AKADEMIA GÓRNICZO-HUTNICZA

IM. STANISŁAWA STASZICA W KRAKOWIE



Wydział Inżynierii Metali i Informatyki Przemysłowej



PROJEKT INŻYNIERSKI

pt.

„Test-Driven Development jako zwinna technika programowania”

Imię i nazwisko dyplomanta: **Tomasz Piętka**

Kierunek studiów: **Informatyka Stosowana**

Nr albumu: **255391**

Promotor: dr inż. Dariusz Jędzejczyk

Recenzent: …………………………………

Podpis dyplomanta: Podpis promotora:

Kraków 2015

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszy projekt inżynierski wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Kraków, dnia ………….… Podpis dyplomanta ……………………

Spis treści

[1. Wstęp 4](#_Toc422323748)

[2. Część teoretyczna 6](#_Toc422323749)

[2.1. Programowanie zwinne („Agile”) 6](#_Toc422323750)

[2.2. Extreme programming 8](#_Toc422323751)

[2.3. Testy jednostkowe 8](#_Toc422323752)

[2.4. Testy integracyjne 9](#_Toc422323753)

[2.5. Test-Driven development 10](#_Toc422323754)

[2.6. Dane statystyczne dotyczące TDD oraz zalety i wady techniki 12](#_Toc422323755)

[3. Badania własne 14](#_Toc422323756)

[3.1. Charakterystyka aplikacji Planning Poker oraz omówienie jej składowych 14](#_Toc422323757)

[3.1.1. Technologia 16](#_Toc422323758)

[3.1.1.1. C#.Net 16](#_Toc422323759)

[3.1.1.2. Nunit 16](#_Toc422323760)

[3.1.1.3. Fluent Assertions 16](#_Toc422323761)

[3.1.1.4. Linq 16](#_Toc422323762)

[3.1.1.5. WPF – Windows Presentation Foundation 16](#_Toc422323763)

[3.1.2. Struktura testów 16](#_Toc422323764)

[3.1.2.1. Testy jednostkowe 16](#_Toc422323765)

[3.1.2.2. Testy integracyjne 20](#_Toc422323766)

[3.1.3. Struktura aplikacji 21](#_Toc422323767)

[3.1.3.1. Klasa Player 21](#_Toc422323768)

[3.1.3.2. Klasa Story 21](#_Toc422323769)

[3.1.3.3. Klasa PlanningPoker 22](#_Toc422323770)

[3.1.3.4. Klasa Estimation 22](#_Toc422323771)

[3.1.4. Graficzny interfejs użytkownika 22](#_Toc422323772)

[3.1.5. GIT 25](#_Toc422323773)

[3.2. Przykład tworzenia aplikacji 26](#_Toc422323774)

[3.2.1. Definiowanie testu krok po kroku 26](#_Toc422323775)

[3.2.2. Refactoring 28](#_Toc422323776)

[3.2.3. Wyjątki 30](#_Toc422323777)

[4. Wnioski 32](#_Toc422323778)

[5. Bibliografia 33](#_Toc422323779)

# Wstęp

Programowanie zwinne (ang. Agile software development), czyli grupa metodyk wytwarzania oprogramowania oparta na programowaniu iteracyjno-przyrostowym prężnie rozwija się od kilkudziesięciu lat. Wielokrotnie zostało udowodnione, że projekty, w których stosuje się metodyki zwinne odnoszą większe sukcesy od projektów tradycyjnych.

Pojęcie zwinnego oprogramowania zostało zaprezentowane w tak zwanym Agile Manifesto w 2001 roku. Sygnatariusze manifestu przedkładali ludzi i interakcje ponad procesy i narzędzia, działające oprogramowanie ponad obszerną dokumentację, współpracę z klientem ponad formalne ustalenia oraz reagowanie na zmiany ponad podążanie za planem[[1]](#endnote-1). Oznacza to, iż w przeciwieństwie do dotychczasowego, tradycyjnego podejścia występuje większy nacisk na działające oprogramowanie i satysfakcję klienta.

Manifest z 2001 roku był wypadkową wielu ruchów, które miały swoje początki w latach 80 i 90 XX wieku. Ken Schwaber już w 1996 roku zdefiniował i sformalizował najpopularniejszą obecnie metodykę o nazwie Scrum.

Wymieniając kolejne metodyki należy wspomnieć o Kanban, Lean Software development, Feature Driven Development czy najważniejszym w kontekście tego opracowania Programowaniu Ekstremalnym (ang Extreme Programming).

Programowanie Ekstremalne to paradygmat zdefiniowany przez Kenta Becka w 1999 w książce Extreme Programming Explained*[[2]](#endnote-2).* Extreme programing to tak naprawdę zbiór technik, jedną z nich jest Test-Driven Developement (w skrócie TDD).

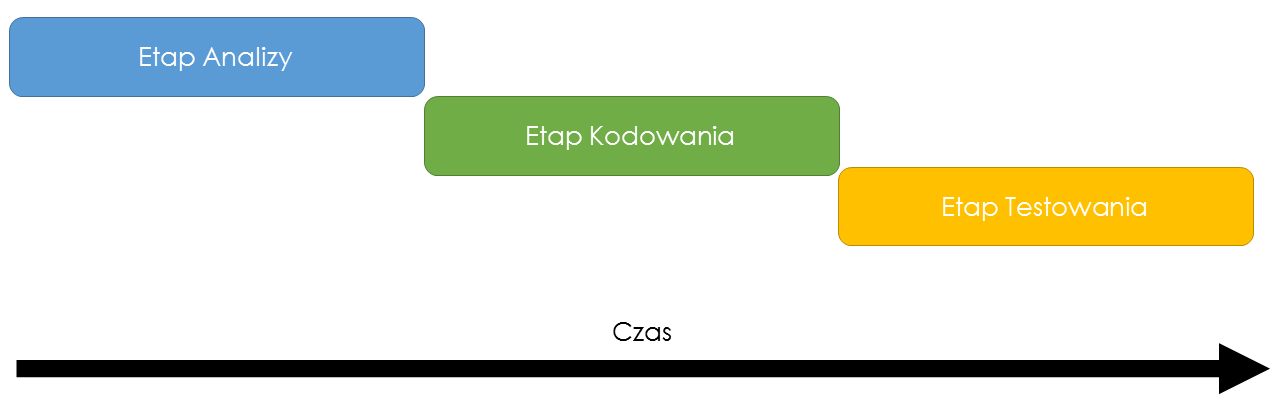
Mimo tego, iż TDD jest znane od wielu lat, nie jest ono używane w wystarczającym stopniu w obecnie prowadzonych projektach. Głównym powodem jest to, iż założone jest, że programiści radykalnie zmienią swój sposób pracy oraz jej filozofię – tak zwane Kaikaku (jap radykalna zmiana). Pisanie testów przed napisaniem działającego kodu jest najczęściej trudne do zrozumienia oraz zastosowania. Kaikaku to koncepcja przeciwna Kaizenowi (który zakładał ciągłą zmianę w procesach produkcyjnych), skupiająca się na małych, radykalnych, inkrementalnych zmianach. Kaikaku idealnie opisuje to, co musi zajść w sposobie myślenia programisty używającego tradycyjne metody programowania, aby mógł korzystać ze wszystkiego co daje poprawne stosowanie TDD.

Jako entuzjasta i praktyk zwinnych metodyk oprogramowania, w tej pracy postaram się udowodnić, iż programowanie w technice TDD jest dużo bardziej wydajne niż programowanie tradycyjne. TDD skupia się także na jakości oprogramowania. Aplikacja, która jest częścią tego projektu umożliwia przeprowadzanie sesji tzw Planning Poker – kolejnej techniki programowania zwinnego polegającej na ułatwieniu procesu estymacji pracy.[[3]](#endnote-3)

# Część teoretyczna

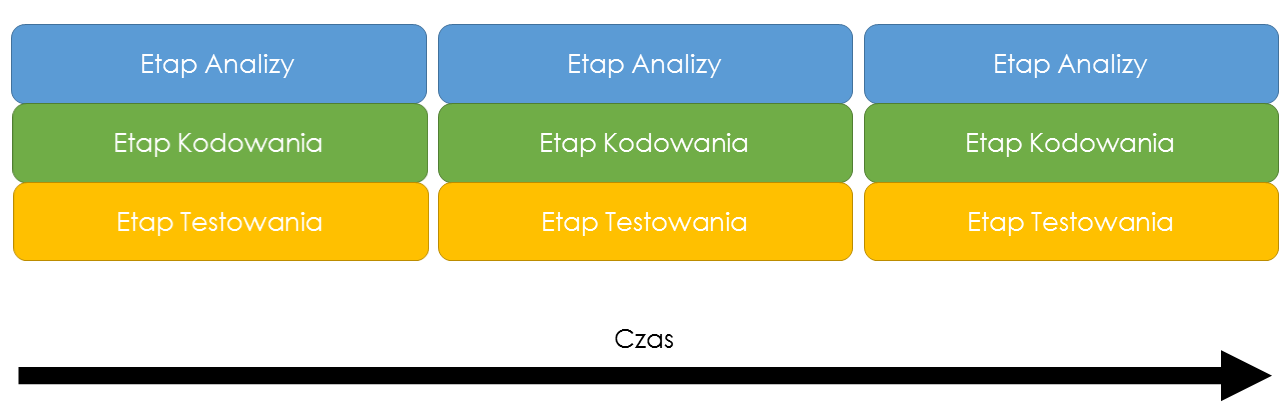
## Programowanie zwinne („Agile”)

Agile to sposób programowania, który przypomina nam, że mimo tego, że to komputery uruchamiają kod to ludzie go tworzą i utrzymują[[4]](#endnote-4). Aby zrozumieć to co dzieje się od kilkunastu lat należy cofnąć się do połowy XX wieku. Inżynieria oprogramowania jako dziedzina biznesu powstała około 40 lat temu. Wtedy urodziło się tradycyjne podejście do projektów, a z racji tego, że programowanie do dziedzina bardzo specyficzna, inżynierowie byli zmuszeni zapożyczać techniki z istniejących dziedzin biznesu. Stąd zapożyczenia między innymi z budownictwa widoczne do dzisiaj – programiści i inni ludzie związani z wytwarzaniem oprogramowania nazywają siebie inżynierami, tworzą „buildy” (budynki) itp. Zapożyczone zostały również techniki prowadzenia projektów. Do niedawna dominowało przeświadczenie iż każdą fazę tworzenia oprogramowania należy wykonywać po sobie, sekwencyjnie. Tradycyjne podejście zostało przedstawione na poniższym rysunku.



Rysunek 1. Tradycyjne podejście do projektów programistycznych, tzw „wodospad” (ang waterfall), opracowanie własne

Takie podejście nazywamy obecnie wodospadem (ang waterfall). Każdy z etapów następuje po sobie, jest od siebie zależny. Głównym problemem takiego podejścia jest problem sprzężenia zwrotnego (ang feedback loop). Każdy etap wymaga osobnego zespołu. Zespoły w jednym momencie czasu pracują nad różnymi projektami. Gdy problem z wymaganiami – re definicja wymagań, niedostateczne ich zrozumienie itp – zostaje zidentyfikowany na jednym z ostatnich etapów (najczęściej testy akceptacyjne) cały proces musi odbyć się jeszcze raz. Tego typu podejście przez lata nastręczało wiele problemów – stąd rozwój metodyk zwinnych, których głównym zadaniem jest skrócenie czasu reakcji w sprzężeniu zwrotnym. Poniższy rysunek obrazuje iteracyjne podejście do prowadzenia projektu.



Rysunek 2. Zwinne podejście do projektów programistycznych, tzw „Agile”, opracowanie własne

Tutaj etapy projektu prowadzone są równocześnie, w krótkich iteracjach (najczęściej 1-4 tygodni). Metodyki Agile skupiają się najczęściej na dostarczeniu małej części działającego oprogramowania (z metodyki Scrum, „przyrost” - ang product increment[[5]](#endnote-5)). Dzięki temu reakcja na sprzężenie zwrotne jest krótsza – zespół może modyfikować założenia w trakcie pisania projektu. Zespoły są odpowiedzialne za całość działającego oprogramowania (a nie za poszczególne etapy) co powoduje występowanie lepszego zrozumienia tego co aktualnie się dzieje[[6]](#endnote-6). Wspomniany wcześniej Manifest Agile wygląda następująco:

|  |  |
| --- | --- |
| Manifest Zwinnego Tworzenia Oprogramowania | |
| Wytwarzając oprogramowanie i pomagając innym w tym zakresie,  odkrywamy lepsze sposoby wykonywania tej pracy.  W wyniku tych doświadczeń przedkładamy: | |
| **Ludzi i interakcje** ponad procesy i narzędzia. | |
| **Działające oprogramowanie** ponad obszerną dokumentację. | |
| **Współpracę z klientem** ponad formalne ustalenia. | |
| **Reagowanie na zmiany** ponad podążanie za planem. | |
| Doceniamy to, co wymieniono po prawej stronie,  jednak bardziej cenimy to, co po lewej. | |
| Kent Beck  Mike Beedle  Arie van Bennekum  Alistair Cockburn  Ward Cunningham  Martin Fowler  James Grenning  Jim Highsmith  Andrew Hunt | Ron Jeffries  Jon Kern  Brian Marick  Robert C. Martin  Steve Mellor  Ken Schwaber  Jeff Sutherland  Dave Thomas |

Tabela 1. Manifest Zwinnego Tworzenia Oprogramowania, http://agilemanifesto.org/iso/pl/

## Extreme programming

Programowanie ekstremalne (ang. Extreme Programming) to zbiór praktyk programowania, które skupiają się na ulepszaniu jakości oprogramowania i zarządzania reakcją na zmieniające się wymagania klientów[[7]](#endnote-7). Zalecenia XP:

- **Iteracyjność** – program tworzy się w iteracjach i planuje się tylko kolejną iterację.

- **Nie projektować z góry** – architektura systemu powinna być tworzona na bieżąco. Nie można z góry przewidzieć jaka architektura będzie najlepsza przed rozpoczęciem projektu.

- **Testy jednostkowe –** należy je pisać przed pisaniem kodu. Jest to de facto definicja techniki Test-Driven Development. Niniejsze opracowanie w dalszej części skupia się na tej technice.

- **Ciągłe modyfikacje architektury** – architektura rozwija się wraz z powstawaniem oprogoramowania.

- **Programowanie parami** – kod należy pisać parami – jedna osoba pracuje przy klawiaturze, druga jest obserwatorem.

- **Stały kontakt z klientem** – klient musi wiedzieć co dzieje się w projekcie oraz przekazywać informacje o zmieniających się wymaganiach.[[8]](#endnote-8)

## Testy jednostkowe

Testy jednostkowe jako pierwsze zostały zdefiniowane w latach 70 XX wieku i przedstawione jako koncepcja wraz z językiem Smalltalk przez Kenta Becka. Testy jednostkowe w programowaniu obiektowym to jeden z najlepszych sposobów dzięki któremu programista może poprawić jakość swojego kodu zyskując dogłębne zrozumienie wymagań funkcjonalnych tworzonej klasy lubi modelu. Angielskojęzyczna Wikipedia definiuje unit test jako (tłumaczenie własne):

**Definicja 1:** Test jednostkowy to kawałek kodu (najczęściej metoda) który wywołuje inny kawałek kodu i sprawdza jego poprawność zgodnie z pewnymi założeniami. Jeśli założenia okazują się błędne, test jednostkowy zawodzi (fails).

Bardziej dokładną definicję proponuje Roy Osherove w książce The Art of Unit Testing. Aby ją zrozumieć, należy wcześniej zdefiniować jednostkę pracy (ang unit of work). Za Osherove (tłumaczenie własne):

**Definicja 2**: Jednostka pracy jest sumą akcji, które mają miejsce przed wywołaniem metody publicznej w systemie oraz pojedynczy, zauważalny rezultat końcowy testu tego systemu. Zauważalny rezultat końcowy może być zaobserwowany bez sprawdzania stanu systemu i tylko poprzez publiczne API i publiczne zachowania. Rezultat końcowy to jedno z poniższych:

- Wywołana publiczna metoda która zwraca wartość (funkcja nie jest typu void)

- Istnieje zauważalna zmiana stanu lub zachowania systemu przed i po wywołaniu, która może być określona bez sprawdzania stanu prywatnego (na przykład system może zalogować wcześniej nie istniejącego użytkownika (...))

- Istnieje wywołanie do innego systemu nad którym test nie ma kontroli oraz ten inny system nie zwraca żadnej wartości, lub wartość zwracana z tego systemu jest ignorowana.

Idea jednostki pracy mówi, że jednostka to pojedyncza, atomiczna metoda i kilka klas i funkcji, które pomagają osiągnąć jej cel. Dzięki zrozumieniu powyższego konceptu można zastosować następującą definicję testu jednostkowego:

**Definicja 3**: Test jednostkowy to kawałek kodu (najczęściej metoda) który wywołuje jednostkę pracy i sprawdza jeden specyficzny rezultat końcowy tej jednostki pracy. Jeśli założenie rezultatu końcowego okaże się złe, test jednostkowy zawodzi (fails). Zakres testu jednostkowego ma tak małą rozpiętość jak metoda lub tak dużą jak wiele klas[[9]](#endnote-9).

## Testy integracyjne

Prostą definicję testu integracyjnego proponuje Microsoft (tłumaczenie własne):

**Definicja 4**: Test integracyjny testuje różne komponenty rozwiązania, które składają się na jedną, większą całość[[10]](#endnote-10).

Bardziej kompleksowa definicja opierająca się na wcześniej zdefiniowanej jednostce pracy to, za Osherove (tłumaczenie własne):

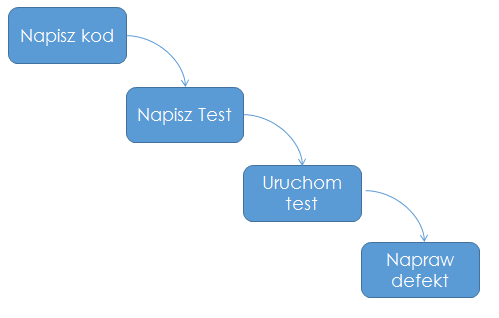
**Definicja 5**: Testy integracyjne testują jednostkę pracy bez posiadania pełnej kontroli nad nią i używają jedną lub więcej jej zależności, takich jak czas, sieć, baza danych, wątki itp[[11]](#endnote-11).

Czyli jak łatwo zauważyć, testy integracyjne dopełniają testy jednostkowe a celem obu jest stworzenie oprogramowania jak najlepszej jakości.

## Test-Driven development

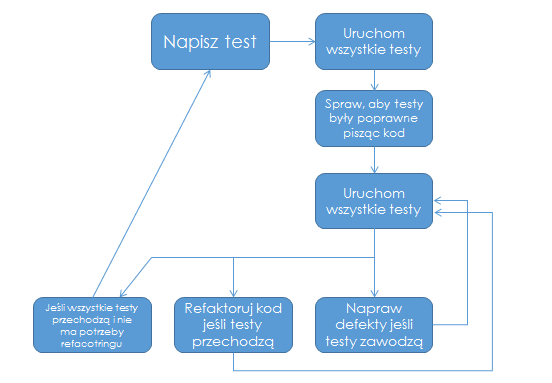
Do tej pory w opracowaniu zdefiniowane zostały rodzaje testów, jednak pozostaje pytanie kiedy je napisać. Głównym założeniem techniki Test-Driven Development (w skrócie TDD) jest pisanie testów jednostkowych i integracyjnych przednapisaniem kodu aplikacji.

Poniższe rysunki przedstawiają różnicę pomiędzy tradycyjnym podejściem do pisania testów jednostkowych a podejściem TDD.



Rysunek 3. Tradycyjne podejście do pisania testów, opracowanie własne za Roy Osherove

Jak widać na powyższym rysunku, w tradycyjnym podejściu najpierw powstaje aplikacja, metoda lub inna część kodu a później test. Głównym założeniem techniki Test-Driven Development (w skrócie TDD) jest pisanie testów jednostkowych i integracyjnych przed napisaniem kodu aplikacji.



Rysunek 4. Podejście TDD, opracowanie własne za Roy Osherove

TDD to przede wszystkim technika programowania – testy są tutaj ważne, jednak najważniejszą rzeczą pozostaje działające oprogramowanie. Mamy do czynienia ze swoistą mantrą TDD, którą można opisać w 3 prostych krokach:

- Krok Czerwony – napisz test, który zawodzi ponieważ nie ma jeszcze działającego kodu

- Krok Zielony – napisz kod, tak aby wcześniej napisany test się powiódł.

- Krok Refactor – gdy test przechodzi, można pisać kolejny test lub zmienić implementacje kodu aby był bardziej czytelny[[12]](#endnote-12).

Należy pamiętać, że w praktyce uruchamia się testy wielokrotnie. Wymaga to dyscypliny.

Przeanalizujmy jak wygląda pisanie protego testu i implementacji kodu w technice TDD. Przypuśćmy, że implementujemy prosty kalkulator. Zaczniemy więc od napisania testu sprawdzającego poprawność operacji dodawania dwóch liczb typu integer.

**Przykład kodu 1:**

[Test]

public void Dodaj\_DodajeDwieLiczby\_Obliczone()

{

//Arrange

    var calc = new Kalkulator();

//Act

    int suma = calc.Dodaj(2, 2);

//Assert

    Assert.AreEqual(4, suma);

}

Analizując powyższy kod od początku widzimy iż:

- Nasz kod jest metodą nie zwracającą żadnej wartości.

- Nazwa metody jest dodana w konwencji: Nazwa testowanej metody, wykonana akcja, efekt końcowy

- W sekcji oznaczonej Arrange (zaaranżuj) tworzy się nową instancję klasy Kalkulator.

- W sekcji oznaczonej Act (działaj) deklarujemy zmienną suma i wywołujemy metodę Dodaj z argumentami 2, 2. Metoda powinna dodać te dwie liczby do siebie.

- W sekcji oznaczonej Assert (zapewnij) sprawdzamy, czy wynik dodawania to 4.

Oczywiście w tym momencie gdy uruchomimy ten test, test się nie powiedzie. Nie ma jeszcze klasy kalkulator oraz jej metod. Jest to etap czerwony (ponieważ w większości aplikacji do uruchamiania testów - test będzie zaznaczony na czerwono). Kolejnym krokiem jest napisanie logiki klasy Kalkulator tak, aby test się powiódł

**Przykład kodu 2:**

public class Kalkulator

{

    public int Dodaj(int a, int b)

    {

        return a + b

    }

}

Po napisaniu powyższej klasy test napisany wcześniej się powiedzie i programista może skupić się na pisaniu kolejnego testu lub refaktoryzacji kodu.[[13]](#endnote-13)

## Dane statystyczne dotyczące TDD oraz zalety i wady techniki

Dwaj naukowcy z Uniwersytetu w Północnej Karolinie, USA – Bobie Gorge i Laurie Williams - przeprowadzili badania nad skutecznością techniki TDD. Eksperyment to 24 programistów, którzy mieli za zadanie napisanie prostej aplikacji w języku Java pracując zgodnie z zasadami Programowania Parami (ang Pair Programming, technika opisana wcześniej w sekcji dotyczącej Programowania Ekstremalnego). Połowa programistów (6 par) pracowała zgodnie z tradycyjnymi metodami programowania (najpierw kod, później testy jednostkowe), a druga połowa zgodnie z TDD.

Efektywność techniki TDD była analizowana pod kątem:

- czasu jaki programiści potrzebowali aby napisać aplikację;

- rezultat testów, które miały potwierdzić poprawność napisanych funkcjonalności – przed eksperymentem utworzona 20 przypadków testowych, które miały sprawdzić działającą aplikację;

- pokrycie kodu testami.

Zostały także przeprowadzone ankiety wśród uczestników eksperymentu dotyczące niemierzalnych aspektów prowadzenia projektu takich jak produktywność, efektywność pracy i trudności adaptacji techniki.

Z eksperymentu wynika, iż kod pisany przez programistów używających TDD pokrywa średnio o 18% więcej przypadków testowych niż kod pisany tradycyjnymi metodami.

Jeśli chodzi o pokrycie kodu testami standard w całym biznesie to około 80-90%, idealnie 100%. Grupa programistów pisząca w TDD osiągnęła następujące współczynniki, które są zdecydowanie wyższe niż standard na rynku:

- 98% pokrytych metod w kodzie;

- 92% pokrytych deklaracji w kodzie;

- 97% pokrycia brancha (ang branch coverage).

Jeśli chodzi o czas tworzenia aplikacji, TDD w eksperymencie wypada gorzej od metody tradycyjnych o 16%, czyli programiści piszący w TDD pracowali średnio o 16% dłużej niż ich koledzy pracującymi metodami tradycyjnymi.

Jeśli chodzi o czas pracy, należy pamiętać, iż badana była prosta aplikacja. Gdy stosujemy TDD w większych projektach, ogólnie rzecz biorąc czas powstawania aplikacji powinien być krótszy niż w przypadku metod tradycyjnych. Wiąże się to z lepszą architekturą, którą osiągamy w TDD.

Jeśli chodzi o wyniki ankiet, większość programistów uznała, że technika TDD poprawia zrozumienie wymagań aplikacji (87,5%) oraz zmniejsza wysiłek debuggowania aplikacji (95,8%). Gorzej jest z czasem – tylko połowa programistów uznała, iż TDD pozytywnie wpływa na czas powstawania aplikacji.

Jeśli chodzi o efektywność, 92% programistów uznało, iż TDD pozytywnie wpływa na jakość pisanego kodu, 79% twierdziło, że TDD pozytywnie wpływa na polepszenie architektury kodu.

56% programistów uznało, że zaadoptowanie techniki TDD było trudne, tylko 23% programistów twierdziło, że brak początkowej fazy projektowania w TDD był przeszkodą.

Podsumowując, TDD:

- Poprawia jakość kodu;

- Poprawia efektywność pracy;

- Jest trudne do wdrożenia;[[14]](#endnote-14)

# Badania własne

## Charakterystyka aplikacji Planning Poker oraz omówienie jej składowych

Wszystkie metody, klasy i testy i inne elementy składowe kodu są napisane w języku angielskim. Graficzny interfejs użytkownika oraz wszystko co jest widoczne dla użytkownika końcowego – np treści wiadomości o błędach – są napisane w języku polskim.

Aplikacja Planning Poker w założeniu ma być używana w części planowania pracy nad projektem zgodnym z filozofią Agile – estymacji.

W sesji Planning Poker biorą udział wszyscy członkowie zespołu, omawia się jednostki pracy zwane często historyjkami użytkownika (ang user stories). Celem sesji jest nadanie historyjkom wartości estymacji w jednostkach zwanych „story point” – czyli umowną jednostką określającą wielkość danej historyjki.

Aplikacja tworzona na potrzeby tego opracowania:

- pozwala na przeprowadzenie gry Planning Poker z minimum 3 graczami

- umożliwia dodanie obiektów historyjki w tej grze, które wymagają estymacji

- umożliwia każdemu z graczy oddanie głosu na każdą z historyjek. Wartości głosów są zgodne z dostępnymi na rynku kartami do gry w planning poker a ich wartości oparte są na ciągu Fibonacciego. Dokładny opis wartości znajduje się w opisie klasy Estimation poniżej

- umożliwia każdemu z graczy edycje oddanego głosu

- udostępnia 3 tryby rozgrywki dzięki którym ustalana jest ostateczna szacowana wielkość historyjki:

- średnia – obliczana jest średnia z oddanych głosów

- wszystkie równe – wszyscy gracze muszą zgodzić się co do wartości historyjki

- usuń skrajności, średnia – usuwa najniższy i najwyższy głos i oblicza średnią z reszty oddanych głosów

- umożliwia każdemu z graczy zagranie tzw karty specjalnej:

- przerwa – powinna nastąpić przerwa w grze

- za duże – historyjka musi zostać podzielona na mniejsze, gdyż według jednego z graczy jest zbyt duża do oszacowania

- ? – historyjka musi zostać doprecyzowana, nie jest możliwe oszacowanie jej wielkości

- gdy którykolwiek z graczy zagra jedną z kart specjalnych – historyjka nie będzie miała wartości estymacji.

- każdy z graczy musi zagrać jedną z kart, która nie jest kartą specjalną, aby wielkość historyjka mogła być oszacowana w programie

### Technologia

### C#.Net

Aplikacja została napisana w języku C#.Net za pomocą zintegrowanego środowiska programistycznego Visual Studio Community 2013 firmy Microsoft. W pisaniu kodu przydatny okazał się dodatek ReSharper w wersji 9.1 firmy JetBrains. Oprogramowanie jest udostępniane studentom w celach akademickich za darmo.

### Nunit

Nunit to framework użyty do uruchamiania testów w aplikacji udostępniany na licencji Open Source.

### Fluent Assertions

Fluent assertions to rozszerzenie frameworku Microsoft .Net umożliwiające bardziej naturalne asercje. Rozszerzenie zostało stworzone z myślą o TDD i jest udostępniane na zasadzie licencji Open Source.

### Linq

Linq to część technologii [Microsoft](http://pl.wikipedia.org/wiki/Microsoft) [.NET](http://pl.wikipedia.org/wiki/.NET_Framework) umożliwiająca zadawanie pytań na obiektach. Składnia języka LINQ jest prosta i przypomina SQL.

### WPF – Windows Presentation Foundation

To silnik graficzny i API służące do tworzenia graficznego interfejsu użytkownika.

### Struktura testów

W trakcie pisania aplikacji powstało 39 testów, z czego 16 jednostkowych i 23 testy integracyjne.

### Testy jednostkowe

Testy jednostkowe zostały tworzone według następującego klucza:

- Nazwa w formacie kiedy ... , wtedy... (when ... , then ...). Na przykład:

public void WhenGameAddedWith3Players\_ThenEveryPlayerHaveName()

Czyli: „Kiedy gra jest tworzona z 3 graczami, wtedy każdy gram ma imię”.

- Każdy test posiada sekcje Arrange, Act, Assert (Zaaranżuj, Działaj, Zapewnij). Niektóre testy – testy konstruktorów - mają sekcję Arrange i Act (Zaaranżuj i Działaj) jako jedną operację. Na przykład:

public void WhenGameAddedWith3Players\_ThenEveryPlayerHaveName()

{

//Arrange & Act

var game = new PlanningPoker("Jan", "Andrzej", "Zdzisław");

//Assert

game.Players.Should().HaveCount(3);

game.Players.Should().ContainSingle(x => x.Name == "Jan");

game.Players.Should().ContainSingle(x => x.Name == "Andrzej");

game.Players.Should().ContainSingle(x => x.Name == "Zdzisław");

}

- Testy jednostkowe są podzielone i ich implementacja znajduje się w następujących klasach:

PlanningPokerTests.cs – testy konstruktorów dodawania instancji gry Planning Poker:

|  |  |  |  |
| --- | --- | --- | --- |
| **Metoda** | **Argumenty** | **Testowane zachowanie** | **Liczba przypadków testowych** |
| WhenGameAddedWith3Players\_ThenEveryPlayerHaveName() |  | Gdy gra jest dodana z 3 graczami, każdy z graczy powinien mieć imię | 1 |
| WhenGameAddedWith1Player\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z 1 graczem, aplikacja powinna zwrócić wyjątek | 1 |
| WhenGameAddedWith0Player\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z 0 graczami, aplikacja powinna zwrócić wyjątek | 1 |
| WhenGameAddedWithNULLPlayer\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z NULL graczami, aplikacja powinna zwrócić wyjątek | 1 |
| WhenGameAddedWith2IdenticalPlayers\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z dwoma graczami o identycznych imionach, aplikacja powinna zwrócić wyjątek | 1 |
| WhenGameAddedWithAPlayerWithNoName\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z jednym graczem bez imienia, aplikacja powinna zwrócić wyjątek | 1 |
| WhenGameAddedWithAPlayerWithNameTooShort\_ThenExceptionIsThrown() |  | Gdy gra jest dodana z graczem o zbyt krótkim imieniu, aplikacja powinna zwrócić wyjątek | 1 |

Tabela 1. Testy konstruktorów PlanningPoker

EstuationTests.cs – testy metod zagrywania kart estymacji

|  |  |  |  |
| --- | --- | --- | --- |
| **Metoda** | **Argumenty** | **Testowane zachowanie** | **Liczba przypadków testowych** |
| WhenPlayerAddsCorrectEstimation\_EstimationValueIsStored(int a) | int a | Gdy gracz zagra poprawną kartę estymacji, wartości karty powinna być zachowana | 3 |
| WhenPlayerAddsIncorrectEstimation4\_ExceptionIsThrown() |  | Gdy gracz zagra niepoprawną kartę estymacji, aplikacja powinna zwrócić wyjątek | 1 |
| WhenPlayerEditsCorrectEstimation\_EstimationValueIsStored() |  | Gdy gracz poprawnie wyedytuje zagraną kartę estymacji, wartość karty estymacji powinna być zachowana | 1 |
| WhenPlayerEditsIncorrectEstimation\_ExceptionIsThrown() |  | Gdy gracz niepoprawnie wyedytuje zagraną kartę estymacji, aplikacja powinna zwrócić wyjątek | 1 |

Tabela 2. Testy obiektu Estimation

StoryTests.cs – testy konstruktorów dodawania obiektu Story

|  |  |  |  |
| --- | --- | --- | --- |
| **Metoda** | **Argumenty** | **Testowane zachowanie** | **Liczba przypadków testowych** |
| WhenStoryAddedWithNoName\_ThenExceptionIsThrown() |  | Gdy obiekt Story jest dodany bez nazwy, aplikacja powinna zwrócić wyjątek | 1 |
| WhenStoryAddedWithNameTooShort\_ThenExceptionIsThrown() |  | Gdy obiekt Story jest dodany z za krótką nazwą, aplikacja powinna zwrócić wyjątek | 1 |
| WhenStoryAddedWithName\_TheStoryHasProperName() |  | Gdy obiekt Story jest dodany z poprawną nazwą, story ma nazwę | 1 |

Tabela 3. Testy obiektu Story



Rysunek 5. Testy w Nunit

- Części testów działa w kilku przypadkach testowych (test cases). W kodzie wygląda to następująco:

[TestCase(-2)]

[TestCase(2)]

[TestCase(0)]

public void WhenPlayerAddsCorrectEstimation\_EstimationValueIsStored(int a)

{

//Arrange & Act

var estimation = new Estimation("Jan", "Story Name", a);

//Assert

estimation.PlayerName.Should().Be("Jan");

estimation.StoryName.Should().Be("Story Name");

estimation.EstimationValue.Should().Be(a);

}

Oznacza to, że dany test jest uruchamiany 3 razy dla 3 różnych wartości.

### Testy integracyjne

Powstało 23 testy integracyjne (licząc z przypadkami, test cases) testujące różne klasy i zależności między nimi. Konwencja nazewnictwa jest taka sama jak w przypadku testów jednostkowych, struktura każdego testu jest taka sama jak w przypadku testów jednostkowych.

Wszystkie testy integracyjne dokumentuje tabela:

|  |  |  |  |
| --- | --- | --- | --- |
| **Metoda** | **Argumenty** | **Testowane zachowanie** | **Liczba przypadków testowych** |
| WhenAtLeastOnePlayerPlaysBreak\_ApplicationIsInBreakMode() |  | Gdy jeden z graczy zagra kartę "przerwa", aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysTooBigInAllEqualMode\_ThenStoryIsTooBig() |  | Gry jeden z graczy zagra kartę "za duże" w trybie "wszystkie równe" aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysUnknownInAllEqualMode\_ThenStoryIsUnknown() |  | Gdy jeden z graczy zagra kartę "?" w trybie "wszystkie równe" , aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysTooBigInAverageMode\_ThenStoryIsTooBig() |  | Gry jeden z graczy zagra kartę "za duże" w trybie "średnia" aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysUnknownInAverageMode\_ThenStoryIsUnknown() |  | Gdy jeden z graczy zagra kartę "?" w trybie "średnia" , aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysTooBigInAverageRemoveMinMaxMode\_ThenStoryIsTooBig() |  | Gry jeden z graczy zagra kartę "za duże" w trybie "średnia, usuń skrajności" aplikacja powinna zwracać wyjątek | 1 |
| WhenAtLeastOnePlayerPlaysUnknownInAverageRemoveMinMaxMode\_ThenStoryIsUnknown() |  | Gdy jeden z graczy zagra kartę "?" w trybie "średnia, usuń skrajności" , aplikacja powinna zwracać wyjątek | 1 |
| WhenApplicationInAllEqualModeEachEstimationIsEqual\_ThenStoryHasEstimation(int a) | int a | Gdy aplikacja jest w trybie "wszystkie równe" i każda zagrana karta jest równa, estymacja powinna być równa wartości tej karty | 3 |
| WhenApplicationInAllEqualModeEachEstimationIsNotEqual\_ThenThrowException(int a, int b, int c) | int a, int b, int c | Gdy aplikacja jest w trybie "wszystkie równe" i nie każda zagrana karta jest równa, aplikacja powinna zwracać wyjątek | 3 |
| WhenApplicationInAverageMode\_ThenCalculateAverage(int a, int b, int c, int d) | int a, int b, int c, int d | Gdy aplikacja jest w trybie "średnia", aplikacja powinna obliczyć średnią zagranych kart | 3 |
| WhenApplicationInAverageRemoveMinMaxMode\_ThenCalculateAverage(int a, int b, int c, int d, int e, int f) | int a, int b, int c, int d, int e, int f | Gdy aplikacja jest w trybie "średnia, usuń skrajności", aplikacja powinna usunąć najniższą i najwyższą wartość oraz obliczyć średnią zagranych kart | 7 |

Tabela 4. Testy integracyjne

### Struktura aplikacji

### Klasa Player

Klasa Player jest klasą pomocniczą. Służy do identyfikacji gracza zagrywającego karty estymacji. Posiada ona atrybut Name z metodą get i set. Name jest ustawiane w konstruktorze. Klasa Player jest testowana przez testy konstruktorów PlanningPoker.

public class Player

{

public Player(string name)

{

Name = name;

}

public string Name { get; set; }

}

### Klasa Story

Klasa Story służy w aplikacji do przechowywania informacji o historyjkach użytkownika i ich estymacji. Posiada metody umożliwiające obliczanie ostatecznej wartości estymacji. Jest klasą, w której znajduje się główna logika biznesowa automatycznego obliczania wartości szacowania.

### Klasa PlanningPoker

W klasie PlanningPoker umożliwia tworzenie gry Planning Poker. Posiada ona logikę umożliwiającą dodawanie odpowiedniej ilości graczy oraz sprawdzanie poprawności ich imion.

### Klasa Estimation

Klasa Estimation pozwala graczom na zagrywanie kart estymacji. Częścią klasy Estimation jest enum EstimationCard który określa jakie wartości kart każdy gracz może zagrać.

public enum EstimationCard

{

Break = -3,

TooBig = -2,

Unknown = -1,

Zero = 0,

One = 1,

Two = 2,

Three = 3,

Five = 5,

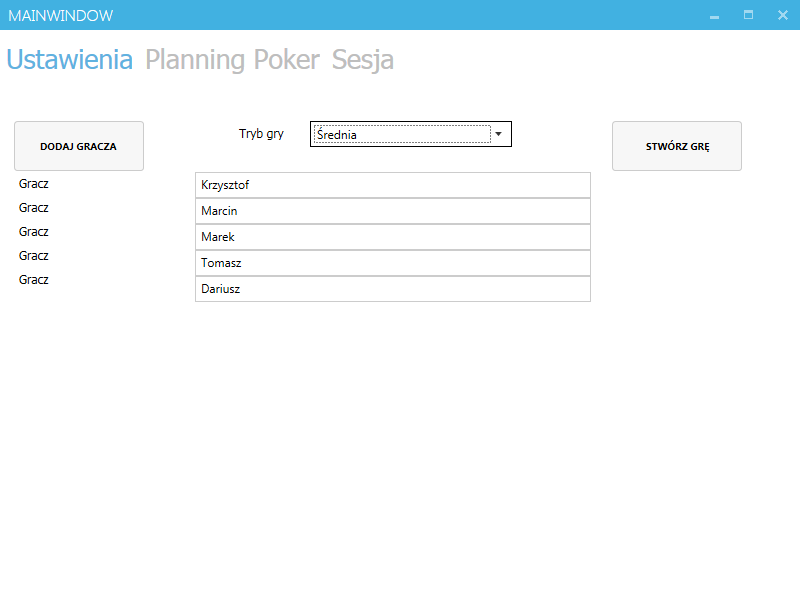
Eight = 8,

Thirteen = 13

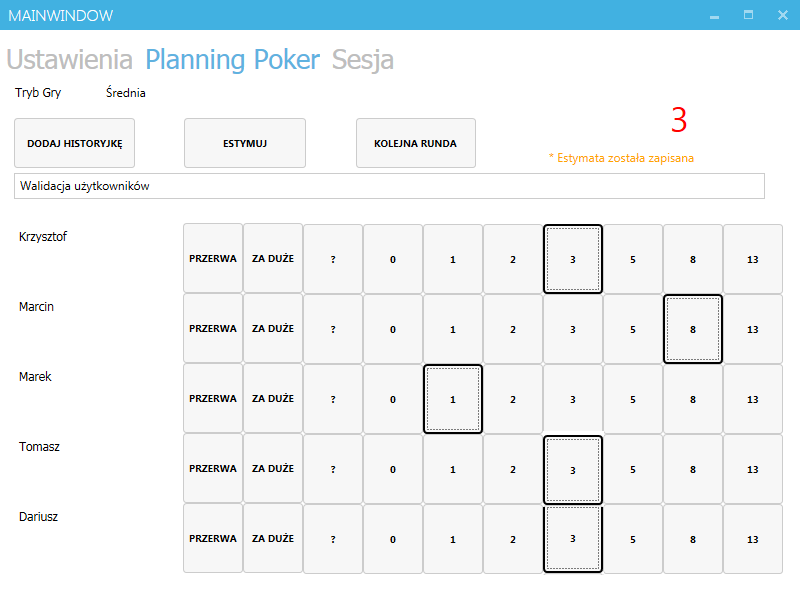
}

### Graficzny interfejs użytkownika

Graficzny interfejs użytkownika (GUI) napisany w technologii WPF służy do łatwej prezentacji funkcjonalności aplikacji. Jest to także kolejny element testujący napisaną wcześniej logikę biznesową. GUI powstało po napisaniu testów i logiki biznesowej Planning Poker. Użyto tutaj darmowego stylu metro MahApps.[[15]](#endnote-15)



Rysunek 6. Ekran uruchamiania gry, 5 graczy, tryb średnia



Rysunek 7. Ekran gry, oddane głosy, historyjka poddana estymacji

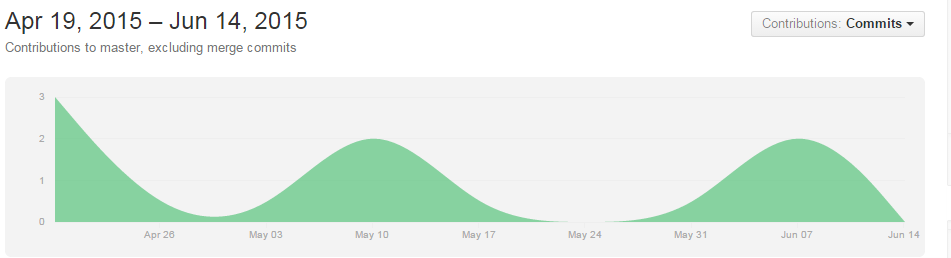


Rysunek 8. Zapis sesji

### GIT

Do kontroli wersji aplikacji użyto hostingowego serwisu internetowego GitHub. Cała aplikację wraz z dokumentacją jest dostępna pod linkiem:

<https://github.com/tompietka/PlanningPoker>



Rysunek 9. Praca nad programem Planning Poker w serwisie GitHub, kontrybucje w czasie

## Przykład tworzenia aplikacji

W tym rozdziale postaram się przedstawić, ja wygląda pisanie w technice TDD na przykładzie aplikacji Planning Poker.

### Definiowanie testu krok po kroku

Tak jak opisywałem wcześniej, zaczynam od pisania testu jednostkowego. Prześledzę krok po kroku jak powstaje test i później testowana logika. Wybieram do tego celu test integracyjny sprawdzający, czy aplikacja poprawnie oblicza średnią z wartości zagranych kart wszystkich graczy. W technice TDD zaczynami od dodania testu. Test składa się z 3 części – Arrange, Act, Assert opisanych w części teoretycznej:

[Test]

public void WhenApplicationInAverageMode\_ThenCalculateAverage()

{

//Arrange

//Act

//Assert

}

Można zauważyć, iż:

* Test jest metodą typu typu void.
* Nazwa testu jest w określonej konwencji. Składa się z 2 części – When (kiedy coś się stanie) Then (efekt)

Następnie dopisuję logikę biznesową testu, najpierw sekcja Arrange:

[Test]

public void WhenApplicationInAverageMode\_ThenCalculateAverage()

{

//Arrange

const string player1 = "Jan";

const string player2 = "Tomasz";

const string player3 = "Józek";

const string storyName = "Walidacja użytkowników";

//Act

//Assert

}

W tej sekcji definiujemy zmienne, 3 graczy z imionami oraz nazwę historyjki do estymacji.

Kolejną sekcją jest sekcja Act, test wygląda następująco:

[Test]

public void WhenApplicationInAverageMode\_ThenCalculateAverage()

{

//Arrange

const string player1 = "Jan";

const string player2 = "Tomasz";

const string player3 = "Józek";

const string storyName = "Walidacja użytkowników";

//Act

var story = new Story(storyName);

var estimation1 = new Estimation(player1, storyName, 3);

var estimation2 = new Estimation(player2, storyName, 8);

var estimation3 = new Estimation(player3, storyName, 1);

story.EstimateAverageMode(estimation1, estimation2, estimation3);

//Assert

}

W tej sekcji:

- tworzę obiekt story z nazwą zdefiniowaną wcześniej;

- tworzę 3 obiekty estymacji przekazując jako parametry kolejno: imię gracza, historyjkę oraz wartość zagranej karty;

- uruchamiam metodę, która jest odpowiedzialna za obliczanie średniej.

Do tego, aby test był kompletny brakuje na tym etapie sekcji Assert. Po jej napisaniu nasz kompletny test wygląda następująco:

[Test]

public void WhenApplicationInAverageMode\_ThenCalculateAverage()

{

//Arrange

const string player1 = "Jan";

const string player2 = "Tomasz";

const string player3 = "Józek";

const string storyName = "Walidacja użytkowników";

//Act

var story = new Story(storyName);

var estimation1 = new Estimation(player1, storyName, 3);

var estimation2 = new Estimation(player2, storyName, 8);

var estimation3 = new Estimation(player3, storyName, 1);

story.EstimateAverageMode(estimation1, estimation2, estimation3);

//Assert

story.Estimation.Should().Be(3);

}

Dopisane wywołanie metody bazującej na Fluent Assertions (patrz sekcja technologia) sprawdza, czy wcześniej wywołana metoda w sekcji Act rzeczywiście obliczyła poprawną średnią.

Kolejnym etapem jest uruchomienie testu (patrz etap Red, z teorii). Test nie może się udać, ponieważ metoda story.EstimateAverageMode nie jest jeszcze napisana. Nunit informuje nas o tym w następujący sposób:



Rysunek 10. Nunit, test się nie powiódł

Zgodnie z TDD dopisuję metodę EstimateAverageMode jako część klasy story:

public void EstimateAverageMode(params Estimation[] estimations)

{

Estimations = estimations.ToList();

Estimation = (int)Estimations.Average(estimation => estimation.EstimationValue);

}

Metoda:

- Przyjmuje parametry o typie obiekt Estimation.

- Może przyjąć nieskończenie wiele obiektów Estimation

- Zapisuje obiekty Estimation do listy

- Oblicza średnią i „castuje” wynik do typu integer (aby pozbyć się miejsc po przecinku).

Wracam do testu, który wcześniej się nie powiódł i uruchamiam go jeszcze raz aby sprawdzić, czy moja metoda spełnia założenia. Nunit informuje mnie, że test się powiódł w następujący sposób:



Rysunek 11. Nunit, test się powiódł

Po tym etapie, zgodnie z zasadami TDD mogę wykonać następujące akcje:

- Refactoring lub

- Napisanie kolejnego testu.

W celach pokazowych, wybieram refactoring. Chcę tym samym testem sprawdzić kilka przypadków, aby mieć pewność, że moja metoda jest poprawna.

### Refactoring

Na etapie refactoring, dodaję kilka przypadków testowych. W tym celu muszę nieznacznie zmodyfikować mój test:

[TestCase(3, 8, 1, 3)]

[TestCase(3, 3, 3, 3)]

[TestCase(13, 5, 1, 6)]

public void WhenApplicationInAverageMode\_ThenCalculateAverage(int a, int b, int c, int d)

{

//Arrange

const string player1 = "Jan";

const string player2 = "Tomasz";

const string player3 = "Józek";

const string storyName = "Walidacja użytkowników ";

//Act

var story = new Story(storyName);

var estimation1 = new Estimation(player1, storyName, a);

var estimation2 = new Estimation(player2, storyName, b);

var estimation3 = new Estimation(player3, storyName, c);

story.EstimateAverageMode(estimation1, estimation2, estimation3);

//Assert

story.Estimation.Should().Be(d);

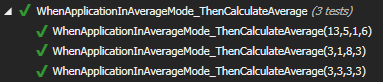
}

Zmiany, w stosunku do pierwszego testu:

- Dodano przypadki testowe. Pierwszy przypadek jest identyczny z tym, który był napisany wcześniej. Dwa kolejne sprawdzają inne wartości.

- Metoda przyjmuje teraz 4 argumenty typu integer, 3 pierwsze to wartości zagranych kart estymacji, ostatnia to spodziewany wynik

Bez dopisania ani jednej linijki kodu w klasie story (metoda EstimateAverageMode wygląda identycznie jak powyżej) uruchamiam testy. Tym razem mam 3 przypadki testowe, każdy z nich kończy się sukcesem. Oto jak Nunit informuje mnie, że wszystko jest w porządku:



Rysunek 12. Nunit, test case, testy się powiodły

Oczywiście za każdym razem, gdy uruchamiam testy, uruchamiam wszystkie do tej pory napisane – nie tylko jeden, który zostaje dodany. W ten sposób zapewniam siebie, że nie ma problemu z tak zwaną regresją – czyli nowa funkcjonalność nie zaburzyła działania aplikacji w innym miejscu. Jest to szczególnie istotne w przypadku pisania testów integracyjnych testujących całość działania systemu.

### Wyjątki

Szczególnym rodzajem testów są testy wyjątków w aplikacji. W projekcie inżynierskim pokazuje dwa sposoby testowania wyjątków w aplikacji. Pierwszy sposób to użycie klasy Action i wyrażenia lambda (=>, takie że):

[Test]

public void WhenGameAddedWith1Player\_ThenExceptionIsThrown()

{

//Arrange & Act

Action a = () => new PlanningPoker("Jan");

//Assert

a.ShouldThrow<ArgumentException>();

}

Część kodu testowana przez powyższy test to metoda Validate klasy PlanningPoker:

private static void Validate(string[] playerNames)

{

if (playerNames == null) throw new ArgumentException("Dodaj więcej graczy");

if (playerNames.Length <= 2) throw new ArgumentException("Gra musi mieć conajmniej 3 graczy");

if (playerNames.Any(x => x.Length <= 2)) throw new ArgumentException("Gracze muszą mieć imiona dłuższe niż 2 znaki");

if (playerNames.Any(x => x == string.Empty)) throw new ArgumentException("Każdy gracz musi mieć imię");

if (playerNames.Distinct().Count() != playerNames.Length) throw new ArgumentException("Nie można dodać gry z graczami o tym samym imieniu");

}

Drugim sposobem jest przekazanie oczekiwanego wyjątku podczas definicji testu, ten sposób został użyty w testowaniu logiki biznesowej. Zastępuje on sekcję Assert testu:

[Test]

[ExpectedException(typeof(ArgumentException))]

public void WhenPlayerEditsIncorrectEstimation\_ExceptionIsThrown()

{

//Arrange

var estimation = new Estimation("Jan", "Story Name", 2);

//Act & Assert

estimation.Edit(estimation, 4);

}

Część kodu testowana przez powyższy test to metoda Validate klasy Estimation:

private static void Validate(int estimationValue)

{

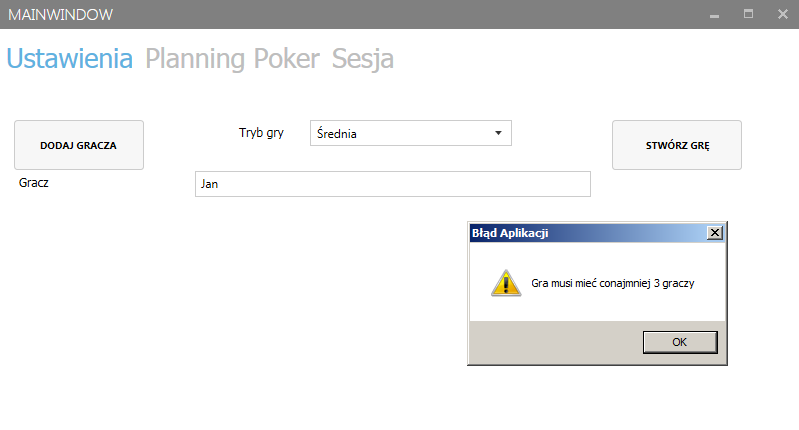
if(estimationValue == -3) throw new ArgumentException("Przerwa w grze");

var validationFlag = Enum.IsDefined(typeof (EstimationCard), estimationValue);

if(validationFlag == false) throw new ArgumentException("Wrong Estimation Value");

}

Wyjątki są przechwytywane przez aplikację GUI. Prezentacja wyjątku użytkownikowi końcowemu wygląda następująco:



Rysunek 13. Graficzny interfejs użytkownika, wyjątek podczas dodawaniu gry z jednym graczem

# Wnioski

Następujące wnioski mogą zostać wysnute po przeprowadzeniu projektu:

- Test-Driven Development jest techniką, której największą zaletą jest dobre zaprojektowanie projekt (ang Design). Często skrót TDD rozwija się jako Test-Driven Design. Dzięki temu, iż aplikacja nie jest zaprojektowana wcześniej mamy do czynienia ze zmianiającą się architekturą w trakcie pisania kodu. Jest to zjawisko bardzo dobrze wpisujące się w zasady zdefiniowane w ruchu Agile.

- Dzięki technice TDD zarządzanie projektem w czasie jest bardzo proste. Większość kodu jest pokryta testami jednostkowymi i dużo łatwiej jest zmieniać implementacje kodu na lepszy, a którego efekt końcowy jest taki sam. Po napisaniu aplikacji, gdy za jakiś czas programista ma pomysł na zmianę zaimplementowanych metod może to zrobić używając testów. Tego typu czynności są znacznie ułatwione.

- Programista piszący w TDD nie musi na bieżąco testować swojego kodu ręcznie w takim stopniu, w jakim testowałby pisząc go tradycyjnie. Napisane testy okazują się w tym przypadku niezwykle pomocne, ponieważ jest pewność, iż powstający kod spełnia założenia biznesowe.

- TDD wymaga dyscypliny i może być trudne do zastosowania dla programistów, którzy wiele lat spędzili na kodowaniu metodami tradycyjnymi. Jednak uważam TDD za świetną metodę do nauki programowania od podstaw – wtedy nie trzeba pozbywać się wcześniej nabytych nawyków.

- Może wydawać się, iż pisanie kodu w technice TDD zajmuje więcej czasu. Na początku na pewno tak jest, jednak z nabieraniem wprawy w kodowaniu w tej technice czas ten się skraca. W perspektywie całego projektu czas powinien być krótszy niż w przypadku metod tradycyjnych – ponieważ większość wysiłku poświęcone jest na kodowanie – nie na architekturę i testowanie manualne.

# Bibliografia

1. http://agilemanifesto.org/ [↑](#endnote-ref-1)
2. Extreme Programming Explained: Embrace Change, 2nd Edition, Kent Beck, Cynthia Andres – 26 Listopad 2004 [↑](#endnote-ref-2)
3. http://wikipedia.org/ [↑](#endnote-ref-3)
4. The Agile Samurai – How Agile Masters Deliver Great Software, Jonathan Rasmusson, The Pragmatic Bookshelf, Wrzesień 2010, s 12 [↑](#endnote-ref-4)
5. Scrum Guide, Przewodnik po Scrumie: Reguły gry, Ken Schweber, Jeff Sutherland, Lipiec 2013 [↑](#endnote-ref-5)
6. Agile Principles as Software Engineering Principles: An Analysis, Normand S´eguin, Guy Tremblay, and Houda Bagane D´epartement d’informatique, Universit´e du Qu´ebec `a Montr´eal, Mardzec 2012 [↑](#endnote-ref-6)
7. http://www.extremeprogramming.org/ [↑](#endnote-ref-7)
8. Extreme Programming Explained: Embrace Change, 2nd Edition, Kent Beck, Cynthia Andres, 26 Listopad 2004 [↑](#endnote-ref-8)
9. The art of unit testing with examples in C#, Second Edition, Roy Osherove, Manning, 2014 [↑](#endnote-ref-9)
10. Better Unit Testing with Microsoft® Fakes, Microsoft Corporation, 2013 [↑](#endnote-ref-10)
11. The art of unit testing…, Osherove [↑](#endnote-ref-11)
12. The art of unit testing…, Osherove [↑](#endnote-ref-12)
13. http://dariuszwozniak.net/ [↑](#endnote-ref-13)
14. An Initial Investigation of Test Driven Development in Industry, Boby George, Laurie Williams [↑](#endnote-ref-14)
15. https://github.com/MahApps/MahApps.Metro [↑](#endnote-ref-15)