Effizientes Programmieren

Praktikum

Testgetriebene Entwicklung (13.06.2017)

Christopher Pietsch

UNIVERSITÄT SIEGEN

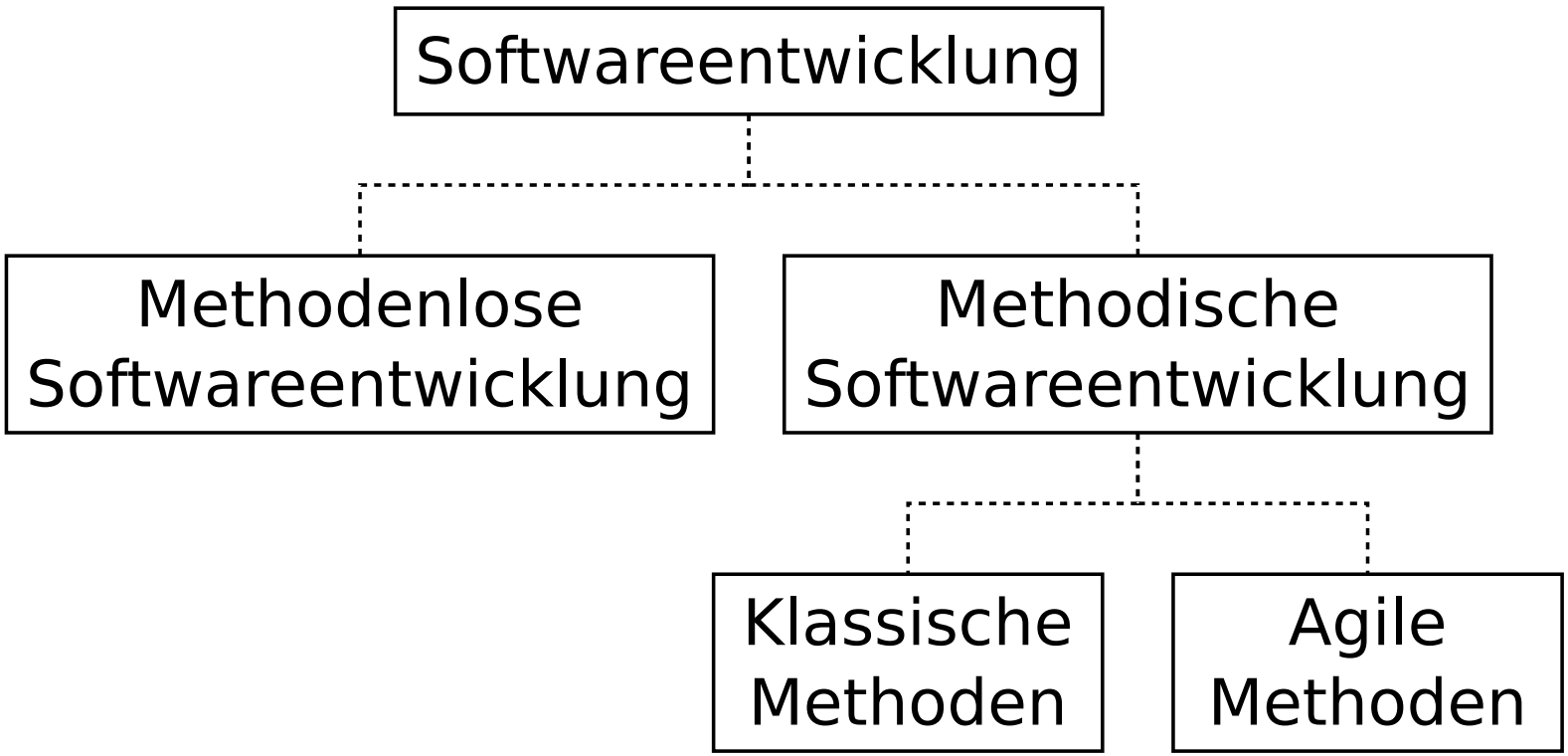
Praktische Informatik

Agenda

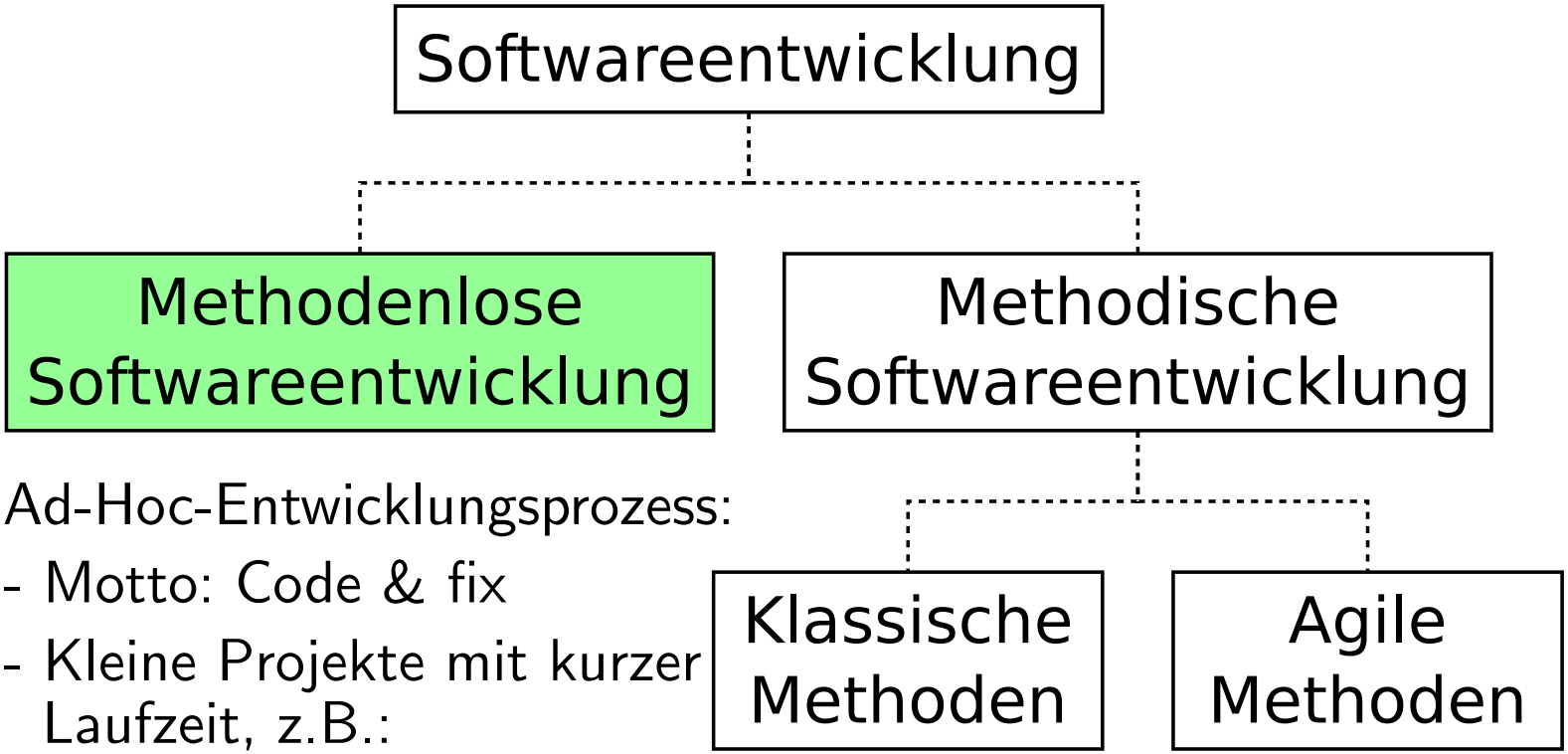
Teil 1

Klassifikation von Softwareentwicklungsmethoden

Klassifikation von Softwareentwicklungsmethoden



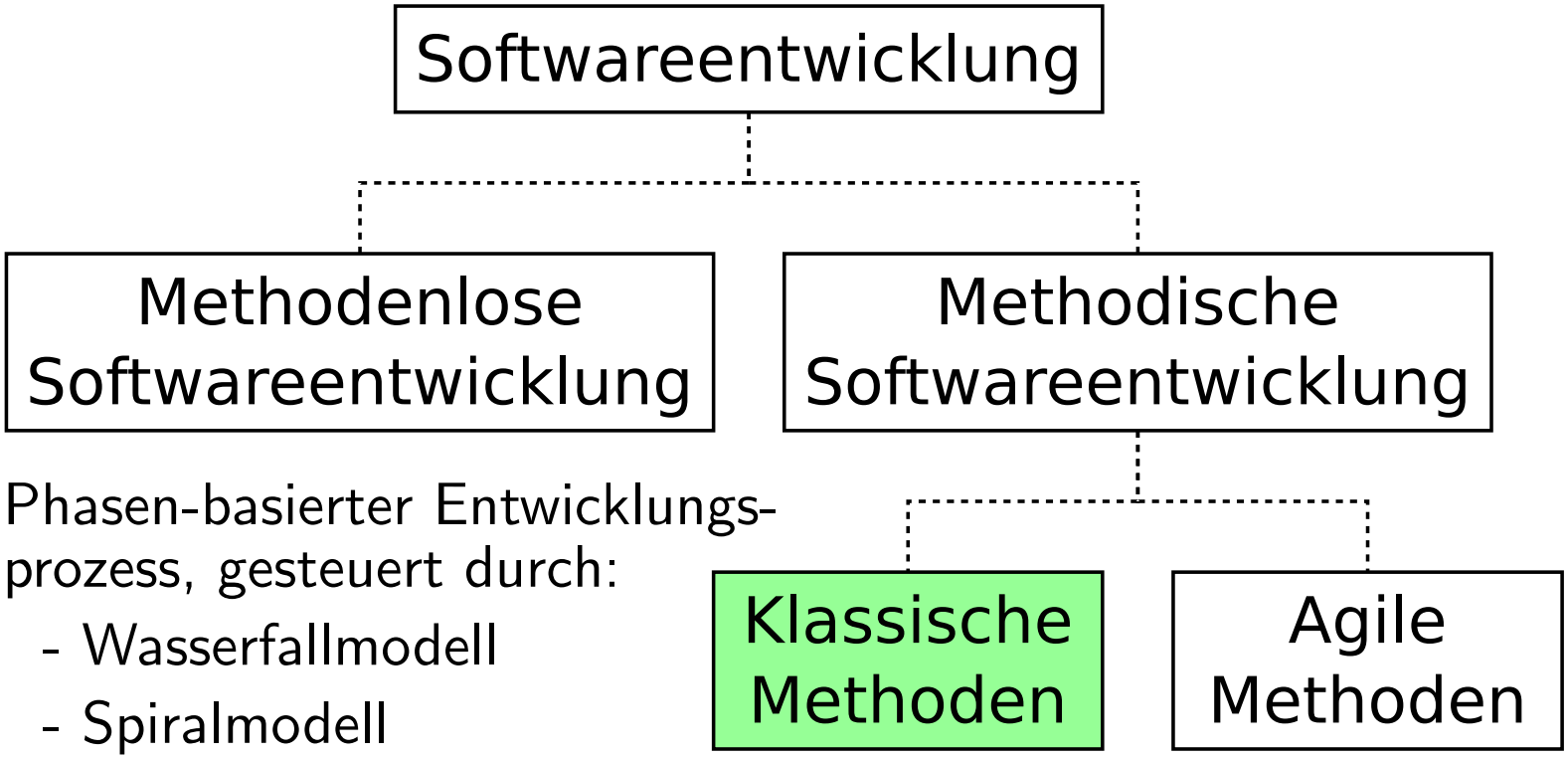
Klassifikation von Softwareentwicklungsmethoden



Ad-Hoc-Entwicklungsprozess:

- Motto: Code & fix
- Kleine Projekte mit kurzer Laufzeit, z.B.:
  - (Wegwerf-)Prototyp
  - Proof-of-Concept
  - Demo

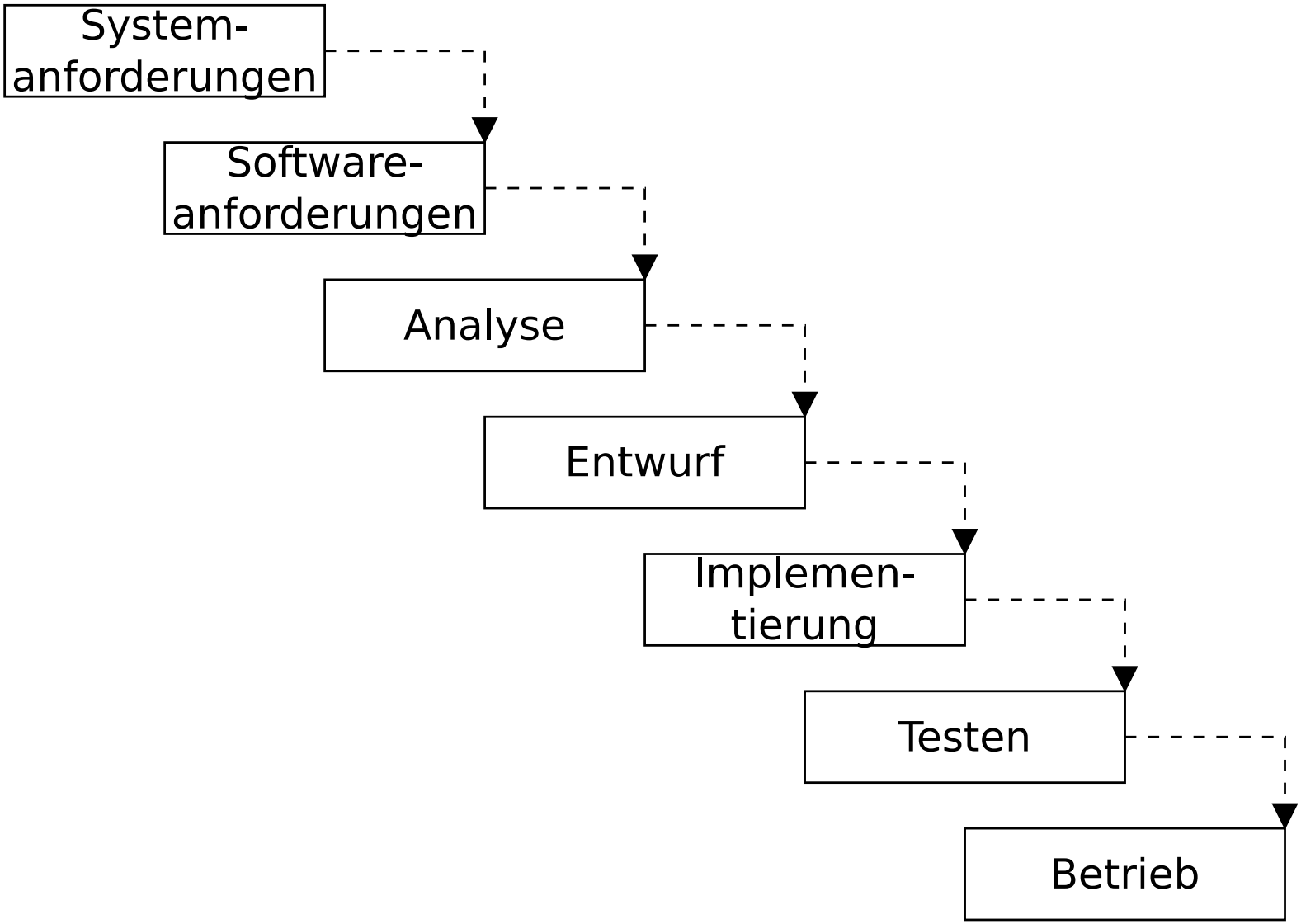
Klassifikation von Softwareentwicklungsmethoden



Phasen-basierter Entwicklungsprozess, gesteuert durch:
- Wasserfallmodell
- Spiralmodell
- V-Modell

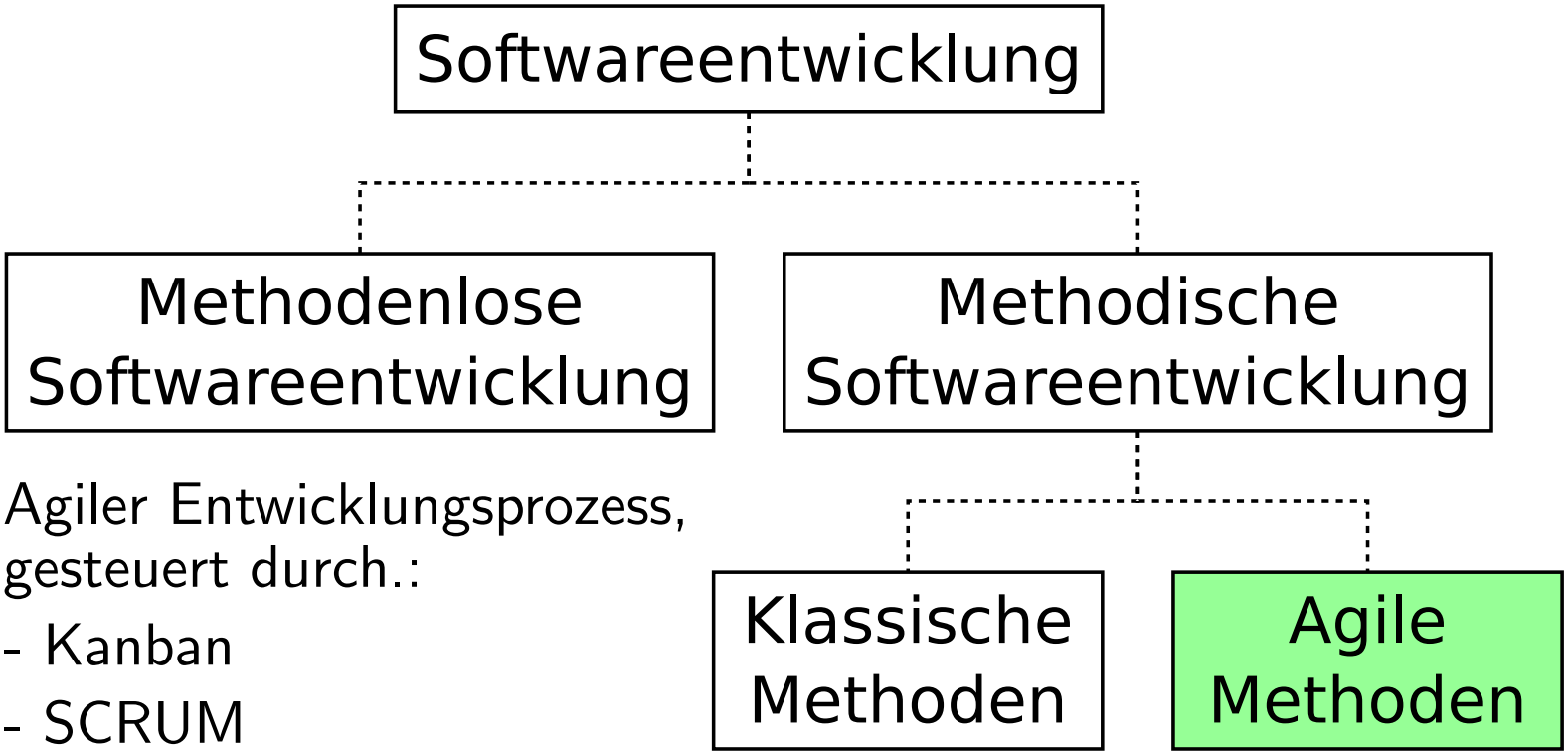
- Projekte mit (relativ) stabilem Umfeld

Beispiel: Wasserfallmodell nach Royce



[Roy87]

Klassifikation von Softwareentwicklungsmethoden



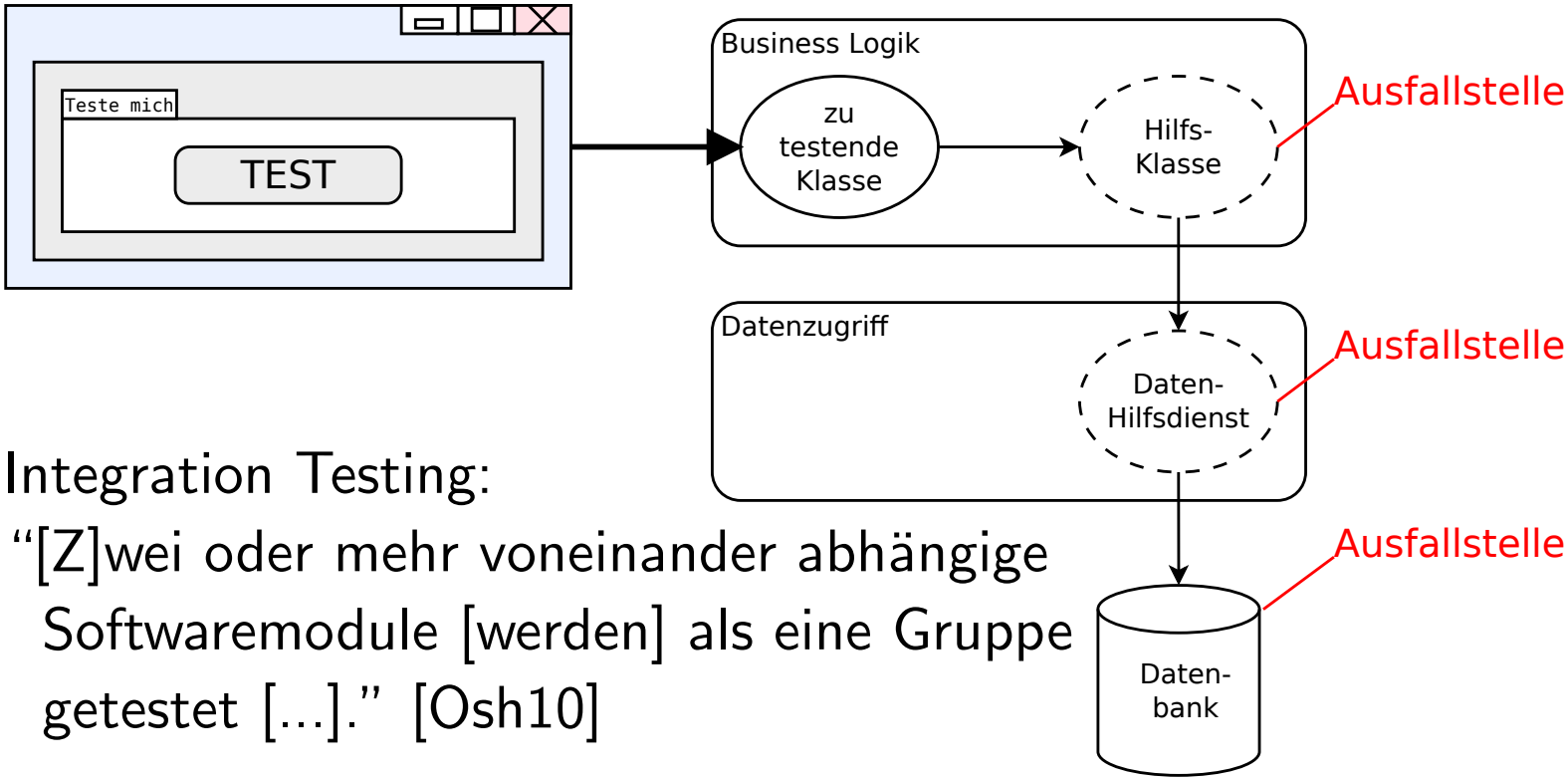
Agiler Entwicklungsprozess, gesteuert durch.:

- Kanban
- SCRUM
- Extreme Programming Praktiken:
  - Paar-Programmierung
  - Testgetriebene Entwicklung
  - ...
  - Retrospektive

Teil 2

Testgetriebene Entwicklung

Integration Tests



Integration Testing:

"[Z]wei oder mehr voneinander abhängige Softwaremodule [werden] als eine Gruppe getestet [...]." [Osh10]

(Gute) Unit Tests

Definition: Unit

"Eine *Unit* ist eine Methode oder Funktion [...]", die *logischen Code* enthält. [Osh10]

Logischer Code:

- Bedingte Anweisungen: `if`, `switch`
- Schleifen: `while`, `for`
- Operationen: `+`, `-`, ...
- Ausnahmen: `throws`

Bemerkung: *Getters/Setters* nur dann, wenn sie logischen Code enthalten.

(Gute) Unit Tests

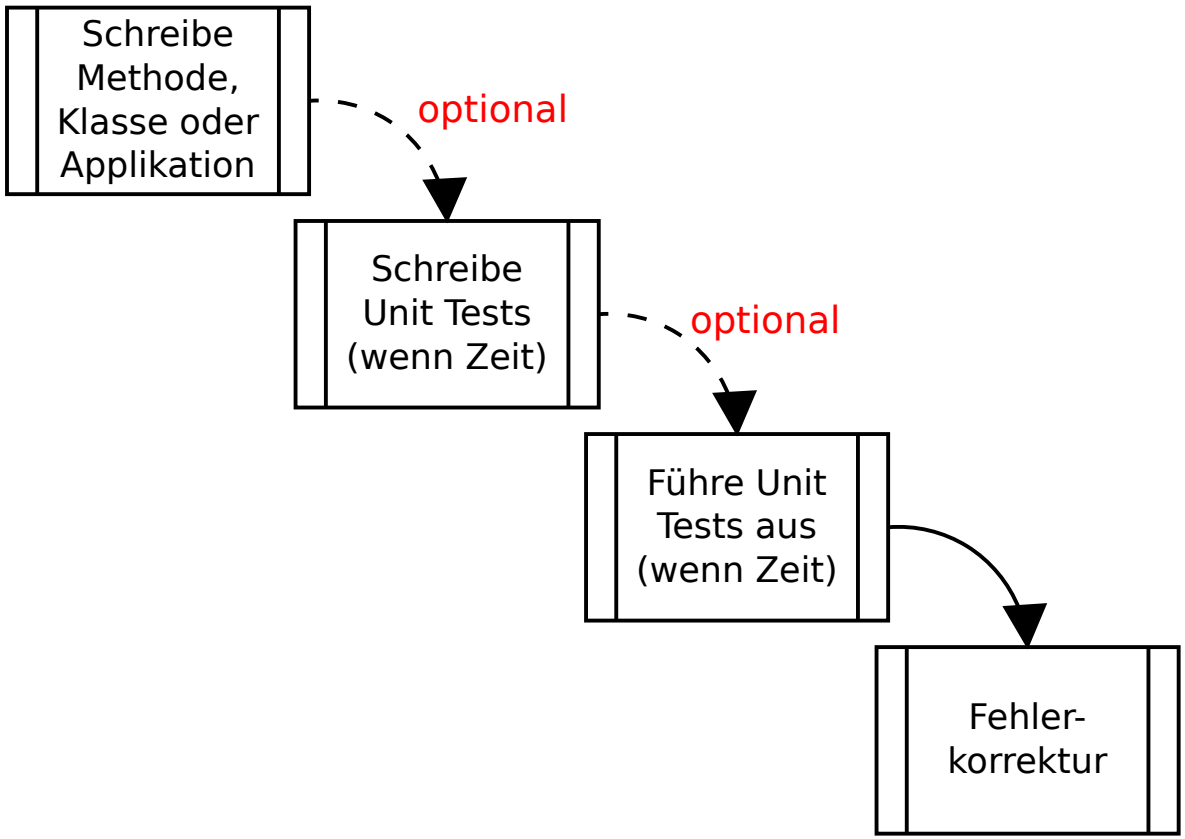
Definition: Unit Test

"Ein *Unit Test* ist ein automatisiertes Stück Code, das eine zu testende Methode oder Klasse aufruft und dann einige Annahmen über das logische Verhalten dieser Methode oder Klasse prüft[...].
[Ein guter Unit Test] kann einfach geschrieben und schnell ausgeführt werden. Er ist vollständig automatisiert, vertrauenswürdig, lesbar und wartbar." [Osh10]

(Gute) Unit Tests

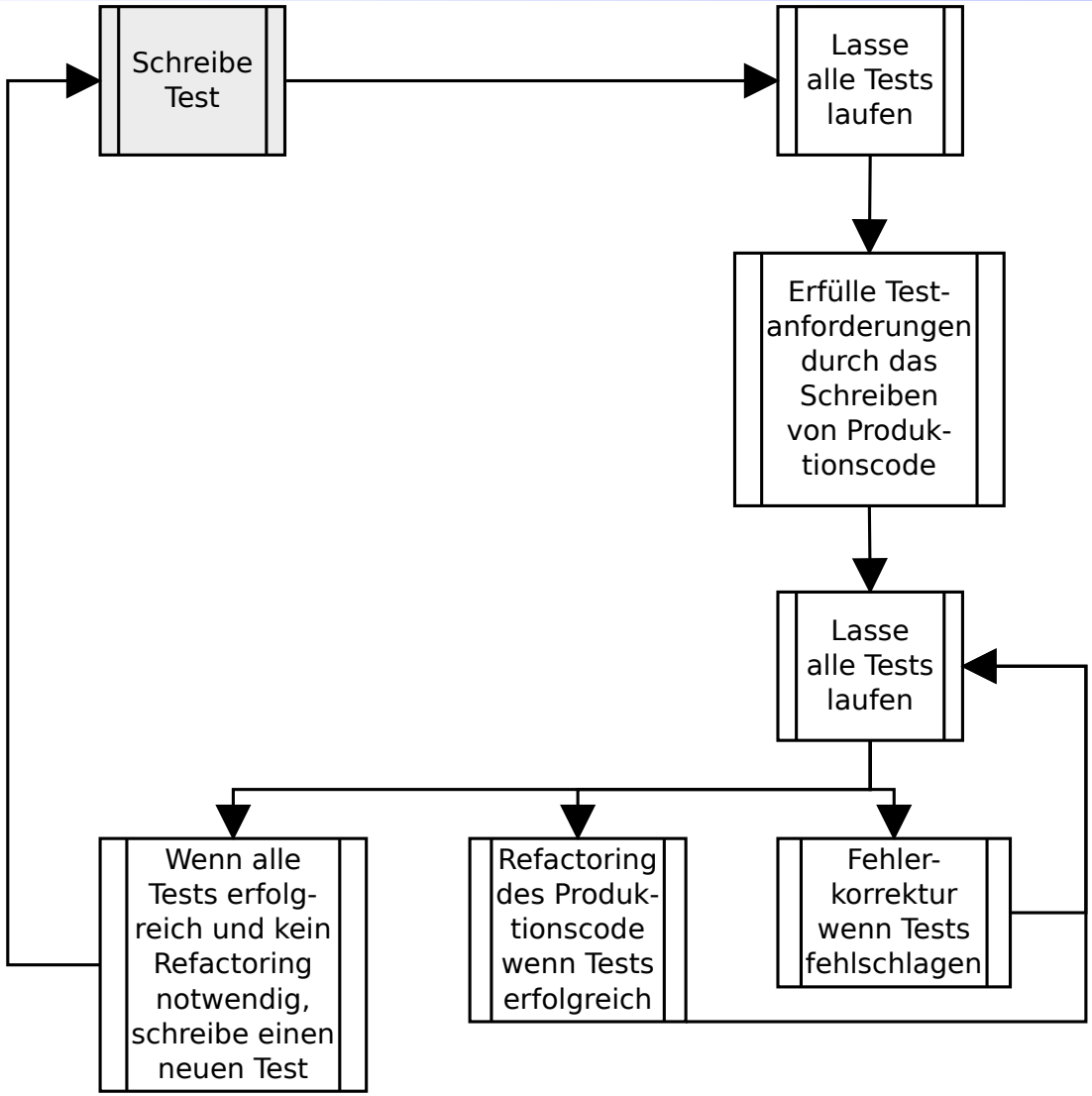
- Automatisierbar und jederzeit wiederholbar → **Accidental Bugging**, **Refactorings**
- Einfach zu implementieren → erhöhter Abdeckungsgrad
- Jeder sollte den Test ausführen können → **Collective Ownership**
- Ausführung erfolgt auf Knopfdruck → keine Konfiguration
- Läuft schnell → erhöhte Ausführungsrate

Traditionelles Unit Testen



[Osh10]

'Test-First' Ansatz

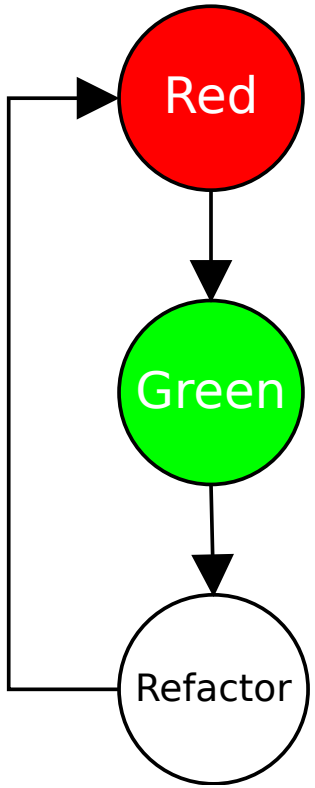


[Osh10]

'Test-First' Ansatz

'Red-Green-Refactor' Mantra:

1. Schreiben einer Testmethode die fehlschlägt: **Fail** (Testen des Tests)
2. Schreiben des Produktionscode: **Pass**
3. Überarbeitung des Codes



Unit Testing Framework

Framwork unterstützt den Entwickler beim

1. Schreiben von Tests → stellt Klassenbibliothek bereit
2. Ausführen von Tests → ermöglicht die automatisierte Ausführung eines oder mehrerer Tests auf Knopfdruck
3. Auswerten von Tests → gibt diverse Informationen bzgl. der ausgeführten Tests zurück

xUnit Frameworks:

- CppUnit: C++
- **JUnit**: Java
- NUnit: .Net
- ...

Teil 3

JUnit

JUnit Testframework

- Annotation-basiertes Testframework für Java
- Projektseite: `http://junit.org`
- Vollständige Integration in Eclipse IDE

Struktur von JUnit Testprojekten

- **Testfall** (Test Case): *Klasse* bestehend aus
  - **Setup-Methode**: *Methode* zum initialisieren der Testumgebung
  - **Testmethode**: *Methode* zum Ausführen der zu testenden Funktionalität und zum Prüfen der jeweiligen Annahmen
  - **Teardown-Methode**: *Methode* zum Aufräumen der Testumgebung
- **Test Suite**: *Klasse* zum Ausführen mehrerer Testfälle

JUnit Konventionen

- **Testfall**: `[Klassenname]`*`Test`*`.java`
- **Testmethode**: *`test`* `[Methodenname()]`
  Anmerkung: u.U. gibt es mehrere Testmethoden für eine Klassenmethode:

  `[Methodenname]_[Testbedingungen]_[ErwartetesVerhalten]()`

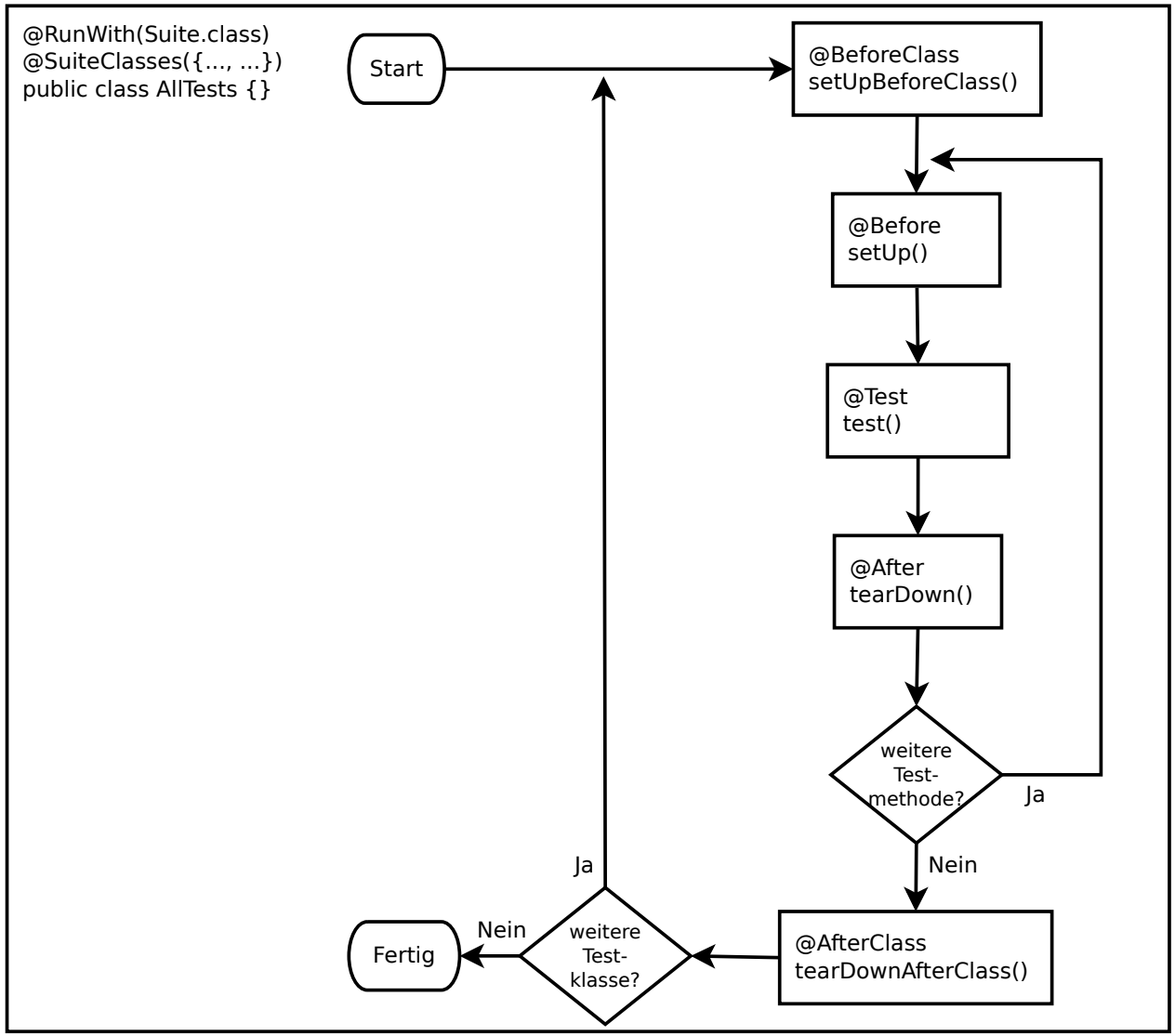
JUnit Annotationen

- `@BeforeClass`: Annotierte Methode wird vor jedem Testfall aufgerufen
- `@AfterClass`: Annotierte Methode wird nach jedem Testfall aufgerufen
- `@Before`: Annotierte Methode wird vor jeder Testmethode aufgerufen
- `@After`: Annotierte Methode wird nach jeder Testmethode aufgerufen

JUnit Annotationen

- `@Test`: Annotierte Methode führt den zu testenden Code aus und überprüft das erwartete Verhalten
- `@Test(expected = Exception.class)`: Annotierte Methode führt zu testenden Code aus und erwartet eine entsprechende Ausnahme
- `@Test(timeout=100)`: Annotierte Methode führt zu testenden Code aus und überprüft deren Ausführungsdauer
- `@Ignore`: Annotierte Methode wird ignoriert

Testphasen von JUnit Testprojekten



JUnit Assertions

Klasse `Assert` (siehe `http://junit.sourceforge.net/javadoc/org/junit/Assert.html`):

- `fail(String)`: Methode schlägt immer fehl
- `assertTrue([message], boolean condition)`: Prüft ob die Bedingung wahr ist
- `assertFalse([message], boolean condition)`: Prüft ob die Bedingung falsch ist
- ...

Beispiel: Calculator.java


```
1  public class Calculator {
2
3    private Stack<Long> mem;
4
5    public Calculator() {...}
6
7    public long add(long n1, long n2) {
8      mem.push(n1+n2);
9      return mem.peek();
10   }
11
12   public long sub(long n1, long n2) {...}
13
14   public long mult(long n1, long n2) {...}
15
16   public long div(long n1, long n2) throws ArithmeticException {...}
17
18   public void clear() {...}
19
20   public Stack<Long> getMem() {...}
21 }
```

Beispiel: CalculatorTest.java


```
1  public class CalculatorTest {
2
3    Calculator calculator;
4
5    @BeforeClass
6    public static void setUpBeforeClass() throws Exception {...}
7
8    @AfterClass
9    public static void tearDownAfterClass() throws Exception {...}
10
11   @Before
12   public void setUp() throws Exception {
13     calculator = new Calculator();
14   }
15
16   @After
17   public void tearDown() throws Exception {
18     calculator = null;
19   }
```

Beispiel: CalculatorTest.java


```
1   @Test
2   public void testAdd() {
3     calculator.add(5, 5);
4     assertTrue(calculator.getMem().peek() == 10);
5   }
6
7   @Test
8   public void testSub() {...}
9
10  @Test
11  public void testMult() {...}
12
13  @Test
14  public void testDiv() {...}
```

Beispiel: CalculatorTest.java


```
1   @Test(expected = ArithmeticException.class)
2   public void testDiv_ArithmeticException() {
3     calculator.div(5, 0);
4   }
5
6   @Test
7   public void testClear() {...}
8 }
```

Teil 4

Stubs und Mocks

Problem: Externe Abhängigkeiten

Definition: Externe Abhängigkeit

"Eine *externe Abhängigkeit* ist ein Objekt [...], mit dem der zu testende Code interagiert und" welches außerhalb der Kontrolle des Entwickler liegt. [Osh10]

Beispiele:

- Dateisystem
- Threads
- Zeit
- ...

Lösung: Stubs

Definition: Stub

"Ein *Stub* ist ein kontrollierbarer Ersatz für eine vorhandene Abhängigkeit [...] im System. Durch die Verwendung von Stubs kann der Code getestet werden, ohne die Abhängigkeit direkt handhaben zu müssen." [Osh10]

Lösung: Mocks

Definition: Mock

"Ein *Mock-Objekt* ist ein nachgeahmtes Objekt im System, das entscheidet, ob ein Unit Test funktioniert hat oder fehlgeschlagen ist. Es macht dies, indem es verifiziert, ob das zu testende Objekt mit dem nachgeahmten Objekt wie erwartet interagiert." [Osh10]

Stub vs. Mock

Stub:

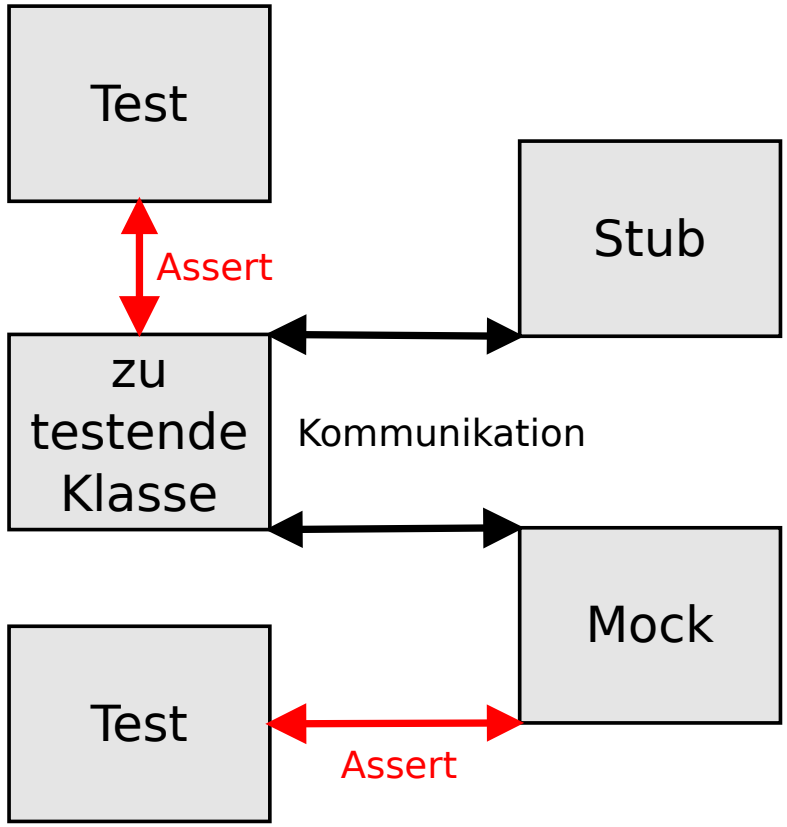
- Ersetzt Abhängigkeit beim *zustandsbasierten* Testen
- Kann nicht fehlschlagen

Mock:

- Ersetzt Abhängigkeit beim *interaktionsbasierten* Testen
- kann fehlschlagen

$\Rightarrow$ Integration durch Indirektionsschicht

Stub vs. Mock



vgl. [Osh10]

Teil 5

Aufgabe

Aufgabenstellung

Implementierung einer digitalen Geldbörse (Money Bag) nach dem 'Test-First' Ansatz

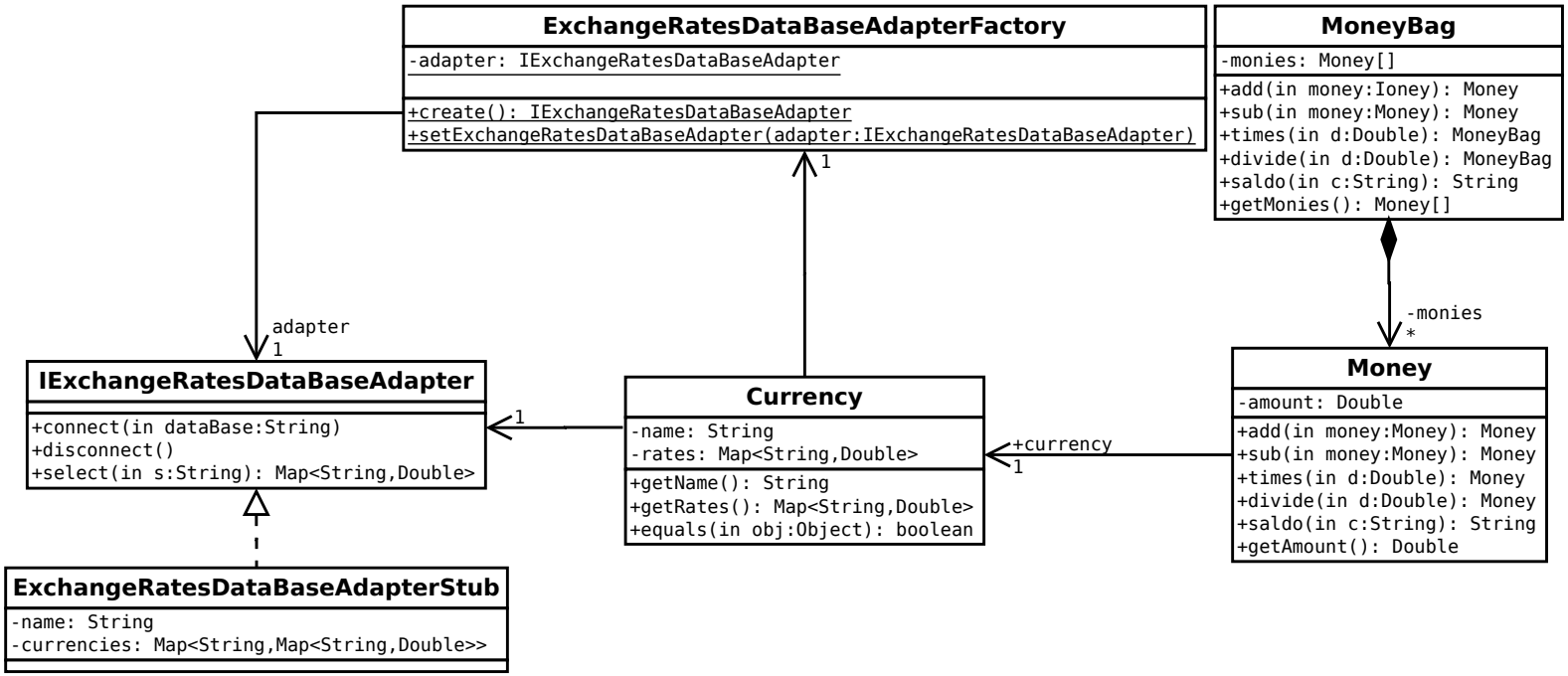


Abbildung: Projekt: `2015s_pep_junit_money`

Teil 6

Literatur

Literatur

Quellen:

[Osh10] R. Osherove.
The Art of Unit Testing - Deutsche Ausgabe.
mitp, 2010.

[Roy87] W. W. Royce.
Managing the Development of Large Software Systems: Concepts and Techniques.
In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

Weiterführende Literatur:

- `http://junit.org/`
- `http://www.vogella.com/tutorials/JUnit/article.html`