



Testowanie oprogramowania TDD

Szymon Kociuba

Autor:

Prawa do korzystania z materiałów posiada Software Development Academy

Witajcie na zajęciach testowania oprogramowania



Prowadzący te zajęcia:

- Szymon Kociuba
- szymon.kociuba@gmail.com
- Kim jestem
 - Pracuję jako programista.
 - Studiowałem fizykę
 - Hobby: programowanie i testowanie (automatyzacja), tenis ziemny, sporty motorowe



Plan na dziś

- Czym jest testowanie
- Testowanie jednostkowe - JUnit
- Biblioteki matcherów
- Testy parametryzowane
- Testowanie wyjątków
- Testy integracyjne – Mockito
- TDD



Czym jest testowanie

Proces związany z wytwarzaniem oprogramowania.

Jest to jeden z procesów zapewnienia jakości oprogramowania.

Testowanie ma na celu **weryfikację** oraz **validację** oprogramowania.

Źródło: Wikipedia



Jakość oprogramowania - ogół cech produktu decydujących o jego zdolności do zaspakajania stwierdzonych lub przewidywanych potrzeb.



Testowanie może ujawnić obecność błędów, ale nigdy ich braku



Znane przypadki

Gdy brak testowania/niewystarczające
testowanie doprowadziło do awarii



Airbus 320 – 290 zabitych

Zestrzelenie samolotu Airbus 320 – lipiec 1988, krążownik USS Vincennes patrolował wody zatoki Perskiej z racji narzuconego embargo przez Stany Zjednoczone na Iran, został zaatakowany przez łodzie Irańskie. Odpowiedział ogniem, niestety w międzyczasie przelatywał pasażerski samolot Airbus 320. Z powodu defektu w systemie śledzenia obiektów samolot został uznany za wrogi wojskowy samolot F-14 i został zestrzelony. Zginęli wszyscy.

Źródło: B. Wiszniewski, B. Bereza-Jarociński, *Teoria i Praktyka testowania programów*, Warszawa 2006



Katastrofa rakiety Ariane 5

40 sekund po startie zmieniła kurs lotu, rozpadła się na części i wybuchła z powodu błędu przepełnienia bufora (cache'a).

Błąd nie został odkryty, ponieważ nie została przetestowana odpowiedzialna za to funkcja. Myślenie, że przecież dotychczas działała bezbłędnie – jest najbardziej mylnym założeniem, ale niestety popularnym.

Budowa ~ 7 000 000 000\$, Straty ~ 500 000 000\$

Źródło: B. Wiszniewski, B. Bereza-Jarociński, *Teoria i Praktyka testowania programów*, Warszawa 2006



Śmierć człowieka – Therac-25

Maszyna do radioterapii Therac-25 w latach 80. Na skutek błędnego działania oprogramowania przynajmniej pięciu pacjentów otrzymało zbyt duże dawki promieniowania, co było bezpośrednią przyczyną ich śmierci.

Na etapie badania przyczyn wypadku (post mortem) okazało się, że producent w celu ograniczenia kosztów zrezygnował z implementacji w tym produkcie niektórych mechanizmów. Na skutek sytuacji wyścigu, czyli błędnej implementacji współbieżności zadań, podawała nadmierne dawki promieniowania chorym.

Źródło: <https://en.wikipedia.org/wiki/Therac-25>



Podział testów według sposobu przeprowadzania

- Testy jednostkowe
- Testy integracyjne
- Testy automatyczne
- Testy manualne
- Testy systemowe
- Testy akceptacyjne
 - w fazie rozwoju funkcji
 - po wdrożeniu funkcji



Podział testów według sposobu przeprowadzania

- **Testy jednostkowe**
najtańsze, umiarkowanie skuteczne, powtarzanie testu “za darmo” (piszesz raz)
- **Testy integracyjne**
nadal dosyć tanie, skuteczne, powtarzanie testu “za darmo” (piszesz raz)
- **Testy automatyczne**
droższe od powyższych, ale nadal w miarę tanie, dobrze napisane w pełni skuteczne, powtarzanie testu “za darmo” (piszesz raz)
- **Testy manualne**
 - w fazie rozwoju funkcji - droższe, do powtórzenia testu konieczne przeklikanie
 - po wdrożeniu funkcji - najdroższe



Jaki powinien być idealny test

- Powtarzalny (patrz również punkt niżej)
- Wyizolowany:
- Nie opiera się na danych z systemów/zasobów zewnętrznych - nie zakładamy, że próbując pobrać www.onet.pl zawsze otrzymamy stronę internetową, co jeśli testy będą przeprowadzane na urządzeniu bez internetu?
- Powinien sprawdzać wszystkie przypadki brzegowe
- Pokrycie kodu testami nie musi być równe 100%



Test jednostkowy - przykład



Źródło: <http://www.rpmgo.com/automotive-articles/car-parts.html>



Test integracyjny - przykład



Źródło: <http://indianautosblog.com/2008/12/iihs-suzuki-sx4-crash-test>

Jaki powinien być idealny test - jednostkowy i integracyjny



Test jednostkowy

- najczęściej testuje pojedynczą funkcję
- powinien być szybki
- nie inicjalizuje bazy danych, nie ściąga danych z sieci itd., bazuje na mockach (co to?)

Test integracyjne

- sprawdza czy kilka komponentów/klas/systemów współpracują ze sobą
- może testować wybrany przypadek biznesowy/wymaganie
- nie musi być szybki, może uruchamiać bazę danych, uruchamiać serwer www onetu itd.



Co zapewniają dobrze napisane testy

- Rola ucząca
- Częściowa ochrona przed regresją
- Bezpieczna refaktoryzacja
- Szybkie testowanie zakodowanych rzeczy (żeby uruchomić całą aplikację/system często potrzebnych jest kilka minut)



Biblioteka wspomagająca pisanie testów, zapewnia między innymi assertje, wspomaga przygotowywanie i wygaszanie środowiska (np. zamykanie połączenia z bazą danych)



JUnit- adnotacje

@Test – występuje przed każdą metodą testową i oznacza pojedynczy test jednostkowy.

@BeforeClass – występuje przed metodą, która jest wykonywana przed uruchomieniem testów. Można w niej utworzyć obiekt klasy testowej, zainicjować jej atrybuty itp.

@AfterClass – oznacza metodę, która ma być wywołana po zakończeniu wszystkich testów. Można w niej np. zamknąć połączenie z bazą danych, zapisać plik itp.

@Before – metoda oznaczona tą adnotacją, będzie wywoływana przed wykonaniem każdej metody testowej.

@After – oznacza metodę, która ma być wywoływana po każdej metodzie testowej

@Ignore – oznacza test który będzie ignorowany



JUnit- asercje

Asercje są to metody statyczne pozwalające na porównanie wyników działania określonej funkcji z oczekiwana wartością. Najpopularniejsze asercje to:

Asercja	Opis
assertEquals(expected, actual)	Porównanie dwóch wartości – otrzymanej po działaniu i oczekiwanej
assertNotNull(object)	Sprawdzenie czy obiekt został zainicjalizowany
assertNull(object)	Sprawdzenie czy obiekt nie został zainicjalizowany
assertTrue(value)	Sprawdzenie czy pole lub metoda zwraca wartość true w przypadku typu boolean
assertFalse (value)	Sprawdzenie czy pole lub metoda zwraca wartość false w przypadku typu boolean
assertArrayEquals(expected, resultArray)	Porównanie dwóch tablic czy posiadają tą samą zawartość



JUnit- adnotacje

1. Nowa klasa testowa
2. Testowany obiekt
3. Metoda poprzedzona
adnotacją @Before będzie
wywołana przed każdym
testem
4. Metoda testująca dodawanie
5. Assercja sprawdzająca czy
otrzymana wartość po
operacji dodawania jest
wartością oczekiwana

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        int result = calculator.add(3, 5);  
        assertEquals(8, result);  
    }  
}
```



Testy jednostkowe – wskazówka

Popularnym „stylem” pisania testów jest podejście ***Given – When – Then.***

Dzięki temu testy są usystematyzowane oraz bardziej czytelne. Bardzo wiele osób dodaje także komentarze oddzielające opisywane sekcje.

- **Given** określa sekcję w której tworzymy początkowe założenia. Ustawiamy stan, tworzymy instancję obiektów, ładujemy konfigurację itp. na potrzebny do testów.
- **When** określa sekcję w której wykonujemy akcję którą chcemy testować.
- **Then** określa sekcję w której wykonujemy sprawdzenia czy aplikacja zachowała się zgodnie z oczekiwaniami. Najczęściej poprzez wykorzystanie asercji lub interakcji z mockami.

```
@Test  
public void testAdding() {  
    //Given  
    Calculator calculator = new Calculator();  
  
    //When  
    int result = calculator.add(3, 5);  
  
    //Then  
    Assert.assertEquals(8, result);  
}
```



Zadanie 1

Otwórz zadanie1.pdf



Przerywnik – rozmowa rekrutacyjna

1. Czym są magic strings, czy powinniśmy je zostawiać w kodzie?
2. Co to code review?
3. Jakie są metody inicjowania obiektów klasy String
4. Czym jest POJO?
5. Jak posortować Listę?
6. Czym jest „String Intern”?
7. Po co jest StringBuilder, StringBuffer?
8. Mając komputer z 512 kB RAM i 2 TB cyfr na dysku w przedziale 1-1000 jakiego algorytmu do ich posortowania użyjesz?
10. Jaki jest kontrakt pomiędzy equals, a hashCode?



Biblioteki matcherów



Czym są i po co używać

Biblioteki matcherów powstały aby poprawić jakość oraz czytelność pisanych przez nas testów, największe ich zalety to:

1. Poprawa czytelności oraz przejrzystości testów (możemy je odczytywać jakby były zapisane normalnym językiem)
2. Łatwiejsza analiza napisanych testów
3. Testy mogą być bardziej dokładne



AssertJ – przykłady

Sprawdzenie czy wartość pola name obiektu frodo równa się „Frodo”

Sprawdzenie obiekt frodo jest tym samym obiektem co obiekt sauron.

```
// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
    .endsWith("do")
    .isEqualToIgnoringCase("frodo");
```

Test bardziej złożony pozwalający sprawdzić wiele różnych warunków.

Sprawdzone zostało czy wartość pola name obiektu frodo rozpoczyna się od „Fro”, kończy na „do”. Dodatkowo sprawdzone zostało czy nazywa się „frodo” jednakże przy porównywaniu ignorowane są wielkości liter.



AssertJ – przykłady

```
assertThat(fellowshipOfTheRing).hasSize(9)
    .contains(frodo, sam)
    .doesNotContain(sauron);

// as() is used to describe the test and will be shown before the error message
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(33);

// Java 8 exception assertion, standard style ...
assertThatThrownBy(() -> { throw new Exception("boom!"); }).hasMessage("boom!");
```

Sprawdzenie czy lista fellowshipOfTheRing zawiera 9 elementów a w śród nich obiekty frodo oraz sam jak również nie zawiera obiektu sauron.

Sprawdzenie wieku frodo. Nowością jest metoda **as** dzięki czemu dodany jest komunikat który będzie wyświetlony w momencie niepowodzenia się testu.

Ciekawostka!
Testowanie wyjątków oraz komunikatów w Javie 8 z wykorzystaniem AssertJ.



Zadanie 2

Otwórz zadanie2.pdf



Testy parametryzowane



Testy parametryzowane

W jaki sposób przetestować określoną funkcjonalność na podstawie wielu różnych danych wejściowych? Np:

1. Danych poprawnych
2. Danych błędnych
3. Skrajne przypadki
4. Itd...

Jakie błędy i problemy może powodować takie podejście?

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup(){  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testAdding() {  
        Assert.assertEquals(8, calculator.add(3, 5));  
        Assert.assertEquals(0, calculator.add(1, -1));  
        Assert.assertEquals(0, calculator.add(0, 1));  
        Assert.assertEquals(0, calculator.add(5, -5));  
        Assert.assertEquals(-5, calculator.add(-10, 5));  
        Assert.assertEquals(1, calculator.add(3, -2));  
    }  
}
```



Testy parametryzowane

1. Bardzo wiele asercji w tej samej metodzie testującej.
2. Możemy wydzielić asercje dla każdego przypadku ale wówczas powstanie nam duża nadmiarowość kodu w postaci wielu testów.
3. W razie niepowodzenia nie wiemy w którym miejscu test się nie powiodł.



Testy parametryzowane

1. Parametryzowane testy jednostkowe służą do testowania tego samego kodu w różnych warunkach.
2. Dzięki parametryzowanym testom możemy sprawdzić poprawność działania określonej metody, dla wielu różnych danych wejściowych.
3. Ogólną ideą jest łatwiejsze testowanie różnych warunków przy użyciu tej samej metody testowania, co ogranicza ilość pisanej kodu i sprawia, że kod testowy jest bardziej czytelny i spójny.



Testy parametryzowane

Do pisania testów parametryzowanych można wykorzystać gotowe narzędzia. Np.

1. Parametrized

- 1. Wbudowane narzędzie w JUnita
- 2. Nie potrzebujemy dodatkowych bibliotek
- 3. Więcej informacji i dokumentacja:
- <http://junit.org/junit4/javadoc/4.12/org/junit/runners/Parameterized.html>

2. JUnitParams

- 1. Projekt polskiej firmy Pragmatics
- 2. Łatwa i szybka integracja z JUnitem
- 3. Dużo więcej możliwości i czytelniejsze testy
- 4. Więcej informacji i dokumentacja:
- <https://github.com/Pragmatists/JUnitParams>



Testy parametryzowane

Definiowanie testu parametryzowanego

Zmienna przechowująca wartości kolejnych parametrów

Utworzenie konstruktora przez który przekazywane będą wartości do kolejnych wywołań testowych

Adnotacja @Parameters definiuje metodę z parametrami do kolejnych testów

Zdefiniowanie tablicy obiektów która zawiera listę przypadków/wartości do przetestowania

Wartość przekazana z metody getParameters()
jako kolejny przypadek do testowania

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

    private Calculator calculator;
    private Integer value;

    public CalculatorTest(Integer value) {
        this.value = value;
    }

    @Parameters
    public static Collection getParameters() {
        return Arrays.asList(
            new Integer[][] {
                { 1000 },
                { 2000 },
                { 3000 }
            });
    }

    @Test
    public void testAdding() {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```



Testy parametryzowane

Definiowanie testu parametryzowanego

```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTest {

    private Calculator calculator;

    @Test
    @Parameters({ "10", "20", "30" })
    public void testAdding(int value) {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```

Adnotacja @Parameters zawierająca kolejne wartości do przetestowania

Wartość przekazana z metody adnotacji jako kolejny przypadek do testowania

Wyniki działania testu parametryzowanego

Finished after 0,018 seconds

Runs: 3/3 Errors: 0 Failures: 0

pl.lbednarski.test.CalculationDate.CalculatorTest

testAdding (0,000 s)

- testAdding(10) [0] (0,000 s)
- testAdding(20) [1] (0,000 s)
- testAdding(30) [2] (0,000 s)



Testy parametryzowane

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

    private Calculator calculator;
    private Integer value;

    public CalculatorTest(Integer value) {
        this.value = value;
    }

    @Parameters
    public static Collection getParameters() {
        return Arrays.asList(
            new Integer[][] {
                { 1000 },
                { 2000 },
                { 3000 }
            });
    }

    @Test
    public void testAddition() {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```

```
@RunWith(JUnitParamsRunner.class)
public class CalculatorTest {

    private Calculator calculator;

    @Test
    @Parameters({ "10", "20", "30" })
    public void testAddition(int value) {
        calculator = new Calculator();

        int result = calculator.add(value, value);

        assertEquals(value + value, result);
    }
}
```



Testy parametryzowane

Główne ulepszenia dostępne w JUnitParams w stosunku do Parametrized:

- Bardziej jawne – przypadki testowe są w parametrach metody testowej, a nie w polach klasy.
- Mniej kodu - nie potrzebujesz konstruktora do ustawiania parametrów.
- Można mieszać parametryzowane z nieparametryzowanymi metodami w jednej klasie.
- Parametry można przekazać pobierając je z pliku CSV lub klasy dostawcy parametrów.
- Klasa dostawcy parametrów może mieć tyle parametrów, ile ma być dostarczone, tak aby można było grupować różne przypadki



Zadanie 3

Otwórz zadanie3.pdf



Testowanie wyjątków



Testowanie wyjątków

Wyjątki w Javie to specjalne obiekty, które poza standardowymi operacjami na obiektach możemy także rzucać za pomocą słowa kluczowego throw, co powoduje natychmiastowe przerwanie działania wątku (w najprostszym przypadku – aplikacji) oraz przejście do pierwszego napotkanego miejsca, które ten wyjątek jest w stanie obsłużyć. Nieobsłużony wyjątek uśmierca bieżący wątek.



Testowanie wyjątków

- NullPointerException – rzucany kiedy próbujesz wywołać metodę nazmiennej, której wartość to null
- IllegalArgumentException – rzucany, kiedy przekazywany argument jest z jakiegoś powodu nieprawidłowy (walidacja wewnątrz metod)
- IOException (wyjątki po nim dziedziczące) – rzucany w przypadku problemów z systemem wejścia/wyjścia, np. kiedy wystąpi problem przy pracy z plikami lub z transmisją danych za pośrednictwem internetu
- NumberFormatException – rzucany, kiedy próbujemy zamienić na liczbę np. obiekt typu String, który zawiera nie tylko cyfry
- IndexOutOfBoundsException – rzucany, kiedy próbujemy się odwołać do nieistniejącego elementu tablicy lub listy



Testowanie wyjątków

Po co testować wyjątki ?

1. Chcemy mieć pewność i świadomość że system poprawnie reaguje np. na błędne wartości (np. dzielenie przez 0).
2. Chcemy mieć pewność że podczas określonych problemów system „rzuca” wyjątki odpowiednich klas
3. Chcemy mieć pewność że wiadomość/informacja zawarta w rzuconym wyjątku jest poprawna

Przykładowe sposoby testowania wyjątków:

1. Try - catch
2. Słowo kluczowe – expected
3. Annotacja @Rule



Testowanie wyjątków

Przykład testowania z wykorzystaniem adnotacji expected. Minusem takiego podejścia jest fakt że nie mamy możliwości zweryfikowania przesłanego komunikatu.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test(expected = ArithmeticException.class)  
    public void testDivision() {  
        Double result = calculator.divide(new Double(1), new Double(0));  
        assertNull(result);  
    }  
}
```

Deklaracja testu który oczekuje
że podczas jego działania rzucony
zostanie wyjątek.

Wywołanie metody rzuca wyjątkiem
więc test zakończy się pozytywnie.

Dodatkowa asercja
sprawdzająca wynik metody.

WAŻNE! W parametrze expected należy podawać konkretną klasę wyjątku.
Nie należy więc umieszczać tam wyjątków typu Exception.class ponieważ taki
test staje się całkowicie bezużyteczny, pomimo iż pozornie może działać
poprawnie.



Testowanie wyjątków

Przykład testowania z wykorzystaniem adnotacji expected. Minusem takiego podejścia jest fakt że nie mamy możliwości zweryfikowania przesłanego komunikatu.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test(expected = ArithmeticException.class)  
    public void testDivision() {  
        Double result = calculator.divide(new Double(1), new Double(0));  
        assertNull(result);  
    }  
}
```

Deklaracja testu który oczekuje
że podczas jego działania rzucony
zostanie wyjątek.

Wywołanie metody rzuca wyjątkiem
więc test zakończy się pozytywnie.

Dodatkowa asercja
sprawdzająca wynik metody.

WAŻNE! W parametrze expected należy podawać konkretną klasę wyjątku.
Nie należy więc umieszczać tam wyjątków typu Exception.class ponieważ taki
test staje się całkowicie bezużyteczny, pomimo iż pozornie może działać
poprawnie.



Testowanie wyjątków

Konstrukcja try - catch służy do obsługi wyjątków w Javie w trakcie działania programu. Dzięki temu możliwe jest wyłapywanie wyjątków i obsługa nieoczekiwanej błędu. Możemy też try-catch wykorzystać do przeprowadzenia testów jednostkowych.

Wywołanie metody divide w bloku try.
„Złapanie” rzuconego wyjątku.
Możliwość sprawdzenia poprawności
wiadomości dodanej do wyjątku.

```
@Test
public void testDivision() {
    Double result = null;

    try {
        result = calculator.divide(new Double(1), new Double(0));
    } catch (ArithmetricException exception) {
        assertNull(result);
        assertEquals(exception.getMessage(), "Incorect value");
    }
}
```

Finished after 0,02 seconds

Runs: 1/1 Errors: 0 Failures: 1

pl.lbednarski.test.CalculationDate.CalculatorTest [Runner: JUnit 4]
testDivision (0,000 s)

Failure Trace

org.junit.ComparisonFailure: expected:<Incor[r]ect value> but was:<Incor[]ect value>
at pl.lbednarski.test.CalculationDate.CalculatorTest.testDivision(CalculatorTest.java:2)



Testowanie wyjątków

Adnotacja Rule pojawiła się w JUnit dopiero w wersji 4.7
Służy do bardziej rozbudowanego testowania wyjątków.

Utworzenie obiektu klasy ExpectedException z
adnotacją @Rule do wyłapywania wyjątków.

Zdefiniowanie jakiej klasy wyjątku oczekujemy
oraz jaką wiadomość powinien zawierać.

Wywołanie metody divide która rzuca wyjątkiem.

Dodatkowa asercja sprawdzająca czy wynik na
pewno nie został zwrócony.

```
public class CalculatorTest {  
  
    private Calculator calculator;  
  
    @Rule  
    public ExpectedException thrown = ExpectedException.none();  
  
    @Before  
    public void setup() {  
        calculator = new Calculator();  
    }  
  
    @Test  
    public void testDivision() {  
        thrown.expect(ArithmeticException.class);  
        thrown.expectMessage("Incorrect value");  
  
        Double result = calculator.divide(new Double(1), new Double(0));  
  
        assertNull(result);  
    }  
}
```



Zadanie 3

Otwórz zadanie3.pdf



Mockowanie

Autor: Prawa do korzystania z materiałów posiada Software
Development Academy



Mockowanie - definicje

„Atrapa obiektu (ang. mock object) – symulowany obiekt, który w kontrolowany sposób naśladuje zachowanie rzeczywistego obiektu. Programista tworzy zazwyczaj atrapy obiektów w celu przetestowania zachowania jakiegoś innego obiektu, podobnie jak projektanci samochodów wykorzystują manekiny do symulacji dynamiki zachowania ludzkiego ciała podczas zderzenia pojazdów.”



Mockowanie - zalety

1. Atrybuty obiektów posiadają identyczny interfejs jak obiekty które naśladują
2. Możemy symulować wiele różnych scenariuszy zachowań
3. Atrapa może być wielokrotnie uruchamiana dzięki czemu możemy sprawdzić czy metody zawsze zwracają ten sam wynik
4. Nie ma potrzeby tworzenia i przygotowywania wielu obiektów
5. Testowana klasa może być uruchamiana w wielu różnych konfiguracjach co pozwala na bardziej elastyczne testy.



Mockowanie - zalety

Czy metoda została wywołana?

```
import static org.mockito.Mockito.*;  
  
// mock creation  
List mockedList = mock(List.class);  
  
// using mock object - it does not throw any "unexpected interaction" exception  
mockedList.add("one");  
mockedList.clear();  
  
// selective, explicit, highly readable verification  
verify(mockedList).add("one");  
verify(mockedList).clear();
```

Tworzenie mocka - atrapy

Wywołanie metod

Sprawdzenie czy metody zostały wcześniej wywołana

Wynik testu JEŚLI metoda add nie zostałaby wywołana z parametrem „one”

Finished after 1,205 seconds

Runs: 1/1 Errors: 0 Failures: 1

pl.sdacademy.Calculator.CalculatorTest [Runner: JUnit 4] (1,161 s) Failure Trace

isAddedToList (1,161 s)

Wanted but not invoked:
list.add("one");



Mockowanie - zalety

Weryfikowanie ilości wywołań metody

```
//using mock
mockedList.add("once");

mockedList.add("twice");
mockedList.add("twice");

mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//following two verifications work exactly the same - times(1) is used by default
verify(mockedList).add("once");
verify(mockedList, times(1)).add("once");

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("three times");
verify(mockedList, atMost(5)).add("three times");
```

Verify pozwala między innymi na bardzo konkretne zweryfikowanie ilości wywołań określonej metody wraz z przekazywanym argumentem.

Czy metoda add z parametrem „once” wywołana była tylko raz?

Czy metoda add z parametrem „twice” wywołana była tylko raz a z parametrem „tree times” 3 razy?

Czy metoda add z parametrem „never happened” nie była wywołana ani razu?

Bardziej dokładne testowanie wywołań metod, np. sprawdzenie czy metoda została wywołana jako ostatnia



Mockowanie - zalety

Zwracanie konkretnej wartości po wywołaniu metody

```
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(0)).thenReturn("first"); ↔ //stubbing using built-in anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("element");
when(mockedList.get(anyInt())).thenReturn("element");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

Bardzo przydatna funkcjonalność podczas symulowania konkretnego scenariusza, np. kiedy chcemy aby nasza metoda zwracająca wartości losowe, w teście zwracała konkretne wartości, np. mniejsze niż 0.



Mockowanie - zalety

Główne anotacje wykorzystywane podczas korzystania z JUnit i Mockito

```
public class Person {  
    private Car car;  
  
    public Person(Car car) {  
        this.car = car;  
    }  
}
```

```
@RunWith(MockitoJUnitRunner.class)  
public class SampleTest {  
  
    @Mock  
    Car car;  
  
    Person person;  
  
    @Before  
    public void setUp() {  
        person = new Person(car);  
    }  
}
```

```
@RunWith(MockitoJUnitRunner.class)  
public class SampleTest {  
  
    @Mock  
    Car car;  
  
    @InjectMocks  
    Person person;  
}
```

Anotacja dostarczająca tzw. Runner który pozwala korzystać z funkcjonalności Mockito

Anotacja tworząca atrapę obiektu klasy Car

Anotacja tworząca obiekt klasy person oraz wstrzyknięcie do niej wszystkich wymaganych obiektów.

Zamiast anotacji `@InjectMocks` możemy obiekt klasy Car wstrzyknąć za pomocą konstruktora



Test Driven Development



TDD - definicja

„Test-driven development (TDD) – technika tworzenia oprogramowania, zaliczana do metodyk zwinnych. Pierwotnie była częścią programowania ekstremalnego (ang. extreme programming), lecz obecnie stanowi samodzielna technikę (...)

Programowanie techniką test-driven development wyróżnia się tym, że najpierw programista zaczyna od pisania testów do funkcji, która jeszcze nie została napisana. Na początku testy mogą nawet się nie kompilować, ponieważ może nie być jeszcze elementów kodu (metod, klas) które są w testach użyte. Na początku zaczyna się od przypadku, który nie przechodzi testu – zapewnia to, że test na pewno działa i może wyłapać błędy.”

https://pl.wikipedia.org/wiki/Test-driven_development



TDD - definicja

Test Driven Development (TDD) to metodyka tworzenia oprogramowania, w której testy tworzy się jeszcze przed napisaniem właściwego kodu. W TDD cykl życia oprogramowania wygląda następująco:

- 1) Napisanie testów do (jeszcze) nieistniejącej funkcjonalności.
- 2) Uruchomienie testów – na tym etapie testy nie powinny się powieść.
- 3) Napisanie właściwego kodu, w założeniu przechodzącego testy.
- 4) Uruchomienie testów i upewnienie się, że się powodzą.
- 5) Refaktoryzacja kodu w celu poprawy jego jakości.

Powyższe kroki powtarza się dla kolejnych funkcjonalności.



TDD - definicja

Czym TDD nie jest?

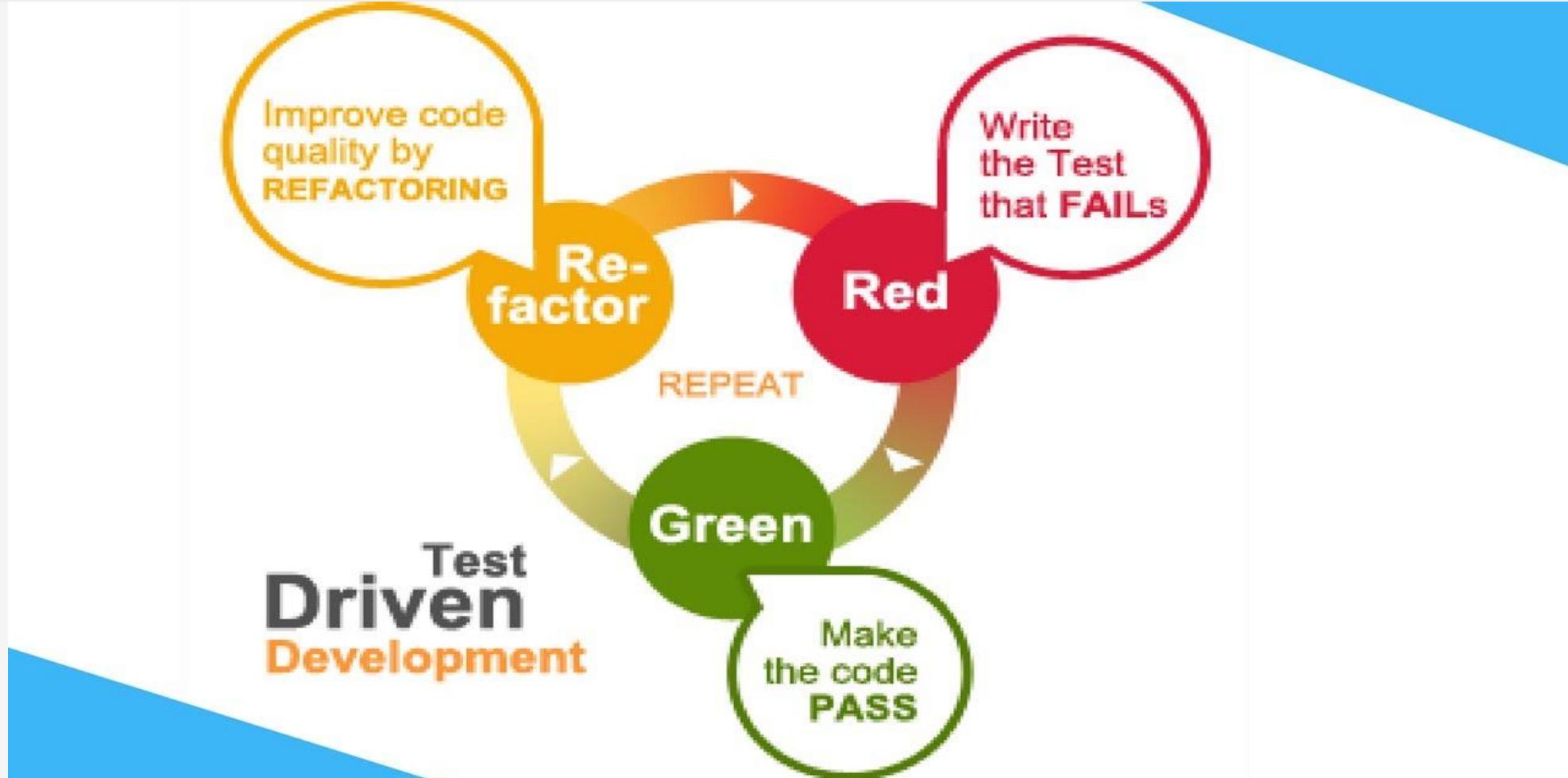
1. Nie jest to technika służąca do pisania testów
2. Głównym celem Test Driven Development pomimo nazwy nie jest pisanie testów
3. Nie chodzi także o wyeliminowanie potrzeby pracy testera

Więc czym jest TDD?

Procesem tworzenia oprogramowania, który opiera się na powtarzaniu krótkich cykli rozwoju, w trakcie których najpierw tworzymy test a później dopiero konkretną implementację. Dzięki takiemu podejściu kod jest (powinien być) przede wszystkim bardziej przemyślany, mniej złożony i krótszy ponieważ implementujemy tylko tyle, ile potrzebujemy aby test był zakończony sukcesem. „Efektem ubocznym” jest duże pokrycie kodu testami.



TDD - definicja



Zródło: <https://twitter.com/TechPrimers>



Zadanie 5

Otwórz zadanie5.pdf