JUnit 5
Design und Architektur eines Frameworks

Software Architecture Summit

@johanneslink

johanneslink.net

Softwaretherapeut

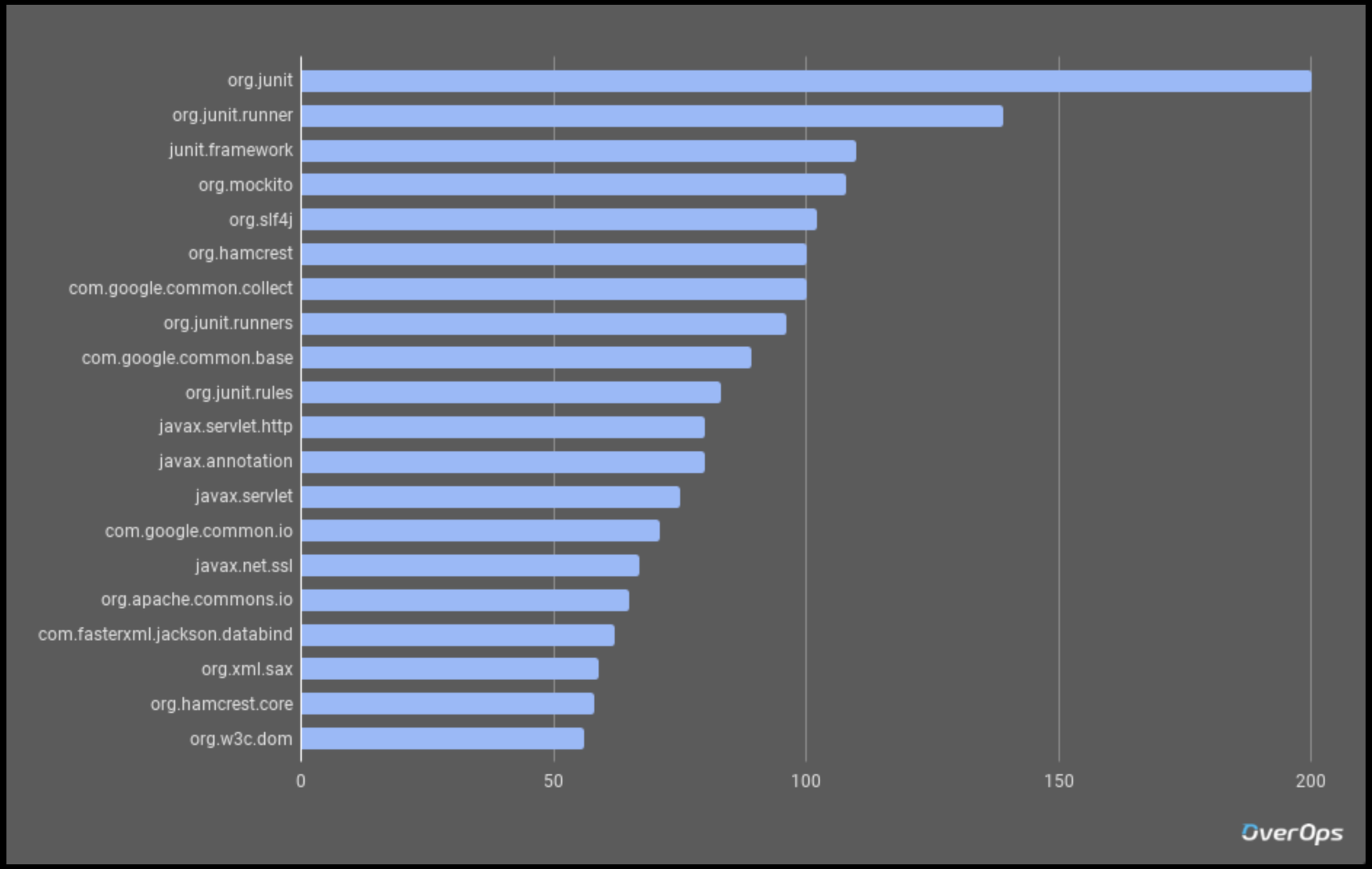
"In Deutschland ist die Bezeichnung Therapeut allein oder ergänzt mit bestimmten Begriffen gesetzlich nicht geschützt und daher kein Hinweis auf ein erfolgreich abgeschlossenes Studium oder auch nur fachliche Kompetenz." Quelle: Wikipedia

- JUnit 5 Initiator - zusammen mit Marc Philipp - und Core-Committer im ersten Jahr
- jqwik: Externe JUnit5-Test-Engine für Property-Based Testing

themenübersicht

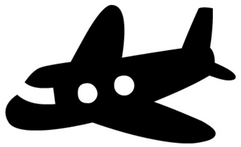
- JUnit 5: Vision einer Plattform
- Architektur der Plattform
- Wie gut funktioniert die Vision?
- Extensibility: Zwei Ansätze im Vergleich

Warum braucht die Welt ein neues JUnit?



http://blog.takipi.com/the-top-100-java-libraries-in-2017-based-on-259885-source-files/

A Brief History of JUnit

1997 JUnit 1.0 

2006 JUnit 4.0 Runner

2009 JUnit 4.7 Rules

2015 JUnit Lambda Campaign

2017 JUnit 5.0.0

Wartbarkeit

ComparisonFailure.java

```
12 public class ComparisonFailure extends AssertionError {
13     /**
14      * The maximum length for expected and actual strings.
15      *
16      * @see ComparisonCompactor
17      */
18     private static final int MAX_CONTEXT_LENGTH = 20;
19     private static final long serialVersionUID = 1L;
20
21     /*
22      * We have to use the f prefix until the next major re
23      * serialization compatibility.
24      * See https://github.com/junit-team/junit/issues/976
25      */
26     private String fExpected;
27     private String fActual;
```

4.11

```
Done: 0 of 1  Failed: 1  (6.853 s)

expected:<[Expected] message> but was:<[Actual] messa
Expected :Expected message
Actual   :Actual message
 <Click to see difference>

org.junit.ComparisonFailure: expected:<[Expected] mes
    at ExampleTest.failingTest(ExampleTest.java:9) <
```

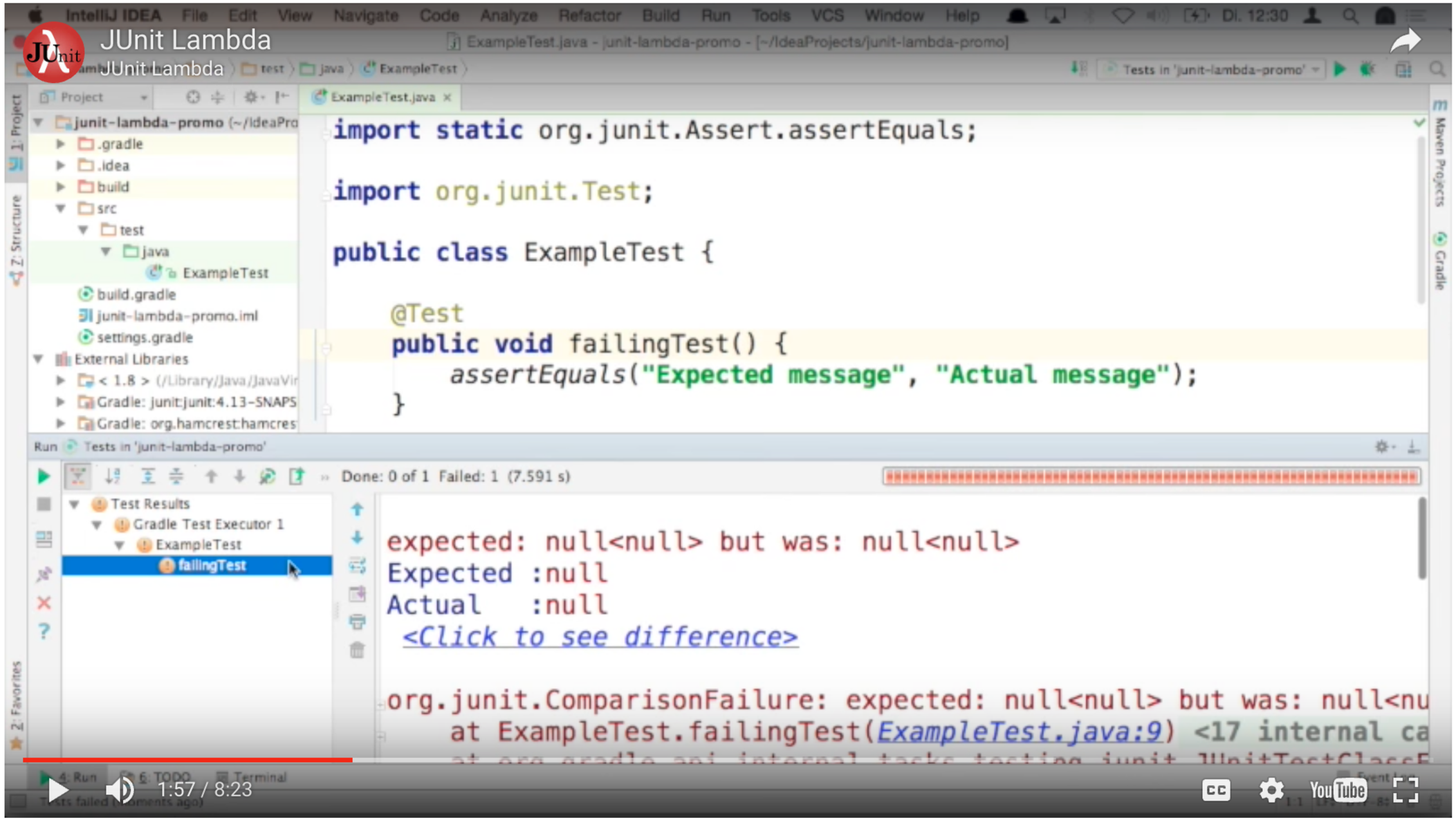
ComparisonFailure.java

```
12 public class ComparisonFailure extends AssertionError {
13     /**
14      * The maximum length for expected and actual strings.
15      *
16      * @see ComparisonCompactor
17      */
18     private static final int MAX_CONTEXT_LENGTH = 20;
19     private static final long serialVersionUID = 1L;
20
21     /*
22      * We have to use the f prefix until the next major re
23      * serialization compatibility.
24      * See https://github.com/junit-team/junit/issues/976
25      */
26     private String expected;
27     private String actual;
```

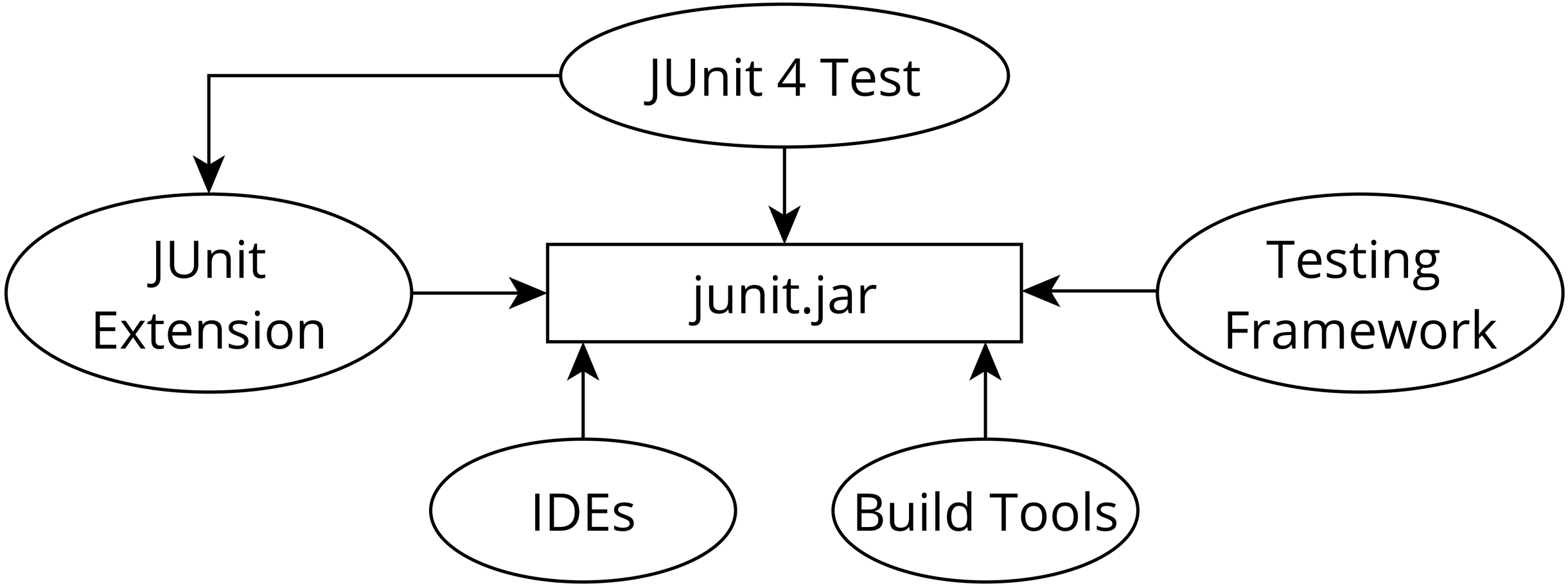
4.12-beta-1

```
Done: 0 of 1  Failed: 1  (7.591 s)

expected: null<null> but was: null<null>
Expected :null
Actual   :null
 <Click to see difference>

org.junit.ComparisonFailure: expected: null<null> but
    at ExampleTest.failingTest(ExampleTest.java:9) <1
```

Crowdfunding JUnit Lambda #fundJUnit



JUnit 4 "Architektur"



Erfolg von JUnit 4 als Plattform verhindert Weiterentwicklung von JUnit 4 als Werkzeug!

JUnit-5 Design-Ziel

- Trennung der Aspekte
  JUnit als Testwerkzeug und
  JUnit als Plattform
- JUnit 4 und 5 nebeneinander,
  um Adaption und Migration zu erleichtern

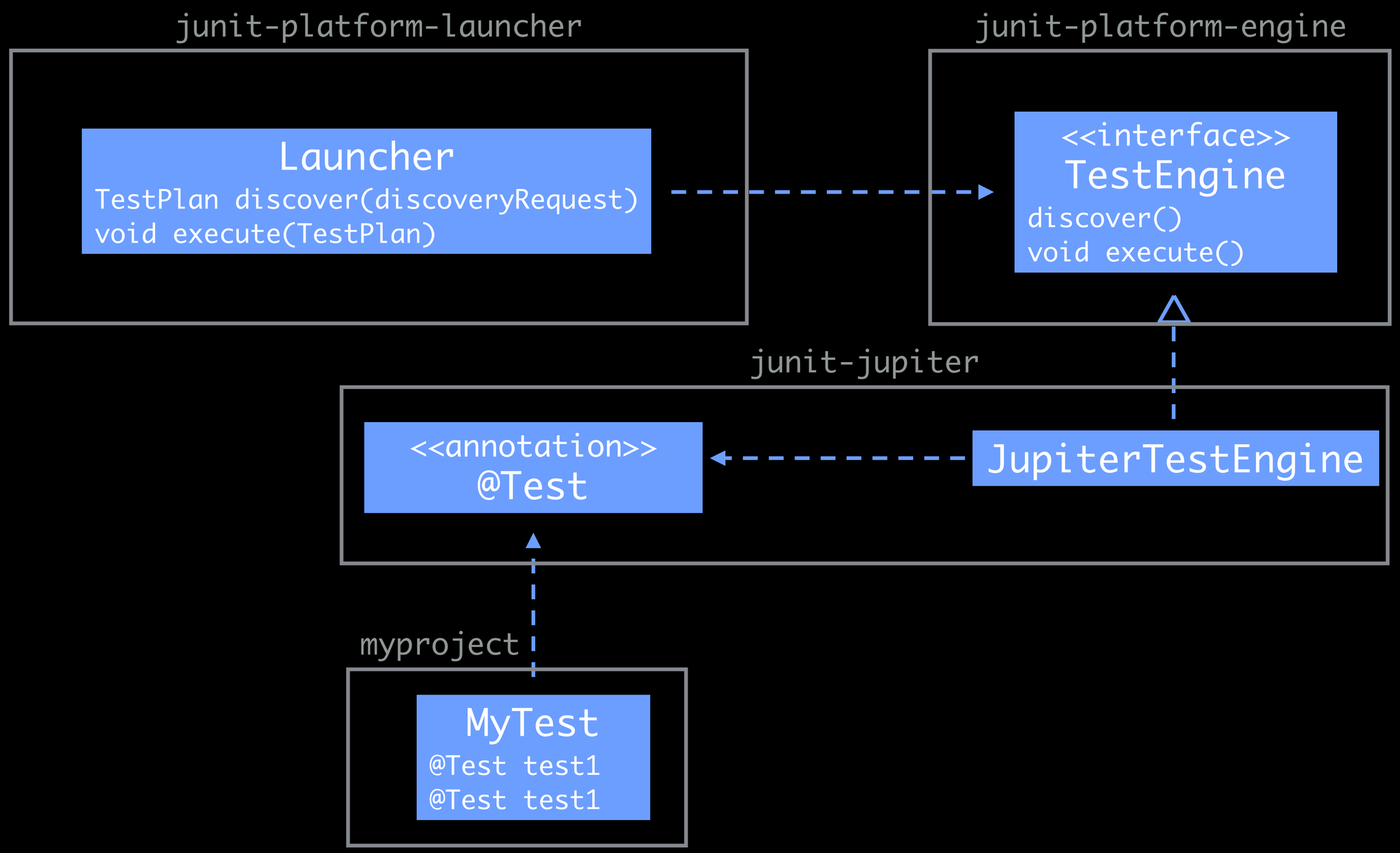
DIY Architecture: Step by Step

Allgemein erwünschte Eigenschaften eines Frameworks

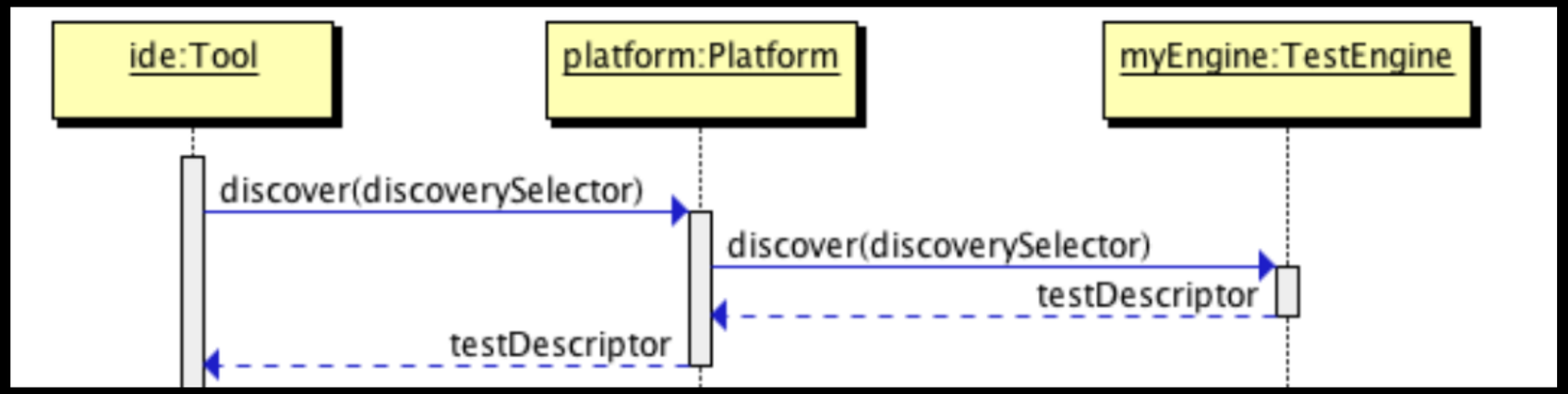
- Usability
- Separation of Concerns
- Freie Kombinierbarkeit mit anderen Frameworks und Bibliotheken
- Erweiterbarkeit
- Abwärts- und Aufwärtskompatibilität
- Keine oder wenig Abhängigkeiten

Schritt 1: Trennung von Framework-Nutzung und Framework-Anbindern

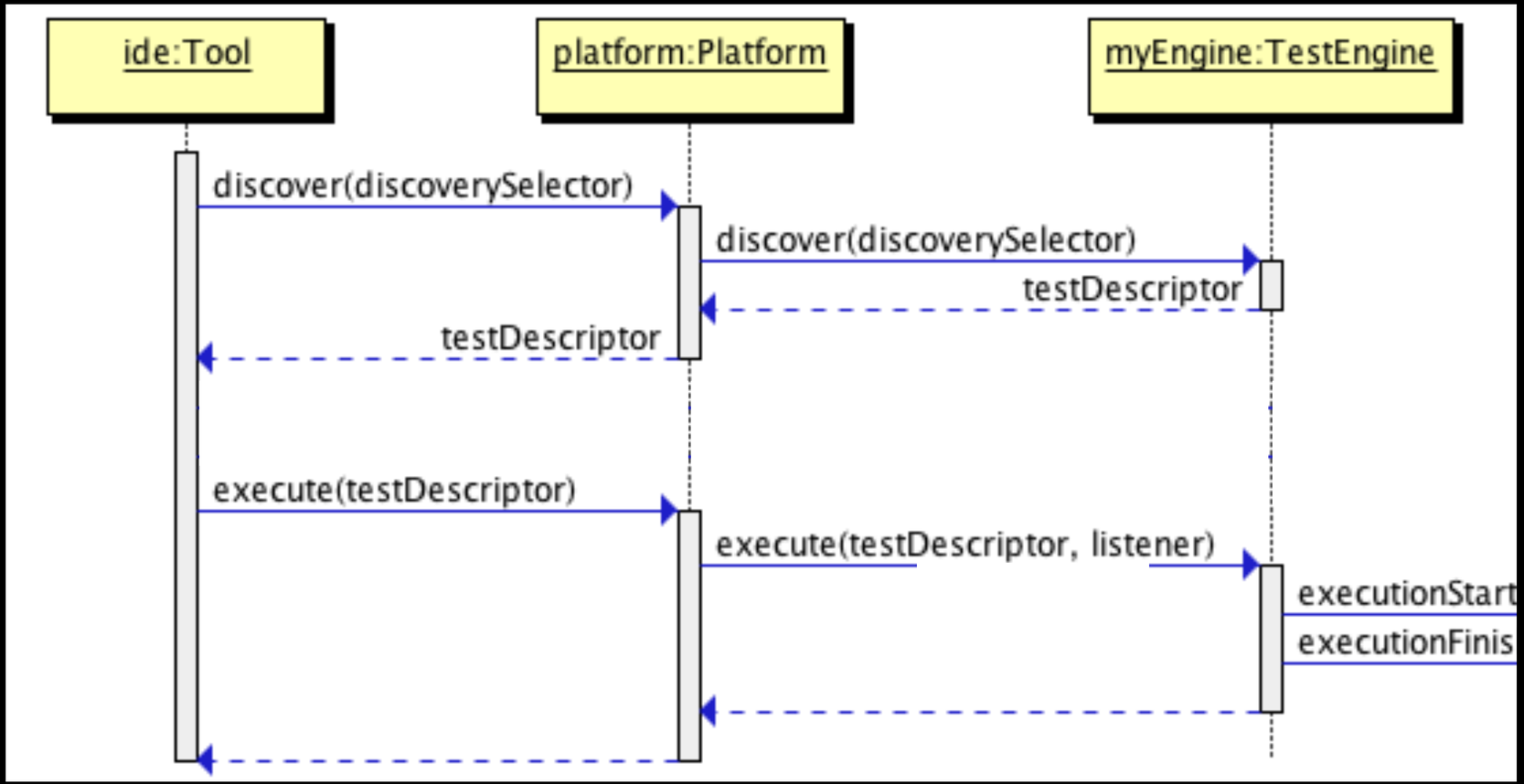
- Nutzung (API):
  - IDE: Anzeigen, Auswahl und Anstoßen von Tests
  - Programmierer: Schreiben/Spezifizieren von Tests
- Anbinder (SPI)
  - Interpretation von Testspezifikationen


junit-platform-launcher
Launcher
TestPlan discover(discoveryRequest)
void execute(TestPlan)
junit-platform-engine
<<interface>>
TestEngine
discover()
void execute()
junit-jupiter
<<annotation>>
@Test
JupiterTestEngine
myproject
MyTest
@Test test1
@Test test1

Trennung von Entdecken und Ausführen

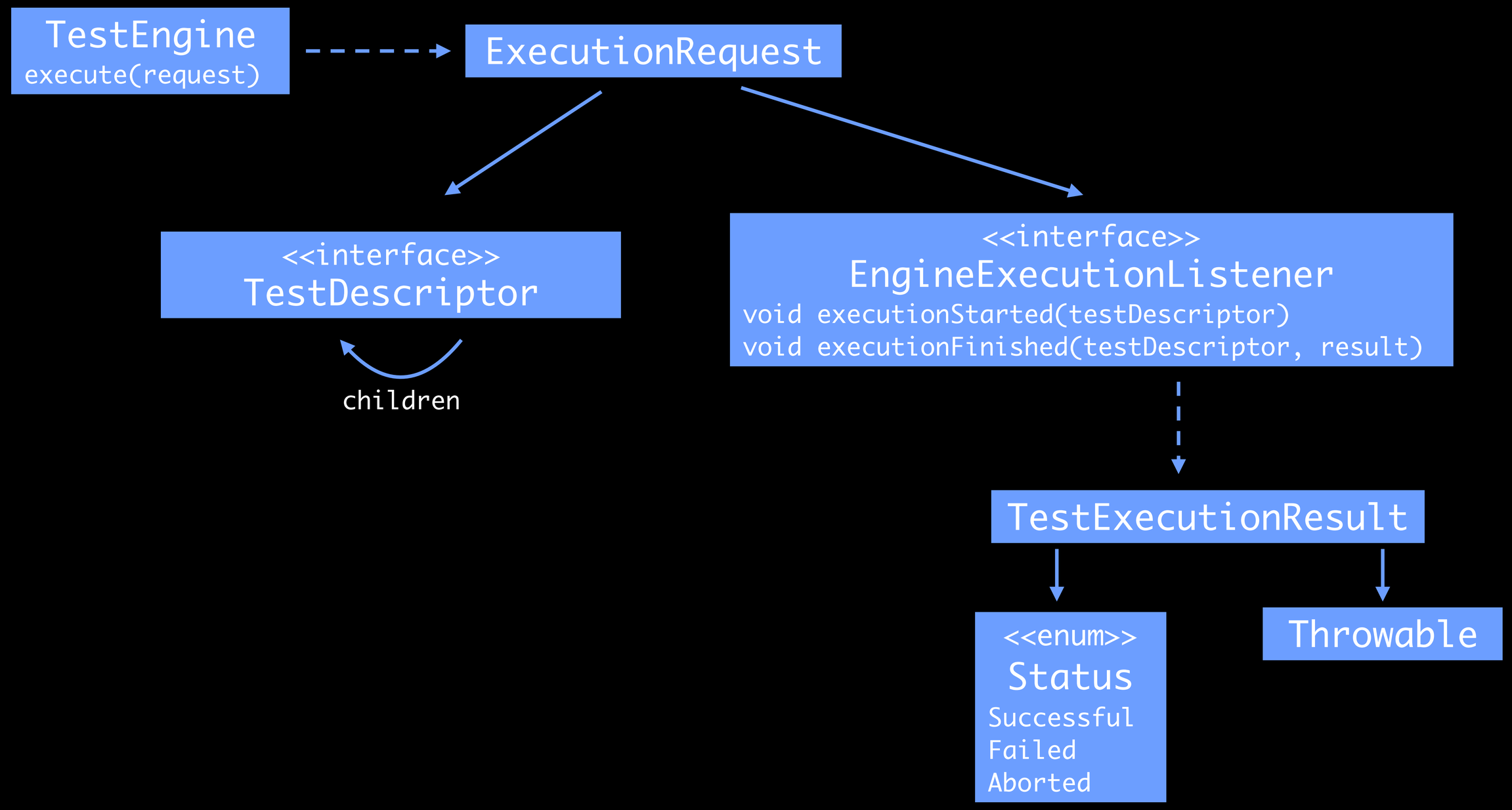


Trennung von Entdecken und Ausführen



Schritt 2: Ständiger Fortschrittsbericht eines Testlaufs

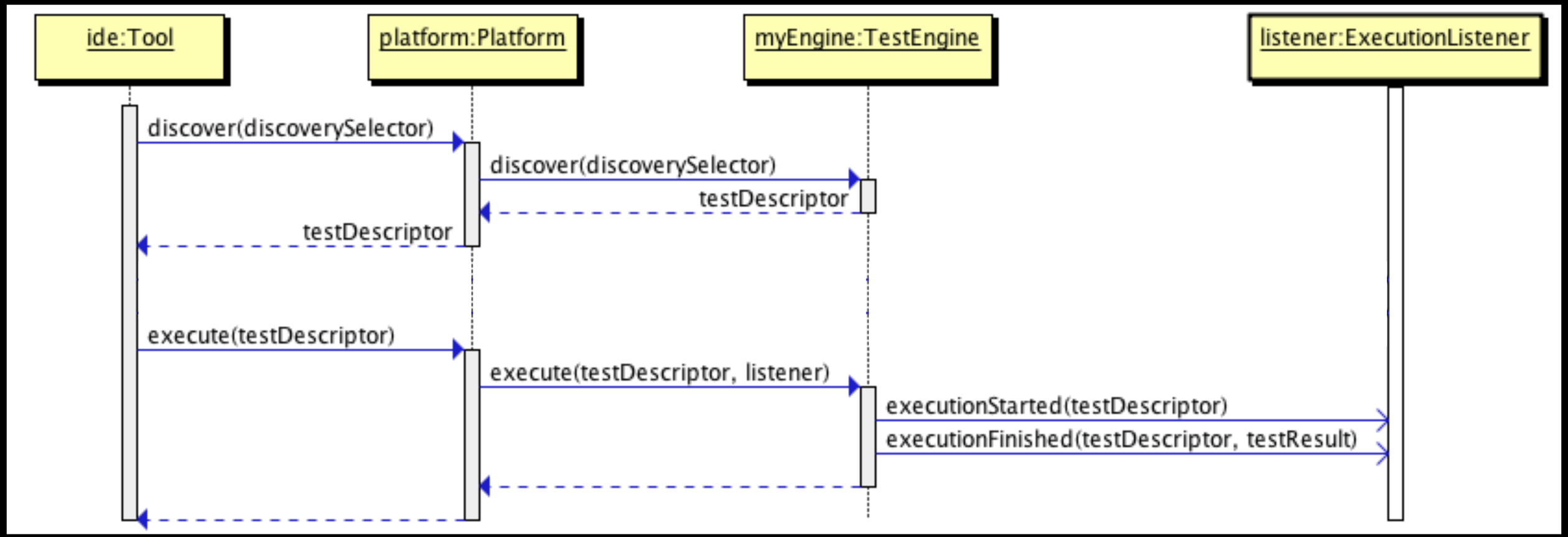
- Events / Nachrichten statt Rückgabe eines Ergebnisses


TestEngine
execute(request)
ExecutionRequest
<<interface>>
TestDescriptor
children
<<interface>>
EngineExecutionListener
void executionStarted(testDescriptor)
void executionFinished(testDescriptor, result)
TestExecutionResult
<<enum>>
Status
Successful
Failed
Aborted
Throwable

EngineExecutionListener	
dynamicTestRegistered(TestDescriptor)	void
executionSkipped(TestDescriptor, String)	void
executionStarted(TestDescriptor)	void
executionFinished(TestDescriptor, TestExecutionResult)	void
reportingEntryPublished(TestDescriptor, ReportEntry)	void

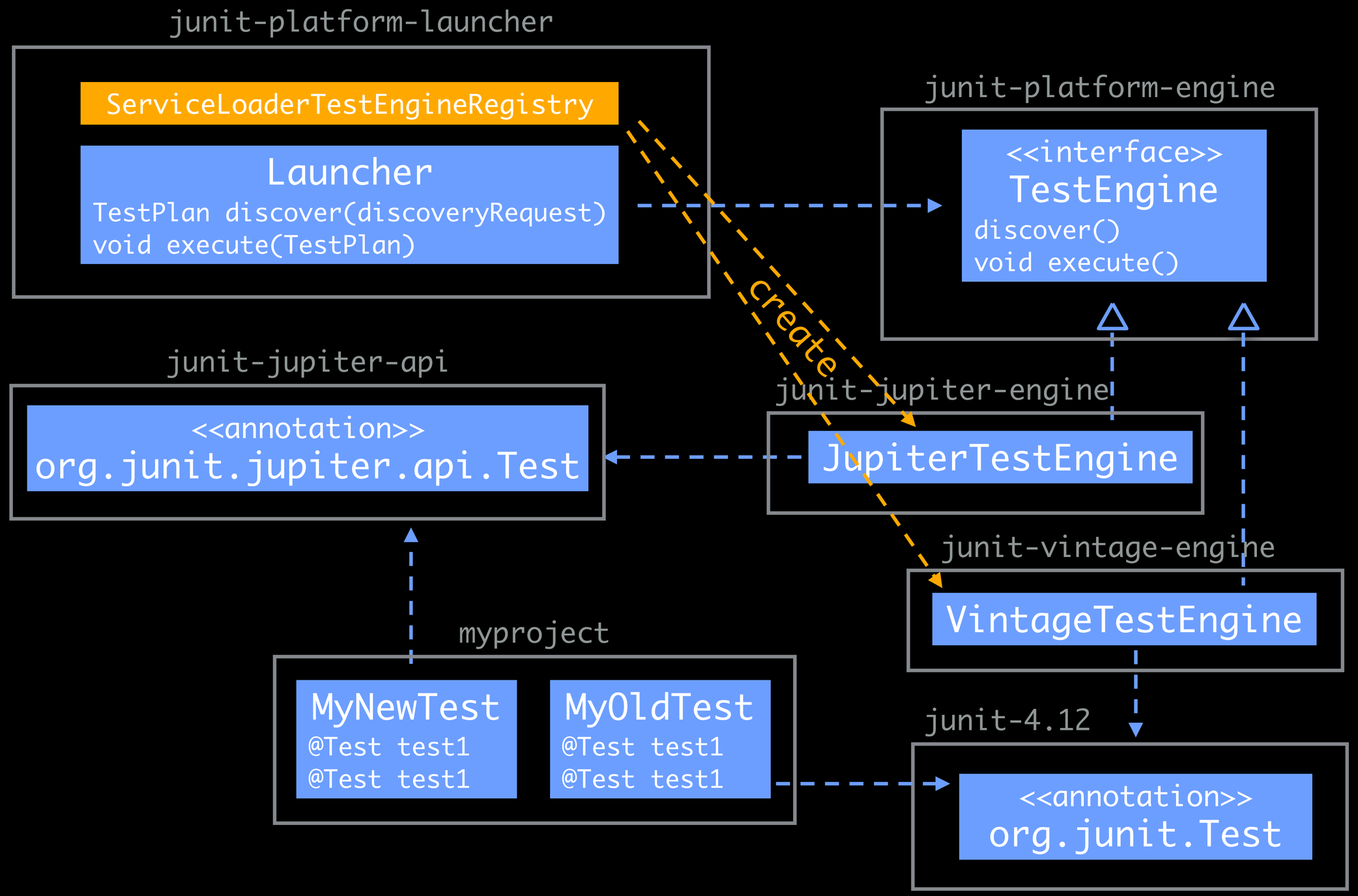
TestExecutionResult	
successful()	TestExecutionResult
aborted(Throwable)	TestExecutionResult
failed(Throwable)	TestExecutionResult
getStatus()	Status
getThrowable()	Optional<Throwable>
toString()	String

Execution Listener



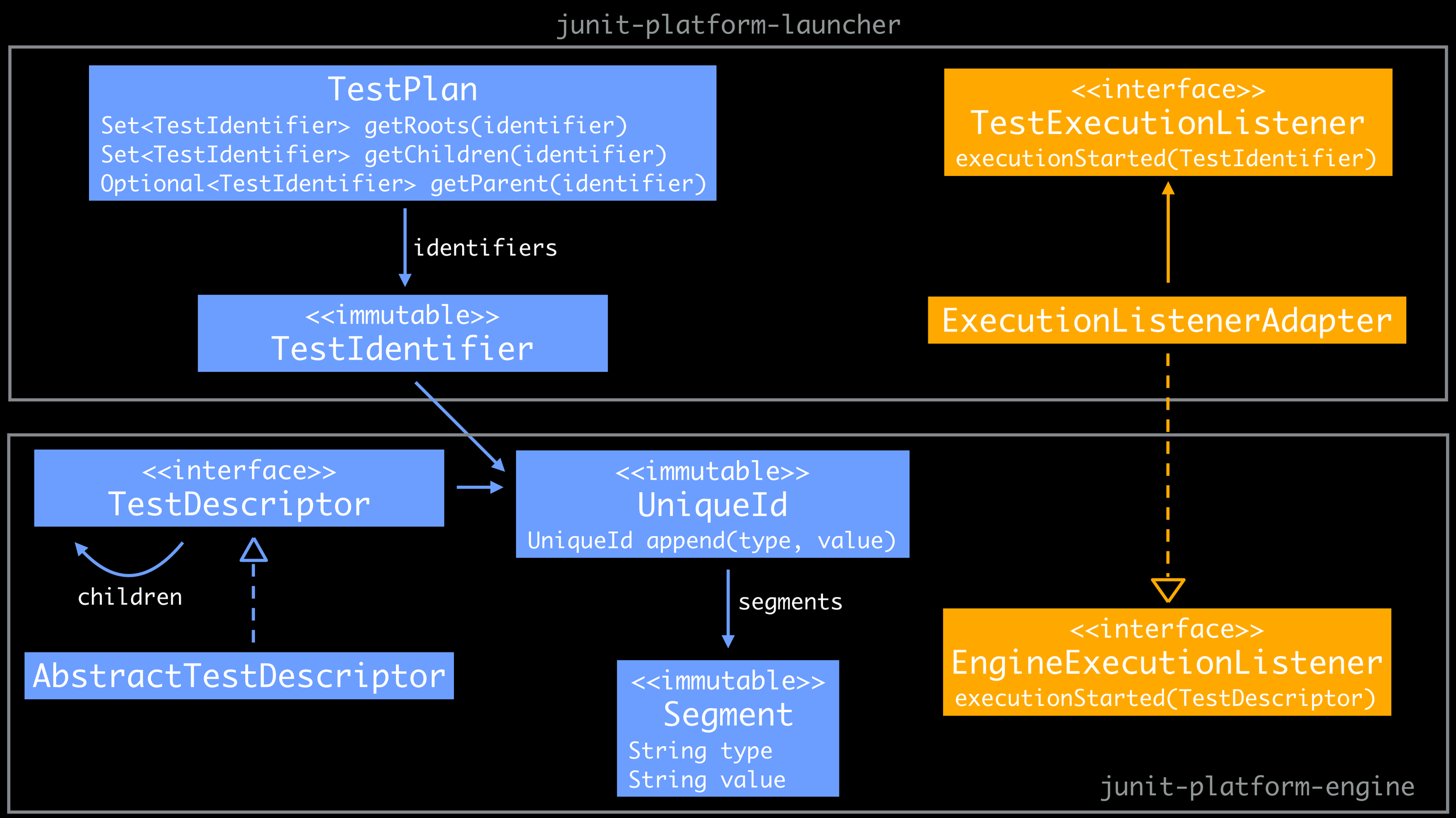
Schritt 3: Gleichzeitiger Einsatz mehrerer Test-Engines

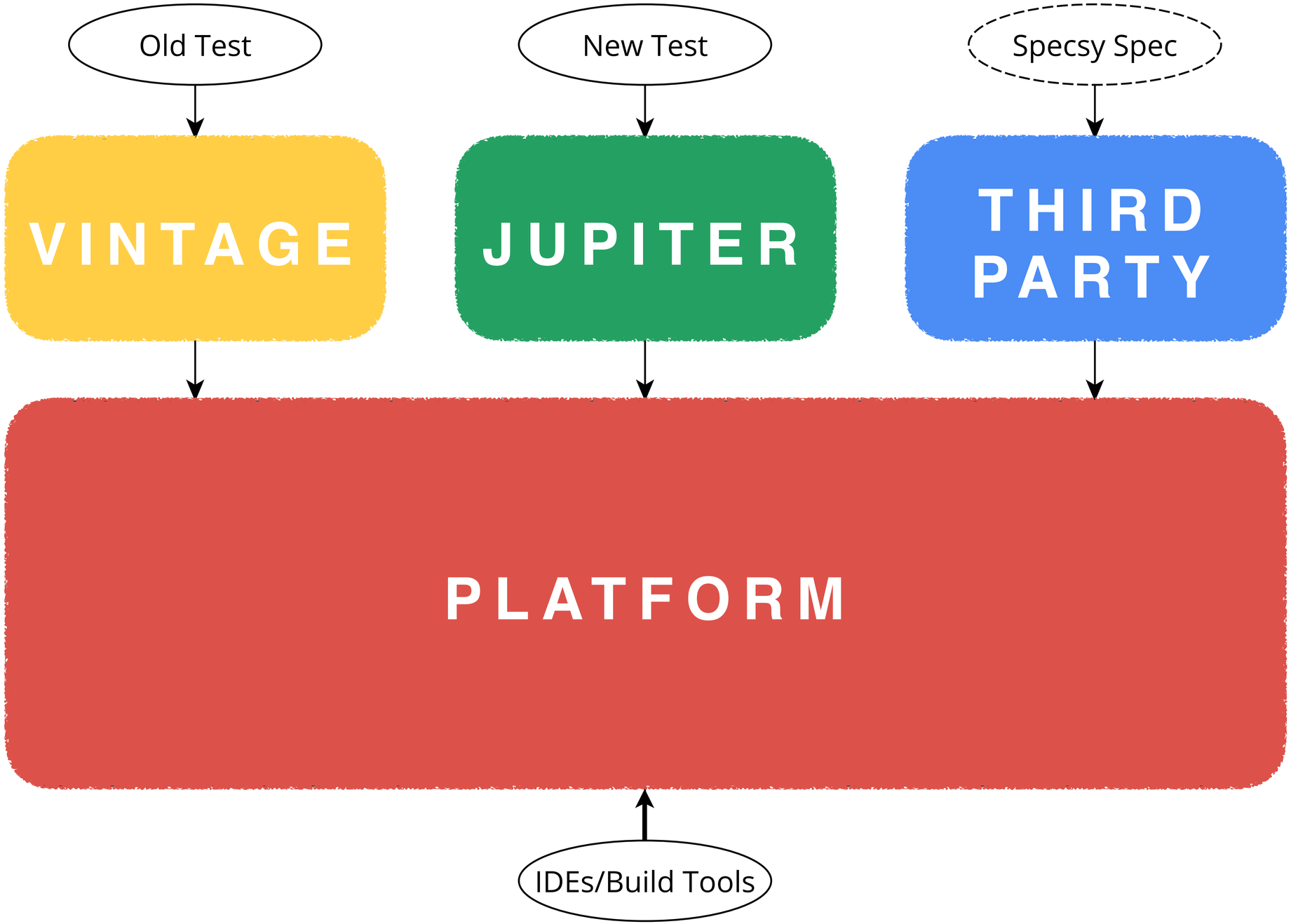
- Jede Test-Engine hat ihre eigene API zur Testfallspezifikation
- Sevice-Provider-Mechanismus von Java zur Registrierung einer Test-Engine

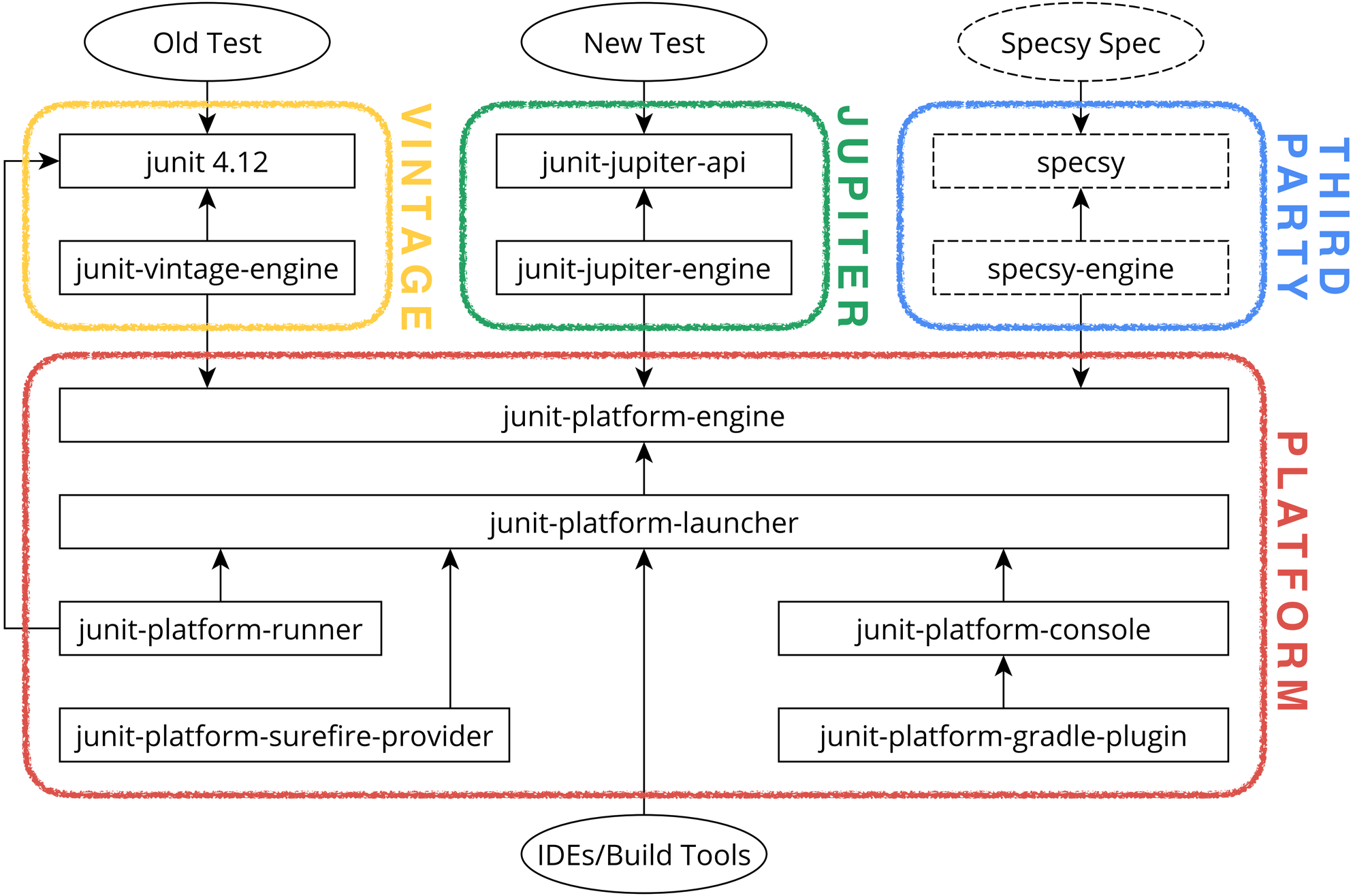

junit-platform-launcher
ServiceLoaderTestEngineRegistry
Launcher
TestPlan discover(discoveryRequest)
void execute(TestPlan)
junit-platform-engine
<<interface>>
TestEngine
discover()
void execute()
create
junit-jupiter-api
<<annotation>>
org.junit.jupiter.api.Test
junit-jupiter-engine
JupiterTestEngine
junit-vintage-engine
VintageTestEngine
myproject
MyNewTest
@Test test1
@Test test1
MyOldTest
@Test test1
@Test test1
junit-4.12
<<annotation>>
org.junit.Test

Schritt 4: Leichte Serialisierbarkeit aller SPI-Objekte

- Notwendig um Inter-Prozess-Kommunikation zur ermöglichen
- `UniqueId` als verbindendes Element
- TestDescriptor vs Testidentifier und TestPlan


junit-platform-launcher
TestPlan
Set<TestIdentifier> getRoots(identifier)
Set<TestIdentifier> getChildren(identifier)
Optional<TestIdentifier> getParent(identifier)
identifiers
<<immutable>>
TestIdentifier
<<interface>>
TestExecutionListener
executionStarted(TestIdentifier)
ExecutionListenerAdapter
<<interface>>
TestDescriptor
children
AbstractTestDescriptor
<<immutable>>
UniqueId
UniqueId append(type, value)
segments
<<immutable>>
Segment
String type
String value
<<interface>>
EngineExecutionListener
executionStarted(TestDescriptor)
junit-platform-engine


Old Test
New Test
Specsy Spec
VINTAGE
JUPITER
THIRD PARTY
PLATFORM
IDEs/Build Tools


Old Test
New Test
Specsy Spec
junit 4.12
junit-vintage-engine
VINTAGE
junit-jupiter-api
junit-jupiter-engine
JUPITER
specsy
specsy-engine
THIRD PARTY
junit-platform-engine
junit-platform-launcher
junit-platform-runner
junit-platform-surefire-provider
junit-platform-console
junit-platform-gradle-plugin
PLATFORM
IDEs/Build Tools

Andere Architekturaspekte

- API Lifecycle-Management mit `apiguardian`: `@API(status=Stable)`
  - Mögliche Status:
    Stable, Maintained, Experimental, Deprecated, Internal
- Test auf Zyklenfreiheit mit `degraph`
- Strikte Formattierungsregeln via `checkstyle`

Platform Usability

Test-Engines nutzen und selbst entwickeln

Warum braucht die Welt mehr als zwei Test-Engines?

- Andere JVM-Sprache
- Anderes Spezifikationsmodell
- Anderes Ausführungsmodell

- Für *Erweiterungen* des Test-Modells genügt häufig eine Jupiter-Extension

Test-Engine benutzen

1. TestCompile-Dependency hinzufügen:

   `org.myorg:my-engine:x.y.z`

2. Tests schreiben

Test-Engine benutzen

1. TestCompile-Dependency hinzufügen:

   `org.myorg:my-engine:x.y.z`

   `org.junit.jupiter:junit-jupiter-engine:5.0.0`

2. Tests schreiben

Test-Engine Kochrezept

1. Compile-Dependencies hinzufügen
2. `TestEngine`-Interface implementieren
3. Engine registrieren
4. Tests in IDE starten

DEMO

Die kleinste Test-Engine der Welt

http://github.com/jlink/jax2017

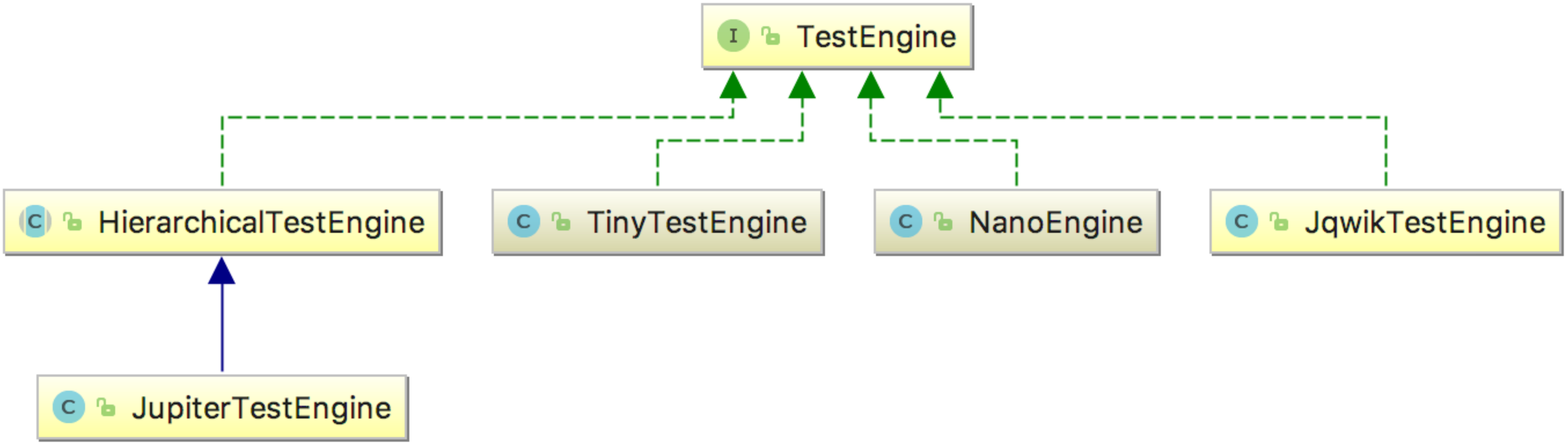
```
git clone https://github.com/jlink/jax2017.git
cd jax2017/empty-engine/
gradle test
```

```
dependencies {
  compile("org.junit.platform:junit-platform-engine:1.0.0“)
  compile("org.junit.platform:junit-platform-commons:1.0.0")

  // For writing integration tests
  testCompile("org.junit.platform:junit-platform-launcher:1.0.0")

  // Only necessary to enable IntelliJ support
  testRuntime("org.junit.jupiter:junit-jupiter-engine:5.0.0")
}
```

Test-Engine registrieren

/META-INF/services/
org.junit.platform.engine.TestEngine

```
nano.NanoEngine
```


TestEngine
HierarchicalTestEngine
TinyTestEngine
NanoEngine
JqwikTestEngine
JupiterTestEngine

TestEngine-Interface implementieren

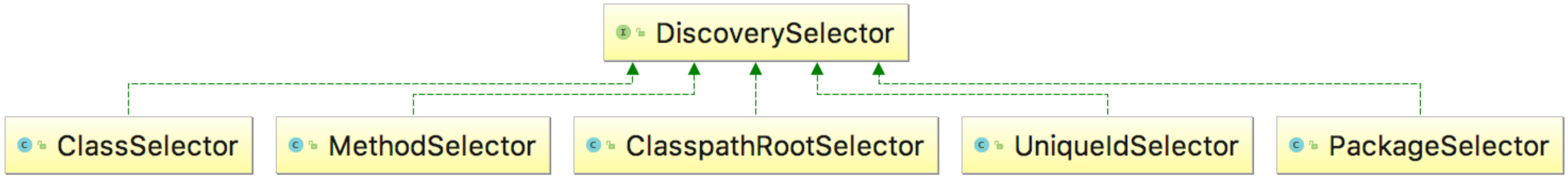
1. Create TestEngine class
2. Implement discover()
3. Implement execute()

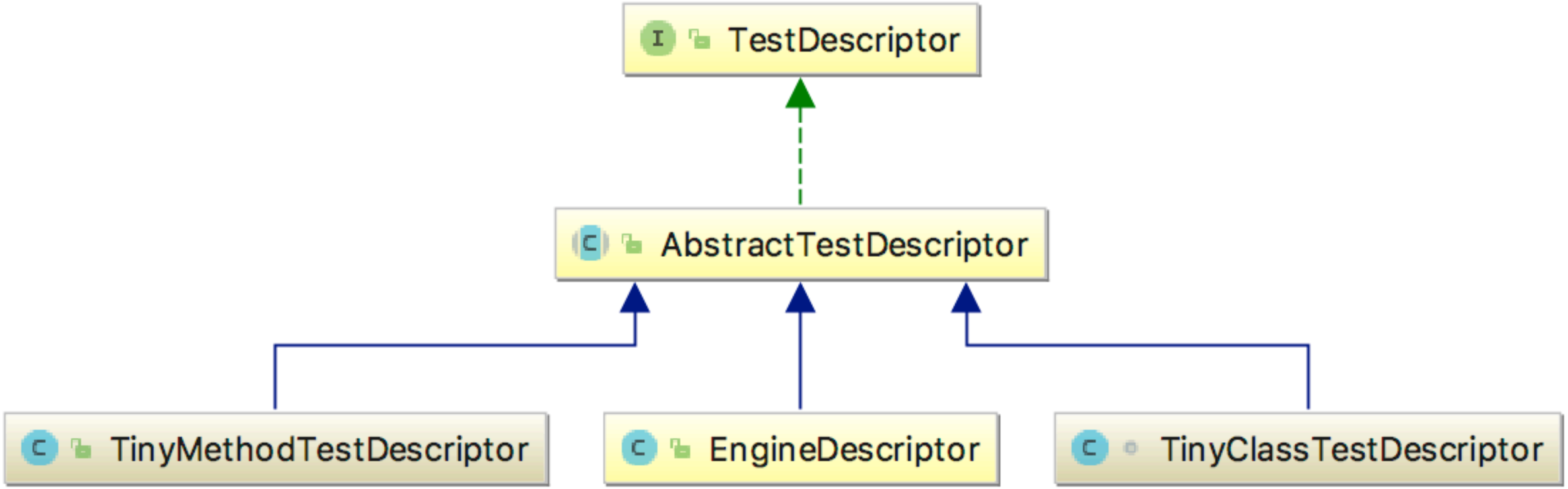
Was fehlt zu stärkerer Test-Engine?

- Nano kann von IDE ausgeführt werden, aber ist sehr beschränkt
- Test-Spezifikation durch Klassen und Methoden

TinyTest

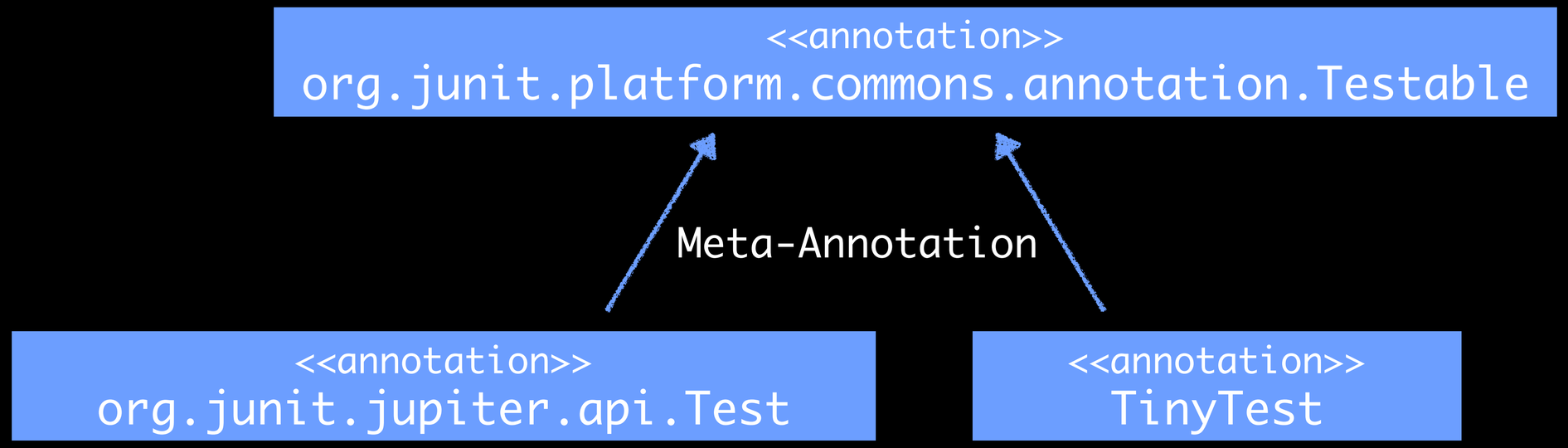
```
@TinyTest
public class A_tiny_test {

  final int theAnswer = 42;

  public boolean this_should_return_true() {
    return theAnswer == 42;
  }

  public boolean this_returns_FALSE() {
    return theAnswer == 43;
  }
}
```


DiscoverySelector
ClassSelector
MethodSelector
ClasspathRootSelector
UniqueIdSelector
PackageSelector


TestDescriptor
AbstractTestDescriptor
TinyMethodTestDescriptor
EngineDescriptor
TinyClassTestDescriptor

Tests in IDE identifizieren

- Theoretisch ist jedes zur Laufzeit ermittelbare Spezifikationsmodell umsetzbar
- Aber: Alles immer zu kompilieren ist für (manche) IDEs nicht realisierbar



Was geht noch?

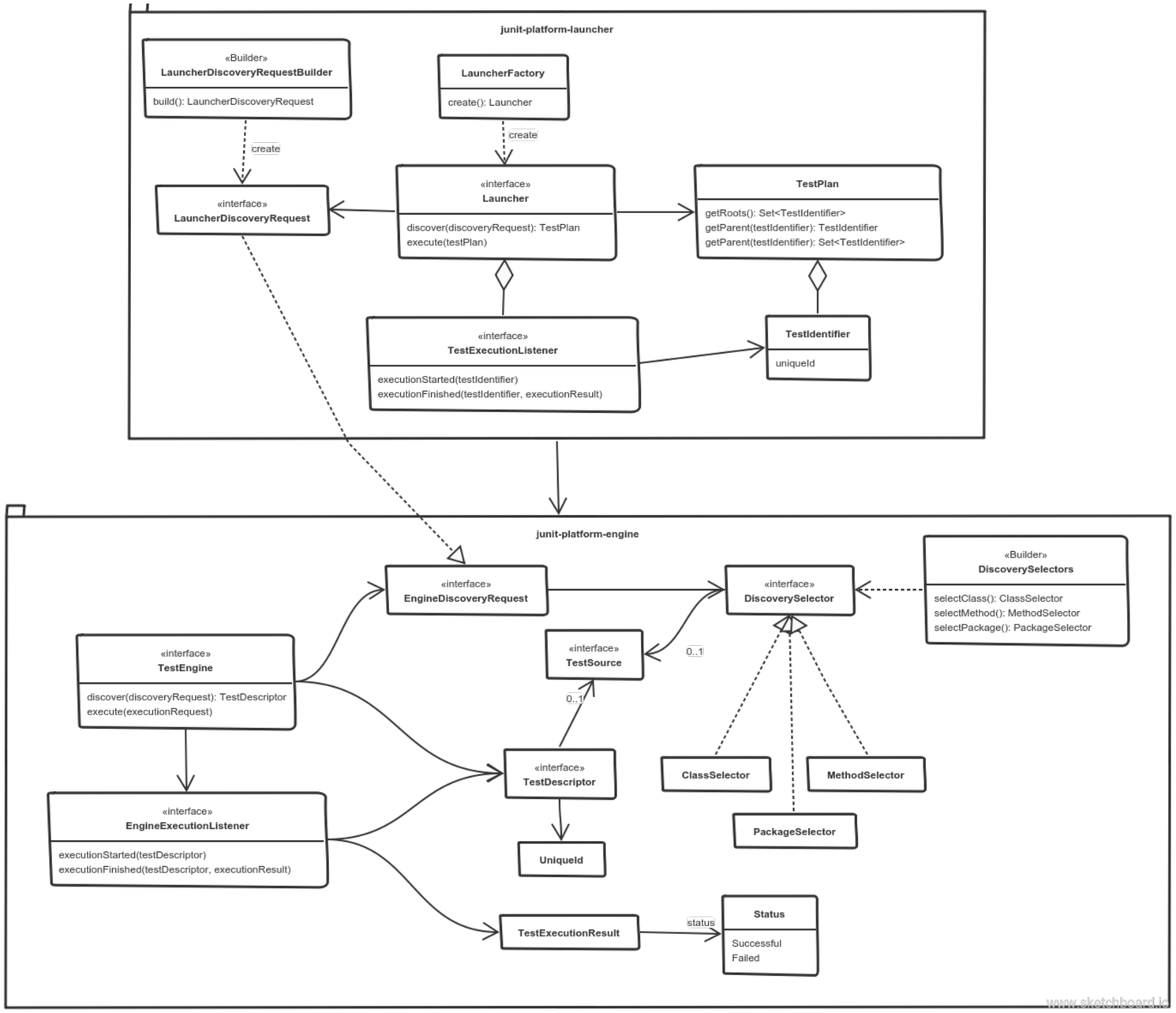
- Ausführen von Dateien, URLs, und anderen Ressourcen
- jqwik.net: Property Testen in Java
- Specsy: Alles mit Lambdas
- Wrapper für Cucumber, Spock, Fitnesse etc

Tool-Support: State of the Union

- IntelliJ
  - @Testable erforderlich für Run-Symbol am Source-Code
  - Gezielter Teststart nur für Klasse/Methode/Package
  - und andere Unstimmigkeiten...
- Eclipse
  - Geplant für 4.7.1
- Gradle und Maven vom JUnit-Team gepflegt

Architekturbewertung der JUnit-5-Plattform

```
git clone https://github.com/junit-team/junit5.git
```


junit-platform-launcher
«Builder»
LauncherDiscoveryRequestBuilder
build(): LauncherDiscoveryRequest
LauncherFactory
create(): Launcher
create
create
«interface»
LauncherDiscoveryRequest
«interface»
Launcher
discover(discoveryRequest): TestPlan
execute(testPlan)
TestPlan
getRoots(): Set<TestIdentifier>
getParent(testIdentifier): TestIdentifier
getParent(testIdentifier): Set<TestIdentifier>
«interface»
TestExecutionListener
executionStarted(testIdentifier)
executionFinished(testIdentifier, executionResult)
TestIdentifier
uniqueId
junit-platform-engine
«interface»
EngineDiscoveryRequest
«interface»
DiscoverySelector
«Builder»
DiscoverySelectors
selectClass(): ClassSelector
selectMethod(): MethodSelector
selectPackage(): PackageSelector
«interface»
TestEngine
discover(discoveryRequest): TestDescriptor
execute(executionRequest)
«interface»
TestSource
0..1
0..1
«interface»
TestDescriptor
ClassSelector
MethodSelector
PackageSelector
«interface»
EngineExecutionListener
executionStarted(testDescriptor)
executionFinished(testDescriptor, executionResult)
UniqueId
TestExecutionResult
status
Status
Successful
Failed
www.sketchboard.io

Bewertungskriterien

- Usability
- Separation of Concerns
- Freie Kombinierbarkeit mit anderen Frameworks und Bibliotheken
- Erweiterbarkeit
- Auf- und Abwärtskompatibilität

JUnit 5 Architektur: Usability

- Für den Engine-Verwender denkbar einfach
- Für den Engine-Entwickler
  - Einfache Schnittstellen
  - IDE- und Build-Support bekommt man „umsonst“
  - Viel Logik (Discovery + Execution) muss selbst implementiert werden
    - Support-Packages für Reflection, Classpath-Scanning, Exception-Handling und Hierarchische Testspecs

JUnit 5 Architektur: Separation of Concerns

- Gute Trennung von APIs, SPIs, Runnern, Engines, Build-Support
- Aber: Komplizierte Abhängigkeitsstruktur in Build-Files

JUnit 5 Architektur: Kombinierbarkeit mit anderen Frameworks

- Kaum Abhängigkeiten, dadurch kaum Konfliktpotenzial
- Aber: Die Plattform übernimmt den kompletten Ausführungslebenszyklus

JUnit 5 Architektur: Erweiterbarkeit

- Engines sind völlig unabhängig voneinander und frei miteinander kombinierbar
- Aber: Funktionalitätssharing zwischen Engines nicht vorgesehen

JUnit 5 Architektur: Auf- und Abwärtskompatibilität

- 99,9 % abwärtskompatibel mit JUnit 4.12
- API Annotationen unterstützen bei der sinnvollen API-Nutzung
- Aber: Stabilität der API muss sich noch beweisen

Extensibility: Zwei Ansätze

- JUnit 4:
  - Test-Runner: Übernimmt die komplette Testausführung
  - Rules: Generic Statement Wrapping
- JUnit 5 Jupiter: Specialized Extension Points

JUnit 4 Statement

Statement

abstract void evaluate() throws Throwable

```
class BlockJUnit4ClassRunner…
 protected Statement methodBlock(FrameworkMethod method) {
     Object test;
     try {
         test = new ReflectiveCallable() {
             @Override
             protected Object runReflectiveCall() throws Throwable {
                 return createTest();
             }
         }.run();
     } catch (Throwable e) {
         return new Fail(e);
     }

     Statement statement = methodInvoker(method, test);
     statement = possiblyExpectingExceptions(method, test, statement);
     statement = withPotentialTimeout(method, test, statement);
     statement = withBefores(method, test, statement);
     statement = withAfters(method, test, statement);
     statement = withRules(method, test, statement);
     return statement;
 }
```

JUnit 4 Rule

<<interface>>
TestRule
Statement apply(Statement)

```
public class MyTestclass…
 @Rule
 public MyRule myRule = new MyRule();

 @Test
 public void aTest() {…}
}
```

```
public class MyRule implements TestRule {
 @Override
 public Statement apply(final Statement base) {
  return new Statement() {
   @Override
   public void evaluate() throws Throwable {

    // Do something before standard behaviour
    base.evaluate();
    // Do something after standard behaviour

   }
  };
 }
}
```

Vor- und Nachteile?

+ Einfache Abstraktion
+ Einfache Komponierbarkeit
- Nur von Innen nach Außen komponierbar
- Keine Möglichkeit die Schachtelung zu verändern
- Seiteneffekte zwischen Rules nicht kontrollierbar

5

Jupiter Extension API

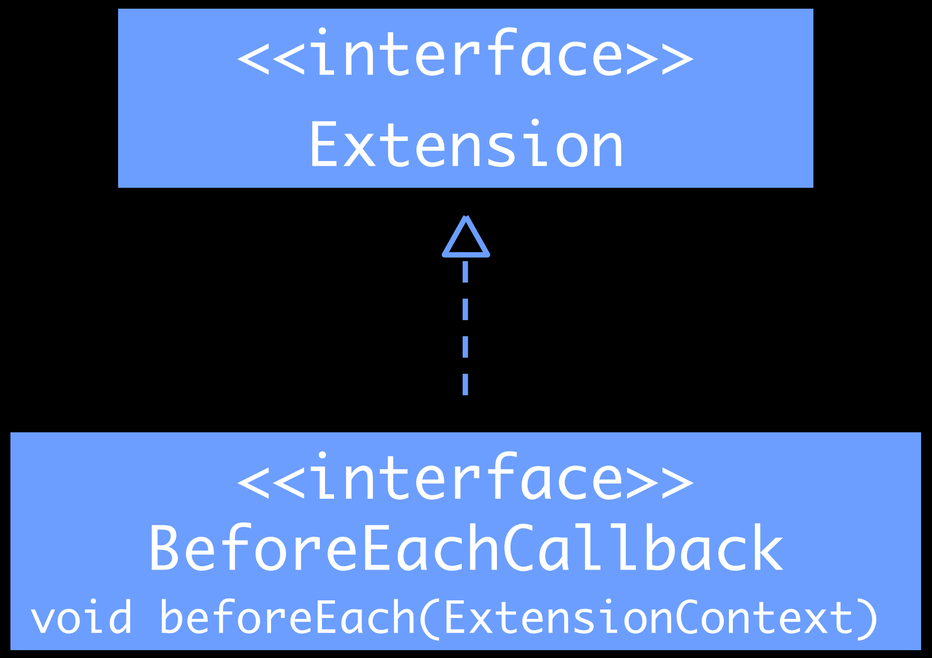
Jupiter Extensions

```java
@API(Experimental)
public interface Extension {
}
```

```java
@API(Experimental)
public @interface ExtendWith {

    Class<? extends Extension>[] value();

}
```

```java
@Test
@ExtendWith(CustomTypeParameterResolver.class)
@ExtendWith(CustomAnnotationParameterResolver.class)
void testMethodWithExtensionAnnotation(CustomType customType,
                                       @CustomAnnotation String value) {...}
```

Jupiter Callbacks



Hierarchy

Hierarchy Subtypes of Extension

Scope: Production

- Extension (org.junit.jupiter.api.extension)
  - AfterEachMethodAdapter (org.junit.jupiter.engine.execution)
  - BeforeEachMethodAdapter (org.junit.jupiter.engine.execution)
  - AfterTestExecutionCallback (org.junit.jupiter.api.extension)
  - ContainerExecutionCondition (org.junit.jupiter.api.extension)
    - DisabledCondition (org.junit.jupiter.engine.extension)
  - BeforeTestExecutionCallback (org.junit.jupiter.api.extension)
  - TestExecutionCondition (org.junit.jupiter.api.extension)
    - DisabledCondition (org.junit.jupiter.engine.extension)
  - AfterEachCallback (org.junit.jupiter.api.extension)
    - ExternalResourceSupport (org.junit.jupiter.migrationsupport.rules)
    - ExpectedExceptionSupport (org.junit.jupiter.migrationsupport.rules)
    - AbstractTestRuleSupport (org.junit.jupiter.migrationsupport.rules)
    - VerifierSupport (org.junit.jupiter.migrationsupport.rules)
  - BeforeAllCallback (org.junit.jupiter.api.extension)
  - AfterAllCallback (org.junit.jupiter.api.extension)
  - BeforeEachCallback (org.junit.jupiter.api.extension)
    - ExternalResourceSupport (org.junit.jupiter.migrationsupport.rules)
    - AbstractTestRuleSupport (org.junit.jupiter.migrationsupport.rules)
  - TestTemplateInvocationContextProvider (org.junit.jupiter.api.extension)
    - RepeatedTestExtension (org.junit.jupiter.engine.extension)
    - ParameterizedTestExtension (org.junit.jupiter.params)
  - TestExecutionExceptionHandler (org.junit.jupiter.api.extension)
    - ExpectedExceptionSupport (org.junit.jupiter.migrationsupport.rules)
    - AbstractTestRuleSupport (org.junit.jupiter.migrationsupport.rules)
      - TestRuleMethodSupport (org.junit.jupiter.migrationsupport.rules)
      - TestRuleFieldSupport (org.junit.jupiter.migrationsupport.rules)
  - ParameterResolver (org.junit.jupiter.api.extension)
    - ParameterizedTestParameterResolver (org.junit.jupiter.params)
    - TestInfoParameterResolver (org.junit.jupiter.engine.extension)
    - RepetitionInfoParameterResolver (org.junit.jupiter.engine.extension)
    - TestReporterParameterResolver (org.junit.jupiter.engine.extension)

Test Lifecycle Callbacks	Conditional Execution	Allgemeine Extensions
BeforeAllCallback	ContainerExecutionCondition	ParameterResolver
BeforeEachCallback	TestExecutionCondition	TestExecutionExceptionHandler
BeforeTestExecutionCallback		TestInstancePostProcessor
AfterTestExecutionCallback		
AfterEachCallback		
AfterAllCallback		

```
BeforeAllCallback (1)
    @BeforeAll (2)
        BeforeEachCallback (3)
            @BeforeEach (4)
                BeforeTestExecutionCallback (5)
                    @Test (6)
                    TestExecutionExceptionHandler (7)
                AfterTestExecutionCallback (8)
            @AfterEach (9)
        AfterEachCallback (10)
    @AfterAll (11)
AfterAllCallback (12)
```

Lifecycle Callbacks (`@ExtendWith(Extension)`) User code: methods of the test class

Extensions sind...

- Komponierbar
- Zustandslos
- Können Zustand über einen Store transportieren und austauschen
  - Namespaces
  - Hierarchisch

Beispiel: Mockito-Extension

```java
public class MockitoExtension implements TestInstancePostProcessor, ParameterResolver {

  @Override
  public void postProcessTestInstance(Object testInstance, ExtensionContext context) {
    MockitoAnnotations.initMocks(testInstance);
  }

  @Override
  public boolean supportsParameter(ParameterContext parameterContext, ExtensionContext extensionContext) {
    return parameterContext.getParameter().isAnnotationPresent(Mock.class);
  }

  @Override
  public Object resolveParameter(ParameterContext parameterContext, ExtensionContext extensionContext) {
    return getMock(parameterContext.getParameter(), extensionContext);
  }

  private Object getMock(Parameter parameter, ExtensionContext extensionContext) {
    Class<?> mockType = parameter.getType();
    Store mocks = extensionContext.getStore(Namespace.create(MockitoExtension.class, mockType));

    return mocks.getOrComputeIfAbsent(mockType.getCanonicalName(), key -> mock(mockType));
  }
}
```

Beispiel: Spring-Extension

```
public class SpringExtension implements
  BeforeAllCallback,
  AfterAllCallback,
  TestInstancePostProcessor,
  BeforeEachCallback,
  AfterEachCallback,
  ParameterResolver
```

https://github.com/sbrannen/spring-test-junit5

Vor- und Nachteile?

+ Feingranulare Eingriffe in den Lebenszyklus möglich
+ Gute Komponierbarkeit unterschiedlicher Extensions
+ State nur wenn nötig
- Zahlreiche unterschiedliche Callbacks notwendig
- Komplexer Life-Cycle in der Testausführung
- Nicht alle unerwünschten Interaktionen zwischen Extensions sind kontrollierbar

Links

- Artikel über JUnit-5-Architektur
  https://blog.codefx.org/design/architecture/junit-5-architecture/
- Artikel über Jupiter-Extension-Modell
  https://blog.codefx.org/design/architecture/junit-5-extension-model/
- Framework Design Principles: http://www.davidtanzer.net/node/118
- JUnit 5: http://junit.org/junit5
- Jqwik-Engine: http://jqwik.net