

**JUnit. HierarchicalContextRunner. Mehr Struktur.
TDD. Clean Code. Verantwortung. Skills. Namics.**

JUnit



**XP
DAYS GERMANY**



Stefan Bechtold. Principal Software Engineer.

16. Oktober 2014

Wer bin ich...?



Stefan Bechtold.
Principal Software Engineer.

E-Mail: stefan.bechtold@namics.com
Twitter: [@bechte](https://twitter.com/bechte)

**Querdenker. Zielstrebig.
Herzblut. Java. TDD.
Commerce. Namics.**

Berufserfahrung
11 Jahre
bei Namics seit 2006

Berufliche Aktivitäten bei Namics:

- Technical Architect mit Schwerpunkt auf Java-basierte Commerce und Content Management Systeme
- Coaching für Themen rund um:
TDD, Clean Code, Software Architecture & Design

Aktivitäten in der Open-Source-Community:

- Entwicklung des HierarchicalContextRunner
- JUnit-Framework Committer

Worum geht's im Vortrag...?

→ Motivation für

- saubere (Unit-) Tests
- gut strukturierte (Unit-) Tests

→ JUnit – HierarchicalContextRunner

- Funktionsweise
- Limitierungen

→ Showcase / Code Samples

Und wer hört eigentlich zu...?

→ **Mit welchen Programmiersprachen entwickelt ihr?**

- C/C++? Ruby? Scala? Java?

→ **Wer verwendet JUnit für das (Unit-) Testing?**

- Ja, immer? Nein? Andere?

→ **Wer kennt den Enclosed Runner von JUnit?**

- Nutze ich gelegentlich Nie davon gehört!

→ **Wer kennt den HierarchicalContextRunner?**

- Nutze ich gelegentlich Nie davon gehört!

Sauber strukturierte (Unit-) Tests

Die Grundregeln

- #1 Behandle Test-Code wie Produktiv-Code
- #2 Verwende Test Patterns für bessere Lesbarkeit
- #3 Vermeide unzuverlässige Tests
- #4 Teste im angemessenen Level (Test-Pyramide)
- #5 Verwende (die richtigen) Test-Doubles

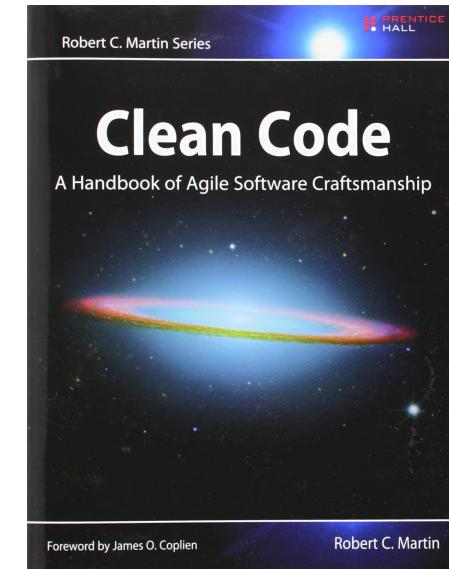
Quelle: Marcos Brizeno – Write Better Tests in 5 Steps (<http://bit.ly/1sydhiC>)

Die Grundregeln

#1 Behandle Test-Code wie Produktiv-Code

- Die Tests sind die Spezifikation des Systems
- Fehlgeschlagende Tests müssen Feedback geben
- Lesbarkeit der Tests ist das höchste Gut!

→ Clean Code: A Handbook of
Agile Software Craftsmanship
Robert Martin, <http://cleancoders.com>



Die Grundregeln

#2 Verwende Test Pattern für bessere Lesbarkeit

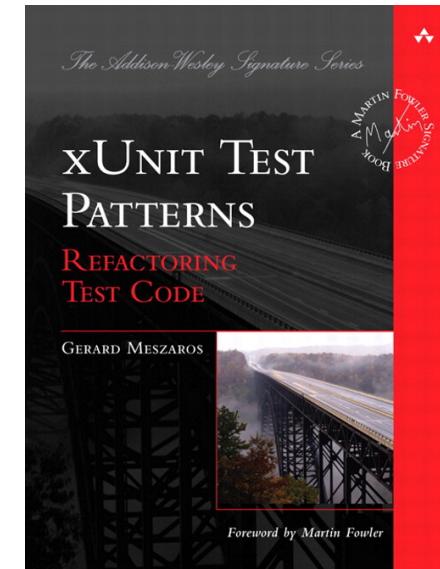
→ Pattern verbessern auch den Test Code ;-)

→ Zusätzliche Pattern speziell für Tests:

- Arrange, Act, Assert (the 3-As)
- Given, When, Then
- viele weitere

→ XUnit Test Patterns, Refactoring Test Code

Gerard Meszaros, <http://xunitpatterns.com>



Die Grundregeln

#3 Vermeide unzuverlässige Tests

→ (Unit-) Tests müssen deterministisch sein!

→ Nachteile nicht-deterministischer Tests:

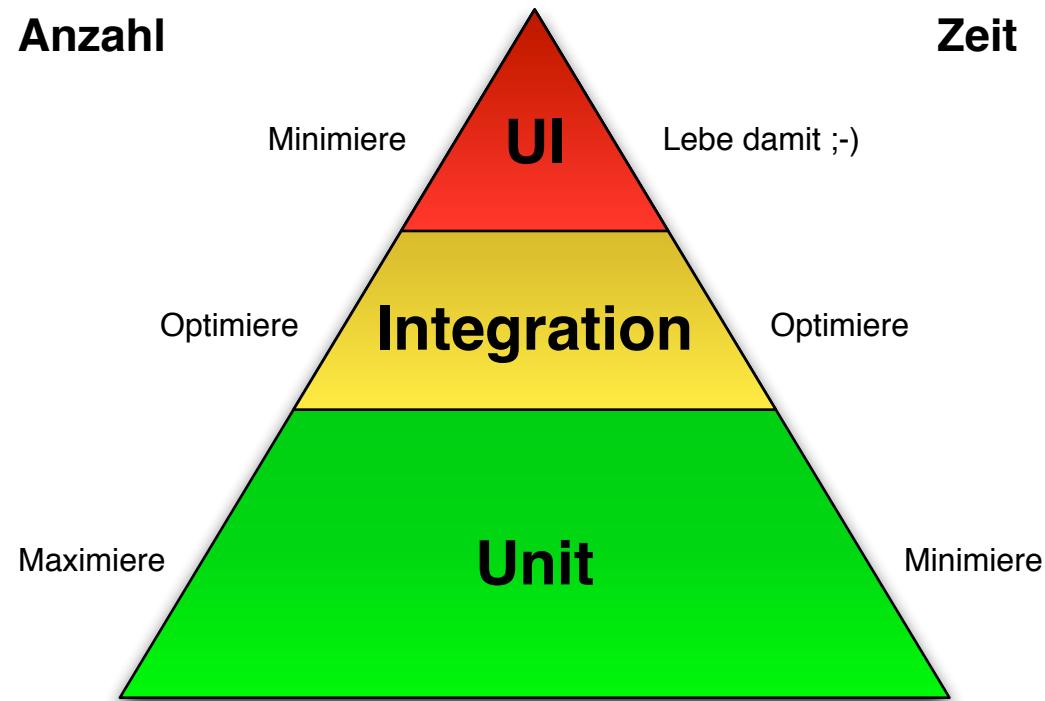
- Unerwünschte und unvorhersehbare Fehler in Tests
- Seiteneffekte, die ganze Test-Suites zerstören können
- Verlust des Vertrauens in die Test-Suite

→ “Indeed you really ought to throw a non-deterministic test away, since if you don’t it has an infectious quality.”

Martin Fowler, <http://martinfowler.com/articles/nonDeterminism.html>

Die Grundregeln

#4 Teste im angemessenen Level (Test-Pyramide)



Die Grundregeln

#5 Verwende (die richtigen) Test-Doubles

→ Verschiedene Arten von Test-Doubles:

Name	Test-Daten	Inspektion	Verifikation	Echte Logik
Dummy	Nein	Nein	Nein	Nein
Stub	Ja	Nein	Nein	Nein
Spy	Ja	Ja	Nein	Nein
Mock	Ja	Ja	Ja	Nein
Fake	Ja	Ja	Ja	Ja



JUnit - HierarchicalContextRunner

Funktionsweise

→ **Kontext anlegen durch innere Klassen**

- Semantische Gruppierung von Tests
- Reduktion Test-Fixtures auf das Nötigste

→ **Unbegrenzte Kontext-Hierarchie**

- Jeder Kontext stellt eigene Test-Fixtures bereit
- Abhängigkeiten von Tests abbildbar

→ **Eigenschaften entlang der Hierarchie verfügbar**

- Objekte wiederverwendbar, z.B. @Rule

→ **Ideale Abbildung von Given-When-Then**

- Kontext repräsentiert den „Given“-Teil

Funktionsweise

→ Ein einfaches Beispiel:

```
@RunWith(HierarchicalContextRunner.class)
public class SocketServerTest {
    private SocketServer server;
    private int port;

    @Before
    public void setup() {...}

    @After
    public void tearDown() throws Exception {...}

    public class WithClosingSocketService {...}

    public class WithReadingSocketService {...}

    public class WithEchoSocketService {...}
}
```

Funktionsweise

→ **Evaluieren je Test-Klasse:**

- Runner führt @BeforeClass aus
- Runner evaluierst alle Tests der Klasse
- Runner ruft sich rekursive für alle Kontexte auf
- Runner führt @AfterClass aus

→ **Evaluierung je Test-Methode:**

- Runner erzeugt Test-Object (top-down)
- Runner führt @Before aus (top-down)
- Runner startet den Test
- Runner führt @After aus (bottom-up)

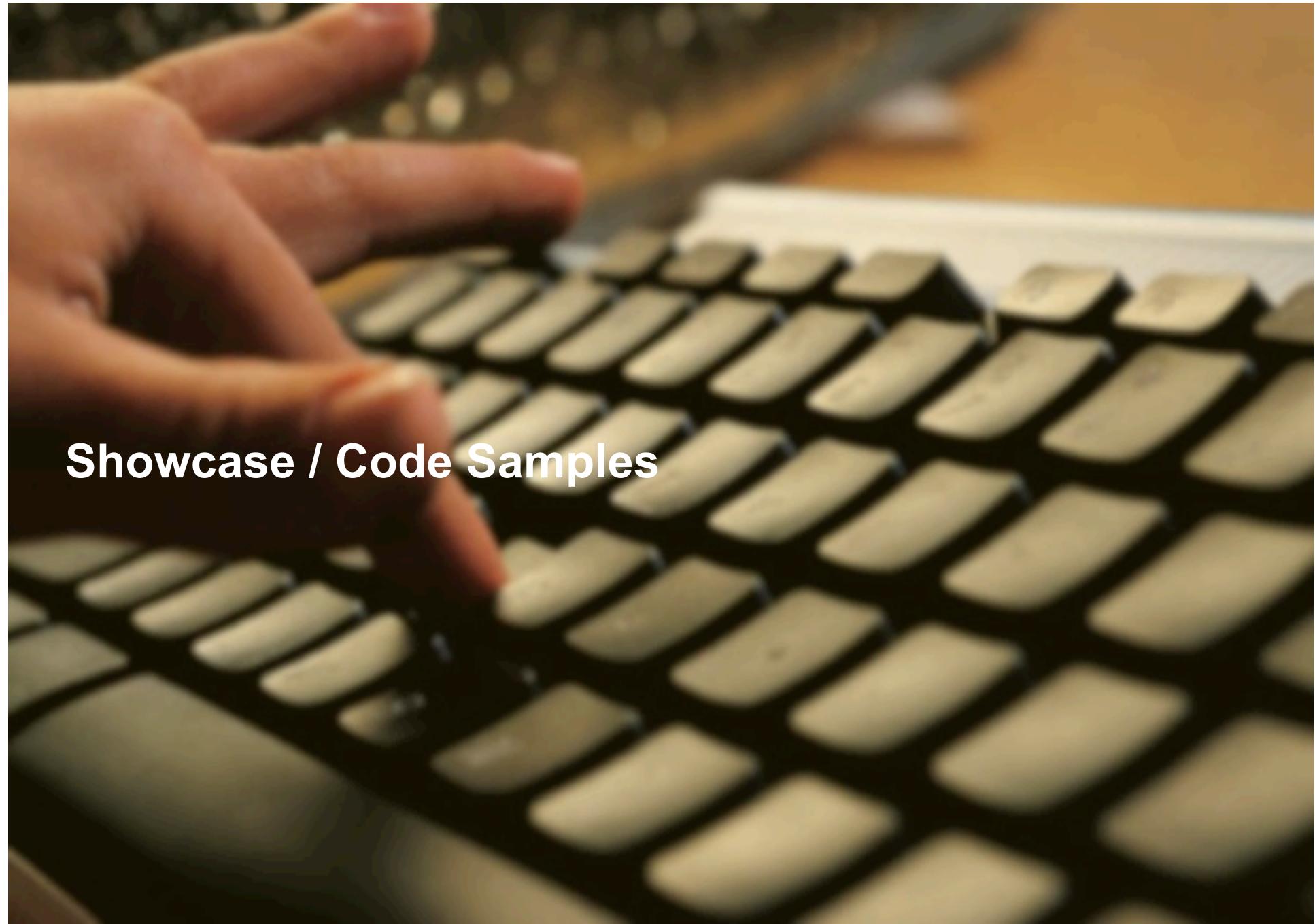
Limitierungen

→ Allgemeine Limitierungen

- Keine statischen Klassen innerhalb eines Kontexts
- Keine statischen Methoden innerhalb eines Kontexts
- Ausführung einzelner Test-Methoden (noch) nicht möglich

→ JUnit Limitierungen

- Support ab Version JUnit 4.11
- Operiert nicht mit anderen JUnit-Runner zusammen
- Beschreibung im Ergebnis (noch) nicht optimal



Showcase / Code Samples

**JUnit. HierarchicalContextRunner. Mehr Struktur.
TDD. Clean Code. Verantwortung. Skills. Namics.**

JUnit



**XP
DAYS GERMANY**

Stefan Bechtold. Principal Software Engineer.

E-Mail: stefan.bechtold@namics.com

Twitter: @bechte