

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

**JUnit**



**XP  
DAYS GERMANY**



**Stefan Bechtold. Principal Software Engineer.**

**16. Oktober 2014**

# Ein typischer (Unit-) Test...

---

```
package xpdays2014.samples.stack.defaultrunner;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import xpdays2014.samples.stack.Stack;

public class StackTest {
    private Stack stack;

    @Before
    public void setUp() throws Exception {
        stack = Stack.make(2);
    }

    @Test(expected = Stack.IllegalCapacity.class)
    public void whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity()
    throws Exception {
        Stack.make(-1);
    }

    @Test(expected = Stack.Overflow.class)
    public void whenCreatingStackWithZeroCapacity_AnyPushShouldOverflow() throws
Exception {
        stack = Stack.make(0);
        stack.push(1);
    }

    @Test
    public void newlyCreatedStack_ShouldBeEmpty() throws Exception {
        assertTrue(stack.isEmpty());
        assertEquals(0, stack.getSize());
    }

    @Test(expected = Stack.Underflow.class)
    public void whenEmptyStackIsPopped_ShouldThrowUnderflow() throws Exception {
        stack.pop();
    }

    @Test(expected = Stack.Empty.class)
    public void whenStackIsEmpty_TopThrowsEmpty() throws Exception {
        stack.top();
    }

    @Test(expected = Stack.Empty.class)
    public void withZeroCapacityStack_topThrowsEmpty() throws Exception {
        stack = Stack.make(0);
        stack.top();
    }

    @Test
    public void givenStackWithNoTwo_FindTwoShouldReturnNull() throws Exception {
        assertNull(stack.find(2));
    }

    @Test
    public void afterOnePush_StackSizeShouldBeOne() throws Exception {
        stack.push(1);
        assertFalse(stack.isEmpty());
        assertEquals(1, stack.getSize());
    }

    @Test
    public void whenOneIsPushed_OneIsOnTop() throws Exception {
        stack.push(1);
        assertEquals(1, stack.top());
    }

    @Test
    public void whenOneIsPushed_OneIsPopped() throws Exception {
        stack.push(1);
        assertEquals(1, stack.pop());
    }

    @Test
    public void afterOnePushAndOnePop_ShouldBeEmpty() throws Exception {
        stack.push(1);
        stack.pop();
        assertTrue(stack.isEmpty());
        assertEquals(0, stack.getSize());
    }

    @Test
    public void whenOneAndTwoArePushed_TwoAndOneArePopped() throws Exception {
        stack.push(1);
        stack.push(2);
        assertEquals(2, stack.pop());
        assertEquals(1, stack.pop());
    }

    @Test
    public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {
        stack.push(1);
        stack.push(2);
        int oneIndex = stack.find(1);
        int twoIndex = stack.find(2);
        assertEquals(1, oneIndex);
        assertEquals(0, twoIndex);
    }

    @Test(expected = Stack.Overflow.class)
    public void whenPushedPastLimit_ShouldThrowOverflow() throws Exception {
        stack.push(1);
        stack.push(1);
        stack.push(1);
    }
}
```

---

# Ein typischer (Unit-) Test...

## → Kurze Analyse:

- Übersichtliche Tests
- Sehr gute Lesbarkeit
- Gutes Naming
- Klar und verständlich

## → Alles super!

```
@Test  
public void whenOneIsPushed_OneIsOnTop() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.top());  
}  
  
@Test  
public void whenOneIsPushed_OneIsPopped() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void afterOnePushAndOnePop_ShouldBeEmpty() throws Exception {  
    stack.push(1);  
    stack.pop();  
    assertTrue(stack.isEmpty());  
    assertEquals(0, stack.getSize());  
}  
  
@Test  
public void whenOneAndTwoArePushed_TwoAndOneArePopped() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    assertEquals(2, stack.pop());  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    int oneIndex = stack.find(1);  
    int twoIndex = stack.find(2);  
}
```

# Ein typischer (Unit-) Test...

## → Kritisches Publikum:

- Übersichtliche Tests?
- Sehr gute Lesbarkeit?
- Gutes Naming?
- Klar und verständlich?

## → Alles super?

```
@Test  
public void whenOneIsPushed_OneIsOnTop() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.top());  
}  
  
@Test  
public void whenOneIsPushed_OneIsPopped() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void afterOnePushAndOnePop_ShouldBeEmpty() throws Exception {  
    stack.push(1);  
    stack.pop();  
    assertTrue(stack.isEmpty());  
    assertEquals(0, stack.getSize());  
}  
  
@Test  
public void whenOneAndTwoArePushed_TwoAndOneArePopped() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    assertEquals(2, stack.pop());  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    int oneIndex = stack.find(1);  
    int twoIndex = stack.find(2);  
}
```

# Ein typischer (Unit-) Test...

## → Was sagt der Bauch?

- Redundanz im Test-Setup
- Redundanz im Naming
- Naming nicht eindeutig

## → Gibt es eine Lösung?

- Test-Setup Helper?
- Test-Object Builder?
- Code-Reviews einführen?

```
@Test  
public void whenOneIsPushed_OneIsOnTop() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.top());  
}  
  
@Test  
public void whenOneIsPushed_OneIsPopped() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void afterOnePushAndOnePop_ShouldBeEmpty() throws Exception {  
    stack.push(1);  
    stack.pop();  
    assertTrue(stack.isEmpty());  
    assertEquals(0, stack.getSize());  
}  
  
@Test  
public void whenOneAndTwoArePushed_TwoAndOneArePopped() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    assertEquals(2, stack.pop());  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    int oneIndex = stack.find(1);  
    int twoIndex = stack.find(2);  
}
```

# Wer bin ich...?



**Stefan Bechtold.**  
Principal Software Engineer.  
Namics GmbH, Frankfurt

E-Mail: [stefan.bechtold@namics.com](mailto: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:

- Java Commerce- und Content-Management-Systeme
- Coach 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**

- sauber strukturierte (Unit-) Tests

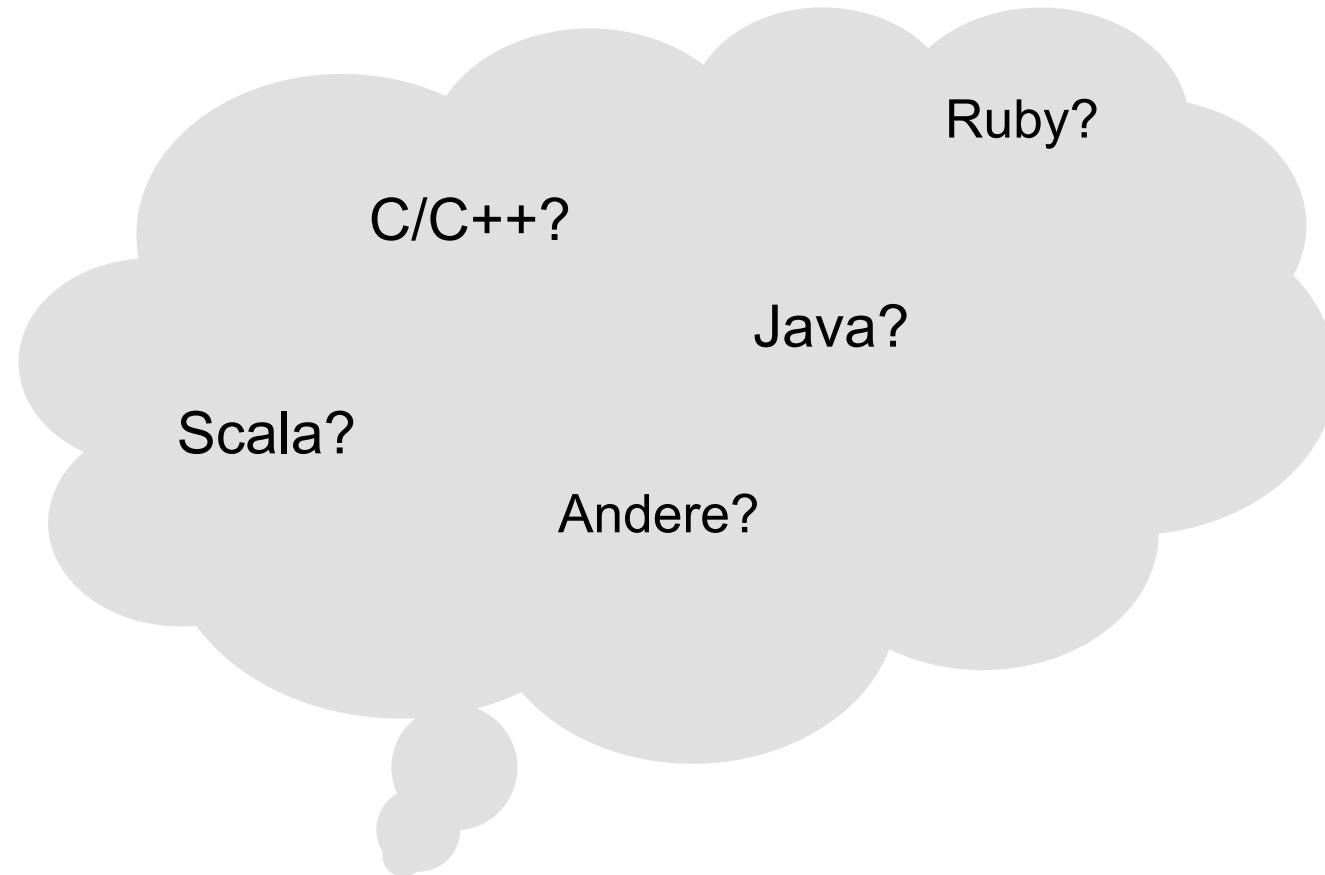
→ **JUnit – HierarchicalContextRunner**

- Einführung
- Funktionsweise
- Limitierungen

→ **Showcase / Code Samples**

## Und wer hört eigentlich zu...?

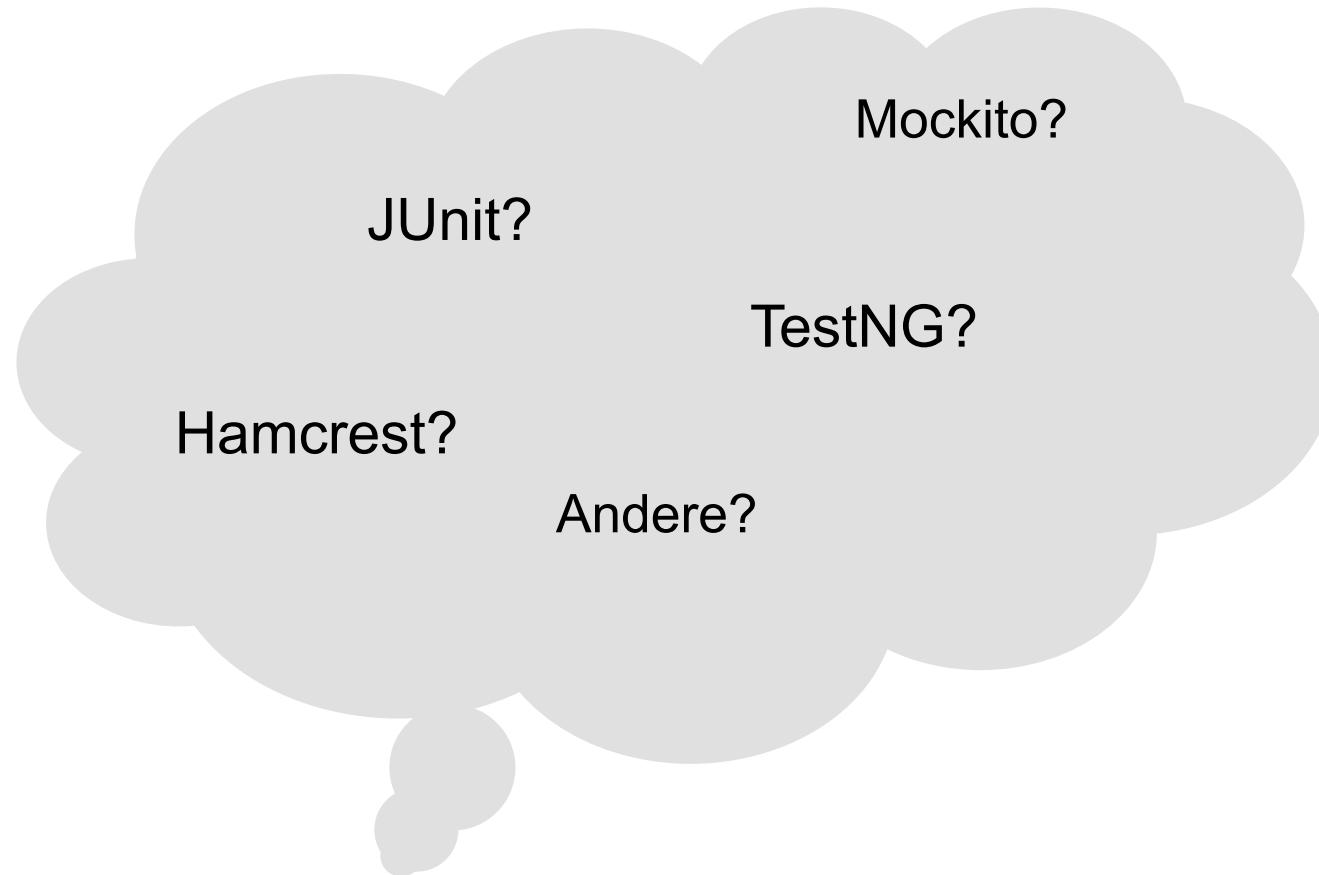
---



→ **Mit welchen Programmiersprachen entwickelt Ihr?**

## Und wer hört eigentlich zu...?

---

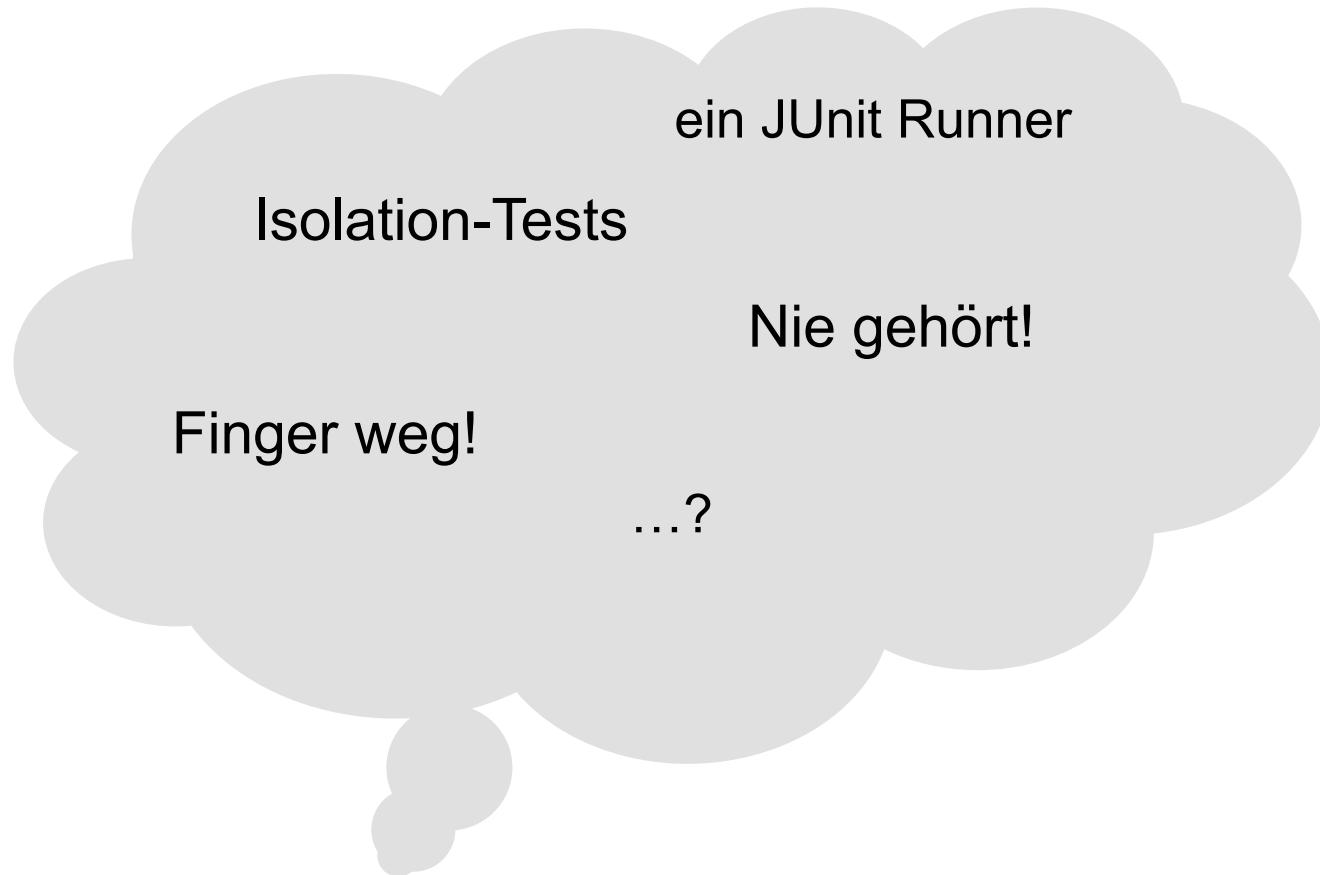


→ **Welche Frameworks nutzt Ihr für (Unit-) Tests in Java?**

---

## Und wer hört eigentlich zu...?

---

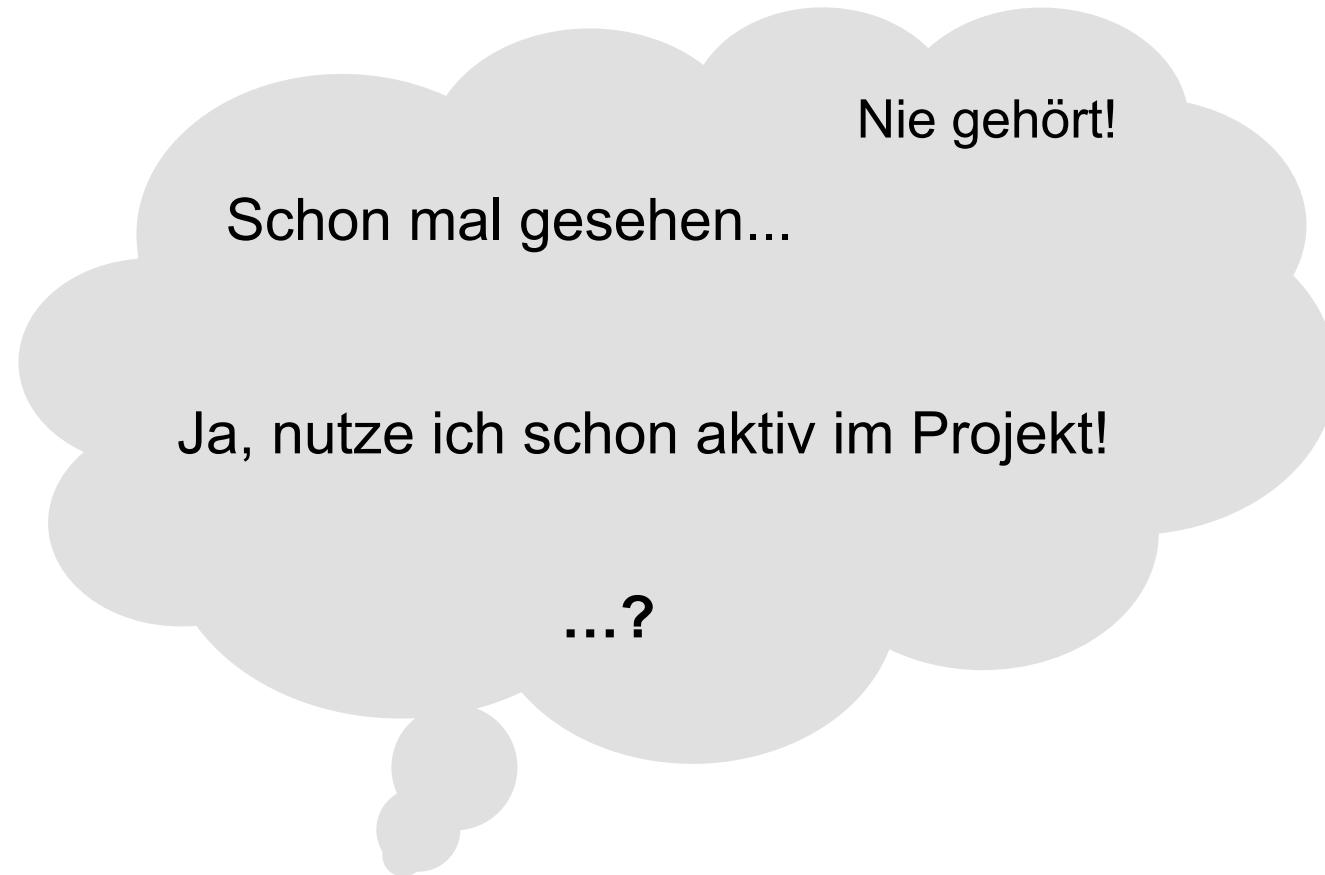


→ Was fällt Dir ein bei: “`@RunWith(Enclosed.class)`”?

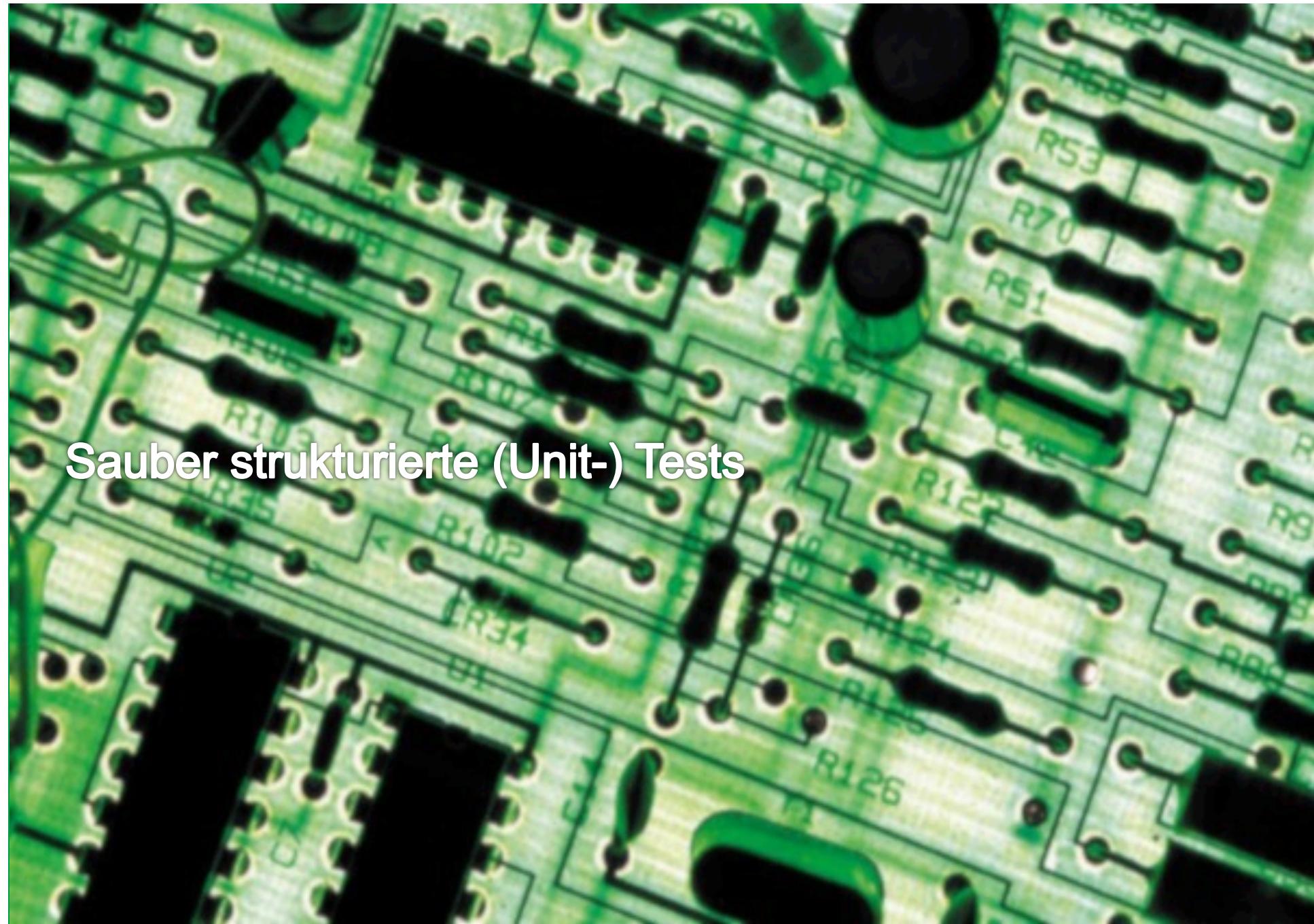
---

## Und wer hört eigentlich zu...?

---



→ Kennst Du den JUnit HierarchicalContextRunner schon?

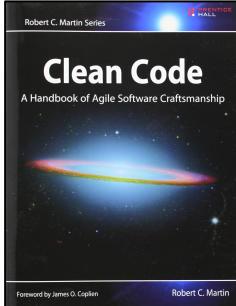


Sauber strukturierte (Unit-) Tests

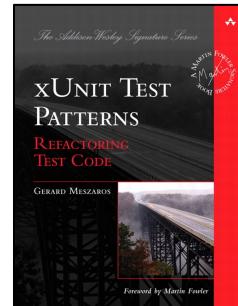
Test. Code. Refactor. Jetzt auch mit Struktur. **Namics.**

# 5 Grundregeln

---



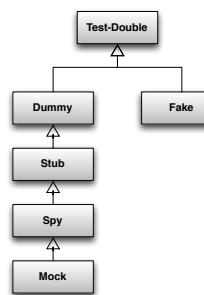
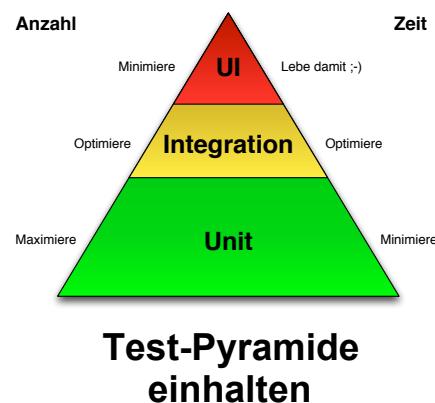
**“Test-Code ist  
Produktiv-Code”**  
Robert Martin



**Test-Pattern, z.B.:  
Arrange, Act, Assert**  
Gerard Meszaros



**“Nicht deterministische  
Tests gehören gelöscht!”**  
Martin Fowler



**Verwendung der  
richtigen Test-Doubles**

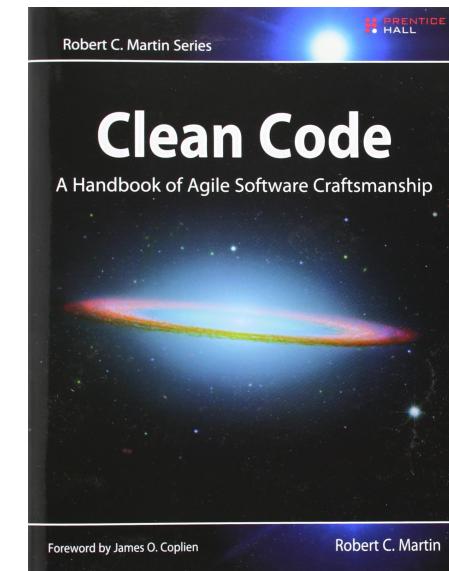
# Die Grundregeln

---

## #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

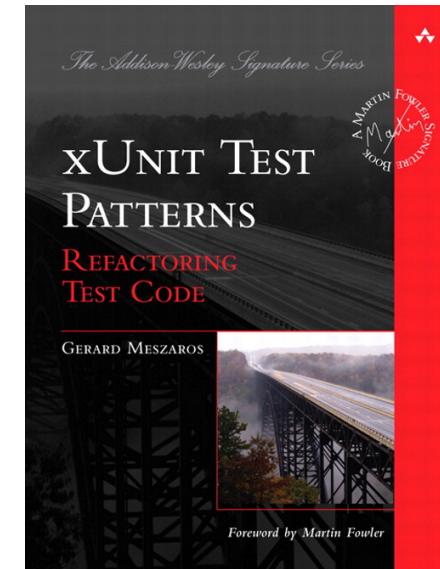
→ Pattern verbessern auch den Test Code ;-)

→ 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

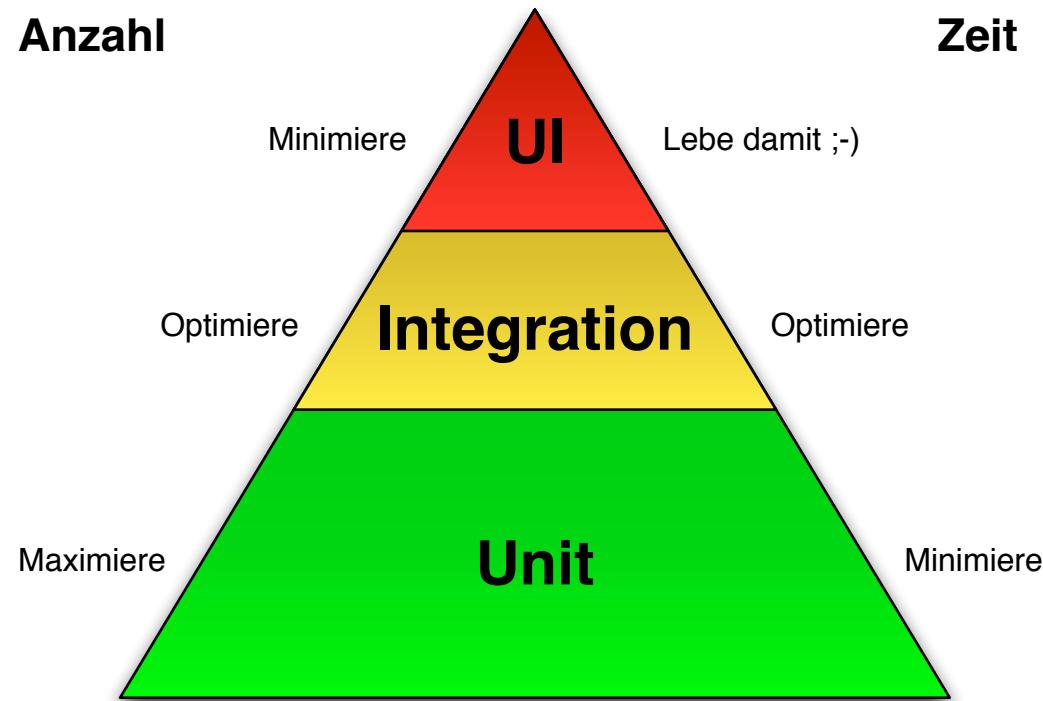
→ “Indeed you really ought to throw a non-deterministic test away, since if you don’t it has an infectious quality.”

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

# Die Grundregeln

---

## #4 Teste im angemessenen Level (Test-Pyramide)



# Die Grundregeln

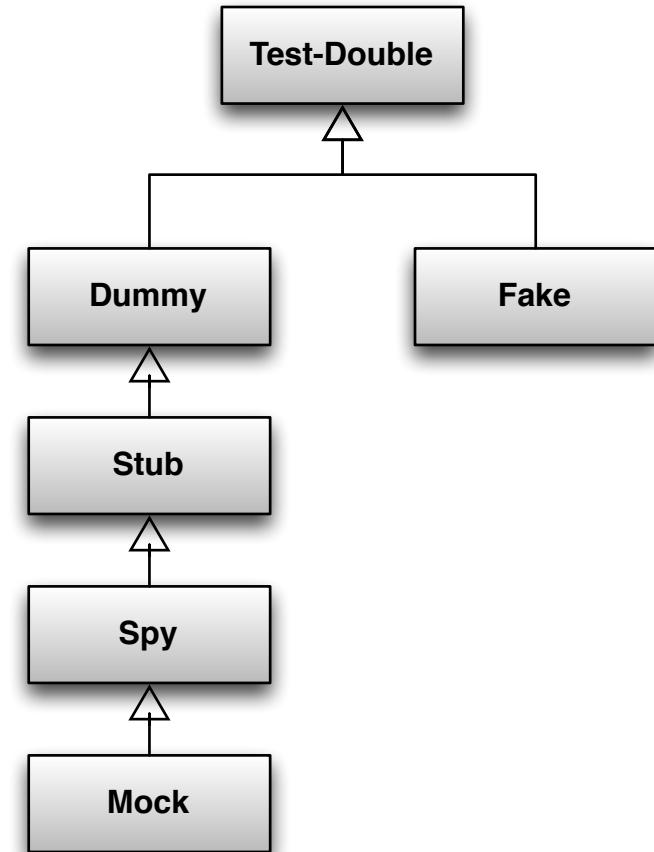
---

## #5 Verwende (die richtigen) Test-Doubles

### → Einsatz von Mocking Frameworks:

- Niemals für: Dummy, Stub & Spy
- Manchmal sinnvoll für: Mock / Fake

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





## JUnit - HierarchicalContextRunner

# Ein typischer (Unit-) Test...

## → Was sagt der Bauch?

- Redundanz im Test-Setup
- Redundanz im Naming
- Naming nicht eindeutig

## → Und die Lösung?

```
@Test  
public void whenOneIsPushed_OneIsOnTop() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.top());  
}  
  
@Test  
public void whenOneIsPushed_OneIsPopped() throws Exception {  
    stack.push(1);  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void afterOnePushAndOnePop_ShouldBeEmpty() throws Exception {  
    stack.push(1);  
    stack.pop();  
    assertTrue(stack.isEmpty());  
    assertEquals(0, stack.getSize());  
}  
  
@Test  
public void whenOneAndTwoArePushed_TwoAndOneArePopped() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    assertEquals(2, stack.pop());  
    assertEquals(1, stack.pop());  
}  
  
@Test  
public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {  
    stack.push(1);  
    stack.push(2);  
    int oneIndex = stack.find(1);  
    int twoIndex = stack.find(2);  
}
```

## Die Outline der Testklasse

- `whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity`
- `whenCreatingStackWithZeroCapacity_AnyPushShouldOverflow`
- `newlyCreatedStack_ShouldBeEmpty`
- `whenEmptyStackIsPopped_ShouldThrowUnderflow`
- `whenStackIsEmpty_TopThrowsEmpty`
- `withZeroCapacityStack_topThrowsEmpty`
- `givenStackWithNoTwo_FindTwoShouldReturnNull`
- `afterOnePush_StackSizeShouldBeOne`
- `whenOneIsPushed_OneIsOnTop`
- `whenOneIsPushed_OneIsPopped`
- `afterOnePushAndOnePop_ShouldBeEmpty`
- `whenOneAndTwoArePushed_TwoAndOneArePopped`
- `givenStackWithOneTwoPushed_FindOneAndTwo`
- `whenPushedPastLimit_ShouldThrowOverflow`

## Die Outline der Testklasse

- `whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity`
- `whenCreatingStackWithZeroCapacity_AnyPushShouldOverflow`
- `withZeroCapacityStack_topThrowsEmpty`
- `newlyCreatedStack_ShouldBeEmpty`
- `whenEmptyStackIsPopped_ShouldThrowUnderflow`
- `whenStackIsEmpty_TopThrowsEmpty`
- `givenStackWithNoTwo_FindTwoShouldReturnNull`
- `afterOnePush_StackSizeShouldBeOne`
- `whenOneIsPushed_OneIsOnTop`
- `whenOneIsPushed_OneIsPopped`
- `afterOnePushAndOnePop_ShouldBeEmpty`
- `whenOneAndTwoArePushed_TwoAndOneArePopped`
- `givenStackWithOneTwoPushed_FindOneAndTwo`
- `whenPushedPastLimit_ShouldThrowOverflow`

## Die Outline der Testklasse

- `whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity`
- `GivenStackWithZeroCapacity`
  - `anyPushShouldOverflow`
  - `topThrowsEmpty`
- `newlyCreatedStack_ShouldBeEmpty`
- `whenEmptyStackIsPopped_ShouldThrowUnderflow`
- `whenStackIsEmpty_TopThrowsEmpty`
- `givenStackWithNoTwo_FindTwoShouldReturnNull`
- `afterOnePush_StackSizeShouldBeOne`
- `whenOneIsPushed_OneIsOnTop`
- `whenOneIsPushed_OneIsPopped`
- `afterOnePushAndOnePop_ShouldBeEmpty`
- `whenOneAndTwoArePushed_TwoAndOneArePopped`
- `givenStackWithOneTwoPushed_FindOneAndTwo`
- `whenPushedPastLimit_ShouldThrowOverflow`

## Die Outline der Testklasse

- `whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity`
- `GivenStackWithZeroCapacity`
  - `anyPushShouldOverflow`
  - `topThrowsEmpty`
- `GivenEmptyStack`
  - `shouldBeEmpty`
  - `popShouldThrowUnderflow`
  - `topThrowsEmpty`
  - `findTwoShouldReturnNull`
- `afterOnePush_StackSizeShouldBeOne`
- `whenOneIsPushed_OneIsOnTop`
- `whenOneIsPushed_OneIsPopped`
- `afterOnePushAndOnePop_ShouldBeEmpty`
- `whenOneAndTwoArePushed_TwoAndOneArePopped`
- `givenStackWithOneTwoPushed_FindOneAndTwo`
- `whenPushedPastLimit_ShouldThrowOverflow`

## Die Outline der Testklasse

- **whenCreatingAStackWithNegativeSize\_ShouldThrowIllegalCapacity**
- **GivenStackWithZeroCapacity**
  - **anyPushShouldOverflow**
  - **topThrowsEmpty**
- **GivenEmptyStack**
  - **shouldBeEmpty**
  - **popShouldThrowUnderflow**
  - **topThrowsEmpty**
  - **findTwoShouldReturnNull**
- **GivenOnePushed**
  - **stackSizeShouldBeOne**
  - **oneIsOnTop**
  - **oneIsPopped**
  - **stackShouldBeEmptyAfterPop**
- **whenOneAndTwoArePushed\_TwoAndOneArePopped**
- **givenStackWithOneTwoPushed\_FindOneAndTwo**
- **whenPushedPastLimit\_ShouldThrowOverflow**

## Die Outline der Testklasse

- **whenCreatingAStackWithNegativeSize\_ShouldThrowIllegalCapacity**
- **GivenStackWithZeroCapacity**
  - **anyPushShouldOverflow**
  - **topThrowsEmpty**
- **GivenEmptyStack**
  - **shouldBeEmpty**
  - **popShouldThrowUnderflow**
  - **topThrowsEmpty**
  - **findTwoShouldReturnNull**
- **GivenOnePushed**
  - **stackSizeShouldBeOne**
  - **oneIsOnTop**
- **GivenOnePop**
  - **oneIsPopped**
  - **stackShouldBeEmptyAfterPop**
- **GivenTwoPushed**
  - **twoAndOneArePopped**
  - **findOneAndTwo**
  - **anotherPushShouldThrowOverflow**

# Erkenntnisse

---

## → Beobachtungen:

- (Unit-) Tests sind hierarchisch abbildbar
- Einzelne Tests können gruppiert werden
- Gültigkeit & Sichtbarkeit von Fixtures
- Keine Redundanz & unnötigen Fixtures
- Leider keine Unterstützung in JUnit ☹

## → Anforderungen:

- (Einfache) Definition eines Kontext notwendig
- SetUp / TearDown je Hierarchie-Stufe benötigt

# JUnit - HierarchicalContextRunner

---

## → Basiert vollständig auf JUnit

- Wiederverwendung der Annotations (@Test, @Ignore, etc.)
- Formulierung der (Unit-) Tests unverändert
- JUnit Core Features direkt einsetzbar
- Bestehende Test Suites sofort ausführbar

## → Zusätzliche Funktionen:

- Erlaubt die Erstellung von Kontexten
- Kontexte können verschachtelt werden
- Erlaubt SetUp / TearDown in jedem Kontext

# Ein typischer (Unit-) Test ... revisited

```

@RunWith(HierarchicalContextRunner.class)
public class StackTest {
    private Stack stack;

    @Test(expected = Stack.IllegalCapacity.class)
    public void whenCreatingAStackWithNegativeSize_ShouldThrowIllegalCapacity() throws Exception {
        Stack.make(-1);
    }

    public class ZeroSizeStackContext {
        @Before
        public void setUp() throws Exception {
            stack = Stack.make(0);
        }

        @Test(expected = Stack.Overflow.class)
        public void anyPushShouldOverflow() throws Exception {
            stack.push(1);
        }

        @Test(expected = Stack.Empty.class)
        public void topShouldThrowEmpty() throws Exception {
            stack.top();
        }
    }

    public class GivenNewStack {
        @Before
        public void setUp() throws Exception {
            stack = Stack.make(2);
        }

        @Test
        public void shouldBeEmpty() throws Exception {
            assertTrue(stack.isEmpty());
            assertEquals(0, stack.getSize());
        }

        @Test(expected = Stack.Underflow.class)
        public void popShouldThrowUnderflow() throws Exception {
            stack.pop();
        }

        @Test(expected = Stack.Empty.class)
        public void topShouldThrowEmpty() throws Exception {
            stack.top();
        }

        @Test
        public void findTwoShouldReturnNull() throws Exception {
            assertNull(stack.find(2));
        }
    }

    public class GivenOnePushed {
        @Before
        public void pushOne() {
            stack.push(1);
        }

        @Test
        public void stackSizeShouldBeOne() throws Exception {
            assertFalse(stack.isEmpty());
        }
    }

    @Test
    public void oneIsOnTop() throws Exception {
        assertEquals(1, stack.top());
    }

    public class GivenOnePopped {
        private int poppedElement;

        @Before
        public void popOne() {
            poppedElement = stack.pop();
        }

        @Test
        public void oneIsPopped() throws Exception {
            assertEquals(1, poppedElement);
        }

        @Test
        public void stackShouldBeEmpty() throws Exception {
            assertTrue(stack.isEmpty());
            assertEquals(0, stack.getSize());
        }
    }

    public class GivenTwoPushed {
        @Before
        public void pushTwo() {
            stack.push(2);
        }

        @Test
        public void givenStackWithOneTwoPushed_FindOneAndTwo() throws Exception {
            int oneIndex = stack.find(1);
            int twoIndex = stack.find(2);
            assertEquals(1, oneIndex);
            assertEquals(0, twoIndex);
        }

        @Test
        public void twoAndOneArePopped() throws Exception {
            assertEquals(2, stack.pop());
            assertEquals(1, stack.pop());
        }

        @Test(expected = Stack.Overflow.class)
        public void anotherPushShouldThrowOverflow() throws Exception {
            stack.push(1);
        }
    }
}

```

## Funktionsweise

---

### → Evaluieren Test-Klasse

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

### → Evaluierung Test-Methode

- Runner erzeugt  
Test-Objekt (top-down)
- Runner führt  
@Before aus (top-down)
- Runner startet  
die Test-Methode
- Runner führt  
@After aus (bottom-up)

# Limitierungen

---

## → Allgemeine Limitierungen

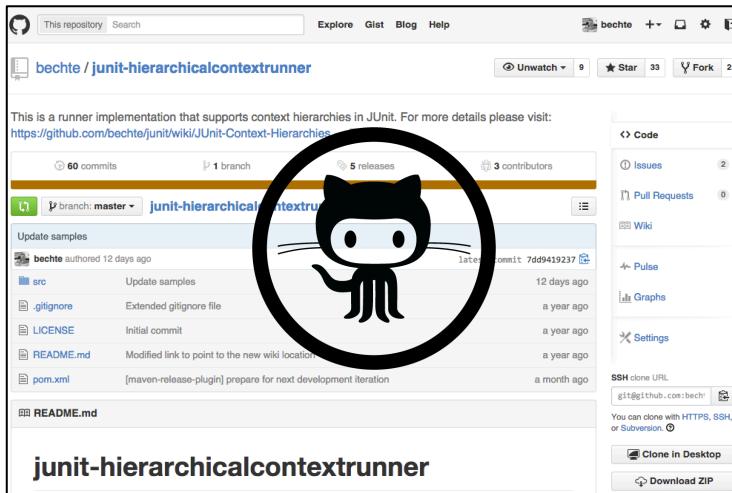
- Keine statischen Klassen innerhalb eines Kontexts
- Keine statischen Methoden innerhalb eines Kontexts
- Ausführung einzelner Test-Methoden (noch) nicht möglich

## → JUnit Limitierungen

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

# Weitere Literatur

---



github Repository  
HierarchicalContextRunner



Clean Coders Screencasts  
CleanCode Episodes: 20, 21  
JavaCaseStudy Episodes: 3, 4



## Showcase / Code Samples

Alle Beispiele und die Präsentation sind zu finden unter:  
<https://github.com/bechtle/XPDays2014>

**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](mailto:stefan.bechtold@namics.com)**

**Twitter: @bechte**