

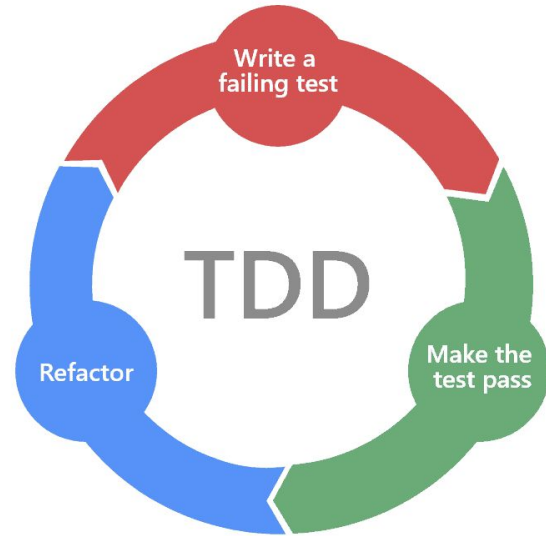


# Applied Java

Daniel Lewandowski, Andrzej Kałuża

# Test Driven Development

- software development process
- repetition of a very short development cycle
  - add test cases as requirement implementation
  - improve code so that the tests pass
  - repeat
- focuses more on the implementation of a feature
- written in a language similar to the one used for feature development
- unit tests





## TDD anti - patterns

- test cases depend on system state manipulated from previously executed test cases
- dependencies between test cases
- execution order should not be presumed
- interdependent tests can cause cascading false negative
- testing precise execution behavior timing or performance
- testing implementation details
- slow running tests
- building "all-knowing oracles"
- implementation focusing on tests (test - code coupling)



# TDD patterns

Unit tests - typically automated tests written and run by software developers to correct behaviour of a section of an application (known as the "unit"):

- easy to write
- readable
- reliable
- fast
- truly unit, not integration



# Acceptance Test Driven Development

- development methodology
- communication between the business customers, the developers, and the testers
- specification by example
- understanding the customer's needs prior to implementation
- customers express requirements domain language
- a single acceptance test is written from the user's perspective
- writing acceptance tests

```
Given Credentials entered in a login form
```

```
And User registered on the system
```

```
When User checks an account balance
```

```
Then List of transactions is printed
```



# Behavior Driven Development

- software development process
- encourages collaboration among developers, QA and non-technical or business participants
- motivates to formalize a shared understanding of how the application should behave
- understanding requirements

**Feature:** Subscribers see different sets of stock images based on their subscription level

**Scenario:** Free subscribers see only the free articles

**Given** Free Frieda has a free subscription

**When** Free Frieda logs in with her valid credentials

**Then** she sees a Free article on the home page

**Scenario:** Subscriber with a paid subscription can access both free and paid articles

**Given** Paid Patty has a basic-level paid subscription

**When** Paid Patty logs in with her valid credentials

**Then** she sees a Free article and a Paid article on the home page



# JUnit

- simple framework to write repeatable tests
- instance of the xUnit architecture
  - Test runner - an executable program that runs tests and reports the test results
  - Test case - the most elemental class executes a unit with specified arguments
  - Test fixtures - the set of preconditions or state needed to run a test
  - Test suites - a set of tests that all share the same fixture
  - Test execution - an individual unit test proceeds as follows:
    - setup
    - specific code
    - teardown
  - Test result formatter - produces results in one or more output formats
  - Assertions - a function or macro that verifies the behavior (or the state) of the unit under test

Test class contains  
all unit tests for  
Calculator class

@Test annotation  
marks a method as  
a unit test

assertEquals checks equality of  
expected value and actual value.  
Raises `AssertionFailedError`  
exception with a given message

@DisplayName overwrites a  
message identifying a unit  
tests

```
class CalculatorTests {

    @Test
    @DisplayName("1 + 1 = 2")
    void addsTwoNumbers() {
        Calculator calculator = new Calculator();
        assertEquals( expected: 2, calculator.add( a: 1, b: 1), message: "1 + 1 should equal 2");
    }

    @ParameterizedTest(name = "{0} + {1} = {2}")
    @CsvSource({
        "0, 1, 1",
        "1, 2, 3",
        "49, 51, 100",
        "1, 100, 101"
    })
    void add(int first, int second, int expectedResult) {
        Calculator calculator = new Calculator();
        assertEquals(expectedResult, calculator.add(first, second),
            () -> first + " + " + second + " should equal " + expectedResult);
    }
}
```

Test Results	73 ms
✓ CalculatorTests	73 ms
✓ 1 + 1 = 2	34 ms
✓ add(int, int, int)	39 ms
✓ 0 + 1 = 1	35 ms
✓ 1 + 2 = 3	2 ms
✓ 49 + 51 = 100	1 ms
✓ 1 + 100 = 101	1 ms



JUnit 5 Test Results

Test Results	73 ms
CalculatorTests	73 ms
1 + 1 = 2	34 ms
add(int, int, int)	39 ms
0 + 1 = 1	35 ms
1 + 2 = 3	2 ms
49 + 51 = 100	1 ms
1 + 100 = 101	1 ms

@ParametrizedTest marks a unit test to be executed multiple times with different parameters

A parameterized unit test with arguments for a function evaluation and a result verification

```
class CalculatorTests {

    @Test
    @DisplayName("1 + 1 = 2")
    void addsTwoNumbers() {
        Calculator calculator = new Calculator();
        assertEquals( expected: 2, calculator.add( a: 1, b: 1), message: "1 + 1 should equal 2");
    }

    @ParameterizedTest(name = "{0} + {1} = {2}")
    @CsvSource({
        "0, 1, 1",
        "1, 2, 3",
        "49, 51, 100",
        "1, 100, 101"
    })
    void add(int first, int second, int expectedResult) {
        Calculator calculator = new Calculator();
        assertEquals(expectedResult, calculator.add(first, second),
            () -> first + " + " + second + " should equal " + expectedResult);
    }
}
```

@CsvSource defines an array of the parameters applied to an unit test

An assertion with an automatically generated message

# An exercise

---

Regular test class

Regular test class

Mocked object have the same methods as a list. It remembers what and how is used.

```
public class MockitoExampleTest {  
    @Test  
    void simpleMockito() {  
        //mock creation  
        List<String> mockedList = mock(List.class);  
  
        //using mock object  
        mockedList.add("one");  
        mockedList.clear();  
  
        //verification  
        verify(mockedList).add("one");  
        verify(mockedList).clear();  
    }  
}
```

Mocked object pretending to be an instance of a list

It can be verified how a mocked class was used.

```
public class StubTest {  
    @Test  
    void simpleStub() {  
        //mock creation  
        List<String> mockedList = mock(List.class);  
  
        //stubbing using built-in anyInt() argument matcher  
        when(mockedList.get(anyInt())).thenReturn("element 0");  
  
        //following prints "element 0"  
        System.out.println(mockedList.get(999));  
  
        mockedList.add("element 2");  
  
        //you can also verify using an argument matcher  
        verify(mockedList).get(999);  
  
        //argument matchers can also be written as Java 8 Lambdas  
        verify(mockedList).add(argThat(someString → someString.length() > 5));  
    }  
}
```

Verification if get method of mocked object was called with an integer value 999.

Mocked object pretending to be an instance of a list.

Mocked object returns "element 0" string when get methods is called with any integer value.

A bit more advanced example using custom argument matcher.

# An exercise

---



# References

- <https://en.wikipedia.org/>
- <https://docs.oracle.com/en/java/>
-