Best Practice in Developing Java Applications

Unit Tests

Dott. Marco Solinas, PhD

12 Nov 2021

Unit Tests

Agenda

- What is and what isn’t a Unit Test
- Unit Tests among testing phases
- Benefits and Principles
- Best Practices
- Common Approaches
- Adoption
- Working with JUnit
- Mocking
- Working with Mockito

Unit Tests
What is a Unit Test?

A unit test (UT) is a procedure to verify that a particular module of source code is working properly (behaves as expected).

Note: a UT does not verify that the system works properly as a whole

Unit tests:

- are written and run by developers for all classes and methods to:
  - quickly fix a regression introduced by a code change
  - ensure that code meets its design and behaves as intended
- are live specifications from developer to developer
- are a safety net for refactoring
- allow to test all the minimum details and edge cases
- are close to the code

Unit Tests

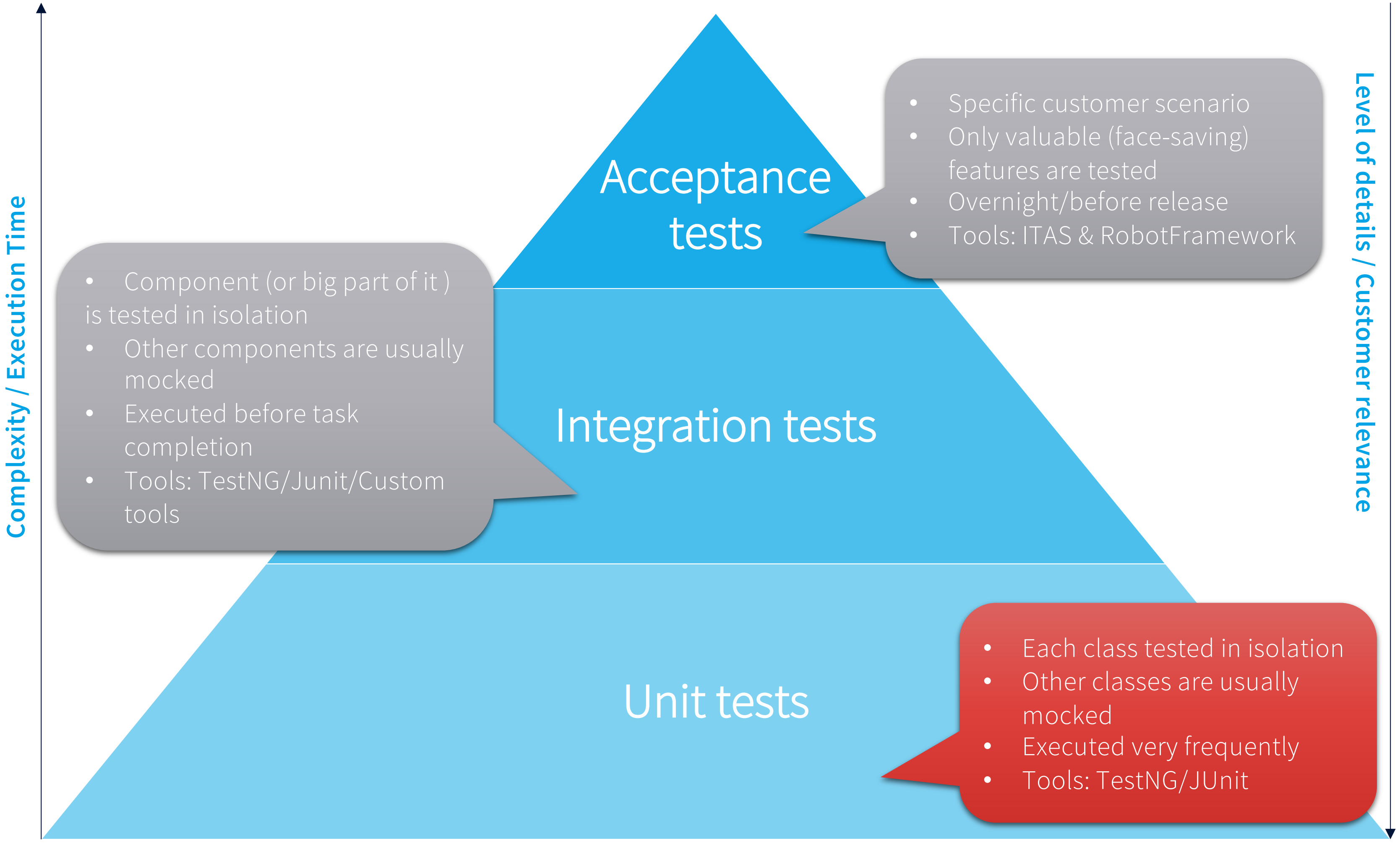
What isn’t a Unit Test?

- A test is not a unit test if:
    - it communicates across the network
    - it touches the file system
    - it talks to the database
    - it runs threads
    - forces you to do special things to your environment (such as editing config files) to run it

- Rationale:
    - Speed
    - Fragility
    - We want to test our code (behaviors), not the third-party one!

Unit Tests

Testing phases



Complexity / Execution Time

Level of details / Customer relevance

Acceptance tests

- Specific customer scenario
- Only valuable (face-saving) features are tested
- Overnight/before release
- Tools: ITAS & RobotFramework

Integration tests

- Component (or big part of it) is tested in isolation
- Other components are usually mocked
- Executed before task completion
- Tools: TestNG/Junit/Custom tools

Unit tests

- Each class tested in isolation
- Other classes are usually mocked
- Executed very frequently
- Tools: TestNG/JUnit

Unit Tests

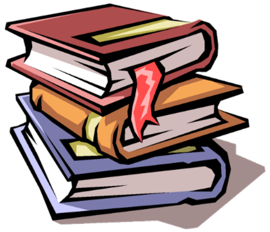
What are the benefits?

- Immediate feedback on bug introduction
  - Readily-available tests make it easy to check whether a piece of code is still working properly
- Facilitates changes
  - Allow for refactoring
  - Performance optimization
- Drive and improve the design
  - Promotes low coupling and high cohesion
  - Promotes OOP principles
- Provide living documentation
- Sustained maintenance for complex environments
  - Real world example (ION 2.0 Framework):
  - ~148k lines, ~3.7k classes
  - ~8k tests, ~71% code coverage

Unit Tests

Principles

- Unit Test should be mainly automatic
- Test run must be fast
- Each software module shall be tested in isolation
- Each test shall not rely on any other test, nor should it depend on tests being run in a specific order
- Test code should be designed, maintained and re-factored as the production code



Unit Tests

Best Practices (1)

- One class, (at least) one fixture
  - Add a fixture for each class you add to the project
  - For different scenarios, use different fixtures

⚠ `<ClassName>Test (FooTest)`

- One test, one assert
  - Don’t use more than one assert in each test (Single Responsibility Principle)
  - Write many tests against the same class method

⚠ `test<NameOfTheTest> (testSaveFile)`

- Each test must be short
  - A test shouldn’t be longer than 10 lines of code

- Test name must clearly specify the goal of the test
  - Don’t use name generic names, use long verbose names

⚠ `public void GIVEN_author_names_available_WHEN_search_by_prefix_THEN_books_matching_startWith_returned()`

Unit Tests

Best Practices (2)

- Tests must be easy to maintain
  - Use the DRY principle (Don’t Repeat Yourself)
  - Use the KISS principle (Keep It Simple and Stupid)
- Tests must be simple
  - Possibly avoid complex logic (if, switch, for and so on)
- Avoid dependencies between tests
  - A test should be able to stand on its own
  - Result must not depend on tests being run in a specific order
  - Avoid statics and singletons
- Tests must be fast and automatic
  - Avoid special (manual) configurations and lengthy operations

Unit Tests
Best Practices (3)

- Test boundary cases
  - Numbers: 0, positive, negative, infinity, NaN, etc.
  - Strings: empty, single-char, etc.
  - Dates: Jan 1, Feb 29, Dec 31, etc.
  - Collections: empty, one element, first, last, etc.
  - Exceptions thrown
- Write a test for a bug
- Cover the code properly
  - Don’t leave production code uncovered (apart GUI stuff, 3rd party libs).

Unit Tests

Best Practices (4)

- You break it, you fix it
  - Who breaks a test is also responsible to fix it, as soon as possible

- Tests must be easily readable
  - The most important one. A test is a documentation about a specific feature!

Unit Tests

Common Approaches (1)

- First approach:
  - Code, code, code for some days
  - Then test, test, test for some days

Wrong

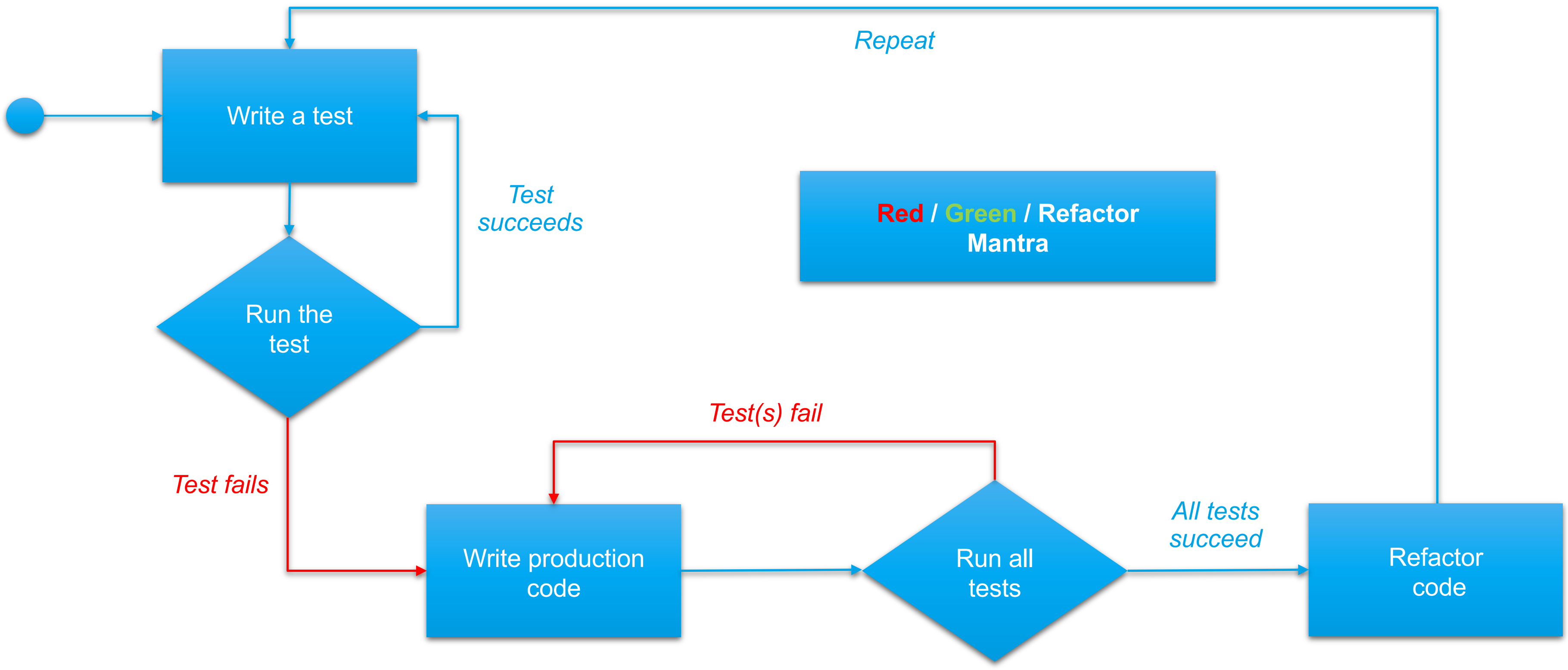
- Cons:
  - Expensive and hard
  - Lost a lot of benefits

- Better approach: 👍
  - Code a little, test a little, code a little, test a little…

- Benefits:
  - Spot immediately design problems (you wear the hat of the user too!)
  - Recognize early unused/not necessary code
  - Intercept errors as soon as possible!

Common Approaches (2)

- Optimal approach (Test Driven Development, or TDD):
  - Write test first, and then the minimum amount of production code to let the test passes



Unit Tests
Adoption

- New developments
  - No new code shall be written with no associated test

- Existing code
  - Start writing/refactoring tests when you
    - Add a new feature
    - Fix a bug
    - Perform refactoring

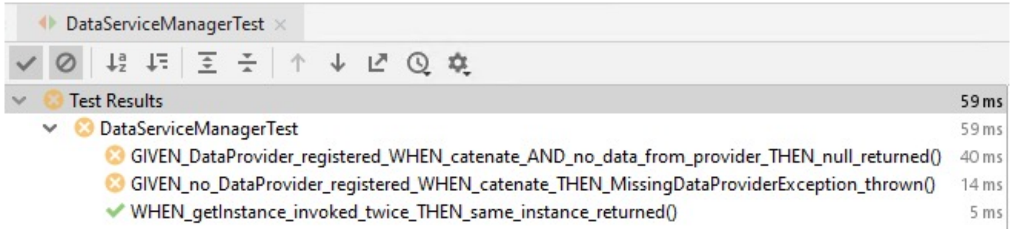
- Goal
  - Incrementally adapt existing test without stopping the normal developments

Unit Tests

Working with JUnit (1)

JUnit is a unit testing framework for the Java programming language

- Open source
- Provides annotations to identify test methods
  - @Test
  - @BeforeEach, @BeforeAll
  - @AfterEach, @AfterAll
- Provides assertions for checking expected results
  - Assertions.assertEquals (1, sut.processAndReturnInteger(“p”));
  - Assertions.assertTrue(sut.isInitialized());
- Can be run automatically and provides immediate feedback after checking the expectations
- Exposes the concept of *test suite*



```
Test Results                                                                                          59 ms
  DataServiceManagerTest                                                                              59 ms
    GIVEN_DataProvider_registered_WHEN_catenate_AND_no_data_from_provider_THEN_null_returned()       40 ms
    GIVEN_no_DataProvider_registered_WHEN_catenate_THEN_MissingDataProviderException_thrown()         14 ms
    ✓ WHEN_getInstance_invoked_twice_THEN_same_instance_returned()                                     5 ms
```

Unit Tests

Working with JUnit (2)

It’s a set of dependencies to be added in pom files

Parent POM

```
<properties>
    <junit.platform.version>1.6.2</junit.platform.version>
    <junit.version>5.6.2</junit.version>
</properties>

<dependencyManagement>
    <dependencies>
        … other dependencies not scoped for tests, e.g. guava
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-commons</artifactId>
            <version>${junit.platform.version}</version>
            <scope>test</scope>
        </dependency>
...
```

```
...
        <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>junit-platform-engine</artifactId>
            <version>${junit.platform.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Module POM

```
<dependencies>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
    </dependency>
</dependencies>
```

Unit Tests

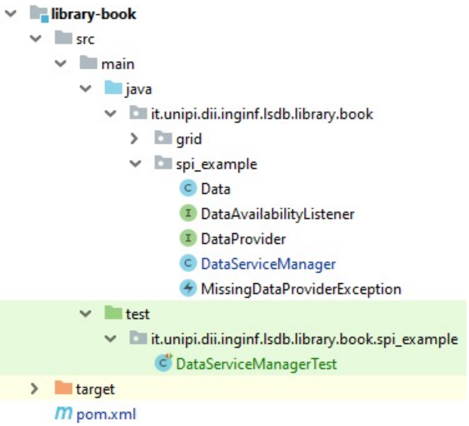
Working with JUnit (3)

package scope to grant access to test suite

```java
package it.unipi.dii.inginf.lsdb.library.book.spi_example;

import …

public class DataServiceManager {

    private static DataServiceManager instance;

    private DataProvider dataProvider;

    @VisibleForTesting
    DataServiceManager(){}

    public static DataServiceManager getInstance() {  // it's a singleton instance
        if (instance == null) {
            instance = new DataServiceManager();
        }
        return instance;
    }

    public void registerDataProvider(DataProvider dataProvider) {
        this.dataProvider = dataProvider;
    }

    public Data catenate(String dataId1, String dataId2) throws MissingDataProviderException { … }
}
```

VisibleForTesting annotation on constructor because we increased its scope (it was private before)

```java
package it.unipi.dii.inginf.lsdb.library.book.spi_example;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class DataServiceManagerTest {

    private DataServiceManager sut;

    @BeforeEach
    public void init() {
        sut = new DataServiceManager();
    }

    @Test
    public void WHEN_getInstance_invoked_twice_THEN_same_instance_returned() {
        DataServiceManager instance = DataServiceManager.getInstance();
        Assertions.assertEquals(instance, DataServiceManager.getInstance());
    }

    @Test
    public void GIVEN_no_DataProvider_registered_WHEN_catenate_THEN_MissingDataProviderException_thrown() {
        Assertions.assertThrows(MissingDataProviderException.class, () -> {
            sut.catenate("","");
        });
    }

}
```

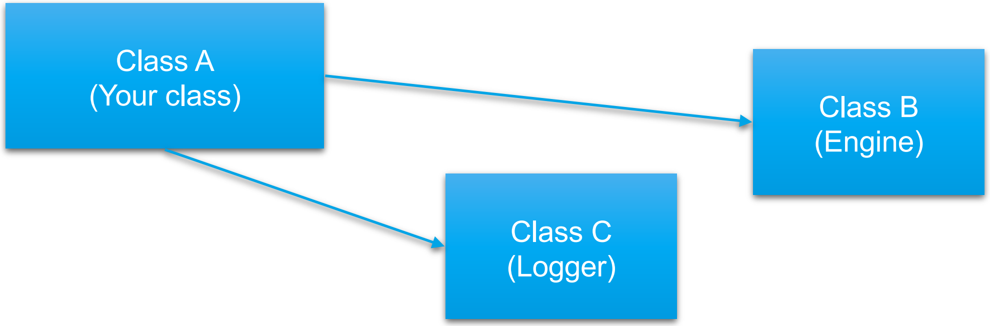


```
library-book
  src
    main
      java
        it.unipi.dii.inginf.lsdb.library.book
          grid
          spi_example
            Data
            DataAvailabilityListener
            DataProvider
            DataServiceManager
            MissingDataProviderException
    test
      it.unipi.dii.inginf.lsdb.library.book.spi_example
        DataServiceManagerTest
  target
  pom.xml
```

Unit Tests

Mocking (1)

- A mock is a substitute implementation to emulate domain code
- A mock must be used when a class:
  - is expensive or impractical to instantiate
  - belongs to a third-party library
  - Accesses the network / file system / database



Class A (Your class)

Class B (Engine)

Class C (Logger)

Unit Tests

Mocking (2)

- A mock object must:
  - be simpler than the real code
  - not contain any kind of logic
  - allow to setup private state
  - be stupid!

- The emphasis in mock implementations is on absolute simplicity, rather than completeness

- Available approaches:
  - Manual mocking
  - Mocking frameworks

Unit Tests

Manual mocking (1)

- Refactor the above class in the following way:
  - wrap the dependency (Engine) using a proper interface
  - provide a default implementation using the concrete type
  - provide a mock implementation using the interface
  - allow dependency injection for the new interface
  - eventually, provide a default initialization in the constructor

```
public class DummyEngineStarter {
   public void start() {
        Engine.getInstance().start();
   }
}
```

Unit Tests

Manual mocking (2)

```
public interface IEngine {
   void start();
}

public class EngineImpl implements IEngine {
   public void start() {
      Engine.getInstance().start();
   }
}

public class MockEngine implements IEngine {
   boolean started = false;

   public void start() {
      started = true;
   }

   public boolean wasStarted() {
      return started;
   }
}
```

```
public class DummyEngineStarter {
   IEngine eng;
   public DummyEngineStarter(IEngine eng) {
      this.eng = eng;
   }
   public start() {
      eng.start();
   }
}

@Test
public void testStartLaunchesEngine() {
   // GIVEN
   MockEngine mockEng = new MockEngine();
   DummyEngineStarter starter = new
        DummyEngineStarter(mockEng);

   // WHEN
   starter.start();

   // THEN
   assertTrue(mockEng.wasStarted());
}
```

Unit Tests
Mocking frameworks

- Sometimes manual mocking is not feasible:
  - you cannot modify the “bad” class (legacy code)
  - you cannot even subclass the “bad” class (final class or simply impractical)

- In such cases a solution is provided by frameworks like Mockito
- Pros:
  - it allows to mock interfaces/classes
  - easy to use
- Cons:
  - tests become more fragile
  - it requires deeper knowledge of the code under test
  - lower test readability
  - does not encourage the sharing of mocks in the team (everyone tends to have their own mocks)

Unit Tests

Working with Mockito (1)

Mockito is a mocking framework, JAVA-based library that is used for effective unit testing of JAVA applications

- Used to mock the deps of a system under test
- Provides annotations to declare mocks
  - @Mock private DataProvider mockDataProvider;
- Provides static methods to specify the behavior of a mock
  - Mockito.when(mockDataProvider.getDataById(Mockito.anyString())).thenReturn(new Data(“value”));
- The main benefit is that you do not have to write your mocks manually

Unit Tests

Working with Mockito (2)

It’s a set of dependencies to be added in pom files

Parent POM

```
<properties>
    <junit.platform.version>1.6.2</junit.platform.version>
    <junit.version>5.6.2</junit.version>
    <mockito.version>3.0.0</mockito.version>
</properties>

<dependencyManagement>
    <dependencies>
        … other dependencies not scoped for tests, e.g. guava
        … other dependencies scoped for tests, e.g. junit
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-junit-jupiter</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Module POM

```
<dependencies>
    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
    </dependency>
</dependencies>
```

Unit Tests

Working with Mockito (3)

```java
public class DataServiceManager {

    private DataProvider dataProvider;
    // ...
    public void registerDataProvider(DataProvider dataProvider) {
        this.dataProvider = dataProvider;
    }

    public Data catenate(String dataId1, String dataId2) throws MissingDataProviderException {
        if (dataProvider != null) {
            Data data1 = dataProvider.getDataById(dataId1);
            Data data2 = dataProvider.getDataById(dataId2);

            Data resultData = null;
            if (data1 != null && data2 != null) {
                resultData = doCatenate(data1, data2);
            }
            return resultData;
        }
        throw new MissingDataProviderException();
    }
}
```

```java
public interface DataProvider {
    Data getDataById(String dataId);
}
```

```java
package it.unipi.dii.inginf.lsdb.library.book.spi_example;

import …;

public class DataServiceManagerTest {

    @Mock
    private DataProvider mockDataProvider;
    private DataServiceManager sut;

    // ...

    @Test
    public void GIVEN_DataProvider_registered_AND_no_data_available_WHEN_catenate_THEN_null_returned() throws MissingDataProviderException {
        // GIVEN
        Mockito.when(mockDataProvider.getDataById(Mockito.anyString())).thenReturn(null);
        sut.registerDataProvider(mockDataProvider);

        // WHEN
        Data actual = sut.catenate("id1", "id2");

        // THEN
        Data expected = null;
        Assertions.assertEquals(expected, actual);
    }

    @BeforeEach
    public void init() {
        MockitoAnnotations.initMocks(this);
        sut = new DataServiceManager();
    }
...
```

```java
...
    @Test
    public void GIVEN_DataProvider_registered_AND_data_available_WHEN_catenate_THEN_data_values_are_concatenated()
                                                                                    throws MissingDataProviderException {
        // GIVEN
        Mockito.when(mockDataProvider.getDataById("id1")).thenReturn(new Data("value1"));
        Mockito.when(mockDataProvider.getDataById("id2")).thenReturn(new Data("value2"));
        sut.registerDataProvider(mockDataProvider);

        // WHEN
        Data actual = sut.catenate("id1", "id2");

        // THEN
        Data expected = new Data("value1value2");
        Assertions.assertNotNull(actual);
        Assertions.assertEquals(expected.getValue(), actual.getValue());
    }
}
```