**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

[Wstęp 2](#_Toc328569987)

[Rozdział 1. ZAPOZNANIE Z POJĘCIAMI ORAZ TECHNOLOGIAMI 3](#_Toc328569988)

[1.1. .NET / C# 3](#_Toc328569989)

[1.2. Programowanie Obiektowe, pardadygmaty 4](#_Toc328569990)

[1.3. SOLID 6](#_Toc328569991)

[1.4. UML 8](#_Toc328569992)

[1.5. Problemy projektowania nowoczesnych systemów informatycznych 11](#_Toc328569993)

[Rozdział 2. Wzorce Projektowe 14](#_Toc328569994)

[2.0. Dwie książki 14](#_Toc328569995)

[2.1. Pojęcie wzorca projektowego 14](#_Toc328569995)

[2.2. Klasyfikacja wzorców, użyteczne tabele 14](#_Toc328569995)

[2.3. Factory Method 14](#_Toc328569995)

[2.4. Abstract Factory 14](#_Toc328569995)

[2.5. Singleton 14](#_Toc328569995)

[2.6. Adapter 14](#_Toc328569995)

[2.7. Facade 14](#_Toc328569995)

[2.8. Composite 14](#_Toc328569995)

[2.9. Proxy 14](#_Toc328569995)

[2.10. Command 14](#_Toc328569995)

[2.11. Iterator 14](#_Toc328569995)

[2.12. Observer 14](#_Toc328569995)

[2.13. Strategy 14](#_Toc328569995)

[2.14. MVC, MVP 14](#_Toc328569995)

[Rozdział 3. Napisanie aplikacji webowej z użyciem wzorców 28](#_Toc328570005)

[3.1. Krótki opis użytych technologii 28](#_Toc328570006)

[3.2. Aplikacja ze strony biznesu 28](#_Toc328570006)

[3.3. Projektowanie aplikacji 28](#_Toc328570006)

[3.4. Szczegóły implementacji 28](#_Toc328570006)

[Zakończenie 42](#_Toc328570009)

[Bibliografia 43](#_Toc328570010)

# Wstęp

Celem mojej pracy jest przedstawienie najważnieszych 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 przerażającym 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 pozióm 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 danną chwile technologii.

# Rozdział 1. ZAPOZNANIE Z POJĘCIAMI ORAZ TECHNOLOGIAMI

## .NET / C#

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

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

C# kompiluje się w CIL (Common Intermidiate Language). Poprzednie zdanie to kompletna bzdura, o której propaguje większość nieświadomych ‘specjalistów’ .NET. 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 (Just In Time). Także w wykonaniu uczęstniczy 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 krosplatformowości 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 standardyzowany jako ECMA-334 oraz ISO/IEC 23270. C# tak samo jak Java oraz C++ należy do rodziny języków z C-podobnym syntaksysem. Pan Anders Hejlsberg, twórca C#, złączył najlepsze konstrukcje syntaksyczne z innych języków programowania. Pod tym względem jest to unikatowy oraz na danny moment najbogatszy syntaksycznie język programowania. To znaczy, że programiści C# mają bogaty wybór, wiele możliwości realizacji a najgłówniejsze łatwość realizacji. To pozwala pisać aplikacje szybciej, bardziej 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 napisanie najbardziej prostych programów z użyciem zmiennyh, pętli, funkcji. Modelowanie pozwala sformować wysykopoziomowe, 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.

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

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

Dobre zrozumienie każdego paradygmaty można otrzymać z książki Code Complete.

Hermetyzacja(Encapsulation) – ukrycie informacji.

Jest kylka rodzajów ukrycia informacji a stąd hermetyzacji:

* Ukrycie typów danych

Wykorzystanie dynamic oraz var.

* Ukrycie realizacji

Wykorzystanie modyfikatorów dostępu wewnątrz klasy, rzutowanie do typu wyzszego 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(Inheritance) – możliwość zdefiniować nowe klasy według istniejących. Możemy dziedziczyć klasy oraz interfejsy. Ważnym jest rozumienie róźnicy 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(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(Abstraction) – celem jest formowanie zbiorowych pojęć.

Message passing – organizacja potoków informacyjnych między obiektami. Pozwala jednemu obiektu przekazać powiadomienie innemu.

Reusable – przy pomocy tworzenia klas, metod, dziedziczenia, kompozycji. Powtórne wykorzystanie przy pomocy tworzenia podklas nazywa się przejzorczystym 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 zadady bazują się na wzorcach projektowych. Warto zauważyć, że jest dużo więcej zasad opisanych we wzorcach. Nie wiadomo czemu dokładnie te 5 zasad były wyróźnione 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.

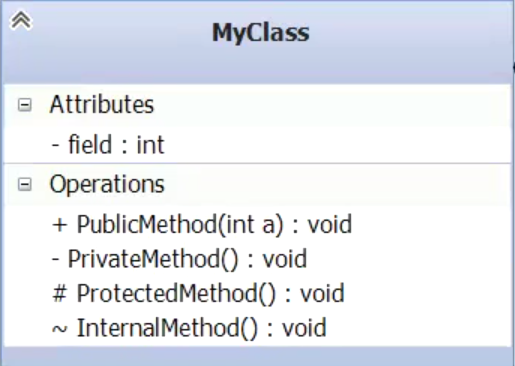
* Dependency inversion

Abstragowanie procesu tworzenia obiektu przy pomocy użycia abstraktnych klas produktów.

## UML

UML(Unified Modeling Language) – pozwala przedstawić system w wyglądzie diagram.

Jest wiele rodzajów UML. W danej pracy jest używany diagram klasów, żeby pokazać związki między klasami we wzorcach.



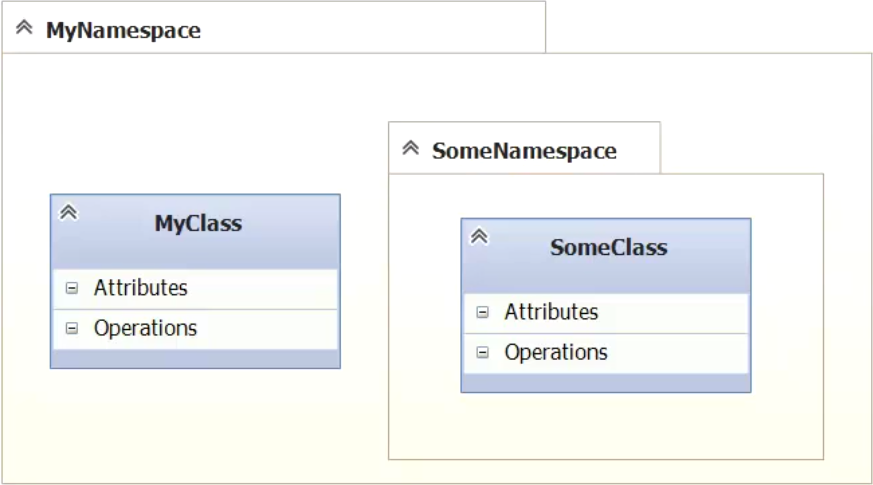
Rysunek 1

Na rysunku 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.

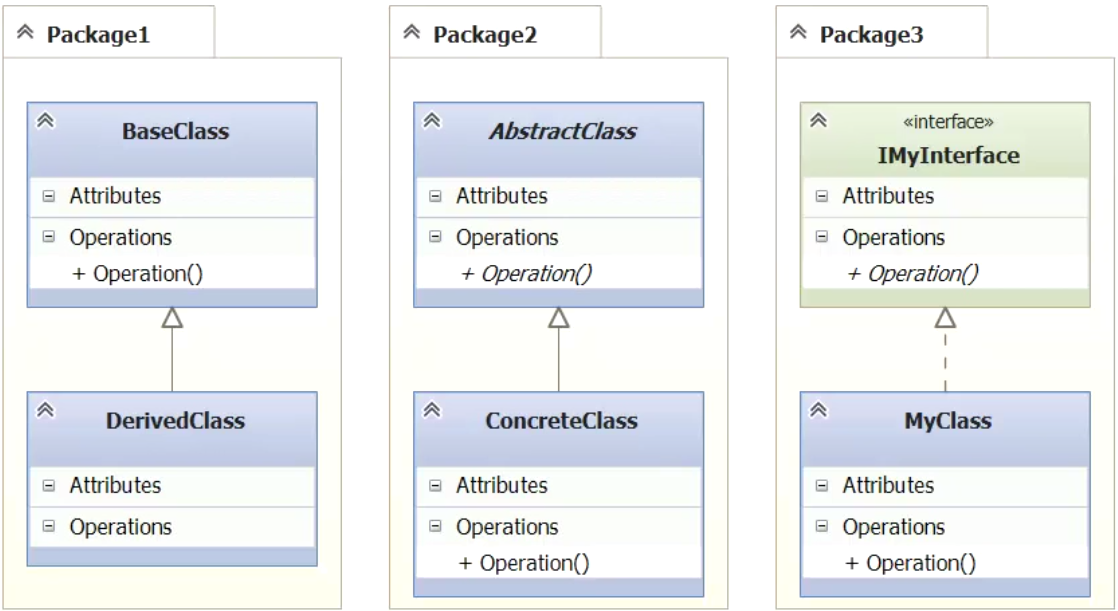
Także, jeżeli mówimy o opisaniu stereotypów, w języku UML mamy klasyfikator ‘Interface’ oraz klasyfikator ‘Enumeration’.

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



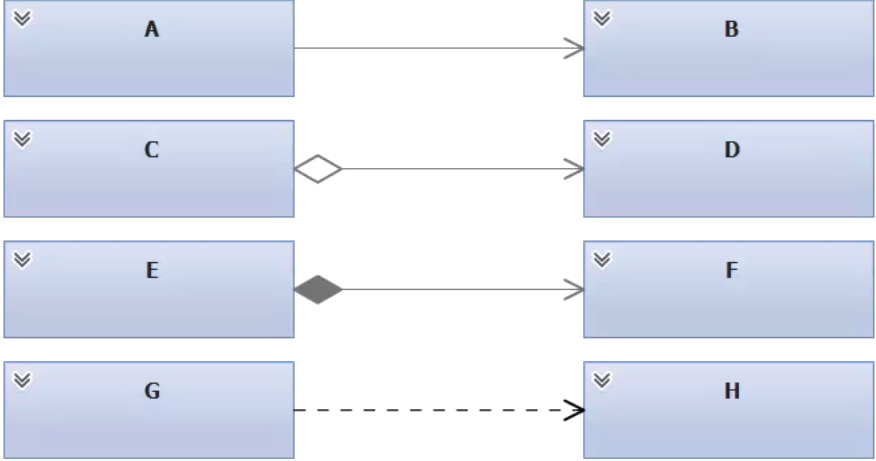
Rysunek 2

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



Rysunek 3

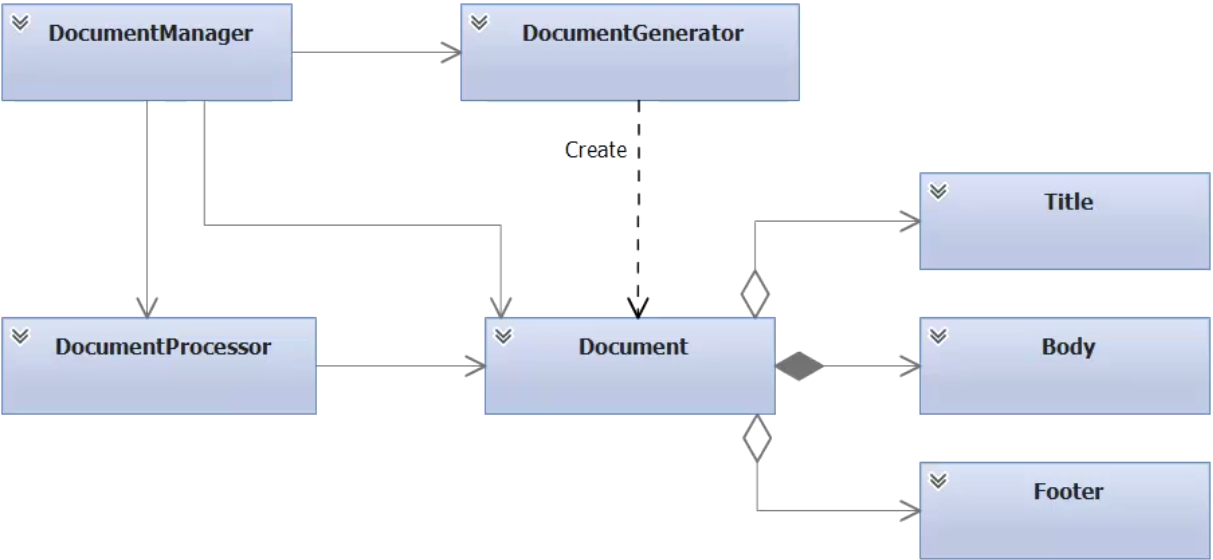
Na rysunku 3 widzimy dziedziczenie w języku UML. Warto zauważyć że w przypadku ostatnim my nie tylko realizujemy interfejs ale też go dziedziczymy. Pierwsze dwa dziedziczenia nazywamy dziedziczeniem realizacji, trzeci nazywamy dziedziczeniem interfejsu.



Rysunek 4

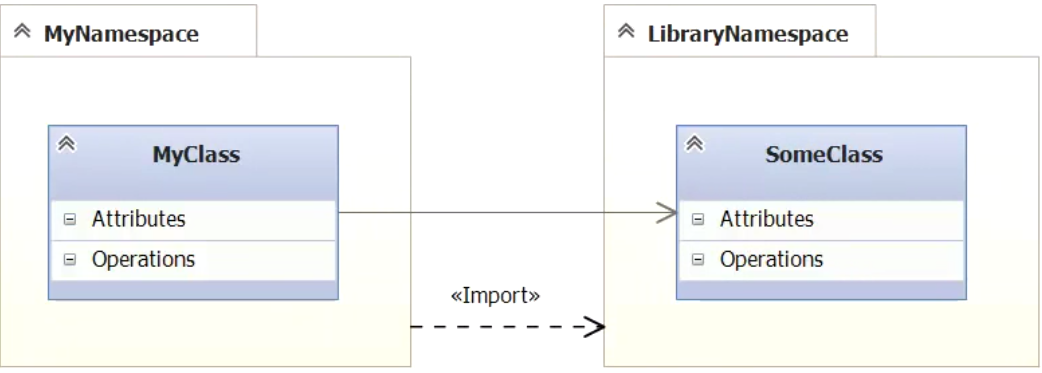
Na rysunku 4 widzimy jeszcze cztery rodzaje związków między klasami:

* 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 związkam. Jest 13 stereotypów zależności.



Rysunek 5

Na rysunku widzimy przykład wykorzystania wszystkich przedstawionych wcześcniej związków oprócz dziedziczenia.



Rysunek 6

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.

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 klasów jeden od drugiego.

Zależność (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 egzemplaru podobnej klasy będzie zależeć od pracy egzemplarów 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 dwuch postaciach: dobre – biznesowe przywiązania lub złe – przywiązania techniczne.

Przywiązania biznesowe wyrażają wymagania biznesu. Naprzykład 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 róźnych warstwach systemu. Naprzykład 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 minimizować ilość przywiązań ale też 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 pozióm abstrakcji w tworzeniu systemów programistyczncyh. Każdy wzorzec to zestaw kompleksnych procesów oraz komend który daje nam odpowiedni rezultat. W dzisiejszym świecie programistycznym mówimy wzorcami. Już nie musimy zastanawiać się nad szczegółami implementacji tylko korzystać z gotowego mechanizmu.

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, użyteczne tabele

\* SKRADNIETE Z WIKI

Początkowo *Banda Czterech* założyła, że istnieją dwa podstawowe rodzaje klasyfikacji wzorców. Pierwsza z nich dotyczy rodzaju wzorca i opisuje to, co on robi. Według tego kryterium wzorce możemy podzielić na trzy rodziny:

* **kreacyjne (konstrukcyjne)** – opisujące proces tworzenia nowych obiektów Ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów, klas oraz innych typów danych
* **strukturalne** – opisujące struktury powiązanych ze sobą obiektów;
* **czynnościowe (behawioralne, operacyjne)** – opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów.

Drugi model dzieli wzorce na kategorie według ich zakresów. Kategoryzacja polega na określeniu czy wzorzec dotyczy klas, czy obiektów. Tutaj wzorce dzielimy na:

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

## Factory Method

Nazwa: Factory method

Drugie nazwy: Virtual Constructor

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

Jak często się używa: bardzo często

Przeznaczenie: być fundamentem wszystkich wzorców kreacyjnych

Opis – 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 ztworzyć.

\*Strona architekta:

Architekt tworzy abstraktowy szkielet mechanizmu tworzenia produktu.

Architekt deleguje realizatoru 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.

Kiedy użyć:

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 kreacyjnych wzorców.

C# osobliwości:

Wykorzystanie Generics – pozwala nie tworzyć kylka ConcreteCreator klasów, a 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>();

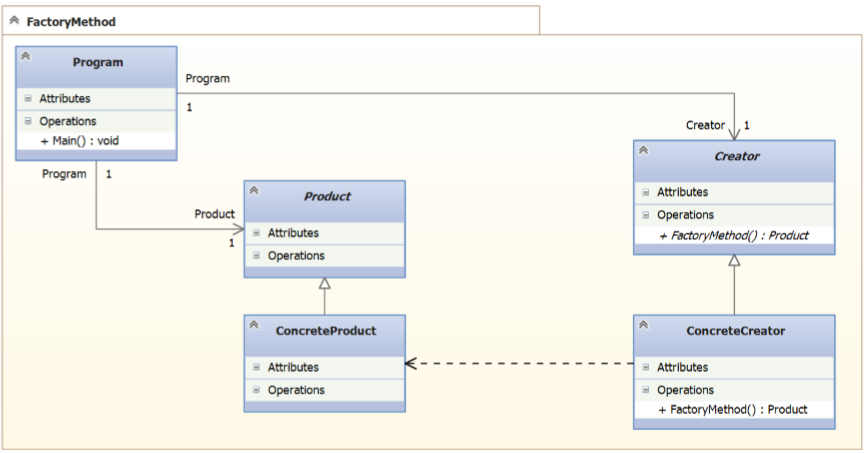
Nazewnictwo:

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

Wykorzystanie w .NET:

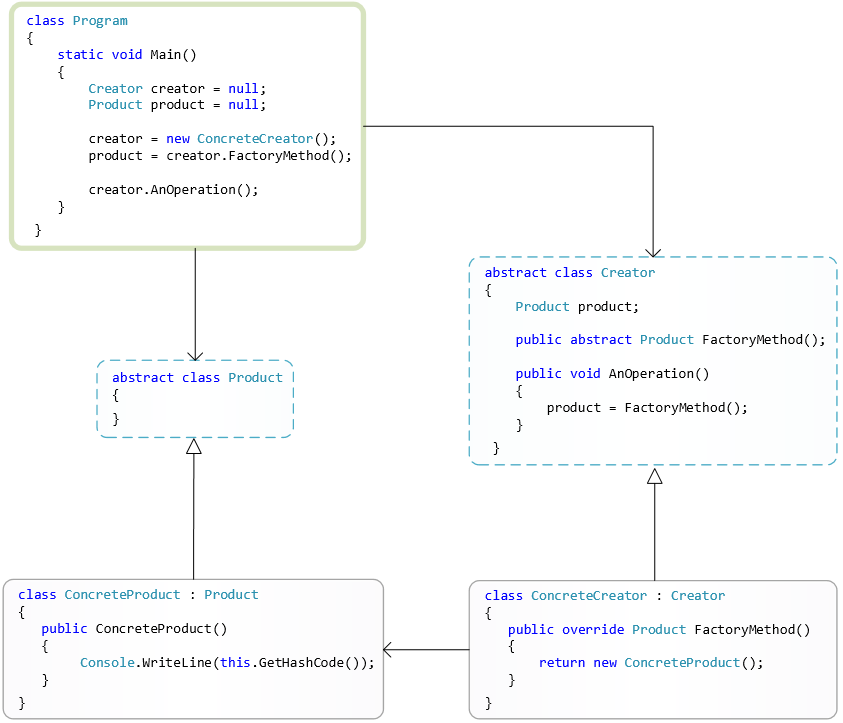
Jest fundamentem wszystkich kreacyjnych wzorców, czyli jest wykorzystywany wszędzie gdzie można spotkać wzorce kreacyjne.

UML



Rysunek

C#



Rysunek

## Abstract Factory

Nazwa: Abstract Factory

Drugie nazwy: Kit

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

Jak często się używa: bardzo często

Przeznaczenie: Tworzenie rodzin współdziałających obiektów.

Opis – wzorzeć opisuje algorytm tworzenia interfejsu pozwalającym instancjować rodziny powiązanych lub zależnych między sobą obiektów-produktów. 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 butełkę lub puszkę. Te dwa produkty muszą między sobą współpracować żeby otrzymać ostateczny produkt. Przykładem antywzorcu może być próba użyć produkt 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 http://msdn.microsoft.com/library/system.activities.presentation.model.modelfactory.aspx

System.Data.Common.DbProviderFactory

http://msdn.microsoft.com/library/system.data.common.dbproviderfactory.aspx

System.Data.EntityClient.EntityProviderFactory http://msdn.microsoft.com/library/system.data.entityclient.entityproviderfactory.aspx

System.Data.Odbc.OdbcFactory

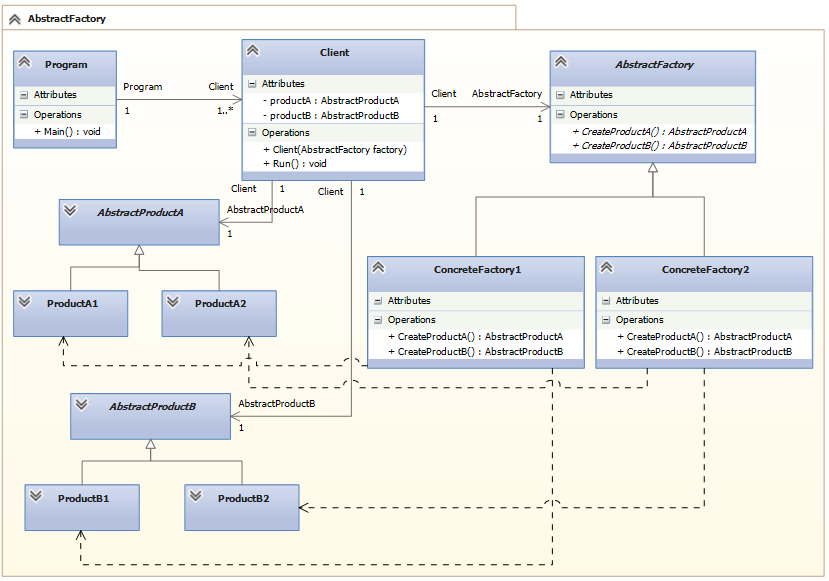
http://msdn.microsoft.com/library/system.data.odbc.odbcfactory.aspx

System.Data.OleDb.OleDbFactory

http://msdn.microsoft.com/library/system.data.oledb.oledbfactory.aspx

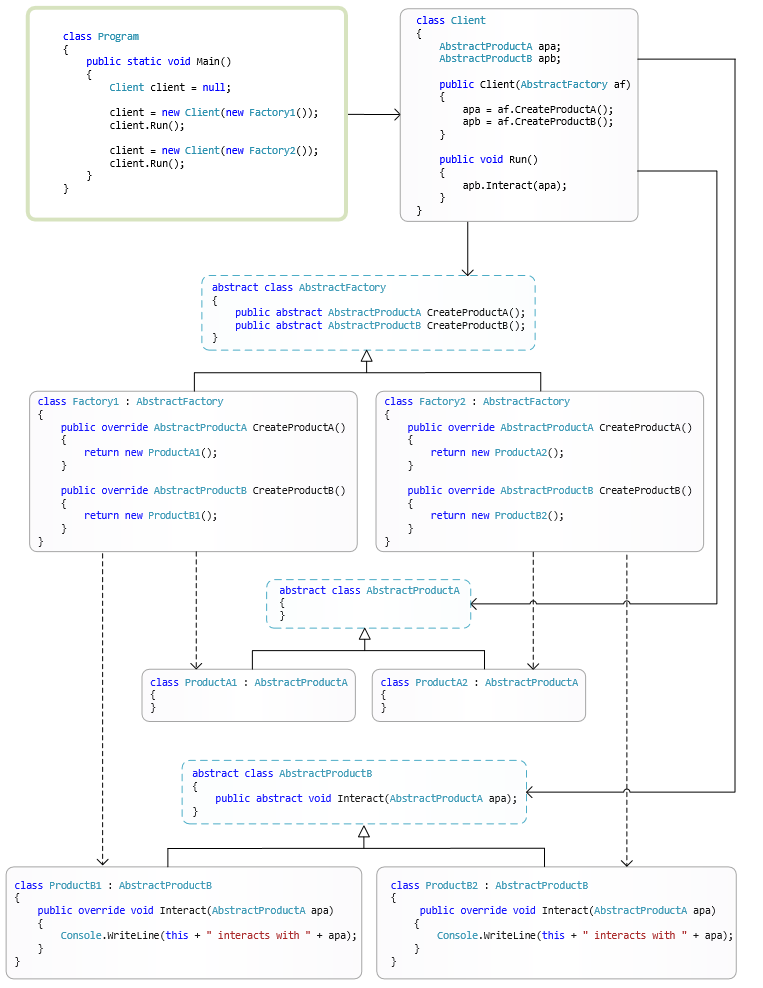
Oraz inne

UML



Rysunek

C#



Rysunek

## Singleton

Nazwa: Singleton

Drugie nazwy: Solitaire

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

Jak często się używa: często

Przeznaczenie: gwarantuje instancjowanie wyłącznie jednego egzemplarza klasy

Opis – metaforycznie możemy sobie wyobrazić Słońce. 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 wzorcu architekt może być pewien tego, że programista nie stworzy zbędny egzemplarz klasy.

Kiedy użyć:

Użyć kiedy w systemie musi być ograniczona ilość egzemplarów jednej klasy

C# osobliwości:

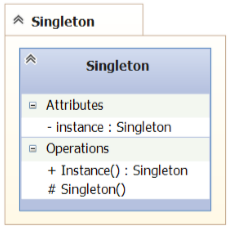
Jedną z możliwości realizacji tego wzorcu w C# jest użycie klas statycznych. Według mnie jest to preferowany sposób realizacji, aczkolwiek ma swoje wady: nie pasuje w przypadku potrzeby tworzenia kylkóch instancji klasy singleton. Klasy statyczne nie mogą uczęstniczać w dziedziczeniu a to znaczy nie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

Przy użyciu wielowątkowości zostaje ryzyko tworzenia kylkóch egzemplarów. To ryzyko można zeliminować przy pomocy konstrukcji lock(){} wewnątrz której i będzie instancjacja egzemplaru obiektu.

Wykorzystanie w .NET

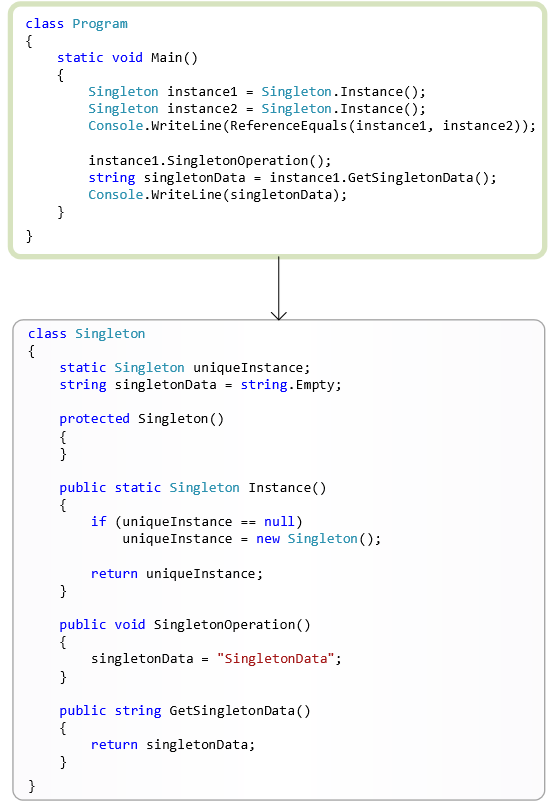
* System.ServiceModel.ServiceHost
* System.Data.DataRowComparer

UML



Rysunek

C#



Rysunek

## Adapter

## Facade

## Composite

## Proxy

## Command

## Iterator

## Observer

## Strategy

## MVC, MVP

# Bibliografia

|  |  |
| --- | --- |
| [1] | Design Patterns Via C# |
| [2] | CLR Via C# |
| [3] | Code complete |
| [4] | Framework Design Guidelines |