Test-Driven-Development

AVM Workshop Poster

Ambs, Brandenburger, Jerke, Klimpel, Tögel
HTWG Konstanz, Fakultät Informatik

Contact Information:
Fakultät Informatik
HTWG Konstanz
Alfred-Wachtel-Str. 8 78462 Konstanz

GitHub: msi-se/avm-workshop-tdd

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Zusammenfassung

Test-Driven Development (TDD) ist eine Softwareentwicklungsmethodik, bei der die Entwicklung um das Schreiben von Tests herum strukturiert ist. Ziele von TDD sind durch hohe Codequalität eine robuste und besser wartbare Software zu entwickeln. Der TDD Prozess besteht aus den Schritten "red, green, refactor". Im Schritt "red"wird ein Tests geschrieben, der alle Anforderungen an Funktionalitäten abdeckt. Im Schritt "green"werden die Funktionalitäten implementiert bis die Tests bestehen. Im Schritt "refactor"wird der Code optimiert. Nachteile von TDD sind ein hoher initialer Zeitaufwand, ein erforderliches Umdenken der Entwickler, es ist nicht für alle Projekte geeignet und die Gefahr für Overengineering. TDD kann agil betrieben werden und ist mit den agilen Prinzipien vereinbar, wenn es effektiv praktiziert wird. Best Practices für TDD sind Tests klein und unabhängig zu halten, so wenig Code wie nötig für das bestehen der Tests, schreiben von positiven und negativen Testsfällen und das wiederholende ausführen der Tests.

Grundlagen und Ziele

Test-Driven Development (TDD) ist eine Softwareentwicklungsmethodik, die sich durch das Ziel auszeichnet, die Qualität und Zuverlässigkeit von Code zu verbessern. Der Kerngedanke hinter TDD ist es, den Entwicklungsprozess um das Schreiben von Tests herum zu strukturieren. Diese Tests definieren die Anforderungen und das gewünschte Verhalten der Software, noch bevor die eigentliche Implementierung beginnt. TDD fördert damit ein tieferes Verständnis für die Anforderungen und hilft, Fehler frühzeitig zu erkennen und zu beheben. Durch diesen Ansatz wird eine kontinuierliche Überprüfung und Verbesserung des Codes ermöglicht, was zu robusteren und besser wartbaren Softwareprodukten führt. [3]

Ablauf

Bei Test-Driven-Development (TDD) wird die Entwicklung von Software durch einen Zyklus aus Testen und Implementieren gesteuert. Dabei werden die Tests optimalerweise vor der Implementierung geschrieben, in Einzelfällen auch parallel dazu. Der genaue Ablauf eines Zyklus unterscheidet sich zwischen verschiedenen Quellen leicht, im Kern ist er aber immer gleich. In diesem Poster werden die typischen drei Schritten („red, green, refactor" z. B. nach dem Vorgehen von „Agile-Ikone" Kent Beck (vgl. [1] S. x)) um einen weiteren verbreiteten Schritt erweitert, um möglichst viel abzudecken.

- **0. Schritt: „Think"**
  Zunächst sollten die Anforderungen an die gerade zu implementierende Funktionalität klar sein. Wenn dies nicht der Fall ist, muss es erneute Absprachen geben. Der/die Entwickler:in sollte sich hier aus Sichtweise des Nutzers der Softwarekomponente in die Funktionalität hineinversetzen. In diesem Schritt muss auch entschieden werden, welche Anforderungen genau in diesem Zyklus umgesetzt werden sollen und welche erst in einer zukünftigen Iteration in Betracht gezogen werden.
- **1. Schritt: „Red"**
  Im ersten richtigen Schritt wird ein Test geschrieben, der alle Anforderungen an die Funktionalität abdeckt. Hierzu können verschiedene Testarten verwendet werden, z. B. Unit-Tests, Integrationstests, etc. Wichtig: Der Test muss fehlschlagen, da die Funktionalität noch nicht implementiert ist („red"). So wird sichergegangen, dass auch wirklich neue Funktionalität getestet und implementiert wird. Oft lassen sich die Tests noch nicht einmal kompilieren, da beispielsweise die zu testende Methode noch gar nicht existiert.
- **2. Schritt: „Green"**
  Bei der anschließenden Implementierung wird nun darauf geachtet, dass der Test so schnell wie möglich erfolgreich durchläuft („green"). Es kommt hier nur auf die Funktionalität an, die Implementierung muss nicht „schön/sauber" sein („quick and dirty"), sprich „so wenig Code wie möglich".
- **3. Schritt: „Refactor"**
  Sind alle Tests erfolgreich, wird „aufgeräumt". Der Code wird optimiert, Duplikate werden entfernt und die Lesbarkeit wird verbessert. Dabei werden die Tests immer wieder ausgeführt, um sicherzustellen, dass die Funktionalität bestehen bleibt.

(vgl. [1] S. x, [4] S. 153, [2])

Vorteile

- **Hochwertige Software**
  TDD ermölicht eine effektive und effiziente Erstellung der Software, indem unnötiger Code vermieden wird. Es wird immer nur der Code geschrieben, der für die Erfüllung der Anforderungen benötigt wird.
- **Geringerer Wartungsaufwand**
  Durch die Vermeidung von Fehlern durch Tests wird der Wartungsaufwand reduziert. Dadurch können Entwickler:innen mehr Zeit in die Entwicklung neuer Features investieren.
- **Frühe Fehlererkennung**
  TDD ermöglicht eine frühzeitige Erkennung von Fehlern, wodurch die Kosten für spätere Korrekturen minimiert werden. Ebenfalls wird dadurch nicht getesteter Code vermieden und dadurch die korrekte Funktionalität der gesamten Software von Beginn an sichergestellt.

Nachteile

- **Hoher initialer Zeitaufwand**
  TDD kann zeitaufwändig sein, da Tests in kleinen Schritten entwickelt werden müssen. Ebenfalls wird der initiale Zeitaufwand durch das Schreiben von Tests bevor die eigentliche Implementierung erfolgt ist, erhöht.
- **Erfordert Umdenken**
  TDD erfordert ein Umdenken in der Art und Weise, wie Software entwickelt wird, was für einige Entwickler:innen eine Herausforderung darstellen kann.
- **Nicht für alle Projekte geeignet**
  TDD ist nicht für alle Projekte geeignet, insbesondere für kleinere Projekte. Für kleinere Projekte gibt es Ansätze, die hierfür besser geeignet sind.
- **Gefahr für Overengineering**
  TDD kann zu Overengineering führen, wenn Entwickler:innen zu viele oder zu komplexe Tests schreiben, damit alle Anforderung abgedeckt weren können.

Vereinbarkeit mit den agilen Prinzipien

TDD lässt sich im agilen Umfeld ähnlich zu klassischeren Entwicklungsprozessen, bei denen erst nach der Implementierung getestet wird, umsetzen. Dabei können die agilen Prinzipien größtenteils erfüllt werden.
Aufgrund der Nachteile von TDD haben Entwicklungsteams allerdings oft Schwierigkeiten TDD anzunehmen und effektiv zu betreiben [5]. Das liegt daran, dass Entwickler nicht genau wissen mit welchem Test sie starten und welchen sie als nächstes schreiben sollen [5].
Aufgrund des höheren Zeitaufwands lässt sich argumentieren, dass Prinzipien 1 und 3 des agilen Manifests deshalb nicht komplett erfüllt werden können.
Allerdings ist es möglich hochwertige Software mit geringerem Wartungsaufwand zu entwickeln. Das agile Prinzip *„Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität."* kann somit erfüllt werden und klassische Entwicklungsprozesse möglicherweise sogar übertroffen werden.
Die Vereinbarkeit von TDD mit den agilen Prinzipien hängt also stark davon ab, wie effektiv es im Entwicklungsprozess praktiziert wird.
Um die Nachteile zu umgehen und TDD einfacher zu lernen und praktizieren, wurde deshalb basierend auf TDD das Entwicklungsverfahren Behavior-Driven Development (BDD) von Dan North erfunden [5].

Best Practices

Kleine unabhänige Tests

Beim Erstellen von Tests sollte darauf geachtet werden, dass sie klein und unabhängig sind. Das bedeutet, dass ein Test nur eine spezifische Funktionalität testet und nicht von anderen Tests abhängig ist. Dadurch wird die Wartbarkeit der Tests erhöht und die Ursache von Fehlern ist leichter zu finden.

So wenig Code wie nötig

Es sollte nur so viel Code geschrieben werden, wie erforderlich ist, um den Test zu bestehen. Dabei sollte vermieden werden, Code zu implementieren der nicht direkt mit dem aktuellen Testfall zusammenhängt. Dies trägt dazu bei, die Codebasis schlanker und besser wartbar zu machen, da sich jeder geschriebene Code auf spezifische Anforderungen und Funktionalitäten konzentriert.

Positve und Negative Tests

Beim Schreiben von Tests ist es wichtig, sowohl positive als auch negative Szenarien zu testen. Das bedeutet, dass sowohl Fälle getestet werden sollten, in denen der Code korrekt funktionieren soll, als auch solche, in denen er scheitern soll. So können mögliche Probleme mit dem Code erkannt und sichergestellt werden, dass er robust und zuverlässig ist.

Testen Testen Testen

Tests sollten immer wieder ausgeführt werden. So kann sichergestellt werden, dass der Code auch nach Änderungen noch korrekt funktioniert. Darüber hinaus ermöglicht es, Fehler frühzeitig zu erkennen und eine zuverlässige Codebasis aufrechtzuerhalten.

Codebeispiel

Anforderung (Schritt 0)

Es ist eine Funktion benötigt, die die Zahlen von 1-3 quadrieren kann. Die Funktion soll als Input eine Zahl entgegennehmen und als Output die quadrierte Zahl zurückgeben. Die Funktion soll den Namen „square" haben.

Testfälle (Schritt 1 „Red")

```
assert(square(1) == 1) # RED
assert(square(2) == 4) # RED
assert(square(3) == 9) # RED
```

Implementierung (Schritt 2 „Green")

```
def square(x):
    if number == 1:
        return 1
    elif number == 2:
        return 4
    elif number == 3:
        return 9

assert(square(1) == 1) # GREEN
assert(square(2) == 4) # GREEN
assert(square(3) == 9) # GREEN
```

Refactoring (Schritt 3 „Refactor")

```
def square(x):
    return x * x

assert(square(1) == 1) # GREEN
assert(square(2) == 4) # GREEN
assert(square(3) == 9) # GREEN
```

Fazit

TDD ist ein Ansatz in der Softwareentwicklung, der mit seinen spezifischen Vorteilen wie Fehlerreduzierung und Verbesserung der Codequalität überzeugt, allerdings auch Herausforderungen wie höheren Zeitaufwand und Eignungsbegrenzungen mit sich bringt. Es ist eine Methode unter vielen, die in bestimmten Kontexten Vorteile bietet, aber nicht universell die optimale Lösung darstellt. Die Entscheidung für oder gegen TDD sollte daher projekt- und teamspezifisch getroffen werden.

Literatur

[1] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional. Google-Books-ID: zNnPEAAAQBAJ.

[2] IONOS. Test driven development: So funktioniert die methode.

[3] Robert C. Martin. Professionalism and test-driven development. *IEEE Software*, 24(3):32–36, May 2007.

[4] Alexander Schatten, Markus Demolsky, Dietmar Winkler, Stefan Biffl, Erik Gostischa-Franta, and Thomas Östreicher. Qualitätssicherung und test-driven development. In Alexander Schatten, Markus Demolsky, Dietmar Winkler, Stefan Biffl, Erik Gostischa-Franta, and Thomas Östreicher, editors, *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*, pages 113–162. Spektrum Akademischer Verlag.

[5] John Ferguson Smart and Jan Molak. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster, 2023.