**Katolicki Uniwersytet Lubelski**

**Wydział Matematyki, Informatyki i Architektury Krajobrazu**

Studia licencjackie

na kierunku Informatyka

specjalność: Programowanie i przetwarzanie informacji

**Vladyslav Verenich**

Praca dyplomowa

**Wzorce projektowe w języku C#**

**Design patterns via C#**

Praca dyplomowa

napisana pod kierunkiem

dr Andrzeja Bobyka

Lublin, 2016

# 

# Spis treści

[Spis treści 1](#_Toc453676059)

[Wstęp 1](#_Toc453676060)

[Rozdział 1. ZAPOZNANIE Z POJĘCIAMI ORAZ TECHNOLOGIAMI 1](#_Toc453676061)

[1.1. NET / C# 1](#_Toc453676062)

[1.2. Programowanie Obiektowe, paradygmaty 2](#_Toc453676063)

[1.3. SOLID 3](#_Toc453676064)

[1.4. UML 4](#_Toc453676065)

[1.5. Problemy projektowania nowoczesnych systemów informatycznych 8](#_Toc453676066)

[Rozdział 2. Wzorce projektowe 11](#_Toc453676067)

[2.1. Pojęcie wzorca projektowego 11](#_Toc453676069)

[2.2. Klasyfikacja wzorców 11](#_Toc453676070)

[2.3. Factory Method 12](#_Toc453676071)

[2.4. Abstract Factory 14](#_Toc453676072)

[2.5. Singleton 17](#_Toc453676073)

[2.6. Adapter 19](#_Toc453676074)

[2.7. Facade 22](#_Toc453676075)

[2.8. Composite 24](#_Toc453676076)

[2.9. Proxy 28](#_Toc453676077)

[2.10. Command 30](#_Toc453676078)

[2.11. Iterator 32](#_Toc453676079)

[2.12. Observer 34](#_Toc453676080)

[2.13. Strategy 38](#_Toc453676081)

[Rozdział 3. NAPISANIE APLIKACJI WEBOWEJ Z UŻYCIEM WZORCÓW 41](#_Toc453676082)

[2.14. Opis aplikacji 41](#_Toc453676083)

[2.15. Warstwy aplikacji 45](#_Toc453676084)

[2.16. Szczegóły implementacji 46](#_Toc453676085)

[Zakończenie 49](#_Toc453676086)

[Bibliografia 51](#_Toc453676087)

# Wstęp

Celem mojej pracy jest przedstawienie najważniejszych oraz najczęściej używanych wzorców projektowych oraz architektonicznych przy tworzeniu nowoczesnych aplikacji Webowych na platformie .NET w języku C#.

Czasami nie zdajemy sobie sprawy o ważności ich świadomego wykorzystywania w codziennej pracy nad projektem. Bez nich byłoby trudno wyobrazić sobie możliwość tworzenia tak skomplikowanego oprogramowania nowoczesnego. Dla mnie bardzo przerażające było to, że niektórzy koledzy nie mogli nazwać, nawiać trzech wzorców z katalogu GoF. Jest to bardzo zła tendencja o ile nieznanie wzorców projektowych w nowoczesnym projektowaniu rozwiązań informatycznych można przyrównać do nieznania całek w nowoczesnej Matematyce oraz Fizyce. Wzorce są nowym poziomem abstrakcji w informatyce, tak samo, jak programowanie obiektowe. Abstrakcja to pojęcie, które określa progres nie tylko technologiczny, ale też całej ludzkości. Im większy jest poziom abstrakcji tym bardziej złożony system można zbudować.

W rozdziale pierwszym robię wprowadzenie do pozostałych części pracy. Tu opowiem o krokach niezbędnych do zrozumienia wzorców projektowych w języku programistycznym C#. W podrozdziałach opowiem o platformie .NET wyjaśnię główne stereotypy języka C#. Opowiem o programowaniu obiektowym, pojęciach SOLID. Także zrobię wstęp do języka modelowania UML oraz opowiem o problemach, z którymi się spotyka architekt przy projektowaniu systemu informatycznego.

W drugim rozdziale skupię się na wzorcach projektowych. Dokładnie wyjaśnię czym jest wzorzec, a potem klasyfikuję wszystkie wzorce projektowe. Potem przeanalizuję 11 najczęściej używanych wzorców projektowych w .NET oraz 2 wzorce architektoniczne.

W ostatnim rozdziale opowiem o aplikacji webowej napisanej z użyciem wzorców oraz najbardziej nowoczesnych na daną chwile technologii.

# Rozdział 1. Zapoznanie z pojęciami oraz technologiami

## NET / C#

CLI (Common Language Infrastructure) – specyfikacja stworzona przez Microsoft, zawierająca:

* VES (ang. *Virtual Execution System*) – specyfikacja opisująca tworzenie maszyny wirtualnej, w której będą uruchamiane programy pisane dla danej CLI.
* CLS (ang. *Common Language Specification*) – specyfikacja opisująca reguły niezbędne dla tworzenia kompilatora odpowiedniego języka programowania, który będzie używany w CLI.
* CTS (ang. *Common Type System*) – podzbiór CLS, opisuje podstawowe typy danych współdzielone przez języki adaptowane pod CLS. CTS jest fundamentem i podzbiorem FCL (ang. *Framework Class Library*).

.NET – realizacja CLI działająca na systemie Windows. Podobnie do .NET mamy CLI o  nazwie MONO (Linux, Windows), ROTOR (FreeBSD, Windows), Portable .NET (FreeBSD, Linux) [2].

C# **nie** kompiluje się w CIL (ang. *Common Intermidiate Language*). CIL tak samo jak C#, VB, C++ mają realizowane w .NET według CLS swoje kompilatory. W wyniku kompilacji otrzymujemy bajt-kod. Ten bajt-kod wykonuje się przez CLR przy pomocy mechanizmu JIT (ang. *Just In Time*). Także w wykonaniu uczestniczy FCL, o ile bajt-kod zawiera wywołania metod, operacje na typach, które zawierają się w FCL.

Przykładowy ciąg uruchomienia aplikacji napisanej w języku C# przy użyciu .NET: **C#** kod, **csc.exe** (kompilator C#), **bajt-kod**, **.NET** (**CLR**, czyli VES + **FCL**, czyli CTS oraz rozszerzenia), **kod natywny**.

W taki sposób widać, że celem CLI jest wspieranie naraz wielu platform o ile bajt-kod możemy przekazać do dowolnej CLI, oczewiście, jeżeli ona będzie zainstalowana w systemie, na  którym chcemy uruchomić program.

C# - język obiektowy stworzony przez Microsoft dla platformy .NET, został standaryzowany jako ECMA-334 oraz ISO/IEC 23270 [7]. C# tak samo, jak Java oraz C++ należy do rodziny języków z C-podobnym syntaksem. Pan Anders Hejlsberg, twórca C#, złączył najlepsze konstrukcje syntaktyczne z innych języków programowania. Pod tym względem jest to unikatowy oraz na dany moment najbogatszy syntaktycznie język programowania. To znaczy, że programiści C# mają bogaty wybór, wiele możliwości realizacji a najgłówniejsze łatwość realizacji. To pozwala pisać aplikacje szybciej, optymalnie oraz z większą jakością. Niema w świecie pod tym względem żadnego języka programowania lepszego od C#.

## Programowanie Obiektowe, paradygmaty

Programowanie obiektowe bazuje się na dwóch głównych pojęciach: algorytmu oraz modelu. Algorytmizacją jest napisaniem najprostszych programów z użyciem zmiennych, pętli, funkcji. Modelowanie pozwala sformować wysokopoziomowe, abstrakcyjne pojęcia, czyli klasy oraz obiekty, a klasy już w sobie zawierają potrzebne mu algorytmy. Także warto zaznaczyć, że  programowanie obiektowe już w sobie zawiera programowanie proceduralne. Z innej strony, języki proceduralne nie zezwalają tworzyć modele [3].

Głównym zadaniem OOP jest rozbicie systemu na obiekty. Kiedy mówimy, że dany obiekt jest egzemplarzem klasy to pod tym rozumiemy, że on wspiera interfejs który definiuje dana klasa [5].

Czym róźni się klasa od typu? Wyobraźmy, że mamy 2 klasy: Dog, Cat, każdy ma operacje Eat() oraz Go(). W danej chwili klasy są róźne, ale rozpatrujemy je jak jeden typ, czyli zwierzę domowe. Jeżeli rozszerzymy klasę Cat o operację CatchMouse(), to otrzymujemy nowy typ, czyli łapacz myszy. Klasa Dog już nie może być za to odpowiedzialna.

Istnieje 6 paradygmatów programowania obiektowego o dokładnej numeracji:

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction
5. Message passing
6. Reusable

Hermetyzacja (ang. *Encapsulation*) – ukrycie informacji.

Jest kilka rodzajów ukrycia informacji a stąd hermetyzacji:

* Ukrycie typów danych

Wykorzystanie dynamic oraz var.

* Ukrycie realizacji

Wykorzystanie modyfikatorów dostępu wewnątrz klasy, rzutowanie do typu wyższego w grafie dziedziczenia.

* Ukrycie części systemów programistycznych (Hermetyzacja Wariacji)

Pozwala przedstawić użytkownikowi ogólny interfejs. Hermetyzacji wariacji formuje abstrakcję. Przykład: formowanie Dokumentu z Title, Body oraz Footer jako pól prywatnych. Dokument to zbiórne pojęcie z części. Dokument nie istnieje, dokument – abstrakcja. Przy formowaniu abstrakcji hermetyzacja jest wtórną a abstrakcja pierwotną.

Dziedziczenie (ang. *Inheritance*) – możliwość zdefiniować nowe klasy według istniejących. Możemy dziedziczyć klasy oraz interfejsy. Ważne jest rozumienie różnicy między dziedziczeniem klasy oraz interfejsu. W przypadku dziedziczenia klasy, realizacja obiektu definiuje się według realizacji drugiego obiektu. Dziedziczenie interfejsu powoduje tworzenie podtypów, co pozwala jeden obiekt użyć zamiast innego.

Polimorfizm (ang. *Polymorphism*) – przy pomocy powiązania dynamicznego pozwala podstawić zamiast jednego obiektu inny o wspólnym interfejsie. Jest 2 rodzaje polimorfizmu: klasyczny oraz AD-HOC. Klasyczny otrzymujemy przy pomocy użycia virtual, rzutowania typów oraz możliwości podmienić obiekt pod czas wykonania.

Abstrakcja (ang. *Abstraction*) – celem jest formowanie zbiorowych pojęć.

Przesyłanie powiadomień (ang. *Message passing*) – organizacja potoków informacyjnych między obiektami. Pozwala jednemu obiektu przekazać powiadomienie innemu.

Powtórne użycie (ang. *Reusable*) – przy pomocy tworzenia klas, metod, dziedziczenia, kompozycji. Powtórne wykorzystanie przy pomocy tworzenia podklas nazywa się przezroczystym pudełkiem o ile wewnętrzna realizacja rodziców jest widoczna dla podklasy. Przy pomocy kompozycji otrzymujemy powtórne wykorzystanie, które nazywa się czarnym pudełkiem o ile aspekty realizacji wewnętrznej obiektów zostają ukryte. Także do reusable należy wykorzystanie bibliotek oraz frameworków.

## SOLID

SOLID – to pięć ważnych zasad w większości zdefiniowanych przez Roberta Martina, które warto brać pod uwagę przy tworzeniu systemu programistycznego. Te zasady bazują się na wzorcach projektowych. Warto zauważyć, że jest dużo więcej zasad opisanych we wzorcach. Nie wiadomo czemu dokładnie te 5 zasad były wyróżnione przez inne.

* Single responsibility

Klasa/obiekt musi wykonywać jedno ogólne zadanie.

* Open/closed

Klasa musi być otwarta dla rozszerzenia oraz zamknięta dla modyfikacji. Rozszerzyć możemy przy pomocy kompozycji lub dziedziczenia.

Jeżeli klasa nie odpowiada niektórym wymaganiam, ale działa dobrze według swego celu, to  nie warto ją modyfikować, tylko stworzyć swoją klasę. Jeżeli część funkcjonału pasuje, to  trzeba zwrócić uwagę na wartość tworzenia nowej klasy. Jeżeli klasę nie można szybko przeanalizować i na jej podstawie stworzyć nową, to warto ją rozszerzyć. Ceną tego rozwiązania jest zbędna ilość elementów w rozszerzonej klasie a stąd większe zapotrzebowanie resursów komputera.

* Liskov substitution

Dziedziczenie warto używać, tylko jeżeli pochodna klasa jest rozszerzeniem klasy bazowej, a  nie zamienia ją. Klienci muszą mieć możliwość wykorzystywania podklas przez interfejs klasy bazowej nie obserwując żadnych rozbieżności. Metody klasy bazowej muszą mieć to samo znaczenie w podklasach.

* Interface segregation

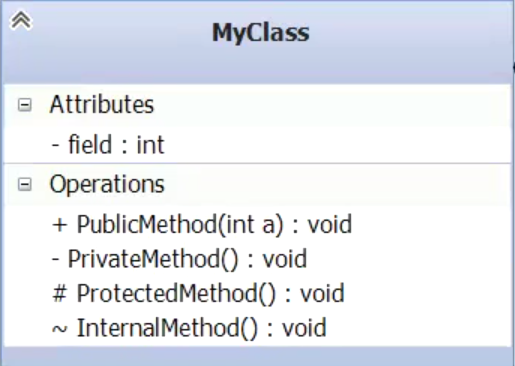
Zamiast jednego dużego interfejsu warto zrobić hierarchie szerokich oraz wąskich interfejsów przy pomocy dziedziczenia.

* Dependency inversion

Abstrahowanie procesu tworzenia obiektu przy pomocy użycia abstrakcyjnych klas produktów.

## UML

UML (ang. *Unified Modeling Language*) – pozwala przedstawić system w wyglądzie diagram. Jest wiele rodzajów diagram UML. W danej pracy jest używany diagram klas, żeby pokazać związki między klasami we wzorcach projektowych.



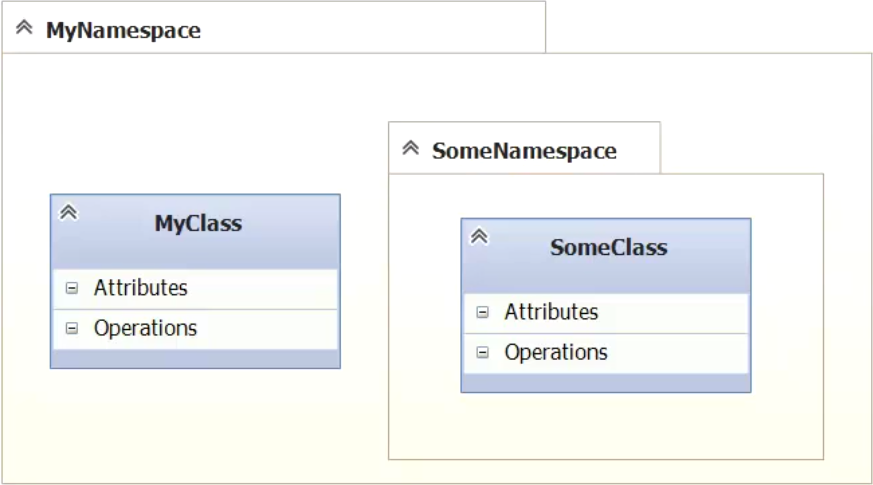
Rysunek 1. Klasyfikator w języku UML

Na Rysunku 1 widzimy klasyfikator, który składa się z trzech sekcji: nagłówku, atrybutów oraz operacji.

* Znak ‘+’ przy atrybutach lub operacjach oznacza, że dany atrybut/operacja jest publiczny.
* Znak ‘-’ przy atrybutach lub operacjach oznacza, że dany atrybut/operacja jest prywatny.
* Znak ‘#’ przy atrybutach lub operacjach oznacza, że dany atrybut/operacja jest chroniony.
* Znak ‘~’ przy atrybutach lub operacjach oznacza, że dany atrybut/operacja jest wewnętrzny.

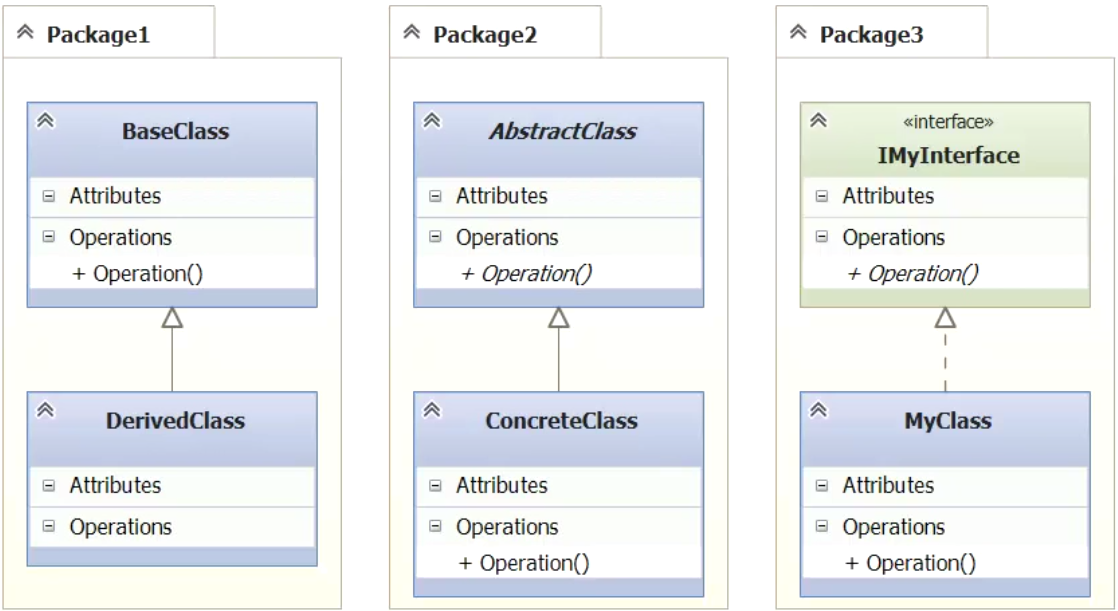
Także, jeżeli mówimy o opisaniu stereotypów, w języku UML mamy klasyfikator ‘Interface’ oraz klasyfikator ‘Enumeration’.

Stereotyp - pojęcie zbiorowe dla klas, struktur, interfejsów, enumeracji oraz delegatów.



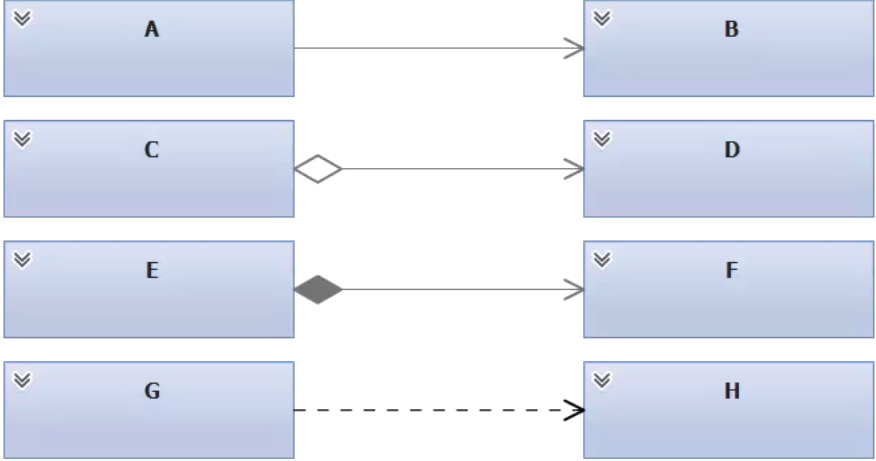
Rysunek 2. Pakiety UML

Na Rysunku 2 widzimy nic innego jak pakiety UML. Pakiet w języku UML reprezentuje przestrzenie nazw (ang. *namespace*) w języku C#.



Rysunek 3. Dziedziczenie UML

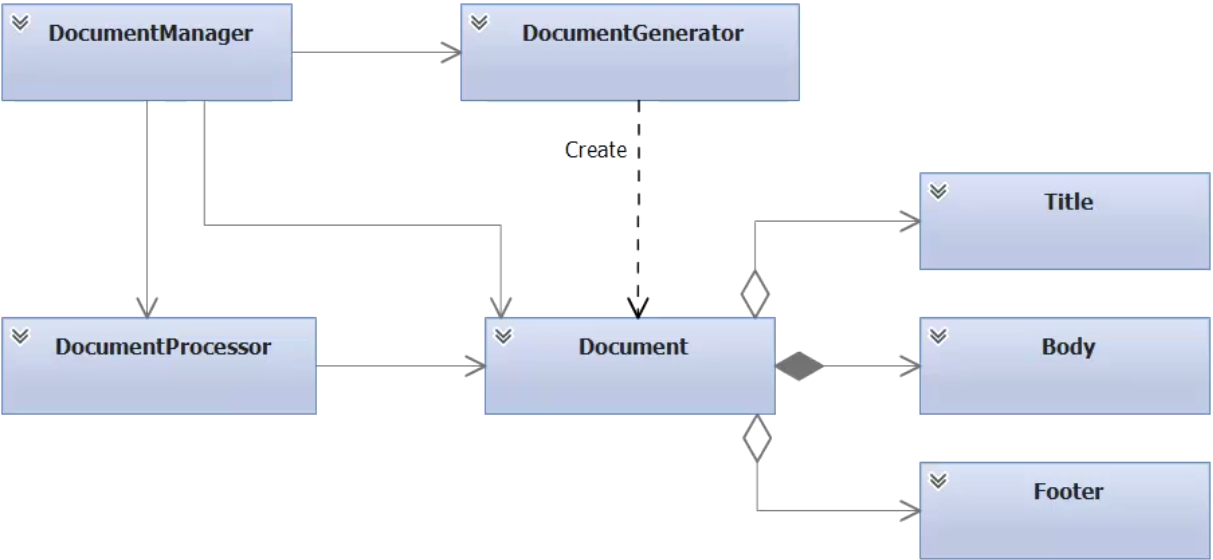
Na Rysunku 3 widzimy dziedziczenie w języku UML. Warto zauważyć, że w przypadku ostatnim my nie tylko realizujemy interfejs, ale też go dziedziczymy. Pierwsze dwa dziedziczenia nazywamy dziedziczeniem realizacji, trzeci nazywamy dziedziczeniem interfejsu.



Rysunek 4. Rodzaje związków w języku UML

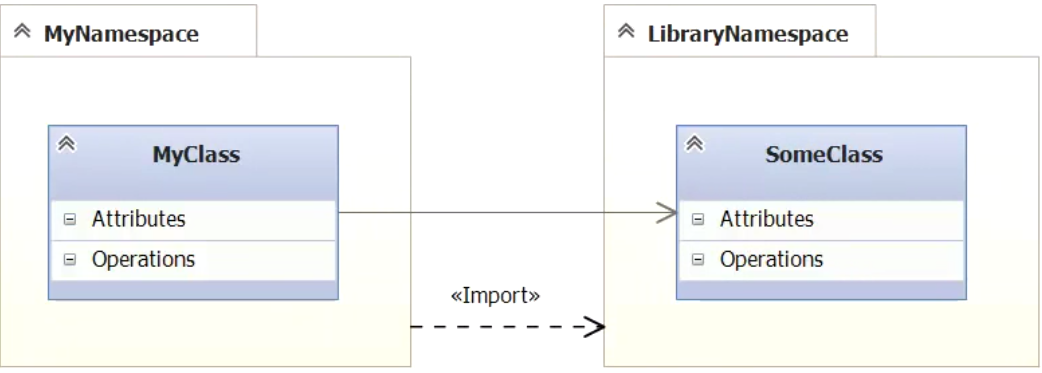
Na Rysunku 4 widzimy jeszcze cztery rodzaje związków między klasami:

* Klasa A jest związana związkiem asocjacji z klasą B. To znaczy, że klasa A zna klasę B lub A wykorzystuje B.
* Klasa C jest związana związkiem agregacji z klasą D. To znaczy, że klasa C składa się z  klasy D lub klasa D jest częścią klasy C, lub klasa C zawiera w sobie klasę D.
* Klasa E jest związana związkiem kompozycji z klasą F. Kompozycja jest rodzajem agregacji. Kompozycja znaczy, że E zawiera F oraz nie może istnieć bez klasy F.
* Klasa G jest związana związkiem zależności z klasą H. Ten związek jest alternatywą wszystkim innym związkom. Jest 13 stereotypów zależności.



Rysunek 5. Przykład wykorzystania związków UML

Na Rysunku 5 widzimy przykład wykorzystania wszystkich przedstawionych wcześniej związków oprócz dziedziczenia.



Rysunek 6. Import Pakietów w języku UML

Na Rysunku 6 obserwujemy import pakietu LibraryNamespace. Jest to analogiczne użyciu using namespace w języku C#.

## Problemy projektowania nowoczesnych systemów informatycznych

Głównym problemem projektowania nowoczesnych systemów informatycznych jest zmniejszenie ilości związków między poszczególnymi klasami [1].

Duże systemy programistyczne są dość trudne w projektowaniu i wspieraniu, dla tego, dla ułatwienia pracy, duże systemy rozbijamy na podsystemy. Głównym zadaniem jest zmniejszenie ilości związków-relacji między klasami, czyli zmniejszenie ilości zależności klas jeden od  drugiego.

Zależność (ang. *dependency*) – to termin techniczny opisujący ilość istniejących związków-relacji.

Jeżeli na diagramie widzimy, że jakaś klasa ma dużo związków z innymi – mówiono jest, że taka klasa jest mocno uzależniona od innych. Także można powiedzieć, że praca egzemplarzu podobnej klasy będzie zależeć od pracy egzemplarzów innej. Zależności klas powodują tak zwane przywiązania.

Przywiązanie – termin logiczny, który zmusza developera zamyślić się nad sensem zależności. Ze strony programowania obiektowego, przywiązania mogą istnieć w dwóch postaciach: dobre – biznesowe przywiązania lub złe – przywiązania techniczne.

Przywiązania biznesowe wyrażają wymagania biznesu. Na przykład klasa Customer jest związana związkiem-relacją asocjacji z klasą Order.

Przywiązania techniczne, z innej strony wyrażają potrzeby systemowe. Naprzykład, kiedy klasa Customer jest związana związkiem asocjacji z klasą DataSet.

Niestety bez niektórej ilości przywiązań technicznych nie da się zaprojektować system. Przywiązanie techniczne może być nazwane dobrym wtedy i tylko wtedy, kiedy zależne encje są zamieszczone w różnych warstwach systemu. Na przykład Customer w Business Layer, DataSet w Data Layer. W takim przypadku przy analizie logiki biznesowej systemu można zignorować związki prowadzące do warstw o niższych poziomach abstrakcji.

Przy projektowaniu systemu musimy zmniejszyć ilość przywiązań oraz pamiętać o  zachowywaniu logicznej całości sensu procesu, który modelujemy. Moc zależności można określić przy pomocy terminu ‘coupling’.

Coupling jest miarą zależności. Coupling posiada wiele metryk pozwalających na  określenie mocy zależności. Opis tych metryk nie wchodzi w kontekst danej pracy.

# Rozdział 2. Wzorce projektowe



## Pojęcie wzorca projektowego

Wzorzec – nowy poziom abstrakcji w tworzeniu systemów programistycznyh. Każdy wzorzec to zestaw złożonych procesów oraz komend które dają nam odpowiedni rezultat. W dzisiejszym świecie programistycznym mówimy wzorcami. Już nie musimy zastanawiać się nad szczegółami implementacji tylko korzystać z gotowego mechanizmu [1].

Ważność wzorców projektowych w programowaniu jest taka sama jak całek w matematyce. Wzorce projektowe pozwolili tworzyć systemy o większej złożoności.

Technicznie wzorzec – to zestaw klas o pewnych związkach. Także wzorcem można nazwać kombinację prymitywnych technik OOP. Ja lubię myśleć o wzorcach jako przykładach pokazujących sposoby organizacji współdziałań klas oraz obiektów.

Wzorce projektowe zezwalają definiować interfejsy, wskazując je główne elementy oraz dane, które możemy przekazywać przez interfejs. Wzorzec też może ‘powiedzieć’ co nie powinno przekazywać się przez interfejs.

## Klasyfikacja wzorców

Banda czterech zaproponowała dwa rodzaje klasyfikacji wzorców. Pierwsza z nich jest klasyfikacją według rodzaju wzorca opisując to, co on robi. W taki sposób dzielimy wzorce na trzy kategorie

* **kreacyjne (konstrukcyjne)** – opisują proces instancjacji egzemplarzów klas.
* **strukturalne** – opisują powiązania między poszczególnymi obiektami oraz klasami
* **czynnościowe** – zachowanie oraz współpraca obiektów.

Drugi rodzaj pozwala kategoryzować wzorce według ich zakresów. Wzorzec może dotyczyć klas lub obiektów.

* **klasowe** – opisujące związki pomiędzy klasami.
* **obiektowe** – opisujące związki pomiędzy obiektami.

## Factory Method

*Nazwa*: Factory method

*Inne nazwy*: Virtual Constructor

*Klasyfikacja*: według celu – kreacyjny, według stosowania – do klas

*Częstość użycia*: bardzo duża

*Przeznaczenie*: być fundamentem wszystkich wzorców kreacyjnych

Jest to bardzo prosty wzorzec, który jest głównym, fundamentalnym składnikiem pozostałych wzorców kreacyjnych. Wzorzec zapewnia abstraktowy interfejs, który możemy użyć do tworzenia obiektów-produktów. Użytkownicy mają możliwość wybierać konkretny produkt, który trzeba stworzyć.

* Strona architekta:

Architekt tworzy abstraktowy szkielet mechanizmu tworzenia produktu.

Architekt deleguje realizatorowi odpowiedzialność za wybór konkretnego produktu do  stworzenia.

* Strona programisty-realizatora:

Programista realizuje konkretne klasy produktów.

Programista przyjmuje odpowiedzialność i decyduje jaki konkretny produkt trzeba stworzyć.

*Przypadki użycia*:

Jeżeli jest wątpliwość jaki wzorzec kreacyjny użyć do tworzenia obiektu, najlepszym rozwiązaniem jest użycie Factory Method o ile mamy możliwość łatwego transformowania do  dowolnego z pozostałych wzorców kreacyjnych.

*Osobliwości C#:*

Wykorzystanie Generics – pozwala nie tworzyć kilka ConcreteCreator klas oraz przy tworzeniu podać typ tworzonego produktu:

ICreator creator = new StandardCreator();

IProduct productA = creator.CreateProduct<ProductA>();

IProduct productB = creator.CreateProduct<ProductB>();

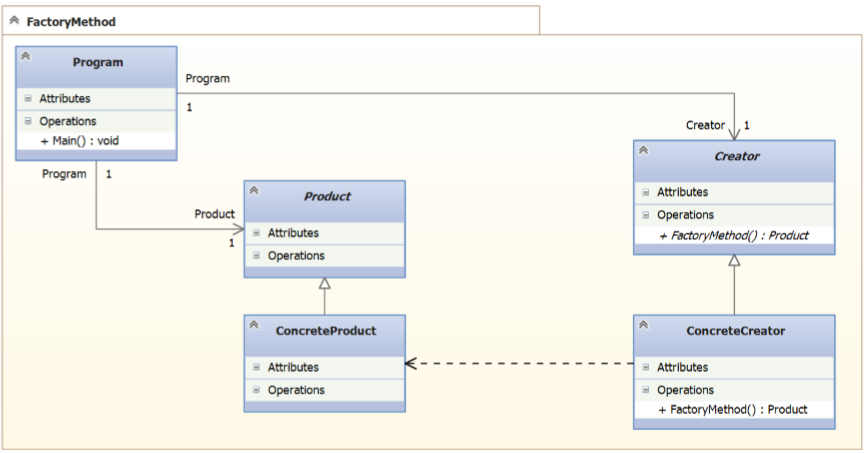
IProduct productC = creator.CreateProduct<ProductC>();

Nazewnictwo:

Warto nadawać takie nazwy, żeby było łatwo zrozumieć, że jest używany kreacyjny wzorzec. Przykładem jest CreateApplication jako fabryczna metoda tworząca aplikację.

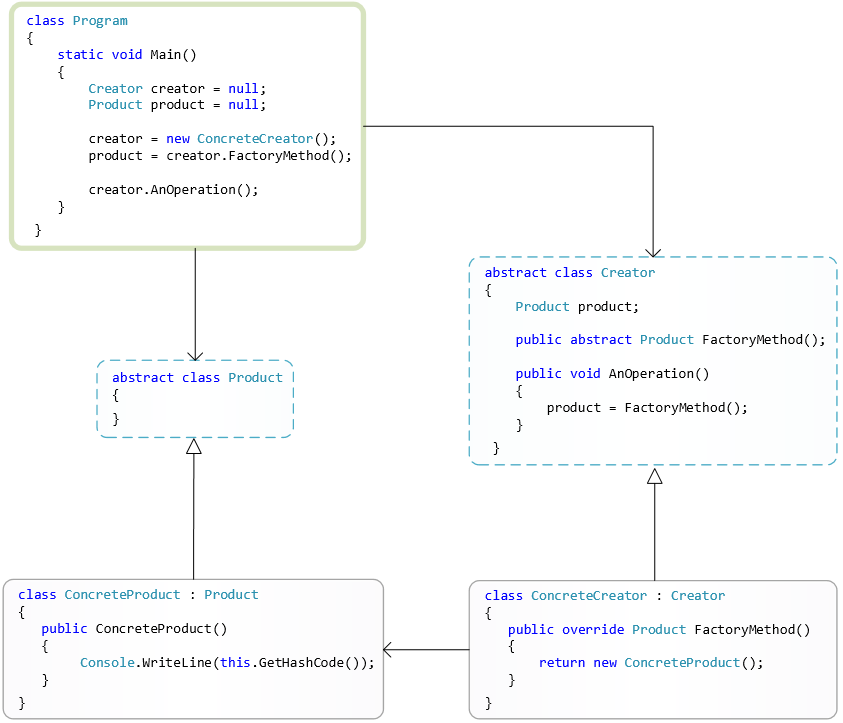
Wykorzystanie w .NET:

Jest fundamentem wszystkich kreacyjnych wzorców, czyli jest wykorzystywany wszędzie gdzie można spotkać wzorce kreacyjne.



Rysunek 7. Struktura Factory Method w języku UML

Na Rysunku 7 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 8. Struktura Factory Method w języku C#

Na Rysunku 8 widzimy szkielet wzorca w języku C#.

## Abstract Factory

*Nazwa*: Abstract Factory

*Inne nazwy*: Kit

*Klasyfikacja*: według celu – strukturalny, według stosowania – do obiektów

*Częstość użycia*: bardzo duża

*Przeznaczenie*: Tworzenie rodzin współdziałających obiektów.

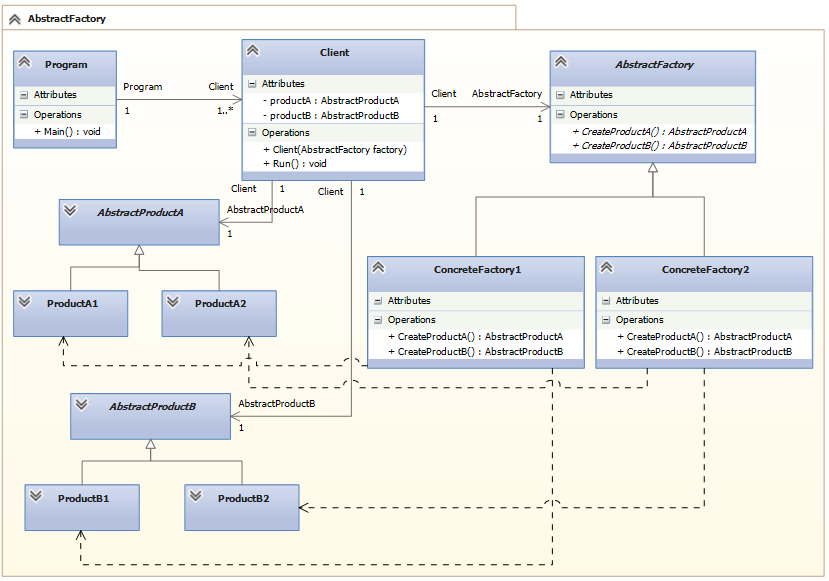
Jest to wzorzeć, który opisuje algorytm tworzenia interfejsu pozwalającego instancjować rodziny powiązanych lub zależnych między sobą obiektów-produktów [5]. Także hermetyzuje informację o konkretnych klasach tworzonych obiektów-produktów.

Żeby zrozumieć do czego służy ten wzorzec można wyobrazić sobie fabrykę Coca Cola. Fabryka produkuje 2 produkty: napój oraz butelkę lub puszkę. Te dwa produkty muszą między sobą współpracować, żeby otrzymać ostateczny produkt. Przykładem antywzorca może być próba użyć produktu z jednej rodziny produktów do współpracy z produktem innej rodziny: nie możemy wlewać cole w butełkę pepsi.

Wykorzystanie w .NET:

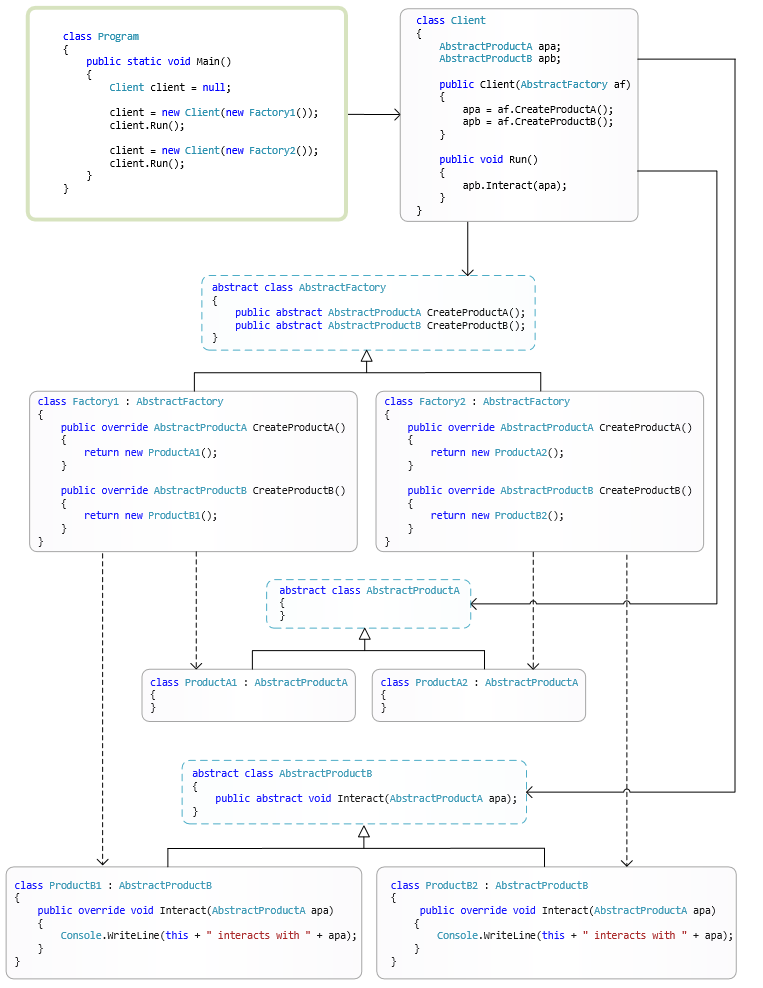
* System.Activities.Presentation.Model.ModelFactory
* System.Data.Common.DbProviderFactory
* System.Data.EntityClient.EntityProviderFactory
* System.Data.Odbc.OdbcFactory
* System.Data.OleDb.OleDbFactory

Oraz inne



Rysunek 9. Struktura Abstract Factory w języku UML

Na Rysunku 9 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 10. Struktura Abstract Factory w języku C#

Na Rysunku 10 widzimy szkielet wzorca w języku C#.

## Singleton

*Nazwa*: Singleton

*Inne nazwy*: Solitaire

*Klasyfikacja*: według celu – kreacyjny, według stosowania – do obiektów

*Częstość użycia*: duża

*Przeznaczenie*: gwarantuje instancjowanie wyłącznie jednego egzemplarza klasy

Żeby zrozumieć ten wzorzec, warto użyć metafory, czyli Słońca. Jeżeli dla naszej planety stworzymy jeszcze jeden egzemplarz Słońca, to całkiem zniszczymy istniejący system. Żeby tak się nie stało, mamy specjalny wzorzec – Singleton.

Strona architekta:

* Po wykorzystaniu tego wzorca architekt może być pewien tego, że programista nie stworzy zbędny egzemplarz klasy.

Kiedy użyć:

* Użyć, kiedy w systemie musi być ograniczona ilość egzemplarzów jednej klasy

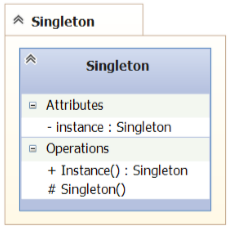
C# osobliwości:

Jedną z możliwości realizacji tego wzorca w C# jest użycie klas statycznych. Według mnie jest to preferowany sposób realizacji, aczkolwiek ma swoje wady: nie pasuje w przypadku potrzeby tworzenia kilkóch instancji klasy singleton. Klasy statyczne nie mogą uczęstniczyć w dziedziczeniu, a to znaczy nie mogą mieć relacje polimorficzne.

Przy użyciu wielowątkowości zostaje ryzyko tworzenia kilkóch egzemplarzów. To ryzyko można wyeliminować przy pomocy konstrukcji lock(){} wewnątrz której i odbędzie się instancjacja egzemplarzu obiektu.

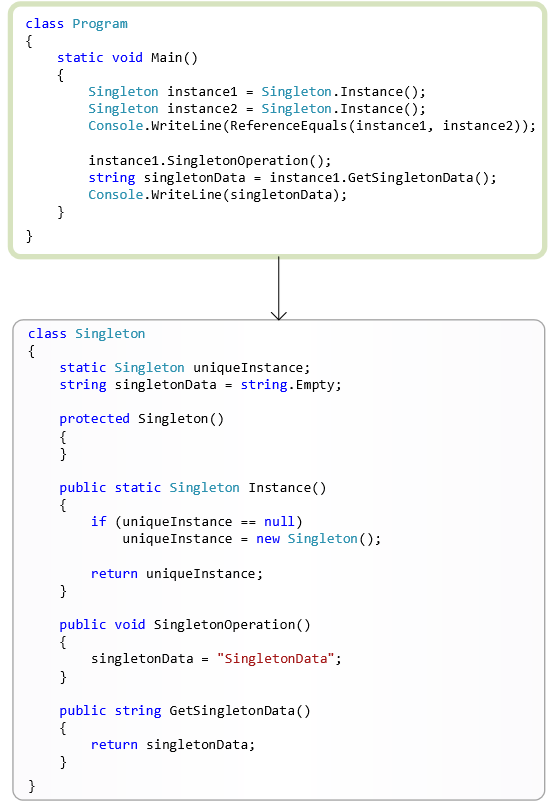
Wykorzystanie w .NET

* System.ServiceModel.ServiceHost
* System.Data.DataRowComparer



Rysunek 11. Struktura Singleton w języku UML

Na Rysunku 11 możemy zobaczyć reprezentację wzorca singleton w wyglądzie pojedynczego klasyfikatora.



Rysunek 12. Struktura Singleton w języku C#

Na Rysunku 12 widzimy szkielet wzorca w języku C#.

## Adapter

*Nazwa*: Adapter

*Inne nazwy*: Wrapper

*Klasyfikacja*: według celu – strukturalny, według stosowania – do klas oraz obiektów

*Częstość użycia*: duża

*Przeznaczenie*: uproszczenie oraz adaptacja interfejsu klasy do potrzeb użytkownika klasy

Jest to prosty wzorzec celem którego w pierwszej kolejności jest nadanie prostego interfejsu użycia funkcji obiektu oraz w razie potrzeby adaptacja interfejsu klasy do potrzeb bieżącego użytkownika.

* Strona architekta:

W zależności od poziomu kompetencji programistów decyduje jak bardzo trzeba uprościć wykorzystanie obiektów.

Pozwala ponownie użyć realizacji klas z nie kompatybilnym interfejsem, w taki sposób zmniejszając koszt projektu.

* Strona programisty-realizatora:

Programista wykorzystuje zrozumiały dla niego interfejs dostępny na wywoływanym obiekcie.

Kiedy użyć:

* Kiedy potrzebujemy uprościć interfejs użycia obiektu dla niekompetentnego programisty.
* Kiedy potrzebujemy zaadaptować interfejs już stworzonej klasy do bardziej pasującego w  danym przypadku użycia interfejsu.

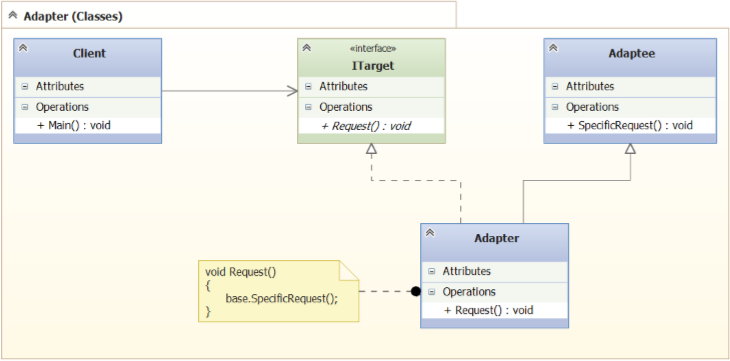
O ile w C# nie mamy wielokrotnego dziedziczenia, warto używać Adaptery poziomu obiektów ,czyli tworzyć egzemplarz klasy do adaptacji wewnątrz klasy-adptera zamiast uzyskania jej interfejsu dziedziczeniem.

Nazewnictwo:

Dobrym przykładem nazewnictwa jest dopisanie po nazwie klasy postfiksu Adapter.

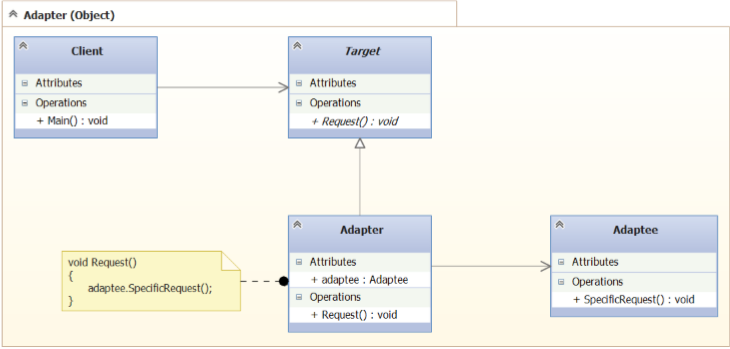
Wykorzystanie w .NET:

* System.Data.Common.DbDataAdapter
* System.Data.OleDb.OleDbDataAdapter
* System.Data.OracleClient.OracleDataAdapter
* System.Web.UI.Adapters.ControlAdapter
* System.Web.UI.Adapters.PageAdapter



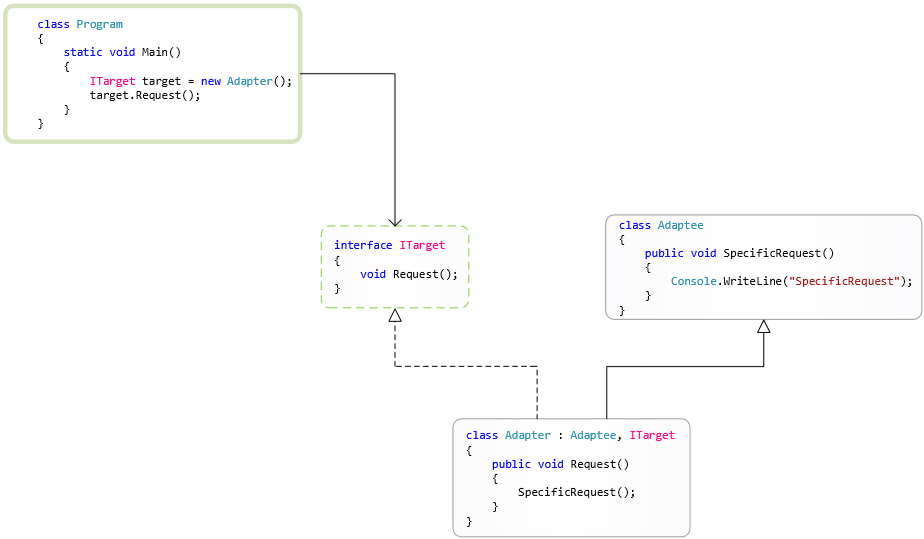
Rysunek 13. Struktura Adapteru poziomu klas w języku UML

Na Rysunku 13 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku Adapteru poziomu klas przy pomocy diagramu UML.



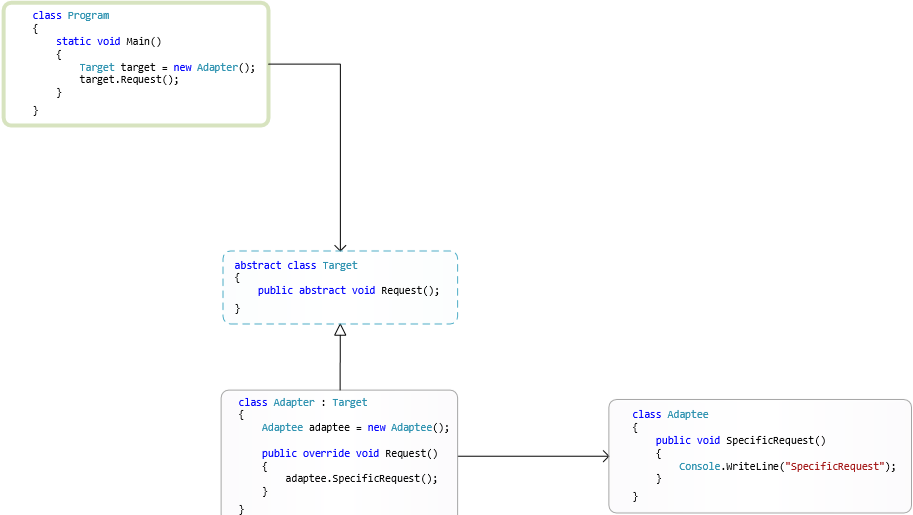
Rysunek 14. Struktura Adapteru poziomu obiektów w języku UML

Na Rysunku 14 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku Adapteru poziomu obiektów przy pomocy diagramu UML.



Rysunek 15. Struktura Adapteru poziomu klas w języku C#

Na Rysunku 15 widzimy szkielet Adapteru poziomu klas w języku C#.



Rysunek 16. Struktura Adapteru poziomu obiektów w języku C#

Na Rysunku 16 widzimy szkielet wzorca Adapter poziomu obiektów w języku C#.

## Facade

*Nazwa*: Facade

*Inne nazwy*: Glue

*Klasyfikacja*: według celu – strukturalny, według stosowania – obiektów

*Częstość użycia*: bardzo duża

*Przeznaczenie*: zwiększenie poziomu abstrakcji istniejącego interfejsu podsystemu lub systemu

Jest to bardzo ważny wzorzec, który używa się nie tylko do podsystemów ale również do architektury ogólnej systemu. W większości seminarów tematem których, jest tworzenie architektury aplikacji spotkamy ten wzorzec.

Dobrą metaforą facade może być interfejs samochodu. My jako użytkownicy mamy bardzo uproszczony interfejs relacji ze złożonym podsystemem (autem). Mamy 2 pedały oraz kierownice. Odpalając samochód nawiać nie musimy zastanawiać się nad złożonymi procesami, które dzieją się pod spodem. Inżyniery aut maksymalnie hermetyzują interfejsy podzespołów auta. Dziś zwykły człowiek nie musi wiedzieć nic o swoim aucie, żeby go komfortowo używać. Dokładnie ten efekt chcemy otrzymać używając facade.

We wzorcu tworzymy klasę, zawierającą instancje wszystkich podsystemów-klas oraz nowy interfejs, w którym używamy te podsystemy.

* Strona architekta:

Facade pozwala kontrolować złożoność interfejsów podzespołów systemu jak również całego systemu. Pozwala to przygotować odpowiedni do kompetencji programistów interfejs systemu w  wyniku czego obniżyć końcowy koszt projektu. Także przy pomocy tego wzorca architekt może wyeliminować zbędne związki obiektów (ang. *acomplish low coupling*).

* Strona programisty-realizatora:

Programista wykorzystuje przygotowany dla niego interfejs.

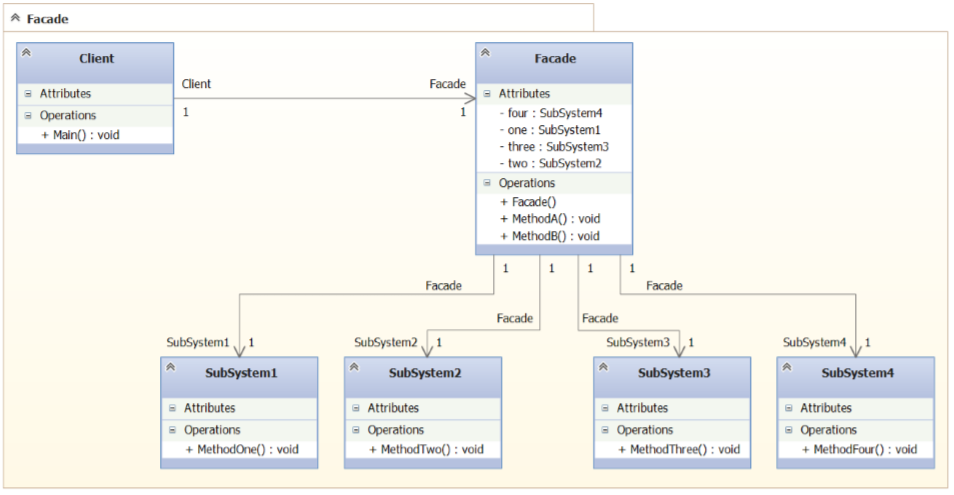
Programista nie zniszczy projekt popełniając błędy spowodowane zbędnymi zależnościami między obiektami.

Kiedy użyć:

* Kiedy potrzebujemy stworzyć prosty interfejs dla złożonego podsystemu.
* Kiedy trzeba zmniejszyć ilość związków
* Stworzenie warstw systemu

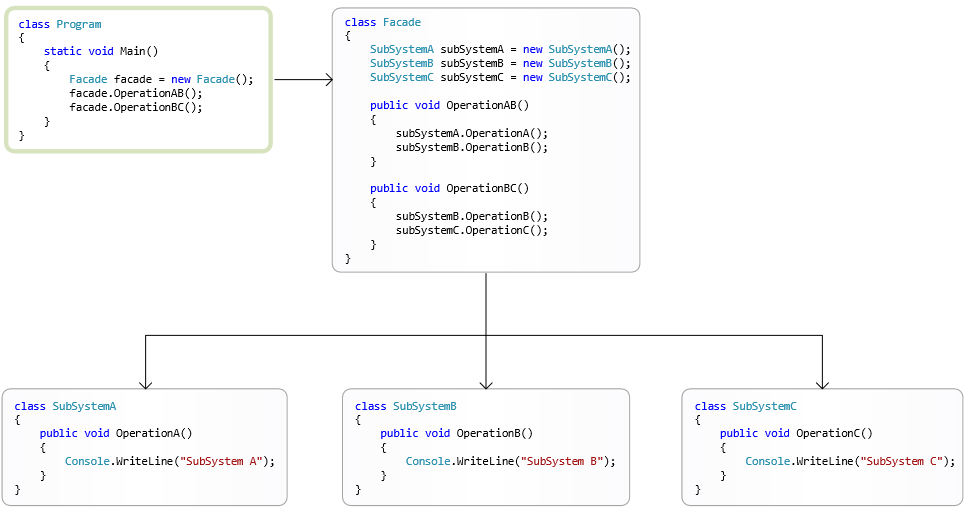
Wykorzystanie w .NET:

Wszędzie.



Rysunek 17. Struktura Facade w języku UML

Na Rysunku 17 możemy zobaczyć jak Facade składający się z czterech podsystemów przy pomocy diagramu UML.



Rysunek 18. Struktura Facade w języku C#

Na Rysunku 18 widzimy szkielet wzorca w języku C#.

## Composite

*Nazwa*: Composite

*Inne nazwy*: brak

*Klasyfikacja*: według celu – strukturalny, według stosowania – do obiektów

*Częstość użycia*: duża

*Przeznaczenie*: zbierać obiekty w dżewopodobne struktury.

Wzorzec Composite opisuje algorytm stworzenia drzew obiektów. Drzewo jest bardzo użyteczną strukturą danych w świecie programistycznym. Drzewo zawiera trzy główne składniki: korzeń-gałęź, gałęź, liście.

* Korzeń jest specjalnym rodzajem gałęzi, z której zaczyna się drzewo.
* Z gałęzi wychodzą nowe gałęzie oraz liście.
* Liście – elementy końcowe drzewa.

Każdy programista potrafi stworzyć drzewo, więc czasami jest trudno zrozumieć po co używać Composite. Composite opisuje zestaw reguł używanych dla budowy drzewa, jest to  ważne, kiedy będziemy chcieli stosować rekursywną kompozycję (wzorzec Interpretator), a także rekursywne obejście drzewa.

Ważną cechą wzorca jest to, że klient nie powinien wiedzieć, z jaką częścią drzewa pracuje: gałęzią czy liściem. Żeby tak się stało stosuje się klasa abstraktowa Component, która musi zawierać jak najwięcej wspólnych operacji dla liścia oraz gałęzi.

* Strona architekta:

Pozwala zbudować dżewopodobną strukturę danych hermetyzując elementy drzewa.

* Strona programisty-realizatora:

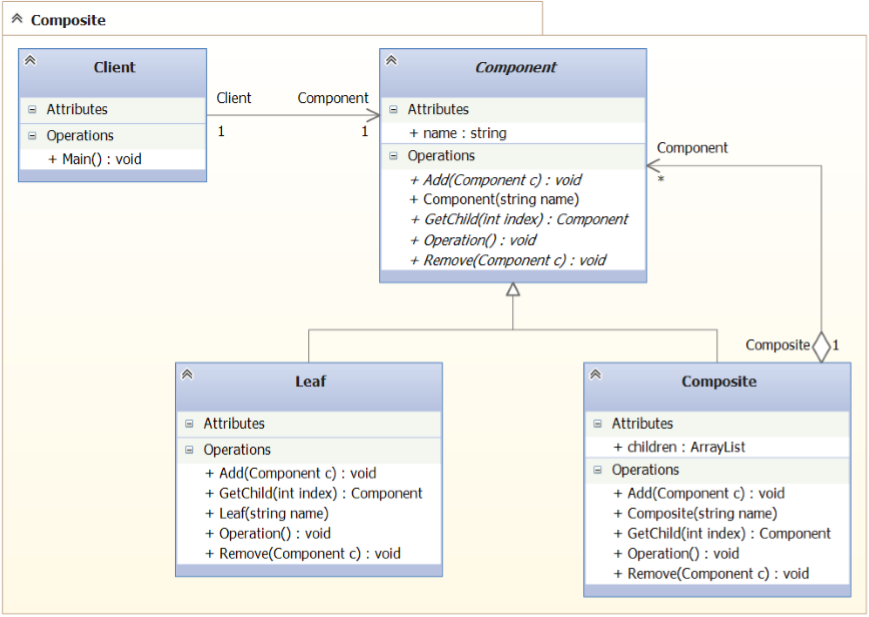
Programista nie musi wiedzieć, z jaką częścią drzewa pracuje

Kiedy użyć:

Kiedy potrzebujemy stworzyć dżewopodobną strukturę lub heterogeną hierarchię obiektów.

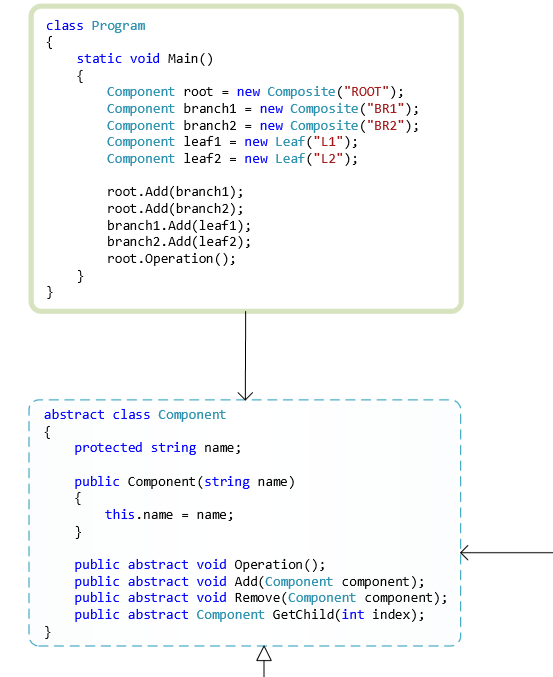
Wykorzystanie w .NET:

* System.Web.UI.WebControls.CompositeControl
* System.Windows.Data.CompositeCollection
* System.Web.UI.Design.WebControls.TreeViewDesigner
* System.Web.UI.WebControls.TreeNodeCollection
* System.Web.UI.WebControls.TreeView
* System.Windows.Controls.TreeView

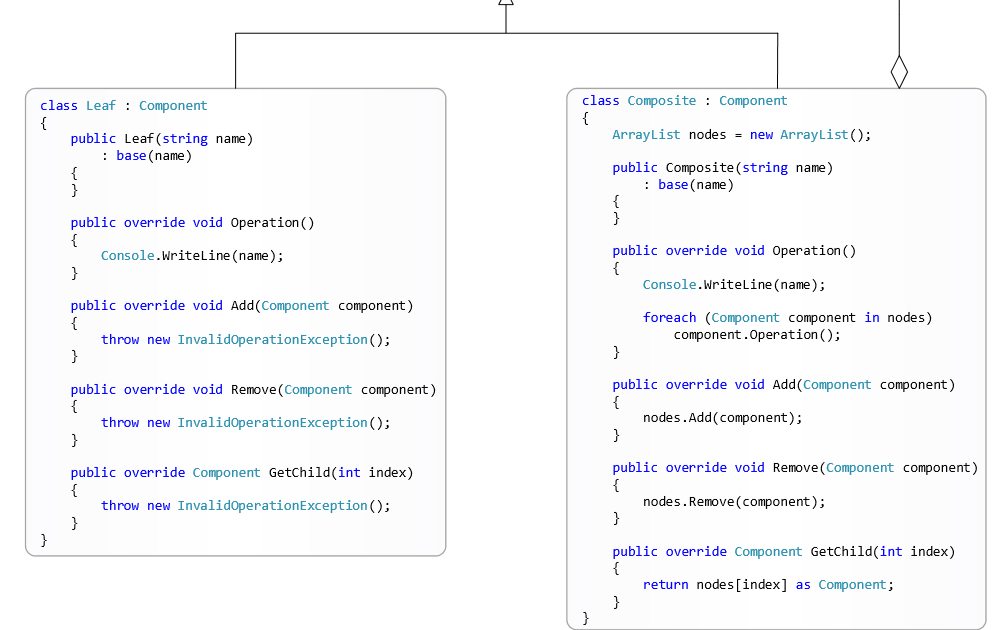


Rysunek 19. Struktura Composite w języku UML

Na Rysunku 19 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 20. Struktura Composite w języku C#



Rysunek 21. Struktura Composite w języku C#

Na Rysunkach 20 oraz 21 widzimy szkielet wzorca w języku C#.

## Proxy

*Nazwa*: Proxy

*Inne nazwy*: Surrogate

*Klasyfikacja*: według celu – strukturalny, według stosowania – do obiektów

*Częstość użycia*: duża

*Przeznaczenie*: nadać zastępczy obiekt

Dla rozumienia tego wzorca potrzebna nam jest metafora. Wyobraźmy, że mamy robota, który jest całkowitą naszą kopią i mamy możliwość go kontrolować. W tym przypadku wszystkie działania możemy wykonywać nie sobą a tylko tym robotem. Plusem takiego podejścia jest to, że jeżeli robot, którego kontrolujemy trafi pod samochód, nam tak naprawdę nic się nie stanie. Możemy w takim przypadku zrobić nowego robota lub wyjść na ulicę osobno (o ile mamy z robotem wspólny interfejs). Taki, czasami poręczny model możemy realizować przy pomocy wzorca Proxy.

W tym wzorcu jest dwie główne części realizacji – dziedziczenie wspólnego interfejsu dwuma klasami – operatorem oraz surogatem oraz przechowywanie w klasie surogatu referencji na operatora. Używając tej referencji będziemy mogli komunikować z operatorem.

Kiedy użyć:

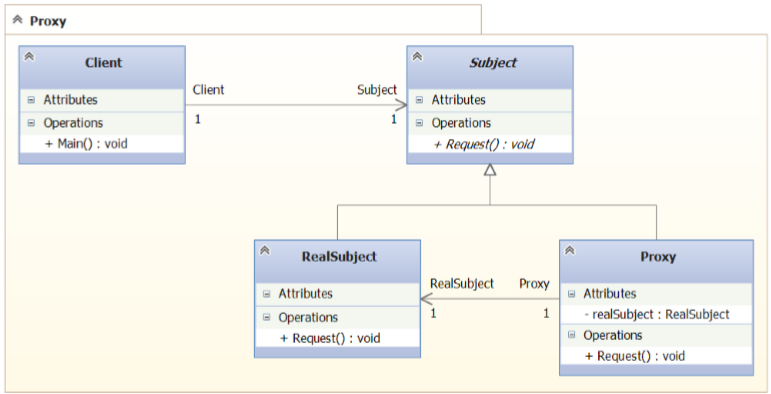
* kiedy trzeba przedstawić obiekt w innym środowisku (delegacja) WCF
* tworzenie ciężkich obiektów w zależności od potrzeby
* zastępowanie obronne, prawa dostępu

Nazewnictwo:

Warto używać sufiksu ‘Proxy’ w klasach-surogatach

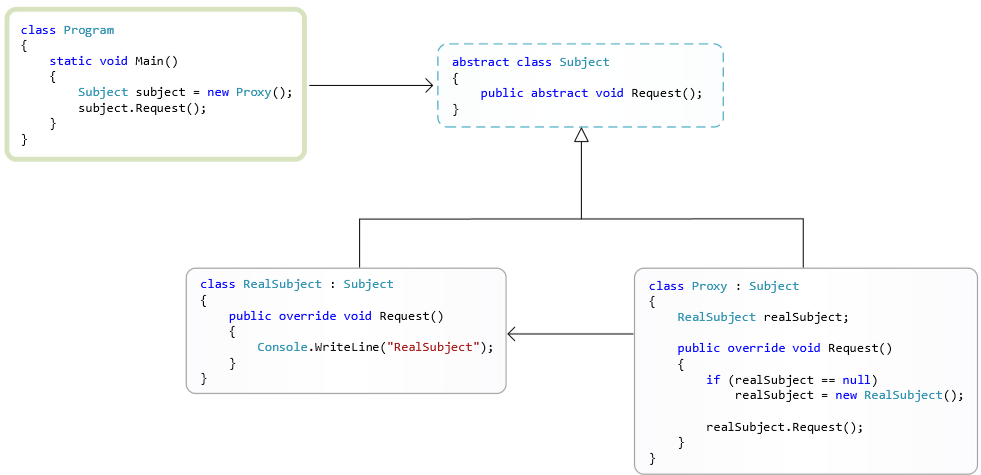
Wykorzystanie w .NET

* Microsoft.Build.Tasks.Deployment.ManifestUtilities.ProxyStub
* System.Runtime.Remoting.Proxies.ProxyAttribute
* System.Web.Script.Services.ProxyGenerator
* System.Web.UI.WebControls.WebParts.ProxyWebPart
* System.Net.Configuration.DefaultProxySection
* System.Net.GlobalProxySelection
* System.Net.IWebProxy



Rysunek 22. Struktura Proxy w języku UML

Na Rysunku 22 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 23. Struktura Proxy w języku C#

Na Rysunku 23 widzimy szkielet wzorca w języku C#.

## Command

*Nazwa*: Command

*Inne nazwy*: Action, Transaction

*Klasyfikacja*: według celu – czynnościowy, według stosowania – obiektów

*Częstość użycia*: duża

*Przeznaczenie*: przedstawić request w wyglądzie obiektu

Wzorzec Command służy do przedstawienia żądania w wyglądzie obiektu co pozwala na  konfigurację żądania klientem, kolejkowanie żądań-obiektów oraz wspieranie rezygnacji operacji.

Dobrym przykładem metaforycznym jest złożenie zamówienia w restauracji, gdzie osoba obsługująca klienta zapisuje na papierku zamówienie (konfiguruje obiekt) a potem go odkłada do  pozostałych. W tym przypadku możemy zrezygnować z zamówienia, dopóki leży ono w stopce innych.

Do wywołania komend wykorzystuje się obiekt-invoker, który także służy to przechowania konkretnych komend. Obiek-invoker potem przekazuje komendę do Recievera – obiektu, który musi obsłużyć obiekt-request, czyli komendę.

* Strona architekta:

Architekt tworzy niezbędny interfejs wzorca, aczkolwiek w zależności od kompetencji może być to zlecone deweloperowi.

* Strona programisty-realizatora:

Programista realizuje konkretne instancje obiektów-komend

Kiedy użyć:

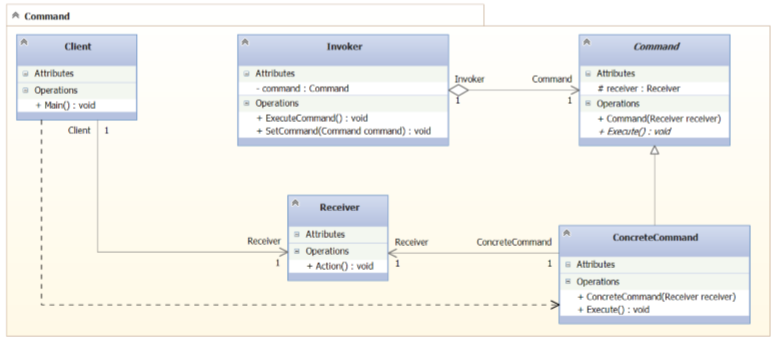
* Undo & Redo
* Kolejkowanie żądań w czasie
* Logowanie zmian

Nazewnictwo:

Dobrym przykładem nazewnictwa jest dopisanie po nazwie klasy-komendy postfiksu Command a także nazwanie wywołującej komendy klasy ‘Invoker’.

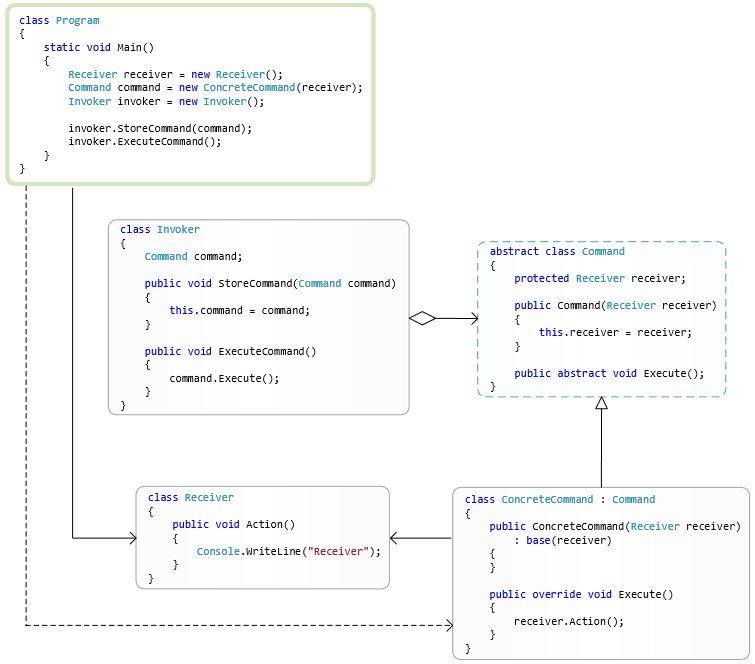
Wykorzystanie w .NET:

* System.Windows.Input.MediaCommands
* System.Windows.Input.NavigationCommands
* System.Windows.Input.RoutedCommand
* System.Windows.SystemCommands
* System.Workflow.ComponentModel.Design.WorkflowMenuCommands



Rysunek 24. Struktura Command w języku UML

Na Rysunku 24 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 25. Struktura Command w języku C#

Na Rysunku 25 widzimy szkielet wzorca w języku C#.

## Iterator

*Nazwa*: Iterator

*Inne nazwy*: brak

*Klasyfikacja*: według celu – czynnościowy, według stosowania – obiektów

*Częstość użycia*: bardzo duża

*Przeznaczenie*: zapewnić obiekt-iterator dla dostępu do kolekcji

Iterator - to wzorzec, a lepiej powiedzieć obiekt który gwarantuje bezpieczność dostępu do  elementu kolekcji.

Dobrą metaforą będzie bank czyli kolekcja pieniędzy. W danym przypadku iteratorem będzie osoba obsługująca przy kasie. Taka osoba nie pozwoli nam wziąć nie swoje pieniądze, czyli ograniczy dostęp do elementów kolekcji.

Trzeba rozumieć, że iterator nie tylko używamy do owinięcia łatwych tablic lub wektorów, a  też przy pracy ze złożonymi obiektami. Nie wolno dawać możliwość developerowi pracować ze  złożonymi konstrukcjami a tylko z pięknymi obiektami-wraperami. Developer prosi element, a  już iterator wie gdzie pójść i jak dostać potrzebny element.

Jeżeli potrzebujemy stworzyć iterator który będzie działał na różnych kolekcjach, możemy użyć technikę tworzenia polimorfnego iteratora.

* Strona architekta:

Zastosowanie iteratora pozwoli ukryć złożone obiekty-elementy kolekcji oraz otrzymać bezpieczny dostęp do elementów.

* Strona programisty-realizatora:

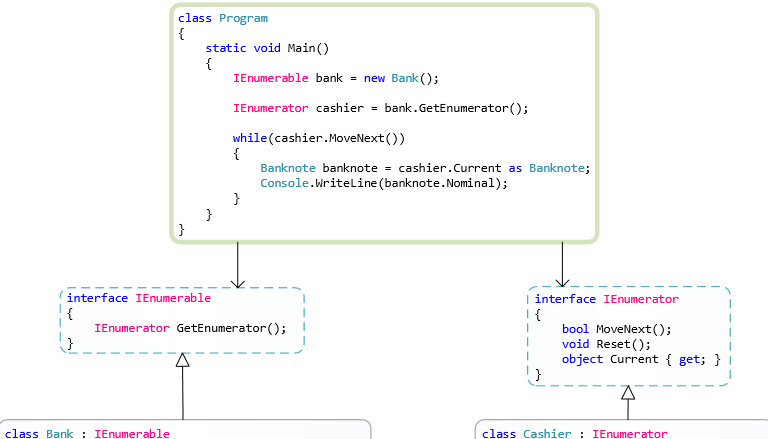
Programista wykorzystuje przygotowany dla niego mechanizm przy pracy z kolekcją.

Kiedy użyć:

* kiedy potrzebujemy poręczny oraz bezpieczny sposób dostępu do elementów kolekcji.
* kiedy potrzebujemy mieć kilka sposobów iterowania po kolekcji
* dla polimorficznej iteracji

C# osobliwości:

Wzorzec Iterator jest ekskludowany z .NET o ile Microsoft daje alternatywny wzorzec, nazwijmy go ‘Enumerator’. Jako implementację tego wzorca w .NET wykorzystujemy implementację dwóch najważniejszych interfejsów wbudowanych w .NET: Ienumerable oraz Ienumerator. Także mamy ICollection, IList oraz inne interfejsy. Ienumerable oraz Ienumerator – najbardziej prymitywne interfejsy do zadania interfejsu złożonego obiektu kolekcji



Rysunek 26. Implementacja Iteratora w języku C#

Na Rysunku 26 widzimy implementację iteratora w języku C# poprzez interfejsy IEnumerable oraz IEnumerator.

## Observer

*Nazwa*: Observer

*Inne nazwy*: Dependents, Publisher-Subscriber

*Klasyfikacja*: według celu – czynnościowy, według stosowania – obiektów

*Częstość użycia*: bardzo duża

*Przeznaczenie*: opisać technikę publisher-subscriber

Wzorzec Observer służy do realizacji mechanizmu zdarzeń. Są 2 modele: model pushowania – zdarzenie nie da się zignorować oraz wyciągnięcia – zależy od developera czy chcemy reagować na  zdarzenie. Observer w .NET jest zamieszczony nowym mechanizmem, więcej informacji w  podrozdziale C# osobliwości.

* Strona architekta:

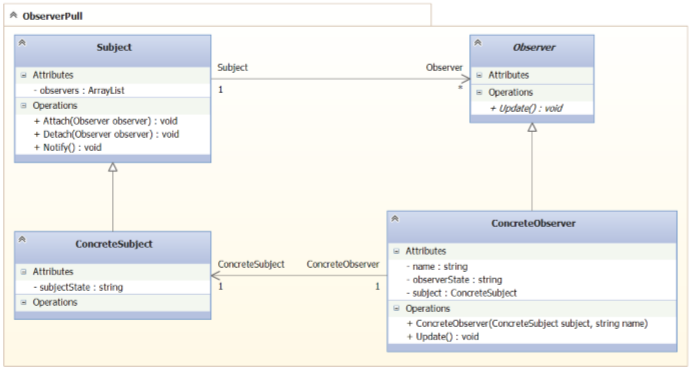
Zwykle rezygnuje z użycia tego wzorca na korzyść wbudowanego w .NET mechanizmu zdarzeń.

Kiedy użyć:

* Kiedy potrzebujemy realizować mechanizm zdarzeń

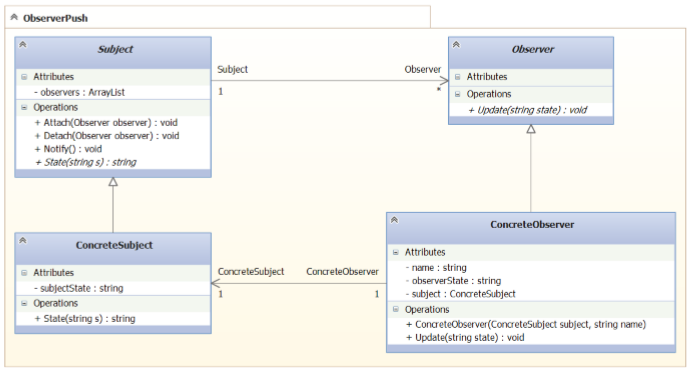
C# osobliwości:

Ten wzorzec leży w fundamencie modelu zdarzeń .NET, realizowany przy pomocy delegatów. Inne języki, które nie posiadają mechanizmu delegatów oraz nie mogą sobie pozwolić zaimplementować ten mechanizm technicznie lub finansowo zmuszają programistę do  implementowania tego wzorca ręcznie. Rekomendującym jest wykorzystanie .NET podejścia do  realizacji mechanizmu zdarzeń. Czyli zamiast tworzenia złożonych obiektów w .NET podpisujemy się na zdarzenia przy pomocy delegatów.



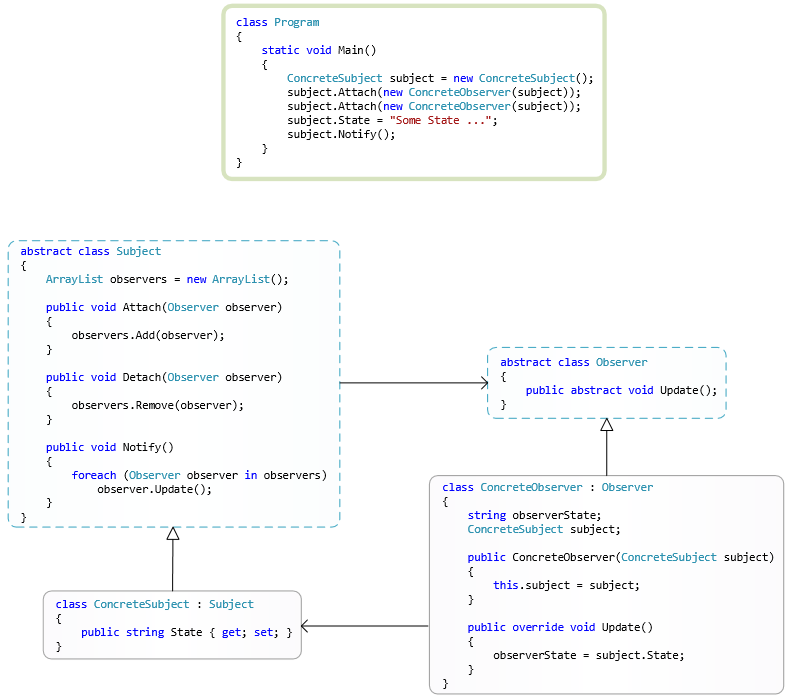
Rysunek 27. Struktura Observer modelu Pull w języku UML

Na Rysunku 27 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku modelu Pull przy pomocy diagramu UML.



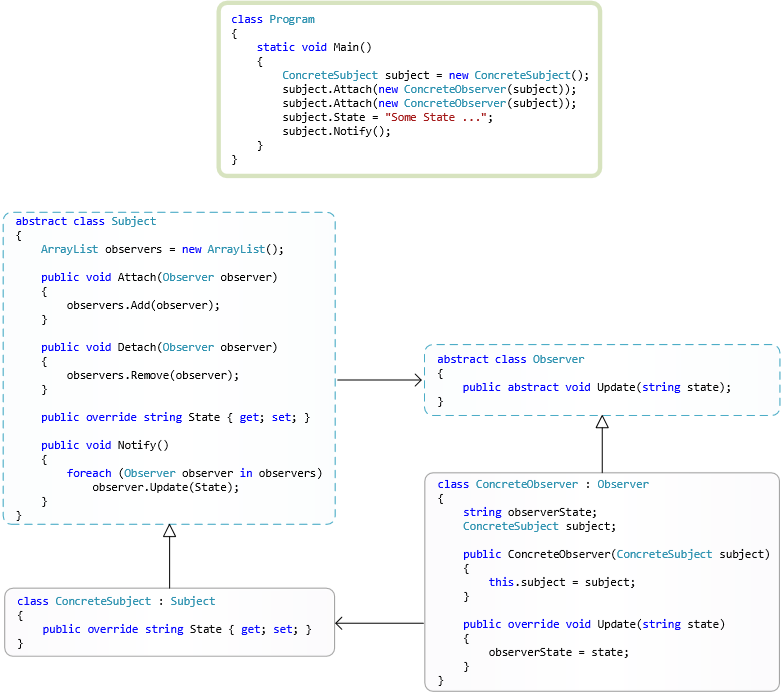
Rysunek 28. Struktura Observer modelu Push w języku UML

Na Rysunku 28 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku modelu push przy pomocy diagramu UML.



Rysunek 29. Struktura Observer modelu Pull w języku C#

Na Rysunku 29 widzimy szkielet wzorca Observer modelu Pull w języku C#.



Rysunek 30. Struktura Observer modelu push w języku C#

Na Rysunku 30 widzimy szkielet wzorca Observer modelu push w języku C#.

## Strategy

*Nazwa*: Strategy

*Inne nazwy*: Policy

*Klasyfikacja*: według celu – czynnościowy, według stosowania – do obiektów

*Częstość użycia*: duża

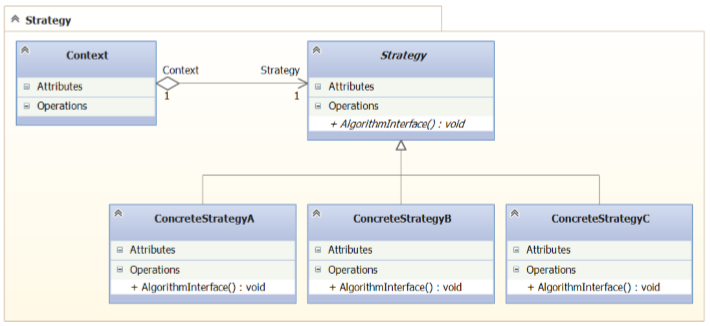
*Przeznaczenie*: podmiana algorytmów-strategii

Wzorzec Strategy pozwala wybierać odpowiednią do sytuacji algorytm-strategię. Minusem tego wzorca jest to, że klient musi wiedzieć wewnętrzną realizacje klas-strategii, żeby dokonać odpowiedniego, poprawnego wyboru.

Plusem wzorca jest hermetyzacja algorytmów w metodach klas oraz możliwość podmiany algorytmu bez względu na klienta. Trzeba również pamiętać, że wybór poprawnej strategii może mieć krytyczny wpływ na działanie systemu. Wzorzec ten także pomaga efektywnie wykorzystywać resursy systemu.

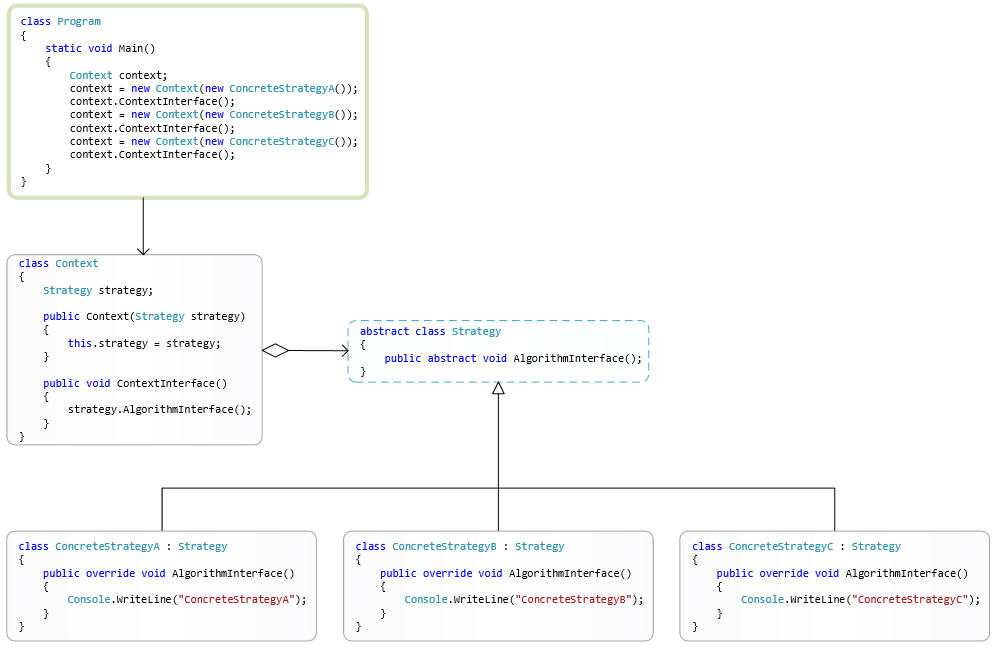
Kiedy użyć:

* kiedy potrzebujemy mieć kilka rodzajów jednego algorytmu: sortowanie, czytanie z pliku i  t.d.
* kiedy mamy kilka klas należących do jednej rodziny różniących się tylko zachowaniem.



Rysunek 31. Struktura Strategy w języku UML

Na Rysunku 31 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



Rysunek 32. Struktura Strategy w języku C#

Na Rysunku 32 widzimy szkielet wzorca w języku C#.

# Rozdział 3. Stworzenie APLIKACJI WEBOWEJ Z UŻYCIEM WZORCÓW



## Opis aplikacji

‘SPA’ aplikacja Finances została napisana głównie w językach C# oraz TypeScript przy użyciu platformy .NET Core oraz zestawu najnowszych na daną chwilę technologii Google oraz Microsoft.

Zostały wyodrębnione następujące przypadki użycia:

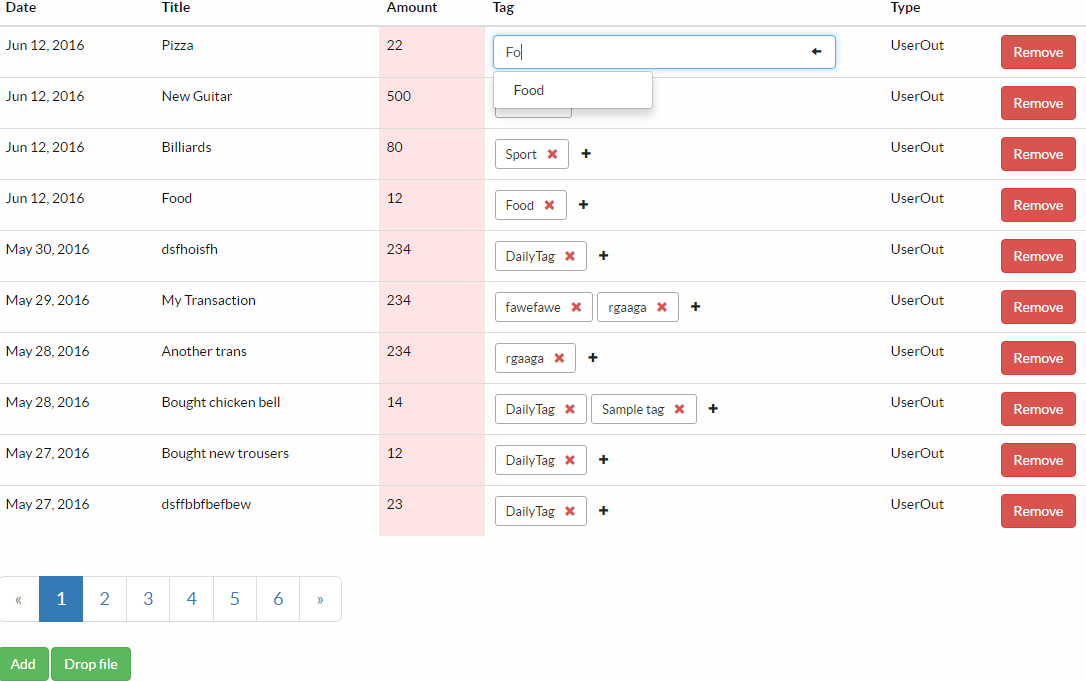
* Logowanie do systemu oraz wylogowanie się z systemu.
* Rejestracja użytkownika w systemie.
* Wczytywanie transakcji użytkownika z .csv pliku, ściąganego z mBanku.
* Możliwość ręcznego dodawania oraz usunięcia transakcji.
* Tworzenie tagów z limitowanymi miesięcznymi wydatkami oraz z możliwością śledzenia tagu na stronie codziennych wydatków.
* Możliwość podpięcia tagów do transakcji z możliwością momentalnego wyszukiwania tagu.
* Dashboard odpowiadający za śledzenie oraz analizę wydatków według odpowiedniego periodu czasu. Interaktywne wykresy na tych podstronach zezwalają nie tylko zobaczyć wydatki za odpowiedni period ale też przejść do szczegół wydatków tego periodu czasu.
* Strona ustawień zezwala zmienić dane użytkownika oraz obecne hasło.
* Strona informacyjna aplikacji.

Autoryzacja użytkownika w aplikacji jest obowiązkowa, żeby zidentyfikować transakcje użytkownika.

Spis użytych technologii:

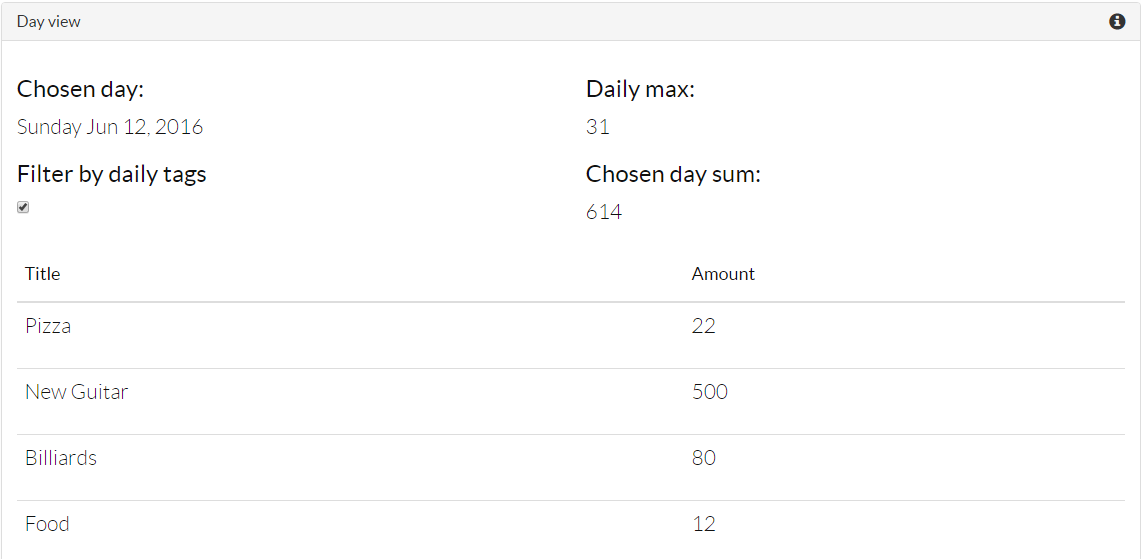
* HTML, SASS, TypeScript
* Bootstrap 3, Angular 2 beta 9, Observables
* Lato fonts, GlyphIcons
* Ng2-Charts
* C#, .NET
* ASP.NET Core 1.0 RC1
* Entity Framework Core 1.0 RC1
* Identity Framework 3.0

Na poniższych rysunkach prezentują się główne elementy aplikacji Finances.



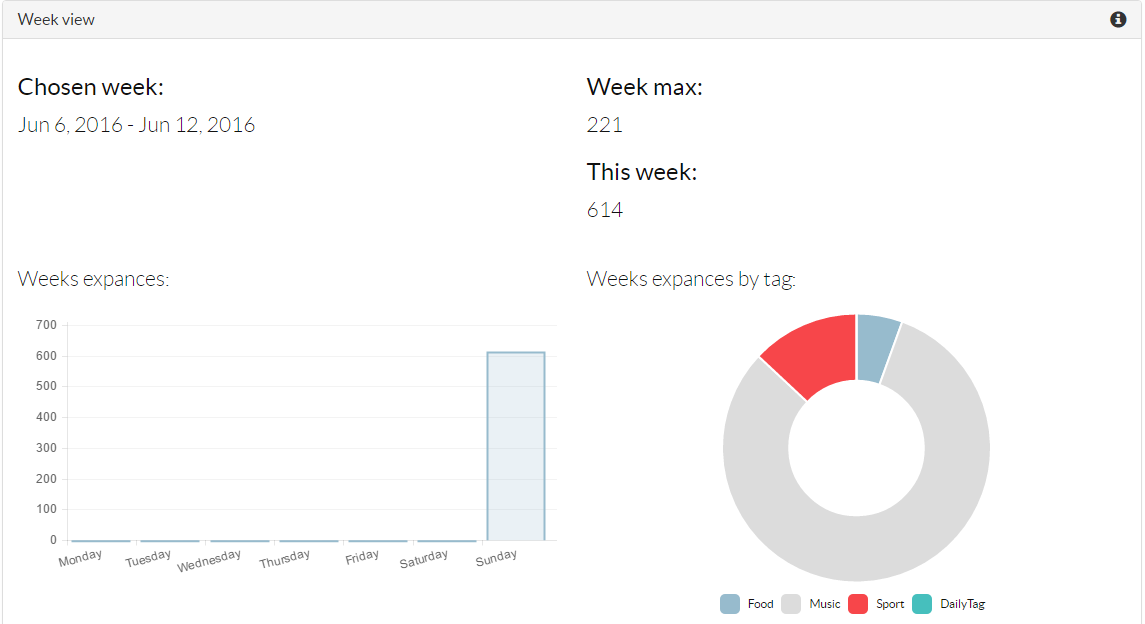
Rysunek 33. Wyszukiwanie oraz podpięcie tagu do transakcji

Na Rysunku 33 widzimy stworzyny przy użyciu Angular 2 grid, zawierający wszystkie transakcję. Do każdej transakcji możemy podpiąć tag, który możemy wyszukać przy pomocy dynamicznej wyszukiwarki.



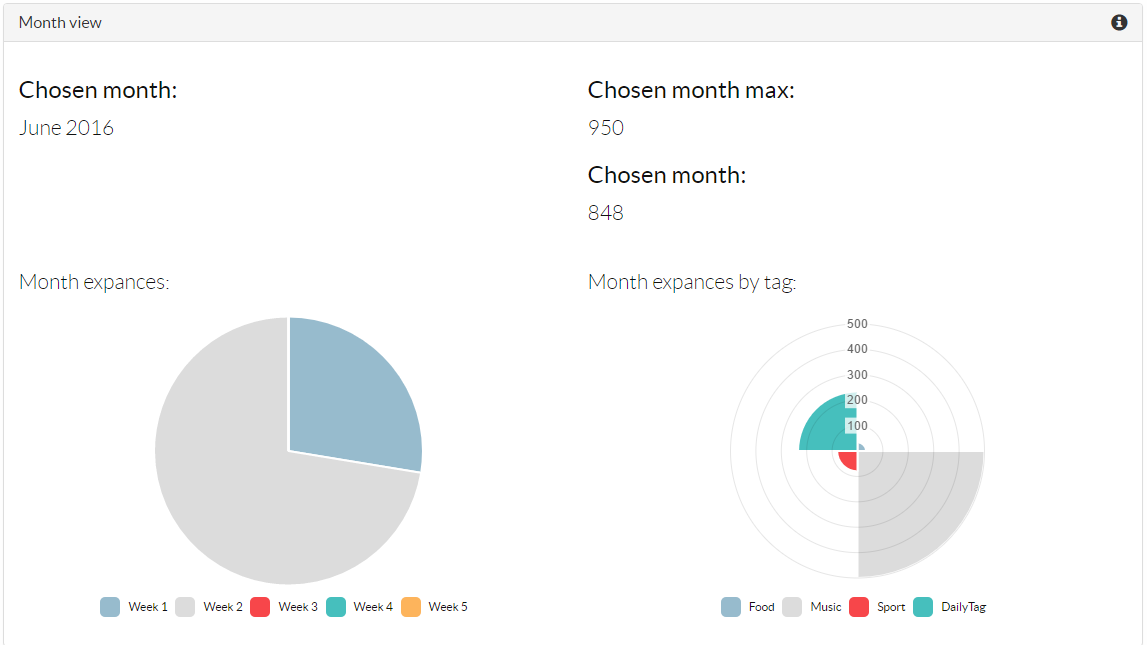
Rysunek 34. Dashboard. Widok pojedynczego dnia

Na Rysunku 34 widzimy dashboard z włączonym widokiem pojedynczego dnia. W danej chwili widzimy tylko wydatki oraz filtrujemy je według typu tagu podpiętego pod transakcję.



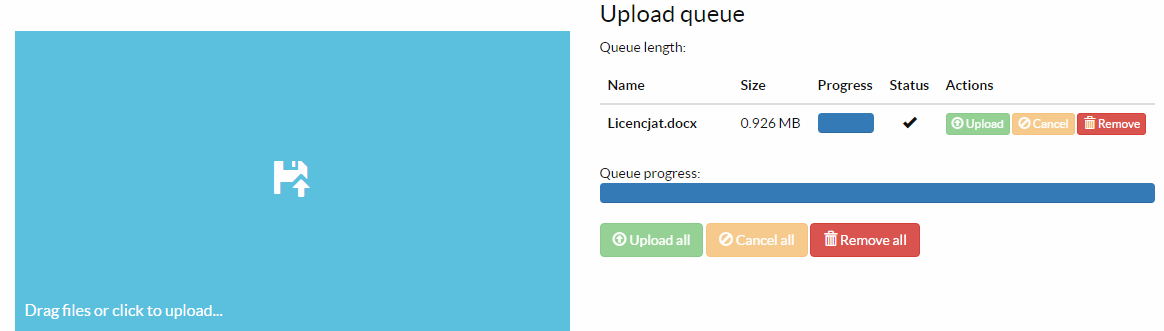
Rysunek 35. Dashboard. Widok tygodnia

Na Rysunku 35 widzimy dashboard z włączonym widokiem tygodnia. Widzimy wykres wydatków za każdy pojedynczy dzień oraz wykres wydatków według tagu. Także mamy sumę wydatków tygodniowych, które możemy porównać do ustawionej kwoty maksymalnej. Klikając na słupek dowolnego dnia, przechodzimy na widok danego dnia.



Rysunek 36. Dashboard. Widok miesięca

Na Rysunku 36 widzimy widok bieżącego miesięca. Tu na wykrasach widzimy wydatki według tygodnia oraz wydatki według tagu. Klikając na odpowiedni tydzień na wykresie, przechodzimy na widok wybranego tygodnia.

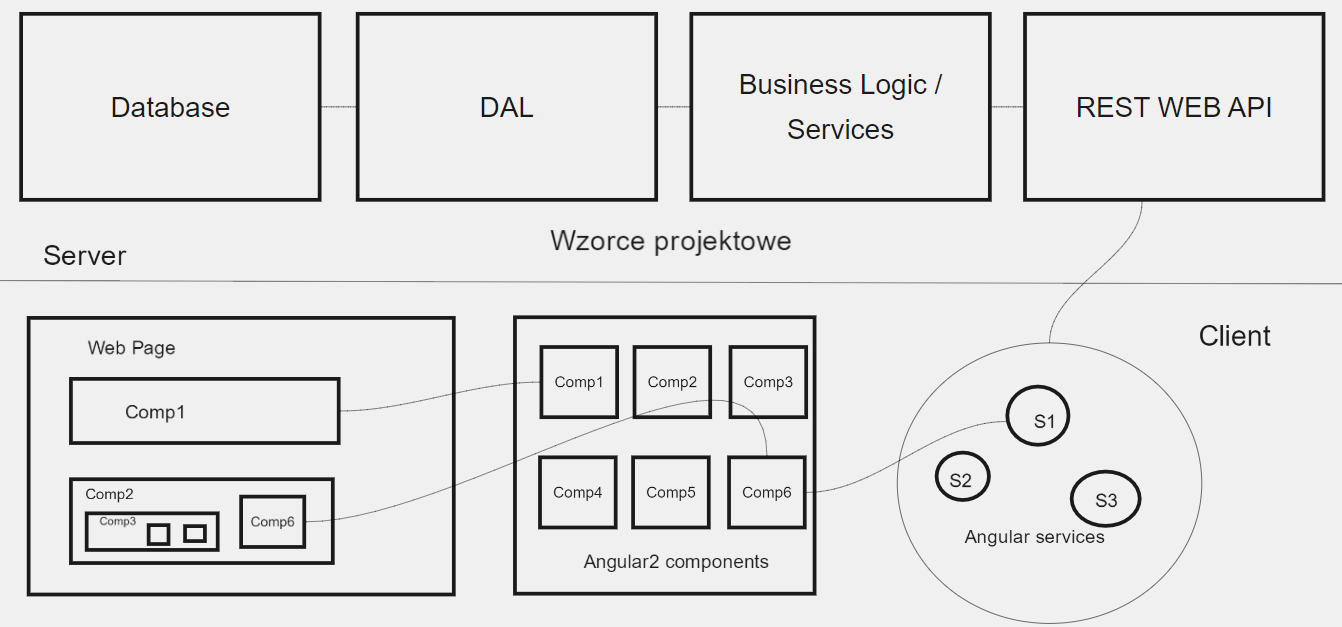


Rysunek 37. Komponent ładowania plików transakcji bankowych

Na Rysunku 37 widzimy komponent odpowiedzialny za ładowanie .csv plików z transakcjami bankowymi. Możemy ładować kylka plików na raz oraz używać drag and drop. Też widać progres ładowania pliku. Jest możliwość momentalnego usunięcia pliku z serwera po naciśnięciu przycisku Remove.

## Warstwy aplikacji

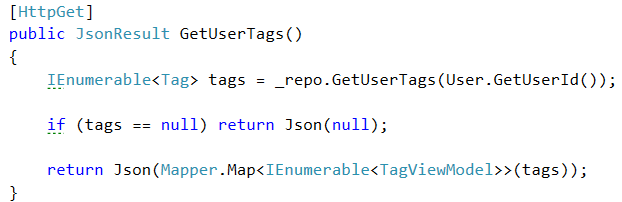
Na poniższym rysunku można zobaczyć warstwy aplikacji Finances. Aplikacja składa się z dwóch części – strony klienta oraz serwerowej.



Rysunek 38. Warstwy aplikacji Finances

Po stronie klienta mamy html stronę, która składa się z poszczególnych komponentów do których są podpięte klasy typescriptowe, odpowiadające za logikę danego komponentu. Komponent może zawierać w sobie inne komponenty. Wewnątrz siebie komponent ma możliwość używać serwisy, czyli logikę współdzieloną. W Finances głównym serwisem jest http service, celem którego jest wysyłanie danych do serwisów restowych uruchomionych na serwerze.

Po stronie serwera mamy mechanizm ASP 5 WEB API, który przyjmuje http zapytania ze strony klienta oraz wysyła odpowiednią do zapytania odpowiedź. W najprostszym przypadku, po otrzymaniu zapytania mechanizm restowy używa wprost DAL, w sercu którego leży wzorzec repository, który obrabia oraz zwraca dane z bazy danych. Także w rest serwisie odbywa się mapowanie encji modelu bazy danych do view modelu [9].



Rysunek 39. Najprostszy Action w kontrolerze restowym

Też są bardziej złożone ścieżki po stronie serwera, na przykład użycie biznes logiki oraz serwisów, które już będą odpowiedzialne za komunikację z DAL.

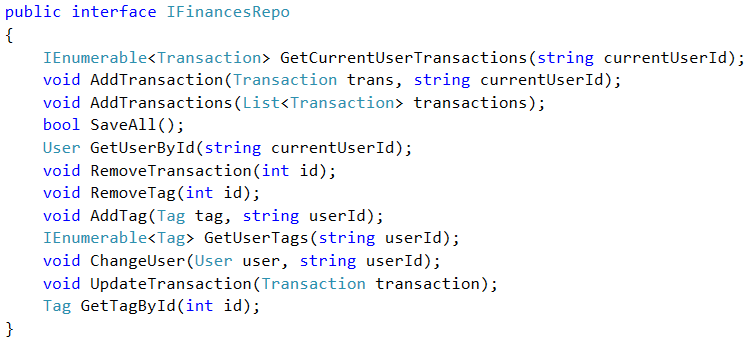


Rysunek 40. Złożony Action w kontrolerze restowym

## Szczegóły implementacji

O ile ASP.NET Core już zawiera wszystkie niezbędne mechanizmy ułatwiające projektowanie aplikacji, dość niełatwym zadaniem okazało się odnalezienie miejsc, gdzie można by było sensownie wstrzyknąć wzorce projektowe. W aplikacji był użyty Facade, żeby przedstawić poręczny interfejs zarządzania bazą danych. W danym przypadku jest to też tak zwany repository pattern, o ile przedstawia sobą jedną klasę do zarządzania danymi.

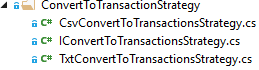
Interfejs klasy wygląda następująco:



Rysunek 41. Interfejs IFinancesRepo

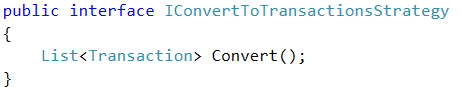
Dany mechanizm wykorzystują rest serwisy, serwisy zwykłe oraz logika biznesowa aplikacji. Także w aplikacji został użyty wzorzec strategy, do zarządzania ładowaniem plików. W taki sposób tworzymy różne strategie dla ładowania plików .csv, .txt oraz innych rozszerzeń.

Do konwertowania używamy oddzielnych klas, które dziedziczą oraz realizują interfejs IConvertToTransactionStrategy:



Rysunek 42. Struktura plików według użytego wzorca

Interfejs strategii jest bardzo prosty:



Rysunek 43. Interfejs strategii

Także w aplikacji był użyty mechanizm dependency injection, który pozwala nie tworzyć ręcznie egzemplarze konkretnych klas.

Warto zauważyć, że ciągle podczas tworzenia aplikacji mi towarzyszą już gotowe, microsoftowe, implementacje wzorców, jako mechanizmy platformy. Przykładem może być standardowy interfejs IEnumerable oraz mechanizm delegatów.

# Zakończenie

Podsumowując swoją pracę, mam nadzieje, że zrealizowałem postawiony na początku cel. W  pierwszym rozdziale zrobiłem krótki opis .Net oraz C#, także opisałem podstawy programowania obiektowego oraz SOLID. W końcu pierwszego rozdziału rozpatrzyłem konstrukcje języku UML. Rozdział drugi poświęciłem opisaniu najczęściej używanych w .NET wzorców projektowych. Każdy wzorzec był także opisany w języku C# oraz UML. W ostatnim rozdziale opowiedziałem o swojej aplikacji, która używa najbardziej nowoczesnych technologii Google oraz Microsoft na dany moment. Także opowiedziałem o wzorcach projektowych, które używałem przy tworzeniu aplikacji.

Starałem się, aby z dwudziestu trzech wzorców opisanych dekady temu wybrać te  najważniejsze, najczęściej używane wzorce projektowe, który każdy deweloper musi brać pod uwagę pod czas pracy nad projektem programistycznym. Jestem bardzo wdzięczny za pomoc doktora Andrzeja Bobyka, który pomógł mi ściśle określić cel pracy. Dzięki tej pracy zdobyłem doświadczenie, które będę używał na co dzień w pracy nad złożonymi projektami w przyszłości.

# Bibliografia

1. A. Shevchuk, D Okhrimenko, A. Kasyanow - *Design Patterns Via C#*, Ebook, 2015.
2. J. Richter - *CLR Via C# (4ht Edition)*, Microsoft Press, 2012.
3. S. McConnell, *Code complete*, Microsoft Press, 2004.
4. K. Cwalina, B. Abrams, *Framework Design Guidelines*, Microsoft Press, 2005.
5. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns. Elements of Reusable Object-oriented Software*, Addisson-Wesley Proffessional Computing Series, 1995.
6. T.Wright, *Learning JavaScript. A Hands-On Guide to the Fundamentals of Modern JavaScript*, Wydawnictwo Addison-Wesley, 2012.
7. A. Troelsen, *C# 6.0 and the .NET 4.6 Framework*, Apress, 2015.
8. S. Millett, [*Professional ASP.NET Design Patterns*](http://shop.oreilly.com/product/9780470292785.do), O'Reilly, 2010.
9. F.Muhammad, *Real World ASP.NET Best Practices*, Apress, 2003.
10. M.J. Price, [*C# 6 and .NET Core 1.0: Modern Cross-Platform Development*](http://shop.oreilly.com/product/9781785285691.do), Apress, 2016.
11. B. Joshi, *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*, Apress, 2016.
12. R. Ferguson, *Javascript Recipes*, Apress, 2016.
13. A. Prabhu, *Beginning CSS Preprocessors with SASS, Compass and Less*, Apress, 2015.
14. S. Fenton, *Pro TypeScript*, Apress, 2014.
15. *ASP.NET Core Documentation*: https://docs.asp.net/en/latest/
16. D. Esposito, *Modern Web Development: Understanding domains, technologies, and user experience*, Microsoft Press, 2015.