

# TDD Schulung

## JavaLand / Phantasialand Brühl

---

# Inhalte

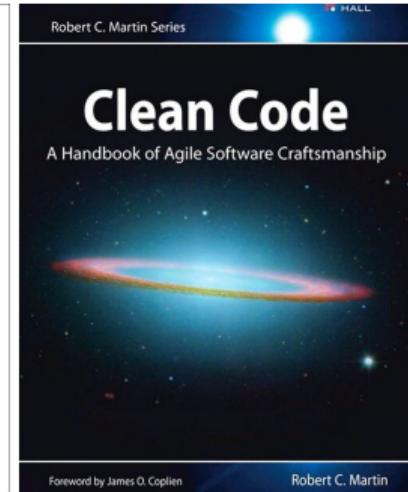
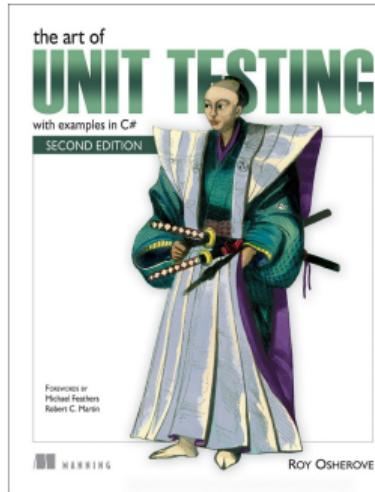
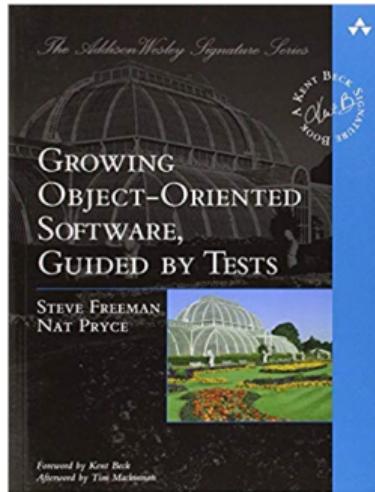
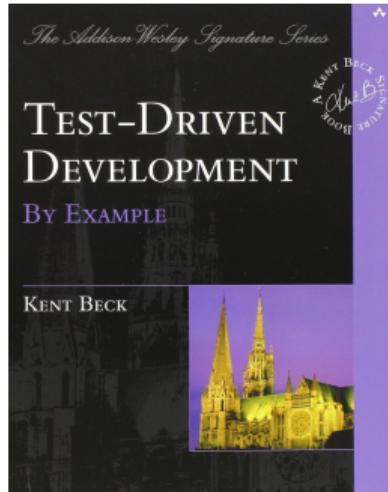
---



# Grundlagen



# Literaturempfehlungen



# 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.

# Test Level

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

Automatisierung von Tests auf allen Ebenen und verschiedener Testarten ist Voraussetzung für *Continuous Integration* bzw. *Continuous Delivery*.

# 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

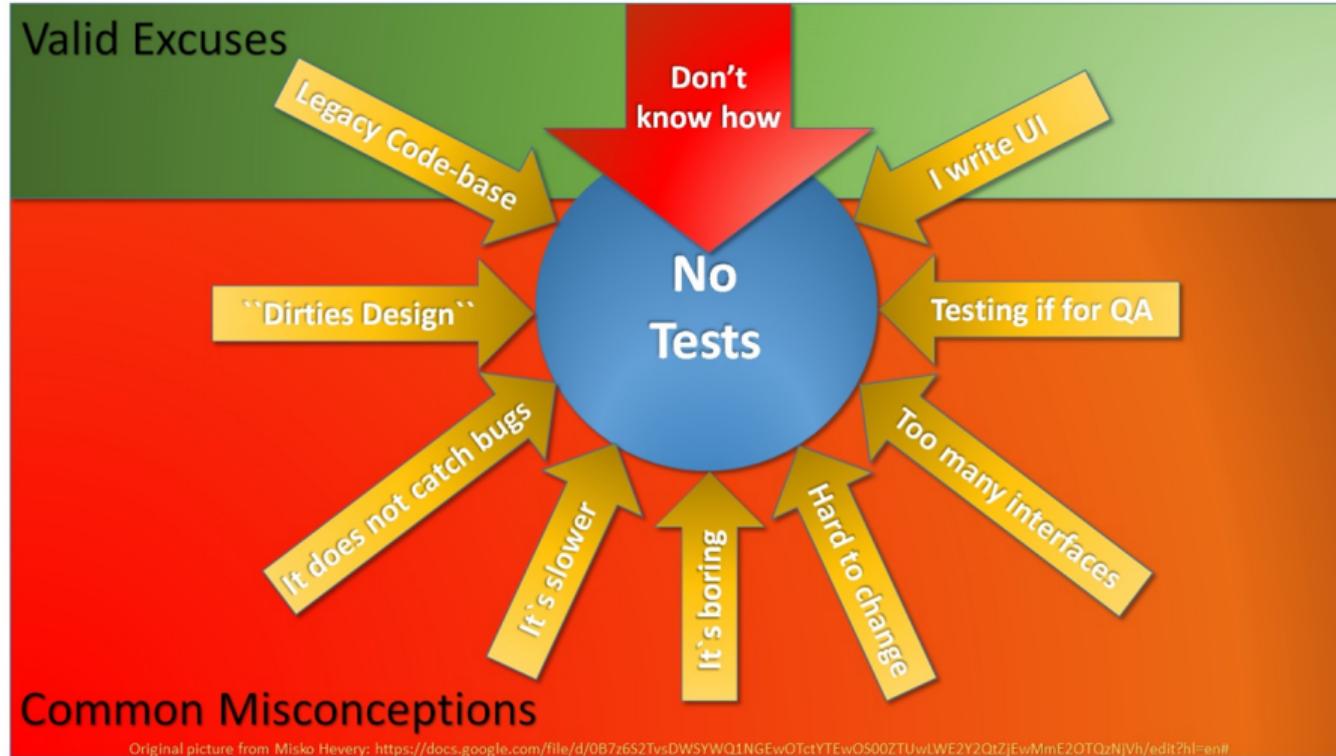
## UnitTest

- 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/0B7z6S2TvsDWSYWQ1NGEwOTctYTEwOS00ZTUwlWE2Y2QtZjEwMmE2OTQzNjVh/edit?hl=en)

## persönliche, technische und soziale Voraussetzungen

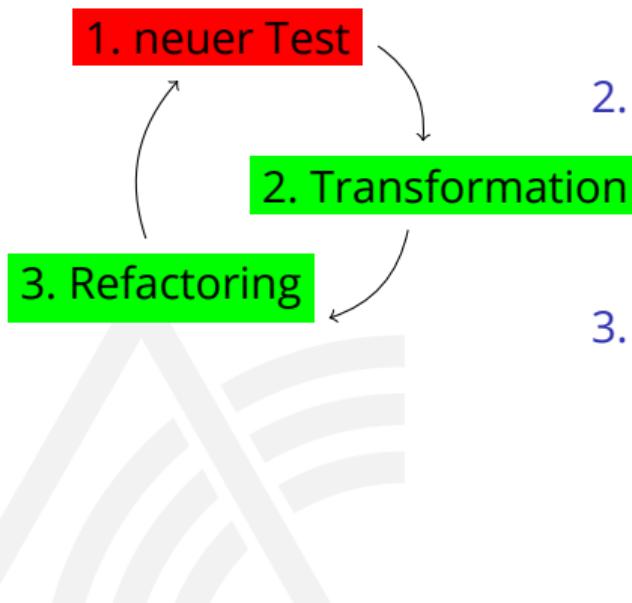
- UnitTests schreiben ist eine Fertigkeit und muss ständig geübt werden.
- Technische Voraussetzungen müssen sichergestellt sein.
- Team und Vorgesetzte müssen automatisiertes Testen unterstützen.



# 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

# was ist Test Driven Developement?



1. Schreibe einen neuen Test, gerade so viel dass er fehl schlägt  
(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 berchen und ohne neue Funktinalität (Geschäftslogik) hinzuzufügen.

# Anforderungen an UnitTests



# Was ist ein guter UnitTest?

- Fast
  - Independent
  - Repeatable
  - Self Checking
  - Timely
- 
- Readable
  - Trustworthy
  - Fast
  - Maintainable

## FIRST - Was bedeutet schnell?

- Tests sollen sehr häufig ausgeführt werden (wo möglich nach jedem Speichern). Dadurch erhalten wir sofort Feedback, ob die letzte Änderung einen Fehler verursacht hat.
- Aufwendige und potenziell langsame Logik in Abhängigkeiten der Unit müssen ersetzt werden.



## FIRST - Was bedeutet *unabhängig*?

- Ein Test soll nur fehlgeschlagen, wenn die getestete Unit fehlerhaft ist
- Änderungen der Logik anderer Units hat keinen Einfluss.
- ⇒Mocking-Framework einsetzen



## FIRST - Was bedeutet *wiederholbar*?

- beliebig oft ausführbar
- Ergebnis ist immer gleich



## FIRST - Was bedeutet *selbs-verifizierend*?

- Ergebnis ist eindeutig
- binär
- kein Beurteilung durch einen Menschen Notwendig



## FIRST - Was bedeutet *zeitnah*?

- entstehen gemeinsam mit dem Produktionscode
- TDD → Test vor Produktionscode



# RTFM - 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  
Variablenamen sorgsam wählen
- **Welche Eigenschaften der Ergebnisse sind wichtig?**  
Explizit Variablen für Ergebnisse anlegen  
Variablenamen sorgsam wählen  
Verifizierungsmethoden mit sprechenden Namen.

# RTFM - 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

# RTFM - 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

# RTFM - Was bedeutet schnell?

- Tests sollen sehr häufig ausgeführt werden (wo möglich nach jedem Speichern). Dadurch erhalten wir sofort Feedback, ob die letzte Änderung einen Fehler verursacht hat.
- Aufwendige und potenziell langsame Logik in Abhängigkeiten der Unit müssen ersetzt werden.



## RTFM - Was bedeutet *wartbar*?

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



# Anforderungen an zu testenden Code



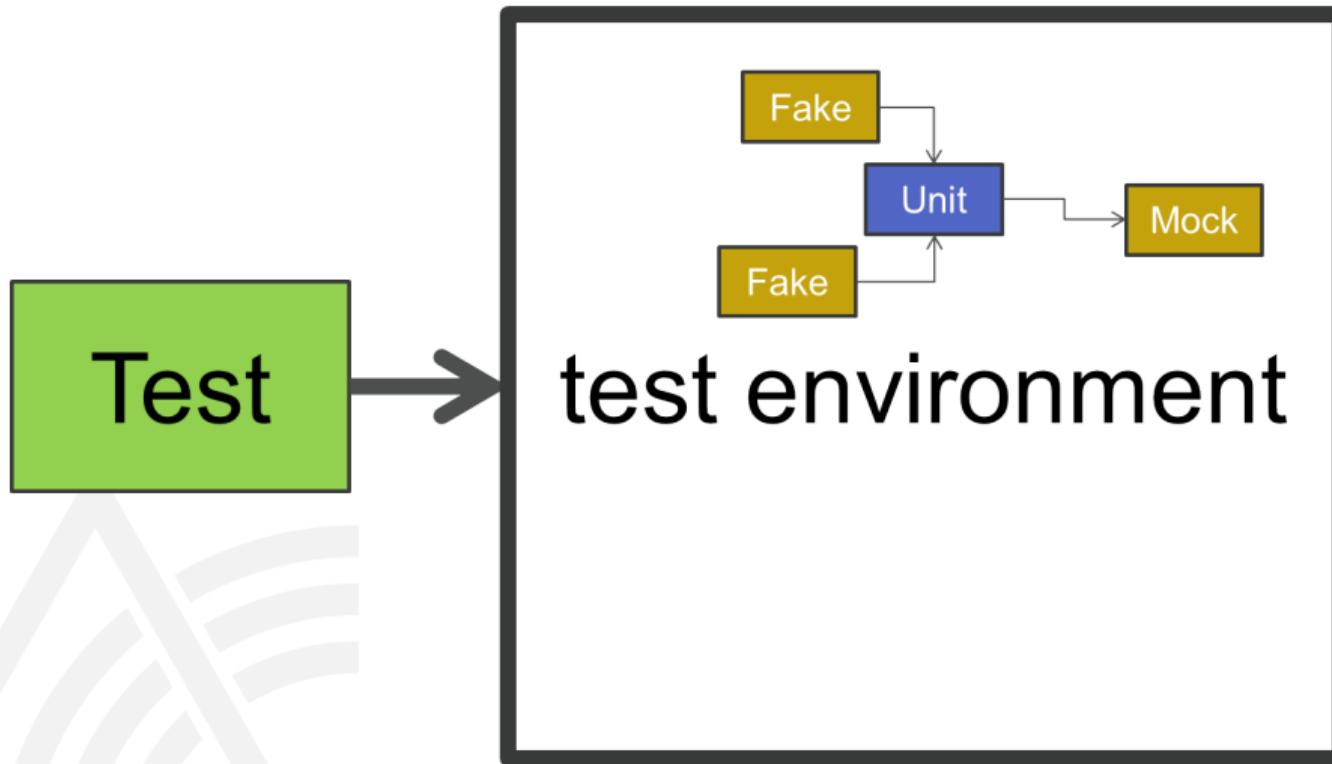
# Testbarkeit von produktivem Code

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 Injection / Inversion of Control
- 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"<sup>1</sup>

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

---

<sup>1</sup>"Clean Code" by Robert C Martin, ISBN-13: 978-0132350884

# UnitTest in Java



# Starterkit für Java Entwickler

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

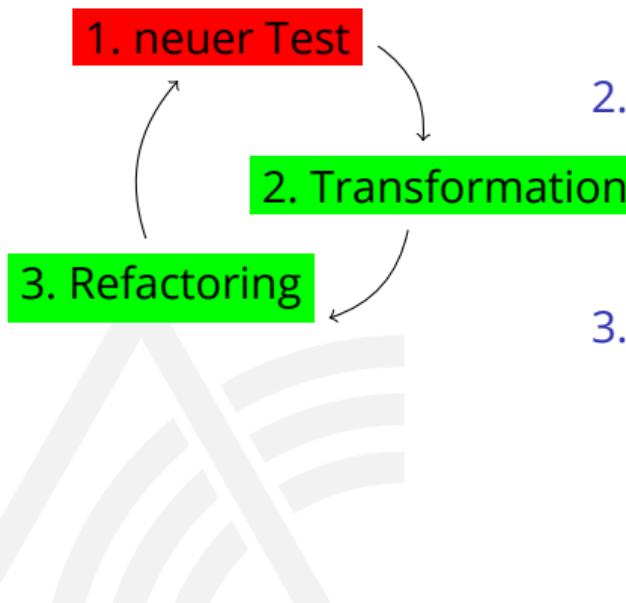
# einen JUnit Test schreiben

```
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("allzerosin", // 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("simulate_error"))  
10            .when( mockObject ).anyMethod();  
11        doReturn(mockObject, null, mockObject)  
12            .thenThrow(new RuntimeException("simulate_error_after_4th_call"))  
13            .when( spyObject ).methodWithReturnValue();  
14  
15         // alternative for non void methods:  
16         .when( spyObject.methodWithReturnValue() )  
17             .thenReturn(mockObject, null, mockObject)  
18             .thenThrow(new RuntimeException("simulate_error_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 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 berchen und ohne neue Funktinalität (Geschäftslogik) hinzuzufügen.

# Transformationsprämissen<sup>1</sup>

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

---

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

Vielen Dank für die Aufmerksamkeit.



## Thomas Papendieck

Senior-Consultant

Norsk-Data-Straße 4  
61348 Bad Homburg

[thomas.papendieck@opitz-consulting.com](mailto:thomas.papendieck@opitz-consulting.com)

+49 6172 66260 1523



[WWW.OPITZ-CONSULTING.COM](http://WWW.OPITZ-CONSULTING.COM)



[@OOC\\_WIRE](https://twitter.com/OOC_WIRE)



[OPITZCONSULTING](https://www.youtube.com/OPITZCONSULTING)



[opitzconsulting](https://www.linkedin.com/company/opitzconsulting)



[opitz-consulting-bcb8-1009116](https://www.x.com/opitz-consulting-bcb8-1009116)