**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](#_Toc10210070)

[Cel Pracy 5](#_Toc10210071)

[1. Test-Driven Development 6](#_Toc10210072)

[1.1 Koncepcja TDD 6](#_Toc10210073)

[1.2 Trzy prawa TDD 6](#_Toc10210074)

[1.3 Cykl Red-Green-Refactor 7](#_Toc10210075)

[1.4 Mocne punkty TDD 8](#_Toc10210076)

[1.5 Rodzaje testów oprogramowania: Testy jednostkowe i integracyjne 8](#_Toc10210077)

[1.6 Wprowadzenie do testów jednostkowych/modułowych 8](#_Toc10210078)

[1.7 Atrybuty testów jednostkowych - F.I.R.S.T. 9](#_Toc10210079)

[2. Projektowanie obiektowe pod kątem testów - zagadnienia związane z testowaniem 10](#_Toc10210080)

[2.1 S.O.L.I.D 10](#_Toc10210081)

[2.1.1 Zasada pojedynczej odpowiedzialności 10](#_Toc10210082)

[2.1.2 Zasada otwarte-zamknięte 10](#_Toc10210083)

[2.1.3 Zasada podstawiania Liskov 10](#_Toc10210084)

[2.1.4 Zasada segregacji interfejsów 11](#_Toc10210085)

[2.1.5 Zasada odwrócenia odpowiedzialności 11](#_Toc10210086)

[2.2 Dependency Injection 11](#_Toc10210087)

[2.3 Refaktoryzacja kodu 11](#_Toc10210088)

[2.3.1 Duplikacja kodu 11](#_Toc10210089)

[2.3.2 Zły kod / code smell 12](#_Toc10210090)

[2.3.3 Przeprowadzenie refaktoryzacji 12](#_Toc10210091)

[3. Środowiska testowe w językach obiektowych 13](#_Toc10210092)

[3.1 Zasada Arrange – Act - Assert 13](#_Toc10210093)

[3.2 GoogleTest C++ Testing Framework 14](#_Toc10210094)

[3.2.1 Tworzenie Testu 14](#_Toc10210095)

[3.2.2 Asercje 15](#_Toc10210096)

[3.2.3 Faza konfiguracji testu - Test fixtures 17](#_Toc10210097)

[3.2.4 Wykonanie Testu 17](#_Toc10210098)

[3.3 Google Mock 18](#_Toc10210099)

[3.3.1 Tworzenie Atrapy 18](#_Toc10210100)

[3.3.2 Użycie Obiektu pozorującego w teście 19](#_Toc10210101)

[3.3.3 Ustawienia akcji domyślnych 20](#_Toc10210102)

[3.3.5 Matchers – Dopasowania 24](#_Toc10210103)

[3.3.6 Akcje 24](#_Toc10210104)

[3.3.7 Użycie wielu oczekiwań 25](#_Toc10210105)

[3.3.8 Uninteresting call 26](#_Toc10210106)

[3.4 Java: JUnit 26](#_Toc10210107)

[3.5 t 26](#_Toc10210108)

[3.6 .Net: xUnit.net 28](#_Toc10210109)

[3.7 Python: PyUnit 28](#_Toc10210110)

[4. Techniki Izolacji testów 28](#_Toc10210111)

[4.1 Obiekty pozorujące - Atrapy 28](#_Toc10210112)

[5. Przebieg tworzenia aplikacji 30](#_Toc10210113)

[5.1 Specyfikacja aplikacji 30](#_Toc10210114)

[5.2 Klasa Coord 30](#_Toc10210115)

[5.3 Klasa Snake 32](#_Toc10210116)

[5.4 Klasa Apple 34](#_Toc10210117)

[5.5 Klasa Board 35](#_Toc10210118)

[6. Bibliografia 38](#_Toc10210119)

# 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.

# Cel Pracy

Praca nad jakością oprogramowania i jego walidacja rozwija się nieprzerwanie od dekad tworząc i doskonaląc rożne metody testowania.

W tej pracy będę chciał przybliżyć technikę testowania jednostkowego określaną jako Test-Driven Development, opisać koncepcje i główne zasady tej filozofii. Stworzę i opisze każdy krok tworzenia aplikacji opartej na filozofii TDD w odniesieniu do różnych frameworków w kilku językach programowania. Powstałe aplikacje będą korzystały z kilku bardzo obszernych bibliotek testowy, w pracy będą opisane te elementy bibliotek, które były niezbędne i bardzo przydatne do napisania aplikacji.

# 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 będzie zaimplementowany 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 istnieje 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

Można 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 na mikro - fazach, które są powtarzane jeden po drugim.

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, jest oznaczona kolorem czerwonym. Zakończenie testu powodzeniem, na tym etapie oznacza, że 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 pisany jest kod, nie zwracając uwagi na jakość pisanego kodu, ważny jest jak najszybszym czas implementacji i odzwierciedlenie testu w kodzie. Następnie uruchomione zostają 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 wprowadzane są zmiany w kodzie, bez naruszana jego funkcjonalności, kod jest ulepszany, tworzone są klasy i metody zapobiegające duplikacjom. Ta faza 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

Najmocniejszym punktem TDD, jest szybkość wyłapywania błędów w pisanym kodzie, co zmniejsza czas i koszty odnajdywania błędów w oprogramowaniu. Błędy zostają wykryte już na etapie kompilacji pojedynczych modułów, programista jest klientem własnego kodu, angażując mniej osób i minimalizując czasu potrzebny przy naprawie niedziałającego kodu. Kod jest przemyślany, a daną funkcjonalność można testować bez uruchamiania całego programu.

## Rodzaje testów oprogramowania: Testy jednostkowe i integracyjne

Z punktu widzenia TDD istnieje kilka podstawowych 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.

Bardzo istotne jest 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, test integracyjny natomiast to 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.

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
* Redukują ryzyko
* Umożliwiają bezpieczne przeprowadzenie refaktoringu
* Powinny być szybkie, w pełni zautomatyzowane i powtarzalne (patrz FIRST)
* Testy powinny być łatwe w pisaniu, uruchamianiu i utrzymaniu
* Powinny być proste i czytelne - testy stają się zbyt skomplikowane, gdy pojedynczy test weryfikuje 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 jest odpowiedzialna tylko za jedną rzecz. Tworzenia dużych klas, mających wiele zastosowań, jest bardzo niewskazane w kodzie, w zaistniałym wypadku konieczne jest rozbicie ich na mniejsze pojedyncze klasy o pojedynczej funkcjonalności, co wpływa łatwość refaktoryzacji i obsługi kodu.

### Zasada otwarte-zamknięte

Oprogramowanie (klasy, moduły funkcje) musi być tworzone w sposób otwarty na rozbudowę, a zamknięty na modyfikacje. Bazuje na budowie i zarządzaniem interfejsami (extension points), oraz budowie klas pochodnych z klasy bazowej, wszystko w ramach rozbudowy ich funkcjonalności. Zasada OCP jest kombinacją hermetyzacji i wyodrębniania. Szczególnie istotna, przy pracy z dużymi projektami, gdzie pojedyncza modyfikacja w kodzie produkcyjnym ciągnąć za sobą olbrzymie zmiany w wielu miejscach z ta zmiana powiązanych.

### Zasada podstawiania Liskov

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

### Zasada segregacji interfejsów

Interface segregation principle - 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 można używać żadnych nazw konkretnych klas, tylko interfejsów i klas abstrakcyjnych. Również klasy nie powinny dziedziczyć po konkretnych klasach, jedynie po klasach abstrakcyjnych i interfejsach.

## 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 refaktoryzacje:

### 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 jest wymagana znajomośc 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) .
* 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.

## Zasada Arrange – Act - Assert

Zasada 3A jest wzorcem postępowania podczas pisania testów jednostkowych. Akronim AAA rozwijany 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 należy mieszać 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

## Tworzenie Testu

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

Tworzenie testu rozpoczyna się od utworzenia makra TEST() aby zdefiniować i nazwać funkcję testową. Pełna nazwa testu składa się zarówno z przypadku testowego, jak i nazwy testu. Pierwszym argumentem jest „TestSuiteName” czyli ogólną nazwę przypadku testowego, drugi to „TestName”, czyli bardziej szczegółowa nazwa testu w przypadku testowym.

Nazwa testu może zawierać tylko litery i cyfry. Symbole, w tym podkreślenia, są niedozwolone a same nazwy należy tworzyć zgodnie z konwencja języka C++. Google Framework grupuje wyniki testów według przypadków testowych, więc testy logiczne powinny być w tym samym przypadku testowym.

TEST(TestSuiteName, TestName)

{

ASSERT\_\*

EXPECT\_\*

…

}

Wynik testu jest określony przez asercje, jeśli jedna z nich zakończy się niepowodzeniem, cały test kończy się z niepowodzeniem, gdy wszystkie warunki asercji są spełnione test przechodzi pomyślnie weryfikację.

## Asercje

Asercje zwana również predykatem, jest wyrażeniem logicznym które sprawdza czy podany warunek jest prawdziwy. W bibliotece Google test asercja jest makrem, które przypomina wywołanie metod.

Dwa główne typy asercji:

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

Preferuje się używanie asercji EXPECT\_ \*, która pokazuje więcej informacji o błędzie i pozwala na wyświetlenie więcej niż jednego błędu w teście.

Google Framework składa się w wielu makr asercji, ich podstawowe rodzaje to:

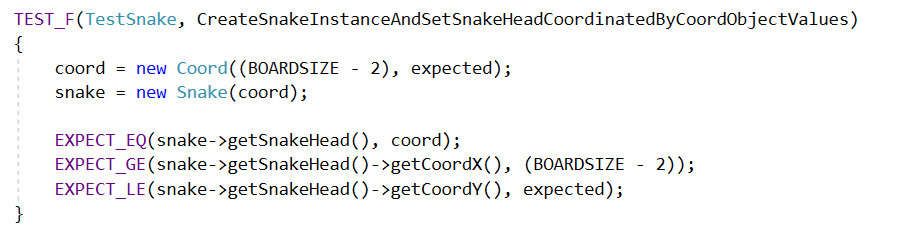
| **Asercja Krytyczna** | **Asercja Niekrytyczna** | **Wynik** |
| --- | --- | --- |
| ASSERT\_TRUE(warunek); | EXPECT\_TRUE(warunek); | Warunek jest prawda |
| ASSERT\_FALSE(warunek); | EXPECT\_FALSE(warunek); | Warunek jest fałszem |

| **Asercja Krytyczna** | **Asercja Niekrytyczna** | **Wynik** |
| --- | --- | --- |
| ASSERT\_EQ(expected, actual); | EXPECT\_EQ(expected, actual); | Expected == actual |
| ASSERT\_NE(val1, val2); | EXPECT\_NE(val1, val2); | Val1 != val2 |
| ASSERT\_LT(val1, val2); | EXPECT\_LT(val1, val2); | Val1 < val2 |
| ASSERT\_LE(val1, val2); | EXPECT\_LE(val1, val2); | Val1 <= val2 |
| ASSERT\_GT(val1, val2); | EXPECT\_GT(val1, val2); | Val1 > val2 |
| ASSERT\_GE(val1, val2); | EXPECT\_GE(val1, val2); | Val1 >= val2 |

Podział na asercje porównujące dwie wartości:

Te porównania działają z podstawowymi typami danych, łańcuchami String i dowolnymi klasami, które mają odpowiadający im przeciążony operator, w przypadku wskaźników test porównuje adresy pamięci (a nie zawartość wskaźników). Dwa wskaźniki są równe, jeśli wskazują dokładnie tę samą lokalizację pamięci.

Testowane wyrażenie powinno być umieszczone w polu „actual”, reprezentujące wartość testowanej funkcji, natomiast oczekiwana wartość, najczęściej obliczana ręcznie w polu „expected”. Wynika to z kolejności w jakiej biblioteka Gtest optymalizuje informacje o błędzie.

Rys. Przykład testu gogle test

Asercje mogą działać również z typem zdefiniowanym przez użytkownika, ale tylko wtedy, gdy jest zdefiniowany odpowiedni operator porównania (np. ==, <, itd.). Wcześniej jednak, może być konieczne użycie ASSERT\_TRUE () lub EXPECT\_TRUE (), aby potwierdzić równość dwóch obiektów typu zdefiniowanego przez użytkownika.

| **Asercja Krytyczna** | **Asercja Niekrytyczna** | **Wynik** |
| --- | --- | --- |
| ASSERT\_STREQ(str1, str2); | EXPECT\_STREQ(str1, str2); | Dwa łańcuchy mają tą samą treść. |
| ASSERT\_STRNE(str1, str2); | EXPECT\_STRNE(str1, str2); | Dwa łańcuchy mają inną treść. |
| ASSERT\_STRCASEEQ(str1, str2); | EXPECT\_STRCASEEQ(str1, str2); | Dwa łańcuchy mają tą samą treść, ignorując wielkość liter. |
| ASSERT\_STRCASENE(str1, str2); | EXPECT\_STRCASENE(str1, str2); | Dwa łańcuchy mają inną treść, ignorując wielkość liter. |

Asercje porównujące ciągi znaków (łańcuchu string):

Do testowanie dwóch obiektów typu string, powinno się użyć asercji porównania, takich jak EXPECT\_EQ. Aby poprawić czytelność po wystąpieniu błędu, można dodać komunikaty o błędach do asercji za pomocą operatora << po nawiasie kończącym asercje.

## Faza konfiguracji testu - Test fixtures

Gdy pisane testy operują na tych samych lub podobnych danych, zalecane jest użycie tzw. Test fixture, czyli specjalnie oznaczonych funkcji, które są wykonywane automatycznie przed wykonaniem każdego z tej grupy testów. Klasa ta pochodzi z klasy testowej :: testing :: zadeklarowanej w gtest.h, a jej użycie wymaga użycia makra TEST\_F () zamiast TEST (), które pozwala to na dostęp do obiektów i podprogramów w funkcji.

Zasady tworzenie test fixture:

* Alokacja lub inicjalizacja zasobów dbywa się w metodzie SetUp lub konstruktorze.
* Dealokacja zasobów wykonywana jest w procedurze TearDown lub destruktora.

Użycie obsługi wyjątków, wymaga użycia metody TearDown, ponieważ wyrzucenie wyjątku z destruktora powoduje niezdefiniowane zachowanie.

* Ten sam test fixture nie jest używany w wielu testach. Dla każdego nowego testu jednostkowego framework tworzy nowe urządzenie testowe.

## Wykonanie Testu

Po napisaniu wszystkich testów w pliku wymagane jest umieszczenie następującej funkcji:

int main (int argc, char \*\* argv)

{

:: testing :: InitGoogleTest (& argc, argv);

return RUN\_ALL\_TESTS ();

}

Wykonanie testów polega na utworzeniu programu testowego w kompilatorze i uruchomieniu pliku wykonywalnego tworzonego przez kompilator.

Polecenie do wpisania w kompilatorze:

* g ++ test.cpp SnakeGameTdd.cpp -lgtest -lpthread -o test

To polecenie kompiluje pliki test.cpp i SnakeGameTdd.cpp i tworzy plik wykonywalny o nazwie test. Komenda –lgtest kieruje kompilator do korzystania z biblioteki Google Testing Framework, komenda –lpthread kieruje do korzystania z biblioteki Pthread, biblioteki wymaganej przez Google Testing Framework. Następnie wystarczy, że uruchomienie komendy

* ./test

Aby uruchomić testy, tworzony jest oddzielny program testowy, który korzysta z własnej funkcji main.

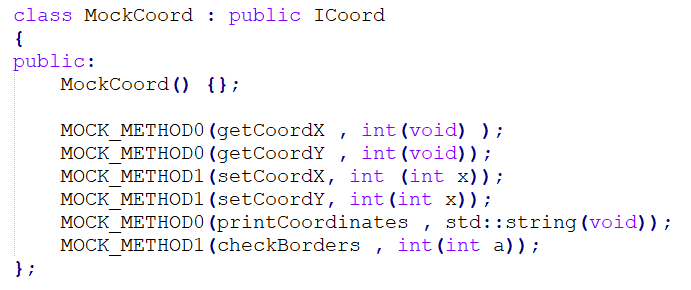
## Google Mock

Jest toozszerzenie frameworku Google Test służące do pisania i używania klas pozorujących Stub/Mock, posiadające duży zestaw akcji i dopasowań (metchers).

## Tworzenie Atrapy

Nowo tworzona klasa mockowa musi dziedziczyć po interfejsie klasy wirtualnej, możliwe jest stworzenie jej na nie-wirtualnych metodach za pomocą szablonów, wymaga to jednak dużo większego nakładu pracy. Definicje funkcji zawierane są w publicznej sekcji, niezależnie od tego czy funkcje mokowane są w klasie bazowej private, protected, public.

Definicja klasy mockowej, musi zawierać się w makrze MOCK\_METHODn (lub MOCK\_CONST\_METHODn () jeśli odnosi się do metody const), gdzie n jest liczbą argumentów które ta metoda przyjmuje. Pierwszym argumentem w makrze jest nazwa funkcji, drugim typ zwracany oraz typ argumentu, który jest przekazany do funkcji.

Rys. Przykład obiektu pozorującego (Mock) klasy ICoord.

## Użycie Obiektu pozorującego w teście

Aby użyć Atrapy w teście należy zastosować kilka prostych kroków:

1. Zaimportowanie potrzebnych nazw Google Mock. Wszystkie nazwy Google Mock znajdują się w przestrzeni nazw testowych, poza makrami
2. Utworzenie Atrapy.
3. Ustawienie oczekiwań na Atrapach.
4. Wywołanie kodu, który zawiera obiekty pozorująca, ewentualnie użycie asercji w celu sprawdzenia wyniku wywołania.
5. Opcjonalnie ustawienie domyślnych akcji obiektów Atrapy.
6. Gdy Atrapa zostanie zniszczona, Google Mock automatycznie sprawdza, czy wszystkie jej oczekiwania zostały spełnione.

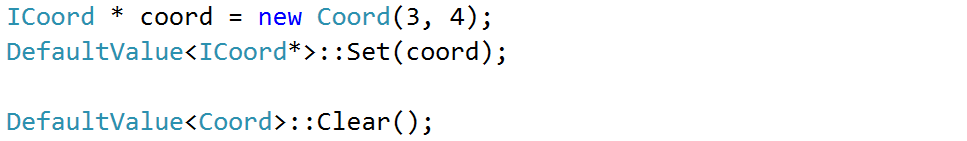


Rys. Wywołanie obiektu pozorującego na klasie Snake.

Narzędzie testowe GMock wymaga, aby oczekiwania były ustawione, zanim nastąpi wywołanie funkcji atrapy, w innym razie zachowanie może nie być poprawne. W szczególności nie wolno przeplatać EXPECT\_CALL () i wywoływać funkcji atrapy. Oznacza to, że EXPECT\_CALL () powinien być odczytany jako oczekiwanie, które nastąpi w przyszłości, a nie jako wywołanie, które już nastąpiło.

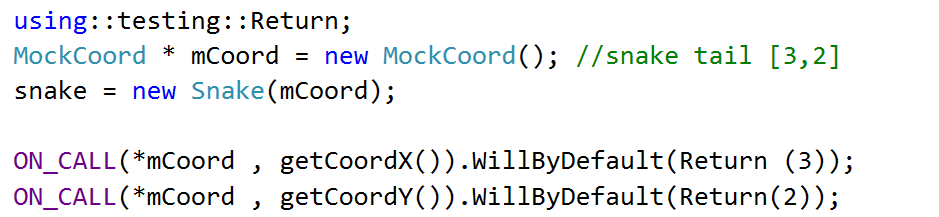
## Ustawienia akcji domyślnych

Google Mock ma wbudowaną domyślną akcję dla każdej funkcji, która zwraca void, bool, wartość liczbową lub wskaźnik i wynosi ona zero. Gdy wymagana jest zmiena tej akcji niezbędne jest użycie szablonu klasy:: testing::DefaultValue.



Rys. Użycie szablonu klasy:: testing::DefaultValue.

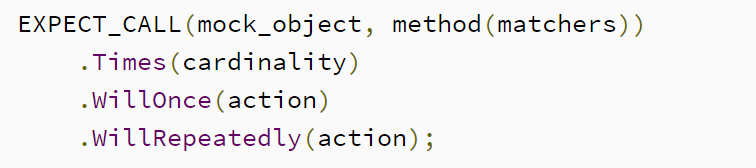
W przykładzie powyżej DefaultValue ustawia domyślną wartość zwracaną dla typu ICoord\*, po czym Clear ją usuwa. Dostosowanie domyślnej akcji dla konkretnej metody, wymaga użycia makra ON\_CALL () wraz z instrukcją WillByDefault, która domyślnie ustawia zwracaną wartość wywołanej metody.



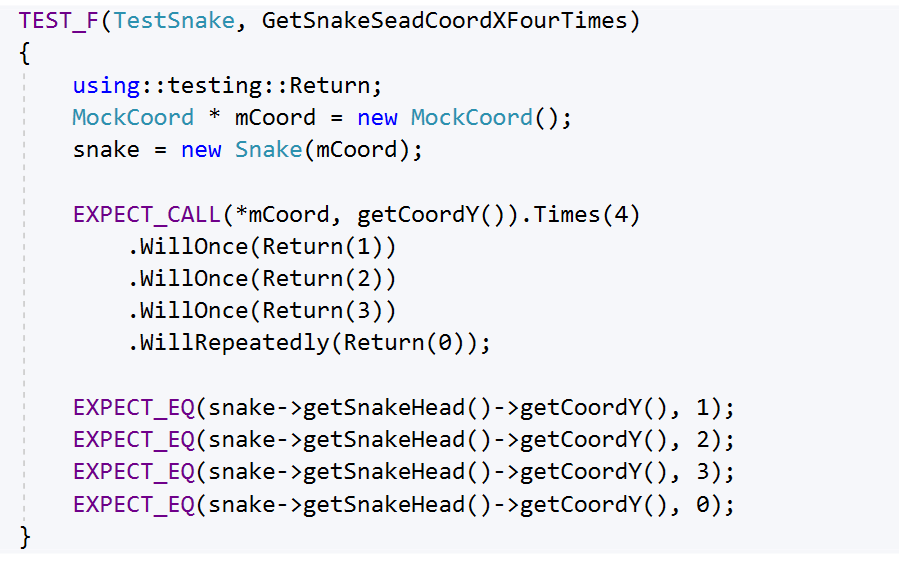
Rys. Użycie makra ON\_CALL z WillByDefault()

* + 1. Ustawienia oczekiwań

Framework Google Mock używa dwóch konstrukcji makr: EXPECT\_CALL oraz ON\_CALL, aby ustawiać zachowanie atrapy. Zasadnicza różnica polega na tym, że ON\_CALL definiuje ca dzieje się w trakcie wywołania metody obiektu, nie definiując żadnych oczekiwań do wywołanych metod, natomiast EXPECT\_CALL ustawia oczekiwanie, że metoda zostanie wywołana z podanymi argumentami, dla podanej liczby razy. Ogólna składnia to:

[[3]](#footnote-3)

Makro składa się z dwóch atrybutów, pierwszy jest instancją obiektu pozorującego, natomiast drugi jego metodą z argumentami, warto zauważyć, że są one oddzielone przecinkiem, po makrze następują klauzule specjalne.



Jak w powyższym przykładzie, który mówi, że na obiekcie pozorującym mCoord zostanie wywołana funkcja getCoordY cztery razy i zwróci następujące wartości: za pierwszym razem 1, za drugim wywołaniem 2, za trzecim trzy i za każdym kolejnym 0. Następnie oczekiwanie jest sprawdzane w asercji.

Kiedy Funkcja atrapy (MOCK) przyjmuje argument, konieczne jest sprecyzowanie parametru, który ma być spodziewany w jej definicji.

EXPECT\_CALL(\*mSnake, changeSnakeHeadCoordinates(‘l’));

W przykładzie powyżej, ustawione jest oczekiwanie wywołania metody changeSnakeHeadCoordinates, wraz ze sprecyzowanym argumentem ‘l’. Jeśli w wywołaniu metody podana zostanie inna zmienna np. ‘m’, oczekiwanie to nie zostało by spełnione.

W przypadku, gdy test dotyczy wywołania samej funkcji changeSnakeHeadCoordinates, bez sprecyzowanego argumentu, zalecane jest wykorzystanie znak „\_” (tzw. Matchers) który jest odczytywany jako każdy możliwy argument spełniający oczekiwanie.

EXPECT\_CALL(\*mSnake, changeSnakeHeadCoordinates(\_));

Pierwszą z klauzul specjalnych występujących po EXPECT\_CALL jest klauzula Times (), która jako argument przyjmuje tzw. Liczebność. Informuje ona o tym, ile razy nastąpi wywołanie, co zapobiega powielaniu pisania tego samego oczekiwania. Argument liczebności nie musi być dokładnie sprecyzowany, czego przykładem jest użycie AtLeast() wewnątrz klauzuli Times.

Ciekawym przykładem jest użycie Times(0), które oczekuje , że dany wywołanie z podanym argumentem nigdy nie nastąpi, a pojawienie się takiego wywołania, będzie skutkowało zakończeniem testu niepowodzeniem.

Framework Google Mock nie wymaga użycia Times(), można ją pominąć a biblioteka sama uzupełni wartość liczebności wg. Następujących zasad:

* Brak w EXPECT\_CALL() klauzul WillOnce () i WillRepeatedly () spowoduje ustawienie liczność na Times (1).
* Wystąpienie n razy ( gdzie n>=1) WIllOnce() bez WillRepeatedly () spowoduje ustawienie liczność na Times (n).
* Jeśli jest n WillOnce () i jeden WillRepeatedly (), gdzie n> = 0, liczność to Times (AtLeast (n)).

Wewnątrz WillOnce() oprócz Return () można także umieścić jedną ze zdefiniowanych akcji z biblioteki Google Mock. Trzeba pamiętać, że instrukcja EXPECT\_CALL ocenia klauzulę działania tylko raz, nawet jeśli akcja może zostać wywołana wiele razy, co powoduje niewskazane skutki uboczne:

int BOARDSIZE = 15;

EXPECT\_CALL(\*mSnake, changeSnakeHeadCoordinates(‘l’))

.Times(3). WillRepeatedly( Return(BOARDSIZE ++));

W powyższym przykładzie mimo oczekiwania wywołania metody 3 razy, wartość BOARDSIZE zawsze będzie zwracana jako 15, ponieważ BOARDSIZE ++ wywołany jest tylko raz jako oddzielny EXPECT CALL().

## Matchers – Dopasowania

Matcher jest predykatem, porównywany do podanej wartości, używanym w makrach ON\_CALL i EXPECT\_CALL lub bezpośrednio porównywanym z podanym atrybutem.

Podstawowy podział dopasowań wg. biblioteki GMock:

| **Dopasowanie** | **Opis** |
| --- | --- |
| Eq(wartość) or wartość | argument == wartość |
| Ge(wartość) | argument >= wartość |
| Gt(wartość) | argument > wartość |
| Le(wartość) | argument <= wartość |
| Lt(wartość) | argument < wartość |
| Ne(wartość) | argument != wartość |
| IsNull() | argument jest NULL pointerem |
| NotNull() | argument nie jest null pointerem |
| Ref(zmienna) | argument jest referencja do zmiennej. |

Można budować złożone elementy dopasowania z tych istniejących przy użyciu AllOf (), AnyOf () i Not () lub dowolnie innych określonych w dokumentacji Google MOCK Framework.

## Akcje

Akcje określają, co powinna zrobić funkcja Atrapy po wywołaniu.

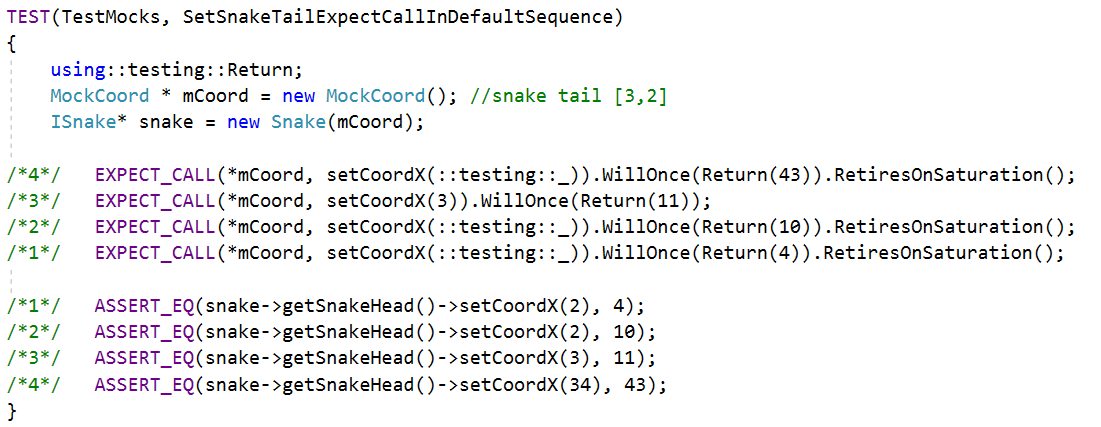
Podstawowy podział wg biblioteki GMock:

| **Matcher** | **Description** |
| --- | --- |
| Return() | Zwracane przez pustą funkcje Atrapy |
| Return(wartość) | Zwraca podaną wartość |
| ReturnArg<N>() | Zwraca N-ty argument. |
| ReturnNew<T>(a1, ..., ak) | Zwraca nowy obiekt T(a1, ..., ak); Za każdym razem, jest tworzony nowy obiekt. |
| ReturnNull() | Zwraca null pointer. |
| ReturnPointee(ptr) | Zwraca wartość wskazaną przez ptr. |
| ReturnRef(variable) | Zwraca referencję do zmiennej. |
| ReturnRefOfCopy(wartośc) | Zwraca referencje do kopi wartości, która żyje tak długo dopóki istnieje akcja. |

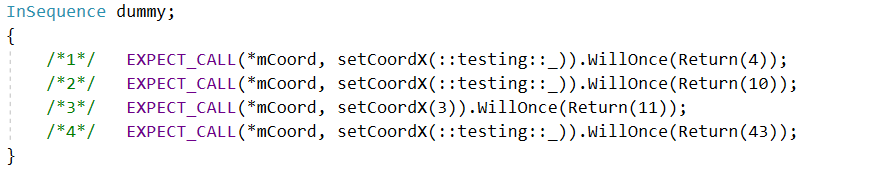
## Użycie wielu oczekiwań

Gdy wywołana jest metoda Atrapy, domyślnie Google Mock Framework przeszuka oczekiwania w odwrotnej kolejności, w jakiej zostały zdefiniowane i zatrzyma się, gdy zostanie znalezione poprawnie zdefiniowane oczekiwanie, które pasuje do argumentów. Oznacza to, że instrukcja zdefiniowane najpóżniej, zawsze ma pierwszeństwo.

Pozwala to użytkownikowi na ustawienie domyślnych oczekiwań w konstruktorze obiektu testowego lub fazie konfiguracji (tets fixtures) urządzenia testowego, a następnie dostosowanie atrapy, poprzez napisanie bardziej szczegółowych oczekiwań w ciele testowym. Tak więc, jeśli masz dwa oczekiwania wobec tej samej metody, a chcesz umieścić tę z bardziej konkretnymi dopasowaniami po drugiej, trzeba definiować najpierw tą bardziej ogólną, a następnie bardziej konkretną po niej. Instrukcja RetiresOnSaturation odpowiada, za to, że gdy wywołana metoda Atrapy posiada EXPECT\_CALL w teście, po jego spełnieniu , oczekiwanie nie będzie więcej brane pod uwagę.



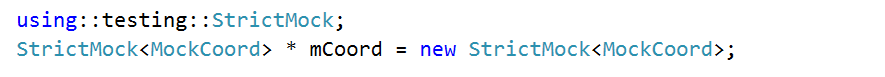
Rys. Domyślna kolejność wywołania oczekiwań

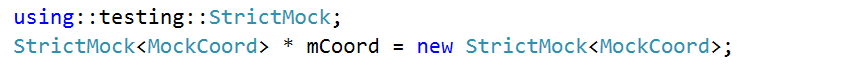
Możliwe jest ustawienie wywołań oczekiwań w ściśle określonej kolejności, służy do tego procedura InSeqence. 

Utworzenie obiektu typu InSeqence powoduje umieszczenie wszystkich wywołań w kolejności ich napisania od góry.

## Uninteresting call

Jeśli w teście zostanie wywołana metoda obiektu pozorującego, lecz brak w nim EXPECT\_CALL dla danej metody, framework wyświetli ostrzeżenie o tzw.” uninteresting call”. Nie jest ono traktowane jako błąd, ponieważ można dodawać nowe metody do interfejsu już po napisaniu testu, ale może także oznacza o niedopatrzeniu, w teście które, może doprowadzić do błędu.

Gdy test wymaga, aby wszystkie wywołania miały swoje oczekiwania EXPECT\_CALL, niezbędne jest użycie StrictMock, który spowoduje, że każde ostrzeżenie będzie potraktowane jako błąd. Odwrotną sytuację powoduje użycie NiceMock w teście, powodując, że ostrzeżenie uninteresting call będzie pominięte.



## Java: JUnit

## t

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.”[[4]](#footnote-4)*

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. Do dyspozycji są 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

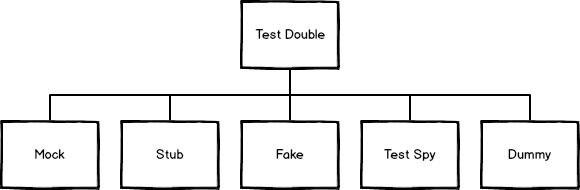
# Techniki Izolacji testów

## Obiekty pozorujące - Atrapy

Test jednostkowy testując pojedynczy element w izolacji, bez zewnętrznych 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 i wyłącznie dla potrzeb testu. Jej zadaniem jest symulowanie zachowania prawdziwych zależności, w oparciu o dane wejściowe które zostaną zdefioniwane.

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. Z reguły własnoręcznie napisana klasa, która posiada minimalną funkcjonalność, żeby spełnić założenia interakcji.
* 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 także ilość wywołań.

# Przebieg tworzenia aplikacji

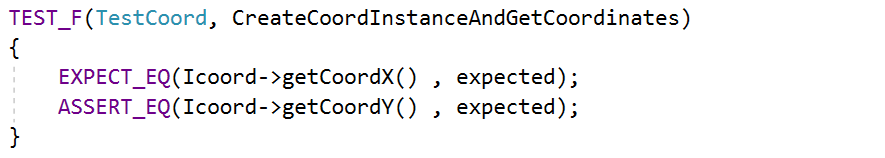
## Specyfikacja aplikacji

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

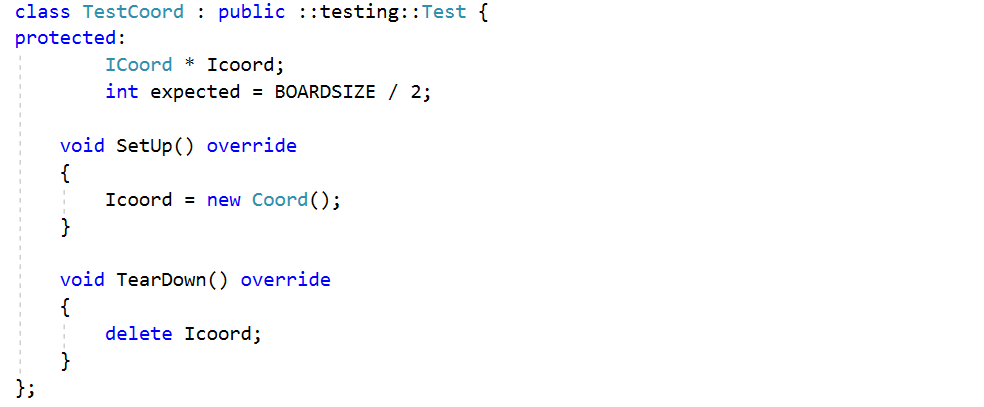
* Gra działa w czasie rzeczywistym
* Wąż porusza się w czterech kierunkach
* Wąż zjada jabłko
* Wąż rośnie
* Wąż może zginąć
* Wąż może przekraczać granice planszy
* Gra kończy się, gdy wąż zje wszystkie jabłka lub zginie.

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 oraz rozrzeszeniu Google Mock. Aplikacja jest stworzona w oparciu o wymienione biblioteki Google, lecz ze względu na obszerny opis bibliotek, w swojej pracy przytoczę i wyjaśnię tylko przykłady które wykorzystałem podczas tworzenia aplikacji.

## Klasa Coord

Zaczynam od tworzenia 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. W fazie konfiguracji testu tworzę funkcje SetUp / TearDown i wykorzystując podstawowe asercje, testuję nowo powstała klasę. 

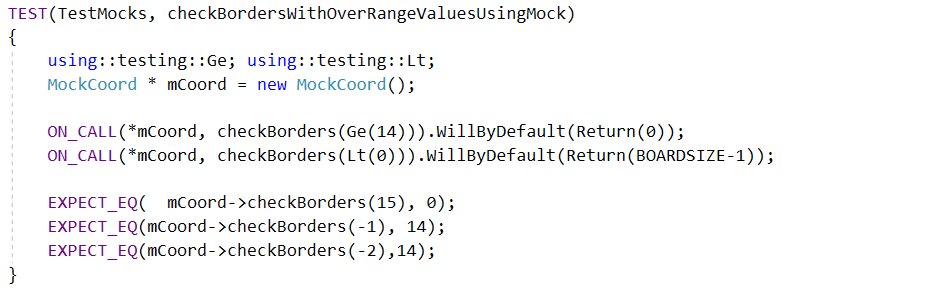
Rys. Podstawowe asercje biblioteki Gtest



Rys. Faza konfiguracji testu

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.

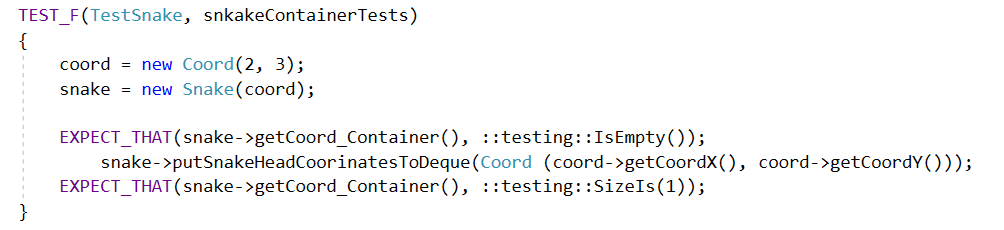
Deklaruję metodę checkBorders() która sprawdza, czy ustawione pola obiektu Xcolumn oraz Yrow nie przyjmują wartości większych niż rozmiar tablicy, a jeśli to nastąpi odpowiednio je modyfikuje. Na podstawie kroków opisanych z działu Użycie Obiektu pozorującego w teście, tworzę test dla metody checkBorders, a następnie ją w kodzie produkcyjnym.



Rys. Atrapa Mock pozorująca wywołanie metody checkBorders

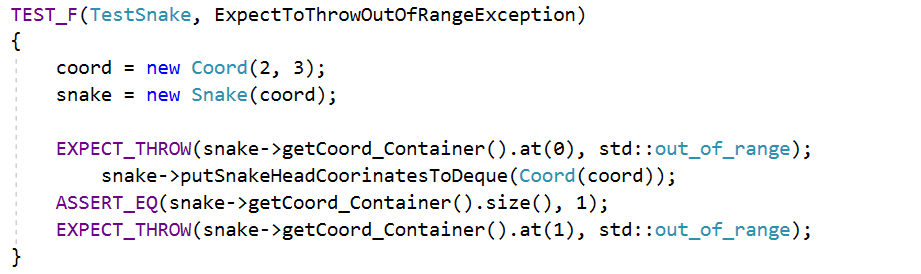
## Klasa Snake

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.

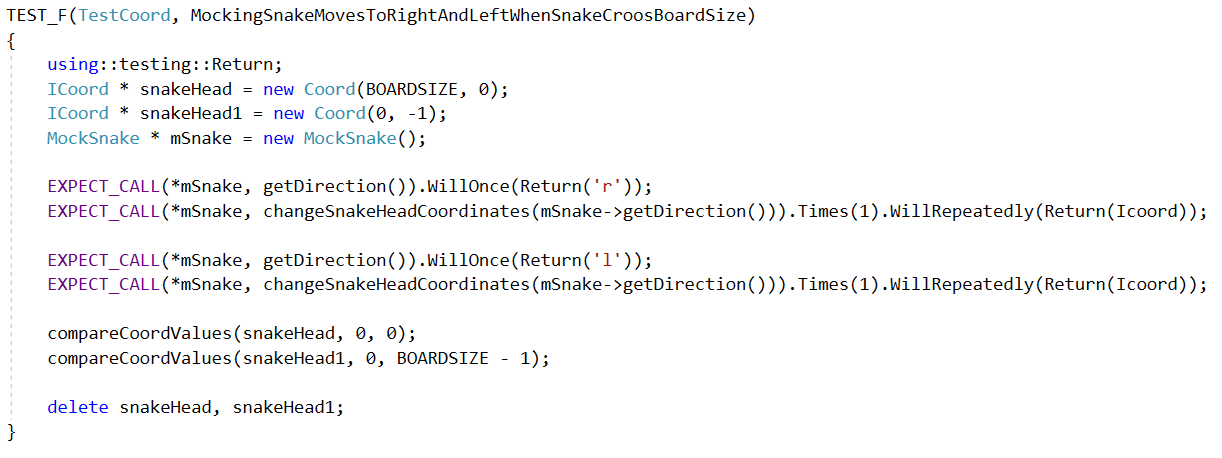


Rys. Użycie wbudowanej asercji obsługującej kontenery

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:

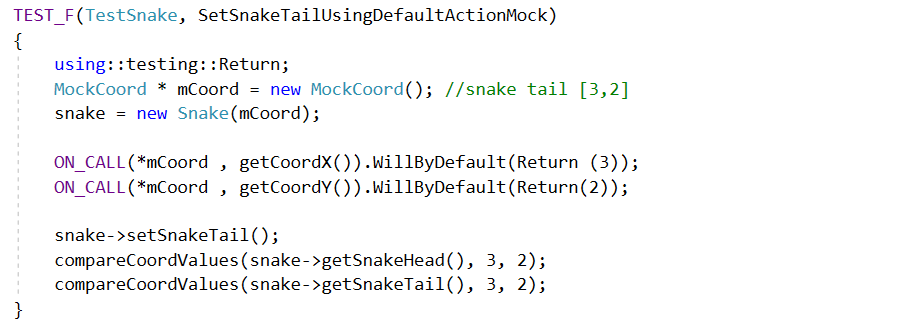
Rys. Użycie asercji EXPECT\_THROW w teście

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.



Rys. Symulowanie zachowania obiektu pozorującego Snake na planszy

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, oraz zmienna score zwieksza się o jeden. Testuję ustawianie pól za pomocą obiektów pozorujących i wywołania ON\_CALL().



Rys. Użycie makra ON\_CALL które zwraca ustawioną wartość domyślną

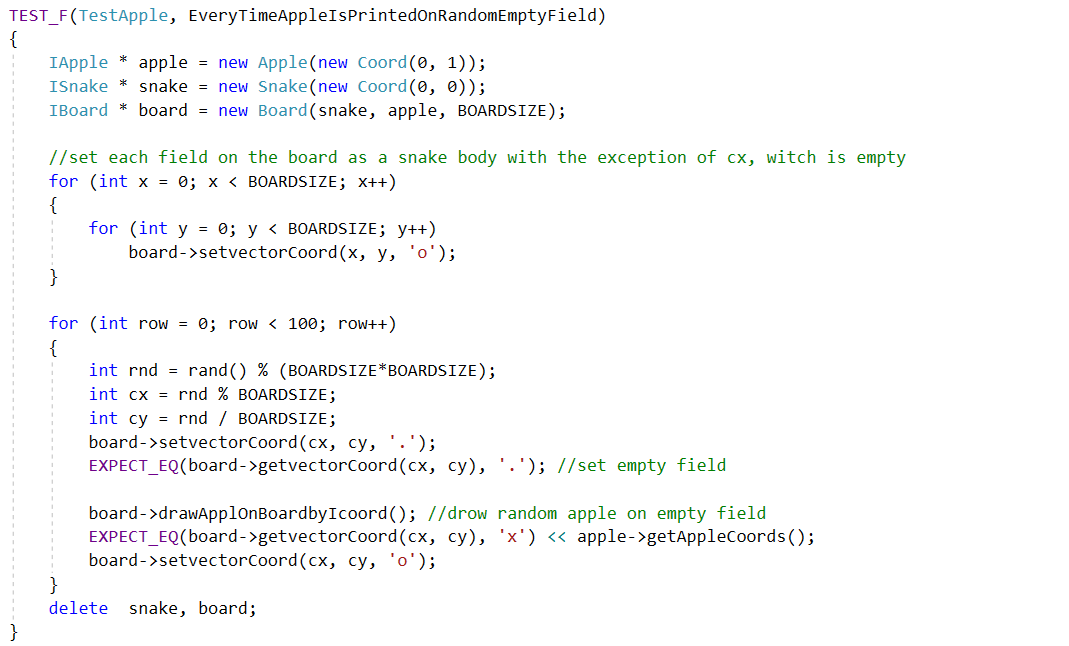
Wąż ginie w momencie, gdy współrzędne jego głowy pokrywają się ze współrzędnymi obecnie wpisanymi w Coord\_Container, czyli z jednym z pól ciała węża, jest to sprawdzane w metodzie snakeHitItself(), po czym zostaje wyświetlony informacja o zakończeniu gry i uzyskanym wyniku.

Gdy wąż zjada ostanie jabłko na planszy a na nowe nie ma już miejsca, grą kończy się zwycięstwem i zostaje wyświetlony komunikat i uzyskany wynik w rozgrywce.

## Klasa Apple

Jest niewielką klasą z polem AppleCoord które przetrzymuje współrzędne obiektu Apple wyświetlanego na szachownicy Board. Istotną metodą jest putRandomAppleOnboard() która losowo wybiera współrzędne jabłka na planszy, w klasie Board sprawdzając jednocześnie czy nie pokrywają się one z już zajętym polem.

Do testowania metody tworze przypadek testowy który wielokrotnie będzie ustawiał jedno wolne pole na planszy, a następnie wywoływał losowanie współrzędnych jabłka na planszy. Asercja wielokrotnie sprawdza, czy wylosowane współrzędne pokrywają się ze współrzędnymi wolnego pola, inny niż oczekiwany wynik powoduje przerwanie testu.

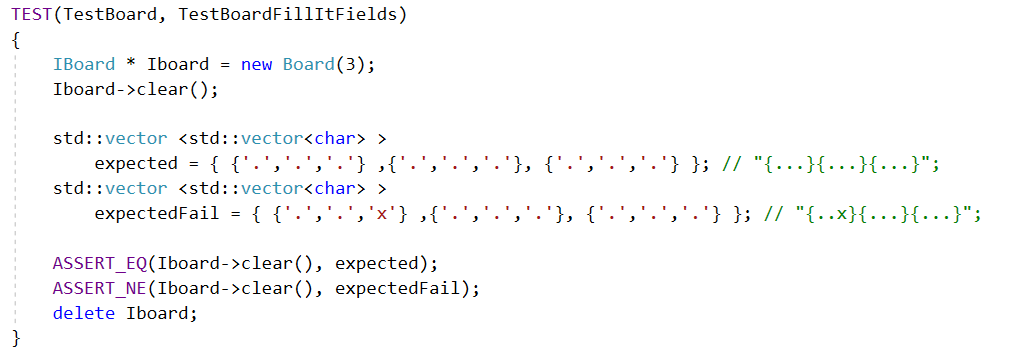


Rys. Wielokrotne wywołanie asercji dla różnych wyników

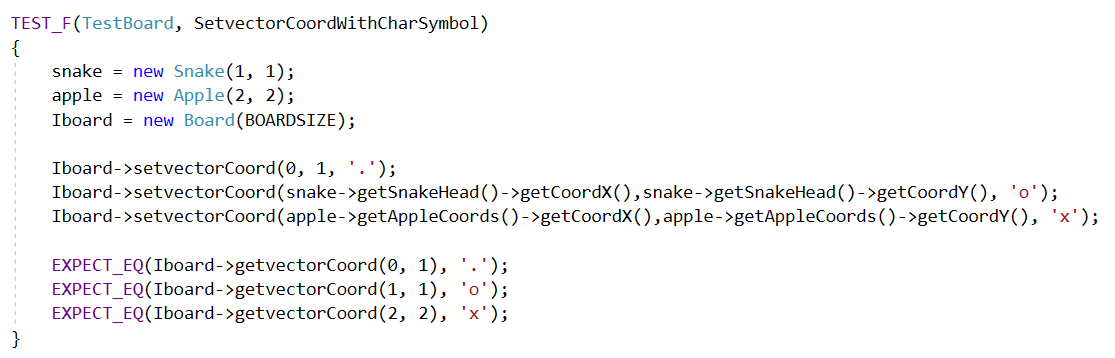
## Klasa Board

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: vector <vector<char> > vec2D.

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 będą użyte asercje EX PECT\_EQ oraz EXPECT\_NE, oraz wbudowanych dopasowani z biblioteki Gtest tzw. Container Matchers.



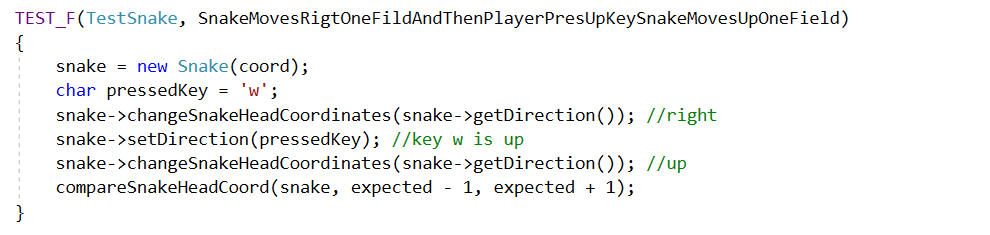
Rys. Porównanie dwóch kontenerów za pomocą asercji.

Tworze pole score które zlicza uzyskany wynik, czyli liczbę zjedzonych jabłek przez węża. Każda z instancji obiektów Snake i Apple jest rysowana na planszy Board, odpowiadają za to funkcje drawSnakeOnBoardbyIcoord() oraz drawApplOnBoardbyIcoord(), które pobierają z obiektu pole z koordynatami i na podstawie tych współrzędnych rysują dany symbol char który będzie wyświetlany w konsoli, funkcja setVectorCoord():

Rys. Porównanie znaku char za pomocą asercji

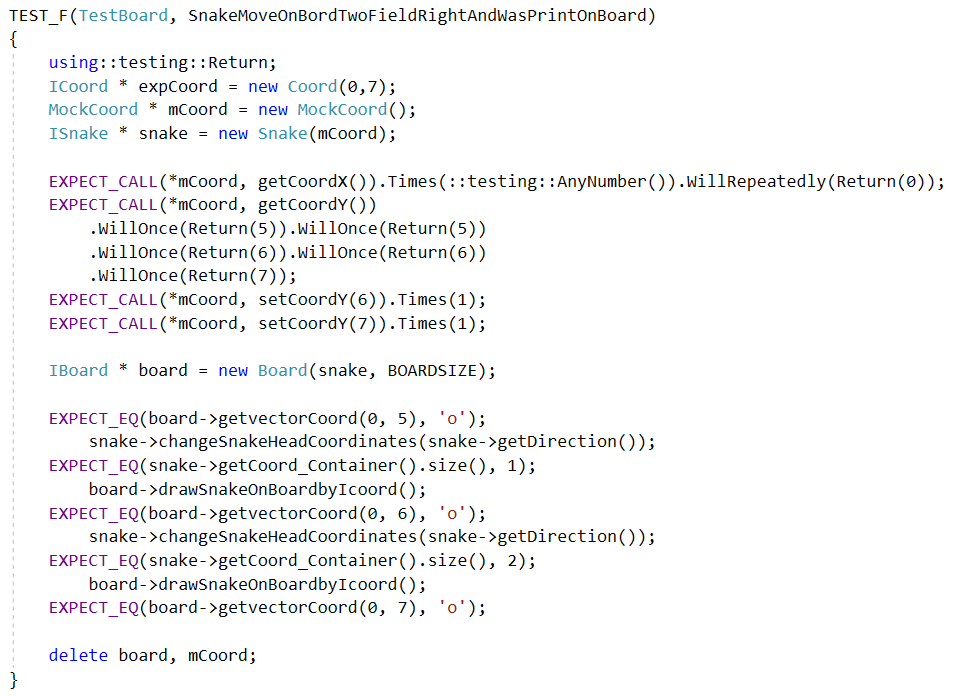
Metoda printVector() jest odpowiedzialna ze, rysowanie planszy w konsoli wraz z odpowiadającym im symbolom pod wskazanymi koordynatami, cyklicznie będąc odświeżana po każdym pojedynczym ruchu obiektu Snake. Aplikacja zakłada, że wąż jest ciągle w ruchu, możliwa jest zmiana kierunku przez wciśnięcie jednego z klawiszy na klawiaturze odpowiedzialnego, za ruch: metoda setDirection(char direction), która przyjmuje jako argument symbol wciśniętego klawisz i na jego podstawie określa kolejny kierunek ruchu, ważne jest tutaj oby obiekt nie mógł bezpośrednio zmienić kierunku na przeciwny, w momencie kiedy wąż idzie w górę nie można zmienić kierunku na dół, co mogłoby spowodować pokrycie się współrzędnych głowy węża z jego ciałem i zakończenie gry.

Weryfikuje to metoda setDirection, która ignoruje takie próby i pozwala na zmianę kierunku na każdy inny możliwy niż przeciwny.



Rys. Porównanie współrzędnych po zmianie kierunku ruchu węża

Do testowania ruchów węża na tablicy używam również biblioteki Gmock, w którym tworze atrapę obiektu Coord i symuluje zmianę jego współrzędnych.



Rys. Symulacja ruchu węża po tablicy

Metoda SnakeEatsApple() jest funkcja która jest bardzo istotna dla aplikacji. Po pierwsze sprawdza za każdym razem czy współrzędne jabłka i głowy węża się pokrywają, a gdy ten warunek jest spełniony długość węża oraz wynik zwiększają się o jeden, następnie przeszukany zostaje vector planszy w poszukiwaniu wolnych pól, w momencie, gdy żadne wolne pole (w aplikacji oznaczone jako ‘.’) nie zostaje znalezione, oznacza to, że wąż zjadł wszystkie jabłka i pole SnakeEatAllApple zostanie ustawione na true co spowoduje zakończenie gry i wyświetlenie komunikatu o zwycięstwie. W momencie znalezienia wolnego pola funkcja checkVectorForFreeSpace() przerywa swoje działanie i zostają wylosowane nowe współrzędne jabłka które zostaną wpisane do kontenera. Gdy koordynaty jabłka i węża różnią się od siebie pole eatApple przyjmuje wartość false. To pole odpowiada również za to czy zmienią się współrzędne pola do wykasowania, zastąpienia symbolem ‘.’



Rys. Test symulujący ciągły ruch węża po planszy

aż do osiągnięcia długości równej dwadzieścia.

Testem weryfikującym zachowanie aplikacji jest test w którym symuluje ciągły ruch węża po całej planszy, gdzie losowo pojawiają się jabłka a momencie pokrycia się współrzędnych jabłka i głowy węża, długość węża zwiększa się o jeden, aż do momentu w którym osiągnie on długość 20.

# 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. <https://chromium.googlesource.com/external/github.com/google/googletest/+/HEAD/googlemock/docs/ForDummies.md> (data odczytu 13 maj 2019) [↑](#footnote-ref-3)
4. Joanna Nowakowska, Lucjan Stapp „Test Driven Development”, kwartalnik TESTER.PL, nr 4, str.34 [↑](#footnote-ref-4)