TESTY JEDNOSTKOWE

AssertJ - Mockito - TDD

TESTY

Ani połowy spośród was nie znam nawet do połowy tak dobrze, jak bym pragnął; a mniej niż połowę z was lubię o połowę mniej, niż zasługujecie.

J.R.R. Tolkien, Drużyna Pierścienia

Materiały do warsztatów

https://github.com/infoshareacademy/jjdz8-materialy-junit.git

Życie zostawało za nim we mgle, a przed nim była ciemność i przygoda.

J.R.R. Tolkien, Drużyna Pierścienia

Tak rozpoczęła się nasza przygoda z testami jednostkowymi...

- wiemy czym jest test jednostkowy
- znamy lokalizację testów w projekcie
- znamy podstawowe biblioteki do testów
- potrafimy nadać testom odpowiednie nazwy i adnotacje
- znamy podstawowe asercje z pakietu Assertions
- wiemy jak testować wyjątki
- potrafimy sterować testami
- wiemy jak sprawdzić test coverage

Zadanie 1

- Zapoznaj się z działaniem klasy:
  - com.isa.strings.**StringReverseService**
- utwórz klasę testową **StringReverseServiceTest**
- utwórz metodę testującą odwracanie kolejności liter w wyrazie ***“gollum”***
- metoda powinna mieć adnotację **@DisplayName**
- metoda powinna mieć asercję **assertEquals** z pakietu org.junit.jupiter.api.Assertions

10 minut

Struktura testu

Zasada 3 x A


```
// given
User user = new User("admin");
                                    // when
                                    String login = user.getLogin();
                                                                        // then
                                                                        assertEquals("admin", login);
```

Zadanie 2

- Zapoznaj się z działaniem klasy:
  - com.isa.person.**Person**
- utwórz klasę testową **PersonTest**
- przetestuj działanie metody ***isAdult()***
- zastosuj w metodach testowych wzorzec **3xA**
- metody testowe powinny korzystać z asercji ***assertTrue*** oraz **assertFalse** z pakietu org.junit.jupiter.api.Assertions

10 minut

AssertJ

Zawsze powtarzałem, że jest w nim więcej niż wzrok potrafi dostrzec.

J.R.R. Tolkien, Drużyna Pierścienia

AssertJ

Po co nam kolejna biblioteka?

Czytelność warunków oraz konieczność używania uogólnionych, statycznych metod może często zniechęcić programistę do tworzenia bardziej rozbudowanych, skomplikowanych testów.

Bardzo częstym problemem jest mieszanie kolejności wartości oczekiwanej z aktualnie występującą.

```
assertEquals(expected, actual)
```

🤔

AssertJ

Co nam to daje?

AssertJ dla odmiany, realizuje model zwany fluent programming interface, który ma na celu wywoływania potoku metod, które zwracają kolejno referencję do obiektu. Pozwala to na płynne realizowanie testów bez oddzielnych wywołań statycznych metod.

```
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");
```

AssertJ

Co nam to daje?

Dodatkowo, na każdym etapie mamy dostęp tylko do tych metod, które są adekwatne do przetwarzanych danych.

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

// in the example below fellowshipOfTheRing is a  List<TolkienCharacter>
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

AssertJ

Maven dependency

```
<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.16.1</version>
    <scope>test</scope>
</dependency>
```

AssertJ

AssertJ vs JUnit

Nie!

JUnit + AssertJ

Pamiętajmy, że biblioteki te w żadnym wypadku się nie wykluczają ani nie zamieniają. Biblioteki te się **uzupełniają**.

AssertJ

Budowanie kryteriów

Każdy test rozpoczynamy od wskazania wartości rzeczywistej (aktualnej) jako argumentu metody assertThat:

```
Assertions.assertThat(value);
```

Dostępnej w pakiecie:

```
import static org.assertj.core.api.Assertions.*;
```

!!! Zwróć uwagę na pakiet, z którego pochodzą metody.

AssertJ

Łańcuch warunków

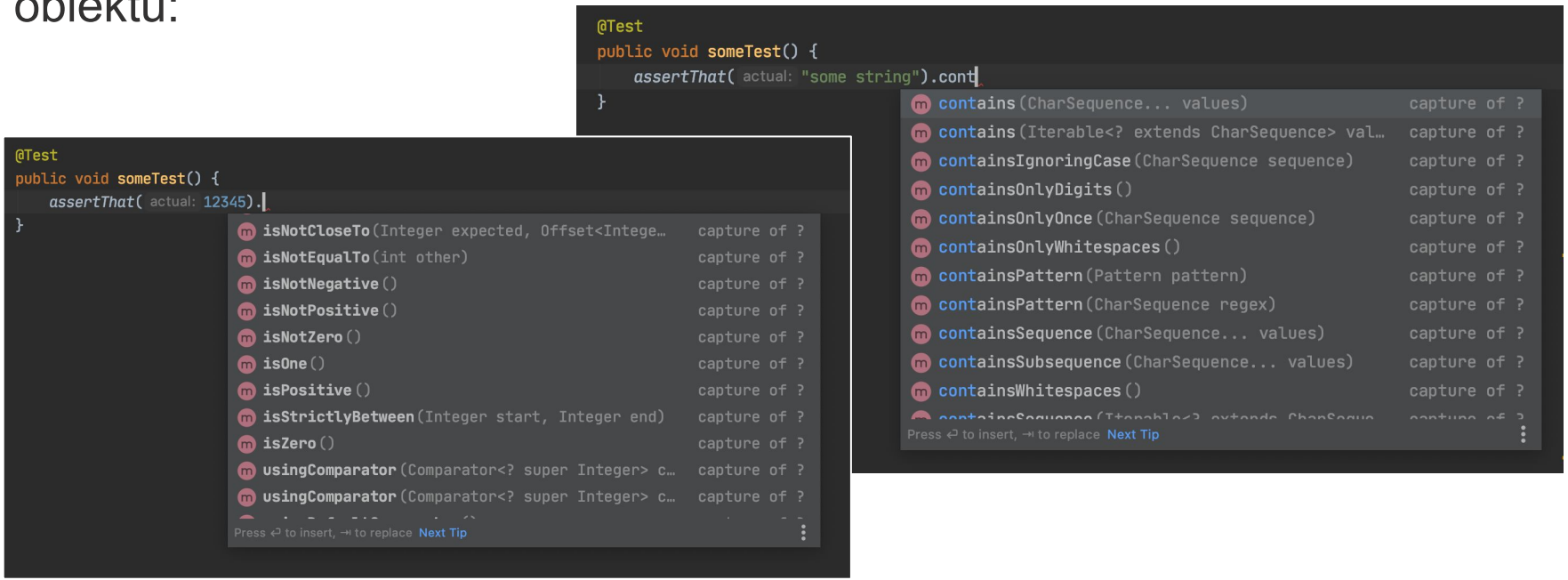
Każdy kolejny warunek zwraca obiekt stanowiący testowaną podstawę.
Dlatego też możliwy jest zapis:

```
assertThat("The Lord of the Rings").isNotNull()
                                   .startsWith("The")
                                   .contains("Lord")
                                   .endsWith("Rings");
```

AssertJ

Autodetekcja

IntelliJ wspiera wykrywanie dopuszczalnych metod dla typu testowanego obiektu:



Zadanie 3

- zapoznaj się z działaniem klasy:
  - com.isa.geometry.**Circle**
- w klasie testowej:
  - com.isa.geometry.**CircleTest**
- wykorzystując AssertJ dokończ metodę testującą poprawność wyliczania obwodu koła
- test powinien sprawdzać:
  - czy wynik jest liczbą
  - czy wynik jest równy

5 minut

Zadanie 4

10 minut

- wykorzystując AssertJ uzupełnij klasę testów:
  - com.isa.geometry.**CircleTest**
- dodaj test weryfikujący poprawność wyliczania pola powierzchni koła
- test powinien sprawdzać:
  - czy wynik jest liczbą
  - czy wynik jest liczbą dodatnią
  - czy wynik jest bliski z dokładnością 4-go miejsca po przecinku

Zadanie 5

- wykorzystując AssertJ uzupełnij klasę testów:
  - com.isa.geometry.**CircleTest**
- napisz test, który wykonuje obliczenia obwodu dla ujemnej wartości promienia
- wnioski? 😉

5 minut

AssertJ

Testowanie wyjątków

Asercje dla wyjątków:

```
assertThatThrownBy(() -> { throw new Exception("boom!"); })
            .isInstanceOf(Exception.class)
            .hasMessage("boom!");

assertThatExceptionOfType(IOException.class)
            .isThrownBy(() -> { throw new IOException("boom!"); })
            .withMessage("%s!", "boom")
            .withMessageContaining("boom")
            .withNoCause();
```

AssertJ

Testowanie wyjątków

Asercje dla popularnych wyjątków:

- `assertThatNullPointerException`
- `assertThatIllegalArgumentException`
- `assertThatIllegalStateException`
- `assertThatIOException`

```
assertThatIOException().isThrownBy(() -> { throw new IOException("boom!"); })
                       .withMessageContaining("boom");
```

AssertJ

Testowanie wyjątków

Możemy przetestować kod, który nie powinien rzucić wyjątku:

```
assertThatCode(() -> {
  new Circle(3.0).calculateArea();
}).doesNotThrowAnyException();
```

Zadanie 6

- wykorzystując AssertJ uzupełnij klasę testów:
  - com.isa.geometry.**CircleTest**

- dodaj testy weryfikujące wystąpienie wyjątku IllegalArgumentException dla pola powierzchni oraz obwodu koła w przypadku kiedy wartość promienia nie została ustawiona

- testy powinien sprawdzać:
  - typ wyjątku
  - wiadomość

10 minut

AssertJ

Niewłaściwe użycie aserecji

Największą pułapką jest umieszczenie obiektu w metodzie assertThat i pominięcie właściwej asercji:

❌ `assertThat(actual.equals(expected)); // to niczego nie weryfikuje`
✔️ `assertThat(actual).isEqualTo(expected);`
✔️ `assertThat(actual.equals(expected)).isTrue();`

❌ `assertThat(1 == 2); // to niczego nie weryfikuje`
✔️ `assertThat(1).isEqualTo(2);`
✔️ `assertThat(1 == 2).isTrue();`

Zadanie 7

10 minut

- zapoznaj się z działaniem klasy:
  - com.isa.sorter.**MapSorter**
- dodaj test sprawdzający czy sortowanie po wartości działa poprawnie
- testy powinien sprawdzać:
  - czy mapa jest prawidłowo posortowana po wartości
  - czy posortowana mapa zawiera te same elementy

Zadanie 8

10 minut

- do klasy:
  - com.isa.sorter.**MapSorter**
- dopisz metodę sortującą po kluczu
- dodaj test sprawdzający czy sortowanie po kluczu działa poprawnie
- testy powinien sprawdzać:
  - czy mapa jest prawidłowo posortowana po kluczu
  - czy posortowana mapa zawiera te same elementy
- wykorzystaj metody @BeforeEach/@AfterEach do sterowania testami

AssertJ

Filtrowanie kolekcji

Przy testowaniu kolekcji możemy dodatkowo wykorzystać filtrowanie przed asercją:

```
assertThat(fellowshipOfTheRing).filteredOn(character -> character.getName().contains("o"))
                               .containsOnly(aragorn, frodo, legolas, boromir);
```

Q&A

mockito

[...] niech nie ślubuje przebrnąć przez ciemności nocy, kto nie widział jeszcze zmroku.

J.R.R. Tolkien, Drużyna Pierścienia

mockito
framework

- mockito to framework, który znacznie ułatwia testowanie kodu
- najpopularniejszy framework do mockowania dla javy (ponad 2 mln użytkowników)

mockito
framework

- dzięki mockowaniu testy mogą być małe i sprawdzać wyizolowaną logikę
- możemy testować kod, który zależy od niedostępnych funkcjonalności (np. jeszcze niedziałających poprawnie) lub czasochłonnych (np. połączenie z bazą danych)

mockito
framework

- pozwala tworzyć obiekty zastępcze (mocki), co ułatwia testowanie
- umożliwia pisanie testów bez wykorzystywania zewnętrznych systemów – np. usług sieciowych
- testy są bardzo czytelne, wyizolowane od pozostałej logiki
- pozwala sprawdzać zachowanie metod, np. ilość wywołań danej metody, przekazane argumenty
- używany z JUnit

mockito

Podstawowe pojęcia

- **fake** – **uproszczona implementacja** obiektu, który chcemy zastąpić, np. baza danych w pamięci RAM
- **stub** – „**prawdziwy obiekt**”, który posiada implementację, ale można mu zmienić **konkretne zachowanie**, np. wybraną metodę
- **mock** – obiekt, który **nie posiada implementacji**, ale wymaga skonfigurowania specjalnie dla testu

mockito

Podstawowe pojęcia

- **spy** – działa jak proxy (pośrednik), część metod wywołuje na „prawdziwym obiekcie”, a część **symuluje**
- **dummy** – obiekt, który jest wykorzystywany jako argument tylko na potrzeby poprawnej kompilacji (nie jest używany w teście)

mockito

Maven dependency

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.3.3</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.3.3</version>
    <scope>test</scope>
</dependency>
```

Tworzenie mocka

Metoda statyczna mock()

Najprościej utworzyć mock przez wywołanie statycznej metody mock() w ciele metody

Parametrem jest klasa, która chcemy zamockować (np. UserDao.class)

```
private UserDao userDao;
private UserService userService;

@BeforeEach
public void setUp() {
    userDao = mock(UserDao.class);
    userService = new UserService(userDao);
}
```

Tworzenie mocka

Adnotacja @Mock

Możemy także użyć adnotacji **@Mock** do utworzenia mocka i **@InjectMock** do wstrzyknięcia mocka do klasy, która potrzebuje go jako zależności.

Należy pamiętać o adnotacji **@ExtendWith** na klasie z testami!!!

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserDao userDao;
    @InjectMocks
    private UserService userService;
```

Tworzenie mocka

Brak zdefiniowanego zachowania

Co się stanie, jeśli wywołamy na mocku metody bez wcześniejszego zdefiniowania ich zachowania?

```
@Test
public void returnFromMock() {
    User user = userDao.findUser("admin");
    System.out.println(user);

    List<User> users = userDao.getAllUsers();
    System.out.println(users);
}
```

Zadanie 1

10 minut

- zapoznaj się z działaniem klas:
  - com.isa.user.**UserService**
  - com.isa.user.**UserDao**

- dodaj klasę testową:
  - com.isa.user.**UserServiceTest**

- dodaj test sprawdzający zachowanie metod klasy UserSevice ze wstrzykniętym mockiem UserDao bez zdefiniowanego żadnego zachowania
  - co zwraca metoda getAllUsers?
  - co zwraca metoda findUsers?
  - co zwraca metoda findUser klasy UserDao?

Tworzenie mocka

Brak zdefiniowanego zachowania

Co się stanie, jeśli wywołamy na mocku metody bez wcześniejszego zdefiniowania ich zachowania?

```
@Test
public void returnFromMock() {
    User user = userDao.findUser("admin");
    System.out.println(user);

    List<User> users = userDao.getAllUsers();
    System.out.println(users);
}
```

domyślna wartość dla obiektu: **null**

domyślna wartość dla kolekcji: **pusta kolekcja []**

Stubbing

Definiowanie zachowania

Dla każdej metody mocka możemy zdefiniować zachowanie – tj. co dana metoda zwraca (wartość).

```
// given
List<User> users = Arrays.asList(
    new User("admin"), new User("janek")
);
when(userDao.getAllUsers()).thenReturn(users);
when(mock.metoda()).thenReturn(zwracanaWartość)
// when
List<User> result = userService.getAllUsers();

// then
assertThat(result).hasSize(2);
```

Zadanie 2

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- dodaj test ***shouldGetAllUsers()***:

  - zdefiniuj prawidłowe zachowanie mocka **userDao** dla metody getAllUsers() (powinna zwracać listę 4 różnych użytkowników)

  - zweryfikuj działanie metody **userService.getAllUsers()** (powinna zwrócić listę tych samych użytkowników)

Zadanie 3

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- dodaj test ***shouldReturnAllUsersMatchingPattern***:

  - test powinien weryfikować, czy metoda ***userService.findUsers()***, zwróci listę 2 użytkowników zawierających **“Ar**” w nazwie

  - zdefiniuj zachowanie mocka **userDao** dla metody *getAllUsers()*, tak aby zwracał listę kilku użytkowników, w tym dokładnie dwóch użytkowników zawierających **“Ar”** w nazwie

Stubbing

Definiowanie zachowania dla parametrów

Ta sama metoda może zwracać różne wartości w zależności od przyjmowanych argumentów. W tym celu stosuje się tzw. **ArgumentMatcher**.

```
// given
when(userDao.findUser(anyString())).thenReturn(new User("Janek"));
when(mock.metoda( ArgumentMatcher )).thenReturn( zwracana wartość )
// when
boolean result = userService.doesUserExist("Janek");

// then
assertThat(result).isEqualTo(true);
```

Stubbing

ArgumentMatchers

- pozwalają na zdefiniowanie zachowania dla mockowanej metody w zależności od parametrów wejściowych – przekazywanych przy wywołaniu metody
- zastępują parametry w wywołaniu metody when(mock.metoda(…))
- any(), anyBoolean(), anyString(), anyInt(), … - wartości dowolne (metoda przyjmie wszystko)
- eq(T value) – konkretna wartość zdefiniowana przez programistę

Stubbing

Definiowanie zachowania dla parametrów

Zwrócenie konkretnej wartości w zależności od parametru (***matcher eq()***):

```
// given
when(userDao.findUser(eq("Janek")).thenReturn(new User("Janek"));

// when
boolean result = userService.doesUserExist("Janek");

// then
assertThat(result).isEqualTo(true);
```

Zadanie 4

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- dodaj test ***shouldReturnTruelfUserExists***:

  - test powinien weryfikować, czy metoda ***userService.doesUserExist()***, zwraca prawdę dla użytkownika o loginie ***“admin”***

Zadanie 5

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- dodaj test ***shouldReturnFalseIfUserDoesNotExist***:

  - test powinien weryfikować, czy metoda ***userService.doesUserExist()***, zwraca fałsz dla użytkownika o loginie ***“frodo”***

5 minut

Stubbing

Rzucanie wyjątków

Jeśli dana metoda ma rzucić wyjątek, musimy użyć konstrukcji

doThrow(wyjątek).when(mock).nazwaMetody(parametry)

```
// given
doThrow(new RuntimeException()).when(userDao).deleteUser(any());

// when & then
assertThrows(RuntimeException.class,
    () -> userService.deleteUser(new User("test")));
```

Zadanie 6

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- dodaj test *shouldThrowExceptionWhenDeletingNonExistingUser:*

  - zdefiniuj zachowanie mocka **userDao**, aby metoda **deleteUser()** rzucała wyjątek **RuntimeException**

  - test powinien weryfikować, czy metoda ***userService.deleteUser()*** rzuca wyjątek **RuntimeException** przy próbie usunięcie użytkownika o loginie ***“sauron”***

Mockito

Weryfikacja wywołania

- Często zdarza się, że nasz kod nie zwraca konkretnych wyników tylko wywołuje kolejne metody z określoną liczbą parametrów (np. z innych klas).

- W celu przetestowania takiego kodu należy sprawdzić, czy parametry wywołania zewnętrznych metod są zgodne z oczekiwaniami.

ArgumentCaptor

Sprawdzanie parametrów

- Aby sprawdzić wartości parametrów przekazanych do mocka, należy skorzystać z klasy ArgumentCaptor
- Obiekt ArgumentCaptor tworzymy poprzez wywołanie statycznej metody ArgumentCaptor.forClass(klasa)

```
ArgumentCaptor.forClass(String.class);
```

ArgumentCaptor

Sprawdzanie parametrów

- W wywołaniu mock() w miejscu przekazania parametrów wywołujemy metodę ArgumentCaptor.capture()

```
when(userDao.findUser(loginCaptor.capture()))
        .thenReturn(new User("sam"));
```

- Po użyciu mocka należy sprawdzić wartość poprzez wywołanie ArgumentCaptor.getValue()

```
assertThat(loginCaptor.getValue()).isEqualTo("sam");
```

ArgumentCaptor

Sprawdzanie parametrów

```
// given
ArgumentCaptor<String> loginCaptor = ArgumentCaptor.forClass(String.class);
when(userDao.findUser(loginCaptor.capture())).thenReturn(new User("sam"));

// when
boolean result = userService.doesUserExist("sam");

// then
assertThat(result).isTrue();
assertThat(loginCaptor.getValue()).isEqualTo("sam");
```

Zadanie 7

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- rozszerz test ***shouldReturnTrueIfUserExists***

  - dodaj sprawdzenie parametrów przekazanych do metody **userDao.findUser()**

  - wykorzystaj klasę **ArgumentCaptor**

Metoda *verify()*

Sprawdzanie parametrów

- Wywołanie metody (a także przekazane parametry) możemy sprawdzić za pomocą metody ***verify()***

```
verify(mock, times(ileRazy)).nazwaMetody(parametry)
```

- Aby zweryfikować, że na mocku nie wykonano żadnych innych akcji, możemy wywołać metodę *verifyNoMoreInteractions(mock)*

```
verifyNoMoreInteractions(mock)
```

Metoda *verify()*

Sprawdzanie parametrów

```
// given
when(userDao.findUser("legolas")).thenReturn(new User("legolas"));

// when
boolean result = userService.doesUserExist("legolas");

// then
assertThat(result).isEqualTo(true);
verify(userDao, times(1)).findUser("legolas");
verifyNoMoreInteractions(userDao);
```

Zadanie 8

5 minut

- w klasie testowej:
  - com.isa.user.**UserServiceTest**

- skopiuj test ***shouldReturnTrueIfUserExists***

  - zmień sposób sprawdzania parametrów przekazanych do metody **userDao.findUser()**

  - wykorzystaj metodę ***verify()***

Mockito

Czego mockito nie potrafi?

- nie można mockować metod prywatnych
- brak możliwości mockowania metod statycznych
- brak możliwości mockowania konstruktora
- brak możliwości mockowania metod: equals(), hashCode()

Testy

Zalety kodu przetestowanego

- masz (większą) pewność, że działa
- zadowolenie klientów
- łatwość zmian
- szybszy “debugging”, błyskawiczna odpowiedź o stanie kodu
- “samopisząca” się dokumentacja
- możesz polegać na członkach zespołu
- oszczędzasz czas nie musząc wykonywać tak wielu testów manualnych
- czujesz się lepiej, śpisz spokojniej

Testy

Dlaczego nie piszemy testów jednostkowych?

- **Czas developmentu**. Początkowe etapy projektu wymagają dodatkowej pracy na przygotowanie testów jednostkowych.
- **Czas utrzymania**. Przygotowane zestawy testów trzeba z czasem utrzymywać by nadal przynosiły korzyści.
- brak odpowiedniej wiedzy programistów
- strach kierownictwa przed wyższymi kosztami
- przekonanie, że testerzy wyłapią wszystkie błędy

Q&A

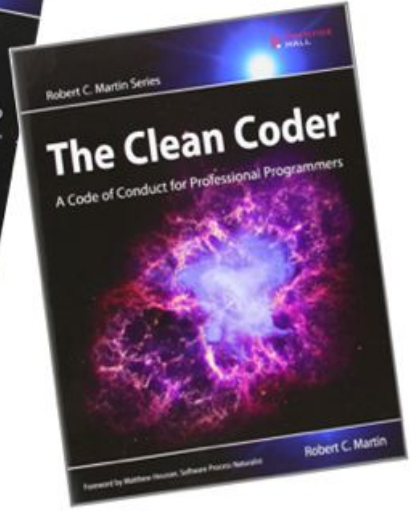
TDD

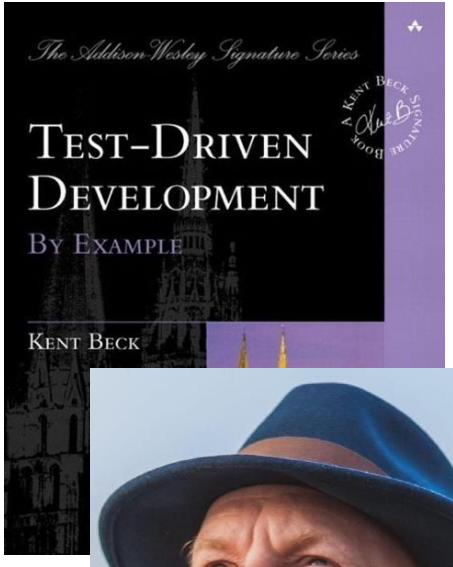
Najtrudniej skończyć tę robotę, której się nie zaczęło.

J.R.R. Tolkien, Drużyna Pierścienia

TDD

Tworzenia oprogramowania sterowane przez testy





TDD

Trzy zasady

- Najpierw programista pisze automatyczny test sprawdzający dodawaną funkcjonalność. Test w tym momencie nie powinien się udać.
- Później następuje implementacja funkcjonalności. W tym momencie wcześniej napisany test powinien się udać.
- W ostatnim kroku programista dokonuje refaktoryzacji napisanego kodu, żeby spełniał on oczekiwane standardy.

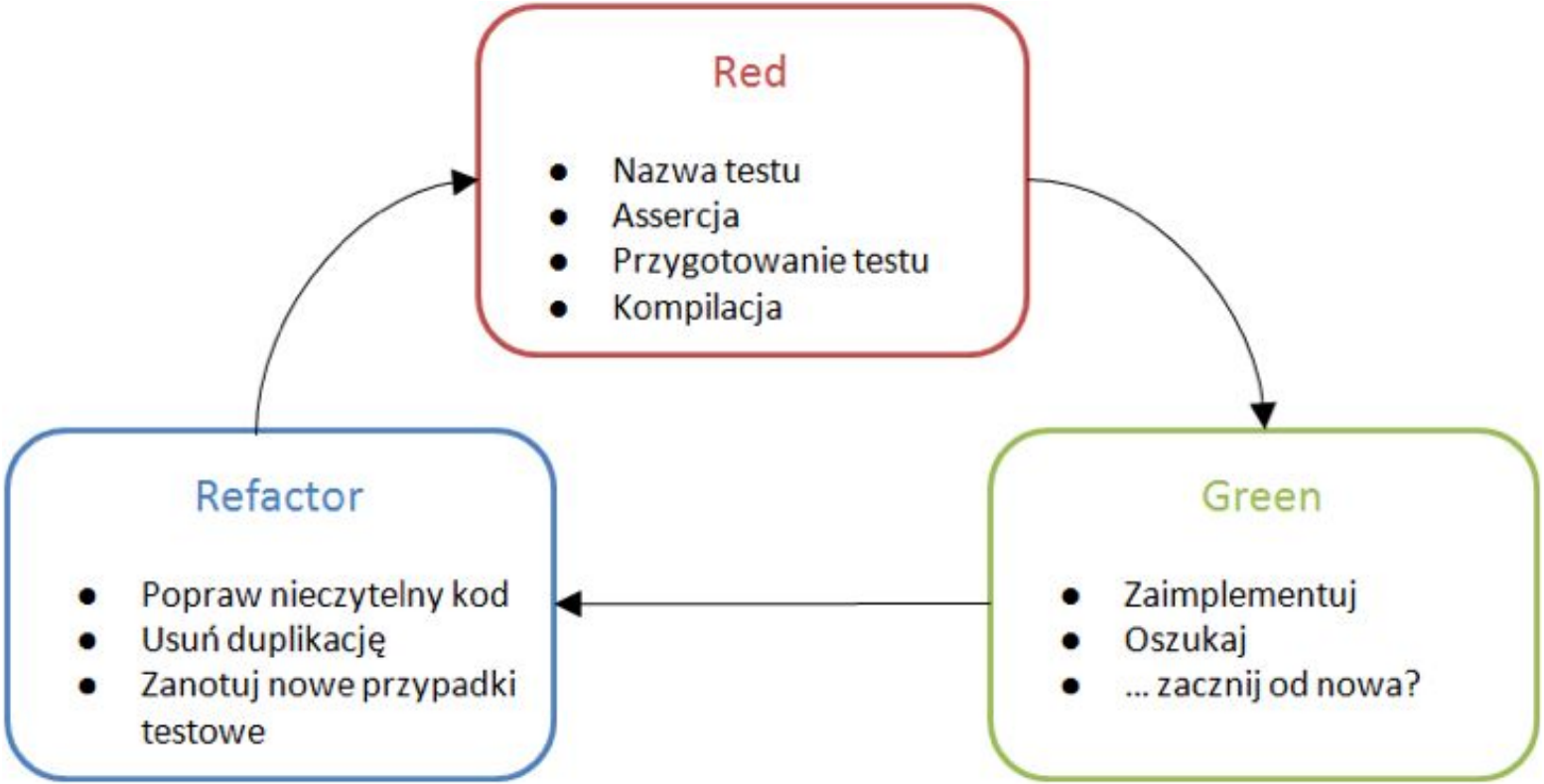
TDD

Pamiętaj!

- Możesz pisać kod produkcyjny tylko, jeżeli napisałeś do niego test jednostkowy.
- Możesz napisać tylko tyle kodu testowego, żeby test uruchamiał się, ale nie przechodził.
- Możesz napisać tylko tyle kodu produkcyjnego, żeby przeszedł test, który napisałeś.

TDD

Cykl


Red
Nazwa testu
Assercja
Przygotowanie testu
Kompilacja
Green
Zaimplementuj
Oszukaj
... zacznij od nowa?
Refactor
Popraw nieczytelny kod
Usuń duplikację
Zanotuj nowe przypadki testowe

TDD

Cykl

Red - Green - Refactor

- napisz testy sprawdzające pisaną funkcjonalność
- testy nie przechodzą lub nawet się nie kompilują, gdyż sprawdzana funkcjonalność jeszcze nie istnieje ❌

TDD

Cykl

Red - **Green** - Refactor

- implementacja testowanej funkcjonalności, tak aby testy napisane w fazie Red zaczęły przechodzić ✔️
- w tej fazie dążymy do działającej funkcjonalności najmniejszym możliwym wysiłkiem

TDD

Cykl

Red - Green - **Refactor**

- polega na wyczyszczeniu kodu napisanego w fazie Green tak, aby uczynić go czytelniejszym i bardziej zrozumiałym, nie zmieniając jego zachowania
- uruchamiając testy napisane w fazie Red możemy upewnić się, że refaktorując kod nie zepsuliśmy żadnej funkcjonalności

TDD

Wady?

- wymaga dodatkowego czasu na stworzenie testów jednostkowych
- wymaga czasu w na utrzymanie testów

TDD

Zalety

- szybkie wychwytywanie błędów
- błędy wykryte przez autora kodu i poprawiane na bieżąco kosztują niewiele, ponieważ angażują tylko jedną osobę
- bardziej przemyślany kod
- możliwość przetestowania funkcjonalności bez uruchamiania całego oprogramowania
- tworzenie swoistej dokumentacji

Q&A

Zadanie

Bowling game kata

Napisz serwis który sumuje ilość zdobytych punktów w kręglach.

Punktacja w kręglach:

1. gra ma dziesięć rund (frame)
2. w każdej rundzie gracz ma dwie rzuty (roll), żeby strącić dziesięć kręgli
3. gracz zdobywa tyle punktów ile strącił kręgli, plus bonusy za “strike” i “spare”
4. “strike” jest wtedy, gdy gracza strąci dziesięć kręgli w jednym rzucie, dostanie wtedy dodatkowo tyle punktów ile strąci kręgli w dwóch następnych rzutach
5. “spare” to sytuacja, gdy gracz strąci dziesięć kręgli w dwóch rzutach, dodatkowo dostaje wtedy tyle punktów ile strąci kręgli w następnym rzucie
6. jeżeli gracz zdobędzie “strike” albo “spare” w dziesiątej rundzie to ma dwa (jeden w przypadku “spare”) dodatkowe rzuty na zdobycie dodatkowych punktów
7. punkty zdobyte po ostatniej rundzie naliczane są tylko raz

http://www.bowlinggenius.com/

Zadanie

Reverse words kata

Napisz serwis który:

- Odwróci kolejność wyrazów w zdaniu.
- Rzuci wyjątek gdy:
  - zdanie nie zaczyna się z wielkiej litery
  - zdanie nie kończy się kropką
  - zdanie składa się z jednego wyrazu
- Ustawi wielką literę oraz kropkę w odpowiednich miejscach.

Zadanie

String calculator kata

Napisz kalkulator który:

- Przyjmuje zero, jedną lub dwie liczby oddzielone spacjami, a następnie zwraca ich sumę.
  - Jeżeli String będzie pusty zwróci wartość zero.
  - Jeżeli String będzie zawierał inne znaki niż liczby, rzuci IllegalArgumentException z odpowiednią wiadomością.
  - Jeżeli liczb będzie zbyt dużo rzuci on IllegalArgumentException z odpowiednią wiadomością.
- Dodaj wsparcie dla więcej niż dwóch liczb.
- Rzuć exception jeżeli jedna z liczb jest ujemna.
- Zignoruj wszystkie liczby które są większe od 1000.
- Dodaj wsparcie dla innych separatorów niż spacja.
  - Jeżeli użytkownik chce użyć innego ogranicznika powinien string zapisać w następujący sposób: '//<separator> <cyfry>'
    Przykład: //| 1|2|3 powinno dać wynik 6

Zadanie

Accident application eligibility checker kata

Napisz prosty serwis, który sprawdzi czy zgłoszenie jest poprawne.

- Aplikacja powinna przyjmować następujące dane od użytkownika:
  - imię
  - nazwisko
  - datę szkody (w formacie yyyy-MM-dd)
  - rodzaj szkody: ACCIDENT lub TOTAL_LOSS
- Dane powinny zostać zwalidowane.
- Aplikacja powinna określić wynik zgłoszenia na podstawie następujących reguł:
  - jeżeli ACCIDENT i nazwisko poszkodowanego krótsze niż 5 znaków -> REJECTED
  - jeżeli ACCIDENT i data szkody powyżej 5 lat -> REJECTED
  - jeżeli ACCIDENT i data szkody do 5 lat -> APPROVED
  - jeżeli ACCIDENT i data szkody do 2 tygodni -> POSTPONED
  - jeżeli TOTAL_LOSS -> POSTPONED
- Wynik zgłoszenia powinien zawierać następujące informację:
  - unikalny nr w formacie: <lp>-<pierwsze-litery-imienia-i-nazwiska>-<data-szkody>, np. 1-MK-160819
  - status zgłoszenia
  - datę rozpatrzenia zgłoszenia
- Wynik należy zapisać do pliku o nazwie <timestamp>-result.<rozszerzenie>
  - na początek aplikacja powinna wspierać formaty csv i json
- Aplikację napisać w taki sposób, aby możliwe było jej łatwe rozszerzenie o dodatkowe zasady walidacyjne oraz formaty.

Dzieki

mszymanski500@gmail.com

linkedin.com/in/mariuszszymanskipl