JUNIT5 TEST-ENGINES

FROM @TEST TO YOUR CUSTOM TESTENGINE

@SVENRUPPERT

vaadin}>

SVEN RUPPERT

Developer Advocate @ Vaadin

CODING JAVA SINCE 1996

DISTRIBUTED SYSTEMS SINCE 2002

CONSULTING WORLD WIDE

JOINED VAADIN 2017

ORACLE Developer Champion

Private Sector: Automotive / Aerospace / SMB /

Public Sector: Military / Government

NonProfit / NonGov: World Bank / UN / YPARD / CGIAR

vaadin}>

Intro

THE WORLD OF UNIT-TESTING

@SvenRuppert

Intro - the world of Unit Testing

Global Preparations

Preparations

Subject to test

Clean Up

Preparations

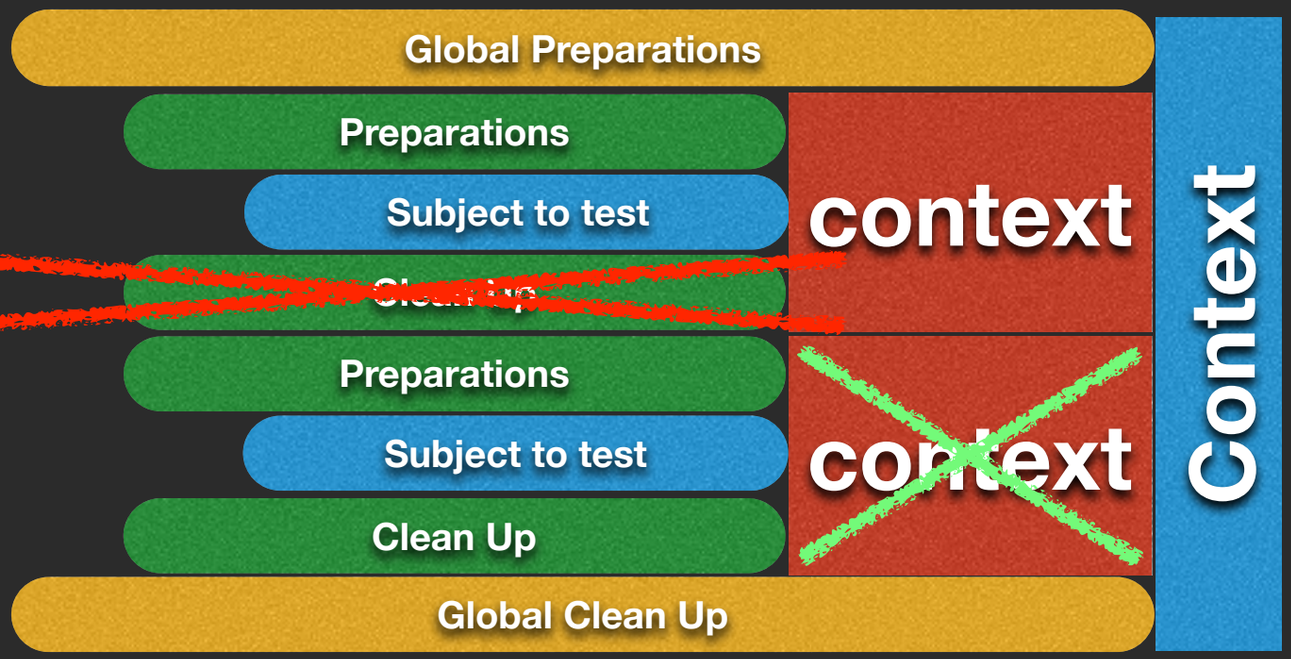
Subject to test

Clean Up

Global Clean Up

Intro - the world of Unit Testing

@SvenRuppert



Preparations / Clean up must be fast

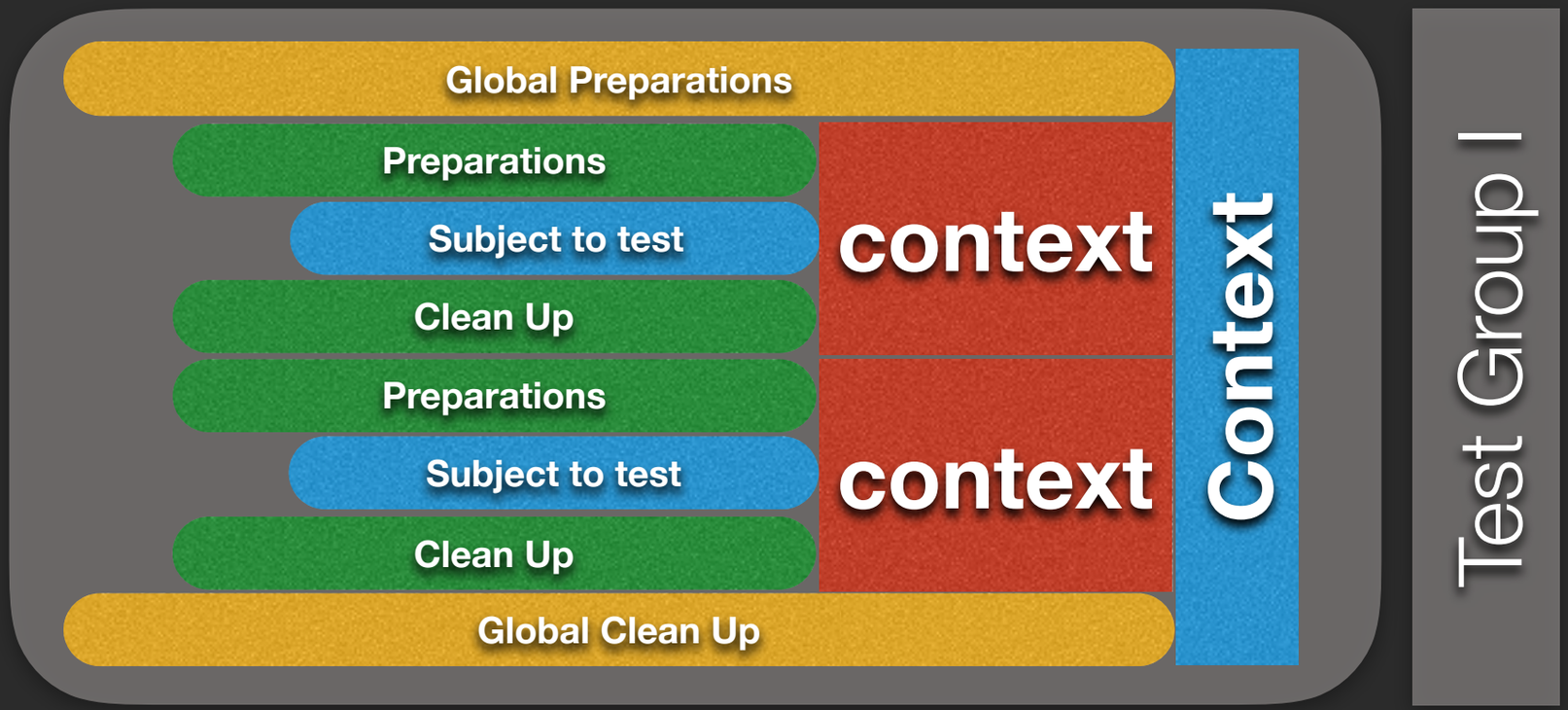
Easy if this is symmetric

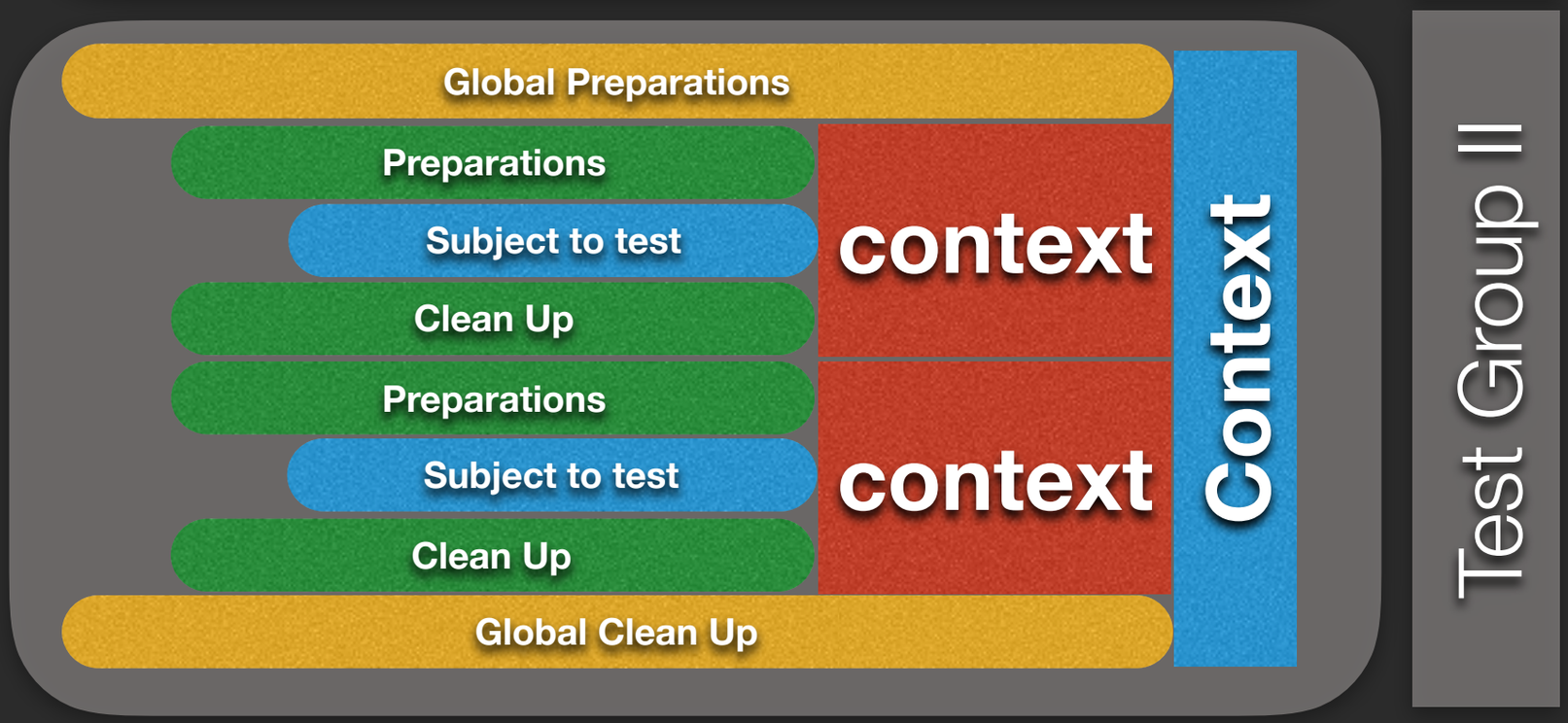
How to deal with exceptions?

Different Tests - needs different preparations / context

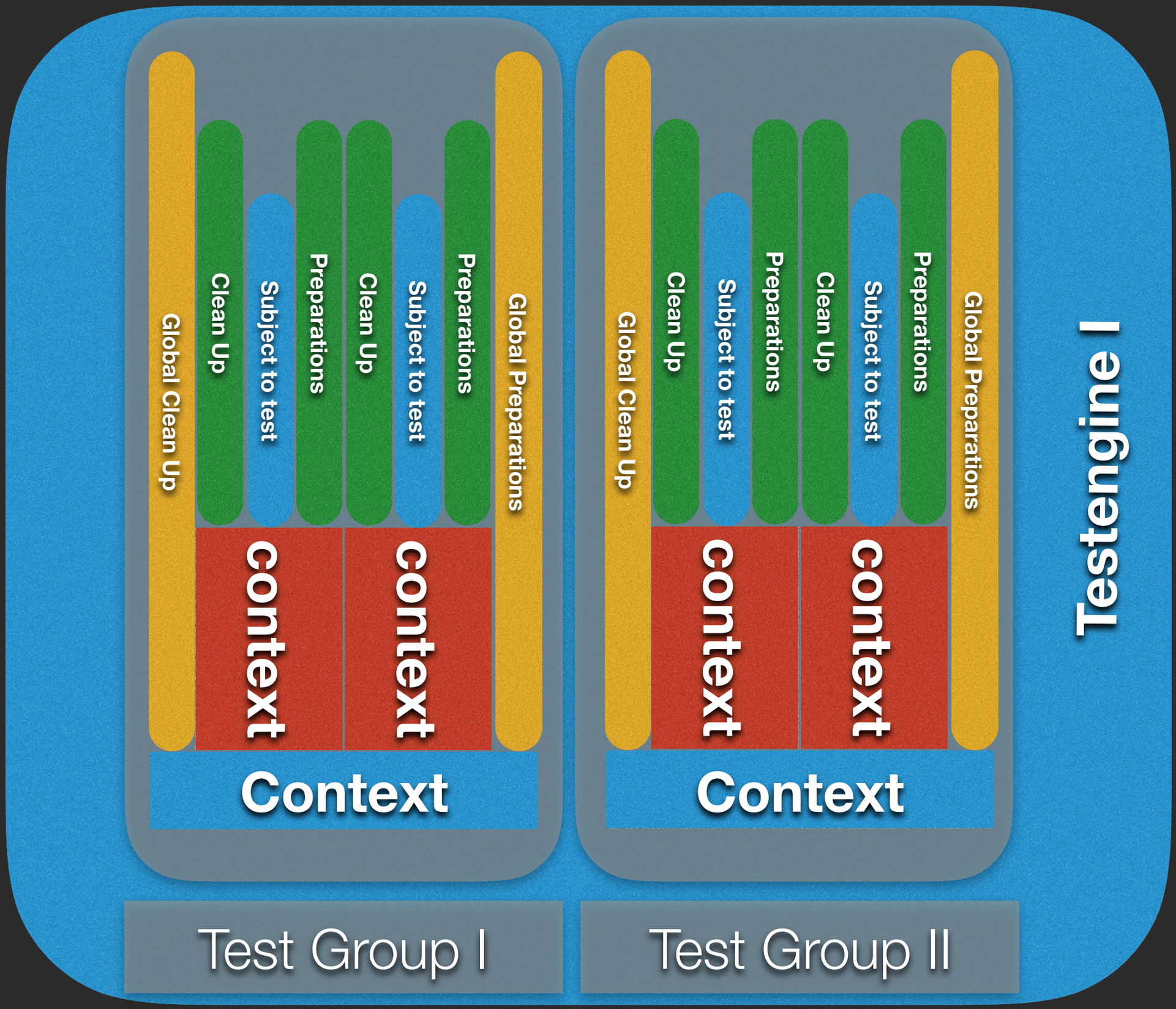
Intro - the world of Unit Testing

@SvenRuppert





Different Tests - needs different preparations / context

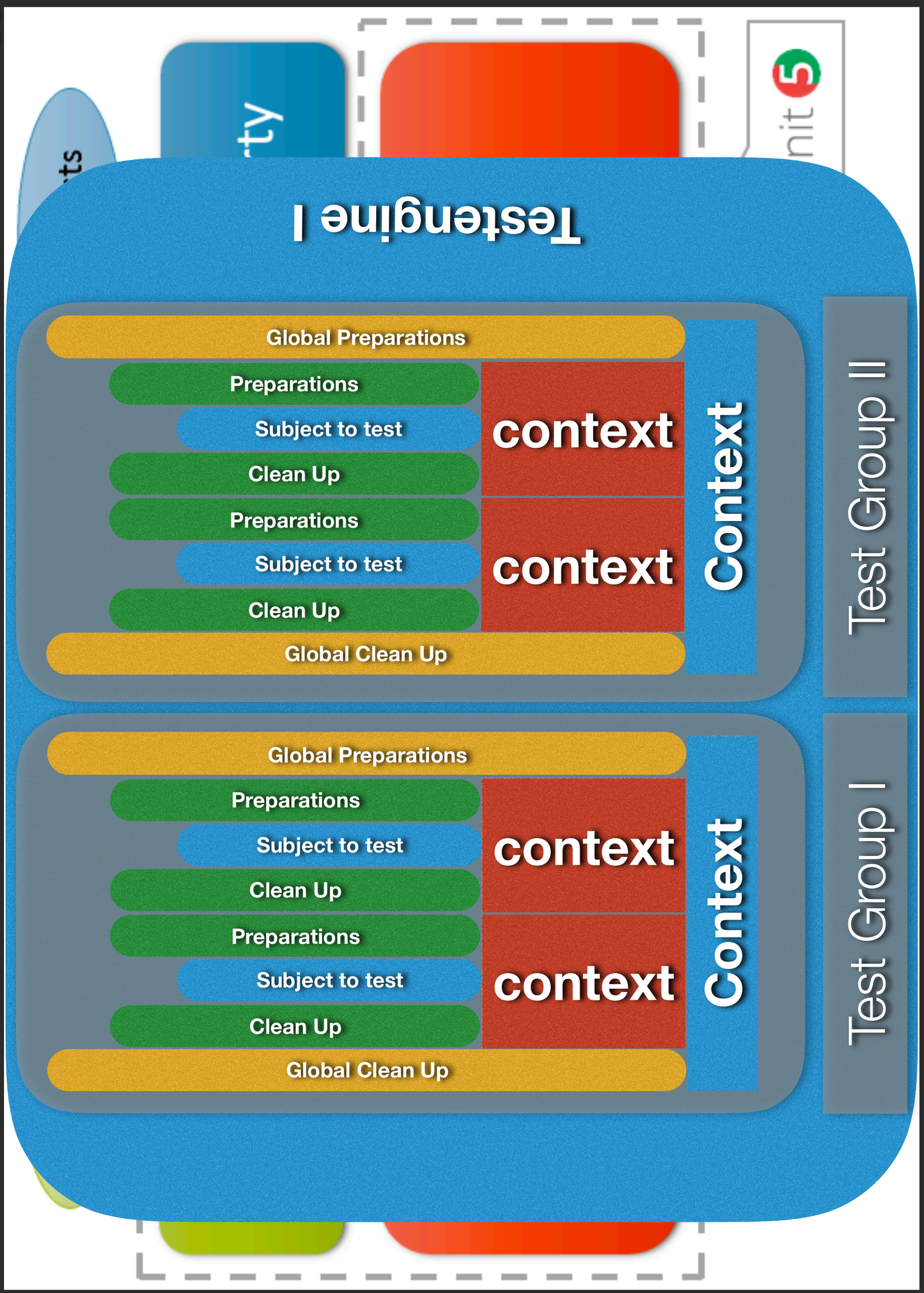


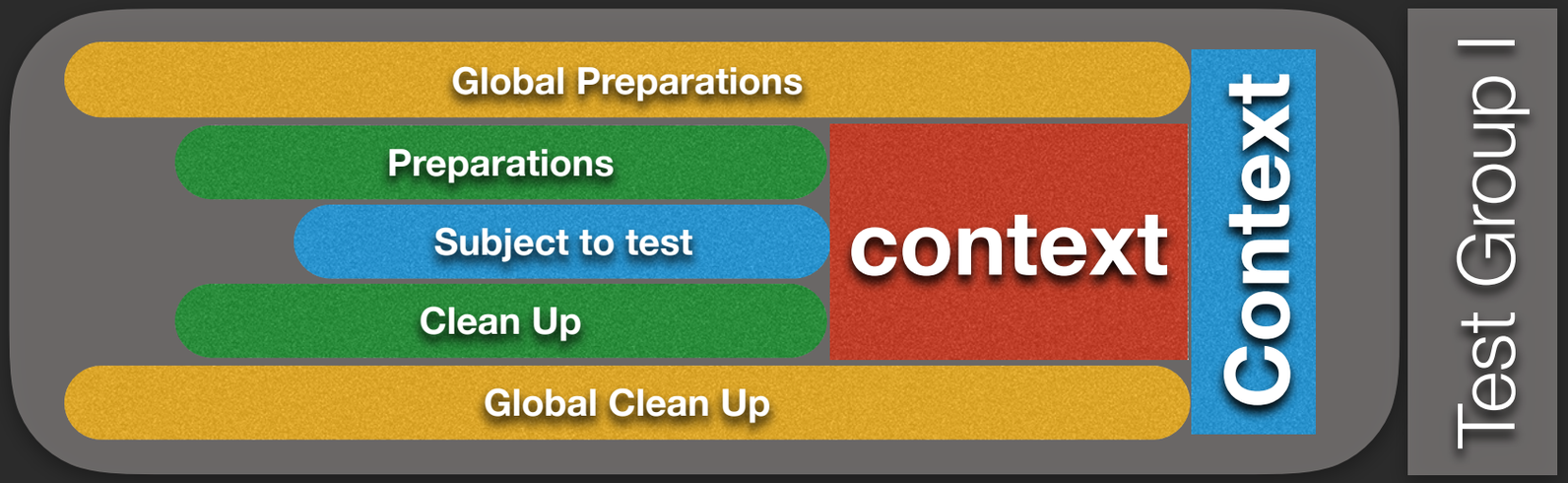
Different Tests - needs different preparations / context

JUnit 5 Basics

HOW TO START WITH @TEST

@SvenRuppert


Testengine I
Global Preparations
Preparations
Subject to test
Clean Up
context
Preparations
Subject to test
Clean Up
context
Global Clean Up
Context
Test Group II
Global Preparations
Preparations
Subject to test
Clean Up
context
Preparations
Subject to test
Clean Up
context
Global Clean Up
Context
Test Group I



LifeCycle - Management

JUnit5 Basics

@SvenRuppert


```
package junit.org.rapidpm.junit.basics;

import org.junit.jupiter.api.Test;

class Basic00Test {

  @Test
  void test001() {

  }
}
```

JUnit5 Basics

@SvenRuppert


```
public class Basic01Test
        implements BeforeAllCallback, BeforeEachCallback,
                   AfterAllCallback, AfterEachCallback {

    @Override
    public void beforeAll(ExtensionContext extensionContext) throws Exception { }

    @Override
    public void beforeEach(ExtensionContext extensionContext) throws Exception { }

    @Override
    public void afterEach(ExtensionContext extensionContext) throws Exception { }


    @Override
    public void afterAll(ExtensionContext extensionContext) throws Exception { }

    @Test
    void test001() { }
}
```

JUnit5 Basics

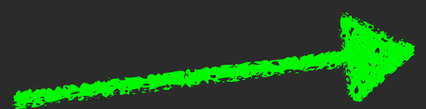
@SvenRuppert


```
public class Basic02Test {

    public static class MyExtension
            implements BeforeEachCallback, AfterEachCallback {

        @Override
        public void beforeEach(ExtensionContext extensionContext) throws Exception { }

        @Override
        public void afterEach(ExtensionContext extensionContext) throws Exception { }
    }

    @ExtendWith(MyExtension.class)
    public static class ExtendedTestClass {
        @Test
        void test001() { }
    }
}
```

JUnit5 Basics

@SvenRuppert


```java
public class Basic03Test {
  public static class MyExtensionA
      implements BeforeEachCallback {
    @Override
    public void beforeEach(ExtensionContext extensionContext) throws Exception { }
  }

  public static class MyExtensionB
      implements BeforeEachCallback {
    @Override
    public void beforeEach(ExtensionContext extensionContext) throws Exception { }
  }

  @ExtendWith(Basic03Test.MyExtensionA.class)
  @ExtendWith(Basic03Test.MyExtensionB.class)
  public static class ExtendedTestClass {
    @Test
    void test001() { }
  }
}
```



```java
@ExtendWith(Basic03Test.MyExtensionA.class)
@ExtendWith(Basic03Test.MyExtensionB.class)
public static class ExtendedTestClass {
  @Test
  void test001() { }
}
```

```java
@ExtendWith(Basic03Test.MyExtensionB.class)
@ExtendWith(Basic03Test.MyExtensionA.class)
public static class ExtendedTestClass {
  @Test
  void test001() { }
}
```

JUnit5 Basics

@SvenRuppert


```
@ExtendWith(Basic03Test.MyExtensionB.class)
@ExtendWith(Basic03Test.MyExtensionA.class)
```

BeforeAll - B

BeforeAll - A

BeforeEach - B

BeforeEach - A

AfterEach - A

AfterEach - B

AfterAll - A

AfterAll - B

JUnit5 Basics

@SvenRuppert


```
@Extensions({
  @ExtendWith({Basic03Test.MyExtensionB.class}),
  @ExtendWith({Basic03Test.MyExtensionA.class})
})
public @interface BothExtensions { }

@BothExtensions
public static class MyExtendedTestClass {
  @Test
  void test001() { }
}
```


```
public static final String KEY = "KEY";

@Override
public void beforeEach(ExtensionContext ctx) throws Exception {
  final ExtensionContext.Store store = ctx.getStore(Namespace.GLOBAL);
  //create and start something
  store.put(KEY, o1:"something");
}

@Override
public void afterEach(ExtensionContext ctx) throws Exception {
  final ExtensionContext.Store store = ctx.getStore(Namespace.GLOBAL);
  //stop something again
  final String value = store.get(KEY, String.class);
}
```

JUnit5 Basics

@SvenRuppert


```
@Test
void test001(Demo demo){
  Assertions.assertEquals("Hello World", demo.value());
}
```


```
public static class DemoParameterResolver implements ParameterResolver {
  @Override
  public boolean supportsParameter(ParameterContext parameterContext,
                                   ExtensionContext extensionContext)
    throws ParameterResolutionException {
   final Class<?> type = parameterContext.getParameter().getType();
   return Demo.class.isAssignableFrom(type);
  }

  @Override
  public Object resolveParameter(ParameterContext parameterContext,
                                 ExtensionContext extensionContext)
    throws ParameterResolutionException {
   return new Demo(1, "Hello World");
  }
}
```

JUnit5 Basics

@SvenRuppert


```
@Test
@ExtendWith(Basic05Test.DemoParameterResolver.class)
void test001(Demo demo){

  Assertions.assertEquals("Hello World", demo.value());
}
```

LoginServiceTest

Write an Extension to ramp up the DataSource
Write an Extension to ramp up the LoginService

make the LoginService for a test available

JUnit 5 TDD

HOW TO TEST YOUR TESTENGINE

@SvenRuppert

How to test a TestEngine?

We need a way to test a new TestEngine

As first exercise we will test an existing TestEngine

```
public static class DemoTestClass {
  @Test
  void test001() { }

  @Test
  void test002() {  throw new RuntimeException("I must fail");}

  @Test
  @Disabled
  void test003() { }
}
```

```
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-testkit</artifactId>
  <version>${junit5.version}</version>
  <scope>test</scope>
</dependency>
```

Three tests are defined

```
public static class DemoTestClass {
  @Test
  void test001() { }

  @Test
  void test002() {  throw new RuntimeException("I must fail");}

  @Test
  @Disabled
  void test003() { }
}
```

One test is failing

Two tests are started / one test is disabled

Three tests are defined

```
@Test
void verifyJupiterTestStats() {
  EngineTestKit.engine( engineId: "junit-jupiter") EngineTestKit.B
                .selectors(selectClass(DemoTestClass.class))
                .execute() EngineExecutionResults
                .tests() Events
                .assertStatistics(stats -> stats.started(2)
                                           .succeeded(1)
                                           .skipped(1)
                                           .failed(1));
}
```

One test is failing

Two tests are started / one test is disabled

```java
@Test
void test002() { throw new RuntimeException("I must fail");}
```

```java
@Test
void verifyTest002() {
  final String methodName = "test002";
  final Events events = EngineTestKit.engine( engineId: "junit-jupiter")
                                     .selectors(selectMethod(DemoTestClass.class, methodName))
                                     .execute()
                                     .tests();

  events.assertStatistics(stats -> stats.failed(1));

  events.assertThatEvents()
        .haveExactly( times: 1, event(test(methodName),
                                finishedWithFailure(
                                    instanceOf(RuntimeException.class),
                                    message( expectedMessage: "I must fail"))));
}
```

JUnit 5 TDD

HOW TO START YOUR TESTENGINE

@SvenRuppert

How to start a custom Test-Engine to test it?

1) Discovery the Tests
2) prepare the Launcher
3) create the TestPlan
4) register Listeners
5) execute Launcher
6) consume the Result

How to start a Test-Engine?

1) Discovery the Tests 2) prepare the Launcher 3) create the TestPlan

4) register Listeners 5) execute Launcher 6) consume the Result

```
LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
                                                                  .build();

Launcher launcher = LauncherFactory.create();
TestPlan testPlan = launcher.discover(request);

// Register a listener of your choice
SummaryGeneratingListener listener = new SummaryGeneratingListener();
launcher.registerTestExecutionListeners(listener);
launcher.execute(request);
TestExecutionSummary summary = listener.getSummary();
summary.printTo(new PrintWriter(out));
testPlan.getRoots() Set<TestIdentifier>
        .stream() Stream<TestIdentifier>
        .filter(TestIdentifier::isContainer) Stream<Te
        .map(TestIdentifier::getDisplayName) Stream<Str
        .map(name -> "TestEngineName: " + name) Stream
        .collect(Collectors.toList()) List<String>
        .forEach(out::println);
```

How to start a Test-Engine?

1) Discovery the Tests 2) prepare the Launcher 3) create the TestPlan

4) register Listeners 5) execute Launcher 6) consume the Result

```
LauncherDiscoveryRequest request = request().selectors(selectPackage( packageName: "junit"))
                                            .build();

final SummaryGeneratingListener summaryGeneratingListener = new SummaryGeneratingListener();
final LauncherConfig config = LauncherConfig.builder()
                                            .enableTestEngineAutoRegistration(false)
                                            .enableTestExecutionListenerAutoRegistration(false)
                                            .addTestEngines(new JupiterTestEngine())
                                            .addTestExecutionListeners(summaryGeneratingListener)
                                            .build();

Launcher launcher = LauncherFactory.create(config);

TestPlan testPlan = launcher.discover(request);

launcher.execute(request);

TestExecutionSummary summary = summaryGeneratingListener.getSummary();
```

Useless -Engine

THE BASIC STRUCTURE OF A TESTENGINE

@SvenRuppert

Basic Structure to build a Custom TestEngine

No real Test-Discovery in the first run

Production code - the Test-Engine

Test code - the Test-Engine-Tests

Test the custom Test-Engine with the Jupiter-Engine

Maven dependencies

```
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-engine</artifactId>
  <version>${junit.platform.version}</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-commons</artifactId>
  <version>${junit.platform.version}</version>
  <scope>compile</scope>
</dependency>

<!--    needed for writing integration tests-->
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-launcher</artifactId>
  <version>${junit.platform.version}</version>
  <scope>compile</scope>
</dependency>
```

Useless - Engine

@SvenRuppert

```
public class UselessEngine
    implements TestEngine, HasLogger {
  @Override
  public String getId() {
    return UselessEngine.class.getSimpleName();
  }

  @Override
  public TestDescriptor discover(EngineDiscoveryRequest request, UniqueId engineID) {
    EngineDescriptor rootNode = new EngineDescriptor(engineID,  displayName: "The UselessEngine");
    return rootNode;
  }
```

```
  @Override
  public void execute(ExecutionRequest request) {
    TestDescriptor           engine   = request.getRootTestDescriptor();
    EngineExecutionListener listener = request.getEngineExecutionListener();
    listener.executionStarted(engine);
    engine.getChildren()
          .forEach(child -> {
            listener.executionStarted(child);
            listener.executionFinished(child, successful());
          });
    listener.executionFinished(engine, successful());
  }
}
```

Useless - Engine

@SvenRuppert

Test run finished after 18 ms

```
[         1 containers found      ]
[         0 containers skipped    ]
[         1 containers started    ]
[         0 containers aborted    ]
[         1 containers successful ]
[         0 containers failed     ]
[         0 tests found           ]
[         0 tests skipped         ]
[         0 tests started         ]
[         0 tests aborted         ]
[         0 tests successful      ]
[         0 tests failed          ]
```

TestEngineName: The UselessEngine

Nano -Engine

HOW TO DEFINE A TEST

@SvenRuppert

Nano - Engine

Find all tests with the Annotation **@NanoTest**

Execute all discovered tests

IDE should be able to execute the test

Maven should be able to execute the test

Nano - Engine

@SvenRuppert

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Testable //for IDE support
public @interface NanoTest { }
```

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Testable
public @interface NanoTestClass {
//params
}
```

@Testable ?

```
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@API(status = STABLE, since = "1.0")
public @interface Testable {
}
```

This is for IDE´s only. They need this for their support.

`@Testable` is used to signal to IDEs and tooling vendors that the annotated or meta-annotated element is *testable*.
In this context, the term "testable" means that the annotated method or class can be executed by a `TestEngine` as a test or test container on the JUnit Platform.

Some clients of the JUnit Platform, notably IDEs such as IntelliJ IDEA, operate only on sources for test discovery. Thus, they cannot use the full runtime discovery mechanism of the JUnit Platform since it relies on compiled classes. `@Testable` therefore serves as an alternative mechanism for IDEs to discover tests by analyzing the source code only.

Nano - Engine

@SvenRuppert

```
public class NanoEngine
    implements TestEngine, HasLogger {

  public static final String ENGINE_ID = NanoEngine.class.getSimpleName();

  @Override
  public String getId() { return ENGINE_ID; }
```

Nano - Engine

@SvenRuppert

```
protected static  Predicate<Class<?>> isTestClass() {
  return classCandidate -> match(matchCase(
      () -> failure("this class is not a supported by this TestEngine - " + classCandidate.getSimpleName())),
                          matchCase(() -> isAbstract(classCandidate), () -> failure(
                              "no support for abstract classes" + classCandidate.getSimpleName())),
                          matchCase(() -> isPrivate(classCandidate), () -> failure(
                              "no support for private classes" + classCandidate.getSimpleName())),
                          matchCase(() -> isAnnotated(classCandidate, NanoTestClass.class),
                                  () -> success(Boolean.TRUE))).ifFailed(msg -> Logger.getLogger(NanoEngine.class).info(msg))
                                                                .ifPresent(b -> Logger.getLogger(NanoEngine.class).info(
                                                                    s: "selected class " + classCandidate))
                                                                .getOrElse(() -> Boolean.FALSE);
}
```

For demonstration purpose -> filter Annotation on Class-Level

Selection is based on Reflection

Nano - Engine

@SvenRuppert

```
protected static Predicate<Method> isTestMethod() {
  return method -> {
    if (ReflectionUtils.isStatic(method)) return false;
    if (ReflectionUtils.isPrivate(method)) return false;
    if (ReflectionUtils.isAbstract(method)) return false;
    if (method.getParameterCount() > 0) return false;
    return AnnotationSupport.isAnnotated(method, NanoTest.class)
        && method.getReturnType().equals(void.class);
  };
}
```

Identify if this method is a Test-Method for this Test-Engine

Nano - Engine

@SvenRuppert

```
@Override
public TestDescriptor discover(EngineDiscoveryRequest request, UniqueId engineID) {
  EngineDescriptor rootNode = new EngineDescriptor(engineID,  displayName: "The NanoEngine");

  request.getSelectorsByType(ClasspathRootSelector.class)
         .forEach(selector -> appendTestInRoot(rootNode, selector));

  request.getSelectorsByType(PackageSelector.class)
         .forEach(selector -> appendTestInPackage(selector.getPackageName(), rootNode));

  request.getSelectorsByType(ClassSelector.class)
         .forEach(classSelector -> appendTestInClass(classSelector.getJavaClass(), rootNode));

  request.getSelectorsByType(MethodSelector.class)
         .forEach(selector -> appendTestInMethod(selector.getJavaMethod(), rootNode));

  return rootNode;
}
```

Deal with different entry level

Nano - Engine

@SvenRuppert

```
private void appendTestInRoot(EngineDescriptor rootNode, ClasspathRootSelector selector) {
  URI classpathRoot = selector.getClasspathRoot();
  ReflectionUtils.findAllClassesInClasspathRoot(classpathRoot, isTestClass(), (name) -> true)
                 .forEach(clazz -> appendTestInClass(clazz, rootNode));
}
```

Nano - Engine


```
private void appendTestInPackage(String packageName, EngineDescriptor rootNode) {

    ReflectionSupport.findAllClassesInPackage(packageName, checkClass(), name -> true) List<Class<
                     .stream() Stream<Class<?>>
                     .peek((e) -> logger().info( s: "class in package -> " + e.getSimpleName()))
                     .map(javaClass -> new NanoEngineClassTestDescriptor(javaClass, rootNode))
                     .forEach(rootNode::addChild);
}
```

Nano - Engine

@SvenRuppert

```
private void appendTestInClass(Class<?> javaClass, EngineDescriptor rootNode) {
  if (checkClass().test(javaClass)) rootNode.addChild(new NanoEngineClassTestDescriptor(javaClass, rootNode));
}
```

Nano - Engine

@SvenRuppert

```
private void appendTestInMethod(Method javaMethod, EngineDescriptor rootNode) {
  Class<?> declaringClass = javaMethod.getDeclaringClass();
  if (isTestClass().test(declaringClass)) {
    final NanoEngineMethodTestDescriptor child = new NanoEngineMethodTestDescriptor(javaMethod, declaringClass,
                                                                                    rootNode.getUniqueId());
    rootNode.addChild(child);
  }
}
```

What is a TestDescriptor?

A TestDescriptor is a Node inside the TestPlan

A Node can be a Container or a Test

A Test is a leaf from this tree

A Descriptor can be a Container and a Test at the same time

A good start is the **AbstractTestDescriptor**

Nano - Engine


```
public class NanoEngineMethodTestDescriptor
    extends AbstractTestDescriptor {

  private final Method testMethod;
  private final Class  testClass;

  public NanoEngineMethodTestDescriptor(Method testMethod, Class testClass, UniqueId uniqueId) {
    super(uniqueId.append( segmentType: "method",
                          testMethod.getName()),
          testMethod.getName(),
          MethodSource.from(testMethod));
    this.testMethod = testMethod;
    this.testClass  = testClass;
  }

  @Override
  public Type getType() { return Type.TEST; }

  public Method getTestMethod() { return testMethod; }

  public Class getTestClass() { return testClass; }
}
```

Nano - Engine

@SvenRuppert

```
public class NanoEngineClassTestDescriptor
    extends AbstractTestDescriptor
    implements HasLogger {

  private final Class<?> testClass;

  public NanoEngineClassTestDescriptor(Class<?> testClass, UniqueId uniqueId) {
    super(uniqueId.append( segmentType: "class",
                          testClass.getSimpleName()),
          testClass.getSimpleName(),
          ClassSource.from(testClass));
    this.testClass = testClass;
    addChildren();
  }

  private void addChildren() {
    ReflectionUtils.findMethods(testClass, isTestMethod())
                   .forEach(method -> {
                     final NanoEngineMethodTestDescriptor child = new NanoEngineMethodTestDescriptor(
                         method, testClass, this.getUniqueId());
                     addChild(child);
                   });
  }

  @Override
  public Type getType() { return Type.CONTAINER; }
}
```

Nano - Engine

@SvenRuppert

How to execute a TestPlan

```
public class NanoEngineTestExecutor
    implements HasLogger {

  public void execute(ExecutionRequest request, TestDescriptor rootNode) {
    // all could be non-blocking / async
    if (rootNode instanceof EngineDescriptor) executeContainer(request, rootNode);
    if (rootNode instanceof NanoEngineClassTestDescriptor) executeContainer(request, rootNode);
    if (rootNode instanceof NanoEngineMethodTestDescriptor) executeMethod(request,
                                                                          (NanoEngineMethodTestDescriptor) rootNode);
  }
```

Nano - Engine

@SvenRuppert

How to execute a TestPlan

```
private void executeContainer(ExecutionRequest request, TestDescriptor rootNode) {
  final EngineExecutionListener engineExecutionListener = request.getEngineExecutionListener();
  engineExecutionListener.executionStarted(rootNode);

  rootNode.getChildren().forEach(c -> execute(request, c));

  engineExecutionListener.executionFinished(rootNode, TestExecutionResult.successful());
}
```

Nano - Engine

@SvenRuppert

How to execute a TestPlan

```
private void executeMethod(ExecutionRequest request, NanoEngineMethodTestDescriptor descriptor) {
  request.getEngineExecutionListener().executionStarted(descriptor);
  TestExecutionResult executionResult = executeTestMethod(descriptor);

  request.getEngineExecutionListener().executionFinished(descriptor, executionResult);
}
```

Nano - Engine

@SvenRuppert

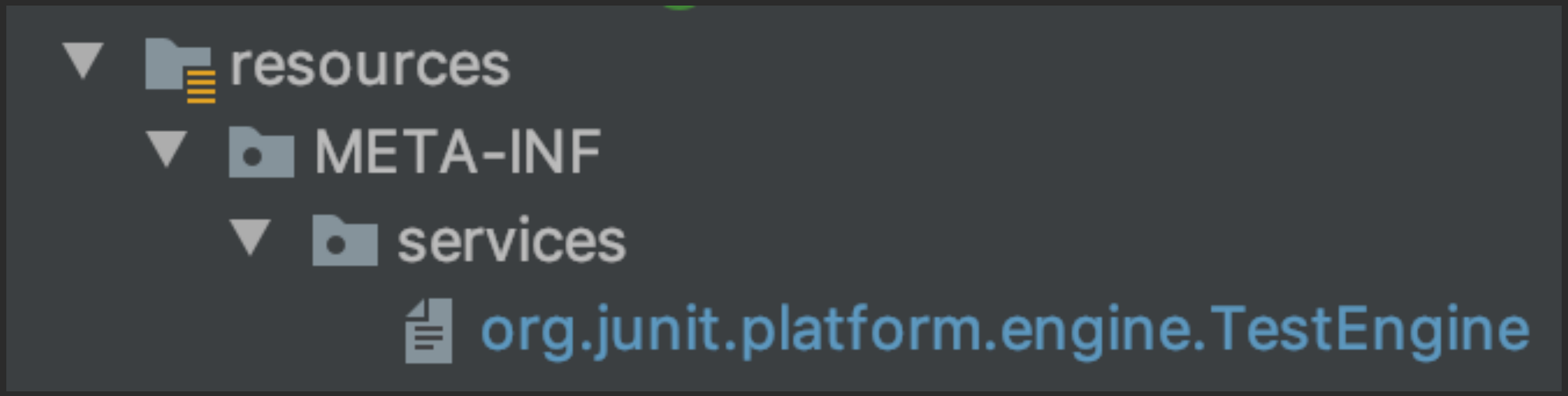
How to execute a TestPlan

```
private TestExecutionResult executeTestMethod(NanoEngineMethodTestDescriptor descriptor) {
  try {
    Object newInstance = ReflectionUtils.newInstance(descriptor.getTestClass());
    ReflectionUtils.invokeMethod(descriptor.getTestMethod(), newInstance);
    return TestExecutionResult.successful();
  } catch (Exception e) {
    logger().warning(e.getMessage());
    return TestExecutionResult.failed(e);
  }
}
```

Nano - Engine


```
public class NanoEngine
    implements TestEngine, HasLogger {

  public static final String ENGINE_ID = NanoEngine.class.getSimpleName();

  @Override
  public String getId() { return ENGINE_ID; }

  protected static Predicate<Class<?>> isTestClass() {...}

  protected static Predicate<Method> isTestMethod() {...}

  @Override
  public TestDescriptor discover(EngineDiscoveryRequest request, UniqueId engineID) {...}

  private void appendTestInRoot(EngineDescriptor rootNode, ClasspathRootSelector selector) {...}
  private void appendTestInMethod(Method javaMethod, EngineDescriptor rootNode) {...}
  private void appendTestInClass(Class<?> javaClass, EngineDescriptor rootNode) {...}
  private void appendTestInPackage(String packageName, EngineDescriptor rootNode) {...}

  @Override
  public void execute(ExecutionRequest request) {
    TestDescriptor rootNode = request.getRootTestDescriptor();
    new NanoEngineTestExecutor().execute(request, rootNode);
  }
}
```

How to activate the Test-Engine?



Core JDK Service-Locator

Nano - Engine - What is achieved until now?

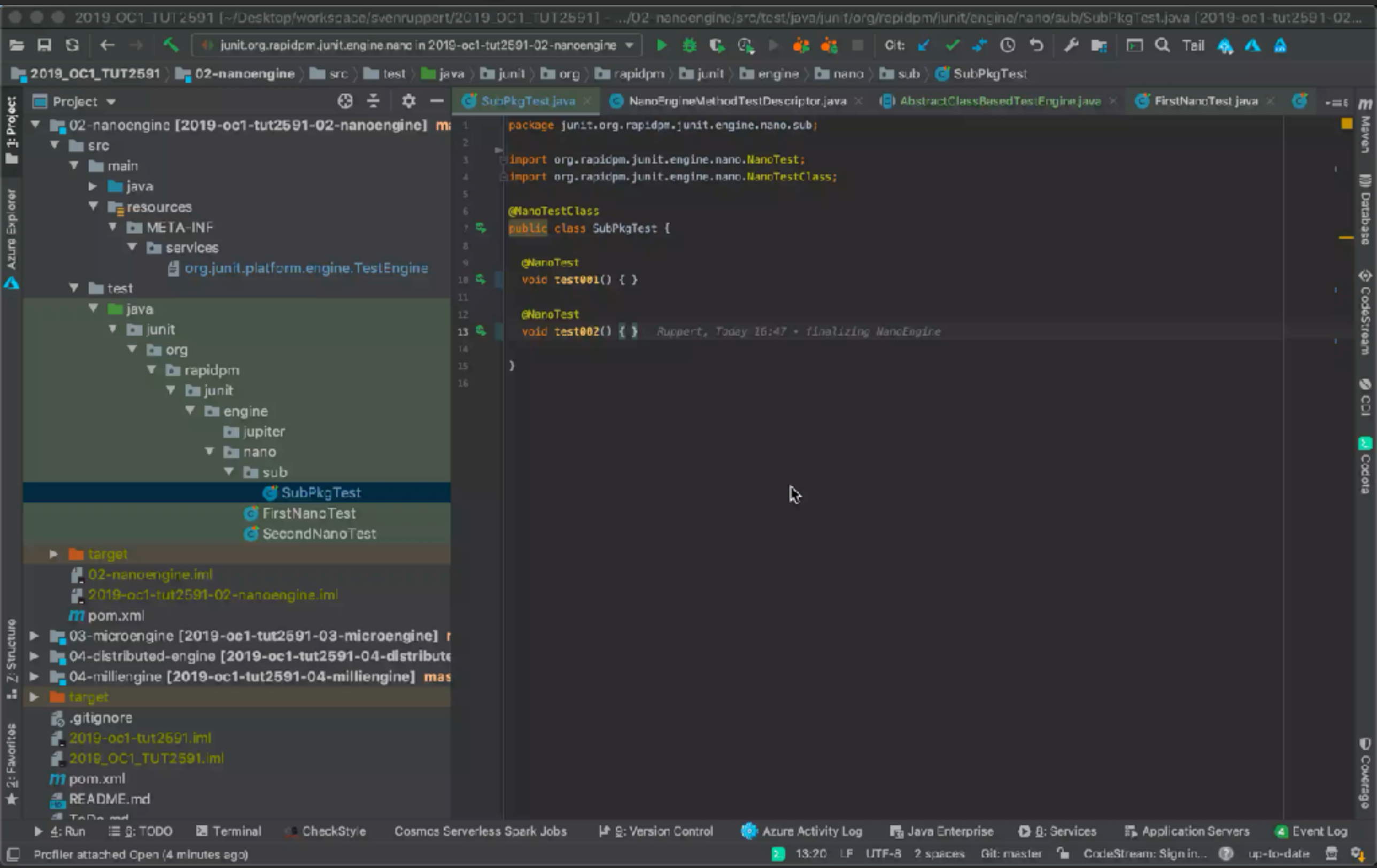
We can define Tests

```
@NanoTestClass
public class FirstNanoTest {
  @NanoTest
  void test001_A() { }
  @NanoTest
  void test001_B() { }

  //@NanoTest
  void test002() { }
}
```

Nano - Engine - What is achieved until now?

@SvenRuppert

We can use this Engine inside an IDE



Micro - Engine

ADD SOME SPECIAL FEATURES

@SvenRuppert

Micro - Engine

Target: Add CDI Support

Target: Add Random Execution Support

Target: Fixing the Pkg Bug from the NanoEngine

Micro - Engine - CDI Support

@SvenRuppert

```
public class MicroEngine
    implements TestEngine, HasLogger {

  private final Weld weld = new Weld();


  @Override
  public void execute(ExecutionRequest request) {
    TestDescriptor rootNode  = request.getRootTestDescriptor();
    WeldContainer  container = weld.initialize();
    new MicroEngineTestExecutor(container)
            .execute(request, rootNode);
    container.shutdown();
  }
```

Micro - Engine - CDI Support


```
public class MicroEngineTestExecutor
    implements HasLogger {

 private WeldContainer container;



 private TestExecutionResult executeTestMethod(
              MicroEngineMethodTestDescriptor descriptor) {
   try {
     Class<?> testClass = descriptor.getTestClass();
     Objects.requireNonNull(testClass);
     Object obj = (descriptor.useCDI())
                ? container.select(testClass).get()
                : ReflectionUtils.newInstance(testClass);
     final Method testMethod = descriptor.getTestMethod();
     ReflectionUtils.invokeMethod(testMethod, obj);
     return successful();
   } catch (Exception e) {
     return TestExecutionResult.failed(new RuntimeException(e));
   }
}
```

Micro - Engine - CDI Support


```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Testable
public @interface MicroTestClass {
  boolean forceRandomExecution() default false;
  boolean useCDI() default false;
}
```

```
@MicroTestClass(useCDI = true)
public class FirstMicroTest {

  @Inject
  private MyMockedService service;

  @MicroTest
  void test001_A() {
    Assertions.assertEquals("someWork", service.doSomeWork());
  }
}
```

One Test selected: No Random possible

One TestClass selected: Random Testmethod execution

Root/Pkg selected: Random per class? Per method?

Micro - Engine - Random Execution Support

@SvenRuppert

```
public class MicroEngineClassTestDescriptor
    extends AbstractTestDescriptor
    implements HasLogger {


  private void addChildren() {
    findAnnotation(testClass, MicroTestClass.class).ifPresent(a -> {
      forceRandomExecution = a.forceRandomExecution();
      useCDI               = a.useCDI();
    });

    ReflectionUtils.findMethods(testClass, isTestMethod())
                   .stream()
                   .collect(collectingAndThen(toList(), elements -> {
                     if (forceRandomExecution) shuffle(elements);
                     return elements.stream();
                   }))
                   .map(method -> new MicroEngineMethodTestDescriptor(
                                       method,
                                       testClass,
                                       getUniqueId()))
                   .forEach(this::addChild);
  }
```

Micro - Engine - Random Execution Support

@SvenRuppert

```
@MicroTestClass(forceRandomExecution = true, useCDI = false)
public class ForceRandomExecTest
    implements HasLogger {

  //Test this with the Testkit !!!

  @MicroTest
  void test001() {
    logger().info("test001");
  }

  @MicroTest
  void test002() {
    logger().info("test002");
  }
```

Micro - Engine - Random Execution Support

@SvenRuppert

1 , 3 , 9

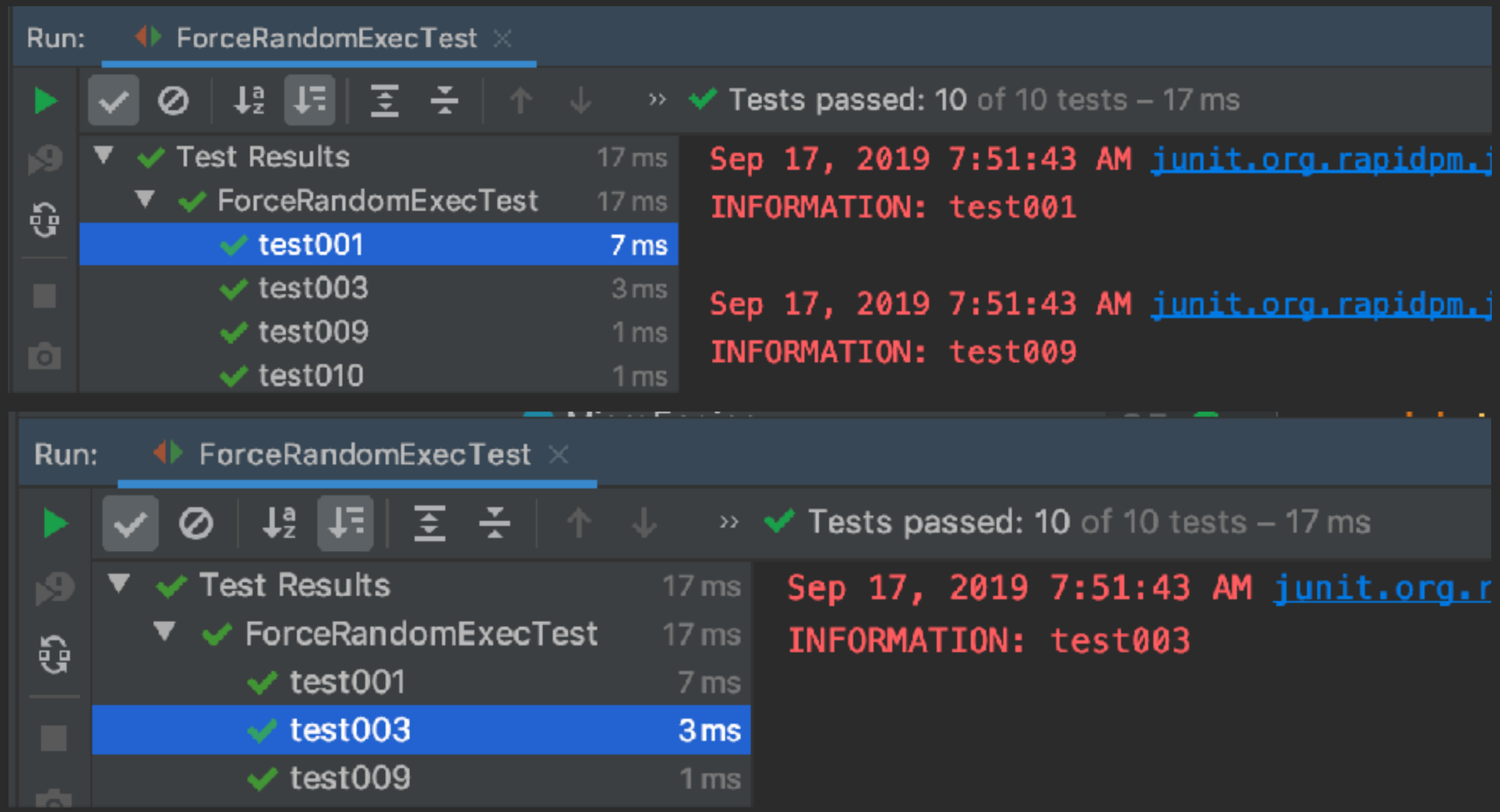
```
Sep 17, 2019 7:51:43 AM junit.org.
INFORMATION: test001

Sep 17, 2019 7:51:43 AM junit.org.
INFORMATION: test009

Sep 17, 2019 7:51:43 AM junit.org.
INFORMATION: test003
```

1 , 9 , 3

Don´t trust your IDE ;-)

Micro - Engine - Random Execution Support

@SvenRuppert



Don´t trust your IDE ;-)

Access to the Include-/Exclude-ClassnameFilters #2001

Closed svenruppert opened this issue 14 days ago · 9 comments

svenruppert commented 14 days ago Contributor

If you are writing a custom TestEngine, the discovery method will be the entrance for the selection of the TestClasses. If you want to select a package and you are using an IDE (IntelliJ in my case) the IDE will give you a ClasspathRooSelector in combination with an Include- and / ExcludeClassnameFilter.

`request.getFiltersByType(ClassNameFilter.class);` is not enough, because the implementing classes are not public.

Deliverables

- [ ] access to the pieces of information to build a proper TestDescriptor

Provide class-based TestEngine support #2011

Open **sormuras** opened this issue 3 days ago · 0 comments

sormuras commented 3 days ago — Member

The AbstractClassBasedTestEngine already implements the base logic needed to support a (Java) class-based `TestEngine`. Derived classes only have to specify what a test class and what a test method is, and all "communication" with the JUnit Platform is handled under the hood.

Deliverables

- [ ] Decide if JUnit Platform should provide such abstract `TestEngine` support class
- [ ] Another abstract support class `AbstractAnnotationBasedTestEngine<MyTest>` that handles a custom method annotation.

Enable CDI / Scopes

ForceRandomExecution

Concurrent Execution

Vaadin - Engine

A CUSTOM WEB ENGINE

@SvenRuppert

Vaadin - Engine

Manage the ServletContainer

Manage the WebDriver

Collect Statistics
Collect Results

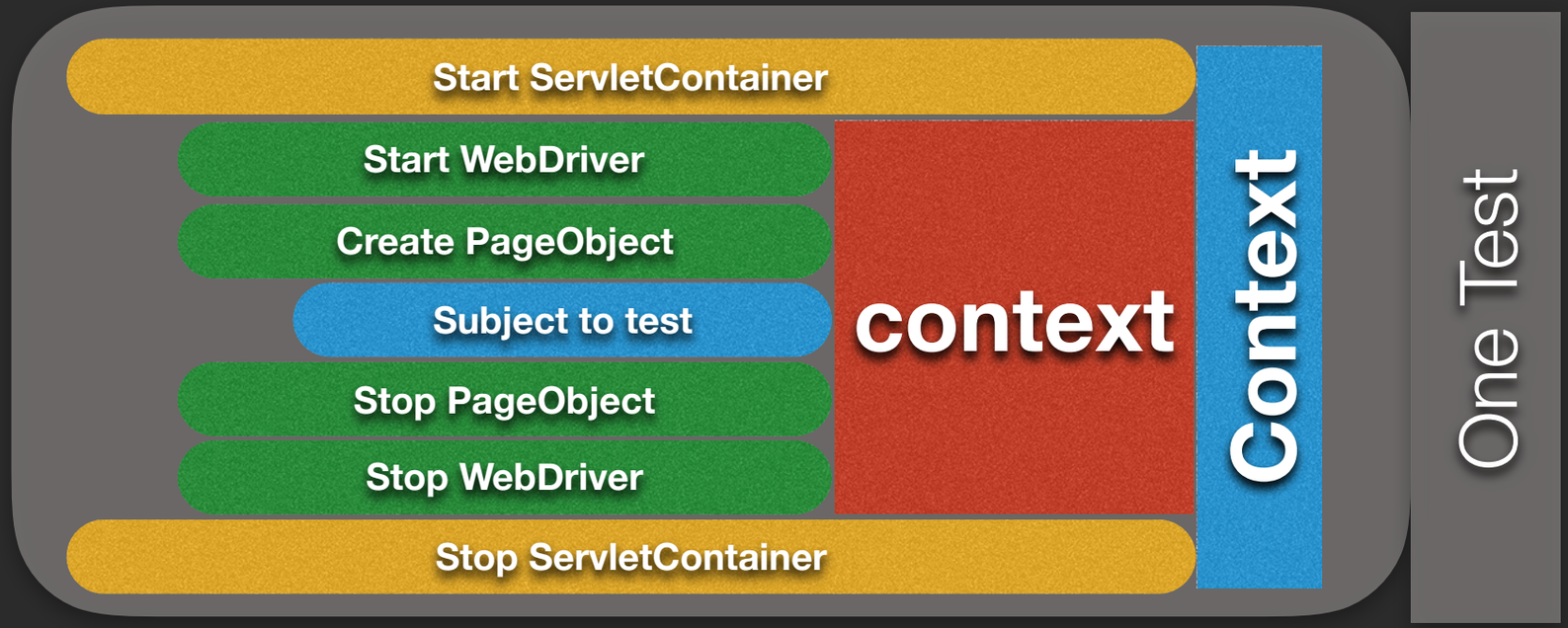
Configuration from outside to decide how many invocations are needed

Integrate Persistence

Concurrent Execution

Vaadin - Engine

@SvenRuppert



Dynamic config

The Jupiter Engine needs to create all input Values before the Test is started. **Stream<InputType>**

Lazy creation must be implemented by yourself.

Start before starting with the TestPlan is not implemented

Demo with Extensions

Distributed - Engine

HOW TO SCALE ON METHOD BASE

@SvenRuppert

Distributed - Engine

POC - Work In Progress

Summary

places to read more about it

www.distributed-unit-testing.com

Summary

Thank You !!!

please, follow me ;-)

@SvenRuppert