Testing

This is when you try different ways to break the code, to see if it works. This is to make sure that the code works as intended.

Different test categories

- Unit (high level) - Single functionally / class, one pice of code is being tested, and makes sure it is working correctly.
- Integration - Packages, multi class
- System (low level)- Multiple test under the whole thing. Going from the very start to the end of the application.

Testing approaches

- Black-box - testing focus on the app aka the external behavior and not the code. You don't know how it is implemented.
- White box - is when you look through the code or test it using code whose sole purpose is to test the code. You know how it is implemented. This allows you to test every part of your code, to see if you tested everything you could use codecov to see how much of the code is being tested.

Pros and cons of black-box and white-box testing

Black-box testing: Pros:

1. BlackBox testing allows for independent testing by individuals or teams who are not involved in the development process, leading to uncovering of issues that where overlooked during development.
2. Black-box testing simulates real-world user scenarios, ensuring that the software meets the user's requirements and expectations.
3. Black-box testing help identify issues that can be from miscommunication between developers and users. Black-box testers can uncover issues such as missing or misinterpreted features, incorrect user interface elements, or usability problems that may have been overlooked during the development process.

White-box testing: Pros:

1. Thorough coverage: White-box testing allows testers to see the internal structure and code of the software, allowing for more detailed test coverage of all possible paths and code segments.
2. Efficient defect identification: By having knowledge of the internal workings, white-box testing facilitates the identification of the exact location and cause of defects, leading to more efficient debugging.
3. Validation of internal design: White-box testing can verify that the software has been implemented correctly based on the internal design specifications.
4. If a change is made, and you write tests for the other parts of the code, you can just rerun the tests and it will automatically test the whole thing.

Test driven development

- Write a test (red) This is shown first you write a test that will fail. This is to show that the test is working, and you still haven't made the code. Aka (product specification)
- Write the code (green) This is where you write the code that will make the test pass. This will just be a rough implementation of the code.
- Refactor This is when you clean your code, with things like making the variable names clearer, removing duplicate code. This is to make it more readable and easier to understand.

JUnit

Junit is a framework for testing java code. In industry there are two main versions of Junit, Junit 4 and Junit 5. Junit 4 is the most used version of Junit.

Junit 5 is the newest version of Junit, and it is not as used as Junit 4. Junit 5 is a complete rewrite of Junit 4, and it is not backwards compatible.

To run a test in Junit you need to annotate the method with `@Test`. This will tell Junit that this is a test method.

Junit lifecycle

Junit have a lifecycle that it follows when running a test. The lifecycle is as follows:

- @BeforeAll - This is run before all the tests are run. This is used to setup the test environment. (This needs to be static)
- @BeforeEach - This is run before each test. This is used to setup the test environment.
- @Test - This is the actual test. This is where the test is run.
- @AfterEach - This is run after each test. This is used to clean up the test environment.
- @AfterAll - This is run after all the tests are run. This is used to clean up the test environment. (This needs to be static)

Junit parameterized tests

Parameterized tests are a way to run the same test with different parameters. This is done by annotating the test with `@ParameterizedTest` and then adding the parameters to the test method. The parameters are added with `@ValueSource` and then the parameters are added to the test method.

```
@ParameterizedTest
@ValueSource(ints = {1, 2, 3})
void test(int number) {
    Assertions.assertEquals(number, number);
}
```

Junit assert

Assertion are used to check if the test is passing or not. This is done by using the `assertEquals()` method. This method takes in two parameters, the first one is the expected value, and the second one is the actual value. If the expected value is the same as the actual value, then the test will pass. If the expected value is not the same as the actual value, then the test will fail. Usually it is best practice to have on assert per test.

There are also other assertions that can be used. These are:

- `assertTrue()` - This will check if the value is true.
- `assertFalse()` - This will check if the value is false.
- `assertNull()` - This will check if the value is null.
- `assertNotNull()` - This will check if the value is not null.
- `assertArrayEquals()` - This will check if the arrays are the same.
- `assertIterableEquals()` - This will check if the iterables are the same.
- `assertLinesMatch()` - This will check if the lines are the same.
- `assertAll()` - This will check if all the assertions are true.

Junit 4 vs Junit 5

Architecture: JUnit 4 has a monolithic architecture where all the testing features are bundled together in the `junit.jar` file. In contrast, JUnit 5 has a modular architecture divided into three main components: JUnit Platform, JUnit Jupiter, and JUnit Vintage. This modular design allows better extensibility and flexibility.

The **Junit platform** is the thing that runs the tests. It also defines the TestEngine API for developing a testing framework that runs on the platform.

JUnit Jupiter is holds the annotations and methods that are used to write tests.

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

Junit 4 has some different annotations than Junit 5, for example in Junit 4 to ignore something the annotation is `@Ignore`, whereas in Junit 5 it is `@Disabled`. Or things like `@Before` and `@After` are now `@BeforeEach` and `@AfterEach` in junit 5.

Mocking

With unit testing you want to test a single piece of code. This means that you don't want to test the whole application, but just a single piece of code. When you want to test a part of code that uses another part of code, you don't want to test the other part of code. This is because you want to test the code in isolation, so you know if that part of the code works. This is where mocking comes in.

Mocking allows you to replace real dependencies with simulated objects, called mocks, which mimic the behavior of the real objects. This is done by creating a mock object that will be used instead of the actual object. This is done by using a mocking framework like Mockito.

Mocking also allows you to verify that certain methods are called on the mock object. This is done by using the `verify()` method. This method takes in the mock object and then the method that you want to verify is called on the mock object.

** Not using mocking **

```
class StudentServicesTest{
    private StudentRepository studentRepository;
    private StudentServices studentServices;

    @BeforeEach
    void setup(){
        studentRepository = new StudentRepository();
        studentServices = new StudentServices(studentRepository);
    }

    @Test
    void canGetAllStudents(){
        List<Student> students = studentServices.getAllStudents();
        assertEquals(3, students.size());
    }
}
```

Using mocking

```
class StudentServicesTest{
    @Mock private StudentRepository studentRepository;
    private StudentServices studentServices;
    private AutoCloseable autoCloseable;

    @BeforeEach
    void setup(){
        @Mock
        studentRepository = mock(StudentRepository.class);
        autoCloseable = MockitoAnnotations.openMocks(this); // This will
open the mock AutoCloseable is used to close the mock after the test is
done.
        studentServices = new StudentServices(studentRepository);
    }

    @AfterEach
    void tearDown() throws Exception {
        autoCloseable.close();// This will close the mock after the test
is done.
    }

    @Test
    void canGetAllStudents(){
        verify(studentRepository).findAll(); // This will verify that the
method findAll() is called on the studentRepository.
    }
}
```

Mocking could also simulate behavior, for example if you are testing a javafx gui application, you could simulate the user clicking a button, or with something like android you could simulate it using a framework called espresso.

javafx example

```
public class testAll(){

    public void testAll(){
    var model = new CalModel();
    var view = new CalView(model);

    view.getTextField().setText("1");
    view.getTextField2().setText("2");

    view.getAddButton().fire();

    assertEquals("3", view.getOutField().getText());

    }
}
```

```
public void throwSomething() throws Exception {
    throw new Exception("Something went wrong");
}

Assertions.assertEquals("Something went wrong", exception.getMessage());
```

JUnit 5 testing

To start with Junit you need to add it as a dependency, which can be done in the pom.xml file.

Scope test is used to signal that the dependency is only used for testing.

```
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.8.1</version>
        <scope>test</scope>
        </dependency>
```

Example of using Junit

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

```java
public class TestMain {
    @Test
    void testAdd() {
        SimpleCalculator simpleCalculator = new SimpleCalculator();
        Assertions.assertEquals(2, simpleCalculator.add(1, 1));
    }
}
```

JUnit 4 testing

To start with Junit you need to add it as a dependency, which can be done in the pom.xml file.

Scope test is used to signal that the dependency is only used for testing.

```xml
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
```

Example of using Junit

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

```java
public class TestMain {
    @Test
    public void testAdd(){
        Main main = new Main();
        int result = main.add(1, 2);
        assertEquals(3, result);
        //First is the expected value, second is the actual value

        assertNotEquals(4, result);
```

```
        assertTrue(result == 3);
        assertFalse(result == 4);
        assertNull(null); // only pass if the value is null
        assertNotNull(1); // only pass if the value is not null
    }
}
```

System testing

We are going to test the whole MVC application, basically we are going to test the whole application as a user would. Without running the application, we are going to test the whole thing.

```
public class CalculatorSystemTest{


    @Test
    public void testAll(){
        CalcModel calcModel = new CalcModel();
        CalcView calcView = new CalcView();
        CalcController calcController = new CalcController(calcModel,
calcView);

        calcController.numberButtonPressed(1);
        calcController.numberButtonPressed(2);
        calcController.getAddButton().fire();

        Assert.assertEquals("3",
calcView.getCalculationTextField().getText());

    }
}
```