JUnit and Test Driven Development (TDD)

Gian Enrico Conti / Niccolò Izzo

Prova Finale - Ingegneria del Software - AA 2017/18

Unit testing

def:

A **unit test** is an automated piece of code that invokes a unit of work in the system and checks a **single assumption** about the behavior of that unit of work

Unit testing

- Unit of work:
  - a single method
  - a single class
  - multiple classes that work together (providing a single logical unit of work that should be verified)

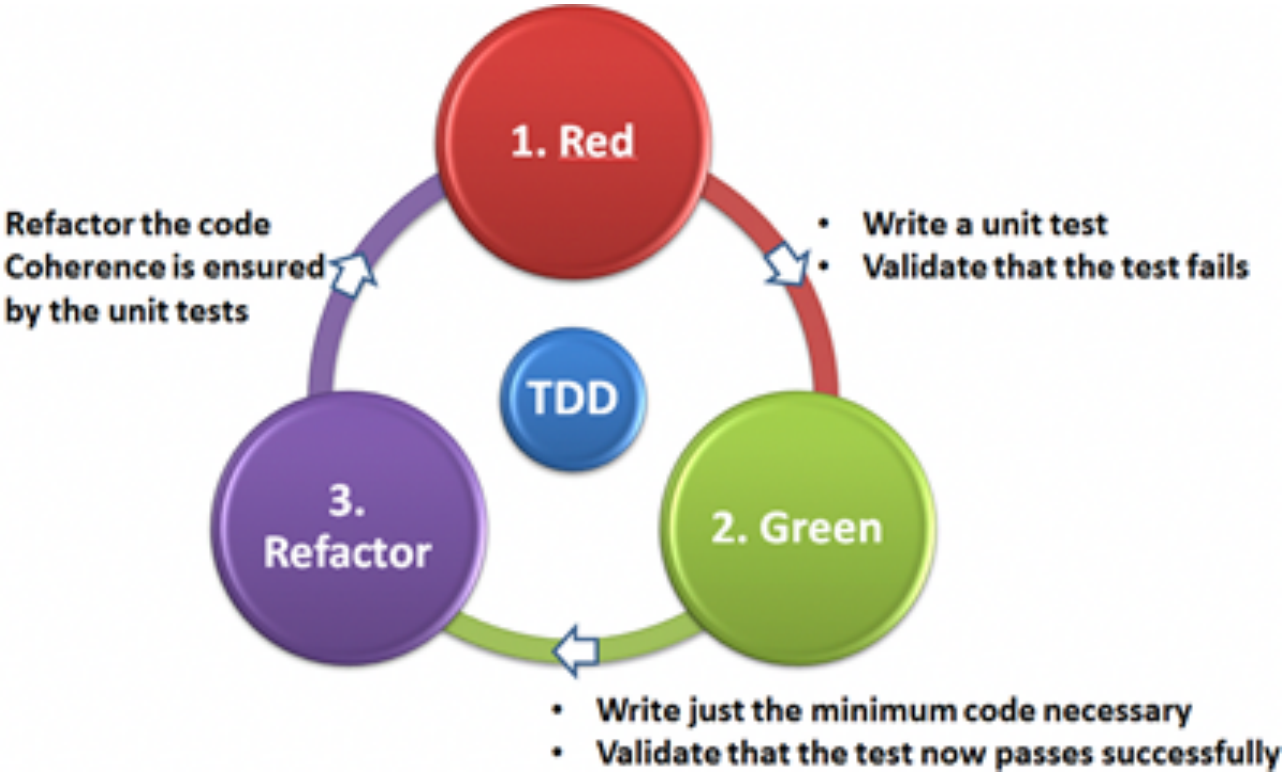
Unit testing

- The **%** of code that is tested by unit tests is called **test coverage**
- not suitable for testing complex user interface or component interaction (for that you have **integration tests**)

Unit test properties

- A unit test:
  - should be **fully automated**
  - should have a **known input** and an **expected output**
  - should **not provide side effects**
    - each test in a set of tests should run in **any order**
  - should return the **same result**
  - should run **fast and in memory**
  - should test a **single logical concept** of the application
  - should provide two tests for each requirement/functionality
    - **positive test**: checks the system with valid (expected) input data
    - **negative test**: checks the system with invalid (bad or unexpected) data
  -

Test Driven Development (TDD)



JUnit (1)

- JUnit is a unit testing framework for the Java programming language
- It allows to use Test Driven Development (TDD)

- Features
  - Fixtures (test start-up and shutdown + cleaning routines)
  - Test suites (the actual test code)
  - Test runners (the test executors)
  - JUnit Classes (Assert, TestCase, TestResult)

JUnit (2)

- Test Fixtures
  - fixed state of a set of objects used as a baseline for running tests.
  - **purpose**: ensure that the environment where tests take place is well-known and fixed
  - allow **repeatability** of tests
  - It includes:
    - *setUp()* method, executed *@Before* each test invocation
    - *tearDown()* method, executed *@After* each test invocation

- Test Suites
  - bundles of unit test cases that are run together
  - used by annotating class with:
    - *@RunWith(Suite.class)*
    - *@SuiteClasses(TestClass1.class, ...)*

Hands on - TDD

- Problem: Create a method that converts USD to CHF
- Base requirement: multiply two numbers together
  - define test case
  - let the test run (and **fail**)
  - provide the code to make the test **succeed**
  - refactor

Best practices

- Do not setup test case inside test constructor
- Do not rely on test execution order
- Do not write test cases with side effects
- Use relative paths to load files
- Bundle the test with all the needed data
- Choose meaningful test names
  - test class name starts always with Test (e.g. *TestClassUnderTest.java*)
  - test methods should describe what is tested
- Use *assert* and *fail* methods of Unit correctly
- Comment tests with Javadoc
- Keep test short and quick

Recommendations (1)

- Private methods
  - test it by means of a public method that makes use of it
  - other approaches possible but not suggested

- Do not test GUI
- Do not test networking
- Do not test CLI

- What should be tested?
  - Controversial topic.
  - **Our advice**: The more a piece of code is visible from outside the class, the more it has to be tested.

Recommendations (2)

- Test location
  - Unit tests are created in a separate project or separate source folder (to separate the real code from the tests)
  - Maven quickstart archetype set up a test folder for you in *src/test/java*

- Further readings
  - Official JUnit 5 Project page (https://junit.org/junit5/)
  - Junit tutorial (http://www.tutorialspoint.com/junit/index.htm)