Putting TDD to the test

Edwin Günthner, IBM Germany Development Lab

Der englische Ausdruck „Putting something to the test" bedeutet so viel wie „etwas auf den Prüfstand stellen." Genau darum geht es in diesem Artikel: den Nutzen der Methodik „Test Driven Development" (TDD) anhand von praktischen Erfahrungen aus dem realen Umfeld in einem Großunternehmen zu bestimmen und auszuloten. Dabei dient die persönliche Lernkurve des Autors als Ausgangsbasis für weiterführende Diskussionen rund um das Thema „TDD und Unit Testing".

Sie lesen richtig: Wir schreiben das Jahr 2017, und ich schreibe einen Artikel über TDD. Zu einem Zeitpunkt, an dem andere erklären, weshalb diese Methode in der Praxis versagt hat *[1]*. Heute zählen doch Agilität und die Fähigkeit, sich schnell wechselnden Anforderungen zu stellen. Da bleibt keine Zeit für rigide Strukturen beim Entwickeln – und überhaupt – sind Menschen nicht wichtiger als Prozesse? Sollte es nicht darum gehen, den Entwicklern zu vertrauen? Natürlich. Was dabei jedoch gerne unter den Tisch fällt: Damit das alles effektiv funktionieren kann, bedarf es auch des Vertrauens des Entwicklers in den geschriebenen Code.

Ohne dieses Vertrauen kommen wir in eine Situation, die Michael Feathers in „Working effectively with Legacy Code" wie folgt beschreibt: „What do you think about when you hear the term legacy code? If you are at all like me, you think of tangled, unintelligible structure, code that you have to change but don't really understand." Wir verstehen: Mit Legacy Code zu arbeiten, ist schwierig und unangenehm. Aber was zeichnet solchen Code aus? Feathers findet eine bestechend einfache und zugleich tiefgründige Antwort: „To me, legacy code is simply code without tests." Wer also vermeiden will, sich morgen über Code zu beschweren, den er gestern selbst geschrieben hat, für den lohnt sich auch heute die Beschäftigung mit dem Thema „Unit-Tests" und mit einer der zentralen Methoden zum Schreiben eben dieser Tests – TDD!

Dieser Artikel ist keine vollständige Einführung ins Thema „wie schreibe ich Unit-Tests mit TDD" – dazu gibt es schon genügend gute Bücher und Tutorials. Es ist allerdings dennoch wichtig sicherzustellen, dass jeder das gleiche Verständnis der verwendeten Begriffe hat.

Betrachten wir zunächst den Begriff „Unit Testing". Bei *[2]* findet sich: „In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use." Mit anderen Worten: Alles was man tut, um ein Programm irgendwie zu testen, ist ein Unit-Test. Eine so weit gefasste Definition erlaubt aber keinerlei Abgrenzung zu anderen Arten von Tests. Daher ist sie wenig hilfreich.

Dieser Artikel basiert stattdessen auf dem Verständnis guter Unit-Tests, wie sie die Website „ArtOfUnitTesting" *[3]* oder der Wikipedia-

Artikel zum Thema „Modultest" [4] beschreiben. Was man jedoch nicht vergessen darf: Die erstgenannte, unscharfe Definition ist durchaus gebräuchlich. Ich erlebe häufig, dass Kollegen über Unit-Tests sprechen, aber eigentlich Funktions- beziehungsweise Integrationstests meinen. In aller Regel ist es (leider) effektiver, sich selbst immer wieder diese unterschiedlichen Sichtweisen bewusst zu machen als zu versuchen, die eingefahrene Sichtweise anderer zu verändern.

Der Begriff „TDD" [5] ist eindeutig definiert: Im Kern geht es darum, dass eine Anforderung zuerst als ausführbarer Test spezifiziert wird. Erst danach erfolgt die eigentliche Implementierung. Diese ist erst abgeschlossen, wenn der neu hinzugefügte sowie alle anderen, bereits existierenden Tests erfolgreich durchlaufen worden sind. Bevor die nächste Anforderung bearbeitet wird, widmet man sich explizit der Qualität des neu geschriebenen Codes – man verbessert dann ausschließlich die Art und Weise, wie die Anforderung umgesetzt wird. Die existierenden Tests stellen sicher, dass die gewünschte Funktionalität durch die zusätzlichen Änderungen am Code nicht verändert wird. Das Ganze erfolgt in zyklischen Iterationen – wobei jeder einzelne Zyklus idealerweise nur wenige Minuten dauern sollte.

Nach Klärung dieser Begrifflichkeiten können wir uns jetzt den praktischen Aspekten zuwenden – wie schon angedeutet aus einer (zunächst) subjektiven Perspektive: Der erste Kontakt zu TDD hat sich für mich vor etwa fünf Jahren ergeben, als ich innerhalb der IBM zum Bereich „System Z Firmware" gewechselt bin. Sowohl im Kontakt zu Kollegen als auch bei Vorträgen beim Java Forum Stuttgart entstand der Eindruck, in TDD eine vielversprechende Methodik zu finden, deren Anwendung in kurzer Zeit die Qualität des geschriebenen Codes verbessern kann. Auch die Kollegen im neuen Team waren schnell überzeugt – das „wir machen das jetzt einfach" kam tatsächlich von Herzen. Nach wenigen Monaten stellte sich Ernüchterung ein: „Das hilft ja gar nicht". Konkret zeigte sich:

- Die meisten Tests entstanden nachträglich – sie wurden erst geschrieben, nachdem der Produkt-Code schon relativ weit fortgeschritten war.
- Die Tests waren umständlich, schwer zu verstehen – und vor allem: Sie haben mehr Probleme gemacht als gefunden.

Die direkte Konsequenz war eine spürbare Verringerung der Motivation einiger Team-Mitglieder, sich weiter mit der Thematik zu beschäftigen. Glücklicherweise blieb aber die Bereitschaft, die TDD-Aktivitäten Einzelner nicht einzuschränken.

Das bedeutet konkret, dass ich zunächst regelmäßig kleinere Aufgaben mit TDD bearbeitet habe. Im Jahr 2015 hat sich dann für mich die Möglichkeit ergeben, eine größere Problemstellung vollständig mit TDD umzusetzen. Ich arbeite innerhalb des Bereichs IBM Z und schreibe dort Java-Code für die Hardware Management Console (HMC) [6].

Die HMC ist das Management Interface für IBM Z Mainframes. Das Projekt „Dynamic Partition Manager" (DPM) [7] hat sich die Aufgabe gestellt, Teile dieser Management-Funktionalität in einer zeitgemäßen Art und Weise neu zu implementieren. Ein zentrales Element ist hierbei das Starten/Stoppen von Gast-Systemen auf dem Mainframe.

Mein Team hatte bereits einen Prototyp der neuen Start/Stopp-Funktion implementiert. Es zeigte sich allerdings, dass der neue Code erhebliche Defizite aufwies. Zum Beispiel fehlte eine durchgehende Fehlerbehandlung unter Berücksichtigung der notwendigen (kontextabhängigen) Rollback-Schritte. Da es auch keine Unit-Tests gab, waren selbst einfache Refactoring-Änderungen mit erheblichem Test-Aufwand verknüpft. Mit anderen Worten: Der neu geschriebene Code war innerhalb weniger Monate zu Legacy Code geworden. Da die notwendige Funktionalität überschaubar war, entschloss ich mich dazu, die ruhige Zeit am Jahresende zu nutzen, um eine komplett neue

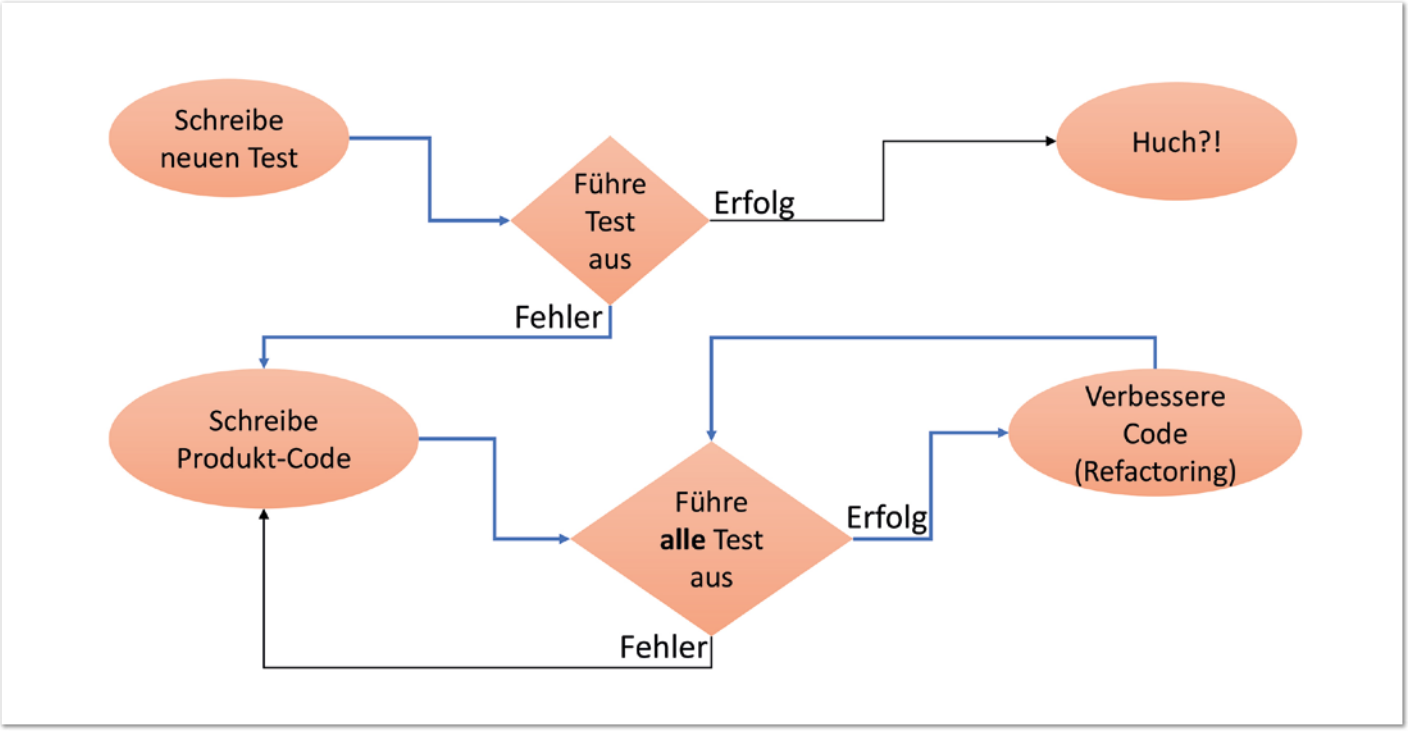

Abbildung 1: TDD im Überblick

Implementierung zu schreiben – und da ich es besser machen wollte, mit der Vorgabe, konsequent TDD einzusetzen. Die Erfahrungen, die ich dabei gesammelt habe, möchte ich im Folgenden darstellen.

Die erste und wichtigste Erkenntnis hat überraschenderweise nichts mit Technik oder Code-Qualität zu tun. Sie lautet nämlich: Konsequentes TDD macht Spaß. Rufen wir uns zunächst mit *Abbildung 1* in Erinnerung, wie TDD abläuft (die blauen Pfeile stehen für den normalen Geradeaus-Pfad beim Einsatz der Methodik).

Der erste Schritt besteht darin, eine neue Anforderung als ausführbaren Test zu formulieren. Danach ist es wichtig sicherzustellen, dass der neue Test beim Ausführen zu einem Fehler führt – falls der Test funktioniert, stimmt irgendetwas nicht – entweder ist der Test inhaltlich falsch oder aber unsere Erwartungshaltung. Anschließend wird Produkt-Code erzeugt (idealerweise so wenig wie möglich – es geht nur darum, diesen einen, neuen Test erfolgreich zu bestehen). Dann werden alle Tests ausgeführt (vorzugsweise: genau die Tests, die für den betroffenen Produkt-Code existieren).

Schlagen Tests fehl, gilt es die Ursache zu bestimmen und zu beseitigen. Wenn alle Tests erfolgreich sind, widmet man sich entweder der nächsten Anforderung – oder verwendet Zeit darauf, die Qualität des bisher geschriebenen Codes durch Refactoring zu verbessern. In Anlehnung an die grafische Darstellung von Tests in IDEs wie Eclipse spricht man auch vom „red/green/refactor"-Kreislauf. Wie bereits erwähnt, erfolgen diese Iterationen in sehr schneller Abfolge – ein einzelner Zyklus sollte nur wenige Minuten dauern (und den überwiegenden Teil dieser Zeit verbringt man idealerweise in der Refactoring-Phase). Diese Arbeitsweise führt beinahe automatisch zu einem positiven Arbeitsgefühl:

- Es ergibt sich eine kontinuierliche Serie von Herausforderungen (neuer, roter Test), auf die jeweils umgehend eine Belohnung (neuer Produkt-Code und grüne Tests) folgt.
- Gleichzeitig laufen diese Vorgänge stets im gleichen Kontext ab: in der Entwicklungsumgebung des Entwicklers.

Wenn neben der Anwendung dieser Vorgehensweise noch die Möglichkeit besteht, Störungen von außen zu minimieren oder (zumindest zeitweise) gänzlich zu eliminieren – dann kann sich sehr schnell ein echtes Flow-Erlebnis einstellen: Man kann sich mit voller Konzentration genau einer Aufgabe widmen. Kreative Ideen werden nicht nur sofort umgesetzt. Man erhält auch umgehend Feedback darüber, ob der neu geschriebene Code sich wie erwartet verhält.

Übrigens gilt auch die Umkehrung: Je länger man auf solches Feedback warten muss, desto unbefriedigender fühlt sich die eigene Arbeit an. In meinem Fall sind für echte Funktionstests die folgenden Schritte notwendig, um einen Patch zu testen:

1. Identifizieren genau derjenigen Java-Klassen, die von der Änderung betroffen sind
2. Erstellen eines Archivs mit allen zu Punkt 1 korrespondierenden, binären Artefakten
3. Einspielen des Archivs in einem (eventuell vorher zu erzeugenden) Test-System

Bestenfalls benötige ich für diese Schritte fünf bis zehn Minuten – bei veränderten Randbedingungen gerne auch dreißig Minuten oder mehr. Die Frage nach flüssigen Arbeitsabläufen stellt sich an dieser Stelle nicht mehr.

Diese drei Punkte habe ich nach ungefähr vier Wochen mit „purem TDD" abgearbeitet, um die Frage „Funktioniert der neue Code denn nun wirklich?" beantworten zu können. Überraschenderweise lautete die Antwort „nein". Bei der anschließenden Analyse zeigte sich schnell, dass die Ursache für den fehlgeschlagenen Versuch ein Fehler in Schritt 2 war – der erzeugte Patch war schlicht und ergreifend unvollständig. Neuer Patch, neuer Test – alles funktioniert wie erwartet.

Weiteres Nachdenken über den Fehlschlag führte zu der Erkenntnis: „Der neue Code hätte diesen Fehler anders verarbeiten müssen". Ich kam also nach vier Wochen intensiver Arbeit zu dem Schluss, dass im neuen Produkt-Code ein Design-Fehler steckt. Das Problem ließ sich aber wieder als Unit-Test darstellen und somit mit TDD bearbeiten.

In der Folge war es notwendig, etliche Klassen im Produkt-Code zu ändern. Nach wenigen Stunden waren alle zugehörigen Tests angepasst und konnten wieder erfolgreich ausgeführt werden. Und ich konnte mir sicher sein, dass sowohl das neue Problem behoben war, als auch, dass durch die vorgenommenen Änderungen keine anderen Fehler eingebaut wurden. Eine komplexe Änderung im Produkt-Code konnte also dank vollständiger Abdeckung der Code-Basis durch Unit-Tests in kürzester Zeit durchgeführt werden.

Nach etlichen weiteren Wochen Programmieren mit TDD waren alle Anforderungen implementiert. Das Ergebnis der anschließenden Test-Phase lautete: Keine Bugs gefunden. Im Detail:

- Keine Übertragungsfehler: Die neu geschriebene Komponente verhielt sich wie die alte Komponente.
- Die neu hinzugefügte Fehlerbehandlung (zur Bearbeitung komplexer Rollback-Szenarien) lieferte die erwarteten Ergebnisse. Gleiches gilt für neu eingebaute „Debug Support"-Funktionen.

Auch in den etwa achtzehn Monaten, in denen die neue Komponente genutzt wurde, wurden keine Fehler gefunden.

Bevor ich mich wieder der Theorie und der Frage nach den Voraussetzungen für den erfolgreichen Einsatz von TDD zuwende, ist es mir wichtig, einige Punkte zu benennen, die in meinem Experiment nicht funktioniert haben:

- Meine Abschätzung des Aufwands lag bei vier bis maximal acht Wochen. Tatsächlich habe ich eher sechzehn bis zwanzig Wochen benötigt.
- Es ist mir nicht gelungen, meine eigenen Erwartungen an Code-Qualität konsequent umzusetzen; zum Beispiel fanden Peer Reviews gar nicht beziehungsweise zu spät statt.

Aber: Die Methode TDD erhebt nicht den Anspruch, alle Probleme zu lösen. Sie gibt einen Rahmen vor, in dem man sehr kreativ und effizient agieren kann – mehr allerdings auch nicht.

Was unterscheidet diese Erfolgsgeschichte von den ersten Erfahrungen in meinem Team, die eher von Frustration gekennzeichnet waren? Die naheliegende Vermutung wäre: „Ich habe gelernt, bessere Tests zu schreiben." Das ist richtig, aber nur die halbe Wahrheit

– der wesentliche Punkt ist nämlich: „Ich habe gelernt, Produkt-Code zu schreiben, der testbar ist."

Es ist letztendlich sehr einfach: Wer eine Klasse schreibt, die zwanzig Felder hat und in der jede Methode fünf Parameter erwartet, um dann über mehr als hundert oder mehr Zeilen hinweg mehrere Dinge gleichzeitig tun – der muss sich nicht wundern, wenn diese Klasse gar nicht oder zumindest nur schwer und unvollständig testbar ist, und zwar unabhängig davon, ob Unit-Tests oder funktionale Tests am echten System zum Einsatz kommen. Im Umkehrschluss bedeutet das: Wenn ich eine Klasse mit Unit-Tests in den Griff bekommen will, sollten die folgenden Bedingungen für die Klasse beziehungsweise ihre Methoden erfüllt sein:

- Die Klasse kann in Isolation betrachtet werden
- Es gibt die Möglichkeit, die Klasse von ihren Abhängigkeiten zu entkoppeln
- Der Code implementiert genau eine Funktion

Oder, um es auf einen Nenner zu bringen: Der Produkt-Code muss mit dem Wissen über „Clean Code" geschrieben werden. Daraus folgt: Wer TDD und Unit Testing sinnvoll einsetzen möchte, muss sich nicht nur mit dem Themenkomplex „TDD" beschäftigen – also beispielsweise Bücher wie „JUnit-Profiwissen" (Michael Tamm) oder „xUnit Test Patterns" (Gerard Meszaros) intensiv bearbeiten. Nein: Darüber hinaus ist es unerlässlich, auch die Klassiker zur Code-Qualität – etwa „Clean Code" beziehungsweise „Agile Principles" (Robert Martin) oder „Refactoring" (Kent Beck) – zu verinnerlichen und immer wieder praktisch einzuüben.

Innerhalb unseres Teams hat es sich an dieser Stelle als besonders wertvoll erwiesen, regelmäßige Code Reviews durchzuführen, die ausschließlich die Clean-Code-Prinzipien *[8]* behandeln. Die kontinuierliche Beschäftigung mit beiden Themenkomplexen ist zwingend notwendig, um die initiale Frustration zu überwinden und zu echten Verbesserungen gelangen zu können. Übrigens ist das keine neue Erkenntnis – zum Beispiel hat Michael Feathers bereits im Jahr 2013 dazu den empfehlenswerten Vortrag „The deep synergy between testability and good design" *[9]* gehalten.

Eine kurzfristige Anordnung des Managements an die Entwickler: „Ihr macht jetzt mal zwei Monate TDD", am besten verbunden mit „und danach erwarten wir, dass die Fehler-Quote um x Prozent sinkt", kann also gar nicht funktionieren. Das Management muss sich vielmehr auf jahrelange Lernprozesse einstellen. Darüber hinaus wollen die Entwickler jetzt auch noch erheblichen Aufwand in das Schreiben von Tests investieren. Fragen wie: „Rentiert sich das überhaupt?" und „Werden wir nicht viel langsamer, wenn die Entwickler die Hälfte der Zeit mit dem Schreiben von Tests verbringen?", liegen dann natürlich nahe.

Wie ich jedoch bereits ausgeführt habe – das Gegenteil kann der Fall sein: TDD und Unit-Tests können dazu führen, dass die Entwicklung schneller vorangeht. Man muss sich klarmachen: Das Ziel bei der Entwicklung von Software ist nicht das Schreiben von Code. Das Ziel besteht vielmehr darin, am Ende ein Produkt liefern zu können, dass sowohl im Funktionsumfang als auch in der Qualität den Erwartungen der (zahlenden) Kunden entspricht.

Wenn ich als Entwickler zunächst m Zeilen Test-Code schreibe, um dann n Zeilen Produkt-Code folgen zu lassen, führt das genau dann zu einem positiven Return on Investment, wenn die m Zeilen Test-Code mir helfen, die erwartete Funktion insgesamt schneller zu liefern. Es geht nicht darum, die Anzahl der geschriebenen Zeilen Code zu minimieren, sondern darum, die Zeit bis zur Auslieferung der nächsten Funktion (mit hoher Qualität) möglichst kurz zu halten. TDD/Unit-Tests verkürzen diese Zeit deswegen, weil sie die Zeitspanne (massiv) verkürzen, die ich als Entwickler auf Feedback warte.

Aus dieser Beobachtung lässt sich ableiten, in welchen Fällen TDD/Unit Testing nicht oder nur bedingt hilfreich ist, nämlich genau dann, wenn die Entwickler über andere Wege schnelles Feedback erhalten können. Fred George beschreibt in *[10]* ein System, in dem die Entwickler Änderungen sofort ins Produktions-System einstellen können. Ist die neue Funktion gut, steigen die Gewinne wenige Sekunden nach dem Aktivieren der Änderung. Fallen die Gewinne jedoch, wird umgehend die alte Funktionalität wiederhergestellt. In diesem System benötigt man kein TDD, weil die meisten Änderungen gut sind – und weil jede Sekunde, die eine Änderung früher verfügbar ist, sofort als Gewinn in die Bilanz eingeht. Aber Entwickler, die in einem solchen System arbeiten, stellen vermutlich die große Ausnahme dar. Alle anderen, die schnelles Feedback erhalten wollen, können einstweilen auf TDD/Unit Testing zurückgreifen.

Fazit

TDD und Unit-Tests sind Methoden, mit deren Hilfe Entwickler schnelles Feedback über Code erhalten können. Um TDD sinnvoll zu nutzen, ist es weniger wichtig, sich stur an die Regeln der Methodik zu halten. Vielmehr geht es darum, das komplexe Zusammenspiel von Codequalität (gemäß Clean Code) und hilfreichen Unit-Tests regelmäßig einzuüben und durch geeignete Maßnahmen zu unterstützen. Dann kann TDD seine volle Wirkung entfalten – und gut gelaunte Entwickler dabei leiten, hochwertigen Produkt-Code zu schreiben.

Links

[1] *https://blogs.msdn.microsoft.com/ericgu/2017/06/22/notdd*
[2] *https://en.wikipedia.org/wiki/Unit_testing*
[3] *http://artofunittesting.com/definition-of-a-unit-test*
[4] *https://de.wikipedia.org/wiki/Modultest*
[5] *https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung*
[6] *http://ibm.biz/redbook-hmc*
[7] *http://ibm.biz/dpm-intro*
[8] *http://clean-code-developer.de*
[9] *https://www.youtube.com/watch?v=4cVZvoFGJTU*
[10] *http://www.se-radio.net/2016/03/se-radio-episode-253-fred-george-on-developer-anarchy*

Edwin Günthner
edwin.guenthner@de.ibm.com

Der Autor hat an der Universität Karlsruhe studiert und dort im Jahr 1999 den Abschluss als Diplom-Informatiker gemacht. Seit dem Jahr 2001 arbeitet er bei der IBM Deutschland Research & Development GmbH in Böblingen in wechselnden Rollen in den Bereichen zSystems-Hardware und Firmware-Entwicklung.