Week 14 - Software Testing

Louis Botha, `louis.botha@tuni.fi`

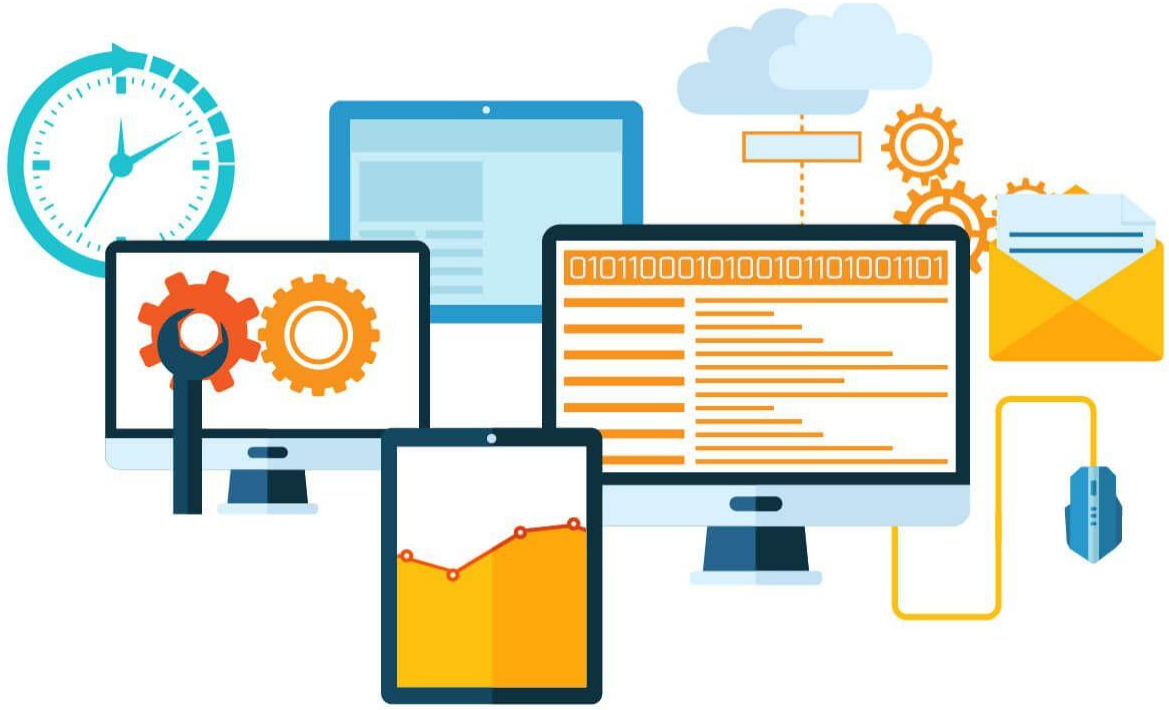


Image Source: Effective software testing and quality assurance - 11 Tips

Software testing is the process of evaluating a software system or application with the intent of finding defects or identifying gaps between expected and actual system behavior. The goal of software testing is to ensure that the software meets the specified requirements and is fit for use by its intended users.

It involves executing software applications under controlled conditions to detect and report any errors, bugs, or vulnerabilities that may compromise the quality, reliability, or security of the software. Testing is an important part of the software development life cycle, as it helps to identify and resolve issues early in the development process, reducing the cost and effort required to fix them later.

Types of Testing

1. Unit Testing: Unit testing is the testing of individual units or components of a software application in isolation. It is usually done by developers and focuses on verifying that each unit works as intended and meets its requirements.

2. Integration Testing: Integration testing is the testing of the interfaces and interactions between different components of a software application. It verifies that the components can work together as intended and that the integration points are functioning correctly.

3. System Testing: System testing is the testing of the entire system as a whole, including its functionality, performance, security, and usability. It verifies that the system meets its overall requirements and objectives.

4. Acceptance Testing: Acceptance testing is the testing that is performed to determine whether the software application meets the customer's requirements and expectations. It is usually done by the customer or end-user and focuses on verifying that the software is fit for purpose and meets the user's needs.

5. Regression Testing: Regression testing is the testing that is performed to verify that changes made to the software do not introduce new defects or break existing functionality. It is usually done after a new version or release of the software is developed.

6. Performance Testing: Performance testing is the testing that is performed to determine the responsiveness, throughput, and scalability of the software application under different workloads and conditions.

7. Security Testing: Security testing is the testing that is performed to verify that the software application is secure against various types of attacks, such as hacking, unauthorized access, and data theft.

8. Usability Testing: Usability testing is the testing that is performed to determine how easy the software application is to use, how intuitive the user interface is, and how well the software meets the user's needs.

Testing Techniques

Here are some commonly used software testing techniques along with examples to illustrate their use:

Black Box Testing
Black box testing is a technique where testers do not have any knowledge of the internal workings of the software application. Testers only examine the input and output behavior of the software application. The goal of black box testing is to identify defects in the functionality of the software. Examples of black box testing include functional testing, usability testing, and acceptance testing.

For example, in functional testing, testers would verify that the software application behaves according to the functional requirements. In usability testing, testers would verify that the software application is easy to use and meets the user's needs. In acceptance testing, testers would verify that the software application meets the customer's requirements and expectations.

White Box Testing
White box testing is a technique where testers have access to the internal workings of the software application. Testers examine the source code, data structures, and algorithms used in the software application. The goal of white box testing is to identify defects in the logic and structure of the software. Examples of white box testing include unit testing, integration testing, and code review.

For example, in unit testing, developer would write unit tests that would verify the behavior of individual units or components of the software application. In integration testing, developers and testers would verify the interactions between different components of the software application. In code review, developers and testers would examine the source code to identify defects in the logic, structure, or style of the code.

Regression Testing
Regression testing is a technique where testers re-run a set of test cases to verify that the software application still works as intended after changes have been made. The goal of regression testing is to catch defects that may have been introduced as a result of the changes. Examples of regression testing include functional regression testing, performance regression testing, and compatibility regression testing.

For example, in functional regression testing, testers would re-run the functional test cases to verify that the software application still behaves as intended after changes have been made. In performance regression testing, testers would re-run the performance test cases to verify that the software application still meets the performance requirements after changes have been made. In compatibility regression testing, testers would re-run the compatibility test cases to verify that the software application still works with other systems or software after changes have been made.

Exploratory Testing
Exploratory testing is a technique where testers use their knowledge and experience to explore the software application and discover defects. Testers do not follow a predefined set of test cases but instead, use their intuition and creativity to uncover defects. The goal of exploratory testing is to discover defects that may have been missed by other testing techniques. Examples of exploratory testing include usability testing, security testing, and ad-hoc testing.

For example, in usability testing, testers would use the software application as a user would and look for usability issues that may not have been identified in previous testing. In security testing, testers would try to hack or exploit the software application to uncover security vulnerabilities. In ad-hoc testing, testers would explore the software application in an unstructured and unplanned way to discover defects.

The Testing Pyramid

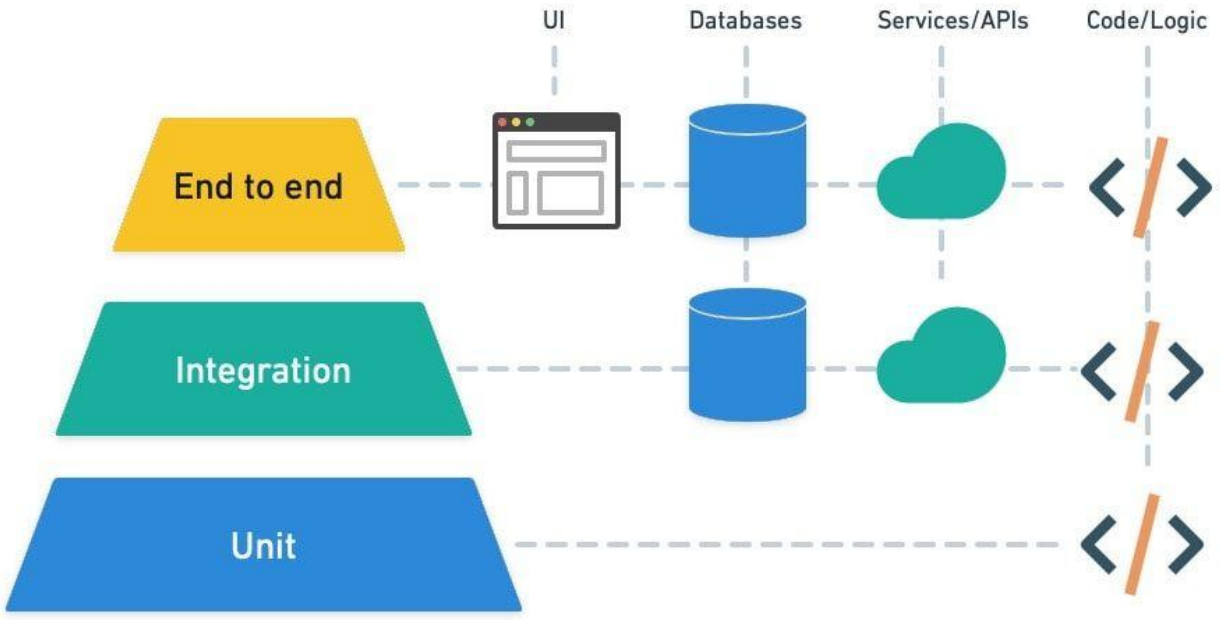


Image Source: Sempahore - The Testing Pyramid: How to Structure Your Test Suite

The testing pyramid is a popular concept in automated software testing that represents the ideal balance between different types of tests needed for a software application. Essentially, the testing pyramid, also referred to as the test automation pyramid, lays out the types of tests that should be included in an automated test suite. It also outlines the sequence and frequency of these tests.

> *The whole point of automated testing is to offer **immediate feedback** to ensure that code changes do not disrupt existing features.*

The pyramid is typically divided into three layers, with each layer representing a different type of test:

1. Unit Tests: At the bottom of the pyramid are the unit tests, which are automated tests that verify the behavior of individual units or components of a software application in isolation. Unit tests are fast, cheap, and reliable, and they help to catch defects early in the development process.
2. Integration Tests: In the middle of the pyramid are the integration tests, which are automated tests that verify the behavior of the interactions between different components of a software application. Integration tests are more complex and slower than unit tests but are still essential for ensuring that the software works as intended and that the different components can interact with each other.
3. End-to-End Tests: At the top of the pyramid are the end-to-end tests, which are automated tests that verify the behavior of the entire system, including the user

interface, backend systems, and external dependencies. End-to-end tests are the slowest and most complex type of test but are still important for verifying that the software application meets the customer's requirements and expectations.

By focusing on unit tests and integration tests early in the development process and using end-to-end tests sparingly, teams can achieve a balance between testing coverage, test speed, and test reliability, which is essential for delivering high-quality software applications.

Unit Testing with Java and JUnit5

Unit testing is a software testing technique in which individual units or components of a software application are tested in isolation from the rest of the system. The purpose of unit testing is to ensure that each unit of code functions as expected and meets its requirements.

In unit testing, a unit is typically a single function, method, or class, although it can also be a small group of related functions or methods. Each unit is tested independently by providing a set of inputs and verifying that the output is correct.

Unit tests are typically automated, so they can be executed frequently during the development process to catch errors early and ensure that changes to the code don't introduce new bugs.

Unit testing can provide many benefits, including:

- Finding and fixing bugs early in the development process, before they become more difficult and expensive to fix.
- Ensuring that each unit of code meets its requirements and works as expected, which can improve overall software quality.
- Making it easier to modify and maintain the code by catching regressions and ensuring that changes don't break existing functionality.
- Providing a safety net that allows developers to make changes to the code with confidence.

Overall, unit testing is an essential part of modern software development, and it is widely used in agile and DevOps environments to ensure that code is reliable, maintainable, and high-quality.

Getting Started

For a Gradle project you will need to need the `junit-jupiter` artifact as a dependency in `build.gradle`:

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.8.0")
}
```

Now all we need is to tell to use the JUnit platform in the tests:

```
test {
    useJUnitPlatform()
}
```

The JUnit Gradle plugin discovers tests under `src/test/java` directory by default.

Now we can run our tests on the command line with:

```
$ gradle test
```

We should see output similar to this:

```
:test
BUILD SUCCESSFUL in 2s
```

My First Unit Test

No instructions on unit tests will be complete if it doesn't use the good old calculator example.

Create a class class called `Calculator` in a file called `Calculator.java` in the `src/main/java`.

```
public class Calculator {
    public int add(int first, int second) {
        return first + second;
    }
}
```

Create a file called `CalculatorTest.java` in the `src/test/java` directory.

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

public class CalculatorTest {

    @Test
    void addNumbers() {
        Calculator calculator = new Calculator();
        int sum = calculator.add(1, 2);
        Assertions.assertEquals(3, sum);
    }
}
```

Run the test.

```
❯ gradle test

BUILD SUCCESSFUL in 450ms
3 actionable tasks: 3 up-to-date
```

The Arrange, Act, Assert (AAA) pattern

The Arrange, Act, Assert (AAA) pattern is a commonly used structure for organizing unit tests. It helps to make test cases more readable, maintainable, and understandable. Here's a breakdown of each step in the AAA pattern:

Arrange

In the first step of the AAA pattern, you set up the test environment by initializing any necessary objects, creating mock objects, or providing input data. This is the "arrange" step, where you prepare the system under test (SUT) and its dependencies for the test case.

Act

In the second step of the AAA pattern, you execute the SUT by calling a method or invoking some behavior. This is the "act" step, where you perform the action that you want to test. The act step should be simple and focused on testing a single behavior or method.

Assert

In the final step of the AAA pattern, you check the results of the test case by verifying that the expected output or behavior was produced. This is the "assert" step, where you compare the actual results of the SUT with the expected results. The assert step should include one or more assertions that validate the correctness of the SUT's output.

Let’s identify the AAA pattern in the test of above

```
public class CalculatorTest {
    @Test
    void addNumbers() {
        // Arrange
        Calculator calculator = new Calculator();
        // Act
        int sum = calculator.add(1, 2);
        // Assert
        Assertions.assertEquals(3, sum);
```

```
    }
}
```

Assertions

We already checked **assertEquals**. There is some more assertion that we can use.

assertTrue and assertFalse

The `assertTrue` and `assertFalse` assertions test that a boolean condition is true or false, respectively. Here's an example:

```
@Test
void testIsEven() {
    Calculator calculator = new Calculator();
    assertTrue(calculator.isEven(2));
    assertFalse(calculator.isEven(3));
}
```

In this example, we use `assertTrue` to test that the result of calling the `isEven` method of the `Calculator` class with the argument `2` is `true`, and `assertFalse` to test that the result of calling the same method with the argument `3` is `false`.

assertNull and assertNotNull

The `assertNull` and `assertNotNull` assertions test that a value is null or not null, respectively. Here's an example:

```
@Test
void testDivisionByZero() {
    Calculator calculator = new Calculator();
    Integer result = calculator.divide(5, 0);
    assertNull(result);
}
```

In this example, we use `assertNull` to test that the result of calling the **`divide`** method of the `Calculator` class with the arguments **`5`** and **`0`** is null, since dividing by zero is not a valid operation.

assertThrows

The `assertThrows` assertion tests that a method throws an exception of a specific type. Here's an example:

```
@Test
void testDivisionByZeroThrowsException() {
    Calculator calculator = new Calculator();
    assertThrows(ArithmeticException.class, () -> {
        calculator.divide(5, 0);
    });
}
```

assertIterableEquals

The `assertIterableEquals` assertion in JUnit5 is used to test that two iterables contain the same elements in the same order. It is similar to the `assertEquals` assertion, but it is specifically designed for iterables.

Here's an example of how to use `assertIterableEquals` in a test:

```
@Test
void testSort() {
    List<Integer> numbers = Arrays.asList(4, 1, 3, 2);
    Collections.sort(numbers);
    assertIterableEquals(Arrays.asList(1, 2, 3, 4), numbers);
}
```

In this example, we have a list of integers that is not sorted. We sort the list using the `Collections.sort` method, and then we use `assertIterableEquals` to test that the sorted list contains the same elements as the expected list, which is created using the `Arrays.asList` method.

If the two iterables contain the same elements in the same order, the `assertIterableEquals` assertion passes. If the two iterables contain different elements or the elements are in a different order, the assertion fails.

assertArrayEquals

The `assertArrayEquals` assertion in JUnit5 is used to test that two arrays are equal. It is similar to the `assertEquals` assertion, but it is specifically designed for arrays.

Here's an example of how to use `assertArrayEquals` in a test:

```
@Test
void testArray() {
    final int[] array = { 3, 2, 1 };
    final int[] expected = { 1, 2, 3 };
    Arrays.sort(array);
    assertArrayEquals(expected, array);
}
```

In this example, we have two arrays of integers, and we use `assertArrayEquals` to test that the two arrays are equal.

If the two arrays have the same length and the same values in the same order, the `assertArrayEquals` assertion passes. If the two arrays have different lengths or the values are in a different order, the assertion fails.

Using `assertArrayEquals` can be helpful when you need to test that a method returns an array of objects with specific values. It is especially useful when testing algorithms that generate or manipulate arrays.

For more complex assertions, the JUnit 5 documentation recommends using third-party assertion libraries, such as Hamcrest, AssertJ or Truth.

Lifecycle Methods

JUnit5 provides several lifecycle methods that you can use to set up and tear down test fixtures, perform actions before or after each test, and manage resources used by the tests. Here are some of the most commonly used JUnit5 lifecycle methods:

@BeforeAll

The `@BeforeAll` annotation is used to annotate a method that should be executed once before all the tests in a test class. This method is typically used to set up any resources that will be shared across all the tests, such as database connections or external services.

```
@BeforeAll
static void setUpClass() {
    // set up resources for all tests in the class
}
```

@AfterAll
The `@AfterAll` annotation is used to annotate a method that should be executed once after all the tests in a test class have been executed. This method is typically used to clean up any resources that were set up in the `@BeforeAll` method.

```
@AfterAll
static void tearDownClass() {
    // clean up resources for all tests in the class
}
```

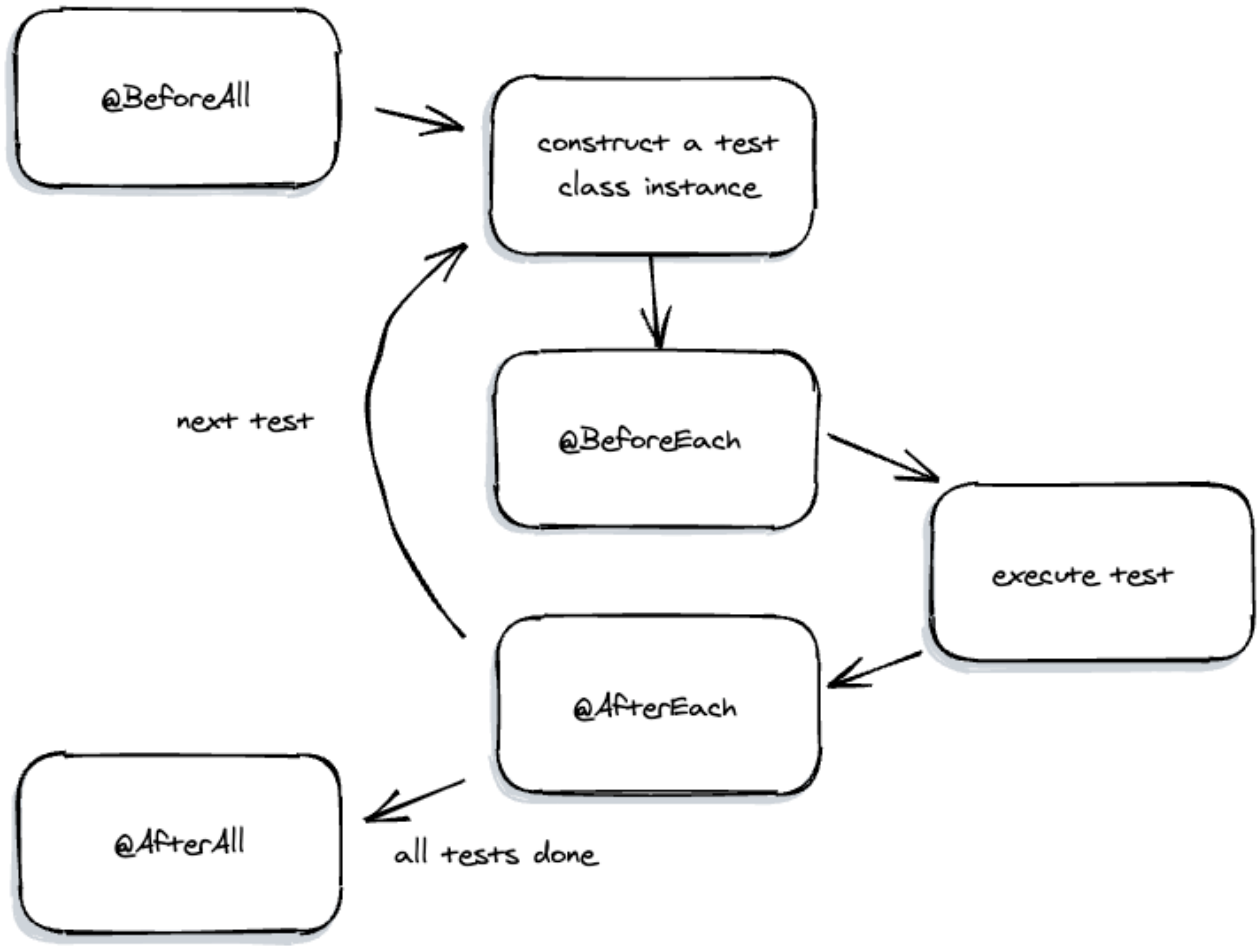
@BeforeEach
The `@BeforeEach` annotation is used to annotate a method that should be executed before each test method in a test class. This method is typically used to set up any resources that are required for each individual test.

```
@BeforeEach
void setUp() {
    // set up resources for each individual test
}
```

@AfterEach
The `@AfterEach` annotation is used to annotate a method that should be executed after each test method in a test class has been executed. This method is typically used to clean up any resources that were set up in the `@BeforeEach` method.

```
@AfterEach
void tearDown() {
    // clean up resources for each individual test
}
```

These lifecycle methods provide a way to manage test fixtures and resources in a consistent and reliable way. By using them, you can ensure that your tests are always executed in a predictable and repeatable environment.



Source: Arho Huttunen

Let's use these in a test file and see when these are called.

```
public class LifetimeTest {
    public LifetimeTest() {
```

```
        System.out.println("LifetimeTest Constructor");
    }

    @BeforeAll
    static void setupBeforeAll() {
        System.out.println("Before the entire test fixture");
    }

    @AfterAll
    static void teardownAferAll() {
        System.out.println("After the entire test fixture");
    }

    @BeforeEach
    void setupBeforeEachTest() {
        System.out.println("Before each test");
    }

    @AfterEach
    void tearDownAfterEachTest() {
        System.out.println("After each test");
    }

    @Test
    void firstTest() {
        System.out.println("First test");
    }

    @Test
    void secondTest() {
```

```
        System.out.println("Second test");
    }
}
```

When running the test case we see the following output

```
Before the entire test fixture
LifetimeTest Constructor
Before each test
First test
After each test
LifetimeTest Constructor
Before each test
Second test
After each test
After the entire test fixture
```

Annotations

There are several other important annotations in JUnit5 that you can use to write tests with different features and capabilities. Here are some additional JUnit5 annotations that you might find useful:

@Disabled

The `@Disabled` annotation is used to annotate a test or a test class that should be temporarily disabled. When a test or a test class is disabled, it will be skipped during test execution.

```
@Disabled
@Test
void testMethod() {
    // this test will be skipped
}
```

@Nested

The `@Nested` annotation is used to create nested test classes, which can help organize your tests and make them more readable. You can use the `@Nested` annotation to create inner classes that contain related tests.

```
public class OuterClass {

    @Nested
    public class InnerClass {

        @Test
        void testMethod() {
            // this test is part of the InnerClass
        }
    }
}
```

@DisplayName

The `@DisplayName` annotation is used to provide a custom display name for a test or a test class. You can use this annotation to give your tests more meaningful and descriptive names.

```
@DisplayName("A test for some feature")
@Test
void testMethod() {
    // this test has a custom display name
}
```

@Timeout

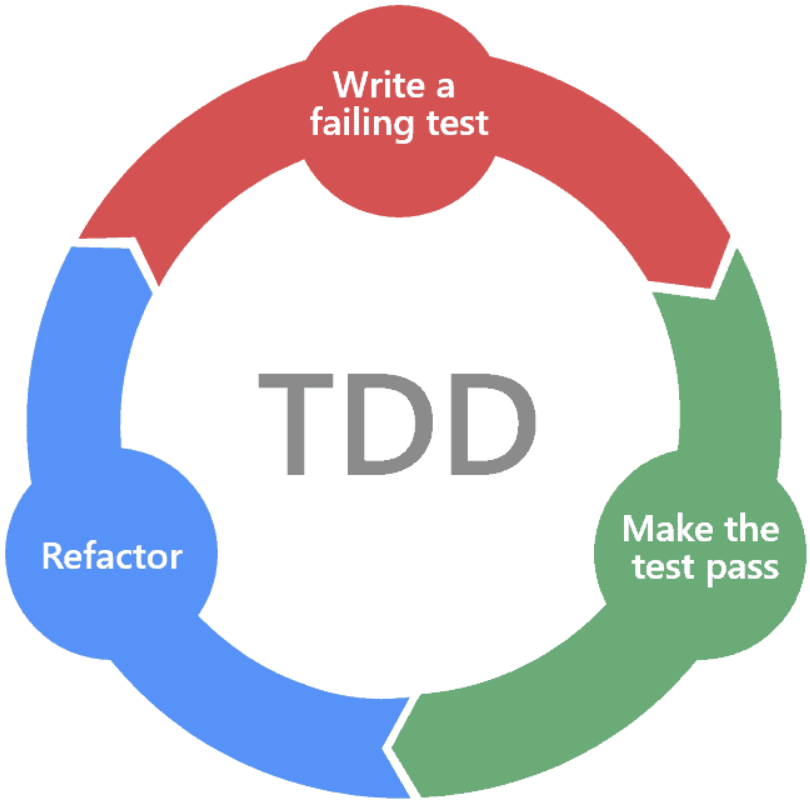
The `@Timeout` annotation is used to set a maximum time limit for a test or a test method. If the test or method takes longer than the specified time limit, it will be marked as a failure.

```
@Test
@Timeout(5) // the test will fail if it takes longer than 5 se
conds
void testMethod() throws InterruptedException {
```

```
    Thread.sleep(6000); // this will cause the test to fail
}
```

TDD

We cant talk about unit tests and not mention TDD. TDD stands for Test-Driven Development. It is a software development process that emphasizes writing automated tests before writing the actual production code.



Why Test-Driven Development (TDD) | Marsner Technologies

The TDD process typically follows these steps:

1. Write a test: Write a small automated test that checks for a specific behavior or feature in the system. At this point, the test should fail, because the feature does not exist yet.

2. Write the code: Write the minimum amount of production code that is necessary to make the test pass. At this point, the test should pass.

3. Refactor the code: Once the test is passing, you can refactor the code to make it more readable, maintainable, and efficient. The goal is to improve the quality of the code without changing its behavior.

4. Repeat: Go back to step 1 and write another test for the next feature or behavior.

The key idea behind TDD is that writing tests first helps you think more deeply about the design of your code, because you have to think about how to test the code before you write it. This can help you identify potential issues and design flaws early in the development process, when they are easier and cheaper to fix. TDD can also help you write more maintainable code, because the tests provide a safety net that allows you to make changes to the code without worrying about breaking existing functionality.

Practice

Create a class called `Person` in the `src/main/java` directory. The class has the following properties:

- private String name;
- private int age;
- private List<String> hobbies;
- public void setName(String name);
- public void setAge(int age);
- public String getName();
- public int getAge();
- public void addHobby(String hobby);
- public List<String> getHobbies();
- public String toString();

Create a file called `PersonTest.java` in the `src/test/java` directory. Implement the `Person` class and add unit tests for the class.

Think of all the edge cases and test for it, e.g age can not be less then 0.

Videos

Java Unit Testing with JUnit - Tutorial - How to Create And Use Unit Tests
https://youtu.be/vZm0lHciFsQ