

# Zasady SOLID

# Paradygmaty programowania

# Paradygmaty programowania

- programowanie imperatywne
- programowanie proceduralne
- programowanie strukturalne
- programowanie obiektowe
- ...
- programowanie deklaratywne
- programowanie funkcjonalne
- programowanie genetyczne
- programowanie kwantowe

# Zasady dobrego programowania

## Zasady dobrego programowania

- Keep it Simple Stupid (KISS) - BUZI (Bez Udziwnień Zapisu, Idioto)
- Don't Repeat Yourself (DRY)
- Tell Don't Ask
- You Aren't Gonna Need It (YAGNI)
- Separation of Concerns

# KISS

- kod ma być prosty i zrozumiały
- unikanie skomplikowanych zapisów
- nie oznacza upraszczania za wszelką cenę

Program - KISS.

## Don't Repeat Yourself (DRY)

- unikanie powtórzeń w kodzie – u nas to np. polimorficzność, stałe
- oddzielenie powtarzającej się części od metody
- zalety: unikanie błędów, poprawki możemy dodać w jednym miejscu

Program - DRY.

## Tell Don't Ask

- związane z hermetyzacją i podziałem obowiązków
- należy mówić obiektom jakie akcje mają wykonywać
- nie należy uzależnia wykonania akcji od stanu w jakim znajdują się obiekty

Program - TDA, TDAGOOD.



## You Aren't Gonna Need It (YAGNI)

- tworzenie tego co jest potrzebne i niezbędne
- usuwanie dodatków, które mogą się przydać
- np. usunięcie zbędnych usingów, zwolnienie zasobów, usuwanie nie używanych zmiennych, metod

Cytat: Antoine de Saint-Exupéry

*Perfekcję osiąga się wtedy, gdy nie można już nic odjąć, a nie dodać.*

## Separation of Concerns

- elementy składowe (np. klasy i metody) powinny być rozłączne i mieć oddzielne zastosowanie
- te elementy nie powinny współdzielić odpowiedzialności

Cytat: Albert Einstein

Kod: SCA1, SCA2.

*Wszystko trzeba robić tak prosto, jak to tylko jest możliwe, ale nie prościej.*

# Zasady SOLID

# Zasady SOLID

SOLID – mnemonik zaproponowany przez Roberta C. Martina, opisujący pięć podstawowych założeń programowania obiektowego:

- zasada jednej odpowiedzialności (ang. single responsibility),
- zasada otwarte-zamknięte (ang. open-close),
- zasada podstawienia Liskov (ang. Liskov substitution principle),
- zasada segregacji interfejsów (ang. interface segregation principle),
- zasada odwrócenia zależności (ang. dependency inversion principle).

## Zasada jednej odpowiedzialności

- każdy obiekt powinien mieć tylko jeden cel i odpowiedzialność
- nie powinien istnieć więcej niż jeden powód do modyfikacji klasy
- jeśli jest kilka odpowiedzialności, powinniśmy podzielić klasę
- podobne do Separation of Concerns

## Zasada otwarte-zamknięte

- klasa powinna być otwarta na rozbudowę ,ale zamknięta do jej własnej modyfikacji
- możemy dodawać nowe pola i metody, ale bez zmiany w wewnętrzzną strukturę
- zmiana istniejącej struktury może mieć wpływ na inne elementy
- hermetyzacja, dziedziczenie, polimorfizm, delegaty
- unikamy instrukcji warunkowych

## Zasada podstawienia Liskov

- powinniśmy być w stanie używać klasy pochodnej w miejsce klasy nadrzędnej i ona zachowuje się w taki sam sposób, bez modyfikacji
- klasa pochodna nie ma wpływu na zachowanie klasy nadrzędnej
- w hierarchii klas powinno dać się traktować obiekt klas pochodnych jak obiekt klas bazowej
- dziedziczenie ma być stosowane tylko gdy chcemy skorzystać z polimorfizmu, a nie tylko gdy chcemy wyciągnąć wspólne cechy

## Zasada segregacji interfejsów

- dzielenie interfejsów na mniejsze grupy, tak by klasa dziedzicząca po nich, nie miała do dyspozycji niepotrzebnych metod
- nie powinno być jednego dużego interfejsu



## Zasada odwrócenia zależności

- Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obie grupy modułów powinny zależeć od abstrakcji.
  - moduły wysokopoziomowe – logika biznesowa
  - moduły niskopoziomowe – komunikacją z bazą danych, ftp, API, algorytmy do liczenia

Abstrakcje nie powinny zależeć od szczegółowych rozwiązań. To szczegółowe rozwiązania powinny zależeć od abstrakcji.

<http://devman.pl/pl/techniki/zasady-solid-5-zasada-odwrocenia-zaleznosci-dependency-inversion/>

# Wzorce projektowe

## Wzorce projektowe

Wzorzec projektowy (ang. design pattern) – uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych. Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz pielęgnację kodu źródłowego. Jest opisem rozwiązania, a nie jego implementacją. Wzorce projektowe stosowane są w projektach wykorzystujących programowanie obiektowe.

## Elementy wzorca projektowego

Wzorzec projektowy składa się z czterech podstawowych elementów:

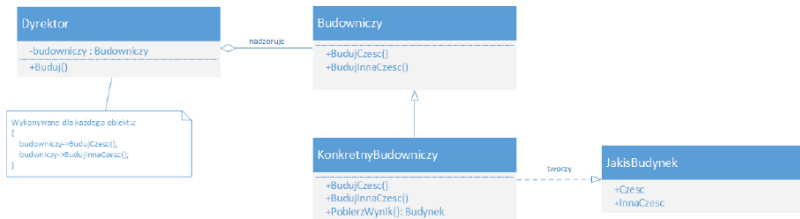
- nazwy wzorca;
- problemu – opisuje sposoby rozpoznawania sytuacji, w których możemy zastosować dany wzorzec oraz warunki jakie muszą zostać spełnione, by jego zastosowanie miało sens;
- rozwiązania – opisuje elementy rozwiązania: ich relacje, powiązania oraz obowiązki, zawiera także wskazówki implementacyjne dla różnych technologii;
- konsekwencji – zestawienie wad i zalet stosowania wzorca, uwzględniające informacje o jego brakach oraz kosztach rozwoju i utrzymania systemu wykorzystującego dany wzorzec.

## Wzorce kreatywności

Wzorce konstrukcyjne (kreatywne) pozwalają w sposób abstrakcyjny tworzyć i konfigurować obiekty w celu ich wielokrotnego użycia i zachowania niezależności systemu od sposobu ich tworzenia.

## Budowniczy

- cel: rozdzielenie sposobu tworzenia obiektów od ich reprezentacji
- proces tworzenia obiektu podzielony jest na kilka mniejszych etapów a każdy z tych etapów może być implementowany na wiele sposobów
- zastosowanie:
  - węzły XMLa
  - konwertowanie tekstu, zdjęć, filmów z jednego formatu na drugi.



Rysunek 1: Diagram UML Budowniczego

Kod do analizy: [https://github.com/pjastr/ZPO\\_programy/tree/master/Budowniczy/Budowniczy](https://github.com/pjastr/ZPO_programy/tree/master/Budowniczy/Budowniczy)



## Fabryka abstrakcyjna

Przeanalizujmy przykład: <http://tomaszjarzynski.pl/fabryka-abstrakcyjna-wzorzec-projektowy-abstract-factory/>

## Zastosowania:

- komunikacja z różnymi api
- wyświetlanie tekstu z różnych formatów plików
- obsługa różnych typów baz danych

# Metoda wytwórcza

Przykład do przeanalizowania: <http://tomaszjarzynski.pl/metoda-wytworcza-wzorzec-projektowy-factory-method/>

## Zastosowania:

- ukrycie przed klientem implementacji
- sterowniki

# Prototyp

Prototyp – kreatywny wzorzec projektowy, którego celem jest umożliwienie tworzenia obiektów danej klasy bądź klas z wykorzystaniem już istniejącego obiektu, zwanego prototypem.

Przykład do analizy: [https://pl.wikibooks.org/wiki/Kody\\_%C5%BAr%C3%B3d%C5%82owe/Prototyp\\_\(wzorzec\\_projektowy\)#C#](https://pl.wikibooks.org/wiki/Kody_%C5%BAr%C3%B3d%C5%82owe/Prototyp_(wzorzec_projektowy)#C#)

## Zastosowania

- duża liczba obiektów tego samego typu
- chcemy otrzymać prawidłową kopię obiektu

# Singleton

Singleton - kreatywny wzorzec projektowy, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji oraz zapewnienie globalnego dostępu do stworzonego obiektu

```
public sealed class Singleton
{
    private static Singleton instance = null;

    private Singleton() { }

    public static Singleton Instance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```



## Wzorce strukturalne

- Adapter
- Dekorator
- Fasada
- Kompozyt
- Most
- Pełnomocnik
- Pyłek

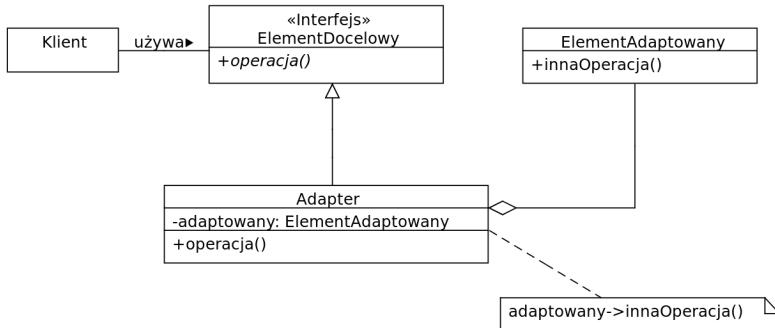
Wzorce strukturalne opisują struktury powiązanych obiektów oraz sposoby składania obiektów w większe struktury.

# Adapter

Adapter (także: opakowanie, ang. wrapper) – strukturalny wzorzec projektowy, którego celem jest umożliwienie współpracy dwóm klasom o niekompatybilnych interfejsach. Adapter przekształca interfejs jednej z klas na interfejs drugiej klasy.

Przykład:

- [https://github.com/pjastr/ZPO\\_programy/tree/master/AdapterWzorzecProjektowy](https://github.com/pjastr/ZPO_programy/tree/master/AdapterWzorzecProjektowy)



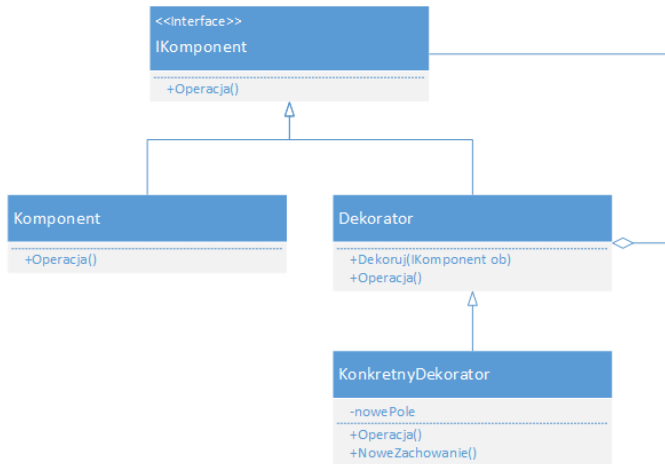
Rysunek 2:

# Dekorator

Dekorator – wzorzec projektowy należący do grupy wzorców strukturalnych. Pozwala na dodanie nowej funkcji do istniejących klas dynamicznie podczas działania programu. Dekoratory są alternatywą dla dziedziczenia. Dziedziczenie rozszerza zachowanie klasy w trakcie kompilacji, w przeciwieństwie do dekoratorów, które rozszerzają klasy w czasie działania programu.

Przykład:

[https://github.com/pjastr/ZPO\\_programy/tree/master/Dekorator](https://github.com/pjastr/ZPO_programy/tree/master/Dekorator)



Rysunek 3:

## Zastosowania:

- dynamiczna zmiana wyglądu okna w zależności od potrzeb,
- dodawanie funkcji do istniejących klas w czasie działania programu.

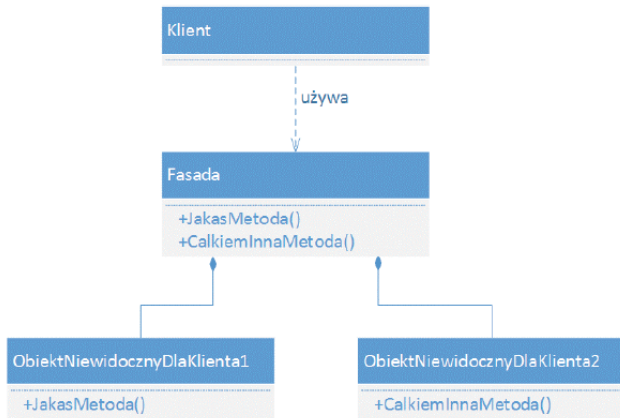
# Fasada

Fasada – wzorec projektowy należący do grupy wzorców strukturalnych. Służy do ujednoczenia dostępu do złożonego systemu poprzez wystawienie uproszczonego, uporządkowanego interfejsu programistycznego, który ułatwia jego użycie.

Przykład:

[https://github.com/pjastr/ZPO\\_programy/tree/master/Fasada](https://github.com/pjastr/ZPO_programy/tree/master/Fasada)





Rysunek 4:

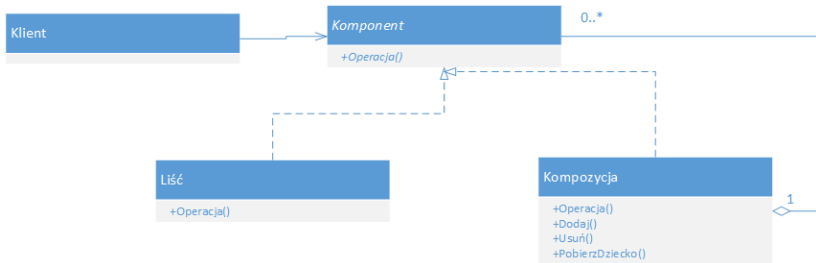
- ukrycie części systemu przed klientem (np. w banku, sklepie) i zmniejszenie liczby zależności pomiędzy klientem a systemem,
- API do połączenia z naszą aplikacją.

# Kompozyt

Kompozyt – strukturalny wzorzec projektowy, którego celem jest składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt.

Przykład: [https:](https://github.com/pjastr/ZPO_programy/tree/master/Kompozyt)

[//github.com/pjastr/ZPO\\_programy/tree/master/Kompozyt](https://github.com/pjastr/ZPO_programy/tree/master/Kompozyt)



Rysunek 5:

- Liść (Leaf) – reprezentuje prymitywny obiekt nie posiadający potomków,
- Kompozyt (Composite) – reprezentuje grupę obiektów, składającą się z „liści”, implementuje akcje interfejsu Komponent,
- Komponent (Component) – interfejs, który implementują obiekty, definiuje ich domyślne zachowanie,
- Klient – operuje na obiektach zawartych w układzie.

## Zastosowania

- kiedy potrzebujemy systemu z możliwością łatwego rozszerzania o nowe komponenty,
- np. sprzedaż produktów, zestawów produktów, usług.

# Most

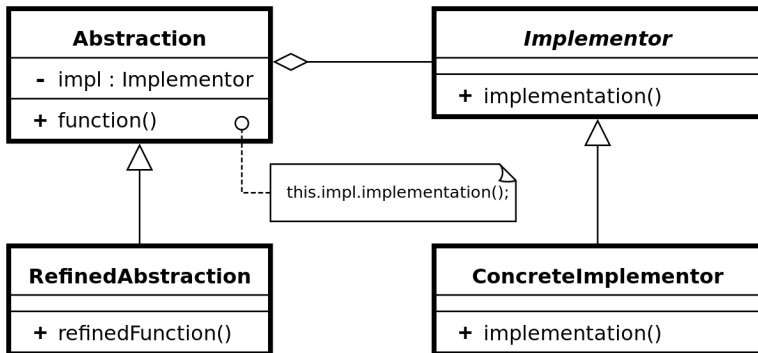
Wzorzec mostu (ang. Bridge pattern) – strukturalny wzorzec projektowy, który pozwala oddzielić abstrakcję obiektu od jego implementacji.

Zaleca się stosowanie tego wzorca aby:

- odseparować implementację od interfejsu,
- poprawić możliwości rozbudowy klas, zarówno implementacji, jak i interfejsu (m.in. przez dziedziczenie),
- ukryć implementację przed klientem, co umożliwi zmianę implementacji bez zmian interfejsu.

Przykład: [https:](https://github.com/pjastr/ZPO_programy/tree/master/MostWzorzec)

[//github.com/pjastr/ZPO\\_programy/tree/master/MostWzorzec](https://github.com/pjastr/ZPO_programy/tree/master/MostWzorzec)



Rysunek 6:

# Pełnomocnik

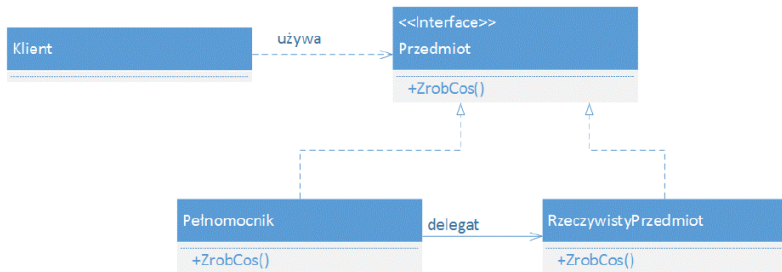
Pełnomocnik (ang. proxy) – strukturalny wzorzec projektowy, którego celem jest utworzenie obiektu zastępującego inny obiekt. Stosowany jest w celu kontrolowanego tworzenia na żądanie kosztownych obiektów oraz kontroli dostępu do nich.

Przykład: [https://github.com/pjastr/ZPO\\_programy/tree/master/PełnomocnikWzorzec](https://github.com/pjastr/ZPO_programy/tree/master/PełnomocnikWzorzec)



## Rodzaje:

- wirtualny – przechowuje obiekty, których utworzenie jest kosztowne; tworzy je na żądanie
- ochraniający – kontroluje dostęp do obiektu sprawdzając, czy obiekt wywołujący ma odpowiednie prawa do obiektu wywoływanego
- zdalny – czasami nazywany ambasadorem; reprezentuje obiekty znajdujące się w innej przestrzeni adresowej
- sprytne odwołanie – czasami nazywany sprytnym wskaźnikiem; pozwala na wykonanie dodatkowych akcji podczas dostępu do obiektu, takich jak: zliczanie referencji do obiektu czy ładowanie obiektu do pamięci

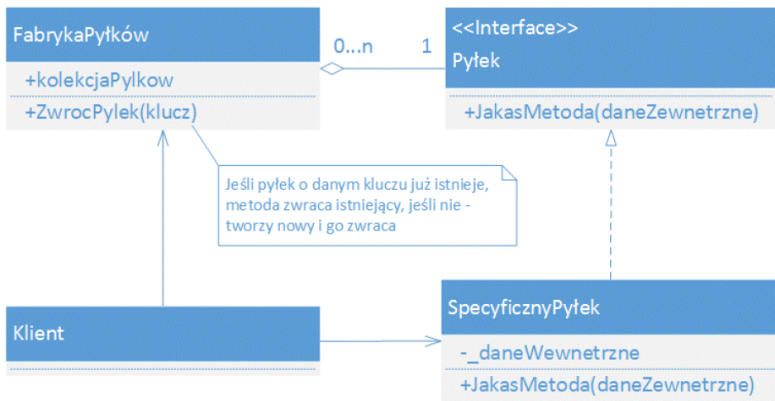


Rysunek 7:

## Pyłek

Pyłek (ang. Flyweight) – strukturalny wzorec projektowy, którego celem jest zmniejszenie wykorzystania pamięci poprzez poprawę efektywności obsługi dużych obiektów zbudowanych z wielu mniejszych elementów poprzez współdzielenie wspólnych małych elementów.

Przykład: [https://github.com/pjastr/ZPO\\_programy/tree/master/Py%C5%82ekWzorzec](https://github.com/pjastr/ZPO_programy/tree/master/Py%C5%82ekWzorzec)



Rysunek 8:

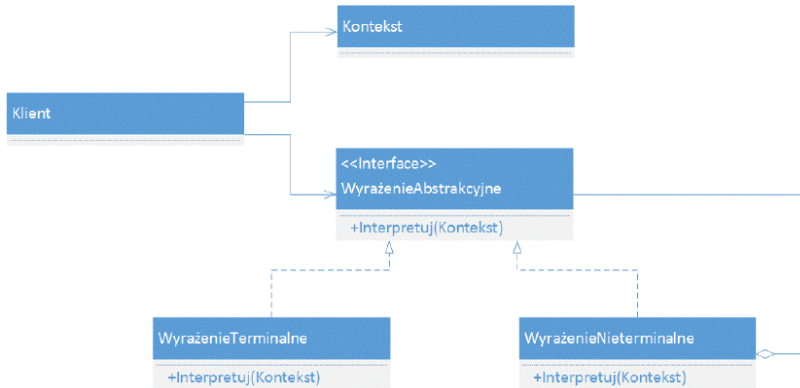
## Wzorce operacyjne

Wzorce operacyjne (czynnościowe) opisują zachowanie obiektów, komunikację pomiędzy nimi i ich odpowiedzialność.

- Interpreter
- Iterator (kursor)
- Łańcuch zobowiązań
- Mediator
- Metoda szablonowa
- Obserwator
- Odwiedzający (wizytator)
- Pamiątka (znacznik)
- Polecenie
- Stan
- Strategia (polityka)

# Interpreter

Interpreter – czynnościowy wzorzec projektowy, którego celem jest zdefiniowanie opisu gramatyki pewnego języka interpretowalnego, a także stworzenie dla niej interpretera, dzięki któremu będzie możliwe rozwiązanie opisanego problemu.



Rysunek 9:

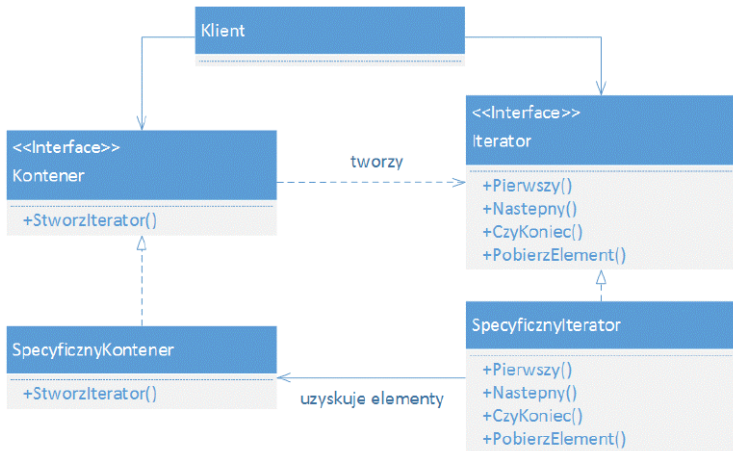


## Zastosowania:

- interpretacja języka
- kompilatory
- odwrotna notacja polska, . . .

## Iterator (kursor)

Iterator – czynnościowy wzorzec projektowy (obiektowy), którego celem jest zapewnienie sekwencyjnego dostępu do podobiektów zgrupowanych w większym obiekcie.



Rysunek 10:

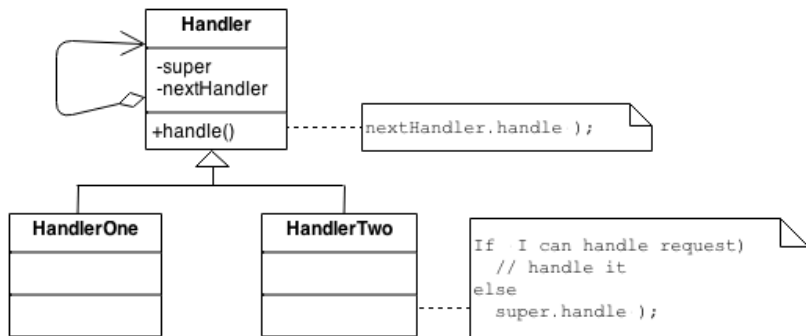
## Zastosowania:

- przetwarzanie zróżnicowanych kolekcji,
- aplikacje, w których dane są przechowywane w kolekcjach różnych typów.

## Łańcuch zobowiązań

Łańcuch zobowiązań - czynnościowy wzorec projektowy, w którym żądanie może być przetwarzane przez różne obiekty, w zależności od jego typu.

```
public void operacja(żądanie: Żądanie)
{
    jeśli potrafimy obsłużyć dany typ żądania żądanie:
        obsłuż żądanie
    w przeciwnym wypadku:
        przekaz żądanie następnikowi
}
```



Rysunek 11:

## Zastosowanie:

- Wzorzec znajduje zastosowanie wszędzie tam, gdzie mamy do czynienia z różnymi mechanizmami podobnych żądań, które można zaklasyfikować do różnych kategorii. Dodatkową motywacją do jego użycia są często zmieniające się wymagania.

## Zalety:

- elementy łańcucha mogą być dynamicznie dodawane i usuwane w trakcie działania programu,
- zmniejszenie liczby zależności między nadawcą a odbiorcami,
- implementacja pojedynczej procedury nie musi znać struktury łańcucha oraz innych procedur.

## Wady:

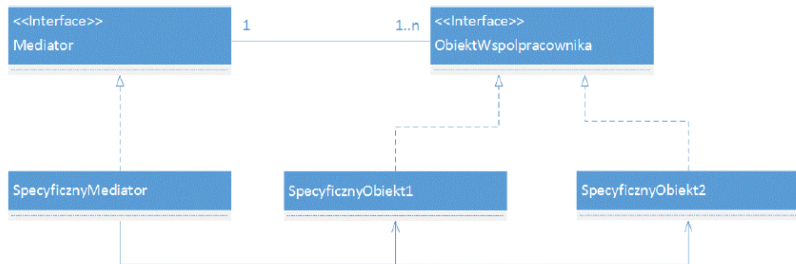
- wzorzec nie gwarantuje, że każde żądanie zostanie obsłużone,
- śledzenie i debugowanie pracy działania łańcucha może być

# Mediator

Mediator – wzorec projektowy należący do grupy wzorców czynnościowych. Mediator zapewnia jednolity interfejs do różnych elementów danego podsystemu.

Wzorec mediatora umożliwia zmniejszenie liczby powiązań między różnymi klasami, poprzez utworzenie mediatora będącego jedyną klasą, która dokładnie zna metody wszystkich innych klas, którymi zarządza. Nie muszą one nic o sobie wiedzieć, jedynie przekazują polecenia mediatorowi, a ten rozsyła je do odpowiednich obiektów.

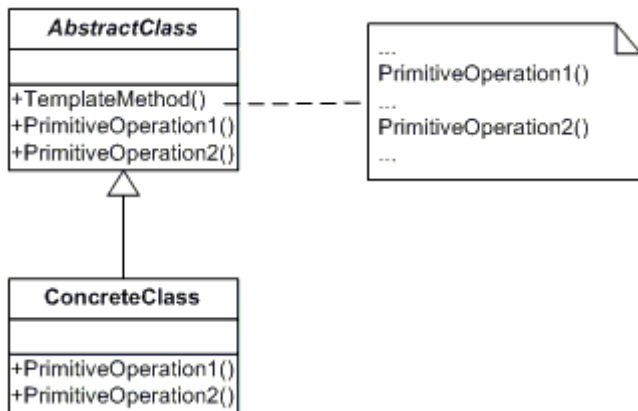




Rysunek 12:

## Metoda szablonowa

Metoda szablonowa – czynnościowy wzorzec projektowy. Jego zadaniem jest zdefiniowanie metody, będącej szkieletem algorytmu. Algorytm ten może być następnie dokładnie definiowany w klasach pochodnych. Niezmienna część algorytmu zostaje opisana w metodzie szablonowej, której klient nie może nadpisać. W metodzie szablonowej wywoływane są inne metody, reprezentujące zmienne kroki algorytmu. Mogą one być abstrakcyjne lub definiować domyślne zachowania. Klient, który chce skorzystać z algorytmu, może wykorzystać domyślną implementację bądź może utworzyć klasę pochodną i nadpisać metody opisujące zmienne fragmenty algorytmu.

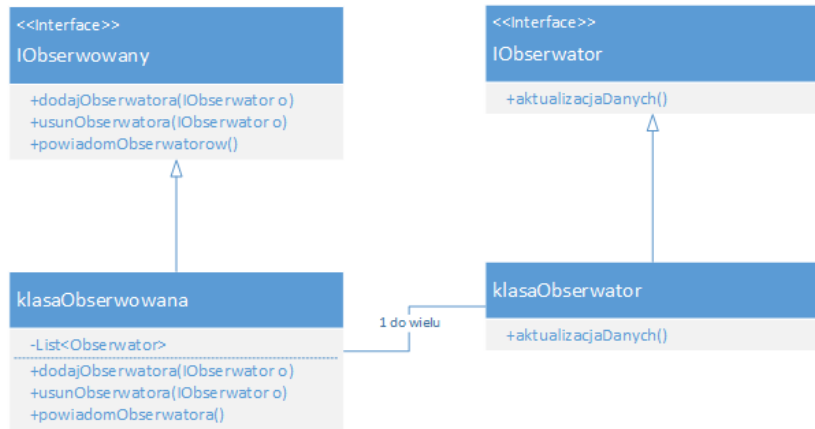


Rysunek 13:

# Obserwator

Obserwator – wzorzec projektowy należący do grupy wzorców czynnościowych. Używany jest do powiadamiania zainteresowanych obiektów o zmianie stanu pewnego innego obiektu.

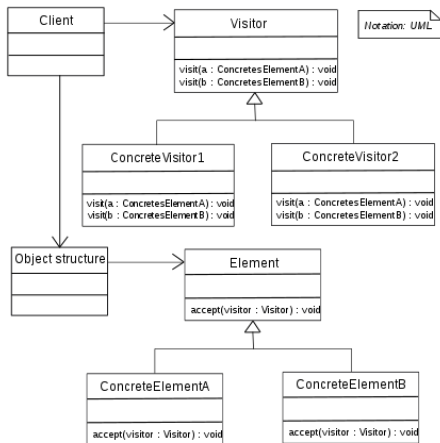
```
public void powiadomObserwatorow() {  
    dla każdego obserwatora obserwator z listy obserwatorzy  
        wywołaj obserwator.aktualizacja(obserwator);  
}
```



Rysunek 14:

## Odwiedzający (wizytator)

Odwiedzający (wizytator) – wzorzec projektowy, którego zadaniem jest odseparowanie algorytmu od struktury obiektowej na której operuje. Praktycznym rezultatem tego odseparowania jest możliwość dodawania nowych operacji do aktualnych struktur obiektów bez konieczności ich modyfikacji.

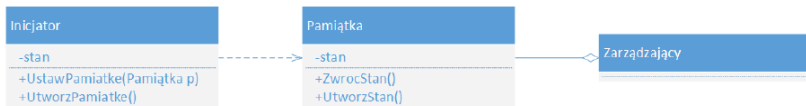


Rysunek 15:

## Pamiętka (znacznik)

Pamiętka – czynnościowy wzorzec projektowy. Jego zadaniem jest zapamiętanie i udostępnienie na zewnątrz wewnętrznego stanu obiektu bez naruszania hermetyzacji. Umożliwia to przywracanie zapamiętanego stanu obiektu.

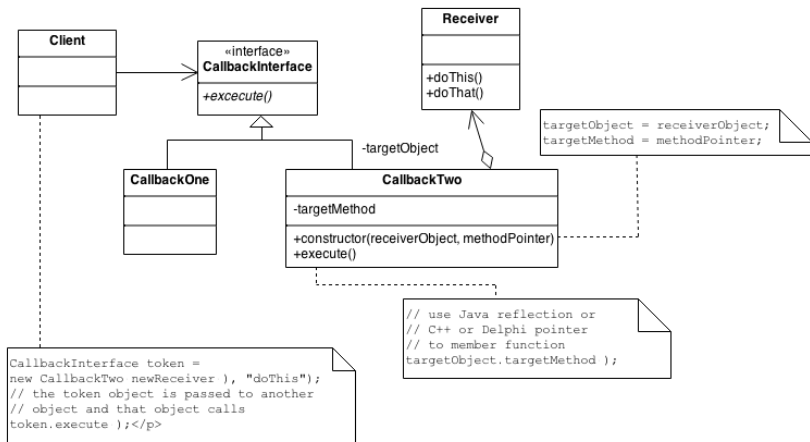




Rysunek 16:

## Polecenie

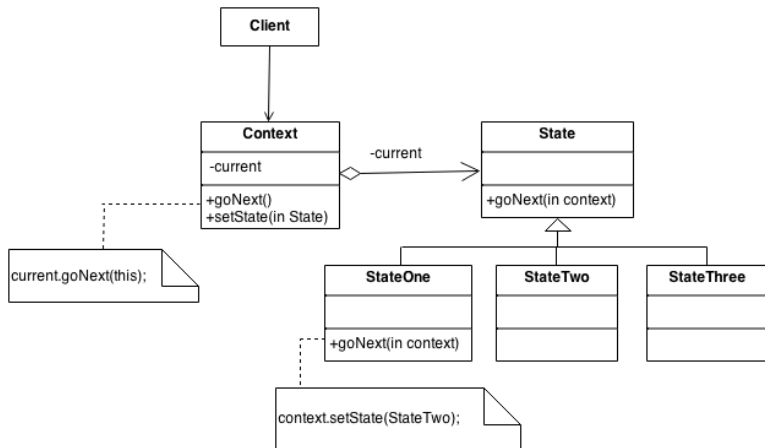
Polecenie – czynnościowy wzorzec projektowy, traktujący żądanie wykonania określonej czynności jako obiekt, dzięki czemu mogą być one parametryzowane w zależności od rodzaju odbiorcy, a także umieszczane w kolejkach i dziennikach.



Rysunek 17:

# Stan

Stan – czynnościowy wzorzec projektowy, który umożliwia zmianę zachowania obiektu poprzez zmianę jego stanu wewnętrznego.  
Innymi słowy – uzależnia sposób działania obiektu od stanu w jakim się aktualnie znajduje.

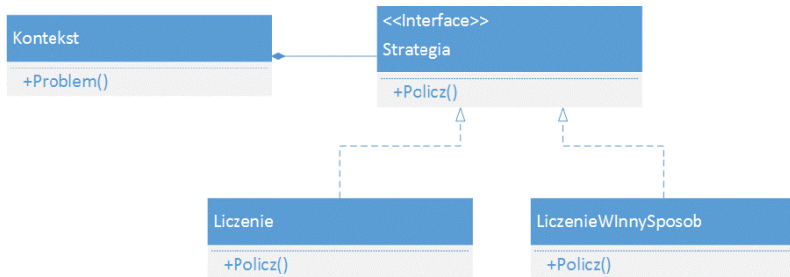


Rysunek 18:

## Strategia (polityka)

Strategia – czynnościowy wzorzec projektowy, który definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Umożliwia wymienne stosowanie każdego z nich w trakcie działania aplikacji niezależnie od korzystających z nich użytkowników.

[http://www.altcontroldelete.pl/artykuly/  
worzec-strategia-przykladowa-implementacja-w-c/](http://www.altcontroldelete.pl/artykuly/worzec-strategia-przykladowa-implementacja-w-c/)



Rysunek 19:

Co dalej?



## co dalej?

<https://bulldogjob.pl/news/550-zegnaj-programowanie-objektowe>

## Bibliografia

- Wikipedia, Wikibooks na licencji CC.
- Joseph Albahari, Ben Albahari, C# 7.0 w pigułce, wyd. Helion, 2018.
- Krzysztof, Sopyła, Wykłady z ZPO, <http://wmii.uwm.edu.pl/~ksopyla/>, dostęp online 19.04.2019.
- <http://itcraftsman.pl/>  
uzyteczne-koncepty-projektowe-kiss-dry-yagni-tda-oraz-separation-of-  
dostęp online 5.05.2019.
- [https://www.p-programowanie.pl/uml/diagramy-klas-uml/  
#Zwiazki-pomiedzy-klasami-w-UML](https://www.p-programowanie.pl/uml/diagramy-klas-uml/#Zwiazki-pomiedzy-klasami-w-UML), dostęp online 01.03.2019.
- Daniel Krasnokucki, Wzorce projektowe. Leksykon kieszonkowy, Wyd. Helion 2017.

- <http://tomaszjarzynski.pl/metoda-wytworcza-wzorzec-projektowy-factory-method/>, dostęp online 1.03.2019.
- <http://tomaszjarzynski.pl/fabryka-abstrakcyjna-wzorzec-projektowy-abstract-factory/>
- <http://www.altcontroldelete.pl/artykuly/wzorzec-adapter-przykladowa-implementacja-w-c/>. dostęp online 10.03.2019.
- <http://lukaszkosiorowski.pl/programowanie/dekorator-decorator/>, dostęp online 10.03.2019.
- <http://lukaszkosiorowski.pl/programowanie/kompozyt-composite/>, dostęp online 10.03.2019.
- <http://devman.pl>, dostęp online 15.03.2019.
- <https://sourcemaking.com/>, dostęp online 15.03.2019.

- <https://www.oodesign.com/visitor-pattern.html>, dostęp online 15.03.2019.
- <https://medium.com/@sawomirkowalski/design-patterns-state-6e4ad27df0d7>, dostęp online 15.03.2019.