JUnit5 in a nutshell

10 Feb 2019

https://goo.gl/rTGmk5

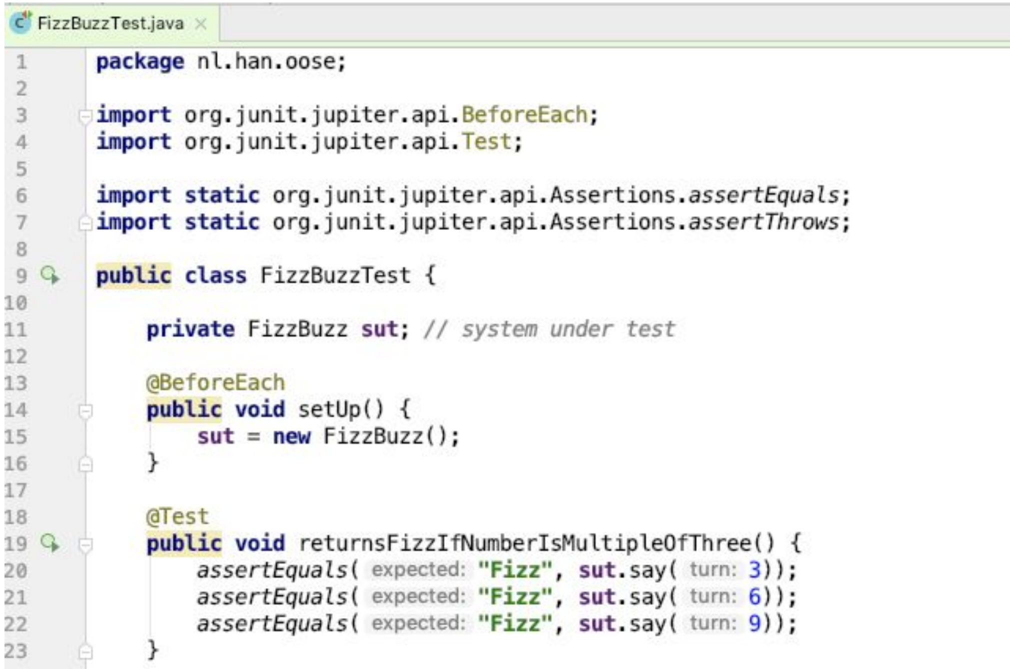
Hogeschool van Arnhem en Nijmegen
HAN University of Applied Sciences

What you will learn

- What is JUnit
- Why do we automate test
  - Verify specifications
  - Support modifications
  - Test-driven code development
- What types of tests exist
- Which types of tests can be automated with JUnit
- What is the standard structure of tests
  - Arrange - Act - Assert
- How to write unit tests
  - Test doubles, mocks and stubs

What is JUnit (https://junit.org/junit5)

- A **framework** supporting developers to write **automated tests** in Java and other JVM-based languages
- The de-facto testing standard in the Java-world
- Supported by
  - all major IDEs
    - IntelliJ
    - Eclipse
    - ..
  - all major build systems
    - Maven
    - Gradle
    - ..


FizzBuzzTest.java

```java
package nl.han.oose;

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

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

public class FizzBuzzTest {

    private FizzBuzz sut; // system under test

    @BeforeEach
    public void setUp() {
        sut = new FizzBuzz();
    }

    @Test
    public void returnsFizzIfNumberIsMultipleOfThree() {
        assertEquals( expected: "Fizz", sut.say( turn: 3));
        assertEquals( expected: "Fizz", sut.say( turn: 6));
        assertEquals( expected: "Fizz", sut.say( turn: 9));
    }
```

Why do we automate tests?

- Because we can 😉
- To (repeatedly) verify that the code meets expectations
  - when it was just finished and
  - after modifications or additions (aka **regression testing**)
- A **regression** is a new error introduced by a modification, i.e. something that is broken now, but worked before
- In **test-driven development** (TDD), tests are used as **executable specifications** written before the actual code (i.e. **production code**) is written
- Because tests call production code in the expected way, they also **serve as documentation**, or as example in use

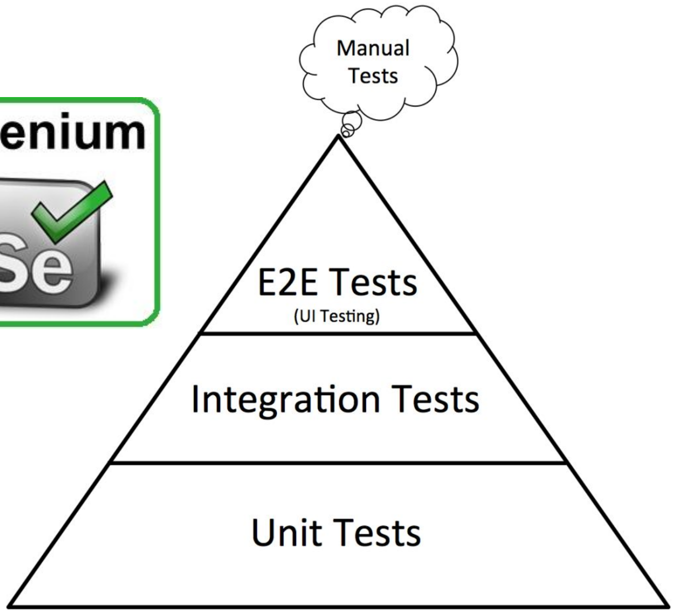
What types of tests exist

- As **developers**, we are at least responsible for:
  - **Unit tests**: make sure that a small unit of code (typically methods) works as expected. **Executed very often**, e.g. after each significant change. Have a **narrow scope**, called external code needs to be isolated so it is not implicitly tested along; need to be quick.
  - **Integration tests**: focus on the proper **integration of different modules** (e.g. classes), including code over which developers have no control. This **usually requires some resources** (e.g. database, filesystem) and because of this the tests run more slowly.
  - **End-to-End tests**: verify that your code works **from the client's point of view** and put **the system as a whole** to the test, mimicking the way the user would use it.

Source: Kaczanowski, T. (2013). *Practical Unit Testing with JUnit and Mockito*. Tomasz Kaczanowski.

What types of tests can be automated with JUnit

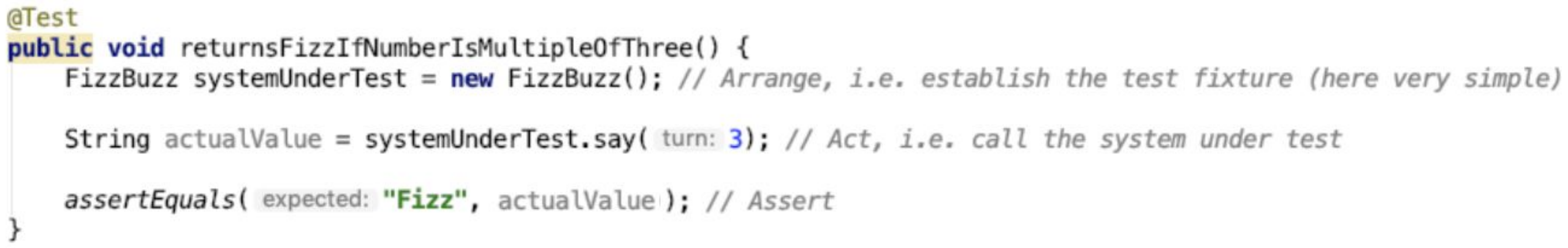
- JUnit5 is basically a **test-automation platform** that supports different kinds of tests
- All three types of developer tests are either natively supported or covered by popular JUnit5 extensions.



What is the standard structure of tests (1/2)

- First, become aware of the **test scope** and decide on the **test goal**, i.e. what you are going to test in the **testcase**
- Tests often use the following pattern
  - Testname resembles the test goal
  - In the test-body
    - Arrange the **test fixture**; i.e. create and configure objects so the test runs in the desired context
    - Act, i.e. call the production-code to be tested (e.g. a method)
    - **Assert**, i.e. check if the actual effects of the call are as expected; usually multiple asserts are used in one testcase
- Typically, we define multiple testcases to cover a single unit; one test-method per test-goal

What is the standard structure of tests (2/2)



```
@Test
public void returnsFizzIfNumberIsMultipleOfThree() {
    FizzBuzz systemUnderTest = new FizzBuzz(); // Arrange, i.e. establish the test fixture (here very simple)

    String actualValue = systemUnderTest.say( turn: 3); // Act, i.e. call the system under test

    assertEquals( expected: "Fizz", actualValue ); // Assert
}
```

Test fixture

- **Test-fixture**: Something used to **consistently test** some item, device, or piece of software; repeated tests need to give the same results
- In JUnit: A configuration of one or more objects required to test the behavior of the SUT in a specific situation/context.

Example:

test-fixture for mobile phone displays

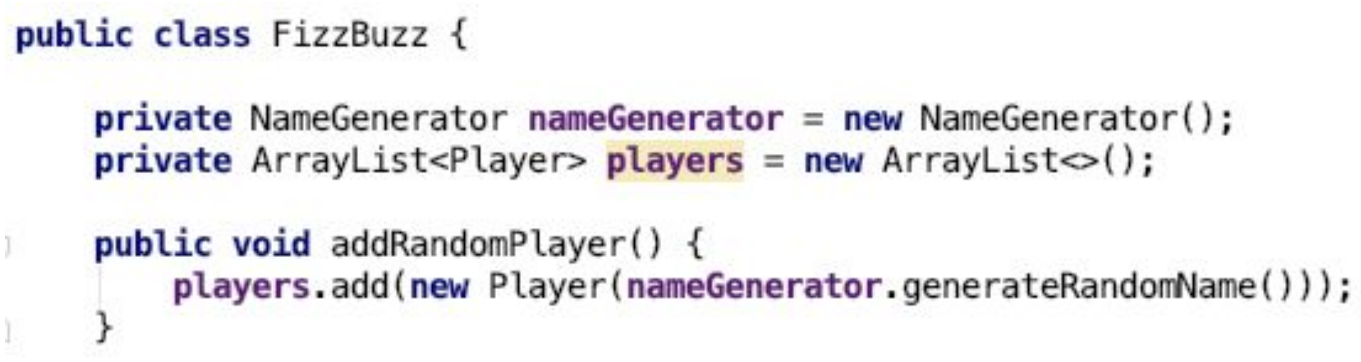
JUnit5 assertions

- assert**Equals**(...) / assertNotEquals(...)
- assert**True**(boolean) / assertFalse(boolean)
- assert**Throws**(...)
- assert**Timeout**(...)
- assert**Null**(Object) / assertNotNull(Object)
- assert**Same**(...) / assertNotSame(...)
- assert**ArrayEquals**(...)
- assert**IterableEquals**(...)
- assert**LinesMatch**(...)
- assert**All**(...)

See: https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-assertions-with-junit-5-api/

How to write unit tests

- In unit tests we need to make sure that we only call code that is inside our testscope. Consider the following example:

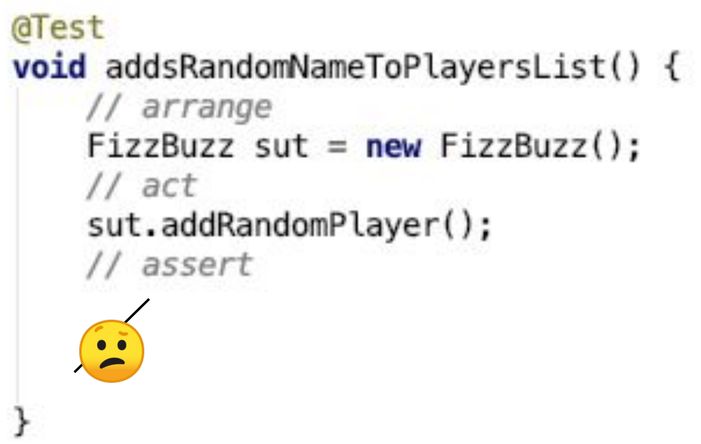


```
public class FizzBuzz {

    private NameGenerator nameGenerator = new NameGenerator();
    private ArrayList<Player> players = new ArrayList<>();

    public void addRandomPlayer() {
        players.add(new Player(nameGenerator.generateRandomName()));
    }
```

Test-scope

- **Test-goal**: Verify that the method addRandomPlayer creates a player object with the string returned by nameGenerator and adds it to the players list.
- The **collaborator** NameGenerator is out-of-scope and **must not** be called in the test.

How to write unit tests



```java
@Test
void addsRandomNameToPlayersList() {
    // arrange
    FizzBuzz sut = new FizzBuzz();
    // act
    sut.addRandomPlayer();
    // assert

}
```

Problems:

- How to observe the expected effect of addRandomPlayer() ?
- The player name is random, so what to assert?
- How to test addRandomPlayer w/o implicitly testing the collaborator NameGenerator?

How to write unit tests

Solution:

- Create a fake NameGenerator that always returns the same String
- Make sure FizzBuzz uses this fake generator when being invoked from the test method
- Change the visibility of players to package-private so we can “see” it from the test.

How to write **unit** tests

Fake collaborator

(here a stub)

```java
@Test
void addsRandomNameToPlayersList() {
    // arrange
    NameGenerator nameGeneratorFake = new NameGenerator() {
        @Override
        public String generateRandomName() {
            return "Uwe";
        }
    };

    FizzBuzz sut = new FizzBuzz(nameGeneratorFake);

    // act
    sut.addRandomPlayer();

    // assert
    Player expectedPlayer = new Player( playerName: "Uwe");
    assertEquals(expectedPlayer, sut.players.get(0));
}
```

```java
public class FizzBuzz {

    ArrayList<Player> players = new ArrayList<>();

    private NameGenerator nameGenerator;

    public FizzBuzz(NameGenerator nameGenerator) {
        this.nameGenerator = nameGenerator;
    }

    public void addRandomPlayer() {
        players.add(new Player(nameGenerator.generateRandomName()));
    }
```

How to write unit tests

- In unit-tests, we focus on one element of the software at a time. To make a single unit work, we often need other units that are out of the test-scope.
- Therefore, we often make use of **test doubles**, which are pretend-objects used in place of a real object for testing purposes.
- Stubs and mocks are different types of test doubles:
  - **Stubs** provide pre-determined values when being called from a test, or no values at all; they are simply required to make the production code run
  - **Mocks** are special stubs which we use to verify the behavior of the sut; i.e. we verify the correct interaction of the sut with the mock

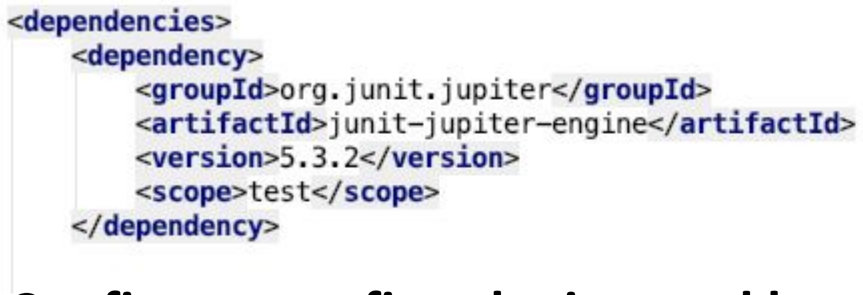
See: https://martinfowler.com/articles/mocksArentStubs.html

All new terms in one place

- test-framework
- regression, regression testing
- unit test, integration test, end-to-end test
- test scope, test goal, test case
- test fixture
- assertion
- system-under-test (sut)
- collaborator
- Test double, stub, mock

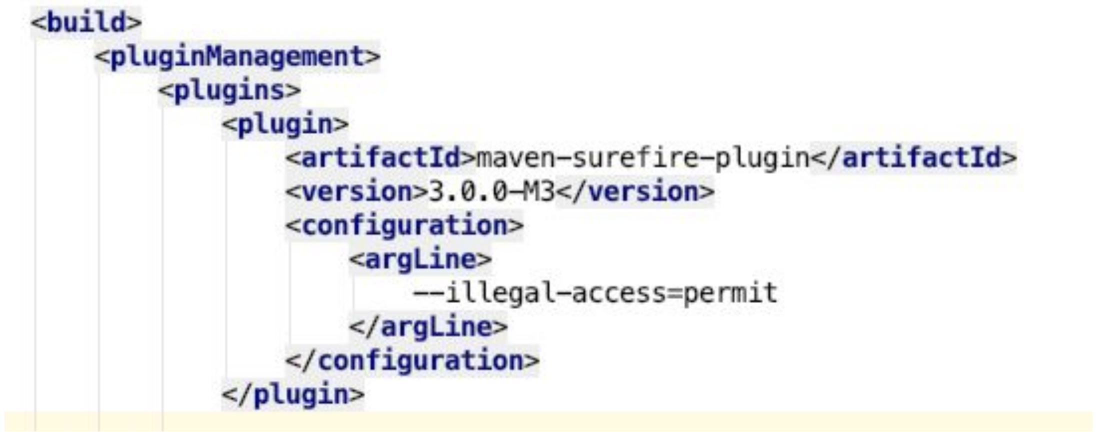
How to use JUnit5

Add maven dependency



```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.3.2</version>
        <scope>test</scope>
    </dependency>
```

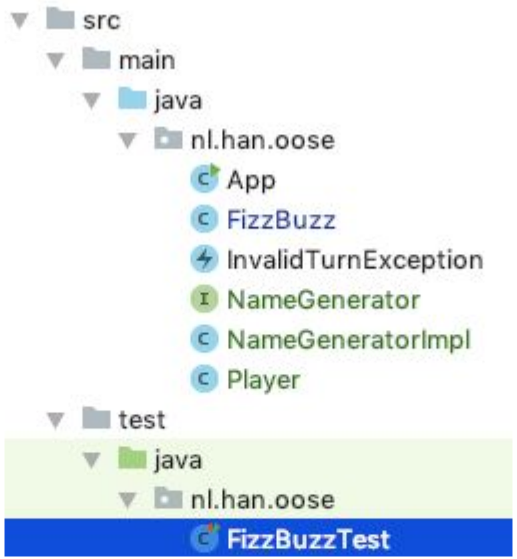
Configure surefire plugin, used by maven to run JUnit tests



```xml
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M3</version>
                <configuration>
                    <argLine>
                        --illegal-access=permit
                    </argLine>
                </configuration>
            </plugin>
```

How to use JUnit5

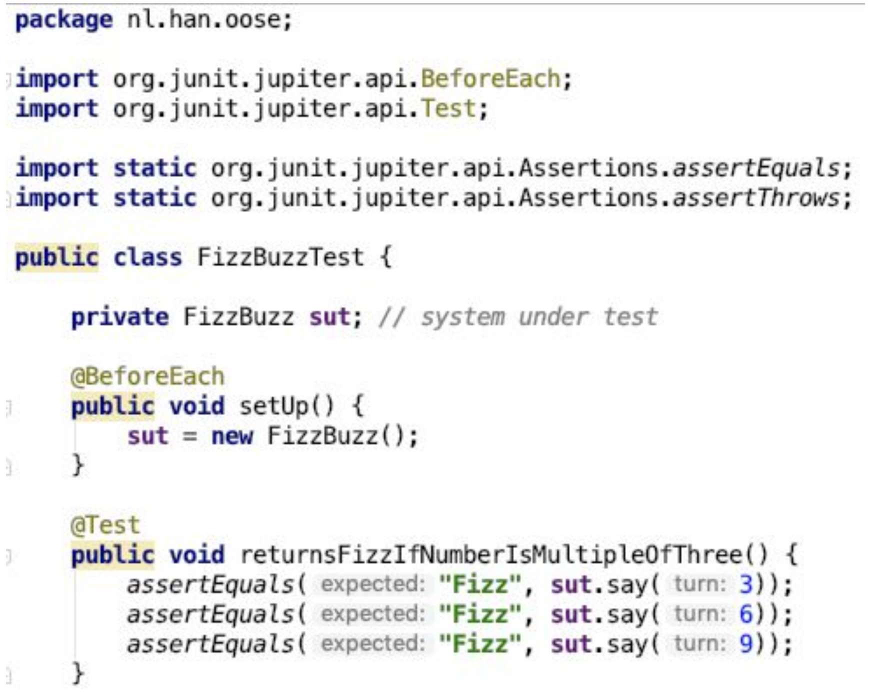
Create new class in test folder (src/test/java)



```
src
  main
    java
      nl.han.oose
        App
        FizzBuzz
        InvalidTurnException
        NameGenerator
        NameGeneratorImpl
        Player
  test
    java
      nl.han.oose
        FizzBuzzTest
```

Make sure the test class uses “Test” in its name and place it in the same Java package as the class under test

How to use JUnit5

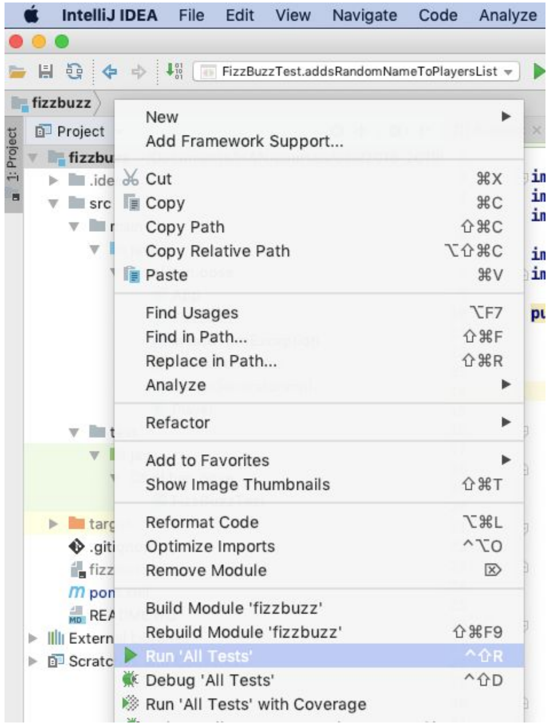


```
package nl.han.oose;

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

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

public class FizzBuzzTest {

    private FizzBuzz sut; // system under test

    @BeforeEach
    public void setUp() {
        sut = new FizzBuzz();
    }

    @Test
    public void returnsFizzIfNumberIsMultipleOfThree() {
        assertEquals( expected: "Fizz", sut.say( turn: 3));
        assertEquals( expected: "Fizz", sut.say( turn: 6));
        assertEquals( expected: "Fizz", sut.say( turn: 9));
    }
```

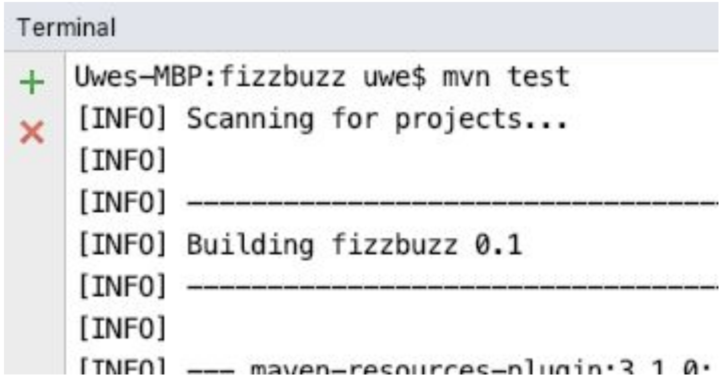
- use **@Test** for all test-cases
- **@BeforeEach** method is called right before each test-case
- **@AfterEach** method is called right after each test-case
- **@BeforeAll** is called once before the first test-case
- **@AfterAll** is called once after the last test-case

How to use JUnit5

Run with your favorite IDE

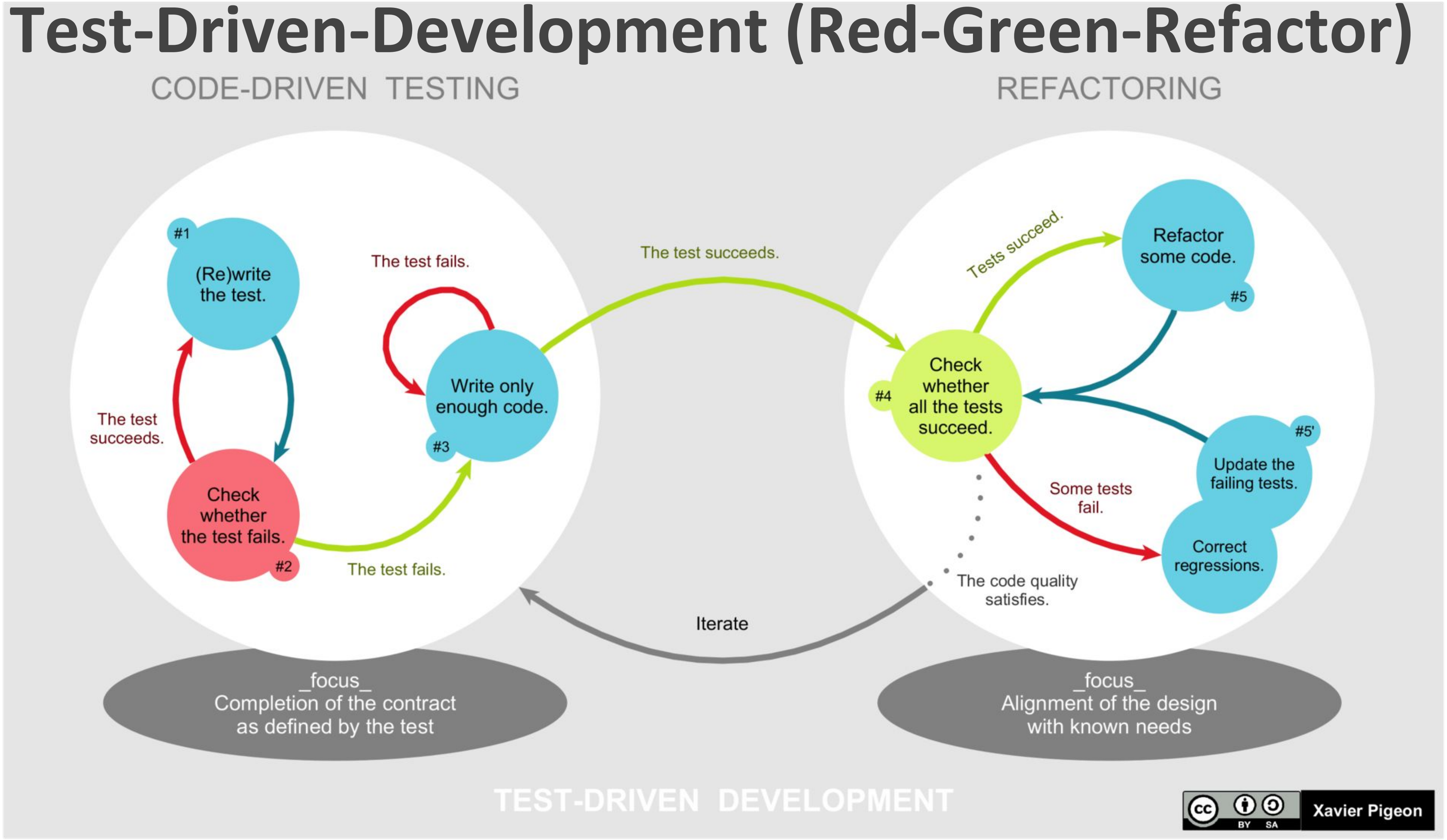


Run using Maven



```
Uwes-MBP:fizzbuzz uwe$ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------------------
[INFO] Building fizzbuzz 0.1
[INFO] -----------------------------------
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:
```

Test-Driven-Development (Red-Green-Refactor)



Source: https://twitter.com/XEngineer/status/666943798363090944