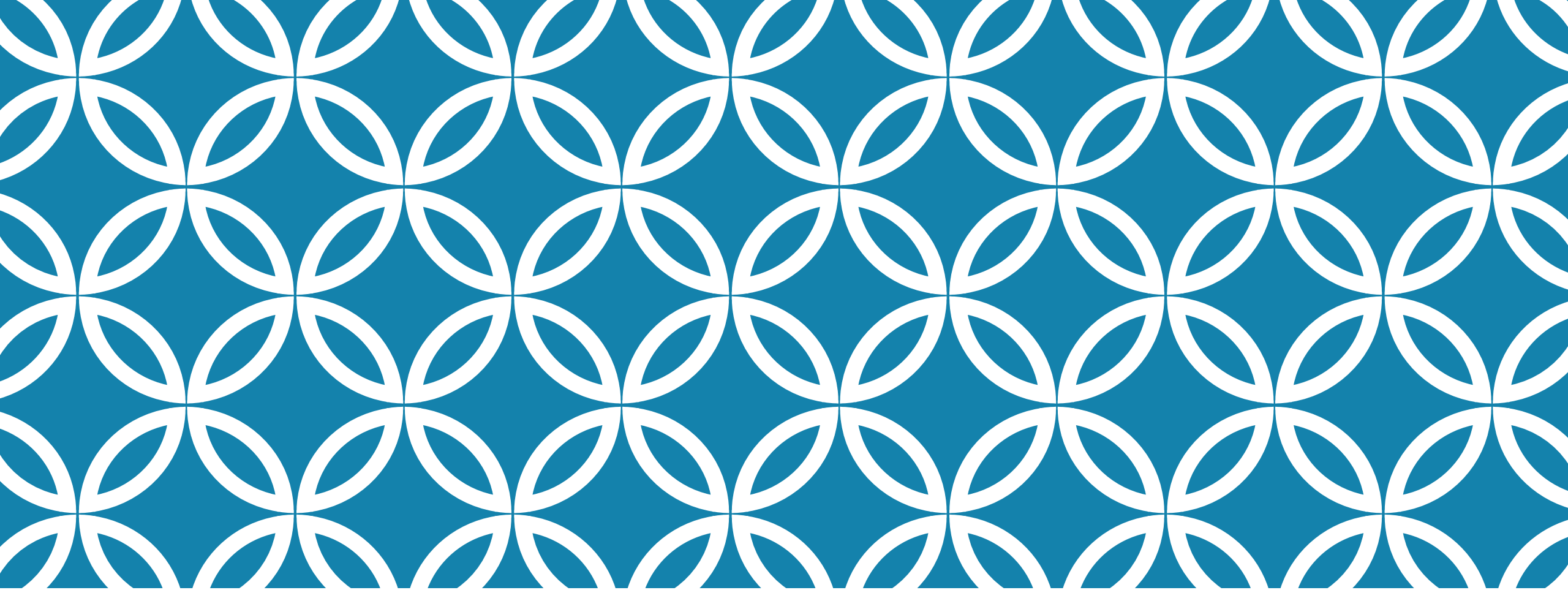
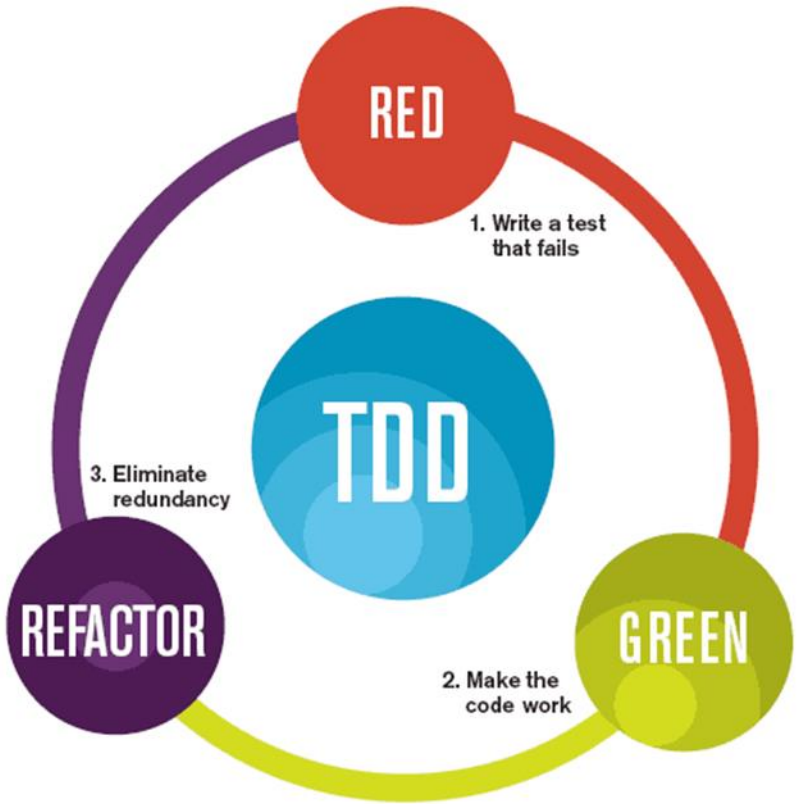


TESTOWANIE JEDNOSTKOWE

JUnit
AssertJ

TEST DRIVEN DEVELOPMENT

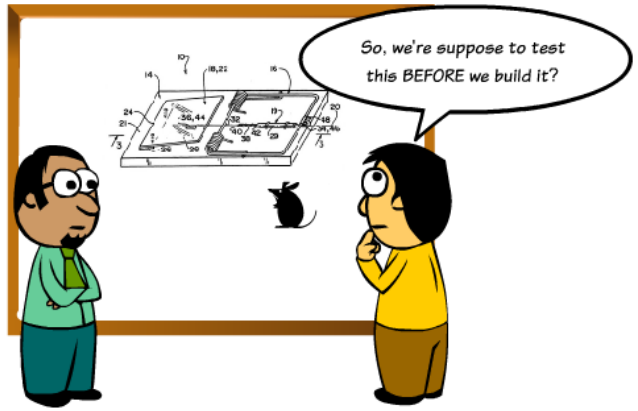
- Testing is cool
- Testing makes Your life easier
- Testing makes You work faster
- Testing makes Your code better
- Unit testing vs. Integration Testing
- Write test then code
- Writing good tests is hard and takes practice



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

UNIT TESTING – F.I.R.S.T

- **Fast**
  - milliseconds per test
  - running after any change
  - run multiple times in reasonable time
- **Isolated**
  - tests one thing/feature
  - clear error message
  - no dependencies with external resources (files/databases)
  - has no influence on other tests
- **Repeatable**
  - the same result every time
  - no initial state
  - running order has no impact on results
- **Self-validating**
  - gives simple answer: test passes vs test fails
  - no need to do additional results analysis
- **Timely**
  - prepare in good time
  - prepare just before start coding given feature (method)



WHAT SUPPORTS (OR NOT) UNIT TESTS

Structures that support testing:

- composition over inheritance
- small classes
- small methods
- dependency injections
- using interfaces

Anti-patterns for unit tests:

- singleton pattern
- static methods
- final elements

JUNIT – SIMPLE EXAMPLE

```java
public class CalculatorTest {

  @Test
  public void shouldAddTwoNumbers() {
    //given
    Calculator sut = new Calculator();
    //when
    int result = sut.add(1, 2);
    //then
    assertEquals(3, result);
  }
}
```

JUNIT – INITIALIZATION AND CLEANING

```
public class RemoteServerIntegrationTest {

  @BeforeClass
  public static void initClass() {
    startEmbeddedTomcatServer(); //raz przed pierwszym testem
  }
  @Before
  public void init() {
    createRequiredCollectionsInMongoDB(); //przed każdym testem
  }

  @After
  public void tearDown() {
    cleanupCollectionsInMongoDB(); //po każdym teście
  }
  @AfterClass
  public static void tearDownClass() {
    shutdownEmbeddedTomcatServer(); //raz po ostatnim teście
  }

  @Test
  public void shouldDoSomething() { //... }
  @Test
  public void shouldDoSomethingElse() { //... }
}
```

JUNIT – BASIC ASSERTIONS

- `assertEquals(String expected, String actual)`
- `assertEquals(String message, String expected, String actual)`
- `...`
- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
- `assertNull(boolean condition)`
- `assertNotNull(boolean condition)`
- `assertTrue(String message, boolean condition)`
- `...`

JUNIT – FRAMEWORK EXTENDING

- **@RunWith**
  - runs test class in specific, defined way
- **@Rule**
  - executing additional steps in single cycle
  - provides additional information that are available during test execution

JUNIT – PARAMETERS FROM ANNOTATION

```java
@RunWith(JUnitParamsRunner.class)
public class AdditionCalculatorJUnitParamsTest {

  @Test
  @Parameters({"1, 2, 3", "2, 3, 5", "3, 5, 8"})
  public void shouldSumTwoNumbers(int first, int second, int expectedResult) {
    //given
    Calculator sut = new Calculator();
    //when
    int result = sut.add(first, second);
    //then
    assertEquals(expectedResult, result);
  }
}
```

JUNIT – PARAMETERS FROM SEPARATE METHOD

```java
@RunWith(JUnitParamsRunner.class)
public class AdditionCalculatorJUnitParamsMethodTest {

  @Test
  @Parameters(method = "parametersForShouldSumTwoNumbers")
  public void shouldSumTwoNumbers(int first, int second, int expectedResult) {
    //given
    Calculator sut = new Calculator();
    //when
    int result = sut.add(first, second);
    //then
    assertEquals(expectedResult, result);
  }

  private Object[] parametersForShouldSumTwoNumbers() {
    return $(
        $(1, 2, 3),
        $(2, 3, 5),
        $(3, 5, 8)
    );
  }
```

JUNIT – DRAWBACKS

- poor implementation – just basic verification
- sometimes, there is a need to add own logic
- *(expected, actual)* or *(actual, expected)*

ASSERTJ – YOU CAN EXPECT MORE

- simplifies the writing of assert statements in tests
- improves the readability of asserts statements
- has a fluent interface for assertions, which makes the code completion easy to write
- base method for AssertJ assertions is the *assertThat* method followed by the assertion

UNIT TESTING – EXCEPTIONS

- a need to verify what type of exception has been thrown
- additional needs to verify:
  - error message
  - exception field
  - cause
  - root cause
  - …

EXCEPTION TESTING – EXPECTED

- easy and readable
- limited – no control where the exception has been thrown

```
@Test(expected = CommunicationException.class)
public void shouldThrowBusinessExceptionOnCommunicationProblem() {
  //given
  RequestSender sut = new RequestSender();
  //when
  sut.sendPing(TEST_REQUEST_ID);
  //then
  //exception expected
}
```

EXCEPTION TESTING – TRY…CATCH

- complex and unreadable
- **not recommended**

```
@Test
public void shouldThrowBusinessExceptionOnCommunicationProblem() {
  RequestSender sut = new RequestSender();
  try {
    sut.sendPing(TEST_REQUEST_ID);
    fail("Exception should be thrown");
  } catch (CommunicationException e) {
    assertEquals("Communication problem when sending requestId" +
                 " with id: 5", e.getMessage());
    assertEquals(TEST_REQUEST_ID, e.getRequestId());
    assertEquals(ConnectException.class, getRootCause(e));
  }
}
```

EXCEPTION TESTING – EXPECTEDEXCEPTION

- better than previous one, but it is still no perfection due to poor readiness and no advanced assertion checks

```
@Rule
public ExpectedException exception = ExpectedException.none();

@Test
public void shouldThrowBusinessExceptionOnCommunicationProblem() {
  RequestSender sut = new RequestSender();

  exception.expect(CommunicationException.class);
  exception.expectMessage("Communication problem when sending " +
          "requestId with id: 5");
  //TODO: Problem with cause and rootCause
  sut.sendPing(TEST_REQUEST_ID);
}
```

EXCEPTION TESTING – ASSERTJ

- more readable and has more useful features, however it is external (additional) library

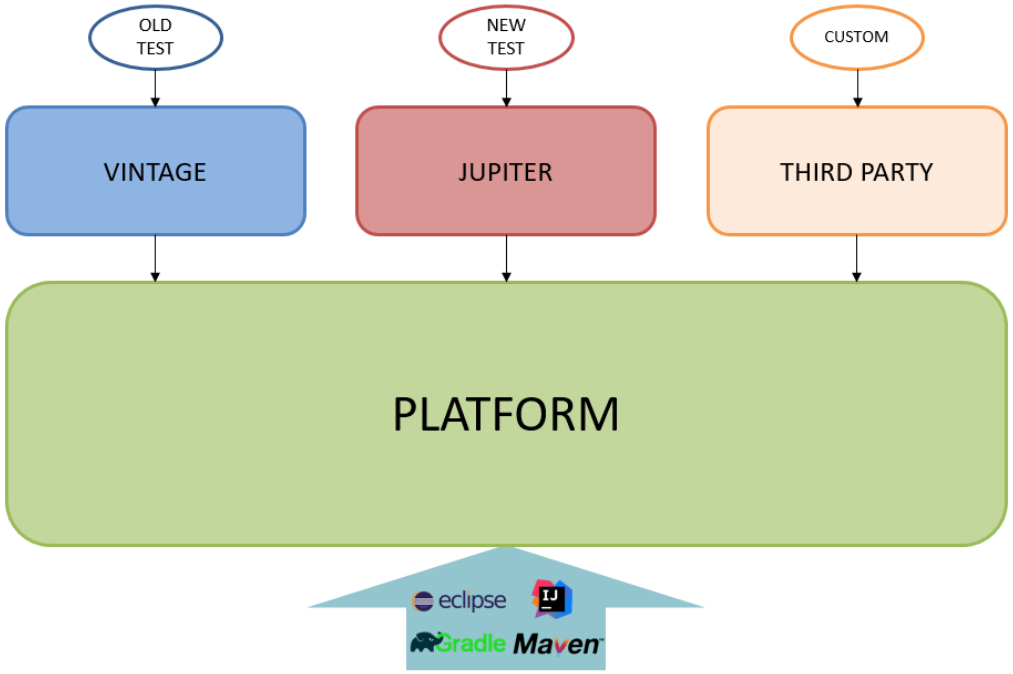
```
@Test
public void shouldThrowBusinessExceptionOnCommunicationProblem() {
  //given
  RequestSender sut = new RequestSender();

  when(sut).sendPing(TEST_REQUEST_ID);

  then(caughtException())
        .isInstanceOf(CommunicationException.class)
        .hasMessage("Communication problem when sending requestId " +
                "with id: 5")
        .hasRootCauseInstanceOf(ConnectException.class);
}
```

JUNIT 5

TIME TO BE A FULLY FUNTIONAL FRAMEWORK

JUNIT 5 – MODULARITY

- JUnit 5 is not a library any more…
- Now it is a **framework**!
- **JUNIT 5 = PLATFORM + JUPITER + VINTAGE**
- **JUnit Platform**
  - Foundation for launching testing frameworks on the JVM
  - Launcher and TestEngine APIs
  - ConsoleLauncher, Gradle plugin, Maven Surefire provider
- **JUnit Jupiter**
  - New programming model and extension API for JUnit 5
- **JUnit Vintage**
  - TestEngines for running JUnit 3 and 4



JUNIT 5 – JUPITER API

- New annotations API
  - @BeforeAll
  - @BeforeEach
  - @Test
  - @AfterEach
  - @AfterAll
- New assertions
  - assertThrows(…)
  - assertAll(…)
  - assertTimeout(…)
- Assumptions
- @DisplayName
- @Nested
- Dynamic tests
  - @TestFactory
- On/Off
  - @Disabled
  - @EnableOnJRE
  - @Disabled
  - Custom…
- @RepeatedTest
- Test method parameters
  - TestInfo
- @ParametrizedTest
  - @ValueSource
  - @CsvSource
  - @MethodSource
  - @CsvFileSource
  - @EnumSource
  - @ArgumentSource

JUNIT 5 – EXTENSION API

- Declaratively via `@ExtendWith`
- Test instance post processing
  - `TestInstancePostProcessor`
- Conditional test execution
  - `ExecutionCondition`
- Parameters resolution
  - `ParameterResolver`
- Callbacks
  - `BeforeAllCallback`
  - `BeforeEachCallback`
  - `BeforeTestExecutionCallback`
  - `AfterTestExecutionCallback`
  - `AfterEachCallback`
  - `AfterAllCallback`
- Exception handling
  - `TestExcecutionExceptionHandling`