**WYŻSZA SZKOŁA ZARZĄDZANIA I BANKOWOŚCI**

**W KRAKOWIE**

**WYDZIAŁ ZARZĄDZANIA, FINANSÓW I INFORMATYKI**

**KIERUNEK: INFORMATYKA**

**SPECJALNOŚĆ: TECHNOLOGIE INTERNETOWE**

**PRACA DYPLOMOWA**

**Artur Kaliszuk**

**Test-driven development w wybranych językach obiektowych**

PROMOTOR:

**dr inż. Tomasz Jurczyk**

**KRAKÓW 2019**

Spis Treści

[Wstęp 4](#_Toc9006696)

[Cel Pracy 5](#_Toc9006697)

[1. Test-Driven Development 6](#_Toc9006698)

[1.1 Koncepcja TDD 6](#_Toc9006699)

[1.2 Trzy prawa TDD 6](#_Toc9006700)

[1.3 Cykl Red-Green-Refactor 7](#_Toc9006701)

[1.4 Mocne punkty TDD 8](#_Toc9006702)

[1.5 Rodzaje testów oprogramowania: Testy jednostkowe i integracyjne 8](#_Toc9006703)

[1.6 Wprowadzenie do testów jednostkowych/modułowych 8](#_Toc9006704)

[1.7 Atrybuty testów jednostkowych - F.I.R.S.T. 10](#_Toc9006705)

[2. Projektowanie obiektowe pod kątem testów - zagadnienia związane z testowaniem 11](#_Toc9006706)

[2.1 S.O.L.I.D 11](#_Toc9006707)

[2.1.1 Zasada pojedynczej odpowiedzialności 11](#_Toc9006708)

[2.1.2 Zasada otwarte-zamknięte 11](#_Toc9006709)

[2.1.3 Zasada podstawiania Liskov 11](#_Toc9006710)

[2.1.4 Zasada segregacji interfejsów 12](#_Toc9006711)

[2.1.5 Zasada odwrócenia odpowiedzialności 12](#_Toc9006712)

[2.2 Dependency Injection 12](#_Toc9006713)

[2.3 Refaktoryzacja kodu 12](#_Toc9006714)

[2.3.1 Duplikacja kodu 12](#_Toc9006715)

[2.3.2 Zły kod / code smell 13](#_Toc9006716)

[2.3.3 Przeprowadzenie refaktoryzacji 13](#_Toc9006717)

[3. Środowiska testowe w językach obiektowych 14](#_Toc9006718)

[3.1 Zasada Arrange – Act - Assert 14](#_Toc9006719)

[3.2 GoogleTest C++ Testing Framework 14](#_Toc9006720)

[3.3 Java: JUnit 16](#_Toc9006721)

[3.4 .Net: xUnit.net 17](#_Toc9006722)

[3.5 Python: PyUnit 17](#_Toc9006723)

[4. Techniki Izolacji testów 17](#_Toc9006724)

[4.1 Obiekty pozorujące - Atrapy 17](#_Toc9006725)

[4.2 Code coverage 18](#_Toc9006726)

[4.3 Biblioteka Framework Mock 19](#_Toc9006727)

[5. Przebieg tworzenia aplikacji 19](#_Toc9006728)

[5.1 Klasa Coord 20](#_Toc9006729)

[5.2 Klasa Snake 21](#_Toc9006730)

[5.3 Klasa Apple 23](#_Toc9006731)

[5.4 Klasa Board 23](#_Toc9006732)

[6. Bibliografia 25](#_Toc9006733)

# Wstęp

Badania i rozwój nad uczynieniem technik wytwarzania oprogramowania łatwiejszymi, szybszymi i bardziej wydajnymi, istnieje tak długo jak sama idea programowania, a jednak ciągle w pracy nad projektami sporą część poświęconego czasu pochłania przepisywanie i poprawianie istniejącego kodu, którego można by uniknąć lub przynajmniej naprawić mniej kosztownie. Błędy w oprogramowaniu, bywają bardzo trudne do zlokalizowania, pochłaniając ogromne ilości czasu, a refaktoring i dostosowanie już istniejącego kodu do nowych funkcjonalności potrafi pogrążyć cały projekt. Dlatego tak ważne stało się przestrzeganie dobrych zasad tworzenia oprogramowania jak i gruntowne testowanie kodu, najlepiej w jak najwcześniejszym etapie powstawania projektów. Praca nad jakością oprogramowania i jego walidacja rozwija się nieprzerwanie od dekad tworząc i doskonaląc rożne metody testowania.

# Cel Pracy

W tej pracy będę chciał przybliżyć technikę testowania jednostkowego określaną jako Test-Driven Development, opisać koncepcję i główne zasady tej filozofii. Stworzę i dokładnie opisze każdy krok tworzenia aplikacji opartej na filozofii TDD w odniesieniu do różnych frameworków w kilku językach programowania.

# Test-Driven Development

## Koncepcja TDD

Test-Driven Development (TDD) jest to technika tworzenia oprogramowania stymulowana przez testy, pozwalająca na pisanie przejrzystego, elastycznego i prostego w utrzymaniu kodu. W uproszczeniu polega na napisaniu niewielkich ilości kodu (testów jednostkowych/modułowych), które opisują nowe zachowanie kodu, który chcemy zaimplementować w projekcie, zanim sam kod zostanie w nim dodany.

Cały cykl TDD zamyka się w trzech etapach, pierwszym z nich jest napisanie testów, następnie pisanie kodu produkcyjnego a końcową fazą refaktoring, co powoduje, że mamy do dyspozycji kompletny zestaw testów, jeszcze przed implementacją danej funkcjonalności.

Idealnie sprawdza się w dużych projektów, gdzie zaangażowana jest znaczna ilość programistów, odpowiedzialnych tylko za kawałki kodu, który tworzą w całym projekcie.

Historia powstania Test-Driven Development sięga do wczesnych latach sześćdziesiątych, kiedy powstaje technika eXtreme Programming (XP), której TDD początkowo było częścią, jednak z czasem zyskiwała ona coraz większa popularność, ewoluując w samodzielną technikę.

## Trzy prawa TDD

Mając na uwadze, że TDD wymaga pisania testów jednostkowych na samym początku, jeszcze przed napisaniem kodu produkcyjnego.

Możemy zdefiniować trzy podstawowe prawa TDD:

* Nie można zacząć pisać kodu produkcyjnego przed zakończeniem pisania testu jednostkowego, który nie jest spełniony.
* Kod testu jednostkowego powinien być tylko tak długi, aby wystarczył do niespełnienia testu, a błędna kompilacja jest jednocześnie nieudanym testem.
* Nie można pisać większej ilości kodu, niż jest wymagana do przejścia testu jednostkowego.

## Cykl Red-Green-Refactor

TDD to podejście, które składa się z trzech faz, które łączą się w jeden cykl. Cały proces pisania kodu opiera się z właśnie takich mikro - fazach, które powtarzasz jeden po drugim. Cykl to trzy fazy:

1. Faza RED – W TDD każda nowa funkcjonalność, zaczyna się od napisania do niej testu, który musi skończyć się niepowodzeniem, przyczyną jest brak kodu, który mam pokrycie w teście jednostkowym. W IDE taki sytuacja, często jest oznaczona kolorem czerwonym. Zakończenie testu powodzeniem, na tym etapie oznacza, że albo dana funkcjonalność już jest zaimplementowana, albo test jest wadliwy. Ta faza wymaga gruntowego przemyślenia, programista musi dokładnie zrozumieć specyfikację i wymagania danej funkcjonalności, wykorzystując do tego przypadki użycia (use cases) i zapis wymagań klienta (user stories), zanim przystąpi do implementacji funkcjonalności.
2. Faza GRENN – Do istniejącego testu piszemy kod, nie zwracając tutaj uwagi na jakość pisanego kodu, skupiamy się na jak najszybszym czasie implementacji i odzwierciedleniu testu w kodzie. Uruchamiamy wszystkie do tej pory napisane testy, które muszą mieć już swoje pokrycie w kodzie. Zakończeniem tej fazy jest udana kompilacja kodu, w IDE często oznaczona kolorem zielonym. Jeśli wszystkie testy zakończą się pomyślnie na tym etapie, programista może być pewien, że kod spełnia wszystkie testowane wymagania.
3. Faza REFACTOR – Bardzo istotna faza, w której wprowadzamy zmiany w kodzie nie zmieniając jego funkcjonalności - ulepszamy, upraszczamy, tworzymy klasy lub funkcje, żeby uniknąć powielania. *Na tym etapie warto korzystać z zasad (Kent Beck's Four Simple Design Rules)* \*. Nie zawsze jest konieczna, w przypadku dobrze napisanego kodu, lecz jej brak może prowadzić do powstawania ciężkiego kodu, który będzie trudny w utrzymaniu z długim czasem kompilacji.

## Mocne punkty TDD

Zdecydowanie najmocniejszym punktem TDD, jest szybkość wyłapywania błędów w pisanym kodzie, co zmniejsza czas i koszty odnajdywania błędów w oprogramowaniu. Błędy zostaną wykryte już na etapie kompilacji pojedynczych modułów, programista jest klientem własnego kodu, co angażuje mniej osób i minimalizuje czasu potrzebny przy naprawie niedziałającego kodu. Kod zawsze jest przemyślany, a daną funkcjonalność możemy testować bez uruchamiania całego programu.

## Rodzaje testów oprogramowania: Testy jednostkowe i integracyjne

Z punktu widzenia TDD możemy wyróżnić kilka najistotniejszych rodzajów testów oprogramowania:

• Jednostkowe – testujące jak najniższe, pojedyncze jednostki kodu (klasa, metoda).

• Integracyjne – testują wiele jednostek kodu, modułów jednocześnie.

• Regresyjne – badają czy wcześniejsze zmiany w kodzie, nie wpływają na inne elementy systemu

• Akceptacyjne – odpowiadają na pytanie, czy aplikacja spełnia wymagania biznesowe.

Szczególnie istotne jest tutaj rozróżnienie testów Jednostkowych i Integracyjnych. Kiedy najmniejszą jednostką kodu jest klasa a testem jednostkowym dla niej jest poprawne wykonanie się kodu testowego korzystającego z tej klasy, testem integracyjnym nazwiemy test korzystający z zależności między kilkoma klasami i sprawdzający ich końcowy wynik (tzw. black box).

## Wprowadzenie do testów jednostkowych/modułowych

Główną przyjętą zasadą tworzenia oprogramowania jest testowanie na jak najwcześniejszym etapie, w praktyce oznacza to, że testy powinny być wykonywane na jak najmniejszych porcjach kodu, czyli na jednostkach. Zamiennie używane są również nazwy **“testy modułowe” lub “testowanie komponentów”.**

Cała koncepcja TDD opiera się wyłącznie na testach jednostkowych (Unit test), wg. Definicji jest to „metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym. Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, zgłoszone wyjątki) z oczekiwanymi wynikami – tak pozytywnymi, jak i negatywnymi (niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu).” [[1]](#footnote-1) Opcjonalnie są pisane testy integracyjne.

Pracując z kodem staramy się zawsze sprawdzać jego poprawność, czy to przez ręcznie testowanie różnych funkcjonalności, wywołanie ich z interfejsu oczekując na wynik, po uruchomienie całej aplikacji i sprawdzenie poprawności jej funkcjonowania. Wszystkie te metody są bardzo czasochłonne i mało odporne na modyfikacje i błędy, dlatego dobre testy jednostkowe, są w tym wypadku bardzo przydatne. Mając testy pokrywające wymaganą funkcjonalność oraz kod spełniający w teorii te założenia, nie musimy uruchamiać całego systemu. Uruchamianie testów i ich wykonanie musi trwać bardzo krótki odcinek czasu. To znacząco wpływa na szybkość testowania naszego kodu.

Warto zatem wyodrębnić główne cele testów jednostkowych:

* Testy powinny pomagać w poprawianiu jakości
* Pomagają specyfikować zachowanie systemu w różnych scenariuszach definiowanych w uruchamialnej formie (“executable specification”)
* Wyłapują i lokalizują błędy
* Testy powinny pomóc w zrozumieniu systemu
* Działają jak dokumentacja
* Testy powinny redukować ryzyko
* Umożliwiają bezpieczne przeprowadzenie refaktoringu
* Testy powinny być łatwe w uruchamianiu
* Powinny być szybkie, w pełni zautomatyzowane i powtarzalne (patrz FIRST)
* Testy powinny być łatwe w pisaniu i utrzymaniu
* Powinny być proste i czytelne - testy stają się zbyt skomplikowane, gdy chcemy w pojedynczym teście zweryfikować więcej niż jedną funkcjonalność
* Testy powinny wymagań niewielkiego nakładu pracy przy ich utrzymaniu w trakcie rozwoju systemu

## Atrybuty testów jednostkowych - F.I.R.S.T.

Dobre testy jednostkowe powinny spełniać pięć zasad:

* Szybkie (Fast) - Testy powinny być szybkie.
* Niezależne (Independent) - Testy nie powinny zależeć od siebie.
* Powtarzalne (Repeatable) - Testy powinny być powtarzalne w każdym środowisku.
* Samokontrolujące (Self-Validating) - Testy powinny mieć jeden parametr wyjściowy typu logicznego. Mogą się powieść albo nie.
* O czasie (Timely) - Testy powinny być pisane w odpowiednim momencie. Testy jednostkowe powinny być pisane bezpośrednio przed tworzeniem kodu produkcyjnego.

# Projektowanie obiektowe pod kątem testów - zagadnienia związane z testowaniem

## S.O.L.I.D

SOLID to skrót od pięciu głównych zasad odnoszących się do programowania zorientowanego obiektowo

### Zasada pojedynczej odpowiedzialności

Single responsibility principle - Klasa powinna mieć tylko za jedną rzecz (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy). Unikamy tworzenia dużych klas, które mają wiele zastosowań, lepszym rozwiązaniem, jest rozbicie ich na mniejsze pojedyncze klasy, które mają pojedyncze funkcjonalności, co znacznie ułatwia refaktoryzacje i ubsługę kodu.

### Zasada otwarte-zamknięte

Oprogramowanie (klasy, moduły funkcje) budujemy w sposób otwarty na rozbudowę, zamknięty na modyfikacje. Szczególnie istotna, przy pracy z dużymi projektami, gdzie mała pojedyńcza modyfikacja w kodzie produkcyjnym może ciągnąć za sobą olbrzymie zmiany w wielu miejscach z ta zmiana powiązanych. Bazuje na budowie i zarządzaniem interfejsami (extension points) poprzez które podpinamy się pod istniejący kod, oraz budowie klas pochodnych z klasy bazowej, wszystko w ramach rozbudowy ich funkcjonalności. Zasada OCP jest kombinacją hermetyzacji i wyodrębniania.

### Zasada podstawiania Liskov

Musi istnieć możliwość podstawiania typów pochodnych w miejsce ich typów bazowych. Zasada zachowania zgodności interfejsu z wszystkimi metodami, dotyczy prawidłowo zaprojektowanego dziedziczenia. Klasa bazowa wyznacza jak klasy dziedziczące po niej powinny się zachowywać, posiadając taką samą funkcjonalność. Tworząc klasę bazową, musimy mieć pewność, że jesteśmy w stanie użyć jej zamiast klasy pochodnej, nie nadpisując przy tym metod klasy bazowej, ewentualnie je rozszerzając.

### Zasada segregacji interfejsów

Interface segregation principle - interfejsy powinny być małe i konkretne, aby klasy nie implementowały metod, których nie potrzebują. Tworzone interfejsy powinny zawierać minimalną liczbę deklaracji metod, a same metody powinny być są ze sobą ściśle powiązane poprzez obszar swojej funkcjonalności. Budowanie krótkich interfejsów, które nie będą wymuszały implementacji niepotrzebnych metod w klasie.

### Zasada odwrócenia odpowiedzialności

Dependency inversion principle – wszystkie zależności powinny w jak największym stopniu zależeć od abstrakcji a nie od konkretnego typu. W klasach i metodach nie powinniśmy używać żadnych nazw konkretnych klas, mogą nazwy tylko interfejsów i klas abstrakcyjnych. Również klasy nie powinny dziedziczyć po konkretnych klasach, jedynie po klasach abstrakcyjnych i interfejsach. Polegajmy na interfejsach i klasach abstrakcyjnych, tam gdzie to wydaje się być użyteczne w przyszłości.

## Dependency Injection

Wstrzykiwanie zależności (DI) jest mechanizmem oddzielenia konstrukcji obiektu od jego użytkowania. Kod jest bardziej podatny na rozszerzenia, będąc mniej związany z innymi elementami. Głównym założeniem jest przeniesienie tworzenia obiektów oraz wiązania ich między sobą poza kod aplikacji. Obiekty tworzy i wiąże osobna biblioteka nazywana kontener DI. Kontener w celu powiązania obiektów posługuje się konfiguracją, która określa, jak obiekty powinny być powiązane.

## Refaktoryzacja kodu

Proces modyfikowania projektu/aplikacji, bez zmiany jego funkcjonalności, którego głównym celem jest ulepszenie istniejącego kodu.

Sytuacje, w których musimy bezwzględnie przeprowadzić refaktoryzację:

### Duplikacja kodu

Występowanie takich samych elementów kodu, w różnych miejscach, jest wyznacznikiem źle zaprojektowanego kodu. Jej eliminacja, nie zawsze poprawia funkcjonalność aplikacji, lecz sprawia, że kod staje się bardziej przejrzysty i czytelny, a tym samym łatwiejszy w modyfikacji.

### Zły kod / code smell

Cechy kodu źródłowego mówiące o złym sposobie implementacji i będące sygnałem do refaktoryzacji. Przykłady:

* Długa metoda (ang. Large method) - istnieją bardzo długie metody.
* Duża klasa (ang. Large class) - istnieją klasy posiadające zbyt wiele odpowiedzialności. Należy przeorganizować strukturę klas w projekcie.
* Zazdrość o kod (ang. Feature envy) - istnieją metody intensywnie korzystające z danych innej klasy, metoda taka powinna być przeniesiona do klasy, z której danych korzysta.
* Zbyt mała intymność (ang. Inappropriate intimacy) - istnieją klasy, których działanie jest zależne od implementacji innych klas. Jest to sprzeczne z ideą hermetyzacji, gdzie nie musimy znać szczegółów implementacyjnych innych klas, a jedynie ich interfejs.
* Odrzucony spadek (ang. Refused bequest) - istnieją klasy pochodne, które przeciążają metodę z nadklasy tak iż naruszają jej kontrakt. Jest to naruszenie zasady podstawienia Liskov.
* Leniwa klasa (ang. Lazy class) - istnieją klasy posiadające bardzo mały zakres odpowiedzialności.
* Powielony kod (ang. Duplicated code) - ten sam fragment kodu powtarza się w kilku miejscach; utrudnia to wprowadzanie zmian (muszą zostać odnalezione wszystkie miejsca w kodzie, które realizują to samo zadanie)[2].
* Contrived Complexity - użyte zostały skomplikowane wzorce projektowe, gdzie zastosowanie znacznie prostszych byłoby wystarczające. [[2]](#footnote-2)

### Przeprowadzenie refaktoryzacji

Proces Refaktoryzacji jest związany z testami automatycznymi, które będą sprawdzały poprawność wprowadzonych zmian w kodzie i ostrzegały przed ewentualnymi błędami. Refaktoryzacja jest przeprowadzana małymi krokami, a po każdym z nich uruchamiane są testy, sprawdzające poprawność kodu produkcyjnego. W momencie wykrycia błędu w ostatnim etapie, należy wycofać ten krok i spróbować ponownie.

# Środowiska testowe w językach obiektowych

xUnit jest zestawem frameworków do pisania i automatycznego wykonywania testów, gdzie „X” oznacza język programowania, np. JUnit dla Java, C-unit dla C, Cpp-Unit, DUnit, NUnit itp. Framework bazuje na wykorzystaniu asercji, czyli wyrażeń logicznych, które porównują rezultat oczekiwany z rzeczywistym. ASSERT (expected, actual).

## Zasada Arrange – Act - Assert

Zasada 3A jest wzorcem postępowania podczas pisania testów jednostkowych. Akronim AAA rozwijamy jako Arrange(aranżacja) Act(akcja) Assert(asercja). Wzorzec wprowadza logiczny porządek w pojedynczym unit tecie, gdzie część przygotowania danych wejściowych jest odseparowana od części weryfikacyjnej. Ponadto, nie mieszamy asercji w trakcie wywołania testowanego obiektu.

* Arrange: wszystkie dane wejściowe i preconditions,
* Act: działanie na metodzie/funkcji/klasie testowanej,
* Assert: upewnienie się, że zwrócone wartości są zgodne z oczekiwanymi.

Najpopularniejsze framworki i biblioteki wspierające tworzenie i wykonywanie testów:

* Java: JUnit, TestNG
* PHP: PHPUnit
* C: MinUnit, CUnit, (biblioteka assert.h)
* C++: GoogleTest, Glib , Boost Test Library, CppUnit, Cantata++
* .Net: Visual Studio Unit Testing Framework, NUnit, xUnit.net
* Delphi: DUnit, Fortran → pFUnit

## GoogleTest C++ Testing Framework

Cały framework opiera się na architekturze xUnit, wykorzystując wbudowane asercje oraz symulacyjne środowisko testowania obiektów (Google Mock). Google C++ obsługuje nie tylko testy jednostkowe ale także testy funkcjonalne i integracyjne, dzieląc te testy na trzy odrębne kategorie: małe, średnie i duże.

Wyróżniamy dwa typy asercji:

* ASSERT\_\* - asercje krytyczne, które w razie niepowodzenia, powodują zakończenie przypadku testowego.
* EXPECT\_ \*- asercje niekrytyczne, które w razie niepowodzenia, kontynuują działanie testu.

| **Test** | **Fatal** | **NonFatal** |  |
| --- | --- | --- | --- |
| Prawda | ASSERT\_TRUE(*warunek*) | EXPECT\_TRUE(*warunek*) |  |
| Fałsz | ASSERT\_FALSE(*warunek*) | EXPECT\_FALSE(*warunek*) |  |
| Równe | ASSERT\_EQ(*arg1*,*arg2*) | EXPECT\_EQ(*arg1*,*arg2*) | arg == arg |
| Nie Równe | ASSERT\_NE(*arg1*,*arg2*) | EXPECT\_NE(*arg1*,*arg2*) | arg != arg |
| Mniejsze Niż | ASSERT\_LT(*arg1*,*arg2*) | EXPECT\_LT(*arg1*,*arg2*) |  |
| Mniejsze lub Równe | ASSERT\_LE(*arg1*,*arg2*) | EXPECT\_LE(*arg1*,*arg2*) |  |
| Większe Niż | ASSERT\_GT(*arg1*,*arg2*) | EXPECT\_GT(*arg1*,*arg2*) |  |
| Większe lub Równe | ASSERT\_GE(*arg1*,*arg2*) | EXPECT\_GE(*arg1*,*arg2*) |  |
| C String Równe | ASSERT\_STREQ(*str1*,*str2*) | EXPECT\_STREQ(*str1*,*str2*) |  |
| C String Nie Równe | ASSERT\_STRNE(*str1*,*str2*) | EXPECT\_STRNE(*str1*,*str2*) |  |
| C String Case Equal | ASSERT\_STRCASEEQ  (*str1*,*str2*) | EXPECT\_STRCASEEQ  (*str1*,*str2*) |  |
| C String Case Not Equal | ASSERT\_STRCASENE  (*str1*,*str2*) | EXPECT\_STRCASENE  (*str1*,*str2*) |  |
| Verify that exception is thrown | ASSERT\_THROW  (*statement*,*exception\_type*) | EXPECT\_THROW  (*statement*,*exception\_type*) |  |
| Verify that exception is thrown | ASSERT\_ANY\_THROW  (*statement*) | EXPECT\_ANY\_THROW  (*statement*) |  |
| Verify that exception is NOT thrown | ASSERT\_NO\_THROW  (*statement*) | EXPECT\_NO\_THROW  (*statement*) |  |

Testy używają asercji do weryfikacji kodu, który jest testowany. Jeżeli test się nie powiedzie lub wystąpi asercja krytyczna, wynikiem testu jest niepowodzenie, w każdym innym przypadku test kończy się pozytywnym wynikiem. Aby utworzyć test, należy użyć makra TEST() w celu zdefiniowania nazwy testu. TEST() jest zwykłą funkcją C++ niezwracającą żadnej wartości. Wewnątrz ciała funkcji można umieszczać dowolne wyrażenia języka.

* Death tests – testy sprawdzające poprawne zakończenie całej pracy programu

## Java: JUnit

JUnit to popularny framework do tworzenia testów jednostkowych w Java. Obecnie najnowsza wydana wersją jest JUnit5.

Elementy biblioteki JUnit:

* „***abstrakcyjna klasa TestCase****, z wykorzystaniem której tworzony jest przypadek testowy (klasa zawiera metody setUp() i tearDown() – co umożliwia ładowanie danych wejściowych potrzebnych do wykonania przypadku oraz czyszczenie po wykonaniu testu);*
* ***klasa Assert****, zawierająca zestaw asercji porównujących wyniki oczekiwane z rzeczywistymi;*
* ***klasy TestRunner*** *(junit.textui.TestRunner i junit.swingui.TestRunner) umożliwiające wykonanie przypadku testowego;*
* ***klasa TestSuite****, umożliwiająca grupowanie większej ilości przypadków testowych.”[[3]](#footnote-3)*

Testowanie polega na sprawdzaniu asercji, które w bibliotece JUnit są metodami statycznymi w klasie Assert. Najczęściej stosowanych asercje:

* *assertTrue* sprawdza czy przekazany argument ma wartość true,
* *assertFalse* sprawdza czy przekazany argument ma wartość false,
* *assertNull* sprawdza czy przekazany argument ma wartość null,
* *assertNotNull* sprawdza czy przekazany argument nie jest nullem,
* *assertEquals* przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, jeśli są różne rzuca wyjątek,
* *assertNotEquals* przyjmuje dwa parametry wartość oczekiwaną i wartość rzeczywistą, rzuci wyjątek, jeśli są równe.
* *assertThat*() - OPIS

Pisząc testy jednostkowe, powszechne jest stworzenie kilku podobnych obiektów, jeszcze przed ich uruchomieniem. Biblioteka JUnit przychodzi z pomocą w takich przypadkach. Mamy do dyspozycji 4 adnotacje, które pozwalają na wykonanie fragmentów kodu przed/po testach:

* @After – metoda oznaczona tą adnotacją uruchamiana jest po każdym teście jednostkowym,
* @AfterClass – metoda statyczna z tą adnotacją uruchamiana jest raz po uruchomieniu wszystkich testów z danej klasy,
* @BeforeClass – metoda statyczna z tą adnotacją uruchamiana jest raz przed uruchomieniem pierwszego testu z danej klasy.
* @Before - adnotacje do publicznej metody, powoduje, że zostanie ona uruchomiona przed każdym z testów.

## .Net: xUnit.net

Platforma testowa open-source , wspiera dwa różne rodzaje testów jednostkowych :

Fakty i Teorie

Fakty reprezentują testami, które zawsze zwracają prawdę. Testują niezmienne warunki.

Teorie natomiast to testy, które zwracają prawde tylko dla określonego zestawu danych.

## Python: PyUnit

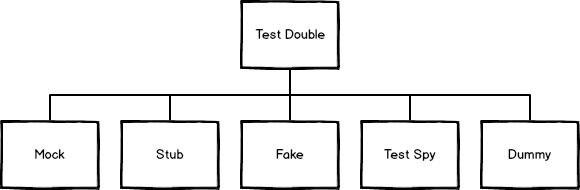
# Techniki Izolacji testów

## Obiekty pozorujące - Atrapy

Test jednostkowy testując pojedynczy element, z założenia działa w izolacji, nie zwraca uwagi na zewnętrzne zależności. Programując obiektowo jednak takie zależności występują niemal na każdym kroku, gdzie pojedyncze moduły korzystają z innych klas, metod lub interfejsów.

W tym właśnie celu została stworzona „atrapa”, będąca imitacją konkretnego obiektu, stworzoną tylko dla potrzeb testu. Jej zadaniem jest symulowanie zachowania isniejących zależności, w oparciu o dane wejściowe które sami zdefiniujemy.

Podział atrap ze względu na cel i zachowanie, wprowadzony przez Gerarda Meszarosa w książce xUnit Test Patterns na mock, stub, fake, test spy, dummy:



* Dummy – podstawowa atrapa, nie ma żadnego zadania poza imitacja istnienia obiektu
* Stub – prosta implementacja interfejsu. Metody stub’a zwykle zwracają zakodowane na sztywno wartości.
* Fake - Bardziej zaawansowana konstrukcja niż stub. Alternatywna implementacja interfejsu. Fake wygląda i działa jak prawdziwy obiekt, a stub tylko wygląda.
* Mock - Najbardziej zaawansowany obiekt pozorujący. Mock używa asercji do sprawdzenia oczekiwanej współpracy z innymi obiektami w czasie testu. W zależności od implementacji, może zwracać zakodowane na sztywno wartości lub dostarczać naśladujące implementacje logiki. Zwykle jest generowany za pomocą odpowiednich frameworków i bibliotek takich jak gmock, ale może być również implementowany ręcznie.
* Spy - mock z dodatkową funkcjonalnoscią. O ile mock rejestrował czy dana składowa została wywołana, to spy sprawdza dodatkowo ilość wywołań.

## Code coverage

Test pokrycia „(ang. coverage test) obejmuje szereg statycznych testów strukturalnych w dziedzinie testowania oprogramowania. Testy pokrycia są wykonywane na grafach przepływu sterowania (control flow graph, CFG). Przy pomocy testów pokrycia są produkowane przypadki testowe (test cases), które testują wewnętrzną strukturę oprogramowania i ewentualnie ujawniają błędy w kodzie źródłowym.” [[4]](#footnote-4)

Jest to procentowy współczynnik pokrycia kodu testami, lecz nie może być traktowany jako wyznacznik jakości kodu, ze względu na sposób w jaki można nim manipulować. Pomaga programiście w wyszukaniu obszarów które nie są pokryte testami i ewentualnie mogły by prowadzić do błędów i nic ponad to.

Link do : [Mocks Aren't Stubs](http://martinfowler.com/articles/mocksArentStubs.html)

## Biblioteka Framework Mock

Google Mock – framework do tworzenia stub/mock, bardzo często stosowany w połączeniu z Google Test, posiadający duży zestaw akcji i dopasowani(metchers).

Podstawowe Mocki tworzone są na funkcjach wirtualnych, możliwe jest również stworzenie ihj na nie-wirtualnych metodach za pomocą szablonów, wymaga to jednak dużo większego nakładu pracy.

Tworząc nowego Mocka, musimy pamiętać aby definicje funkcji zawierać w publicznej sekcji, niezależnie od tego czy funkcje mokowane są w klasie bazowej private, protected, public.

Pozwala to metodzie ON\_CALL i EXPECT\_CALL do odwołania się do funkcji spoza mockowej klasy.

# Przebieg tworzenia aplikacji

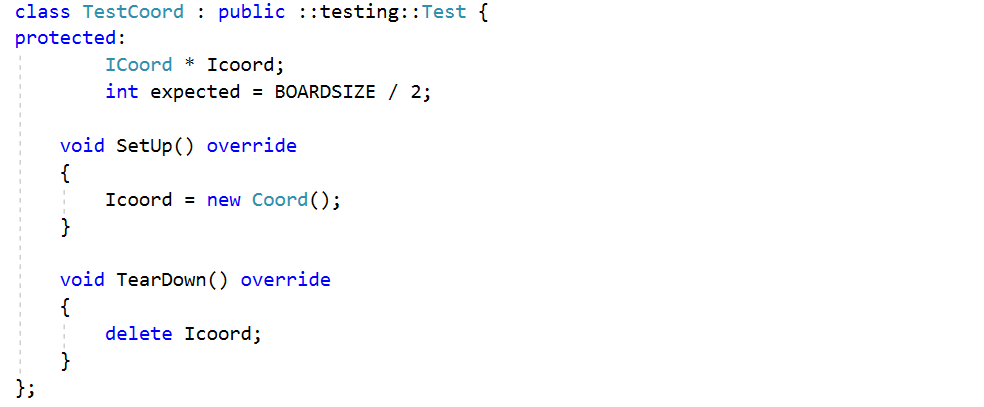
Tworzenie aplikacji zaczynam od specyfikacji, w której zawarte są podstawowe informacje o grze:

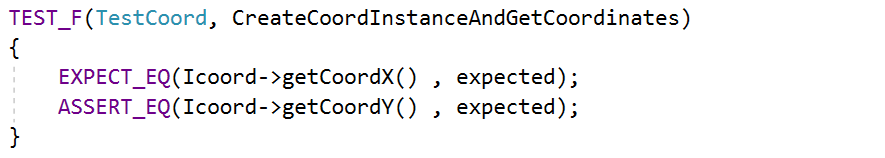
* Gra działa w czasie rzeczywistym
* Wąż porusza się w czterech kierunkach
* Wąż zjada jabłko
* Wąż rośnie
* Wąż może zginąć
* Wąż może przekraczać granice planszy
* Grę kończy się gdy wąż zje wszystkie jabłka lub zginie

Aplikacja składa się z 4 głównych klas: Snake, Coord, Apple i Board, które dziedziczą po klasach czysto wirtualnych(interfejsach), niezbędnych do testów jednostkowych we framworku Google Test.

## Klasa Coord

Zaczynam od stworzenia definicji klasy Coordinates, jest to krótka klasa, która zawiera dwa pola Xcolumn oraz Yrow, odzwierciadlające położenie obiektu na osi, zmienne nie mogą przyjmować wartości minusowych. Stworzona jest również zmienną globalna BOARDSIZE, która jest zmienną odpowiedzialną za rozmiar planszy, po której będzie poruszał się obiekt snake. Wykorzystując podstawowe asercje, oraz funkcje SetUp()/TearDown() testuję nowo powstała klasę.



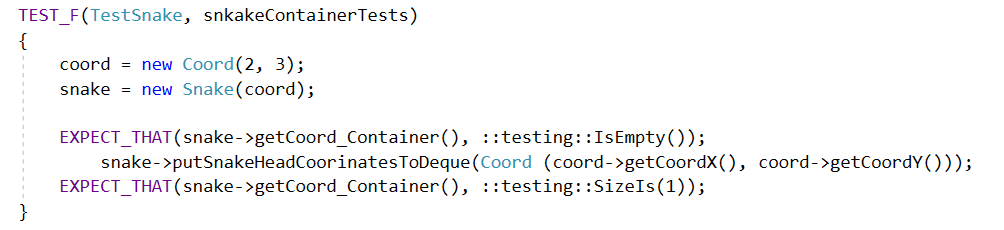


Zarówno asercje EXPECT\_EQ i ASSERT\_EQ sprawdzają czy podany argument jest równy oczekiwanej wartości, zasadniczą różnicą jest tutaj to, że w razie niepowodzenia, test kończy się na nieprawidłowym porównaniu i w przeciwieństwie do expect nie kontynuuje dalszego wykonywania się. Preferuje się użycie asercji EXPECT\_ \*, która pokazuje więcej informacji o błędzie i pozwala na wyświetlenie więcej niż jednego błędu w teście. Istnieje możliwość dostarczenia własnych komunikatów o błędzie, który zostanie wypisany obok ewentualnego błędu. Odbywa się to za pomocą operatora ≪.

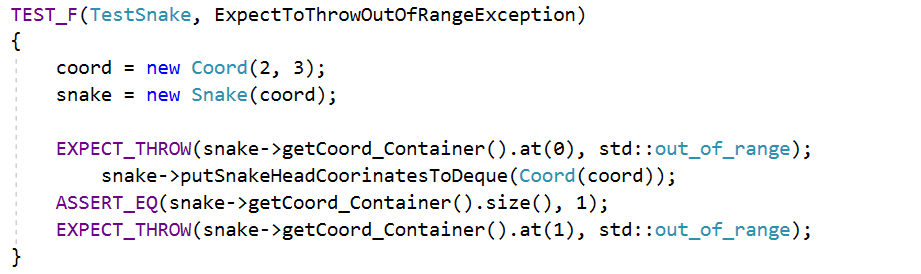
Deklaruję metodę checkBorders() która będzie sprawdzała, czy pola obiektu Xcolumn oraz Yrow nie przyjmują wartości większych niż rozmiar tablicy, a jęsli to nastąpi będzie odpowiednio je modyfikowała.

## Klasa Snake

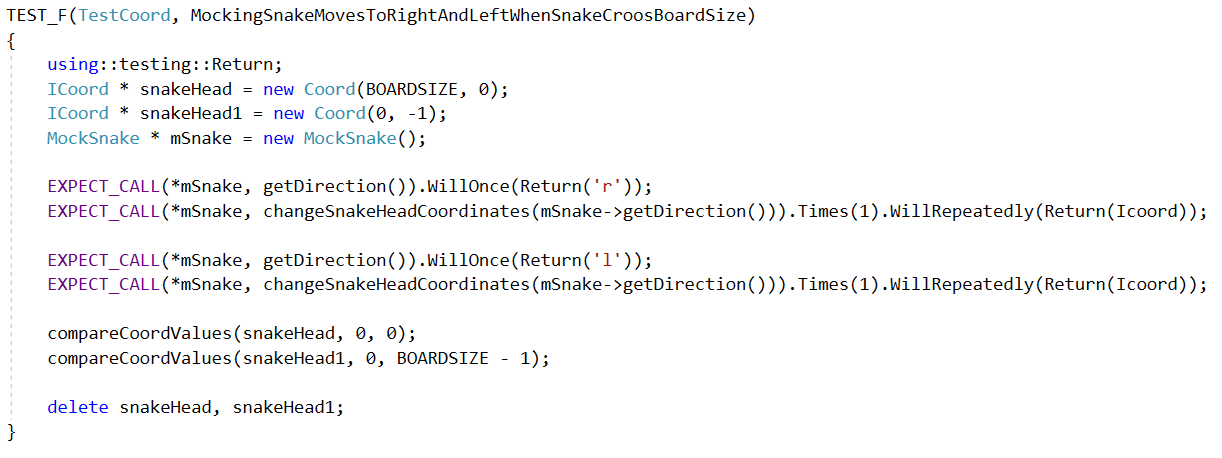
Po zdefiniowaniu klasy tworzę w niej pola snakeHead, snakeTail, fieldToClear które są wskaźnikami typu Coord, zawierającymi informacje o koordynatach na tablicy (obiekt Board), oraz pole Coord\_Container typu deque, w który to kontenerze będą przetrzymywane informacje o współrzędnych każdego z elementów ciała węża.



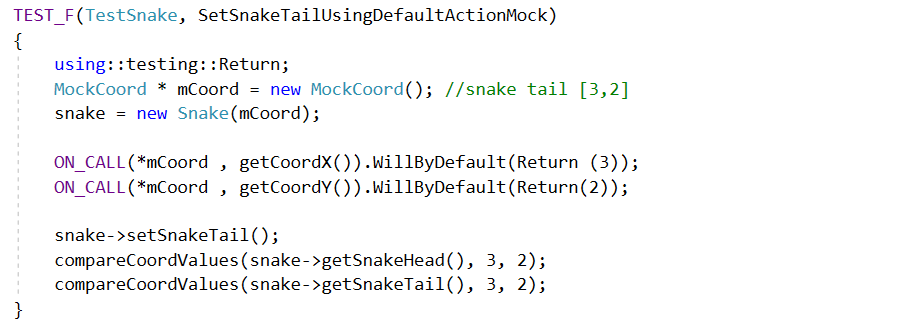
Do pobierania współrzędnych z kontenera używam wbudowanej funkcji .at() która w razie odwołania się do elementu spoza wielkości kontenera rzuca wyjątkiem out\_of\_range. Do sprawdzenia tego przypadku w testach będę używał asercji EXPECT\_THROW:



Do zmiennej typu char direction, będzie przypisany wciśnięty klawisz z klawiatury i zamieniony na odpowiedni kierunek ruchu. Sterowanie odbywa się tylko za pomocą klawiszy a, s, d, w, p, każdy inny klawisz jest ignorowany. Domyślnie wąż zaczyna od ruchu prawo, metoda changeSnakeHeadCoordinates() pobiera wartość z pola direction i na jej podstawie zmienia współrzędne obiektu Snake na planszy. Gdy koordynaty pola snakeHead pokrywają się z granicą wielkości planszy, wąż pojawia się z drugiej strony planszy, za co odpowiedzialna jest metoda checkBorders() w klasie Coord. Do testowania tego przypadku używam Obiektów pozorujących tzw. Mocków, w tym przypadku jest to Mock Interfejsu Klasy Snake, na którym symuluje ruch węża poza granicę planszy i zmianę jego współrzędnych na odpowiadające przeciwnej stronie planszy po jakiej może się on poruszać.



Za wielkość węża, jest odpowiedzialne pole snakeLength, początkowa przyjmujące wartość dwa, a po każdym razem, kiedy koordynaty pola snakeHead i AppleCoord się pokryją zwiększające ono swoją wartość o jeden. Pole SnakeTail jest wyznacznikiem końca węża na planszy, jest to zarazem pierwszy element w kontenerze typu deque, elementem zerowym jest pole fieldToClear, które jest wskazują na współrzędne które obiekt snake właśnie opuszcza, ostatnim elementem kontenera są współrzędne pola SnakeHead. W momencie, gdy wąż zjada jabłko, współrzędne pól SnakeTail i fieldToClear nie ulegają zmianie. Testuję ustawianie pól za pomocą obiektów pozorujących i wywołania ON\_CALL().



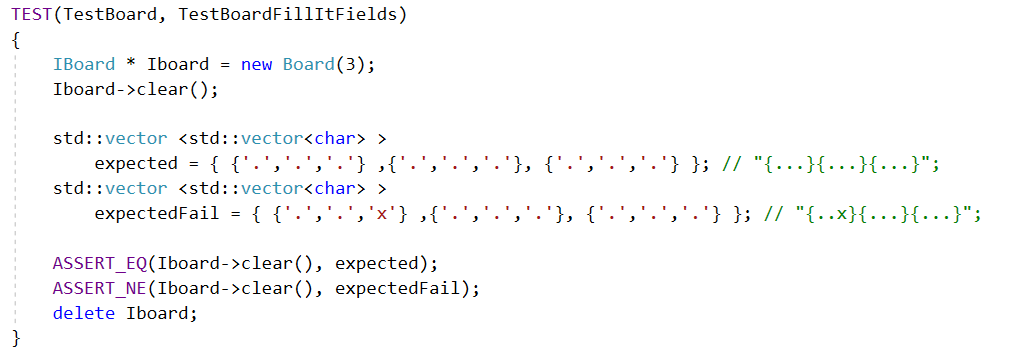
Wąż ginie w momencie, gdy współrzędne jego głowy pokrywają się ze współrzędnymi już znajdującymi się w Coord\_Container, czyli jedno z pól ciała węża, jest to sprawdzane w metodzie snakeHitItself();

## Klasa Apple

Jest niewielką klasa przetrzymującą współrzędne obiektu Apple który będzie wyświetlany na szachownicy Board. Tworze również pole eatApple typu bool które będzie informowało mnie o momencie, w którym koordynaty apple i snakeHead się pokryją.

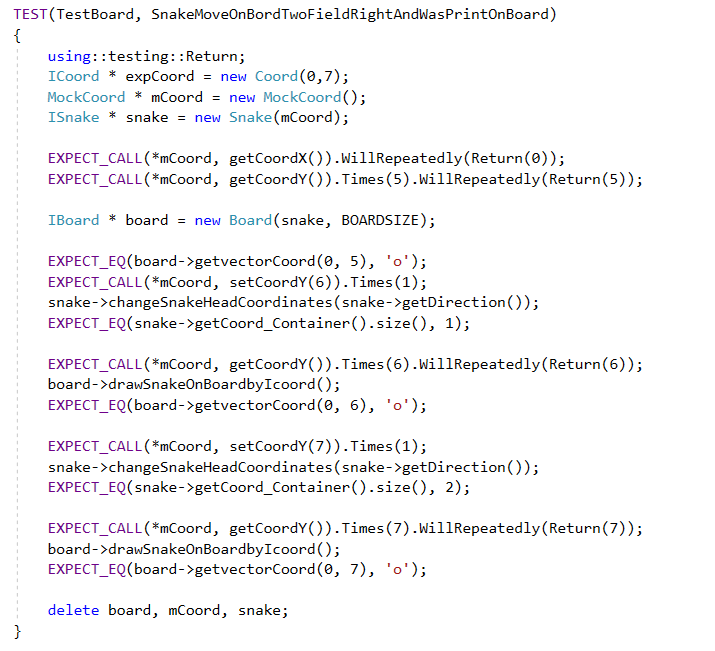
## Klasa Board

Ta klasa odzwierciedla obszar, po którym porusza się wąż, jak w szachownicy każde pole ma swoje wymiary. Przedstawione to jest w aplikacji w formie dwuwymiarowego vectora przechowującego typ char. Zaczynam od ustawienia wymiary szachownicy i uzupełnieniu jej pustymi polami. W razie próby ustawienia tablicy na wartość poniżej zera, zostaje to wyłapane i zmienione na wartość domyślną. Do porównania zawartości wektorów mogę użyć EX PECT\_EQ oraz EXPECT\_NE, asercje te obsługują kontenery.



Tworze również pole score które będzie zliczać wynik liczony w liczbie zjedzonych jabłek przez węża.

Do testowania ruchów węża na tablicy używam frameworku GMock który tworzy atrapy obiektów i symuluje ich zachowania:



# Bibliografia

TDD. Sztuka tworzenia dobrego kodu - Kent Beck

TDD. Programowanie w Javie sterowane testami

Growing Object-Oriented Software Guided by Tests - Steve Freeman, Nat Pryce

MOJE:

http://www.samouczekprogramisty.pl/test-driven-development-na-przykladzie/

https://infotraining.bitbucket.io/cpp-tdd/

https://mfiles.pl/pl/index.php/Test\_driven\_development

http://komputeks.pl/tdd-programowanie-javie-sterowane-testami-p-9607.html

<https://github.com/BillSchofield/TDDIntro>

https://www.vogella.com/tutorials/JUnit/article.html

TetrisGame: (Snake)

* <https://www.youtube.com/watch?v=KD7wHKN22DQ>
* <https://www.youtube.com/watch?v=fmddhJQIIcc>

<https://github.com/selcouthlyBlue/agile_katas> - GRY

<https://richard.jp.leguen.ca/tutoring/soen343-f2010/tutorials/getting-started-on-rock-paper-scissors/>

https://www.oreilly.com/library/view/java-extreme-programming/0596003870/ch04s17.html#jextprockbk-CHP-4-EX-13

<https://www.pluralsight.com/guides/mvp-with-testing-part-2>

<http://www.learnjavacoding.com/definitions/snake/>

<https://www.youtube.com/watch?v=rMKHLz3liuk>

<https://www.youtube.com/watch?v=rMKHLz3liuk&list=UUpw6gOyEUCeSJDtcFhS_7Zw&index=4>

http://articulos.conclase.net/?tema=juegos&art=snake&pag=000

1. <https://pl.wikipedia.org/wiki/Zapachy_kodu> (data odczytu 17 styczeń 2019) [↑](#footnote-ref-1)
2. https://pl.wikipedia.org/wiki/Test\_jednostkowy (data odczytu 20 styczeń 2019) [↑](#footnote-ref-2)
3. Joanna Nowakowska, Lucjan Stapp „Test Driven Development”, kwartalnik TESTER.PL, nr 4, str.34 [↑](#footnote-ref-3)
4. https://pl.wikipedia.org/wiki/Test\_pokrycia (data odczytu 20 styczeń 2019) [↑](#footnote-ref-4)