Testy jednostkowe

Stuby i mocki

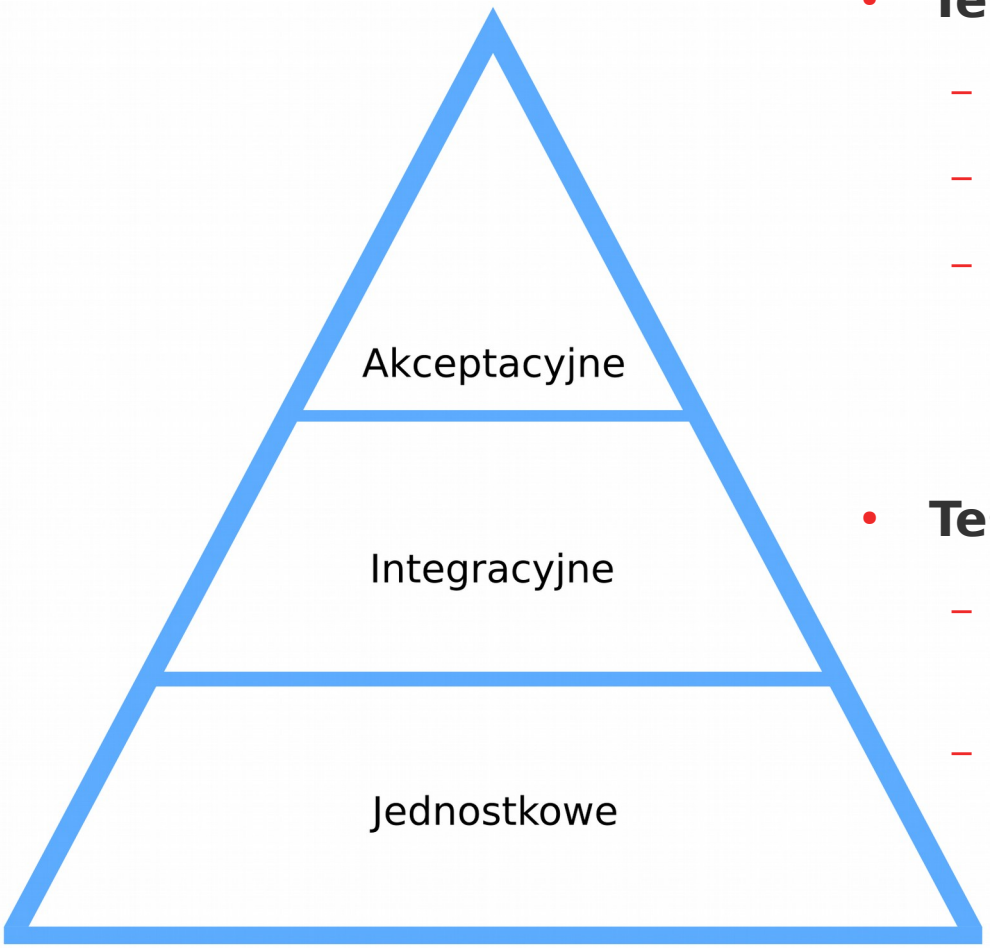
Testy jednostkowe

Idea

- Ideą testów jest zapewnienie pewności, że produkcyjny kod działa i każda refaktoryzacja nie będzie wstanie go zepsuć
- Dzięki testom w łatwy sposób tworzona jest aktualna dokumentacja programu
- Testujemy przede wszystkim logikę biznesową
- Zastosowanie mniej oczywiste (testy przy pomocy testów):
  - Testy umożliwiają w prosty sposób nauczyć się danej biblioteki
  - W szybki sposób można sprawdzić komunikację z zewnętrznym serwisem (świadomie łamiąc idee testów jednostkowych)

Testy jednostkowe

Piramida testów



- **Testy akceptacyjne (end to end)**
  - zajmują około 10% wszystkich testów
  - znajdują się na same górze hierarchii
  - testują one całą funkcjonalność od początku do końca, symulują zachowanie użytkowników i najczęściej testują też UI
- **Testy integracyjne**
  - testy zajmują około 20% wszystkich testów
  - testują one integrację z innymi komponentami, które nie są wyizolowane, np. baza danych, system plików

Testy jednostkowe

Definicja

- Testy zajmują około 70% wszystkich testów
- To automatyczny fragment kodu, który wywołuje testowaną jednostkę pracy, a następnie sprawdza pewne założenia dotyczące pojedynczego wyniku końcowego tej jednostki. Testy jednostkowe pisze się łatwo, są „tanie” i można je szybko uruchomić. Są wiarygodne, czytelne i łatwe w utrzymaniu. Są spójne w swoich wynikach, o ile kod produkcyjny się nie zmienił. Ponadto, testy te uruchamiane są w izolacji, oznacza to, że nie jesteśmy w żaden sposób związani z innymi elementami systemu
- Jednostka pracy może obejmować metodę lub wiele klas i funkcji potrzebnych do osiągnięcia celu. Mówi się, że dobrze jest minimalizować naszą jednostkę pracy, aby test był jak najmniejszy jednak nie zawsze jest to dobre podejście. Czasem zbyt mocne zmniejszenie rozmiaru powoduje, że testujemy tylko pośrednie kroki zamiast otrzymać zauważalny efekt końcowy testowanego API.
- Jednostka pracy jest to suma działań, które mają miejsce pomiędzy wywołaniem publicznej metody w systemie, a pojedynczym zauważalnym efektem końcowym. Publiczna metoda może za sobą kryć prostą metodę zwracającą wartość lub należeć do fasady, wywołując pod sobą wiele różnych zależności nie widocznych z zewnątrz

Testy jednostkowe

Dobry test jednostkowy

- Dobry test jednostkowy powinien być:
  - zautomatyzowany i powtarzalny
  - łatwy do zaimplementownania
  - istotny także w dzień po napisaniu
  - każdy powinien być go w stanie uruchomić w prosty sposób
  - szybki
  - zwracać zawsze ten sam wyniki
  - mieć pełną kontrolę nad testowaną jednostką
  - w pełni niezależny od innych testów

Testy jednostkowe

Test jednostkowy, a integracyjny

- Różnica między tymi dwoma typami testów jest niewielka, ale istotna. Testy integracyjne charakteryzują się tym, że nie da się ich przeprowadzić szybko i spójnie, wykorzystuje jedną lub więcej rzeczywistych zależności
- Przykładem testu integracyjnego jest test wykorzystujący rzeczywisty czas systemowy, prawdziwy system plików lub rzeczywistą bazę danych. Jeżeli np. nie mamy kontroli nad czasem systemowym w testowanej metodzie to w gruncie rzeczy każde wywołanie takiego testu jest innym testem, który w efekcie może dawać różne wyniki
- Testy integracyjne są dobrym uzupełnieniem testów jednostkowych

Testy jednostkowe

Nazewnictwo

- Nadawanie nazw klasom zawierającym testy
  - nazwa jest analogiczna do klasy testowanej przy czym na końcu dodajemy słowo Test
  - dobrą praktyką jest rozdzielenie kodu produkcyjnego od testów
- Nadawanie nazw metodą testowym
  - nazwaJednostkiPracy_testowanyScenariusz_oczekiwaneZachowanie
    - nazwaJednostkiPracy – powinna odpowiadać nazwie metody jeżeli ją testujemy lub być bardziej abstrakcyjna jeżeli jest to przypadek użycia obejmujący wiele metod lub klas
    - testowanyScenariusz – warunki na jakich jednostka jest testowana, np. „zły login”, „nieprawidłowy użytkownik” lub „dobre hasło”
    - oczekiwaneZachowanie – określa jakiego działania oczekujemy od testowanej metody

Testy jednostkowe

Prosty test

- Nasz przypadek testowy
  - pogram zajmuje się rezerwacją pokoi hotelowych. Jego zadaniem jest przyjęcie i zwalidowanie otrzymanych danych oraz odpowiednia reakcja poprzez zawiadomienie o błędzie lub zapisie do bazy danych (lub wysłanie poprzez http do serwera zewnętrznego)

Testy jednostkowe

Prosty test

Metoda produkcyjna sprawdza czy wpisane imię i nazwisko jest niepuste

```
public boolean isValidName(String firstName,String lastName)
{
    if(firstName.isEmpty() || lastName.isEmpty())
    {
        return false;
    }
    return true;
}
```

Metoda testowa

```
@Test
public void isValidName_EmptyName_ReturnFalse() {
    HotelReservation reservation= new HotelReservation();

    boolean result = reservation.isValidName("", "");

    assertFalse(result);
}
```

Test jednostkowy zazwyczaj obejmuje trzy główne zadania:

<- konfigurację obiektów (given)

<- wykonanie operacji na obiektach (when)

<- asercję oczekiwanego rezultatu (then)

Testy jednostkowe

Testy z parametrami

```
@RunWith(Parameterized.class)
public class HotelReservationTest1 {

    private String firstName;
    private String lastName;
    private boolean expected;

    public HotelReservationTest1(String firstName, String lastName, boolean expected) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.expected = expected;
    }

    @Parameterized.Parameters
    public static Collection validNameList() {
        return Arrays.asList(new Object[][]{{"", "", false}, {"a", "b", true}});
    }

    @Test
    public void isValidName_EmptyName_ReturnFalse() {
        HotelReservation reservation = new HotelReservation();
        boolean result = reservation.isValidName(firstName, lastName);

        assertEquals(expected, result);
    }

}
```

Sposób implementacji wykorzystywany w junit4

Testy jednostkowe

Testy z parametrami

Można też wykorzystać zewnętrzną klasę @RunWith(JUnitParamsRunner.class)

```
@Test
@Parameters(value = {"15", "30", "150"})
public void shouldReturnFizzBuzzIfDiv3And5(int p) throws Exception {
    assertEquals("FizzBuzz", sut.fizzBuzz(p));
}
```

Metoda znacznie czytelniejsza i umożliwia wykorzystanie
kilku sparametryzowanych testów w jednej klasie testowej

Testy jednostkowe

Atrybuty Before i After

```
public class HotelReservationTest2 {

    private HotelReservation reservation;

    public HotelReservationTest2() {
    }

    @Before
    public void init() {
        reservation = new HotelReservation();
    }

    @Test
    public void isValidName_EmptyName_ReturnFalse() {

        boolean result = reservation.isValidName("", "");

        assertFalse(result);
    }

    @After
    public void finish() {
        reservation = null;
    }

}
```

Metoda init() z adnotacją @Before tworzy przed każdym uruchomieniem nowego testu obiekt HotelReservation

Metoda finish() z adnotacją @After kasuje nasz obiekt

Wykorzystując przedstawione atrybuty należy pamiętać, że im częściej będziemy z nich korzystać, tym testy będą mniej czytelne. Aby je zrozumieć, osoby analizujące kod, będą musiały go czytać w kilku miejscach.

Dobrą praktyką jest wykorzystanie zamiast nich metody fabryk.

Testy jednostkowe

Testowanie wyjątków

```
public boolean isValidAge(int age)throws Exception
{
    if(age<18)
    {
        throw new Exception("osoba niepełnoletnia");
    }
    return true;
}
```

Przykładowa metoda sprawdzająca czy osoba jest pełnoletnia.
Jeżeli ma mniej niż 18 lat, wtedy zwraca wyjątek

Testy jednostkowe

Testowanie wyjątków

Sposób I

```
@Test(expected = Exception.class)
public void isValidAge_Children_ThrowsException()throws Exception {
    HotelReservation reservation= createHotelReservation();

    reservation.isValidAge(6);

}

private HotelReservation createHotelReservation()
{
    return new HotelReservation();
}
```

Niezalecana metoda testowania wyjątków - działa ona na podobnej zasadzie blok try catch. Niepowodzenie testu pojawia się gdy żaden wyjątek nie zostanie wychwycony, problem pojawia się w momencie gdy inna metoda niż oczekiwana zwróci wyjątek. W ten sposób test się powiedzie, ale zataimy błąd, który może się objawić w najmniej spodziewanym momencie

Sposób II

Lepszym rozwiązaniem jest albo zastosowanie bloku try catch tylko dla wybranej metody (junit4), albo wykorzystanie asercji assertThrows (junit5). Istnieje mniejsze ryzyko, że test nas okłamie

Testy jednostkowe

Stuby (namiastki)

- Stub to obiekt, który w testach służy do imitowania właściwej implementacji. Jego zadaniem jest wyłącznie zwrócenie zadanej wartości

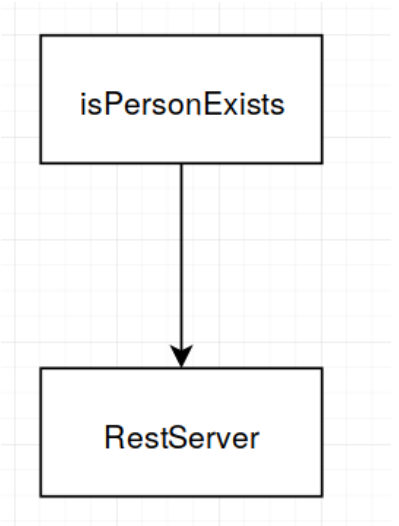
Przykładowa metoda isPersonExists, przyjmująca jako parametr numer pesel, sprawdza w centralnym serwerze, czy dana osoba już istnieje w bazie danych

```
public boolean isPersonExists(String pesel)
{
    RestServer rest = new RestServer();
    return rest.isPersonExists(pesel);
}
```

```
public class RestServer {

    public boolean isPersonExists(String pesel) {
        //cialo metody, zwraca true gdy osoba istnieje w systemie
        return true;
    }

}
```



Testy jednostkowe

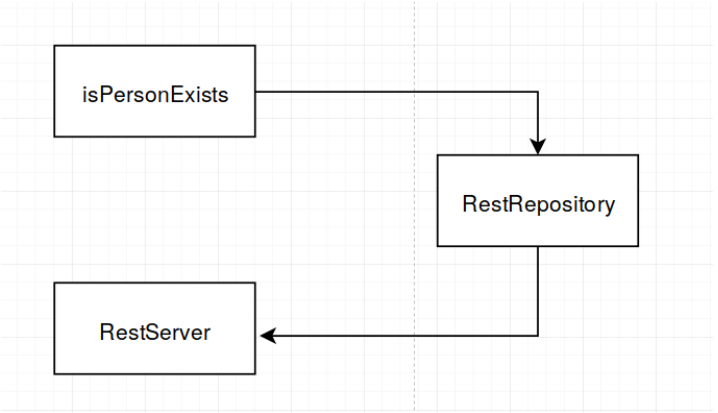
Stuby

- Zewnętrzna zależność uniemożliwia wykonanie tradycyjnego testu
- Zależność ta może przyczynić się do niepowodzenia testu pomimo tego, że logika metody jest poprawna
- „Nie istnieje taki problem obiektowy, którego nie można by było rozwiązać poprzez dodanie warstwy pośredniej, z wyjątkiem, co oczywiste, zbyt wielu warstw pośrednich”
- Zgodnie z tą zasadą działa stub. Dodaje warstwę, która opakowuje wywołanie zewnętrznej zależności i naśladuje jej zachowanie w testach

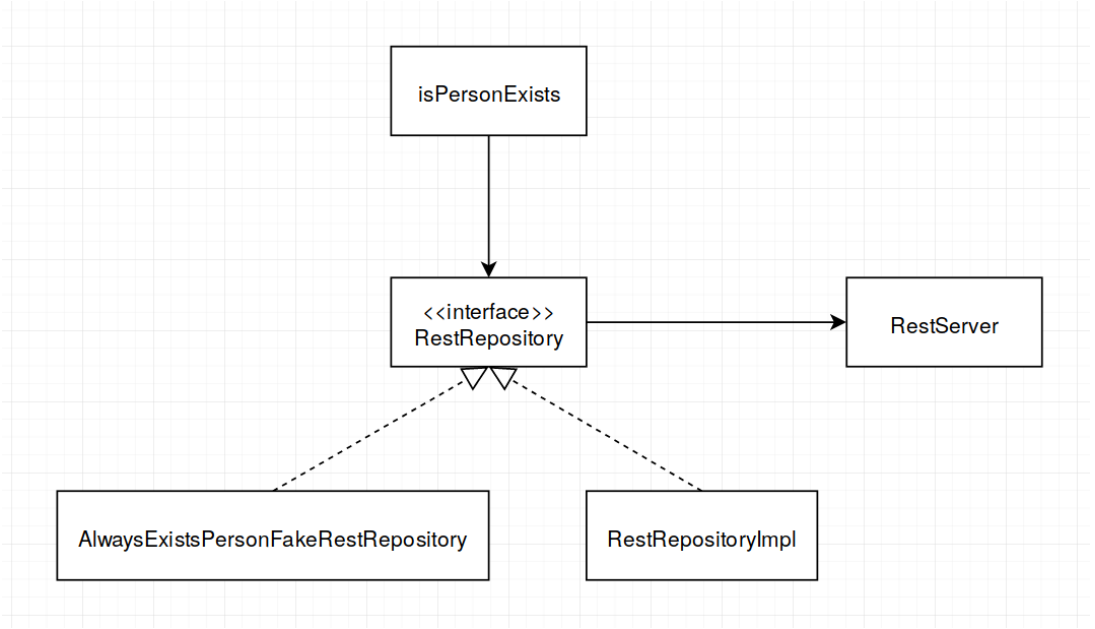
Testy jednostkowe

Stuby

Przenosimy bezpośrednie wywołanie do warstwy pośredniej RestRepository



Interfejs dla RestRepository umożliwia stworzenie namiastki AlwaysExistsPersonFakeRestRepository



Testy jednostkowe

Stuby

```
public class AlwaysExistsPersonFakeRestRepository implements RestRepository {

    @Override
    public boolean isPersonExists(String pesel) {
        return true;
    }

}
```

- Namiastka AlwaysExistsPersonFakeRestRepository zawiera słowo fake, aby jednoznacznie określić iż ten obiekt jest sztuczny
- Dodatkowo rozbudowana nazwa jasno mówi w jaki sposób będzie się zachowywać, ułatwia to czytelność testu

Testy jednostkowe

Stuby - wstrzykiwanie zależności

Przekazanie interfejsu na poziomie konstruktora

```
public class HotelReservation1 {

    private RestRepository repository;

    public HotelReservation1(RestRepository repository) {
        this.repository = repository;
    }

    public boolean isPersonExists(String pesel) {
        return repository.isPersonExists(pesel);
    }
}
```

- Kod tej klasy możemy być wykorzystywany wielokrotnie
- Poprawia czytelności kodu testu
- Wiele parametrów psuje czytelność - jednym z rozwiązań jest stworzenie klasy przechowującej wszystkie wymagane zależności

```
@Test
public void isPersonExists_PersonExists_ReturnTrue1() {
    HotelReservation1 reservation = new HotelReservation1(new AlwaysExistsPersonFakeRestRepository());

    boolean result = reservation.isPersonExists("abc");

    assertTrue(result);
}
```

Testy jednostkowe

Stuby - wstrzykiwanie zależności

Przekazanie interfejsu na poziomie gettera lub settera

```java
public class HotelReservation2 {

    private RestRepository repository;

    public RestRepository getRepository() {
        return repository;
    }

    public void setRepository(RestRepository repository) {
        this.repository = repository;
    }

    public boolean isPersonExists(String pesel) {
        return repository.isPersonExists(pesel);
    }
}
```

```java
@Test
public void isPersonExists_PersonExists_ReturnTrue2() {
    HotelReservation2 reservation = new HotelReservation2();
    reservation.setRepository(new AlwaysExistsPersonFakeRestRepository());
    boolean result = reservation.isPersonExists("abc");

    assertTrue(result);
}
```

Testy jednostkowe

Stuby - wstrzykiwanie zależności

Wykorzystanie sparametryzowanych fabryk

```java
public class HotelReservation3 {

    private RestRepository repository;

    public HotelReservation3() {
        this.repository = Factory.createRepository();
    }

    public boolean isPersonExists(String pesel) {
        return repository.isPersonExists(pesel);
    }
}
```

```java
public class Factory {

    private static RestRepository restRepository;

    public static void setRepository(RestRepository repository) {
        restRepository = repository;
    }

    public static RestRepository createRepository() {
        return restRepository;
    }
}
```

```java
@Test
public void isPersonExists_PersonExists_ReturnTrue3() {
    Factory.setRepository(new AlwaysExistsPersonFakeRestRepository());

    HotelReservation3 reservation = new HotelReservation3();

    boolean result = reservation.isPersonExists("abc");

    assertTrue(result);
}
```

- Test konfiguruje fabrykę, aby ta zwróciła namiastkę.
- Testowana klasa wykorzystuje fabrykę w celu otrzymania egzemplarza

Testy jednostkowe

Mocki

- Mock są podobne do namiastki - zastępuje obiekt, dzięki czemu możemy bezproblemowo przetestować inny obiekt. Określa zachowanie obiektu, który imituje. Dodatkowo daje możliwość weryfikacji.
- Podstawową różnicą między mockiem, a namiastką jest to, że namiastki nie mogą doprowadzić do niepowodzenia testu, natomiast mocki tak

- Dodatkowe założenie przypadku testowego:
  - Posiadamy namiastkę dla obiektu RestRepository, która sprawdza czy dana osoba istnieje w bazie. Dodajmy mocka, który prócz sprawdzania czy dana osoba istnieje w systemie, ma możliwość jej dodania lub usunięcia

Testy jednostkowe

Mocki

```java
public class HotelReservation {

    private RestRepository repository;

    public HotelReservation(RestRepository repository) {
        this.repository = repository;
    }

    public void addNewPerson(String pesel) {
        if (!repository.isPersonExists(pesel)) {
            this.repository.addNewPerson(pesel);
        }

    }

    public void removePerson(String pesel) {
        if (repository.isPersonExists(pesel)) {
            this.repository.removePerson(pesel);
        }
    }
}
```

```java
public class MockRestRepository implements RestRepository {

    private ArrayList<String> list;

    public MockRestRepository() {
        this.list = new ArrayList<>();
    }

    @Override
    public boolean isPersonExists(String pesel) {
        for (String p : list) {
            if (p.equals(pesel)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public void addNewPerson(String pesel) {
        this.list.add(pesel);
    }

    @Override
    public void removePerson(String pesel) {
        this.list.remove(pesel);
    }

}
```

Testy jednostkowe

Mocki

```java
@Test
public void isPersonExists_PersonExists_ReturnTrue1() {
    MockRestRepository mock = new MockRestRepository();
    HotelReservation reservation = new HotelReservation(mock);

    boolean result = mock.isPersonExists("abc");
    assertFalse(result);

    reservation.addNewPerson("abc");
    result = mock.isPersonExists("abc");
    assertTrue(result);

    reservation.removePerson("abc");
    result = mock.isPersonExists("abc");
    assertFalse(result);

}
```

- Mock w przeciwieństwie do namiastki zmienia stan dlatego test może zakończyć się nie powodzeniem
- Możemy wykorzystywać w jednym teście więcej niż jedną namiastkę, ale większa ilość mocków niż jeden może być problematyczna gdyż testujemy więcej niż jedną rzecz

Testy jednostkowe

Zadanie 1

- Napisać program realizujący zadania dodawania, odejmowania, mnożenia, dzielenia,potęgowania i pierwiastkowania wraz testami

Testy jednostkowe

Zadanie 2

- Napisać grę w życie
  - Gra odbywa się w turach na planszy gdzie każde pole może być "żywe" albo "martwe". Stan pola może zmienić się po turze w zależności od stanu jego sąsiadów
  - jeżeli żywa komórka ma mniej niż 2 sąsiadów ginie
  - jeżeli żywa komórka ma 2 albo 3 żywych sąsiadów przeżyje do następnej tury
  - jeżeli żywa komórka ma więcej niż 3 sąsiadów ginie
  - jeżeli martwa komórka ma dokładnie 3 sąsiadów, ożywa
- Pokryć ją testami jednostkowymi