Lambda goes Alpha

JUnit wird weiterentwickelt

Stefan Bechtold, Sam Brannen, Matthias Merdes, Marc Philipp

Abstract fehlt noch.

Hintergrund und Motivation

Das Test-Framework JUnit wurde ursprünglich im Jahr 2000 in der ersten Version veröffentlicht, von Kent Beck und Erich Gamma der Legende nach im Flugzeug per Pair-Programming entwickelt [1]. JUnit war Vorbild für viele andere Test-Frameworks und wurde auf zahlreiche andere Plattformen portiert. Nach einer Untersuchung von Ende 2013 [2] ist JUnit (neben Logging-Frameworks) die am meisten verwendete Java-Bibliothek überhaupt. 43 Millionen Downloads allein in 2014 belegen diesen besonderen Status zusätzlich.

Nach vielen Jahren der Weiterentwicklung und Wartung ist JUnit mittlerweile bei Version 4.12 angelangt. Im Lauf dieser Wartungsarbeiten zeigte sich, dass es an der Zeit war, über die fünfte Generation von JUnit nachzudenken. Zum einen, weil die vorherigen Versionen naturgemäß keine Sprachfeatures von Java 8 unterstützen konnten, zum anderen, weil die zur Verfügung stehenden Konstrukte die Komponierbarkeit von Erweiterungen erschwerten. Nicht zuletzt ist die Art und Weise, in der IDEs JUnit integrieren, hinderlich für die Weiterentwicklung.

Da es sich bei dieser Neukonzeption um ein größeres Unterfangen handelt, wurde zur Mitfinanzierung des JUnit-Lambda-Projekts eine Crowdfunding-Kampagne ins Leben gerufen [3]. Hierbei konnte mehr als das Doppelte des ursprünglich anvisierten Minimalbudgets erlöst werden. Insbesondere zeigt aber die hohe Zahl von 474 Unterstützern das große Interesse der Community an der Vision des JUnit-Lambda-Projektes. Neben 428 privaten Personen haben 36 Unternehmen, darunter große Anwender, insbesondere aber auch Hersteller von Entwicklungswerkzeugen, ihren Anteil zur Unterstützung beigetragen. Mit diesem Rückhalt fand im Oktober 2015 das Kickoff-Treffen in Karlsruhe statt, bei dem die Anforderungen bezüglich Integrierbarkeit mit den Herstellern von Gradle, Eclipse, IntelliJ IDEA und dem Spring Framework diskutiert wurden.

Bald nach dem Workshop wurde der Prototyp für JUnit Lambda vorgestellt, um Feedback der Community einzuholen [4]. Auf Basis dieses Feedbacks wird zur Zeit die Version JUnit 5 Alpha M1 entwickelt. Auf diese Version bezieht sich auch der vorliegende Artikel. Es liegt in der Natur der Sache, dass sich API und darunterliegende Konzepte noch ändern können. Im Folgenden werden wir einen ersten Überblick über die neue Gesamtarchitektur, das Programmiermodell und das zugehörige API, sowie die Themen Erweiterbarkeit und Integrierbarkeit geben.

Grundlegende Architektur

Eines der Hauptziele von JUnit 5 ist die Entkopplung der API zum Schreiben von Tests von der API zum Ausführen derselben. In JUnit 4 sind beide in einem Artefakt enthalten. Alle, die JUnit auf irgendeine Weise verwenden, hängen von diesem ab — egal ob es sich dabei um IDEs, Build Tools, andere Testing Frameworks oder Benutzer, die Tests schreiben, handelt. Diese Verwender benutzen teilweise Interna von JUnit, etwa interne Klassen oder private Instanzvariablen, die gar per Reflection ausgelesen werden. Die Weiterentwicklung von JUnit 4 wurde dadurch immens erschwert, in Teilen sogar nahezu

unmöglich gemacht.

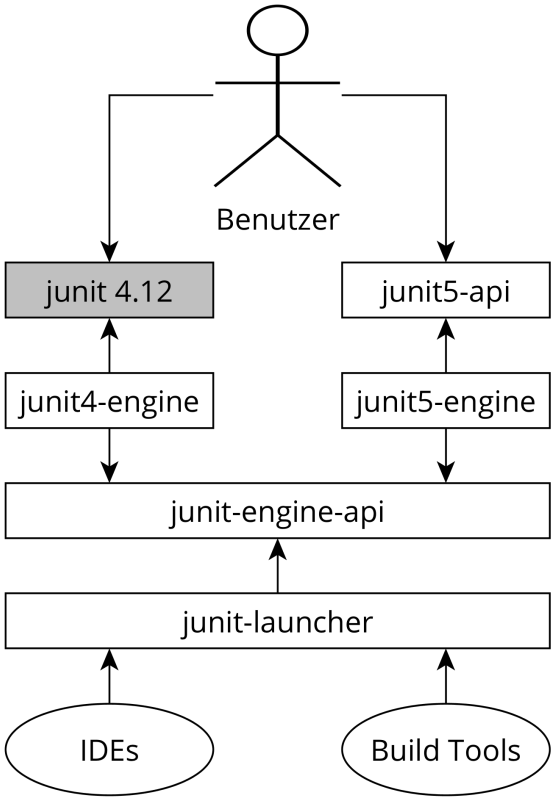

Abbildung 1. JUnit 5 Module

Um dieses Problem in Zukunft zu vermeiden, ist JUnit 5 in verschiedene Module aufgeteilt (s. Abbildung 1). Zum Schreiben von Tests benötigt man lediglich das Modul *junit5-api*, in dem alle Annotations wie etwa `@Test`, grundlegende `Assertions` etc. enthalten sind. Build Tools und IDEs verwenden *junit-launcher* zum Ausführen von Tests. Der darin enthaltene *Launcher* orchestriert das Auffinden und die Ausführung von Tests durch verschiedene *Engines*. Die Engine-Abstraktion ermöglicht die Ausführung von JUnit 4 und JUnit 5 Testfällen im selben Testlauf. Die Engines registrieren sich über den *ServiceLoader*-Mechanismus beim Launcher. Die Engine-Implementierungen hängen dabei nur vom Artefakt *junit-engine-api* ab. JUnit 5 wird zwei Engines mitliefern: *junit4-engine* und *junit5-engine* zum Auffinden und Ausführen von JUnit 4 bzw. 5 Tests. Die Engine API ist sogleich ein Angebot an existierende und zukünftige Testing Frameworks. Wenn sie eine Engine-Implementierung bereitstellen, können ihre Tests in allen IDEs und Build Tools laufen, die JUnit 5 unterstützen.

Programmiermodell und Test API

Die hier vorgestellten Beispiele basieren auf dem Milestone Alpha M1. Bis zum ersten JUnit 5 Release können sich daher noch Veränderungen ergeben. Tests in JUnit 5 sehen den Tests aus der vorherigen Version sehr ähnlich. Auf den ersten Blick ist fast kein Unterschied zu erkennen. Diese Entscheidung wurde bewusst getroffen, damit der Umstieg zu JUnit 5 leicht fällt.

SinglePassingTestSampleClass

```
class SinglePassingTestSampleClass {
    @Test
    void singlePassingTest() {
        System.out.println("Test got executed!");
    }
}
```

Im Detail sind dennoch maßgebliche Veränderungen sichtbar. So ist es nicht mehr notwendig, dass Test-Klassen und Test-Methoden mit dem Keyword `public` versehen werden, um den Test-Code schlanker zu machen.

Alle Annotationen für JUnit 5 befinden sich in einem eigenen Package namens `org.junit.gen5.api`. Spätestens nach einem Blick in dieses Package wird ersichtlich, dass es im Kern deutliche Unterschiede und Verbesserungen zu JUnit 4 gibt, auf die wir nun im Detail eingegangen.

Selbstdefinierte Namen

Jeder Test-Klasse und jeder Test-Methode kann ein selbstdefinierter Name zugewiesen werden. Dieser Name wird im Ergebnis anstelle des Klassen- oder Methodennamens als Beschreibung angezeigt und erlaubt auch den Einsatz von Leer- und Sonderzeichen.

AdvancedSinglePassingTestSampleClass

```
@Name("My custom test-class name")
class AdvancedSinglePassingTestSampleClass {
    @BeforeAll
    static void myBeforeAllMethod() {
        // executed once before the first
        // test method in this test class
    }

    @AfterAll
    static void myAfterAllMethod() {
        // executed once after the last
        // test method in this test class
    }

    @BeforeEach
    void myBeforeEachMethod() {
        // executed before each
        // test method in this test class
    }

    @AfterEach
    void myAfterEachMethod() {
        // executed after each
        // test method in this test class
    }

    @Test
    @Name("My custom test-method name")
    void singlePassingTest() {
        System.out.println("Test got executed!");
    }
}
```

Before / After Methoden

Die Code-Ausführungen vor und nach jedem Tests werden weiterhin unterstützt und sind durch die Verwendung von `BeforeAll`, `BeforeEach`, `AfterEach` und `AfterAll` nun expliziter in der Ausdruckweise der Annotationen. Das Verhalten der Methoden entspricht exakt dem heutigen Prinzip von JUnit 4.

Unter der Haube haben sich gerade in diesem Bereich viele Änderungen ergeben. Im nächsten Kapitel wird das neue Modell für Extensions vorgestellt, das hier tiefere Einblicke mitbringt.

Tests deaktivieren / ignorieren

Die Ausführung von Tests kann in JUnit 4 mit der Annotation `@Ignore` verhindert werden. In JUnit 5 wird das Deaktivieren von Tests flexibler möglich sein. Mithilfe von Conditions können verschiedene Einschränkungen angegeben werden, die bei der Ausführung der Tests dynamisch geprüft werden. Im einfachsten Fall soll ein Test immer deaktiviert werden. Dafür bietet JUnit 5 bereits jetzt die Annotation `@Disabled`. Alle Test-Klassen oder Test-Methoden, die mit dieser Annotation versehen sind, werden bei der Ausführung ignoriert. Optional kann ein Grund für das Deaktivieren mit angegeben und dokumentiert werden.

ConditionTestsSampleClass

```
class ConditionTestsSampleClass {
    @Test
    @Disabled("reason: not yet implemented, see ticket #32")
    void stillFailingTest() {
        fail("not yet implemented!");
    }
}
```

Zum Release können noch weitere Conditions hinzukommen, wie z.B. das Ausführen von Tests in Abhängigkeit von Umgebungsvariablen oder Konfigurationsparametern. Es ist zudem vorgesehen, dass zukünftig eigene Conditions innerhalb des Projekts selbst definiert werden können.

Tags und Meta-Annotations

Neben Conditions erlaubt JUnit 5 auch das Taggen von Tests und die Filterung nach Tests für ein oder mehrere Tags gesetzt wurden. Über dieses Feature können bspw. Test-Gruppen erstellt werden.

TagsAndMetaAnnotationsSampleClass

```
class TagsAndMetaAnnotationsSampleClass {
    @Test
    @Tag("fast")
    void testTaggedAsRunningFast1() throws Exception {
        System.out.println("I am running very fast!");
    }

    @MyFastTest
    void testTaggedAsRunningFast2() throws Exception {
        System.out.println("I am running very fast, too!");
    }
}
```

Zusätzlich werden Meta-Annotations — d.h. Annotationen auf Annotationen — unterstützt, so dass

eigene Gruppen erstellt und im Projekt vorgegeben werden können. Als Beispiel wurde hier eine Gruppe `MyFastTest` angelegt, die eine Methode als Test kennzeichnet und sie mit dem Tag "fast" markiert. Im konkreten Fall werden also beide Methoden identisch behandelt.

Die entsprechende Annotation ist sehr einfach selbst zu schreiben:

MyFastTest

```
@Test
@Tag("fast")
public @interface MyFastTest {
}
```

Entfall von `expected` und `timeout`

Eine weitere Änderung ist der Wegfall der Properties `expected` und `timeout` auf der Annotation `@Test`. Durch den Einsatz von Lambdas lassen sich diese Semantiken ausdrucksstärker formulieren. Hier bieten Assertion Frameworks (z.B. AssertJ) bereits gute Lösungen an, für den Puristen bringt jedoch auch JUnit 5 bereits eine einfache Variante mit:

ExpectedExceptionSampleClass

```
class ExpectedExceptionSampleClass {
    @Test
    void testWithExpectedException() {
        IllegalArgumentException thrown = expectThrows(IllegalArgumentException.class, ()
-> {
            // code that throws exception
        });
        assertEquals("foo", thrown.getMessage());
    }
}
```

Dependency / Parameter Injection

Als neues Feature unterstützt JUnit 5 das Auflösen von Abhängigkeiten in Form von Methoden-Parametern. Ein Beispiel dafür ist das Auswerten des Testnamens. Dies wird bereits von JUnit 5 unterstützt, so dass der Name der Test-Methode in Fehlermeldungen oder Ausgaben verwendet werden kann.

ParameterInjectionTestSampleClass

```
class ParameterInjectionTestSampleClass {
    @Test
    @Name("MyName")
    void injectTestName(@TestName String testname) throws Exception {
        // prints: Test MyName got executed!
        System.out.println("Test " + testname + " got executed!");
    }
}
```

Die Funktionsweise und die Erweiterungsmöglichkeiten dieses Features werden im nachfolgenden Kapitel im Detail erläutert.

Verschachtelung von Test-Klassen

JUnit 5 unterstützt die Verschachtelung von Test-Klassen. Mittels der Annotation `@Nested` können innere Klassen als Test-Klassen deklariert werden. Tests können über dieses Feature hierarchisch zu Gruppen zusammengefasst werden, um den Tests mehr Übersicht zu geben.

NestedTestsSampleClass

```
class NestedTestsSampleClass {
    @Test
    void testOnTopLevel() {
    }

    @Nested
    class NestedTest {
        @Test
        void testOnNestedLevel() {
        }

        @Nested
        class DoubleNestedTest {
            @Test
            void testOnDoubleNestedLevel() {
            }
        }
    }
}
```

Auf diese Art und Weise lassen sich auch Testfälle für Hierarchien übersichtlich aufstellen. Es ist nach wie vor möglich, dass Test-Klassen von bestehenden Test-Klassen ableiten und darüber die Tests der Superklasse erben. Ebenso können statische innere Klassen definiert werden. Diese werden von JUnit

5 nun wie top-level Test-Klassen behandelt und parallel zur enthaltenden Klasse im Testplan angezeigt.

Erweiterbarkeit

Ein grundlegendes Ziel des JUnit Lambda Projekts ist die flexible Erweiterbarkeit des JUnit 5 Frameworks durch kombinierbare Erweiterungspunkte.

Probleme des bisherigen Erweiterungsmodells

JUnit 4 führte das `Runner` API ein, womit Entwickler das Verhalten von JUnit komplett erweitern konnten. Bekannt sind Runners wie `Parameterized` von JUnit selbst und maßgeschneiderte `Runner` von Drittparteien wie zum Beispiel der `SpringJUnit4ClassRunner`, `MockitoJUnitRunner`, `Cucumber`, usw. Ein `Runner` kann durchaus sehr mächtig sein, allerdings lassen sich mehrere `Runner` nicht kombinieren. Spätere Versionen von JUnit 4 haben deshalb Rules eingeführt.

Seit Version 4.7 unterstützt JUnit `MethodRules`, und ab JUnit 4.9 gibt es Unterstützung für `TestRules`. Eine `TestRule` kann auf der Methoden- oder Klassenebene angewendet werden, aber nur eine `MethodRule` hat Zugriff auf die Testinstanz. Im Gegensatz zu Runners lassen sich Rules beliebig kombinieren. Allerdings ist der Einsatz einer Rule beschränkt: eine Rule kann nicht gleichzeitig `TestRule` und `MethodRule` implementieren. Das hat zur Folge, dass manche Drittparteien beide Rule APIs in verschiedenen Klassen implementieren mussten: `SpringClassRule` und `SpringMethodRule` können als konkretes Beispiel dienen.

JUnit 5 Erweiterungsmodell

Um die oben genannten Probleme von JUnit 4 zu vermeiden, führt JUnit 5 ein komplett neues Erweiterungsmodell ein. Für den `Launcher` gibt es das `TestExecutionListener` API. Als Ersatz für Rules gibt es die `TestExtension` APIs. Anstatt Runners können mehrere TestEngines gleichzeitig zum Einsatz kommen, und zwar innerhalb des gleichen Testlaufs.

Launcher und TestEngine

Wie vorher erwähnt (s. Grundlegende Architektur) gibt es die `Launcher` und `TestEngine` APIs als Basis für die Ausführung von Tests.

Einem Launcher können beliebige `TestExecutionListener` hinzugefügt werden. Während der Ausführung werden alle Listener über verschiedene Events informiert, wie z.B. `executionSkipped`, `executionStarted`, `executionFinished` usw. Mitgeliefert werden ein `LoggingListener` und ein `SummaryGeneratingListener`, die zum Logging bzw. zur einfachen Report-Generierung auf der Konsole dienen. Für die Zukunft sind zusätzliche Listener für "Rich Reporting" geplant, z.B. für die Generierung von XML- oder JSON-Reports. Tool-Anbieter können ihre eigenen Listener registrieren, um Reports in der IDE und in Builds anzuzeigen.

Eine Vielfalt von Tests für Java-Anwendungen werden zukünftig mit dem JUnit 5 API entwickelt

werden, das von der JUnit5TestEngine unterstützt wird. Für Tests, die noch auf JUnit 4 oder JUnit 3 basieren, gibt es die JUnit4TestEngine. Für andere Testarten gäbe es die Möglichkeit, eine eigene TestEngine zu implementieren, z.B. um ein anderes Format wie z.B. eine XML-Definition anstatt Java-Klassen zu unterstützen.

Drittparteien können eigene TestEngines über den Java ServiceLoader Mechanismus automatisch registrieren lassen, indem eine Textdatei unter META-INF/services/org.junit.gen5.engine.TestEngine im JAR angelegt wird. Diese Datei muss einfach den kompletten Klassennamen der TestEngine -Implementierung enthalten.

Eigene TestEngines sollten aber nur implementiert werden, wenn bestehende TestEngines nicht ausreichend erweiterbar sind. In allen anderen Fällen sollte die TestExtension APIs der JUnit5TestEngine verwendet werden.

TestExtension und ExtensionPoint APIs

Mit JUnit 5 wird es möglich, beliebige TestExtensions gleichzeitig zu registrieren. Diese Extensions übernehmen die Rollen der TestRule und MethodRule APIs aus JUnit 4. Eine oder mehrere TestExtensions können mittels der @ExtendWith Annotation registriert werden, und zwar auf Klassen- oder Methoden-Ebene. Einige Extensions, die von JUnit mitgeliefert werden, werden auch automatisch registriert, wie z.B. die DisabledCondition und der TestNameParameterResolver.

TestExtension ist selbst nur ein *Marker Interface*. Konkrete Erweiterungspunkte leiten von der Schnittstelle ExtensionPoint ab, die selbst von TestExtension ableitet. Als Erweiterung zur TestExtension kann die Reihenfolge von ExtensionPoints auch beeinflusst werden.

Zum Zeitpunkt des Schreibens gibt es folgende ExtensionPoints für die JUnit5TestEngine:

- ContainerExecutionCondition: entscheidet zur Laufzeit, ob ein Test-Container (e.g., eine Testklasse) ausgeführt werden soll
- TestExecutionCondition entscheidet zur Laufzeit, ob ein Test (e.g., eine Testmethode) ausgeführt werden soll
- InstancePostProcessor: bekommt eine Referenz auf die Testinstanz, um z.B. Abhängigkeiten zu injizieren (i.e., *Dependency Injection*)
- MethodParameterResolver: sorgt dafür, dass Parameter für @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, und @Test Methoden dynamisch aufgelöst werden
- BeforeEachExtensionPoint: dient als *Callback* rund um die Ausführung von @BeforeEach Methoden
- AfterEachExtensionPoint: dient als *Callback* rund um die Ausführung von @AfterEach Methoden
- BeforeAllExtensionPoint: dient als *Callback* rund um die Ausführung von @BeforeAll Methoden
- AfterAllExtensionPoint: dient als *Callback* rund um die Ausführung von @AfterAll Methoden

Da wir nun einen Überblick der möglichen ExtensionPoints bekommen haben, schauen wir uns im Folgenden einige konkrete Beispiele an.

Beispiel: Bedingungsabhängige Test-Ausführung

Nehmen wir an, wir wollen, dass einige Testmethoden nur auf dem CI-Server ausgeführt werden. Es besteht die Möglichkeit, ein bestimmtes Tag zu definieren (z.B., "ci-server"), dieses durch `@Tag("ci-server")` auf den betroffenen Methoden zu deklarieren und das `"ci-server"` Tag beim `Launcher` bzw. beim Build-Plugin oder in der IDE anzugeben. Das würde natürlich funktionieren, aber diese Vorgehensweise ist eher statisch. Wenn wir dynamisch entscheiden müssen, ob ein Test ausgeführt werden sollte, dann können wir eine maßgeschneiderte `TestExecutionCondition` implementieren, wie folgt.

RunOnCiServer

```
import static org.junit...ConditionEvaluationResult.disabled;
import static org.junit...ConditionEvaluationResult.enabled;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(RunOnCiServer.CiServerCondition.class)
public @interface RunOnCiServer {

  static class CiServerCondition implements TestExecutionCondition {

    @Override
    public ConditionEvaluationResult evaluate(TestExtensionContext ctx) {
      boolean onCiServer = // determine if on CI Server

      if (!onCiServer) {
        return disabled("Not on CI Server");
      }

      return enabled("On CI Server");
    }
  }
}
```

`@RunOnCiServer` ist eine Annotation, die selbst mit `@ExtendWith(RunOnCiServer.CiServerCondition.class)` meta-annotiert ist. Wenn eine Testmethode mit `@RunOnCiServer` annotiert ist — wie unten in der `CiServerTests` Klasse — dann kommt die `CiServerCondition` zum Einsatz. In der Methode `evaluate()` von `CiServerCondition` können wir dynamisch, programmatisch entscheiden, ob der jetzige Testlauf auf dem CI-Server stattfindet, z.B. anhand von Umgebungsvariablen. Über den `TestExtensionContext` können wir auf die Testinstanz und die Testmethode zugreifen, falls wir weitere Informationen benötigen.

CiServerTests

```
class CiServerTests {

    @Test
    @RunOnCiServer
    void onlyRunOnCiServer() {
        // ...
    }
}
```

Beispiel: Dependency Injection mit Mockito

Mit JUnit 4 hatte man die Option, Mockito-Mocks über drei Wege in Testklassen zu injizieren: mit dem `MockitoJUnitRunner`, mit der `MockitoRule` oder programmatisch mittels `MockitoAnnotations.initMocks(this)`. Alle Varianten erfordern, dass die entsprechenden Felder mit `@Mock` annotiert sind. Da JUnit 5 weder `Runner` noch Rules unterstützt, müssen wir nun einen anderen Weg finden.

Das JUnit 5 Erweiterungsmodell bietet uns an dieser Stelle zwei Optionen an, die auch leicht kombinierbar sind. Erstens können wir das `InstancePostProcessor` API implementieren, um Mocks in die Felder der Testinstanz zu injizieren. Das deckt die Funktionalität des `MockitoJUnitRunner` sowie der `MockitoRule` ab. Zweitens können wir auch *Dependency Injection* für Methoden-Parameter unterstützen, indem unsere `TestExtension` gleichzeitig das `MethodParameterResolver` API implementiert. Das einzige Problem an der Stelle ist, dass die `@Mock` Annotation von Mockito nicht auf Methodenparameter deklariert werden darf. Das lösen wir aber schnell, indem wir eine eigene `@InjectMock` Annotation wie folgt deklarieren.

InjectMock

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface InjectMock {
}
```

Die nächste Aufgabe ist, die `TestExtension` zu entwickeln. Wie unten sichtbar, implementiert unsere `MockitoExtension` die `InstancePostProcessor` und `MethodParameterResolver` APIs. Die Methode `postProcessTestInstance()` verwendet die vorher erwähnte Methode `MockitoAnnotations.initMocks()` von Mockito, aber an dieser Stelle wird die Testinstanz aus dem jetzigen `TestExtensionContext` geholt. Damit ist *Dependency Injection* für Felder in Testklassen abgedeckt.

Um *Dependency Injection* für Methoden-Parameter zu implementieren, müssen wir noch ein bisschen mehr selbst leisten. Erstens entscheidet die Methode `supports()`, ob die `MockitoExtension` Parameter auflösen kann, die mit `@InjectMock` annotiert sind. Zweitens sorgt die Methode `resolve()` dafür, dass der richtige Mock an die Testmethode weitergegeben wird. Die Mock-Instanzen werden in einer `Map`

nach Typ hinterlegt.

MockitoExtension

```
import static org.mockito.Mockito.mock;

public class MockitoExtension
  implements InstancePostProcessor, MethodParameterResolver {

    private final Map<Class<?>, Object> mocks = new ConcurrentHashMap<>();

    @Override
    public void postProcessTestInstance(TestExtensionContext context) {
        MockitoAnnotations.initMocks(context.getTestInstance());
    }

    @Override
    public boolean supports(Parameter parameter,
            MethodContext methodContext,
            ExtensionContext extensionContext) {

        return parameter.isAnnotationPresent(InjectMock.class);
    }

    @Override
    public Object resolve(Parameter parameter,
            MethodContext methodContext,
            ExtensionContext extensionContext)
                throws ParameterResolutionException {

        return getMock(parameter.getType());
    }

    private Object getMock(Class<?> mockType) {
        return this.mocks.computeIfAbsent(mockType, type -> mock(type));
    }

}
```

Die Klasse `MockitoDependencyInjectionTests` zeigt, wie die `MockitoExtension` verwendet wird.

MockitoDependencyInjectionTests

```
@ExtendWith(MockitoExtension.class)
class MockitoDependencyInjectionTests {

    @Mock
    NumberGenerator numberGenerator;

    @BeforeEach
    void initialize(@InjectMock Person person, @TestName String testName) {
        when(person.getName()).thenReturn(testName);
        when(this.numberGenerator.next()).thenReturn(42);
    }

    @Test
    @Name("Yoda")
    void injectedMocksOne(@InjectMock Person person) {
        assertEquals("Yoda", person.getName());
        assertEquals(42, this.numberGenerator.next());
    }

    @Test
    @Name("Dilbert")
    void injectedMocksTwo(@InjectMock Person person) {
        assertEquals("Dilbert", person.getName());
        assertEquals(42, this.numberGenerator.next());
    }

}
```

Integrationen

Alte JUnit 4 Tests mit JUnit 5 ausführen

Wie bereits erwähnt, ermöglicht es die neue Architektur, JUnit 4 und JUnit 5 Tests im selben Testlauf auszuführen. Ist das Artefakt *junit4-engine* zur Laufzeit der Tests auf dem *Classpath*, werden JUnit 4 Tests gefunden und ausgeführt.

Somit ist es möglich, neue Tests mit JUnit 5 zu schreiben und alte Tests weiterhin JUnit 4 verwenden zu lassen. Jedes Projekt-Team kann dann selbst entscheiden, ob und wann es seine JUnit 4 Tests auf JUnit 5 umstellt. Neue Features werden jedoch nur in JUnit 5 eingebaut werden.

Neue JUnit 5 Tests mit JUnit 4 ausführen

Da es unter Umständen eine Weile dauern wird, bis alle IDEs und Build Tools eine Unterstützung für JUnit 5 mitbringen bzw. nicht jeder Entwickler stets die neuste Version seiner IDE bzw. seines Build Tools verwenden kann, werden wir eine Implementierung von `Runner` (JUnit 4 API) namens `JUnit5` mitliefern, die es ermöglicht, einen JUnit 5 Test mit JUnit 4 auszuführen. Dieser `Runner` wird nicht alle Features unterstützen können, die JUnit 5 bietet, ist als Übergangslösung aber durchaus zu gebrauchen.

Der `Runner` kann entweder für einen einzelnen JUnit 5 Test verwendet werden (s. `SingleJUnit5TestRunWithJUnit4`) oder, ähnlich dem `Suite` Runner, um eine Menge von JUnit 5 Tests auf einmal auszuführen (s. `JUnit4SuiteOfJUnit5Tests`).

Einzelner JUnit 5 Test, der mit JUnit 4 ausführbar ist

```
@RunWith(JUnit5.class)
public class SingleJUnit5TestRunWithJUnit4 {

    @Test
    void test() {
        fail("Hello to JUnit 4 from JUnit 5");
    }
}
```

JUnit 4 Test Suite, die alle Tests im Package "org.junit.example" ausführt

```
@RunWith(JUnit5.class)
@Packages("org.junit.example")
public class JUnit4SuiteOfJUnit5Tests {
}
```

JUnit 4/5 Tests mit Gradle/Maven ausführen

Das JUnit Lambda Team hat in der Prototyp-Phase eigene Plugins geschrieben, die es ermöglichen JUnit 5 Tests mit Gradle und Maven auszuführen. Perspektivisch soll die Weiterentwicklung und Wartung dieser Plugins aber an die Maintainer von Gradle und Maven Surefire abgeben werden. Auf GitHub [5] findet man Beispielprojekte, die demonstrieren, wie man JUnit 5 Tests im Build-Prozess mit Gradle bzw. Maven ausführt.

Fazit und Ausblick

In diesem Artikel haben wir einen ersten Überblick über Architektur und API der nächsten Generation von JUnit gegeben. Ein besonderes Augenmerk beim Design lag auf der einfachen Erweiterbarkeit und

insbesondere auf der Möglichkeit, Erweiterungen frei zu kombinieren.

Ziel ist es weiterhin, eng mit den Tool-Herstellern zusammenzuarbeiten, um eine frühzeitige Integration in die wichtigsten Werkzeuge zu erreichen. Potentielle Early Adopter seien noch einmal auf die prototypischen Gradle- und Maven-Integration hingewiesen, außerdem natürlich auf die Brücke zu JUnit 4, mit der man heute schon JUnit 5 Tests auch in alten Infrastrukturen ausführen kann, sowie auf den Console-Runner, der JUnit 5 Tests auf der Konsole ausführt.

Wie bereits erwähnt, befindet sich JUnit 5 noch im Alpha-Stadium. Es ist abzusehen, dass es noch Änderungen sowohl am API als auch am Funktionsumfang geben wird. Auch hat der gegenwärtige Entwicklungsstand noch keineswegs Produktionsqualität. Weiterhin ist jetzt ein guter Zeitpunkt, um die Entwicklung von JUnit 5 durch kritisches Feedback zu begleiten, was durch Kommentieren des GitHub-Projektes [6] leicht geschehen kann. Wenn die Arbeiten an JUnit 5 weiter planmäßig weiterlaufen, dann ist noch in diesem Jahr mit einem Beta-Release und einer Version 5.0 zu rechnen.

Autoren

Stefan, Sam, Matthias und Marc sind Mitglieder des JUnit Lambda Teams.

Stefan Bechtold ist Software Architect & Engineer bei der Namics Deutschland GmbH mit den Schwerpunkt auf Java basierte Enterprise Plattformen. Er interessiert sich sehr für das automatisierte Testen von Software und hat den HierarchicalContextRunner für JUnit4 entwickelt. Namics ist Hauptsponsor der JUnit Lambda Kampagne und hat ihm ermöglicht, 6 Wochen seiner Arbeitszeit für JUnit Lambda zu verwenden.

Sam Brannen ist Software Consultant mit mehr als 17 Jahren Erfahrung und Mitbegründer der Swiftmind GmbH in Zürich. Er ist auch Trainer und ein gefragter Speaker auf Konferenzen rund um Java, Spring und Testing. Ausserdem ist er aktiver Core Committer für das Spring Framework seit 2007 und Autor des Spring TestContext Frameworks.

Matthias Merdes ist Lead Developer Architecture and Services bei der Heidelberg Mobil International GmbH, einem der Hauptsponsoren von JUnit-Lambda. Er befasst sich seit JDK 1.1 mit Java-Technologien vor allem im Backendbereich und ist begeisterter Groovy-Programmierer.

Allgemein interessiert Matthias sich für alle Technologien, die Enterprise Development einfacher, eleganter und effizienter machen.

Marc Philipp ist Softwareentwickler und Entwickler-Coach bei der andrena objects ag. Außerdem beschäftigt er sich mit Open-Source-Entwicklungswerkzeugen: Er ist einer der Maintainer von JUnit und Project Usus. andrena ist Hauptsponsor der JUnit Lambda Kampagne und hat ihm ermöglicht, 6 Wochen seiner Arbeitszeit für JUnit Lambda zu verwenden.

Links & Literatur

[1]

Martin Fowler: XUnit, http://www.martinfowler.com/bliki/Xunit.html

[2]

Tal Weiss: We Analyzed 30,000 GitHub Projects – Here Are The Top 100 Libraries in Java, JS and Ruby, http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[3]

Interview mit Johannes Link zur JUnit Lambda Crowdfunding-Kampagne, https://jaxenter.de/junit-lambda-gestartet-crowdfunding-kampagne-fuer-ein-junit-5-0-24898

[4]

Matthias Merdes und Dirk Dorsch: Logbuch JUnit Lambda: Zwischen Kickoff und Alpha kommt der Prototyp, https://jaxenter.de/logbuch-junit-lambda-zwischen-kickoff-und-alpha-kommt-der-prototyp-31196

[5]

JUnit 5 Beispielprojekte mit Gradle/Maven, https://github.com/junit-team/junit5-samples

[6]

JUnit Lambda auf GitHub, https://github.com/junit-team/junit-lambda