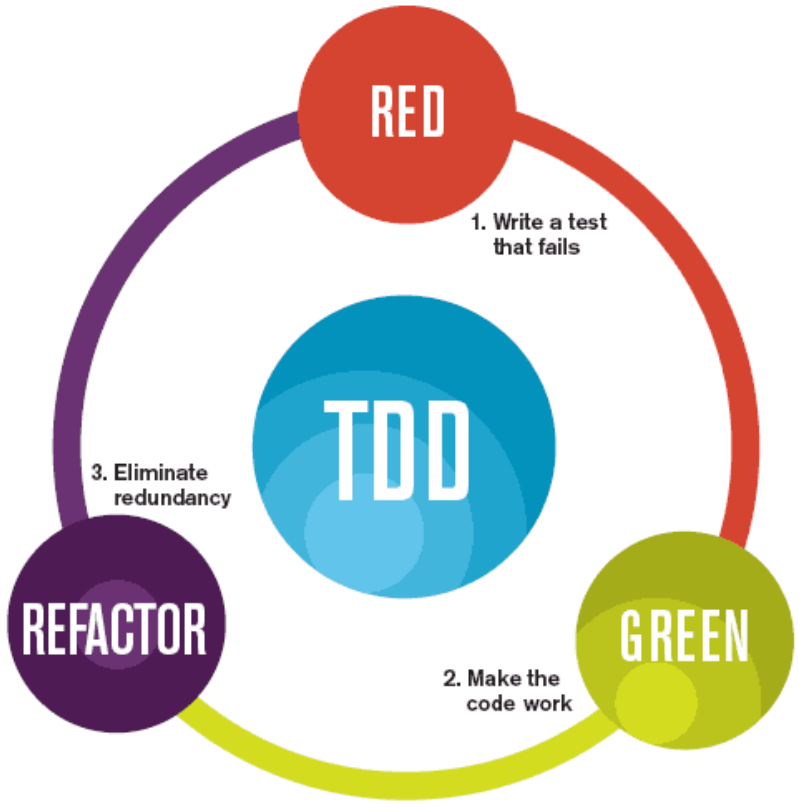
Testowanie jednostkowe

Junit 4, AssertJ, TDD

Why testing ? TDD ?

- 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."

Testowanie jednostkowe

- Dobre testy jednostkowe - F.I.R.S.T.
- **F**ast - Szybkie
- **I**solated - Skupione
- **R**epeatable - Powtarzalne
- **S**elf-validating - Samosprawdzające
- **T**imely - Pisane w odpowiednim momencie

Testowanie jednostkowe

Dobre testy jednostkowe - Fast

- milisekundy na test
- uruchamianie bez obaw z IDE po każdej zmianie
- możliwość uruchomienia kilku tysięcy w rozsądnym czasie

Testowanie jednostkowe

Dobre testy jednostkowe - Isolated

- testujące jedną rzecz
- z jasnym komunikatem błędu

Testowanie jednostkowe

Dobre testy jednostkowe - Repeatable

- z takim samym rezultatem za każdym razem
- bez założonego stanu początkowego
- bez wpływu na inne testy
- niezależne od kolejności uruchamiania
  - samodzielnie lub w grupie
  - z IDE lub systemu budowania
- bez zależności do zasobów zewnętrznych
  - internet, baza danych, inne systemy, pliki *

Testowanie jednostkowe

Dobre testy jednostkowe - Self-validating

- automatyczna asercja - przeszedł lub nie
- bez konieczności ręcznego analizowania wyjścia

Testowanie jednostkowe

Dobre testy jednostkowe - Timely

- pisane w dobrym momencie
- chwilę przed kodem, który spowoduje, że test przejdzie

Testowanie jednostkowe

Konstrukcje wspierające testowalność

- kompozycja zamiast dziedziczenia
- małe klasy
- małe metody
- wstrzykiwanie zależności
- praca na interfejsach

Testowanie jednostkowe

Antywzorce dla testowalnego kodu

- singletony
- metody statyczne
- elementy finalne

JUnit - prosty przykład

```
public class CalculatorTest {

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

JUnit - inicjalizacja i sprzątanie

```
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 - rozszerzenie możliwości

- @RunWith
  - możliwość uruchomienia klasy testowej w specyficzny sposób
- @Rule
  - wykonanie dodatkowych czynności w cyklu wykonywania testu
  - dostarczenie dodatkowych informacji dostępnych wewnątrz testu

Testy parametryzowane - JUnit

- bardzo długo niedostępne
- potem mało wygodny runner *@Parametrized*
  - parametryzacja na poziomie klasy (pola klasy)
  - rozwlekły - np. wymagany dodatkowy konstruktor
  - jedynie testy parametryzowane w danej klasie
  - problem z podglądem wartości parametrów w niedziałającym teście

Testy parametryzowane - JUnitParams

Parametry bezpośrednio w adnotacji

```
@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);
  }
}
```

Testy parametryzowane - JUnitParams

Parametry z oddzielnej metody

```
@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)
    );
  }
```

Standardowe asercje JUnit - minusy

- ubogie - tylko podstawowe weryfikacje
- własna logika potrzebna w wielu przypadkach
- *(expected, actual)* or *(actual, expected)* ?

```
assertEquals(3, receivedNumberOfShips);
```

Standardowe asercje JUnit

- `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)`
- `...`

AssertJ - oczekuj więcej

- kontynuacja FEST Assert
- czytelny DSL
  - `assertThat(receivedNumberOfShips).isEqualTo(3);`
  - z podpowiadaniem w IDE
- wyspecjalizowane asercje zależne od typu
  - m.in. kolekcji/tablic, dat, tekstów, wyjątków
- ekstrakcja pól
- soft assertions

Testowanie wyjątków

- weryfikacja, że wyjątek określonego typu został rzucony
- często dodatkowo weryfikacja:
  - komunikatu błędu (message)
  - biznesowego pola w wyjątku
  - przyczyny (cause)
  - źródłowej przyczyny (root cause)
  - konkretnego wywołania, które go spowodowało

Testowanie wyjątków - expected

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

- proste i czytelne
- ograniczone - bez kontroli, w którym miejscu został rzucony wyjątek

Testowanie wyjątków - try..catch

```
@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));
  }
}
```

- złożone i nieczytelne
- **niezalecane do użytku** - są lepsze sposoby

Testowanie wyjątków - ExpectedException

```
@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);
}
```

- wbudowane w JUnit (4.7+)
- brak zaawansowanych sprawdzeń
- średnio czytelne

Testowanie wyjątków - catch-exception

```
@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);
}
```

- czytelne i o dużych możliwościach (assercje AssertJ)
- nie może być użyte z wyjątkiem z konstruktora oraz statycznym wywołaniem
- zewnętrzna biblioteka (nowa zależność testowa)
- mało jasny komunikat przy braku wyjątku