Studia Podyplomowe: Java EE - produkcja oprogramowania

Produkcja i testowanie (PTE)
Testy jednostkowe

(do użytku wewnętrznego)

Anna Derezińska
Instytut Informatyki, Politechnika Warszawska

Semestr zimowy 2020/21
Grupa: JA20Z (zesp. JA1-A, JA1-B) - semestr pierwszy

1

Automatyzacja testowania

1. **Zadania administracyjne**, np.
   - przechowywanie specyfikacji testów
   - generacja raportów testowych
2. **Realizacja testów**, np.
   - **uruchamianie testów, powtarzanie testów**
   - monitorowanie przebiegu testów
   - testy odtwarzająco-przechwytujące (capture/replay)
3. **Wybór testów, np.**
   - wybór kolejności wykonania (priorytetyzacja)
   - minimalizacja zbioru testów (dla zadanego kryterium pokrycia)
   - wybór testów dla zależnego kodu (regresyjnych)
4. **Generacja testów**, np.
   - generacja testów na podstawie formalnej specyfikacji, modeli
   - **generacja testów na podstawie analizy kodu źródłowego**

2

Literatura

- K. Beck TDD. Sztuka tworzenia dobrego kodu, Helion 2014 (K. Beck, *Test Driven Development by Example*, Addison-Wesley, 2002)
- L. Vogel, Unit Testing with JUnit – Tutorial http://www.vogella.com/tutorials/JUnit/article.html
- K. Beck, E. Gamma, JUnit Test Infected: Programmers love writing tests members.pingnet.ch/gamma/junit.htm
- Andy Hunt, Dave Thomas, JUnit. Pragmatyczne testy jednostkowe w Javie, Helion 2006

3

Testy jednostkowe (unit tests)

- Test w postaci kodu, który sprawdza działanie niewielkiego fragmentu kodu źródłowego (jednostki) wyznaczonego przez strukturę kodu (np. klasa, metoda) i/lub realizowana funkcjonalność
- Tworzone głównie przez osoby implementujące kod

4

Testy jednostkowe (unit tests) c.d.

K – testowana klasa (*CUT – class under test*)
TestK – klasa testująca usługi K
lub metody testujące w klasie K
Weryfikacja stanu obiektu klasy K

Zalety
– budowane przed lub równolegle z kodem
- testowanie regresyjne
Wady
– pracochłonność tworzenia testów
Przyspieszenie generacji testów:
- szkielety metod testujących, tworzenie metod

5

Prosty test z wyrocznią

```
package pl.edu.pw.ii.pte.junit.simple;

import static
org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class AdditionTest {
  private int x = 2;
  private int y = 2;
  @Test
  public void addition() {
      int z = x + y;
      assertEquals(4, z);
      }
}
```

8

Klasa Money – wzorzec analizy

```
class Money {
   private int fAmount;        //wartość
   private String fCurrency;   //jednostka – np. waluta

   public Money(int amount, String currency) {
      fAmount=amount;
      fCurrency= currency;    }

   public int amount()    {
       return fAmount;     }

   public String currency() {
       return fCurrency;   }
   }
```

source members.pingnet.ch/gamma/junit.htm 10

10

Money - dodawnie takiej samej waluty

```
package pl.edu.pw.ii.pte.junit.money;

class Money {
  private int fAmount;
  private String fCurrency;
//.....................

  public Money add(Money m) {
    return new Money(amount()+m.amount(), currency());}
}
```

12

12

Proste porównywanie pieniędzy

```
public boolean equals(Object anObject) {
      if (anObject instanceof Money) {
            Money a = (Money) anObject;
            return a.currency().equals(currency()) &&
                  amount() == a.amount();
      }
      return false;
}
```

14

14

Test porównywania Pieniędzy

```
public class MoneyTest {

@Test
public void testEquals() {
      Money m12CHF = new Money(12, "CHF");
      Money m14CHF = new Money(14, "CHF");

      assertTrue(!m12CHF.equals(null));
      assertEquals(m12CHF, m12CHF);
      assertEquals(m12CHF, new Money(12, "CHF"));
      assertTrue(!m12CHF.equals(m14CHF));
      }
}
```

package junit.money Money, MoneyTest 16

16

Test dodawania Pieniędzy

```
public class MoneyTest {

  @Test
  public void testSimpleAdd() {
      // 1. Tworzenie obiektów
      Money m12CHF = new Money(12, "CHF");
      Money m14CHF = new Money(14, "CHF");
      Money expected = new Money(26, "CHF");
      // 2. Przetwarzanie obiektów
      Money result = m12CHF.add(m14CHF);
      // 3. Weryfikacja rezultatów
      assertTrue(expected.equals(result));
      }
}
```

17

17

Narzędzia X-Unit dla Javy

JUnit junit.org
- JUnit3.x
- JUnit4.x
- JUnit5.x [https://junit.org/junit5/docs/current/user-guide/]

JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*
Java 8 lub wyższa

Dużo kodu odziedziczonego/zastanego (legacy code) z testami w Junit4

testNG testng.org

19

19

Adnotacje JUnit 5.x

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	@Test	@Test
Execute before all test methods in the current class	@BeforeClass	@BeforeAll
Execute after all test methods in the current class	@AfterClass	@AfterAll
Execute before each test method	@Before	@BeforeEach
Execute after each test method	@After	@AfterEach
Disable a test method / class	@Ignore	@Disabled

[https://howtodoinjava.com/junit5/junit-5-vs-junit-4/]

Adnotacje JUnit 4.x

Feature	JUnit 3.x	JUnit 4.x
test annotation	testXXX pattern	@Test
run before the first test method in the current class is invoked	None	@BeforeClass
run after all the test methods in the current class have been run	None	@AfterClass
run before each test method	override setUp()	@Before
run after each test method	override tearDown()	@After
ignore test	Comment out or remove code	@ignore
expected exception	catch exception assert success	@Test(expected = ArithmeticException.class)
timeout	None	@Test(timeout = 1000)

[http://www.asjava.com/junit/junit-3-vs-junit-4-comparison/]

JUnit 5.x – przykład typowego użycia /1/

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import org.junit.jupiter.api.AfterEach;
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.Test;
5
6  public class TournamentTest {
7    Tournament tournament;
8
9    @BeforeEach
10   public void init() throws Exception {
11     System.out.println("Setting up ...");
12     tournament = new Tournament(100, 60);
13   }
14
15    @AfterEach
16   public void destroy() throws Exception {
17     System.out.println("Tearing down ...");
18     tournament = null;
19   }
```


przykład typowego użycia /2/

```
21
22   @Test
23   public void testGetBestTeam() {
24     assertNotNull(tournament);
25
26     Team team = tournament.getBestTeam();
27     assertNotNull(team);
28     assertEquals(team.getName(), "Test1");
29   }
30 }
```

[http://www.asjava.com/junit/junit-3-vs-junit-4-comparison/]

Test Pieniędzy z *Before-* i *AfterEach*

```
public class MoneyTest2 {
  private Money m12CHF;
  private Money m14CHF;
  @BeforeEach
  public void setUp() throws Exception {
      m12CHF = new Money(12, "CHF");
      m14CHF = new Money(14, "CHF");
  }
  @AfterEach
  public void tearDown() throws Exception {
  }

  @Test
  public void testSimpleAdd2() {
      Money expected = new Money(26, "CHF");
      Money result = m12CHF.add(m14CHF);
      assertTrue(expected.equals(result));
  }
}
```

package junit.money Money, MoneyTest2

Kolejność wykonywania testów

- Wykonanie testu nie powinno zależeć od wykonania innych testów!
- Zakłada się, że kolejność wykonania testów może być dowolna
- Jeśli przed wykonaniem każdego testu (lub po) wymagane są czynności inicjalizacyjne (sprzątające) należy użyć adnotacji @BeforeEach (@AfterEach)
- Jeśli wymagane są czynności inicjalizacyjne (sprzątające) przed wszystkimi testami klasy (lub po) należy użyć adnotacji @BeforeAll (@AfterAll)

Gwarantowana kolejność - przykład

```
@BeforeEach
   public void initializeSystem() {
   this.user = userDAO.createUser(1, "Foo", "Bar");
   }
@Test
   public void getUserById() {
   User newUser = userDAO.findUserById(1);
   assertEquals(this.user.getName(),
   newUser.getName());
   assertEquals(this.user.getSurname(),
   newUser.getSurname());
   }
@AfterEach
   public void cleanSystemState() {
   userDAO.deleteUser(this.user);
   }
```

35

35

After/Before All vs Each

@BeforeAll – wykonuje metodę przed początkiem wszystkich testów tej klasy, np. wykonaj połączenie z bazą danych

@AfterAll - wykonuje metodę po zakończeniu wszystkich testów tej klasy, np. odłącz od bazy danych

- Tylko po jednej metodzie w klasie
- Metoda publiczna i statyczna
- @BeforeAll metoda klasy bazowej – uruchamiana przed metodą z danej klasy
- @AfterAll z klasy bazowej – po ...
- Uwaga na zależności pomiędzy testami

package junit.executionOrder AfterBeforeTest

36

36

Określona kolejność wykonania

JUnit 5.4 i późniejsze

@TestMethodOrder

- domyślny porządek deterministyczny (testy wykonywane zawsze w tej samej kolejności), ale nieokreślony
- @TestMethodOrder(OrderAnnotation.class)
  zgodnie z porządkiem adnotacji @Order
- @TestMethodOrder(Alphanumeric.class)
  porządek leksykograficzny nazw metod
- możliwa implementacja własnego porządku testów – wymagana klasa implementująca interfejs *MethodOrderer*

37

37

Określona kolejność wykonania

JUnit 4.11 i późniejsze

@FixMethodOrder

@FixMethodOrder(MethodSorters.DEFAULT)
porządek deterministyczny, ale nieokreślony
(często leksykograficzny nazw metod)

@FixMethodOrder(MethodSorters.JVM)
jak w maszynie wirtualnej, różny dla wykonań

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
porządek leksykograficzny nazw metod

38

38

Porządek leksykograficzny

```
package pl.edu.pw.ii.pte.junit.executionOrder;
...

@TestMethodOrder(Alphanumeric.class)
public class TestMethodOrder {
  @Test
  public void testC() {
      System.out.println("trzy");
  }
  @Test
  public void testA() {
      System.out.println("jeden");
  }
  @Test
  public void testB() {
      System.out.println("dwa");
  }
}
```

40

40

Sprawdzanie warunków

Warunki logiczne
- assertTrue(warunek logiczny, [opcjonalna wiadomość])
- assertFalse(...)

Porównanie wartości
- assertEquals(wartość oczekiwana, wartość porównywana, [wiadomość])
- assertArrayEquals(wartość oczekiwana, wartość porównywana, [wiadomość])

Porównanie identyczności obiektów
- assertSame(oczekiwany, porównywany, [wiadomość])
- assertNotSame (...)

42

[https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions

42

Przykład porównania wartości

```
package pl.edu.pw.ii.pte.junit.assertions;
...

public class AssertTests {
  @Test
  public void testAssertEquals() {
      assertEquals("text", "text",
                   "failure - strings are not equal");
  }  //wynik poprawny
  @Test
  public void testAssertEquals2() {
      assertEquals("text1", "text2",
                   "failure - strings are not equal");
  }  //zgłoszony błąd
...
```

44

44

Porównanie tablic /1/

```
@Test
public void testAssertArrayEquals() {
      byte[] expected = "abcd".getBytes();
      byte[] actual = "abcd".getBytes();
      assertArrayEquals(expected, actual,
                   "failure - byte arrays not same");
} //wynik poprawny

@Test
public void testAssertArrayEquals2() {
      byte[] expected = "abcd".getBytes();
      byte[] actual = "abcdef".getBytes();
      assertArrayEquals(expected, actual,
                   "failure - byte arrays not same");
} //zgłoszony błąd - różna długość tablic
```

46

46

Porównanie tablic /2/

```
@Test
public void testAssertArrayEquals3() {
      byte[] expected = "abcd".getBytes();
      byte[] actual = "cdef".getBytes();
      assertArrayEquals(expected, actual,
                   "failure - byte arrays not
same");
} //zgłoszony błąd

@Test
public void testAssertArrayEquals4() {
      byte[] expected = "abcd".getBytes();
      byte[] actual = "abcf".getBytes();
      assertArrayEquals(expected, actual,
                   "failure - byte arrays not
same");
} //zgłoszony błąd - różnica elementu[3]
```

47

47

Identyczność a równość wartości

```
@Test
public void testAssertArraySame() {
      byte[] expected = "abcd".getBytes();
      assertSame(expected, expected);
} //wynik poprawny - identyczne obiekty

@Test
public void testAssertArraySame2() {
      byte[] expected = "abcd".getBytes();
      byte[] actual = "abcd".getBytes();
      assertSame(expected, actual);
} //zgłoszony błąd - obiekty nie są identyczne choć
mają takie same wartości
```

49

49

Klasa *Money* – Ćwiczenie A.1

- Rozbudować klasę *Money*, o

1) metodę mnożenia pieniędzy przez liczbę,

```
public Money multiplyCurrency(int k){..}
```

- Napisać testy jednostkowe dla zaimplementowanej metody. Uwzględnić sytuacje typowe i graniczne (np. mnożenie przez 1).

50

Klasa *Money* – praca domowa A.2

- Rozbudować klasę *Money*, o

2) metodę dodawania (odejmowania) pieniędzy w różnych walutach (*min 3 różne waluty*) z uwzględnieniem kursów odpowiednich walut,

```
public Money addAnyCurrency(Money m){..}
```

3) porównywanie kwot w różnych walutach, itp.

- Napisać testy jednostkowe dla zaimplementowanych metod. Uwzględnić sytuacje typowe i graniczne (np. dodawanie 0$ do czegoś).

51

W Javie - słowo kluczowe *assert*

Element Javy (1.4 i wyżej) – a nie JUnit
assert <wyrażenie boolowskie>
Zakłada, że wyrażenie jest prawdziwe w czasie wykonania
assert true;
assert 1==1;
Dla wyniku *false*, zgłoszony jest *AssertionError*
assert x>0; //do sprawdzenie poprawności x
- Użycie klucza *-ea JVM* pozwala na raportowanie błędnego wykonania asercji.
- Można zablokować wykonanie asercji w bytcode
javac -disableassertion MyClass.java

52

52

Testy z limitem czasu JUnit 5

```
package pl.edu.pw.ii.pte.junit.timeout;
import static java.time.Duration.ofMillis;
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;
import org.junit.jupiter.api.Test;

public class TimeoutTest {
        @Test
        public void infinity() {
                assertTimeoutPreemptively(
                        ofMillis(10),
                        () -> { while (true) {;} });
        }

        @Test
        void timeoutExceeded() {
                assertTimeout(
                        ofMillis(10),
                        () -> {
                                // Simulate task that takes more than 10 ms.
                                Thread.sleep(100); });
        }
}
```

53

53

JUnit4 – testy z limitem czasu

```
package pl.edu.pw.ii.pte.junit.timeout;

import org.junit.Test;

public class TimeoutTest {

  @Test(timeout = 1000) //zgłoś błąd jeśli metoda trwa
                        //dłużej niż 1000ms
  public void infinity() {
       while (true) {;}
  }
}
//otrzymamy TestTimedOutException
```

54

54

Globalny limit czasu – JUnit 4

```
import org.junit.rules.Timeout;
public class GlobalTimeoutTest {
  public static String log;
  @Rule
  public Timeout globalTimeout = Timeout.seconds(1);
  // 1 second max per method tested
  @Test
  public void infinity() throws Exception {
       while (true) {;}
  }
  @Test
  public void testSleepForTooLong() throws Exception
{
       log += "ran1";
       Thread.sleep(100_000); // sleep for 100 seconds
  }
```

55

55

JUnit4 – wyjątki

```
@Test(expected=nazwa klasy wyjątku)
//testuj czy metoda zgłosiła dany wyjątek

@Test(expected=ArithmeticException.class)
  public void divideByZero() {
            int n = 2 / 0;
  }

@Test(expected=IndexOutOfBoundsException.class)
  public void empty() {
       new ArrayList<Object>().get(0); }
```

56

56

Testowanie wyjątków JUnit5

Assertions.assertThrows(...);
– sprawdza tyko jeden wyjątek, nie testuje wiadomości ani stanu obiektu po zgłoszeniu wyjątku

Schemat testowania wyjątków:

```
try{
        mustThrowException();
        fail();
} catch (Exception e) {
        // oczekiwany wyjątek
        //można również sprawdzać
        //komunikat oczekiwanego wyjątku, ..
}
```

58

58

Testy klasy *Account* – oczekiwany wyjątek

```
public  class TestAccount extends TestCase {
  public void ballanceEqualAfterDeposit() {
    Account o = new Account();
    o.deposit(1000);
    assertTrue(o.getBalance() == 1000);              }
public void withdrawMoreThanBalanceNotAllowed() {
    try {
         Account o = new Account();
         o.deposit(300);
         o.withdraw(1000);
         fail("InsufficientBalanceException expected");}
    catch (InsufficientBalanceException e) {...        };
}
```

59

Testy klasy *Account* (2) – wyjątek jako błąd

```
public void canWithdrawLessThanBalance() {
   try {
        Account o = new Account();
        o.deposit(1000);
        assertTrue(o.getBalance() == 1000);
        o.withdraw(300);
        assertTrue(o.getBalance() == 700);
        }
   catch (Exception e) {
              fail("Unexpected exception");  };
   }
…
}
```

60

Konstrukcja *try/catch* lub ...

```
@Test
public void testExceptionMessage() {
  try {
    new ArrayList<Object>().get(0);
    fail("Expected an IndexOutOfBoundsException to be
          thrown"); }
  catch
  (IndexOutOfBoundsException
anIndexOutOfBoundsException)
    {
   assertEquals("Index: 0, Size: 0",

IndexOutOfBoundsException.getMessage());
    }
  }
```

61

61

Oczekiwany wyjątek – JUnit 5

```
package pl.edu.pw.ii.pte.junit.exceptionRule;

import static org.junit.jupiter.api.Assertions.assertThrows;
import java.util.*;
import org.junit.jupiter.api.Test;

public class ExceptionRuleDemo {
@Test
public void shouldTestExceptionMessage() {
  assertThrows(
    IndexOutOfBoundsException.class,
    ()->{
         List<Object> list = new ArrayList<Object>();
         list.get(0); // wykonanie nie przekroczy tej linii
         }
    );
  }
}
```

62

62

Testy jednostkowe - weryfikacja

1. Właściwa inicjalizacja obiektów
2. Czy określone funkcje lub reguły biznesowe zostały wykonane prawidłowo
3. Sposób niszczenia obiektów i ewentualnego zapisu ich stanu
4. Kolejności wykonania metod i porównania zwracanych przez nie rezultatów ze wzorcem

64

64

JUnit5 – grupowanie testów

```
@RunWith(JUnitPlatform.class)
@SelectClasses({
ClassATest.class,
ClassBTest.class,
ClassCTest.class })

public class JUnit5TestSuiteExample
{
}
```

65

65

JUnit4 - grupowanie testów

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
   TestClass1.class,
   //lista klas oddzielonych przecinkami
   //wykonywane wg kolejności na liście
})
public class AllTests {
   }
```

66

66

JUnit5 – pomijanie testów

@Disabled – pomiń metodę, np. nieaktualny test dla zmienionego kodu

```
@Disabled("out of date after refactoring")
@Test
public void multiplication() {
 }
```

67

67

JUnit4 – pomijanie testów

@Ignore – pomiń metodę, np. nieaktualny test dla zmienionego kodu

```
@Ignore("out of date after refactoring")
@Test
public void multiplication() {
   }
```

68

68

Dobre testy jednostkowe F.I.R.S.T.

1. Fast – ograniczony czas wykonania
2. Independent – niezależne od innych testów
3. Repeatable – wykonanie powtarzalne i deterministyczne
4. Self-validating – jednoznaczny wynik pass/fail
5. Timely – TDD przed kodem, inne – możliwie szybko po kodzie

69

R. Martin, Clean Code: A Handbook of Agile Software Craftmanship (Prentice Hall, 2008)

69

Wspomaganie budowy testów jedn.

- Generowanie szkieletów metod testujących
- Generowanie danych testowych, metod testujących z asercjami na podstawie kodu (i kontraktów) lub wykonania
- Filtr wyboru różnych metod i funkcji (np. publiczne, prywatne)
- Automatyczna walidacja warunków końcowych
- Repozytorium obiektów
- Tworzenie przypadków testowych z modeli (UML, SysML)
- Tworzenie przypadków testowych ze specyfikacji (semi-)formalnych

70

70

Generacja testów jednostkowych

Java

SilverMark's Test Mentor – Java Edition
Parasoft Jtest, Unit Tester z AppPerfect DevTest4J
JCrasher, TestGen4J, JUB, Junit Test Generator,
CodePro Junit Test Case Generation
Agitar Test Runner, Daikon (dynamiczne) z Eclat, Jov,..

Randoop
EvoSuite

75

75