Wzorce projektowe w C#

Marcin Sulecki
marcin.sulecki@gmail.com
github.com/sulmar

VAVATECH

Agenda

- Manifest Agile
- Reguły wytwarzania oprogramowania
- Zasady SOLID
- Wzorce projektowe
- Wzorce złożone

Manifest Agile

Zasady

Ludzie i interakcje
ponad procesy i narzędzia

Działające oprogramowanie
ponad szczegółową dokumentację

Współpraca z klientem
ponad ustalenia w umowie

Reagowanie na zmiany
ponad realizacją planu

agilemanifesto.org

Reguły wytwarzania oprogramowania

Reguły wytwarzania oprogramowania

DRY

(ang. Don't Repeat Yourself) – nie powtarzaj się

KISS

(ang. Keep It Simple, Stupid) – nie komplikuj

Prawo Demeter

Zasada minimalnej wiedzy - rozmawiaj tylko z (bliskimi) przyjaciółmi

Worse is Better

(pol. Gorsze jest lepsze) - prostota implementacji jest ważniejsza niż prostota interfejsu. System powinien skupić się na typowych przypadkach.

Zasady SOLID

Zasady SOLID

Zasada pojedynczej odpowiedzialności

***S**ingle-Responsibility Principle* (SRP)

Zasada otwarte-zamknięte

***O**pen/Closed Principle* (OCP)

Zasada podstawiania Liskov

***L**iskov Substitution Principle* (LSP)

Zasada segregacji interfejsów

***I**nterface Segregation Principle* (ISP)

Zasada odwracania zależności

***D**ependency Inversion Principle* (DIP)

Zasada pojedynczej odpowiedzialności

„Każda klasa powinna być odpowiedzialna za jedną konkretną rzecz"

Zasada otwarte-zamknięte

„Każda klasa powinna być otwarta na rozbudowę ale zamknięta na modyfikację"

Zasada podstawiania Liskov

„Musi istnieć możliwość zastępowania typów bazowych ich podtypami"

Zasada podstawiania Liskov

Historia

Zasada sformułowana przez Barbarę Liskov w 1987 roku.

Wnioski

- Rozszerzając, nie modyfikujmy sposobu działania istniejącego kodu
- W miejscu klasy bazowej można zawsze użyć dowolnej klasy pochodnej

Zasada segregacji interfejsów

„Interfejsy powinny być małe i konkretne
aby klasy nie implementowały metod, których nie potrzebują"

Zasada odwracania zależności

„Wszystkie zależności powinny w jak największym stopniu zależeć od abstrakcji a nie od konkretnego typu"

Zasada odwracania zależności

Zasada

1. Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji.
2. Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.

Przepis

1. Żadna zmienna nie powinna zawierać referencji do konkretnej klasy.
2. Żadna klasa nie powinna dziedziczyć po konkretnej klasie.
3. Żadna metoda nie powinna przykrywać metody zaimplementowanej w którejkolwiek z klas bazowych.

Wzorce projektowe

Podział wzorców projektowych

Wzorce kreacyjne (*creationals patterns*)

opisują tworzenie nowych obiektów klasy

Wzorce strukturalne (*structurals patterns*)

opisują budowę powiązań między obiektami

Wzorce czynnościowe (*behaviorals patterns*)

opisują zachowanie obiektów, które ze sobą współpracują

Wzorce projektowe

Wzorce kreacyjne
Abstract Factory
Builder
Factory Method
Prototype
Singleton

Wzorce strukturalne
Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Wzorce czynnościowe
Chain of Resp.
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Wzorce kreacyjne

Opisują tworzenie nowych obiektów klasy

Wzorce kreacyjne

Fabryka abstrakcyjna (Abstract Factory)

Umożliwia wytworzenie obiektu

Budowniczy (Builder)

Umożliwia rozdzielenie tworzenia skomplikowanych obiektów od ich reprezentacji.

Metoda wytwórcza (Factory Method)

Tworzy szkielet algorytmu, który jest w pełni realizowany dopiero w klasach potomnych.

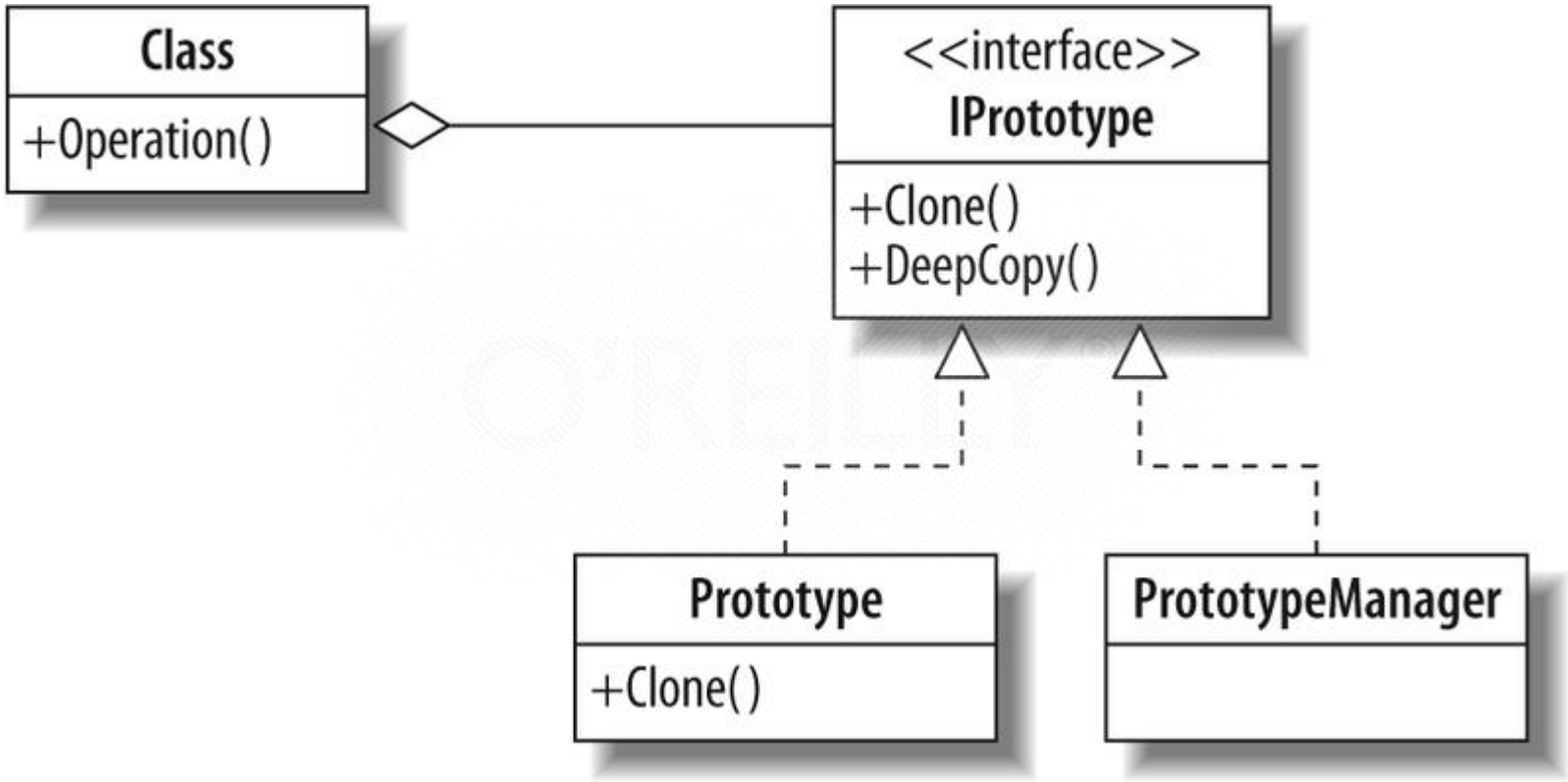
Prototyp (Prototype)

Wytwarza obiekt na podstawie innego obiektu zwanego prototypem poprzez klonowanie

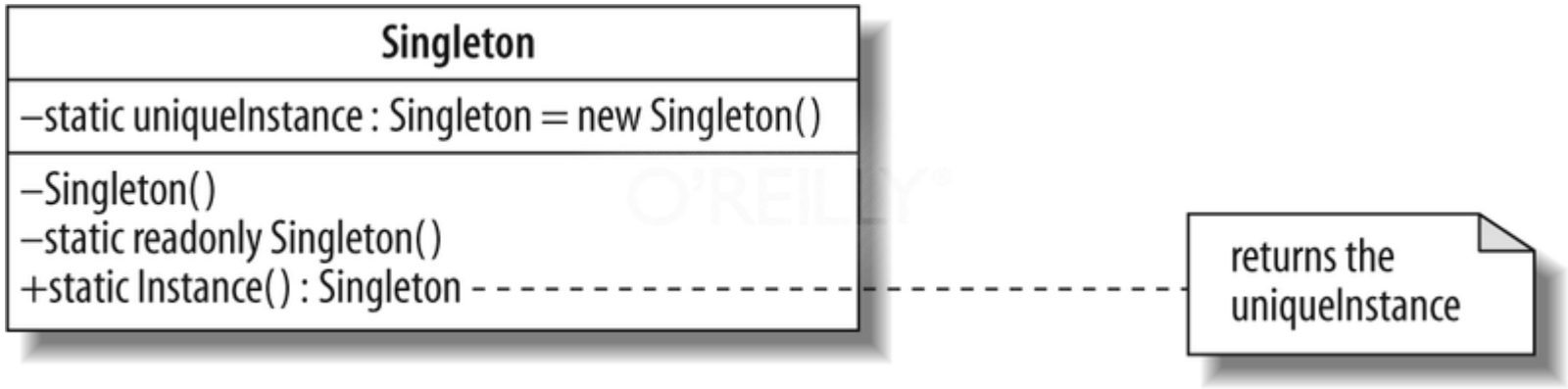
Singleton

Pilnuje, aby klasa była reprezentowana maksymalnie przez jedną instancję

Prototype


Class
+Operation()
<<interface>>
IPrototype
+Clone()
+DeepCopy()
Prototype
+Clone()
PrototypeManager

Singleton


Singleton
–static uniqueInstance : Singleton = new Singleton()
–Singleton()
–static readonly Singleton()
+static Instance() : Singleton
returns the uniqueInstance

Wzorce strukturalne

Opisują budowę powiązań między obiektami

Wzorce strukturalne

Adapter

Umożliwia wykorzystanie istniejącej klasy o niekompatybilnym interfejsie

Most (Bridge)

Pozwala na modyfikowanie implementacji w czasie działania programu

Composite

Składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt

Decorator

Wzbogaca klasę o nowe funkcjonalności w trakcie działania programu (dynamiczne dziedziczenie)

Facade

Dostarcza uproszczony interfejs do podsystemów

Wzorce strukturalne c.d.

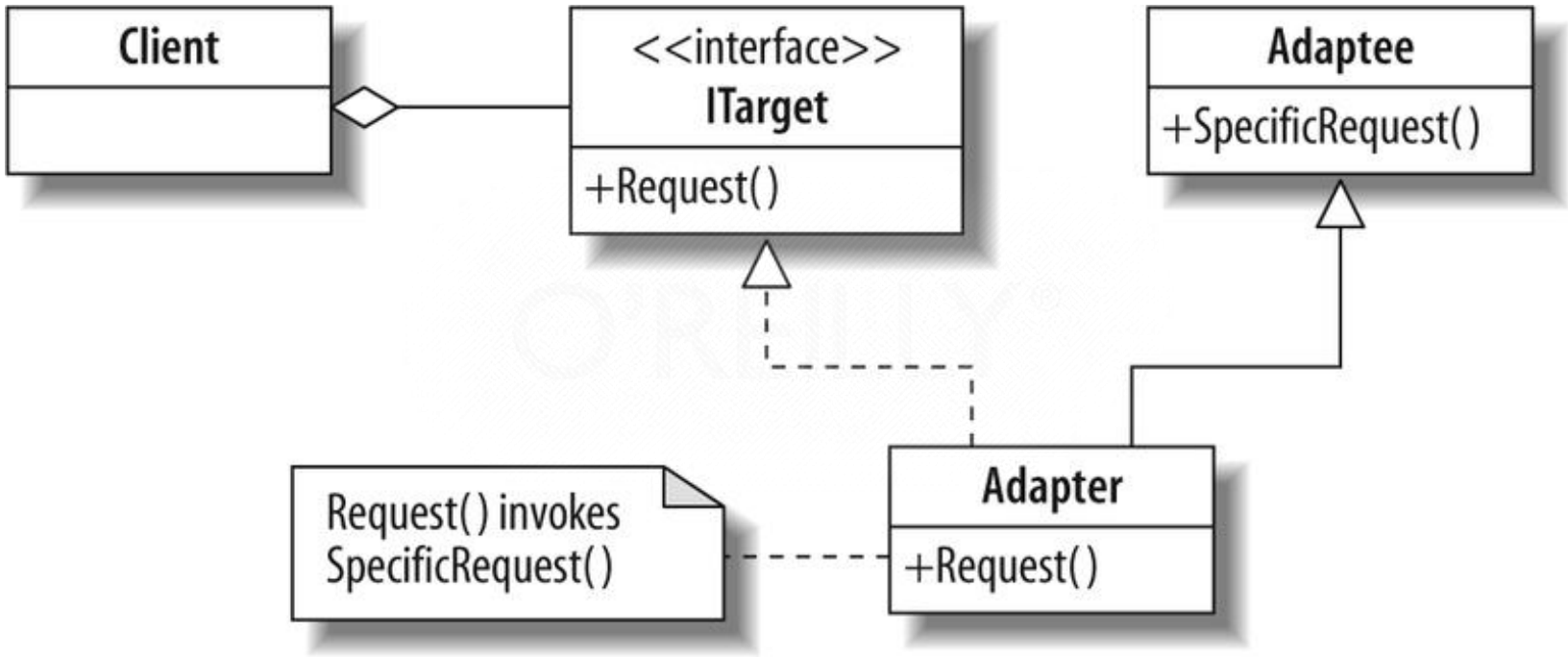
Flyweight

Umożliwia poprawę efektywności obsługi dużych obiektów zbudowanych z wielu mniejszych elementów poprzez współdzielenie wspólnych małych elementów

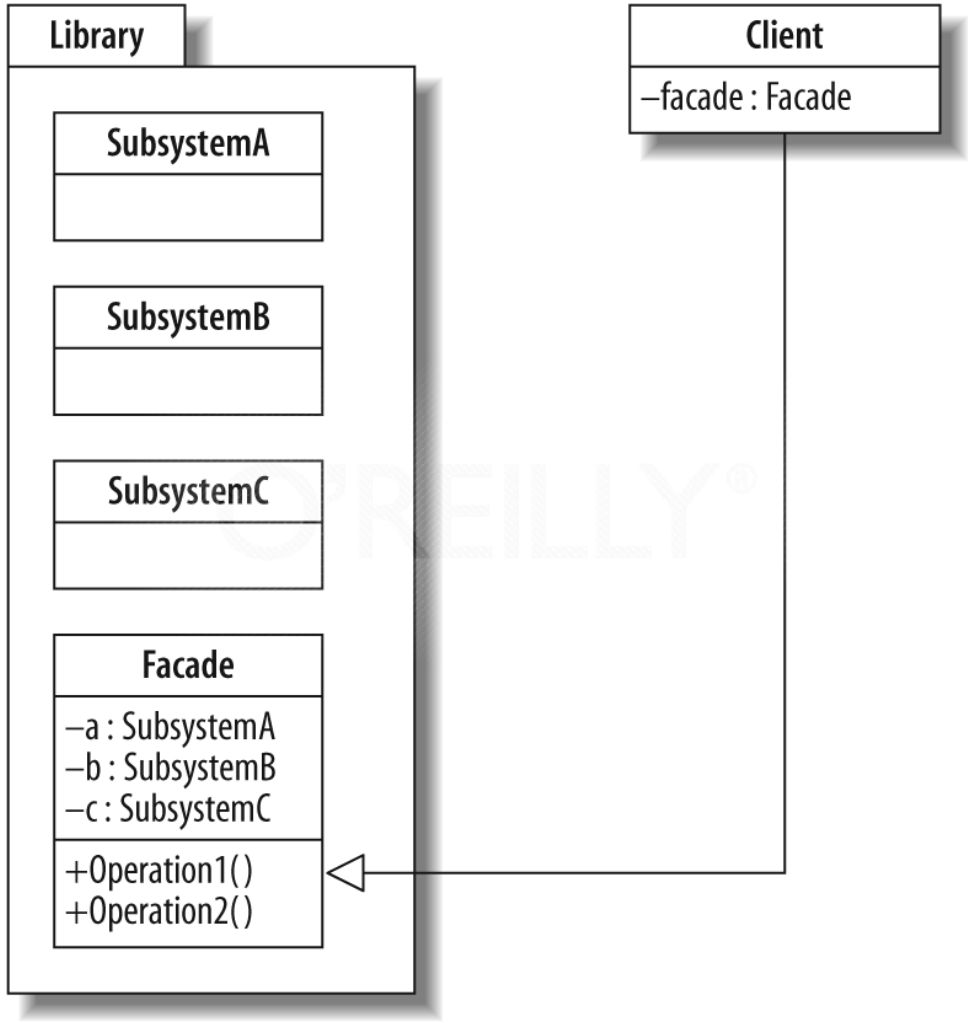
Proxy

Tworzy obiekt zastępujący inny obiekt i umożliwia kontrolowanie do niego dostępu

Adapter


Client
<<interface>>
ITarget
+Request()
Adaptee
+SpecificRequest()
Request() invokes
SpecificRequest()
Adapter
+Request()

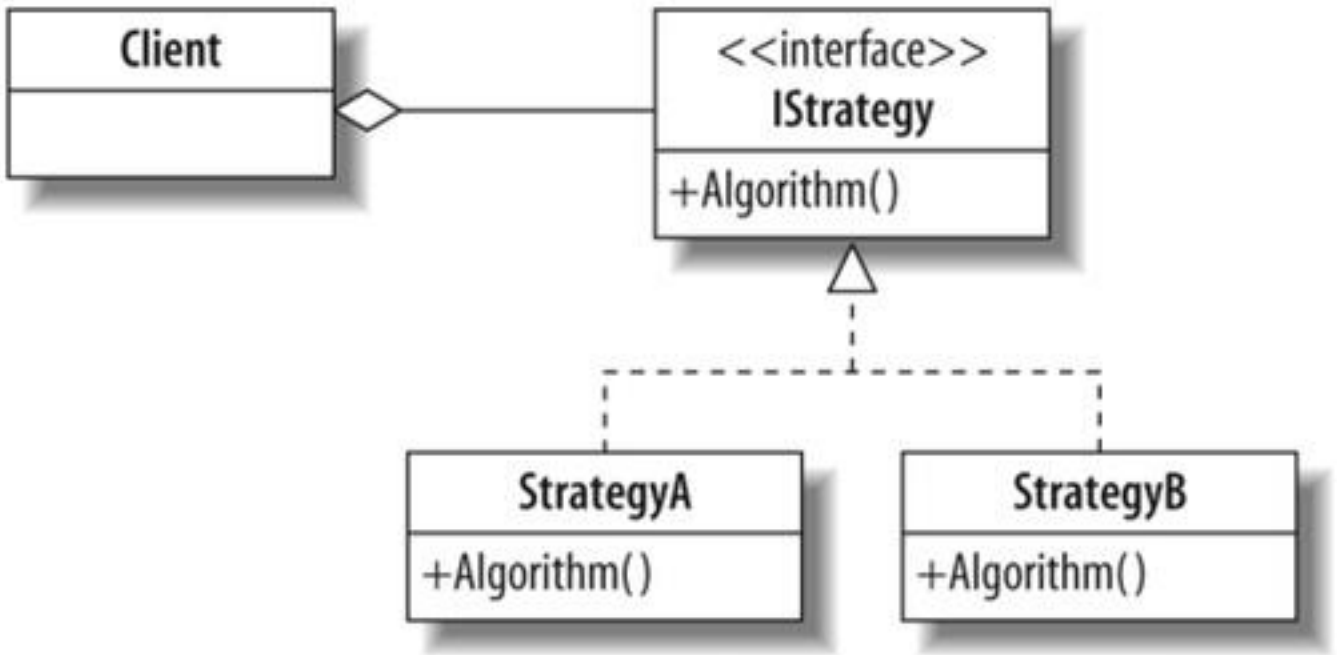
Facade


Library
SubsystemA
SubsystemB
SubsystemC
Facade
–a : SubsystemA
–b : SubsystemB
–c : SubsystemC
+Operation1()
+Operation2()
Client
–facade : Facade

Wzorce czynnościowe

Opisują zachowanie obiektów, które ze sobą współpracują

Strategy


Client
<<interface>>
IStrategy
+Algorithm()
StrategyA
+Algorithm()
StrategyB
+Algorithm()

Wzorce czynnościowe

Adapter

Dostosowuje inne struktury obiektów do siebie.

Command

Czynność podniesiona jest do rangi klasy.

Interpreter

Umożliwia stworzenie interpretera, który będzie przetwarzać polecenia w określonej gramatyce

Iterator

Umożliwia iterację po strukturze obiektu.

Mediator

Umożliwia w scentralizowany sposób powiadamianie wszystkich znanych mu obiektów

Wzorce czynnościowe c.d.

Memento

Przechowuje stan obiektu w celu późniejszego jego wykorzystania

Observer

Informuje obserwatorów o zmianach w obserwowanym obiekcie

State

Maszyna stanów

Strategy

Zachowanie obiektu jest przekazanie z zewnątrz

Template Method

Tworzymy zarys zachowania obiektu, a szczegółowe działanie będzie opisane w klasie potomnej.

Wzorce czynnościowe c.d.

Visitor

Umożliwia przejście po strukturze obiektu

Chain of Responsibility

Umożliwia przetwarzania żądania za pomocą łańcucha obiektów (handlerów)

Złożone wzorce projektowe

Złożone wzorce projektowe

MVC (Model-View-Controller)

MVP (Model-View-Presenter)

MVVM (Model-View-ViewModel)

MVC (Model-View-Controller)

MVP (Model-View-Presenter)

MVVM (Model-View-ViewModel)

Materiały

GitHub

https://github.com/sulmar/Vavatech.201710.DesignPatterns

Zapraszam na szkolenia

WPF + MVVM

Tworzenie aplikacji WPF zgodnie ze wzorcem MVVM

UWP

Tworzenie aplikacji uniwersalnych dla Windows 10

Entity Framework 6

Tworzenie warstwy dostępu do danych

WebApi

Tworzenie usług sieciowych REST API

Crystal Reports

Tworzenie raportów


VAVATECH