

01 Lab

Software Quality and Test Driven Development (TDD)

Mirko Viroli, Roberto Casadei, Gianluca Aguzzi

`{mirko.viroli, roby.casadei, gianluca.aguzzi}@unibo.it`

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2023/2024

Lab 01: Outline

- Software quality, principles and refactoring
- Test Driven Development (TDD)

Lab Setup

- Clone (or fork and clone) the repo at <https://github.com/unibo-pps/pps-23-24-lab01>
- Open the project in Visual Studio Code
 - ▶ File => Open and select the **repository root folder**
 - ▶ You will find a project with two internal modules

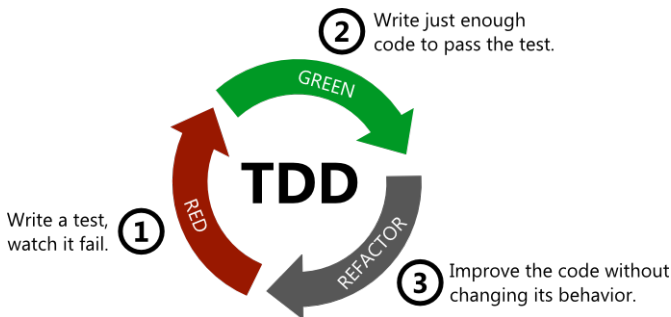
Software Quality Principles (recall)

- **DRY** – Don't Repeat Yourself
- **KISS** – Keep it simple, stupid
- *SOLID Principles*
 - ▶ **SRP** – Single Responsibility Principle
 - ▶ **OCP** – Open/Closed Principle
 - ▶ **LSP** – Liskov' Substitutability Principle
 - ▶ **ISP** – Interface Segregation Principle
 - ▶ **DIP** – Dependency Inversion Principle

On Test Driven Development (TDD) (i)

TDD

- TDD process: **Red-Green-Refactor** cycle
- TDD is about explicitly formalising (and enforcing) the “what” before the “how”.
 - ▶ The term “test” is imprecise.
 - ▶ Your “JUnit code” serves different functions at different times. (Why?)



On Test Driven Development (TDD) (ii)

Guidelines

- **Quality tests:** quality techniques should be applied to test code too!
 - ▶ Systems of tests are software projects on their own!
- Structuring tests: **Arrange-Act-Assert**

```
@Test
void test() {
    // ARRANGE
    final AccountHolder holder = new AccountHolder( name: "Mario", surname: "Rossi", id: 12345);
    final BankAccount account = new SimpleBankAccount(accountHolder, balance: 0);

    // ACT
    account.deposit(holder.getId(), amount: 100);

    // ASSERT
    assertEquals( expected: 100, account.getBalance());
}
```

- Tests should appear as **specifications** or **living documentation**

JUnit 5+ (recall) (i)

Method Annotations (package `org.junit.jupiter.api.*`)

- `@Test` – Denotes that a method is a test method
- `@BeforeEach/@AfterEach` – Denotes that the annotated method should be executed before/after each test method
- `@BeforeAll/@AfterAll` – Denotes that the annotated method should be executed before/after all test method
- `@Disabled` – Used to disable a test class or test method
- `@Timeout` – Used to fail a test if its execution exceeds a given duration

JUnit 5+ (recall) (ii)


Assertions (package `org.junit.jupiter.api.Assertions.*`)

- `assertEqual(Object expected, Object actual)`
 - ▶ Assert that *expected* and *actual* are equal (see also `assertNotEqual`).
- `assertFalse(boolean condition)`
 - ▶ Assert that the supplied *condition* is false.
- `assertTrue(boolean condition)`
 - ▶ Assert that the supplied *condition* is true.
- `assertNull(Object actual)`
 - ▶ Assert that *actual* is null (see also `assertNotNull`).
- `assertSame(Object expected, Object actual)`
 - ▶ Assert that *expected* and *actual* refer to the same object.
- `assertThrows(Class<T> expectedType, Executable executable)`
 - ▶ Assert that execution of the supplied *executable* throws an exception of the *expectedType* and return the exception.
- `fail()`
 - ▶ Fail the test without a failure message.

Exercise 1 – Visual Studio Code Basics, Software Quality and Tests ⁽¹⁾

Steps

1. Analyse the proposed code to understand the application logic of the implemented model (`example.model.*`), then run the application.
2. Analyse and run the proposed test (`SimpleBankAccountTest`).
3. Implement a new version of a bank account, allowing the deposit and the withdrawal also using the ATM. Each transaction done with the ATM implies paying a 1\$ fee.
 - ▶ The new bank account must implement the `BankAccount` interface and coded into a new class `SimpleBankAccountWithAtm`
 - ▶ It is requested to provide a new test class for the new bank account (`SimpleBankAccountWithAtmTest`)
4. Apply the DRY principle to refactor the written code, avoiding repetitions of code
 - ▶ This principle must be applied both to classes and tests.

¹for this exercise refer to [basic-refactoring-exercise module](#) 

Exercise 2 – TDD (²)

Step 1

- Following the TDD approach, provide an implementation for the `tdd.CircularList` interface.
 - ▶ see methods' documentation for details
- *Hints*
 1. Design a test for each method to be implemented for the `CircularList` following the order suggested in the provided interface
 - In some cases, e.g. to test the `next()` method, more than one test may improve the test suite
 2. Think about a simple way to keep the internal state of the list
 3. Think about corner cases as well: pose questions like “what if...?”

²for this exercise refer to `tdd-exercise` module

Exercise 2 – TDD

Step 2

- Create a new version of `CircularList` in a new package, fully adopting TDD
- Remove methods `next`, `previous`, and `clear`, replacing them with a `forwardIterator()` and `backwardIterator()`
- `forwardIterator()` returns a new `java.util.Iterator`, which yields next elements, circularly
- `backwardIterator()` returns a new `java.util.Iterator`, which yields previous elements, circularly

Exercise 2 – TDD

Step 3

- Create a new version of `CircularList` in a new package, fully adopting TDD
- Implement a `filteredNext(?)` method for your `CircularList` that finds and returns the next element that satisfies a given condition
- Choose an appropriate argument, seeking for good generality
- If no such element is found the method should return an empty `Optional`