Unit test crash course

Andrea Ferranti
Francesco Rubino

07/02/2023

Wolters Kluwer

Agenda

First part:

- Theory
- JUnit 5 & Live coding
- Best practices

Second part:

- Test doubles
- Mockito
- Live coding

Why Unit Test matters?

- Automatable
- Replicable
- Virtually error-free (unlike manual ones)
- Enable and facilitate refactoring
- “Document” behaviors

Unit test

- Unit: class or method
- Allow to quickly identify the cause of regression
- Executed at build time (=> must be fast!)
- Classes compiled but not included in the jar

Definitions

- Code coverage: % automatically tested code
- State vs Behavior testing
  - State: verification of the result
  - Behavior: verifies that a defined sequence of operations has been performed

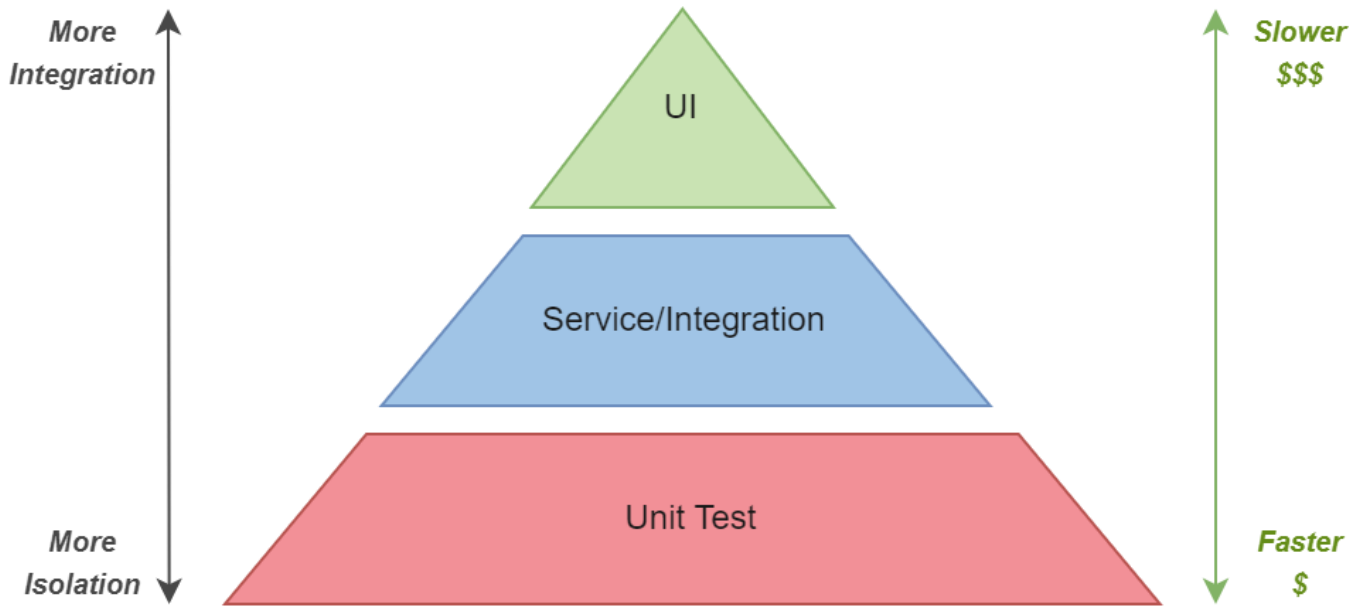
Integration Test

- Also called "Functional Test"
- Combination of modules already tested with unit test

System Test

- Test of entire system
- Manual or Automatic
  - Sikuli
  - Selenium

Test pyramid



JUnit

- Erich Gamma (Gang of Four)
- Standard methodology for making assertions and delivering results in a structured manner
- Integrated with build system and IDE
- JUnit 5
  - 2017
  - Requires Java 8+

JUnit 5 – Annotation

```
1 @Test
2 public void method() { /* … */ }
```

Arbitrary order!

```
1 @BeforeEach
2 public void method() { /* … */ }
```

- Executed before each test
- Used to initialize state

```
1 @AfterEach
2 public void method() { /* … */ }
```

- Executed after each test
- Used to reset state (i.e. remove temporary file)

JUnit 5 – Annotation

```
1 @BeforeAll
2 public static void method() { /* … */ }
```

- Static!
- Run once before all test methods

```
1 @AfterAll
2 public static void method() { /* … */ }
```

- Static
- Run once after all test methods

JUnit 5 - Annotation

```
1 @Disabled("Reason")
2 public class TestClass { /* … */ }
```

- Disable a test class

```
1 @Disabled("Reason")
2 public void method() { /* … */ }
```

- Disable a test method

JUnit 5 - Assertion

- Static method of the Assertions class
- Used to compare the value returned by the object under test with an expected value
- Allows to specify a message describing the eventual failure
- Allows checking for exceptions (assertThrows)

JUnit 5 - Assertion

- `fail("Message")`
- `assertTrue(boolean condition, ["Message"])`
- `assertFalse(boolean condition, ["Message"])`
- `assertEquals(expected, actual, ["Message"])`
- `assertEquals(expected, actual, delta, ["Message"])`
- `assertArrayEquals(expected, actual, ["Message"])`
- `assertNull(object, ["Message"])`
- `assertNotNull(object, ["Message"])`
- `assertSame(expected, actual, ["Message"])`
- `assertNotSame(expected, actual, ["Message"])`
- `assertThrows(Class<T> expectedType, Executable executable)`
- `assertTimeout(Duration duration, Executable executable)`

JUnit 5 – Example Calculator

`Calculator` class implements a simplified version of a calculator: only addition and multiplication are allowed

JUnit 5 – Example ATM

- `Calculator` is simple example! It's a stateless object!
- Let's try to introduce the "state": in this case the object to be tested must be initialized and I must be able to verify that, in a certain instant, its state is the expected one.

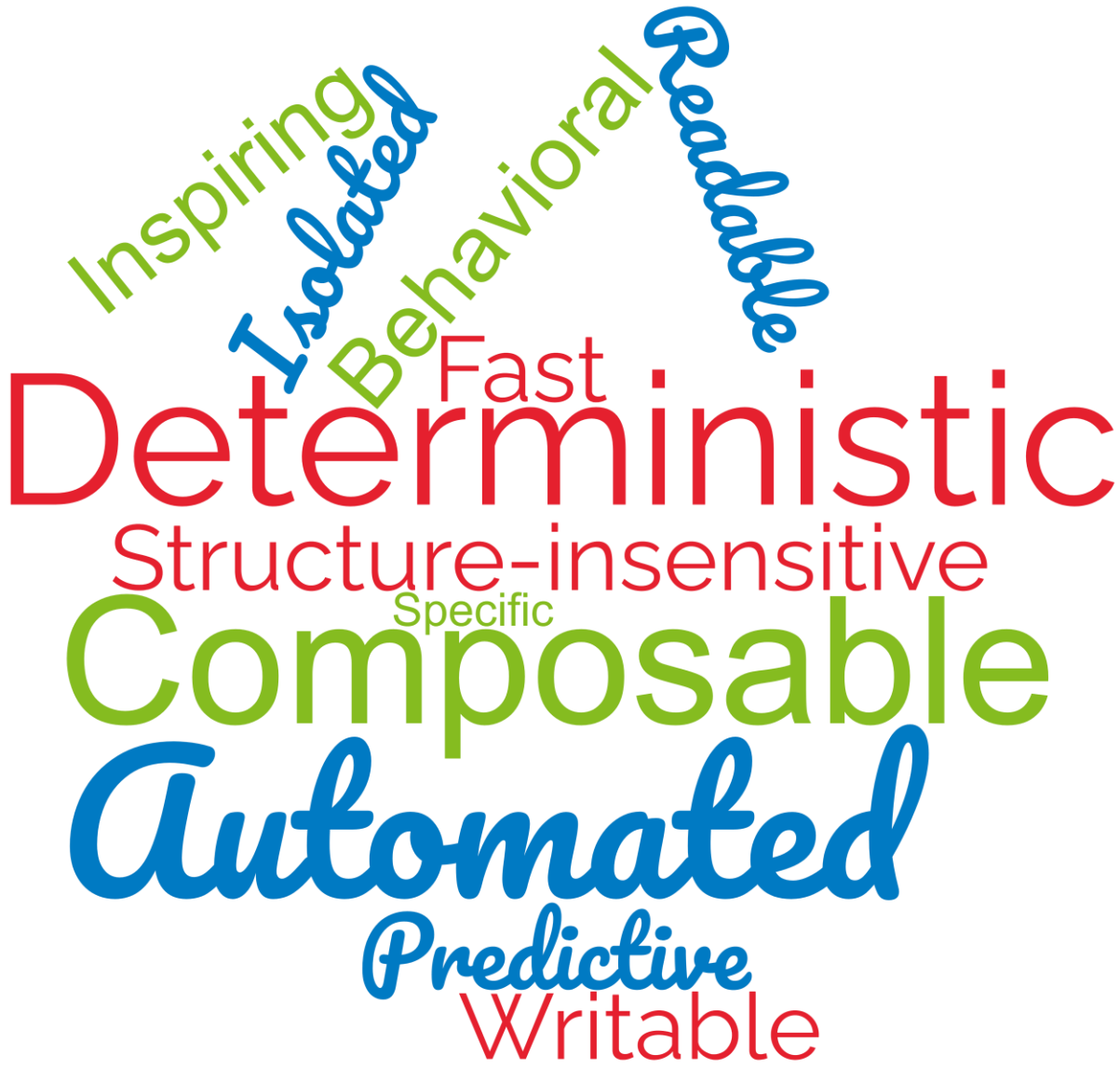
JUnit 5 – Example ATM

- `AtmImpl` implements `Atm` and represents an ATM.
- The ATM can be recharged using the `deposit` method
- Withdrawals are made with the `withdraw` method If you try to withdraw more money than there is in the ATM, an exception is raised

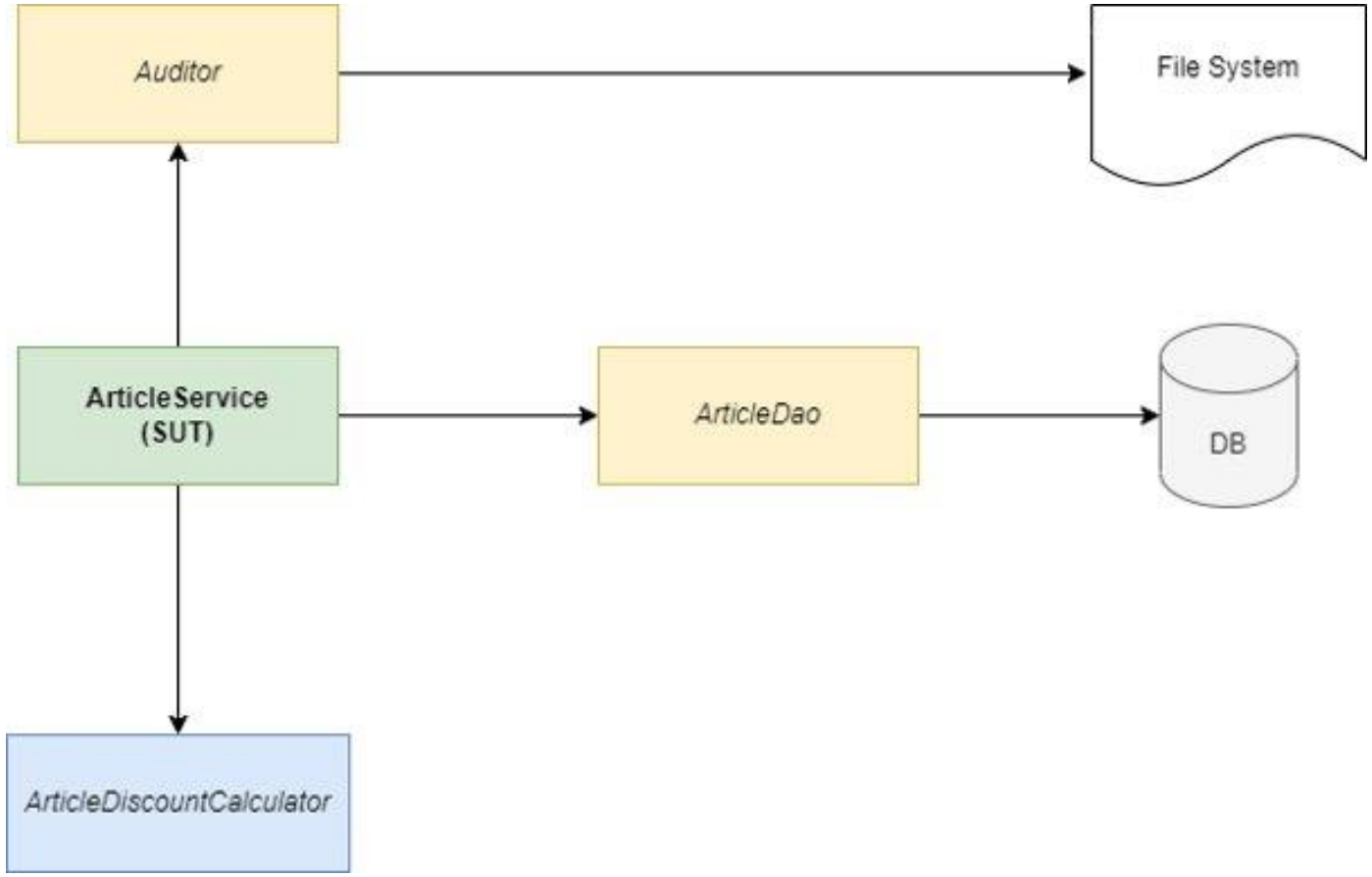
JUnit 5 – Recap

- The structure of a test follows a more or less standard pattern:
  - An initial state is defined
  - The test object is stimulated
  - The results (status or sequence of events produced) are verified
- In literature
  - Given-when-then
  - Four-phase test (setup, exercise, verify, teardown)
  - AAA (Arrange, Act, Assert).

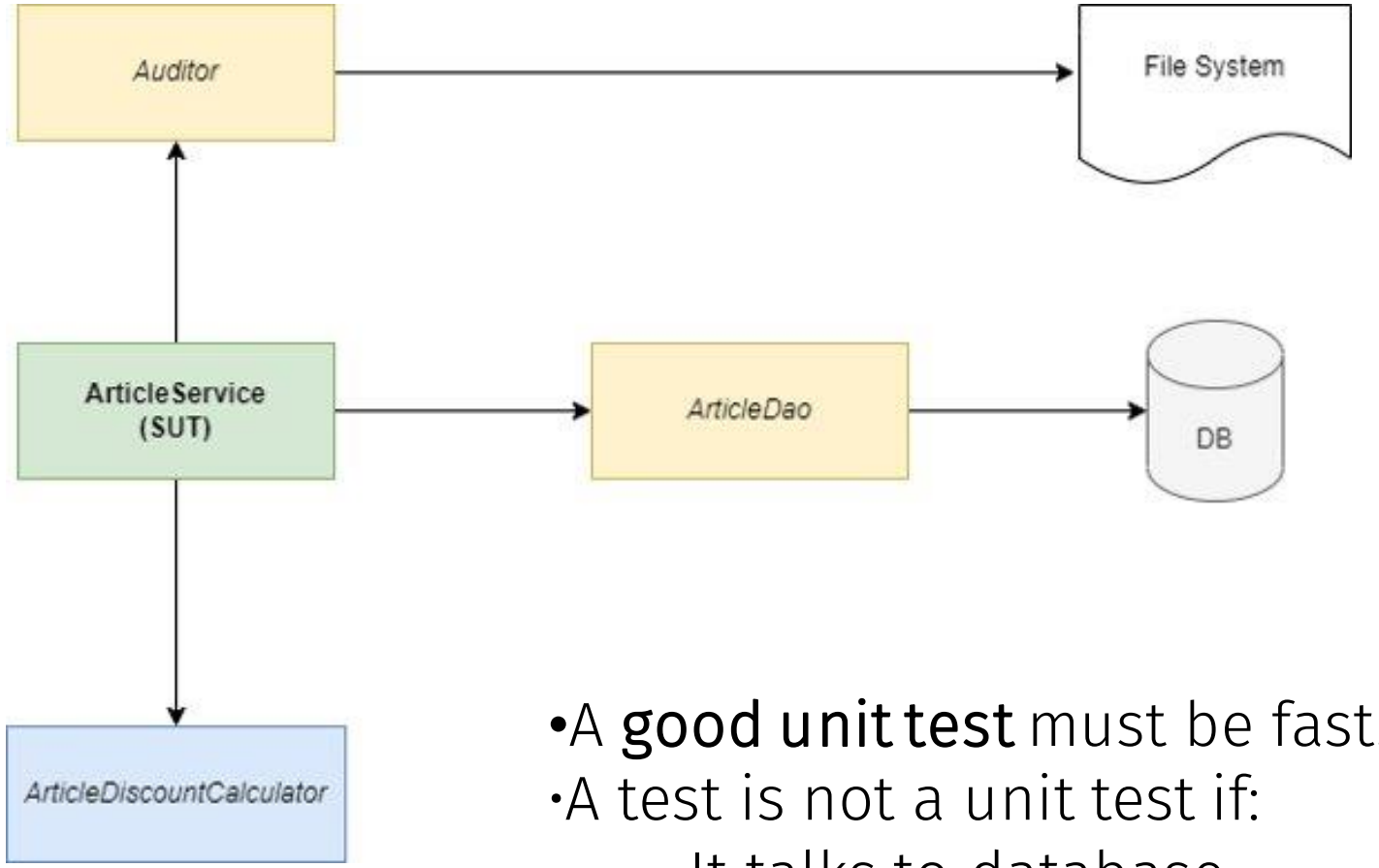
JUnit 5 – Best Practices

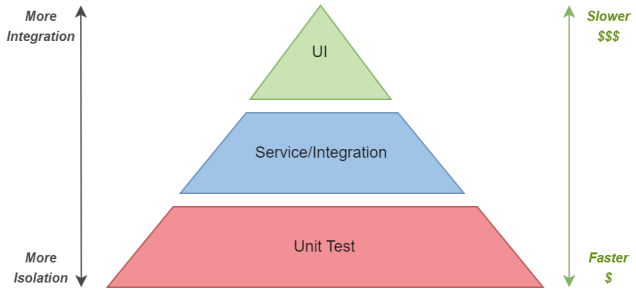


JUnit and
… collaborators



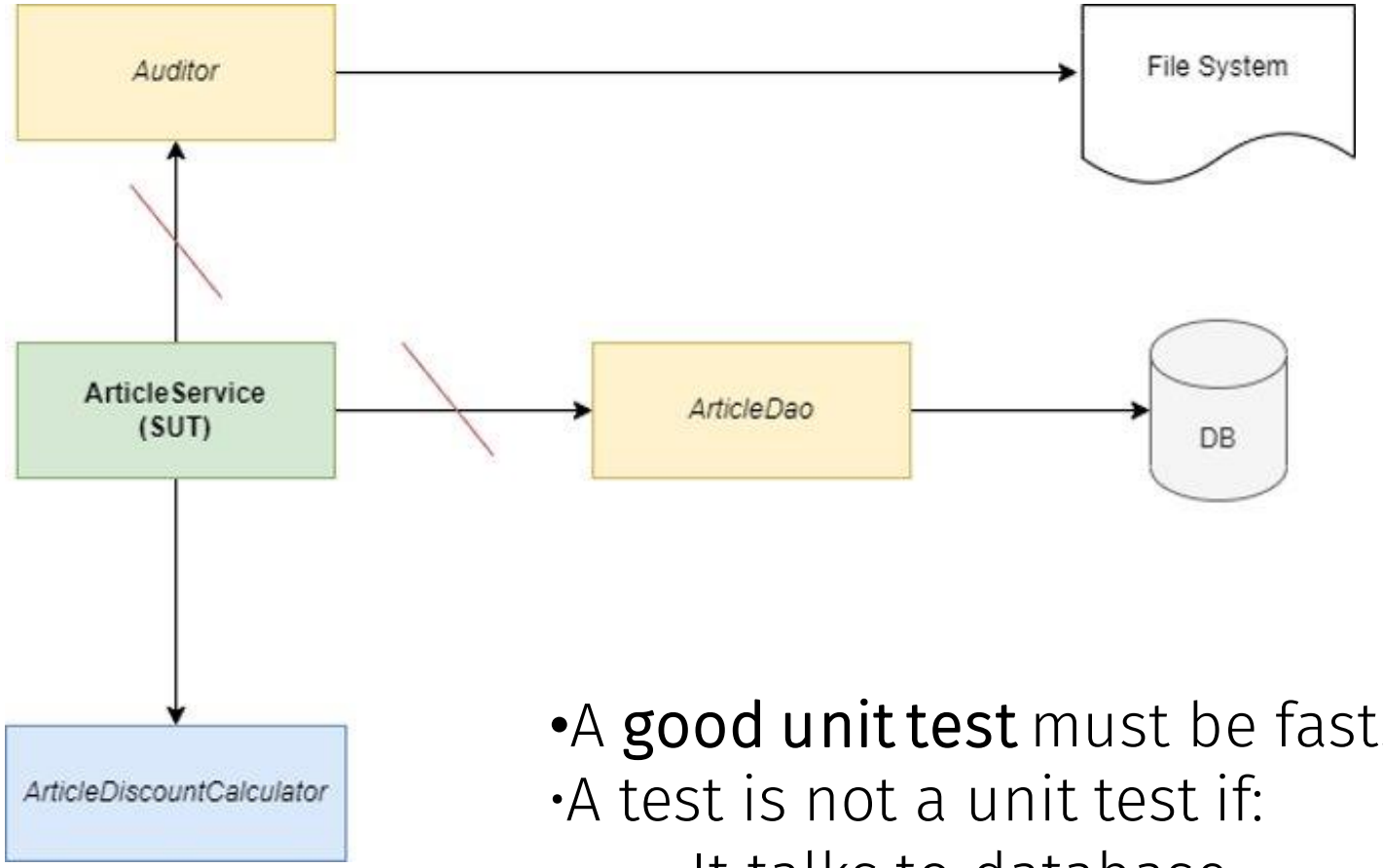
JUnit and
… collaborators

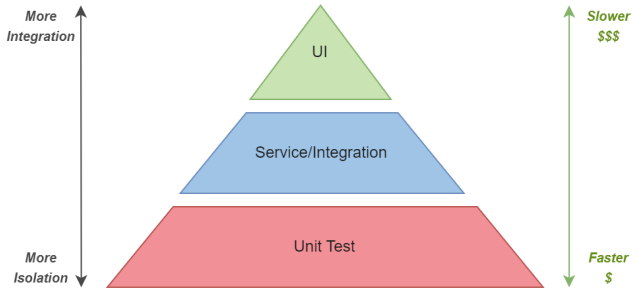




- A **good unit test** must be fast!
- A test is not a unit test if:
  - It talks to database
  - It touches the file system

JUnit and
… collaborators



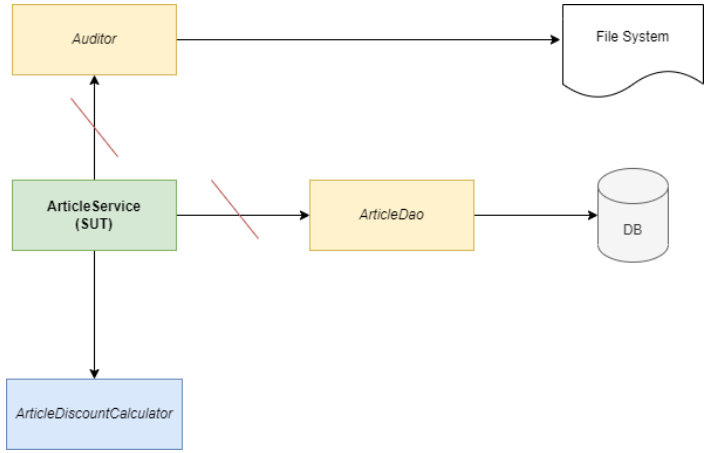


- A **good unit test** must be fast!
- A test is not a unit test if:
  - It talks to database
  - It touches the file system

Test doubles

- Useful for testing classes with many collaborators
- A Double allow us to break the dependency, helping to isolate the code and avoiding external operations, such as:
    - Access to the database
    - Communication across a network
    - Access to file system

- Test doubles can be:
    - Dummy
    - Fake
    - Stub
    - Mock



Test doubles - Dummy

- It's the simplest one
- It is just an object that you pass to satisfy a constructor and the API in general
- It will not have any method implemented and it shouldn't.

```java
public interface Auditor {
  void audit(String message);
}

public class ArticleServiceImpl implements ArticleService {
  private final Auditor auditor;

  public ArticleServiceImpl(Auditor auditor) {
    this.auditor = auditor;
  }

  public double getPriceOf(List<Integer> articleIds) {
    for (int i = 0; i < articleIds.size(); i++) {
      auditor.audit("Reading article with id: " + articleIds.get(i));
      // ...
    }
  }
}

public class AuditorDummy implements Auditor {

    @Override
    public void audit(String message) {
      // Do nothing!
    }
  }
```

Test doubles – Stub

- It provide pre-defined answers for methods of our collaborators
- They still don't have any logic
- Useful to write deterministic and repeatable tests

```java
public interface ArticleDao {
  Article findById(Integer id);
}


public class ArticleServiceImpl implements ArticleService {
  private final Auditor auditor;
  private final ArticleDao articleDao;

  public ArticleServiceImpl(Auditor auditor, ArticleDao articleDao) {
    this.auditor = auditor;
    this.articleDao = articleDao;
  }

  public double getPriceOf(List<Integer> articleIds) {
    for (int i = 0; i < articleIds.size(); i++) {
      Integer articleId = articleIds.get(i);
      Article article = articleDao.findById(articleId);
      auditor.audit("Reading article with id " + articleId + " and price " + article.getPrice());
      // ...
    }
  }
}

public class ArticleDaoStub implements ArticleDao {
  private Article article = new Article(1, "MyArticle", 10.5d );
  @Override
  public Article findById(Integer id) {
    return article;
  }
}
```

Test doubles – Fake

- Stub with a simplified version of the business logic
- Different behaviours based on the provided data provided (input parameters)

```
public class ArticleDaoFake implements ArticleDao {

    @Override
    public Article findById(Integer id) {
      if (id == 1){
        return new Article(1, "My Article", 10.50d);
      } else {
        return new Article(2, "Another Article", 20.5d);
      }
    }
  }
```

Test doubles – Mock

- Pre-programmed objects based on the input parameters
- Set your expectations of the collaborator method…
  - … call the method to test …
  - … and verify if the collaborator method is called at the end.

- Useful libraries
  - Mockito
  - EasyMock

Mockito

- Mockito is a mock framework to use in conjunction with JUnit.
- Allows you to create and configure mock objects to simplify the development of tests for classes with (external?) dependencies.

Mockito – How to create and inject a mock

- Using the static method Mockito.mock(<class>)

```
public class ArticleServiceTest {
  private ArticleService sut;

  @BeforeEach
  public void setUp(){
    ArticleDao articleDao = Mockito.mock(ArticleDao.class);
    this.sut = new ArticleService(articleDao);
  }

}
```

- Or using the annotation @Mock

```
public class ArticleServiceTest {
  private ArticleService sut;
  @Mock
  private ArticleDao articleDao;

  @BeforeEach
  public void setUp(){
    this.sut = new ArticleService(articleDao);
  }

}
```

Mockito – How to extend Junit5

- Use the annotation @ExtendWith(MockitoExtension.class)

```
@ExtendedWith(MockitoExtension.class)
public class ArticleServiceTest {

  private ArticleService sut;
  @Mock
  private ArticleDao articleDao;
}
```

- `@ExtendedWith` annotation on test class allows to specify the extension class of a test

Mockito – How to create a mock

- Mockito cannot mock/spy following:
  - final classes
  - anonymous classes
  - primitive types
  - no arrays
  - no String

```
org.mockito.exceptions.base.MockitoException:
Cannot mock/spy class com.tagetik.junitcrashcourse.mockito.service.internal.ArticleServiceImplTest$MyFinalClass
Mockito cannot mock/spy because :
 - final class
```

Mockito – How to configure a mock

Use it to **configure** a mock to return a particular value when a particular method is called.

```
Mockito.when(<method call>)
  .thenReturn(T value)
  .thenAnswer(Answer<?> answer)
  .thenThrow(Class<? extends Throwable> exception);
```

"*When the x method with certain parameters is called then return y*"

Mockito – How to verify a mock

Use it to **verify** that a particular method of a mock with particular arguments is called (**Behaviour Testing**)

```
Mockito.verify(T mock, VerificationMode? mode)
  .someMethod("arg1", "arg2", ...);
```

- VerificationMode can be:
  - times(x): verify that a method is called x times
  - never(): verify that a method is never called
  - Default: times(1)

- Mockito.verifyNoInteractions(T mock)
  - Verify that mock is never used

Mockito - Tips

- Do not mock types you don’t own
- Don’t mock value objects
- Don’t mock everything
- Show love with your tests!

Thank you!

Wolters Kluwer