Lecture on Testing technologies

Software Architecture and Methodologies - A.A. 2017/2018

Fulvio Patara, PhD
fulvio.patara@unifi.it - https://stlab.dinfo.unifi.it/patara/

Department of Information Engineering
University of Florence

15 May 2018
1. Introduction
   - What software testing is?
   - Why software testing?
   - Software testing levels
   - Software testing methodologies
   - Software testing types

2. JUnit framework
   - Basics
   - Live demo
   - References

3. Mockito framework
   - Basics
   - Live demo
   - References
What software testing is?

- A **software test** is a piece of code, which executes another piece of code.
- It validates if the code-under-test
  - results in the expected state → **State testing**
  - executes the expected sequence of events → **Behavior testing**
Why software testing?

- Assure that the logic of a piece of code is correct
- Support the identification of software regressions introduced by changes
- Enable automatic mechanisms of software verification
- ... and test coverage
  - Percentage of code covered by tests
  - High test coverage allows to continue developing features without having to perform (lots of) manual tests
Software testing levels

- Different stages of the software development lifecycle where testing is conducted

Software Development Process

- Client needs
- Requirements
- Designs
- Coding

Software Testing Levels

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
Software testing levels

▶ Unit testing
  ▶ Individual units of a software are tested to validate that they perform as designed

▶ Integration testing
  ▶ Individual units are *combined* and tested as a single unit with the aim of exposing faults in the *interaction between integrated units*

▶ System testing
  ▶ A *complete integrated* system is tested in order to evaluate the system’s compliance with respect to the specified requirements

▶ Acceptance testing
  ▶ The system is tested for *acceptability* with the purpose of assessing whether it is acceptable for delivery by users, customers or other authorized entities
Software testing methodologies

- Black-box testing
- White-box testing
- Gray-box testing
Black-box testing

Black box

Input → Black box → Output
Black-box testing

- To test the functionality of a system without peering into its internal structure/design/implementation
- The tester is aware of *what* the software is supposed to do
- ...is not aware of *how* it does it (no programming skills required) 😊
- Only a small number of possible inputs can be tested 😞
- Applicable to Integration, System and Acceptance levels
- Test design techniques
  - Equivalence partitioning
  - Boundary Value Analysis
  - Cause-Effect Graphing
  - Use case testing
  - User story testing
  - ...
White-box testing

[Diagram of white box model with input and output processes]
White-box testing

- To test internal structures or workings of a system
- The tester has access to the internal structure of the system
- ... determines all legal and illegal inputs
- ... verifies the outputs against the expected outcomes determined by studying the implementation code
- Testing is more thorough, with the possibility of covering most paths 😊
- Highly skilled resources are required, with a thorough knowledge of programming and implementation 😞
- Mainly applied to Unit testing
- Test design techniques
  - Code coverage (e.g. branch testing, path testing)
  - Fault injection methods
  - ...
Gray-box testing

Gray Box

Input

Output
Gray-box testing

- A combination of white-box testing (for designing test cases) and black-box testing (for testing)
- The tester *partially* knows the internal structure of the system (i.e., architectural view, not on source code or binaries)
- Primarily used in Integration testing
- Test design techniques
  - Matrix Testing
  - Regression testing
  - Pattern Testing
  - Orthogonal array testing
Software testing types

- **Functional testing**
  - The system is tested against the functional requirements/specifications

- **Usability testing**
  - To evaluate how easy to use the system is by testing it with real users

- **Performance testing**
  - To determine how a system performs in terms of responsiveness and stability under a certain load

- **Regression testing**
  - To ensure that changes (enhancements or defect fixes) to the software have not adversely affected it

- **Security testing**
  - To uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders
JUnit
JUnit framework

- A unit testing framework for the Java programming language
- One of a family of unit testing frameworks collectively known as xUnit
- Open source project on Github (latest stable version: 5.2.0)
- (JUnit Plugin for Eclipse)
A **Unit test** executes a specific functionality in the code under test

- Typically, a Unit test targets a **small unit of code**
  - A method or a class
- External dependencies should be removed from Unit tests
  - By replacing the dependency with a test implementation or a mock object created by a test framework (e.g., Mockito)

- **Avoid to test trivial code**
  - Getter/setter methods
  - Code already covered by external tests
  - ...
Test cases and test suites

- Tests are organized in **Test cases**
- Test cases can be organized in **Test suites**
  - `@RunWith(Suite.class)`
  - `@Suite.SuiteClasses({. . . })`
Annotations [JUnit 4.x/JUnit 5.x]

- Annotations are used to identify/configure methods in a test case
  - @Test – Identifies a method as a test method
  - @Before/@BeforeEach – Executed before each test to prepare the environment
  - @After/@AfterEach – Executed after each test to cleanup the environment
  - @BeforeClass/@BeforeAll – Executed once, before the start of all tests, to perform time intensive activities
  - @AfterClass/@AfterAll – Executed once, after all tests have been finished, to perform clean-up activities
  - @Ignore/@Disable – Marks that the test should be disable
  - @Test (expected = Exception.class) – Fails if the method does not throw the named exception
  - @Test(timeout=100) – Fails if the method takes longer than 100 milliseconds
 Assert methods are used to check an expected result vs. the actual result

- `fail(message)` – Marks a test that is incomplete
- `assertTrue([message,] boolean condition)` – Checks that the boolean condition is true
- `assertFalse([message,] boolean condition)`
- `assertEquals([message,] expected, actual)` – Checks that two values are the same
- `assertEquals([message,] expected, actual, tolerance)` – Checks that float or double values match
- `assertNull([message,] object)` – Checks that the object is null
- `assertNotNull([message,] object)`
- `assertSame([message,] expected, actual)` – Checks that both variables refer to the same object
- `assertNotSame([message,] expected, actual)`
The default order of execution of JUnit tests within a class is deterministic but not predictable.

It is recommended that test methods be written so that they are independent of the order that they are executed.

In the case of dependent tests, `@FixMethodOrder` allows the user to choose the order of execution of the methods within a test class.

- e.g., `@FixMethodOrder(MethodSorters.NAME_ASCENDING)`
Parameterized tests

- When running a parameterized test class, instances are created for the cross-product of the test methods and the test data elements
  - @RunWith(Parameterized.class)
  - One static method annotated with the @Parameters annotation
    - The method generates and returns a collection of arrays
    - Each item is used as parameter for the test method
  - @Parameter annotation on public fields to get the test values injected in the test
Rules

- Rules (@Rule) allow very flexible addition or redefinition of the behavior of each test method in a test class

- Some base rules
  - TemporaryFolder - To create files/folders to be deleted when the test method completes
  - ExternalResource - To setup a resource before a test and tear it down afterward
  - ErrorCollector - To continue the execution of a test after the first problem is found
  - Verifier - To turn otherwise passing test methods into failing tests if a verification check is failed
  - ExpectedException - To allow in-test specification of expected exception types and messages
  - ...
Success vs. Error vs. Failure

- A **success** occurs when the test case completed successfully
  - All test criteria were satisfied → **JUnit green bar**

- An **error** is an unexpected behavior that occurs while trying to actually run the test (i.e. unhandled/unforeseen exceptions)
  - The test case crashed without executing fully → **JUnit red bar**

- A **failure** occurs when the test criteria are not met
  - The test case completed successfully but assertions were not satisfied → **JUnit gray bar**
Live demo
References

- https://junit.org/junit4/
- https://junit.org/junit5/
- https://junit.org/junit5/docs/current/user-guide/
Mockito framework

- **Mock framework** for the **Java** programming language
- Open source project on Github (latest stable version: **2.18.5**)
- Allows the creation of **mock objects** at runtime in automated unit tests in order to support **testing isolation**
- Usually used in conjunction with **JUnit**
A **test double** is an object that can stand in for a real object
- **Dummy** - A placeholder passed around but never actually used
- **Fake** - An object with working simplified implementation, usually takes some shortcut which makes it not suitable for production (e.g., in-memory database)
- **Stub** - Provides canned answers (no logic) to calls made during the test, usually not responding at all to anything outside what’s programmed in for the test
- **Spy** - A stub ables to record some information based on how it was called
- **Mock** - An object pre-programmed with expectations which form a specification of the calls it is expected to receive

Only mocks insist upon **behavior verification**

The other doubles can, and usually do, use **state verification**

---

1. [https://martinfowler.com/articles/mocksArentStubs.html](https://martinfowler.com/articles/mocksArentStubs.html)
Mock objects

- A **dummy implementation** for an interface or a class
- Configured to perform a certain behavior during a test
  - e.g. Return a certain output for certain method calls
- Passed to other objects which are tested
- Guarantee that tests are not affected by any side effects
Creating Mock objects

- Several methods to create Mock objects
  - Using the static `mock(ClassToBeMocked.class)` method
  - Using the `@Mock` annotation
    + the `@RunWith(MockitoJUnitRunner.class)`
      to trigger the creation of annotated objects
    - Alternatively, using the static
      `MockitoAnnotations.initMocks(this)` method
Configuring Mock objects

- Mocks can return different values depending on arguments passed into a method
  - The `when(methodCallWithPredefinedParameters).thenReturn(returnValue)` method chain can return
    - a single value
    - multiple values
    - a single value from specified arguments
    - a single value from generic arguments using methods like `AnyString()` or `AnyInt()`
  - Unspecified method calls return empty values
  - Similarly, the `when(methodCallWithPredefinedParameters).thenThrow(exception)` method chain can throw a specified exception
Creating Spy objects

- To partially mock on an existing instance
  - Part of the object will be mocked and part will use real method invocations

- Several methods to create Spy objects
  - Using the static `spy(ClassToBeSpied.class)` method
  - Using the `@Spy` annotation
    + the `@RunWith(MockitoJUnitRunner.class)` to trigger the creation of annotated objects
      - Alternatively, using the static `MockitoAnnotations.initMocks(this)` method

- Need to use the `doReturn(returnValue).when(mock).methodCall()` call chain that works similar to `when(…).thenReturn(…)`
Behavior testing using `verify()`

- **Behavior testing** does not check the result of a method call but it checks that a method is called with the right parameters.

- `verify(mock).methodToCheck(params)`

- `verify()` can be enriched by:
  - `never()` - Method never called
  - `atLeastOne()` - Method called at least one time
  - `atLeast(numberOfTimes)` - Method called at least \# times
  - `atMost(numberOfTimes)` - Method called at most \# times
  - `times(numberOfTimes)` - Method called \# times
Using `@InjectMocks` Mockito tries to inject mocks only either
- by constructor injection
- by setter injection
- or by property injection

Mocks will first be resolved by type
- If there is several property of the same type, mocks will be resolved by the match of the property name and the mock name

If any strategy fails, provide injection by yourself!
- `FieldUtils.writeField(Object target, String fieldName, Object value, boolean forceAccess)`
References

- http://site.mockito.org/
- https://github.com/mockito/mockito