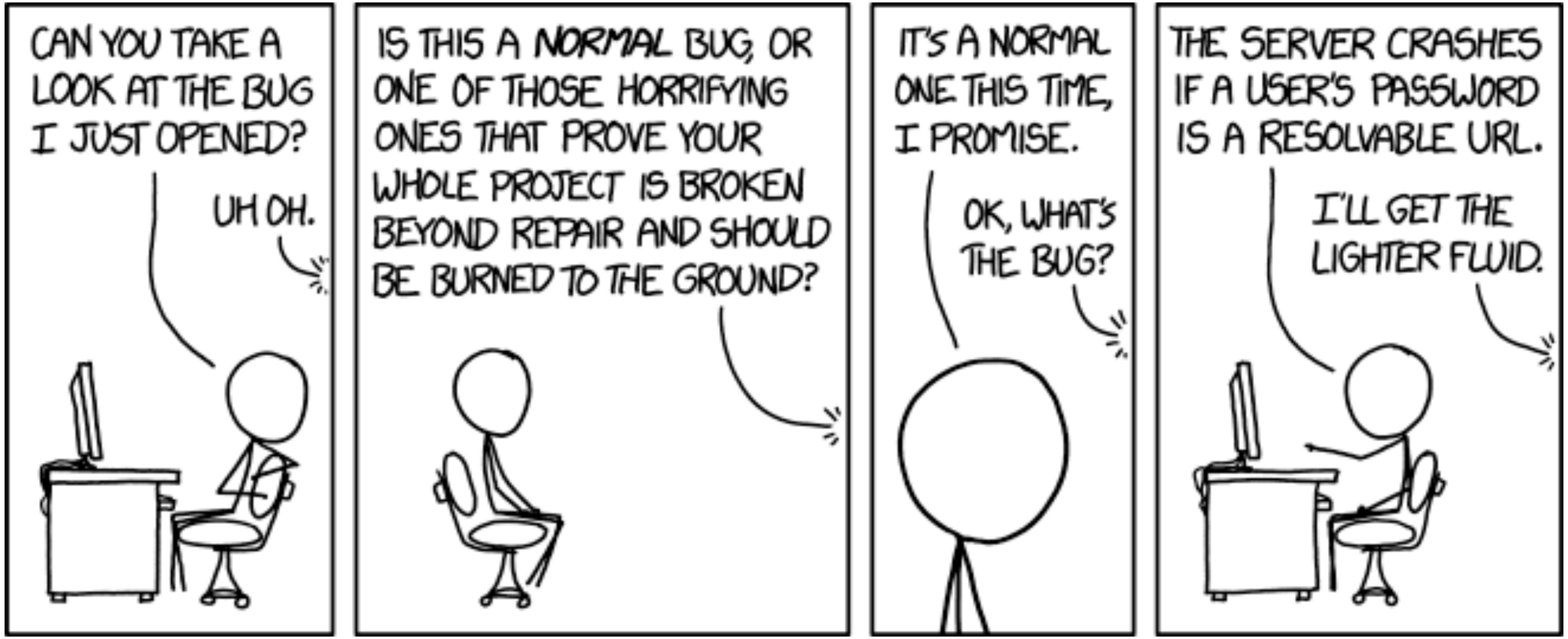
PT2

Unit06: Modultests & JUnit

Sven Köhler
Hasso-Plattner-Institut
2018-06-05



[https://xkcd.com/1700/]

SOUND HORN
STOP
Mahindra
4H 7909

Wie erreichen wir **Vertrauen** in die **funktionale Korrektheit** von **Software**?

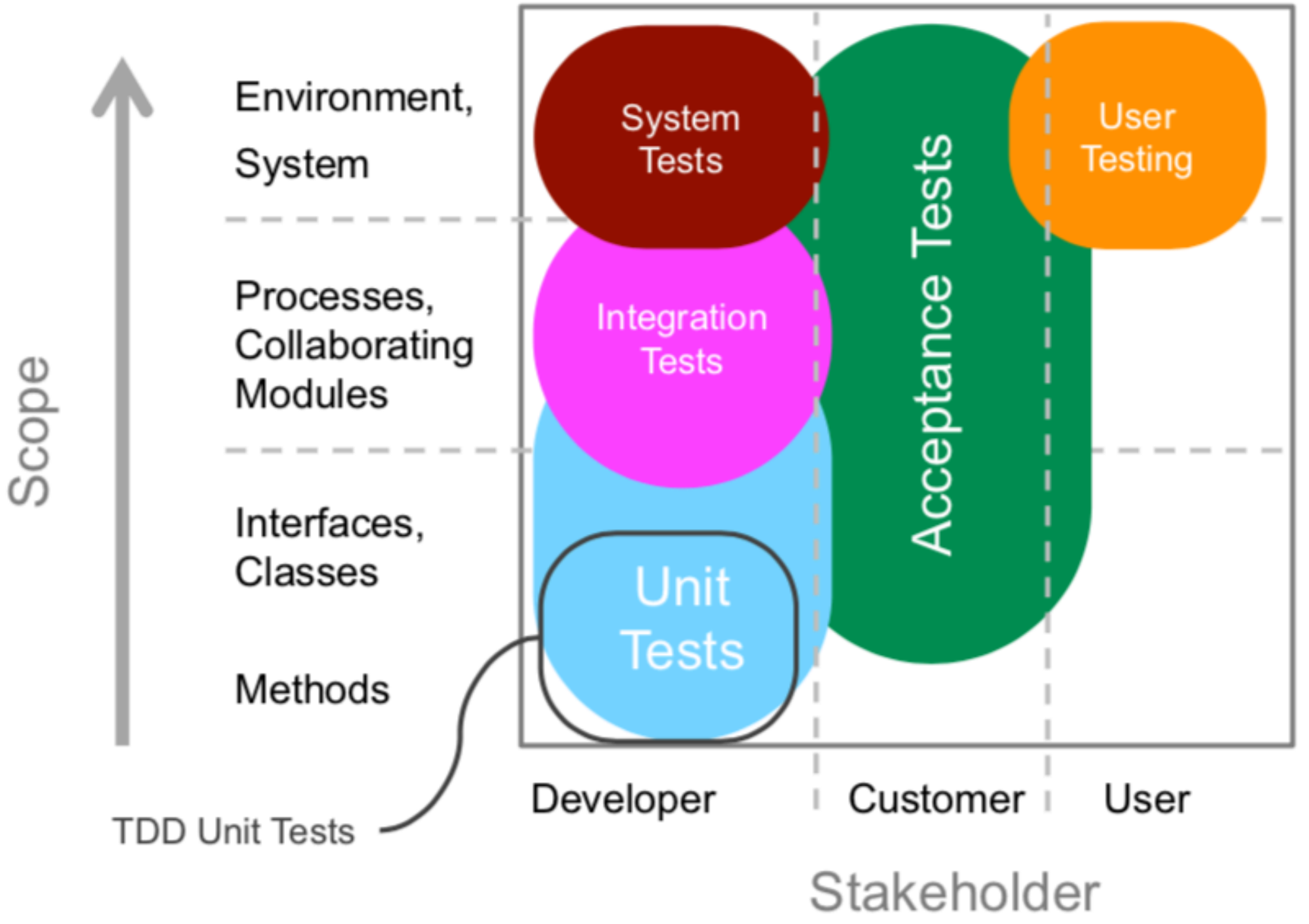
(schon vor der Auslieferung)

```
$ echo -ne '8 7 2 3 4 9 -1' | java Minimum
received: ['2']
expected: ['2']
$ echo -ne '42 1337 21 4 5 -2 3' | java Minimum
received: ['4']
expected: ['4']
$ echo -ne '-3' | java Minimum
received: ['-3']
expected: ['']
```

```
class AccountTest {

    public static void main(String[] args) {
        Account[] list = Account.open("passwd");
        assert list[0].UID == 5316;
        assert list[list.length-1].user_name.equals("Steffen Kensy");
        // ...
```

Test Classifications



PT2

Unit06: Modultests & JUnit

Sven Köhler
Hasso-Plattner-Institut
2018-06-05

1

Grundlagen & Terminologie

Wie erreichen wir Vertrauen in die funktionale **Korrektheit** von **Modulen**?

Übereinstimmung mit (informaler) Spezifikation

kleine, testbare Einheit (Funktion, Klasse)

UUT :: Unit under Test vs. SUT :: System under Test

(Tests sind **kein** Beweis der Abwesenheit von Fehlern.)

Testfall (test case) :: Elementarer Softwaretest

- Vorbedingungen / Umgebungen (**fixtures**)
- Eingaben
- Erwartete Ausgaben
- Erwartete Nachbedingungen

Testziel (test purpose) :: Definition der zu testenden Eigenschaft

Testsammlung (test suite) :: Zum Modul gehörende Testfälle

assertions

Tests sollen **Zusicherungen** über einzelne Eigenschaften eines Modul/System bestätigen.

Testergebnis (verdict) :: Eigenschaft erfolgreich getestet?

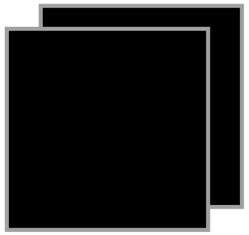
green
pass

red
fail

Andere Ergebnisse:

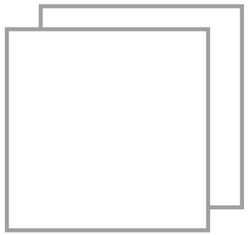
- Unbestimmt (inconclusive)
- Unerwartet bestanden/erwartet nicht bestanden (xpass, xfail)

Test != **Benchmark**



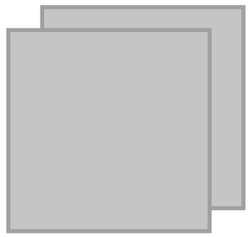
black box

- Testet gegen Spezifikation, ohne Wissen über Implementierung
- Nutzt öffentliche Modulschnittstellen
- Deckt verletzte Anforderungen auf



white box

- Orientiert sich an Quelltext des zu testenden Moduls bekannt
- Testfälle sollen mögliche Kontrollflüsse, Wertebereiche abdecken
- Deckt Fehler im Code auf



grey box

- Quelltext bekannt; Tests auf öffentliche Operationen beschränkt
- …

Mit black box tests sind nicht alle möglichen Kontrollflüsse bekannt.

```
if (cond1)
    A1();
else if (cond2)
    A2();
else
    A3;

if (cond3)
    A2();
```

Zustand von Reihenfolge abhängig (vgl. Floating-point Normalisierung)

Aber auch mit white box tests sind 100% Pfadabdeckung (code coverage) schwer.

Fehler, die aus Kombination von Modulen entstehen sind mit Modultests nicht erkenntlich

-> Integration Tests, System Tests, …

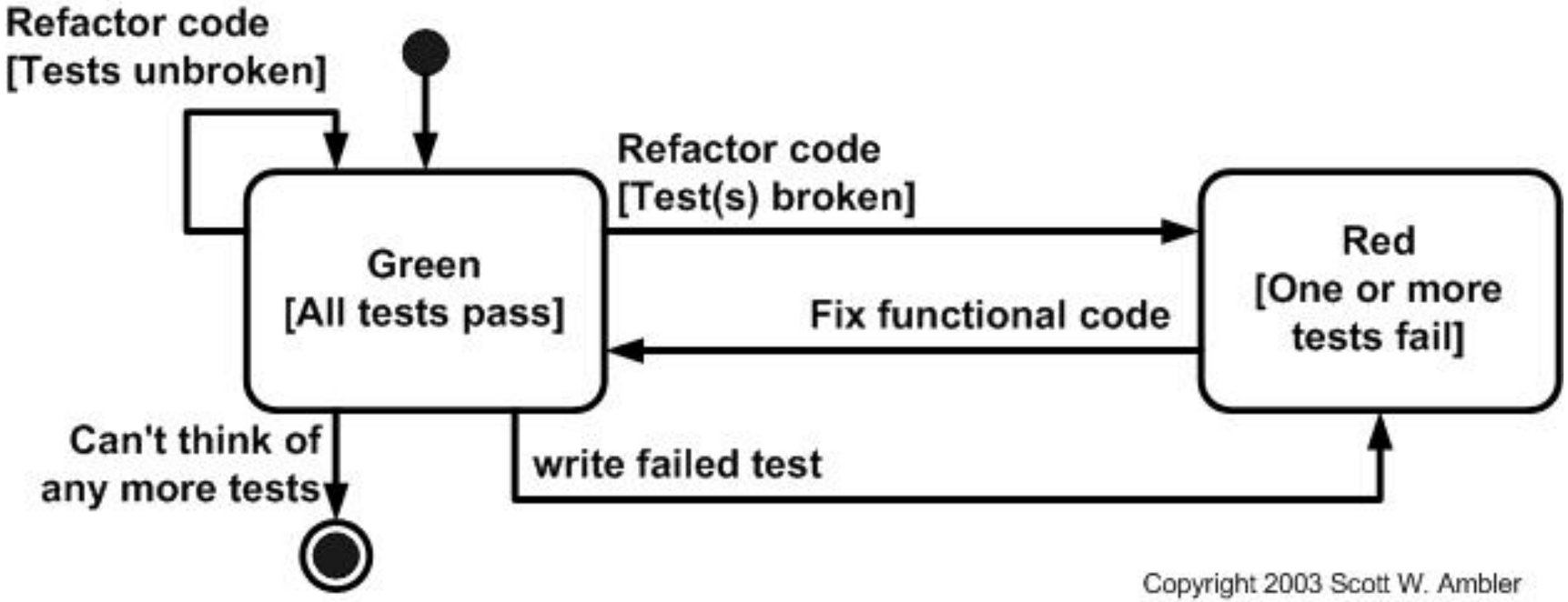
Automatisiertes Testen ::
regelmäßige, automatische Abwicklung aller Testfälle
(z.B. bevor Änderungen in Quelltextverwaltung aufgenommen)

Auswahl der Testfälle:
- Ableitung aus Spezifikation
- Nach Code Coverage
- Hinzufügen eines neuen Testfalls bei gefundenen Fehlern
- Bildung von Äquivalenzklassen, testen eines Repräsentanten
- *(was ihr zwischendurch mit REPLs/printf/… ausprobiert)*

Motivation: Einmal behobene Fehler sollen nicht wieder auftreten.
Besonders wichtig bei Refactoring ("Umstrukturierung") des Codes.

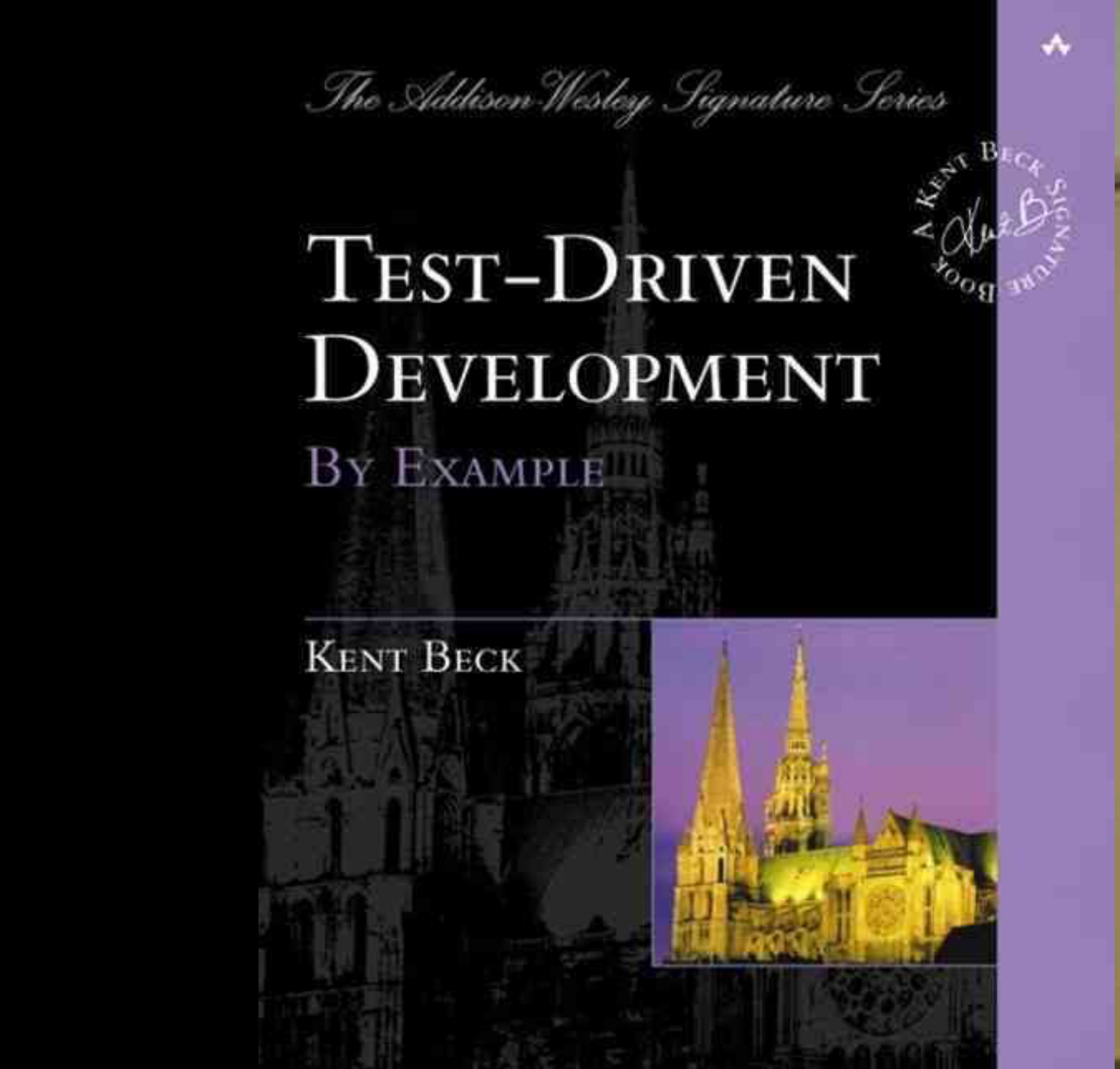
Automatisierte Tests führen empirisch zu besserem Design, höherer Verständlichkeit und leichterer Anpassbarkeit von Software.

… lasst uns einen Entwicklungsprozess daraus bauen.

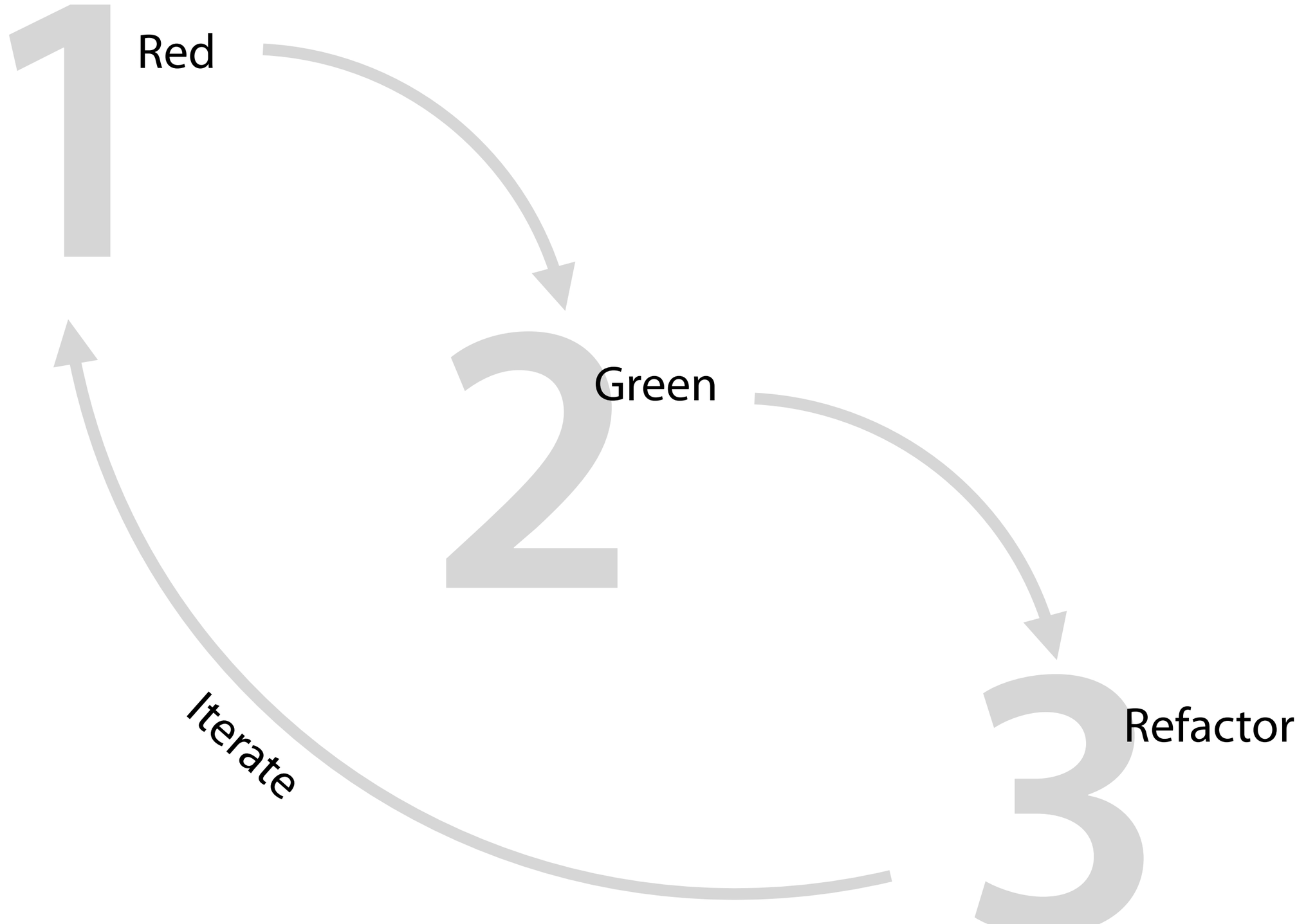


Copyright 2003 Scott W. Ambler

Test-Driven Development


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

Kent Beck



Mantra

TEST FIRST!

1. Quellcode für Klasse und Variablen/Methoden/Konstruktoren anlegen, sodass sich die Compilationseinheit übersetzen lässt
2. Schreibe die API-Dokumentation für Methoden/Konstruktoren erstellen; welche Parameter, Rückgaben, Ausnahmen sind nötig?
3. API an einem Beispiel testen – ist Klasse wie intendiert benutzbar?
4. Implementieren einer Testklasse
5. Implementieren der Programmlogik (Business-Logik)
6. Gibt es durch die Implementierung neue Dinge, die ein Testfall testen sollte? Wenn ja, Testfall erweitern.
7. Tests ausführen und Schritt 5 wiederholen bis alles fehlerfrei läuft

TDD

erzwingt Nachdenken beim Entwurf

verringert Notwendigkeit großer,
teuer Änderungen im Nachhinein (früh Scheitern)

Erleichtert API-Dokumentation

so oft wie möglich testen
besonders:
- vor und nach größeren Designänderungen
- vor Aufnahme in Versionsverwaltung
- Auf build-Server -> continuous integration

S
J
R
Py
C
x

Unit

3

Modultests mit JUnit

JUnit ::
Java-Klassenbibliothek zu **Modultests** aus der xUnit-Familie; basierte ursprünglich auf Kent Becks SUnit (Smalltalk).

Enthält Kommandozeilentools zum Ausführen einer Testsuite.
(junit.jar muss im CLASSPATH enthalten sein).

Integriert in vielen IDEs (Wizards zum Testerstellen, grafische Abarbeitung)

Testfälle bestehen aus drei Schritten:

1. Szenario aufbauen
2. Zu testende Methoden(-kombinationen) gerufen
3. Verdict mit spezifischer API des Testframeworks gefällt

Assertions („stimmt-es-dass“-Methoden) ::
Vergleichen den gewünschten Zustand mit dem tatsächlichen Zustand und Lösen bei Konflikt eine Ausnahme aus (Ausführung dieses Testfalles unterbrochen)

Testframework muss Tests laufen lassen und im Fehlerfall Meldung geben
Test-Runner – kann generisch aufgebaut sein

```java
import static org.junit.Assert.*;
import org.junit.*;

public class MyExampleTest {

    @Before
    public void setUp() throws Exception {
        // ...
    }

    @Test
    public void testCaseA() {
        // ...
        assertXXX(/* ... */);
    }

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

Klasse. Durch import static Klassenmethoden verfügbar

test case

fixture

test (keine Rückgabe)

assertion aus org.junit.Assert.assertXXX

JUnit4 Basisbeispiel

@Annotation ::
Benannte Anmerkungen, Metadaten und Compiler-Steuerung.

Bsp aus java.lang.

```
@Deprecated
public class A {}
```

Oder: @Override, @SuppressWarnings

Können selbst mit @interface definiert werden.
Anforderungen: Beschränkte Rückgabetypen, keine Exceptions, keine Rekursion.

Zur Laufzeit auswertbar (Reflections), oder als Compilerplugin (**Annotation Processor**) zur Compilezeit.

Tests sollen voneinander **unabhängig** sein.
Fixtures kümmern sich um **Seiteneffekte**

@BeforeClass - Klassenmethode, vor allen Tests einmal ausgeführt
@Before - Methode vor jedem Test ausgeführt.
@After - Methode nach jedem Test ausgeführt.
@AfterClass - Klassenmethode, nach allen Tests einmal ausgeführt
@Ignore - Deaktivierter Test (z.B. für unfertige Typen)
@Category - Erlaubt Tests kategorisiert auszuführen

@Test(timeout = 500) - Zeitbeschränkung in Millisekunden

Bsp03/
StringUtilsTest.java

Historisch (bis JUnit3) ohne Annotations, sondern

- Klassen mussten von `junit.framework.TestCase` abgeleitet werden.
- test-Methoden mussten `test...` heißen
- Methoden `setUp`/test.../`tearDown` wurden per Reflection gefunden.

Andere xUnit-Frameworks arbeiten noch nach diesem Prinzip

Ausführung von Tests:

- Exemplar der Testfall-Klasse anlegen
- Aufruf von .run()

Bsp03/
StringUtilsTest3.java

Seit Ende 2017: JUnit05 (abwärtskompatibel zu JUnit4 und JUnit3)

[https://howtoprogram.xyz/2016/08/10/junit-5-vs-junit-4/]

Vordefinierte Methoden in Klasse org.junit.Assert:

```
assertEquals(erwartet, beobachtet)
assertTrue(bedingung), assertFalse(bedingung)
assertNull(object), assertNotNull(object)
assertSame(erwartet, beobachtet), assertNotSame(e, b)
...
```

Scheitern Sie, bricht der Test ab.

Spezifikation fordert Auftreten von **Ausnahmebedingungen**.

`fail("Fehlermeldung")`, lässt Test abbrechen

```
try {
   StringUtils.reverse(null);
   fail("reverse(null) should throw IllegalArgumentException");
} catch (IllegalArgumentException e) {}
```

Problem: Was passiert mit anderen Fehlern (erwartet und un-erw.)?

```
@Test(expected = IllegalArgumentException.class)
public void testReverseException() {
        StringUtils.reverse( null );
}
```

Problem: Tests müssen oft private Elemente verschiedener Pakete erreichen
Lösung 1: Schnittstelle öffentlich machen (= Verletzung Information hiding)

Lösung 2:
Testklassen für jede Klasse im gleichen Paket erzeugen; mehre Testklassen zur (hierarchischen) **Testsuite** zusammenfassen und gemeinsam ausführen.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({ TestJunit1.class ,TestJunit2.class })
// ...
```

Bsp02/
JunitTestSuite.java

ende