01 Lab
Software Quality and Test Driven Development (TDD)

Mirko Viroli, Roberto Casadei
{mirko.viroli,roby.casadei}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2020/2021

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-20-21-lab01`
- Open the project in IntelliJ IDEA
  - `File => Open` and select the **repository root folder**
  - You will find a project with two internal modules
- You must add JUnit 5 to the project, since it is specified as an **external dependency**
  a) Open a test class, move to a JUnit 5 symbol (which will be red, i.e., not resolved), and either click on the hint by the IDE, or press ALT+ENTER and select *Add JUnit '5.4' to classpath*
  b) Work in `File => Project structure => Modules => Dependencies`
- For any other errors
  - You may need to set the project SDK
    - `Setup SDK | Configure.. ==> + ("Add new SDK")` and select JDK
  - You may need to adjust the language level to enable Java features
    - `File => Project structure.. => Project`

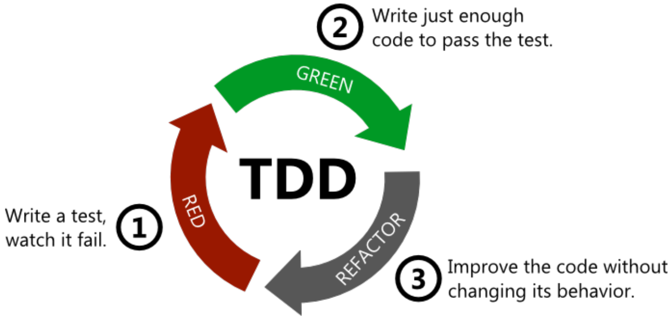
Software Quality Principles (recall)

- **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

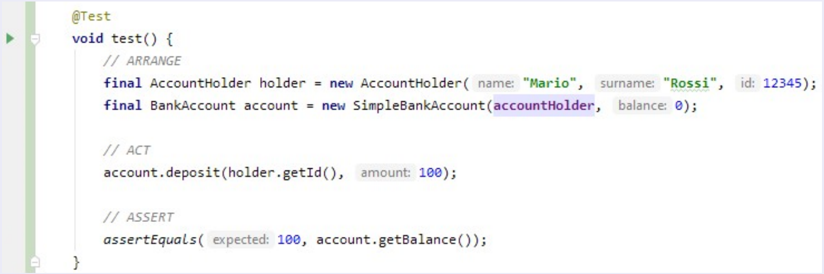
- TDD process: **Red-Green-Refactor** cycle
- TDD is about explicitly formalising (and enforcing) the "what" before the "how".
  - The term "test" is imprecise.
  - Your "JUnit code" serves different functions at different times. (Why?)


2 Write just enough code to pass the test.
GREEN
TDD
RED
Write a test, watch it fail. 1
REFACTOR
3 Improve the code without changing its behavior.

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
  - Note: in JUnit 5, unlike JUnit 4, this annotation does not defines any attributes
- `@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
  - Analogous to JUnit 4's `@Ignore`
- `@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.

JUnit 5+ (recall) (iii)

Assertions vs. Assumptions

- Assertions are used to write testing scenarios for test methods.
  - **If an assertion fails, the test fails.**
- Assumptions are used to specify test-case preconditions.
  - **If an assumptions fails, the test method is skipped.**

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

- `assumeFalse(boolean assumption)`
  - Validate the given *assumption*.
- `assumeTrue(boolean assumption)`
  - Validate the given *assumption*.
- `assumeThat(boolean assumption, Executable executable)`
  - Execute the supplied *executable*, but only if the supplied *assumption* is valid.

Exercise 1 – IntelliJ Basics, Software Quality and Tests ([1])

Steps

1. Analyse the proposed code to understand the application logic of the implemented model (`lab01.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 withdraw using also the ATM. Each transaction done with the ATM implies paying a 1$ fee.
   - The new bank account must implements 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.

[1] for this exercise refer to `pps-lab01-intellij-basic-example` module

Exercise 2 – TDD ([2])

Step 1

- Following the TDD approach, provide an implementation for the `lab01.tdd.CircularList` interface.
  - see methods' documentation for details
  - **for this step ignore the "next with strategy" method of the** interface
- *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...?"

[2] for this exercise refer to `pps-lab01-tdd` module

Exercise 2 – TDD

Step 2

- Implement the "next with strategy" method using the *Strategy Design Pattern*, adding for this purpose a dedicated test method to the suite
  - Note: a select strategy allows to get the next element of the circular list that satisfies the strategy.
- Each strategy must implements the `SelectStrategy` interface. Real Strategies that can be injected are:
  - *evenStrategy*, to get the next even element;
  - *multipleOfStrategy*, to get the next multiple of a given number;
  - *equalsStrategy*, to get the next equal element of a given one.

Exercise 2 – TDD

Step 3

- Consider software quality principles can be applied to the proposed solution and/or to the test suite implementation
  - e.g. DRY or KISS

Step 4

- Refactor the strategy implementation using the *Abstract Factory Pattern* to generate strategies