



OPITZ CONSULTING

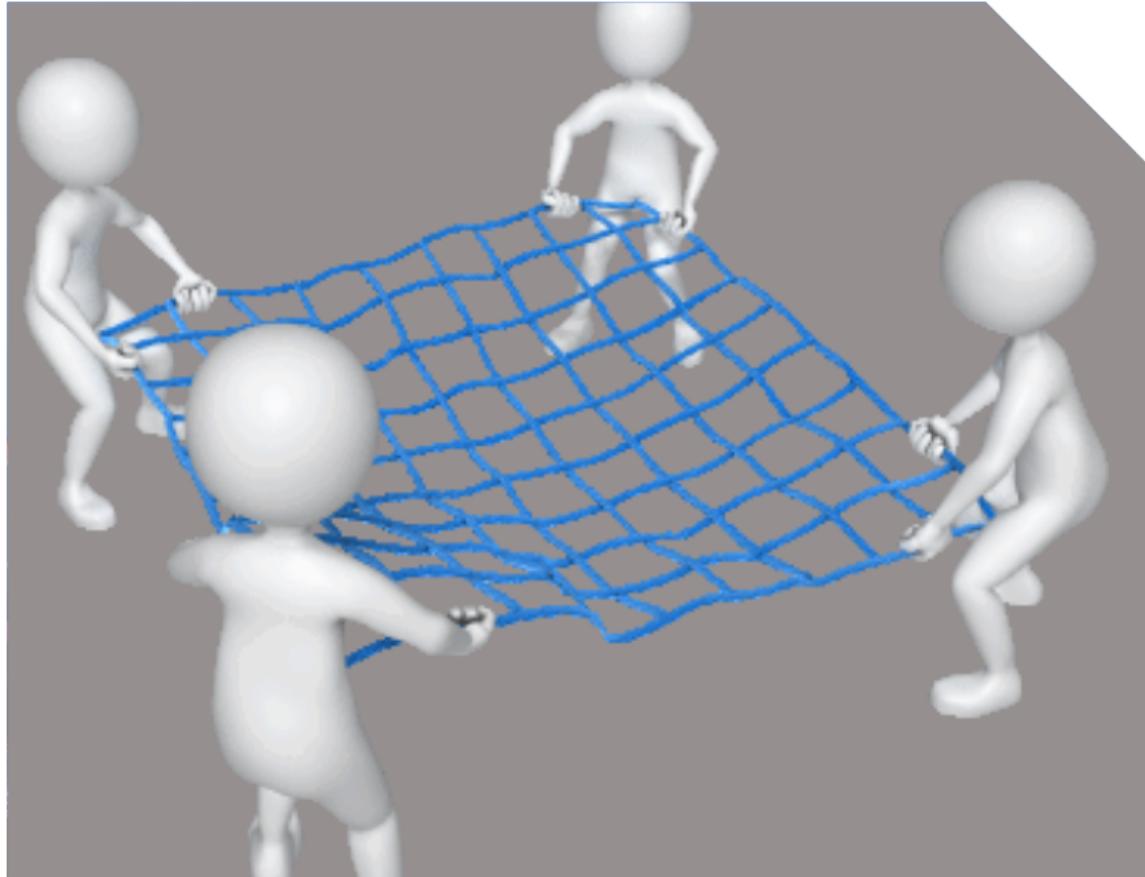
TDD Schulung

Thomas Papendieck

Senior-Consultant

17. 03. 2022, Phantasialand Brühl

JavaLand



INHALT

01

GRUNDLAGEN

02

ANFORDERUNGEN AN
UNITTESTS

03

ANFORDERUNGEN AN
ZU TESTENDEN CODE

04

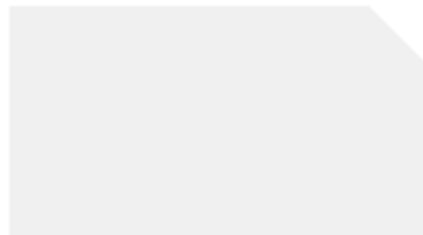
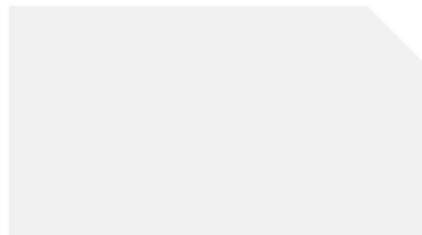
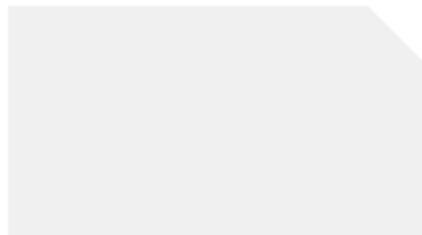
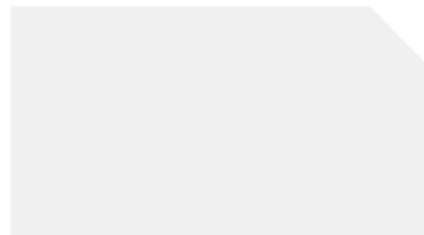
UNITTEST IN JAVA

05

MIND CHANGES

06

HANDS ON



01

GRUNDLAGEN

- Testen im Softwareentwicklungsprozess
- Probleme manueller Tests
- Welche Tests sollten automatisiert werden?
- Testarten
- Hindernisse beim Automatisierten Testen
- Was ist TDD

Relative Kosten von Fehlern

Wie teuer ist ein Fehler in Abhängigkeit vom Zeitpunkt seiner Entdeckung:

während der Entwicklung	1
Wenn der Entwickler das fertige Feature testet	3
Während der Integration	10
Beim Abnahmetest	100
in der Produktion	1000

Nachteile manueller Tests.

Software muss in einem lauffähigen Zustand sein.

Tester benötigt wissen über die Anwendung.

Was ist das gewünschte Verhalten?

Was ist ein Fehler?

Wie testen man Fehlerzustände?

manuelle Tests sind langsam.

finden nur zu regulären Arbeitszeiten statt.

Testen ist langweilig.

- Application Test
 - rejection check formally known as *acceptance test*.
 - performance/stress test
- Module Test
- Unit Test

Unterschied Application/Module-Tests zu UnitTests

Applications und Module Test

- werden spät im Entwicklungsprozess ausgeführt
- Testwerkzeuge sind komplex
- sind aufwendig zu warten
- zeigen, dass ein Fehler existiert, aber nicht wo

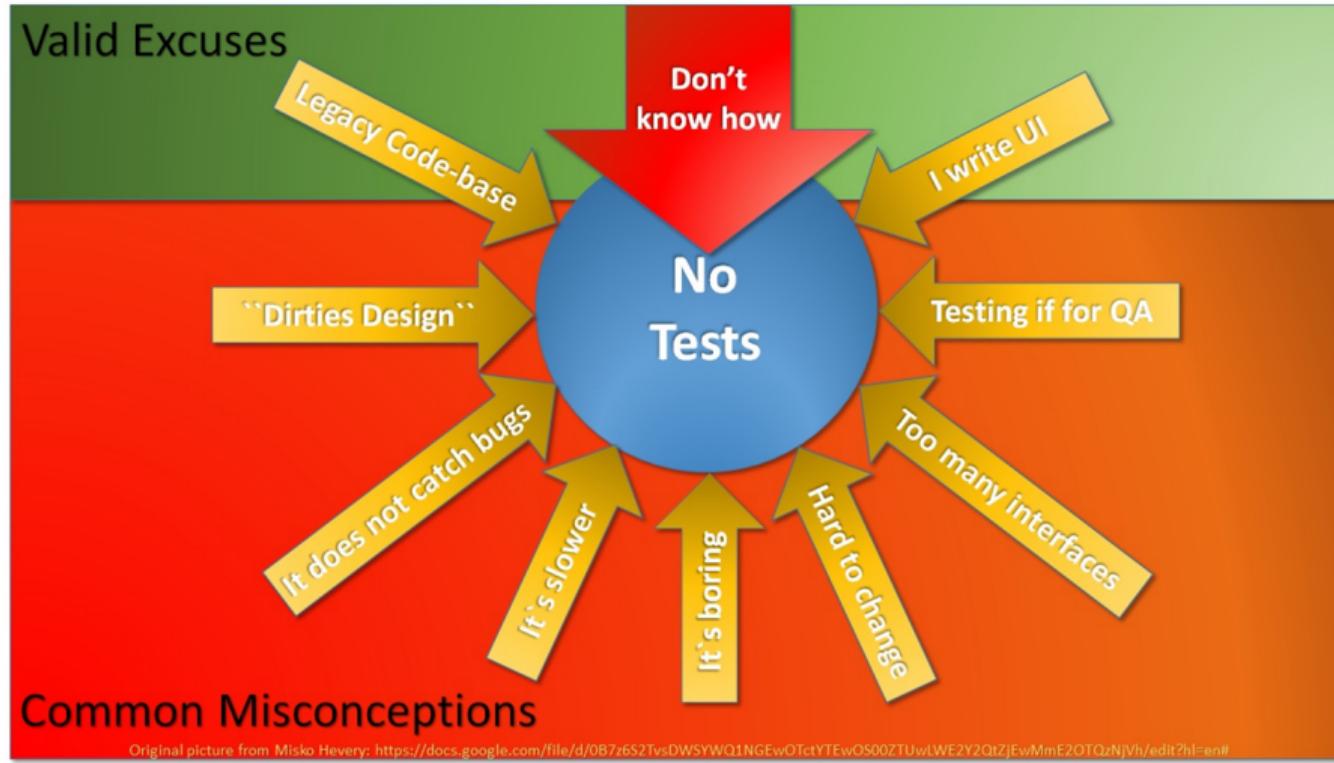
UnitTest

- laufen früh im Entwicklungsprozess (idealer Weise nach jedem Speichern)
- Werkzeuge haben einfache API
- sind stabil gegen Änderungen (anderer Units)
- zeigen welche Anforderung nicht erfüllt wird, wo der Fehler existiert und unter welchen Bedingungen er auftritt.

hoher UnitTest Abdeckung → weniger Test höherer Ebenen

UnitTests prüfen die Geschäftslogik, Application/Module-Tests die "Verdrahtung"

Warum schreiben wir keine automatisierten Tests?



Original picture from Misko Hevery: <https://docs.google.com/file/d/0B7z6S2TvsDWSYQ1NGEwOTctYTEwOS00ZTUwlWE2Y2QtZjEwMmE2OTQzNjVh/edit?hl=en#>

persönliche, technische und soziale Voraussetzungen

- UnitTests schreiben ist eine Fertigkeit und muss ständig geübt werden.
- Technische Voraussetzungen müssen sichergestellt sein.
- Team und Vorgesetzte müssen automatisiertes Testen unterstützen.

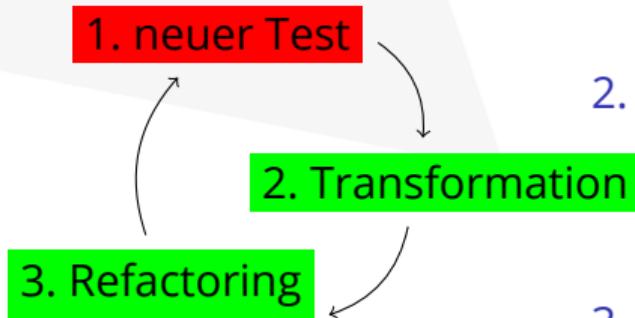
TDD = Testen?

- TDD ist Arbeitsmethode bei der Implementierung
 - fokussiert auf definierte Anforderungen
 - schützt Codebasis gegen ungewollte Änderungen (des Verhaltens).
- Testen prüft die Betriebstauglichkeit des Programms
 - ist immer manuell.
 - Suche nach Umständen, in denen das Programm Fehlverhalten zeigen könnte.
- Verifikation prüft bekanntes (Fehl-) Verhalten.
 - anhand vorhandener Testpläne (die beim Testen entstehen)
 - oft manuell
 - idR. gut automatisierbar

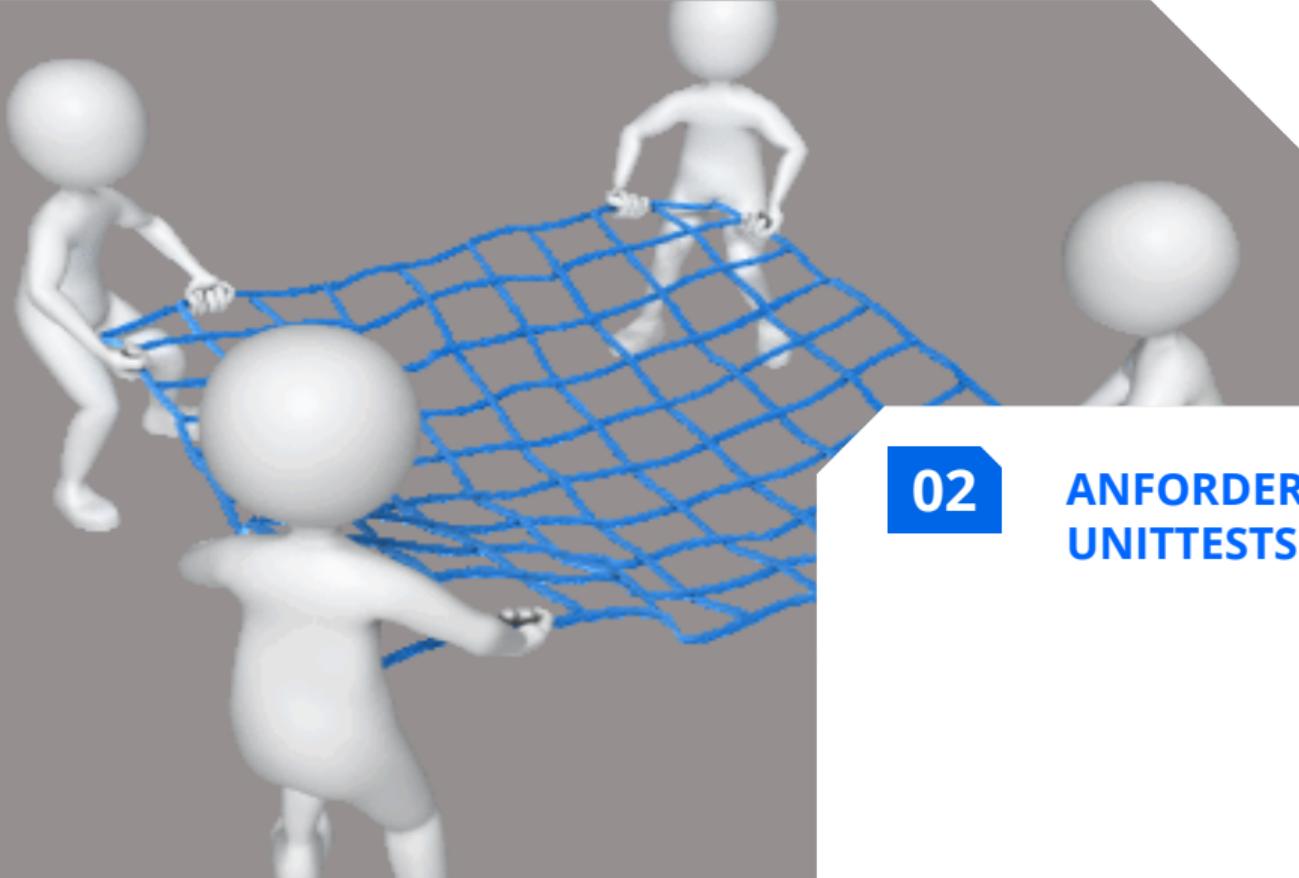
Vorgehensweisen beim Schreiben von Unitests:

- Code first
 - macht den Test grün (statt ist das so richtig?)
 - unter Zeitdruck keine Tests
- Test first
 - Testschreiber != Implementierer
- Test Driven Developement
 - Test und Produktivcode entstehen **gleichzeitig** in Micro-Cycles

Was ist Test Driven Development?



1. Schreibe einen neuen Test, gerade so viel dass er fehlschlägt (nicht kompilieren ist fehlschlagen).
2. Schreibe gerade so viel Produktivcode, dass der Test erfüllt wird. Zukünftige Anforderungen **nicht** beachten! (so simpel wie möglich, alles ist erlaubt, Transformations-Prämissen beachten).
3. Verbessere den Code (Produktion und Test), ohne einen Test zu berühren und ohne neue Funktionalität (Geschäftslogik) hinzuzufügen.



02

ANFORDERUNGEN UNITTESTS

AN

Was macht ein Unitest?

- Unitests sind ausführbare Dokumentation.
- Unitest testen **keinen** Code.
- Unitest verifizieren **von außen beobachtbares gewünschtes Verhalten** von Code.

Sie prüfen *Rückgabewerte* und *Kommunikation mit anderen Units* des zu testenden Codes als Reaktion auf die übergebenen Parameter und/oder den Reaktionen der anderen Units.

- Ein einzelner Test prüft genau **eine Erwartung** an die Unit.
- Unitests **verhindern ungewollte Änderungen**.

Wie schreibt man einen guten UnitTest?

Fast
Independent
Repeatable
Selfevaluating
Timely

Readable
Trustworthy
Fast
Maintainable

Kann nach jedem Speichern ausgeführt werden ohne den Arbeitsablauf zu verzögern.

- primitive Vorbereitung (setup)
- Abhängigkeiten durch Test-Doubles ersetzen

Independent - unabhängig

Jeder Test kann einzeln ausgeführt werden.

- kein Test schafft Voraussetzungen für nachfolgende
- Test können in beliebiger Reihenfolge laufen

Repeatable - wiederholbar

Jede Ausführung (ohne Änderung des getesteten Codes) führt zum selben Ergebnis.

- nicht von zufälligen Größen abhängig
- kein Einfluß durch Änderungen an anderen Units
- kein Einfluß durch Testumgebung (Netzwerk, Festplatte, Position in der Verzeichnisstruktur)

Selfverifying - selbstauswertend

Das Testergebnis ist eindeutig und binär.

- Erfolg oder Mißerfolg sind eindeutig erkennbar
- Ergebnis muss nicht in Logdateien o.ä. gesucht werden
- Ergebnis kann automatisiert weiter verarbeitet werden

Unitests entstehen zeitnah zum getesteten Code.

- **Code first**

Tests werden nach dem zu testenden Code geschrieben.

- **Test first**

Tests werden vor dem zu testenden Code geschrieben.

- **Test Driven Developement**

Test und verifizierter Code entstehen gleichzeitig.

Was bedeutet *lesbar*?

- was wird getestet
 - Name des Tests drückt Vorbedingungen und erwartetes Ergebnis aus.
- kurz
 - wiederkehrende Vorbereitungen in Methoden auslagern
- Welche Eigenschaften der Parameter sind wichtig?
 - Explizit Variablen für Parameterwerte anlegen
 - Variablennamen sorgsam wählen
- Welche Eigenschaften der Ergebnisse sind wichtig?
 - Explizit Variablen für Ergebnisse anlegen
 - Variablennamen sorgsam wählen
 - Verifizierungsmethoden mit sprechenden Namen.

Beispiel Lesbarkeit

Finished after 0,027 seconds

Runs: 2/2 Errors: 0 Failures: 1

BadNamedBowlingCalculatorTest [Runner: JUnit 4] (0,007 s)

- test1 (0,006 s)
- test2 (0,000 s)

Runs: 2/2 Errors: 0 Failures: 1

GoodNamedBowlingCalculatorTest [Runner: JUnit 4] (0,000 s)

- calculate_someIncompleteFrames_sumUpIndividualRolls (0,000 s)
- calculate_worstGame_returnsZero (0,000 s)

Failure Trace

java.lang.AssertionError: 0,2,3,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0 expected:<7> but was:<0>

at GoodNamedBowlingCalculatorTest.calculate_someIncompleteFrames_sumUpIndividualR

Was bedeutet *vertrauenswürdig*?

- Geschäftsanforderungen erfüllt

Wurde tatsächlich implementiert, was der Kunde wollte?

- technisch

Wird der Produktivcode tatsächlich ausgeführt?

- "test first"

Schlägt der Test aus dem richtigen Grund fehl?

- Ersetzen von Abhängigkeiten (Mocking)
 - (weitestgehend) keine Logik in Tests

Wiederholung Folie 17

Was bedeutet wartbar?

- Stabilität gegenüber Änderungen in anderen Units
Abhängigkeiten ersetzen
- stabil gegen Änderungen in der Unit selbst
eine einzelne Anforderung (nicht Codestelle) testen.

03

ANFORDERUNGEN AN TESTENDEN CODE

AN ZU

- Was verbessert die Testbarkeit?
- Isolieren einer Unit
- Isolation ermöglichen

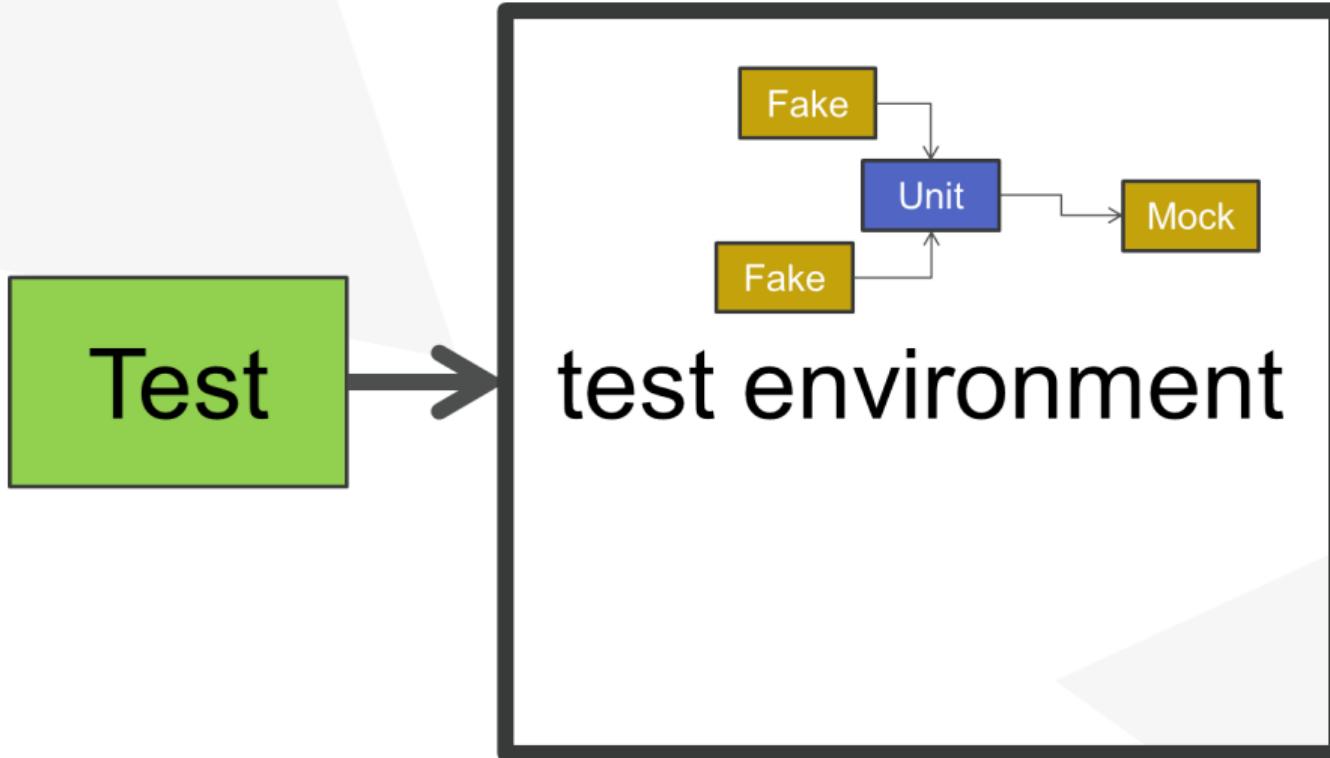
Testbarkeit von produktivem Code

Qualität des produktivem Code (im Sinne von "Clean Code" und dem "S.O.L.I.D." Prinzip) beeinflußt die Qualität der Tests (im Sinne der RTFM-Anforderungen)

die wichtigsten "Clean Code" Regeln sind:

- Separation of concerns / single responsibility
Erzeugung von Objekten der Abhängigkeiten ist **keine** Aufgabe der Geschäftslogik!
- Dependency Inversion
- Tell! Don't ask.
- Law of Demeter (Don't talk to strangers)
vermeide "message chaining" (nicht mit "fluent API" verwechseln)

Testbarkeit von produktivem Code



Arten von Test-Doubles

- Stub
 - leere Implementierung einer Schnittstelle, üblicher Weise generiert
- Fake
 - alternative Implementierung einer Schnittstelle oder Erweiterung einer existierenden Implementierung.
 - extrem vereinfachtes Verhalten (keine Logik)
- Mock
 - "aufgemotztes" Fake
 - konfigurierbares Verhalten
 - Verifizierung vom Methodenaufrufen und der übergebenen Parameter

Was Ermöglicht die Ersetzung von Abhängigkeiten?

- trenne Instanziierung der Abhängigkeiten von der Geschäftslogik
- vermeide den Aufruf des `new` Operators
- Bereitstellen von "seams" (Nahtstellen)
 - dependency injection
 - Getter mit geringer Sichtbarkeit
- keine `static` (public) Methoden
- keine `final` (public) Methoden
- vermeide *Singelton Pattern* (nicht *Singelton* als Konzept)

schreibe "Clean Code"¹

- programmiere gegen Schnittstellen
- SoC/SRP (Feature envy)
- Law of Demeter
- DRY
- same level of abstraction

¹"Clean Code" by Robert C Martin, ISBN-13: 978-0132350884

04

UNITTEST IN JAVA

- Werkzeuge
- Code Vorlagen
- Transformationsprämissen

- IDE including testplugin (eclipse + infinitest / Netbeans + ? /...)
- build – tool (maven / gradle / ant / ...)
- SCM (git / subversion / ...)
- xUnit testing framework (JUnit / NUnit /...)
- mocking framwork (Mockito / ...)

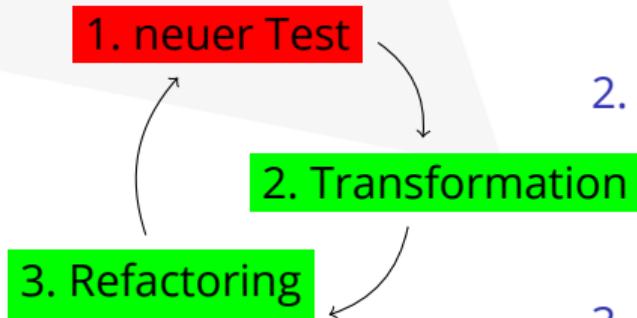
einen JUnit Test schreiben

```
1 class PlainOldJavaClass{  
2  
3     @Test // marks a method so that its been called by JUnit  
4     public // test methods must be public  
5     void // test methods must not return anything  
6     calculate_worstGame_returnsZero() // no parameters  
7     {  
8         // arrange: create and configure dependencies and variables  
9         String playerResultAllZero = "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0";  
10        int expectedResult = 0;  
11  
12        // act: create tested unit (if possible) and call tested method.  
13        int calculatedResult = new BowlingCalculator().calculate(playerResultAllZero);  
14  
15        // assert: check the Result by calling one of JUnits assert methods.  
16        // the assert method throws an exception when the check fails.  
17        assertEquals("allzeros", // give usefull short(!) additional description  
18                      // skip when in doubt  
19                      expectedResult,  
20                      calculatedResult);  
21    }  
22}
```

Mockito verwenden

```
1 class PlainOldJavaClass{  
2     @Test  
3     public void testedMethod__preconditions__expectedBehavior() {  
4         // create mock  
5         MyInterfaceOrClass mockObject = mock(MyInterfaceOrClass.class);  
6         //create a spy  
7         MyClass spyObject = spy(new MyClass());  
8         // configure behavior:  
9         doThrow(new RuntimeException("simulateError"))  
10            .when( mockObject ).anyMethod();  
11         doReturn(mockObject, null, mockObject)  
12            .thenThrow(new RuntimeException("simulateError after 4th call"))  
13            .when( spyObject ).methodWithValue();  
14  
15         // alternative for non void methods:  
16         .when( spyObject.methodWithValue() )  
17            .thenReturn(mockObject, null, mockObject)  
18            .thenThrow(new RuntimeException("simulateError after 4th call"));  
19  
20         // check call of method  
21         verify(spyObject).expectedMethodCall(mockObject, any(OtherInterfaceOrClass.class));  
22     }  
23 }
```

Test Driven Development



1. Schreibe einen neuen Test, gerade so viel dass er fehl schlägt
(nicht komplizieren ist fehlschlagen).
2. Schreibe gerade so viel Produktivcode, dass der Test erfüllt wird. Zukünftige Anforderungen **nicht** beachten! (so simpel wie möglich, alles ist erlaubt, Transformations-Prämissen beachten).
3. Verbessere den Code (Produktion und Test), ohne einen Test zu berchen und ohne neue Funktinalität (Geschäftslogik) hinzuzufügen.

Transformationsprämissen¹

In der Implementierungsphase des TDD-Zyklus ist im Fall, dass mehrere Transformationen möglich sind, diejenige anzuwenden, die in der folgenden Liste am weitesten oben steht:

1. **constant** a value
2. **scalar** a local binding, or variable
3. **invocation** calling a function/method
4. **conditional** if/switch/case/cond
5. **while loop** applies to **for** loops as well
6. **assignment** replacing the value of a variable

¹"Transformation Priority Premise Applied" by Micah Martin, <https://8thlight.com/blog/micah-martin/2012/11/17/transformation-priority-premise-applied.html>

05

MIND CHANGES

Test Driven Development erzeugt keine Tests sondern **ausführbare Spezifikationen**. Diese ausführbaren Spezifikationen sind Teil der Dokumentation des eigenen Codes für andere Entwickler.

UnitTests verifizieren **öffentlich beobachtbares Verhalten**, nicht Code.
Öffentlich beobachtbares Verhalten sind Rückgabewerte und Kommunikation mit
Abhängigkeiten (anderen *Units*).

Mind Change 3:

Code Coverage hat als konkrete Zahl keine Bedeutung. Die einzige "*Coverage*" Metrik von Bedeutung ist **100% Requirements Coverage**. Die kann man aber nicht messen, sondern nur per konsequentem TDD sicherstellen.

Mind Change 4:

Code Coverage hat als konkrete Zahl keine Bedeutung. Es gibt Code, der nur Infrastruktur also keine Geschäftslogik implementiert und *too simple to fail* ist, bzw. dessen Korrektheit durch Integrationstest verifiziert wird. Für solchen Code sollten keine Unitests geschrieben werden.

Das setzt voraus, dass das die Konzepte *Single Layer of Abstraction*(SLA) und *Single Responsibility Pattern*(SRP) konsequent auch auf Objekt/Klassenebene umgesetzt sind.

06

HANDS ON

Conways Game of Life

https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Demo

- Berechne den Folgestatus (TDD pure function).
- Verhalten einer Zelle (Mocks einsetzen).

Bowling Kata

<http://codingdojo.org/cgi-bin/index.pl?KataBowling>

Berechne den Wert eines Bowling-Spiels.



Thomas Papendieck

Senior-Consultant

Am Weidenring 56
61348 Bad Homburg

thomas.papendieck@opitz-consulting.com

+49 6172 66260 1523