Testowanie w Javie

v3.0

Coders Lab
SZKOŁA PROGRAMOWANIA

Plan

1. Wprowadzenie do testowania
2. JUnit
3. Test Driven Development

Coders Lab
SZKOŁA PROGRAMOWANIA

Wprowadzenie do testowania

Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie

Ręczne

- Łatwe na początku.
- Z czasem uciążliwe (a wręcz niemożliwe).
- Jest wolne.
- Podatne na błędy.

Automatyczne

- Wymaga trochę wysiłku.
- Im dłużej trwa projekt, tym bardziej zwraca się testowanie automatyczne.
- Zapewnia nam siatkę bezpieczeństwa.
- Może pomagać w programowaniu.
- Są rzeczy, które testuje się trudno (ale praktycznie wszystko da się przetestować).
- Jest rzetelne (brak czynnika ludzkiego).

Metody testowania

Testowanie statyczne

Polega na statycznej analizie napisanego kodu, sprawdzeniu jego składni i przepływu.

Najczęściej spotykane metody to:

- **code review** czyli przeczytanie kodu przez kilku deweloperów zanim zostanie on dodany do projektu,
- zastosowanie programów sprawdzających wszystkie możliwości przepływu programu.

Testowanie dynamiczne

Testy przeprowadzane na działającym programie. Ich zadaniem jest sprawdzenie, czy wyniki i zachowanie jest takie jak przewidywane przez nas wcześniej.

Testowanie dynamiczne powinno być rozpoczęte zanim program jest ukończony. A najlepiej zanim program zacznie być tworzony.

Metody testowania

Testy funkcjonalne

Testy zakładające, że moduł jest „czarnym pudełkiem”, o którym wiadomo tylko, jak ma się zachowywać. Nazywamy je też testami czarnej skrzynki (**black-box testing**).

Testy strukturalne

Testy skupiające się na wewnętrznej pracy modułu, a nie na jego kooperacji z innymi modułami. Nazywamy je też testami białej skrzynki (**white-box testing**).

Poziomy testowania

Testy jednostkowe (unit testing)

Najniższy poziom testów. Ich zadaniem jest sprawdzenie poszczególnych jednostek logicznych kodu (zazwyczaj klas i ich funkcji).

Tworzone przez deweloperów podczas pracy nad poszczególnymi funkcjonalnościami.

Testy integracji (integration testing)

Testy sprawdzające integracje poszczególnych interfejsów programu i ich wzajemne oddziaływanie.

- **Metoda od dołu** (bottom up) – metoda sprawdzająca najpierw najniższe części programu i budująca testy wzwyż.
- **Metoda od góry** (top down) – metoda sprawdzająca najbardziej ogólne moduły i budująca test w dół.
- **Big bang** – metoda grupująca moduły wedle funkcjonalności.

Poziomy testowania

Testy systemowe (end-to-end testing)

- Testy polegające na sprawdzaniu całego gotowego oprogramowania w celu znalezienia potencjalnych błędów wpływających na użytkownika.
- Testy te mają również na celu sprawdzenie, czy program nie wpływa na system, w którym działa.

Coders Lab
SZKOŁA PROGRAMOWANIA

Typy testów

Testy poczytalności (sanity tests)

Najmniejsza grupa testów sprawdzająca, czy podstawowa funkcjonalność systemu działa.

Testy dymne (smoke tests)

Grupa testów, których wykonanie trwa stosunkowo krótko. Pokazują one, czy można rozwijać dany kod.

Regresja

Niezamierzone zmiany wprowadzone zazwyczaj w komponencie, nad którym nie pracujemy, a który to komponent polega na aktualnie zmienianym.

Testy regresji (regression tests)

Testy sprawdzające, czy funkcjonalność pozostaje bez zmian między wersjami. Często pokazują zmiany, które nie są de facto błędami, ale niechcianymi naruszeniami starego standardu.

Słowniczek

Atrapy

Obiekt w teście, który jest nam potrzebny jako wypełnienie, a nie spełnia żadnego logicznego celu. Możemy w tym celu wykorzystać nawet pustą klasę.

Fake

Obiekt, który zawiera już logikę, ale nie taką jak prawdziwa implementacja. Na przykład tabelka z danymi zamiast bazy danych.

Zaślepka (stub)

Obiekt mający minimalną implementację potrzebnego przez nas interfejsu. Zazwyczaj funkcje zwracają dla jakiejś danej wejściowej predefiniowaną daną wyjściową.

Mock

Obiekt, który poza predefiniowaną daną wyjściową śledzi jeszcze wszystkie interakcje z interfejsem. Zazwyczaj bardziej skomplikowane klasy. Najlepiej skorzystać już z dostępnych wchodzących w skład bibliotek.

JUnit

Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit

JUnit - to framework służący do pisania powtarzalnych testów.

Inaczej rzecz ujmując jest to biblioteka w postaci pliku **jar** dołączana do projektu, która wspomaga nas w pisaniu i uruchamianiu testów jednostkowych.

Strona projektu http://junit.org/junit4/

Cechy:

- klasy opisywane za pomocą adnotacji
- wspiera automatyczne przeprowadzanie testów
- pluginy do integracji z popularnymi IDE

JUnit adnotacje

JUnit używa adnotacji do oznaczenia metod, wyróżniamy:

- **@Test** - oznaczenie metody testującej.
- **@BeforeClass** - oznaczenie metody uruchamianej przed wszystkimi metodami testującymi.
- **@AfterClass** - oznaczenie metody uruchamianej po wszystkich metodach testujących.
- **@Before** - oznaczenie metody uruchamianej przed każdym testem.
- **@After** - oznaczenie metody uruchamianej po każdym teście.
- **@Ignore** - oznaczona metoda tą adnotacją nie zostanie wywołana.

JUnit

Testy są umieszczone w metodach.

Metody reprezentują pewien scenariusz wykonania fragmentu kodu.

Sprawdzanie poprawności danych odbywa się poprzez porównanie wartości oczekiwanej z wynikiem rzeczywiście wykonanego działania.

Do sprawdzania poprawności służą **asercje**.

JUnit

Metody testujące mogą zawierać więcej niż jedną **asercję** , oraz własną dodatkową logikę.

Tak jak w przypadku zwykłych metod muszą się one znajdować w ramach klasy.

Klasa testująca

```
public class SimpleSampleTest {
    @Test
    public void simpleStringTest() {
        String testString = "CodersJava";
        String firstHalf = Main6.firstHalf(testString);
        assertEquals("Coder", firstHalf);
        assertTrue(firstHalf.startsWith("Code")); }
    @Test
    public void simpleTestList() {
        List<Integer> list = Arrays.asList(1, 2, 3, 4);
        assertEquals(list.size(), 4);
        assertEquals((Integer) 2, list.get(1));
        assertEquals((Integer) 4, list.get(3)); }
}
```

Klasa testująca

```
public class SimpleSampleTest {
    @Test
    public void simpleStringTest() {
        String testString = "CodersJava";
        String firstHalf = Main6.firstHalf(testString);
        assertEquals("Coder", firstHalf);
        assertTrue(firstHalf.startsWith("Code")); }
    @Test
    public void simpleTestList() {
        List<Integer> list = Arrays.asList(1, 2, 3, 4);
        assertEquals(list.size(), 4);
        assertEquals((Integer) 2, list.get(1));
        assertEquals((Integer) 4, list.get(3)); }
}
```

Testujemy metodę statyczną dzielącą napis na połowę **`firstHalf`** z zajęć: JEE_Podstawy/a_Zadania/b_Dzien_2/a_Napisy/.

JUnit w Eclipse

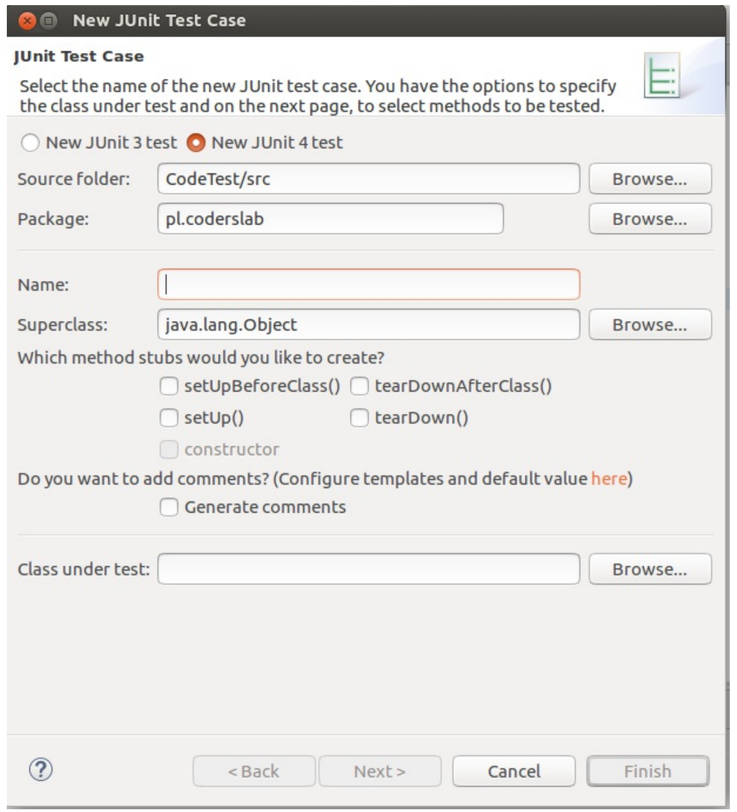
Do korzystania z **JUnit** w **Eclipse** nie potrzebujemy załączać dodatkowych bibliotek.

Tworzymy nowy projekt - **Java Project**.

Następnie tworzymy klasę testującą wybierając z menu kontekstowego **New** a następnie **JUnit Test Case**.

Wybieramy opcję **New JUnit 4 test**.

W polu **Class under test** możemy również wskazać dla jakiej klasy jest tworzona klasa testująca.

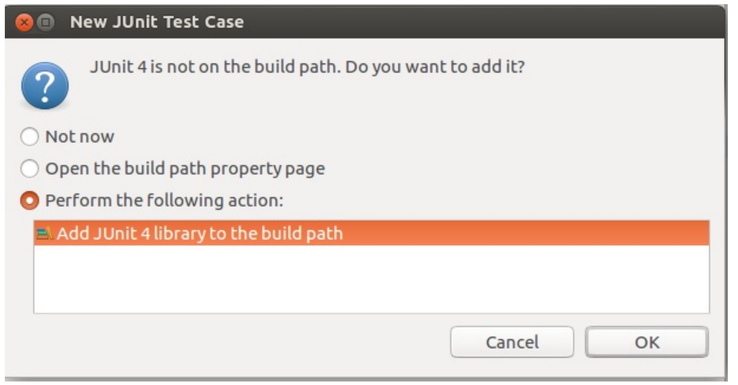


JUnit w Eclipse

W przypadku gdy jest to nasz pierwszy test w projekcie, Eclipse wskaże nam bibliotekę do załączenia.

Możliwe jest również samodzielne załączenie bibliotek do projektu, w tym celu pobieramy **jar** ze strony:

https://github.com/junit-team/junit4/wiki/Download-and-Install



Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit w Eclipse

Po utworzeniu klasy testującej będzie ona miała postać:

```
import static org.junit.Assert.fail;
import org.junit.Test;
public class SimpleSampleTest {
    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit w Eclipse

Po utworzeniu klasy testującej będzie ona miała postać:

```
import static org.junit.Assert.fail;
import org.junit.Test;
public class SimpleSampleTest {
    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Oznaczamy metodę adnotacją **@Test**. Tylko metody oznaczone tą adnotacją będą wyszukiwane przez **JUnit** i uruchamiane jako nasze testy.

Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit w Eclipse

Po utworzeniu klasy testującej będzie ona miała postać:

```
import static org.junit.Assert.fail;
import org.junit.Test;
public class SimpleSampleTest {
    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Oznaczamy metodę adnotacją **@Test**. Tylko metody oznaczone tą adnotacją będą wyszukiwane przez **JUnit** i uruchamiane jako nasze testy.

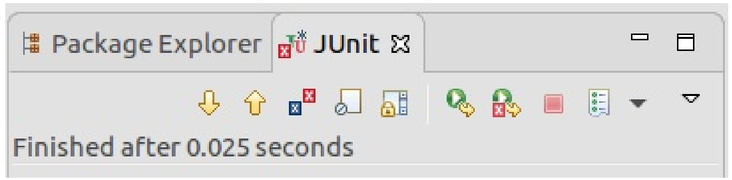
Metoda **`fail()`** czyli wywołanie statycznej metody: **`Assert.fail()`** powoduje zakończenie testu niepowodzeniem.

Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit w Eclipse

Tak utworzony test możemy już wykonać przy pomocy **Eclipse** uruchamiając jak zwykłą klasę Javy z metodą **main**.

Po uruchomieniu otworzy się dodatkowy panel:



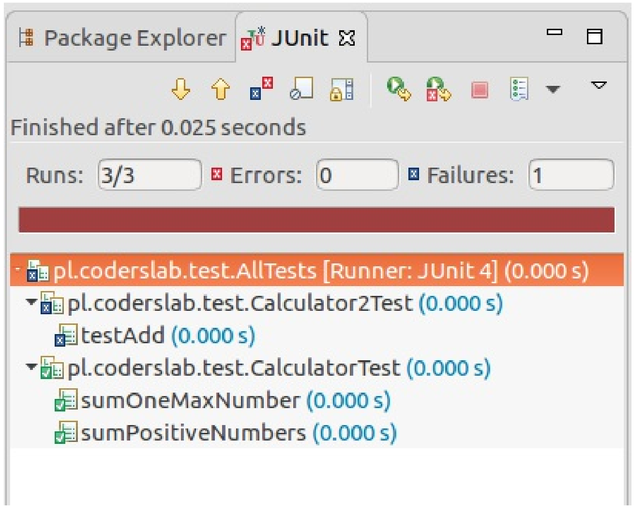
Panel udostępnia dodatkowe możliwości np. **nawigację pomiędzy testami** lub **ponowne uruchomienie testów** rozpoczynając od tych, które zakończyły się niepowodzeniem.

Coders Lab
SZKOŁA PROGRAMOWANIA

JUnit w Eclipse

Pierwsze uruchomienie wygenerowanego przez **Eclipse** testu zakończy się niepowodzeniem, ze względu na umieszczoną w teście metodę `fail("Not yet implemented");`.

Przykład niepowodzenia testów w **Eclipse**:

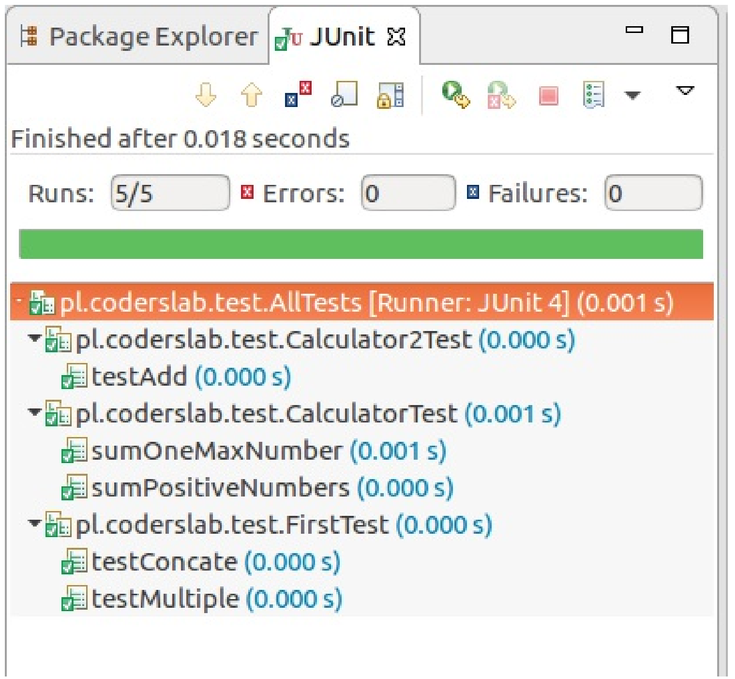


JUnit w Eclipse

Zmodyfikujemy test tak by zakończył się pomyślnie:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class SimpleSampleTest {
    @Test
    public void test() {
        assertEquals(4, 2*2);
    }
}
```

Przykład zakończenia testów pozytywnie w **Eclipse**:



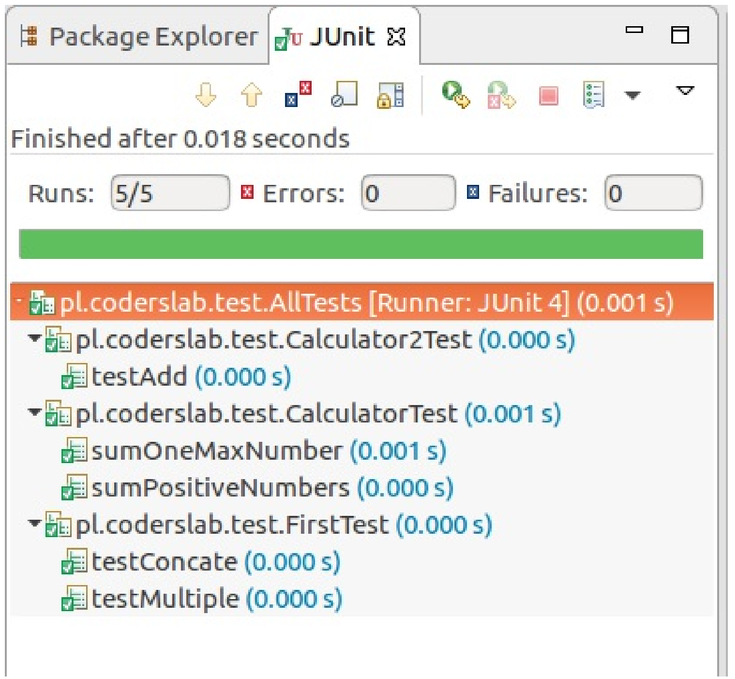
JUnit w Eclipse

Zmodyfikujemy test tak by zakończył się pomyślnie:

```
import static org.junit.Assert.*;
import org.junit.Test;
public class SimpleSampleTest {
    @Test
    public void test() {
        assertEquals(4, 2*2);
    }
}
```

Wykorzystujemy asercję **assertEquals** - ustawiając jako wartość oczekiwaną wartość **4** a jako wartość weryfikowaną wynik działania **2 * 2** .

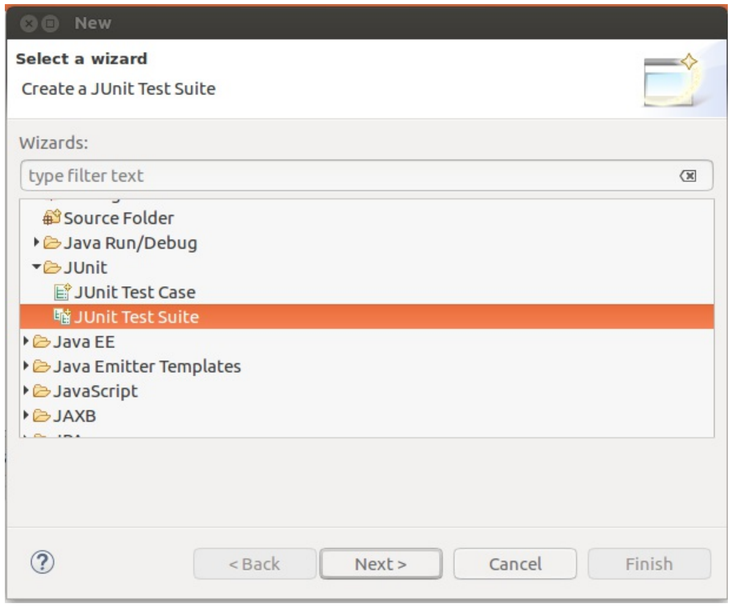
Przykład zakończenia testów pozytywnie w **Eclipse**:



Zestawy testów

Aby wywołać jednocześnie metody z kilku klas testujących możemy utworzyć **TestSuite** - jest to klasa agregująca inne klasy testowe.

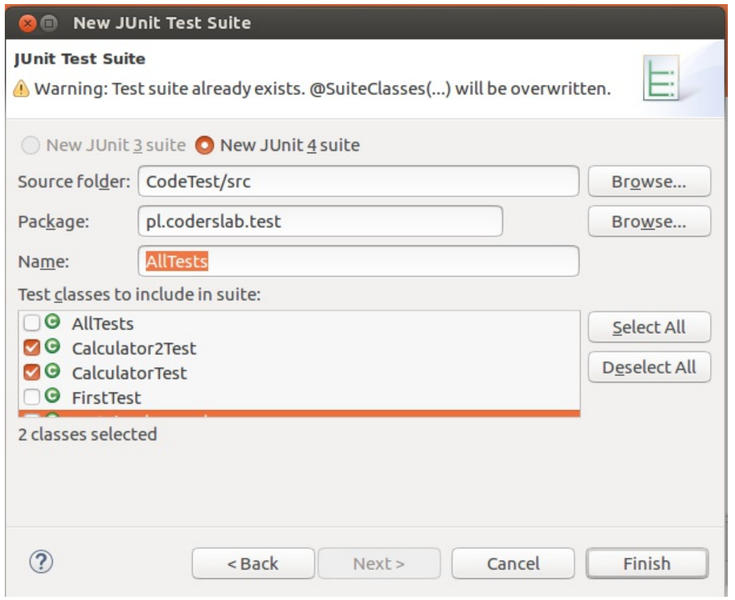
Możemy to wykonać za pomocą **Eclipse** wybierając z menu kontekstowego **New** następnie **Other** na koniec wyszukujemy **JUnit Test Suite**.



Coders Lab
SZKOŁA PROGRAMOWANIA

Zestawy testów

W kolejnym kroku nadajemy nazwę, określamy pakiet, oraz wybieramy, które klasy mają wchodzić w skład grupy.



Zestawy testów

W wyniku otrzymamy poniższą klasę:

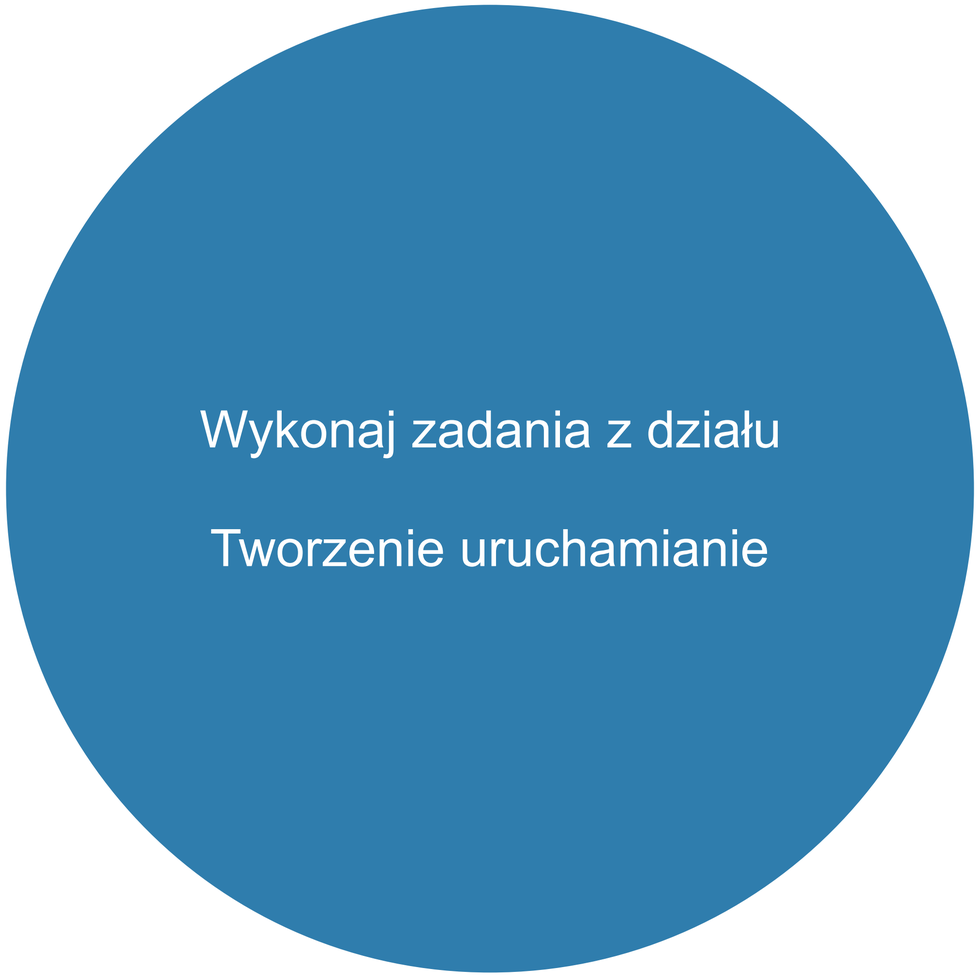
```
package pl.coderslab.test;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses({ Calculator2Test.class, CalculatorTest.class })
public class AllTests {
}
```

Adnotacja **@RunWith** wskazuje klasę uruchamiającą testy - w naszym wypadku klasę **Suite.class** - jest to standardowa klasa uruchamiająca, która pozwala budować zestawy testów z wielu klas.
Dokumentacja : https://github.com/junit-team/junit4/wiki/test-runners

Adnotacja **@SuiteClasses** - wskazuje klasy których metody testujące należy uruchomić.

Coders Lab
SZKOŁA PROGRAMOWANIA

Zadania



Coders Lab
SZKOŁA PROGRAMOWANIA

Konwencje nazewnicze

Istnieje kilka konwencji nazewniczych dotyczących tworzenia testów, najpopularniejsze to:

Nazwa klasy testującej jest zbudowana z nazwy klasy z sufiksem **Test**

np. Dla klasy **Calculator** jej klasa testująca będzie się nazywać **CalculatorTest**.

Nazwy metod zaczynają się lub zawierają słowo **should**

np. **shouldSumNegativeNumber** lub **bookShouldBeUpdated**.

Nazwy metod są zbudowane zgodnie z konstrukcją:

Given[Dane wejściowe] When[Co wykonane] Then[Opis wyniku] np.:

`givenBookInDatabase_whenDeletingBook_thenBookIsDeleted` - dla zwiększenia czytelności ze znakiem podkreślenia.

Coders Lab
SZKOŁA PROGRAMOWANIA

Konwencje nazewnicze

Niezależnie od preferowanej konwencji nazewniczej - powinniśmy się kierować zasadą, że nazwa powinna tłumaczyć jakie zadanie wykonuje test.

Zdecydowanie nie powinniśmy stosować nazw typu **testBook1**.

Jeżeli korzystamy z **Mavena** nazwy klas testujących powinny używać sufiksu **Test** - w ramach zajęć omówimy również przykład wywołania testów z jego użyciem.

Asercje

- **Asercje** to warunki, których spełnienie jest konieczne do zaliczenia testu.
- **Asercje** są podstawą pisania testów.
- Zakładamy, że asercja jest spełniona.
- Test może zawierać jedną lub więcej **asercji** - niepowodzenie którejkolwiek powoduje przerwanie testu z wynikiem negatywnym.

Przykład asercji

Przykład **asercji**:

```
assertEquals([String message], expected, actual);
assertTrue([String message], boolean condition);
```

Składnię i przeznaczenie omówimy dokładnie na kolejnych slajdach, warto zauważyć że są to statyczne metody klasy **Assert** importowane za pomocą instrukcji:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Typy asercji

JUnit posiada wiele przeciążonych metod dla typów prymitywnych, tablic i obiektów.

Opcjonalny pierwszy parametr jest wiadomością w przypadku niepowodzenia.

Oznacza to dla przykładu, że istnieją metody o sygnaturach:

```
assertArrayEquals(int[] expected, int[] actuals)
assertArrayEquals(long[] expected, long[] actuals)
assertArrayEquals(String message,int[] expected, int[] actuals)
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Najważniejsze asercje

```
assertEquals(String message, long expected, long actual)
```

Parametr **expected** jest spodziewanym wynikiem metody - zazwyczaj wpisywany ręcznie. Sami musimy wiedzieć jaki powinien być spodziewany wynik.

Parametr **actual** to wartość rzeczywista uzyskana podczas testu.

Istnieje również odwrotna wersja asercji:

```
assertNotEquals(String message, long unexpected, long actual)
```

Do porównywania używane sa metody **equals** odpowiednich klas.

Coders Lab
SZKOŁA PROGRAMOWANIA

Najważniejsze asercje

Dla typów zmiennoprzecinkowych występuje rozszerzona definicja zawierająca dodatkowy parametr oznaczający dokładność:

```
assertEquals(double expected, double actual, double delta);
```

Ze względu na skończoną dokładność jaką posługują się komputery, wywołanie:

```
assertEquals(3.33, 10.0/3.0, 0);
```

- zakończy się niepowodzeniem

```
assertEquals(3.33, 10.0/3.0, 0.1);
```

- zakończy się powodzeniem

Coders Lab
SZKOŁA PROGRAMOWANIA

Najważniejsze asercje

```
assertNull(java.lang.Object object);
```

Sprawdza czy obiekt jest równy **null**.

```
assertNotNull(java.lang.Object object);
```

Sprawdza czy obiekt jest różny od **null**.

```
assertSame(Object expected, Object actual);
```

Sprawdza czy parametry **actual** oraz **expected** są równe.

```
assertNotSame(Object unexpected, Object actual);
```

Sprawdza czy parametry **actual** oraz **unexpected** są różne.

Metody te posiadają również przeciążone odpowiedniki z możliwością podania wiadomości, która zostanie wyświetlona w przypadku niespełnienia podanych warunków.

Coders Lab
SZKOŁA PROGRAMOWANIA

Najważniejsze asercje

```
assertTrue(boolean condition);
```

Sprawdza czy podany warunek logiczny jest prawdziwy.

```
assertFalse(boolean condition);
```

Sprawdza czy podany warunek logiczny jest fałszywy.

```
fail(String message);
```

Powoduje niepomyślne zakończenie testu wyświetlając komunikat **message** .

Coders Lab
SZKOŁA PROGRAMOWANIA

Przykłady metod testujących

```
@Test
public void testAssertEquals() {
    assertEquals("failure - not equal",
                 "text", "text");
}
@Test
public void testAssertFalse() {
    assertFalse("failure -should be
                false", false);
}
@Test
public void testAssertNotNull() {
    assertNotNull("should not be null",
                  new Object());
}
```

```
@Test
public void testAssertNotSame() {
    assertNotSame("should`t be same",
                  new Object(),
                  new Object());
}
@Test
public void testAssertNull() {
    assertNull("should be null", null);
}
@Test
public void testAssertSame() {
    Integer aNumber =
        Integer.valueOf(768);
    assertSame("should be same",
                  aNumber, aNumber);
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie metod

Do tej pory poznaliśmy podstawy pracy z **JUnit** pokazujące jak używać asercji, jednak naszym celem będzie testowanie naszych własnych klas oraz zawartych w nich metod.

W tym celu utworzymy prostą klasę:

```
public class First {
      public String concatString(String first, String second) {
          return first + second;
      }
      public int multiply(int first, int second) {
          return first * second;
      }
  }
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie metod

Wykorzystując **Eclipse** utworzymy test dla tej klasy.

Wybierając klasę **First** w polu **Class Under Test**, zaznaczając obie metody tej klasy otrzymamy szablon klasy testującej z metodami.

Eclipse wygeneruje dla nas poniższy szablon:

```
public class FirstTest {
    @Test
    public void testConcatString() {
        fail("Not yet implemented");
    }
    @Test
    public void testMultiply() {
        fail("Not yet implemented");
    }
}
```

Testowanie metod

Uzupełniamy kod metod tak, by sprawdzane było prawidłowe wykonanie metody **concatString**.

```
public class FirstTest {
    @Test
    public void testConcate() {
        First first = new First();
        String result =
            first.concatString("one",
                               "two");
        assertEquals("onetwo", result);
    }
}
```

Analogicznie dodajemy kolejną metodę testującą.

Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie metod

Uzupełniamy kod metod tak, by sprawdzane było prawidłowe wykonanie metody **concatString**.

```
public class FirstTest {
    @Test
    public void testConcate() {
        First first = new First();
        String result =
            first.concatString("one",
                               "two");
        assertEquals("onetwo", result);
    }
}
```

Analogicznie dodajemy kolejną metodę testującą.

Tworzymy obiekt klasy **First**.

Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie metod

Uzupełniamy kod metod tak, by sprawdzane było prawidłowe wykonanie metody **concatString**.

```
public class FirstTest {
    @Test
    public void testConcate() {
        First first = new First();
        String result =
            first.concatString("one",
                               "two");
        assertEquals("onetwo", result);
    }
}
```

Analogicznie dodajemy kolejną metodę testującą.

Tworzymy obiekt klasy **First**.

Wywołujemy testowaną metodę.

Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie metod

Uzupełniamy kod metod tak, by sprawdzane było prawidłowe wykonanie metody **concatString**.

```
public class FirstTest {
    @Test
    public void testConcate() {
        First first = new First();
        String result =
            first.concatString("one",
                               "two");
        assertEquals("onetwo", result);
    }
}
```

Analogicznie dodajemy kolejną metodę testującą.

Tworzymy obiekt klasy **First**.

Wywołujemy testowaną metodę.

Porównujemy wartość oczekiwaną **onetwo** ze zmienną, która przechowuje wynik testowanej metody - **result** .

Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie wyjątków

Jeżeli nasza metoda zwraca wyjątek w określonych okolicznościach, również możemy to sprawdzić, adnotacja **@Test** posiada opcjonalny parametr **expected**.

Uzupełnimy klasę o metodę zwracającą wyjątek w przypadku wartości **null**.

```
public void printMessage(String message){
    if(message == null){
        throw new IllegalArgumentException();
    }else{
        System.out.println(message);
    }
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Testowanie wyjątków

Metoda testująca będzie wyglądać następująco:

```
@Test(expected = IllegalArgumentException.class)
public void shouldReturnIllegalArgEx() {
    First first = new First();
    first.printMessage(null);
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

given when then

Jest to popularny sposób pisania metod testujących które dzielimy na 3 sekcje:

- **given** - ustawiamy wartości na potrzeby testu
- **when** - wykonujemy metodę która jest testowana
- **then** - sprawdzamy czy zachowanie jest zgodne z oczekiwaniami

Przykład:

```
@Test
public void sumPositiveNumbers() {
    // given - ustalam założenia początkowe
    int a = 2; int b = 4; int expected = 6;
    // when - wywołanie metody którą sprawdzamy
    Calculator calculator = new Calculator();
    long valueToCheck = calculator.add(a, b);
    // then - sprawdzanie czy wynik jest zgodny z oczekiwaniem
    assertEquals(expected, valueToCheck);
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Timeout

W ramach testów możemy również sprawdzać czy metoda nie wpada w pętlę nieskończoną, (np. ze względu na błędnie ustawione warunki), w tym celu ustawiamy maksymalny czas wykonania.

Służy do tego parament adnotacji **@Test** o nazwie **timeout**.

Przykład:

```
@Test(timeout = 1000)
  public void infinity() {
        // kod którego czas wykonania testujemy
  }
```

Czas określamy w milisekundach.

Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

Hamcrest - jest to narzędzie, które udostępnia dodatkowe metody nazywane **matcherami**.

Jest ono włączone do **JUnit** więc nie musimy nic więcej dołączać.

Pozwala w łatwy sposób łączyć wiele warunków.

W założeniu ma sprawiać by **asercje** były bardziej zwięzłe i czytelne.

Istnieją alternatywne rozwiązania oferujące zbliżone funkcjonalności:

http://joel-costigliola.github.io/assertj/

https://github.com/alexruiz/fest-assert-2.x

Hamcrest

Schematycznie działanie przedstawiamy:

```
assertThat([value], [matcher statement]);
```

value - oznacza wartość sprawdzaną.

matcher statement - zestaw warunków porównania.

Jeżeli eclipse nie podpowiada asercji **assertThat**, lub **matcherów** należy dodać instrukcje importu:

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

W celu pisania warunków z wykorzystaniem **Hamcrest** wykorzystujemy asercję **assertThat**

Przykład:

```
@Test
public void simpleTestAssertThat() {
    Integer factorial = Main9.factorial(3);
    assertThat(factorial, is(6));
    assertThat(factorial, is(not(12)));
}
```


Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

W celu pisania warunków z wykorzystaniem **Hamcrest** wykorzystujemy asercję **assertThat**

Przykład:

```
@Test
public void simpleTestAssertThat() {
    Integer factorial = Main9.factorial(3);
    assertThat(factorial, is(6));
    assertThat(factorial, is(not(12)));
}
```

Sprawdzamy statyczną metodę obliczającą silnie z zadań na zajęciach:
JEE_Podstawy/a_Zadania/a_Dzien_1/b_Metody/.

Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

W celu pisania warunków z wykorzystaniem **Hamcrest** wykorzystujemy asercję **assertThat**

Przykład:

```
@Test
public void simpleTestAssertThat() {
    Integer factorial = Main9.factorial(3);
    assertThat(factorial, is(6));
    assertThat(factorial, is(not(12)));
}
```

Sprawdzamy czy **factorial** jest równy **6**.

Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

W celu pisania warunków z wykorzystaniem **Hamcrest** wykorzystujemy asercję **assertThat**

Przykład:

```
@Test
public void simpleTestAssertThat() {
    Integer factorial = Main9.factorial(3);
    assertThat(factorial, is(6));
    assertThat(factorial, is(not(12)));
}
```

Sprawdzamy czy **factorial** nie jest równy **12**.

Coders Lab
SZKOŁA PROGRAMOWANIA

Hamcrest

Najważniejsze **matchery** :

- **equalTo** - sprawdza czy obiekt za pomocą metody **equals**
- **is** - dekorator dla **equalTo**
- **hasItem** - sprawdza czy kolekcja zawiera sprawdzany element
- **hasItemInArray** - sprawdza czy tablica zawiera sprawdzany element
- **equalToIgnoringCase** - sprawdza napis ignorując wielkość znaków
- **containsString, endsWith, startsWith** - warunki dla napisów
- **everyItem** - sprawdza każdy element
- **allOf** - określa, że wszystkie **matchery** muszą być spełnione
- **anyOf** - określa, że wystarczy jeden spełniony **matcher**

Hamcrest - Przykłady

```
@Test
public void simpleTestAssertThatMather() {
    List<Integer> list = Arrays.asList(1, 2, 3, 4);
    assertThat(list, hasItem(4) );
    assertThat(list, allOf(hasItem(4), hasItem(2)) );
    assertThat(list, allOf(hasItem(4), not(hasItem(12))) );
    assertThat(list, anyOf(not(hasItem(12)), hasItem(12) ));
    assertThat("CodersLab", anyOf(endsWith("Lab"), containsString("rs"),
                not(startsWith(""))));
    assertThat(Arrays.asList("bar", "baz"), everyItem(startsWith("ba")));
}
```

Lista wszystkich dostępnych **matcherów**:
http://junit.org/junit4/javadoc/latest/org/hamcrest/CoreMatchers.html

Lista z podziałem na kategorie w postaci pdf:

http://www.marcphilipp.de/downloads/posts/2013-01-02-hamcrest-quick-reference/Hamcrest-1.3.pdf

Zadania

Coders Lab
SZKOŁA PROGRAMOWANIA

Test Driven Development

Coders Lab
SZKOŁA PROGRAMOWANIA

Co to jest TDD?

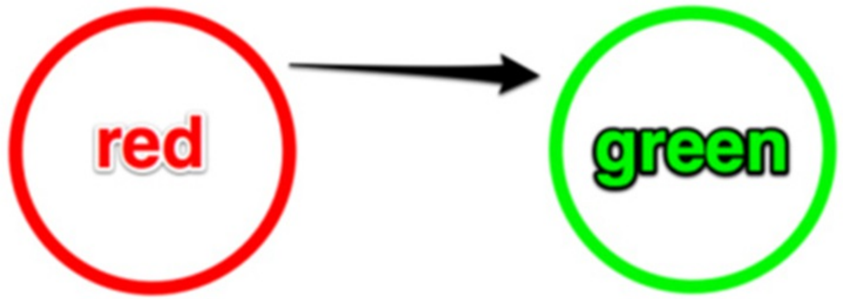
Test driven development (TDD) jest techniką tworzenia oprogramowania zaliczaną do metodyk zwinnych (**Agile**).

Polega na wielokrotnym powtarzaniu trzech kroków:

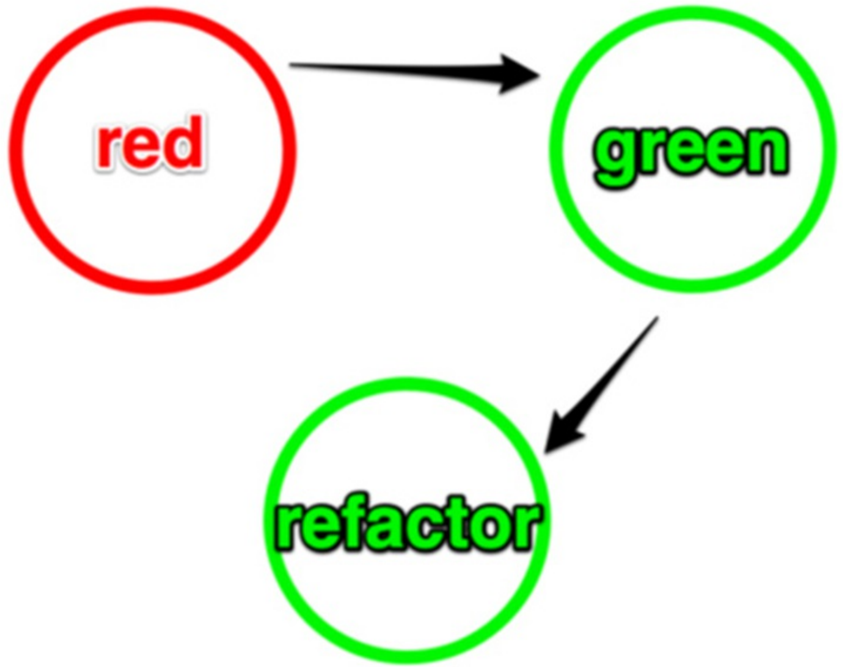
- napisaniu testu automatycznego,
- implementacji funkcjonalności do chwili, gdy wszystkie testy przejdą,
- poprawiania kodu do momentu, w którym spełnia wszystkie wymagania (a testy nadal przechodzą).

Trzy kroki TDD

Trzy kroki TDD

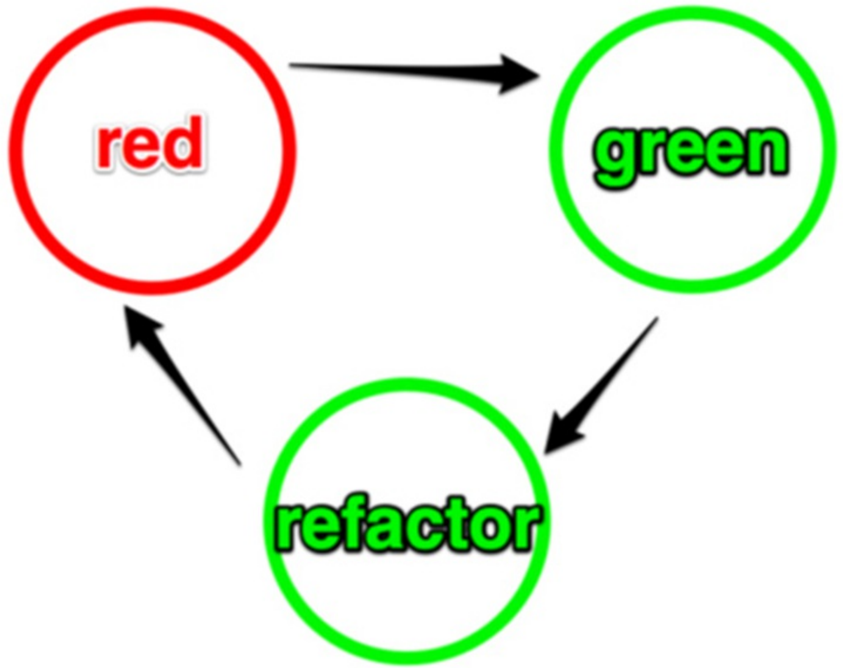


Trzy kroki TDD



Coders Lab
SZKOŁA PROGRAMOWANIA

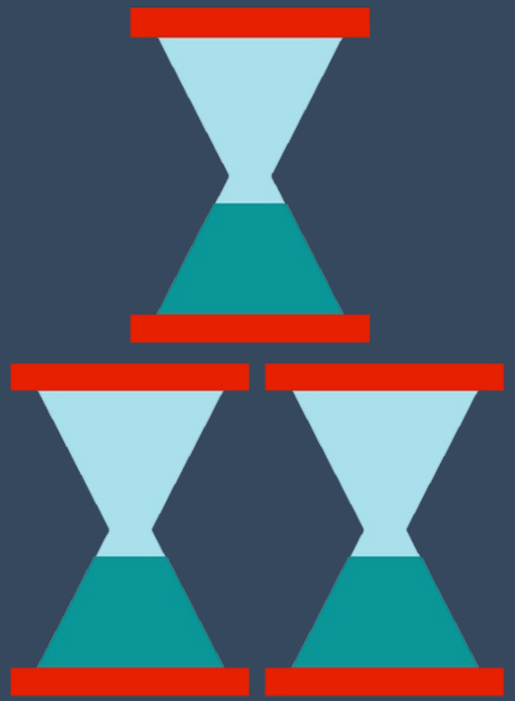
Trzy kroki TDD



Coders Lab
SZKOŁA PROGRAMOWANIA

Najważniejsza zasada TDD

Nie chodzi o często bezmyślne pisanie testów



Chodzi o przemyślany proces pisania testów

Coders Lab
SZKOŁA PROGRAMOWANIA

Zalety i wady TDD

Zalety

- Szybkie wychwytywanie błędów (już na etapie początkowej implementacji logiki).
- Błędy wykrywane i naprawiane przez autora są tańsze w naprawie.
- Kod tworzony za pomocą TDD jest bardziej przejrzysty i łatwiejszy w utrzymaniu.
- Możliwość testowania aplikacji bez potrzeby uruchamiania całego programu.

Wady

- Deweloper potrzebuje dodatkowego czasu przeznaczonego na tworzenie testów.
- Rozpoczęcie pracy nad kodem jest przesunięte w czasie.
- Testy muszą być odpowiednio zarządzane i uaktualniane wraz ze zmianą logiki. Inaczej TDD traci sens.

Coders Lab
SZKOŁA PROGRAMOWANIA

Cztery złote zasady TDD

- Dodawaj kod, gdy czerwone
- Usuwaj kod, gdy zielone
- Usuwaj duplikaty
- Poprawiaj złe nazwy

Nie wiem, jak to napisać (i przetestować)

Metoda Spike

- Brzydko napisz kod metodą prób i błędów.
- Gdy już wiesz, jak rozwiązać problem, usuń kod i zrób to porządnie.

Coders Lab
SZKOŁA PROGRAMOWANIA

Kiedy nasze testy są dobre?

Testy można uznać za dobrze napisane, gdy są:

- **Niezależne**

  (od środowiska i innych testów).

- **Szybkie**

  (dzięki temu mogą być wykorzystane w Continuous Integration).

- **Powtarzalne**

  (za każdym razem dają takie same wyniki).

- **Na bieżąco z kodem**

  (zawsze, gdy zmienia się funkcjonalność, muszą zmienić się testy).

- **Krótkie**
- **Odporne na zmiany**

  (innych części naszej aplikacji).

Coders Lab
SZKOŁA PROGRAMOWANIA

Zadania

Coders Lab
SZKOŁA PROGRAMOWANIA