andrena OBJECTS

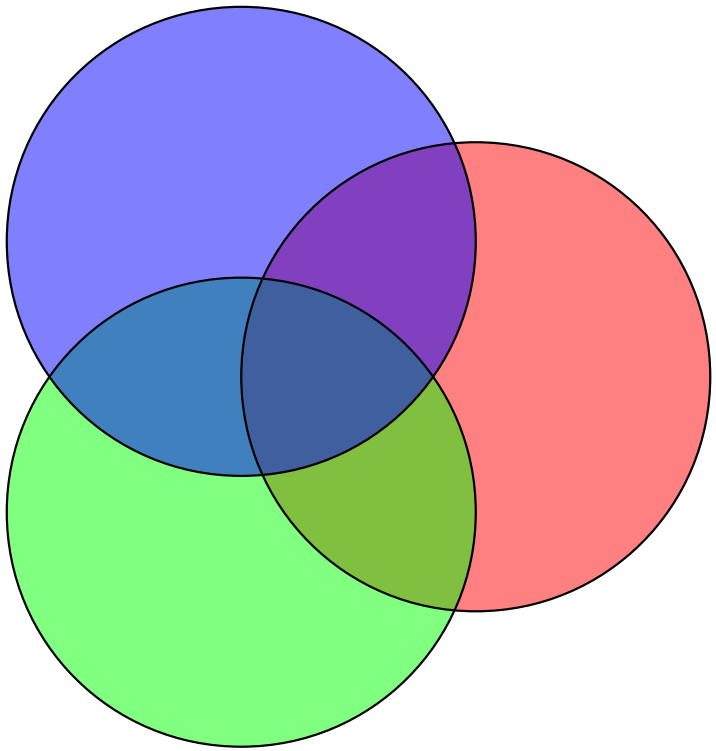
Experts in agile software engineering

Bessere Tests mit JUnit 4.x

Marc Philipp

1. Juli 2010

Über JUnit



Kent Beck:

A programmer-oriented testing framework for Java.

David Saff:

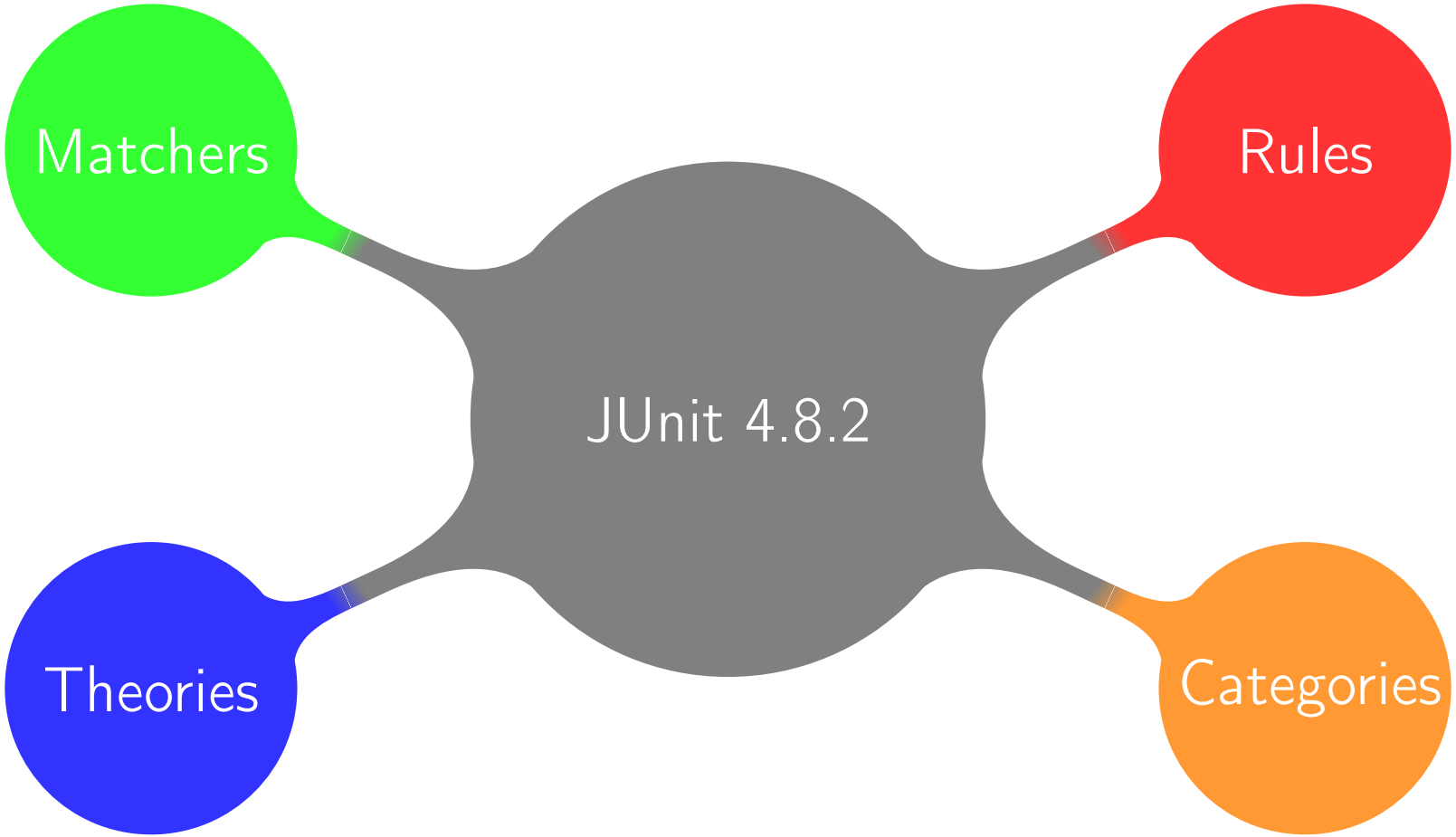
JUnit is the intersection of all possible useful Java test frameworks, not their union.

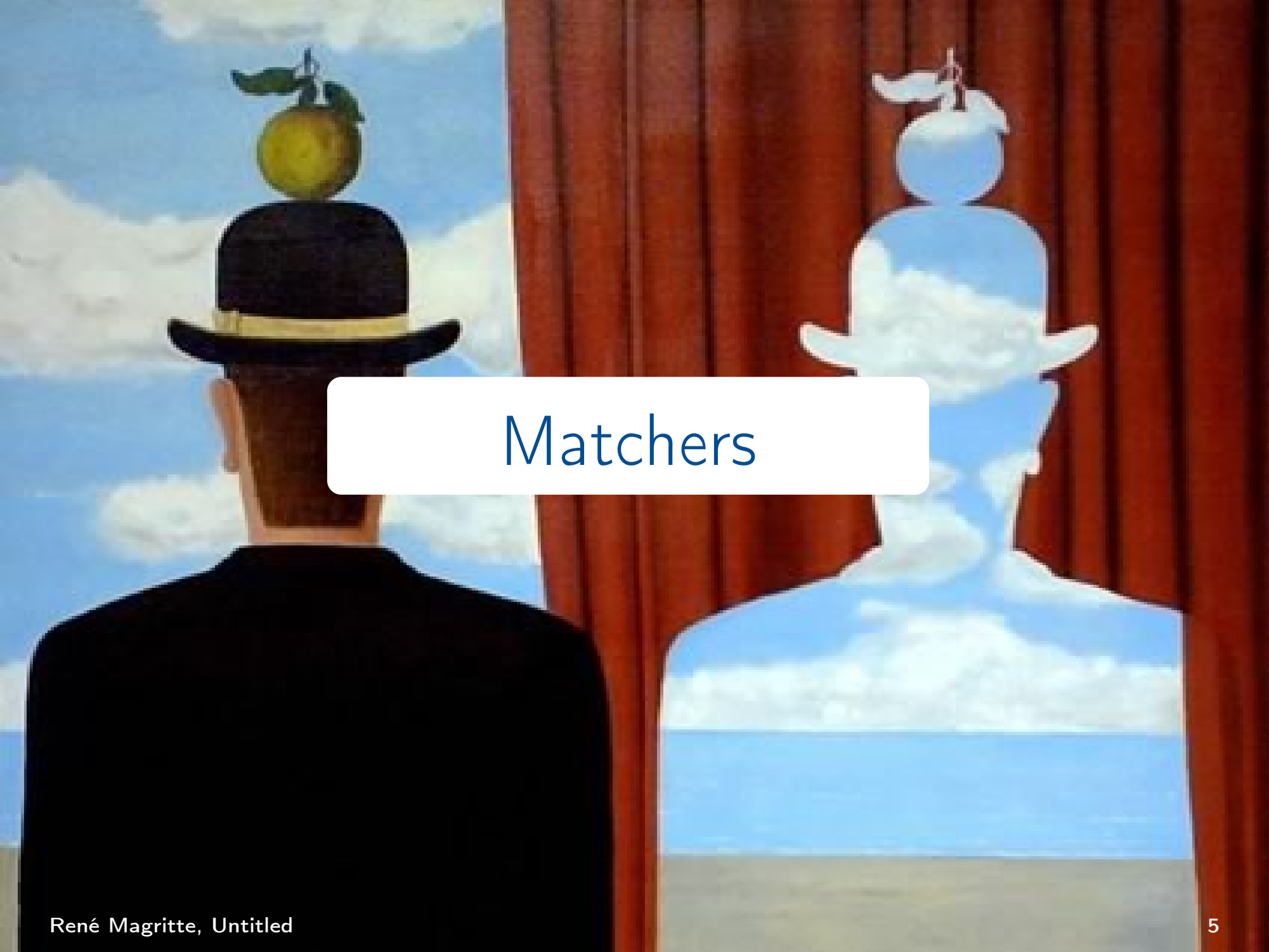
Nicht nur für Unit Tests!

Warum ein Vortrag über JUnit?

- Jeder kennt JUnit.
- Wirklich?
- Jeder meint, JUnit zu kennen.
- Ich meinte es auch.
- Seit Version 4.0 hat sich einiges getan!

Neue Features seit Version 4.0





Matchers

René Magritte, Untitled

Neue Assertion: `assertThat(...)`

- Neue `Assert`-Methoden:

```
<T> void assertThat(T actual, Matcher<T> matcher)
<T> void assertThat(String reason, T actual, Matcher<T> matcher)
```

- Parameter:

reason	Zusätzliche Beschreibung für den Fehlerfall (optional)
actual	Tatsächlicher Wert
matcher	Hamcrest-Matcher überprüft tatsächlichen Wert

Bessere Lesbarkeit

```
import static org.hamcrest.CoreMatchers.is;

public class BetterReadability {

   @Test public void withoutMatchers() {
      assertEquals(2, 1 + 1);
   }

   @Test public void withMatchers() {
      assertThat(1 + 1, is(2));
   }
}
```

- Reihenfolge stimmt
- Häufig besser lesbar als herkömmliche Assertions

Matcher kombinieren

Matcher lassen sich einfach kombinieren:

```
assertThat(1 + 1, is(not(3)));
assertThat(1 + 1, both(greaterThan(1)).and(lessThan(3)));
```

Aussagekräftige Fehlermeldungen

Herkömmliche Zusicherung ohne Beschreibung

```
  assertFalse(asList(1, 2, 3).contains(2));
```

Ergebnis:

```
AssertionError: at de.andrena.junit...
```

Neue Zusicherung mit Matcher

```
  assertThat(asList(1, 2, 3), not(hasItem(2)));
```

Ergebnis:

```
AssertionError:
Expected: not a collection containing <2>
     got: <[1, 2, 3]>
```

Vordefinierte Matcher

- Vielzahl vordefinierter Matcher:

Core	`is`, `not`, `allOf`, `anyOf`, `nullValue`, ...
Strings	`containsString`, `startsWith`, `endsWith`, ...
Collections	`hasItem`, `hasItems`, `isIn`, `empty`, `hasSize`, ...

- Bei JUnit ist nur ein kleiner Teil mit dabei:
  - org.hamcrest.CoreMatchers
  - org.junit.matchers.JUnitMatchers
- Hamcrest bietet weitere Matcher (`hamcrest-all.jar`).
- Darüber hinaus lassen sich eigene Matcher definieren.

Ein eigener Matcher

Implementierung

```
public class IsEmptyCollection extends TypeSafeMatcher<Collection<?>> {

   @Override protected boolean matchesSafely(Collection<?> collection) {
      return collection.isEmpty();
   }

   @Override public void describeTo(Description description) {
      description.appendText("empty");
   }

   @Factory public static Matcher<Collection<?>> empty() {
      return new IsEmptyCollection();
   }
}
```

Ein eigener Matcher

Benutzung

```
public class CollectionMatchersTest {

   @Test public void isEmpty() {
      TreeSet<String> set = new TreeSet<String>();
      assertThat(set, new IsEmptyCollection());   // direkter Aufruf
      assertThat(set, IsEmptyCollection.empty()); // über @Factory-Methode
      assertThat(set, empty());                   // mit statischem Import
      assertThat(set, is(empty()));               // syntactic sugar
   }
}
```

Viele Matcher – Viele Klassen

Problem

- Vielzahl von Klassen mit zu importierenden statischen Methoden
- Manuelle Konfiguration der IDE notwendig
- . . . bei allen Entwicklern

Lösung

Generierung einer eigenen Matcher-Bibliothek

Generierung einer Matcher-Bibliothek

1. Konfiguration in XML

```
<matchers>
   <factory class="org.hamcrest.core.Is"/>
   <factory class="de.andrena.junit.hamcrest.IsEmptyCollection"/>
</matchers>
```

2. Aufruf von `org.hamcrest.generator.config.XmlConfigurator`
3. Alle mit `@Factory` annotierten Methoden werden in einer generierten Klasse zusammengefasst:

```
public class Matchers {
   public static <T> Matcher<? super T> is(T param1) {
      return org.hamcrest.core.Is.is(param1);
   }
   /* ... */
   public static Matcher<Collection<?>> empty() {
      return de.andrena.junit.hamcrest.IsEmptyCollection.empty();
   }
}
```

Fazit: Matchers

- Assertions lassen sich oft eleganter formulieren
- Manchmal sind die alten Assertion-Methoden klarer

Problem

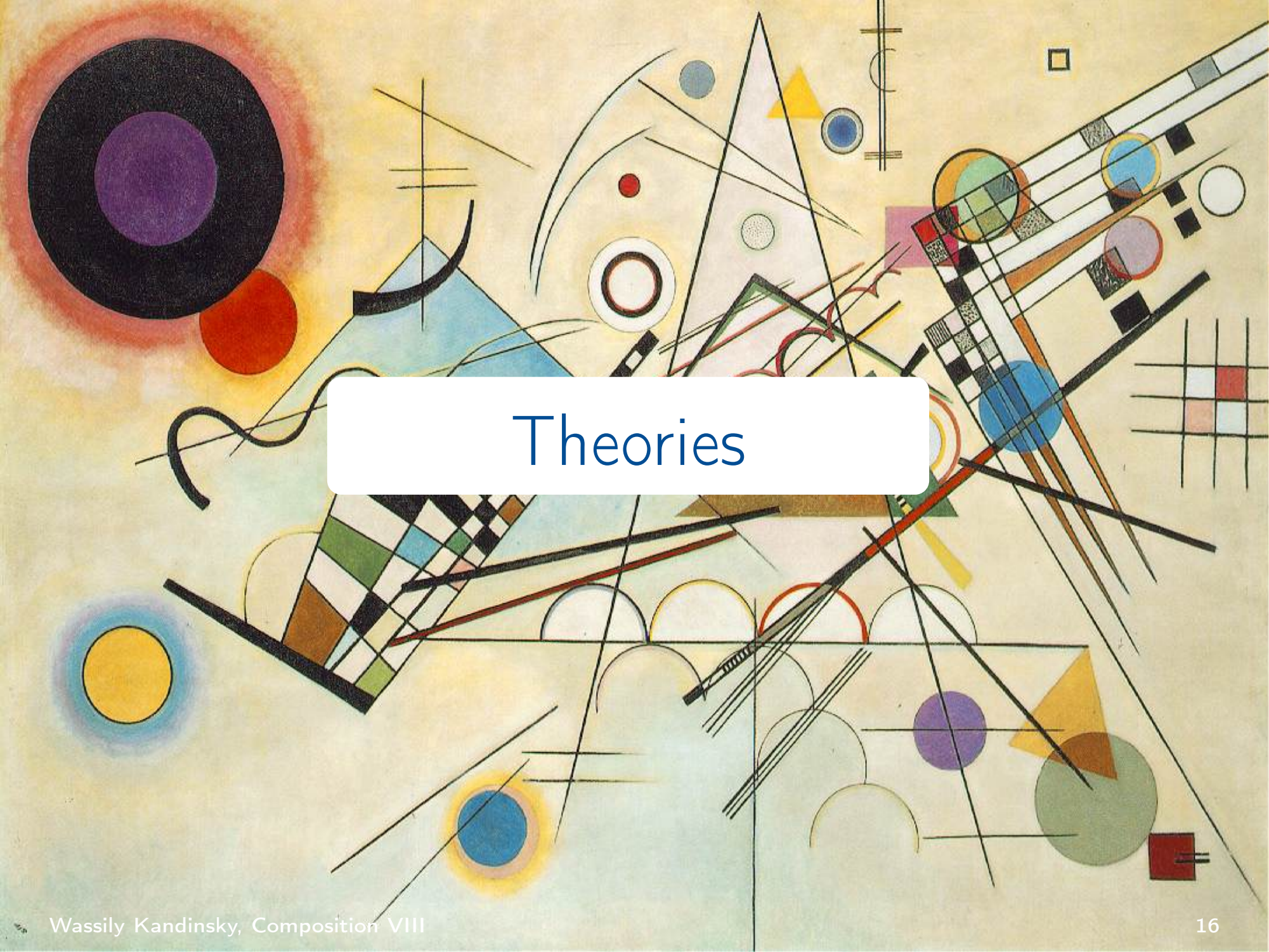
Javas Typsystem macht einem oft einen Strich durch die Rechnung

- Boxing notwendig bei primitiven Typen

```
assertThat(1 + 1, is(2))
```

- Mangelnde Typinferenz

```
assertThat(new TreeSet<String>(), Matchers.<String>empty());
```



Theories

Parametrisierte Tests

- Seit JUnit 4.0 gibt es den `Parameterized` Test Runner.
- Ermöglicht Parametrisierung von Testklassen über Konstruktor
- Argumente werden in einer statischen Methode angegeben:
  - `@Parameters`-Annotation
  - Rückgabetyp: `List<Object[]>`
- Für jedes Paar von Testmethode und Argument wird eine Instanz der Testklasse angelegt und die Testmethode ausgeführt.
- Nützlich, wenn Berechnung mit vielen Eingabeparametern getestet werden soll.

Parametrisierte Tests

Beispiel: Fibonacci-Zahlen

```
@RunWith(Parameterized.class)
public class FibonacciParameterizedTest {

   @Parameters public static List<Object[]> data() {
      return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, { 2, 1 },
            { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 } });
   }

   private final int input, expected;

   public FibonacciParameterizedTest(int input, int expected) {
      this.input = input;
      this.expected = expected;
   }

   @Test public void test() {
      assertEquals(expected, Fibonacci.compute(input));
   }
}
```

Theories

- Neu seit Version 4.4
- Parametrisierte Testmethode mit Vorbedingungen (Assumptions)
  - `assumeThat()`
  - `assumeTrue()`
  - `assumeNotNull()`
- `@Theory` statt `@Test`
- Input für Testmethoden über `@DataPoint(s)`-Annotation

Theories

Beispiel: Fibonacci-Zahlen

```
import static de.andrena.junit.fibonacci.Fibonacci.compute;

@RunWith(Theories.class)
public class FibonacciTheories {

   @DataPoints public static int[] VALUES = { 0, 1, 2, 3, 4, 5, 6 };

   @Theory public void seeds(int n) {        // F(0) = 0,  F(1) = 1
      assumeTrue(n <= 1);
      assertEquals(n, compute(n));
   }

   @Theory public void recurrence(int n) {  // F(n) = F(n-1) + F(n-2)  für  n > 1
      assumeTrue(n > 1);
      assertEquals(compute(n - 1) + compute(n - 2), compute(n));
   }
}
```

Theories

Beispiel: Verschiedene @DataPoints

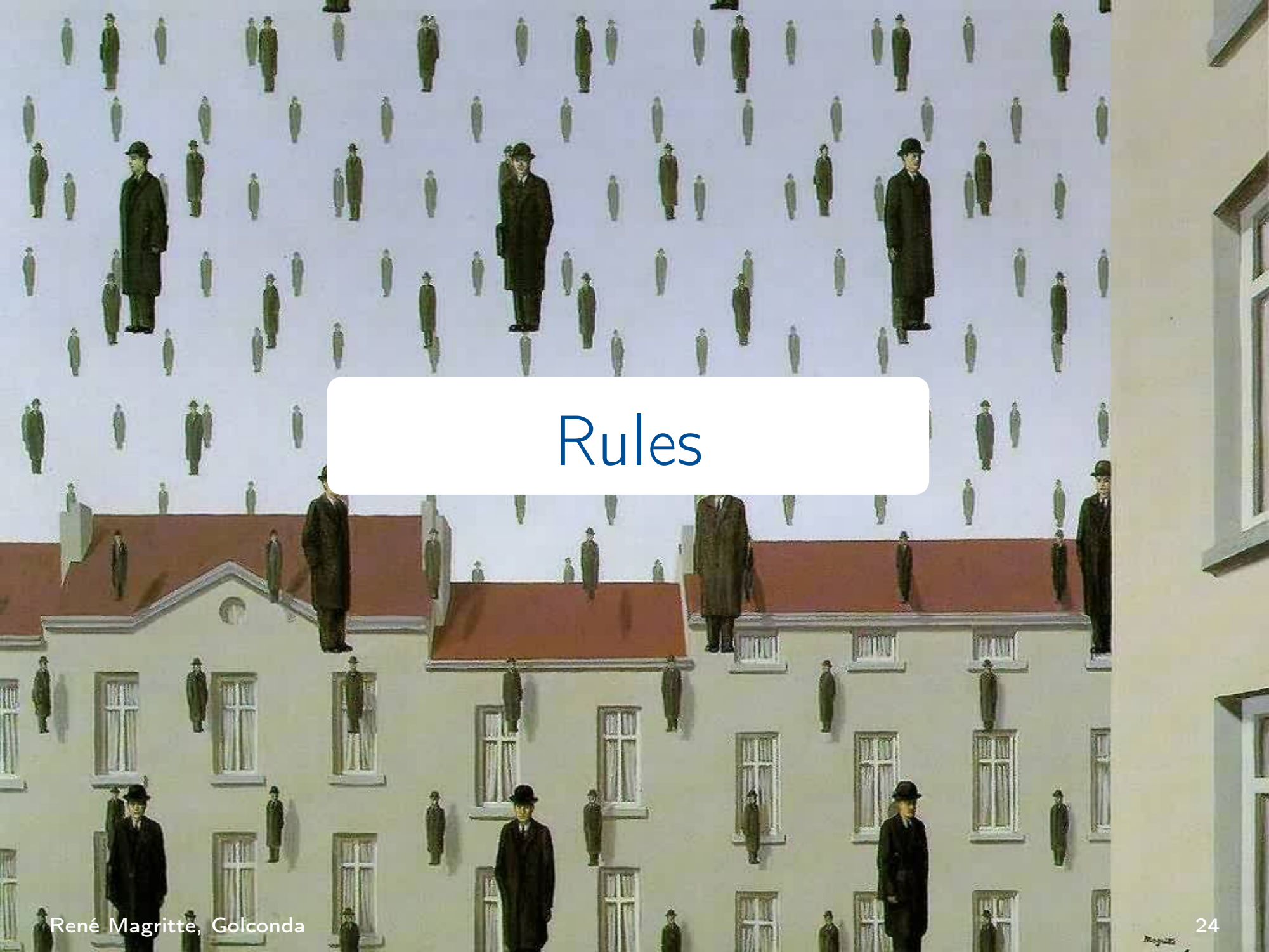
```
@RunWith(Theories.class)
public class DifferentDataPoints {

   @DataPoint public static String A = "a";
   @DataPoint public static String B = "b";
   @DataPoints public static int[] NUMBERS = { 1, 2, 3 };

   @Theory public void theory1(String string, int number) {
      /* ("a", 1), ("a", 2), ("a", 3), ("b", 1), ("b", 2), ("b", 3) */
   }
   @Theory public void theory2(String string) {
      /* ("a"), ("b") */
   }
   @Theory public void theory3(int number) {
      /* (1), (2), (3) */
   }
}
```

Parameterized Tests vs. Theories

- Definition von `@DataPoints` ist flexibler
  - Konstanten oder Methoden
  - Beliebig viele
  - Verschiedene Typen
- Parametrisierung der Test*methoden*
  - Kein Boilerplate-Code (Konstruktor + Instanzvariablen)
  - Verschiedene Parametertypen möglich
- `Theories` können alles, was `Parameterized` auch kann, aber mehr!

Herkömmliche Tests vs. Theories

- Herkömmliche Tests benutzen Beispiele:
  - Überprüfung des Verhaltens unter ausgewählten Eingaben
  - Beispiele sind (hoffentlich) charakteristisch
- Eine Theory verallgemeinert eine Menge von Tests:
  - Vorbedingung wird explizit angegeben
  - Sollte für alle Eingaben gelten, die Vorbedingungen erfüllen
- Eingabewerte können explizit angegeben werden oder automatisch erzeugt werden



Rules

René Magritte, Golconda

Was ist eine Rule?

Erweiterungsmechanismus für Ablauf der Testmethoden

Beispiele:

- Ausführung eigenen Codes vor bzw. nach jeder Testmethode
- Behandlung fehlgeschlagener Tests
- Überprüfung zusätzlicher Kriterien nach einem Tests
- ...

Beispiel: TemporaryFolder

Ohne Benutzung einer Rule

```
public class TemporaryFolderWithoutRule {
   private File folder;

   @Before public void createTemporaryFolder() throws Exception {
      folder = File.createTempFile("myFolder", "");
      folder.delete();
      folder.mkdir();
   }

   @Test public void test() throws Exception {
      File file = new File(folder, "test.txt");
      file.createNewFile();
      assertTrue(file.exists());
   }

   @After public void deleteTemporaryFolder() {
      recursivelyDelete(folder); // passt nicht mehr auf die Folie...
   }
```

Beispiel: TemporaryFolder

Unter Verwendung einer Rule

```
public class TemporaryFolderWithRule {

   @Rule public TemporaryFolder folder = new TemporaryFolder();

   @Test public void test() throws Exception {
      File file = folder.newFile("test.txt");
      assertTrue(file.exists());
   }

}
```

Was ist eine Rule?

Erweiterungsmechanismus für Ablauf der Testmethoden

Beispiele:

- Ausführung eigenen Codes vor bzw. nach jeder Testmethode
- Behandlung fehlgeschlagener Tests
- Überprüfung zusätzlicher Kriterien nach einem Tests
- ...

Beispiel: ExpectedException

Ohne Benutzung einer Rule

```
public class ExpectedExceptionWithoutRule {

   int[] threeNumbers = { 1, 2, 3 };

   @Test(expected = ArrayIndexOutOfBoundsException.class)
   public void exception() {
      threeNumbers[3] = 4;
   }

   @Test public void exceptionWithMessage() {
      try {
         threeNumbers[3] = 4;
         fail("ArrayIndexOutOfBoundsException expected");
      } catch (ArrayIndexOutOfBoundsException expected) {
         assertEquals("3", expected.getMessage());
      }
   }
}
```

Beispiel: ExpectedException

Unter Verwendung einer Rule

```
public class ExpectedExceptionWithRule {

   int[] threeNumbers = { 1, 2, 3 };

   @Rule public ExpectedException thrown = ExpectedException.none();

   @Test public void exception() {
      thrown.expect(ArrayIndexOutOfBoundsException.class);
      threeNumbers[3] = 4;
   }

   @Test public void exceptionWithMessage() {
      thrown.expect(ArrayIndexOutOfBoundsException.class);
      thrown.expectMessage("3");
      threeNumbers[3] = 4;
   }
}
```

Weitere vordefinierte Rules

ErrorCollector

Sammelt fehlgeschlagene Assertions innerhalb einer Testmethode und gibt am Ende eine Liste der Fehlschläge aus.

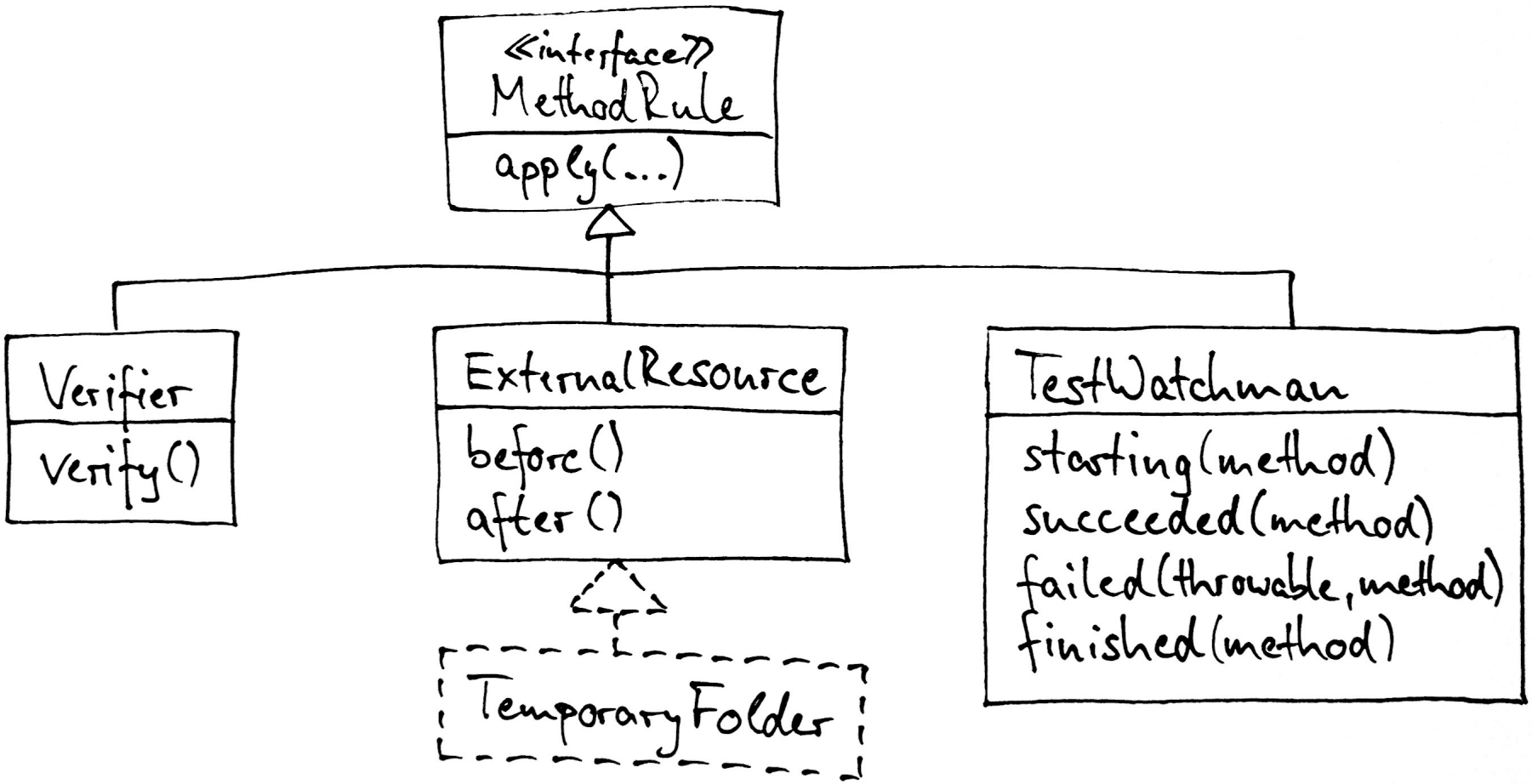
TestName

Merkt sich Namen der aktuell ausgeführten Testmethode und stellt ihn auf Anfrage zur Verfügung.

Timeout

Wendet gleichen Timeout auf alle Testmethoden einer Klasse an.

Schreib deine eigenen Regeln!



Eine eigene Regel

Implementierung

```
public class SystemProperty extends ExternalResource {
   private final String key, value;
   private String oldValue;
   public SystemProperty(String key, String value) {
      this.key = key;
      this.value = value;
   }
   @Override protected void before() {
      oldValue = System.getProperty(key);
      System.setProperty(key, value);
   }
   @Override protected void after() {
      if (oldValue == null) {
         System.getProperties().remove(key);
      } else {
         System.setProperty(key, oldValue);
      }
   }
}
```

Eine eigene Regel

Benutzung

```
public class SomeTestUsingSystemProperty {

   private static final String VALUE = "someValue";
   private static final String KEY = "someKey";

   @Rule public SystemProperty systemProperty = new SystemProperty(KEY, VALUE);

   @Test public void test() {
      assertThat(System.getProperty(KEY), is(VALUE));
   }
}
```

Vorteile von Regeln

Wiederverwendbarkeit

Ermöglichen häufig benötigten Code auszulagern.

Kombinierbarkeit

Beliebig viele Regeln in einem Test verwendbar

Delegation statt Vererbung

Helfen Testklassenhierarchien zu vermeiden!

Erweiterbarkeit

Eigene Regeln schreiben ist einfach.

Categories

Piet Mondrian, Composition with Gray and Light Brown

Tests in Kategorien einteilen

- Kategorie ist Klasse oder Interface:

```
public interface Slow {}
```

- Einzelner Test:

```
@Category(Slow.class)
@Test
public void test() {}
```

- Alle Tests einer Klasse:

```
@Category(Slow.class)
public class B {
   @Test
   public void c() {}
}
```

Tests in bestimmten Kategorien ausführen

- Herkömmliche Test-Suite:

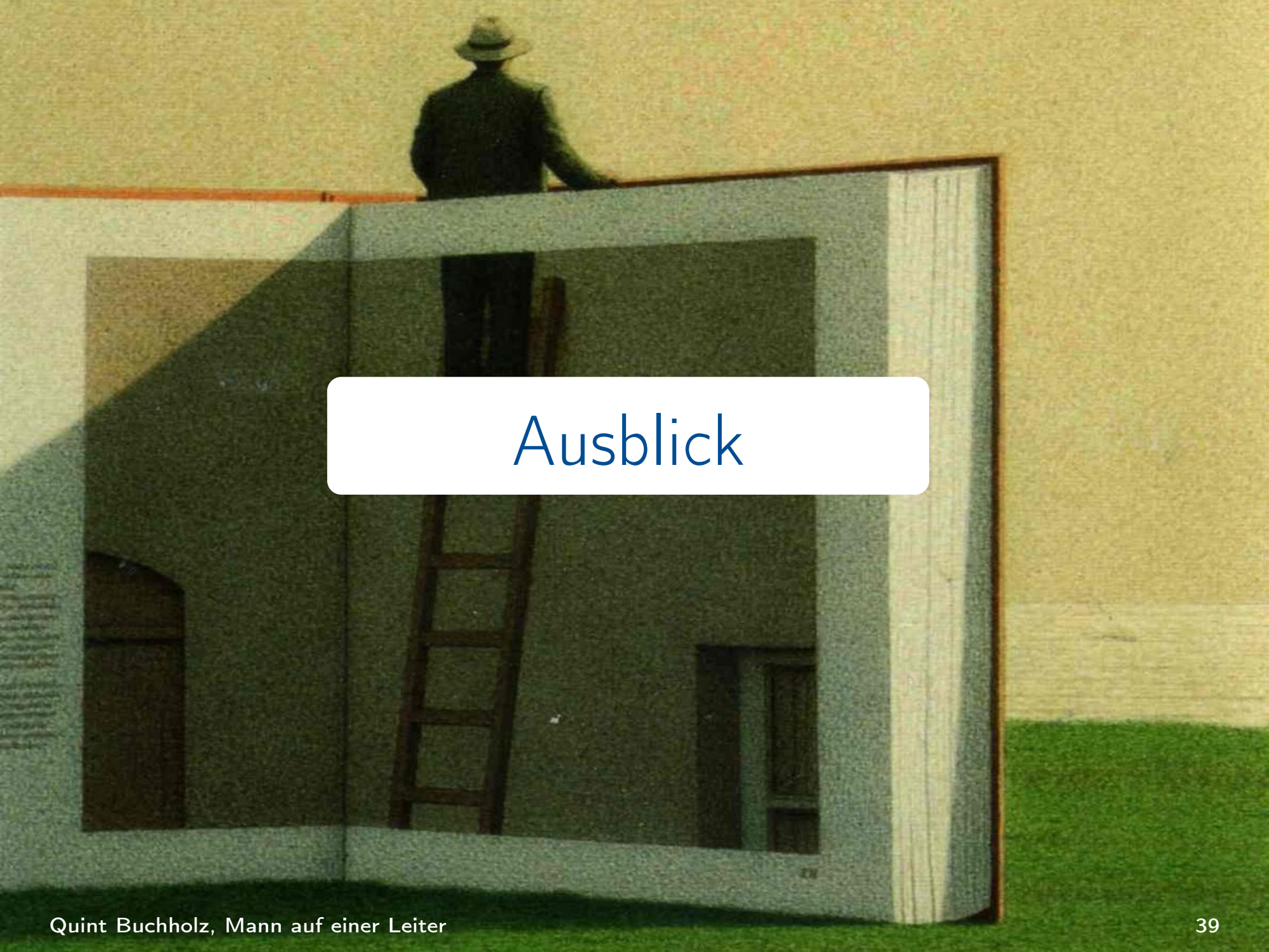
```
@RunWith(Suite.class)
@SuiteClasses( { A.class, B.class })
public class AllTests {}
```

- Alle Tests in der Kategorie `Slow`:

```
@RunWith(Categories.class)
@IncludeCategory(Slow.class)
public class AllSlowTests extends AllTests {}
```

- Alle anderen Tests:

```
@RunWith(Categories.class)
@ExcludeCategory(Slow.class)
public class AllFastTests extends AllTests {}
```



Ausblick

Quint Buchholz, Mann auf einer Leiter

Was jetzt?

- Aktualisierung auf neue Version ist einfach
- Alte Tests funktionieren weiterhin
- Neue Tests profitieren von neuen Features
- Alte Tests können nach und nach vereinfacht werden
- Ausprobieren!

Ausprobieren!

`http://www.junit.org/`

Vielen Dank!

Mail `marc@andrena.de`
Twitter `marcphilipp`
Blog `http://marcphilipp.tumblr.com/`