TDD / BDD

Konzepte, Vorgehen und Werkzeuge

TDD und BDD

- Methoden / Arbeitsweisen die bei der Umsetzung von Anforderungen in Code unterstützen
- es entstehen Tests und damit selbst testender Code
- die Tests dokumentieren die Fachlichkeit
- TDD hat Wurzeln in Xtreme Programming

- durch Tests wird Code überhaupt erst änderbar (refactor) (!)

TDD

Definition TDD

“Test-Driven Development (TDD) is a technique for building software that guides software development by writing tests. It was developed by Kent Beck in the late 1990's as part of Extreme Programming. In essence you follow three simple steps repeatedly:

- Write a test for the next bit of functionality you want to add.
- Write the functional code until the test passes.
- Refactor both new and old code to make it well structured.”

Autor: Martin Fowler (http://martinfowler.com/bliki/TestDrivenDevelopment.html)

Vorteile / Motivation von TDD

Test für Test wird die Funktionalität des Systems aufgebaut.

- es entsteht “SelfTestingCode”, funktionaler Code wird als Antwort auf einen roten Test geschrieben
- durch den Test ist der Entwickler gezwungen zuerst über die Schnittstelle (Interface) des Codes nachzudenken
- Konzentration auf Schnittstelle und Verwendung der Klasse hilft die Schnittstelle besser von der Implementierung zu separieren

Quelle: Martin Fowler - http://martinfowler.com/bliki/TestDrivenDevelopment.html

Was zerstört TDD (Anti-Pattern)

- Der sicherste weg TDD zu zerstören ist den Überarbeitungsschritt (Refactoring) wegzulassen.
- Refactoring des Codes um ihn sauber zu halten ist der Schlüssel des Prozesses.

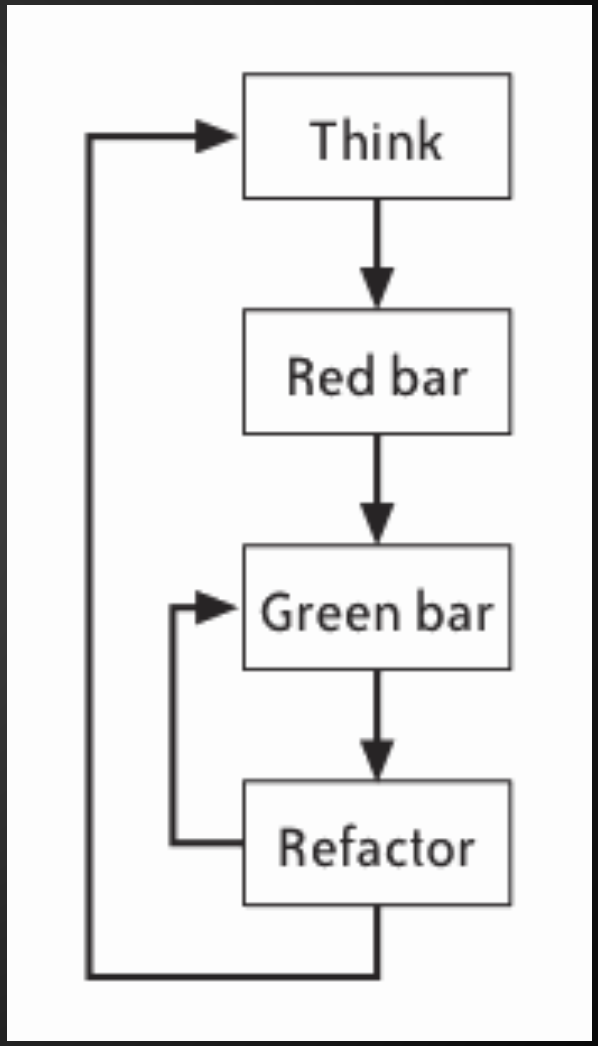
Alternative:

- Entwickler endet mit einer wilden Sammlung von Code-Fragmenten (aber immerhin mit Tests :-))

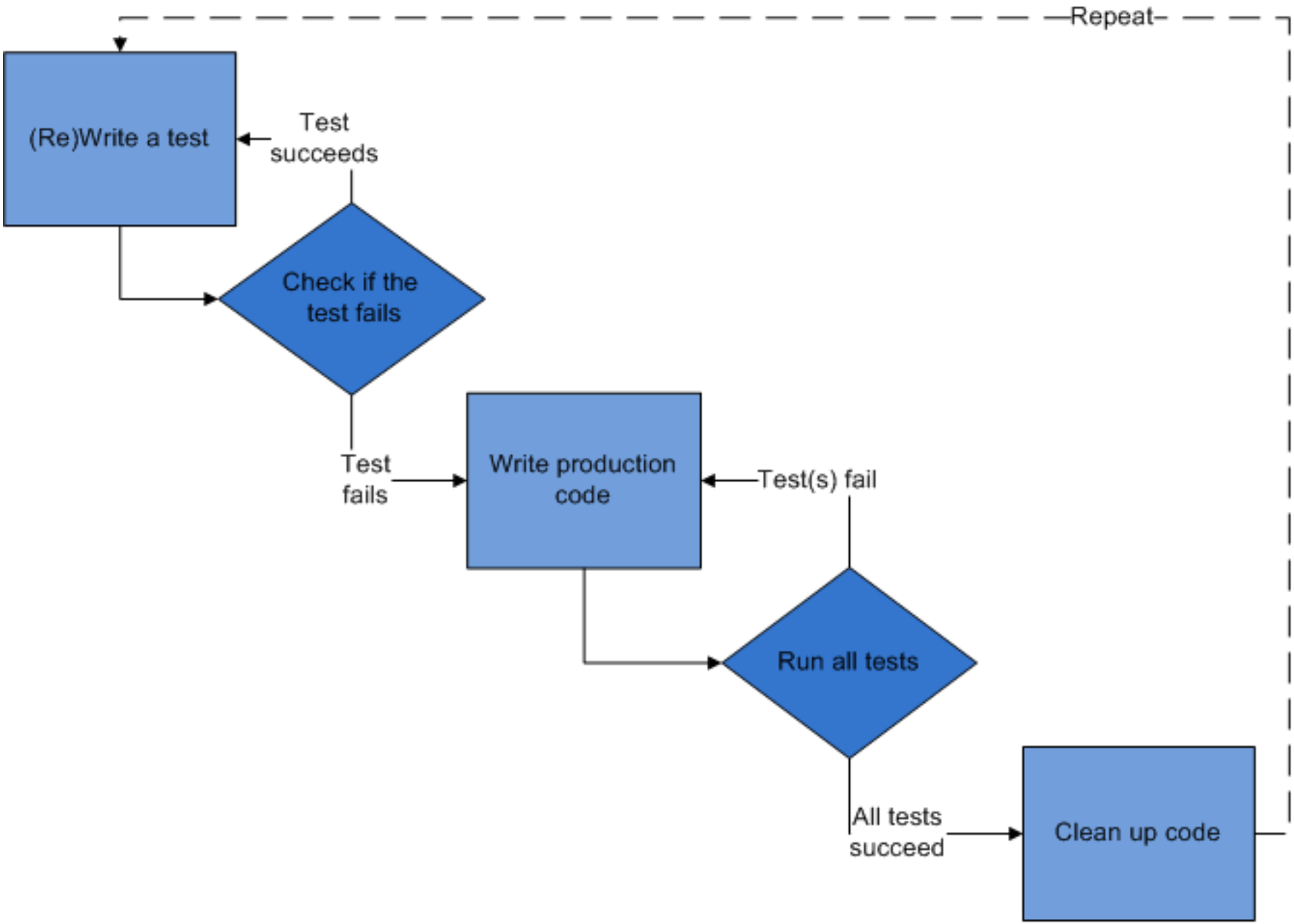
Quelle: Martin Fowler - http://martinfowler.com/bliki/TestDrivenDevelopment.html

Zyklus / Arbeitsweise

- Denken
- Test (rot)
- (Implementieren)
- Test (grün)
- Refactor



Quelle: James Shore - http://www.jamesshore.com/Agile-Book/test_driven_development.html


Repeat
(Re)Write a test
Test succeeds
Check if the test fails
Test fails
Write production code
Test(s) fail
Run all tests
All tests succeed
Clean up code

Ziel von TDD

- Bugs verhindern
- implementieren was tatsächlich gebraucht wird
- sich selbst testender Code
- testbarer Code
- bessere Schnittstellen und Design des Codes

BDD

Problemstellung

- Dokumentation bereits entwickelter und noch fehlender Funktionalität
- Beschreibung von Abnahmetests durch oder für die Fachabteilung
- Übersicht über bereits umgesetzte Funktionalität

Geschichte

- Behavior Driven Development wurde erstmals 2003 durch Dan North als Erweiterung von TDD entwickelt
- Dan North schrieb auch das erste Framework für die Umsetzung von BDD: JBehave (http://jbehave.org)

Quelle: http://de.wikipedia.org/wiki/Behavior_Driven_Development

Behavior Driven Development - BDD

- Textuelle Beschreibung des Verhaltens der Software durch Fallbeispiele
- Verwendung genormter Schlüsselwörter zur Markierung von Vorbedingungen und des gewünschten Verhaltens
- Automatisierung der Fallbeispiele unter Verwendung von Mock-Objekten zur Simulation von noch nicht implementierten Softwareteilen.
- Sukzessive Implementierung der Softwareteile und Ersetzung der Mock-Objekte
- es entsteht eine automatisiert prüfbare Beschreibung der umzusetzenden Software, welche jederzeit die Korrektheit der bereits umgesetzten Teile der Software überprüfen lässt

Vereinfachte Definition

- BDD = Behaviour Driven Development
- BDD versucht vom technischen Testcode zu abstrahieren und führt eine natürlichsprachliche Ebene ein, die der Fachabteilung das Schreiben von Tests ermöglichen soll.
- TDD = erst Test dann Funktionalität
- BDD = erst Verhalten (Behavior) dann Funktionalität

Werkzeuge

- Spock
- easyb
- Cucumber
- JBehave
- RSpec

User Stories vs. Specifications

- User Story - wird agilen Methoden wie Scrum verwendent um Anforderungen zu definieren.
- Specification - ehr technisch und beschreibt das Verhalten der Komponenten

Beispiel (User Story)

Game of Life (in JBehave)

…

```
Given a 5 by 5 game
When I toggle the cell at (2, 3)
Then the grid should look like
```

…

Zeilen im Textdokument so wie sie JBehave benötigt.

Quelle: http://en.wikipedia.org/wiki/Behavior-driven_development

Beispiele (Code)

```
private Game game;
private StringRenderer renderer;

@Given("a $width by $height game")
public void theGameIsRunning(int width, int height) {
    game = new Game(width, height);
    renderer = new StringRenderer();
    game.setObserver(renderer);
}

@When("I toggle the cell at ($column, $row)")
public void iToggleTheCellAt(int column, int row) {
    game.toggleCellAt(column, row);
}
```

BDD auf allen Testebenen

- Vorgehen nach Verhaltensspezifikation kann auf allen Testebenen angewendet werden
- Unit-Test
  - z. B. Zinsberechnungsalgorithmus gekapselt in einer Klasse
- Komponenten-Test
  - z. B. Verhalten einer Eingabemaske im Web
- etc.

Integration in den CI-Prozess

- Frameworks lassen sich meist mit JUnit-Testrunner ausführen, dadurch leichte Integration in Erstellungsprozess (z. B. Maven oder Gradle)
- Bis auf Testerstellung keine weiteren Besonderheiten gegenüber “normalen” automatisierten Tests

Vorteile

- Wenn die Fachseite willens ist die Spezifikation im BDD-Stil abzubilden, dann kann sie sehr gut ins Team und den Entwicklungsprozess integriert werden.
- Hauptvorteil: Nicht technische Dokumentation / Reports des Funktionsumfangs der Software
- Möglichkeit die Abnahmetests für automatisierte Ausführung abzulegen

Nachteile

- aufwendiger Test zu schreiben, meist Pflege von 2 Dateien (Text und Code)
- schwieriger zu debuggen, refactorern und zu warten
- weniger IDE-Unterstützung
- langsamerer Texstausführung (parsing der Texte und Mappen auf Code)
- skalliert schlecht bei wachsenen Testsuites, Frameworks nutzen globalen Zustand zur Kommunikation -> serielle Abarbeitun
- Entwickler gewohnt Code zu lesen, Lesen von reinem Text ist langsamer, Text nicht so komprimiert wie Code
- Fachabteilung schreibt oft nicht die Tests

Quelle: http://jamescrisp.org/2011/05/30/automated-testing-and-the-test-pyramid

BDD bei der E-Post

Name: Living Documentation

Tools: Verwendung von Spock-Framework und Confluence (Wiki)

Ziel: Generieren ein für die Fachseite durchstöber- und lesebare Dokumentation die den aktuellen Funktionsumfang der Software wiederspiegelt und automatisch mit Testausführung aktualisiert wird.

Beispiel

```
15  @JiraId("IAR-43, IAR-44, IAR-107")
16  @ConfluencePath("Identitäten/Co-Registrierung/Reg-Service-Features")
17  @HeaderName("Userdatenabfrage an der Registrierungs-Schnittstelle")
18  @Narrative(role = "Als Betreiber des Epostbriefes", action = " möche ich sicherstellen, da
19  expect = " und dass damit alles funktionieren kann.")
20  class AccountResourceCT extends AbstractComponentCT {
21
79      void "Return 204 NO_CONTENT falls Passwort gültig und verify gesetzt ist"() {
80          given: "Registrierungsdaten mit Passwort"
81          accountRegistrationData()
82          when: "Setzen der Registrierungsdaten mit neuem Passwort mit der Option Verify"
83          putWithVerifyIsCalledOnResourceWith epbAddress, registrationData
84          then: "Die Operation war nicht erfolgreich, Status NO_CONTENT"
85          statusIs 204
86      }
87
```

Confluence

Regel: Return 204 NO_CONTENT falls Passwort gültig und verify gesetzt ist

GEGEBEN Registrierungsdaten mit Passwort

WENN Setzen der Registrierungsdaten mit neuem Passwort mit der Option Verify

DANN Die Operation war nicht erfolgreich, Status NO_CONTENT

Literatur / Links

Wikipedia:

- http://en.wikipedia.org/wiki/Test-driven_development

Bücher:

Kent Beck

- http://www.amazon.de/Driven-Development-Example-Addison-Wesley-Signature/dp/0321146530
- als Google Book: http://books.google.de/books?id=gFgnde_vwMAC&lpg=PP1&pg=PA7&redir_esc=y#v=onepage&q&f=false

Steve Freeman

- http://www.amazon.de/Growing-Object-Oriented-Software-Guided-Signature/dp/0321503627/ref=sr_1_cc_1?s=aps&ie=UTF8&qid=1381480216&sr=1-1-catcorr&keywords=Growing-Object-Oriented-Software-Guided

Web:

- http://cumulative-hypotheses.org/2011/08/30/tdd-as-if-you-meant-it
- http://gojko.net/2009/02/27/thought-provoking-tdd-exercise-at-the-software-craftsmanship-conference