12.–15.09.2010
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Theorien, Regeln und Kategorien

JUnits experimentelle Features

Jens Schauder

LINEAS Informationstechnik

Categories

Slow Tests

DB Tests

GUI Tests

```
public class MixedTest {

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

    @Test
    public void testSomethingThatIsNotSlow() {
    }

    @Category(Gui.class)
    @Test
    public void testSomeGuiThing() {
    }

    @Category({ DB.class, Slow.class })
    @Test
    public void testSomeDBThing() {
    }
}
```

```
@Category(Slow.class)
public class SlowTests {

    @Test
    public void testSomethingSlow() throws Exception {
    }

    @Test
    public void testSomethingElseSlow() throws Exception {
    }

    @Test
    public void testMoreSlowStuff() throws Exception {
    }
}
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Category {
        Class<?>[] value();
}
```

Only X Tests

```
@RunWith(Categories.class)
@IncludeCategory(Slow.class)
@SuiteClasses({ MixedTest.class, SlowTests.class })
public class OnlySlowTestSuite1 {
}
```

Suites
Recyclen

```
@SuiteClasses({ MixedTest.class, SlowTests.class })
public class AllTestSuite {}


@RunWith(Categories.class)
@IncludeCategory(Slow.class)
public class OnlySlowTestSuite2 extends AllTestSuite {}
```

```
@RunWith(ClasspathSuite.class)
public class AllTestsWithCPS {}


@RunWith(Categories.class)
@IncludeCategory(Slow.class)
@SuiteClasses({ AllTestsWithCPS.class })
public class OnlySlowTestSuiteWithCPS {}
```

ClasspathSuite.class
http://johanneslink.net/projects/cpsuite.jsp

Rules

Code Review:

```
@Test
public void testFrameFactory() {
        JFrame frame = new FrameFactory()
                .createFrame("testName");
        assertNotNull(frame);
        assertEquals("testName", frame.getTitle());
}
```

```
@Test
public void testFrameFactory()  throws Exception {
        SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                        JFrame frame = new
                                FrameFactory()
                                .createFrame("testName");
                        assertNotNull(frame);
                        assertEquals("testName", frame.getTitle());
                }
        });
  }
```

```
@Rule
 public SwingRule rule = new SwingRule();

@Test
public void testFrameFactory() {
            JFrame frame = new FrameFactory()
                        .createFrame("testName");
           assertNotNull(frame);
           assertEquals("testName", frame.getTitle());
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

```
public class SwingRule implements MethodRule {
        public Statement apply(final Statement base,
                FrameworkMethod method,
                Object target) {
                return new Statement() {
                        @Override
                        public void evaluate() throws Throwable {
                        SwingUtilities.invokeAndWait(
                                new Runnable() {
                                        public void run() {
                                                base.evaluate();
                                        }
                                });
                        }
                };
        }
}
```

Wozu?

Patterns

Ressourcen

Context

Testverhalten

Theories

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

        @DataPoint
        public static String a = "a";

        @DataPoint
        public static String b = "bb";

        @DataPoint
        public static String c = "ccc";

        @Theory
        public void stringTest(String x, String y) {
                System.out.println(x + " " + y);
        }
}
```

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

        @DataPoint
        public static String a = "a";

        @DataPoint
        public static String b = "bb";

        @DataPoint
        public static String c = "ccc";

        @Theory
        public void stringTest(String x, String y) {
                System.out.println(x + " " + y);
        }
}
```

```
a a
a bb
a ccc
bb a
bb bb
bb ccc
ccc a
ccc bb
ccc ccc
```

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

	@DataPoint
	public static String a = "a";

	@DataPoint
	public static String b = "bb";

	@DataPoint
	public static String c = "ccc";

	@Theory
	public void stringTest(String x, String y) {
		assumeTrue(x.length() > 1);
		System.out.println(x + " " + y);
	}
}
```

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

	@DataPoint
	public static String a = "a";

	@DataPoint
	public static String b = "bb";

	@DataPoint
	public static String c = "ccc";

	@Theory
	public void stringTest(String x, String y) {
		assumeTrue(x.length() > 1);
		System.out.println(x + " " + y);
	}
}
```

```
bb a
bb bb
bb ccc
ccc a
ccc bb
ccc ccc
```

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

        @DataPoints
        public static String[] a = { "a", "bb", "ccc" };

        @DataPoints
        public static Integer[] j = { 1, 2, 3 };

        @Theory
        public void someTest(String x, Integer y) {
                ...
        }
}
```

Gleiche Typen

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

        @Theory
        public void imagineThisIsATest(
                @AllCreditCards String x,
                @AllNames String y) {
                ...
        }
}
```

```
@Retention(RetentionPolicy.RUNTIME)
@ParametersSuppliedBy(CreditCardSupplier.class)
public @interface AllCreditCards {}
```

```
public class CreditCardSupplier extends ParameterSupplier {

        @Override
        public List<PotentialAssignment> getValueSources(
                ParameterSignature signature) {

                ArrayList<PotentialAssignment> result = new
                        ArrayList<PotentialAssignment>();

                result.add(PotentialAssignment.forValue(
                        "Amex", "Amex "
                ));
                …
                return result;
        }
}
```

```
public class CreditCardSupplier extends ParameterSupplier {

        @Override
        public List<PotentialAssignment> getValueSources(
                ParameterSignature signature) {

                List<PotentialAssignment> result = new
                        ArrayList<PotentialAssignment>();

                result.add(PotentialAssignment.forValue(
                        "Amex", "Amex "
                ));
                …
                return result;
        }
}
```

```
@Theory
public void testIntegers(
        @TestedOn(ints = { 2, 3, 4, 7, 13, 23, 42 }) int i) {
        …
}
```

Wozu?

- Theories
- N Parameterquellen
- Tests mit Parametern
- Ein Ergebnis pro Test
- @DataPoint, @DataPoints, ParameterSupplier
- Assume

- Parameterized
- 1 Parameterquelle
- Konstruktor mit Parametern
- Ein Ergebnis pro Parametersatz
- @Parameters

12.–15.09.2010
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Theorien, Regeln und Kategorien

JUnits experimentelle Features

Jens Schauder

LINEAS Informationstechnik