Beim Jupiter

JUnit 5 ist da!

Matthias Merdes

Hintergrund und Motivation

Das Test-Framework JUnit hat seit dem ersten Release im Jahr 2000 wesentlich dazu beigetragen, testgetriebene Entwicklung zu ermöglichen. Der Legende nach wurde es ursprünglich von Kent Beck und Erich Gamma im Flugzeug per Pair-Programming entwickelt [1] und war seitdem Vorbild für viele andere Test-Frameworks. Nach einer aktuellen Untersuchung vom Juli diesen Jahres [2] ist JUnit die am meisten verwendete Java-Bibliothek überhaupt.

Nach vielen Jahren der Weiterentwicklung und Wartung war JUnit 4 bei Version 4.12 angelangt. Im Lauf der Jahre hatte es sich gezeigt, dass die zur Verfügung stehenden Konstrukte die Komponierbarkeit von Erweiterungen erschwerten. Auch die Art und Weise, in der IDEs JUnit integrierten, stellte sich als zunehmend hinderlich für die Weiterentwicklung heraus. Dazu kommt, dass die vorherigen Versionen naturgemäß keine Sprachfeatures von Java 8 unterstützen konnten, weil stets die Kompatibilität zu Java 5 gewahrt bleiben musste.

Nach längerer Zeit ohne neue Releases und ohne grundlegende Modernisierungsmöglichkeit hat im Oktober 2015 die Entwicklung der vollständig überarbeiteten Version 5 begonnen. Nach einer Startfinanzierung durch die Crowdfunding-Kampagne "JUnit Lambda" begann im Oktober 2015 die Planung der neuen Version bei einem Workshop mit dem Core-Committer-Team und Herstellern der Tools Eclipse, Intellij Idea, Gradle und Pivotal (Spring Framework). Nach einem Prototypen und einer Reihe von Milestone-Releases ist im Sommer 2017 mit der Version 5.0.0 GA das erste Produktionsrelease der neuen Version erschienen. Version 5.1 ist bereits in Planung.

Im Folgenden wird ein Überblick über das neue Programmiermodell inklusive API, die neue Gesamtarchitektur, sowie die Themen Erweiterbarkeit und Integrierbarkeit gegeben.

Die wichtigsten Neuerungen für Entwickler

In diesem Abschnitt sollen einige neue Features vorgestellt werden, die für das Schreiben von Tests und damit für alle Entwickler wichtig sind. Alles Folgende bezieht sich auf die neue JUnit Jupiter Testengine, die im Abschnitt über Architektur weiter beschrieben wird.

Hello TestWorld

Als Einstieg soll zunächst ein bewusst minimalistisches Beispiel dienen, bevor einige fortgeschrittene oder grundsätzlich neue Feature vorgestellt werden. Naheliegenderweise heißt die wichtigste Annotation immer noch @Test. Im Unterschied zu JUnit 4 muss man allerdings ein anderes Package importieren: 'org.junit.jupiter.api' anstelle von 'org.junit'. Hierbei bezeichnet 'jupiter' die Testengine, die das neue JUnit 5-Programmiermodell unterstützt. Wie im Abschnitt über die Architektur weiter ausgeführt unterstützt die neue JUnit-Plattform die Koexistenz verschiedener Testengines.

```
@BeforeAll
static void setupServer() { }

@BeforeEach
void prepareDatabase() { }

@Test
void myTest() { }

@AfterEach
void cleanupDatabase() { }

@AfterAll
static void tearDownServer() { }
```

Figure 1. Hello TestWorld!

Man erkennt sogleich, dass sich die Namen der Lifecycle-Methoden im Vergleich zu JUnit 4 geändert haben. Die neue Nomenklatur ist das Ergebnis langer Diskussionen [3] und versucht, die Bedeutung noch deutlicher hervorzuheben, als das bisher der Fall war. Zu den wichtigsten Annotationen gehören die Folgenden: @BeforeAll wird einmal pro Testklasse aufgerufen, @BeforeEach vor jeder einzelnen Testmethode; analoges gilt für die After-Methoden. Erwähnenswert ist ebenfalls, dass der Modifier `public` entfallen kann.

Frei definierbare Namen

Die Annotation @DisplayName ermöglicht es, fast beliebige Namen für Testklassen und einzelnen Testmethoden zu verwenden, so dass sich übersichtliche und gut lesbare Baumdarstellungen in der GUI der IDE erzielen lassen.

```
@DisplayName("HTTP tests for REST product service")
public class DisplayNameDemo {

    @Test
    @DisplayName("GET 'http://localhost:8080/products/4711' user: Bob")
    public void getProduct() {...}

    @Test
    @DisplayName("POST 'http://localhost:8080/products/' user: Alice")
    public void addProduct() {...}
```

Figure 2. Frei definierte Namen im Code

Dies ergibt in der Ausführung eine saubere Darstellung in der IDE. Das Besondere an der Realisierung dieses neuen Features ist die Tatsache, dass die IDE trotzdem noch in der Lage ist, zwischen der Baumansicht und dem Sourcecode zu navigieren. Dies wird dadurch ermöglicht, dass die JUnit Platform der IDE die Möglichkeit gibt, einzelne Tests sauber zu referenzieren ohne reflexiv auf den Methodennamen zuzugreifen.

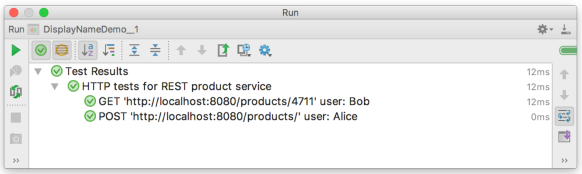

Figure 3. Frei definierte Namen in der IDE

Parameter Injection und Test-Reporting

Eine weitere Neuerung, die vor allem aus dem Spring Framework und anderen Dependency-Injection-Mechanismen bekannt ist, ist die Möglichkeit zur Parameter-Injection.

```
@Test
void reporting(TestReporter reporter) {
    reporter.publishEntry("balance", "47.11");
}
```

Figure 4. Parameter-Injection im Code

Durften Testmethoden in den bisherigen Versionen von JUnit keine Argumente haben, so fällt diese Einschränkung mit JUnit Jupiter weg. Dieser neue Mechanismus ermöglicht es Erweiterungen, Parameter für die Ausführung einer Testmethode bereitzustellen. Dazu können solche ParameterResolver-Extensions registriert werden, um Parameter aufzulösen, z.B. basierend auf einem bestimmten Typen oder einer bestimmten Annotation. Im angegebenen Beispiel wird der bereits mitgelieferte typbasierte 'TestReporterParameterResolver' aktiv, um eine TestReporter-Instanz zu liefern. Mit diesem TestReporter können unabhängig von der Konsole in geordneter Weise Informationen an die ausführende Umgebung geliefert werden.

Tags und Meta-Annotations

Eine weitere Veränderung im Vergleich zu JUnit 4 ist das Tagging von Tests mit bestimmten Labeln, was in JUnit 5 stringbasiert und nicht mehr über den klassenbasierten Category-Mechanismus gelöst ist.

```
@Test
@Tag("fast")
public void taggedTest() {...}
```

Figure 5. Stringbasiertes Tagging

Möchte man nun für eine Menge von Tests (z.B. für alle Integrationstests) ein solches Tag vergeben, so kann es sinnvoll sein, sich eines neu eingeführten Mechanismus' zu bedienen. So wird nämlich für alle im Jupiter-API vorhandenen Annotationen eine Komposition von Annotationen zu Meta-Annotationen unterstützt. Ähnliche Mechanismen sind wieder aus dem Spring Framework oder auch aus der Programmiersprache Groovy bekannt.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
@Test
public @interface FastTest {
}
```

Figure 6. Meta-Annotationen durch Komposition anderer Annotationen

Im Beispiel werden besonders schnell ausführbare Tests durch die Meta-Annotation @FastTest gekennzeichnet, z.B. weil man sie aus Fast-Feedback-Gründen im Build zuerst ausführen möchte. Mit diesem Mechanismus lassen sich Möglichkeiten der Jupiter-API parametrisieren und zu höherwertigen Abstraktionen zusammenfassen, was der konsistenten Verwendung z.B. durch verschiedene Testautoren im Team dienen kann.

Nested Tests

JUnit Jupiter unterstützt direkt die Erstellung von geschachtelten Tests über die Annotation @Nested. Diese Fähigkeit ist grob vergleichbar mit dem aus JUnit 4 bekannten HierarchicalContextRunner.

```
@DisplayName("A stack")
public class TestingAStack {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() { new Stack<>(); }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void init() { stack = new Stack<>(); }

        @Test
        @DisplayName("is empty")
        void isEmpty() { Assertions.assertTrue(stack.isEmpty()); }
```

Figure 7. Nested Tests

Auch hier führt die Ausführung zu einem visuell besonders ansprechenden Ergebnis in der IDE, das in gewisser Weise schon an die spezifikationsnahen Darstellungen aus BDD-Frameworks erinnert.

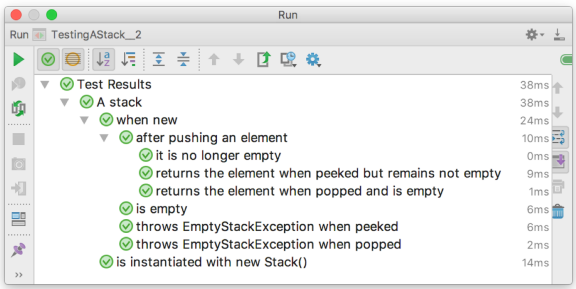

Figure 8. Ausführung von Nested Tests in der IDE

Dynamische Tests: @TestFactory und @ParameterizedTest

Während die grundlegenden Annotationen wie @Test oder @BeforeEach, @AfterAll etc im wesentlichen eine vergleichbare Semantik wie die entsprechenden JUnit 4-Vorgänger haben, so gibt es in JUnit 5 auch grundlegend neue Konzepte. Hierzu zählt insbesondere die Möglichkeit, dynamische Tests zu erstellen.

Neben den schon aus JUnit 4 in verschiedenen Varianten bekannten parametrisierten Tests stellen vor allem die Annotation @TestFactory eine besondere Neuerung dar. Mit dieser Annotation gekennzeichnete Methoden stellen selbst keine Tests dar, sondern liefern eine Stream (oder eine Collection) an Testfällen zurück. Dieser Stream entsteht erst zur Laufzeit, insbesondere ist seine Länge nicht vorab bekannt. Durch die in der JUnit Platform exponierten Listener ist die IDE aber dennoch in der Lage, für jeden Testfall innerhalb des Streams einen sauberen Knoten im Ausführungsbaum darzustellen und das in Verbindung mit dynamischen Namen. Lässt man z.B. das folgende Beispiel in einer IDE ablaufen, so sieht man, dass die einzelnen Knoten tatsächlich erst zur Laufzeit bei der Durchführung des gerade entstandenen Tests zur graphischen Darstellung hinzugefügt werden.

```
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

class DynamicFibonacciDemo {

    @TestFactory
    @DisplayName("all Fibonaccis are odd")
    Stream<DynamicTest> allFibonaccisAreOdd() {
        return IntStream.range(1, 13).boxed()
                .map(this::fibonacci)
                .map(number -> dynamicTest( displayName: "Fibonacci = " + number,
                        () -> isOdd(number))
                );
    }
```

Figure 9. Dynamische Tests im Code

Im Beispiel geht man von einem Stream natürlicher Zahl aus und bildet diesen mit Hilfe der Methodenreferenz 'fibonacci' auf Fibonacci-Zahlen ab. Diese Fibonacci-Folge wird dann über das statisch importierte `org.junit.jupiter.api.DynamicTest.dynamicTest(String displayName, Executable executable)` auf einen Stream von dynamischen Testfällen abgebildet. Auch wenn in diesem Beispiel bereits zum Zeitpunkt der Testerstellung die Folge auf 12 Testfälle begrenzt wird, so könnte man jederzeit ein erst zur Laufzeit berechnetes Abbruchkriterium verwenden.

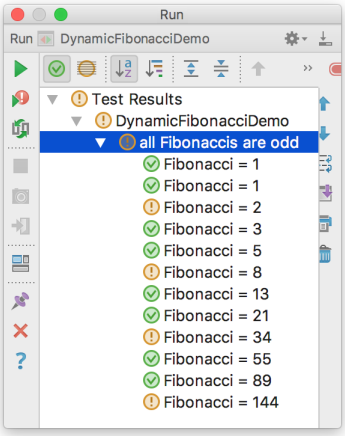

Figure 10. Ausführung Dynamischer Tests in der IDE

Eine solche Funktionalität lässt sich mit Hilfe von Lambda-Ausdrücken elegant realisieren. Die so gewonnene Ausdrucksmächtigkeit für dieses und andere Feature war bei den Vorüberlegungen für das neue JUnit 5 im Jahre 2015 mit ausschlaggebend dafür, die Abwärtskompatibilität zu Java 5 fallen zu lassen. Nicht umsonst trug die zur Anschubfinanzierung der Neuentwicklung initiierte Crowdfunding-Kampagne den Namen JUnit Lambda.

Erwähnenswert ist in diesem Zusammenhang, dass es für diese Art von dynamischen Tests explizit *keine* Ausführung von Lifecycle-Methoden auf der Ebene einzelner Testfälle gibt, sondern nur für die ganze @TestFactory-Methode. Möchte man also, dass z.B. @BeforeEach-Methoden für jeden einzelnen einer Menge von Testfällen aufgerufen werden, so sollte man die Annotation @ParameterizedTest verwenden. Dies Art von Test bietet weiter den Vorteil, dass die Werte, die einzelnen Testfälle parametrisieren, aus diversen vorhandenen (z.B. `CsvSource` oder `EnumSource`) oder selbst implementierbaren Quellen bezogen werden können. Im Gegensatz dazu werden diese Werte bei den zuvor behandelten dynamischen Tests typischerweise direkt im Code berechnet. Im folgenden Beispiel werden die Parameter der Einfachheit halber mit Hilfe der Annotation @MethodSource aus einer lokalen Methode gelesen. @ParameterizedTest ist eine Anwendung des allgemeineren TestTemplate-Konzepts, mit dessen Hilfe auch das verwandte `@RepeatedTest` umgesetzt ist.

```
@ParameterizedTest
@MethodSource("providerMethod")
void testWithParametersFromMethods(String parameter) {
    assertEquals( expected: "two", parameter);
}

static Iterable<String> providerMethod() {
    return asList("one", "two", "three");
}
```

Figure 11. Parametrisierte Tests im Code

Zusammenfassend lässt sich sagen, dass beide Varianten es ermöglichen, eine Gruppe von Testfällen zu erzeugen, deren Anzahl zuvor nicht bekannt ist. Die Auswahl einer Variante kann dadurch geleitet werden, ob man Lifecycle-Support auf Testfall-Ebene benötigt und ob die parametrisierenden Werte sich im Test leicht berechnen lassen oder aus externen Quellen bezogen werden müssen.

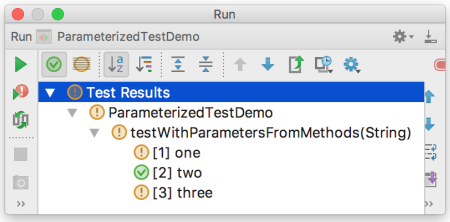

Figure 12. Ausführung Parametrisierter Tests in der IDE

Nach diesem Überblick über einige der interessantesten Neuerungen für Entwickler werden im Folgenden Gesamtarchitektur sowie Erweiterungsmöglichkeiten beschrieben.

Architektur: Platform, Testengines und Extensions

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Eines der Hauptziele bei der Neuentwicklung von JUnit 5 war die logische Trennung der Rollen zum Verfassen eines Tests und zur Ausführung von Tests. Physisch spiegelt sich das in der Entkopplung der APIs zum Schreiben von denen zum Ausführen von Tests wider. In JUnit 4 sind beide APIs in einem Artefakt enthalten. Alle, die JUnit auf irgendeine Weise verwenden, hängen von diesem einzigen jar-File ab, unabhängig davon, ob es sich dabei um IDEs, Build Tools, andere Test-Frameworks oder Testcode handelt. Diese Verwender benutzen teilweise Interna von JUnit, wie etwa interne Klassen oder private Instanzvariablen. So scheiterte z.B. der Versuch, die Version 4.12-beta-1 in Kombination mit einer bekannten IDE zu betreiben, weil die IDE eine private Variable, die umbenannt worden war, per Reflection referenziert hatte. Diese Art der Integration war der Tatsache geschuldet, dass es eben kein ausreichend mächtiges API für die Integration in Tools gab. Die Weiterentwicklung von JUnit 4 wurde dadurch immens erschwert, in Teilen sogar nahezu unmöglich gemacht.

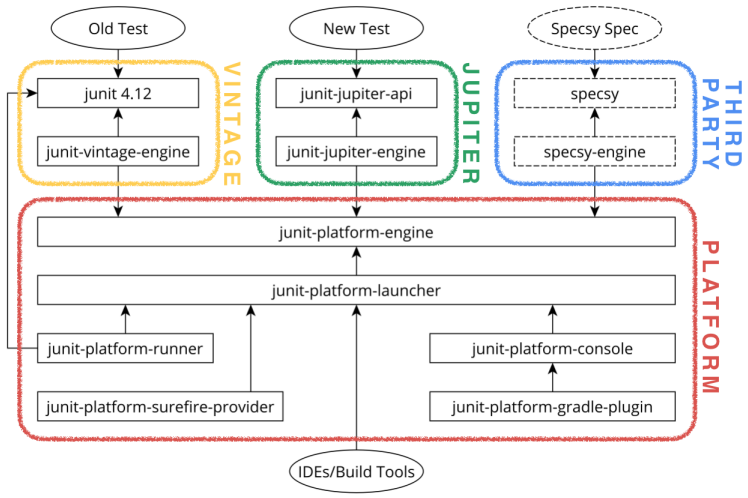

Figure 13. Gesamtarchitektur von JUnit 5

Wie das oben abgebildete Architekturdiagramm zeigt, wurde beim Entwurf von JUnit 5 großer Wert darauf gelegt, eine möglichst saubere Trennung der APIs für die unterschiedlichen Rollen der möglichen Verwender zu erreichen. So benötigt ein Testautor lediglich die Dependency *junit-jupiter-api*, die alle Annotations wie etwa `@Test`, Lifecycle-Annotationen und grundlegende Assertions mitbringt.

Tools, die eine JUnit 5-Integration anbieten wollen, wie etwa IDEs oder Build Tools verwenden das Modul *junit-platform-launcher* innerhalb der JUnit 5-Plattform, um die Ausführung von Tests anzustoßen. Der darin enthaltene *Launcher* orchestriert das Auffinden und die Ausführung von Tests durch verschiedene *Engines*.

Die Engine-Abstraktion ermöglicht die Koexistenz verschiedener Testframeworks bzw. -bibliotheken, und damit insbesondere auch die Ausführung von Testfällen auf Basis von JUnit 4 und JUnit 5 innerhalb eines einzigen Testlaufs. Dies impliziert, dass man bei existierenden Projekten auf Basis von JUnit 4 langsam mit einigen JUnit 5-Tests beginnen kann und keineswegs gezwungen ist, eine Big-Bang-Migration durchzuführen. Weitere Hilfen für die Migration wie etwa der Support ausgewählter Rules finden sich im Artefakt 'junit-jupiter-migrationsupport'. Um von einem integrierenden Tool angesprochen zu werden, müssen Test-Engines lediglich auf dem Klassenpfad vorhanden sein und sich über den normalen *ServiceLoader*-Mechanismus der JVM [4] beim Launcher registrieren.

Die Engine-Implementierungen hängen dabei nur vom Artefakt *junit-platform-engine* ab. JUnit 5 liefert zwei Engines aus: Die *junit-vintage-engine* dient dem Auffinden und der Ausführung von JUnit 4-Tests. Die *junit-jupiter-engine* hingegen implementiert das neue Programmiermodell von JUnit 5. Die Engine API ist dabei im Prinzip ein Angebot an existierende und zukünftige Test-Frameworks. Solche Frameworks müssen nur eine Engine-Implementierung bereitstellen, und profitieren sofort von der Integration, die die Tool-Hersteller für die JUnit 5-Plattform einmalig erstellt haben. Da eine Engine-Implementierung in vielen Fällen recht einfach zu erstellen ist [5], sinkt somit die Schwelle, ein sauber integriertes Test-Framework einem weiteren Benutzerkreis zur Verfügung zu stellen. Einige Projekte haben bereits gegen Milestone-Releases von JUnit 5 eigene Engines implementiert, z.B. die jqwik-Engine für Property-Based Testing [6] oder das BDD-Framework Specsy [7].

Zusammenfassend lässt sich sagen, dass das neue JUnit 5 erstmalig eine Plattform-Abstraktion zur Integration bereitstellt, die sauber von Test-Engines getrennt ist, die ihrerseits ein bestimmtes Programmiermodell für Tests unterstützen. Die JUnit-Jupiter-Engine realisiert dabei das neue JUnit 5-Programmiermodell, die JUnit-Vintage-Engine wird aus Gründen der Abwärtskompatibilität und Migrationsunterstützung mitgeliefert. Alle Beispiele und neuen Features in diesem Artikel beziehen sich auf die neue JUnit-Jupiter-Engine.

Extensions

Im Folgenden soll überblicksartig das neue Extensionmodell vorgestellt werden. Im Gegensatz zu den zuvor beschriebenen Testengines, die es ermöglichen, beliebige Testdefinitionen und Ausführungssemantiken zu unterstützen, dient das im Folgenden beschriebene Extensionmodell zur Verfeinerung der Jupiter-Engine.

Das hybride Erweiterungsmodell von JUnit 4 aus Runner und Rule hatte sich über die Jahre hinweg insbesondere im Bereich der Komponierbarkeit als problematisch herausgestellt. Daher wurde bei JUnit 5 viel Energie darauf verwendet, ein sowohl einheitliches als auch ausdrucksmächtiges Erweiterungsmodell für die Jupiter-Engine zu entwerfen und umzusetzen. Es handelt sich dabei um ein feingranulares Erweiterungsmodell, das einen selektiven Eingriff an vielen Stellen der Testdefinition und -ausführung erlaubt. Insbesondere sind die Erweiterungen aus technischer Sicht beliebig kombinierbar. Diesem Thema ließe sich leicht ein ganzer Artikel widmen, hier soll nur kurz an Hand eines Beispiels die Grundidee erläutert werden.

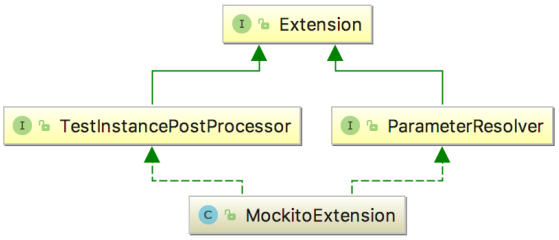

Figure 14. Klassendiagramm zu Extensions

Als Basis aller Erweiterungen für JUnit Jupiter gibt es das Marker-Interface 'Extension'. Nehmen wir nun an, dass eine einfache Extension zur Integration des Mockframeworks Mockito erstellt werden soll. Konkret müsste eine solche 'MockitoExtension' die beiden ExtensionPoints 'TestInstancePostProcessor' und 'ParameterResolver' implementieren, wenn man die Extension dann wie in obiger Abbildung verwenden will. Beide ExtensionPoints erweitern wiederum das genannte Marker-Interface.

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

    @BeforeEach
    void setup(@Mock List<String> list) {...}

    @Test
    void aTest(@Mock List<String> list) {...}

}
```

Figure 15. Einfache Mockito-Extension

In diesem Beispiel wird der ExtensionPoint 'TestInstancePostProcessor' benötigt, um über das Mockito-API die Mocks direkt nach Erstellung der Testinstanz zu initialisieren. Der ExtensionPoint 'ParameterResolver' hingegen dient dazu, die so erzeugten Mocks elegant in die gewünschten Test- oder Lifecycle-Methoden hineinreichen zu können, was präzise ausdrücken lässt, für welche Methoden welche Mocks bestimmt sind.

Neben den beiden hier genannten ExtensionPoints gibt es eine ganz Reihe anderer, etwa zur Handhabung von Ausnahmen und insbesondere zur Interaktion mit den benutzerdefinierten

Methoden des Test-Lifecycle (z.b. @BeforeAll und BeforeAllCallback). Benötigt man in einer Extension einen Zustand über verschiedene Aufrufe hinweg, so sollte man diesen im bereitgestellten `ExtensionContext.Store` ablegen, um Probleme bei der Komposition von Extensions zu vermeiden.

Zusammenfassend lasst sich sagen, dass man entweder auf einer sehr hohen Granularitätsebene eine eigene Testengine implementieren kann oder aber die Standard-Testengine 'Jupiter' durch feingranulare und komponierbare Extensions erweitern kann. Eine Kombination aus eigener Engine und Jupiter-Extensions funktioniert natürlich auch in JUnit 5 nicht. Dies liegt aber in der Natur der Sache, da ein Erweiterungsmodell notwendigerweise die Abstraktionen einer konkreten Engine manipulierbar machen muss und sich nicht sinnvoll über alle denkbaren Engines hinweg verallgemeinern lässt.

Integration und Kompatibilität

Wesentlich für den Erfolg eines Test-Frameworks sind nicht nur moderne oder elegante Features, sondern vor allem auch die Integration in andere Werkzeuge. Die Ausführung von Tests innerhalb einer IDE wurde oben bereits ausführlich gezeigt, mindestends ebenso wichtig ist sicherlich die Integration mit CLI-basierten Buildtools, die in CI-Builds zum Einsatz kommen.

Buildtools

Die meisten Projekte im Java-Umfeld werden heute sicherlich mit Maven oder Gradle gebaut. Daher hat das JUnit 5-Team von Anfang an ein Gradle Plugin [Referenz] und einen Maven Surefire Provider [Referenz] mitentwickelt. Der Surefire Provider wurde bereits an das entsprechende Apache-Projekt übergeben, um dort weiterentwickelt zu werden; für das Gradle Plugin ist ein ähnliches Vorgehen in Vorbereitung. Möchte man eine JUnit 5-Projekt mit Maven oder Gradle neu aufsetzen, so lohnt sich ein Blick auf die entsprechenden Beispielprojekte [8].

Zusätzlich zu dieser Buildtool-Integration gibt es einen `ConsoleLauncher`, der mit zahlreichen Optionen parametrisierbar ist und das Ergebnis einer Testausführung in einer wohlformatierten Baumdarstellung ausgibt. Dazu gitb es im zentralen Maven-Repository unter der Dependency `junit-platform-console-standalone` ein stand-alone Executable, was direkt mit `java -jar` aufgerufen werden kann. Mit diesem Werkzeug wäre auch eine Integration in nicht direkt unterstützte Buildtools (z.B. Ant) realisierbar.

JUnit 4 und 5 (Jupiter)

Wie bereits im Abschnitt über die Architektur erwähnt, war es ein wesentliches Designziel, mehrere Engines in einem einzigen Lauf gleichzeitig ausführen zu können. Dies gilt natürlich insbesondere für die Koexistenz von JUnit 4- und Junit Jupiter-Tests. Für die Ausführung von Junit 4-Tests innerhalb einer neuen JUnit 5-Umgebung existiert die bereits beschriebene JUnit-Vintage-Engine. Umgekehrt lassen sich neue Jupiter-Tests in JUnit 4-Umgebungen ausführen, wenn man den entsprechenden Runner `org.junit.platform.runner.JUnitPlatform` aus der Dependency `junit-platform-runner` verwendet, der das Jupiter-Programmiermodell bestmöglich auf die in JUnit 4 vorhandenen

Abstraktionen übersetzt. Auch wenn beide Integrationen eher nicht für eine dauerhafte Verwendung vorgesehen sind, so lassen sich doch einige Migrationsjahre damit bestreiten.

Java 9

Seit Version 5.0.0 kann JUnit 5 zusammen mit Java 9 verwendet werden, die CI-Builds laufen schon seit einigen Monaten sowohl gegen Java 8 als auch Java 9. Die Modulinformation wird derzeit noch automatisch generiert, d.h. dass im Moment noch alle Klassen physisch verwendbar sind. Um für die Zukunft gewappnet zu sein, sollte man als Endanwender jedoch bereits jetzt auf die Verwendung von Klassen verzichten, die mit `@API(Status.INTERNAL)` annotiert sind.

Fazit

An ein Testframework werden hohe Anforderung in Bezug auf Stabilität und Kompatibilität gestellt, das gilt insbesondere für ein so lange existierendes und so weit verbreitetes Framework wie JUnit. Das JUnit 5-Team hat von Anfang versucht, diese besonderen Anforderungen in Einklang mit den nicht zuletzt durch Java 8 ermöglichten Innovationen zu bringen. Wichtiger noch als eine größere Eleganz bei der Testdefinition oder die vereinfachte Entwicklung von Erweiterungen ist möglicherweise sogar die grundlegend neu konzipierte Architektur, die erstmals zwischen allgemeiner und stabiler Plattform einerseits und verschiedenen Engines andrerseits unterscheidet. Nicht zuletzt ermöglicht die Plattform eine saubere und mächtige Einbindung belieber Engines in IDEs und Build-Tools.

Ein großes Dankeschön geht an die Kollegen Marc Philipp, Sam Brannen, Christian Stein und Stefan Bechtold vom JUnit 5-Team sowie an Johannes Link, der nicht nur diesen Artikel korrekturgelesen hat, sondern vor allem auch die Überlegungen zur grundlegend neuen Architektur wesentlich mitgestaltet hat. Dank gebührt auch den vielen frühen Anwendern, die die Entwicklung von JUnit 5 in den letzten zwei Jahren durch Feedback und Diskussionen begleitet haben.

Links & Literatur

[1]

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

[2]

Henn Idan: The Top 100 Java Libraries in 2017 - Based on 259,885 Source Files, http://blog.takipi.com/the-top-100-java-libraries-in-2017-based-on-259885-source-files/

[3]

GitHub Issue: https://github.com/junit-team/junit5/issues/163

[4]

JavaDoc java.util.ServiceLoader: https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html

[5]

Johannes Link und Matthias Merdes: JUnit-5-Test-Engine selbst gemacht, https://jaxenter.de/junit-5-test-engine-selbst-gemacht-62472

[6]

Johannes Link: Property-Based Testing in Java, http://jqwik.net/

[7]

Specsy Unit Testing Framework, http://specsy.org/

[8]

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