Unit Test

Instructor: Dr.Vahidi Asl

Mohammad-Moein Arabi

Table of contents

Naive Approach

- Print the program’s state to standard output!
- Check the output of each method one by one manually.
- Before releasing software, add Main class to test each function and then remove it! (Not reusing tests)

The problem with this way

- Low speed developing.
- Not cover all lines of code.
- Not reusing tests.
- The programmer must run tests manually.
- The programmer himself must ensure the test passes or not.

Benefits of Auto Testing

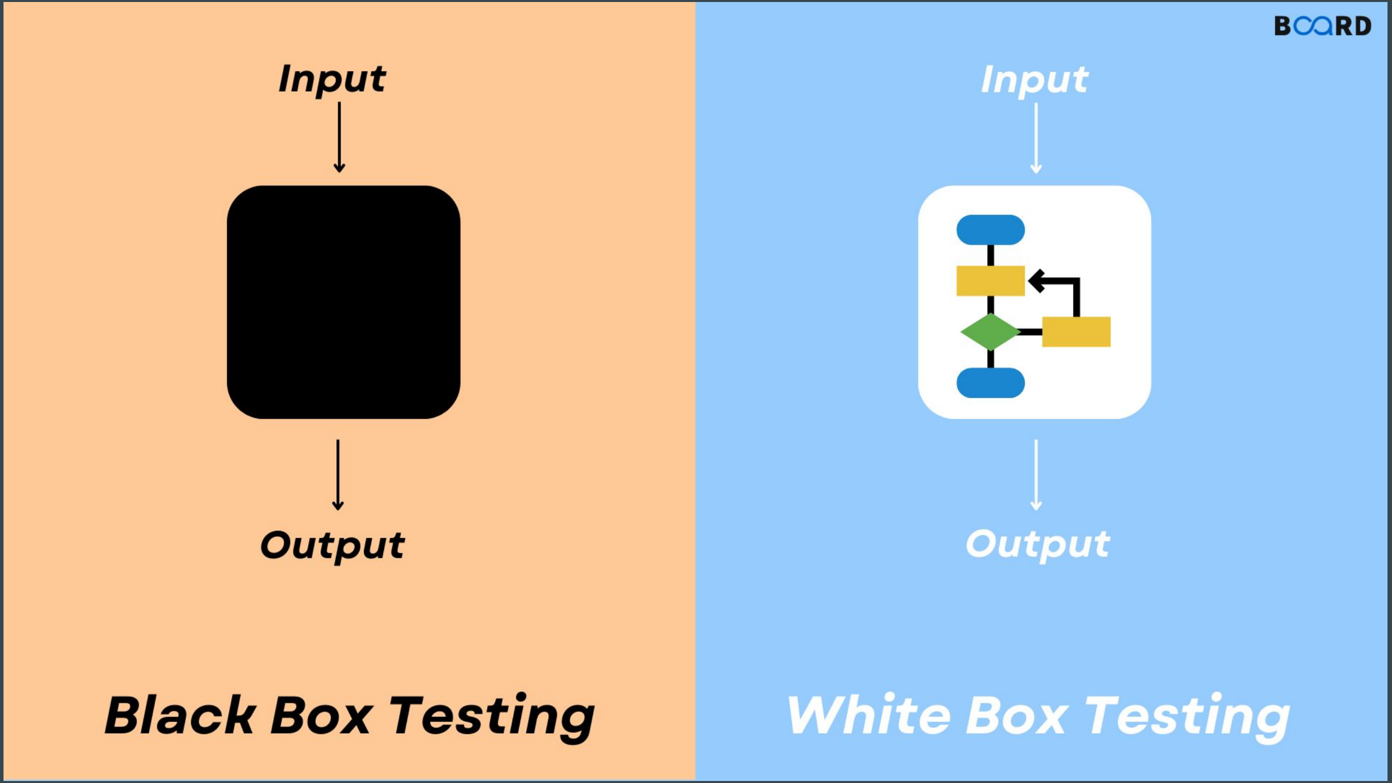
Let a program tests the program!

- High speed developing
- More accurate
- Debug easier after finding bugs
- Reuse tests
- Reports how many tests pass and fails

What is Unit Test?

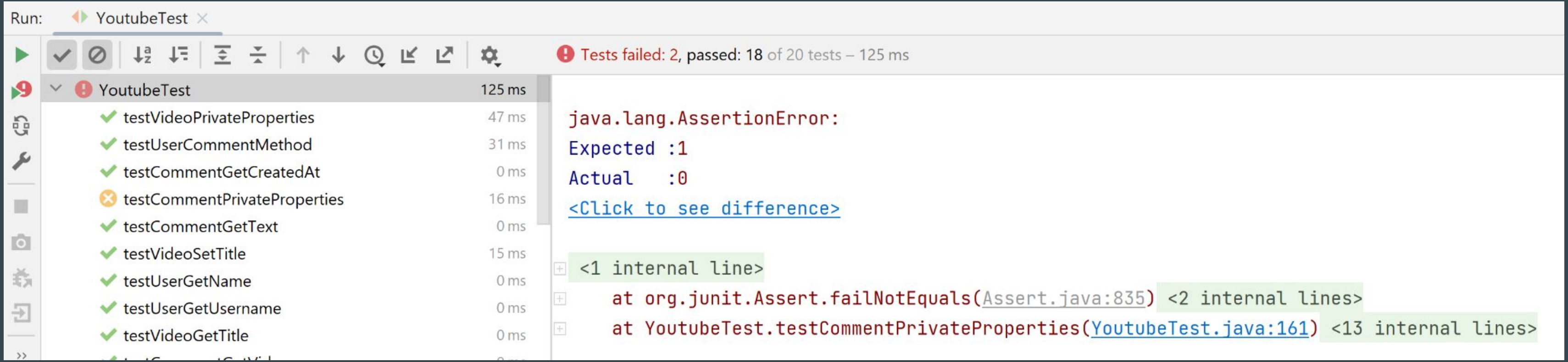
- An automated test code
- Written and maintained by the developer
- Tests a function or an **small** section
- The result of every test is **pass** or **fail**

Type of Testing



Editor Supportings

Intellij:



Tests failed: 2, passed: 18 of 20 tests – 125 ms

YoutubeTest	125 ms
testVideoPrivateProperties	47 ms
testUserCommentMethod	31 ms
testCommentGetCreatedAt	0 ms
testCommentPrivateProperties	16 ms
testCommentGetText	0 ms
testVideoSetTitle	15 ms
testUserGetName	0 ms
testUserGetUsername	0 ms
testVideoGetTitle	0 ms

```
java.lang.AssertionError:
Expected :1
Actual   :0
<Click to see difference>

<1 internal line>
    at org.junit.Assert.failNotEquals(Assert.java:835) <2 internal lines>
    at YoutubeTest.testCommentPrivateProperties(YoutubeTest.java:161) <13 internal lines>
```

Write your first unit tests

Unit Test in Java

JUnit framework

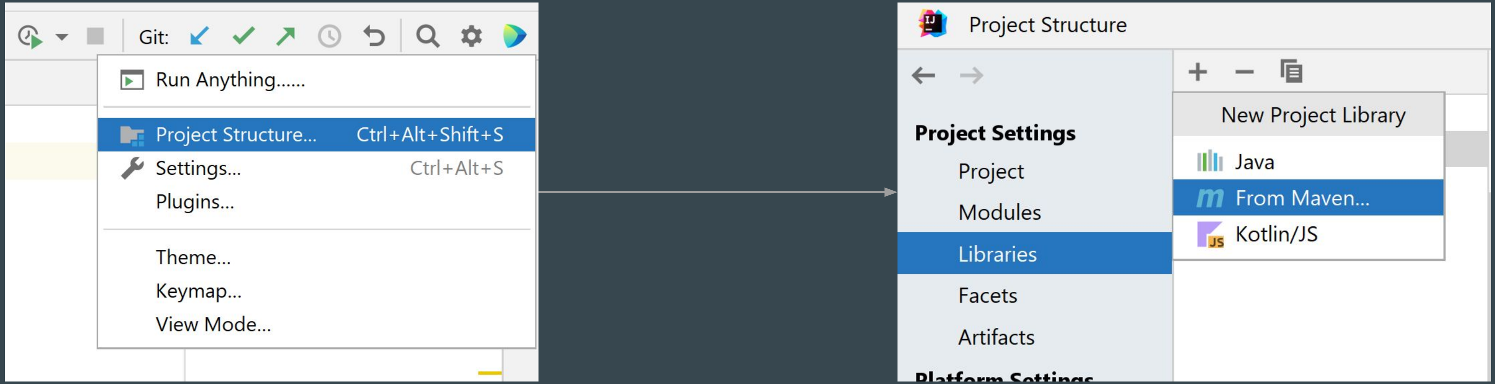
Not officially packed with JVM. Should be installed as a dependency.

JUnit4 last release was v4.13.2 on Feb 2021

JUnit5 is the last major update.

Download Junit

1. The official GitHub page
2. Dependency management in Intellij



Class Calculator

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
        return a * b;
    }
}
```

1. Create `CalculatorTest` class

```java
public class CalculatorTest {







}
```

1. Create `CalculatorTest` class
2. Add a new method to test add method

```java
public class CalculatorTest {

    public void testAdditionMethod() {



    }
}
```

1. Create `CalculatorTest` class
2. Add a new method to test add method
3. Invoke add method in the test method assert the result

```java
import static org.junit.Assert.assertEquals;


public class CalculatorTest {


    public void testAdditionMethod() {

        Calculator calculator = new Calculator();

        int result = calculator.add(1, 1);

        assertEquals(2, result);

    }

}
```

1. Create `CalculatorTest` class
2. Add a new method to test add method
3. Invoke add method in the test method assert the result
4. Add `Test` annotation to the test method

```java
import org.junit.Test;

import static org.junit.Assert.assertEquals;


public class CalculatorTest {

    @Test

    public void testAdditionMethod() {

        Calculator calculator = new Calculator();

        int result = calculator.add(1, 1);

        assertEquals(2, result);

    }

}
```

1. Create `CalculatorTest` class
2. Add a new method to test add method
3. Invoke add method in the test method assert the result
4. Add `Test` annotation to the test method

Junit will only run methods with Test annotation.

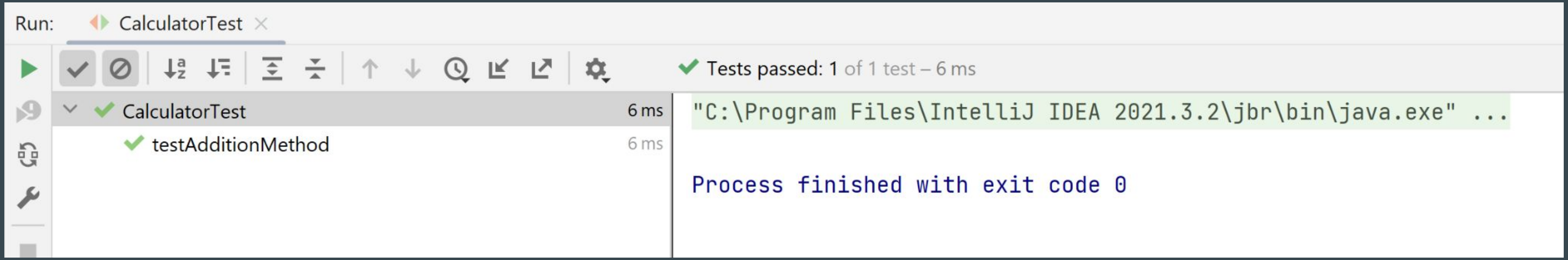
How JUnit knows which method has Test annotation? ***Reflection***

```java
import org.junit.Test;

import static org.junit.Assert.assertEquals;


public class CalculatorTest {

    @Test

    public void testAdditionMethod() {

        Calculator calculator = new Calculator();

        int result = calculator.add(1, 1);

        assertEquals(2, result);

    }

}
```

```java
class Calculator {

    public int add(int a, int b) {

        return a + b;

    }


    public int subtract(int a, int b) {

        return a - b;

    }


    public int multiply(int a, int b) {

        return a * b;

    }

}
```

```java
import org.junit.Test;

import static org.junit.Assert.assertEquals;



public class CalculatorTest {

    @Test

    public void testAdditionMethod() {

        Calculator calculator = new Calculator();

        int result = calculator.add(1, 1);

        assertEquals(2, result);

    }

}
```

5. Run tests...



Tests passed: 1 of 1 test – 6 ms

```
CalculatorTest         6 ms
  testAdditionMethod   6 ms
```

```
"C:\Program Files\IntelliJ IDEA 2021.3.2\jbr\bin\java.exe" ...

Process finished with exit code 0
```

Good Test

The properties of a good test:

1. Use meaningful names for test methods.
2. Each test method should test only one thing (Single Responsibility)
3. Tests should be independent of another test.
4. Use meaningful messages for fail reason.

Advanced JUnit

Assert methods

assertThrows	`assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));`

assertTimeout	`assertTimeout(ofMinutes(2), () -> {`
	`    // Perform task that takes less than 2 minutes.`
	`});`

assertTrue	`assertTrue(5 > 3);`
assertFalse	
assertNotNull	`assertNotNull(getPersonName());`

Startup & Teardown Method

Before and after each test, these

methods will be executed.

```
public class BeforeAndAfterAnnotationsUnitTest {

    private List<String> list;

    @BeforeEach

    public void init() {

        System.out.println("startup");

        list = new ArrayList<>(Arrays.asList("test1", "test2"));

    }

    @AfterEach

    public void teardown() {

        System.out.println("teardown");

        list.clear();

    }

}
```

Startup & Teardown Method

Executes only **once**. Before any tests

and after all tests.

```
class BeforeAllAndAfterAllAnnotationsUnitTest {

    @BeforeAll

    static void setup() {

        System.out.println("startup - creating DB connection");

    }



    @AfterAll

    static void tearDown() {

        System.out.println("closing DB connection");

    }

}
```

Compare JUnit v4 vs v5

JUnit 4	`@Before`	`@After`	`@BeforeClass`	`@AfterClass`
JUnit 5	`@BeforeEach`	`@AfterEach`	`@BeforeAll`	`@AfterAll`

Other Annotations

`@DisplayName`	Declares a custom display name for the test class or test method.
`@Disabled`	Used to disable a test class or test method
`@Timeout`	lifecycle method if its execution exceeds a given duration.

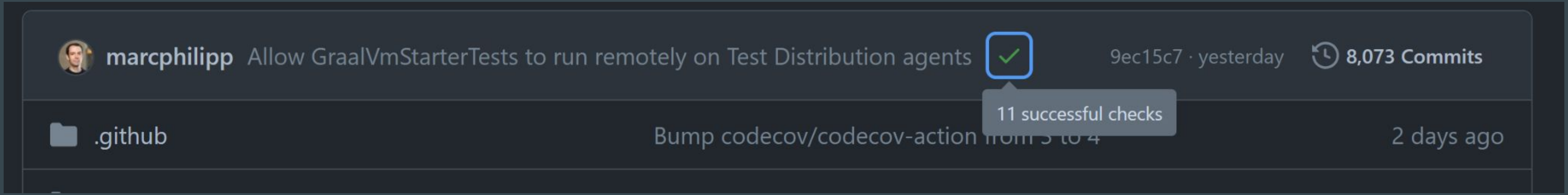
Additional Topics

Test in GitHub

A **GitHub workflow** is a configurable, automated process that executes one or more actions after each event (e.g. after pushing, after merging, ...)

GitHub can run our repository tests on its virtual machines!

Shows passing or failing result to users.



marcphilipp Allow GraalVmStarterTests to run remotely on Test Distribution agents ✓ 9ec15c7 · yesterday 8,073 Commits

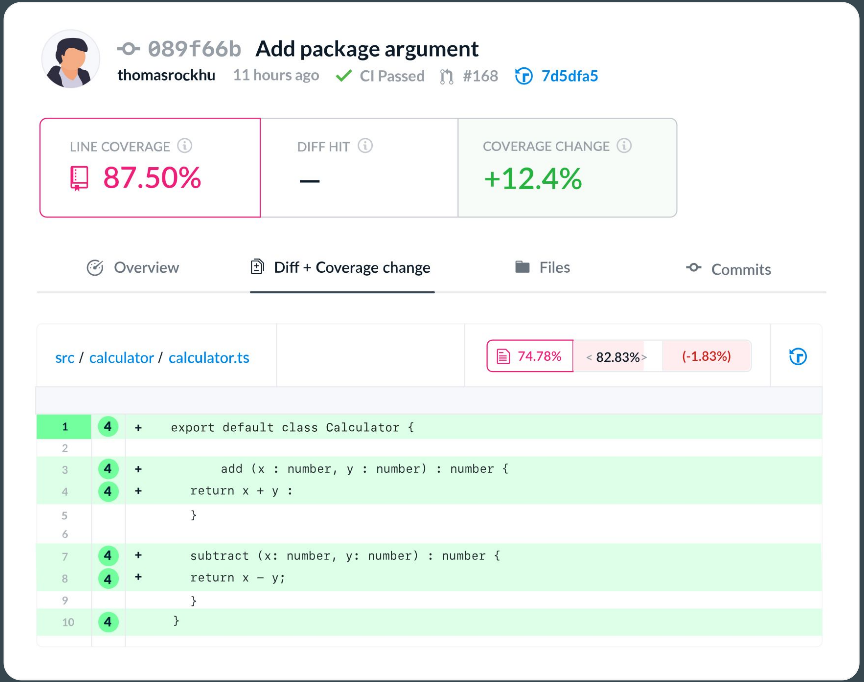
11 successful checks

.github Bump codecov/codecov-action from 3 to 4 2 days ago

Code Coverage

measures how many of your lines of code are executed when you run automated tests.

Many tools are exists. One of them is Codecov



089f66b Add package argument
thomasrockhu 11 hours ago ✓ CI Passed #168 7d5dfa5

LINE COVERAGE	DIFF HIT	COVERAGE CHANGE
87.50%	—	+12.4%

src / calculator / calculator.ts 74.78% < 82.83% > (-1.83%)

```
4 +  export default class Calculator {

4 +        add (x : number, y : number) : number {
4 +    return x + y :
       }

4 +    subtract (x: number, y: number) : number {
4 +    return x - y;
       }
4    }
```