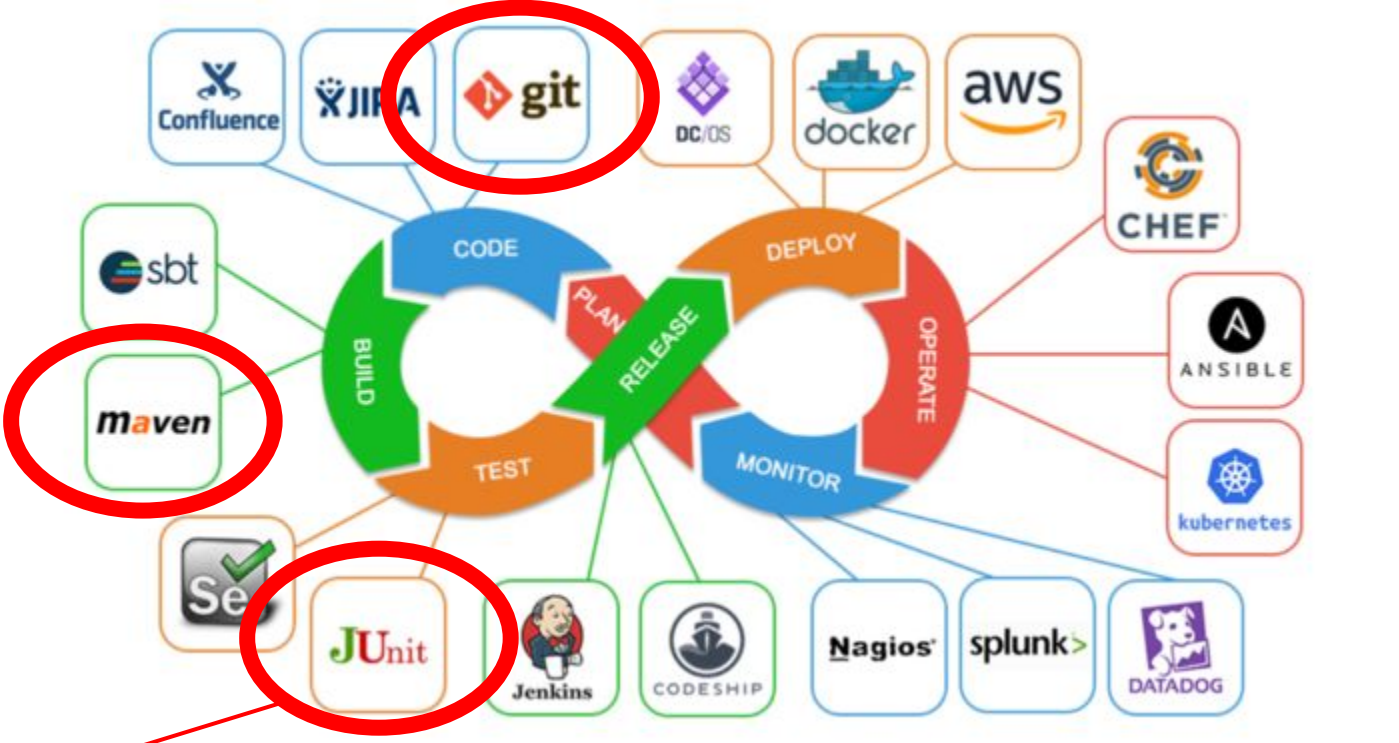# Software Project Management - Laboratory

Lecture n° 7
A.Y. 2021-2022

Prof. Fabrizio Fornari

DevOps

Our Focus

What is Testing?

Testing

Testing is the activity of finding out whether a piece of code (a method, class, or program) produces the intended behavior.
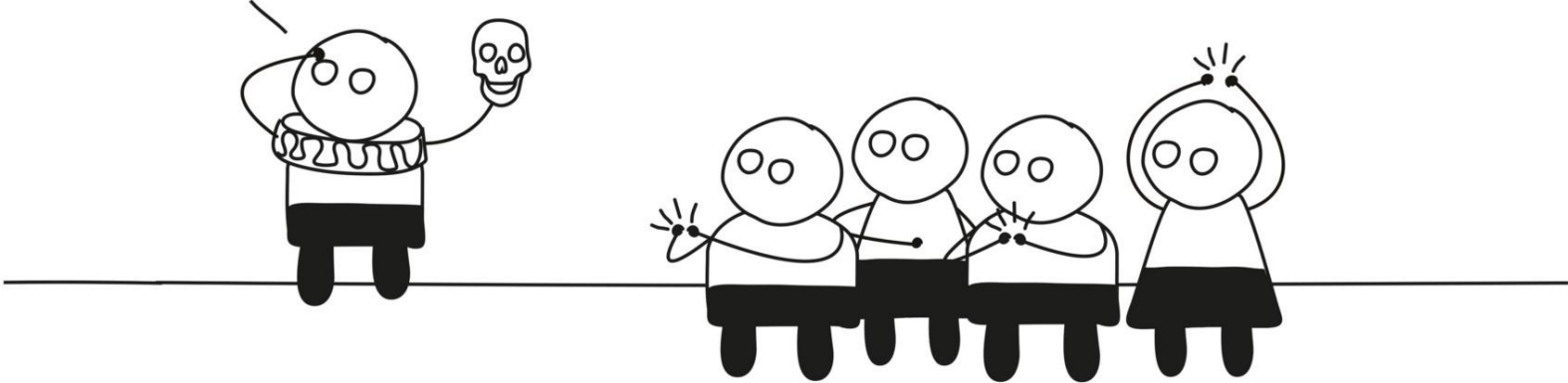
Program testing can be used to show the presence of bugs, but never to show their absence!
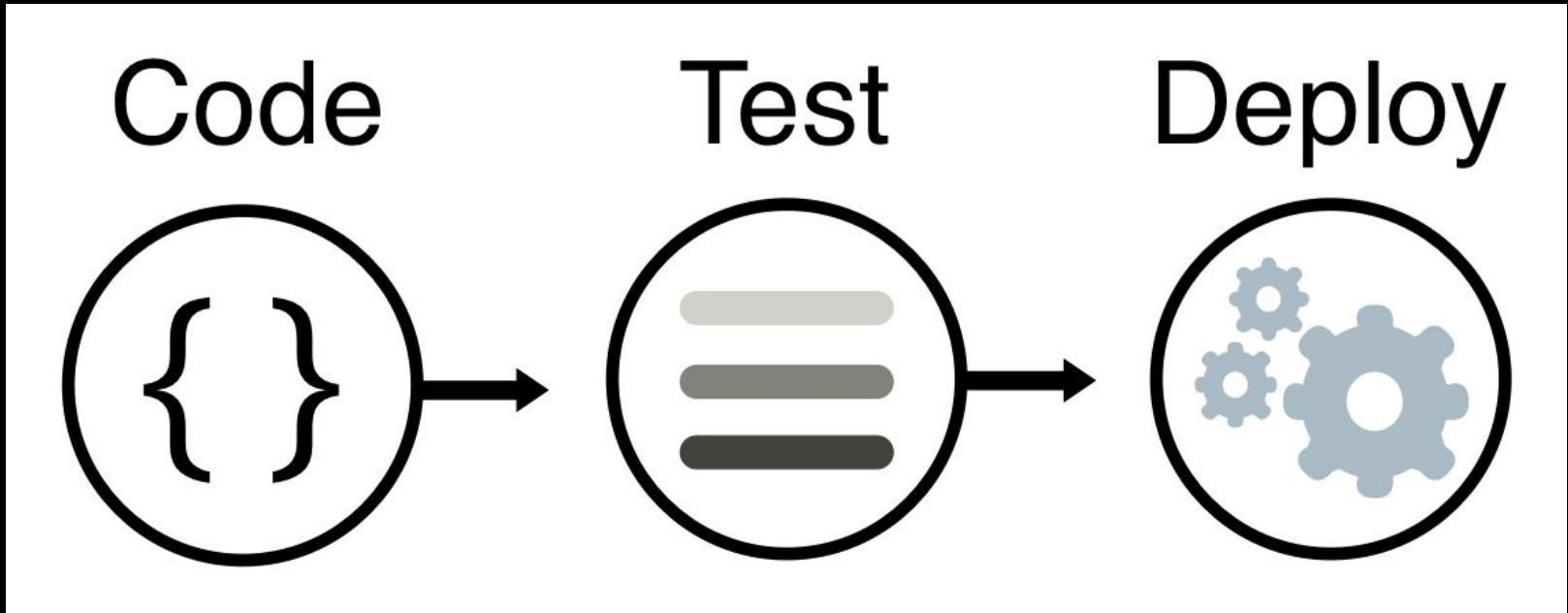
— *Edsger Dijkstra* —

Have you ever done Testing?

How many of you? Raise your hands!!!

Testing vs Debugging

What is the difference?

Testing

The purpose of testing is
to find bugs and errors.

Debugging

Testing

The purpose of testing is to find bugs and errors.

Debugging

The purpose of debugging is to correct those bugs found during testing.

Debugging Untested Code

Debugging
Untested Code

The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

Anger

*#@$!

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.
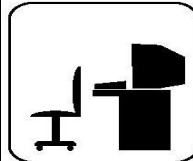
Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake, League of Legends, Fortnite, COD, or whatever you play :D

Testing...when?
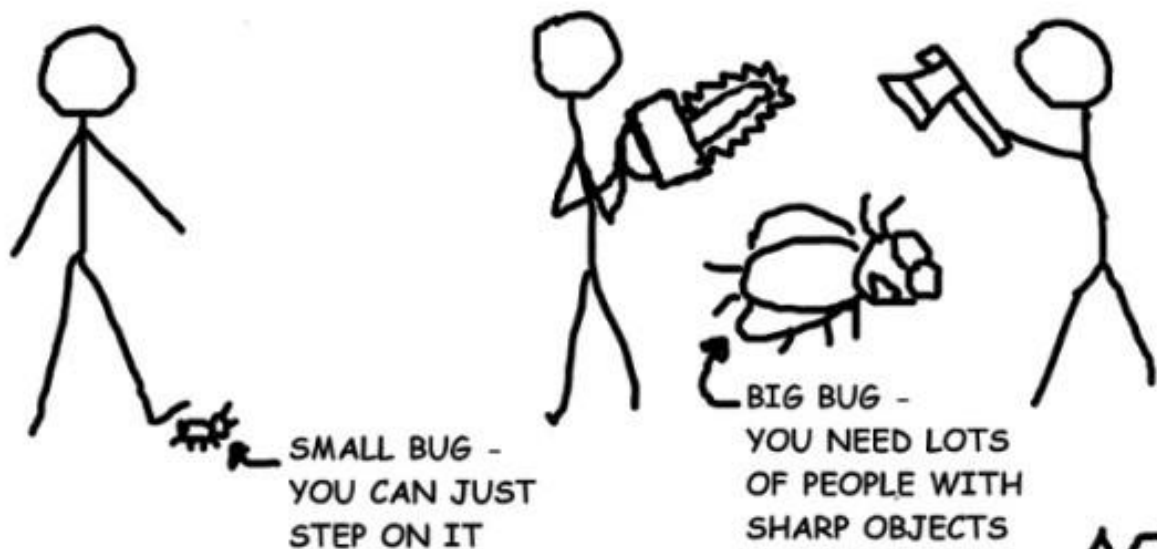
The Continuous Delivery Pipeline
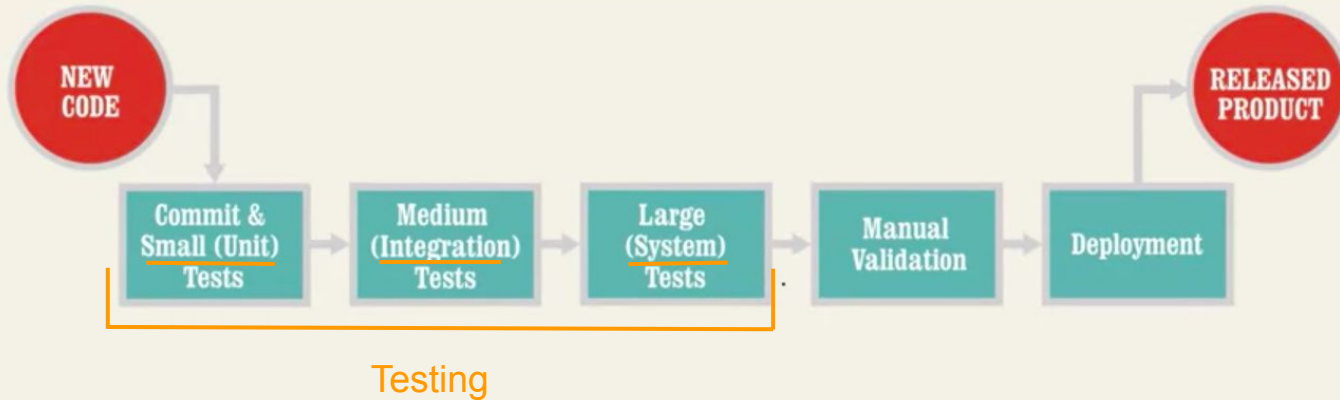
Test Sizes

Size & Time →

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

Google Testing Blog 'Test Sizes'

The Test Stack

System / Large

Integration / Medium
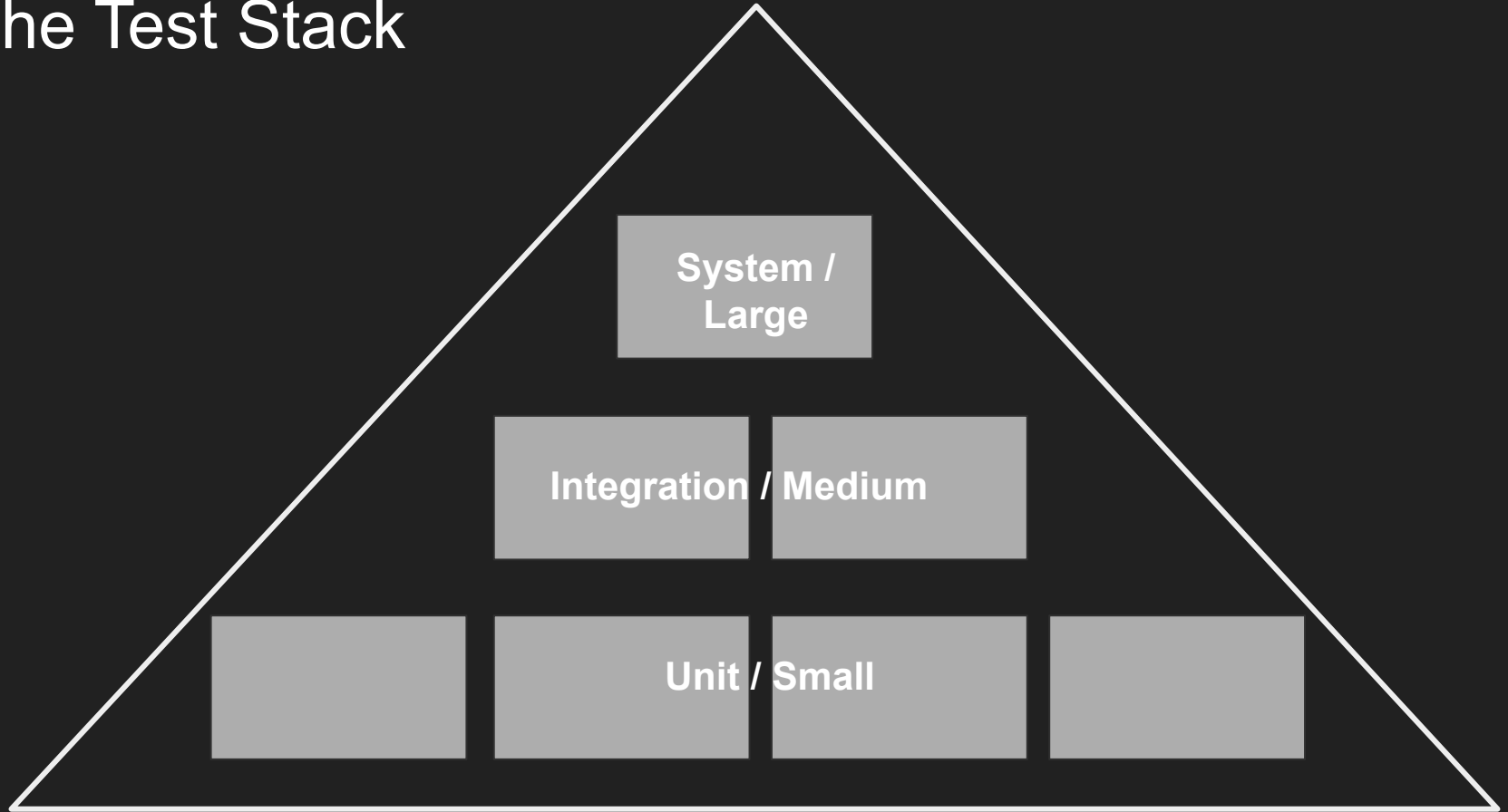
Unit / Small

The Test Stack

Unit testing on individual units of source code (smallest testable part).

System / Large

Integration / **Medium**
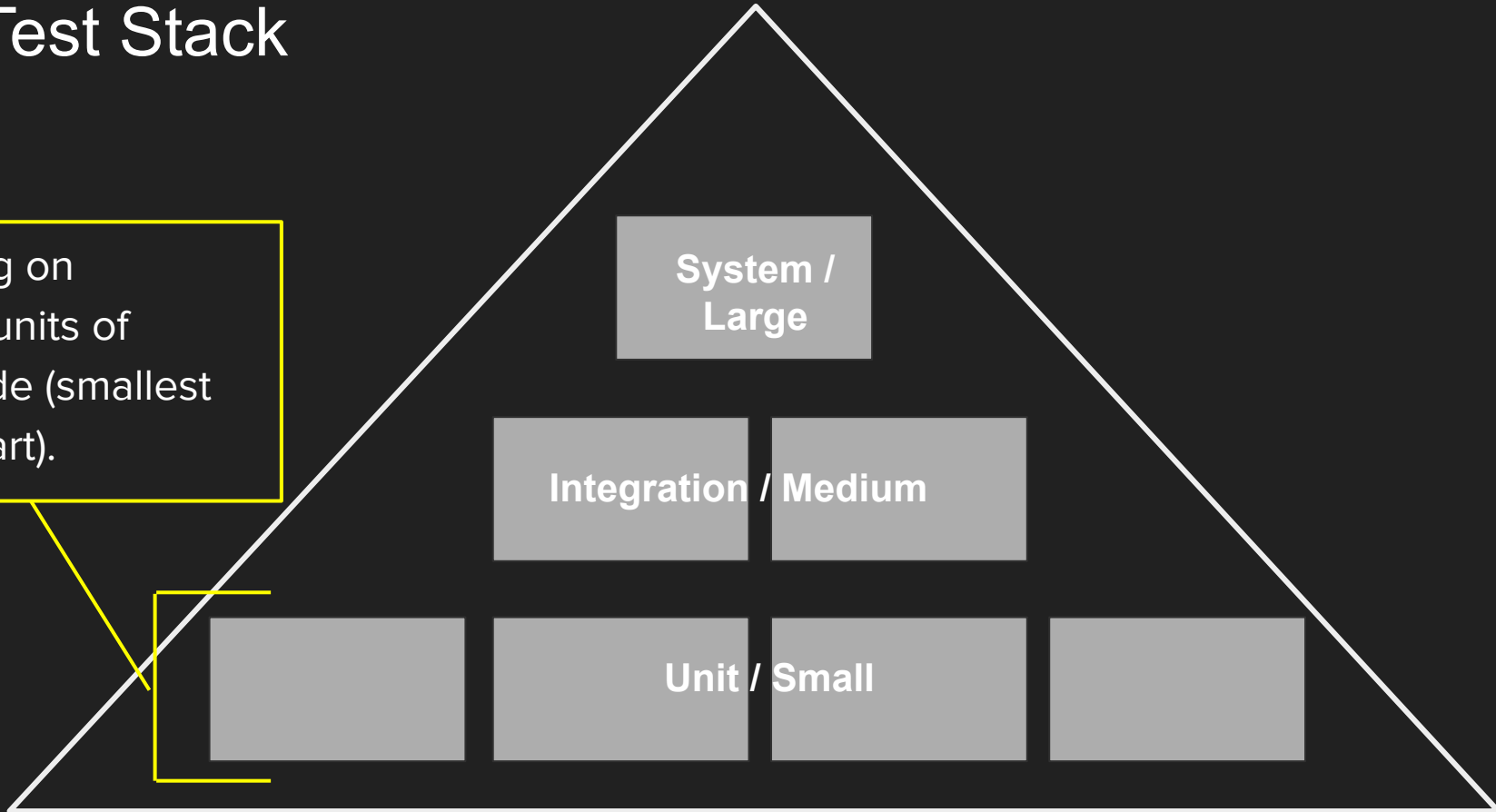
Unit / **Small**
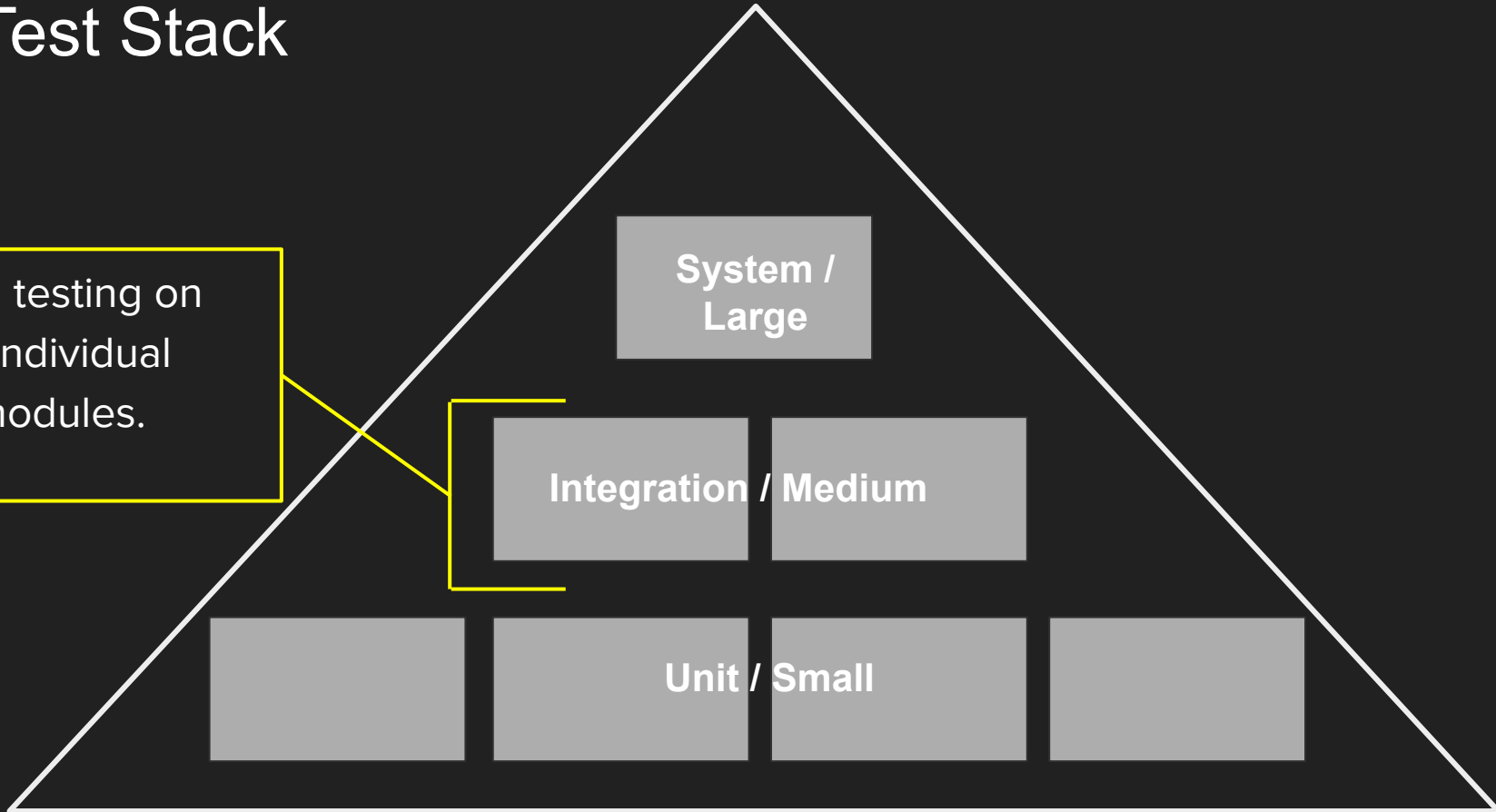
The Test Stack

Integration testing on groups of individual software modules.

System / Large

Integration / Medium

Unit / Small

The Test Stack

System testing on a complete, integrated system (evaluate compliance with requirements)

System / Large

Integration / Medium

Unit / Small
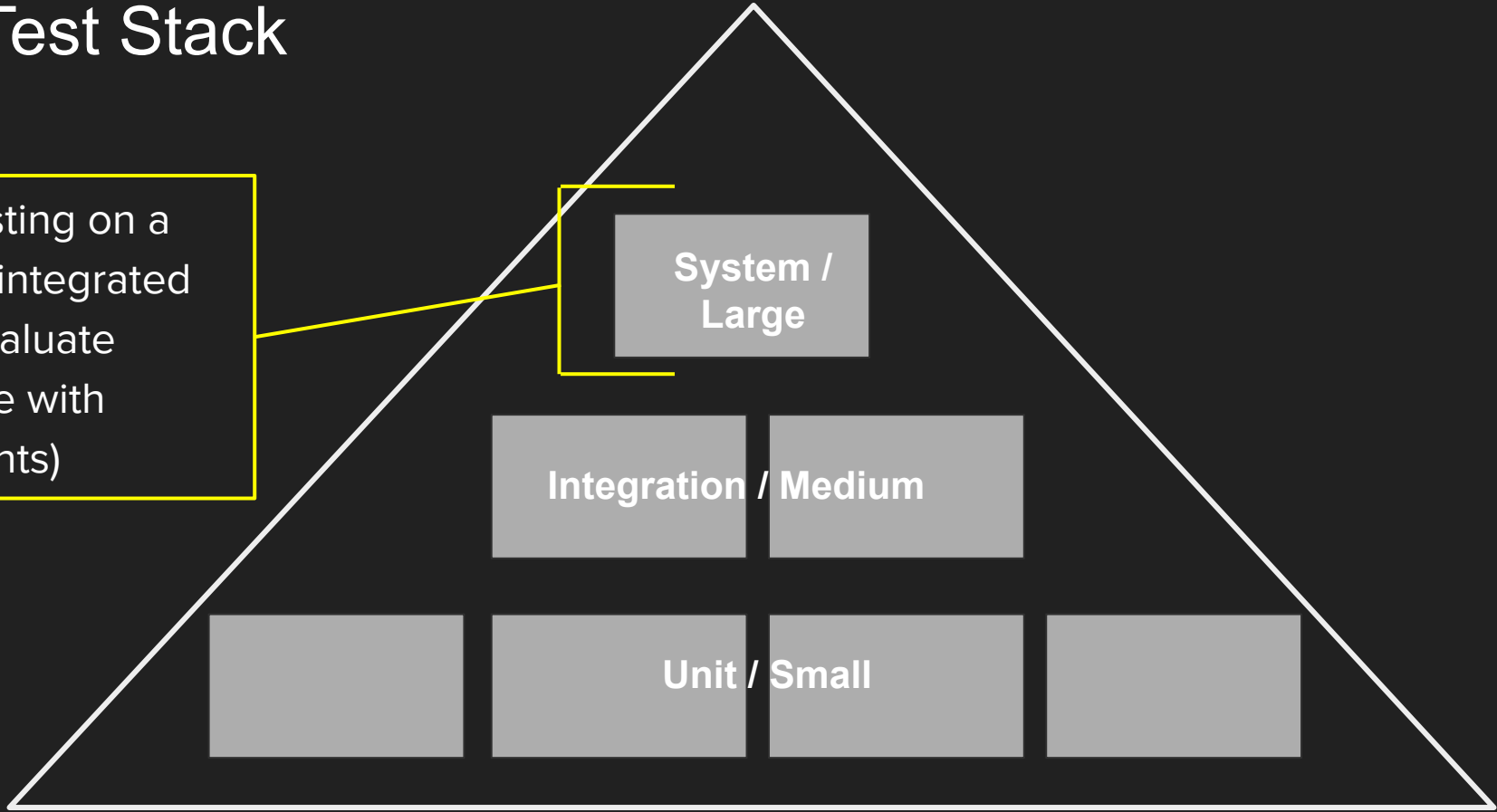
Our Focus…

Unit tests and unit testing

Unit tests and unit testing

- a unit test is a piece of code written by a developer that executes a specific functionality in the code to be tested.

- a unit test targets a small unit of code, e.g., a method or a class

- it ensures that code works as intended, or that it still works as intended in case you need to modify code for fixing a bug or extending functionality.

Unit tests and unit testing

The percentage of code which is tested by unit tests is typically called **test coverage**.

Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

About test coverage in Eclipse:
https://www.eclemma.org/

Unit tests and unit testing

- JUnit (http://junit.org/) is a test framework which uses annotations to identify methods that specify a test. Typically these test methods are contained in a class which is only used for testing. It is typically called a *Test class*.

- Current version is JUnit 5

Tests in JUnit

- Tests are realized as *public void AUsefulTestName()* methods.

- A test typically calls a few methods, then checks if the state matches the expectation. If not, it fails.

Tests in JUnit

- To define that a certain method is a test method, annotate it with the **@Test** annotation.

- This method executes the code under test. You use an ***assert* method**, provided by JUnit or another assert framework, **to check an expected result versus the actual result**. These method calls are typically called ***asserts*** or ***assert statements***.

- You should provide meaningful messages in assert statements. That makes it easier for the user to identify and fix the problem. This is especially true if someone looks at the problem, who did not write the code under test or the test code.

JUnit test example

```java
public class MyClass {

    public int multiply(int a, int b) {

        return (a*b);

    }

}
```

```java
public class MyClassTest {
    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested
        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

JUnit naming conventions

- A widely-used solution for classes is to use the "Test" suffix at the end of test classes names. (The Maven build system automatically includes such classes in its test scope.)

- As a general rule, a **test name** should explain what the test does. If that is done correctly, reading the actual implementation can be avoided.

- One possible convention is to use the "should" in the test method name. For example, "ordersShouldBeCreated" or "menuShouldGetActive". This gives a hint what should happen if the test method is executed.

- Another approach is to use "Given[ExplainYourInput]When[WhatIsDone]Then[ExpectedResult]" for the display name of the test method.

JUnit5

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

With the objective of separating "JUnit the tool" (which we use to write tests) and "JUnit the platform" (which tools use to run our tests) the JUnit team decided to split JUnit 5 into three sub-projects:

- **JUnit Platform**: Contains the engine API and provides a uniform API to tools, so they can run tests

- **JUnit Jupiter:** The API against which we write tests and the engine that understands it.

- **JUnit Vintage**: An engine that allows to run tests written in JUnit 3 and 4 with JUnit 5

JUnit4

Maven Dependency

```xml
28⊖        <dependencies>
29
30         <!-- https://mvnrepository.com/artifact/junit/junit -->
31⊖        <dependency>
32             <groupId>junit</groupId>
33             <artifactId>junit</artifactId>
34             <version>4.13.1</version>
35             <scope>test</scope>
36         </dependency>
37
38⊖        </dependencies>
```

Import JUnit

```java
 3⊖ import static org.junit.Assert.assertTrue;
 4
 5  import org.junit.Test;
 6
 7
 8⊖ /**
 9   * Unit test for simple App.
10   */
11  public class AppTest
12  {
13⊖     /**
14       * Rigorous Test :-)
15       */
16⊖     @Test
17      public void shouldAnswerWithTrue()
18      {
19          assertTrue( true );
20      }
21  }
```

JUnit test example - MyClass

```java
public class MyClass {

  public int multiply(int x, int y) {

    return x / y;

  }

}
```

JUnit test example - MyClassTest

```java
// JUnit 5
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyClassTest {

  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals(50, tester.multiply(10, 5));
  }
}
```

Run Tests
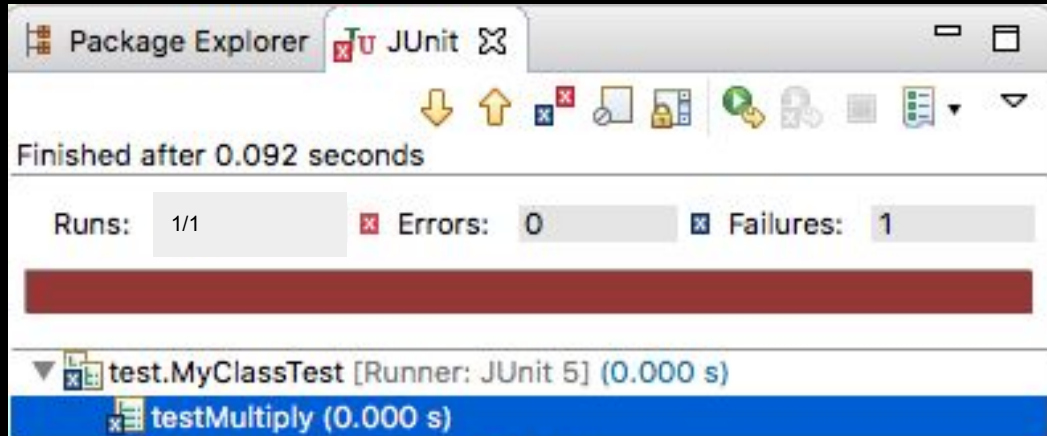
- The test is failing, because our multiplier class is currently not working correctly.

- It does a division instead of multiplication.

- Fix the bug and re-run the test to get a green bar.

JUnit test example - MyClass

```java
public class MyClass {

  public int multiply(int x, int y) {

    return x * y;

  }

}
```
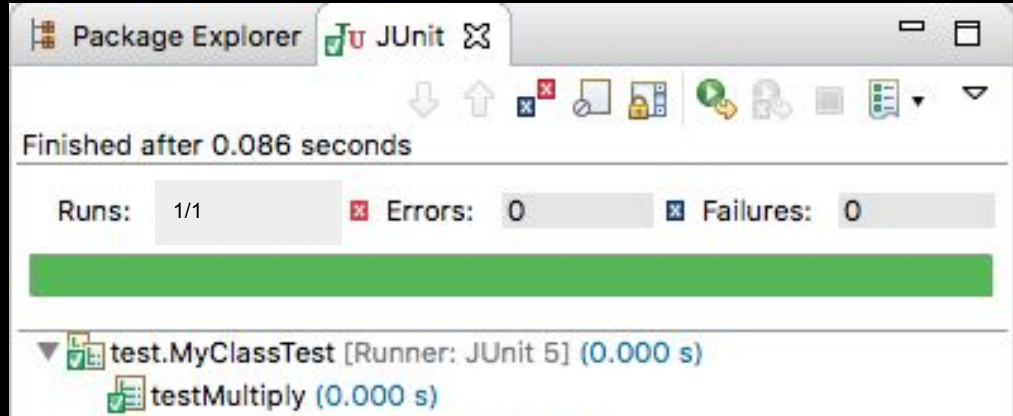
JUnit Eclipse Legend

	JUnit
	test
	currently running test
	successful test
	failing test
	test throwing an exception
	ignored test
	test with an assumption failure

Setup Your Maven Project with JUnit5

To setup JUnit 5
https://maven.apache.org/surefire/maven-surefire-plugin/examples/junit-platform.html

JUnit5 dependencies:

- junit-jupiter-api: for writing JUnit5 tests
- junit-jupiter-engine: for running JUnit5 tests
- junit-platform-xxx: the foundation for JUnit5
- (Optionally) you might want to include junit-vintage-engine for running JUnit4 tests

Available JUnit **annotations**

All core **annotations** are located in the org.junit.jupiter.api package in the junit-jupiter-api module.

Annotation	Description
@Test	Denotes that a method is a test method. Unlike JUnit 4's @Test annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are inherited unless they are overridden
@ParameterizedTest	Denotes that a method is a parameterized test. Such methods are inherited unless they are overridden.
@RepeatedTest	Denotes that a method is a test template for a test that we want to repeat several times. Such methods are inherited unless they are overridden.
@DisplayName	Declares a custom display name for the test class or test method. Such annotations are not inherited.

Available JUnit annotations

Annotation	Description
@BeforeEach	Denotes that the annotated method should be executed before each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4's @Before. Such methods are inherited unless they are overridden.
@BeforeAll	Denotes that the annotated method should be executed before all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4's @BeforeClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" test instance lifecycle is used).
@AfterEach	Denotes that the annotated method should be executed after each @Test, @RepeatedTest, @ParameterizedTest, or @TestFactory method in the current class; analogous to JUnit 4's @After. Such methods are inherited unless they are overridden.
@AfterAll	Denotes that the annotated method should be executed after all @Test, @RepeatedTest, @ParameterizedTest, and @TestFactory methods in the current class; analogous to JUnit 4's @AfterClass. Such methods are inherited (unless they are hidden or overridden) and must be static (unless the "per-class" test instance lifecycle is used).

Test Classes and Methods

A **test method** is any instance method that is directly or meta-annotated with @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, or TestTemplate.

A **test class** is any top level or static member class that contains at least one test method.

Assertions

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
assertTrue(boolean condition)	Checks that the boolean condition is true.
assertFalse(boolean condition)	Checks that the boolean condition is false.
assertEquals(expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals(expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.

Assertion allow to add a message
JUNIT4: assertEquals("10 multiplied by 5 should return 50",50, tester.multiply(10, 5));

JUNIT5: assertEquals(50, tester.multiply(10, 5),"10 multiplied by 5 should return 50");

Assertions

Statement	Description
assertNull(object)	Checks that the object is null.
assertNotNull(object)	Checks that the object is not null.
assertSame(expected, actual)	Checks that both variables refer to the same object.
assertNotSame(expected, actual)	Checks that both variables refer to different objects.
assertArrayEquals(expected, actual)	Checks both array contains same values

Display Names

Test classes and test methods can declare custom display names — with spaces, special characters, and even emojis — that will be displayed by test runners and test reporting.

```java
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
@DisplayName("A special test case")
class DisplayNameDemo {
  @Test
  @DisplayName("Custom test name containing spaces")
  void testWithDisplayNameContainingSpaces() {
  }
  @Test
  @DisplayName("╯°□°）╯")
  void testWithDisplayNameContainingSpecialCharacters() {
  }
  @Test
  @DisplayName(" 😈")
  void testWithDisplayNameContainingEmoji() {
  }
}
```

Disabling tests

- The **@Ignore** annotation allow to statically ignore a test. The **@Disabled** allow to disable a test.
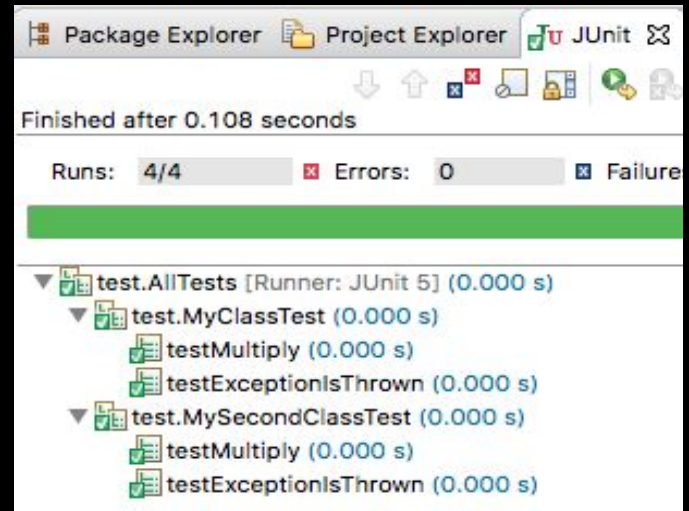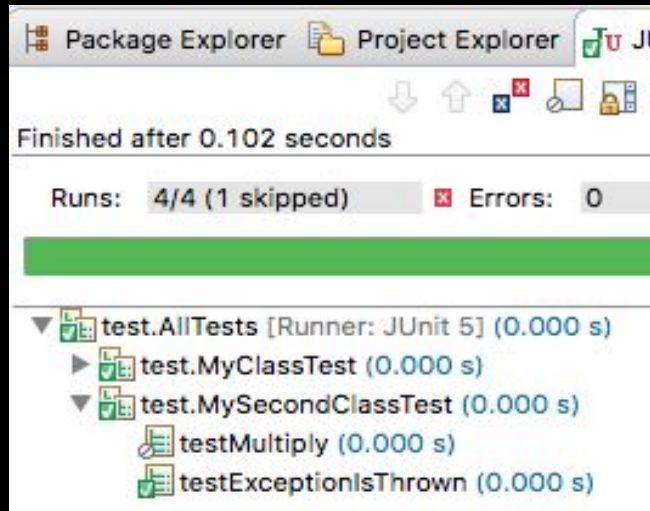
Assumptions

- Alternatively you can use Assume.assumeFalse or Assume.assumeTrue to define a condition for the test.

- Assume.assumeFalse(System.getProperty("os.name").contains("Mac OS X"));

- Assume.assumeTrue(System.getProperty("os.name").contains("Mac OS X"));

- All JUnit Jupiter assumptions are static methods in the org.junit.jupiter.api.Assumptions class.

Conditional Test Execution

The **ExecutionCondition** extension API in JUnit Jupiter allows developers to either enable or disable a container or test based on certain conditions programmatically.

Operating System Conditions A container or test may be enabled or disabled on a particular operating system via the @EnabledOnOs and @DisabledOnOs annotations.

```
@Test
@EnabledOnOs(MAC)
void onlyOnMacOs() {
  // ...
}
@TestOnMac
void testOnMac() {
  // ...
}
```

```
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
  // ...
}
@Test
@DisabledOnOs(WINDOWS)
void notOnWindows() {
  // ...
}
```

Conditional Test Execution

Java Runtime Environment Conditions A container or test may be enabled or disabled

on a particular version of the Java Runtime Environment (JRE) via the @EnabledOnJre and

@DisabledOnJre annotations.

```java
@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}
```

```java
@Test
@EnabledOnJre({ JAVA_9, JAVA_10 })
void onJava9Or10() {
    // ...
}
```

```java
@Test
@DisabledOnJre(JAVA_9)
void notOnJava9() {
    // ...
}
```

Conditional Test Execution

Environment Variable Conditions A container or test may be enabled or disabled based on the value of the named environment variable from the underlying operating system via the **@EnabledIfEnvironmentVariable** and **@DisabledIfEnvironmentVariable** annotations. The value supplied via the matches attribute will be interpreted as a regular expression.

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = "staging-server")
void onlyOnStagingServer() {
  // ...
}

@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*development.*")
void notOnDeveloperWorkstation() {
  // ...
}
```

Conditional Test Execution

Script-based Conditions enable or disable a test based on the evaluation of a script

configured via the **@EnabledIf** or **@DisabledIf** annotation. Scripts can be written in JavaScript,

Groovy, or any other scripting language for which there is support for the Java Scripting API, defined

by JSR 223

```java
@Test // Static JavaScript expression.
@EnabledIf("2 * 3 == 6")
void willBeExecuted() {
  // ...
}


@RepeatedTest(10) // Dynamic JavaScript
expression.
@DisabledIf("Math.random() < 0.314159")
void mightNotBeExecuted() {
  // ...
}
```

```java
@Test // Regular expression testing bound system property.
@DisabledIf("/32/.test(systemProperty.get('os.arch'))")
void disabledOn32BitArchitectures() {
  assertFalse(System.getProperty("os.arch").contains("32"));
}


@Test
@EnabledIf("'CI' == systemEnvironment.get('ENV')")
void onlyOnCiServer() {
  assertTrue("CI".equals(System.getenv("ENV")));
}
```

Parameterized test

- JUnit allows you to use parameters in a tests class. This class can contain one test method and this method is executed with the different parameters provided.

```java
@ParameterizedTest
@ValueSource(strings = { "pippo" , racecar",
"radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
  assertTrue(isPalindrome(candidate));
}
```
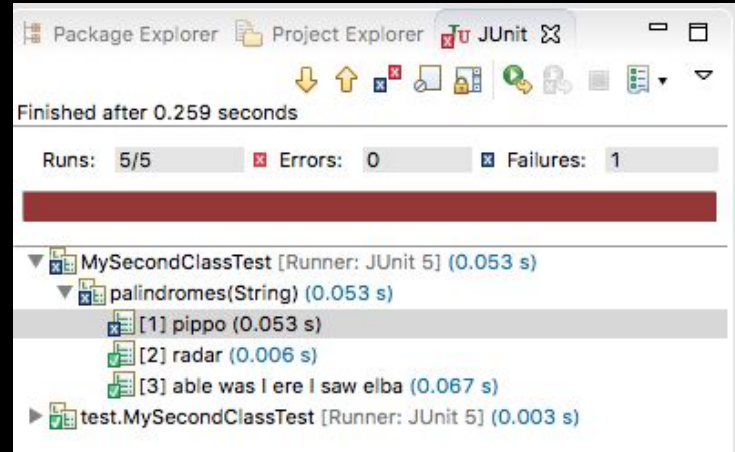
Test Suites

- combine multiple tests into a test suite

- a test suite executes all test classes in that suite in the specified order

- A test suite can also contain other test suites.

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
    MyClassTest.class,
    MySecondClassTest.class })

public class AllTests {

}
```

Tagging and Filtering

Test classes and methods can be tagged via the **@Tag** annotation. Those tags can later be used to filter test discovery and execution.

- A trimmed tag must not contain whitespace.
- A trimmed tag must not contain ISO control characters.
- A trimmed tag **must not contain** any of the following reserved characters:

  ◦ ,: comma

  ◦ (: left parenthesis

  ◦): right parenthesis

  ◦ &: ampersand

  ◦ |: vertical bar

  ◦ !: exclamation point

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

@Tag("fast")
@Tag("model")
class TaggingDemo {

  @Test
  @Tag("taxes")
  void testingTaxCalculation() {
  }

}
```

Test Result from code

JUnit 5 introduces the concept of a Launcher that can be used to discover, filter, and execute tests. The launcher API is in the **junit-platform-launcher** module

```
final LauncherDiscoveryRequest request =
        LauncherDiscoveryRequestBuilder.request()
                                .selectors(selectClass(MyClassTest.class))
                                .selectors(selectClass(HelloWorldTest.class))
                                .build();

final Launcher launcher = LauncherFactory.create();
final SummaryGeneratingListener listener = new SummaryGeneratingListener();

launcher.registerTestExecutionListeners(listener);
launcher.execute(request);

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

Console launcher

- The ConsoleLauncher is a command-line Java application that lets you launch the JUnit Platform from the console. For example, it can be used to run JUnit Vintage and JUnit Jupiter tests and print test execution results to the console.
- An executable **junit-platform-console-standalone-1.3.2.jar** with all dependencies included is published in the central Maven repository under the junit-platform-console-standalone directory.

- java -jar lib/junit-platform-console-standalone-1.3.2.jar --class-path bin --scan-class-path

Github Sample Project

https://github.com/FabrizioFornari/spm2021Template.git

1. Clone (or pull) the repository.
2. Import Maven project in eclipse.

Best practices

- Tests should be written before the code (TDD - Test driven development)

- Test everything that could reasonably break.

- If it can't break on its own, it's too simple to break (like most get and set methods).

- Run all your unit tests as often as possible

Best practices

Whenever you are tempted to type something into a print statement or a debugger expression, **write it as a test instead.**

Martin Fowler

**One of the founding fathers of Extreme Programming**

Extreme Programming (XP)

- A type of Agile software development
  - it advocates frequent "releases" in short development cycles
  - introduce checkpoints at which new customer requirements can be adopted

- Other elements of extreme programming include:
  - programming in pairs or doing extensive code review
  - unit testing of all code
  - avoiding programming of features until they are actually needed
  - code simplicity and clarity
  - expecting changes in the customer's requirements as time passes and the problem is better understood
  - frequent communication with the customer and among programmers
- XP uses Test Driven Development (TDD) and refactoring to help uncover the most effective design.
  - refactoring can be safely achieved only with a strong test system, able to check that the whole software product don't break when you add new code, or when you modify existing ones.

Test-Driven Development (TDD)

- Writing test before code to be tested
  - "a little test, a little code, a little test, a little code, ..."
  - Tests are added gradually during implementation – not in large lump afterwards

- Process of writing tests drives low-level design and programming
  - Tests specify what code should do
  - Tests validate that code does what it should

- Actually, a design and coding practice
- One of the core practices of Extreme Programming
  - Developers have been applying TDD for several decades

TDD cycle

- Proceeds step by step
    a. Write a test.
    b. Design and implement just enough to make the test pass.
    c. Repeat.

- Testing and coding alternate in very small steps
    ○ Duration of one cycle should be a few minutes
    ○ Small steps – difficult to make mistake
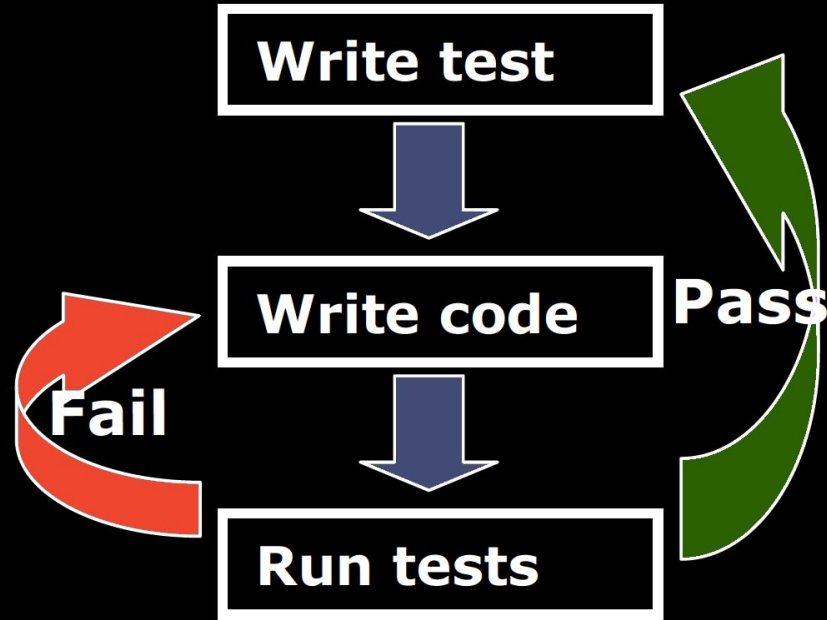
TDD

- TDD procedure is over when you can't write a failing test anymore
  a. Write test for each requirement of the code
  b. Write test for each point that can possibly break

- One cycle at a time
  a. Don't write a bunch of tests at once

- Refactor if you ever see the chance to make the design simpler

- Run all tests after finishing episode
  ○ Make sure you did not break anything else

TDD - claimed benefits (1/2)

- Close feedback loop
  a. TDD cycle is very short – know if code is working right after you programmed it

- Task-orientation
  a. Encourage programmer to decompose problem into manageable programming tasks
  b. Helps to maintain focus
  c. Helps to measure progress and scope work

- Low-level design
  a. Programmer is forced to think which classes and methods to create, how they are used, how to name them, what arguments does a method take, what does a method return

TDD - claimed benefits (2/2)

- Results better code
    a. If the test is too hard to write, the code being tested is too complicated

- Results testable code
    a. Programmer can't end up with code that cannot be tested

- Effect on quality
    a. Testing becomes part of the development process and gets done
    b. Side effect of TDD is that code gets thoroughly unit tested

TDD - Try it!

- The only way to know!

- Personal experiences
  a. Good feeling about the code written
     - General confidence that your code does what you have intended it to do
     - Good feeling when checking your code into version control with all green
  b. Tests really get written when they are written beforehand
     - You always have an up-to-date regression testing suite
     - TDD helps you to keep focus on the current task
     - Program only what is needed to see the green light
  c. Promote best practices
     - System.out.println is used for displaying messages for user – not for developer Debugger is used for debugging

and for anything else…

Check this out: **JUnit 5 User Guide**

https://junit.org/junit5/docs/current/user-guide/index.pdf
or
https://junit.org/junit5/docs/current/user-guide/

and for anything else...

Java Unit Testing with JUnit 5:
Test Driven Development with JUnit 5.

Shekhar Gulati and Rahul Sharma