Software Testing Coursebook

Welcome to the **Software Testing** module!

During this block you will learn the basics of software testing, including mainly:

- creating efficient and valid unit tests
- testing based on the Junit framework
- software testing in TDD methodology

Get ready for class

Check the list below if you have any programs that we will work with during this class:

✔ Java OpenJDK 11

✔ IntelliJ IDEA Community

Introduction

Software testing is a process related to software development. It ensures the quality of the created software, and its main goal is to verify if the software is compliant with the user's expectations (or with the project specification).

Types of tests

Unit tests

- Verify the performance of single parts of code (usually a method).
- Fast and easy to automate.
- Usually run with **every** source code change.

Integration tests

- Verify the performance of a few application modules at once.
- Quite long execution time.
- Many possible points of failure.

Issue	Unit test	Integration test
Dependencies	Single element of code is tested	Many dependencies are tested
Points of failure	One potential point of failure (method)	Many potential points of failure
Speed	Very high	Potentially long execution time due to e.g. database access
Configuration	No additional configuration	Test can require configuration (e.g. database connection)

System tests

System tests are performed when all system components have been integrated. In this phase we can erform e2e (end to end) test – going through the whole process. For example a functionality of account registration should save a user in the database and send an e-mail with a confirmation of the registration. Testing of this whole process is an e2e test.

Acceptance tests

Software tests, which main goal is not finding bugs but getting a formal confirmation of creating a software with a proper quality (according to the specification)

Characteristics of good tests

Unit tests, as well as software development, should be created according to best practices. Below you can find the most important characteristics of good tests, that we should follow.

FIRST rule

- Fast – they should be executed fast, in order to not extend the build time and not waint long for test results
- Isolated/Independent – tests should not depend on each other (be isolated). Single test execution should not impact on other tests in the *test suite* (a single test package, e.g. tests within a single test class).
- Repeatable – should be repeatable on **any** environment. This means, that they should not have dependencies with other systems, e.g. the database. The configuration of other systems should not impact on the recurrence of test execution on different environments. If a test will be executed 10 times, then we should get the same result 10 times. The use of `Random` class, depending on the order of elements in unordered collections or using real timestamps (by using e.g. `LocalDateTime.now()`) can cause problems with the consistency of test results.
- Self-checking – stating if the whole test passed (no manual interpretation).
- Thorough – written together with the production code. We should check the test cases by providing values giving positive results, negative results and testing edge cases. Thanks to that, the tests will check our code more precisely.

Other good practices

- Single responsibility – one test checks one functionality.
- `Second Class Citizens` – test codeis is not a second category code. It should be maintained as well as production code.

Test-driven development

Test-driven development is one of the techniques of creating software. By creating software with TDD, you need to continuously repeat these steps:

- Red
- Green
- Refactor

Stages of creating software

Red

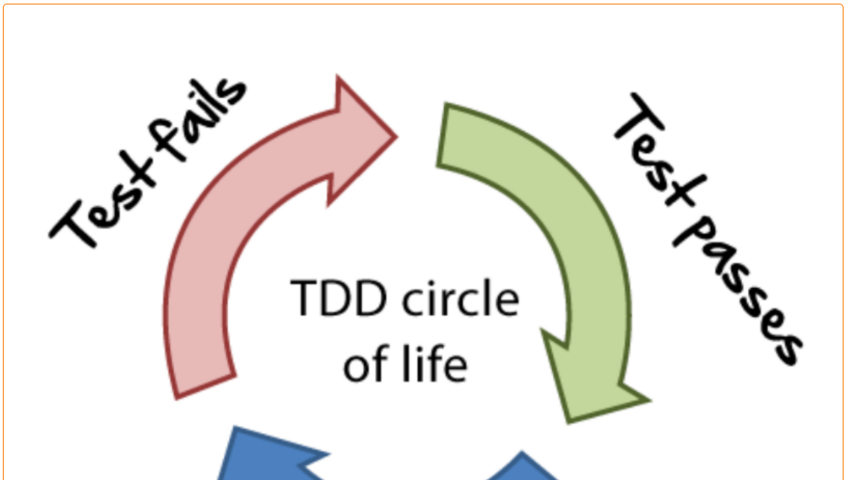
In this step the user needs to create a test which checks a certain functionality. With no code to check against the test will fail. The main code **does not exist**.

Green

The second step in writing the **first** functional revision. At this point the test written in step one should pass but the main code will have some residual errors.

Refactor

The last stage is code refactor which cleans the code so that it fulfills certain standards (like SOLID and DRY).



TDD in details

What can't be understood as TDD.

1. It's not a technique of writing tests. Tests should look like in their "usual" form when you are writing software.
2. It's not a definition of how a code function should look.
3. It's not a way of substituting testers with developers. Testers are always an important part of software development.

Benefits of TDD

There are many benefits of using TDD such as:

- It forces you to cover your code with tests.
- Any conceptual errors are found during writing tests.
- Code modification is safer – we can always check if we had not broken anything by executing exisitng tests.
- Requirements are also showcased by looking at the tests themselves.
- Insufficient requirements are identified at an early stage of application creation.
- The project can be finally written a lot faster than in classical approaches.

Disadvantages of using TDD

Similar to all other approaches TDD has also some drawbacks:

- The project has to have good requirements and documentation written down (which is missing in most cases).
- At the beginning of the project the production code is created rather slowly.
- Any change in requirements implies a lot of change in source code.

Parameterized tests

Introduction

There is often a need to run one unit test several times with different inputs. Such a problem can be solved by duplicating an existing test or extracting a separate method that we call in separate tests. However, such a solution causes the number of unit tests to grow. This problem is solved by parameterized tests, which implement the test code only **once**. The code can be executed multiple times with different sets of input data.

Tests parameterized in JUnit 5

JUnit version 5 provides a set of tools for implementing parameterized tests. For this purpose, the following dependency should be introduced into the project:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.6.2</version> <!-- the current version may change -->
    <scope>test</scope> <!-- scope test means that libraries will be visible only in the
test bundle -->
</dependency>
```

@ParameterizedTest

Parameterized tests are implemented almost identically to regular unit tests, except for the way they are marked. **Instead of** the `Test` annotation, we use `ParameterizedTest`.

Dla poniższej metody statycznej:

```
public class NumbersHelper {
    public static boolean isOdd(int number) {
        return number % 2 != 0;
    }
}
```

The parameterized test may look like this:

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX_VALUE})
void shouldReturnTrueForOddNumbers(int number) {
    assertTrue(NumbersHelper.isOdd(number));
}
```

The `ValueSource` defines the set of inputs that will be passed to the test method as its arguments.

It is the programmer's responsibility to match the number and types of arguments to the source of the arguments. The above example will trigger the unit test six times.

Sources of arguments

JUnit offers several ways to define the argument set passed to a parameterized test. They differ in possibilities and the approach to defining parameters. These annotations are:

- `ValueSource`
- `EnumSource`
- `CsvSource`
- `CsvFileSource`
- `MethodSource`
- `ArgumentsSource`

@ValueSource

The `ValueSource` annotation allows you to pass a single parameter to the test. This parameter can be one of the following types:

Type	Attribute of @ValueSource annotation
short	shorts
byte	bytes
int	ints
long	longs
float	floats
double	doubles
char	chars
java.lang.String	strings
java.lang.Class	classes

> *NOTE:* One limitation of `ValueSource` is that it cannot pass as `null`.

Below are some examples of parameterization tests (for simplicity with an empty body) using `ValueSource`:

```
class ValueSourceExamplesTest {

  @ParameterizedTest
  @ValueSource(doubles = {1, 2.3, 4.1})
  void shouldPassDoubleToParam(double param) {
  }

  @ParameterizedTest
  @ValueSource(strings = {"Ala", "has a", "cat"})
  void shouldPassStringToTest(String word) {
  }

  @ParameterizedTest
  @ValueSource(classes = {String.class, Integer.class, Double.class})
```

```
  void shouldPassClassTypeAsParam(Class<?> clazz) {

  }
}
```

@EnumSource

This annotation allows you to invoke parameterized tests for arguments of enumerated types.

```
public enum TemperatureConverter {
  CELSIUS_KELVIN(cTemp -> cTemp + 273.15f),
  KELVIN_CELSIUS(kTemp -> kTemp - 273.15f),
  CELSIUS_FAHRENHEIT(cTemp -> cTemp * 9 / 5f + 32);

  private Function<Float, Float> converter;

  TemperatureConverter(Function<Float, Float> converter) {
    this.converter = converter;
  }

  public float convertTemp(float temp) {
    return converter.apply(temp);
  }
}
```

For the example above, the parameterized tests may look like this. This test will be run for **every** defined enum value `TemperatureConverter`.

```
@ParameterizedTest
@EnumSource(TemperatureConverter.class)
void shouldConvertToValueHigherThanMinInteger(TemperatureConverter converter) {
  assertTrue(converter.convertTemp(10) > Integer.MIN_VALUE);
}
```

It is also possible to specify specific objects of a given enumeration type using the `names` attribute, e.g.:

```
@ParameterizedTest
@EnumSource(value = TemperatureConverter.class, names = {"CELSIUS_KELVIN",
"CELSIUS_FAHRENHEIT"})
void shouldConvertToTemperatureLowerThanMaxInteger(TemperatureConverter converter) {
  assertTrue(converter.convertTemp(10) < Integer.MAX_VALUE);
}
```

Within this annotation, it is also possible to define a mode that determines whether the values given in the `names` attribute are excluded or included.

mode	values in the names attribute
`EnumSource.Mode.INCLUDE`	includes, this is the default value
`EnumSource.Mode.EXCLUDE`	excludes
`EnumSource.Mode.MATCH_ALL`	matches those that contain *all* the given strings
`EnumSource.Mode.MATCH_ANY`	match those that contain *any* of the given strings

The next example shows the use of the `EnumSource.Mode.EXCLUDE` mode:

```
@ParameterizedTest
@EnumSource(value = TemperatureConverter.class,
    names = {"KELVIN_CELSIUS"},
    mode = EnumSource.Mode.EXCLUDE)
void shouldConvertTemperatureToPositiveValue(TemperatureConverter converter) {
  assertTrue(converter.convertTemp(10) > 0); // the test will run for the values CELSIUS
KELVIN and CELSIUS FAHRENHEIT
}
```

@CsvSource

The `CsvSource` annotation allows you to define test parameters using CSV literals (* comma separated value *). Also:

- Data strings are separated by certain characters (*commas* by default).
- Each separated element is a *separate* parameter taken in the test.
- This mechanism may be used when we want to provide input parameters and the expected value for the purposes of a given unit test.
- The limit of the `CsvSource` annotation is a limited number of types that can be used in the test. All parameter types we want to use in the test must be convertible from the `String` object.

```
public class Strings {
  public static String toUpperCase(String input) {
    return input.trim().toUpperCase();
  }
}
```

```
@ParameterizedTest
@CsvSource({"  test  ,TEST", "tEst ,TEST", "   Java,JAVA"})
void shouldTrimAndUppercaseInput(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

The above parameterized test takes text strings as input. In the process of calling the `Strings.toUpperCase (input)` method the text is stripped of whitespace from the beginning and end of the value and converted to uppercase. It is later compared with the expected value, provied as the second element of the CSV literal .

Within `CsvSource` it is possible to change the separator using the attribute `delimiter`, e.g .:

```
@ParameterizedTest
@CsvSource(value = {"  test  ;TEST", "tEst ;TEST", "   Java;JAVA"}, delimiter = ';')
void shouldTrimAndUppercaseInput(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

@CsvFileSource

The `CsvFileSource` annotation is very similar to the `CsvSource`, except the data is loaded directly from the file. It can define the following parameters:

- `numLinesToSkip` – specifies the number of ignored lines in the source file. This is useful if the data comes from a table that has headers in addition to the value.
- `delimeter` – means a separator between each *element*.
- `lineSeparator` – means the separator between each *sets* of parameters.
- `encoding` – means the file content encoding method.

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1) // the data.csv file must be
in the classpath root, we skip the first line in the file
void shouldUppercaseAndBeEqualToExpected(String input, String expected) {
  String actualValue = Strings.toUpperCase(input);
  assertEquals(expected, actualValue);
}
```

@MethodSource

The `ValueSource`, `@EnumSource`, `CsvSource` and `CsvFileSource` annotations have limitations – the number of parameters or their types. This problem is solved by the argument source defined by the `MethodSource` annotation. The only parameter for this annotation is the * name * of a **static** method in the same class. This method should be argumentless and return stream:

- objects of any type in the case of tests with a single parameter,
- `Arguments` objects for tests with multiple parameters.

Both ways of using the `MethodSource` annotation show the following examples:

```
@ParameterizedTest
@MethodSource("provideNumbers")
void shouldBeOdd(final Integer number) {
  assertThat(number % 2).isEqualTo(1);
```

```
}

static Stream<Integer> provideNumbers() {
  return Stream.of(1, 13, 101, 11, 121);
}
```

The next example uses the `Arguments.of` static method to create a set of arguments for a single parametric test:

```
@ParameterizedTest
@MethodSource("provideNumbersWithInfoAboutParity")
void shouldReturnExpectedValue(int number, boolean expected) {
  assertEquals(expected, number % 2 == 1);
}

private static Stream<Arguments> provideNumbersWithInfoAboutParity() {
  return Stream.of(Arguments.of(1, true),
      Arguments.of(2, false),
      Arguments.of(10, false),
      Arguments.of(11, true));
}
```

@ArgumentsSource

`ArgumentsSource` is the most universal source for defining arguments. Like `MethodSource` it allows you to define *any* number of arguments of *any* type. The main difference between the two annotations is that inside the `ArgumentsSource` annotation we give the type that must implement interface `ArgumentsProvider`.

The method we need to implement, `provideArguments` should return a stream of objects of type `Arguments`.

```
java.util.stream.Stream<? extends org.junit.jupiter.params.provider.Arguments>
provideArguments(org.junit.jupiter.api.extension.ExtensionContext extensionContext)
throws java.lang.Exception;
```

The argument source created this way can be used in many test classes.

```
public class NumberWithParityArgumentsProvider implements ArgumentsProvider {

  @Override
  public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
    return Stream.of(
        Arguments.of(1, true),
        Arguments.of(100, false),
        Arguments.of(101, true)
    );
  }
}
```

A sample test using the `NumberWithParityArgumentsProvider` class might look like this:

```
@ParameterizedTest
@ArgumentsSource(NumberWithParityArgumentsProvider.class)
void shouldReturnExpectedValue(int number, boolean expectedResult) {
  assertEquals(expectedResult, number % 2 == 1);
}
```

Connecting sources

JUnit allows you to combine multiple argument sources. Because the discussed annotations denoting argument sources do not have the annotation `Repeatable`. We can use a specific type of source only once, e.g. the following parameter test will be run 5 times:

```
class CombinedSourcesTest {

  @ParameterizedTest
  @CsvSource(value = "1, true")
  @MethodSource("provideNumbersWithInfoAboutParity")
  void shouldReturnExpectedValue(int number, boolean expected) {
    assertEquals(expected, number % 2 == 1);
  }
```

```
  private static Stream<Arguments> provideNumbersWithInfoAboutParity() {
    return Stream.of(Arguments.of(1, true),
        Arguments.of(2, false),
        Arguments.of(10, false),
        Arguments.of(11, true)
    );
  }
}
```

Support sources

In addition to the standard sources, we have several auxiliary annotations that allow you to add additional parameter values to your existing tests. Those include:

- `NullSource`, which allows you to pass the `null` parameter to the test
- `EmptySource`, which allows you to pass an empty object to the test, e.g.
    - "" in case of `String` class
    - empty collection in case of `Collection`
    - empty array when using array.
- `NullAndEmptySource`, which combines the functionality of the `NullSource` and `EmptySource` annotations.

The following examples show their use:

```
public class Strings {
    public static boolean isBlank(String input) {
        return input == null || input.trim().isEmpty();
    }
}
```

```
@ParameterizedTest
@NullSource
void shouldbeBlankForNull(String input) {
  assertTrue(Strings.isBlank(input));
}
```

```
public class Arrays {
  public static boolean isValid(List<String> values) {
    return values != null && !values.isEmpty();
  }
}
```

```
@ParameterizedTest
@EmptySource
void shouldNotBeValid(List<String> input) {
  assertFalse(Arrays.isValid(input));
}
```

```
@ParameterizedTest
@NullAndEmptySource
void nullAndEmptyShouldBeBlank(String input) {
  assertTrue(Strings.isBlank(input));
}
```

Exception testing

When writing unit tests, we usually start with testing the so-called “happy path”. However, we should not forget to test the scenarios that throw exceptions.

try-catch

We can test exceptions using the `try catch` block and the `fail` assertion from JUnit 5, e.g .:

```
  @Test
```

```
  void shouldThrowException() {
    try {
      numbers.findFirstDigit("");
      fail("Exception was not thrown");
    } catch (final IllegalArgumentException exp) {
      assertEquals("Input cannot be empty", exp.getMessage());
    }
```

assertThrows

Junit 5 provides a set of tools to easily test exceptions in Java. By using dedicated assertions, you can catch an exception and validate its potential details. There are three overloads of such an assertion:

- `public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)`
- `public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable, String message)`
- `public static<T extends Throwable> T assertThrows(Class <T> expectedType, Executable executable, Supplier <String> messageSupplier)`.

The first argument of this method is the expected type of the exception (*or* the type that this exception inherits from). The second argument is the functional interface, in which we should call the code snippet that we think should throw a specific exception. Optionally, we can further personalize the message in the event that such an exception does not occur.

The `assertThrows` assertion returns an instance of the exception thrown. Thanks to this, we can check, e.g. the exception message or the reason for throwing it (i.e. * cause *).

Examples

For the method below:

```
public static float divide(float a, float b) {
  if (b == 0) {
    throw new IllegalArgumentException("dividend can't be 0");
  }
  return a / b;
}
```

the test for the case where an exception is expected might look like this:

```
@Test
void shouldThrowIllegalArgumentException() {
  final IllegalArgumentException exception = assertThrows(IllegalArgumentException.class,
() -> divide(10, 0));

  assertThat(exception).hasMessage("dividend can't be 0")
      .hasNoCause();
}
```

AssertJ

The test for the case where an exception is expected might look like this:

- `assertThatExceptionOfType`
- `assertThatThrownBy`.

Furthermore, AssertJ provides specialized versions of the `assertThatExceptionOfType` method that expects a specific type of exception. It results from the name of the method, e.g .:

- `assertThatNullPointerException`
- `assertThatIllegalArgumentException`.

So we could test the divide method from the previous example in the following ways:

```
@Test
void shouldThrowIllegalArgumentException() {
  assertThatExceptionOfType(IllegalArgumentException.class)
      .isThrownBy(() -> divide(10, 0))
  .withMessage("dividend can't be 0")
  .withNoCause();
}
```

```
@Test
void shouldThrowIllegalArgumentException() {
  assertThatIllegalArgumentException()
      .isThrownBy(() -> divide(10, 0))
      .withMessage("dividend can't be 0")
      .withNoCause();
  }
```

```
@Test
void shouldThrowIllegalArgumentException() {
  assertThatThrownBy(() -> divide(10, 0))
      .isExactlyInstanceOf(IllegalArgumentException.class)
      .hasMessage("dividend can't be 0")
      .hasNoCause();
  }
```

Complete Lesson