AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ ZARZĄDZANIA

KATEDRA INFORMATYKI STOSOWANEJ

Praca licencjacka

Programowanie sterowane testami na przykładzie programu generującego drzewo decyzyjne

Test-Driven Development on the example of program that generates a decision tree

Autor: Jerzy Dariusz Noworolnik
Kierunek studiów: Informatyka i ekonometria
Opiekun pracy: dr Beata Basiura

Kraków, 2020

Spis treści

1. Wstęp i cel pracy

W dzisiejszych czasach, w których jakość i prędkość wytwarzania oprogramowania stają się coraz ważniejszymi czynnikami w produkcji oraz utrzymaniu systemów informatycznych, dobór odpowiedniej techniki tworzenia oprogramowania staje się jednym z ważniejszych kroków przy planowaniu projektów informatycznych. Jedną z technik tworzenia oprogramowania jest Test-Driven Development, która ma na celu usprawnienie procesu pisania i refaktoryzacji kodu.

W tej pracy opisze podstawy testowania oprogramowania i główne zasady techniki Test-Driven Development, zaimplementuję tą technikę w programie mającym na celu tworzenie drzewa decyzyjnego z wykorzystaniem algorytmu ID3 oraz przedstawię zalety i wady techniki w porównaniu z szerzej stosowaną praktyką testowania funkcjonalności programu po napisaniu kodu.

Wysoka jakość tworzonego oprogramowania oraz prędkość tworzenia kodu są bardzo ważnymi cechami dobrego programisty. Opanowanie techniki programowania sterowanego testami może zarówno przyśpieszyć pisanie kodu, jak również polepszyć ostateczną jakość programu, stąd moje zainteresowanie i chęć przetestowania tej techniki.

Celem pracy dyplomowej jest przedstawienie techniki Test-Driven Development oraz stworzenie programu według zasad tej techniki. Praca ma za zadanie ukazać proces tworzenia oprogramowania na podstawie testów, oraz pokazać wady i zalety tego podejścia.

2. Test-Driven Development

W tym rozdziale opisana zostanie technika Test-Driven Development, w skrócie TDD. Omówiona zostanie główna idea i prawa rządzące TDD, zostaną również przedstawione podstawy testowania oprogramowania.

2.1. Cel programowania sterowanego testami

Dodawanie nowej funkcjonalności do gotowego programu jest procesem niezwykle skomplikowanym, a w niektórych przypadkach wręcz niemożliwym w pewnych ramach czasowych. Przed takim właśnie zadaniem stanął znajomy głównego popularyzatora techniki TDD Kenta Becka, Ward Cunningham, i to z obserwacji właśnie tego znajomego Beck wyciągnął wnioski i postanowił opisać cały proces, poczynając od motywacji, poprzez metody, kończąc na okazji.

W przykładzie tym motywacją stała się chęć zaspokojenia wymagań potencjalnego klienta, który potrzebował rozbudowania funkcjonalności programu o operacje na większej ilości walut. Metodą była praca i doświadczenie zebrane przez zespół Warda przy dotychczasowej pracy nad programem. Okazją stała się kombinacja obszernych testów, sfaktoryzowanego programu, i języka programowania pozwalającego na izolację poszczególnych modułów.

Przykłady w książce Kenta Becka nie tylko ukazują prędkość, oraz bezawaryjność programu rozwijanego za pomocą tej techniki, ale przede wszystkim pokazują łatwość przystosowania nowych funkcjonalności do istniejącego projektu, bez ryzyka wywołania błędów w już gotowych modułach.

Kent Beck podsumowuje główną ideę techniki Test-Driven Development w dwóch prostych regułach:

- Przed utworzeniem właściwego kodu, napisz testy automatyczne.
- W tworzonym kodzie likwiduj duplikacje. [1]

2.2. Trzy prawa TDD

Idea techniki programowania sterowanego testami została usystematyzowana oraz rozbudowana przez Roberta C. Martina w 2007 roku. Zdefiniował on następujące trzy prawa:

1. Nie możesz zacząć pisać kodu produkcyjnego, dopóki nie napiszesz niezaliczonego testu jednostkowego.
2. Nie możesz napisać więcej testów jednostkowych, niż jest potrzebnych do przetestowania funkcjonalności.
3. Nie możesz pisać większej ilości kodu produkcyjnego, niż wystarczy do zaliczenia obecnie niezaliczonego testu.

Wykonywanie czynności opisanych w tych trzech prawach, oraz przestrzeganie tych praw, zamyka nas w krótko trwającym cyklu pisania testów, a następnie kodu produkcyjnego które te testy spełnia. [4].

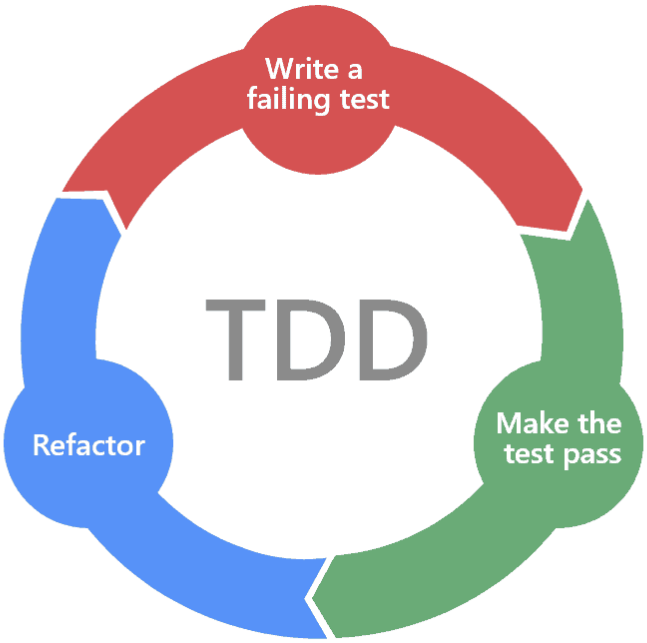
Prawa te nie tylko opisują kolejność działań w technice TDD, ale również zamyka wykonywane czynności w krótkim cyklu, który ma być stale powtarzany. Zapewnia to dokładne pokrycie kodu produkcyjnego testami, jak również zapewnia większą bezawaryjność pisanego kodu.

2.3. Wstęp do implementacji TDD

Przez lata projekty tworzone w zwinnej metodyce prowadziły do nowych i bardziej pragmatycznych praktyk, które miały dostarczyć docelowemu Klientowi program spełniający jego potrzeby w jak najszybszym tempie. W przypadku tego zagadnienia tradycyjne metody oparte na modelu kaskadowym oferują podejście z testami jednostkowymi pisanymi po stworzeniu kodu aplikacji, z angielskiego *"Test-Last"*. Takie podejście do tworzenia oprogramowania pokazało swoje ograniczenia, ponieważ testy pisane po stworzeniu kodu produkcyjnego często były dostosowywane do kodu, a nie na odwrót, lub w niektórych przypadkach całkowicie pomijane.

Sposobem na rozwiązanie problemu jakości testów, może stać się zastosowanie technik TDD, które zakładają podejście *"Test-First"*, w którym testy jednostkowe pisane są na początku. Dzięki temu podejściu to kod jest dostosowywany do testu, a nie na odwrót. Dodatkową zaletą jest większe pokrycie kodu testami jednostkowymi.

Poprzez połączenie programowania, tworzenia testów jednostkowych oraz ciągłej refaktoryzacji kodu, TDD staje się praktyką dzięki której możemy napisać czysty kod, który jest łatwy w utrzymaniu i rozbudowywaniu.



Rysunek 2.1 – 3 phases of TDD. Źródło: [6]

Podczas wdrażania TDD warto zapoznać się z trzema fazami, które stanowią trzon pisania dobrze skomponowanych testów:

- Faza **CZERWONA**. Piszemy test jednostkowy który ma zakończyć się niepowodzeniem.
- Faza **ZIELONA**. Jak najszybciej tworzymy kod który ma za zadanie zaliczenie poprzedniego testu, nie przykuwając większej uwagi do jakości.
- Faza **REFAKTOR**. Przeprowadzamy refaktoryzację kodu produkcyjnego oraz kodu testującego, zarazem uważając by nie zmienić zachowania zewnętrznego ukazanego przez zaliczony test z fazy czerwonej. [5]

2.4. Cykl życia TDD

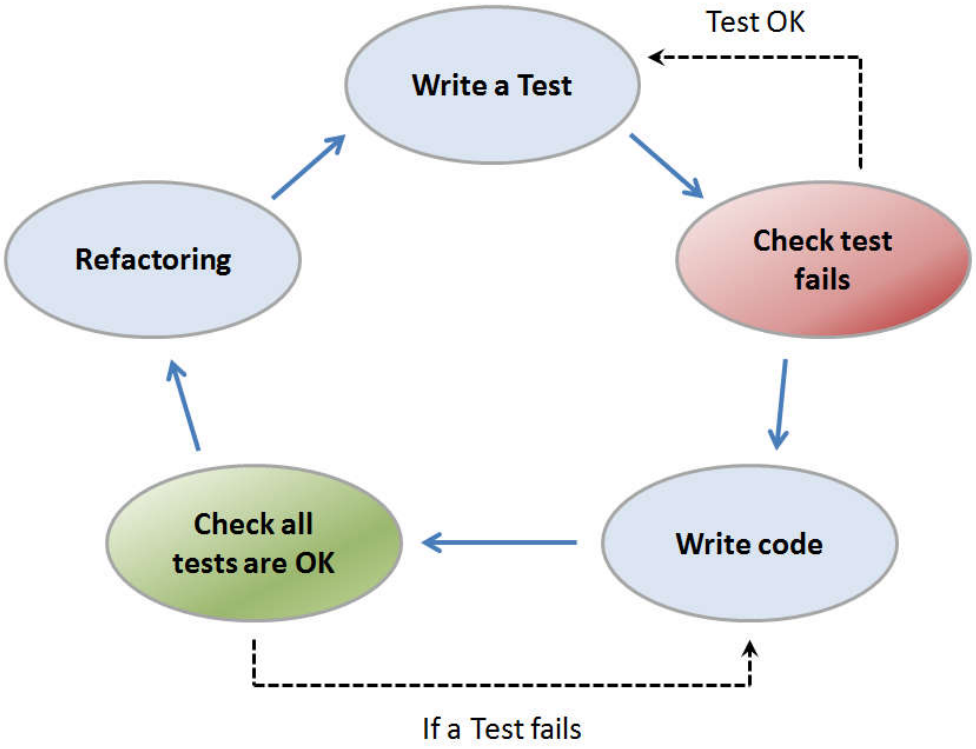
Pisząc zgodnie z techniką Test-Driven Development w pierwszej kolejności należy wiedzieć, co chcemy przetestować oraz jakich wyników oczekujemy od sprawdzanej funkcjonalności. Poprzez taki zabieg lepiej zrozumiemy logikę biznesową i co za tym idzie, nie będziemy programować bez wyznaczonego celu. Idąc tym tropem warto zaimplementować przypadki użycia, czyli z angielskiego *"use cases"* oraz historyjki użytkownika, z angielskiego *"user stories"*. Poprawnie i czytelnie stworzony test będzie stanowił biznesową dokumentację, którą w każdej chwili możemy przekazać do osoby biznesowej mogącej sprawdzić spójność logiki aplikacji.

Po napisaniu przykładowego testu należy go uruchomić i oczekiwać błędu kompilacji. W tym przypadku wystąpi taka sytuacja, ponieważ nie posiadamy jeszcze kodu, który będziemy chcieli przetestować.

W kolejnym kroku zabierzemy się za pisanie funkcjonalności odpowiedzialnych za logikę biznesową. Pisząc kod będziemy starać się o jego skrócenie w jak największym stopniu. Skompresowanie go do jak najbardziej minimalistycznej wersji sprawi, że po uruchomieniu testu przejdzie on w stan "zielony", czyli przeprowadzany test zakończy się pozytywnie.

Następnie zostanie wprowadzony ważny krok, który sprawi, że oprogramowanie będzie w ciągłym tempie rozwijane prawidłowo. Nastąpi refaktoryzacja kodu mająca na celu utrzymanie testowanego kodu przy takich samych standardach, który posiada kod będący na produkcji. Dlatego wykonując tą procedurę, staramy się wykonywać ją równie sprawnie i drobiazgowo na produkcji, jak i przy przeprowadzanych przez nas testach.

Po wykonaniu wszystkich poprzednich kroków cykl się zamyka. Jednak nie musi być to koniec wykonywanych działań. Jeżeli wprowadzone testy nie pokrywają całej logiki biznesowej, to powtarzamy wszystkie operacje ponownie, czyli cykl się powtarza, aż do skutku.



Rysunek 2.2 – TDD Cycle. Źródło: [5]

W całej technice nasuwa się problem długości takich cykli. Założeniem TDD jest pisanie ich jak najkrótszych, czyli tworzenie dużej ilości testów pokrywających małe wycinki implementowanych funkcjonalności. Na pewno pilnując wielkości cyklu warto starać się zachować zdrowy rozsądek. Testy powinny być na tyle długie, aby można było je w prosty sposób zrozumieć i wiedzieć co w nich przetestowano. [7]

2.5. Testy jednostkowe

Testy jednostkowe, nazywane również modułowymi, wykonywane są na etapie wytwarzania kodu. Są przeprowadzane najczęściej przez twórcę kodu produkcyjnego podlegającego testowaniu. Testy te mają na celu weryfikację poprawności zaimplementowanych rozwiązań, oraz sprawdzenia czy realizują one swoją funkcję zgodnie z założeniami. Odnoszą się one do ściśle odizolowanego fragmentu kodu i nie mają żadnego wpływu na zewnętrzne funkcje programu.

Testy te służą wykrywaniu błędów w jak najwcześniejszej fazie produkcji oprogramowania. Dzięki wykrywaniu, oraz wczesnym usuwania błędów, zmniejszamy szansę na wystąpienie negatywnych skutków w innych metodach, oraz minimalizujemy ryzyko propagacji negatywnego wpływu wadliwego kodu ma pozostałe moduły. Implementacja oraz wykonywanie testów jest bardzo ważna, ponieważ kompilator nie jest wstanie zweryfikować poprawności kodu odnosząc się do założonych funkcjonalności. Regularne testowanie krótkich fragmentów kodu zmniejsza prawdopodobieństwo wystąpienia problemów z uruchomieniem gotowej aplikacji, lub pewnych jej funkcjonalności. Usuwanie problemów w fazie implementacji kodu znacznie zwiększa jakość produktu, oraz skraca czas procesu wytwórczego oprogramowania. [3]

2.6. Testy integracyjne

W przypadku rozbudowanego oprogramowania opartego na wielu różnych systemach, podsystemach i aplikacjach, niezbędne są testy sprawdzające heterogeniczność ostatecznie scalonego rozwiązania. Testy te nazywamy testami integracyjnymi.

Przeprowadzenie testów integracyjnych wymaga uprzedniego połączenia modułów, oraz weryfikacji interakcji pomiędzy nimi. Po udanym scaleniu wszystkich poszczególnych podsystemów i po pozytywnym rozstrzygnięciu testów integracyjnych, możemy traktować wszystkie moduły jako jeden system.

Testy integracyjne służą wykazaniu następujących wartości:

- Logiki – czy nie wystąpiły problemy natury technologicznej i czy wzajemnie świadczone usługi spełniają oczekiwania.
- Wzajemnego oddziaływania – jak zachowują się poszczególne elementy systemu, jeśli w innym elemencie dojdzie do błędu lub awarii.
- Logiki biznesowej – czy wzajemnie powiązane podsystemy spełniają założenia procesu biznesowego
- Wielomodułowości - czy infrastruktura zapewnia odpowiednie warunki pracy systemu.
- Pokrycia wymagań – sprawdzenie czy nie ujawniły się luki bądź potrzeby nie przewidziane podczas procesu projektowania. [3]

2.7. Czystość testów

Pisząc kod aplikacji oraz testy do poszczególnych składowych programu powinniśmy starać się tworzyć go jak najlepiej. Podczas oprogramowywania kolejnych projektów polepszamy swój warsztat. Chcemy stawać się coraz lepsi w swojej dziedzinie. Dlatego chcemy, aby nasz kod był coraz lepszy.

Książka pod tytułem "Czysty Kod" napisana przez Roberta C. Martina opisuje najlepsze praktyki pisania dobrego oprogramowania. W rozdziale na temat testów jednostkowych opisany został akronim F.I.R.S.T, który zawiera pięć zasad pisania czystych testów:

- Szybkie (*Fast*). Tworzone testy powinny być szybkie, czyli powinny działać szybko. Jeżeli działają one powoli, przestajemy z nich często korzystać, ponieważ nie są wydajne. Z tego powodu możemy wpaść w problem polegający na nie znalezieniu wystarczająco szybko podatności w programie. Na końcu czystość kodu spadnie i może zacząć się psuć.
- Niezależne (*Independent*). Chcąc napisać test należy pamiętać, aby nie był zależny od innego testu. Konfiguracja warunków nie powinna przechodzić pomiędzy testami. Każdy z nich ma być uruchamiany niezależnie od innych oraz w dowolnej kolejności. W przeciwnym wypadku może zajść sytuacja gdy nastanie lawina błędów, co sprawi, że nie będziemy w stanie zdiagnozować przyczyny awarii.
- Powtarzalne (*Repeatable*). W każdym środowisku test powinien uruchamiać się tak samo. Na prywatnym laptopie, na produkcji, czy stanowisku deweloperskim dążymy do powtarzalności. Nie wykonując tego kroku będziemy mieli wymówkę, gdy się nie powiodą lub nadejdzie problem z ich kompilacją.
- Samokontrolujące się (*Self-Validating*). Każdy test ma posiadać jeden logiczny parametr wyjściowy, który może się udać lub też nie. Prostota sprawi, że nie będziemy wykonywać długiej analizy wyników oraz kodu. Jeżeli testy nie kontrolują swojego działania, to niedługa droga do poważnej awarii.
- O czasie (*Timely*). Ostatnia zasada polega na pisaniu testów w odpowiednich momentach. Na przykład testy jednostkowe tworzymy dokładnie przed napisaniem testowanego kodu produkcyjnego. Pisząc go po tej operacji, dojdziemy do sytuacji, że ciężko będzie dokonać przetestowania poszczególnych funkcjonalności będących już na produkcji. [2]

3. Implementacja aplikacji

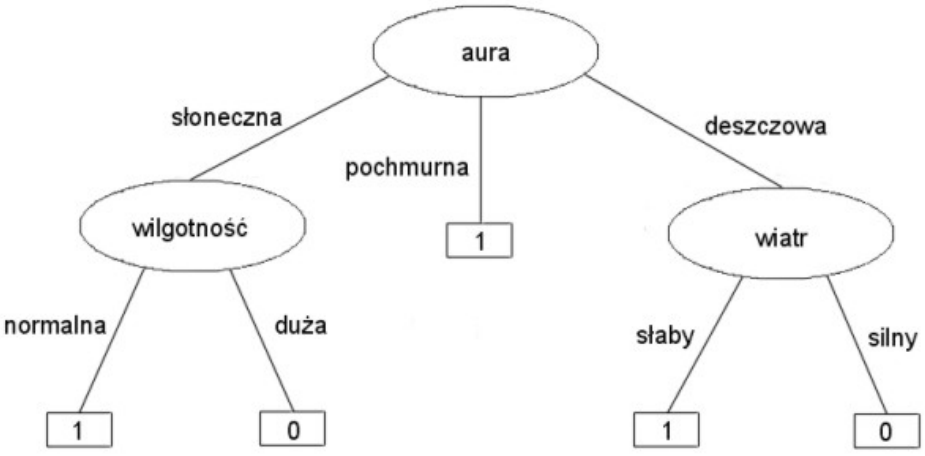
W tym rozdziale zostanie opisany proces tworzenia programu generującego drzewo decyzyjne za pomocą techniki TDD. Na początku zapoznamy się z definicją drzewa decyzyjnego, oraz opisem algorytmu ID3 na podstawie którego zostanie stworzony program. W dalszych podrozdziałach zaprezentuję proces tworzenia testów, pisania kodu oraz refaktoryzacji.

Do tworzenia testów oraz pisania programu wykorzystam środowisko *Microsoft Visual Studio Community 2019.* Cały projekt będzie napisany w języku programowania C#.

3.1. Definicja drzewa decyzyjnego

W projekcie zastosowane są drzewa decyzyjne wykorzystywane do reprezentacji hipotez. Posiadają one strukturę drzewa, jednak różnią się specjalnym znaczeniem liści, węzłów, czy krawędzi. Każdy z nich posiada swoją rolę:

- Węzły odpowiadają za testy, które wykonywane są na poszczególnych wartościach atrybutów przykładów.
- Krawędzie posiadają w sobie wyniki testów.
- Liście zawierają etykiety kategorii.



Rysunek 3.1 – Przykład drzewa decyzyjnego. Źródło: [8]

Posiadając już stworzone drzewo decyzyjne, można wspomnieć o regułach tworzących się za pomocą łączenia, czyli koniunkcji poszczególnych testów posiadających swoją drogę od korzenia do liścia, który reprezentuje daną kategorię.

Zaletą takiego zastosowania na pewno jest umożliwienie zaprezentowania dowolnie skomponowanych zagadnień. Porównując tego typu drzewa do innych reprezentacji hipotez, można stwierdzić, że ich złożoność pamięciowa jest dużo mniejsza. Podobnie sytuacja wygląda z złożonością obliczeniową, która poprzez ilość atrybutów staje się ograniczona liniowo. Dodatkową zaletą jest czytelność takiej struktury, a przejście na reprezentację regułową nie jest utrudnieniem.

Jednak drzewa decyzyjne posiadają także ograniczenia, wśród których zalicza się złożoną strukturę przy rozbudowanych hipotezach, alternatywne warunki są cięższe do ukazania. Rozbudowa takiego drzewa o nowe przykłady i dane także nie należy do łatwego zadania. [8]

3.2. Algorytm ID3

Jednym z algorytmów operujących na drzewach decyzyjnych jest algorytm ID3, którego twórcą jest Ross Quinlan. Cechą wyróżniającą jest posiadanie przez niego wyboru atrybutów, na których następnie przeprowadzane są testy, które końcowo sprawiają, że drzewo staje się najbardziej efektywne oraz najprostsze w swojej budowie. Zwykle używa się go w uczeniu maszynowym, a także jako język przetwarzania naturalnego domen.

Algorytm opiera się na wybraniu pewnego klucza atrybutu, który stanie się pierwszym węzłem. Z niego wyprowadzone zostaną kolejne gałęzie. Ich ilość będzie ograniczona o wartość, którą może przyjąć wybrany atrybut. Gałęzie będą oznaczać wybory wartości z badanej cechy. Na końcach takich gałęzi tworzone będą nowe listy przykładów, których budowa uzależniona zostanie od atrybutów nadrzędnych mających wartość taką jaka jest ukazana w prowadzącej do niej gałęzi. Następnie dobiera się ponownie według opisanych wcześniej operacji atrybut, który stanie się kolejnym testowanym węzłem. W tym momencie może jednak wystąpić warunek stopu. Przy takiej zależności na końcu gałęzi zostanie wstawiony liść odpowiedzialny za kwalifikację przykładów, które w pozytywny sposób przeszły warunki do jednej kategorii atrybutu

decyzyjnego. Jeżeli na samym końcu rozpatrywanej gałęzi znajduje się nowy węzeł, to wyciąga się z niego wszystkie wartości, które mogą być przyjęte przez atrybut powiązany. W ostatnim kroku rozpatruje się odpowiednio atrybuty według kryteriów mających na celu wybranie kolejnego korzenia lub wstawienia go dalej. Chcąc przejść do praktyki należy wykorzystać kryterium maksymalnego przyrostu informacji. Żeby je wyznaczyć konieczne jest wyliczenie ilości informacji, a dokładnie entropii i entropii zbioru przykładów dobieranego ze względu na badany atrybut.

Entropia:

$$I(E) = -\sum_{i=1}^{k} \frac{|E_i|}{|E|} \cdot \log_2 \left(\frac{|E_i|}{|E|} \right)$$

Rysunek 3.2 – Entropia. Źródło: [9]

Gdzie:

k – Liczba wartości atrybutu decyzyjnego.

$|E_i|$ - Liczba przykładów ze zbioru uczącego mających i-tą wartość atrybutu decyzyjnego.

Entropia zbioru przykładów ze względu na analizowany atrybut:

$$I(E,a) = -\sum_{m=1}^{m=L} \frac{|E^{(m)}|}{|E|} \cdot I\left(E^{(m)}\right)$$

Rysunek 3.3 – Entropia zbioru przykładów ze względu na analizowany atrybut. Źródło: [9]

a – Analizowany atrybut.

L – Liczba wartości analizowanego atrybutu.

$E^{(m)}$ – Przykłady, dla których a-ty atrybut miał m-tą wartość.

$|E^{(m)}|$ - Liczba przykładów, dla których a-ty atrybut miał m-tą wartość.

Względny maksymalny przyrost informacji:

$$\triangle I(E,a) = I(E) - I(E,a)$$

Rysunek 3.4 – Względny maksymalny przyrost informacji. Źródło: [9]

Atrybut, dla którego względny maksymalny przyrost informacji będzie największy, zostanie wybrany jako pierwszy węzeł drzewa decyzyjnego. [9]

3.3. Szczegółowe przedstawienie techniki TDD

Pracę nad programem rozpoczynam od dokładniejszego zapoznania się z danymi wejściowymi i rozpoznania potrzeby przekonwertowania wartości 1 i 0 z typu liczb całkowitych na bezpieczniejszy typ logiczny. Konstruuję test:

```
[TestMethod]
0 references
public void UnitTest_TBoolean_False()
{
    string value = "0";
    bool expected = false;

    var testClass = new DrzewoDecyzyjne();

    bool actual = testClass.TBoolean(value);

    Assert.AreEqual(expected, actual);
}
```

Rysunek 3.5 – Pierwszy test metody TBoolean. Źródło: opracowanie własne

Ma on za zadanie sprawdzenie czy w przypadku wprowadzenia wartości wejściowej „0”, metoda poprawnie przyjmie wartość jako fałsz.

Następnie tworzę metodę która ten test ma zaliczyć:

```
public bool TBoolean(string war)
{
    return false;
}
```

Rysunek 3.6 – Kod zaliczający pierwszy test metody TBoolean. Źródło: opracowanie własne

Na tym etapie nie potrzebujemy nic więcej, niż prostą funkcję przyjmującą wartość a następnie zwracającą false. Test został pomyślnie zakończony.

Zakończyłem pierwszy cykl techniki TDD, przechodzę następnie do kolejnego testu:

```
[TestMethod]
0 references
public void UnitTest_TBoolean_True()
{
    string value = "1";
    bool expected = true;

    var testClass = new DrzewoDecyzyjne();

    bool actual = testClass.TBoolean(value);

    Assert.AreEqual(expected, actual);
}
```

Rysunek 3.7 – Drugi test metody TBoolean. Źródło: opracowanie własne

Ten test sprawdza czy funkcja zwraca wartość true, jeśli wartość wejściowa wynosi „1”. Test zostaje uruchomiony i okazuje się, że nie kończy się pomyślnie. Przechodzimy więc do pisania kodu, w tym przypadku modyfikacji klasy TBoolean:

```
public bool TBoolean(string war)
{
    switch (war)
    {
        case "1":
            return true;
        default:
            return false;
    }
}
```

Rysunek 3.8 – Kod zaliczający drugi test metody TBoolean. Źródło: opracowanie własne

Sprawdzamy czy obydwa dotychczasowe testy są wykonywane pomyślnie i kontynuujemy pracę przechodząc do kolejnego cyklu. Piszemy kolejny test:

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
0 references
public void UnitTest_TBoolean_Null()
{
    string value = null;

    var testClass = new DrzewoDecyzyjne();

    bool actual = testClass.TBoolean(value);
}
```

Rysunek 3.9 – Trzeci test metody TBoolean. Źródło: opracowanie własne

Stworzony test jest bardziej skomplikowany i ma na celu wprowadzenie jednoznaczności wprowadzanych danych. Przechodzimy do metody TBoolean i dołączamy wyjątek:

```
public bool TBoolean(string war)
{
    switch (war)
    {
        case "1":
            return true;
        case "0":
            return false;
        default:
            throw new ArgumentException();
    }
}
```

Rysunek 3.10 – Kod zaliczający trzeci test metody TBoolean. Źródło: opracowanie własne

Po tej zmianie wszystkie testy przechodzą pomyślnie.

Przechodzimy do testowania większej ilości przypadków:

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
0 references
public void UnitTest_TBoolean_Default()
{
    string value = "50";
    var testClass = new DrzewoDecyzyjne();
    bool actual = testClass.TBoolean(value);
}

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
0 references
public void UnitTest_TBoolean_Empty()
{
    string value = "";
    var testClass = new DrzewoDecyzyjne();
    bool actual = testClass.TBoolean(value);
}

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
0 references
public void UnitTest_TBoolean_Throw()
{
    string value = "5";
    var testClass = new DrzewoDecyzyjne();
    bool actual = testClass.TBoolean(value);
}
```

Rysunek 3.11 – Kolejne testy metody TBoolean. Źródło: opracowanie własne

Wszystkie testy są zaliczane, metoda TBoolean nie potrzebuje zmian.

3.4. Zastosowanie techniki TDD w całym programie

Na podstawie działań z poprzedniego rozdziału oraz opisu algorytmu ID3 kontynuowałem pracę nad programem, aż do zakończenia projektu.

Kilka przykładowych testów które wykorzystałem oraz kod który został napisany na ich podstawie:

```
[TestMethod]
0 references
public void UnitTest_FunkcjaLogarytmiczna_Expected()
{
    int pozytywI = 20;
    int caloscI = 10;

    double expected = -2;
    var testClass = new DrzewoDecyzyjne();

    double actual = testClass.FunkcjaLogarytmiczna(pozytywI, caloscI);

    Assert.AreEqual(expected, actual);
}

[TestMethod]
0 references
public void UnitTest_FunkcjaLogarytmiczna_Zero()
{
    int pozytywI = -10;
    int caloscI = 20;

    double expected = 0;
    var testClass = new DrzewoDecyzyjne();

    double actual = testClass.FunkcjaLogarytmiczna(pozytywI, caloscI);

    Assert.AreEqual(expected, actual);
}
```

Rysunek 3.12 – Przykładowe testy metody FunkcjaLogarytmiczna. Źródło: opracowanie własne

Wgląd na metodę FunkcjaLogarytmiczna w programie:

```
10 references | 10/10 passing
public double FunkcjaLogarytmiczna(int pozytywI, int caloscI)
{
    double pozI = pozytywI;
    double calI = caloscI;
    double podzielone = pozI / calI;
    double wynik = 0;
    double podzielone_log;
    if (podzielone > 0)
    {
        podzielone_log = Math.Log(podzielone, 2);
        wynik = (podzielone * (-1));
        wynik *= podzielone_log;
        return wynik;
    }
    else
        return wynik;
}
```

Rysunek 3.13 – Metoda FunkcjaLogarytmiczna. Źródło: opracowanie własne

Przykładowe testy dla klasy KoncowaEntropia:

```
[TestMethod]
0 references
public void UnitTest_KoncowaEntropia_Expected()
{
    double entropiaKonkluzji = 20;
    double entropiaPrzeslanki = 10;

    double expected = 10;
    var testClass = new DrzewoDecyzyjne();

    double actual = testClass.KoncowaEntropia(entropiaKonkluzji, entropiaPrzeslanki);

    Assert.AreEqual(expected, actual);
}

[TestMethod]
0 references
public void UnitTest_KoncowaEntropia_NegativeNumber()
{
    double entropiaKonkluzji = 10;
    double entropiaPrzeslanki = 20;

    double expected = -10;
    var testClass = new DrzewoDecyzyjne();

    double actual = testClass.KoncowaEntropia(entropiaKonkluzji, entropiaPrzeslanki);

    Assert.AreEqual(expected, actual);
}
```

Rysunek 3.14 – Przykładowe testy metody KoncowaEntropia. Źródło: opracowanie własne

4. Porównanie dwóch technik tworzenia oprogramowania

W tym rozdziale zostaną opisane wady i zalety pisania kodu wykorzystując technikę *"Test-Last"*, wykorzystywanej na przykład w modelu kaskadowym. Zostaną również przedstawione zalety i wady techniki Test-Driven Development. Bezpośrednie zestawienie tych dwóch technik pozwoli na lepsze zrozumienie w jakich sytuacjach jedna technika zapewnia lepsze wyniki od drugiej.

4.1. Zalety i wady modelu kaskadowego

Zaletami tworzenia oprogramowania z wykorzystaniem modelu kaskadowego jest to, że pozwala on na wydzielenie modułów oraz kontrolę. Rygorystyczny harmonogram może zostać skonstruowany dla każdego etapu i dzięki temu produkt może się posuwać przez kolejne fazy procesu rozwoju.

Zaletami modelu kaskadowego są:

- Model jest prosty i łatwy w użytkowaniu i zarządzaniu
- Fazy są wykonywane i kończone jedna po drugiej
- Działa dobrze w mniejszych projektach, z jasnymi wymaganiami
- Proste do zidentyfikowania kamienie milowe.

Główną wadą modelu kaskadowego jest utrudniona możliwość rewizji kodu. Gdy aplikacja znajdzie się w fazie testowania, bardzo trudno wrócić się i naprawić problem który mógł być źle udokumentowany, bądź pominięty w fazie analizy.

Niektóre wady modelu kaskadowego:

- Brak działającego programu, aż do późnych momentów cyklu życia oprogramowania
- Duże ilości ryzyka i niepewności
- Nie nadaje się do skomplikowanych, bądź zorientowanych obiektowo projektów
- Trudno jest kontrolować postęp wewnątrz poszczególnych etapów
- Nie może sprostać zmieniającym się wymaganiom. [10]

4.2. Zalety i wady Test-Driven Development

Główną wadą techniki Test-Driven Development jest brak możliwości wykorzystania jej do produkcji oprogramowania w środowiskach, które nie pozwalają na ciągłe kompilowanie kodu i testów. Mogą to być środowiska podlegające pakietowemu kompilowaniu, bądź kod może być wykonywany tylko na zewnętrznym urządzeniu, na które trzeba za każdym razem zgrywać kod.

Wśród zalet TDD można wymienić natomiast:

- Elastyczność – dzięki pokryciu kodu testami, programista może dokonywać małych zmian w kodzie i nie obawiać się, że wprowadził błąd do projektu, ponieważ może od razu wykonać wszystkie testy. Daje to bardzo duże możliwości w rozbudowie funkcjonalności kodu.
- Dokumentacja – programista dowiaduje się bardzo dużo sprawdzając testy jednostkowe. Dokładnie napisane testy jednoznacznie wskazują, jaki cel miał programista pisząc dany kod. Prawie całkowite pokrycie kodu produkcyjnego testami, ułatwia śledzenie wykonywanego kodu.
- Zmniejszenie czasu debugowania – dzięki ciągłemu testowaniu pisanego kodu, wszystkie błędy zostają poprawiane w trakcie tworzenia oprogramowania. Oprogramowanie w fazie debugowania jest wolne od większości problemów. Czas poświęcony na pisanie testów w trakcie tworzenia kodu jest więcej niż nadrabiany dzięki zdecydowanie krótszej fazie debugowania, podczas której nie trzeba szukać powodu wystąpienia błędu.
- Lepszy projekt – W projektowaniu obiektowym bardzo ważne jest rozłączność pomiędzy poszczególnymi modułami, zwiększa to jakość projektu i możliwości rozwoju. Dzięki pisaniu testów do każdej funkcji programista jest zmuszany do pisania kodu odłączonego od reszty modułów, ponieważ nie może testować modułu zależnego od innych funkcjonalności. [4]

4.3. Porównanie

W tym rozdziale przybliżone zostały nam dwie techniki programowania, znacząco różniące się kolejnością wykonywanych czynności. Wykorzystując model kaskadowy przeprowadzamy testy bardzo późno w cyklu życia oprogramowania. Z tego powodu wszystkie problemy i błędy odnalezione są bardzo kosztowne, ponieważ odszukanie metod które mogą odpowiadać za ten błąd jest żmudne i czasochłonne. Natomiast użycie w takiej sytuacji techniki Test-Driven Development, o ile wydłuża czas tworzenia kodu, o tyle wszystkie błędy są natychmiast odnajdywane i naprawiane, co skraca czas debugowania.

Test-Driven Development często jest niestety nie możliwy do wykorzystania z powodu środowiska w jakim pracuje programista. Może się okazać, że tworzenie testów i kompilowanie kodu w krótkich, kilku minutowych cyklach jest niemożliwe lub bardzo uciążliwe z różnych powodów, np. wdrażania funkcjonalności w pracującym systemie webowym. W takich momentach metoda kaskadowa sprawdza się znakomicie, ponieważ program jest kompilowany i uruchamiany bardzo późno w cyklu życia oprogramowania.

Techniki programowania sterowanego testami cechują się również elastycznością oraz łatwością późniejszego rozwoju już gotowego oprogramowania, co nie zawsze jest proste w oprogramowaniu stworzonym metodą kaskadową. Tworzenie oprogramowania techniką TDD również uczy dewelopera dobrych zwyczajów programistycznych, szczególnie jeśli pracuje on w językach zorientowanych obiektowo.

5. Zakończenie

W powyższej pracy została zaprezentowana technika tworzenia oprogramowania, która zalicza się do metodyk zwinnych. Z przebiegu całej analizy można wnioskować, że zastosowana w odpowiedni sposób, znacząco wpływa na jakość wytwarzanego oprogramowania. Pewnym jest, że nie każdy problem jesteśmy w stanie rozwiązać za pomocą testów, jednak dzięki nim, zdecydowaną większość problemów jesteśmy w stanie zlokalizować. Jest wiele zalet płynących ze stosowania techniki Test-Driven Development, takich jak prawie całkowite pokrycie kodu produkcyjnego testami, elastyczność związana z pewnością zachowania poprawności działania pomiędzy funkcjonalnościami programu, czy skrócenie czasu debugowania. Ale chyba najważniejszą według mnie zaletą jest walor edukacyjny. Dzięki zastosowaniu techniki programowania sterowanego testami nie tylko uczymy się pisać czysty kod, ale również pomaga nam w nauce nowego języka poprzez zmuszanie nas do wykorzystywania mało skomplikowanych metod, a później stopniowego budowania na nich.

Jednak TDD ma też swoje wady, w małych nieskomplikowanych projektach dodatkowa praca nad testami w trakcie pisania kodu może okazać się stratą czasu, ponieważ program napisany metodą tradycyjną też może nie wymagać długiego debuggingu. Jak również możliwość wystąpienia środowiska, które w żaden sposób nie sprzyja przestrzeganiu trzech praw TDD, i wymusza dłuższy okres pomiędzy kompilacjami kodu.

W dzisiejszych czasach coraz więcej deweloperów ugina się pod presją czasu, i zaniedbuje lub czasami całkowicie porzuca testowanie swojego kodu. Częstsze wykorzystanie techniki programowania sterowanego testami może tylko pozytywnie wpłynąć, zarówno na czas tworzenia oprogramowania, jak również na jego jakość, oraz przejrzystość.

6. Bibliografia

[1] „TDD. Sztuka tworzenia dobrego kodu”, Kent Beck, Wydawnictwo Helion SA 2014

[2] „Czysty kod. Podręcznik dobrego programisty”, Robert C. Martin, Wydawnictwo Helion SA 2014

[3] „Testowanie oprogramowania. Podręcznik dla początkujących”, Rafał Pawlak, Wydawnictwo Helion SA 2014

[4] „Professionalism and Test-Driven Development”, Robert C. Martin, Object Mentor, IEEE Software, maj – czerwiec 2007 (Vol. 24, No. 3), s. 32 – 36

[5] https://medium.com/hackernoon/introduction-to-test-driven-development-tdd-61a13bc92d92 (dostęp: 12.10.2020)

[6] https://marsner.com/blog/why-test-driven-development-tdd/ (dostęp: 12.10.2020)

[7] https://blog.onwelo.pl/czym-jest-technika-tdd-i-jak-wyglada-jej-cykl-zycia/ (dostęp: 13.11.2020)

[8] https://www.ii.pwr.edu.pl/~kwasnicka/tekstystudenckie/apw/decyzyjne.htm (dostęp: 14.11.2020)

[9] http://kajtsweb.webd.pl/pwr/kajt/weka/uczenie_maszyn_projekt.pdf?fbclid=IwAR1NtGDwS0QEEDhxGPjoDwB3s28xW2MsoQ_SSvxE4ai3SI5xC15zc_7w4I8 (dostęp: 16.11.2020)

[10] https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm (dostęp 29.11.2020)

7. Spis Rysunków