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. 2024/2025

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-lab01`
- Open the project in IntelliJ IDEA
  - `File => Open` and select the **repository root folder**
  - You will find a project with two internal modules
- Develop all your exercises on that repo
- Try to commit frequently, at the right "moment" of the TDD process, with good commit messages
- We will communicate the modality for sending us your request for feedbacks: we will anyway look code in your repo

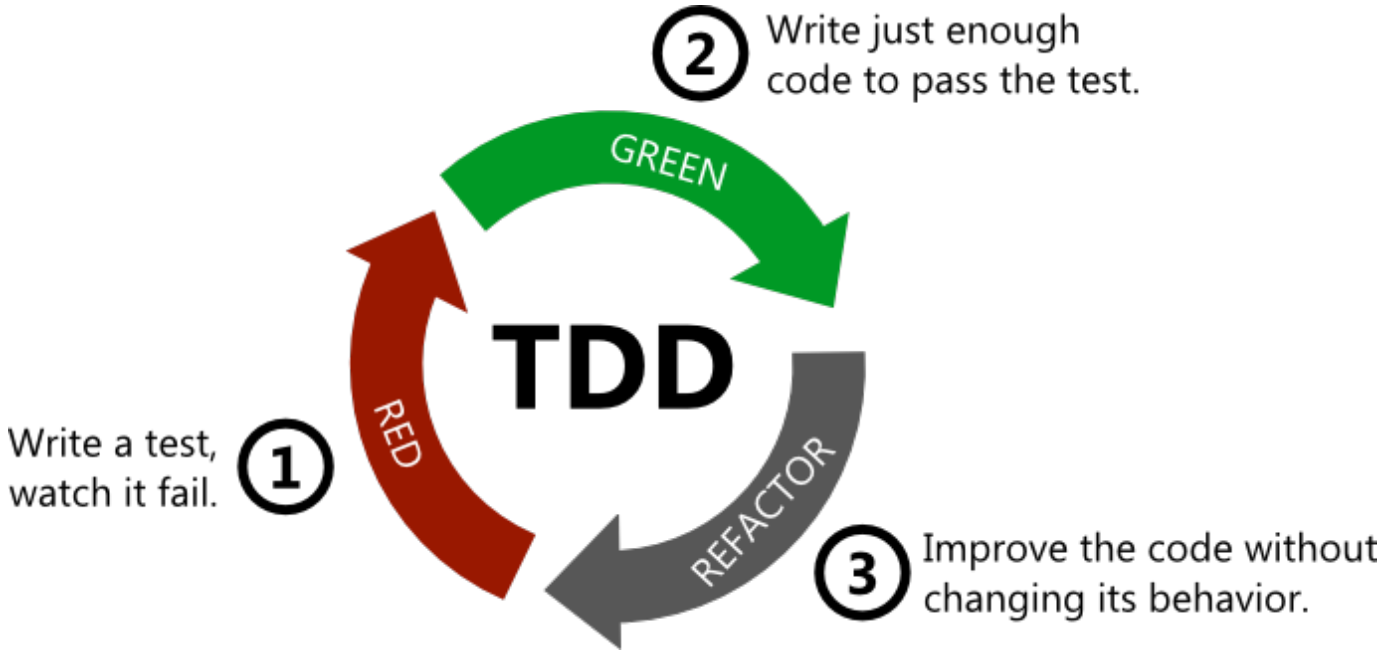
Software Quality Principles for this lab

- **DRY** – Don't Repeat Yourself
- **KISS** – Keep it simple, stupid
- Detect and avoid errors: rigidity, fragility, immobility, viscosity, opacity
- Strictly follow Java conventions
- Use good names for classes, methods, fields, variables
- Use small methods/classes

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 – IntelliJ Idea 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. Refactor and improve test and code following the proposed software quality principles.
4. Adjust the current solution introducing a withdrawal fee equal to 1$.
   - **NB!** Update tests and the implementation using TDD

[1] for this exercise refer to `basic-refactoring-exercise` module

Exercise 2 – TDD ([2])

Step 1

- Following the TDD approach, provide an implementation for the `tdd.SmartDoorLock` interface.
  - see methods' documentation for details
- *Hints*
  - Write tests for the `SmartDoorkLock` interface that verify complete user scenarios.
    - Focus on use cases rather than **isolated** method tests.
    - Design test plans that outline complete use-case scenarios, culminating in a final assertion. Build these tests **incrementally** while ensuring that edge cases are also addressed.
    - Consider both the blocked and locked states within realistic usage scenarios.

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

Exercise 2 – TDD

Step 2

- Implement also the `MinMaxStack` class, following the TDD approach
- *Hints*
  - As with the previous case, design tests that accurately reflect realistic user scenarios and interactions;
  - Think about a simple way to keep track of the minimum and maximum values in the stack
  - Think about corner cases as well: pose questions like "what if...?"
    - What if the stack is empty and you try to get the minimum value?

Exercise 2 – TDD

Step 3

- Finally, implement the whole concept of a `CircularQueue` following the TDD approach
  - In this case, just the description of the interface is provided, so you have to design also the methods' signatures from scratch
  - `CircularQueue` is a queue with a fixed size that, when full, starts overwriting the oldest elements — more details in the interface documentation
- *Hints*
  - Design tests that reflect realistic usage scenarios of the `CircularQueue` object as a whole.
  - Follow the TDD approach also in refining the methods of the `CircularQueue` interface
  - Think about a simple way to implement a circular queue (e.g., using a `List`)
  - Think about corner cases as well: pose questions like "what if...?"
    - What if the queue is full and you try to enqueue an element?