

**Katolicki Uniwersytet Lubelski Jana Pawła II**  
**Wydział Matematyki, Informatyki i Architektury Krajobrazu**  
**Instytut Matematyki i Informatyki**

Informatyka,  
Studia stacjonarne I stopnia

**Vladyslav Verenich**

Nr albumu 134439

**Wzorce projektowe w języku C#**

Praca licencjacka  
napisana na seminarium aplikacje sieciowe i bazy danych pod kierunkiem  
**dr. Andrzeja Bobyka**

Lublin 2016



## Spis treści

WSTĘP.....	5
ROZDZIAŁ 1. ZAPOZNANIE Z POJĘCIAMI ORAZ TECHNOLOGIAMI.....	7
1.1. NET / C# .....	7
1.2. Programowanie obiektowe, paradygmaty .....	8
1.3. SOLID .....	9
1.4. UML.....	10
1.5. Problemy projektowania nowoczesnych systemów informatycznych.....	13
ROZDZIAŁ 2. WZORCE PROJEKTOWE.....	15
2.1. Pojęcie wzorca projektowego .....	15
2.2. Klasyfikacja wzorców .....	15
2.3. Factory Method .....	16
2.4. Abstract Factory .....	18
2.5. Singleton .....	21
2.6. Adapter .....	23
2.7. Facade .....	25
2.8. Composite .....	28
2.10. Proxy .....	31
2.11. Command.....	33
2.12. Iterator .....	35
2.13. Observer .....	37
2.14. Strategy .....	41
ROZDZIAŁ 3. STWORZENIE APLIKACJI WEBOWEJ Z UŻYCIEM WZORCÓW.....	45
3.1. Opis aplikacji .....	45
3.2. Warstwy aplikacji .....	48
3.3. Szczegóły implementacji .....	50
ZAKOŃCZENIE .....	53
BIBLIOGRAFIA .....	55



## Wstęp

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

Czasami nie zdajemy sobie sprawy o ważności świadomego wykorzystywania takich wzorców w codziennej pracy nad projektem. Bez nich byłoby trudno wyobrazić sobie możliwość tworzenia skomplikowanego nowoczesnego oprogramowania. Dla mnie osobiście zaskoczeniem było to, że niektórzy moi koledzy nie mogli nazwać przynajmniej trzech wzorców z katalogu GoF. Jest to bardzo zła tendencja, jako że brak znajomości wzorców projektowych w nowoczesnym projektowaniu rozwiązań informatycznych można przyrównać do braku znajomości całek w nowoczesnej matematyce oraz fizyce. Wzorce projektowe są bowiem nowym poziomem abstrakcji w informatyce, tak samo, jak programowanie obiektowe. Abstrakcja to pojęcie, które określa postęp nie tylko technologiczny, ale też całej ludzkości. Im większy jest poziom abstrakcji, tym bardziej złożony system można zbudować.

W rozdziale pierwszym dokonano wprowadzenia do pozostałych części pracy. Przedstawiono kroki, niezbędne do zrozumienia wzorców projektowych w języku programowania C#. W kolejnych podrozdziałach opisano platformę .NET i wyjaśniono główne stereotypy języka C#. Zaprezentowano pojęcia programowania obiektowego oraz SOLID, ponadto został dokonany wstęp do języka modelowania UML oraz przeanalizowano problemy, z którymi się spotyka architekt przy projektowaniu systemu informatycznego.

W drugim rozdziale skupiono się na wzorcach projektowych. Dokładnie wyjaśniono, czym jest wzorzec, a następnie sklasyfikowano wszystkie wzorce projektowe. Dalej przeanalizowano 11 najczęściej używanych wzorców projektowych w .NET oraz 2 wzorce architektoniczne.

W ostatnim rozdziale przedstawiono aplikację webową, napisaną z użyciem wzorców oraz najbardziej nowoczesnych na daną chwilę technologii.



## Rozdział 1. Zapoznanie z pojęciami oraz technologiami

### 1.1. NET / C#

CLI (ang. *Common Language Infrastructure*) jest to specyfikacja stworzona przez Microsoft, zawierająca:

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

.NET – realizację CLI działającą 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 Intermediate Language*). CIL tak samo jak C#, gdyż VB, C++ mają zaimplementowane w .NET według CLS swoje kompilatory. W wyniku kompilacji otrzymujemy kod bajtowy (zwany też bajt-kodem). Kod ten wykonuje się przez CLR przy pomocy mechanizmu JIT (ang. *Just In Time*). Także w wykonaniu uczestniczy FCL, o ile kod bajtowy zawiera wywołania metod, operacje na typach, które zawierają się w FCL.

Przykładowy ciąg uruchomienia aplikacji napisanej w języku C# przy użyciu .NET byłby zatem następujący: kod C# → **csc.exe** (kompilator C#) → **kod bajtowy** → .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 kod bajtowy możemy przekazać do dowolnej CLI, oczywiście, jeżeli będzie ona zainstalowana w systemie, na którym chcemy uruchomić program.

C# jest językiem obiektowym, stworzonym przez Microsoft dla platformy .NET. Został on zestandaryzowany jako ECMA-334 oraz ISO/IEC 23270 [7]. C# tak samo, jak Java oraz C++ należy do rodziny języków z C-podobną składnią. Anders Hejlsberg, twórca C#, połączył 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 co najważniejsze, łatwość realizacji. Pozwala to pisać aplikacje szybciej, optymalnie oraz z wyższą jakością. Nie ma dziś na świecie pod tym względem żadnego języka programowania lepszego od C#.

## 1.2. Programowanie obiektowe, paradygmaty

Programowanie obiektowe (ang. *Object-Oriented Programming*, OOP) bazuje się na dwóch głównych pojęciach: algorytmu oraz modelu. Algorytmizacja jest to napisanie 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 pozwalają tworzyć modeli [3].

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

Czym różni się klasa od typu? Wyobraźmy, że mamy 2 klasy: Dog, Cat, każdy ma operacje Eat() oraz Go(). W danej chwili klasy są różne, 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. Reusability

Hermetyzacja (ang. *Encapsulation*) – oznacza ukrycie informacji. Istnieje 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. Hermetyzacja wariacji formuje abstrakcję. Przykład: formowanie Dokumentu z Title, Body oraz Footer jako pól prywatnych. Dokument to zborowe pojęcie z części. Dokument nie istnieje, dokument – abstrakcja. Przy formowaniu abstrakcji hermetyzacja jest wtórną a abstrakcja pierwotną.

Dziedziczenie (ang. *Inheritance*) – możliwość zdefiniowania nowych klas 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 użyć jednego obiektu zamiast innego.

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

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

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

Powtórne użycie (ang. *Reusability*) – 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 *reusability* należy wykorzystanie bibliotek oraz frameworków.

### 1.3. SOLID

SOLID – to pięć ważnych zasad w większości zdefiniowanych przez Roberta Martina, które warto brać pod uwagę przy tworzeniu systemu programistycznego. Te zasady bazują na wzorcach projektowych. Warto zauważyć, że jest dużo więcej zasad opisanych we wzorcach. Nie wiadomo, czemu dokładnie te 5 zasad było wyróżnionych ponad 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 wymaganiom, ale działa dobrze według swego celu, to nie warto jej modyfikować, tylko stworzyć swoją klasę. Jeżeli część funkcjonalna pasuje, to trzeba zwrócić uwagę na wartość tworzenia nowej klasy. Jeżeli klasy nie można szybko przeanalizować i na jej podstawie stworzyć nowej, to warto ją rozszerzyć. Ceną tego rozwiązania jest zbędna ilość elementów w rozszerzonej klasie a stąd większe zapotrzebowanie zasobów komputera.

- Liskov substitution

Dziedziczenia 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 hierarchię szerokich oraz wąskich interfejsów przy pomocy dziedziczenia.

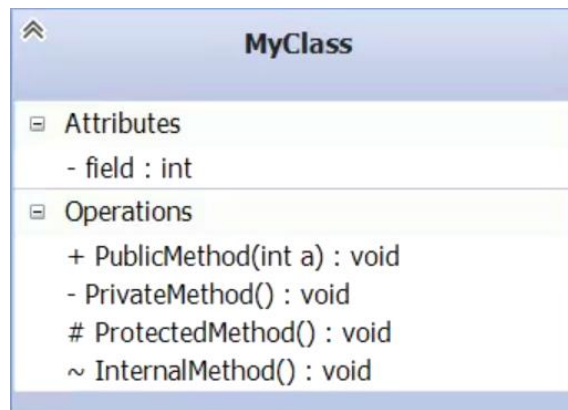
- Dependency inversion

Abstrahowanie procesu tworzenia obiektu przy pomocy użycia abstrakcyjnych klas produktów.

## 1.4. UML

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

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



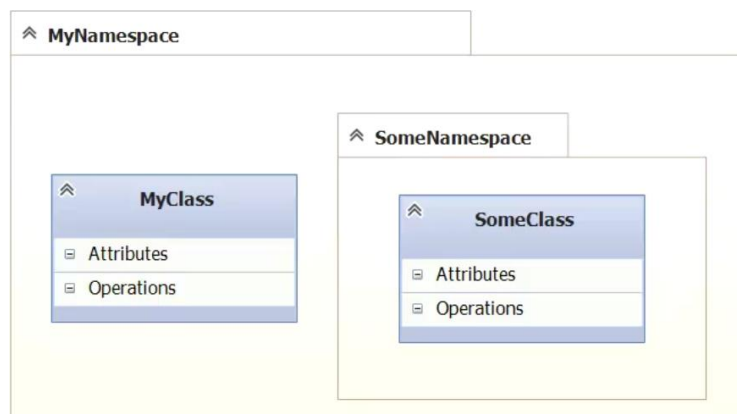
Rysunek 1. Klasyfikator w języku UML

- 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 opisanu stereotypów, w języku UML mamy klasyfikator ‘Interface’ oraz klasyfikator ‘Enumeration’.

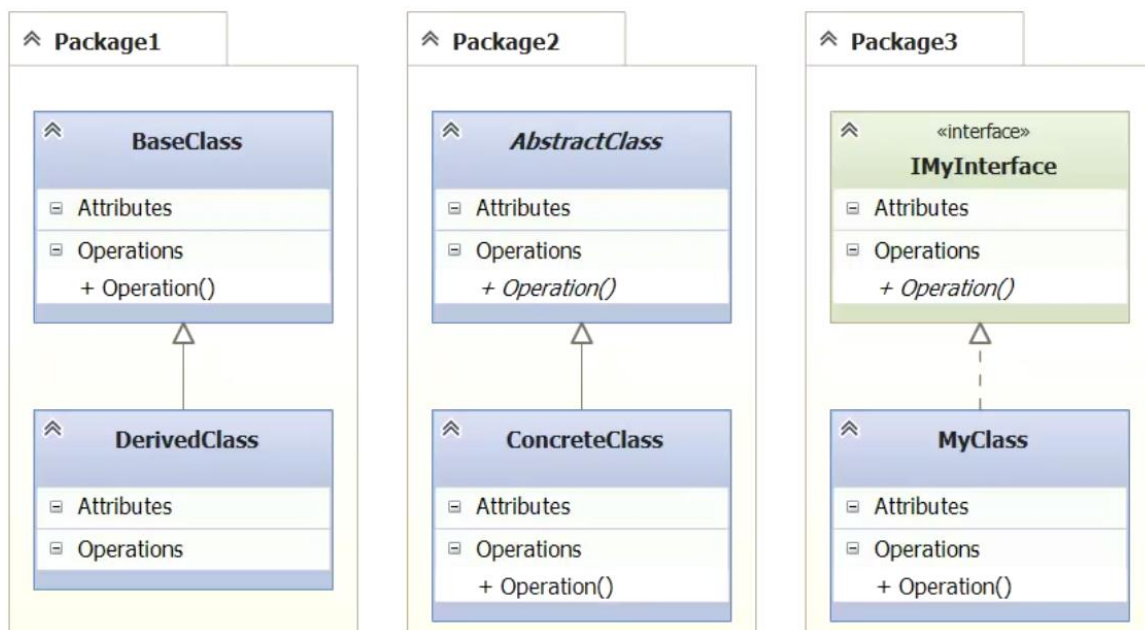
Stereotyp - pojęcie zbiorowe dla klas, struktur, interfejsów, enumeracji oraz delegatów.

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



Rysunek 2. Pakiety UML

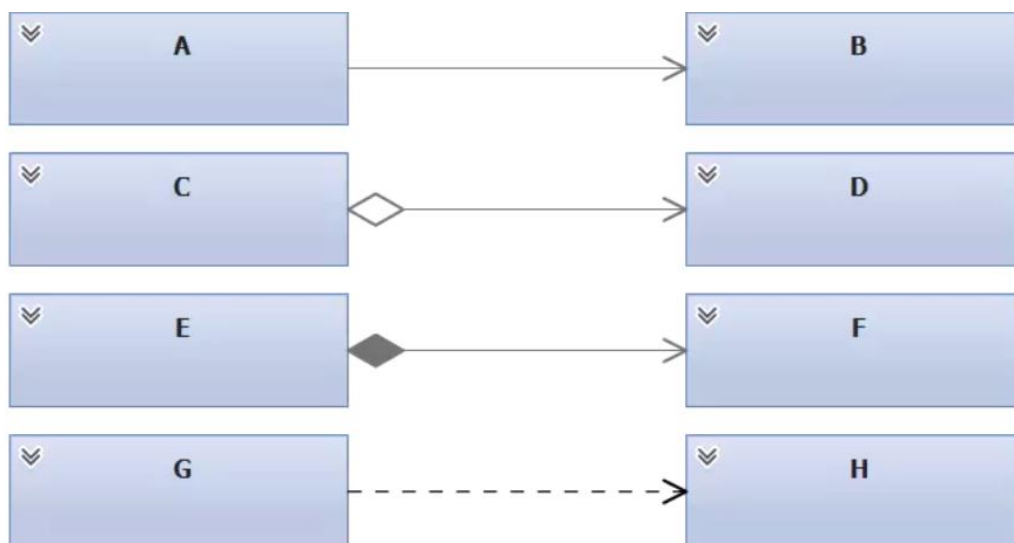
Na Rysunku 3 widzimy dziedziczenie w języku UML. Warto zauważyć, że w przypadku ostatnim nie tylko realizujemy interfejs, ale też go dziedziczymy. Pierwsze dwa dziedziczenia nazywamy dziedziczeniem realizacji, trzeci nazywamy dziedziczeniem interfejsu.



Rysunek 3. Dziedziczenie UML

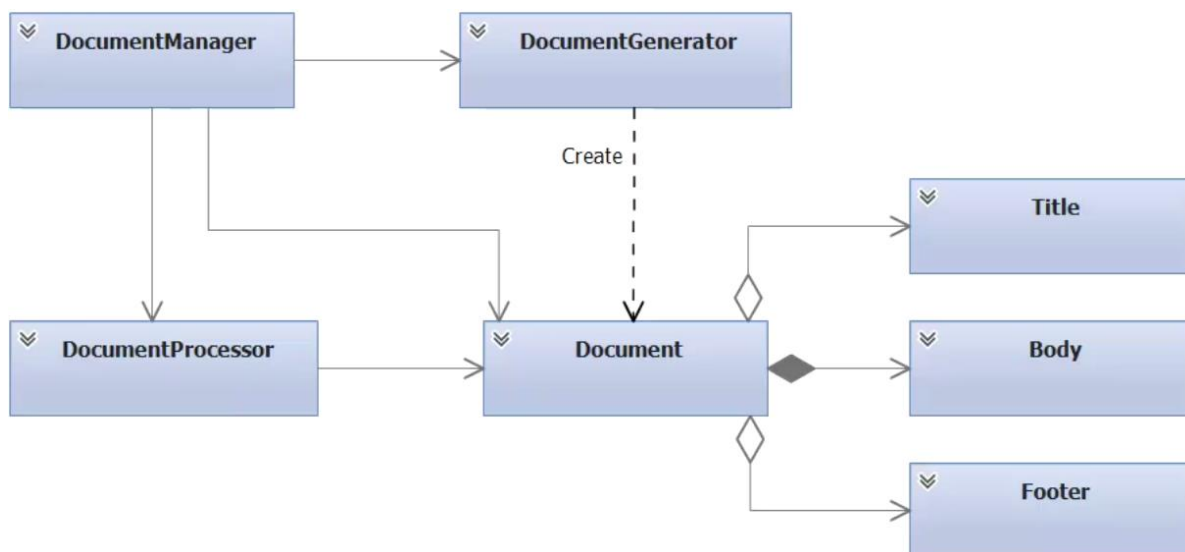
Na Rysunku 4 widzimy jeszcze cztery rodzaje związków między klasami:

- 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.



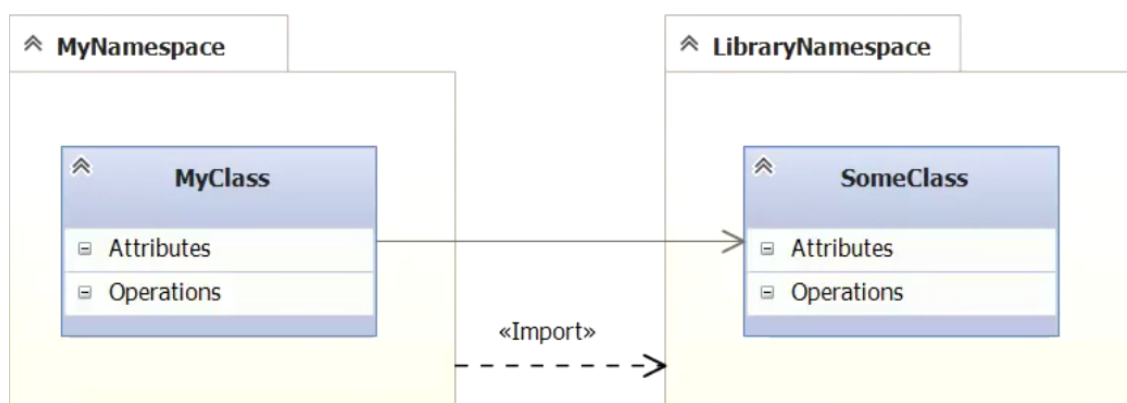
Rysunek 4. Rodzaje związków w języku UML

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



Rysunek 5. Przykład wykorzystania związków UML

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



Rysunek 6. Import Pakietów w języku UML

### 1.5. Problemy projektowania nowoczesnych systemów informatycznych

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

Duże systemy programistyczne są dość trudne w projektowaniu i wspieraniu, dlatego, dla ułatwienia pracy, duże systemy rozbijamy na podsystemy. Głównym zadaniem jest zmniejszenie liczby związków-relacji między klasami, czyli zmniejszenie liczby wzajemnych zależności klas.

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

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

Przywiązanie – termin logiczny, który zmusza developera do zastanowienia 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. Na przykład, kiedy klasa Customer jest związana związkiem asocjacji z klasą DataSet.

Niestety bez pewnej liczby przywiązań technicznych nie da się zaprojektować systemu. Przywiązanie techniczne może być nazwane dobrym wtedy i tylko wtedy, kiedy zależne encje są umieszczone 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 liczbę 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

### 2.1. Pojęcie wzorca projektowego

Wzorzec – nowy poziom abstrakcji w tworzeniu systemów programistycznych. 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 pozwoliły 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 ich główne elementy oraz dane, które możemy przekazywać przez interfejs. Wzorzec też może „powiedzieć”, co nie powinno być przekazywane przez interfejs.

### 2.2. 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 egzemplarzy 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.
- **obiektywne** – opisujące związki pomiędzy obiektami.

## 2.3. 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 abstrakcyjny interfejs, który możemy użyć do tworzenia obiektów-produktów. Użytkownicy mają możliwość wybierać konkretny produkt, który trzeba stworzyć.

- Strona architekta:

Architekt tworzy abstrakcyjny 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, jakiego wzorca kreacyjnego 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 kilku klas ConcreteCreator oraz przy tworzeniu podać typ tworzonego produktu:



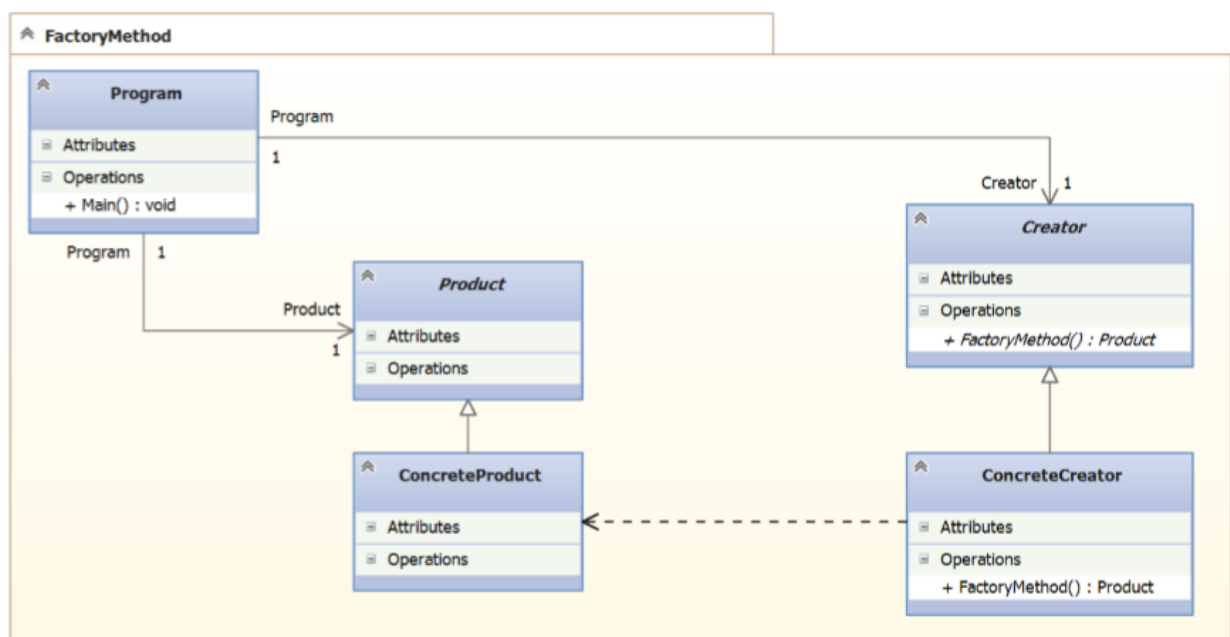
```
ICreator creator = new StandardCreator();  
  
IProduct productA = creator.CreateProduct<ProductA>();  
  
IProduct productB = creator.CreateProduct<ProductB>();  
  
IProduct productC = creator.CreateProduct<ProductC>();
```

#### *Nazewnictwo:*

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

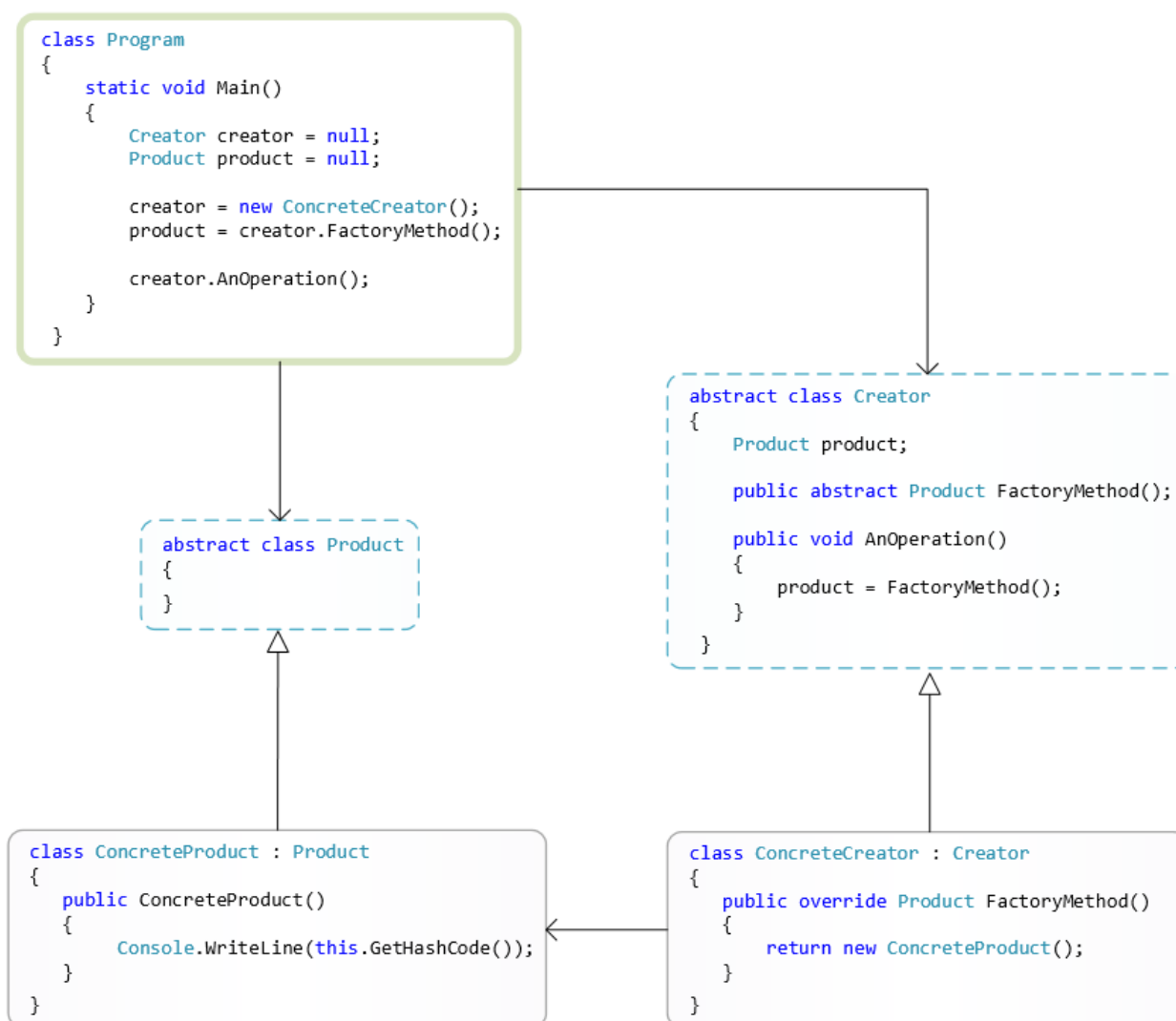
#### *Wykorzystanie w .NET:*

Jest fundamentem wszystkich wzorców kreacyjnych, czyli jest wykorzystywany wszędzie tam, gdzie można spotkać wzorce kreacyjne. Na Rysunku 7 możemy zobaczyć wszystkie związki pomiędzy klasami przy pomocy diagramu UML.



*Rysunek 7. Struktura Factory Method w języku UML*

Na Rysunku 8 widzimy szkielet wzorca w języku C#.



Rysunek 8. Struktura Factory Method w języku C#

## 2.4. 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.

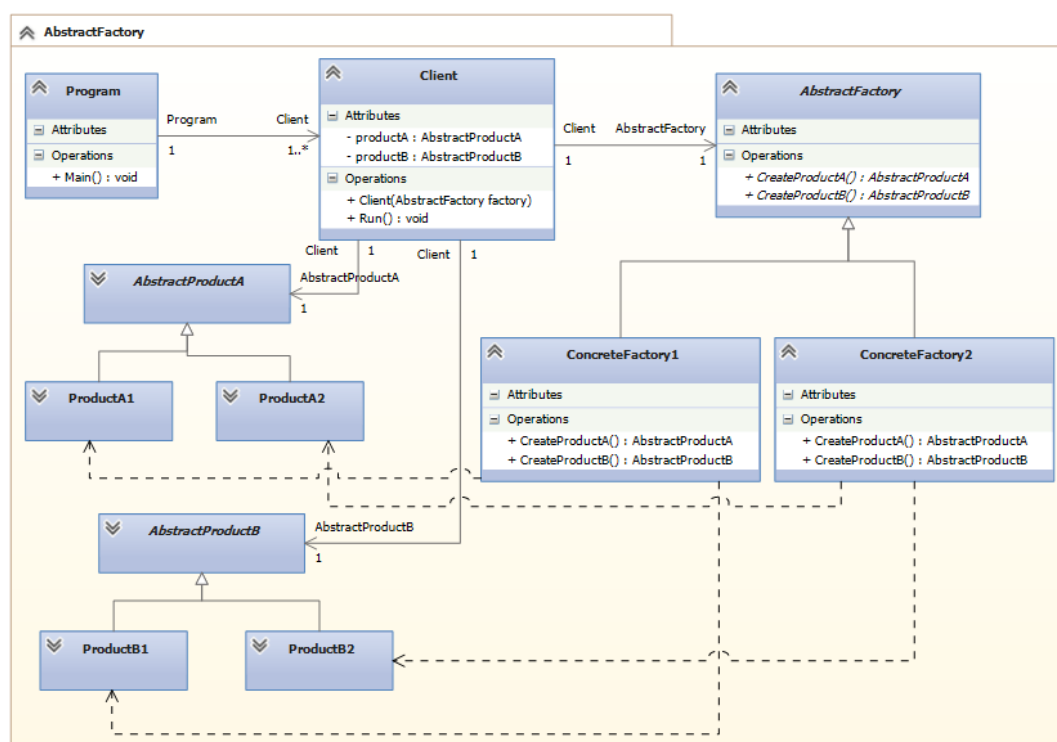
Jest to wzorzec, który opisuje algorytm tworzenia interfejsu pozwalającego instancjować rodziny powiązanych lub zależnych między sobą obiektów-produktów [5]. Także hermetyzuje informację o konkretnych klasach tworzonych obiektów-produktów.

Żeby zrozumieć, do czego służy ten wzorec, 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 użycia produktu z jednej rodziny produktów do współpracy z produktem innej rodziny: nie możemy wlewać cole w butelkę pepsi.

Wykorzystanie w .NET:

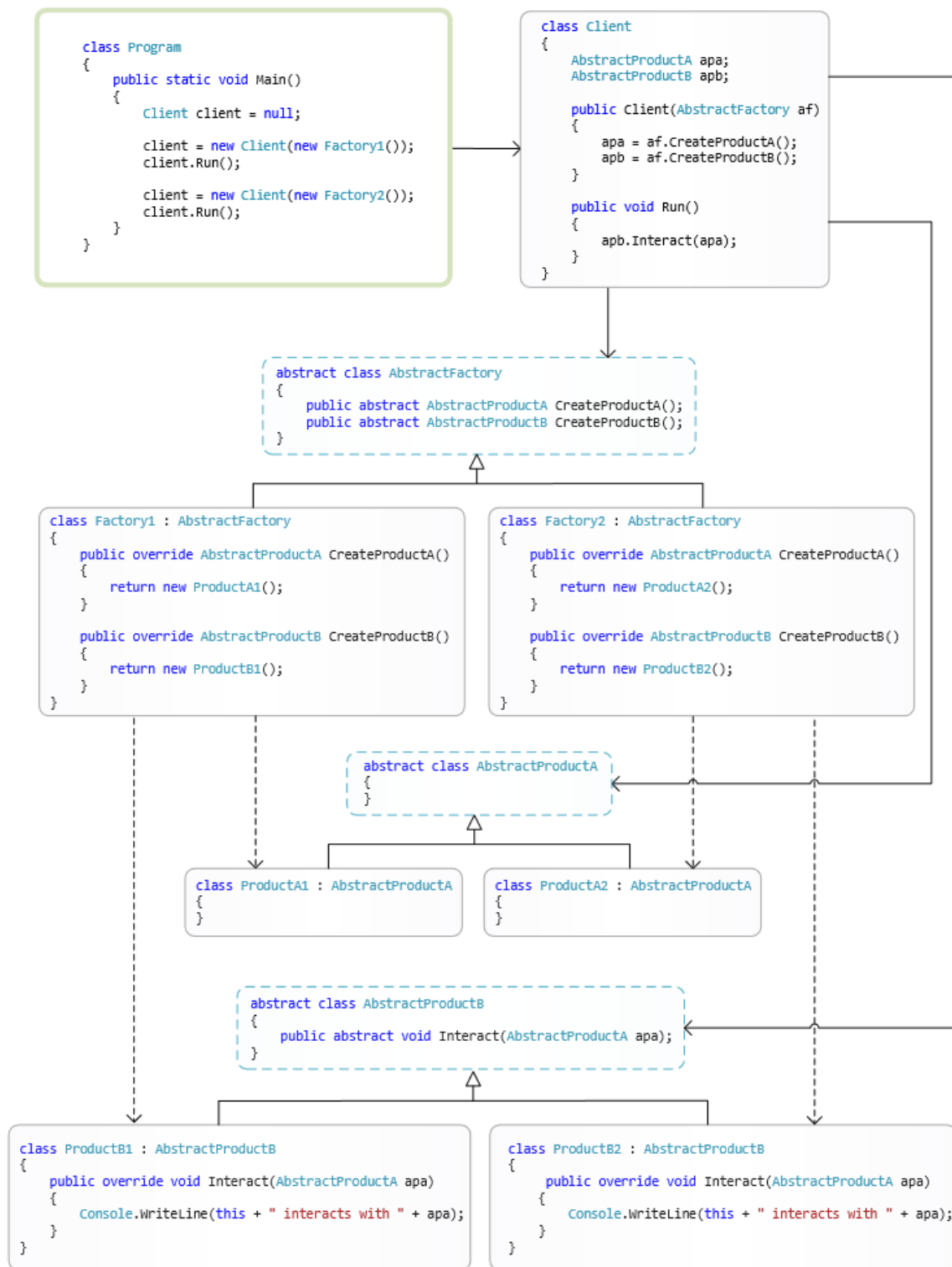
- System.Activities.Presentation.Model.ModelFactory
- System.Data.Common.DbProviderFactory
- System.Data.EntityClient.EntityProviderFactory
- System.Data.Odbc.OdbcFactory
- System.Data.OleDb.OleDbFactory
- oraz inne

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



Rysunek 9. Struktura Abstract Factory w języku UML

Na Rysunku 10 widzimy szkielet wzorca w języku C#.



Rysunek 10. Struktura Abstract Factory w języku C#

## 2.5. 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 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 egzemplarzy jednej klasy

*Osobliwości C#:*

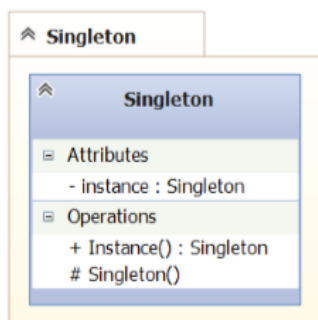
Jedną z możliwości realizacji tego wzorca w C# jest użycie klas statycznych. Według mnie jest to preferowany sposób realizacji, aczkolwiek ma swoje wady: nie pasuje w przypadku potrzeby tworzenia kilku instancji klasy singleton. Klasy statyczne nie mogą uczestniczyć w dziedziczeniu, a to znaczy, że nie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

Przy użyciu wielowątkowości zostaje ryzyko tworzenia kilku egzemplarzy. To ryzyko można wyeliminować przy pomocy konstrukcji `lock(){}` , wewnątrz której odbędzie się instancjacja egzemplarza obiektu.

*Wykorzystanie w .NET*

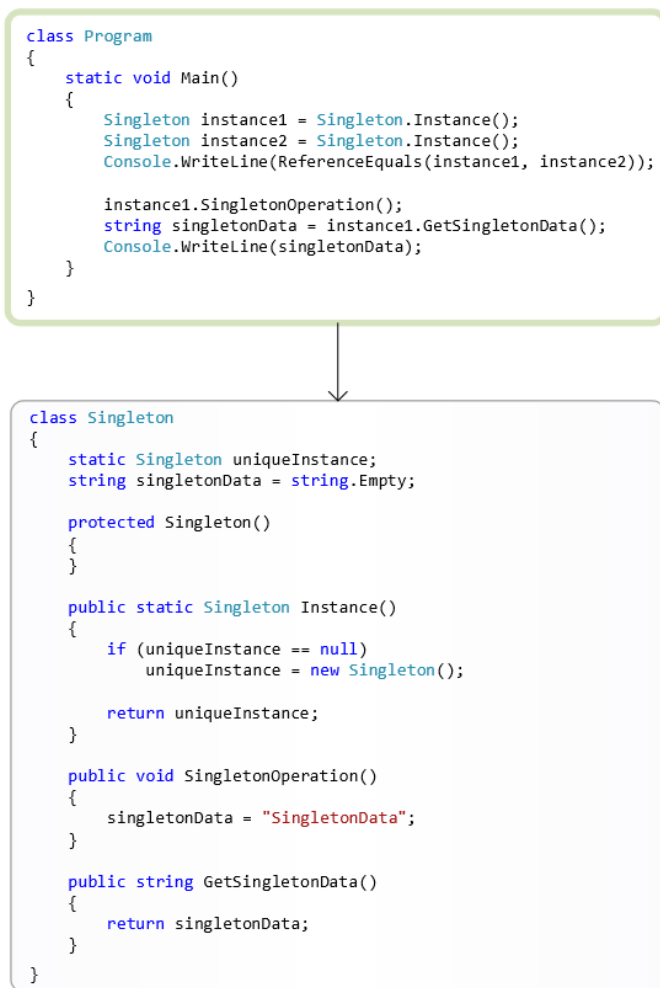
- `System.ServiceModel.ServiceHost`
- `System.Data.DataRowComparer`

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



*Rysunek 11. Struktura Singleton w języku UML*

Na Rysunku 12 widzimy szkielet wzorca w języku C#.



*Rysunek 12. Struktura Singleton w języku C#*

## 2.6. 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, którego celem 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 niekompatybilnym 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 adapterów poziomu obiektów, czyli tworzyć egzemplarz klasy do adaptacji wewnątrz klasy-adaptera zamiast uzyskania jej interfejsu dziedziczeniem.

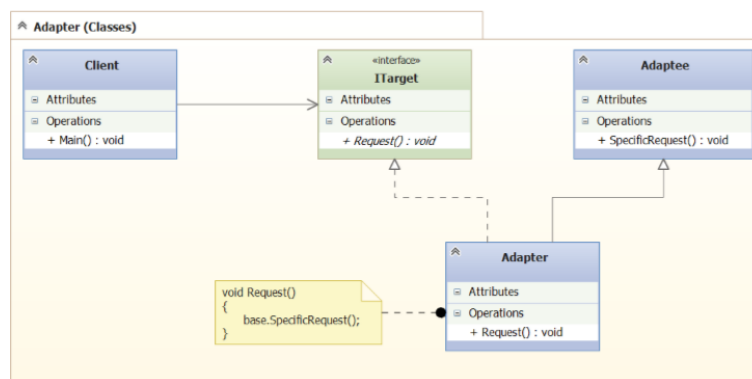
*Nazewnictwo:*

Dobrym przykładem nazewnictwa jest dopisanie po nazwie klasy postfiks Adapter.

Wykorzystanie w .NET:

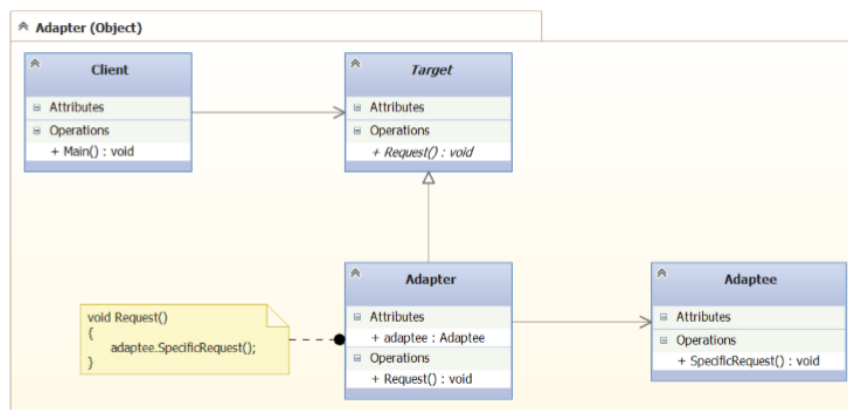
- System.Data.Common.DbDataAdapter
- System.Data.OleDb.OleDbDataAdapter
- System.Data.OracleClient.OracleDataAdapter
- System.Web.UI.Adapters.ControlAdapter
- System.Web.UI.Adapters.PageAdapter

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



Rysunek 13. Struktura Adapteru poziomu klas w języku UML

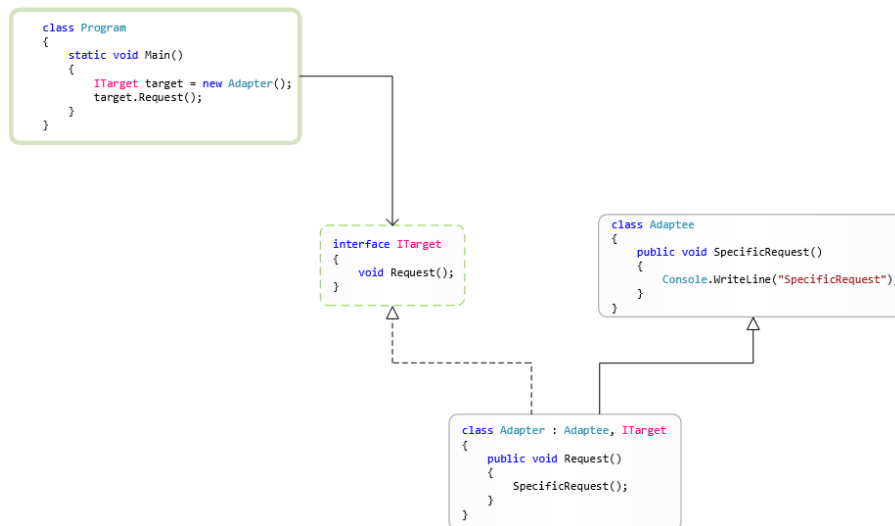
Na Rysunku 14 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku Adaptera poziomu obiektów przy pomocy diagramu UML.



Rysunek 14. Struktura Adapteru poziomu obiektów w języku UML

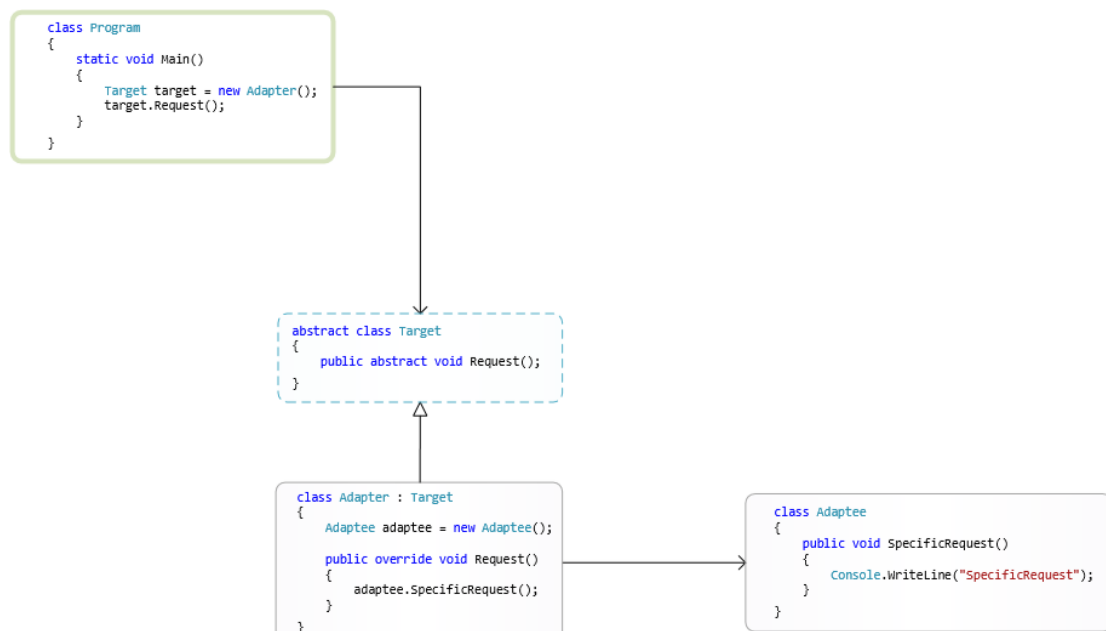
Na Rysunku 15 widzimy szkielet Adaptera poziomu klas w języku C#.





Rysunek 15. Struktura Adapteru poziomu klas w języku C#

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



Rysunek 16. Struktura Adapteru poziomu obiektów w języku C#

## 2.7. 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, którego używa się nie tylko do podsystemów ale również do architektury ogólnej systemu. W większości seminariów, tematem których jest tworzenie architektury aplikacji, spotkamy ten wzorzec.

Dobłą metaforą facade może być interfejs samochodu. My, jako użytkownicy, mamy bardzo uproszczony interfejs relacji ze złożonym podsystemem (autem). Mamy 2 pedały oraz kierownicę. Uruchamiając samochód nawet nie musimy zastanawiać się nad złożonymi procesami, które dzieją się pod jego maską. Inżynierowi 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 tych podsystemów.

*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. *accomplish low coupling*).

*Strona programisty-realizatora:*

Programista wykorzystuje przygotowany dla niego interfejs.

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

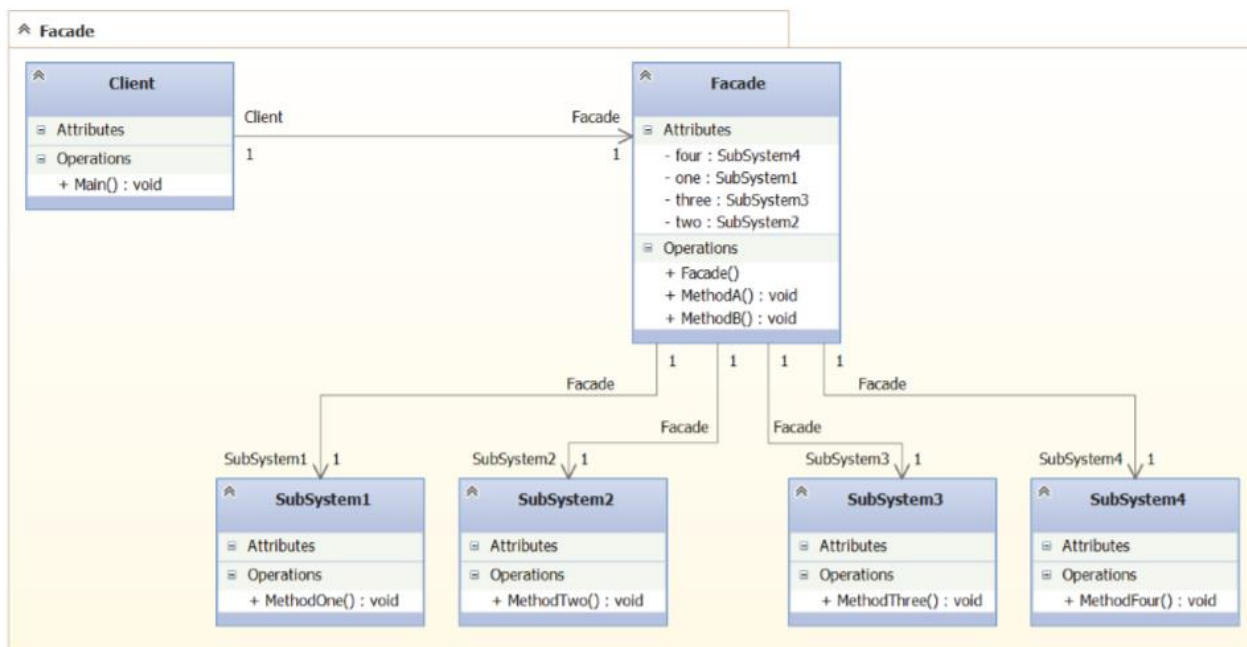
*Kiedy użyć:*

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

*Wykorzystanie w .NET:*

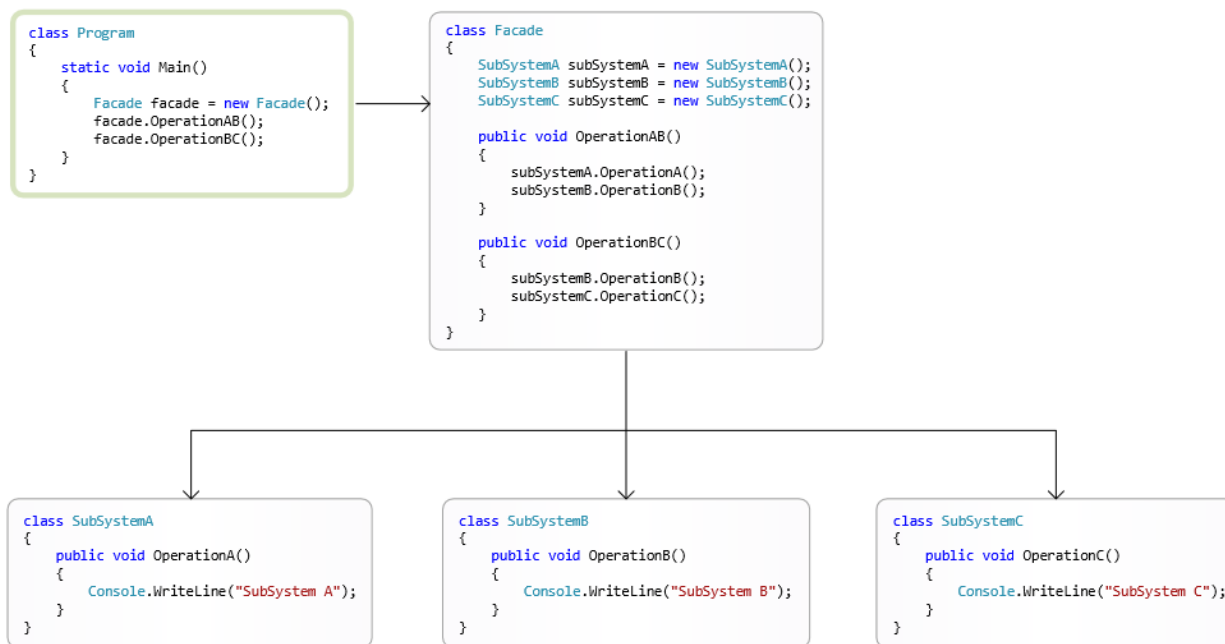
Wszędzie.

Na Rysunku 17 możemy zobrazować przy pomocy diagramu UML wzorzec Facade składający się z czterech podsystemów.



Rysunek 127. Struktura Facade w języku UML

Na Rysunku 18 widzimy szkielet wzorca w języku C#.



Rysunek 138. Struktura Facade w języku C#

## 2.8. Composite

*Nazwa:* Composite

*Inne nazwy:* brak

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

*Częstość użycia:* duża

*Przeznaczenie:* zbieranie obiektów w drzewopodobne struktury.

Wzorzec Composite opisuje algorytm stworzenia drzew obiektów. Drzewo jest bardzo użyteczną strukturą danych w świecie programistycznym. Drzewo zawiera trzy główne składniki: korzeń-gałąź, gałąź, 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 klasę abstrakcyjną Component, która musi zawierać jak najwięcej wspólnych operacji dla liścia oraz gałęzi.

*Strona architekta:*

Pozwala zbudować drzewopodobną strukturę danych, hermetyzując elementy drzewa.

*Strona programisty-realizatora:*

Programista nie musi wiedzieć, z jaką częścią drzewa pracuje.

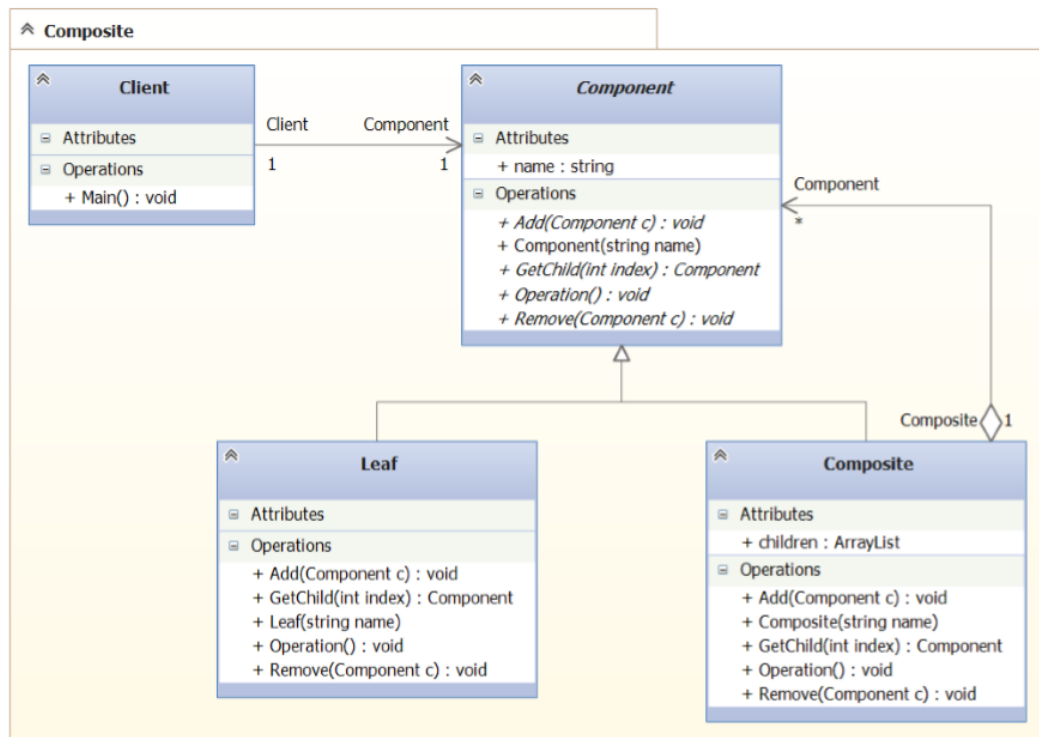
*Kiedy użyć:*

Kiedy potrzebujemy stworzyć drzewopodobną strukturę lub heterogenną hierarchię obiektów.

Wykorzystanie w .NET:

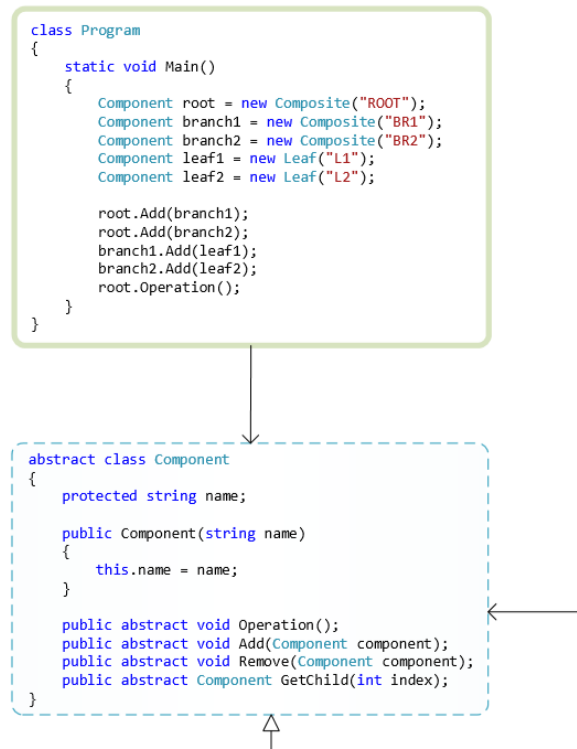
- System.Web.UI.WebControls.CompositeControl
- System.Windows.Data.CompositeCollection
- System.Web.UI.Design.WebControls.TreeViewDesigner
- System.Web.UI.WebControls.TreeNodeCollection
- System.Web.UI.WebControls.TreeView
- System.Windows.Controls.TreeView

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

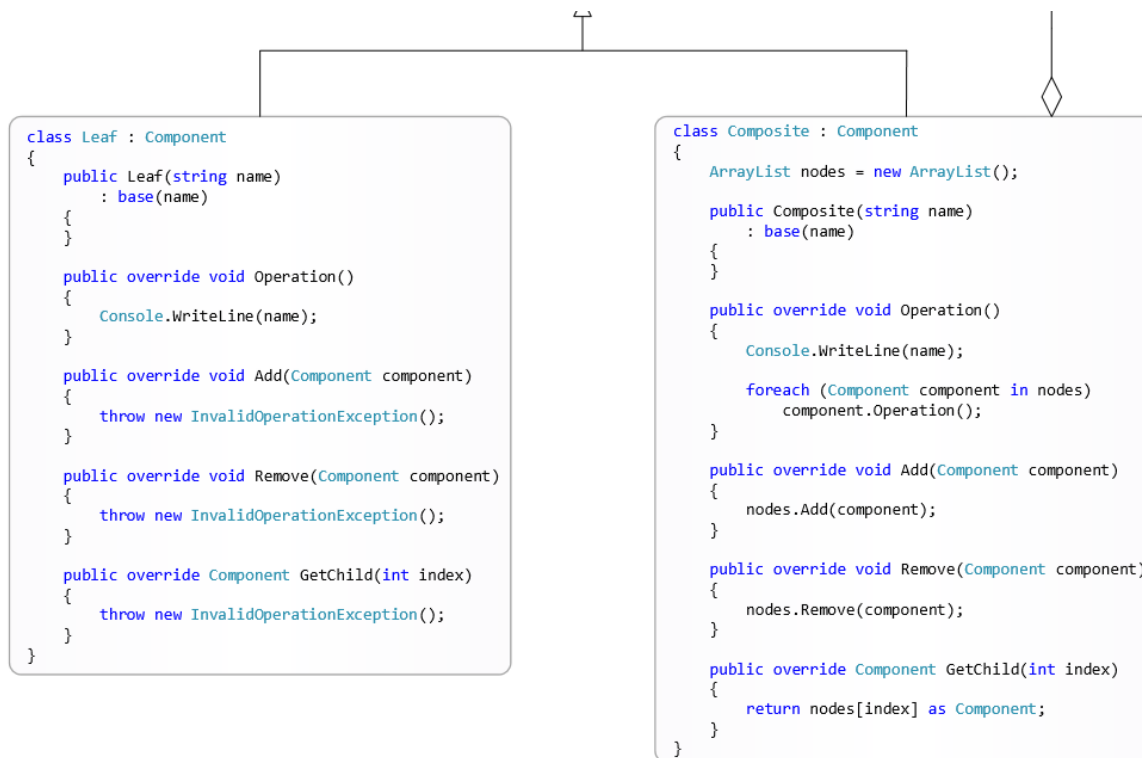


Rysunek 149. Struktura Composite w języku UML

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



Rysunek 20. Struktura Composite w języku C#



Rysunek 21. Struktura Composite w języku C#

## 2.10. 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 samodzielnie (o ile mamy z robotem wspólny interfejs). Taki, czasami poręczny model, możemy realizować przy pomocy wzorca Proxy.

W tym wzorcu są dwie główne części realizacji – dziedziczenie wspólnego interfejsu dwoma 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 obrotne, prawa dostępu.

*Nazewnictwo:*

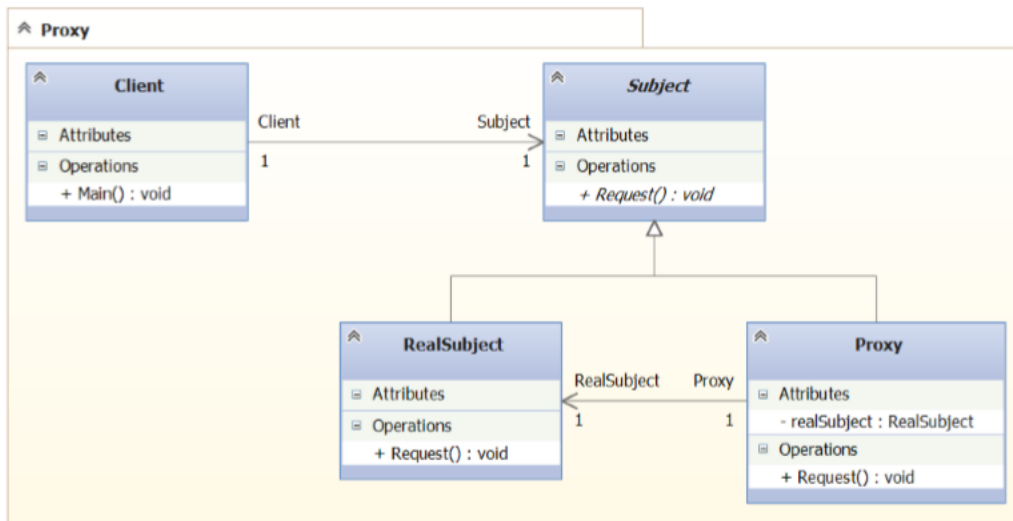
Warto używać sufiksu 'Proxy' w klasach-surogatach

*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

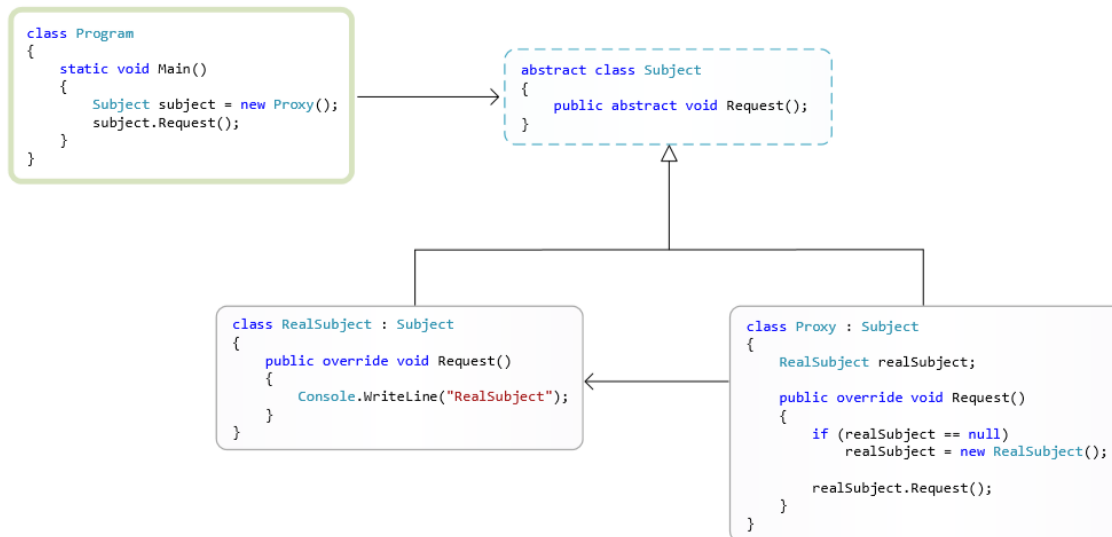
- System.Net.GlobalProxySelection
- System.Net.IWebProxy

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



Rysunek 22. Struktura Proxy w języku UML

Na Rysunku 23 widzimy szkielet wzorca w języku C#.



Rysunek 23. Struktura Proxy w języku C#



## 2.11. 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, kolejkovanie 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
- Kolejkovanie żądań w czasie
- Logowanie zmian

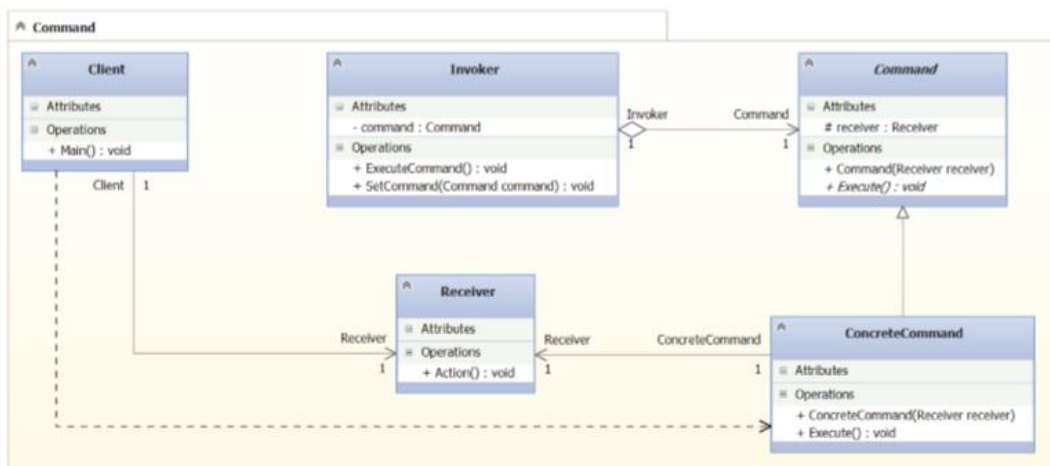
*Nazewnictwo:*

Dobrym przykładem nazewnictwa jest dopisanie po nazwie klasy-komendy postfiks Command a także nazwanie wywołującej komendy klasy 'Invoker'.

*Wykorzystanie w .NET:*

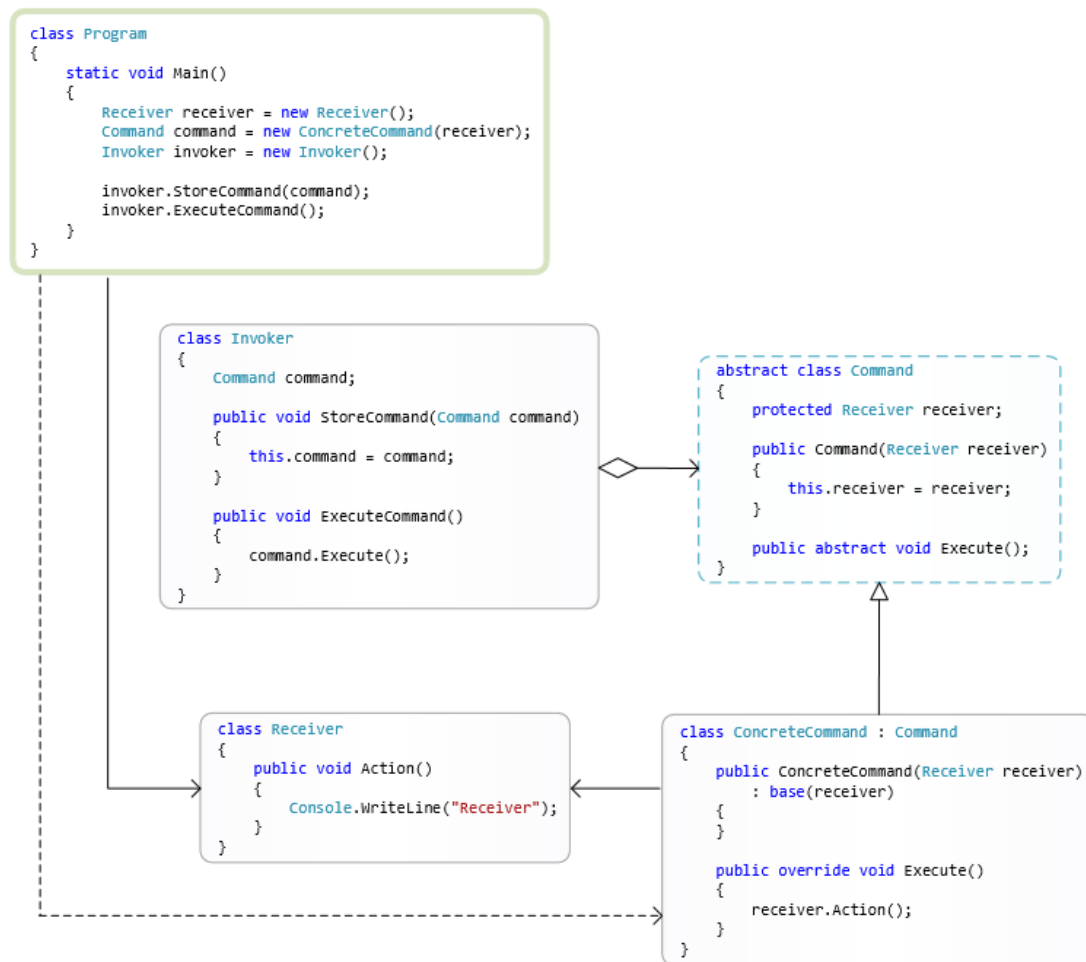
- System.Windows.Input.MediaCommands
- System.Windows.Input.NavigationCommands
- System.Windows.Input.RoutedCommand
- System.Windows.SystemCommands
- System.Workflow.ComponentModel.Design.WorkflowMenuCommands

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



*Rysunek 24. Struktura Command w języku UML*

Na Rysunku 25 widzimy szkielet wzorca w języku C#.



Rysunek 25. Struktura Command w języku C#

## 2.12. 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 bezpieczeństwo dostępu do elementu kolekcji.

Dobłą 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 iteratora 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-wrapperami. 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 iteratora polimorficznego.

*Strona architekta:*

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

*Strona programisty-realizatora:*

Programista wykorzystuje przygotowany dla niego mechanizm przy pracy z kolekcją.

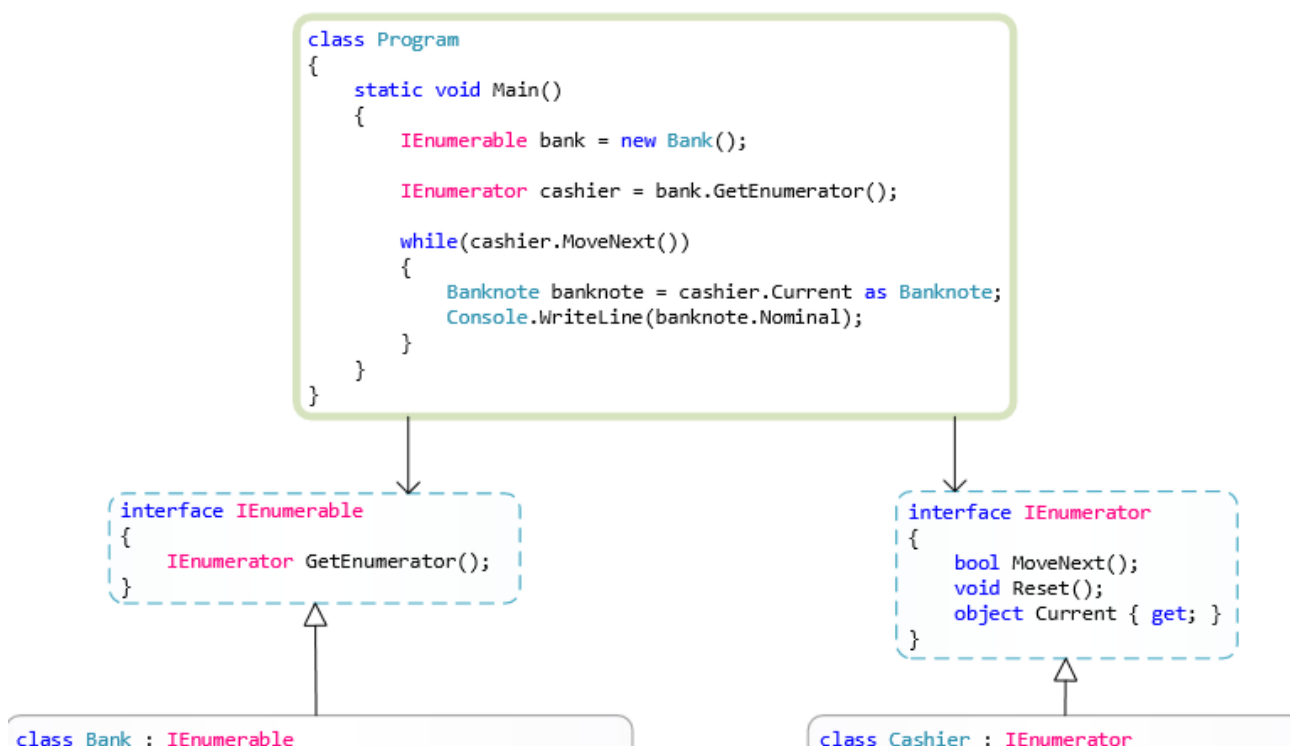
*Kiedy użyć:*

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

*Osobliwości C#:*

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.

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



Rysunek 156. Implementacja Iteratora w języku C#

## 2.13. 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 pchania (ang. *push*) – zdarzenie nie da się zignorować oraz wyciągnięcia (ang. *pull*) – zależy od developera czy chcemy reagować na zdarzenie. Observer w .NET jest zamieszczony nowym mechanizmem (więcej informacji w podrozdziale osobliwości C#).

Strona architekta:

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

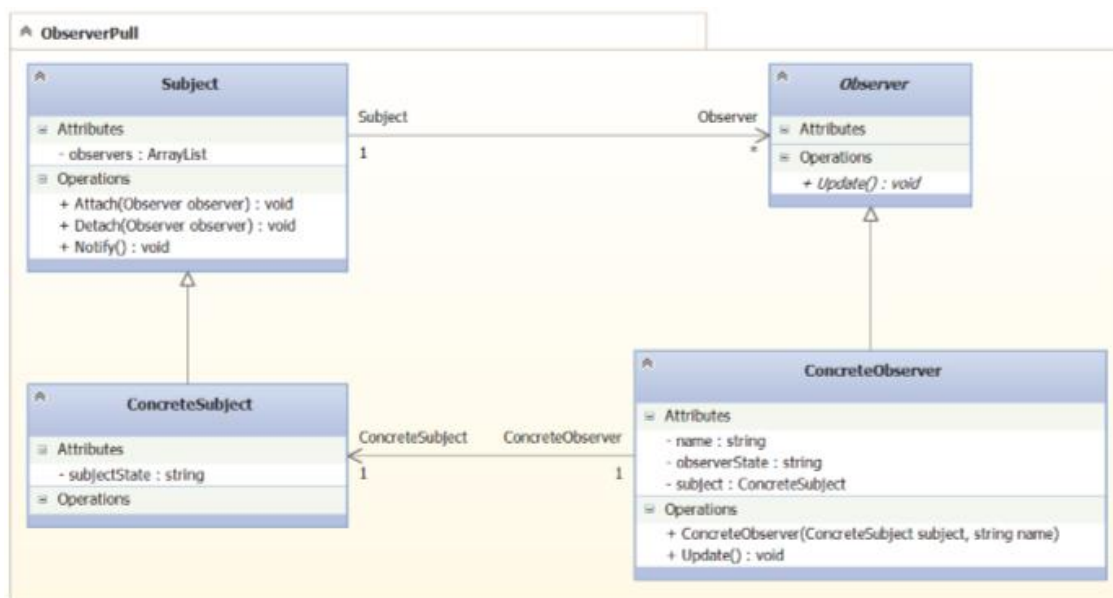
*Kiedy użyć:*

- Kiedy potrzebujemy realizować mechanizm zdarzeń.

*Osobliwości C#:*

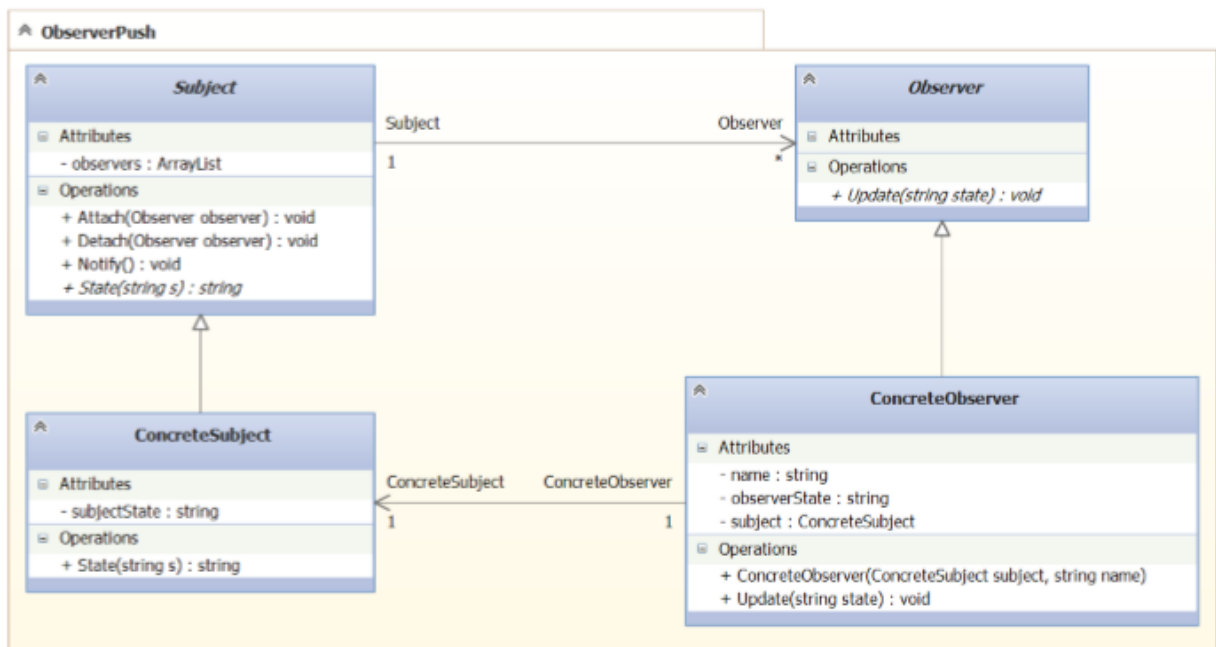
Ten wzorzec leży w fundamencie modelu zdarzeń .NET i realizowany jest przy pomocy delegatów. Inne języki, które nie posiadają mechanizmu delegatów oraz nie mogą sobie pozwolić na zaimplementowanie tego mechanizmu technicznie lub finansowo, zmuszają programistę do implementowania tego wzorca ręcznie.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

Na Rysunku 27 możemy zobaczyć wszystkie związki pomiędzy klasami w przypadku modelu *pull*, wyrażone przy pomocy diagramu UML.



*Rysunek 167. Struktura Observer modelu Pull w języku UML*

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



Rysunek 178. Struktura Observer modelu Push w języku UML

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

```

class Program
{
    static void Main()
    {
        ConcreteSubject subject = new ConcreteSubject();
        subject.Attach(new ConcreteObserver(subject));
        subject.Attach(new ConcreteObserver(subject));
        subject.State = "Some State ...";
        subject.Notify();
    }
}

```

```

abstract class Subject
{
    ArrayList observers = new ArrayList();

    public void Attach(Observer observer)
    {
        observers.Add(observer);
    }

    public void Detach(Observer observer)
    {
        observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (Observer observer in observers)
            observer.Update();
    }
}

```

```

abstract class Observer
{
    public abstract void Update();
}

```

```

class ConcreteSubject : Subject
{
    public string State { get; set; }
}

```

```

class ConcreteObserver : Observer
{
    string observerState;
    ConcreteSubject subject;

    public ConcreteObserver(ConcreteSubject subject)
    {
        this.subject = subject;
    }

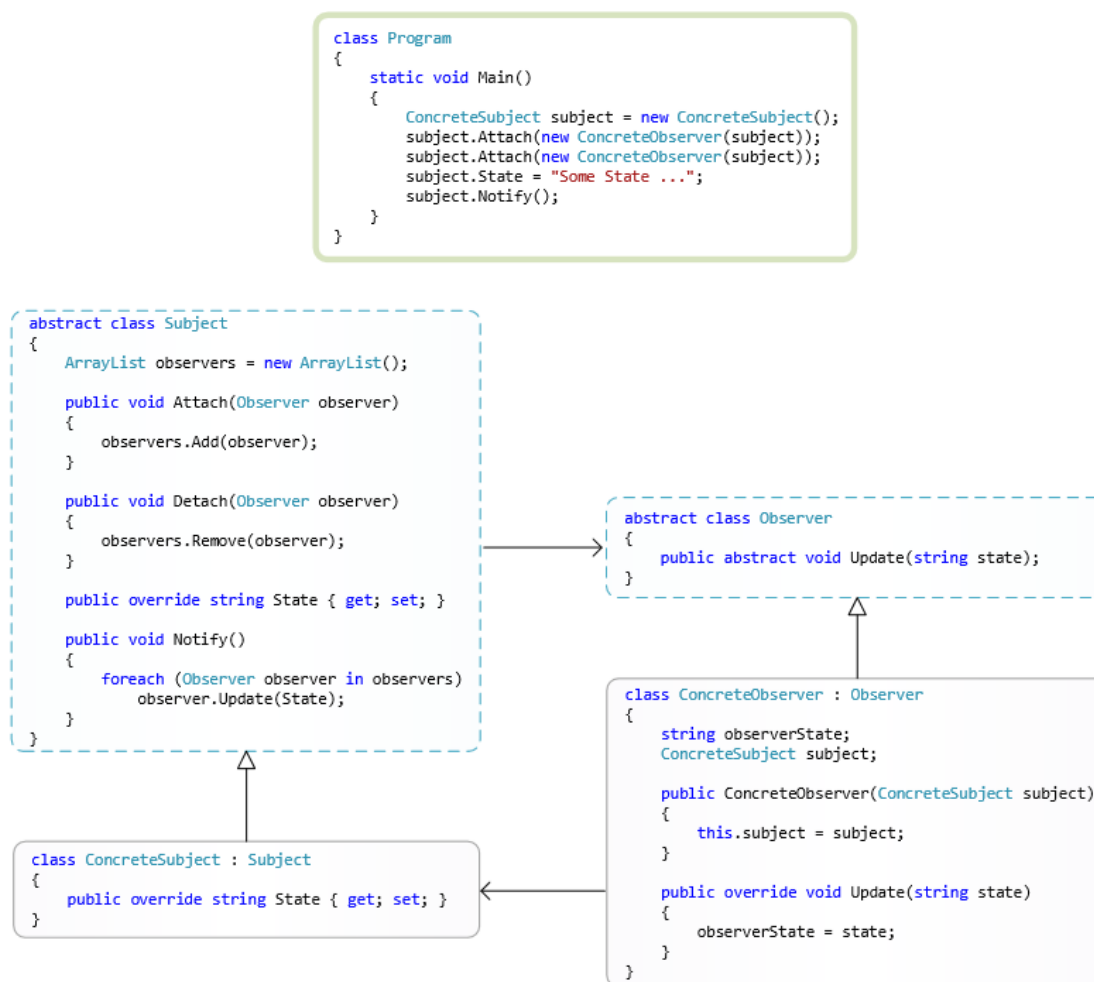
    public override void Update()
    {
        observerState = subject.State;
    }
}

```

Rysunek 189. Struktura Observer modelu Pull w języku C#

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





Rysunek 30. Struktura Observer modelu push w języku C#

## 2.14. Strategy

Nazwa: Strategy

Inne nazwy: Policy

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

Częstość użycia: duża

Przeznaczenie: podmiana algorytmów-strategii

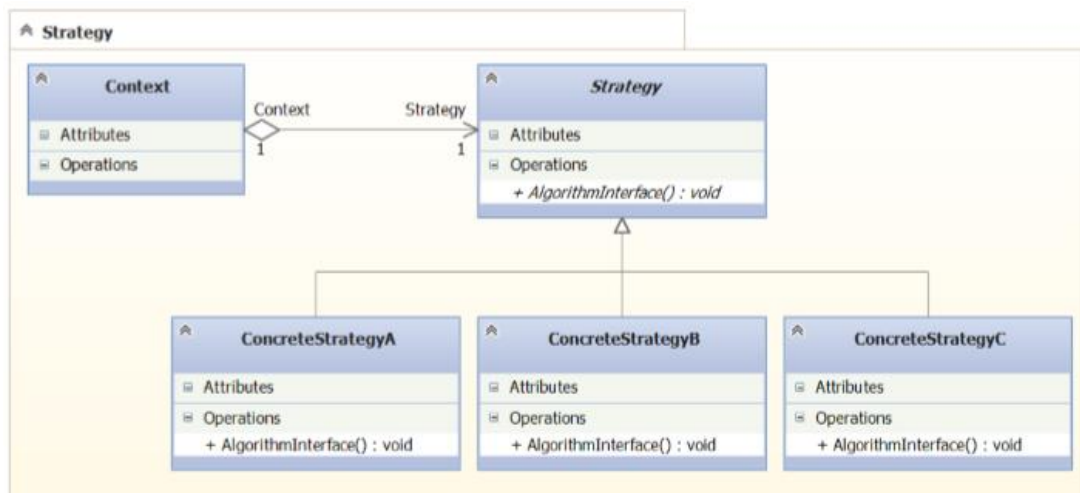
Wzorec Strategy pozwala wybierać odpowiednią do sytuacji algorytm-strategię. Minusem tego wzorca jest to, że klient musi znać wewnętrzną realizację klas-strategii, żeby dokonać odpowiedniego, poprawnego wyboru.

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. Wzorec ten także pomaga efektywnie wykorzystywać zasoby systemu.

*Kiedy użyć:*

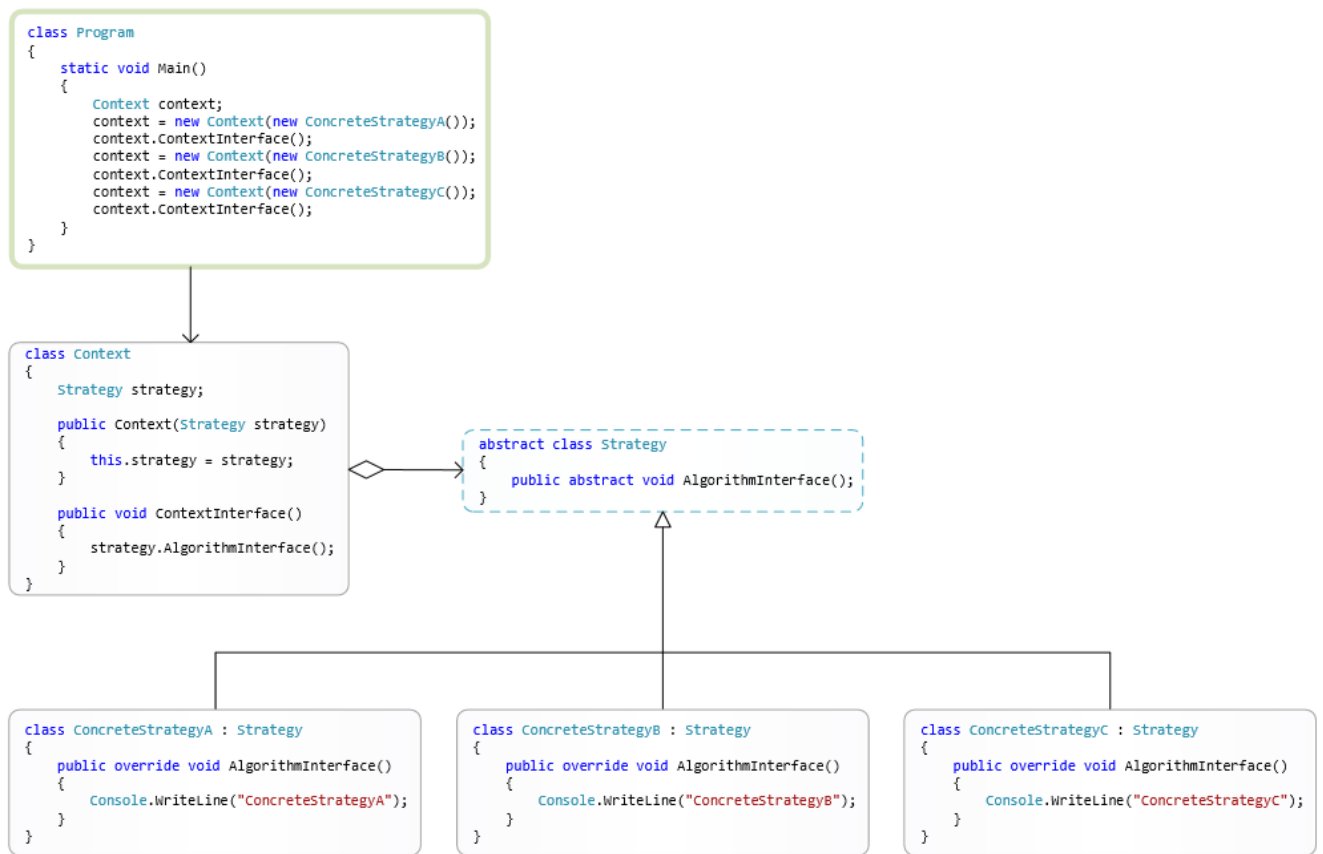
- kiedy potrzebujemy mieć kilka rodzajów jednego algorytmu: sortowanie, czytanie z pliku itd.,
- kiedy mamy kilka klas należących do jednej rodziny różniących się tylko zachowaniem.

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



*Rysunek 31. Struktura Strategy w języku UML*

Na Rysunku 32 widzimy szkielet wzorca w języku C#.



Rysunek 32. Struktura Strategy w języku C#



## Rozdział 3. Stworzenie aplikacji webowej z użyciem wzorców

### 3.1. Opis aplikacji

Aplikacja SPA 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 pliku *.csv*, pobranego 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 okresu. Interaktywne wykresy na tych podstronach zezwalają nie tylko zobaczyć wydatki za odpowiedni okres, ale też przejść do szczegółów wydatków tego okresu.
- Strona ustawień zezwala zmienić dane użytkownika oraz obecne hasło.
- Strona informacyjna aplikacji.

Autoryzacja użytkownika w aplikacji jest obowiązkowa, aby zidentyfikować jego transakcje.

*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.

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

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.

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

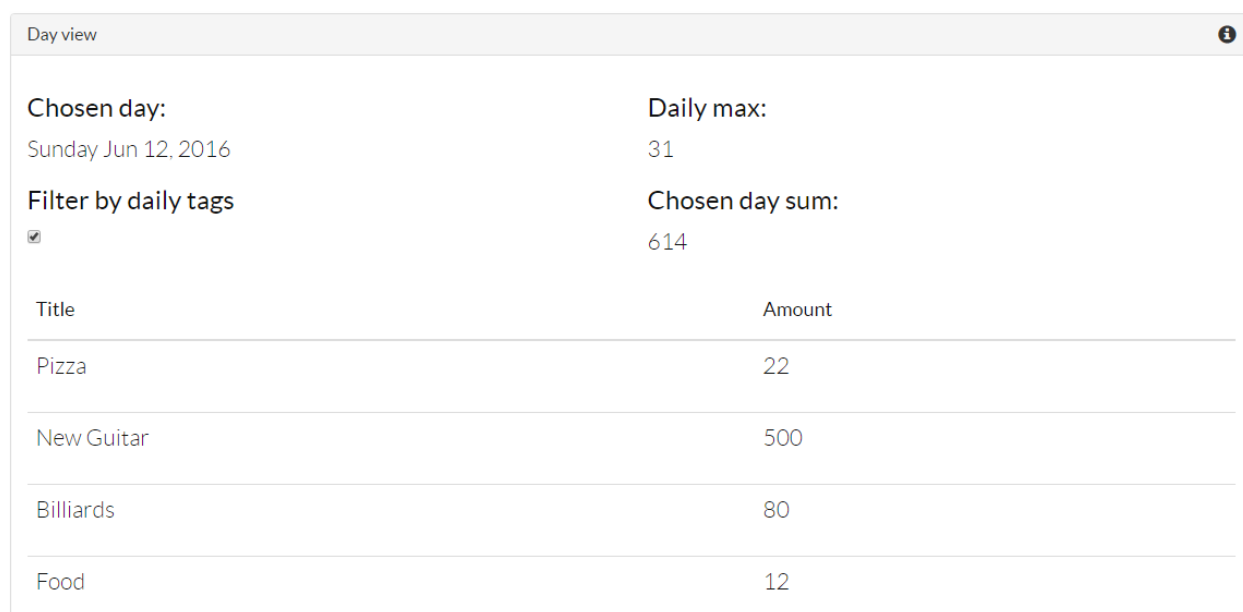
Date	Title	Amount	Tag	Type	
Jun 12, 2016	Pizza	22	Fo	UserOut	Remove
Jun 12, 2016	New Guitar	500	Food	UserOut	Remove
Jun 12, 2016	Billiards	80	Sport ✖ +	UserOut	Remove
Jun 12, 2016	Food	12	Food ✖ +	UserOut	Remove
May 30, 2016	dsfhoisfh	234	DailyTag ✖ +	UserOut	Remove
May 29, 2016	My Transaction	234	fawefawe ✖ rgaaga ✖ +	UserOut	Remove
May 28, 2016	Another trans	234	rgaaga ✖ +	UserOut	Remove
May 28, 2016	Bought chicken bell	14	DailyTag ✖ Sample tag ✖ +	UserOut	Remove
May 27, 2016	Bought new trousers	12	DailyTag ✖ +	UserOut	Remove
May 27, 2016	dsffbbfbefbew	23	DailyTag ✖ +	UserOut	Remove

« 1 2 3 4 5 6 »

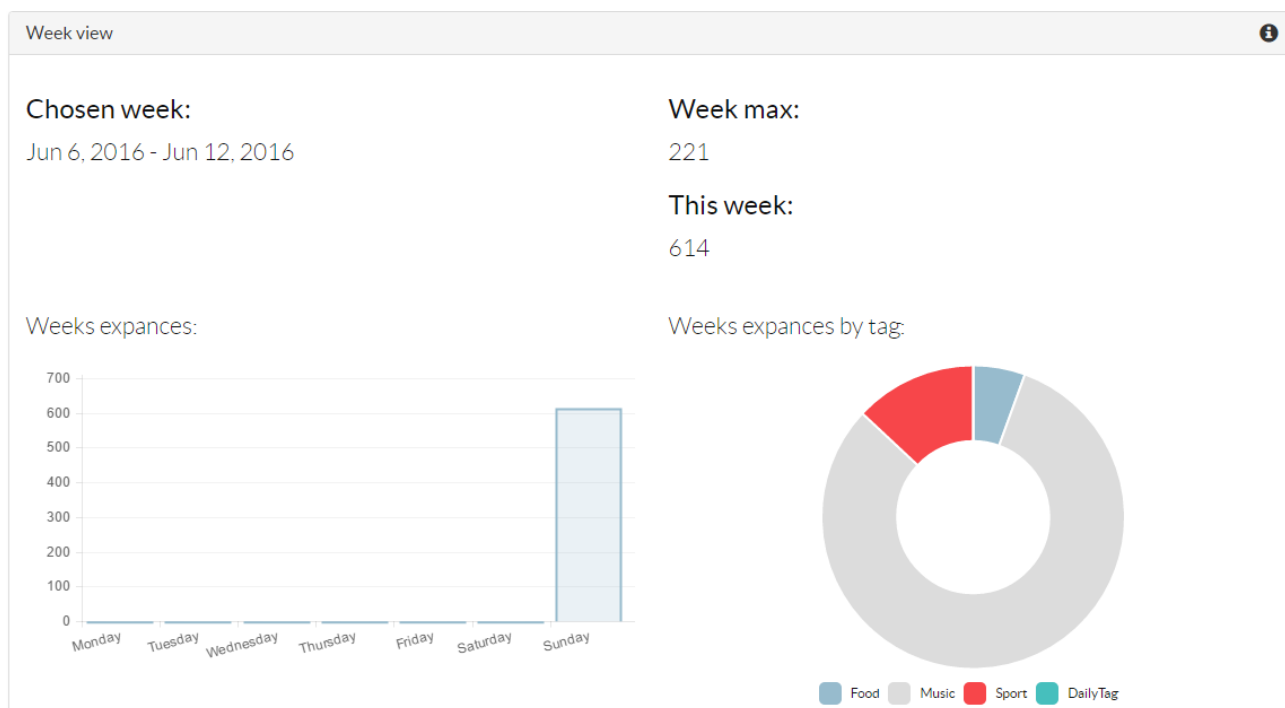
Add Drop file

Rysunek 33. Wyszukiwanie oraz podpięcie tagu do transakcji

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

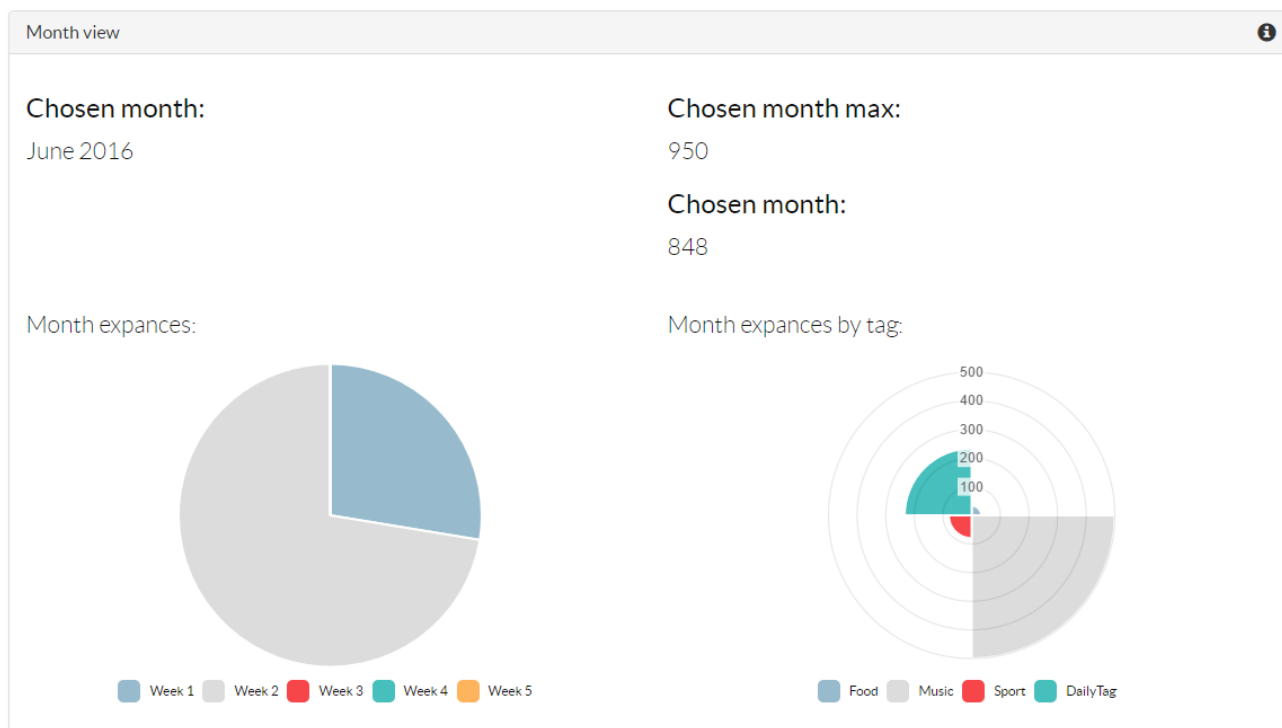


*Rysunek 34. Dashboard. Widok pojedynczego dnia*

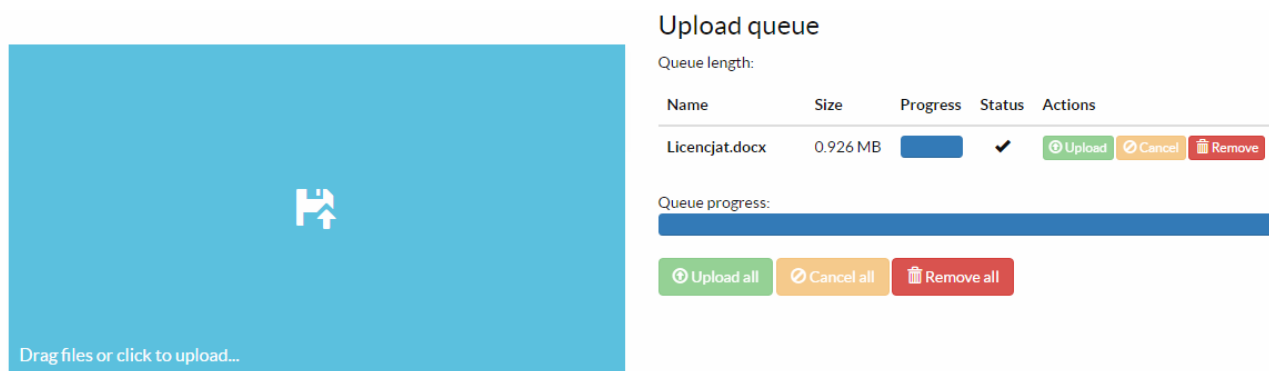


*Rysunek 35. Dashboard. Widok tygodnia*

Na Rysunku 37 widzimy komponent odpowiedzialny za ładowanie plików .csv z transakcjami bankowymi. Możemy ładować kilka plików naraz oraz używać drag and drop. Widać także postęp ładowania pliku. Jest możliwość natychmiastowego usunięcia pliku z serwera po naciśnięciu przycisku Remove.



Rysunek 36. Dashboard. Widok miesiąca



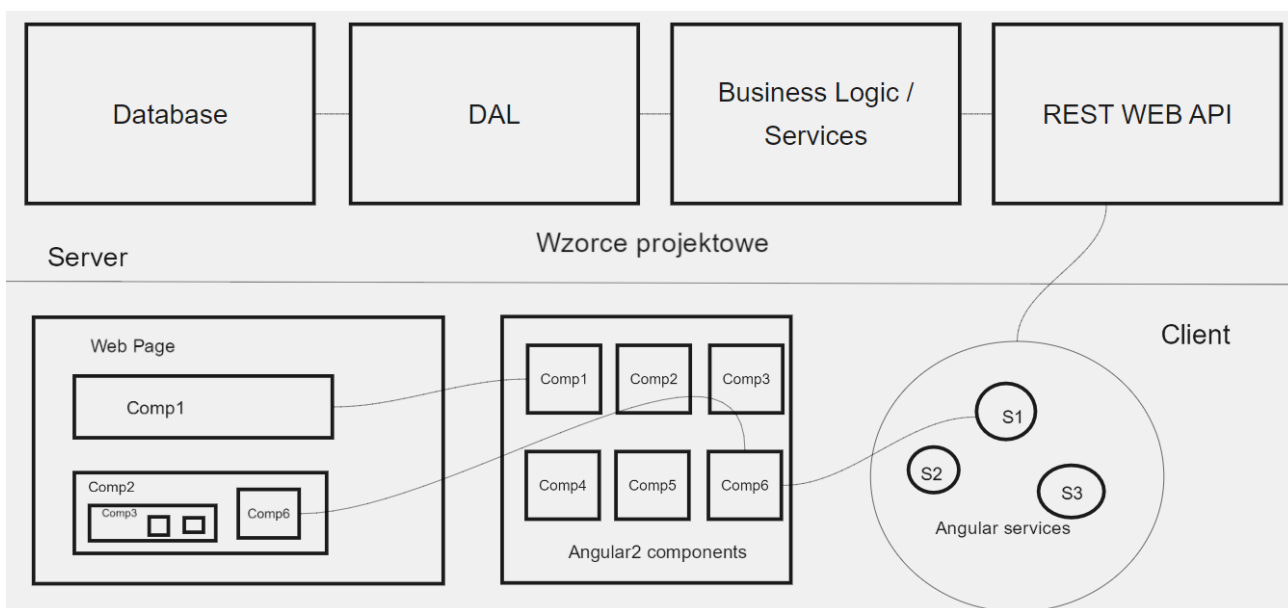
Rysunek 37. Komponent ładowania plików transakcji bankowych

### 3.2. Warstwy aplikacji

Na Rysunku 38 można zobaczyć warstwy aplikacji Finances. Aplikacja składa się z dwóch części – strony klienta oraz serwerowej.

Po stronie klienta mamy stronę HTML, 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 używania serwisów, czyli logiki współdzielonej. W Finances głównym serwisem jest *http service*, celem którego jest wysyłanie danych do serwisów restowych uruchomionych na serwerze.





*Rysunek 38. Warstwy aplikacji Finances*

Po stronie serwera mamy mechanizm ASP 5 WEB API, który przyjmuje zapytania HTTP ze strony klienta oraz wysyła odpowiedź do zapytania odpowiedź. W najprostszym przypadku (Rysunek 29), po otrzymaniu zapytania mechanizm restowy używa wprost DAL, w sercu którego leży wzorec repository, który obrabia oraz zwraca dane z bazy danych. Także w serwisie rest odbywa się mapowanie encji modelu bazy danych do view modelu [9].

```
[HttpGet]
public JsonResult GetUserTags()
{
    IEnumerable<Tag> tags = _repo.GetUserTags(User.GetUserId());

    if (tags == null) return Json(null);

    return Json(Mapper.Map<IEnumerable<TagViewModel>>(tags));
}
```

*Rysunek 39. Najprostszy Action w kontrolerze restowym*

Obecne są także bardziej złożone ścieżki po stronie serwera, na przykład użycie logiki biznesowej oraz serwisów, które są już odpowiedzialne za komunikację z DAL (Rysunek 40).

```

[HttpPost("upload")]
public JsonResult Upload()
{
    var files = Request.Form.Files;
    var uploads = Path.Combine(_environment.WebRootPath, "uploads");
    var user = _repo.GetUserById(User.GetUserId());

    foreach (var file in files)
    {
        if (file.Length > 0)
        {
            var fileName = ContentDispositionHeaderValue.Parse(file.ContentDisposition).FileName.Trim(' ');
            file.SaveAs(Path.Combine(uploads, fileName));
            if (!transactionImportService.ImportTransactions(fileName, user)) return Json(false);
        }
    }
    return Json(true);
}

```

*Rysunek 40. Złożony Action w kontrolerze restowym*

### 3.3. 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 Jak na Rysunku 41:

```

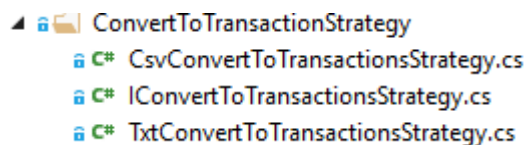
public interface IFinancesRepo
{
    IEnumerable<Transaction> GetCurrentUserTransactions(string currentUserId);
    void AddTransaction(Transaction trans, string currentUserId);
    void AddTransactions(List<Transaction> transactions);
    bool SaveAll();
    User GetUserById(string currentUserId);
    void RemoveTransaction(int id);
    void RemoveTag(int id);
    void AddTag(Tag tag, string userId);
    IEnumerable<Tag> GetUserTags(string userId);
    void ChangeUser(User user, string userId);
    void UpdateTransaction(Transaction transaction);
    Tag GetTagById(int id);
}

```

*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):



```
ConvertToTransactionStrategy
├── CsvConvertToTransactionsStrategy.cs
├── IConvertToTransactionsStrategy.cs
└── TxtConvertToTransactionsStrategy.cs
```

*Rysunek 42. Struktura plików według użytego wzorca*

Interfejs strategii jest bardzo prosty (Rysunek 43):

```
public interface IConvertToTransactionsStrategy
{
    List<Transaction> Convert();
}
```

*Rysunek 43. Interfejs strategii*

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

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



## **Zakończenie**

Podsumowując swoją pracę, mam nadzieję, że zrealizowałem postawiony na początku cel. W pierwszym rozdziale zrobiłem krótki opis platformy .Net oraz języka C#, ponadto 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 zaprezentowałem swoją aplikację, która używa najbardziej nowoczesnych technologii Google oraz Microsoft na dany moment. Także przedstawiłem wzorce projektowe, których używałem przy tworzeniu aplikacji.

Starłem się, aby z dwudziestu trzech wzorców projektowych opisanych dekady temu wybrać te najważniejsze, najczęściej używane, które każdy deweloper musi brać pod uwagę podczas pracy nad projektem programistycznym. Jestem bardzo wdzięczny za pomoc doktora Andrzeja Bobyka, który pomógł mi ściśle określić cel pracy. Dzięki tej pracy zdobyłem doświadczenie, którego 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*, Addison-Wesley Professional 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*, 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*, 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.