Behaviour Driven Testing und automatische Unit-Test-Generierung

Mehr Effizienz im Testen

Johannes Bergsmann, Software Quality Lab

In vielen Entwicklungsorganisationen existiert eine Lücke zwischen der Fachabteilung und der Testautomatisierung. Der Fachbereich spezifiziert Tests oft funktional. Es fehlen hier oft Details zum Verhalten. Umgekehrt wird die Implementierung der automatischen Tests durch den Fachtester oft mangels Entwickler-Knowhow nicht verstanden. Sehr oft werden auch zu wenige Tests spezifiziert und automatisiert und die erreichte Testabdeckung ist in manchen Fällen sogar fahrlässig gering.
BDT (Behavior Driven Testing) ist eine Technik aus dem agilen Entwicklungs-umfeld, die genau diese Lücke zwischen Fachtester und Automatisierer schließt. Als Ergänzung zu einem strukturierten Testautomatisierungsansatz (wie z.B. BDT) ist es oft zusätzlich noch sinnvoll, die Testlücken automatisch durch generierte Unit-Tests abzudecken.

Test Driven Development (TDD) als Vorstufe zu BDT

Michael Feathers ist einer der „Gurus“ in der Software-Entwicklung und hat schon vor längerer Zeit den Satz geprägt:
“I don't care how good you think your design is.
If I **can't** walk in and **write a test** for an arbitrary method of yours **in five minutes** it's not as good as you think it is, …
… and whether you know it or not, **you're paying a price for it!**”
Dieses Zitat betont, dass das Verstehen eines Programms (z.B. für einen Tester) einfacher und besser wird, je besser es strukturiert ist. Damit wird natürlich auch die Qualität der Software insgesamt positiv beeinflusst.
Um das Verstehen und die Qualität der Software schon möglichst früh sicher zu stellen, ist es daher sinnvoll, bereits in einem sehr frühen Stadium der Software-Erstellung die Tester einzubeziehen. Idealerweise noch bevor die erste Zeile programmiert wird.
Dieser Ansatz wird „Testgetriebene Softwareentwicklung“ genannt.
Hier werden durch einen Entwickler oder auch durch einen Tester/Testautomatisierer mit entsprechenden Fähigkeiten die zum Testen der gewünschten Software notwendigen Testfälle bereits VOR der Programmierung spezifiziert und im Idealfall auch automatisiert. Theoretisch ist testgetriebene Softwareentwicklung auch ohne Testautomatisierung umsetzbar. Dieser Ansatz ist jedoch sehr ineffizient. Daher werden bei TDD fast ausschließlich automatisierte Tests verwendet.
Die beschriebenen und automatisierten Testfälle können auch als (detaillierte) Requirements-Spezifikation betrachtet werden. Testfälle sind je im Grunde nur eine andere Form von spezifizierten Anforderungen an die Software.
TDD kann auf mehreren Ebenen angewendet werden:

- Testen im Kleinen (durch Unit Tests) und

- Testen im Großen (durch Integrations- und Systemtests)

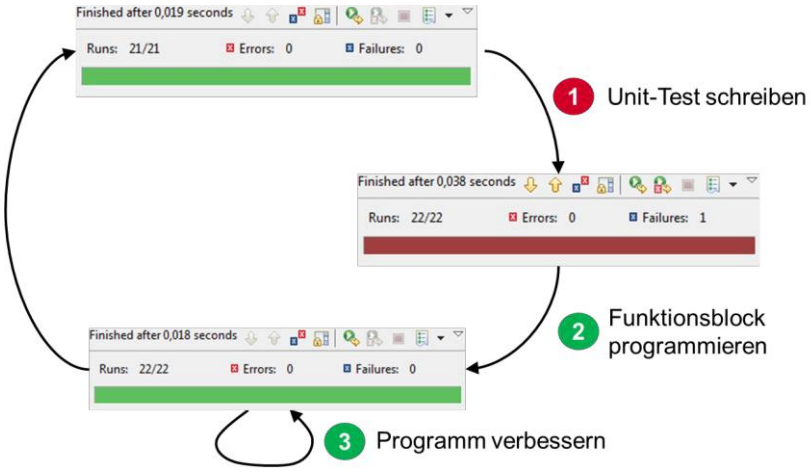


Abb. 1 – Grundlegender Ablauf von Test Driven Development

Ziele von TDD ist es, von Anfang an testbaren und wartbaren Code zu erstellen und die Testabdeckung zu erhöhen. Ein weiterer Vorteil ist, dass in (agilen) Projekten, in denen oft wenig explizite Requirements-Spezifikation erstellt wird, die Anforderungen an die Software hier aufgrund der Vorgehensweise in Form der Testfälle gut spezifiziert und sogar automatisch überprüfbar definiert werden.

Die Vorteile von TDD sind:

- Die Qualität der Entwicklung wird verbessert
- Die Testabdeckung wird erhöht
- Es existiert eine bessere Basis für die Sprint-Abnahme (Test = Requirements)
- Die Effizienz im Sprint wird erhöht
- Es wird eine „lebende“ (Test-)Spezifikation erstellt

Es gibt jedoch auch einige Herausforderungen:

- TDD alleine macht noch keine guten Requirements und Code
- Manches kann nicht vorab als Test automatisiert werden (Usablity, System-kontext, Prozessmodellierung, Architektur, etc.)
- Es ist eine sehr hohe Disziplin erforderlich!

Behaviour Driven Development (BDD) bzw. Behaviour Driven Testing BDT

BDD ist eine Variante des TDD. Diese Methode wurde bereits 2003 von Dan North als Antwort auf Test-Driven Development erarbeitet. Grund war, dass Unit-Tests für den Fachbereich, der ja oft kein Entwickler-KnowHow besitzt, schlecht lesbar sind.
Die Motivation bei BDD ist, dass die User Stories als weit verbreitete Form der Spezifikation in agilen Projekten die Anforderungen an eine Software sehr oft nur funktional beschreiben. Daher soll durch Behaviour Driven Development – wie der Name schon sagt – eine stärkere Verhaltens- und Prozess-Sicht eingebracht werden.
Die Idee bei BDT ist, dass die User Stories nun auch aus Sicht der Tester durch verhaltensorientierte Testbeschreibungen in einer einfachen aber doch strukturierten Form ergänzt werden sollen, sodass die Tests anschließend einfach automatisiert werden können.

Die Beschreibung soll dabei möglichst natürlichsprachlich, leicht verständlich und lesbar erfolgen. Somit ist es möglich, dass auch Nicht-Programmierer einen „Code“ für das automatisierte Testen erstellen.

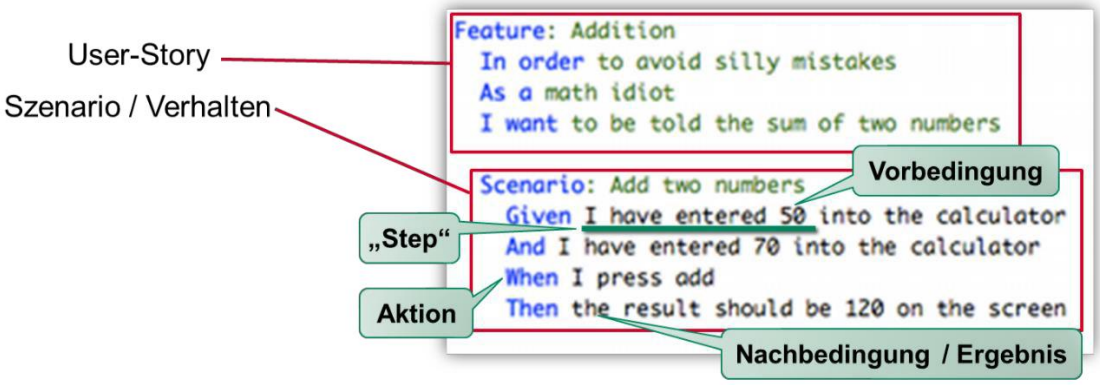


Abb. 2 – Struktur der Spezifikation bei Behaviour Driven Development (BDD)

Der Ablauf bei BDD ist wie folgt:

- Der Fachbereich oder Product-Owner erstellt User-Stories wie bisher
- Zusätzlich wird zu einer User Story nun jedoch auch das Verhalten in Form eines “Szenarios” beschrieben
- Ein Szenario beginnt mit einer Vorbedingung, die aus einem oder mehreren „Steps“ bestehen kann. Ein „Step“ ist ein zusammengehöriges Element, das auch wiederkehrend verwendet werden kann und das einen Teil eines Testfalls (z.B. eine Eingabe oder Ausgabe oder auch eine Aktion) enthalten kann.
- Wenn die Vorbedingungen gegeben sind, kann im nächsten Schritt die “Aktion” (oder der Trigger) ausgelöst werden.
- Dies führt dann im dritten Schritt zur „Nachbedingung“ (bzw. Ergebnis) der Aktion.

Durch diese Schritte wird ein Teil einer Software sowohl funktional als auch von seinem Verhalten beschrieben.
Die Testfälle lassen sich dadurch sehr einfach ableiten. Aus dem oben dargestellten Beispiel können die Tests wie folgt abgeleitet werden:

Trigger	Vorbedingung (Eingabe) Step 1	Vorbedingung (Eingabe) Step 2	Nachbedingung (Erwartetes Ergebnis)
Press Add	50	70	120

Durch die Parametrierung der Steps können für die Testausfühung beliebige Testdatenkombinationen verwendet werden und somit Standarfälle, Grenzwerte und Fehlerfälle mit einem abstrakten Testfall getestet warden.

Damit die in Prosa spezifizierten Tests dann auch einfach automatisiert werden können, gibt es speziflle BDD-Frameworks und BDD-Tools, welche diesen Ansatz unterstützen. Nachfolgend wird dies am Beispiel des Tools „Cucumber“ dargestellt:

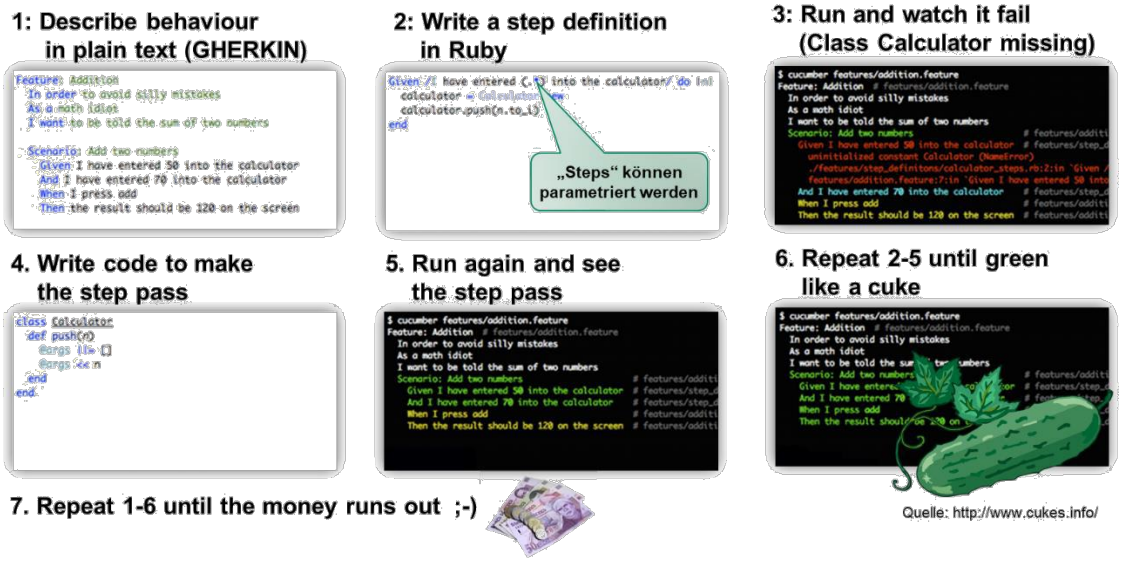


Abb. 3 –Behaviour Driven Development (BDD) am Beispiel Cucumber

BDD ist ein interessanter Ansatz, der versucht, eine bessere Kopplung zwischen den Beteiligten (Fachexperten, Tester und Entwickler) zu erreichen.
Die dabei erstellten automatisierten Regressionstests geben schnelles Feedback, was zu einer „lebenden“ Spezifikation und Dokumentation des Systems führt.

Es gibt jedoch auch hier einige Herausforderungen:

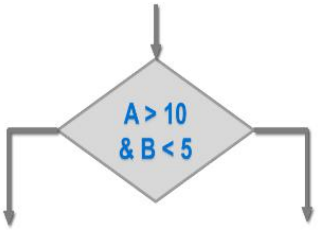
- Die Fachtester arbeiten typischerweise mit anderen Tools (z.B. Testmanagement-Tools)
- Es gibt meist keine tool-unterstützte direkte Kopplung zwischen Testspezifikation und Automatisierungscode, was die Konsistenz der Testfälle mit dem Testcode erschwert. Die in den BDD-Tools verwendeten Code-Aufrufe (Steps) werden oft textbasiert erstellt, ohne diese in der Spezifikationsumgebung mit dem generierten Testautomatisierungscode direkt zu verbinden. Dadurch führt jeder Tippfehler (egal ob beim Tester oder beim Automatisierer) zu einer Lücke im Zusammenhang und bricht damit die Automatisierung. Hier ist noch deutlicher Verbesserungsbedarf bei den Tools.
- Die derzeit am Markt verfügbaren Tools eignen sich ohne Ergänzung primär für entwicklungsnahe Tests (Unit-/Komponenten-Ebene) und kaum für Systemtests.

Generell kann jedoch BDD/BDT als interessanter Ansatz bewertet werden, der mittlerweile in der Praxis immer mehr Verbreitung findet. Der Bruch zwischen Fachtester und Testautomatisierer kann hier deutlich reduziert werden und mit einer passenden Integration der verfügbaren Frameworks (Testmanagement-Tool mit Automatisierungsumgebung und BDD-Tool) kann diese Lücke auch technisch in den Tools geschlossen werden.

Automatische Unit-Test Generierung
Die zunehmende Komplexität von Systemen führt dazu, dass immer mehr Tests notwendig sind, um dieses Systeme nach dem Stand der Technik abzusichern und die Haftung gegenüber Auftraggebern im Falle von Fehlern zu reduzieren.

Parallel dazu werden für die Entwicklung dieser Systeme immer mehr Software-Entwickler benötigt, die jedoch aktuell am Personalmarkt nicht im benötigten Ausmaß zur Verfügung stehen und dann ev. extern eingekauft werden müssen, was die Entwicklung teurer macht.
Entwickler müssen sich aufgrund der größeren Systemkomplexität auch immer mehr mit der Absicherung des Systems durch passende Unit-Tests beschäftigen (ca. 20-30% der Entwicklerzeit werden für die Unit-Testerstellung aufgewendet). Dadurch wird die Geschwindigkeit der eigentlichen Entwicklung gebremst. Wenn Produkte dadurch erst später auf den Markt gebracht werden können, führt dies zu hohen Umsatzverlusten.

Die funktionale Abdeckung durch Tests ist oft viel geringer, als es erforderlich wäre.
Eine einfache Verzweigung als Beispiel (Grafik rechts):
Hier schreibt der Entwickler typischerweise 2 Tests, die einmal den Weg links und einmal rechts gehen. Damit wird 100% Codeabdeckung erreicht und die meisten sind damit zufrieden.
Um die Verzweigung in der Grafik jedoch funktional gut zu testen, sind mindestens 14 Testfälle notwendig:



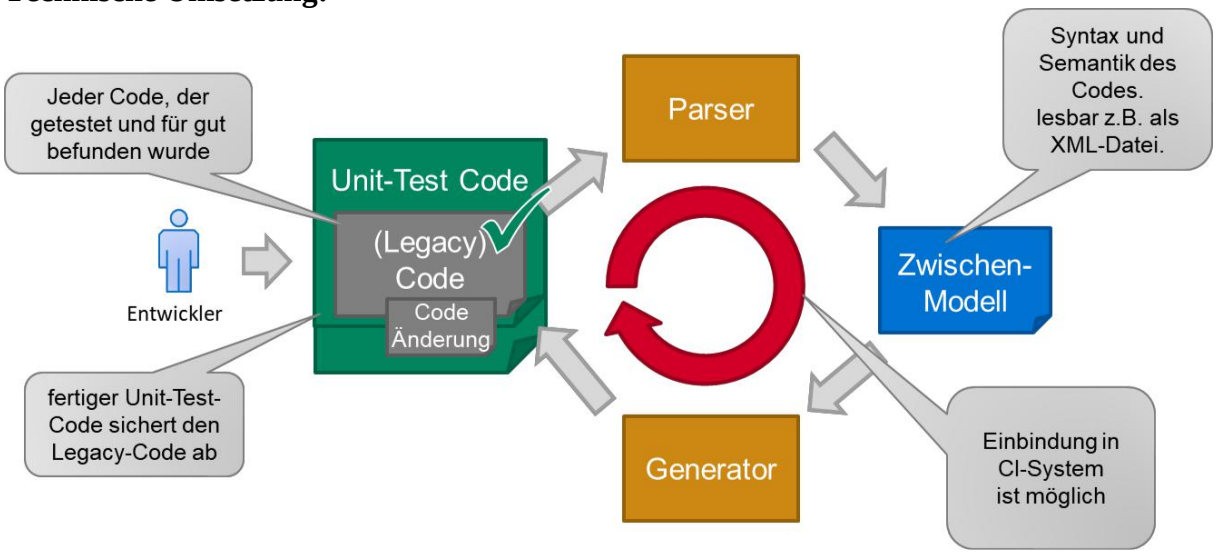
- Jede (Einzel-)Bedingung muss für sich getestet werden (4 Normal-Testfälle)
- Grenzwerte (oben/unten) der beiden Bedingungen (weitere 4 Testfälle)
- Fehlertests außerhalb der Grenzen oben und unten sowie innerhalb der Grenzen mit Fehlerwerten (weitere 6 Testfälle)

	T1	T2	T3	T4
A > 10	1	1	0	0
B < 5	1	0	1	0

Die Erstellung von (Unit-)Tests beansprucht daher immer mehr Zeit und Kosten!

Es ist daher notwendig, auch die Erstellung der Tests zu automatisieren, soweit dies möglich ist.
Nachfolgend wir ein Ansatz vorgestellt, durch den ein großer Teil der heute oft noch manuell erstellten Unit-Tests automatisch erstellt werden kann.

Technische Umsetzung:



Ablauf:

1.) Der Entwickler produziert Code, der aus seiner Sicht passend ist (er findet keine Fehler mehr). Diesen Code übergibt der Programmierer an das Codeverwaltungssystem.
2.) Der übergebene Code wird von einem Parser (Programm zur Code-Analyse) ausgelesen, der daraus ein Zwischenmodell der Software erstellt. Das Zwischenmodell enthält alle testrelevanten Elemente (Eingabe, Ausgabe, Schleifen, Verzweigungen, etc.) und deren Zusammenhänge.
3.) Dieses Modell verwendet ein Testgenerator als Basis, um daraus sinnvolle Unit-Tests mit einer hohen funktionalen Abdeckung für den Programmcode zu generieren.
4.) Die Unit-Tests werden in der zur jeweiligen Programmiersprache passenden Form generiert und können dann in der Programmierumgebung weiterverarbeitet werden, wie wenn diese durch den Entwickler manuell erstellt worden wären.

Die Entwickler können wie bisher einige wenige Unit Tests erstellen, die zum Verständnis der Software hilfreich sind oder ev. nicht automatisiert werden können. Der Unit Test Codegenerator verbessert dann automatisch die Abdeckung des Codes mit passenden Tests.

Resümee

Test Driven Development (TDD) ist ein guter Ansatz, um die Qualität in der Entwicklung zu verbessern. Es erfordert jedoch sehr hohe Disziplin! Außerdem sind die sehr entwicklernahe erstellten automatischen Tests für die Fachtester oft nicht oder schwer lesbar.

Des Weiteren wurde erkannt, dass User Stories meist zu stark funktional spezifiziert werden und die Verhaltens- und Prozess-Sicht fehlt. Daher wurde Behaviour Driven Development (BDD) als Variante von TDD entwickelt. BDD ist für die Zusammenarbeit zwischen Fachtester und TA sehr vielversprechend und hat den Fokus auf dem Verhalten der Software. Die Tester-Sicht wird durch passende Tools nahtlos gekoppelt mit der Testautomatisierung.

Durch die steigende Komplexität in modernen Systemen und auch den Mangel an Software-Entwicklern ist meist eine unzureichende Absicherung durch Unit tests gegeben. Diese Lücke kann aus Kostengründen meist nicht durch manuelle Erstellung von Unit-Tests ausreichend ausgeglichen werden. Es ist daher notwendig, Ansätze für die automatische Absicherung von erstelltem Source-Code anzuwenden. Bestehende Tools bieten hier jedoch meist keine Unterstützung. Bei Software Quality Lab wird gemeinsam mit der technischen Universität Wien ein innovativer Ansatz zur automatischen Test-Code Generierung erforscht und umgesetzt.