Management im Software Engineering
Universität Würzburg
WS 2013/2014

Ausarbeitung

Test Driven Development

Philipp Jeske Benjamin Morgan

15. Dezember 2013

Inhaltsverzeichnis

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 steht 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 vielen 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 schwer wiegender, 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 werden kann. 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 Testen 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 den hohe Codequalität gefordert wird. Bei Testgetriebene Entwicklung (TDD) steht das regelmäßige Testens seines Codes bereits in 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

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 Vorraussetzungen

Benjamin
Morgan

5 Beispiel

Philipp
Jeske

6 Vorteile

Benjamin
Morgan

7 Nachteile

Benjamin
Morgan

8 Fazit

Philipp
Jeske

9 Erfahrungsbericht

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

9.1 Benjamin Morgan

Benjamin Morgan

9.2 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 zum Testen von 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 ab. 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 nicht alle Entwickler die Testenwicklung beibringen zu müssen.

Literaturverzeichnis

Glossary

Extreme Programming Agiles Softwareentwicklungsparadigma. 4, 15

IDE Integrated Development Environment. 6

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.. 6, 15

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

TDD Testgetriebene Entwicklung. 4, 6, 13

Testgetriebene Entwicklung häufig auch testdriven development. 4, 15

XP Extreme Programming. 4