

Introducción y novedades de JUnit 5

MadridJUG - 16/01/2018

Boni García



boni.garcia@urjc.es



<http://bonigarcia.github.io/>



[@boni_gg](https://twitter.com/boni_gg)



<https://github.com/bonigarcia>

Boni García

- Soy doctor en sistemas telemáticos por la Universidad Politécnica de Madrid (UPM) desde 2011
 - Tesis doctoral centrada en las pruebas en el software (*testing*)
- Actualmente trabajo como:
 - Investigador en la Universidad Rey Juan Carlos (URJC)
 - Profesor en el Centro Universitario de Tecnología y Arte Digital (U-tad)
- Participo activamente en múltiples proyectos *open source*:
 - Comunidades: ElasTest, Kurento
 - Proyectos propios: WebDriverManager, selenium-jupiter, DualSub
- Soy autor del libro ***Mastering Software Testing with JUnit 5*** Pack Publishing (octubre 2017)



Contenidos

1. Introducción a las pruebas en el software
2. Motivación y arquitectura de JUnit 5
3. Jupiter: el nuevo modelo de programación de JUnit 5
4. Modelo de extensiones en Jupiter
5. Conclusiones

1. Introducción a las pruebas en el software

- La **calidad en el software** es un término bastante ambiguo en el mundo de la ingeniería de software. Según Roger Pressman:

“ *Software quality is an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it*

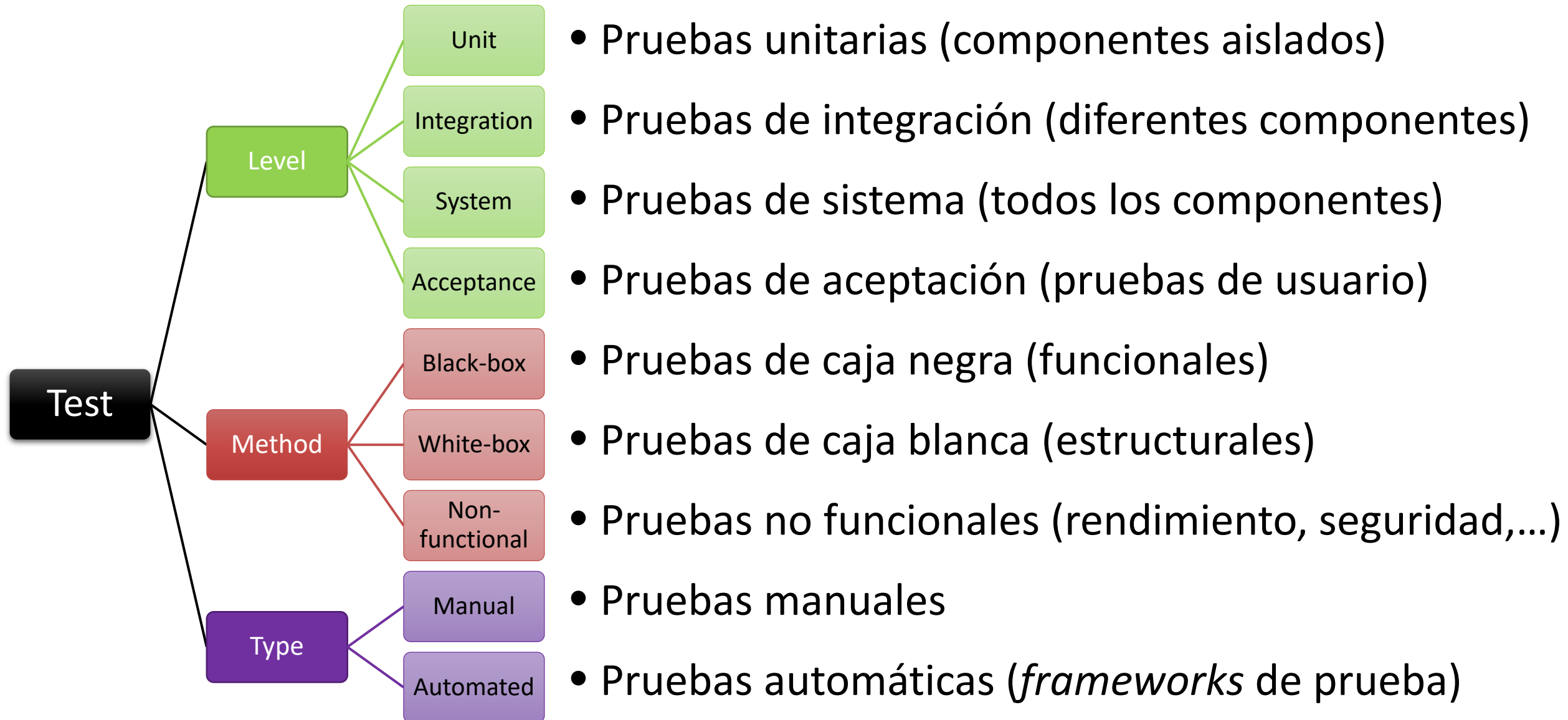
- El conjunto de técnicas destinadas a asegurar la calidad (QA, *Quality Assurance*) en un proyecto software se conocen como **Verificación y Validación** (V&V, *Verification & Validation*). Según Barry Boehm:

“ *Are we building the product right? (verification)*
Are we building the right product? (validation)

1. Introducción a las pruebas en el software

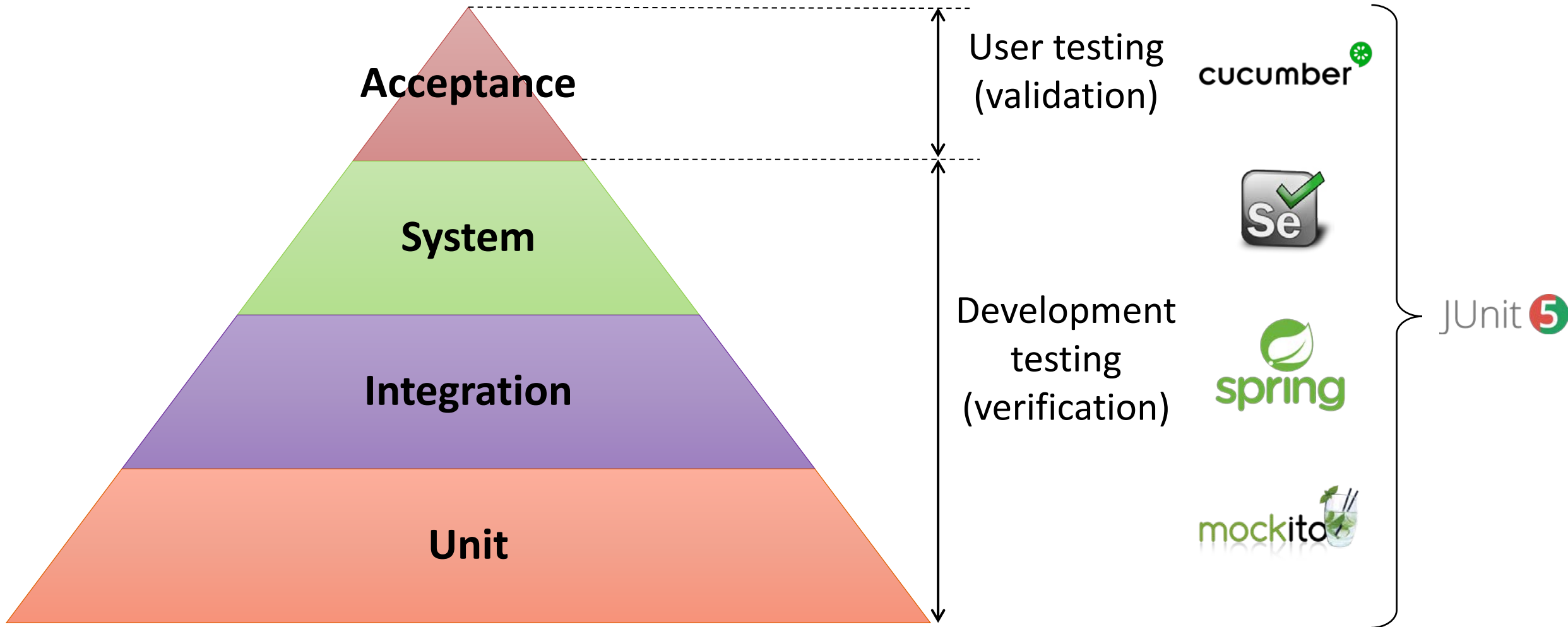
- Hay dos tipos de técnicas dentro de V&V:
 1. **Análisis** (estático): evaluación del software (ya sea código, modelos, documentación, etc.) sin ejecutar el propio software
 - Revisión de pares: Collaborator, Crucible, Gerrit ...
 - Análisis automático (*linters*): SonarQube, Checkstyle, FindBugs, PMD ...
 2. **Pruebas** (dinámico): evaluación del software observando un resultado *esperado* de la *ejecución* de una *parte o todo el sistema* (caso de prueba) y dar un *veredicto* sobre la misma

1. Introducción a las pruebas en el software



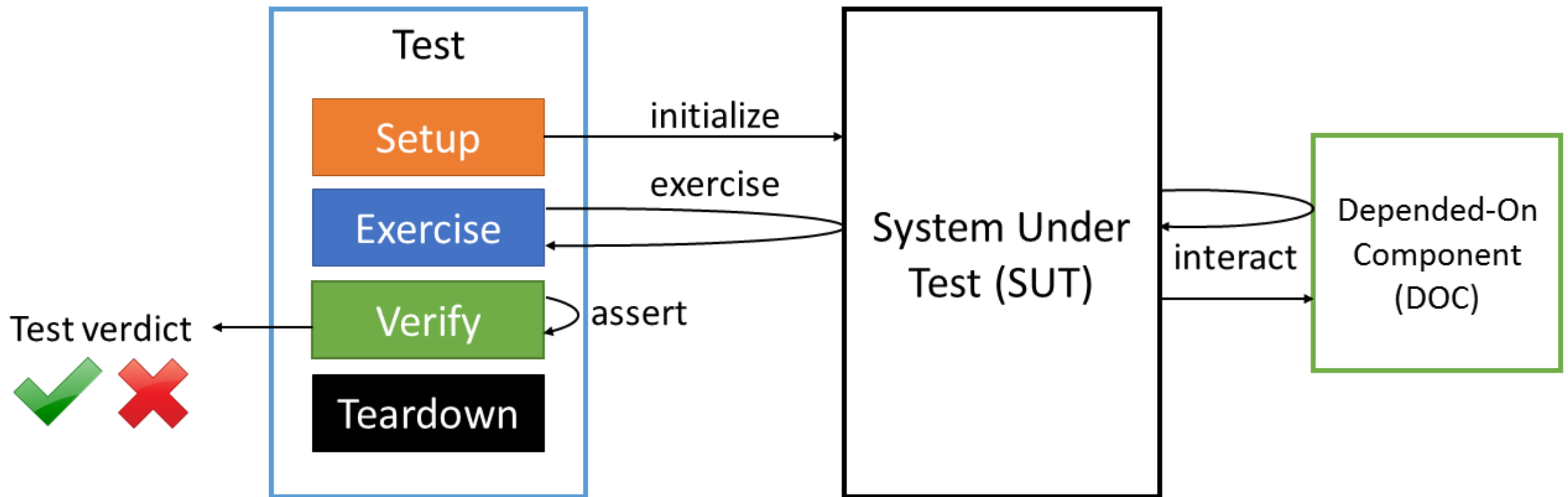
1. Introducción a las pruebas en el software

- Pirámide de tests:



1. Introducción a las pruebas en el software

- Esquema de pruebas unitarias según Gerard Meszaros (*xUnit Test Patterns*)



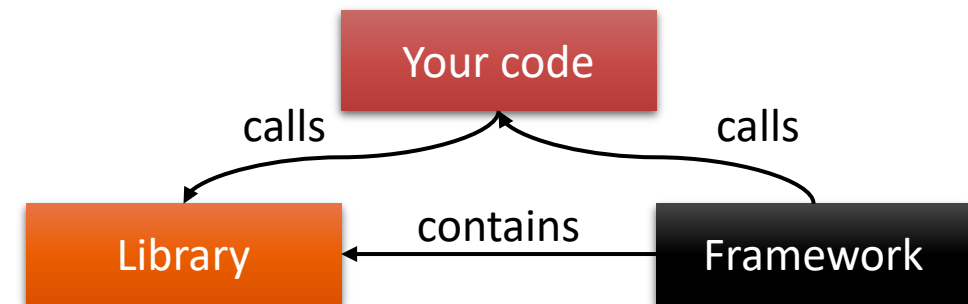
1. Introducción a las pruebas en el software

- Las **pruebas automáticas** según Elfriede Dustin:

” *Application and implementation of software technology throughout the entire software testing life cycle with the goal to improve efficiencies and effectiveness*

- Las pruebas automáticas son más efectivas cuando se implementan en base a un **framework**. Según Martin Folwer:

” *A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client. A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.*

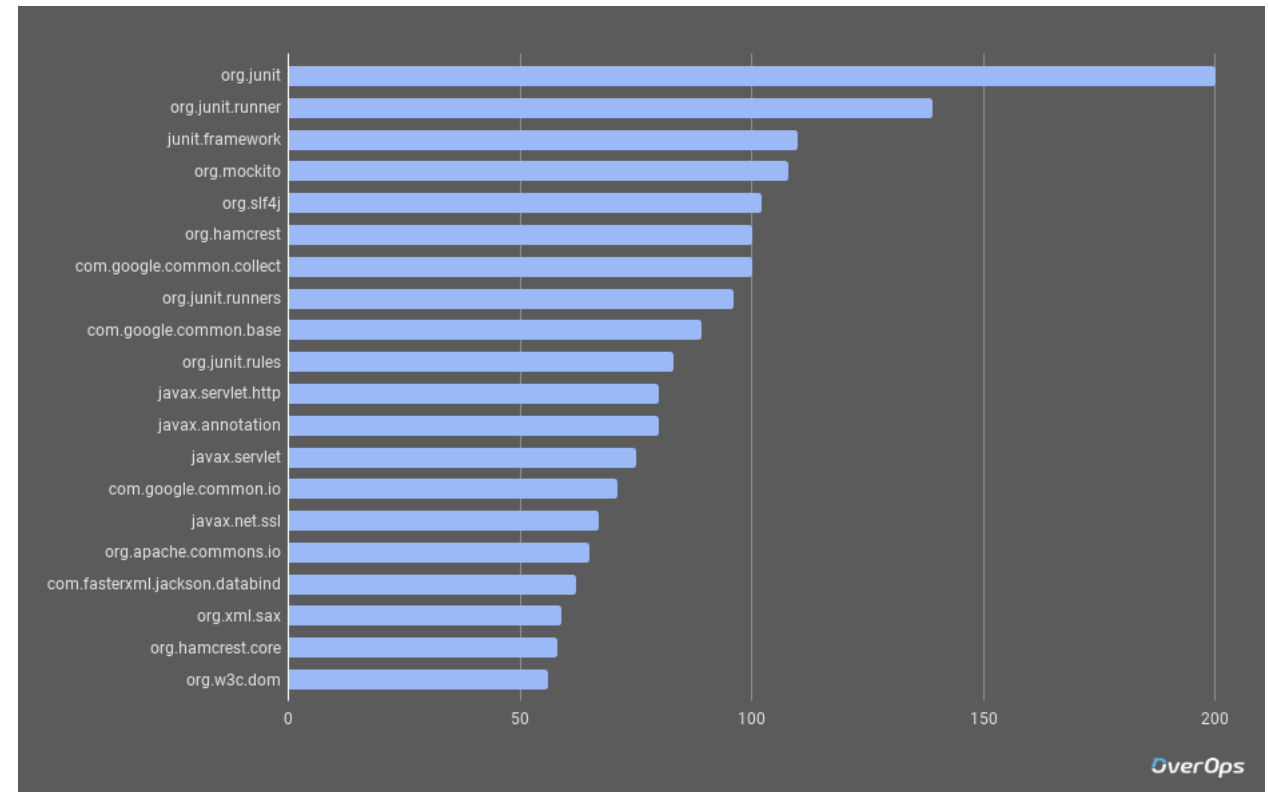


Contenidos

1. Introducción a las pruebas en el software
- 2. Motivación y arquitectura de JUnit 5**
3. Jupiter: el nuevo modelo de programación de JUnit 5
4. Modelo de extensiones en Jupiter
5. Conclusiones

2. Motivación y arquitectura de JUnit 5

- JUnit es el framework más utilizado en la comunidad Java y uno de los más influyentes en la ingeniería de software en general (precursor de la familia xUnit)
- La última versión de JUnit 4 fue liberada en diciembre de 2014
- JUnit 4 tiene importantes limitaciones que han propiciado el rediseño completo del framework en JUnit 5

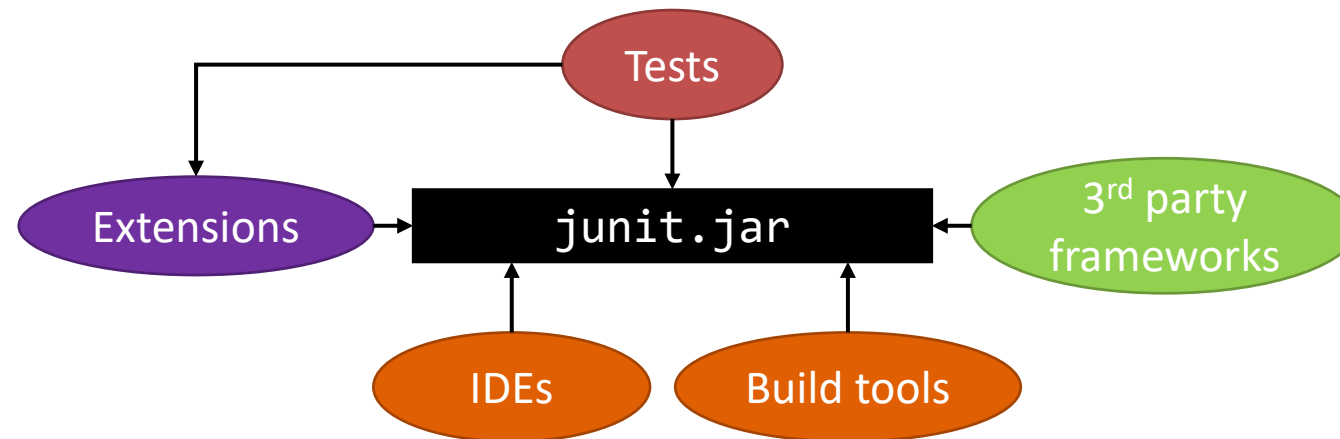


[The Top 100 Java libraries on GitHub \(by OverOps\)](#)

2. Motivación y arquitectura de JUnit 5



- Los principales inconvenientes de JUnit 4 son:
 1. JUnit 4 es **monolítico**. Toda las funciones de JUnit 4 son proporcionadas por un único componente, de forma que mecanismos como el descubrimiento de tests y la ejecución de los mismos están muy acoplados



2. Motivación y arquitectura de JUnit 5



- Los principales inconvenientes de JUnit 4 son:
2. Los casos de prueba se ejecutan en JUnit 4 mediante unas clases especiales llamadas ***Test Runners***. Estos *runners* tienen una limitación fundamental: no se pueden componer

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class MyTest1 {

    @Test
    public void myTest() {
        // my test code
    }

}
```

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
public class MyTest2 {

    @Test
    public void myTest() {
        // my test code
    }

}
```


2. Motivación y arquitectura de JUnit 5



- Los principales inconvenientes de JUnit 4 son:
3. Para mejorar la gestión del ciclo de vida de tests en JUnit 4 se desarrollaron las reglas (***Test Rules***), implementadas con las anotaciones `@Rule` y `@ClassRule`. El inconveniente es que puede llegar a ser complicado gestionar ambos ámbitos de forma simultánea (*runners y rules*)

```
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ErrorCollector;

public class MyTest3 {

    @Rule
    public ErrorCollector errorCollector = new ErrorCollector();

    @Test
    public void myTest() {
        // my test code
    }

}
```

```
import org.junit.ClassRule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

public class MyTest4 {

    @ClassRule
    public TemporaryFolder temporaryFolder = new TemporaryFolder();

    @Test
    public void myTest() {
        // my test code
    }

}
```

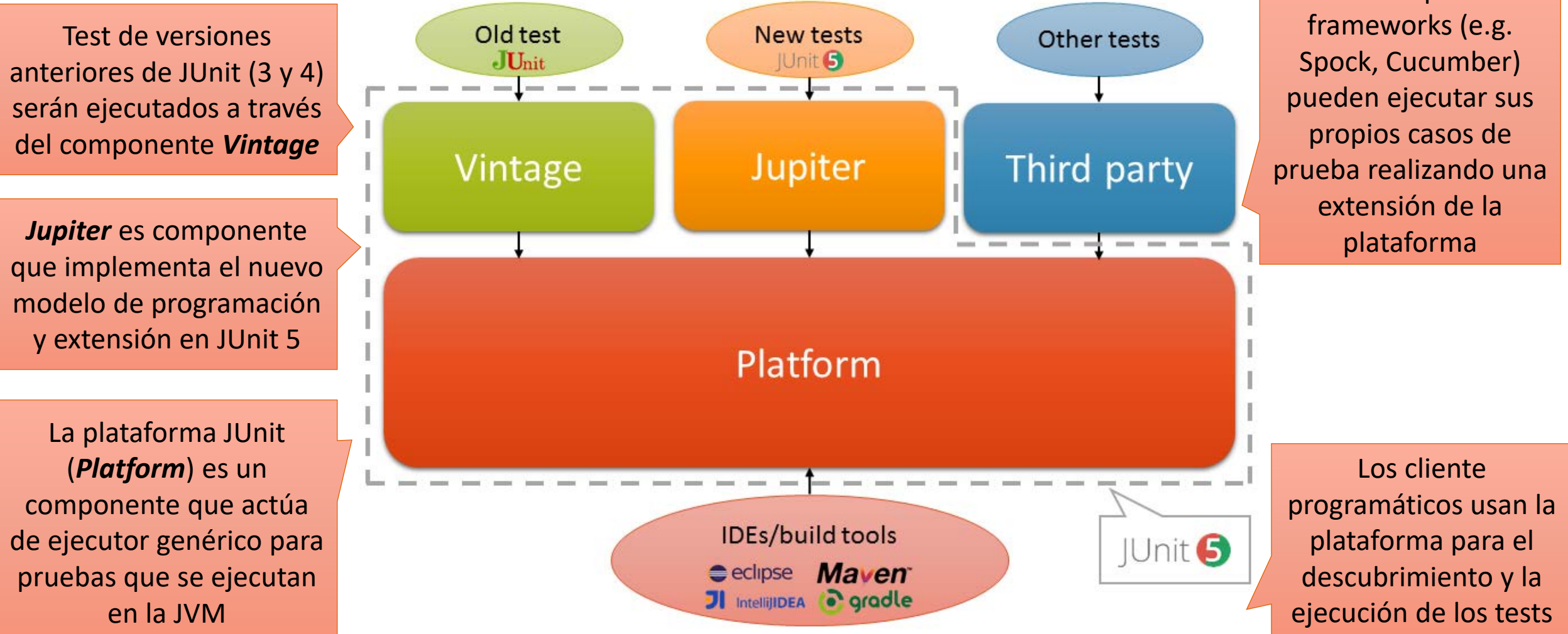
2. Motivación y arquitectura de JUnit 5

- Para intentar solucionar estos problemas, en julio de 2015 Johannes Link y Mark Philipp pusieron en marcha una campaña para recaudar fondos y crear una nueva versión de JUnit
- Esta campaña se conoció como [*JUnit Lambda crowdfunding campaign*](#)
- Gracias a esta campaña se puso en marcha el equipo de **JUnit 5**, con miembros de diferentes compañías (Eclipse, Gradle, e IntelliJ entre otras)



2. Motivación y arquitectura de JUnit 5

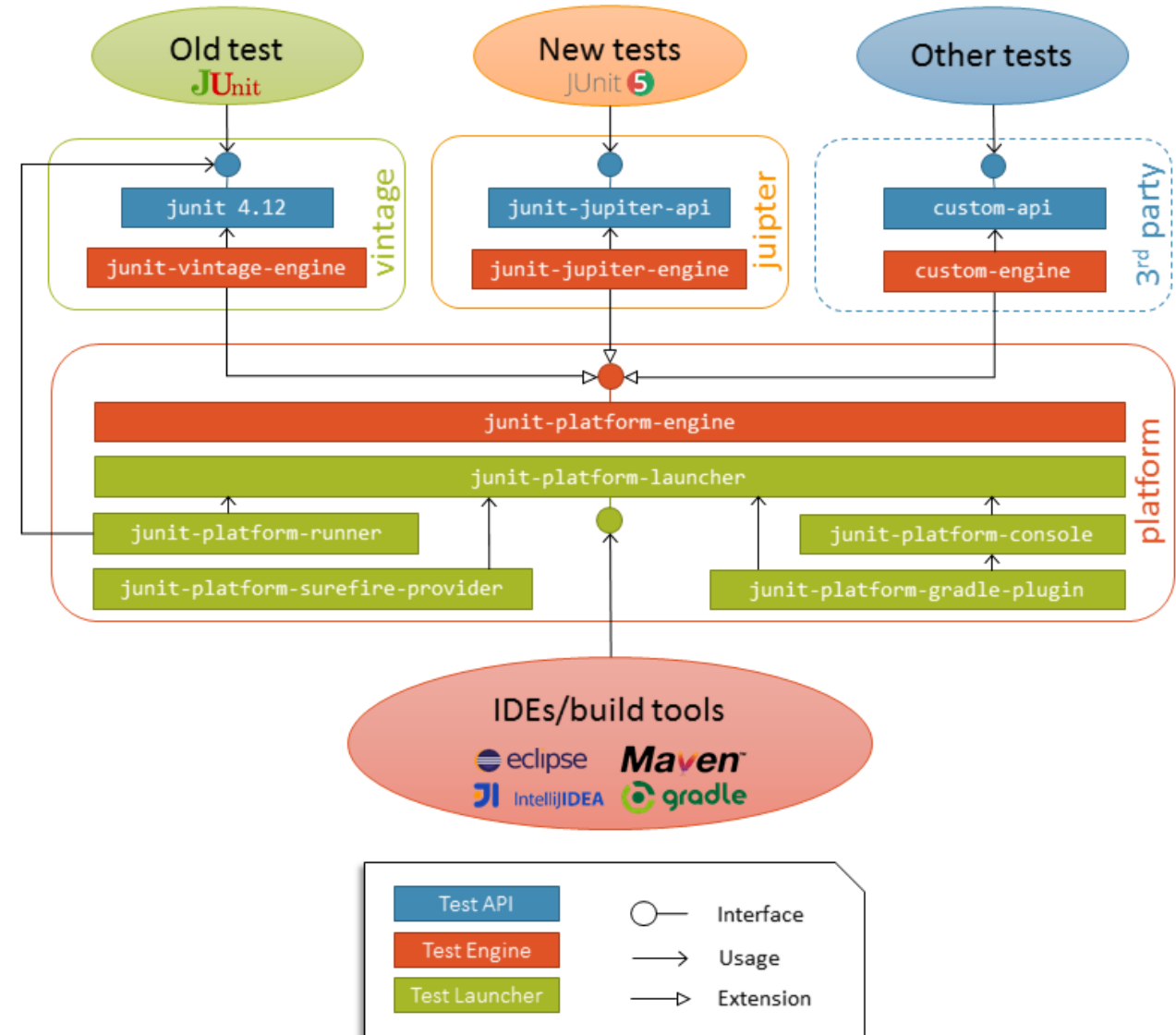
- Arquitectura de JUnit 5:



2. Motivación y arquitectura de JUnit 5

- Hay tres tipos de módulos:

1. **Test API:** Módulos usados por *testers* para implementar casos de prueba
2. **Test Engine SPI:** Módulos extendidos para un *framework* de pruebas Java para la ejecución de un modelo concreto de tests
3. **Test Launcher API:** Módulos usados por *clientes programáticos* para el descubrimiento de tests

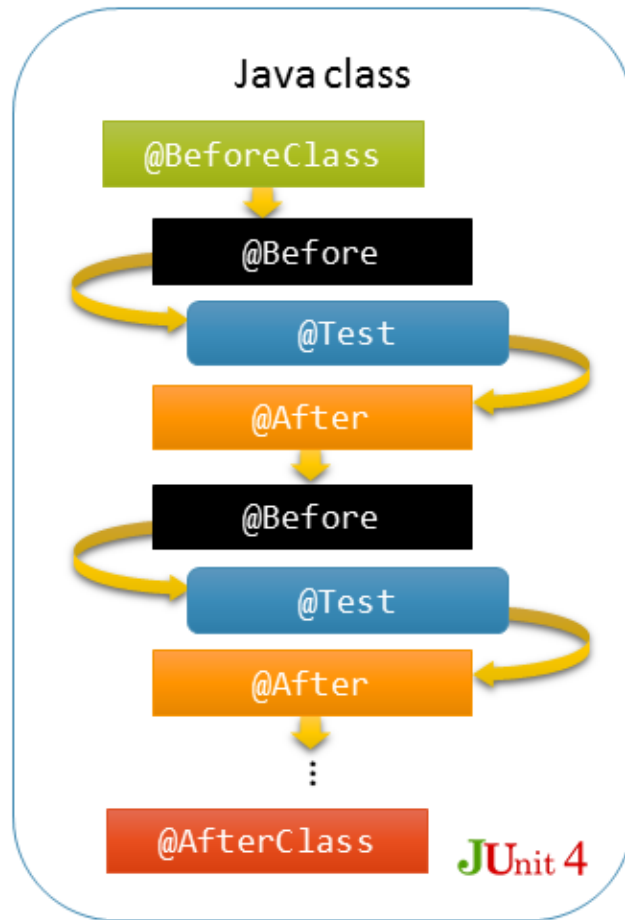


Contenidos

1. Introducción a las pruebas en el software
2. Motivación y arquitectura de JUnit 5
- 3. Jupiter: el nuevo modelo de programación de JUnit 5**
4. Modelo de extensiones en Jupiter
5. Conclusiones

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Los **tests básicos** en Jupiter son muy parecidos a JUnit 4:



JUnit

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class BasicJUnit4Test {

    @BeforeClass
    public static void setupAll() {
        // setup all tests
    }

    @Before
    public void setup() {
        // setup each test
    }

    @Test
    public void test() {
        // exercise and verify SUT
    }

    @After
    public void teardown() {
        // teardown each test
    }

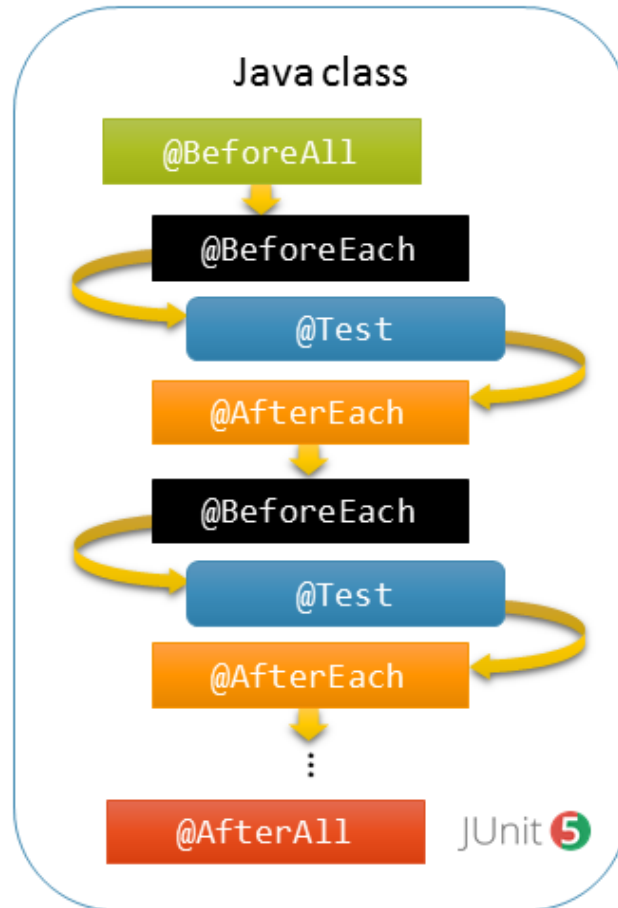
    @AfterClass
    public static void teardownAll() {
        // teardown all tests
    }

}
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Los **tests básicos** en Jupiter son muy parecidos a JUnit 4:

El nombre de las anotaciones que gestionan el ciclo de vida básico de los tests ha cambiado en Jupiter



```
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
class BasicJUnit5Test {
    @BeforeAll
    static void setupAll() {
        // setup all tests
    }

    @BeforeEach
    void setup() {
        // setup each test
    }

    @Test
    void test() {
        // exercise and verify SUT
    }

    @AfterEach
    void teardown() {
        // teardown each test
    }

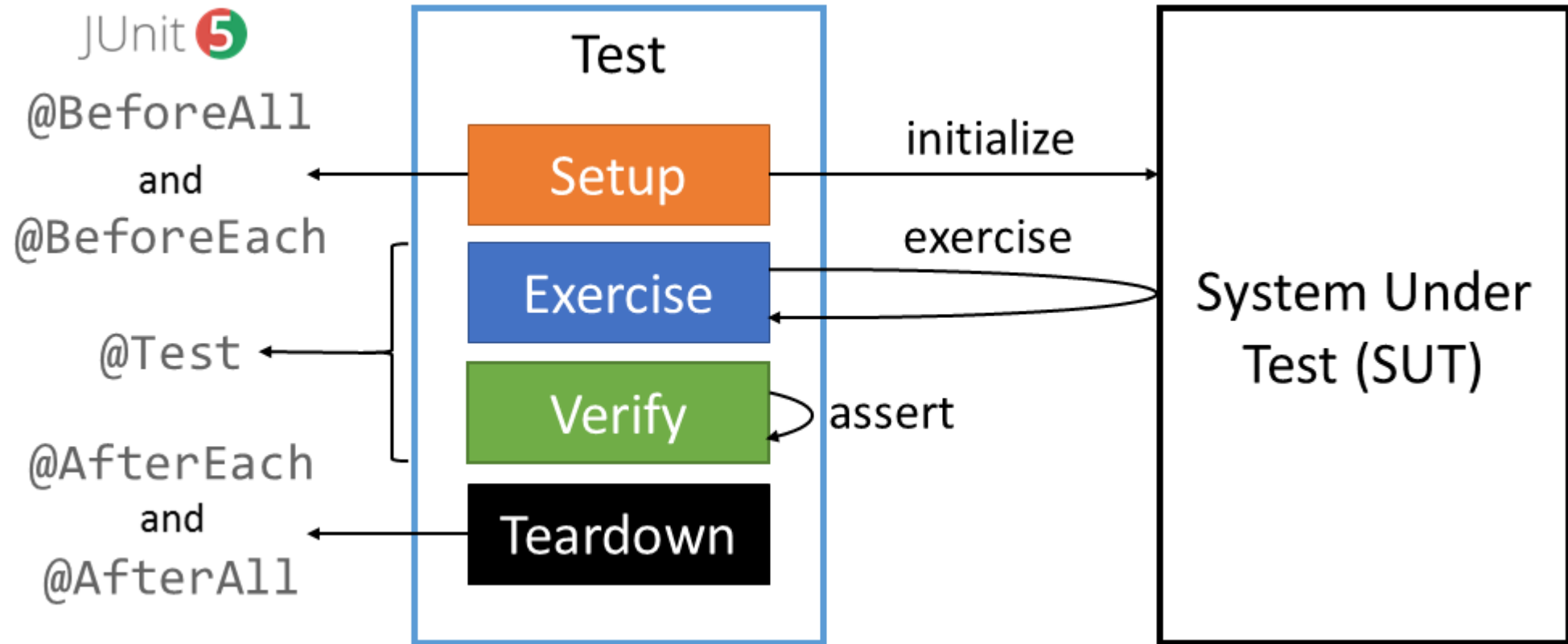
    @AfterAll
    static void teardownAll() {
        // teardown all tests
    }
}
```

JUnit 5

Se elimina la necesidad que las clases y los métodos de prueba tengan que ser **public**

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Podemos ver el **ciclo de vida** de un caso de prueba en JUnit 5 de la siguiente manera:



3. Jupiter: el nuevo modelo de programación de JUnit 5

- Actualmente se pueden **ejecutar** test de JUnit 5 de diferentes formas:

1. Mediante herramienta de gestión y construcción de proyectos Java (*build tools*)

- Maven
- Gradle



2. Mediante un Entorno de Desarrollo Integrado (IDE)

- IntelliJ IDEA 2016.2+
- Eclipse 4.7+



3. Mediante una herramienta propia de JUnit 5

- Console Launcher (*standalone* jar que permite ejecutar tests JUnit 5)

```
java -jar junit-platform-console-standalone-version.jar <Options>
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Ejecutar JUnit 5 desde **Maven**:

```
<properties>
  <!-- Dependency versions -->
  <junit.jupiter.version>5.0.3</junit.jupiter.version>
  <junit.platform.version>1.0.3</junit.platform.version>
  <maven-surefire-plugin.version>2.19.1</maven-surefire-plugin.version>
</properties>

<!-- Jupiter API for writing tests -->
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



Todos los ejemplos están disponibles en GitHub
<https://github.com/bonigarcia/mastering-junit5>

Fork me on GitHub

```
<!-- Maven Surefire plugin to run tests -->
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>${maven-surefire-plugin.version}</version>
      <dependencies>
        <dependency>
          <groupId>org.junit.platform</groupId>
          <artifactId>junit-platform-surefire-provider</artifactId>
          <version>${junit.platform.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter-engine</artifactId>
          <version>${junit.jupiter.version}</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
```


3. Jupiter: el nuevo modelo de programación de JUnit 5

- Ejecutar JUnit 5 desde **Gradle**:

```
buildscript {
    ext {
        junitPlatformVersion = '1.0.3'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.junit.platform:junit-platform-gradle-plugin:${junitPlatformVersion}")
    }
}

repositories {
    mavenCentral()
}

ext {
    junitJupiterVersion = '5.0.3'
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.junit.platform.gradle.plugin'
```

```
compileTestJava {
    sourceCompatibility = 1.8
    targetCompatibility = 1.8
    options.compilerArgs += '-parameters'
}

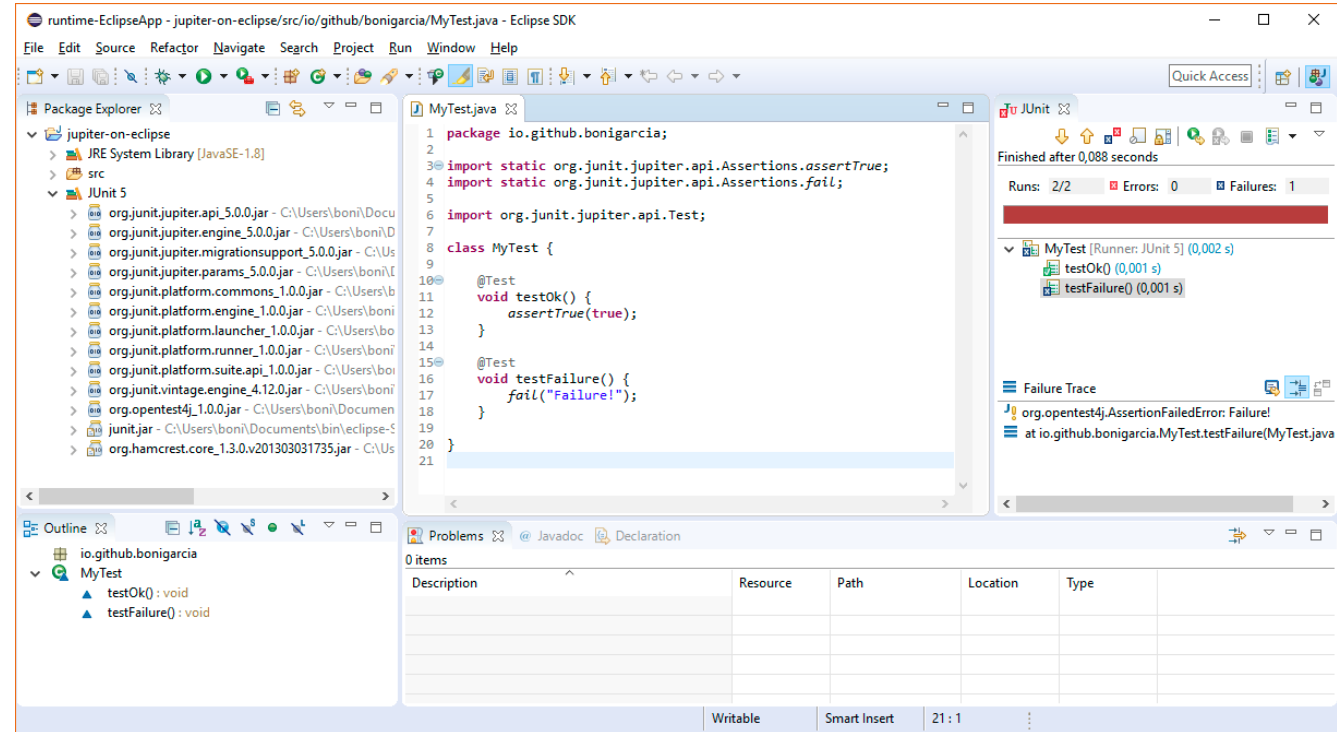
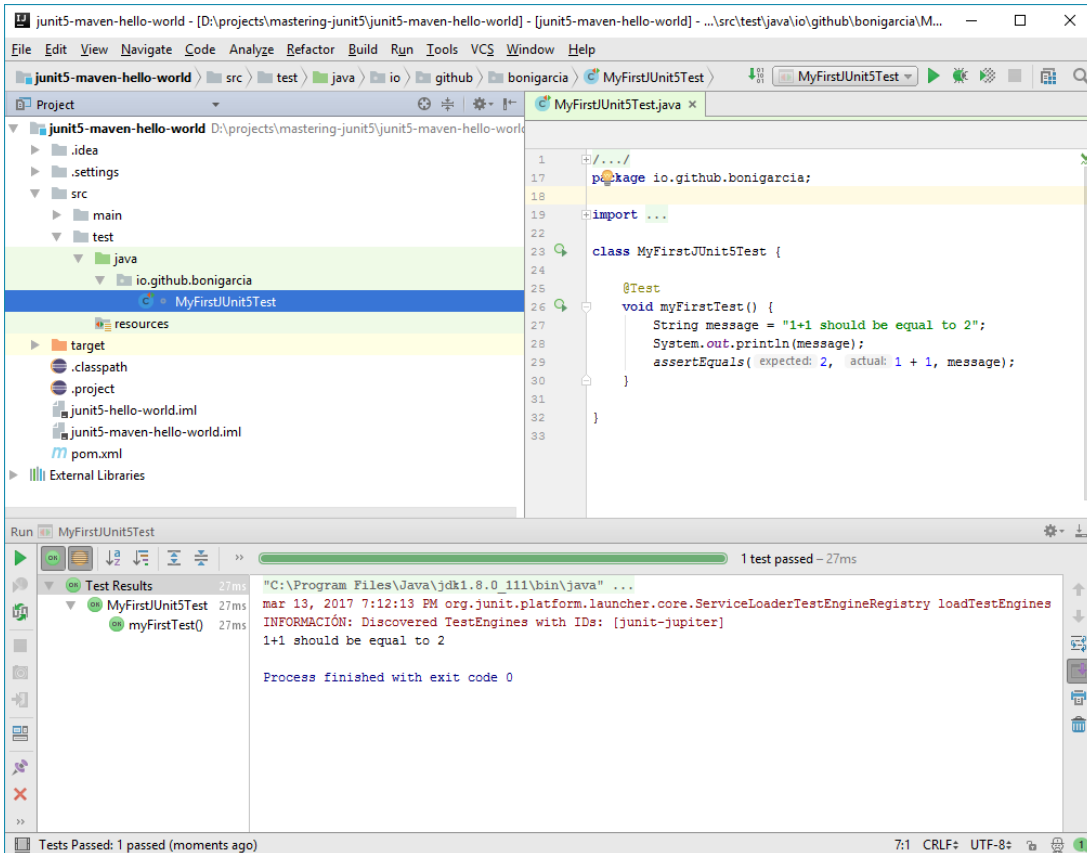
dependencies {
    testCompile("org.junit.jupiter:junit-jupiter-api:${junitJupiterVersion}")
    testRuntime("org.junit.jupiter:junit-jupiter-engine:${junitJupiterVersion}")
}
```



Fork me on GitHub

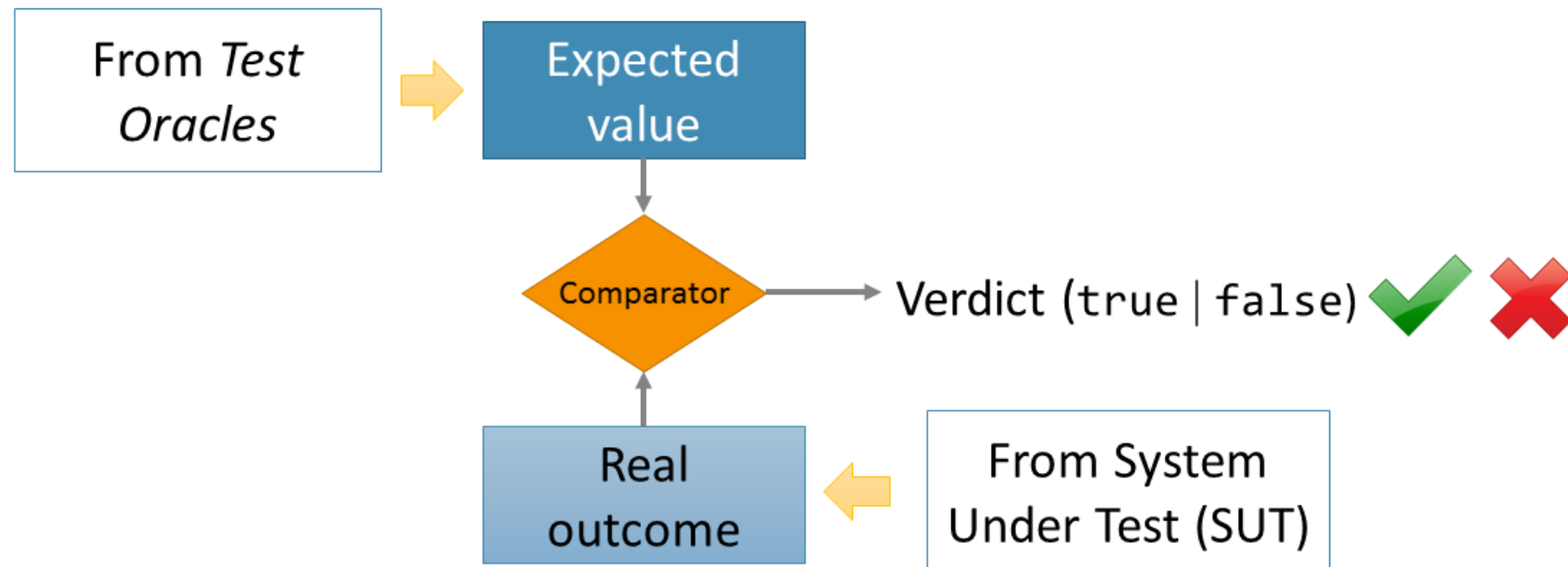
3. Jupiter: el nuevo modelo de programación de JUnit 5

- Ejecutar JUnit 5 desde **IntelliJ** y **Eclipse**:



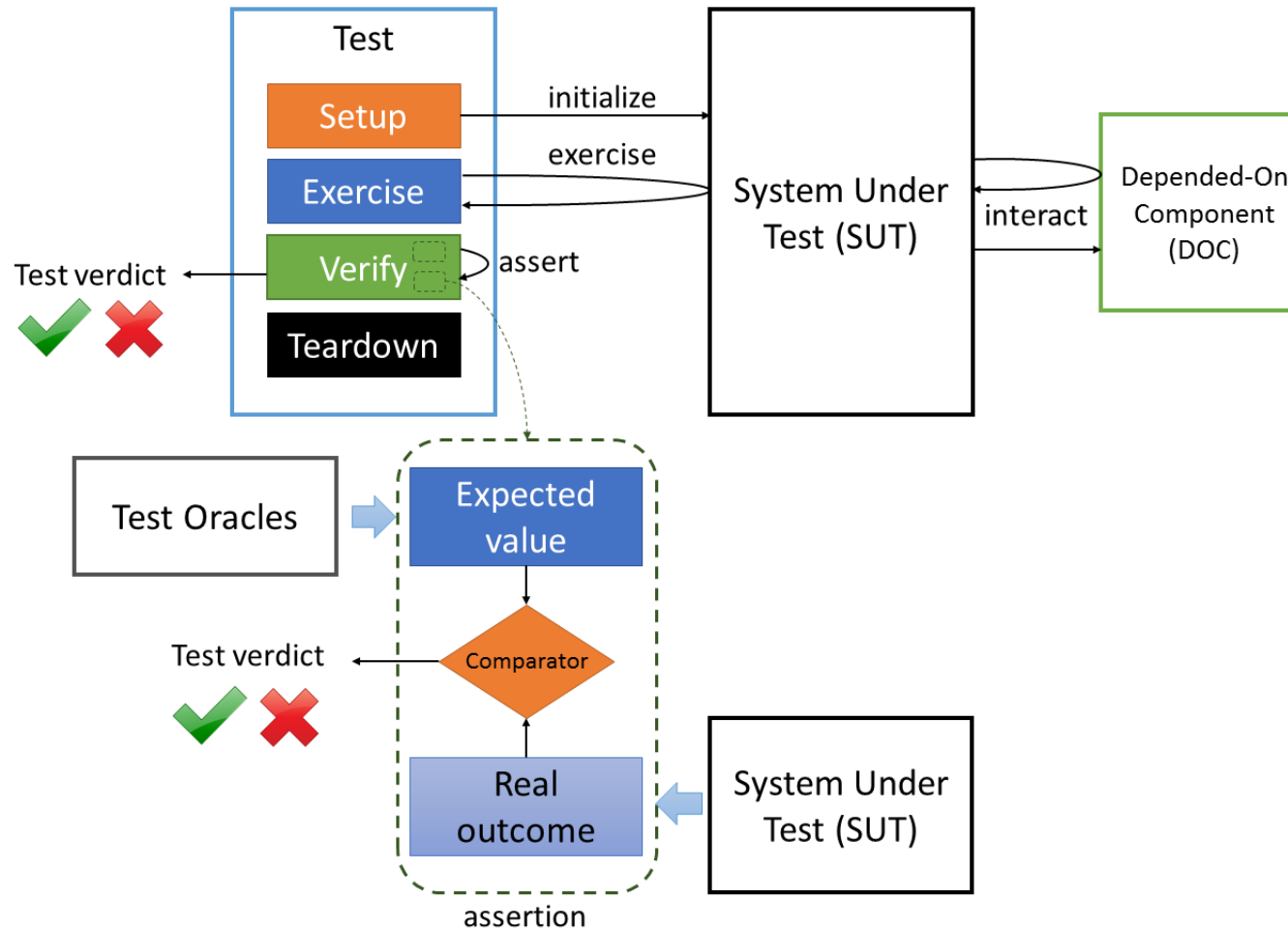
3. Jupiter: el nuevo modelo de programación de JUnit 5

- De forma conceptual, una **aserción** (o predicado) está formada por
 - Datos esperados, obtenidos de lo que se conoce como oráculo (típicamente la especificación del SUT)
 - Datos reales, obtenidos de ejercitar el sistema bajo pruebas (SUT)
 - Un operador lógico que compara ambos valores



3. Jupiter: el nuevo modelo de programación de JUnit 5

- Podemos ver el **ciclo de vida** un caso de prueba en JUnit 5 (con aserciones) de la siguiente manera:



3. Jupiter: el nuevo modelo de programación de JUnit 5

- Las aserciones básicas en Jupiter son las siguientes:

Métodos estáticos
de la clase
Assertions

| Aserción | Descripción |
|-----------------------------------|--|
| <code>fail</code> | Hace fallar un test proporcionando un mensaje de error u excepción |
| <code>assertTrue</code> | Evalúa si una condición es cierta |
| <code>assertFalse</code> | Evalúa si una condición es false |
| <code>assertNull</code> | Evalúa si un objeto es null |
| <code>assertNotNull</code> | Evalúa si un objeto no es null |
| <code>assertEquals</code> | Evalúa si un objeto es igual a otro |
| <code>assertNotEquals</code> | Evalúa si un objeto no es igual a otro |
| <code>assertArrayEquals</code> | Evalúa si un array es igual a otros |
| <code>assertIterableEquals</code> | Evalúa si dos objetos iterables son iguales |
| <code>assertLinesMatch</code> | Evlúa si dos listas de String son iguales |
| <code>assertSame</code> | Evalúa si un objeto es el mismo que otro |
| <code>assertNotSame</code> | Evalúa si un objeto no es el mismo que otro |

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Un grupo de aserciones se evalúa mediante *assertAll*:

```
import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class GroupedAssertionsTest {

    @Test
    void groupedAssertions() {
        Address address = new Address("John", "Smith");

        assertAll("address", () -> assertEquals("John", address.getFirstName()),
            () -> assertEquals("User", address.getLastName()));
    }
}
```

En este ejemplo la
segunda aserción no se
cumple

T E S T S

Running io.github.bonigarcia.GroupedAssertionsTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time
elapsed: 0.124 sec <<< FAILURE! - in
io.github.bonigarcia.GroupedAssertionsTest
groupedAssertions() Time elapsed: 0.08 sec <<< FAILURE!
org.opentest4j.MultipleFailuresError:
address (1 failure)
 expected: <User> but was: <Smith>
 at
io.github.bonigarcia.GroupedAssertionsTest.groupedAssertio
ns(GroupedAssertionsTest.java:32)

Results :

Failed tests:
 GroupedAssertionsTest.groupedAssertions:32 address (1
failure)
 expected: <User> but was: <Smith>

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Las ocurrencia de excepciones se implementa mediante la aserción *assertThrows*:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

import org.junit.jupiter.api.Test;

class ExceptionTest {

    @Test
    void exceptionTesting() {
        Throwable exception = assertThrows(IllegalArgumentException.class,
            () -> {
                throw new IllegalArgumentException("a message");
            });
        assertEquals("a message", exception.getMessage());
    }
}
```

En este ejemplo el test pasará ya que estamos esperando la excepción `IllegalArgumentException`, y de hecho ocurre dentro una expression lambda

```
-----
T E S T S
-----
Running io.github.bonigarcia.ExceptionTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.153 sec
- in io.github.bonigarcia.ExceptionTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Para evaluar *timeouts* podemos usar la aserción *assertTimeout*.

Fork me on GitHub

```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertTimeout;

import org.junit.jupiter.api.Test;

class TimeoutExceededTest {

    @Test
    void timeoutNotExceeded() {
        assertTimeout(ofMinutes(2), () -> {
            // Perform task that takes less than 2 minutes
        });
    }

    @Test
    void timeoutExceeded() {
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms
            Thread.sleep(100);
        });
    }
}
```

Este test
pasará

Este test
fallará

```
-----
T E S T S
-----
Running io.github.bonigarcia.TimeoutExceededTest
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time
elapsed: 0.18 sec <<< FAILURE! - in
io.github.bonigarcia.TimeoutExceededTest
timeoutExceeded() Time elapsed: 0.126 sec <<< FAILURE!
org.opentest4j.AssertionFailedError: execution exceeded
timeout of 10 ms by 90 ms
    at
io.github.bonigarcia.TimeoutExceededTest.timeoutExceeded(
TimeoutExceededTest.java:36)

Results :

Failed tests:
  TimeoutExceededTest.timeoutExceeded:36 execution
exceeded timeout of 10 ms by 90 ms

Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Si queremos aserciones todavía más avanzadas, se recomienda usar librerías específicas, como por ejemplo **Hamcrest**:

Fork me on GitHub

```
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.notNullValue;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

class HamcrestTest {

    @Test
    void assertWithHamcrestMatcher() {
        assertThat(2 + 1, equalTo(3));
        assertThat("Foo", notNullValue());
        assertThat("Hello world", containsString("world"));
    }
}
```

T E S T S

Running io.github.bonigarcia.HamcrestTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.059 sec - in io.github.bonigarcia.HamcrestTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0



<http://hamcrest.org/>

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Podemos declarar **nombres personalizados** a los métodos de prueba mediante la anotación `@DisplayName`:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameTest {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {

    }

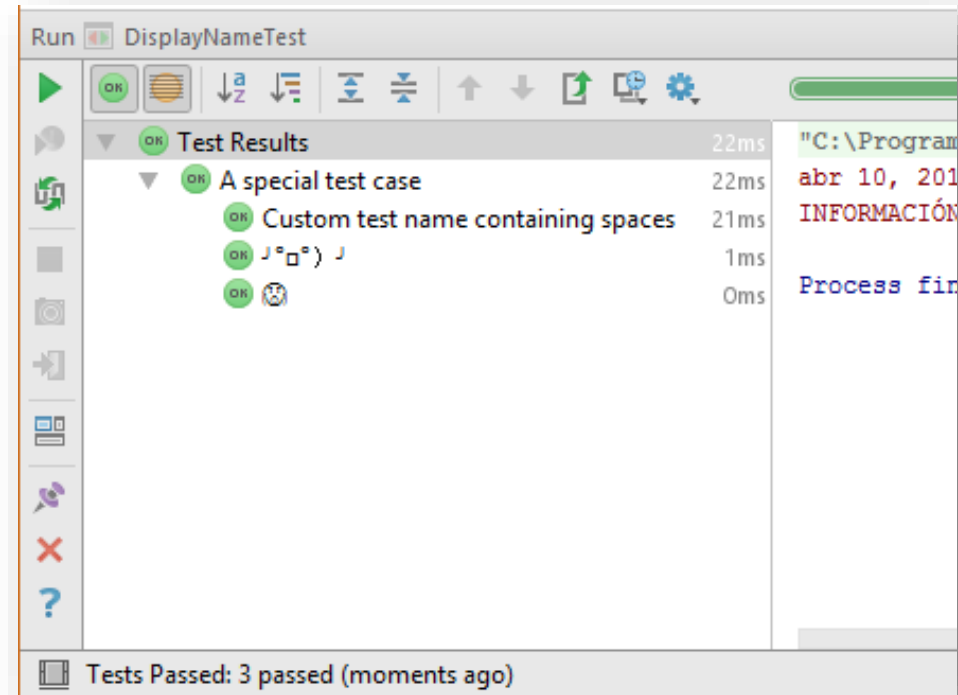
    @Test
    @DisplayName("J o _ o J")
    void testWithDisplayNameContainingSpecialCharacters() {

    }

    @Test
    @DisplayName("🐼")
    void testWithDisplayNameContainingEmoji() {

    }

}
```



IntelliJ IDEA: 2016.2+

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Las clases y métodos de test en JUnit 5 se pueden **etiquetar** usando la anotación `@Tag`
- Estas etiquetas se pueden usar después para el descubrimiento y ejecución de los test (**filtrado**)

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("functional")
class FunctionalTest {

    @Test
    void test1() {
        System.out.println("Functional Test 1");
    }

    @Test
    void test2() {
        System.out.println("Functional Test 2");
    }

}
```

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("non-functional")
class NonFunctionalTest {

    @Test
    @Tag("performance")
    @Tag("load")
    void test1() {
        System.out.println("Non-Functional Test 1 (Performance/Load)");
    }

    @Test
    @Tag("performance")
    @Tag("stress")
    void test2() {
        System.out.println("Non-Functional Test 2 (Performance/Stress)");
    }

    @Test
    @Tag("security")
    void test3() {
        System.out.println("Non-Functional Test 3 (Security)");
    }

    @Test
    @Tag("usability")
    void test4() {
        System.out.println("Non-Functional Test 4 (Usability)");
    }

}
```

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Para filtrar por tags en Maven:



```
-----
T E S T S
-----
Running io.github.bonigarcia.FunctionalTest
Functional Test 2
Functional Test 1
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.075 sec - in
io.github.bonigarcia.FunctionalTest
Running io.github.bonigarcia.NonFunctionalTest
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0 sec - in
io.github.bonigarcia.NonFunctionalTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

```
<!-- Maven Surefire plugin to run tests -->
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>${maven-surefire-plugin.version}</version>
      <configuration>
        <properties>
          <includeTags>functional</includeTags>
          <excludeTags>non-functional</excludeTags>
        </properties>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Para filtrar por tags en Gradle:



```
junitPlatform {  
    filters {  
        engines {  
            include 'junit-jupiter'  
            exclude 'junit-vintage'  
        }  
        tags {  
            include 'functional', 'non-functional'  
            exclude ''  
        }  
        packages {  
            include 'io.github.bonigarcia'  
            exclude 'com.others', 'org.others'  
        }  
        includeClassNamePattern '.*Spec'  
        includeClassNamePatterns '.*Test', '.*Tests'  
    }  
}
```

```
mastering-junit5\junit5-tagging-filtering-gradle>gradle test --rerun-tasks  
:compileJava NO-SOURCE  
:processResources NO-SOURCE  
:classes UP-TO-DATE  
:compileTestJava  
:processTestResources NO-SOURCE  
:testClasses  
:junitPlatformTest  
Functional Test 2  
Functional Test 1  
Non-Functional Test 3 (Security)  
Non-Functional Test 2 (Performance/Stress)  
Non-Functional Test 4 (Usability)  
Non-Functional Test 1 (Performance/Load)  
  
Test run finished after 78 ms  
[      3 containers found      ]  
[      0 containers skipped    ]  
[      3 containers started    ]  
[      0 containers aborted    ]  
[      3 containers successful ]  
[      0 containers failed     ]  
[      6 tests found           ]  
[      0 tests skipped         ]  
[      6 tests started         ]  
[      0 tests aborted         ]  
[      6 tests successful      ]  
[      0 tests failed          ]  
  
:test SKIPPED  
  
BUILD SUCCESSFUL  
  
Total time: 1.977 secs
```

Fork me on GitHub

3. Jupiter: el nuevo modelo de programación de JUnit 5

- La anotación `@Disabled` se usa para **deshabilitar** tests. Se puede usar tanto a nivel de clase como a nivel de método

Fork me on GitHub

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTest {

    @Disabled
    @Test
    void skippedTest() {

    }

}
```

```
-----
T E S T S
-----
Running io.github.bonigarcia.DisabledTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 1, Time
elapsed: 0.03 sec - in io.github.bonigarcia.DisabledTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 1
```

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled("All test in this class will be skipped")
class AllDisabledTest {

    @Test
    void skippedTest1() {

    }

    @Test
    void skippedTest2() {

    }

}
```

```
-----
T E S T S
-----
Running io.github.bonigarcia.AllDisabledTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 1, Time elapsed:
0.059 sec - in io.github.bonigarcia.AllDisabledTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 1
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Las **asunciones** sirven para ignorar un test (o una parte del mismo) en base a una condición
- Hay tres asunciones en JUnit 5: *assumeTrue*, *assumeFalse*, y *assumingThat*

TESTS

```
org.junit.platform.launcher.core.ServiceLoaderTestEngineRegistry
loadTestEngines
INFORMACIÓN: Discovered TestEngines with IDs: [junit-jupiter]
Running io.github.bonigarcia.AssumptionsTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 2, Time elapsed:
0.093 sec - in io.github.bonigarcia.AssumptionsTest
```

Results :

```
Tests run: 3, Failures: 0, Errors: 0, Skipped: 2
```

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeFalse;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;
```

```
import org.junit.jupiter.api.Test;
```

```
class AssumptionsTest {
```

```
    @Test
```

```
    void assumeTrueTest() {
        assumeTrue(false);
        fail("Test 1 failed");
    }
```

```
    @Test
```

```
    void assumeFalseTest() {
        assumeFalse(this::getTrue);
        fail("Test 2 failed");
    }
```

```
    private boolean getTrue() {
        return true;
    }
```

```
    @Test
```

```
    void assumingThatTest() {
        assumingThat(false, () -> fail("Test 3 failed"));
    }
```

```
}
```

Fork me on GitHub

La asunción *assumingThat* se usa para condicionar la ejecución de una parte del test

3. Jupiter: el nuevo modelo de programación de JUnit 5

- La anotación `@RepeatedTest` permite **repetir** la ejecución de un test un número determinado de veces
- Cada repetición se comportará exactamente igual que un test normal, con su correspondiente ciclo de vida

Fork me on GitHub

```
import org.junit.jupiter.api.RepeatedTest;

class SimpleRepeatedTest {

    @RepeatedTest(5)
    void test() {
        System.out.println("Repeated test");
    }

}
```

```
-----
T E S T S
-----
Running io.github.bonigarcia.SimpleRepeatedTest
Repeated test
Repeated test
Repeated test
Repeated test
Repeated test
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.11 sec - in io.github.bonigarcia.SimpleRepeatedTest

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Los **test parametrizados** reutilizan la misma lógica con diferentes datos de prueba
- Para implementar este tipo de test, lo primero es añadir el módulo `junit-jupiter-params` en nuestro proyecto



```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



```
dependencies {
    testCompile("org.junit.jupiter:junit-jupiter-params:${junitJupiterVersion}")
}
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

- Los pasos para implementar un test parametrizado son:
 1. Usar la anotación `@ParameterizedTest` para declarar un test como parametrizado
 2. Elegir un proveedor de argumentos (*argument provider*)

| Arguments provider | Descripción |
|-------------------------------|---|
| <code>@ValueSource</code> | Usado para especificar un array de valores <code>String</code> , <code>int</code> , <code>long</code> , o <code>double</code> |
| <code>@EnumSource</code> | Usado para especificar valores enumerados (<code>java.lang.Enum</code>) |
| <code>@MethodSource</code> | Usado para especificar un método estático de la clase que proporciona un <code>Stream</code> de valores |
| <code>@CsvSource</code> | Usado para especificar valores separados por coma, esto es, en formato CSV (<i>comma-separated values</i>) |
| <code>@CsvFileSource</code> | Usado para especificar valores en formato CSV en un fichero localizado en el classpath |
| <code>@ArgumentsSource</code> | Usado para especificar una clase que implementa el interfaz <code>org.junit.jupiter.params.provider.ArgumentsProvider</code> |

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class ValueSourcePrimitiveTypesParameterizedTest {

    @ParameterizedTest
    @ValueSource(ints = { 0, 1 })
    void testWithInts(int argument) {
        System.out
            .println("Parameterized test with (int) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(longs = { 2L, 3L })
    void testWithLongs(long argument) {
        System.out.println(
            "Parameterized test with (long) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(doubles = { 4d, 5d })
    void testWithDoubles(double argument) {
        System.out.println(
            "Parameterized test with (double) argument: " + argument);
        assertNotNull(argument);
    }
}
```

@ValueSource

Fork me on GitHub

```
-----
T E S T S
-----
Running io.github.bonigarcia.ValueSourcePrimitiveTypesParameterizedTest
Parameterized test with (int) argument: 0
Parameterized test with (int) argument: 1
Parameterized test with (long) argument: 2
Parameterized test with (long) argument: 3
Parameterized test with (double) argument: 4.0
Parameterized test with (double) argument: 5.0
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.134 sec - in
io.github.bonigarcia.ValueSourcePrimitiveTypesParameterizedTest

Results :

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertNotNull;

import java.util.concurrent.TimeUnit;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EnumSource;

class EnumSourceParameterizedTest {

    @ParameterizedTest
    @EnumSource(TimeUnit.class)
    void testWithEnum(TimeUnit argument) {
        System.out.println(
            "Parameterized test with (TimeUnit) argument: " + argument);
        assertNotNull(argument);
    }
}
```

@EnumSource

Fork me on GitHub

TESTS

```
Running io.github.bonigarcia.EnumSourceParameterizedTest
Parameterized test with (TimeUnit) argument: NANoseconds
Parameterized test with (TimeUnit) argument: MICROSECONDS
Parameterized test with (TimeUnit) argument: MILLISECONDS
Parameterized test with (TimeUnit) argument: SECONDS
Parameterized test with (TimeUnit) argument: MINUTES
Parameterized test with (TimeUnit) argument: HOURS
Parameterized test with (TimeUnit) argument: DAYS
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.31 sec - in
io.github.bonigarcia.EnumSourceParameterizedTest
```

Results :

Tests run: 7, Failures: 0, Errors: 0, Skipped: 0

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertNotNull;

import java.util.stream.Stream;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class MethodSourceStringsParameterizedTest {

    static Stream<String> stringProvider() {
        return Stream.of("hello", "world");
    }

    @ParameterizedTest
    @MethodSource("stringProvider")
    void testWithStringProvider(String argument) {
        System.out.println(
            "Parameterized test with (String) argument: " + argument);
        assertNotNull(argument);
    }
}
```

@MethodSource

Fork me on GitHub

```
-----
T E S T S
-----
Running io.github.bonigarcia.MethodSourceStringsParameterizedTest
Parameterized test with (String) argument: hello
Parameterized test with (String) argument: world
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.297 sec - in
io.github.bonigarcia.MethodSourceStringsParameterizedTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class CsvSourceParameterizedTest {

    @ParameterizedTest
    @CsvSource({ "hello, 1", "world, 2", "'happy, testing', 3" })
    void testWithCsvSource(String first, int second) {
        System.out.println("Parameterized test with (String) " + first
            + " and (int) " + second);

        assertNotNull(first);
        assertNotEquals(0, second);
    }
}
```

@CsvSource

Fork me on GitHub

```
-----
T E S T S
-----
Running io.github.bonigarcia.CsvSourceParameterizedTest
Parameterized test with (String) hello and (int) 1
Parameterized test with (String) world and (int) 2
Parameterized test with (String) happy, testing and (int) 3
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.121 sec - in
io.github.bonigarcia.CsvSourceParameterizedTest

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

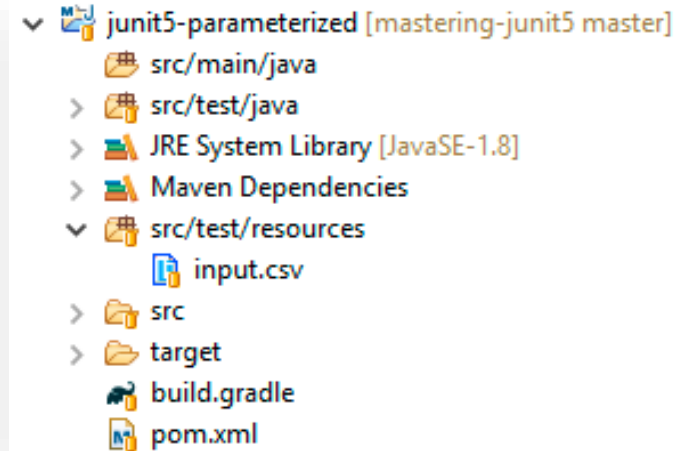
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class CsvFileSourceParameterizedTest {

    @ParameterizedTest
    @CsvFileSource(resources = "/input.csv")
    void testWithCsvFileSource(String first, int second) {
        System.out.println("Yet another parameterized test with (String) "
            + first + " and (int) " + second);

        assertNotNull(first);
        assertEquals(0, second);
    }
}
```

@CsvFileSource



junit5-parameterized [mastering-junit5 master]

- src/main/java
- src/test/java
- JRE System Library [JavaSE-1.8]
- Maven Dependencies
- src/test/resources
 - input.csv
- src
- target
- build.gradle
- pom.xml

Fork me on GitHub

```
TESTS
-----
Running io.github.bonigarcia.CsvFileSourceParameterizedTest
Yet another parameterized test with (String) Mastering and (int) 4
Yet another parameterized test with (String) JUnit 5 and (int) 5
Yet another parameterized test with (String) hi, there and (int) 6
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.235 sec - in
io.github.bonigarcia.CsvFileSourceParameterizedTest

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

```
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ArgumentsSource;

class ArgumentSourceParameterizedTest {

    @ParameterizedTest
    @ArgumentsSource(CustomArgumentsProvider1.class)
    void testWithArgumentsSource(String first, int second) {
        System.out.println("Parameterized test with (String) "
            + first + " and (int) " + second);

        assertNotNull(first);
        assertTrue(second > 0);
    }
}
```

@ArgumentsSource

```
import java.util.stream.Stream;

import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.ArgumentsProvider;

public class CustomArgumentsProvider1 implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(
        ExtensionContext context) {

        System.out.println("Arguments provider [1] to test "
            + context.getTestMethod().get().getName());

        return Stream.of(Arguments.of("hello", 1),
            Arguments.of("world", 2));
    }
}
```

Fork me on GitHub

T E S T S

```
Running io.github.bonigarcia.ArgumentSourceParameterizedTest
Arguments provider to test testWithArgumentsSource
Parameterized test with (String) hello and (int) 1
Parameterized test with (String) world and (int) 2
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.287 sec - in
io.github.bonigarcia.ArgumentSourceParameterizedTest
```

Results :

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

3. Jupiter: el nuevo modelo de programación de JUnit 5

Fork me on GitHub

- A medio cambio entre los test parametrizados y el modelo de extensión nos encontramos las **plantillas de tests**
- Los métodos anotados con `@TestTemplate` se ejecutarán múltiples veces dependiendo de los valores devueltos por el proveedor de contexto
- Las plantillas de test se usan siempre en conjunción con el punto de extensión `TestTemplateInvocationContextProvider`

```
import org.junit.jupiter.api.TestTemplate;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MyTestTemplateInvocationContextProvider.class)
class TemplateTest {

    @TestTemplate
    void testTemplate(String parameter) {
        System.out.println(parameter);
    }

}
```

En este caso deberíamos implementar la extensión `MyTestTemplateInvocationContextProvider`

Contenidos

1. Introducción a las pruebas en el software
2. Motivación y arquitectura de JUnit 5
3. Jupiter: el nuevo modelo de programación de JUnit 5
- 4. Modelo de extensiones en Jupiter**
5. Conclusiones

4. Modelo de extensiones en Jupiter

- El modelo de extensión de Jupiter permite añadir el modelo de programación con funcionalidades personalizadas
- Gracias al modelo de extensiones, frameworks externos pueden proporcionar integración con JUnit 5 de una manera sencilla
- Para poder usar una extensión en un caso de prueba usaremos la anotación `@ExtendWith` (se puede usar a nivel de clase o de método)

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MyExtension.class)
public class MyTest {

    @Test
    public void test() {
        // My test logic
    }

}
```

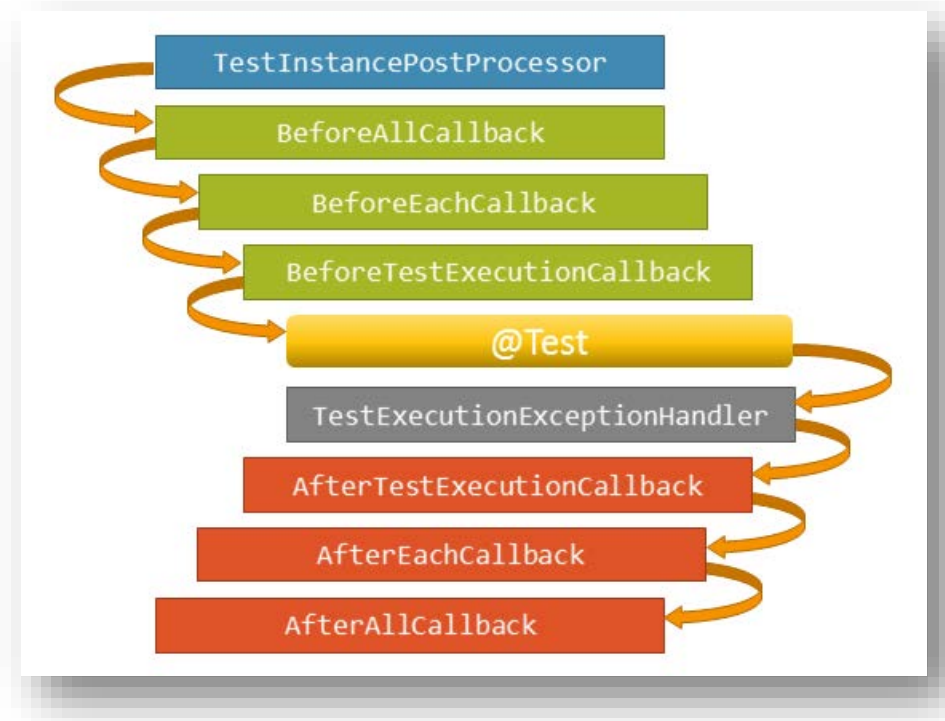
4. Modelo de extensiones en Jupiter

- Las extensiones en Jupiter se implementan haciendo uso de los llamados **puntos de extensión**
- Los puntos de extensión son interfaces que permiten declarar cuatro tipos de operaciones:
 1. Añadir nueva lógica dentro del **ciclo de vida** de tests
 2. Realizar **ejecución condicional** de tests
 3. Realizar **inyección de dependencias** en métodos de tests
 4. Gestionar las **plantillas** de test

4. Modelo de extensiones en Jupiter

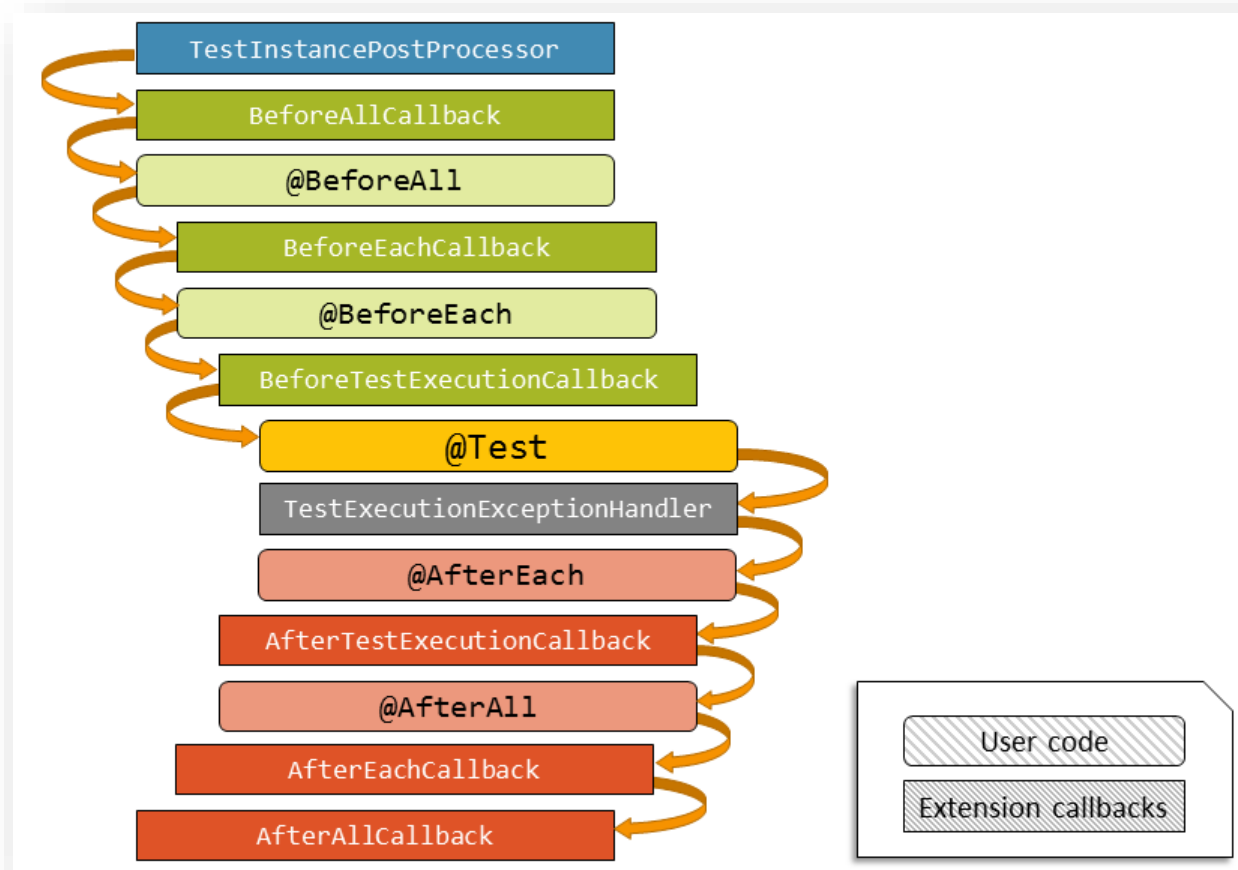
- Los puntos de extensión que controlan el **ciclo de vida** de tests son los siguientes:

| Punto de extensión | Implementadas por extensiones que se ejecutarán... |
|-------------------------------|---|
| TestInstancePostProcessor | Justo después de la instanciación del test |
| BeforeAllCallback | Antes de todos los tests de una clase |
| BeforeEachCallback | Antes de cada test |
| BeforeTestExecutionCallback | Justo antes de cada test |
| TestExecutionExceptionHandler | Justo después de que ocurra una de excepciones en el test |
| AfterTestExecutionCallback | Justo después de cada test |
| AfterEachCallback | Después de cada test |
| AfterAllCallback | Después de todos los tests |



4. Modelo de extensiones en Jupiter

- El ciclo de vida completo teniendo en puntos de extensión y código de usuario (@BeforeAll, @BeforeEach, @AfterEach, @AfterAll) es:



4. Modelo de extensiones en Jupiter

- Ejemplo: extensión que ignora las excepciones de tipo `IOException`

Fork me on GitHub

```
package io.github.bonigarcia;

import java.io.IOException;

import org.junit.jupiter.api.extension.TestExecutionExceptionHandler;
import org.junit.jupiter.api.extension.TestExtensionContext;

public class IgnoreIOExceptionExtension
    implements TestExecutionExceptionHandler {

    @Override
    public void handleTestExecutionException(TestExtensionContext context,
        Throwable throwable) throws Throwable {

        if (throwable instanceof IOException) {
            return;
        }
        throw throwable;
    }
}
```

En este ejemplo el primer caso de prueba pasará mientras el segundo fallará

```
package io.github.bonigarcia;

import java.io.IOException;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

public class ExceptionTest {

    @ExtendWith(IgnoreIOExceptionExtension.class)
    @Test
    public void test1() throws IOException {
        throw new IOException("My IO Exception");
    }

    @Test
    public void test2() throws IOException {
        throw new IOException("My IO Exception");
    }
}
```


4. Modelo de extensiones en Jupiter

- La **ejecución condicional** de tests se realiza mediante el punto de extensión `ExecutionCondition`:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.extension.ConditionEvaluationResult;
import org.junit.jupiter.api.extension.ExecutionCondition;
import org.junit.jupiter.api.extension.ExtensionContext;

public class MyConditionalExtension implements ExecutionCondition {

    @Override
    public ConditionEvaluationResult evaluateExecutionCondition(
        ExtensionContext arg0) {
        boolean condition = ...
        if (condition) {
            return ConditionEvaluationResult.enabled("reason");
        }
        else {
            return ConditionEvaluationResult.disabled("reason");
        }
    }
}
```

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

@ExtendWith(MyConditionalExtension.class)
class MyExtensionTest {

    @Test
    void test() {
        // my test
    }
}
```

En función de lo que devuelva la extensión, los tests se ejecutarán o serán ignorados

4. Modelo de extensiones en Jupiter

- La **inyección de dependencias** se realiza mediante el punto de extensión **ParameterResolver**:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.api.extension.ParameterResolutionException;
import org.junit.jupiter.api.extension.ParameterResolver;

public class MyParameterResolver implements ParameterResolver {

    @Override
    public boolean supportsParameter(ParameterContext parameterContext,
                                    ExtensionContext extensionContext)
        throws ParameterResolutionException {
        return true;
    }

    @Override
    public Object resolveParameter(ParameterContext parameterContext,
                                   ExtensionContext extensionContext)
        throws ParameterResolutionException {
        return "my parameter";
    }
}
```

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;

public class MyTest {

    @ExtendWith(MyParameterResolver.class)
    @Test
    public void test(Object parameter) {
        System.out.println("---> parameter " + parameter);
    }
}
```

```
-----
T E S T S
-----
mar 13, 2017 5:37:36 PM
org.junit.platform.launcher.core.ServiceLoaderTestEngineRegistry
loadTestEngines
INFORMACIÓN: Discovered TestEngines with IDs: [junit-jupiter]
Running io.github.bonigarcia.MyTest
---> parameter my parameter
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.059 sec - in io.github.bonigarcia.MyTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Fork me on GitHub

4. Modelo de extensiones en Jupiter

- La gestión de **plantillas** de tests (@TestTemplate) se implementa mediante el punto de extensión TestTemplateInvocationContextProvider

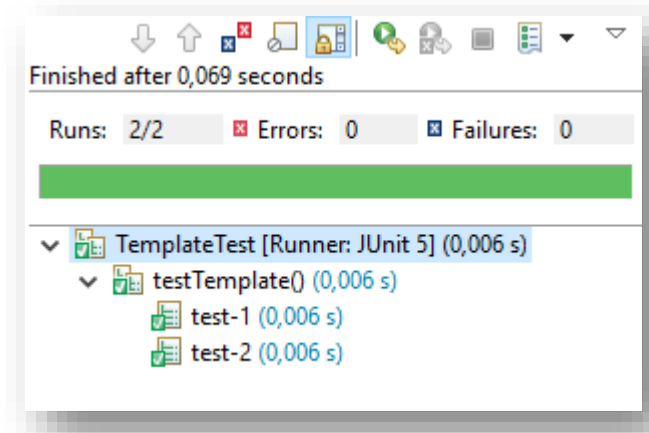
Fork me on GitHub

```
import java.util.stream.Stream;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.TestTemplateInvocationContext;
import org.junit.jupiter.api.extension.TestTemplateInvocationContextProvider;

public class MyTestTemplateInvocationContextProvider implements TestTemplateInvocationContextProvider {
    @Override
    public boolean supportsTestTemplate(ExtensionContext context) {
        return true;
    }

    @Override
    public Stream<TestTemplateInvocationContext> provideTestTemplateInvocationContexts(
        ExtensionContext context) {
        return Stream.of(invocationContext("test-1"), invocationContext("test-2"));
    }

    private TestTemplateInvocationContext invocationContext(String parameter) {
        return new TestTemplateInvocationContext() {
            @Override
            public String getDisplayName(int invocationIndex) {
                return parameter;
            }
        };
    }
}
```



```
package io.github.bonigarcia;

import org.junit.jupiter.api.TestTemplate;
import org.junit.jupiter.api.extension.ExtendWith;

class TemplateTest {

    @TestTemplate
    @ExtendWith(MyTestTemplateInvocationContextProvider.class)
    void testTemplate() {
    }

}
```

4. Modelo de extensiones en Jupiter

- Extensión de **Mockito** para JUnit 5 (pruebas unitarias):

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

import java.lang.reflect.Parameter;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ExtensionContext.Namespace;
import org.junit.jupiter.api.extension.ExtensionContext.Store;
import org.junit.jupiter.api.extension.ParameterContext;
import org.junit.jupiter.api.extension.ParameterResolver;
import org.junit.jupiter.api.extension.TestInstancePostProcessor;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

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));
    String mockName = getMockName(parameter);

    if (mockName != null) {
        return mocks.getOrComputeIfAbsent(mockName,
            key -> mock(mockType, mockName));
    } else {
        return mocks.getOrComputeIfAbsent(mockType.getCanonicalName(),
            key -> mock(mockType));
    }
}

private String getMockName(Parameter parameter) {
    String explicitMockName = parameter.getAnnotation(Mock.class).name()
        .trim();
    if (!explicitMockName.isEmpty()) {
        return explicitMockName;
    } else if (parameter.isNamePresent()) {
        return parameter.getName();
    }
    return null;
}
```



4. Modelo de extensiones en Jupiter

- Extensión de Mockito para JUnit 5 (pruebas unitarias):

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.verifyZeroInteractions;
import static org.mockito.Mockito.when;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import io.github.bonigarcia.mockito.MockitoExtension;
```

```
@ExtendWith(MockitoExtension.class)
class LoginControllerLoginTest {
    // Mocking objects
    @InjectMocks
    LoginController loginController;
    @Mock
    LoginService loginService;
    // Test data
    UserForm userForm = new UserForm("foo", "bar");
```

T E S T S

```
Running io.github.bonigarcia.LoginControllerLoginTest
LoginController.login UserForm [username=foo, password=bar]
LoginController.login UserForm [username=foo, password=bar]
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.739 sec -
in io.github.bonigarcia.LoginControllerLoginTest
```

Results :

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

```
@Test
void testLoginOk() {
    // Setting expectations (stubbing methods)
    when(loginService.Login(userForm)).thenReturn(true);
    // Exercise SUT
    String responseLogin = loginController.login(userForm);
    // Verification
    assertEquals("OK", responseLogin);
    verify(loginService).Login(userForm);
    verifyNoMoreInteractions(loginService);
}
```

```
@Test
void testLoginKo() {
    // Setting expectations (stubbing methods)
    when(loginService.Login(userForm)).thenReturn(false);

    // Exercise SUT
    String responseLogin = loginController.login(userForm);

    // Verification
    assertEquals("KO", responseLogin);
    verify(loginService).Login(userForm);
    verifyZeroInteractions(loginService);
}
```

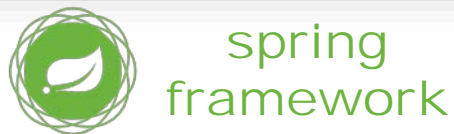
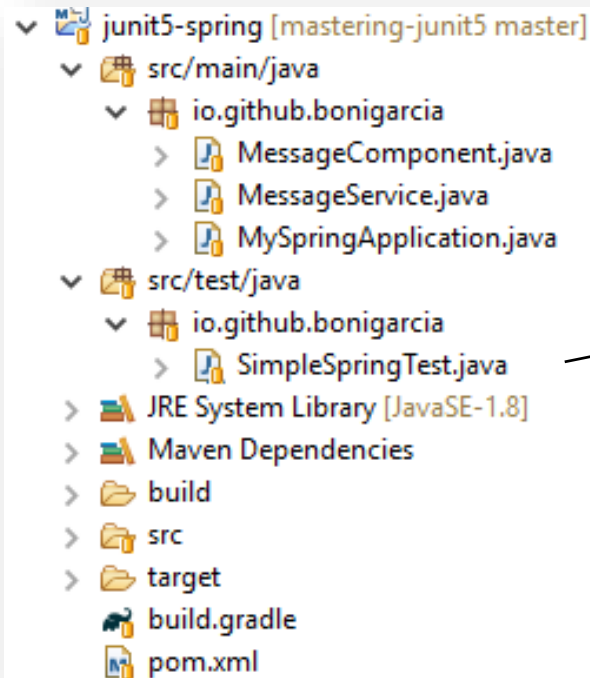
Fork me on GitHub



4. Modelo de extensiones en Jupiter

- Extensión de **Spring** para JUnit 5 (pruebas de integración):

Fork me on GitHub



```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = { MySpringApplication.class })
class SimpleSpringTest {

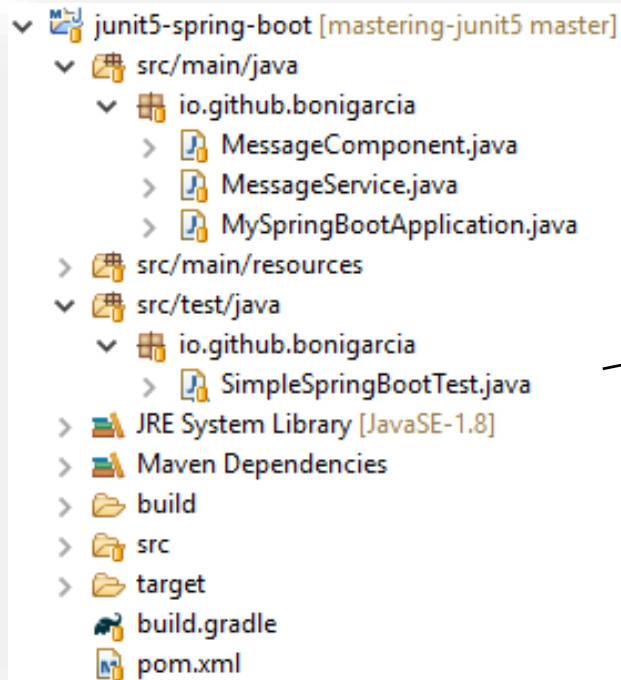
    @Autowired
    public MessageComponent messageComponent;

    @Test
    public void test() {
        assertEquals("Hello world!", messageComponent.getMessage());
    }
}
```

4. Modelo de extensiones en Jupiter

- Extensión de Spring para JUnit 5 (pruebas de integración):

Fork me on GitHub



```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@SpringBootTest
class SimpleSpringBootTest {
    @Autowired
    public MessageComponent messageComponent;

    @Test
    public void test() {
        assertEquals("Hello world!",
            messageComponent.getMessage());
    }
}
```



```
TESTS
Running io.github.bonigarcia.SimpleSpringBootTest

Spring
:: Spring Boot :: (v2.0.0.M3)
2017-08-02 00:39:47.904 INFO 14124 --- [main] i.g.bonigarcia.SimpleSpringBootTest : Starting SimpleSpringBootTest
on LAPTOP-T904060I with PID 14124 (started by boni in D:\dev\mastering-junit5\junit5-spring-boot)
2017-08-02 00:39:47.912 INFO 14124 --- [main] i.g.bonigarcia.SimpleSpringBootTest : No active profile set, falling
back to default profiles: default
2017-08-02 00:39:48.357 INFO 14124 --- [main] i.g.bonigarcia.MySpringBootApplication : *** Hello world! ***
2017-08-02 00:39:48.426 INFO 14124 --- [main] i.g.bonigarcia.SimpleSpringBootTest : Started SimpleSpringBootTest i
n 0.716 seconds (JVM running for 2.045)
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.395 sec - in io.github.bonigarcia.SimpleSpringBootTest
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

4. Modelo de extensiones en Jupiter

- Extensión de Selenium para JUnit 5 (pruebas *end-to-end*):

```
package io.github.bonigarcia;

import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

@ExtendWith(SeleniumExtension.class)
public class LocalWebDriverTest {

    @Test
    public void testWithChrome(ChromeDriver chrome) {
        chrome.get("https://bonigarcia.github.io/selenium-jupiter/");

        assertTrue(chrome.getTitle().startsWith("selenium-jupiter"));
    }

    @Test
    public void testWithFirefox(FirefoxDriver firefox) {
        firefox.get("http://www.seleniumhq.org/");

        assertTrue(firefox.getTitle().startsWith("Selenium"));
    }
}
```

Fork me on GitHub

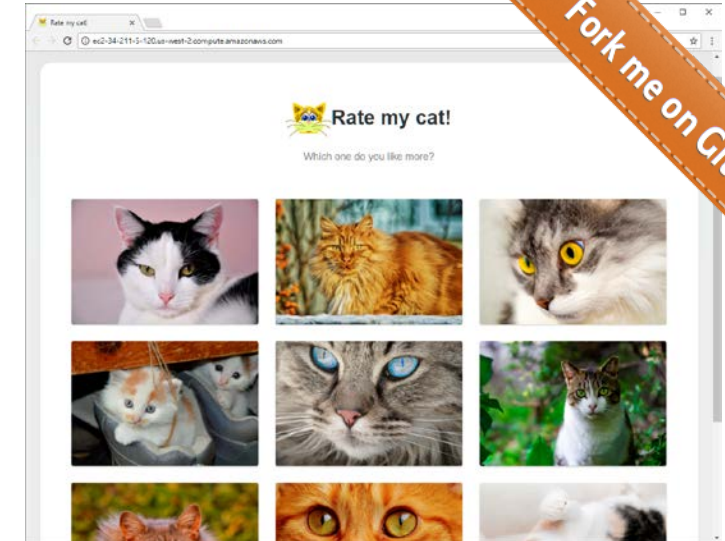


<https://bonigarcia.github.io/selenium-jupiter/>

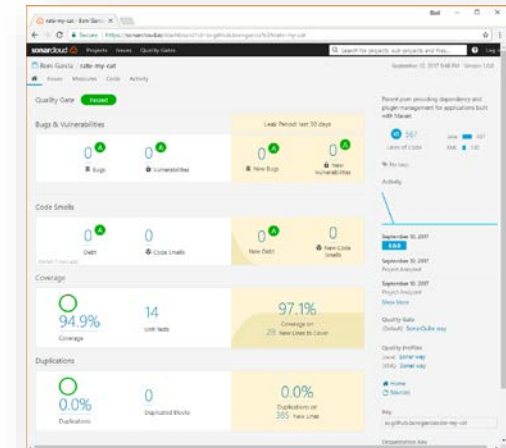
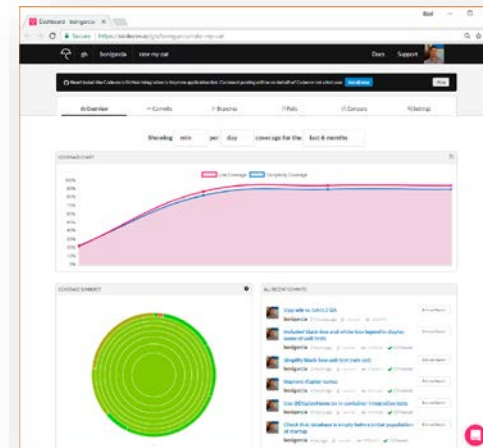
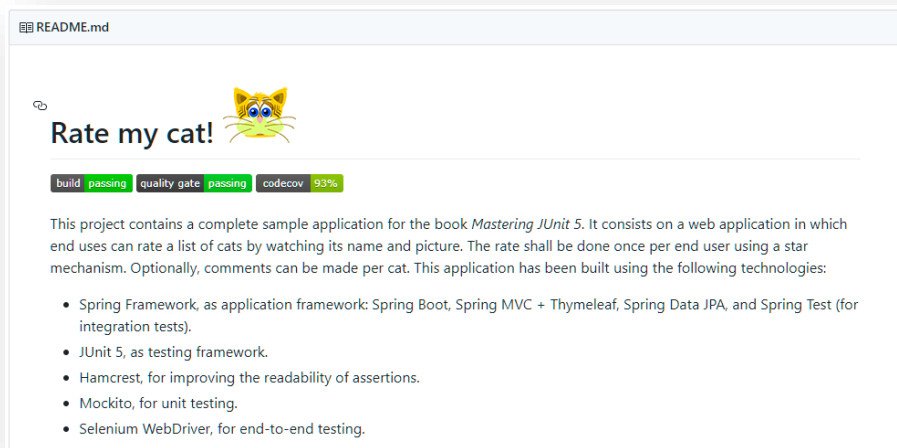
4. Modelo de extensiones en Jupiter

- Ejemplo completo:

- Aplicación web implementada con Spring-Boot
- Pruebas unitarias con Mockito
- Pruebas de integración con Spring
- Pruebas de sistema (e2e) con Selenium
- Ejecución de pruebas mediante Travis CI
- Análisis de código mediante SonarCloud
- Análisis de cobertura mediante Codedov



<https://github.com/bonigarcia/rate-my-cat>



Contenidos

1. Introducción a las pruebas en el software
2. Motivación y arquitectura de JUnit 5
3. Jupiter: el nuevo modelo de programación de JUnit 5
4. Modelo de extensiones en Jupiter
- 5. Conclusiones**

5. Conclusiones

- JUnit 5 ha supuesto el rediseño completo del framework JUnit
- Es modular y está formado por Jupiter, Vintage, Platform
- Ofrece tres tipos de APIs: Test API, Test Engine SPI, Test Launcher API
- El modelo de programación en JUnit 5 (Jupiter) presenta muchas novedades con respecto a JUnit 4
- Jupiter puede crecer mediante un modelo de extensiones basado en diferentes puntos de extensión: ciclo de vida, ejecución condicional, inyección de dependencias, plantillas de test
- La versión 5.1 de JUnit 5 está actualmente en desarrollo. En esta versión se implementarán los escenarios de test (organización de unidades de test), se mejorarán los test dinámicos, etc.

5. Conclusiones

- Algunas *features* de JUnit 5 que no hemos visto en esta charla:
 - Test anidados (`@Nested`)
 - Test dinámicos (`@TestFactory`)
 - Soporte de reglas de JUnit 4 en JUnit 5
 - Uso de interfaces y métodos por defecto para tests
 - Compatibilidad de JUnit 5 y Java 9
- Tampoco hemos visto la integración de JUnit 5 con:
 - Cucumber (hay una extensión propuesta y quieren hacer un *test engine*)
 - Docker (hay una extensión que permite ejecutar contenedores Dockers desde casos de prueba JUnit 5)
 - Android (plugin Gradle para ejecutar tests JUnit 5)
 - Servicios REST (con Spring, REST Assured, o WireMock)
- Otros: integración continua, ciclo de vida del desarrollo, requisitos...

Introducción y novedades de JUnit 5

Muchas gracias

Boni García



boni.garcia@urjc.es



<http://bonigarcia.github.io/>



[@boni_gg](https://twitter.com/boni_gg)



<https://github.com/bonigarcia>