muss auch erhalten und debugged werden. Obwohl die Tests die Qualität des Produktivcodes steigern, ist der Entwickler selber für die Qualität der Tests komplett zuständig.
- Das schreiben von guten, effektiven Tests wie TDD es fordert ist eine Kunst in sich selbst; viele Entwickler können keine *gute* Tests schreiben. [**StackOverflow**]
- TDD macht es sehr teuer, den Code zu tunen; bei vielen Algorithmen kann erst durch Experimentieren die Performance richtig gesteigert werden. [**StackOverflow**]
- Bestehende Projekte auf die TDD Entwickelungsstrategie umzustellen kann eine riesige Herausforderung sein; es gibt keine Garantie, dass alle Codewege gedeckt sind oder dass die Tests wirklich die gesamte Funktionalität abdecken.

Es sieht vielleicht nach viel aus was gegen TDD spricht, aber die meisten Nachteile sind Design bedingt und gelten nicht für alle Projekte. Für die Projekte die zu TDD passen, oder für Entwickler die darin sehr zuhause sind, kann TDD besser sein als alternative Entwicklungsstrategien. Nach Sun Tzu sollte man allerdings seine eigene Schwächen gut kennen, damit man effektiv dagegen wirken kann.

- Dass erfolgreiche durchlaufen von hunderte und tausende von Tests kann leicht ein falsches Gefühl von Sicherheit herbeiführen. [**StackExchange**]
- Die Voraussetzung, dass Code (Unit-) testbar sein muss führt oft zu einem viel komplexeren Design als für das Problem eigentlich nötig ist. [**StackExchange**]
- TDD führt zu einem sehr Schnittstellenzentrierten Code, da dies sich besser testen lässt. Dies mag oft eine gute Eigenschaft sein, für kleine Projekte ist es aber vielleicht unnötig.
- Es ist möglich zu detailliert oder zu genau zu testen, dies kann die Entwickelung verlangsamen und sogar das Refaktorisieren verhindern.
- TDD kann den Produktivcode schnell in ein vorläufiges Design fixieren, da die Tests eine bestimmte API verlangen. Falls die API noch nicht eindeutig ist, oder falls Prototyping nötig ist, wird TFD viel Zusatzarbeit erzeugen. [**StackOverflow**]

- Wenn Entwickler selber die Tests für ihren eigenen Produktivcode schreiben, dann können leicht die Tests die gleichen Lücken wie der Produktivcode aufweisen.
- Oft glaubt man, die Tests würden alle Codewege decken; das mag anfänglich stimmen, kann durchaus aber gerade durch das Refaktorisieren nicht mehr der Fall sein.

5 Fazit

Philipp Jeske

Nachdem im vorigen Kapitel die Vorteile und Nachteile von TDD aufgezeigt wurden, soll in diesem Kapitel versucht werden objektiv einzuordnen, wann und wo TDD sinnvoll eingesetzt werden kann, bevor im folgenden Kapitel eine subjektive empirische Bewertung von TDD der beiden Autoren erfolgt.

Betrachtet man zunächst die nackten Zahlen, so gibt es mehr Nachteile als Vorteile und zusätzlich noch diverse Einschränkungen, welche in Abschnitt 4.1 geschildert sind. Untersucht man jedoch die Vorteile und Nachteile genauer kann unter gewissen Umständen ein Vorteil zu einem Nachteil werden und umgekehrt bzw. lassen sich die Nachteile vernachlässigen, weil die Vorteile einen hinreichend großen Vorteil bringen. Dies bedeutet, dass man allgemein weder zu TDD raten kann noch davon abraten. Dem Einsatz von TDD sollte also immer eine Einzelfallprüfung vorausgehen, ob die Vorteile die Nachteile aufwiegen.

Im Folgenden soll dies an einigen einfachen Beispielen illustriert werden.

Beispiel 5.1. *Es soll eine neue Version eines Autopiloten entwickelt werden. Der Autopilot legt anhand von Zielwerten und Messwerten, wie Windgeschwindigkeit, -richtung, Luftdruck, GPS etc., die Winkel der Steuerflächen und die Leistung der Turbinen fest. Das Management fordert eine hohe Qualität und überprüfbare Korrektheit, damit eine Zulassung für den Luftverkehr erfolgen kann.*

In diesem Szenario sind alle Vorraussetzungen mehr oder weniger erfüllt. Das Management wird sich wahrscheinlich TDD nicht verweigern, da eine nachweisbare Korrektheit gefordert ist. Diese wird durch TDD zu einem hohen Prozentsatz erreicht. Der Weg ist zum Erreichen des Ziels ist klar, da – vereinfacht ausgedrückt – lediglich einige Gleichungen korrekt implementiert werden müssen, die die Eingaben verarbeiten und die Ausgaben erzeugen. Die Umsetzung lässt sich stark modularisieren, so lassen sich beispielsweise pro Sensor ein Modul und pro Ausgabe ein Modul definieren.

Die Vorteile die sich durch den Einsatz von TDD in diesem Fall ergeben sind

- Fortwährende Überprüfung der Korrektheit
- nachweisbare Korrektheit – nur bei hinreichender Test-Coverage

Diese beiden Vorteile überwiegen in diesem Fall den Nachteil des erhöhten Aufwands, da Korrektheit in diesem Beispiel essentiell ist. Somit ist in diesem Fall der Einsatz von TDD zu empfehlen.

Beispiel 5.2. *Es soll eine Applikation entwickelt werden, die über das Netzwerk auf eine Datenbank zugreift und für eine GUI für die CRUD-Operationen bereitstellt. Der*

Kunde möchte damit ein existierendes System ablösen und seinen Workflow verbessern. Es soll auf einem für das Unternehmen neuen Framework aufbauen und Teilberechnungen sollen über eine proprietäre Drittsoftware umgesetzt werden. Um die Akzeptanz der Anwender zu steigern, werden diese regelmäßige nach Verbesserungsvorschlägen befragt.

Dieses Beispiel ist so ziemlich der Super-GAU für den Einsatz von TDD. Es gibt kein fertige Spezifikation, da diese im Laufe der Entwicklung nach Anwenderwunsch regelmäßig angepasst werden soll, eine proprietäre Blackbox soll neben einem Framework eingesetzt werden, für das noch keine Erfahrungswerte im Unternehmen existieren. Ein ebenso kritischer Punkt ist die Schwierigkeit beim Mocken der Kommunkation des Datenbankzugriffes, dies ist zwar möglich, erfordert jedoch immensen Mehraufwand. Ferner ist es generell schwierig eine GUI analog zur Business Logic automatisiert zu testen. Zusammengefasst bedeutet das, dass in diesem Fall von einem Einsatz von TDD abzuraten ist.

So eindeutig wie in diesen beiden Extrembeispielen wird in der Praxis wahrscheinlich selten die Entscheidung fallen. So kann ein gangbarer Weg manchmal sein, TDD nur für einige Teilmodule einzusetzen. Allerdings bleibt das Problem der Testerstellung aus der Spezifikation bestehen. So kann aus einer ungenügenden Spezifikation niemals ein guter Testpool entstehen. Als Alternative zum klassischen Ansatz, exisitert die so genannte Beispielgesteuerte Entwicklung (EDD), die sich lediglich in der Erstellung der Tests vom gängigen TDD unterscheidet. So werden bei der EDD die Testfälle nicht anhand der formalen Spezifikation, sondern anhand gemeinsam mit dem Kunden erarbeiteten Beispielen erstellt.

6 Erfahrungsbericht

Nach dem objektiven Fazit im vorigen Kapitel, werden hier noch die Erfahrungen der beiden Autoren zusammengefasst und ein persönliches Fazit gezogen.

6.1 Benjamin Morgan

Benjamin Morgan

Ich habe an mehreren Softwareprojekte gearbeitet oder mitgearbeitet, jedoch noch nicht mit dem strikten testgetriebenen Entwicklungsstil. Wenn ich meine Sicht trotzdem äußern darf, dann scheint mir die pure TDD etwas zu *masochistisch* für viele Projekte (vorallem die, an denen ich arbeite). Ich glaube eine Mischform die eine umfangreiche aber angemessene Menge von Testing miteinbezieht könnte sich als sehr hilfreich erweisen. Dieser Stil müsste nicht so strikt die Gesetze der TDD einhalten, würde aber troztdem einen ähnlichen testgetriebenem Entwicklungsstil an gewissen Stellen einsetzten.

1. Denke das Design gut durch.
2. Entwickle eine geeignete API für den Lösungsansatz. Das könnte am besten dadurch geschehen indem man
   - auf Papier einen Entwurf macht,
   - verschiedene Anwendungsfälle durchdenkt und durcharbeitet oder
   - Anwendungsbeispiele für die zukünftige API programmiert.

   Es könnte sogar durchaus sinnvoll sein, Prototypen oder gar Produktivcode zu schreiben. Oft merkt man, dass der erste Ansatz doch nicht das ist, was man will; wenn man das spät merkt, ist TDD nachteilig, da man bereits viel Zeit mit konkreten Tests verbummelt hat.
3. Schreibe umfangreiche Tests für die API (nicht aber unbedingt für die kleine privaten Funktionen). Alle möglichen Ein- und Ausgänge sollen berücksichtigt sein.
4. Schreibe den Produktivcode und nutze die zuvor geschriebene Tests, aber vertraue nicht darauf (schalte das Gehirn nicht aus).

Diese Schritte können mit verschiedener Genauigkeit und Umfang angewendet werden. Es sollte möglich sein, mehrere Tests oder mehr Produktivcode am Stück zu schreiben, dadurch erspart man dem Programmierer gezwungene Kontextwechsel, die seine Produktivität negativ beeinflussen.

Das TDD Paradigma fühlt sich so an als würde sich alles (und zwar ohne Ausnahme) um die Tests drehen. Das finde ich ein bisschen extrem; man verliert sich leicht darin und vergisst das der Kunde oder der Nutzer in der Regel den Produktivcode bekommt, nicht die ganze Testsuite. Ich persönlich hätte viel lieber sehr guten Produktivcode als sehr gute Tests (mit schlechtem Produktivcode). Vielleicht kann man von den Programmiersprachen *Go*[1] und *D*[2] lernen, denn sie integrieren das Testing in die Sprache bzw. Tools und Bibliothek auf einer Weise, dass das Testing sehr angenehm macht.

6.2 Philipp Jeske

Philipp Jeske

Ich habe bereits in kleineren Projekten als auch in mittelgroßen Projekten, versucht TDD einzusetzen, jedoch gab es einige Probleme.

So stelle sich heraus, dass der Aufwand bereits bei kleinen Projekten ziemlich groß ist, wenn man den ganzen Netzwerkstack mocken muss bzw. den Zugriff auf eine Datenbank. Ebenso stellte sich heraus, dass es eine Menge Mehraufwand erzeugt, wenn sich Anforderungen ändern, da diese dann auch in den Tests nachgezogen werden müssen.

Aber es gab nicht nur negative Beispiele, so eignet sich TDD meines Erachtens um logische Fehler bereits während der Implementierung zu erkennen. Auch projektinternen Abhängigkeiten der Art „Klasse A erbt von Klasse C und implementiert Schnittstelle C“ können mit Hilfe von TDD sehr gut getestet werden.

Aufgrund meiner persönlichen Erfahrungen würde ich weder zu TDD raten noch davon abraten. Ich würde einen abgespeckten Workflow vorziehen, der den Testaufwand auf logische Fehler reduziert, die einfach überprüft werden können und die Integrationstests nur auf einfache Fälle wie oben beschrieben beschränkt. Für die restlichen Testpunkte bevorzuge ich den klassischen Ansatz, da sich der Aufwand dadurch reduziert.

Ferner macht es Sinn, die Tests von spezialisierten Teams schreiben zu lassen, da dadurch das Problem der Whitebox-Tests weiter umgangen wird und nicht jeder Entwickler tests schreiben will. Ferner spart es Schulungsaufwand.

[1] Siehe die Website `http://golang.org`
[2] Siehe die Website `http://dlang.org`

Glossary

Beispielgesteuerte Entwicklung häufig auch: example driven development, alternative Form der Testgetriebene Entwicklung. 15

EDD Beispielgesteuerte Entwicklung. 15

Extreme Programming Agiles Softwareentwicklungsparadigma. 4, 18

IDE Integrated Development Environment. 7

Integrated Development Environment Bezeichnet einen speziell für die Softwareentwicklung gebauten Texteditor, der häufig auf eine Sprache spezialisiert ist und für diese bereits Compiler und Debugger mitliefert.. 7, 18

Production Code Quelltext, der in das finale Produkt eingebaut wird. 4

TDD Testgetriebene Entwicklung. 4, 7, 14, 15, 17

Testgetriebene Entwicklung häufig auch testdriven development. 4, 18

XP Extreme Programming. 4