

1. Wprowadzenie do wzorców projektowych
2. Jakość kodu źródłowego
3. Refaktoryzacja do wzorców
4. Wzorce GoF
5. Język wzorców w architekturze aplikacji – wybrane wzorce architektoniczne

- # Znajomość podstawowych zagadnień programowania zorientowanego obiektowo: abstrakcja, polimorfizm, dziedziczenie, hermetyzacja
- # Znajomość języka C#
- # Znajomość UML
- # Znajomość środowiska MS Visual Studio C#

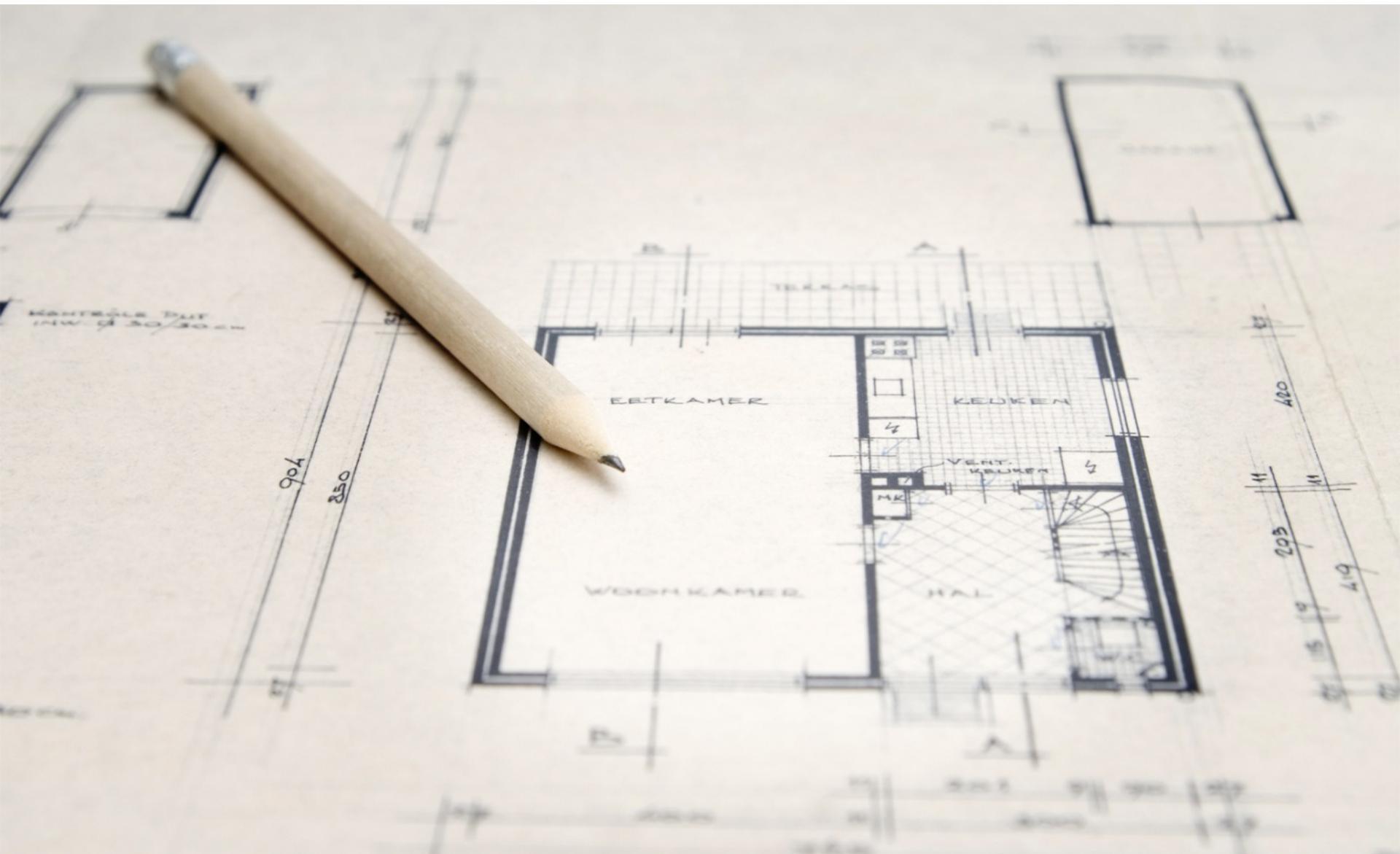


**Imię i nazwisko
Stanowisko
Doświadczenie
Oczekiwania**

Poznajmy się!

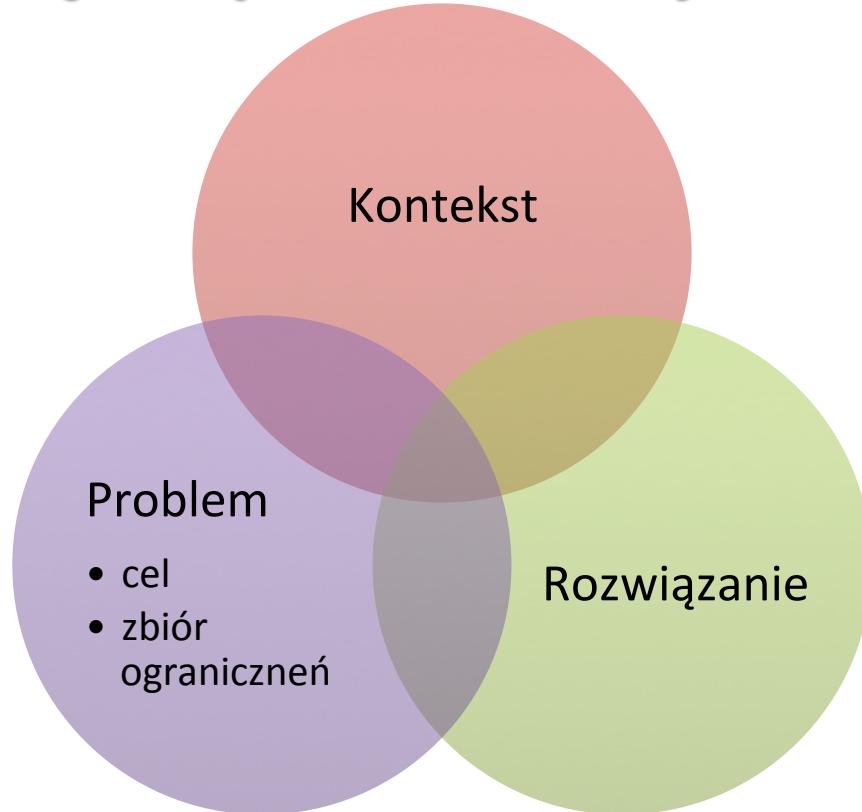
Wprowadzenie do wzorców projektowych

Wzorce projektowe i refaktoryzacja do wzorców



1. Pojęcie wzorca projektowego
2. Historia powstania wzorców
3. Cechy wzorca projektowego
4. Przykłady wzorców projektowych
5. Kategorie wzorców projektowych

to rozwiązanie problemu w danym kontekście



„Każdy wzorzec opisuje problem, który ciągle pojawia się w naszej dziedzinie, a następnie określa zasadniczą część jego rozwiązania w taki sposób, by można było zastosować je nawet milion razy, za każdym razem w nieco inny sposób”

Alexander Christopher, *A Patterns Language*, 1977

Gamma, Helm, Johnson, Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995

- Katalog 23 wzorców projektowych
- Pokazanie zastosowania wzorców projektowych w dziedzinie projektowania oprogramowania

1. Wzorce projektowe GoF

Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995

2. Wzorce architektoniczne

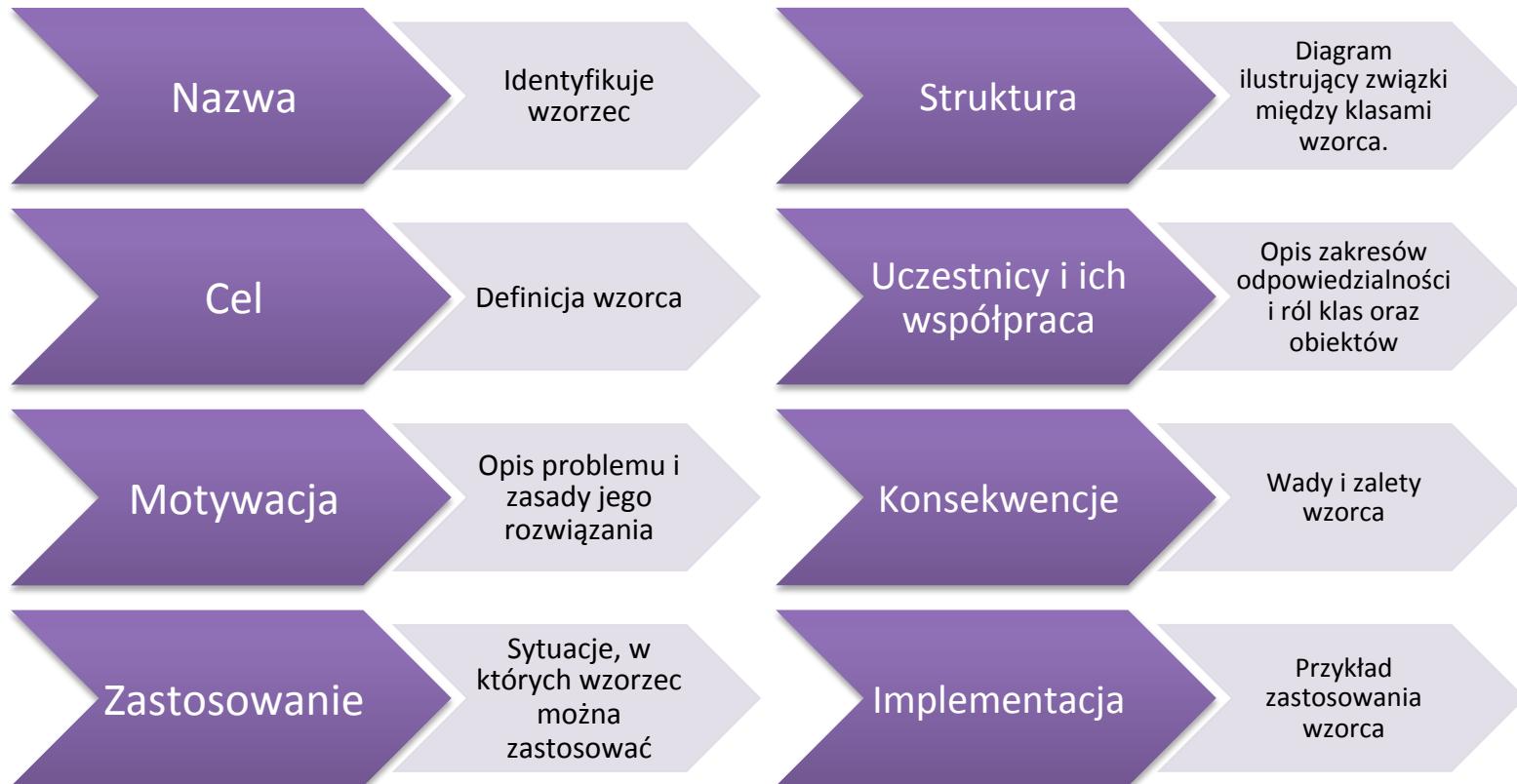
Pattern-Oriented Software Architecture (seria), 1996-2007
Fowler, *Patterns of Enterprise Application Architecture*, 2002

3. Wzorce integracyjne

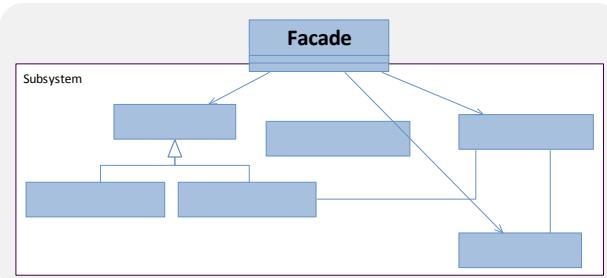
Hohpe, Woolf, <http://www.enterpriseintegrationpatterns.com>

- # Wzorzec projektowy identyfikuje najważniejsze aspekty struktury typowego rozwiązania.
- # Określa uczestniczące w nim klasy i obiekty, ich rolę, współpracę oraz podział odpowiedzialności.
- # Dotyczy konkretnego zagadnienia projektowania obiektowego.

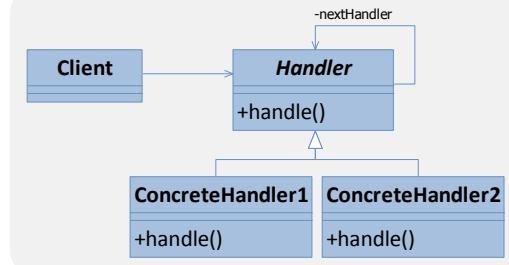
Kluczowe elementy opisu wzorca projektowego



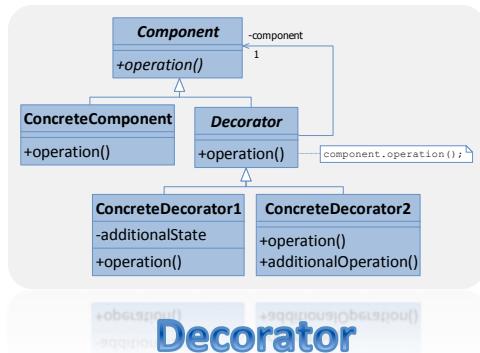
Przykłady wzorców projektowych



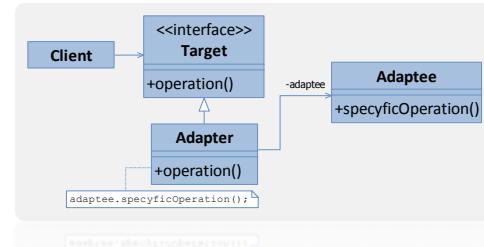
Facade



Chain of Responsibility



Decorator



Adapter

Dlaczego warto znać wzorce?

Ponieważ wzorce projektowe:

- # Powstały na bazie wiedzy i umiejętności ekspertów.
- # Zostały wyodrębnione w skutek analizy sprawdzonych rozwiązań.
- # Sprawdziły się wcześniej wielokrotnie.
- # Tworzą język porozumienia na poziomie projektowym.
- # Umożliwiają i ułatwiają myślenie na wyższym poziomie abstrakcji.
- # Pozwalają dogłębnie zrozumieć zasady programowania zorientowanego obiektowo.
- # Umożliwiają tworzenie elastycznego oprogramowania

Kategorie wzorców projektowych GoF

Kreacyjne

- Simple Factory
- Factory Method
- Builder

Strukturalne

- Adapter
- Decorator
- Facade
- Proxy

Behavioralne

- Command
- Strategy
- Observer
- Chain of Responsibility
- Template Method

Inny podział wzorców projektowych GoF

Wzorce klas

- Template Method
- Factory Method
- Adapter

Wzorce obiektów

- Decorator
- Proxy
- Facade
- Command
- Observer
- Strategy
- Chain of Responsibility
- Builder

Wzorzec projektowy to rozwiązanie problemu w danym kontekście

Na wzorzec projektowy składają się:
unikatowa nazwa, cel, motywacja,
struktura, konsekwencje, ...

Wzorce projektowe GoF dzielą się na:
kreacyjne, strukturalne i behawioralne

```
private $host;
private $username;
private $password;
private $database;
private $charset;

static private $link = null;

static public function connect()
{
    self::$link = mysql_connect(self::$host, self::
```

bns   Wyznaczniki Jakości Kodu
Jakość kodu źródłowego

Praktyki poprawiające jakość kodu

Definiowanie i przestrzeganie odpowiedzialności

Tworzenie czytelnego kodu

Enkapsulowanie

Preferowanie kompozycji ponad dziedziczenie

Programowanie poprzez interfejsy

- # Praktyki są podstawowymi wytycznymi programowaniu obiektowym, z których wynika większość rozwiązań programistycznych i architektonicznych
- # Wzorce projektowe są skutkiem przestrzegania powyższych zasad

Objawy kodu o wysokiej jakości

Dokładanie ponad modyfikacje

- Dodawanie lub zmiana funkcjonalności wiąże się raczej z dodawaniem nowych bytów w systemie niż z modyfikowaniem istniejących

Lokalne zmiany

- Zmiany wprowadzane w kodzie mają zasięg lokalny (blok, metoda, klasa)

Nieinwazyjność zmian

- Zmiany dokonywane w kodzie nie modyfikują drastycznie istniejącej struktury
- Zmiany są przezroczyste zwłaszcza dla klientów zmienianego fragmentu systemu

- # Określaj odpowiedzialność dla zmiennych, metod, klas, interfejsów, pakietów, modułów
- # Dbaj, aby odpowiedzialność była wyłącznie jedna

Konsekwencje stosowania	Konsekwencje zaniedbywania
<ul style="list-style-type: none">• Wiele dobrze zdefiniowanych komponentów współpracujących ze sobą• Nadmiar może spowodować zbytnie rozdrobnienie kodu lub Solution Sprawl	<ul style="list-style-type: none">• Duże (pod względem linii kodu) komponenty• Prawie niemożliwe efektywne testowanie jednostkowe• Niska czytelność

Tworzenie czytelnego kodu

- # Twórz kod, który *czyta się jak książkę*
- # Poruszaj się od ogółu do szczegółu

Konsekwencje stosowania	Konsekwencje zaniedbywania
<ul style="list-style-type: none">• Samodokumentujący się kod• Ograniczenie ilości diagramów i dokumentacji• Wyeliminowanie komentarzy w kodzie• Szybkie rozumienie intencji programisty	<ul style="list-style-type: none">• Kod jest zrozumiały wyłącznie dla jego autora• Duże prawdopodobieństwo pomyłek• Trudne poszukiwanie błędów

- # Zamykaj funkcjonalność w komponentach od dobrze zdefiniowanym interfejsie
- # To co zmienia się w kodzie enkapsuluj w metodę lub klasę

Konsekwencje stosowania	Konsekwencje zaniedbywania
<ul style="list-style-type: none">• Możliwe wielokrotne użycie komponentów• Ułatwione wprowadzanie zmian w działaniu komponentów	<ul style="list-style-type: none">• Silne zależności pomiędzy fragmentami kodu• Powstawanie dużych komponentów

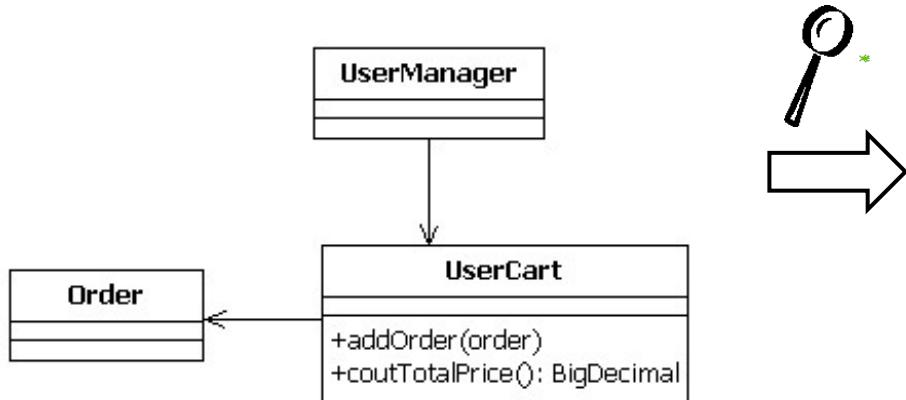
Do uzyskania nowej funkcjonalności używaj delegacji zamiast przeciążania

Konsekwencje stosowania	Konsekwencje zaniedbywania
<ul style="list-style-type: none">• Funkcjonowanie klasy jest od razu zrozumiałe dla programisty• Można dynamicznie zmieniać funkcjonalność komponentu, podmieniając zależności	<ul style="list-style-type: none">• Rozbudowane hierarchie dziedziczenia utrudniają zrozumienie kodu• Brak wyeksponowanych zależności uniemożliwia testy jednostkowe

Definiuj w miarę niezmiennej sposób komunikowania się komponentu z otoczeniem

Konsekwencje stosowania	Konsekwencje zaniedbywania
<ul style="list-style-type: none">• Zmiany są przezroczyste dla klientów• Zmiany mają zazwyczaj zasięg lokalny• Zmiana interfejsu powoduje kaskadowe konsekwencje we wszystkich implementacjach	<ul style="list-style-type: none">• Powstawanie silnych zależności w kodzie• Kaskadowe zmiany w przypadku modyfikowania sposobu działania funkcjonalności

Wzorce projektowe, wzorce implementacyjne



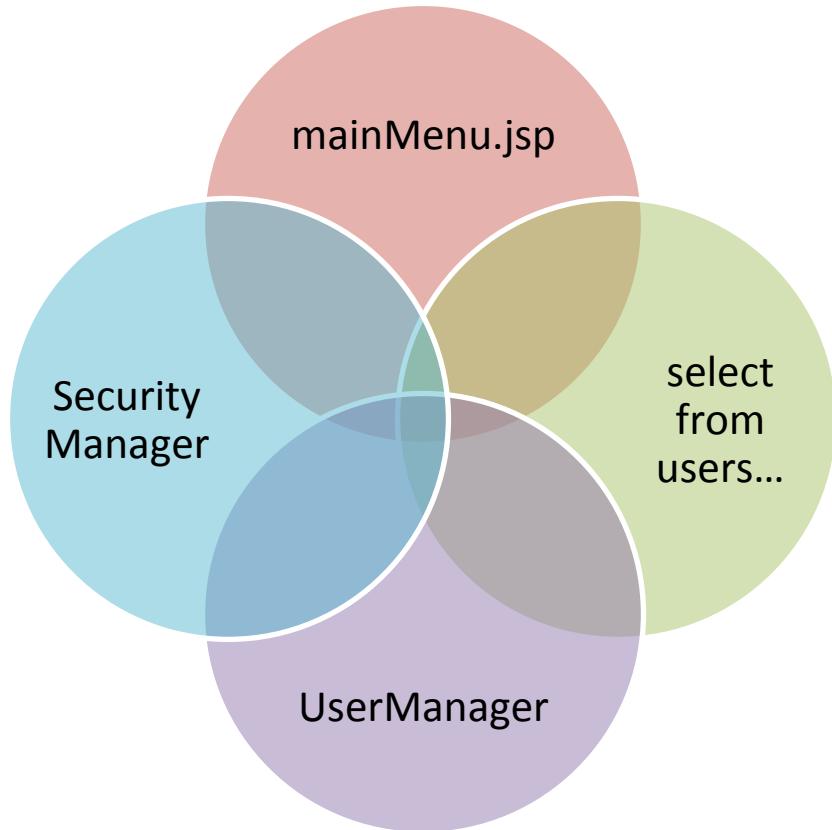
```
public class UserCart{
    public void addOrder( Order order ) {
        //...
    }

    public BigDecimal coutTotalPrice() {
        //...
    }
}
```

- Wzorce projektowe rozwiązuje problemy programistów dostarczając **struktury**, która pomoże poradzić sobie z tym problemem
- **Wzorce implementacyjne** schodzą poziom niżej i formułują zasady **implementowanie** kodu o wysokiej jakości; zajmują się nazewnictwem, czytelnością i elementarnymi konstrukcjami programistycznymi

bns it } Silne zależności
Jakość kodu źródłowego

Zależności w kodzie



- # Jeśli wymiana lub modyfikacja fragmentu systemu powoduje kaskadowe zmiany w innych częściach systemu, to sytuację nazywamy **silną zależnością** (*coupling*) pomiędzy częściami systemu
- # Silne zależności utrudniają utrzymanie oprogramowania

Silna zależność: warstwy

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<html>
<body>
<%
SqlConnection con = //...
SqlDataAdapter adapter = //...
DataSet ds = new DataSet();
adapter.Fill(ds, "Employees");
%>
</body>
</html>
```

- # Zmiana interfejsu użytkownika powoduje konieczność zmiany w niemal całej aplikacji
- # Niemal na pewno pojawią się duplikacje kodu na wielu stronach ASP
- # Strony ASP będą rozrastać się w niekontrolowany sposób

Silna zależność: warstwy

```
public class Department
{
    public void AddEmployee(String name, long depId)
    {
        SqlCommand command = //...
        command.CommandText = "update deps set empnum=" + 7 + " where id=" + depId + ";";
        //...
    }

    public DataSet FindAllEmployees()
    {
        SqlDataAdapter adapter = new SqlDataAdapter("select * from employees;");
        DataSet ds = new DataSet();
        //...
        return ds;
    }
}
```

- # System uzależniony jest od konkretnej bazy danych
- # Duże prawdopodobieństwo pomyłek przy pisaniu zapytań SQL
- # Trudne tworzenie i utrzymywanie testów
- # Słaba czytelność kodu

Silna zależność: elementy statyczne

```
class OrderProcessor
{
    public void Process()
    {
        PricingService pricingService = PricingService.GetInstance();
        //...
    }
}
```

- # Tworzenie zależności, których zmiana ma charakter globalny
- # Trudne testowanie
- # Brak możliwości korzystania mechanizmów obiektowych
- # Utrudnione zrozumienie kodu

Silna zależność: ukryte zależności

```
class OrderProcessor
{
    public void Process()
    {
        PricingService pricingService = new PricingService();
        //...
    }
}
```

- # Utrudnione testowanie
- # Zmiana implementacji *PricingService* pociąga za sobą konieczność modyfikacji i rekompilacji kodu
- # W przypadku konstruktorów programista jest „skazany” na logikę zaszytą w komponencie

Silna zależność: środowisko

```
public class Employee
{
    public void ChangeAddress( HttpRequest req )
    {
        String street = req.Params[ "ADDRESS_STREET" ];
        //...
    }
}
```

- # Uruchomienie aplikacji w innym środowisku niż webowe niesie za sobą wiele zmian w całym systemie
- # Utrudnione testowanie ze względu na konieczność symulowania środowiska webowego

Osłabianie zależności



- # Osłabianie zależności w kodzie to podstawowa wytyczna architektoniczna
- # Wzorce projektowe koncentrują się na tworzeniu kodu ze słabymi zależnościami

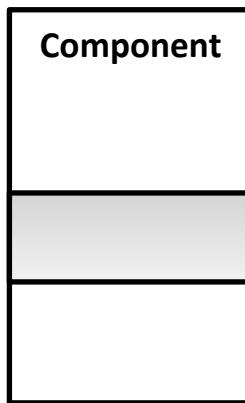
- # Programowanie poprzez interfejsy
- # Wstrzykiwanie zależności
- # Programowanie aspektowe
- # Komunikacja asynchroniczna



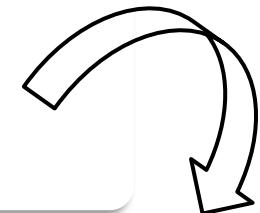
Po co nam interfejsy?

Jakość kodu źródłowego

Definiowanie interfejsu



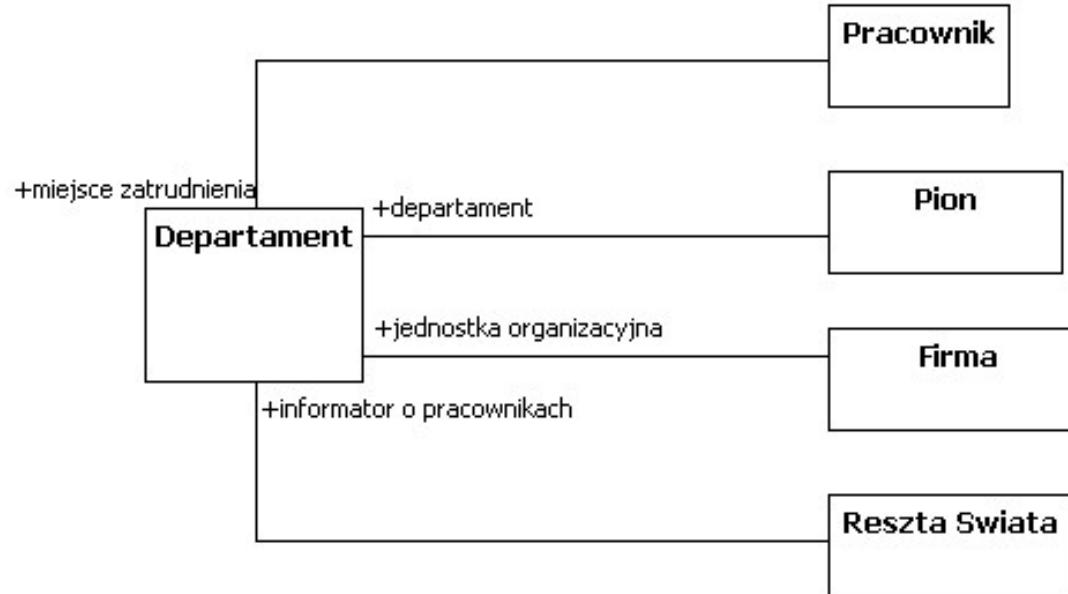
```
class Departement {  
    public void AddEmployee(Employee empl)  
    {  
        //...  
    }  
    public Employee GetDirector()  
    {  
        //...  
    }  
    public Employee NotifyEmployees( Message msg)  
    {  
        //...  
    }  
}
```



```
public interface IOrganizationalUnit {  
    void AddEmployee( Employee empl );  
  
    Employee GetDirector();  
  
    Employee NotifyEmployees( Message msg );  
}  
  
class Departement : IOrganizationalUnit { //...
```

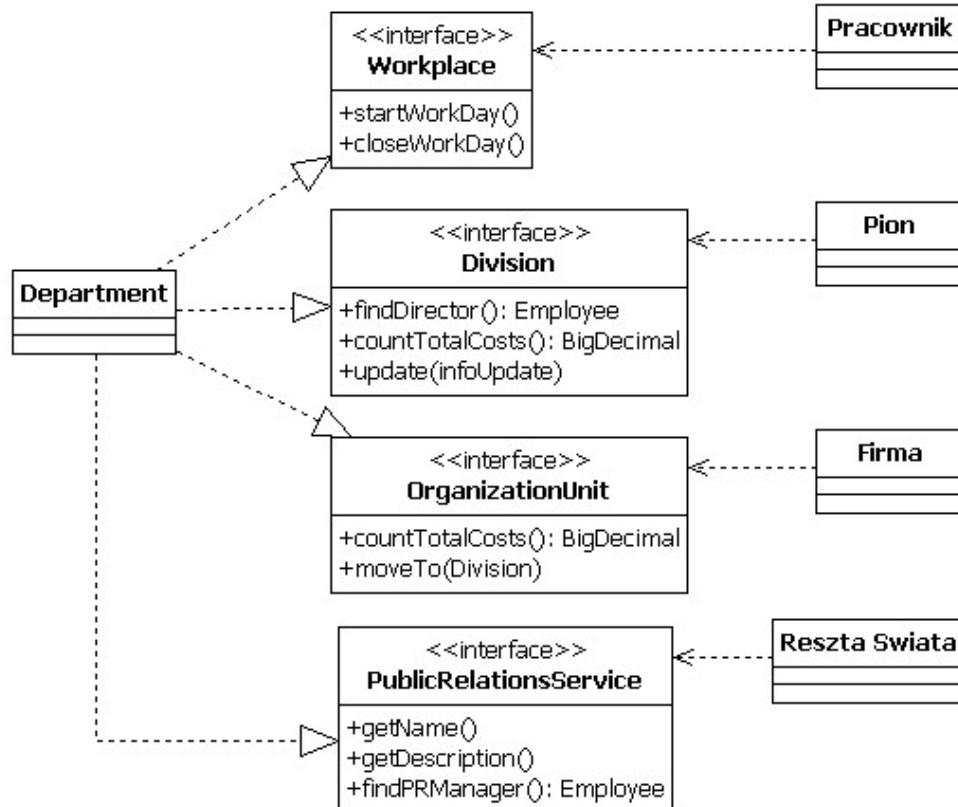
- # Definiuje sposób, w jaki komponent komunikuje się ze światem zewnętrznym
- # Interfejs to zestaw publicznych składników komponentu
- # Niepoprawny interfejs utrudnia rozwijanie systemu
- # Często zmieniający się interfejs utrudnia prace nad systemem

Dzielenie interfejsów



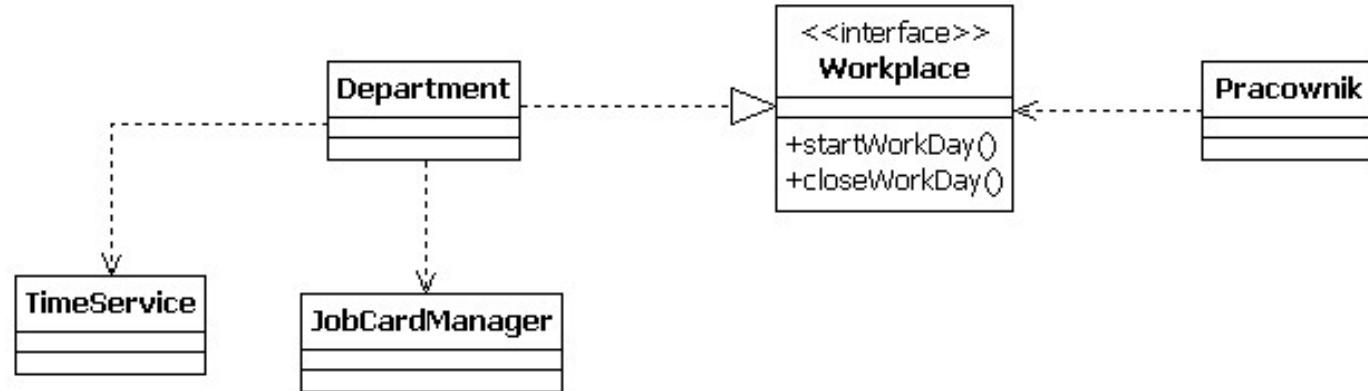
- # W relacjach z różnymi współpracownikami komponent może pełnić różne role
- # Definiuj nowy interfejs dla nowej roli, zamiast modyfikować istniejące interfejsy

Dzielenie interfejsów



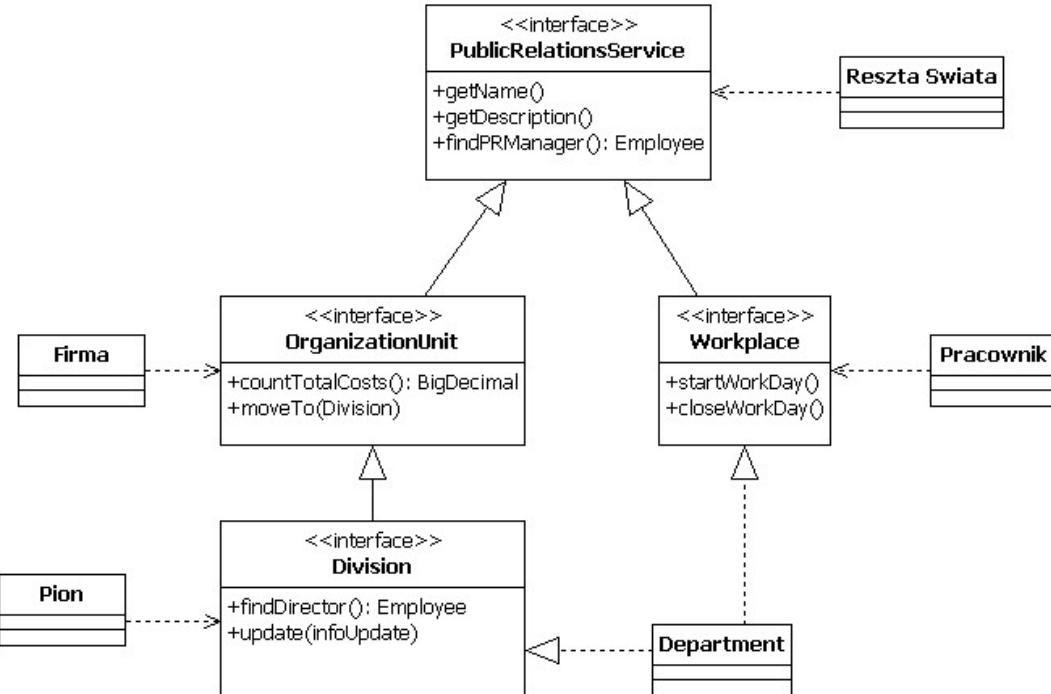
- # Komponent definiuje interfejs dla ról, którą pełni
- # Współpracownicy otrzymują tylko taką funkcjonalność, która jest im niezbędna i nic ponad to

Dzielenie interfejsów



- # Komponent może (często powinien) delegować część odpowiedzialności do innych komponentów współpracujących
- # Uwaga na **antywzorzec Helper** – dla klasy pomocniczej również należy definiować odpowiedzialność

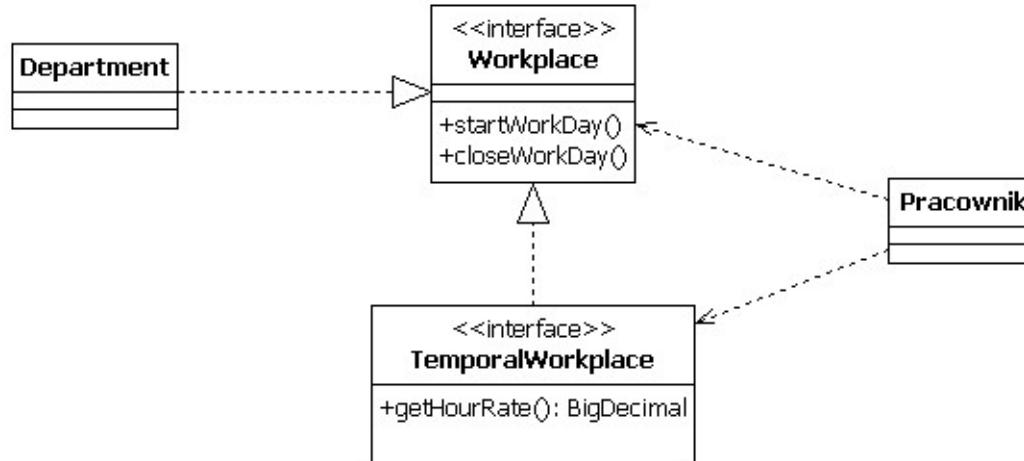
Rozszerzanie interfejsów



Rozszerzanie interfejsów pozwala na ustalenie zakresów dostępu do funkcjonalności obiektu

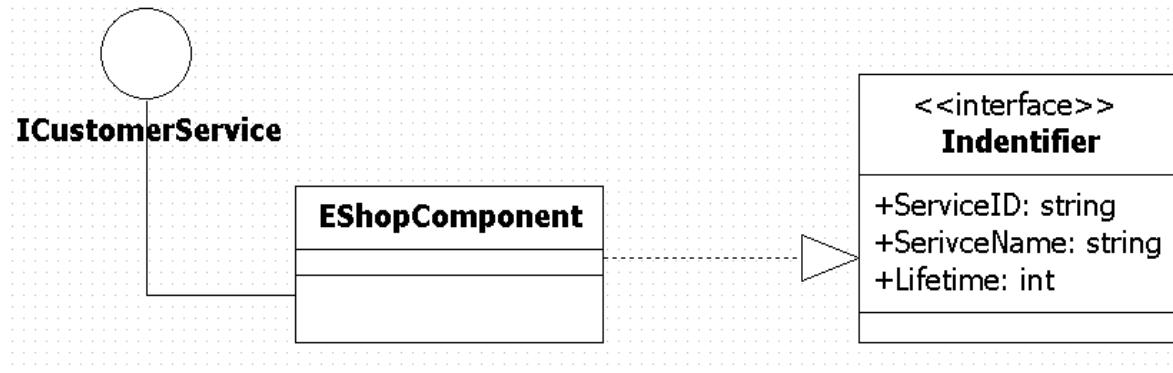
Zapewnia logiczną ciągłość funkcjonalności

Wersjonowanie interfejsów



- # Gdy oczekiwania klientów rosną, interfejs może być **wersjonowany**
- # Rozwiązanie zapewnia ciągłość dostępu do funkcjonalności komponentu
- # Chociaż techniczna implementacja wygląda tak samo jak przy rozszerzaniu interfejsów, **są to rozwiązania o różnych intencjach**

Interfejs retrospekcyjny

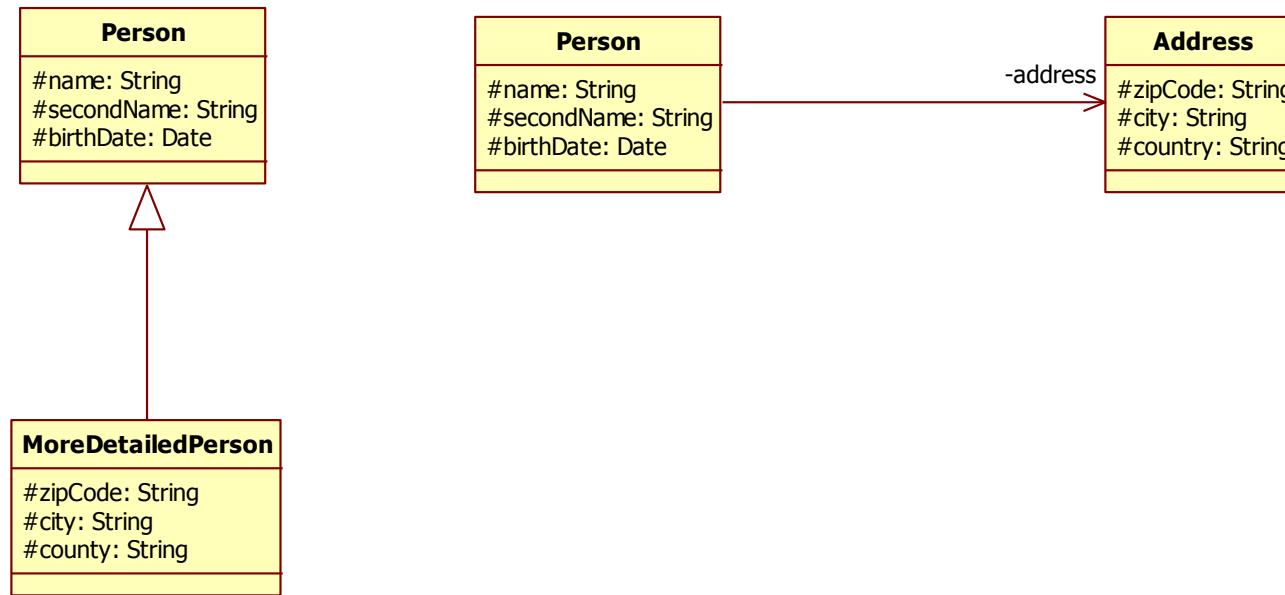


- # **Introspective Interface** jest szczególnym przypadkiem *Extension Interface*
- # Jego odpowiedzialnością jest dostarczenie informacji diagnostycznych o komponencie
- # Tego typu funkcjonalność powinna zostać oddzielona od głównych usług komponentu
- # Szczególnym przypadkiem **Introspective Interface** są klasy *Class, Method, Filed, itd.* z biblioteki standardowej

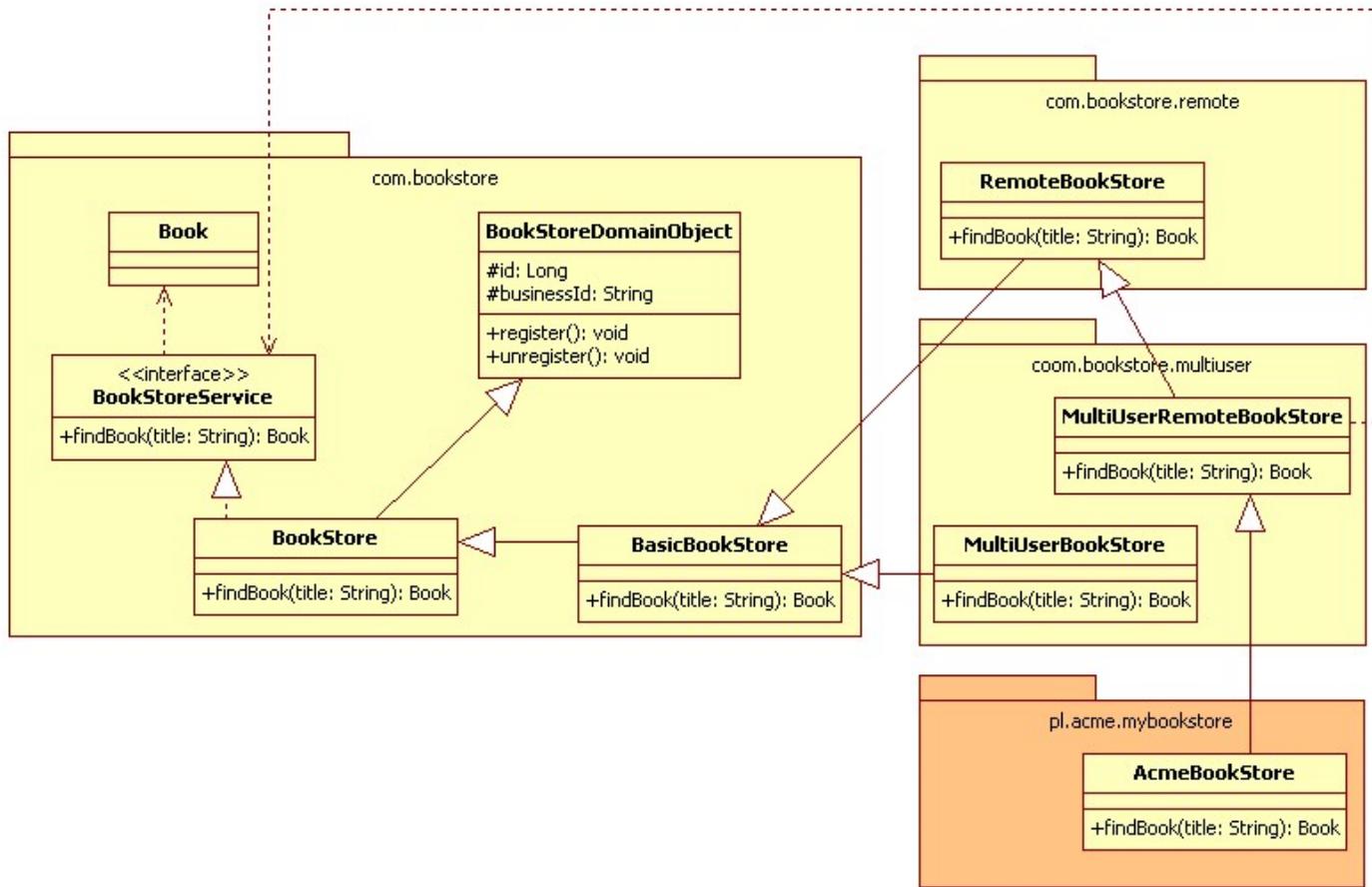
Oznacza, że:	NIE oznacza, że:
<ul style="list-style-type: none">• Interfejs jest najważniejszą częścią komponentu• Interfejs to kontrakt, który zobowiązuje się wypełnić komponent• Dokładnie przemyśl w jaki sposób komponent komunikuje się ze światem zewnętrznym• O ile to możliwe twórz interfejsy, które nie będą zmieniać się w czasie• Powinieneś używać typu interfejsu<ul style="list-style-type: none">• jako typów parametrów metod• jako typów zwracanych przez metody• po lewej stronie deklaracji pól i zmiennych	<ul style="list-style-type: none">• Twórz interfejsy (słowo kluczowe interface) do każdej klasy w systemie• Interfejs zawsze musi być osobnym bytem w systemie (słowo kluczowe interface)• Wszystkie metody publiczne klasy muszą być zadeklarowane w osobnym interfejsie

bns it } # } Kompozycja a dziedziczenie
 Jakość kodu źródłowego

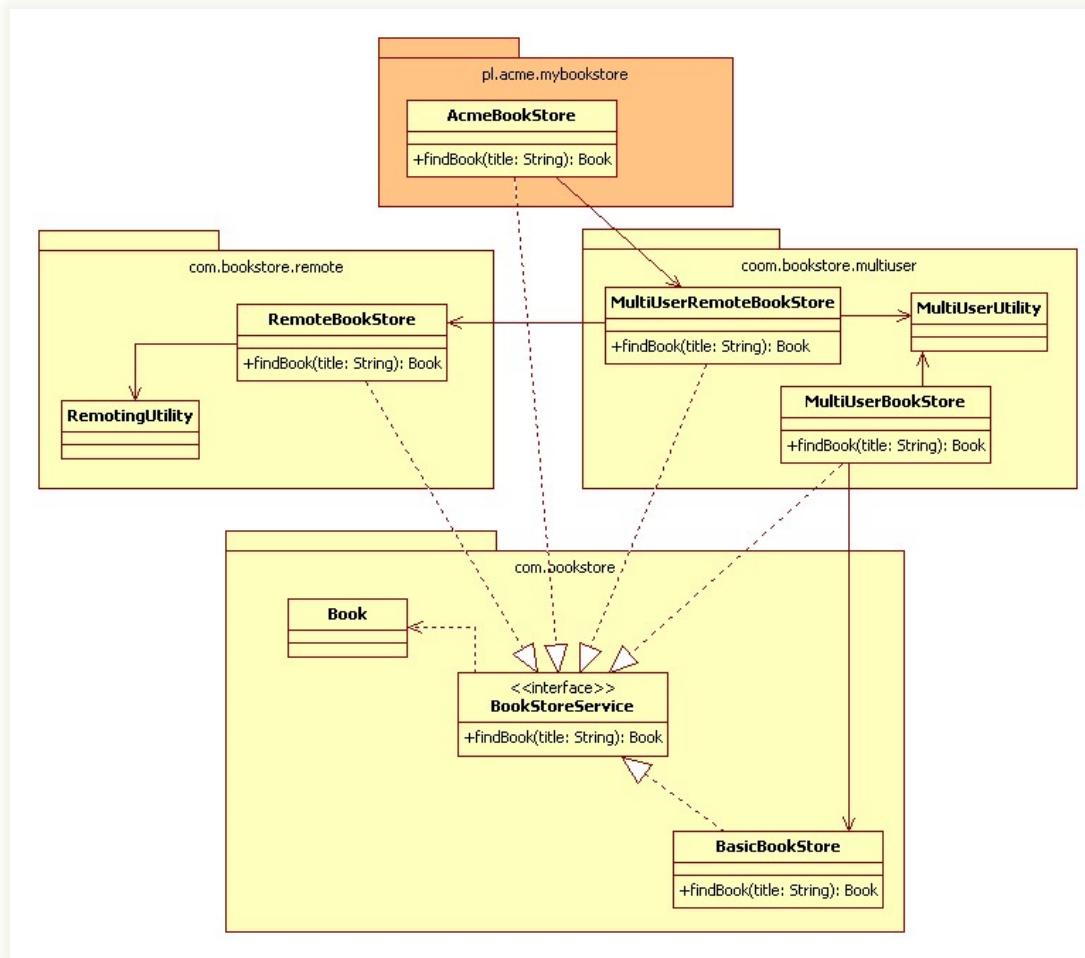
Kompozycja a dziedziczenie (Prosty przykład)



Kompozycja z dziedziczeniem (Bardziej złożony przykład)



Kompozycja z dziedziczeniem (Bardziej złożony przykład - kompozycja)



- # Dziedziczenie jest proste
- # Należy znać hierarchię dziedziczenia, aby zrozumieć metodę
- # Kod metody polimorficznej jest rozrzucony po kilku klasach

- # Nie zawsze jest oczywisty stan obiektu, z którego dziedziczymy
- # Trudno odcinać zależności od nadklas
- # Dodatkowe elementy (pola, metody) są silnie zależne z klasą nadzczną

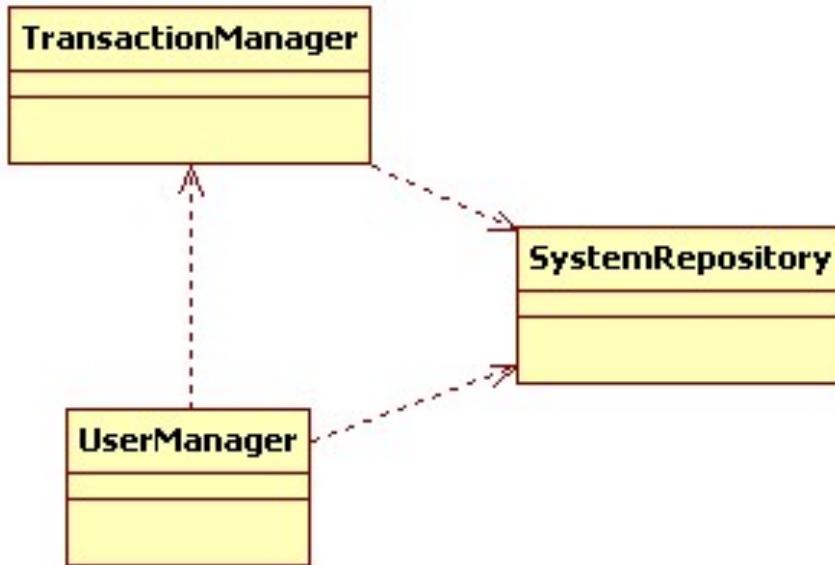
- # Więcej klas
- # Więcej kodu fabrykującego i wiążącego obiekty
- # Niezależność klas zawieranych
- # Można je używać w wielu kontekstach
- # Słabe zależności (*loose coupling*)
- # Łatwiej testować



Wstrzykiwanie zależności

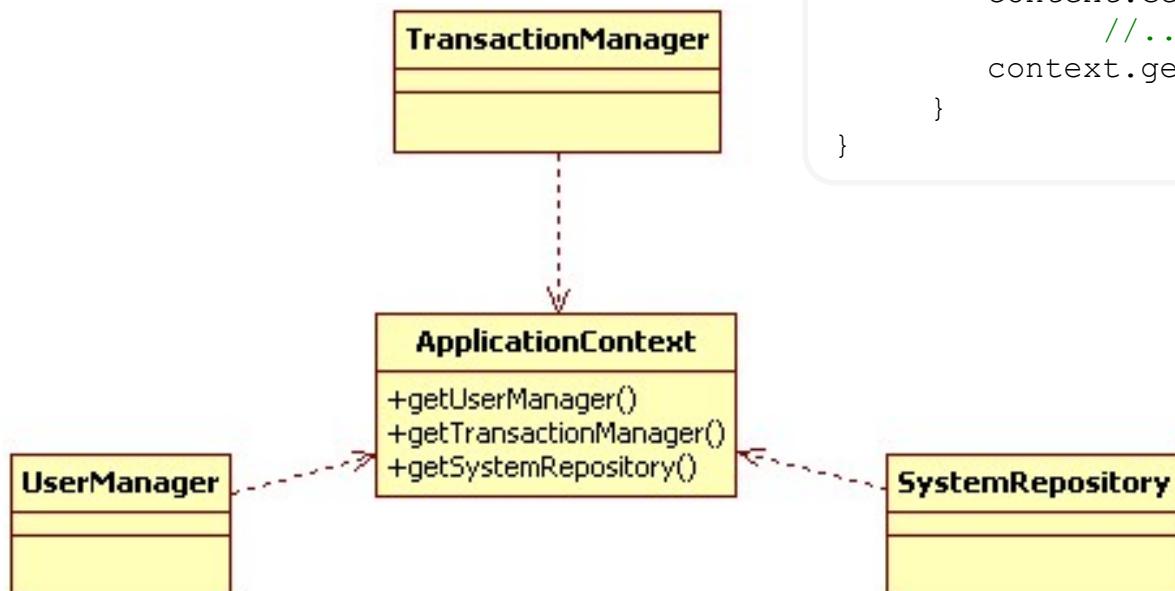
Jakość kodu źródłowego

W czym problem?



- # W jaki sposób klasa ma uzyskiwać dostęp do swoich współpracowników?
- # Użycie statycznego singletona lub tworzenie poprzez operator **new** wprowadzi silne zależności między klasami

Jawne pozyskiwanie zależności



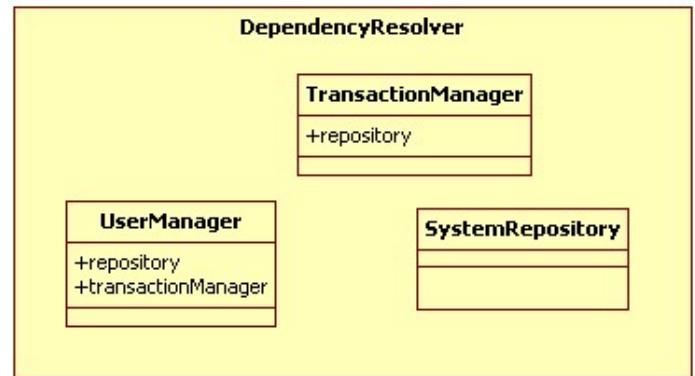
```
class UserManager
{
    private ApplicationContext context;

    public void Order()
        context.GetTransactionManager().begin();
        //...
        context.getTransactionManager.commit();
    }
}
```

Jawne pozyskiwanie zależności

- # Wprowadza uporządkowany sposób pozyskiwania zależności przez klasy
- # Klasa jest obarczona odpowiedzialnością za pozyskanie zależności z kontekstu
- # Obiekt kontekstu będzie rozrastać się nieograniczenie
- # Istnieje pokusa używania obiektu kontekstu jako miejsca do przechowywania globalnych zmiennych

Otwarcie na wstrzykiwanie zależności



```
class UserManager {
    // zależności klasy
    public SystemRepository repository { get; set; }
    public TransactionManager transactionManager; { get; set; }

    //dane prywatne klasy
    private Cart cart;
    private int lifetime;
}
```

- # Klasy deklarują (za pomocą.setterów lub konstruktorów) jakich zależności potrzebują
- # Zależność będą wstrzyknięte
- # Programista może założyć, że wymagane zależności zostaną w jakiś sposób dostarczone i zająć się kodem biznesowym

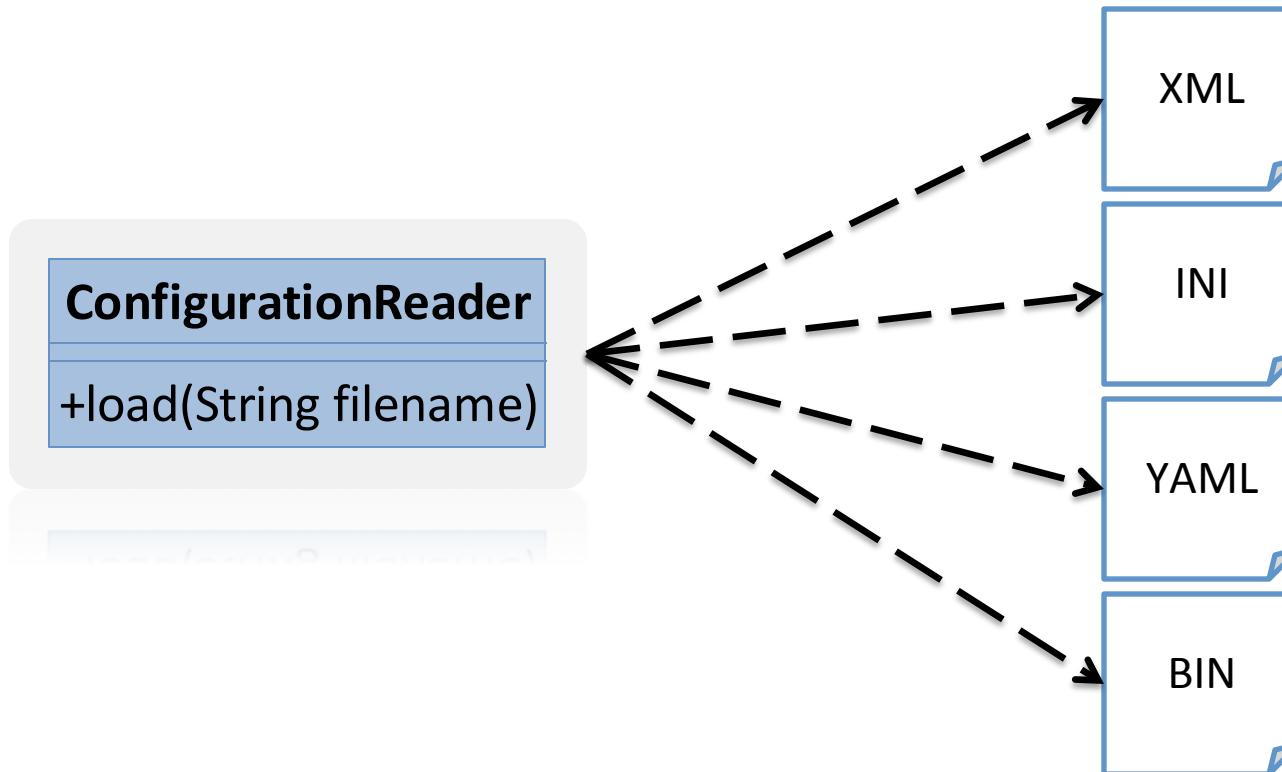
Wzorce kreacyjne GoF

Wzorce projektowe i refaktoryzacja do wzorców



- # Prowadzą do utworzenia obiektu.
- # Separują tworzenie obiektów od klienta, który je tworzy.
- # Ułatwiają budowę systemu, który jest niezależny od sposobu tworzenia i składania obiektów.

bns it } #} Wzorzec Simple Factory
Wzorce kreacyjne



```
String type = determineFileType(filename);
Serializer serializer
    = serializerFactory.create(type);
Object configuration
    = serializer.deserialize(filename);
```

ConfigurationReader

+load(String filename)

SerializerFactory

+create(String type): Serializer

```
if (type.equals("XML"))
    return new XMLSerializer();
else if (type.equals("YAML"))
    return new YAMLSerializer();
...
```

<<interface>>

Serializer

+serialize(Object o, String filename)
+deserialize(String filename): Object

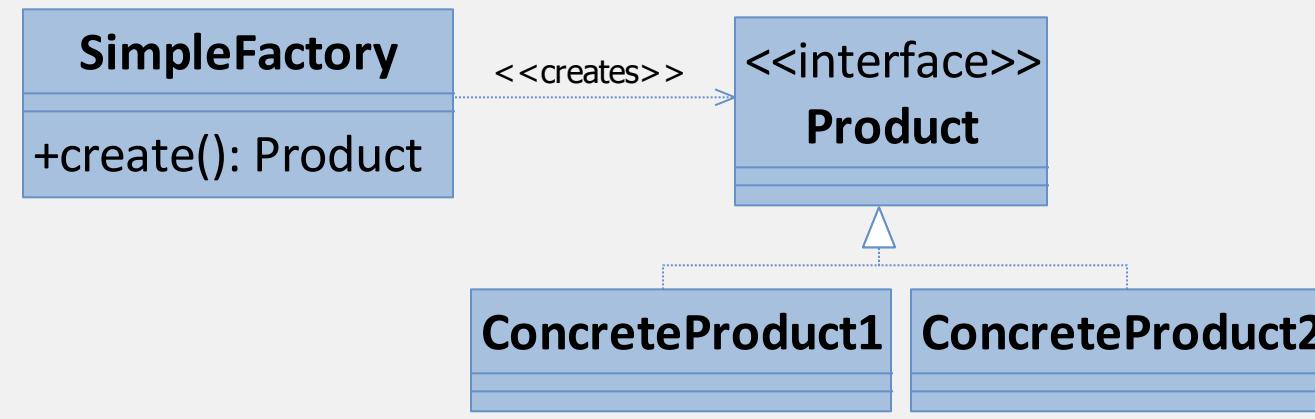
<<creates>>

XMLSerializer

INISerializer

YAMLSerializer

BinarySerializer

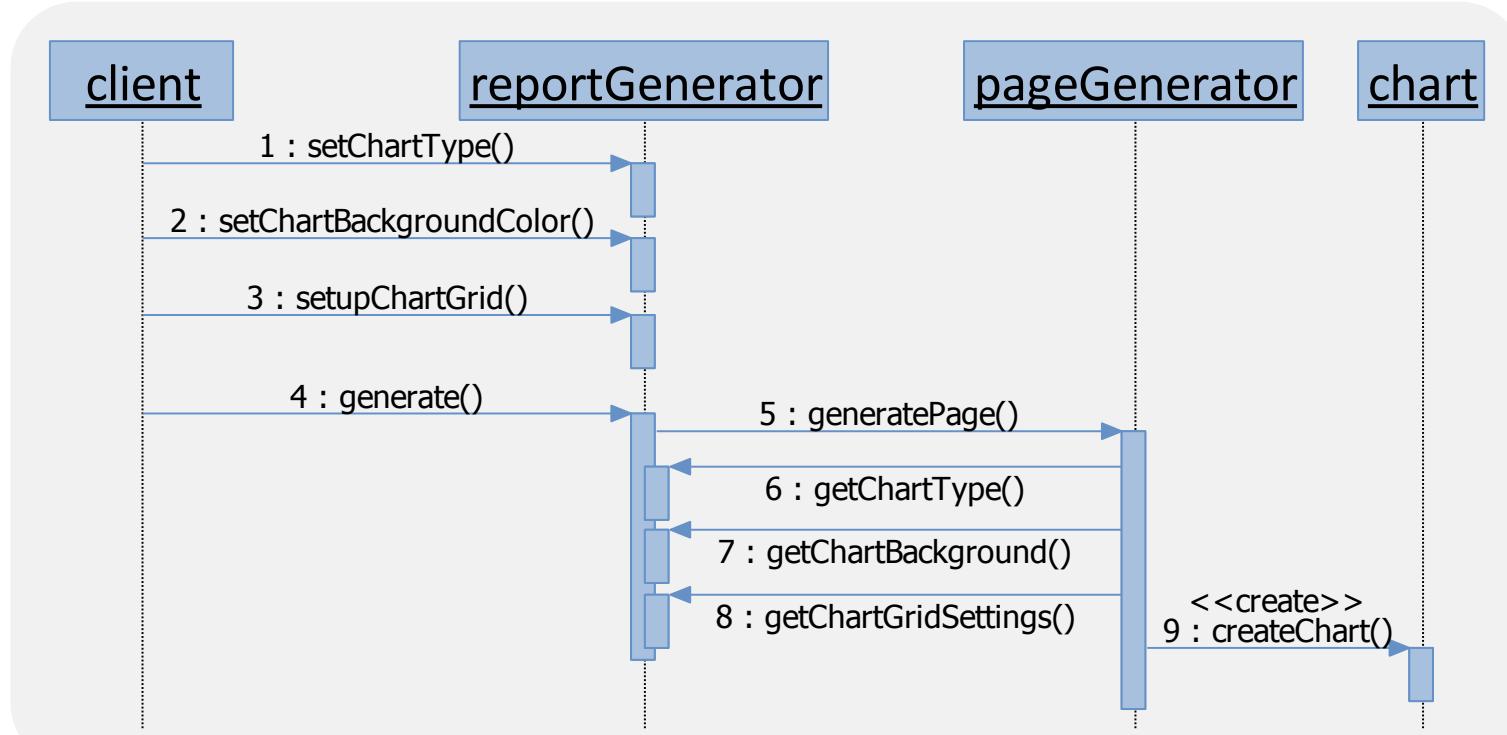


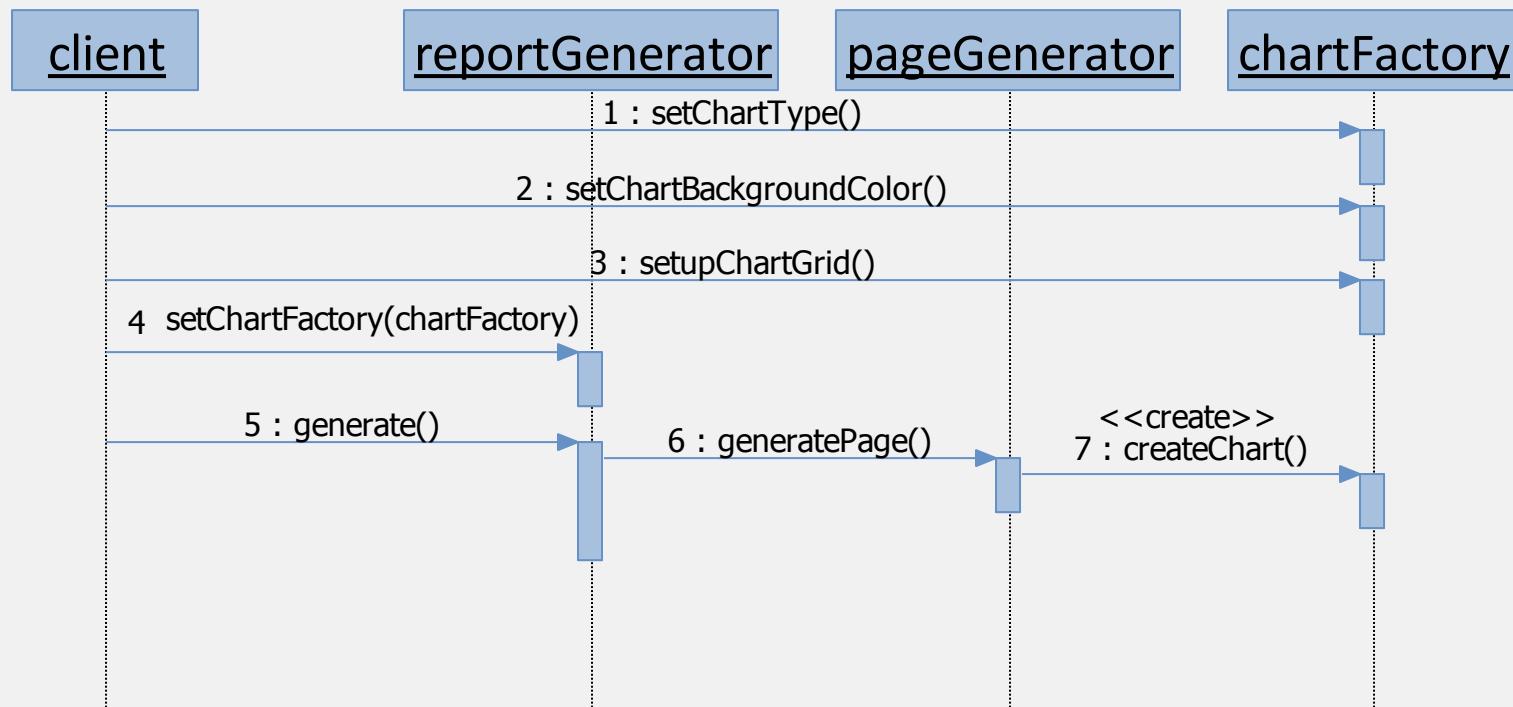
Wzorzec *Simple Factory* hermetyzuje tworzenie rodziny obiektów.

Podejmuje decyzję o tym jaki obiekt należy utworzyć i tworzy go.

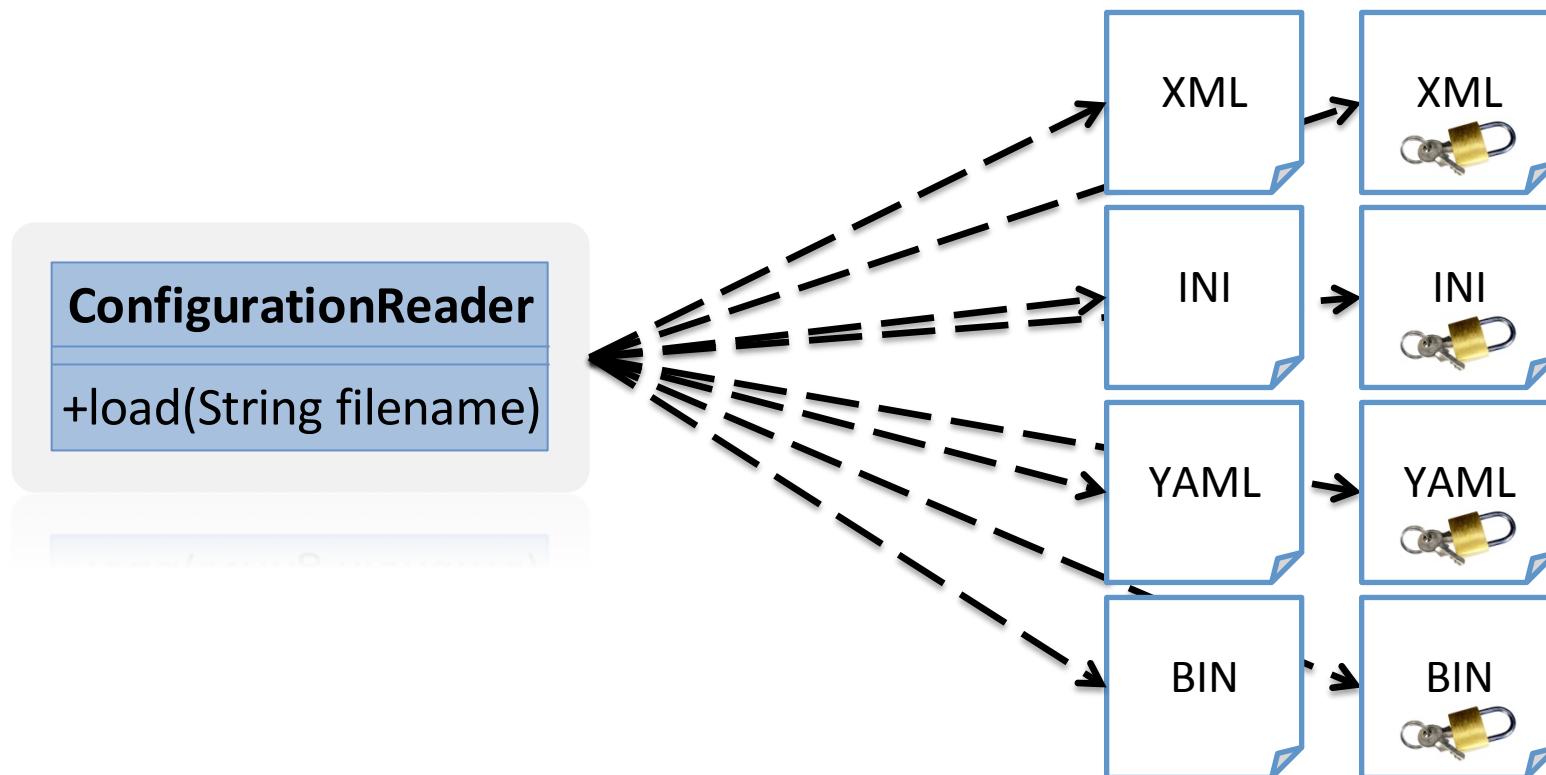
- # Wyodrębnienie tworzenia obiektów do osobnej dedykowanej ku temu klasy
- # Prosta fabryka tworząca obiekt połączenia do bazy danych określonego typu
- # Prosta fabryka tworząca obiekt modelu danych na podstawie jego sygnatury

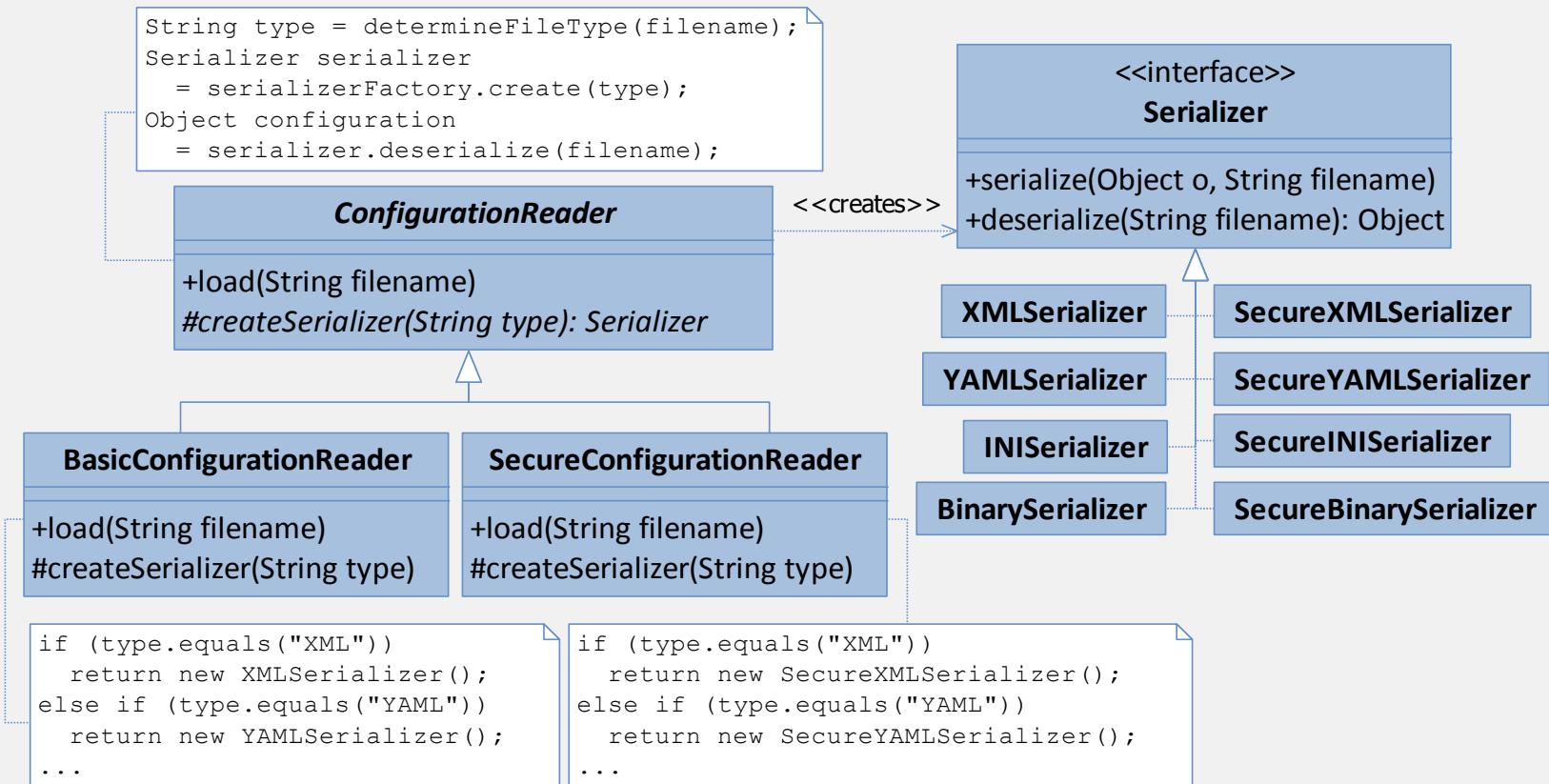
Refaktoryzacja: *Move Creation Knowledge to Factory*



Refaktoryzacja: *Move Creation Knowledge to Factory*

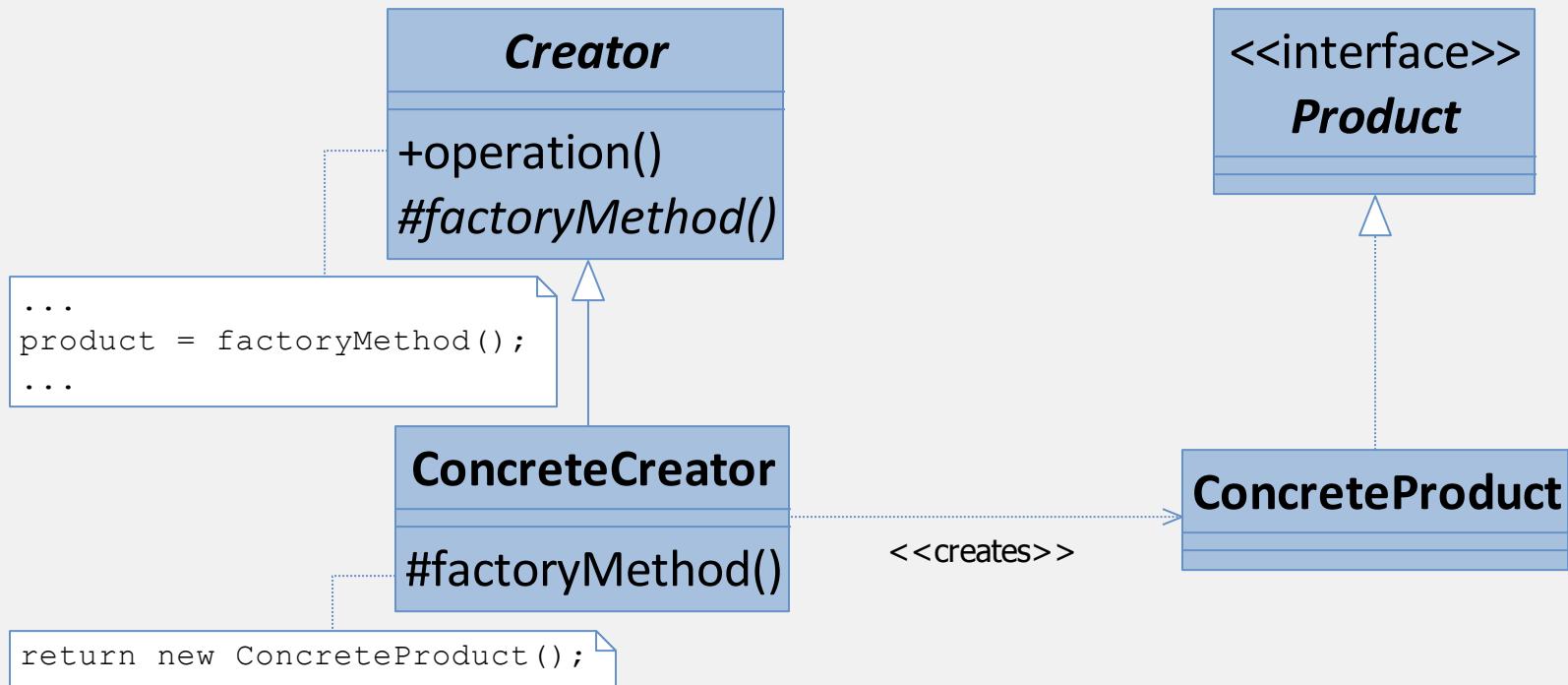
bns it } # Wzorzec Factory Method
Wzorce kreacyjne





```
***  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u
```

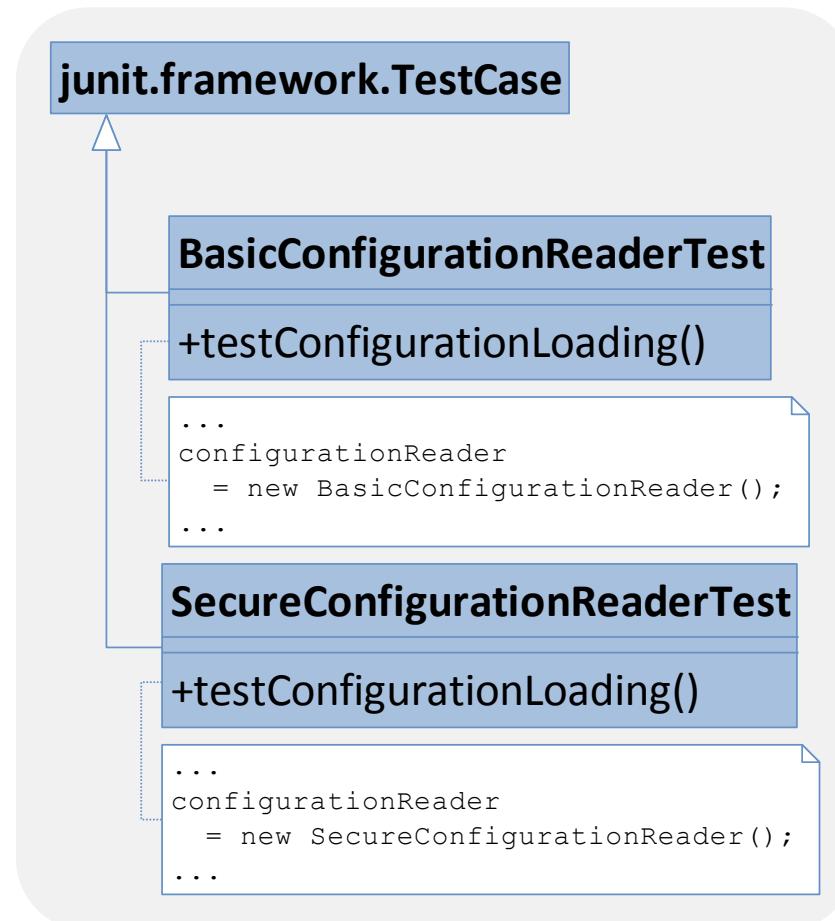
```
***  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u  
:() i e s t l b i t e r i s M A Y w e n u r u f u
```



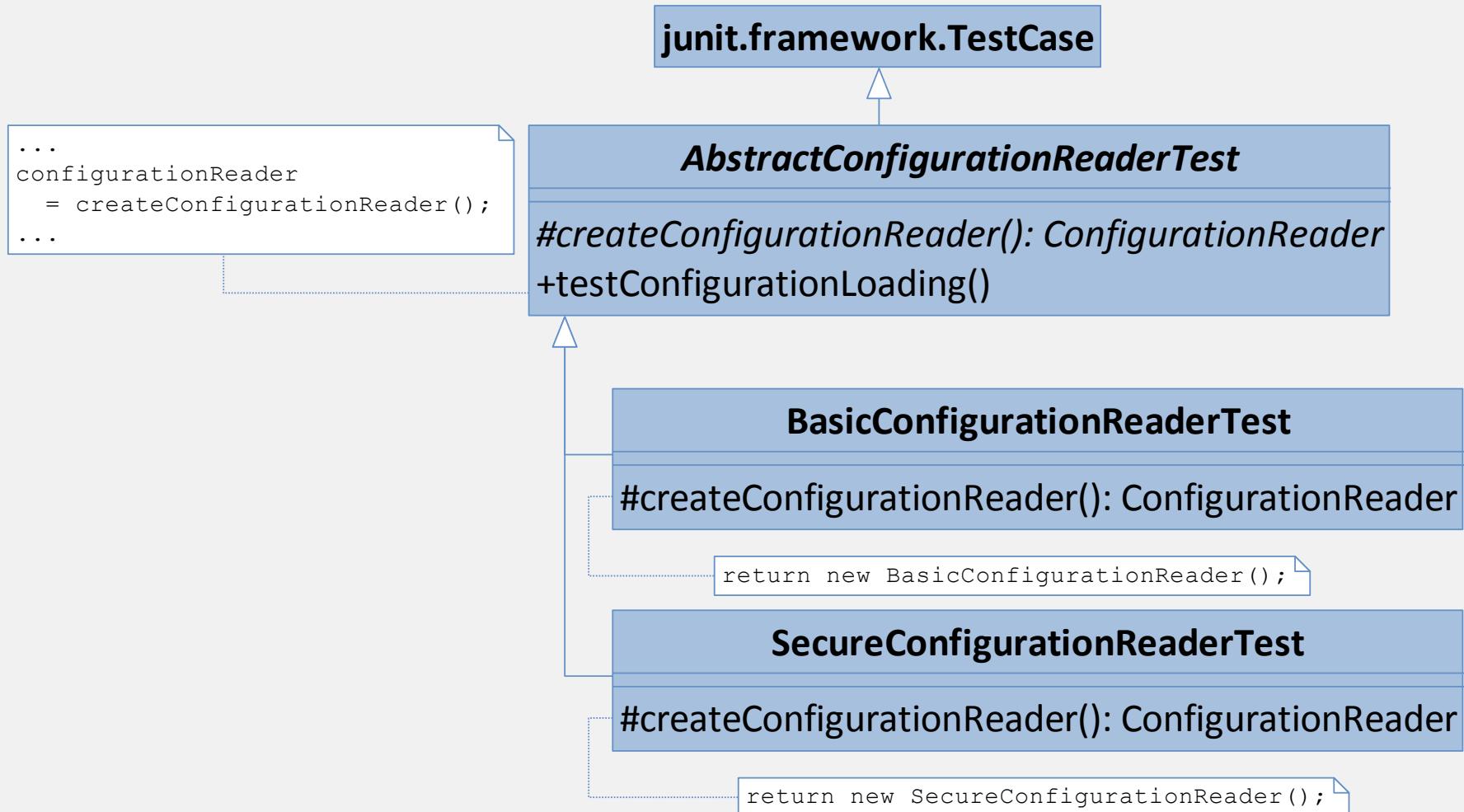
Wzorzec *Factory Method* określa interfejs do tworzenia obiektów, lecz decyzje o tym jaki obiekt ma zostać utworzony pozostawia swoim podklasom.

- # *Factory Method* eliminuje potrzebę współpracy z klasami *ConcreteProduct*. Klient odwołuje się tylko do interfejsu klasy *Product*.
- # W przypadku kiedy klient będzie potrzebował obiektu klasy *ConcreteProduct* ze względu na jego specyficzny interfejs, będzie musiał wykonać rzutowanie.
- # Klasa *Creator* może zapewnić domyślną implementację metody *factoryMethod()* i umożliwić podklasom dostarczenie rozszerzonej wersji obiektu *Product* w przyszłości.

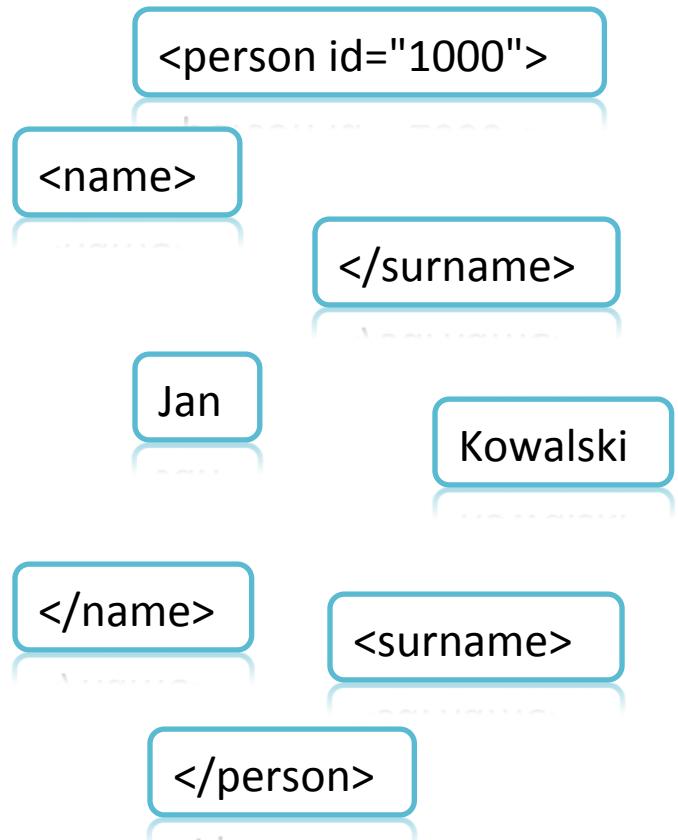
- # W przypadku gdy informacje o tym, jaki rodzaj obiektu ma być utworzony chcemy ulokować w klasach pochodnych.
- # Definicja interfejsu pewnego frameworku, który jest implementowany przez użytkownika tego frameworku.



Refaktoryzacja: *Introduce Polymorphic Creation with Factory Method*



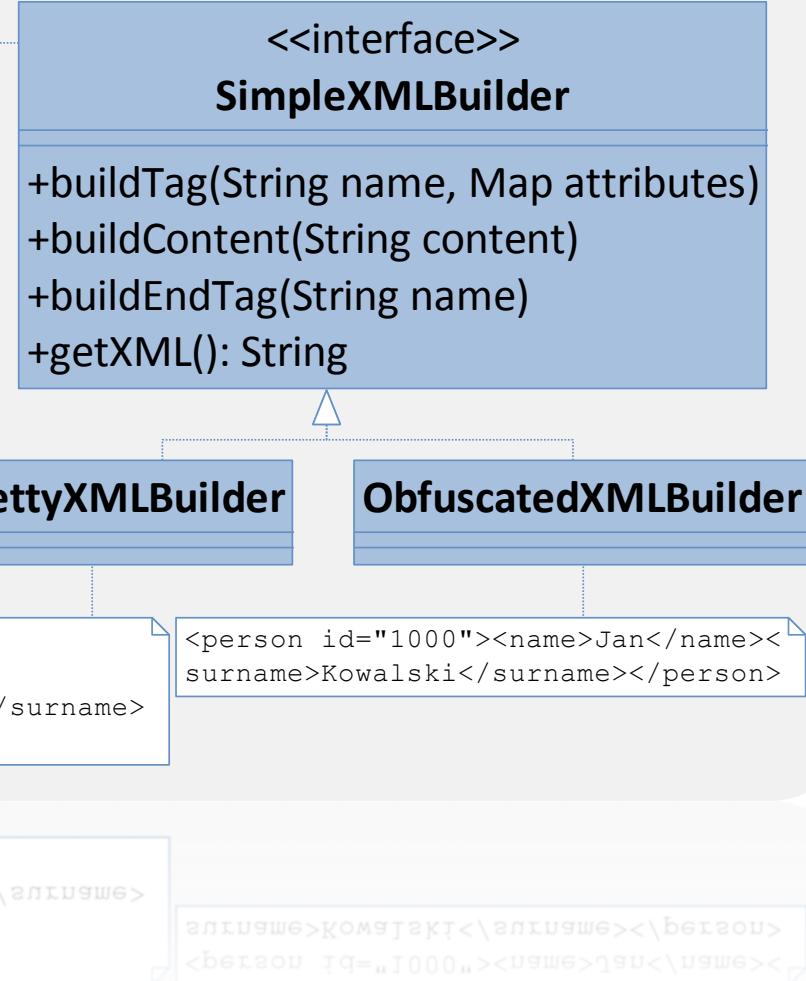
bns it} #} Wzorzec Builder
Wzorce kreacyjne

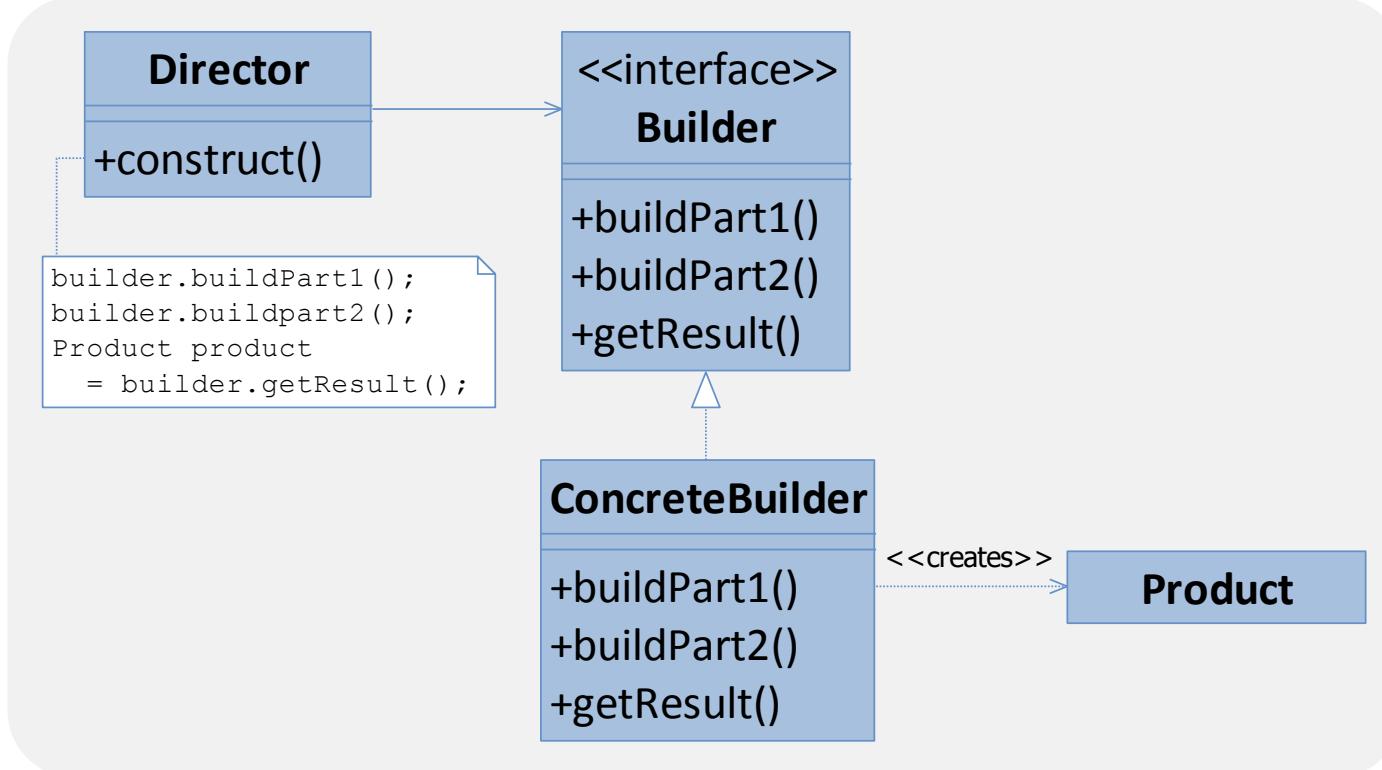


```
<person id="1000">
  <name>Jan</name>
  <surname>Kowalski</surname>
</person>
```

<\person>

```
SimpleXMLBuilder builder
    = new PrettyXMLBuilder();
Map attributes = new HashMap();
attributes.put("id", 1000);
builder.buildTag("person", attributes);
    builder.buildTag("name")
    builder.buildContent("Jan")
    builder.buildEndTag("name")
    builder.buildTag("surname")
    builder.buildContent("Kowalski")
    builder.buildEndTag("surname")
builder.buildEndTag("person")
String xml = builder.getXML();
```





+getResutl()
+buildPart1()
+buildPart2()

Product

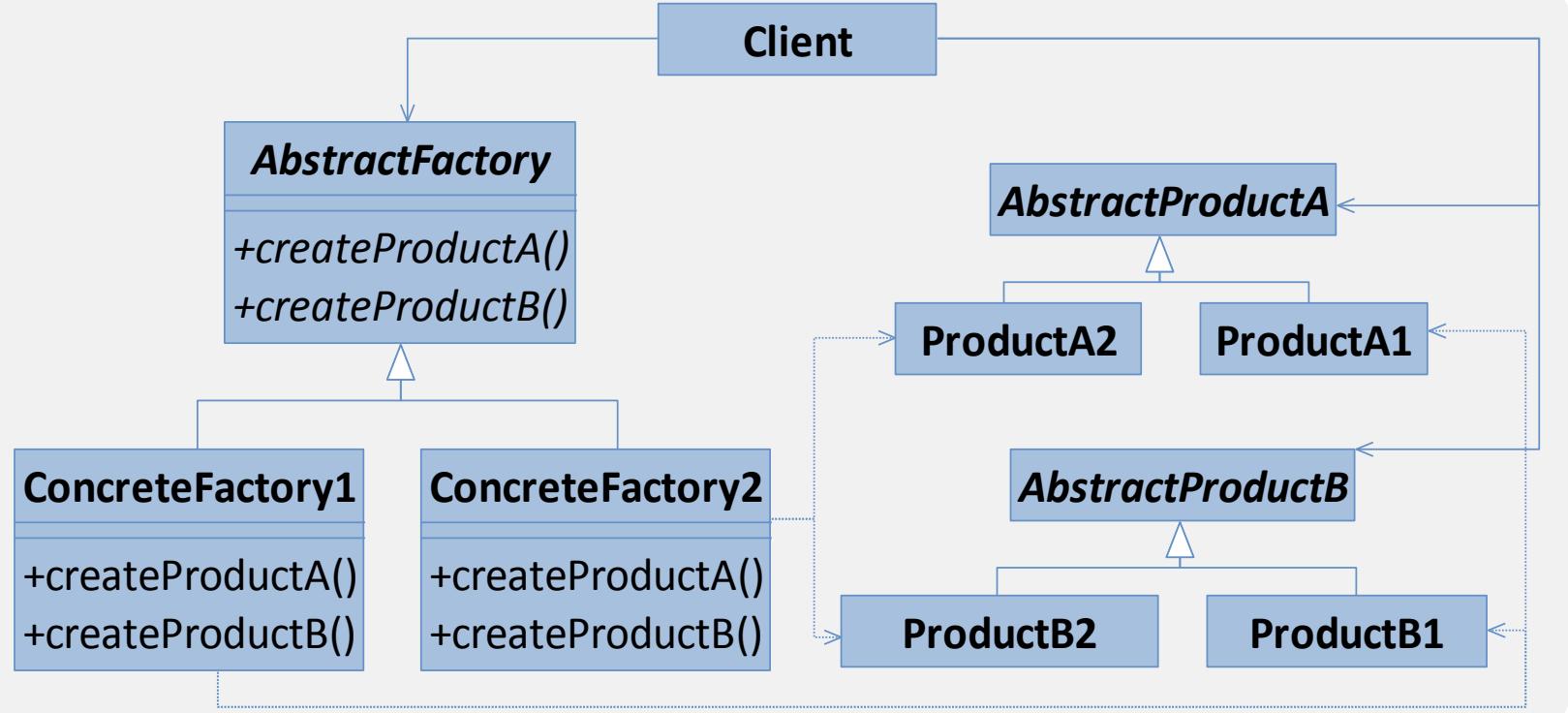
Wzorzec *Builder* pozwala konstruować obiekty z komponentów.

- # Wzorzec *Builder* oddziela kod służący do konstruowania produktu od jego wewnętrznej reprezentacji.
- # Klient nie musi nic wiedzieć o wewnętrznej strukturze obiektu *Product* stworzonego przez obiekt *Builder*.
- # Klient ma wysoką kontrolę nad procesem tworzenia obiektu, gdyż nadzoruje ten proces krok po kroku.

- # W przypadku gdy potrzebujemy skomplikowaną strukturę obiektową konstruować w prosty sposób.
- # Kreator tworzący obiekty w kilku etapach
- # Obiekt odpowiedzialny za składanie tekstu i generowanie raportu

bns it } #} Wzorzec Abstract Factory
Wzorce kreacyjne

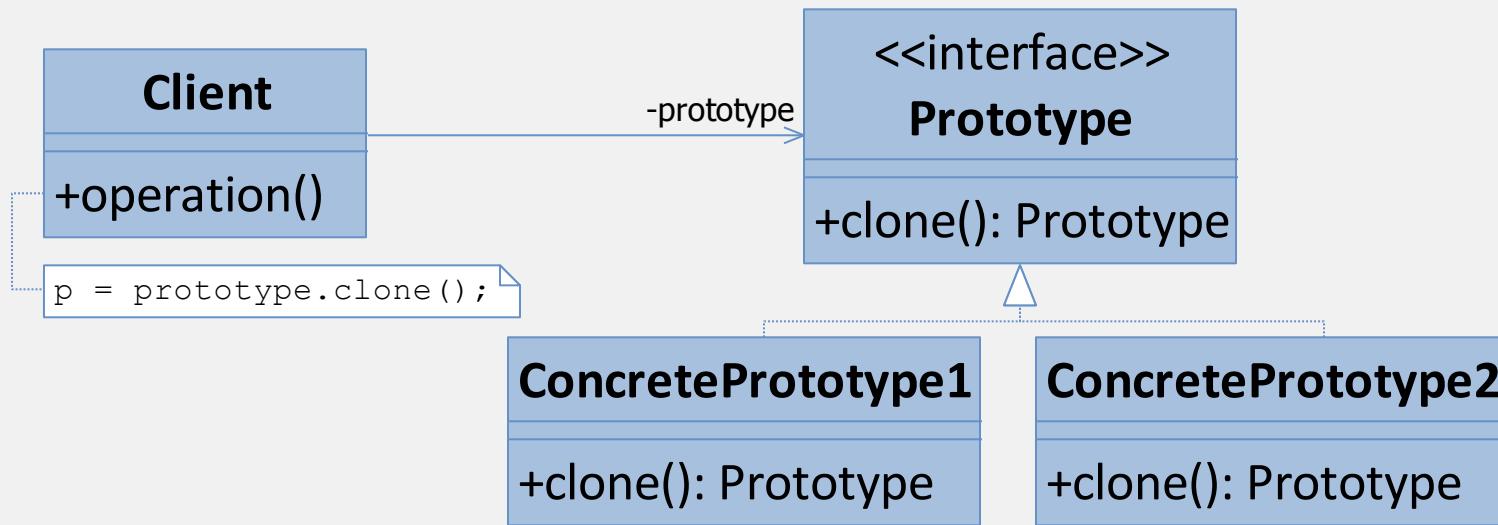
Wzorzec *Abstract Factory* zapewnia interfejs umożliwiający tworzenie wielu rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez specyfikowania ich klas konkretnych.



- # Wzorzec *Abstract Factory* pozwala określić, jakie klasy obiektów tworzy dana aplikacja.
- # Klient nigdy nie tworzy obiektów sam, zawsze zleca to fabryce.
- # Wymiana całej rodziny produktów używanych przez aplikację jest prosta. Ogranicza się do wymiany obiektu *ConcreteFactory*.
- # Obsługa nowej rodziny produktów wymaga rozszerzenia interfejsu *AbstractFactory* i wszystkich jej podklas.

bns it} #} Wzorzec Prototype
Wzorce kreacyjne

Wzorzec *Prototype* tworzy nowe obiekty na podstawie prototypowego egzemplarza.



- # Dodanie nowego produktu do systemu polega na zarejestrowaniu nowego prototypu u klienta.
- # Wzorzec *Prototype* może znacznie zmniejszyć liczbę klas potrzebnych w systemie.
- # Wzorzec *Prototype* umożliwia definiowanie nowego zachowania poprzez składanie obiektów.
- # Każda podklasa klasy *Prototype* musi implementować własną metodę *clone()*.

bns it } #} Wzorzec Singleton
Wzorce kreacyjne

Wzorzec *Singleton* zapewnia, że klasa ma tylko jeden egzemplarz i zapewnia globalny dostęp do niego.

Singleton

-instance: Singleton

-attribute1

-attribute2

+operation()

+getInstance(): Singleton

```
return instance;
```

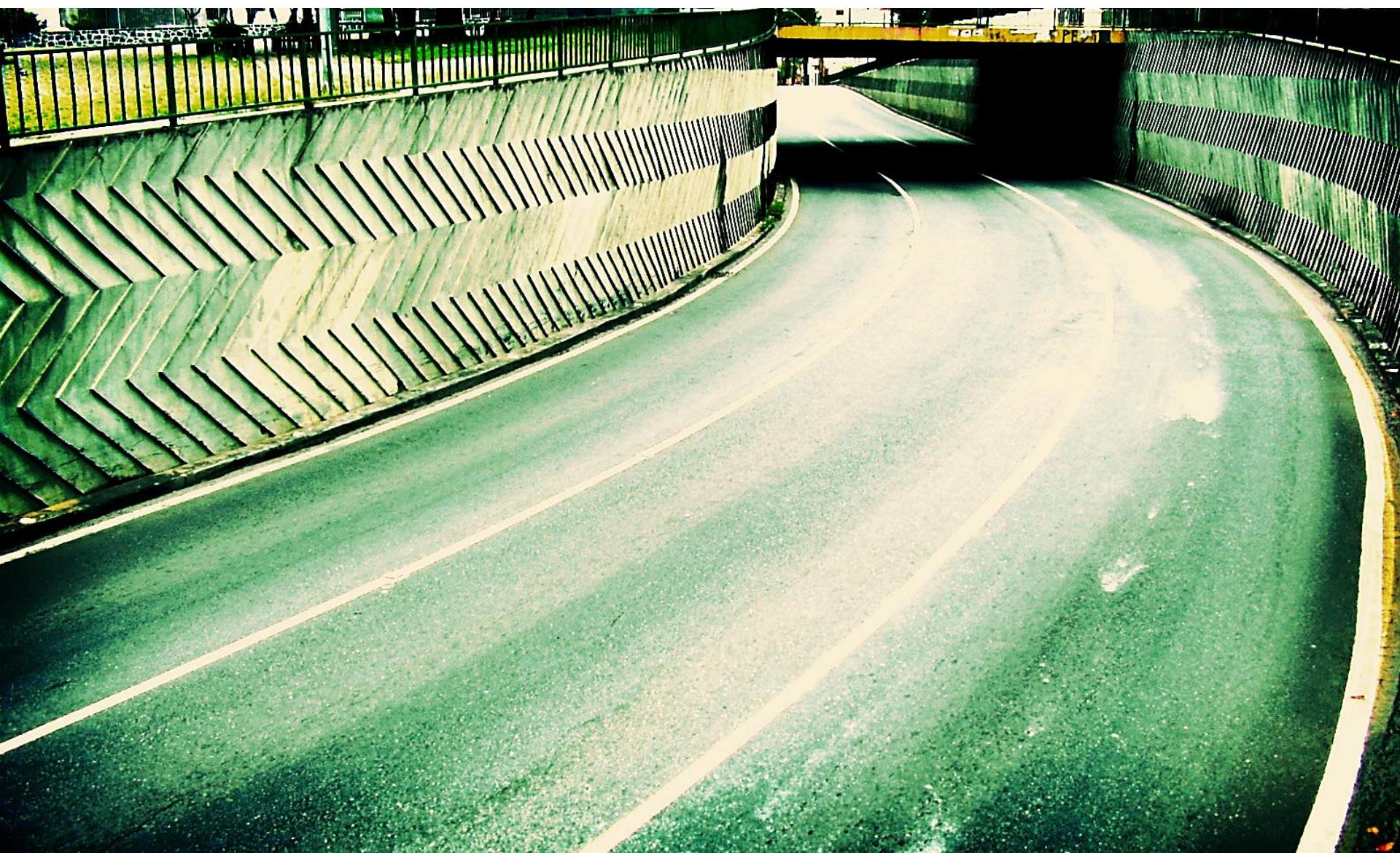
return instance;

Recurrence (Von Ricken)

- # Klasa *Singleton* może ścisłe kontrolować dostęp do swojego jedynego egzemplarza.
- # Można użyć wzorca *Singleton* do kontrolowania liczby obiektów typu *Singleton*.
- # Wzorzec *Singleton* wprowadza zmienną globalną i usztywnia projekt.
- # Należy unikać stosowania wzorca *Singleton*, chyba że jest to konieczne.

Wzorce behawioralne GoF

Wzorce projektowe i refaktoryzacja do wzorców



- # Dotyczą interakcji pomiędzy klasami i obiektami.
- # Omawiają sposoby podziału odpowiedzialności pomiędzy klasami.

bns it } # } Wzorzec Command
Wzorce behawioralne

Przykład

```
> java CommandExample ShowHelp  
ShowHostIP - Display computer IP.  
ShowMachineName - Display name of the computer.  
ListCurrentDirectory - Lists files in current directory.
```



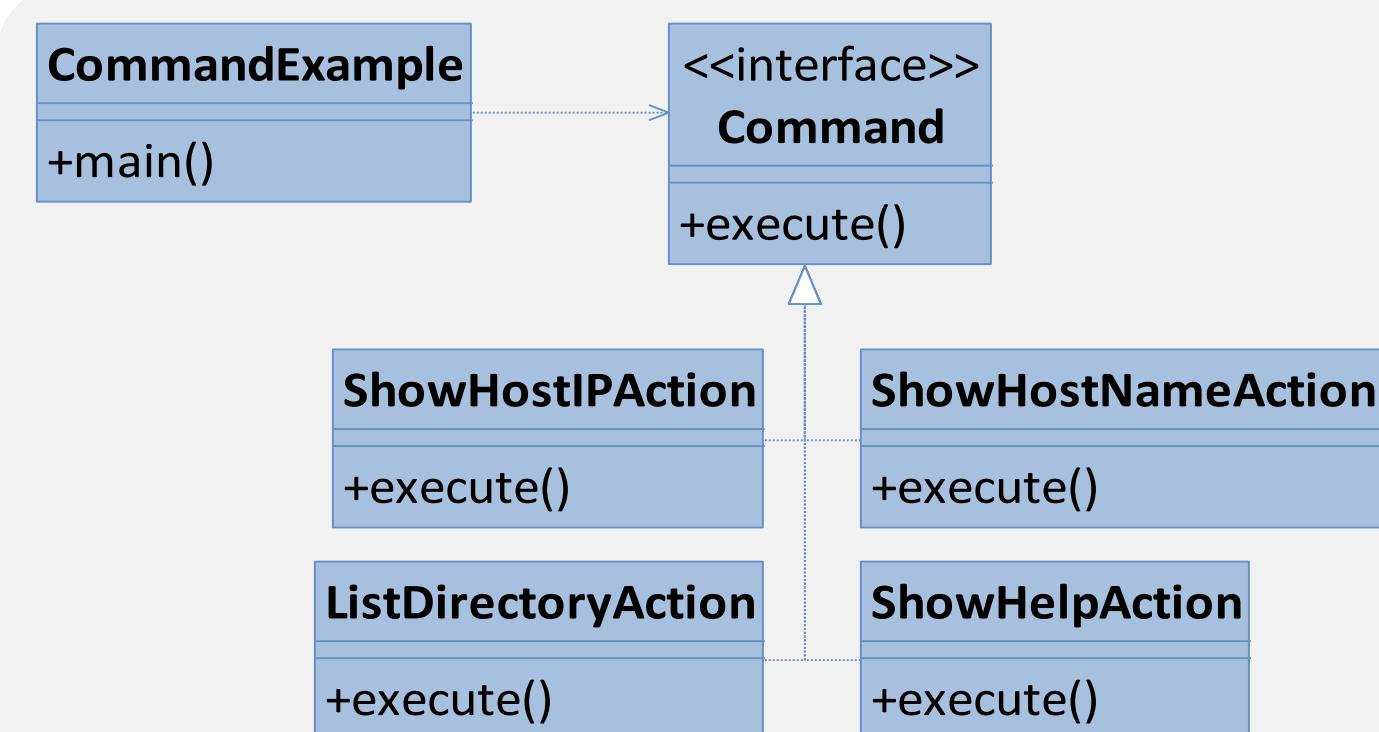
```
> java CommandExample ShowHostIP  
192.168.0.5
```

```
> java CommandExample ShowMachineName  
BNSServer
```



```
> java CommandExample ListDirectory  
File1.txt  
File2.txt
```

Przykład



Kod przykładu

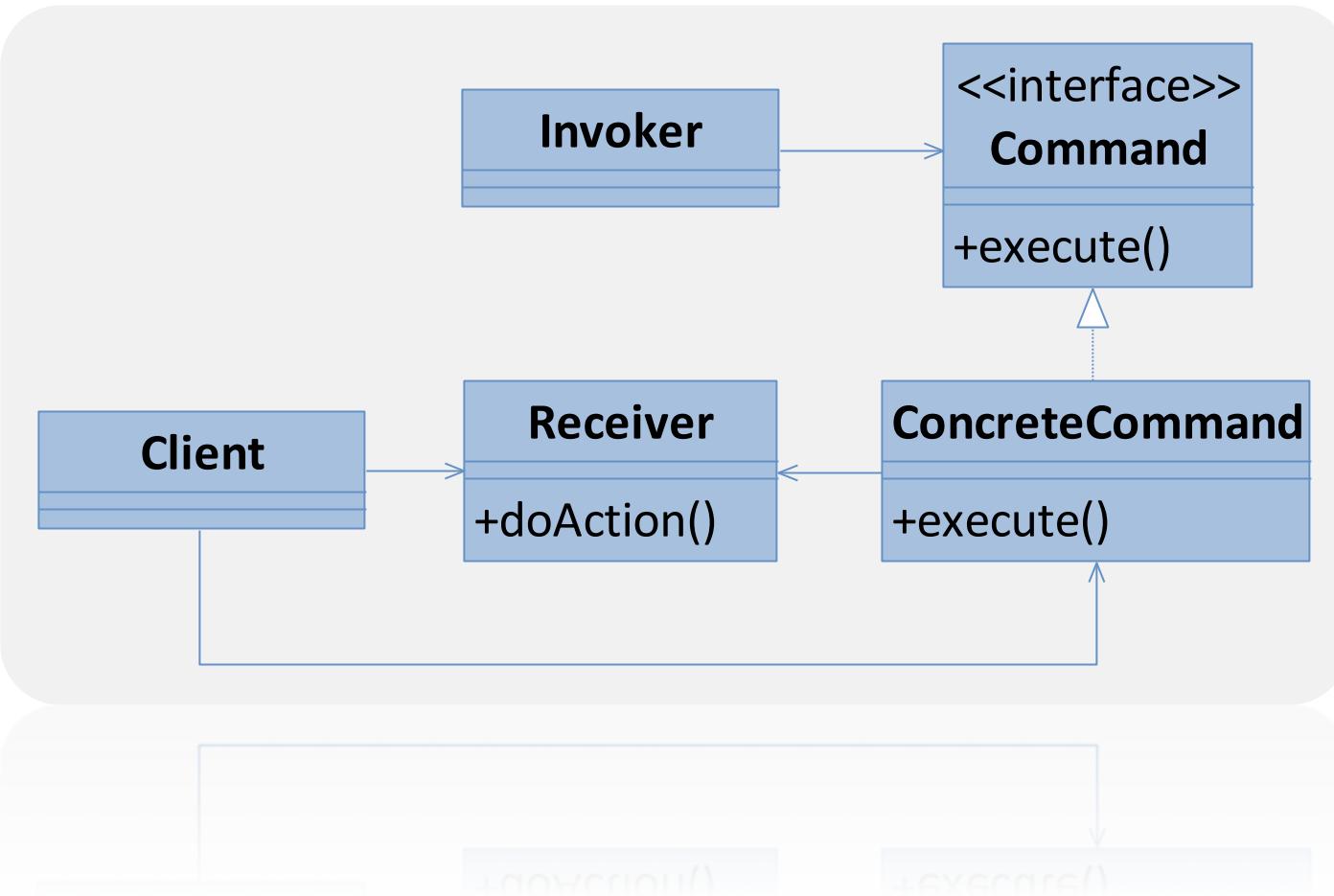
```
interface Command {  
    public void Execute();  
}
```

```
public class ListDirectoryAction  
: Command { ... }
```

```
public class ShowHostNameAction  
: Command { ... }
```

```
public class ShowHostIPAction : Command {  
  
    public void Execute() {}  
  
}
```

```
public class CommandExample {  
    public static void main(String[] args) {  
        if (args.Length == 0) return ;  
  
        String command = args[0];  
        if (command.Equals("ShowHelp")) {  
            new ShowHelpAction().Execute();  
        } else if (command.Equals("ShowHostIP")) {  
            new ShowHostIPAction().Execute();  
        }  
    }  
}
```



Wzorzec *Command* hermetyzuje żądania w postaci obiektów.

- # Wzorzec *Command* separuje obiekt wywołujący polecenie od obiektu, który wie jak je zrealizować.
- # Obiekty *Command* mogą być rozszerzane tak jak inne obiekty.
- # Dodawanie obiektów *Command* nie wymaga modyfikowania istniejących obiektów klas.

- # Implementowanie wycofywalnych operacji
- # Kolejkowanie zadań
- # Księgowanie żądań

Refaktoryzacja: *Replace Conditional Dispatcher with Command*

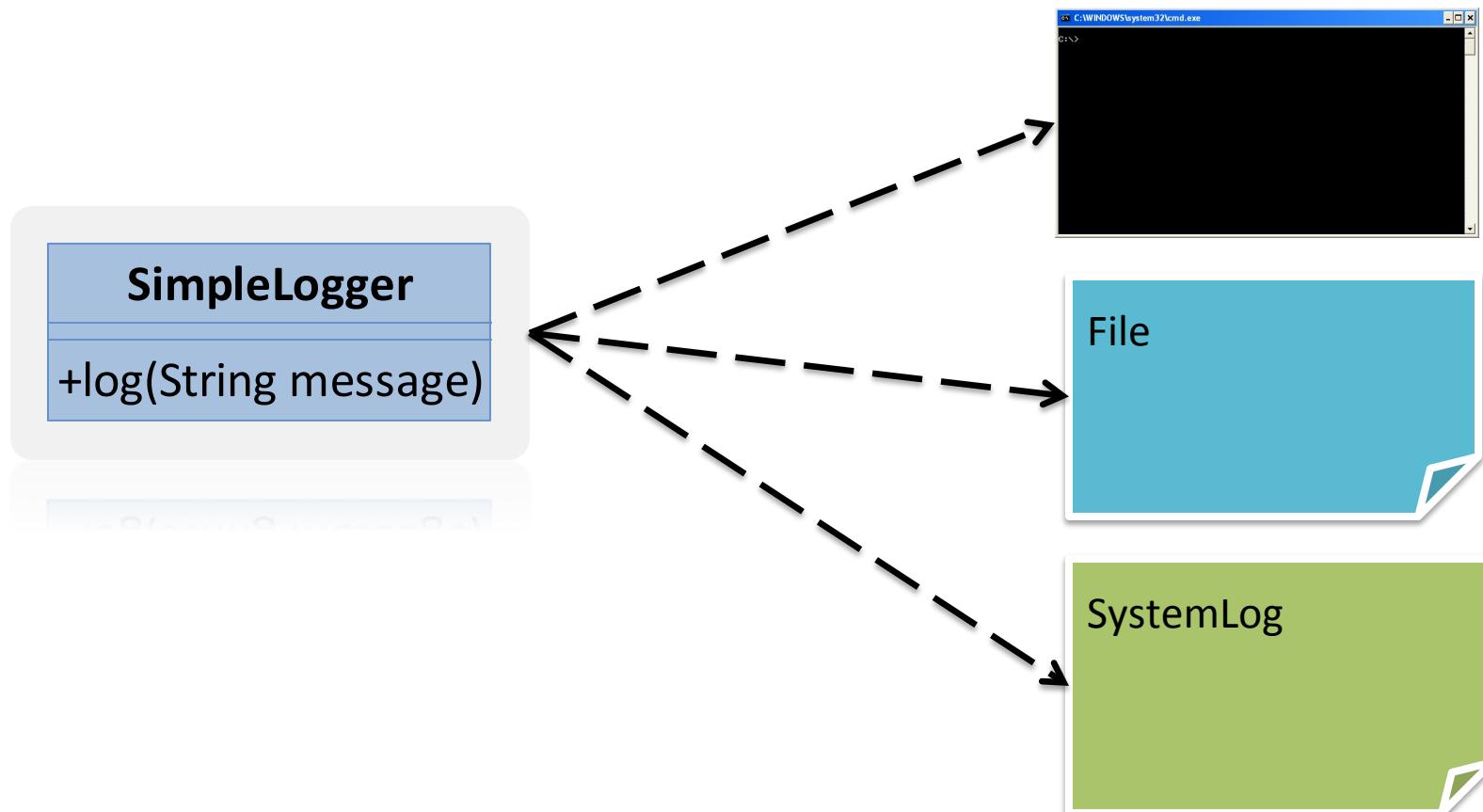
```
if (command.Equals("ShowHelp")) {  
    //...  
} else if (command.equals("ShowHostIP")) {  
    //...  
} else if (command.equals("ShowHostName")) {  
    //...  
} else if (command.equals("ListCurrentDirectory")) {  
    //...  
}
```

Refaktoryzacja: *Replace Conditional Dispatcher with Command*

```
private Dictionary<String, Command> commands = new HashMap<String, Command>();  
  
public CommandFullExample() {  
    commands["ShowHelp"] = new ShowHelpAction();  
    commands["ShowHostIP"] = new ShowHostIPAction();  
    commands["ShowHostName"] = new ShowHostNameAction();  
    commands["ListDirectory"] = new ListDirectoryAction();  
}  
  
public void Main(String[] args) {  
    Command command = commands[args[0]];  
    command.Execute();  
}
```

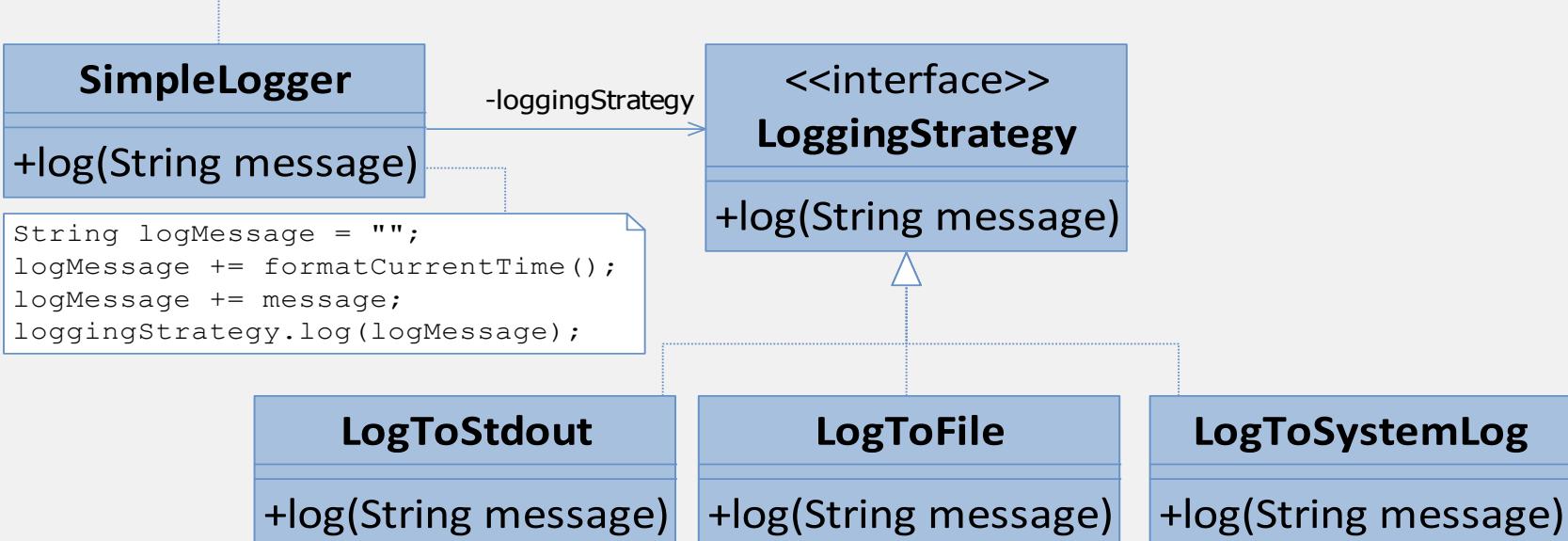
bns it } # Wzorzec Strategy
Wzorce behawioralne

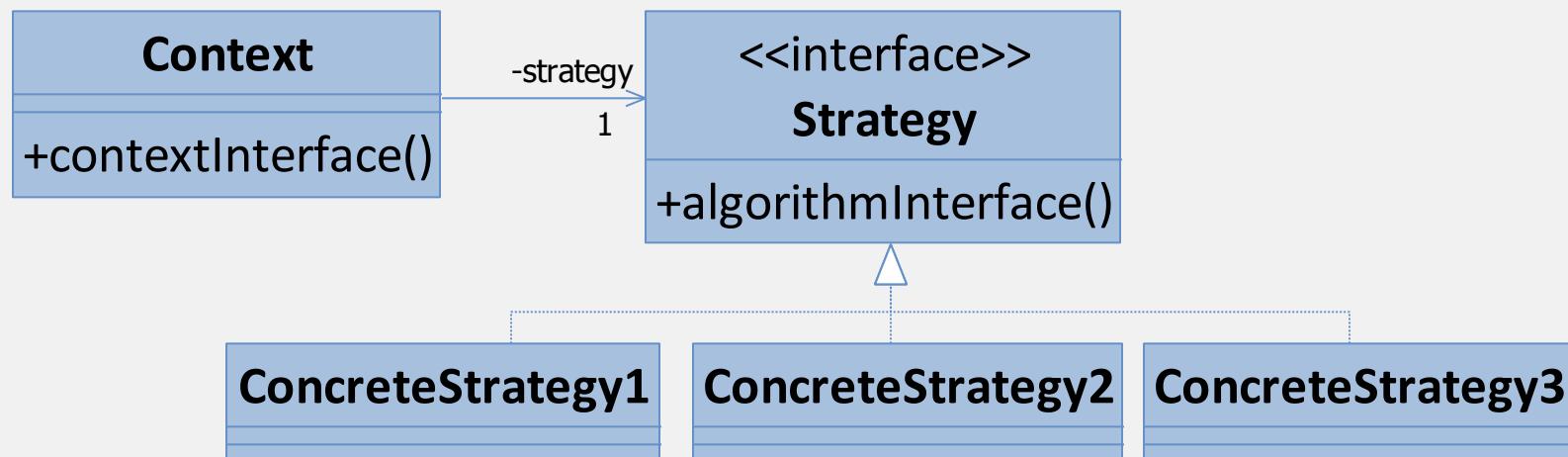
Przykład



Przykład

```
Logger logger = new SimpleLogger(new LogToFile("log.txt"));
logger.log("A very important message.");
```





Wzorec *Strategy* tworzy rodzinę podobnych algorytmów i daje możliwość ich podmiany w trakcie działania programu.

- # Wzorzec *Strategy* definiuje rodzinę algorytmów powiązanych ze sobą.
- # Korzystanie z wzorca *Strategy* może być alternatywą dla korzystania z dziedziczenia w celu wyodrębnienia nowych algorytmów.
- # Hermetyzacja algorytmu w osobnych klasach *ConcreteStrategy* umożliwia jego modyfikowanie niezależnie od klasy *Context*.

- # Wzorzec *Strategy* eliminuje przeładowane instrukcje warunkowe.
- # Klient musi rozumieć, czym różnią się poszczególne strategie, aby móc dokonać dobrego wyboru strategii.
- # Interfejs *Strategy* jest identyczny dla wszystkich algorytmów, co może prowadzić do przekazywania niepotrzebnych parametrów.
- # Nadmierne stosowanie wzorca *Strategy* może doprowadzić do powstania dużej ilości małych obiektów w systemie.

- # Implementacja jednego algorytmu na różne sposoby
- # Serializowanie danych do plików o różnych formatach
- # Wyliczanie fragmentu algorytmu na różne sposoby
- # Zapisywanie obrazka z wykorzystaniem różnych rodzajów algorytmów

Refaktoryzacja: *Replace Conditional Logic with Strategy*

```
Logger logger = new SimpleLogger(LoggerType.STDOUT);
logger.Log("A very important message.");
```

```
public void Log(String message) {
    String logMessage = "";
    logMessage += FormatCurrentTime();
    logMessage += message;

    if (loggerType == LoggerType.STDOUT) {
        Console.WriteLine(logMessage);

    } else if (loggerType == LoggerType.FILE) {
        //...
    } // else if ...

    // else if ...
    // else if ...
}
```

Refaktoryzacja: *Replace Conditional Logic with Strategy*

```
Logger logger = new SimpleLogger(new LogToFileStrategy("log.txt"));
logger.Log("A very important message.");
```

```
public void Log(String message) {
    String logMessage = "";
    logMessage += FormatCurrentTime();
    logMessage += message;

    loggingStrategy.Log(logMessage);
}
```



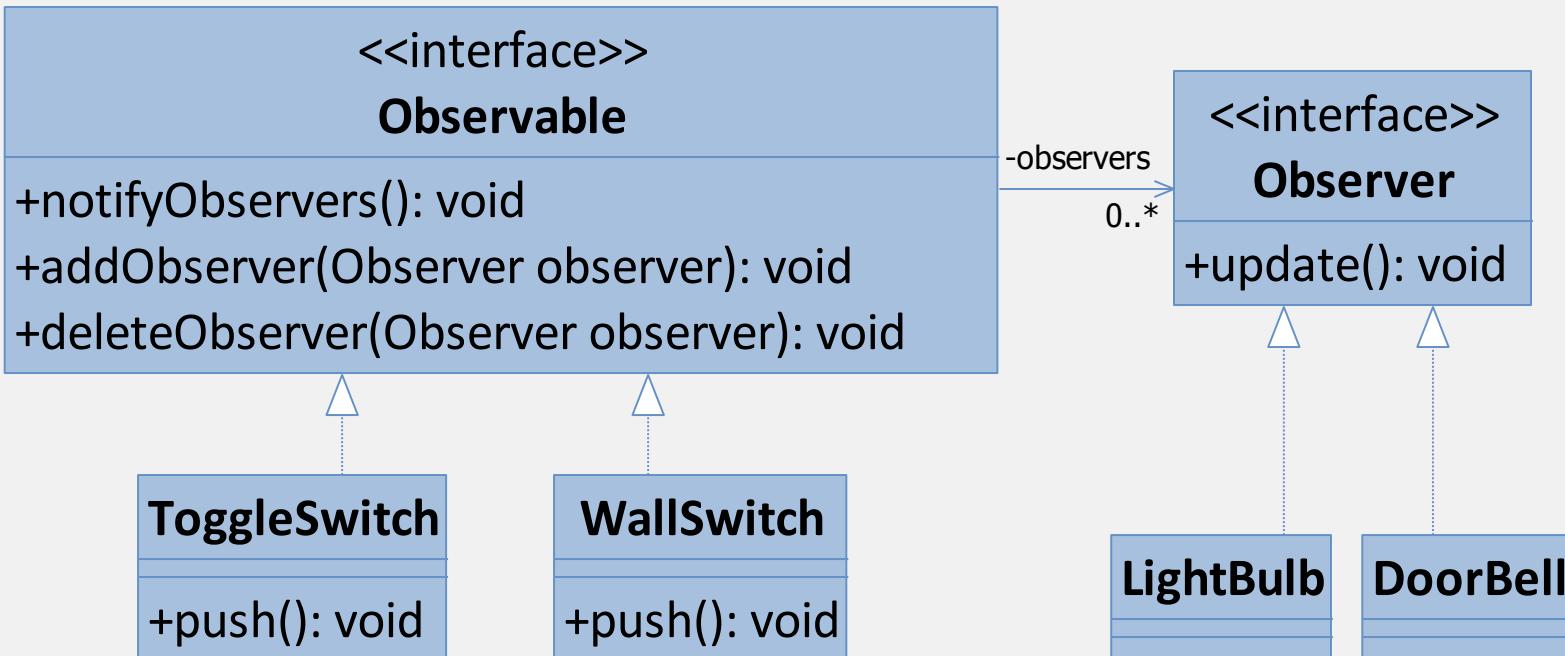
Wzorzec Observer

Wzorce behawioralne

Zagadnienie projektowe



Struktura przykładu



Kod przykładu

```
interface Observable {  
    public void AddObserver(Observer observer);  
    public void DeleteObserver(Observer observer);  
    public void NotifyObservers();  
}  
  
public class WallSwitch : Observable {  
    private List<Observer> observers ...  
  
    public void Push() {  
        NotifyObservers();  
    }  
  
    public void AddObserver(Observer observer) {  
        observers.Add(observer);  
    }  
  
    public void DeleteObserver(Observer observer) {  
        observers.Remove(observer);  
    }  
  
    public void NotifyObservers() {  
        foreach(Observer activeable in observers) {  
            activeable.Update();  
        }  
    }  
}
```

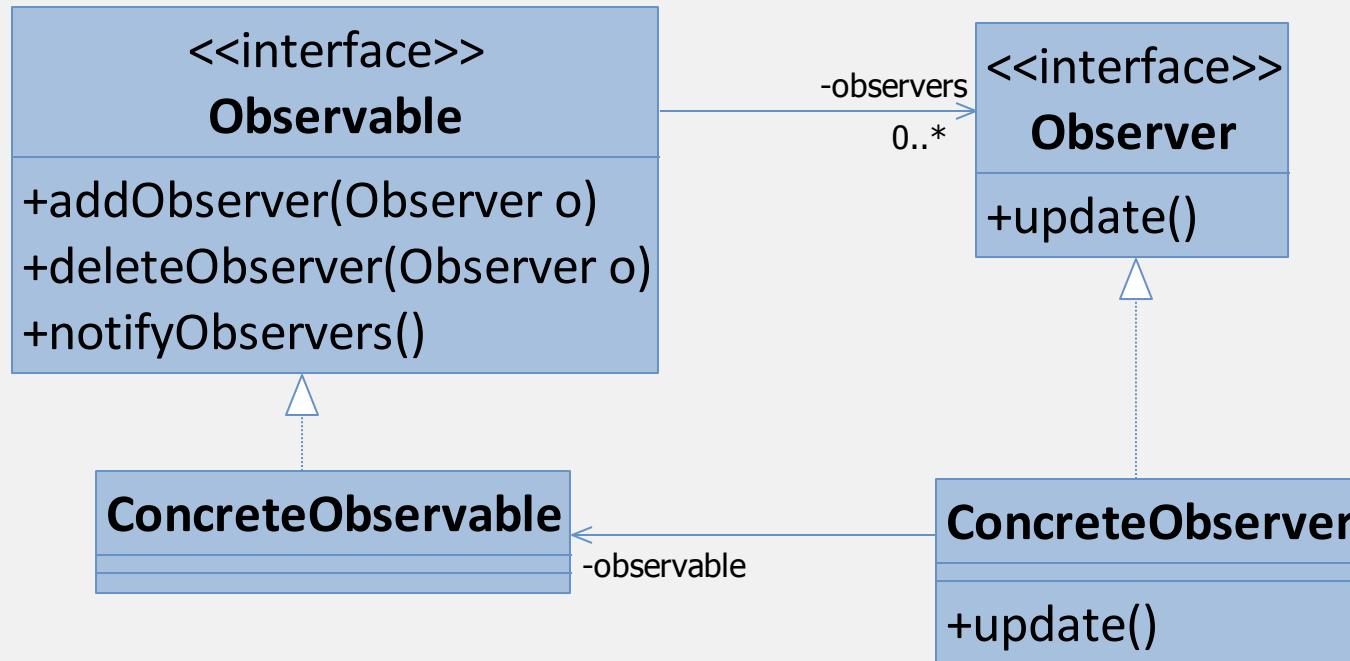
```
public interface Observer {  
    public void Update();  
}  
  
public class DoorBell : Observer {  
    public void Update() {  
        Console.WriteLine("DoorBell: Ring");  
    }  
}  
  
public class LightBulb : Observer {  
    ...  
}
```

Kod przykładu

```
public class ObserverPatternExample
{
    public static void Main(String[] args) {
        LightBulb lightBulb = new LightBulb();
        DoorBell doorBell = new DoorBell();
        ToggleSwitch toggleSwitch = new ToggleSwitch();
        WallSwitch mainSwitch = new WallSwitch();

        mainSwitch.AddObserver(lightBulb);
        mainSwitch.AddObserver(doorBell);
        toggleSwitch.AddObserver(lightBulb);

        mainSwitch.Push();
        toggleSwitch.Push();
    }
}
```



update()

update()

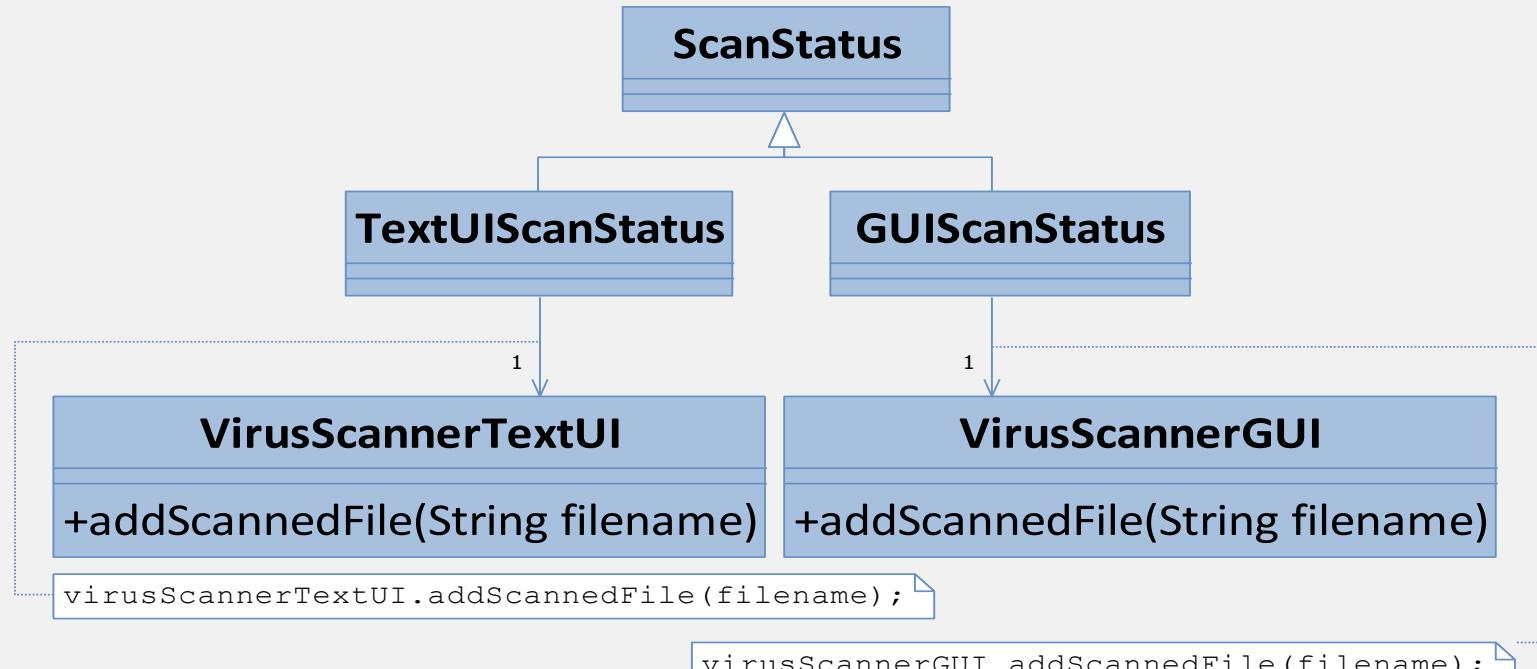
Wzorzec *Observer* umożliwia powiadamianie grupy obiektów o zmianie stanu obiektu obserwowanego.

- # Wzorzec *Observer* zapewnia luźne powiązanie pomiędzy obiektami.
- # Umożliwia niezależne wymienianie obiektów obserwowanych i obserwatorów.
- # Umożliwia dodawanie obserwatorów bez konieczności modyfikowania kodu obiektu obserwowanego.
- # Powiadomienie wysyłane przez obserwowanego nie musi specyfikować odbiorcy.
- # Prosta operacja może wywołać kaskadę kosztownych aktualnień.

Informowanie o wydarzeniach zachodzących w systemie

Informowanie widoku o zmianach w modelu danych

Refaktoryzacja: Replace Hard-Coded Notifications with Observer

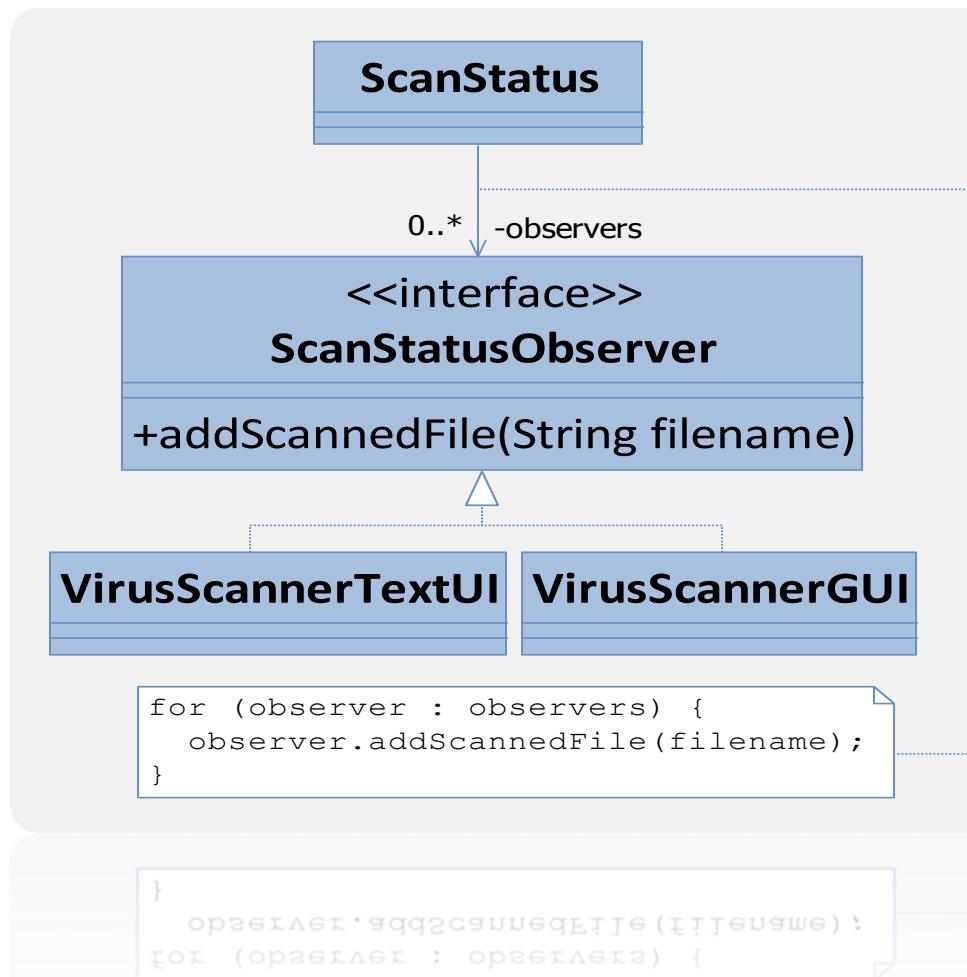


Przykład: Zmiana kodu w aplikacji.

Wystarczy zmienić jedno wejście.

Wystarczy zmienić jedno wejście.

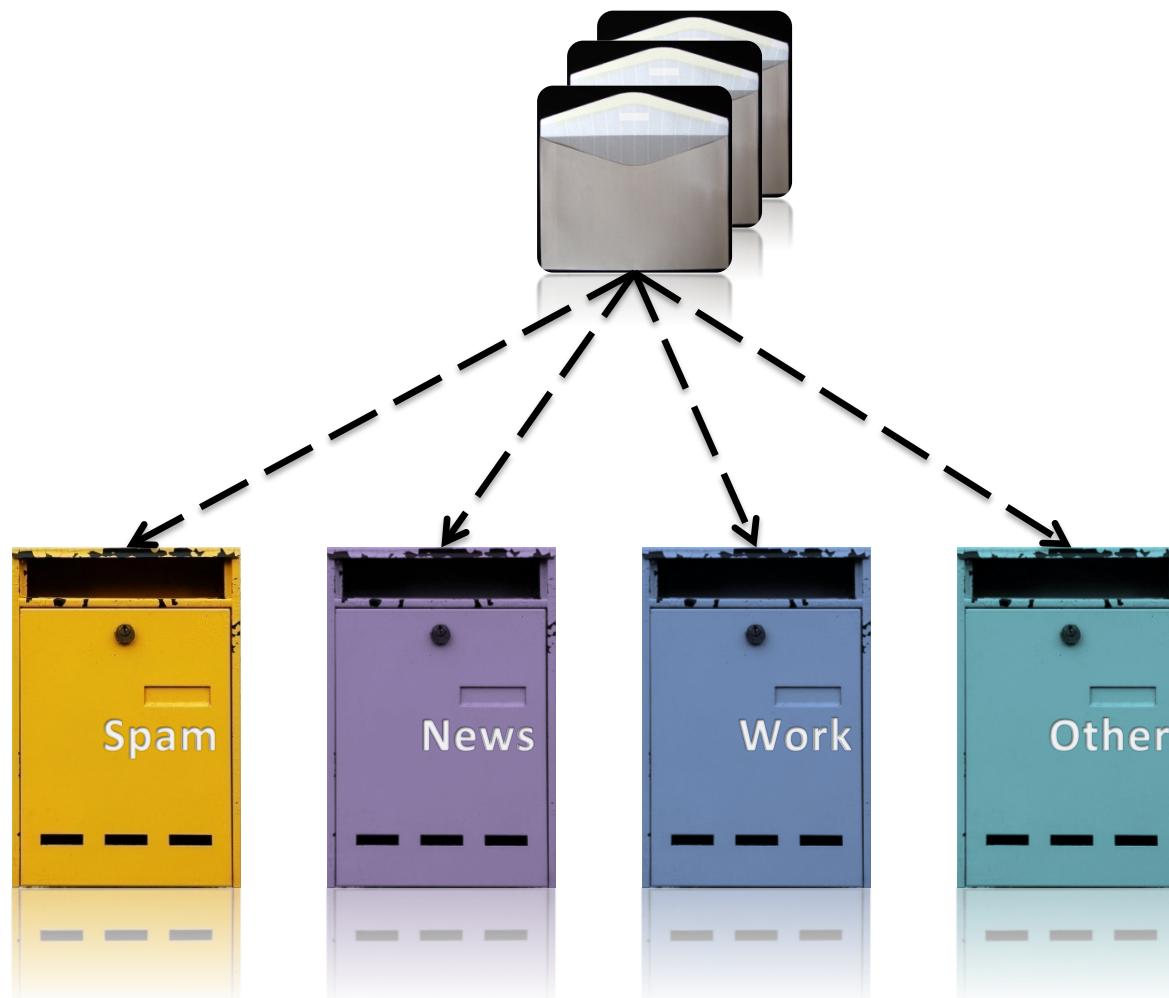
Refaktoryzacja: Replace Hard-Coded Notifications with Observer



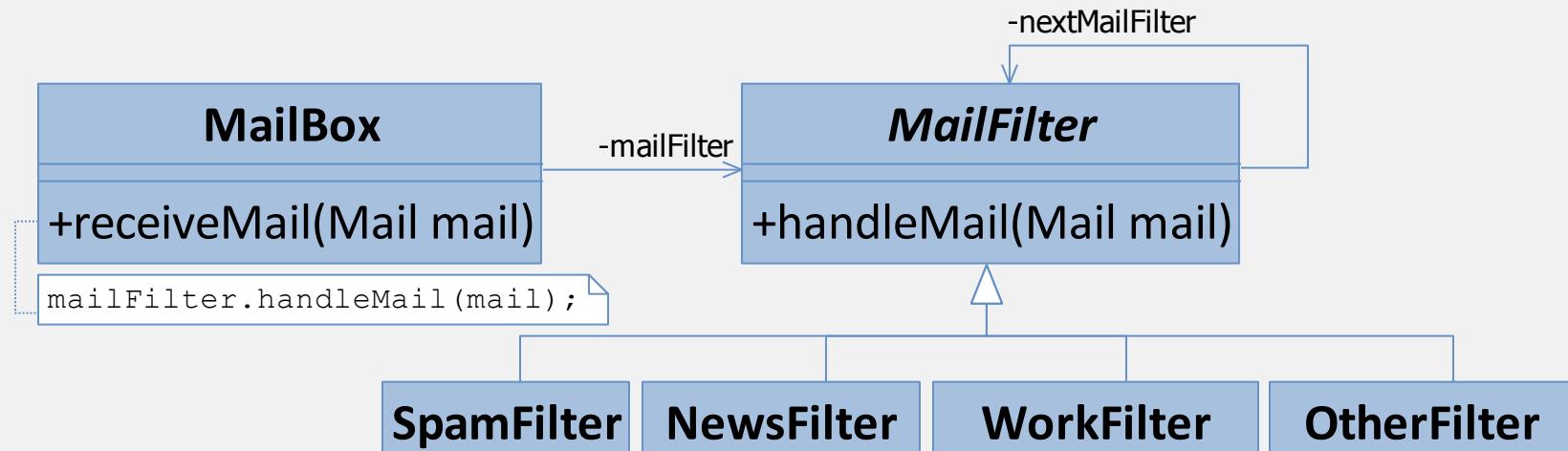
bns it } # } Wzorzec Chain of Responsibility

Wzorce behawioralne

Przykład



bns it} Przykład



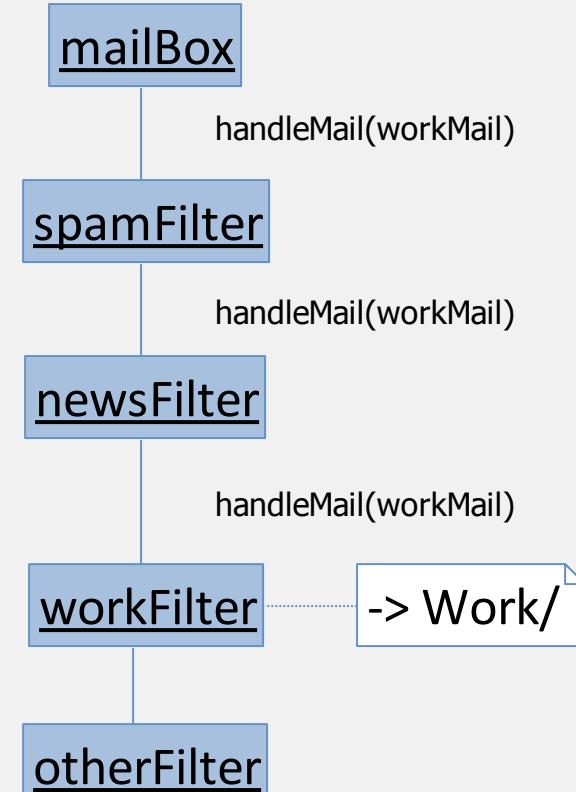
spamFilter
newsFilter
workFilter
otherFilter

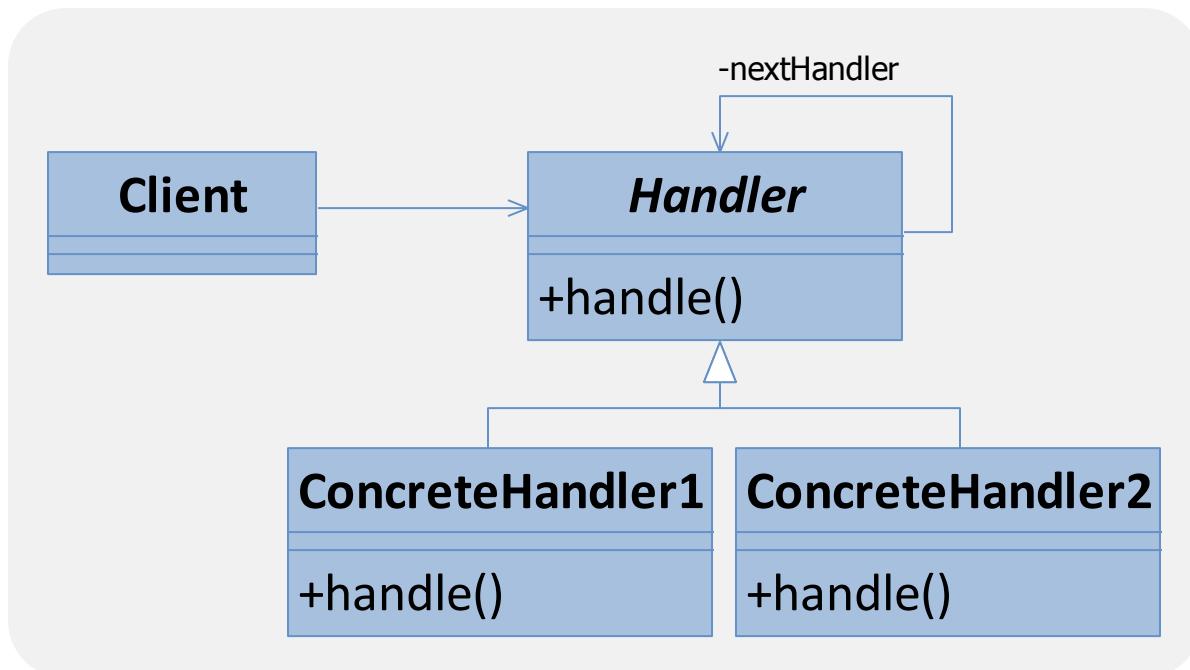
Przykład

Mail ze spamem



Mail dot. pracy





+handle()

+handle()

Wzorzec *Chain of Responsibility* tworzy łańcuch odbiorców i przekazuje wzdłuż niego żądanie, aż jakiś obiekt je obsłuży.

Separuje nadawcę żądania od jego odbiorców i umożliwia dynamiczne określenie odbiorcy żądania.

- # Nadawca i odbiorca żądania nie są ze sobą powiązani.
- # Nadawca żądania nie musi nic wiedzieć o odbiorcy. Wie tylko, że jakiś obiekt je obsłужy.
- # Można dynamicznie dodawać/usuwać obiekty obsługujące żądania.
- # Istnieje możliwość, że żądanie może zostać nieobsłużone, jeżeli łańcuch jest źle skonfigurowany.

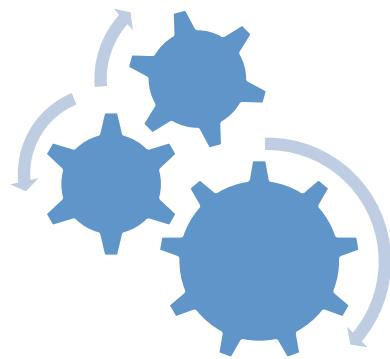
- # Obsługa zdarzeń myszy i klawiatury w systemach okienkowych.
- # Filtrowanie danych w konfigurowalny sposób.



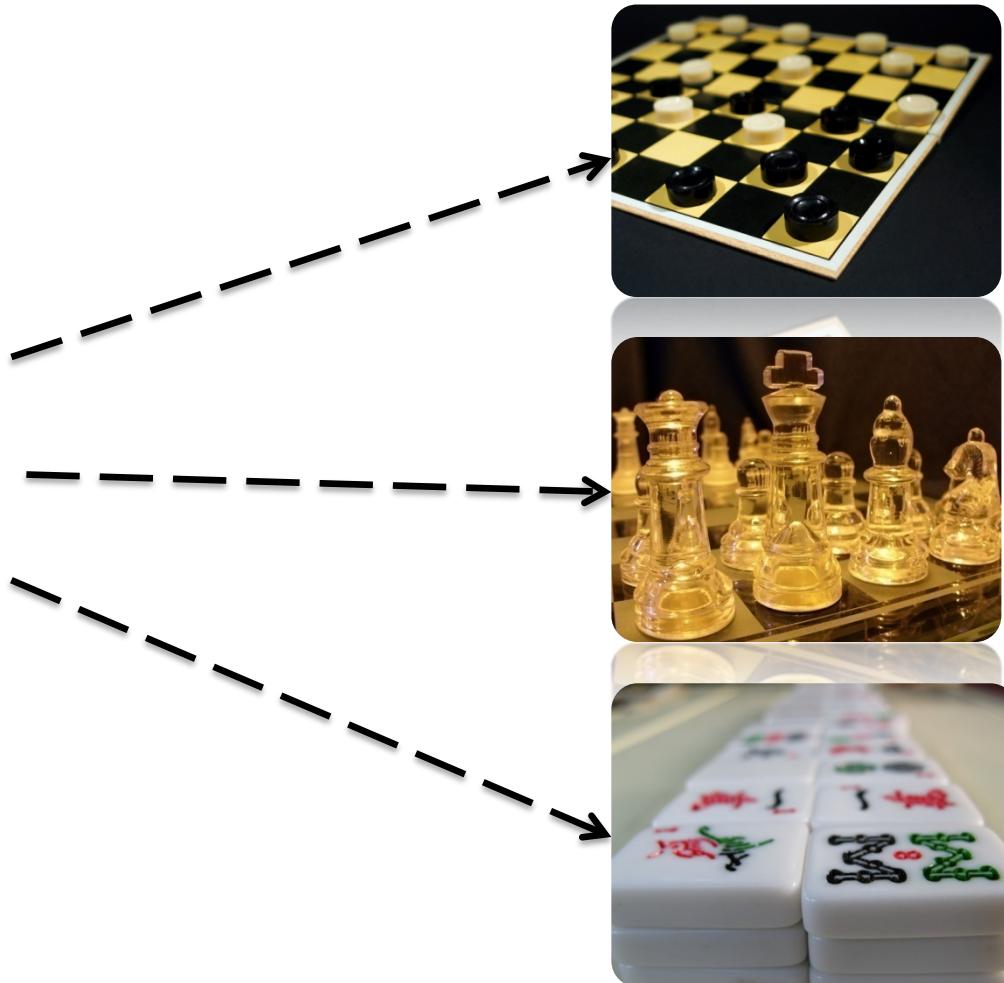
Wzorzec Template Method

Wzorce behawioralne

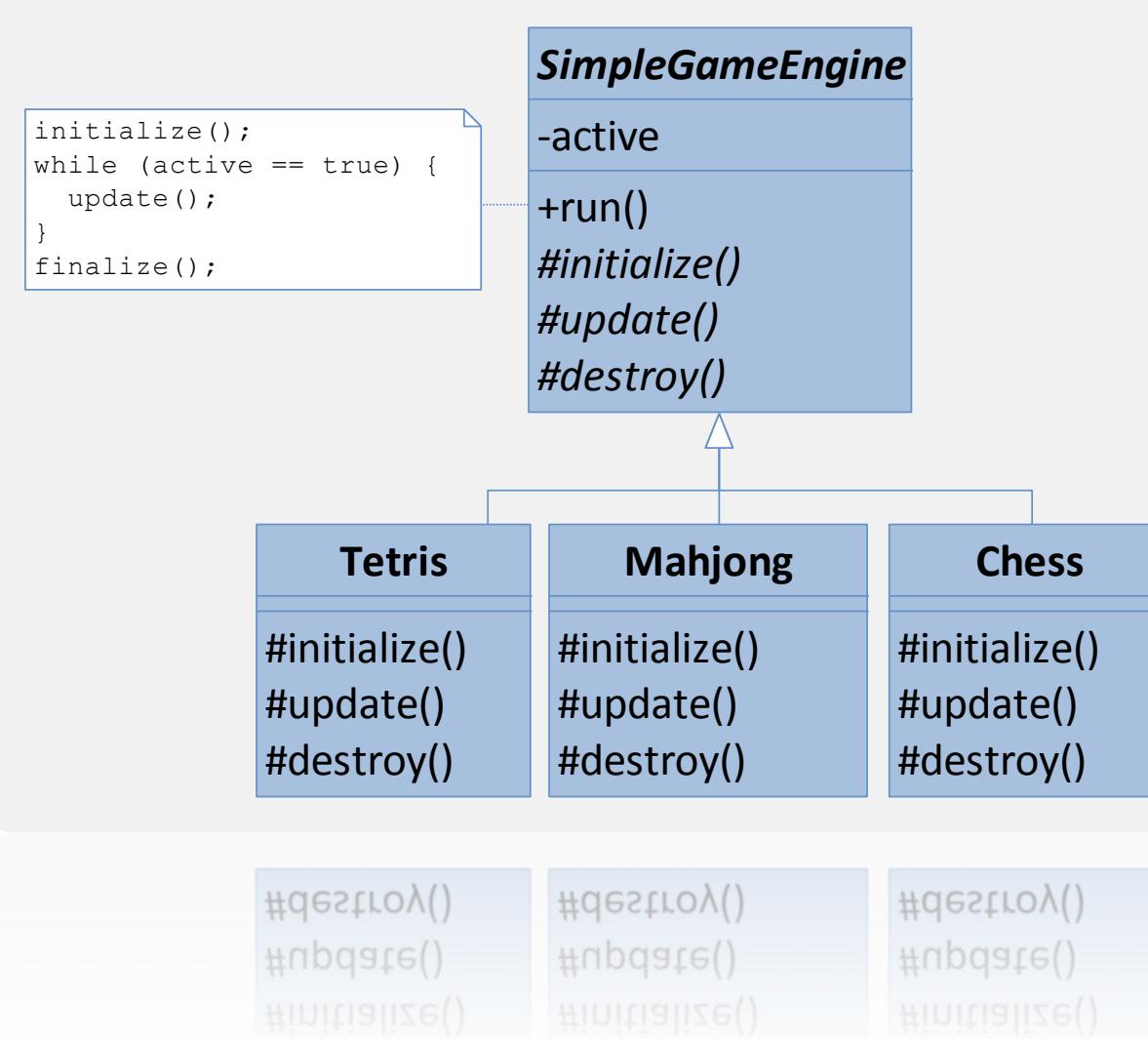
Przykład

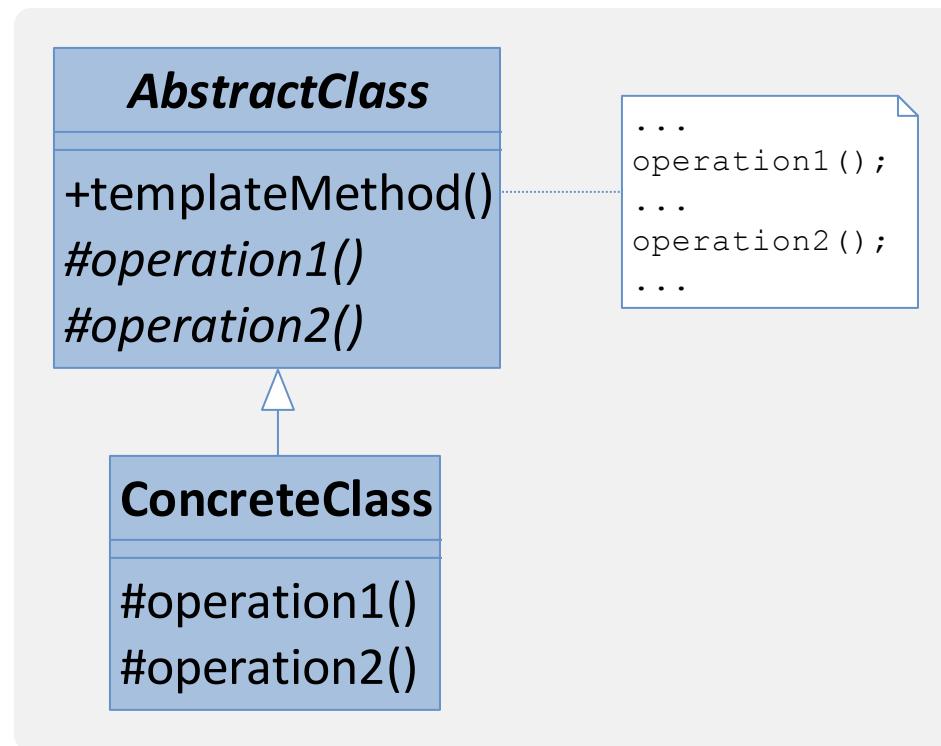


Game Engine



Przykład





#operations()
#operations()

Wzorzec *Template Method* umożliwia podklasom przeddefiniowanie pewnych kroków algorytmu bez zmiany struktury tego algorytmu.

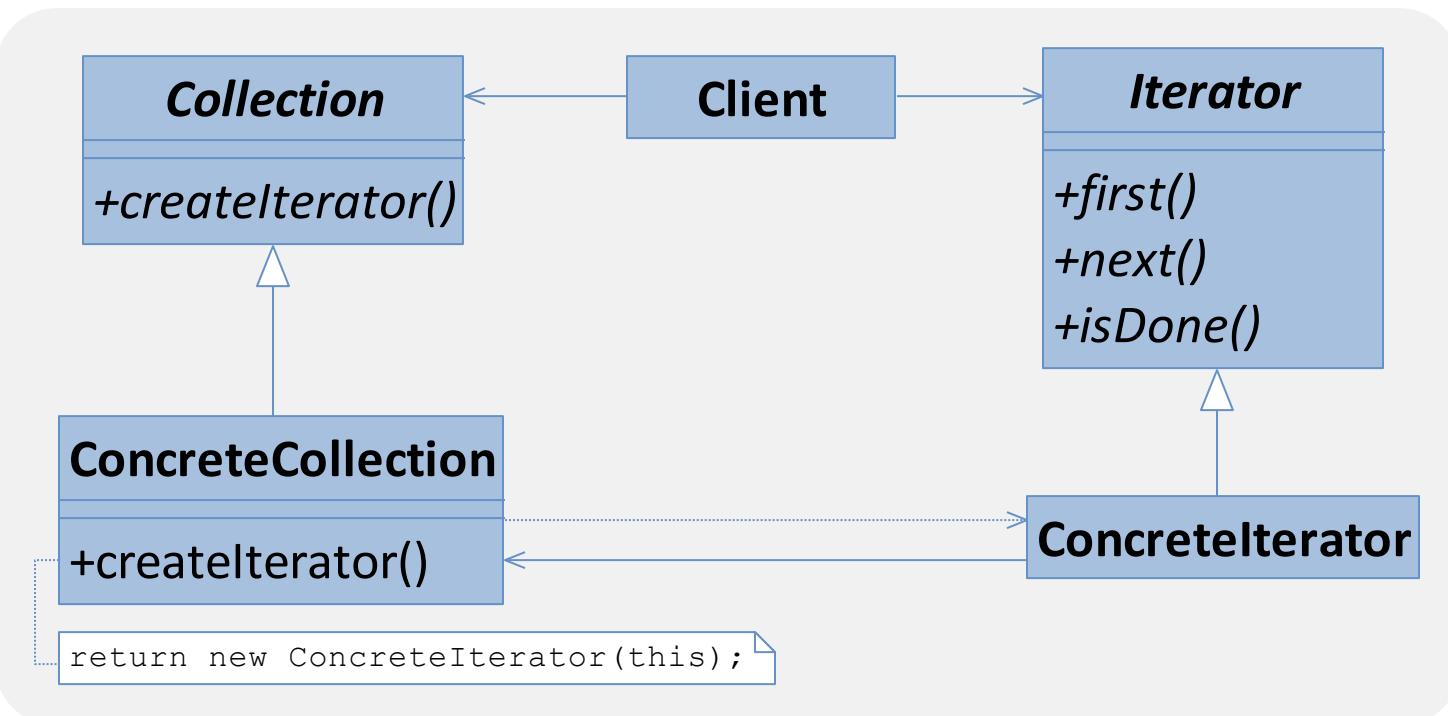
- # W przypadku implementowania wzorca *Template Method* ważne jest określenie operacji, które mogą być przeddefiniowane i tych które muszą być przeddefiniowane.
- # Podklasy mogą rozszerzać działanie operacji z nadklasy poprzez przeciążenie jej i jawne jej wywołanie.
- # Nadklasa definiuje wymagane kroki algorytmu i ich kolejność, natomiast podklasy określają, czy chcą te kroki zaimplementować czy też rozszerzyć ich domyślne wersje.

- # Gotowe szablony do tworzenia dedykowanych rozwiązań.
- # Rodziny algorytmów o podobnym ogólnym schemacie działania.
- # Gotowy szablon algorytmu, który może występować w różnych odmianach.
- # Zastosowanie *Template Method* do obsługi zapytań bazy danych.

bns it} Wrzoeć Iterator

Wzorce behawioralne

Wzorzec *Iterator* zapewnia sekwencyjny dostęp do kolekcji obiektów, bez ujawniania jej wewnętrznej reprezentacji.



return new ConcreteIterator(this);

+createIterator()

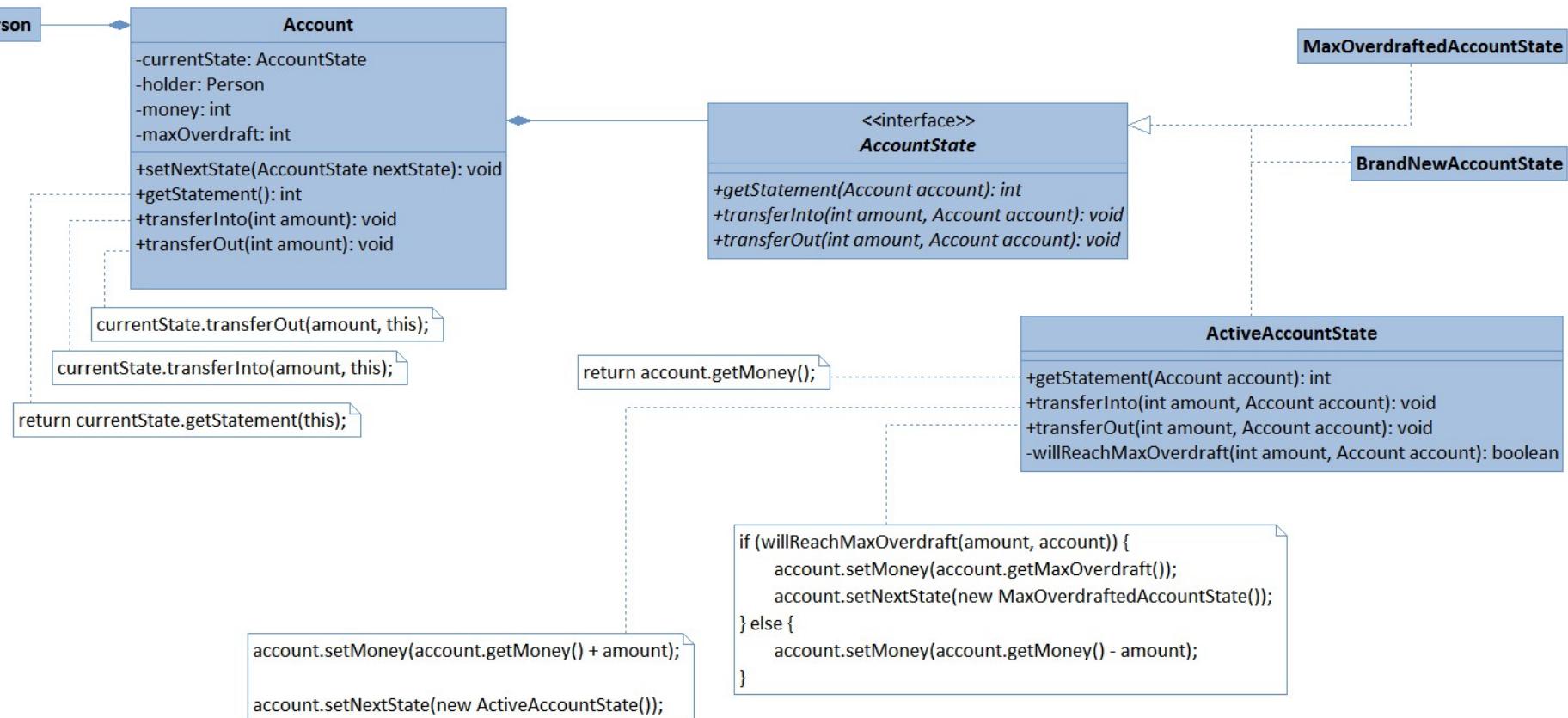
ConcreteCollection

- # Wzorzec *Iterator* umożliwia zaimplementowanie różnych sposobów przehodzenia skomplikowanych kolekcji czy drzew obiektów.
- # Wyodrębnienie obiektu *Iterator* upraszcza interfejs klasy *Collection*.
- # Jednocześnie kilka obiektów typu *ConcretIterator* może przehodzić jedną kolekcję.

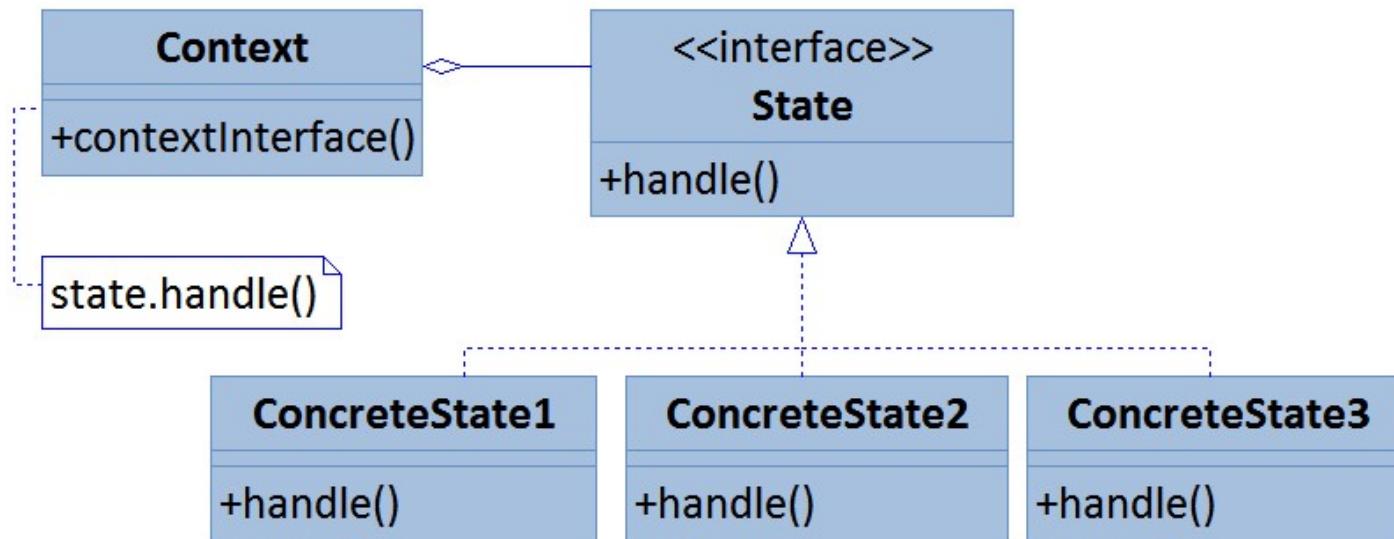
bns it  Wzorzec State
Wzorce behawioralne

Wzorzec *State* umożliwia obiektowi zmianę jego zachowania, jeżeli zmieni się jego stan wewnętrzny.

bns it} Przykład



Struktura

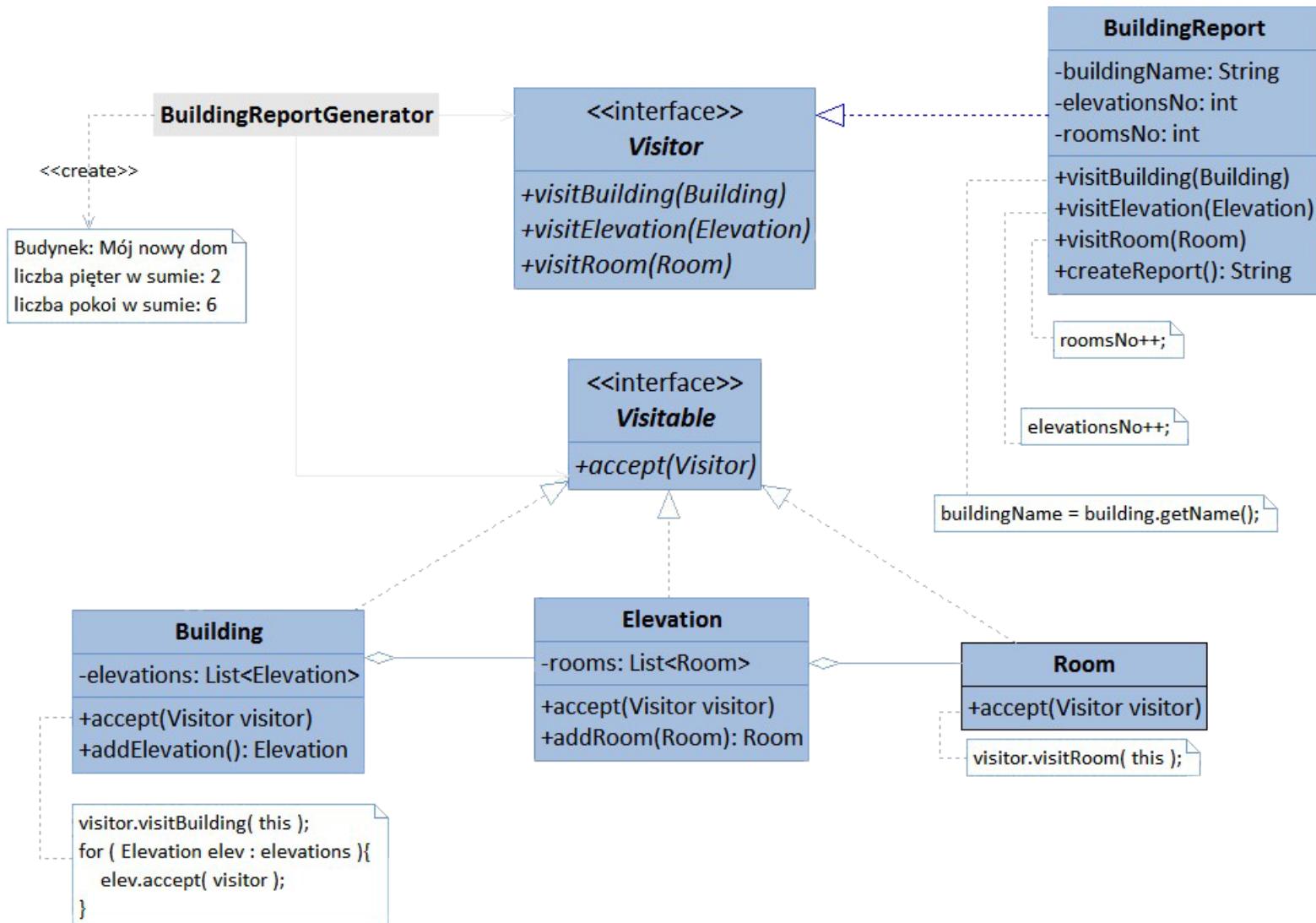


- # Wzorzec *State* zapewnia łatwość kontroli nad aktualnym stanem obiektu.
- # Pozwala uniknąć wielu nieczytelnych instrukcji warunkowych, które sprawdzały wartości zmiennych.
- # Specyficzne zachowania są podklasami klasy *State*, więc można je łatwo dodawać i edytować.
- # Jawność przejść pomiędzy stanami, przejścia pomiędzy stanami są atomowe.
- # Możliwość współdzielenia obiektów reprezentujących stan.

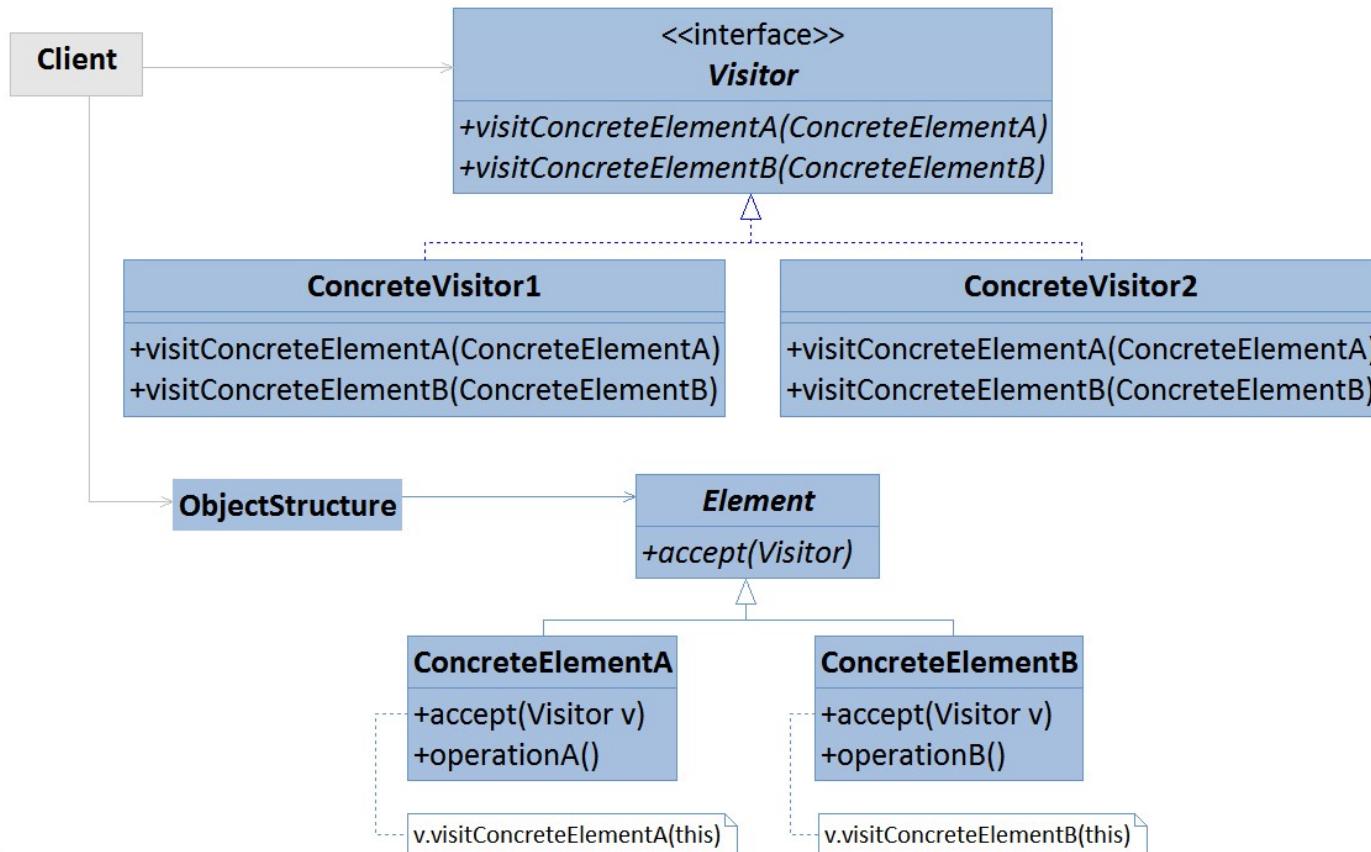
bns it} #} Wzorzec Visitor
Wzorce behawioralne

Wzorzec *Visitor* określa operację, która ma być wykonana na elementach struktury obiektowej. Umożliwia definiowanie nowej operacji bez modyfikowania elementów, na których ona działa.

Przykład



Struktura

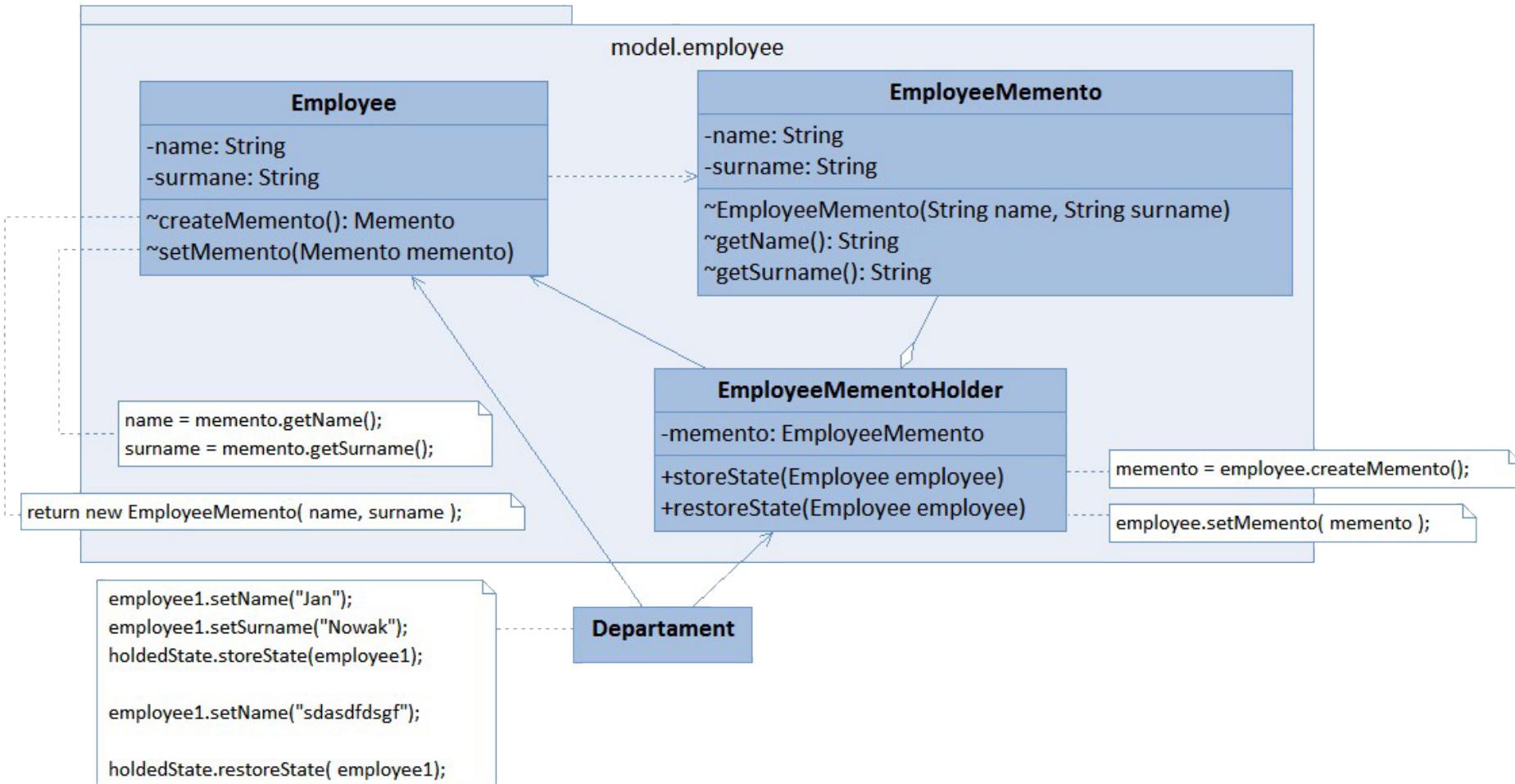


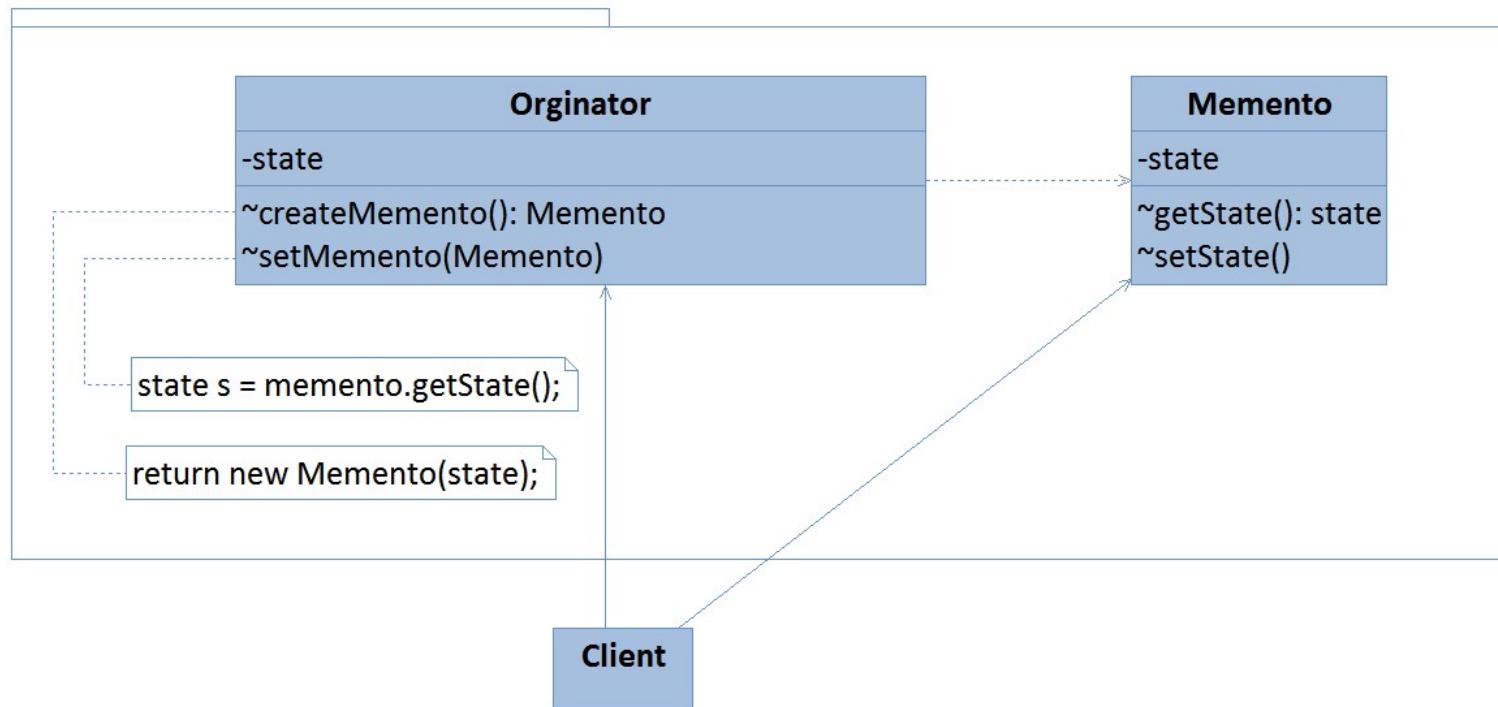
- # Wzorzec *Visitor* zapewnia łatwość dodawania nowych operacji.
- # Grupowanie powiązanych ze sobą operacji w podklasach odwiedzających.
- # Odwiedzanie całej hierarchii klas, bez względu na rodzaje powiązań obiektów.
- # Kumulowanie stanu w konkretnych wizytatorach.
- # Prawdopodobieństwo naruszenia enkapsulacji.

bns it } # } Wzorzec Memento
Wzorce behawioralne

Wzorzec *Memento*, bez naruszania enkapsulacji, zapamiętuje stan wewnętrzny obiektu, dzięki czemu może on zostać później przywrócony.

Przykład





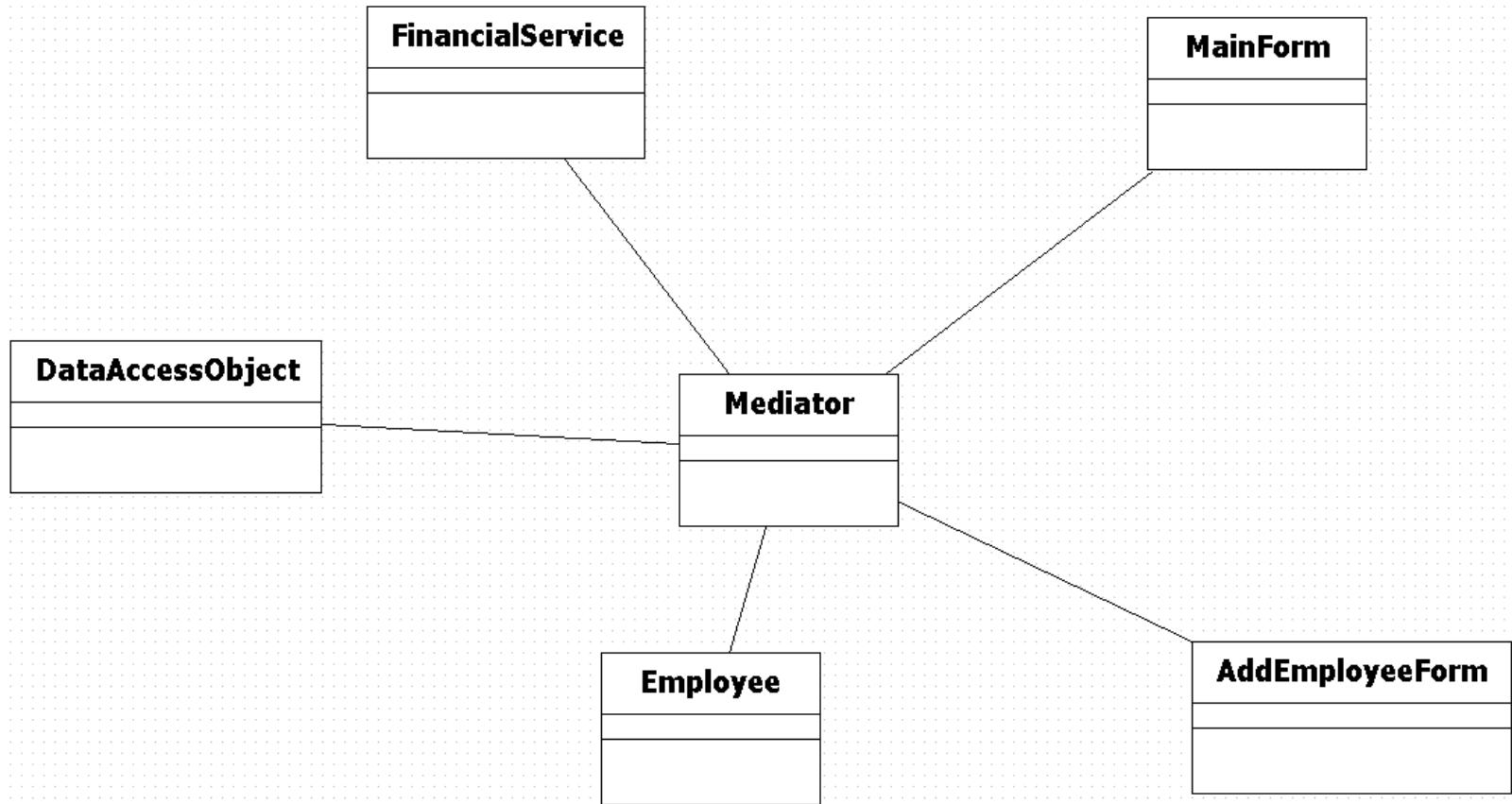
- # Wzorzec *Memento* zapewnia pewność nienaruszalności enkapsulacji, podczas zapamiętywania stanu obiektu.
- # Trzeba zagwarantować, że tylko wybrani klienci mają dostęp do zapisanego stanu, co może okazać się trudne.
- # Tworzenie i obsługa zapisanych stanów może być kosztowne .

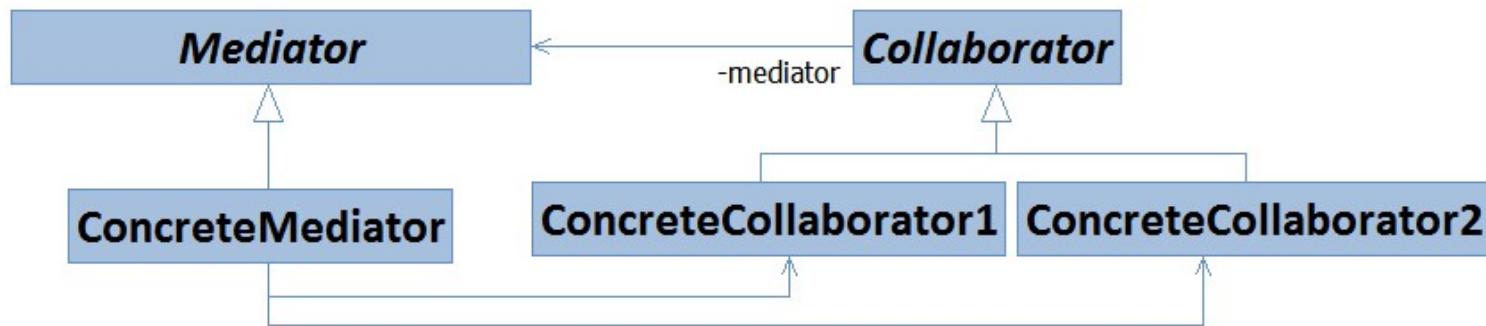
bns it} Wzorzec Mediator
Wzorce behawioralne

Wzorzec *Mediator* separuje obiekty od systemu.

Tworzy obiekt enkapsulujący informacje o współdziałaniu obiektów w strukturze charakteryzującej się złożonością procedur komunikacji i sterowania.

Przykład



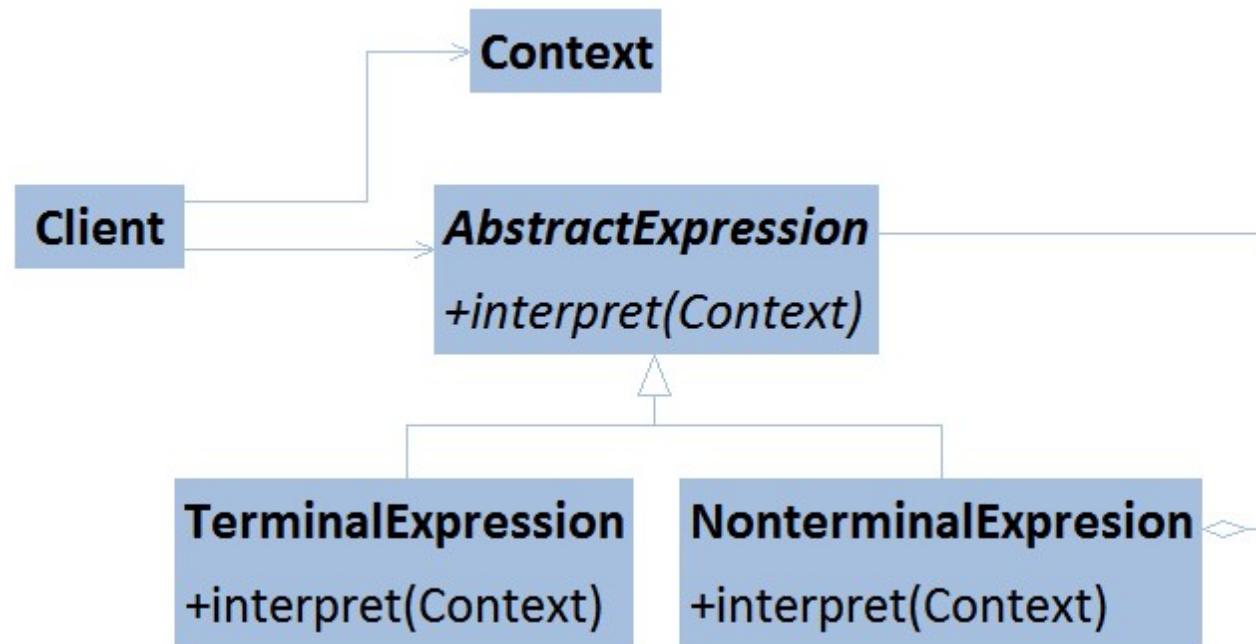


- # Wzorzec *Mediator* zwiększa szanse ponownego wykorzystania obiektów, z którym współpracuje, bo zapewnia ich separację od systemu.
- # Upraszcza system zamieniając związki wiele-do-wiele na jeden-do-wiele.
- # Skupia w jednym miejscu całą logikę sterowania.
- # Niedopracowany projekt może uczynić mediatora bardzo skomplikowanym i trudnym w utrzymaniu.
- # Implementacja jest mocno związana z konkretnym systemem, co utrudnia jej ponowne użycie.

bns it } # } Wzorzec Interpreter

Wzorce behawioralne

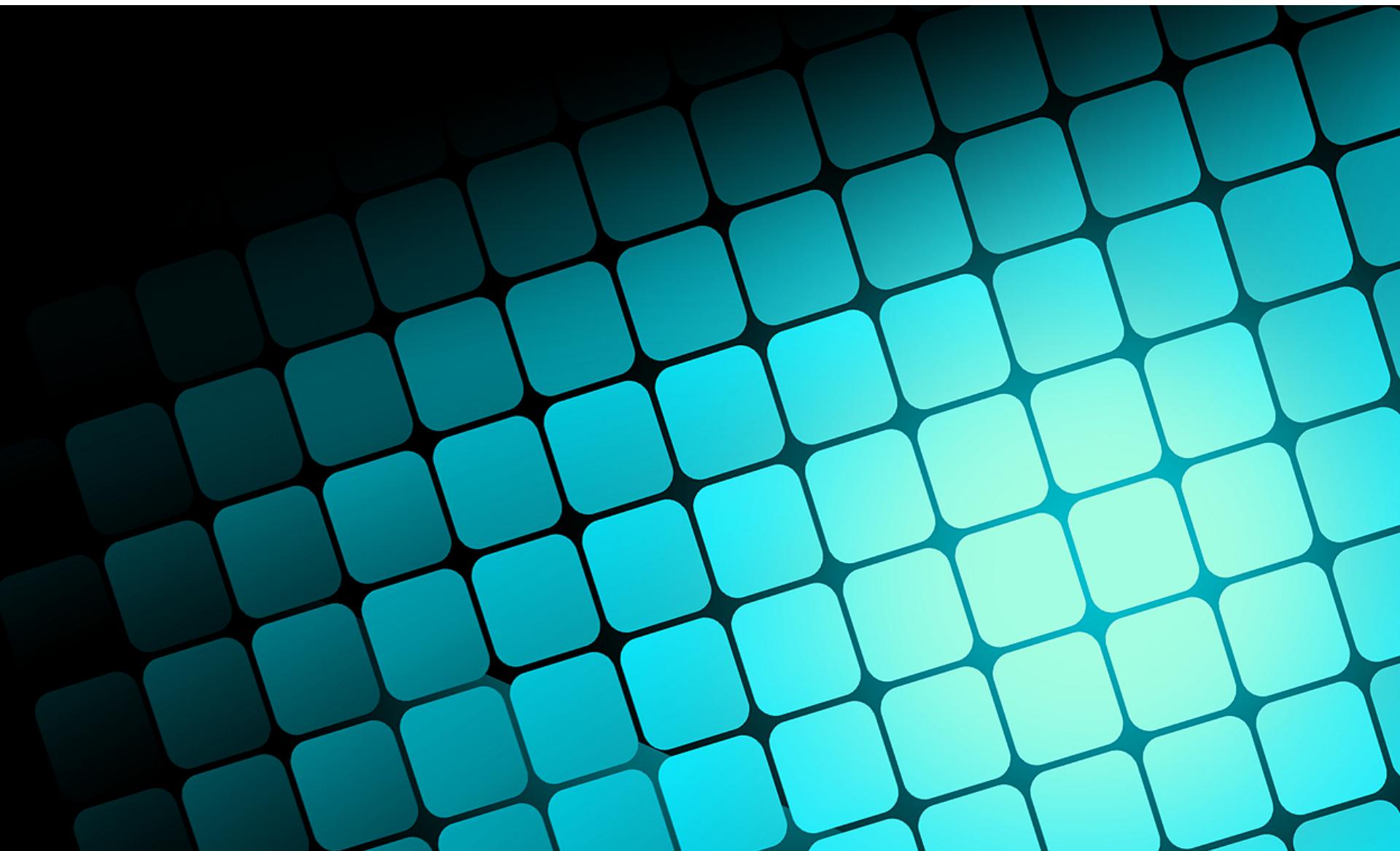
Wzorzec *Interpreter* definiuje opis gramatyki języka i umożliwia jej interpretacje.



- # Wzorzec *Interpreter* zapewnia łatwość zmian i rozszerzenia opisu gramatyki.
- # Łatwość Interpretacji gramatyki dzięki strukturze ich klas.
- # Dodawanie nowych interpretacji również jest proste.
- # Skomplikowana gramatyka skutkuje skomplikowanym interpreterem.

Wzorce strukturalne GoF

Wzorce projektowe i refaktoryzacja do wzorców



Opisuję sposoby łączenia klas i obiektów w większe struktury.

bns it } # Wzorzec Adapter
Wzorce strukturalne

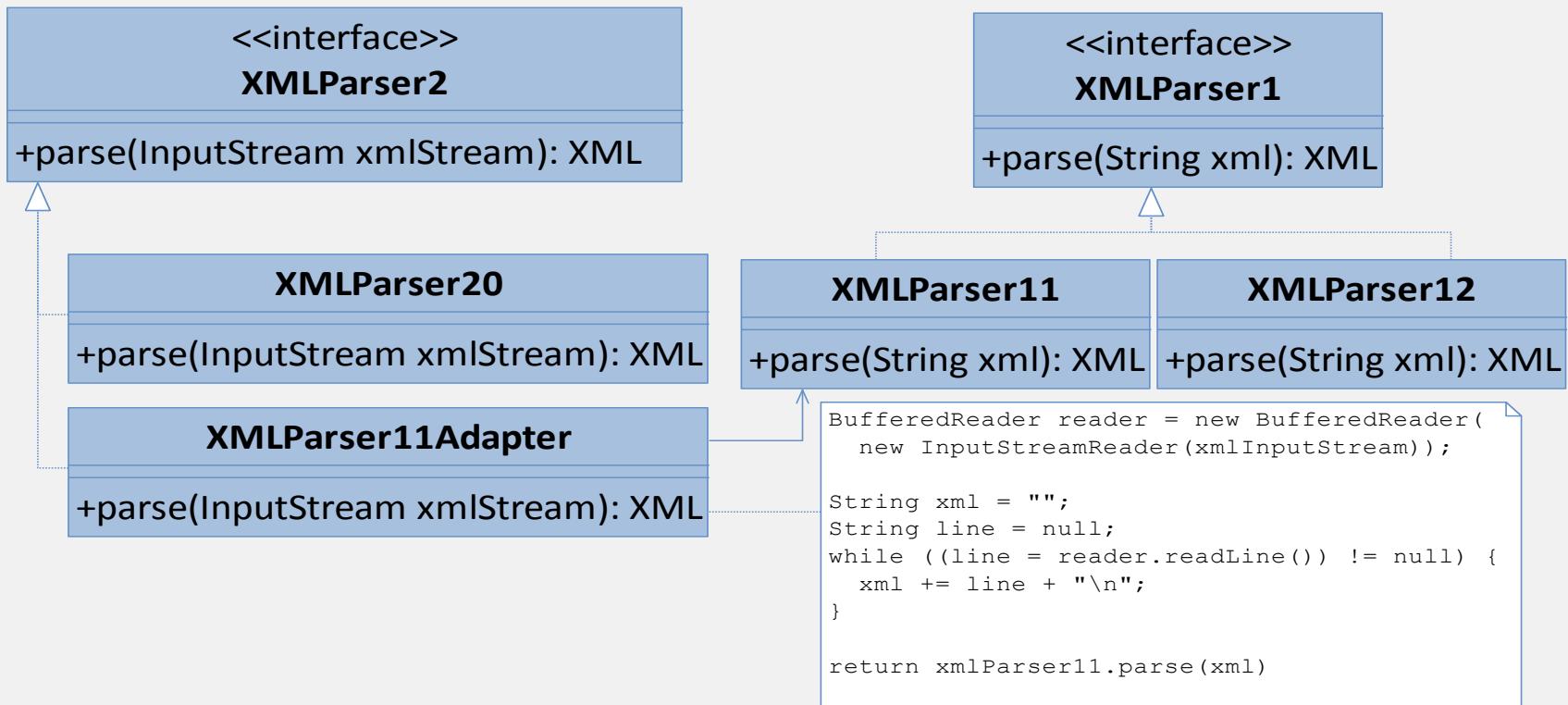
Przykład

XMLParser
Ver. 2.0

XMLParser1Adapter

XMLParser
Ver. 1.0

Przykład



```
return xmlParser11.parse(xml)
}

xml += line + "\n";
while ((line = reader.readLine()) != null) {
    xml += line + "\n";
}
```

bns it} Kod przykładu

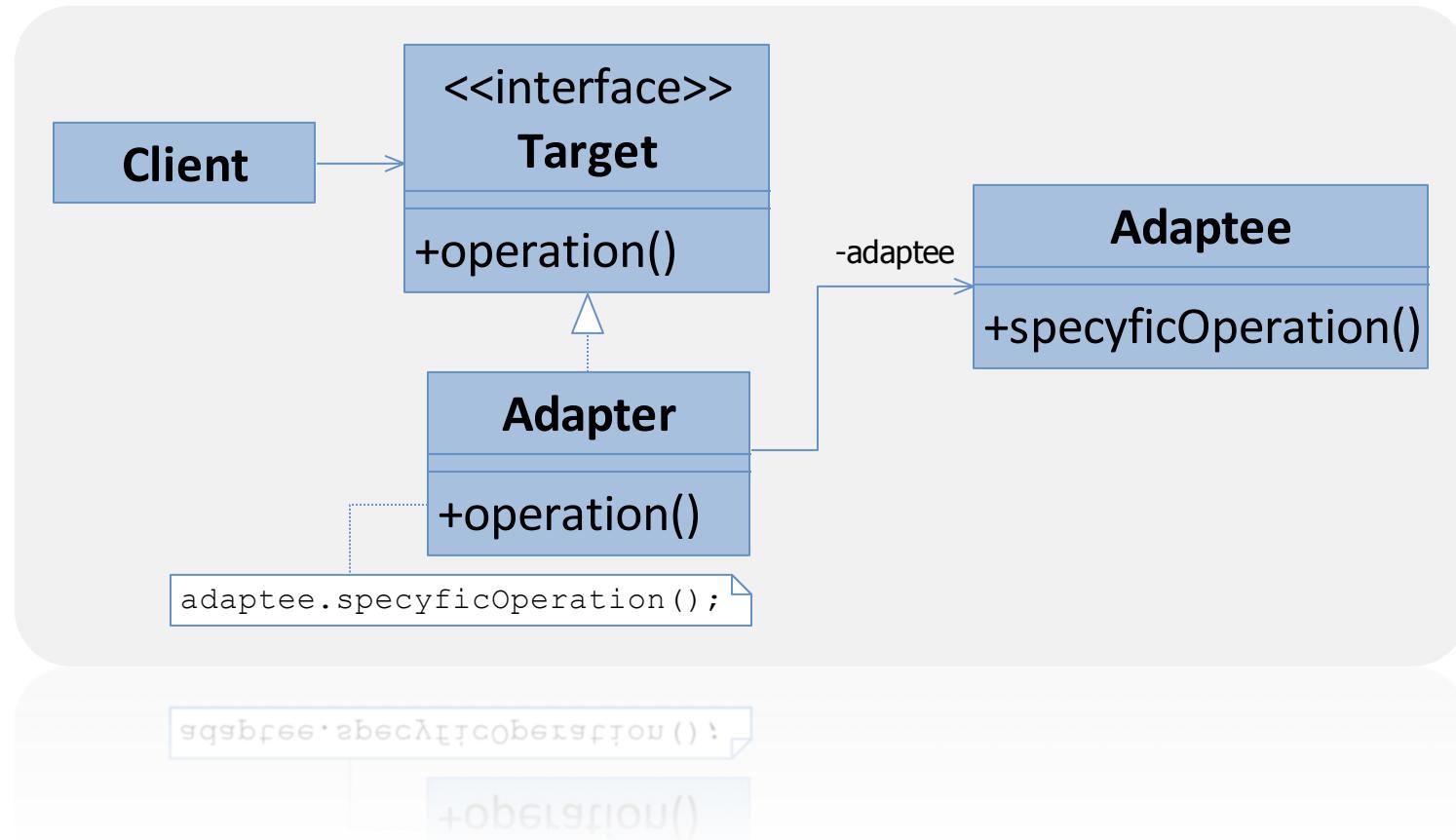
```
public interface XMLParser2
{
    public XML Parse(Stream xmlStream);
}
```

```
public interface XMLParser1 {
    public XML Parse(String xml);
}
```

```
public class XMLParser11 : XMLParser1 {
    public XML Parse(String xml) {
        XML xmlTree = new XML();
        // ...
        return xmlTree;
    }
}
```

```
public class XMLParser11Adapter : XMLParser2 {
    private XMLParser11 xmlParser11 = new XMLParser11();

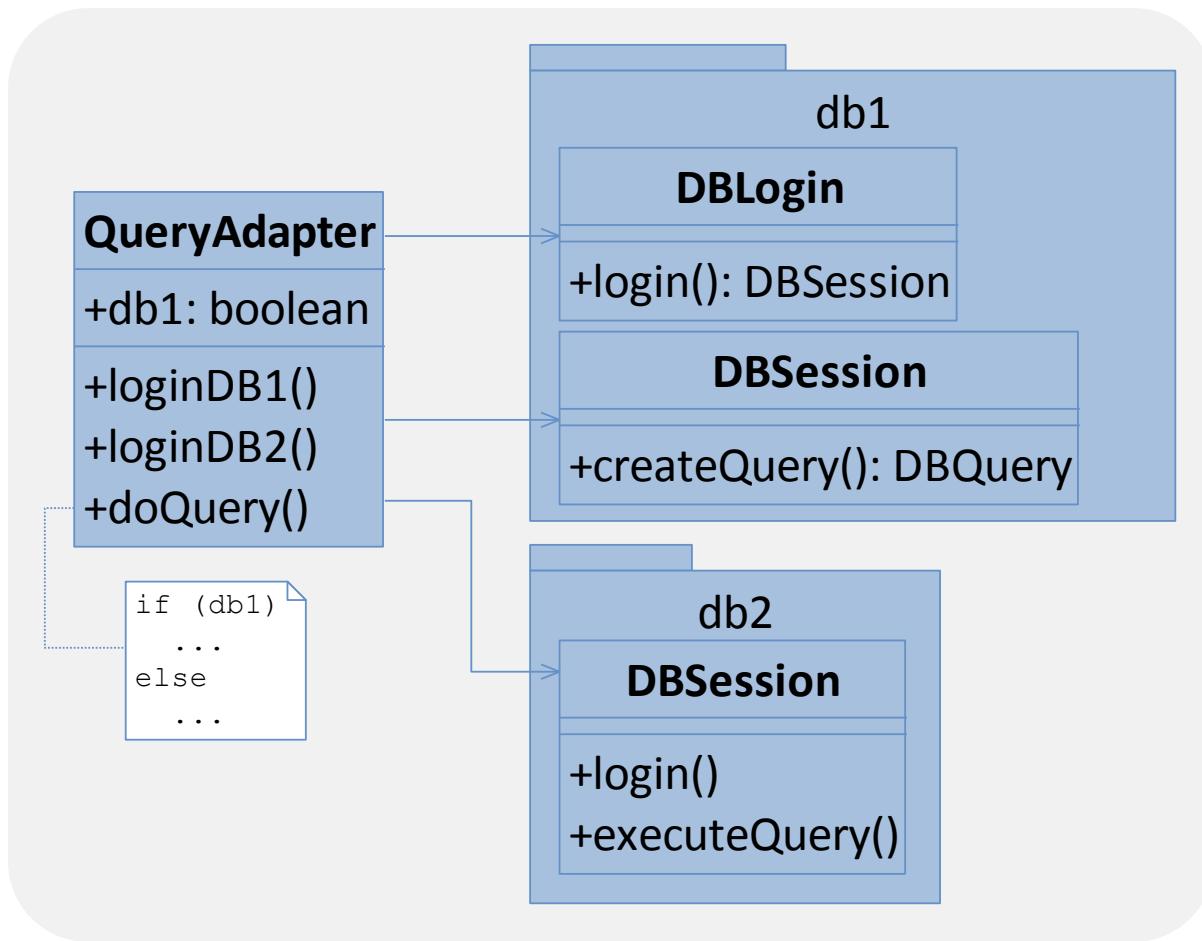
    public XML Parse(Stream xmlStream) {
        String xml = ToXMLString( xmlStream );
        return xmlParser11.Parse(xml);
    }
}
```



Wzorzec *Adapter* przekształca interfejs klasy do postaci, której oczekują klienci.

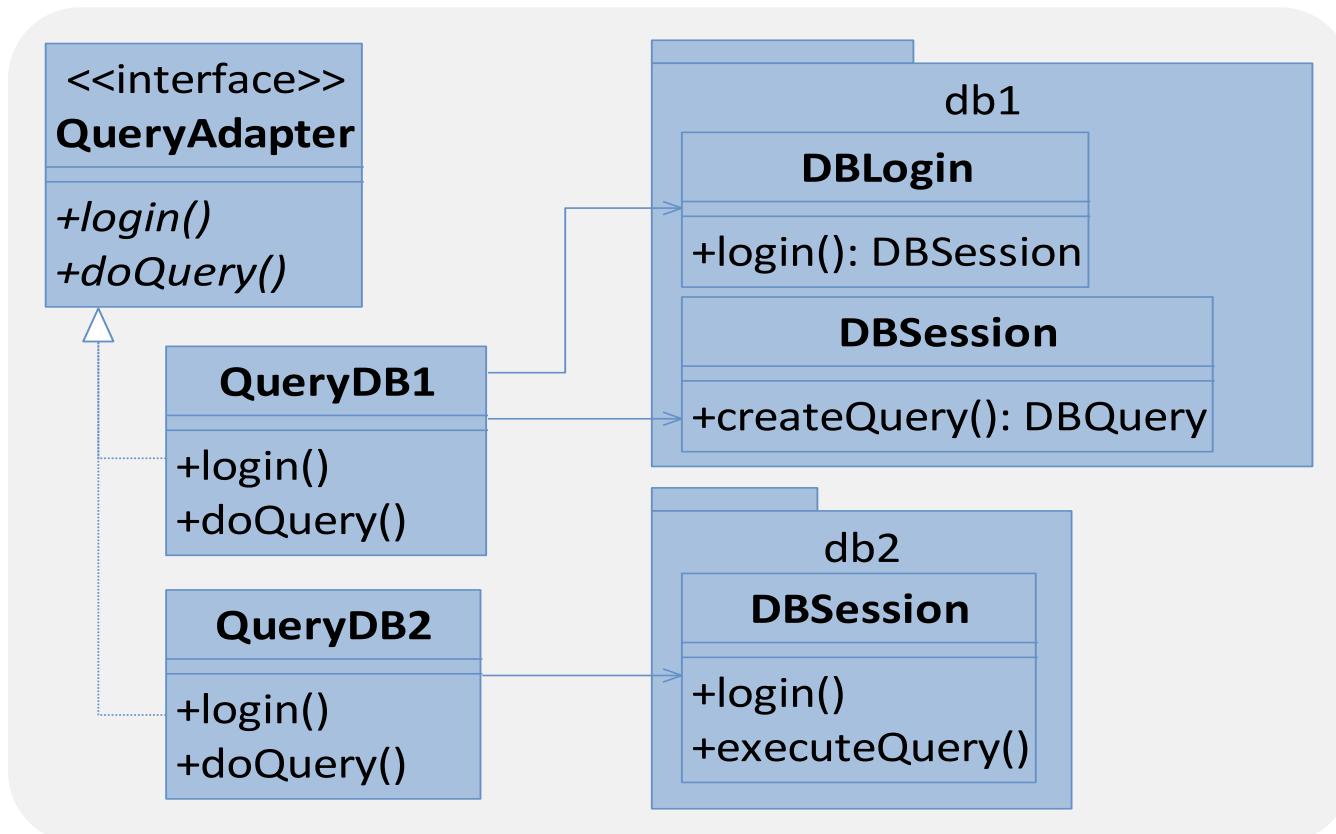
- # *Adapter* może działać również z podklasami obiektu *Adaptee*.
- # Zmiana zachowania obiektu *Adaptee* wymaga tworzenia podklas i odwoływania się obiektu *Adapter* bezpośrednio do nich.
- # *Adapter* musi wykonać dodatkową pracę potrzebną do przystosowania interfejsu. Może to pogorszyć wydajność obliczeniową rozwiązania.

- # Włączanie klasy do systemu, który oczekuje od niej innego interfejsu.
- # Adaptowanie zewnętrznej biblioteki do własnych interfejsów.
- # Biblioteka okienkowa *wxWidgets* – adaptuje biblioteki okienkowe z różnych systemów do wspólnego interfejsu.

Refaktoryzacja: *Extract Adapter*

+executeQuery()
+login()

Refaktoryzacja: *Extract Adapter*



+doQuery()
+login()

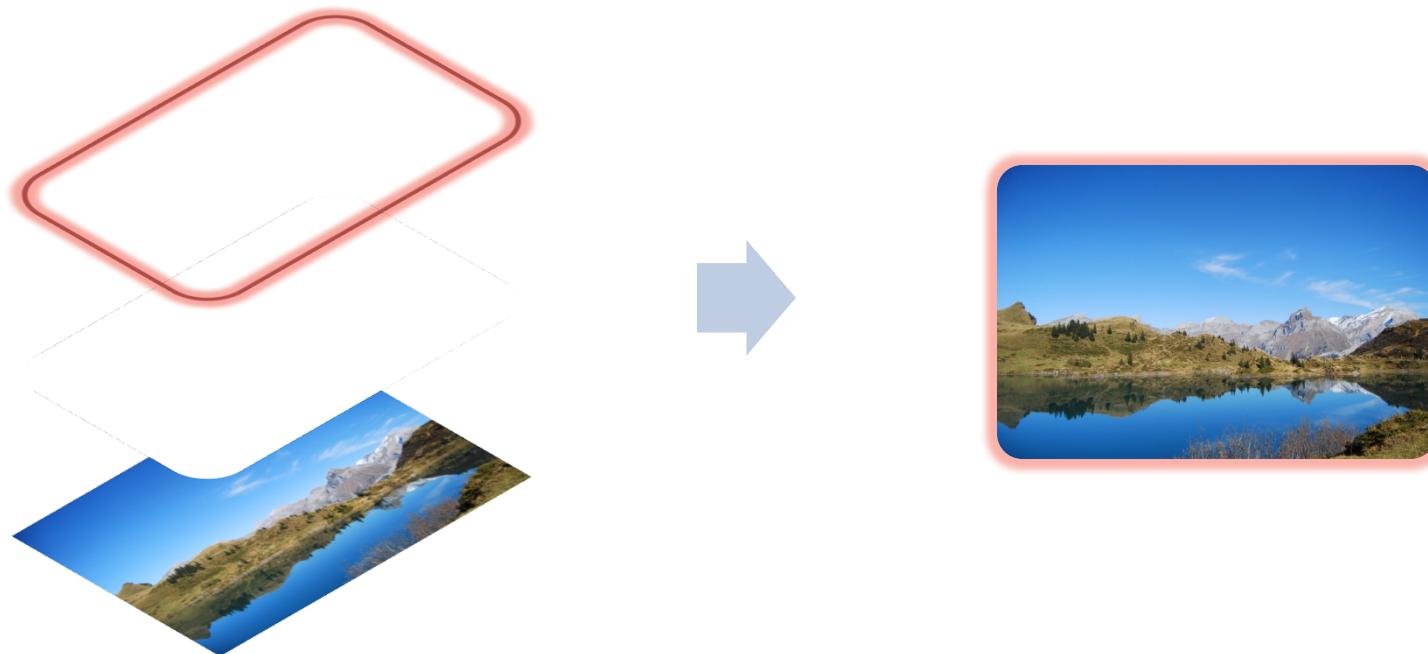
db1

+executeQuery()
+login()

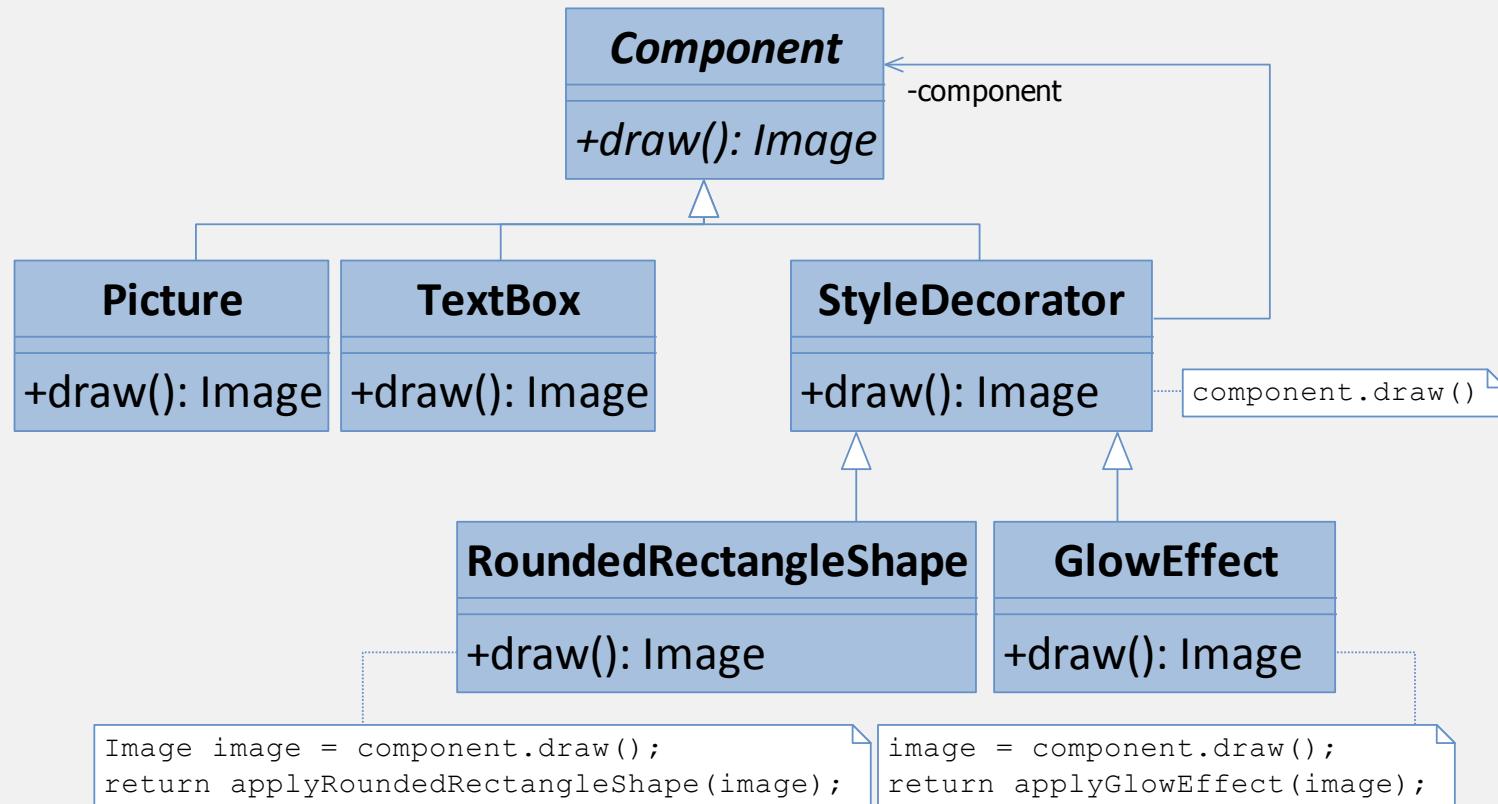
db2

bns it } # Wzorzec Decorator
Wzorce strukturalne

Przykład



Przykład

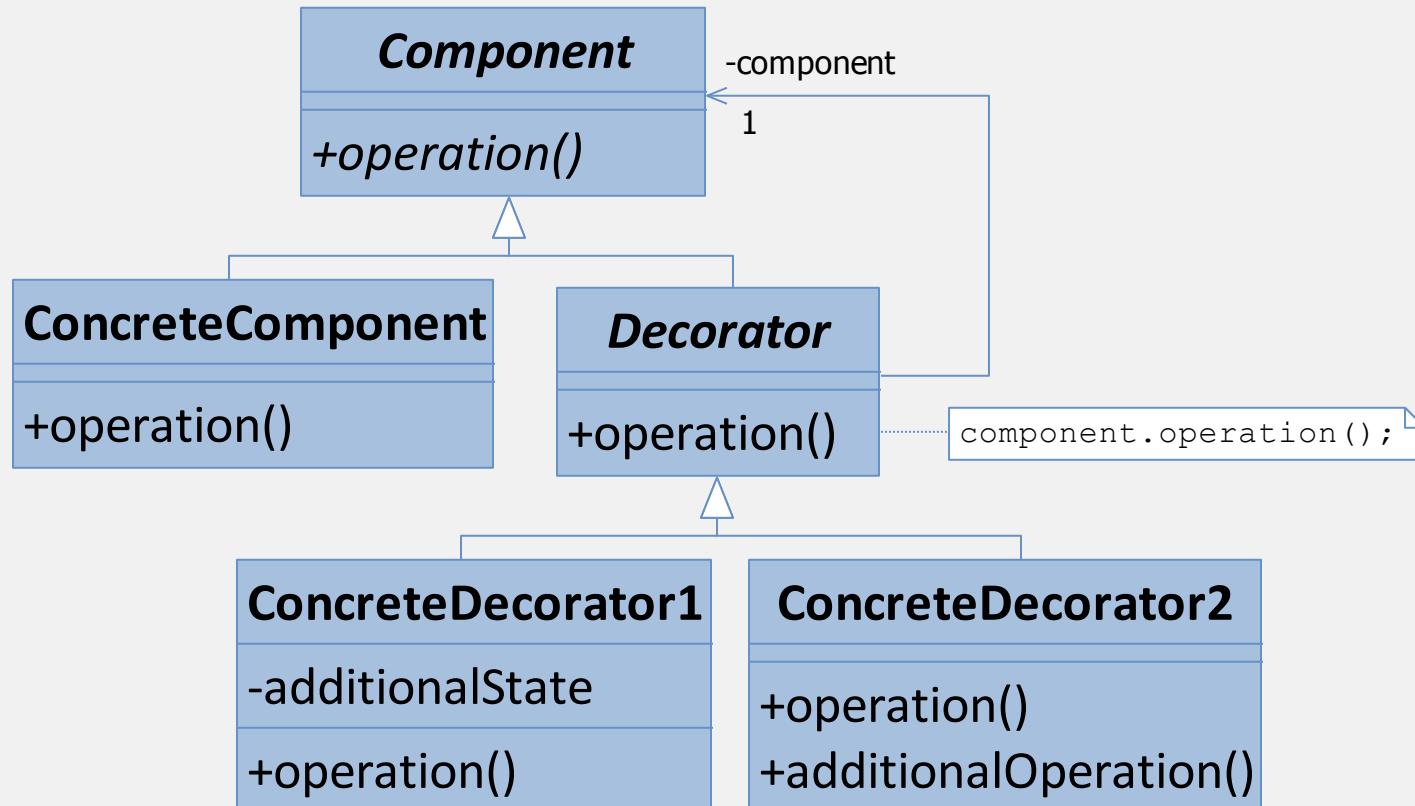


reflexus abbęgłybioruqeqbecfauadżesħaġe (żmäde) :
żmäde = component.draw();

reflexus abbęgłybioruqeqbecf (żmäde) :
żmäde = component.draw();

+draw(): Image

+draw(): Image



Kod przykładu

```
public interface Component {  
    public Image Draw();  
}
```

```
public class Picture : Component {  
    private Image image;  
  
    public Image Draw() {  
        return image;  
    }  
}
```

```
public class DecoratorExample {  
    public static void Main(String[] args) {  
        Component component  
            = new Picture("a/file/path");  
        component  
            = new RoundedRectangleShape(component);  
        component = new GlowEffect(component);  
  
        Image image = component.Draw();  
        //...  
    }  
}
```

```
public class StyleDecorator : Component {  
    protected Component component;  
  
    public StyleDecorator(Component component) {  
        this.component = component;  
    }  
}
```

```
public class RoundedRectangleShape :  
StyleDecorator {  
    public Image Draw() {  
        Image image = component.Draw();  
        return ApplyRoundedRectangleShape(image);  
    }  
}
```

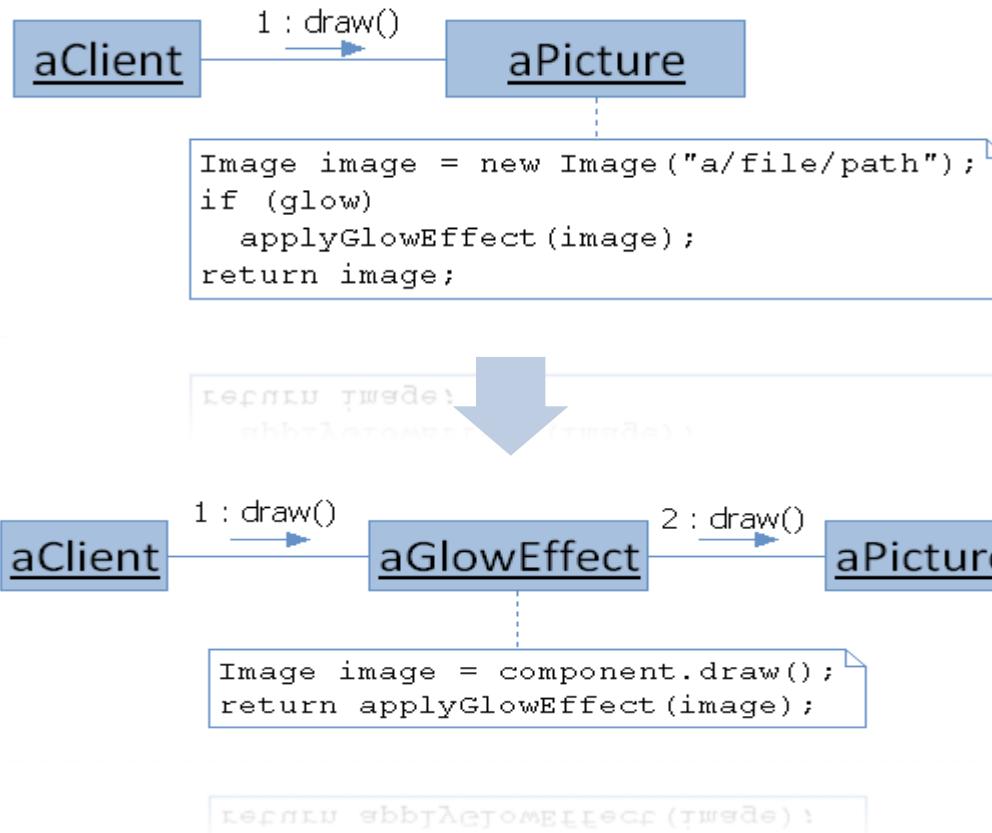
Wzorzec *Decorator* pozwala na dynamiczne rozszerzanie odpowiedzialności obiektu.

- # Wzorzec *Decorator* umożliwia elastyczne rozszerzanie odpowiedzialności obiektu – elastyczny zamiennik dziedziczenia.
- # Umożliwia dynamiczne modyfikowanie odpowiedzialności obiektu w trakcie działania programu.

- # Udekorowany komponent z punktu widzenia identyczności obiektów nie jest taki sam jak komponent bez dekoracji.
- # Systemy korzystające nadmiernie z dekoratorów będą wypełnione wieloma małymi i podobnymi obiektami.
Wprowadzanie zmian i usuwanie błędów może być utrudnione.

- # Dodawanie graficznych upiększaczy do kontrolek
- # Programowanie aspektowe
- # Sprawdzanie w czasie wykonania, typów elementów dodawanych do kolekcji (EmployeeListDecorator, NotNullCollectionDecorator)

Refaktoryzacja: *Move Embellishment To Decorator*

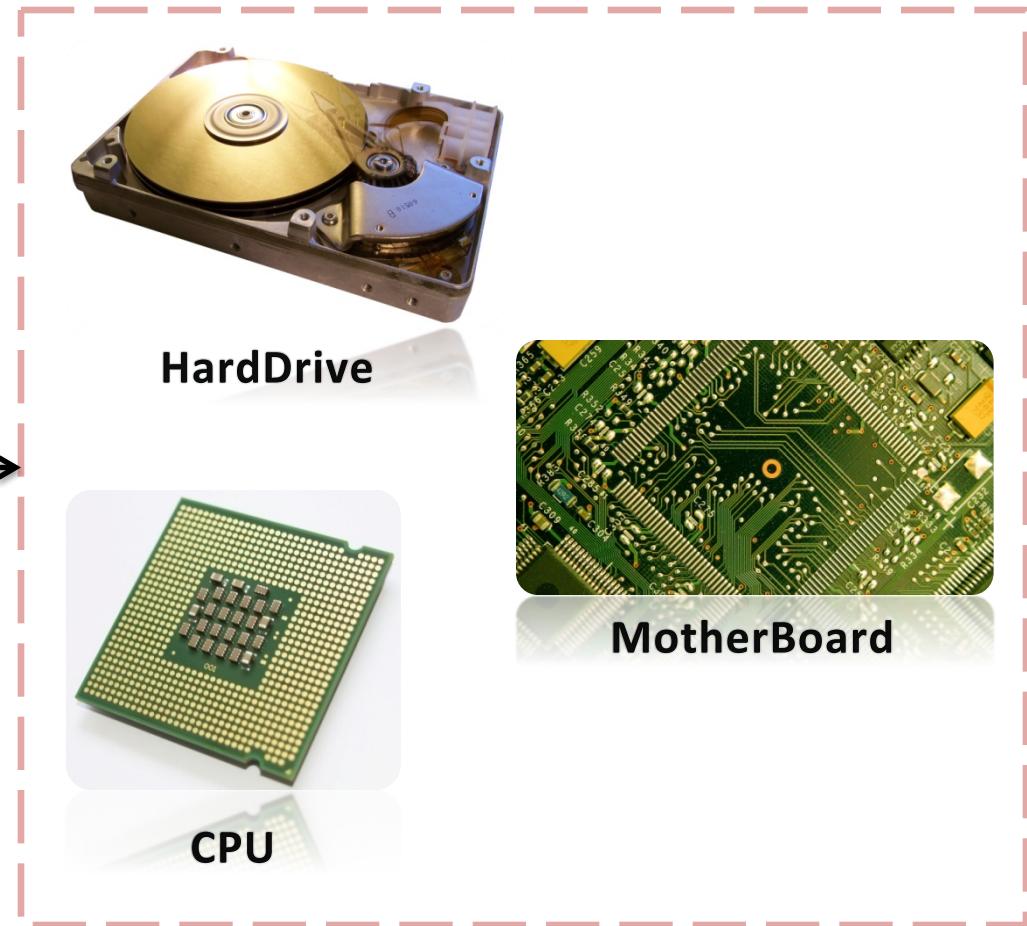


bns it } # } Wzorzec Facade
Wzorce strukturalne

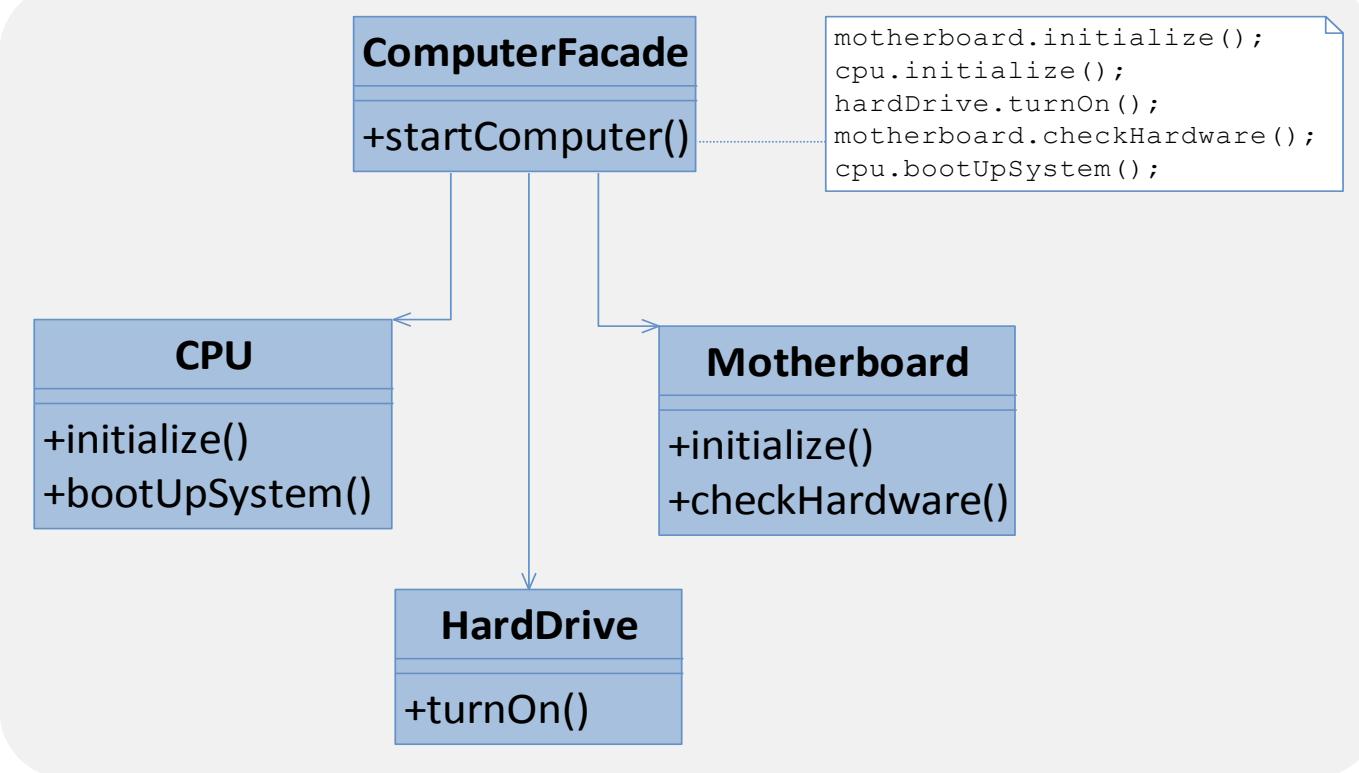
Przykład



start()



Przykład



+turnOn()
HardDrive

bns it} Kod przykładu

```
public class ComputerFacade
{
    private Motherboard motherboard;
    private CPU cpu;
    private HardDrive hardDrive;

    public void StartComputer()
    {
        motherboard.Initialize();
        cpu.Initialize();
        hardDrive.TurnOn();
        motherboard.CheckHardware();
        cpu.BootUpSystem();
    }
}
```

```
public class FacadeExample
{
    public static void Main(String[] args)
    {
        ComputerFacade computerFacade
            = new ComputerFacade();
        computerFacade.StartComputer();
    }
}
```

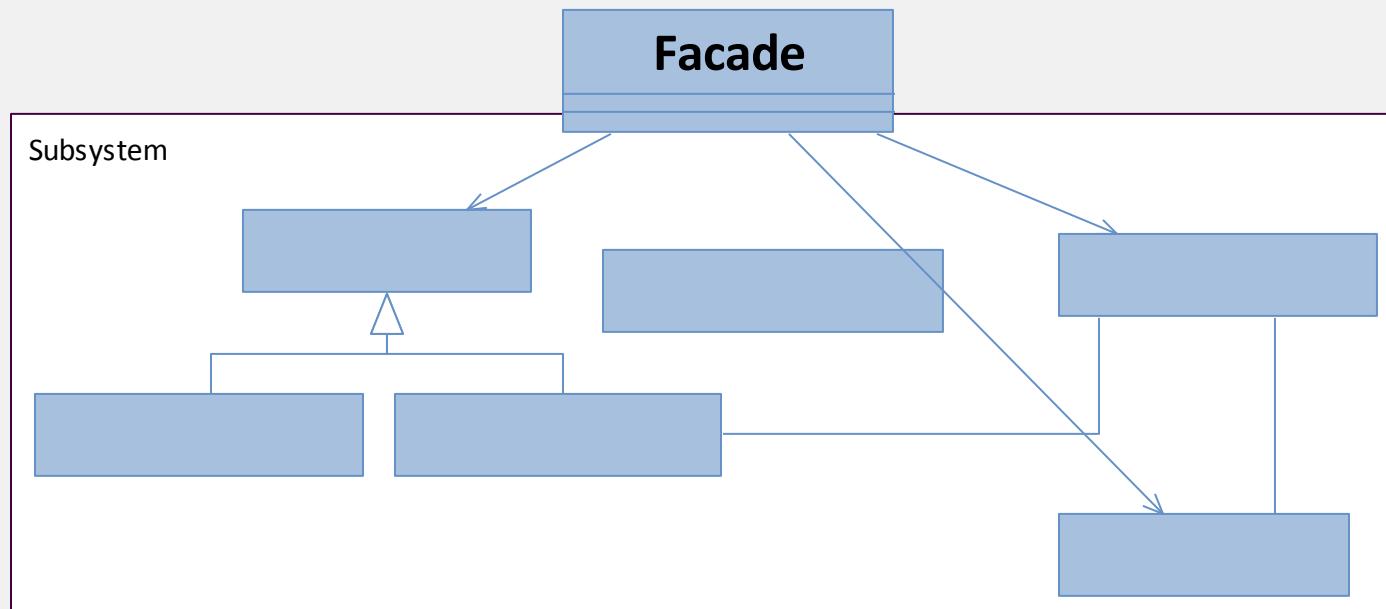
```
public class Motherboard {
    public void Initialize() {
        // ...
    }

    public void CheckHardware() {
        // ...
    }
}
```

```
public class CPU {
    public void Initialize() {
        //...
    }

    public void BootUpSystem() {
        //...
    }
}
```

```
public class HardDrive {
    public void TurnOn() {
        //...
    }
}
```



Wzorzec *Facade* zapewnia jednolity interfejs dla podsystemu.

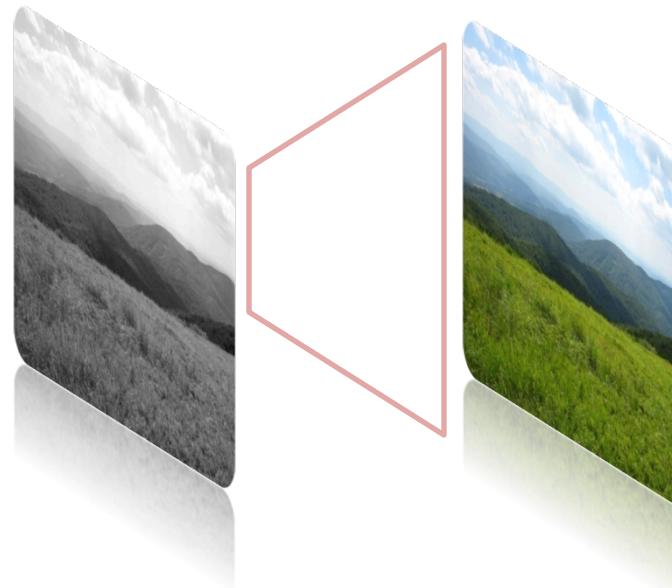
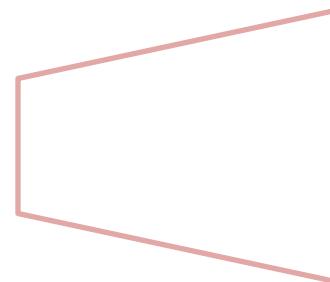
- # Wzorzec Facade zmniejsza ilość obiektów, z którymi klient podsystemu musi współpracować.
- # Tworzy słabe powiązanie klienta z podsystemem co umożliwia modyfikowanie podsystemu bez wprowadzania zmian w kliencie.
- # Klient może wybrać, czy chce komunikować się z podsystemem za pomocą fasady czy za pomocą bezpośrednich odwołań do jego elementów.

- # Ułatwienie korzystania ze skomplikowanego podsystemu
- # API biblioteki definiujące jej zunifikowany i uproszczony interfejs

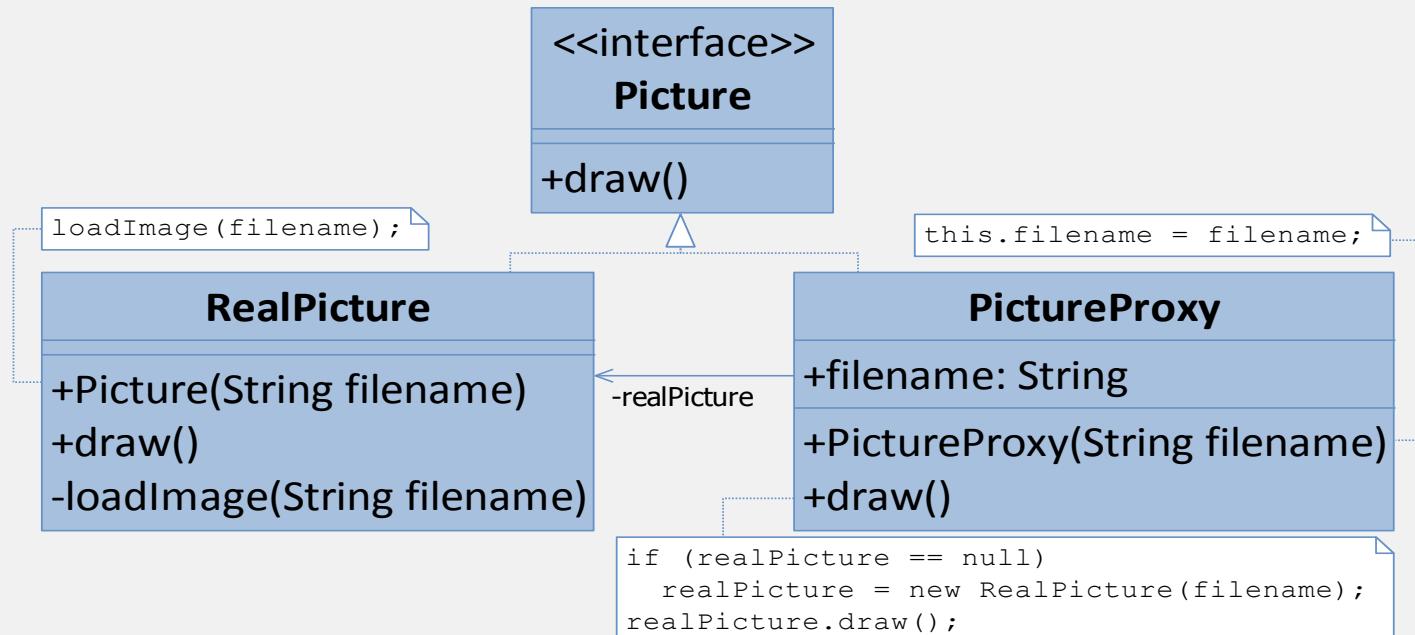
bns it } # } Wzorzec Proxy

Wzorce strukturalne

Przykład



Przykład



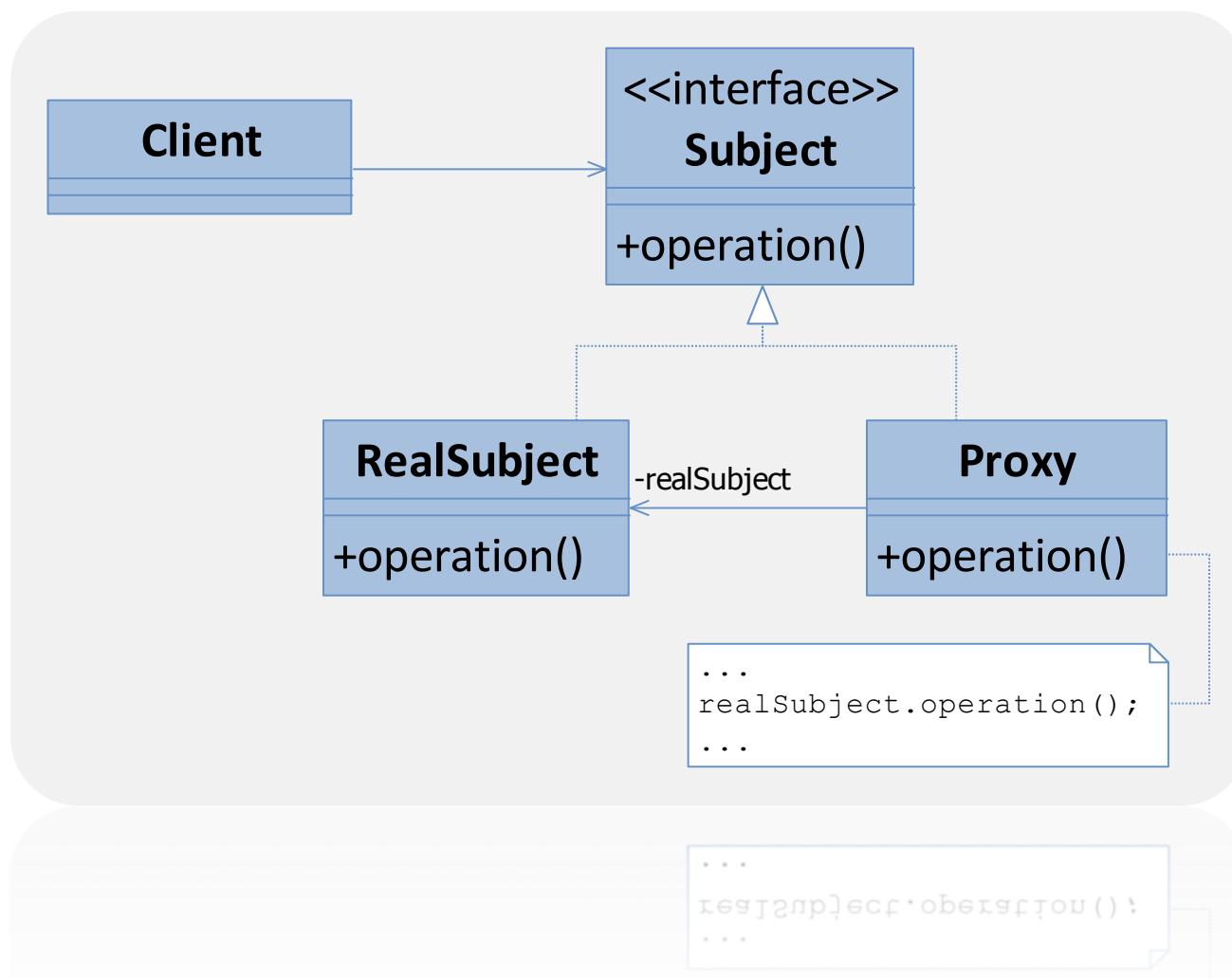
```
Picture picture1 = new PictureProxy("picture1.png");
Picture picture2 = new PictureProxy("picture2.png");
Picture picture3 = new PictureProxy("picture3.png");

picture1.draw()
picture2.draw()
```

Picture picture1
Picture picture2

Picture picture1 = new PictureProxy("picture1.png");
Picture picture2 = new PictureProxy("picture2.png");
Picture picture3 = new PictureProxy("picture3.png");

Wzorce projektowe i refaktoryzacja do wzorców



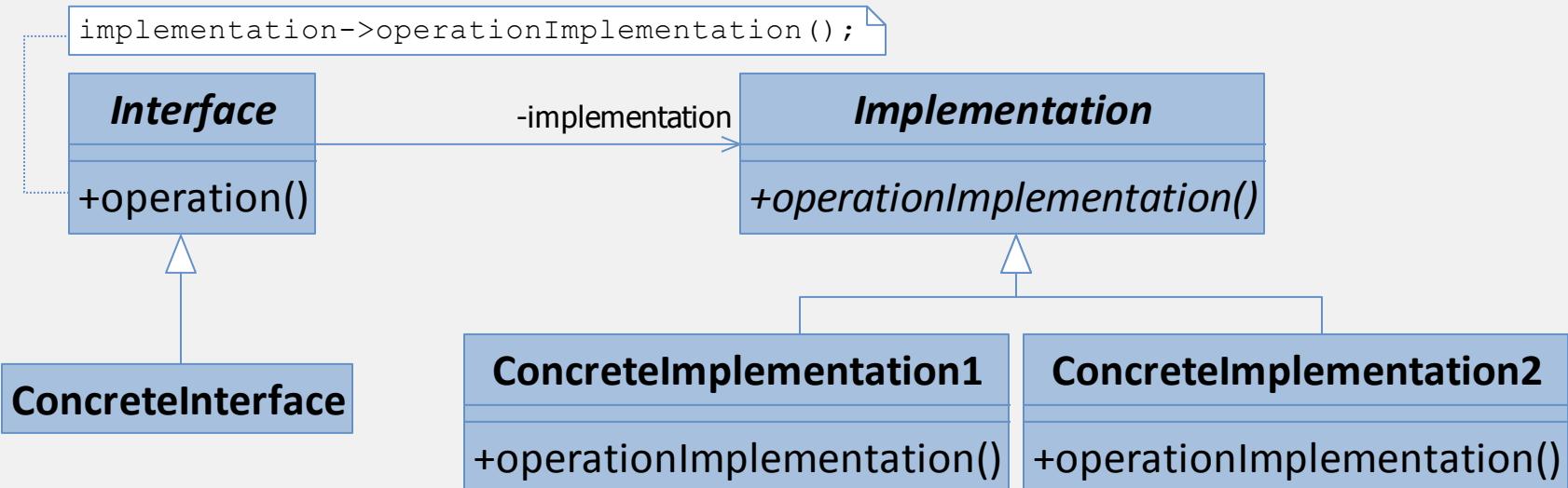
Wzorzec *Proxy* zapewnia reprezentanta obiektu, który steruje dostępem do niego.

Wprowadza dodatkową warstwę przy dostępie do obiektu.

- # *Remote Proxy – lokalny reprezentant zdalnego obiektu*
- # *Virtual Proxy – odsuwa w czasie tworzenia kosztownych obiektów do momentu, kiedy jest to naprawdę konieczne*
- # *Protective Proxy – kontroluje dostęp do oryginalnego obiektu*
- # *Smart Proxy – wtrąca dodatkowe czynności podczas operacji dostępu do obiektu*

bns it } # } Wzorzec Bridge
Wzorce strukturalne

Wzorzec *Bridge* separuje abstrakcję od implementacji, tak aby mogły zmieniać się niezależnie.



ConcreteInterface

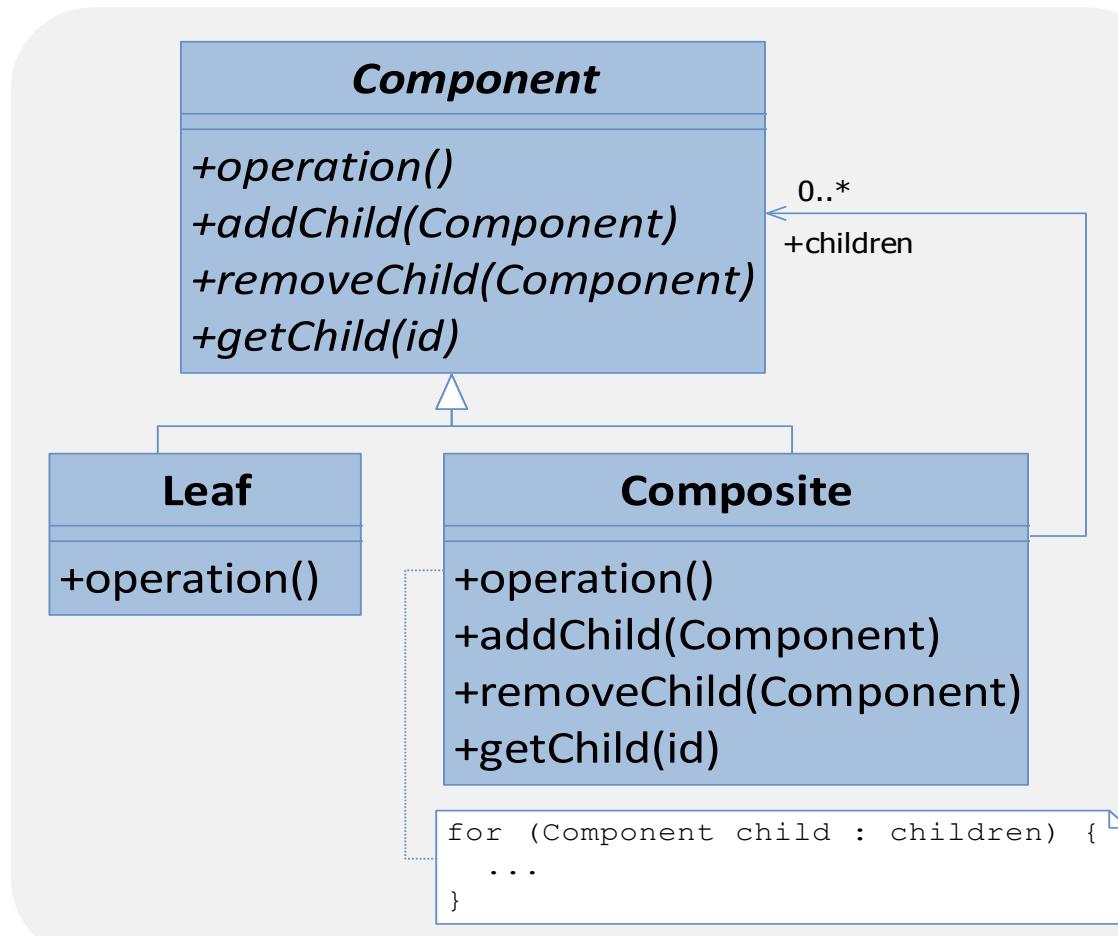
+operationImplementation()

+operationImplementation()

- # Wzorzec *Bridge* umożliwia ustalanie implementacji danego interfejsu w trakcie działania programu.
- # Kilka obiektów może współdzielić jedną implementację.
- # Klasy *Interface* i *Implementation* można rozszerzać niezależnie.
- # Ukrywa szczegóły implementacji przed klientami.

bns it } # } Wzorzec Composite
Wzorce strukturalne

Wzorzec *Composite* składa obiekty w drzewiaste struktury reprezentujące hierarchie część-całość.



```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

```
...
```

```
for (Component child : children) {
```

```
    ...  
}
```

```
}
```

- # Wzorzec *Composite* definiuje hierarchię obiektów prostych i złożonych.
- # Klient może jednakowo traktować struktury proste jak i złożone.
- # Nowo zdefiniowane podklasy klasy *Leaf* bądź *Composite* automatycznie potrafią współpracować z istniejącymi strukturami.
- # Brak ograniczeń dotyczących składania komponentów.

bns it } # Wzorzec Flyweight
Wzorce strukturalne

Wzorzec *Flyweight* wykorzystuje współdzielenie obiektów w celu efektywnej obsługi dużej liczby drobnych obiektów.

```
Flyweight flyweight = flyweights.get(id)
if (flyweight == null) {
    flyweight = createFlyweight();
    flyweights.put(id, flyweight);
}
return flyweight;
```

FlyweightFactory
+createFlyweight(id)

-flyweights

0..*

Flyweight

+operation(externalState)

ConcreteFlyweight
+internalState
+operation(externalState)

NotShareableFlyweight
+wholeState
+operation(externalState)

Client

Client

+operation(externalState)

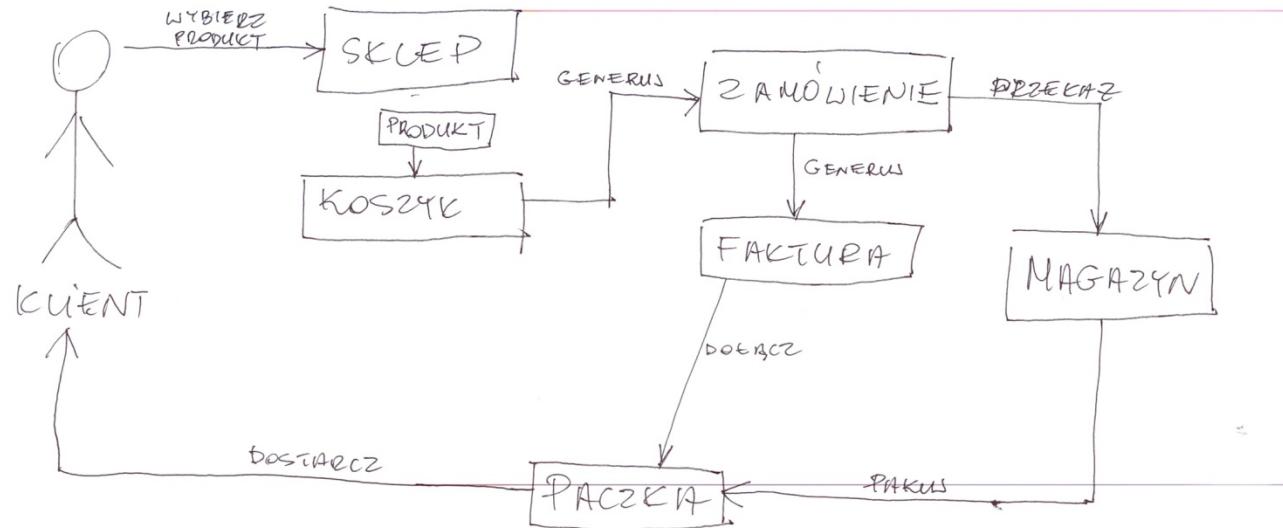
+operation(externalState)

- # Współdzielenie obiektów *Flyweight* skutkuje oszczędnością pamięci na skutek zmniejszenia łącznej ilości obiektów i zwiększenia zakresu współdzielonego stanu.
- # Wzorzec *Flyweight* często łączy się ze wzorcem *Composite* w celu przedstawienia struktury drzewiastej ze współdzielonymi węzłami-liśćmi.

Wybrane wzorce architektoniczne

Wzorce projektowe i refaktoryzacja do wzorców

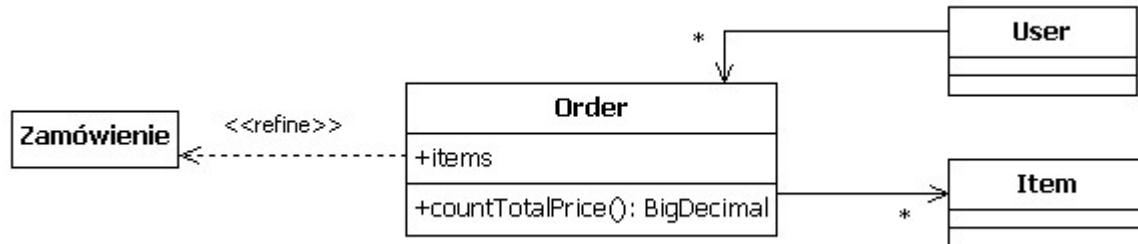




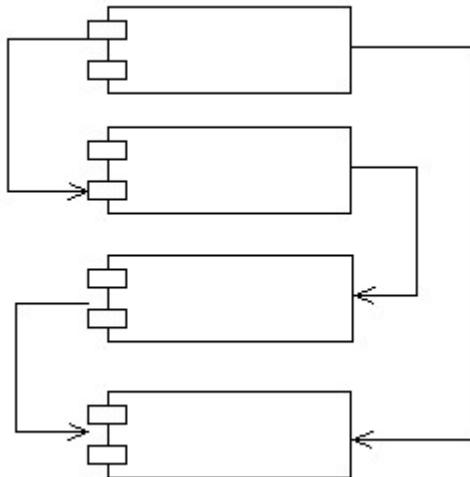
Model dziedziny (Domain Model)

- # Opisuje fragment struktury oraz dynamiki informatyzowanego procesu
- # Definiuje zakres systemu

Obiekt dziedziny



Element modelu dziedziny jest przekształcany w jeden lub wiele **obiektów dziedziny (Domain Object)**, które są programistycznym odwzorowaniem rzeczywistego bytów



- # Warstwa grupuje obiekty o analogicznej odpowiedzialności np.: dostęp do danych, interakcja z użytkownikiem
- # Dzielenie na warstwy jest podstawową techniką strukturyzowania kodu
- # Bezpośrednia komunikacja odbywa się od **warstwy wyższej do warstwy niższej**
- # Brak warstw usztywnia architekturę systemu
- # Zbyt duża ilość warstw wprowadza zbyteczne komplikacje

Presentation Layer (Warstwa prezentacji)

Prezentowanie informacji użytkownikowi i obsługa żądań pochodzących od użytkownika.



Domain Layer (Warstwa dziedziny)

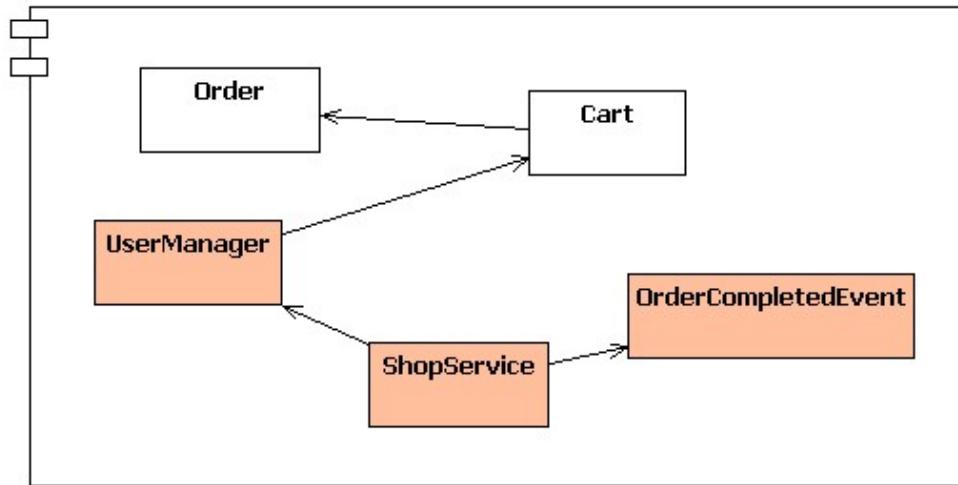
Implementacja zadań biznesowych systemu.



Data Source Layer (Warstwa dostępu do danych)

Komunikacja z bazami danych i innymi systemami, które spełniają zadania na rzecz aplikacji.

Dekomponowanie warstwy



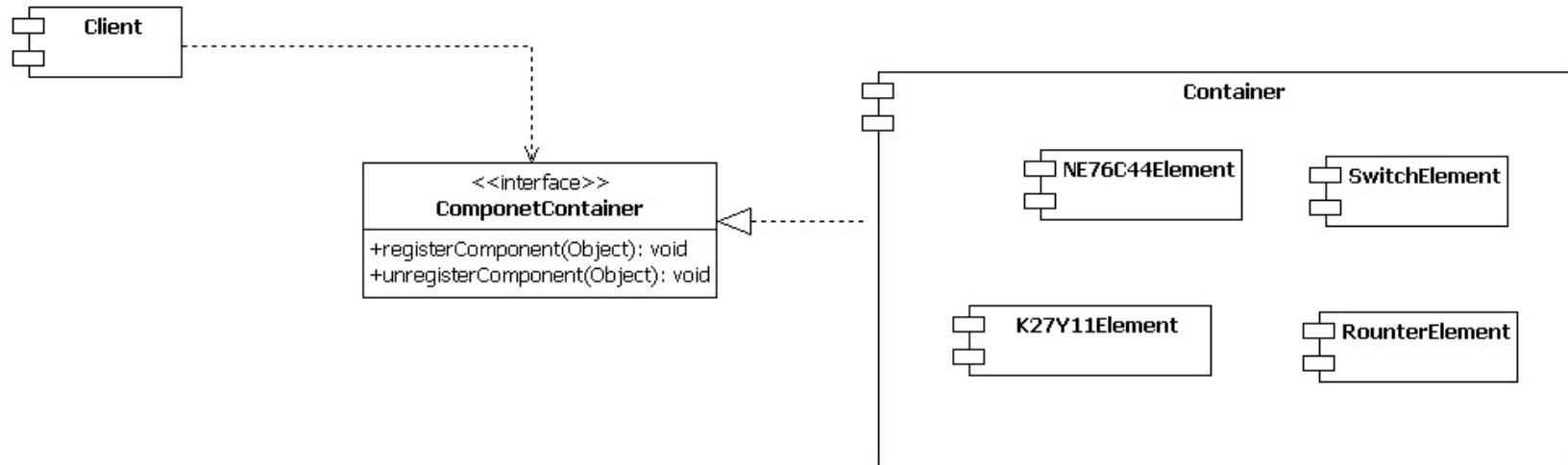
- # Warstwa jest dekomponowana na części przez **obiekty dziedziny**
- # W każdym systemie oprócz elementów modelu znajdują się również **obiekty abstrakcyjne**, które łączą system w całość

Obiekty modelu dziedziny

- Modelują fragmenty rzeczywistości

Obiekty abstrakcyjne

- Przejmują część odpowiedzialności, która jest w systemie niezbędna lecz nie należy do żadnego z obiektów dziedziny
- Pełnią funkcję łącznika pomiędzy elementami



- Osłabia zależności w systemie poprzez mechanizm Dependency Injection
- Klienci mają dostęp do obiektów dzięki interfejsom udostępnianym przez kontener
- Zarządza cyklem życia komponentów

Kontener - popularne implementacje

- # *Unity* - <http://unity.codeplex.com>
- # *Castle Windsor* -
<http://www.castleproject.org/container>
- # *Spring.NET* - <http://www.springframework.net>



Dostęp do danych

Wybrane wzorce architektoniczne

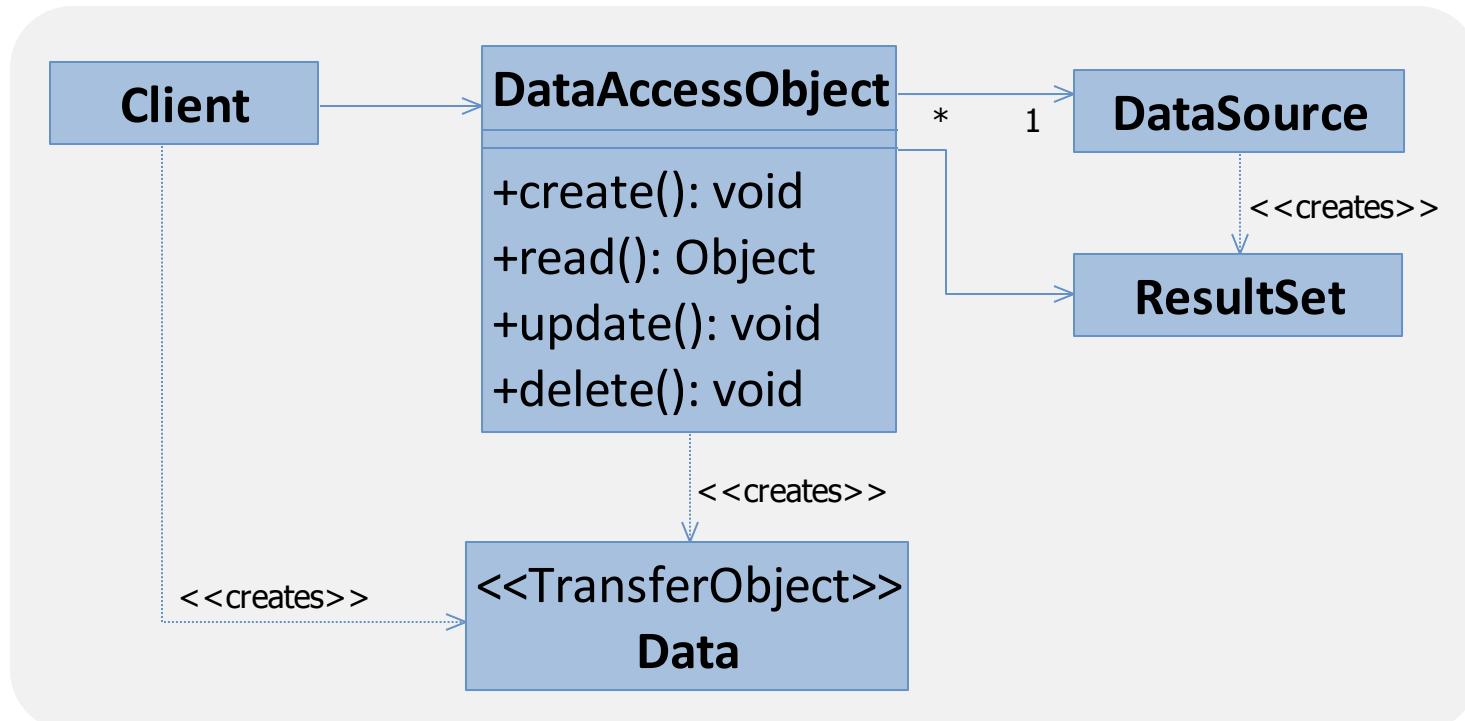
Opisuję sposoby komunikacji warstwy dziedziny problemu ze źródłem danych.

Wzorzec Data Access Object (Table Data Gateway)

Wybrane wzorce architektoniczne

Ukrywa logikę dostępu do danych w osobnej warstwie.

Struktura

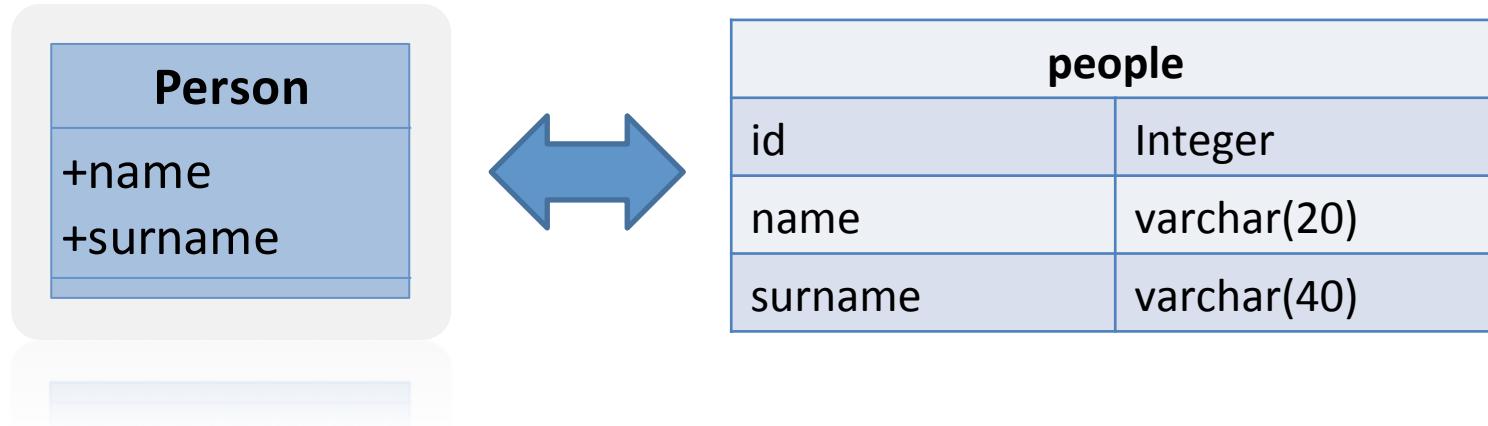


Dat
<<TransferObject>>

bns it} Wzorzec Object-Relational Mapping
Wybrane wzorce architektoniczne

Object-Relational Mapping to sposób odwzorowania obiektowej architektury na bazę danych o relacyjnym charakterze.

Object-Relational Mapping



bns it  **Wzorzec Repository**
Wybrane wzorce architektoniczne

Wzorzec *Repository* pośredniczy pomiędzy warstwami *Domain* i *Data Source* udostępniając interfejs podobny do kolekcji, pozwalający na dostęp do obiektów dziedziny problemu.

Klient konstruuje deklaratywną specyfikację zapytania, które jest obsługiwane przez obiekt *Repository*.

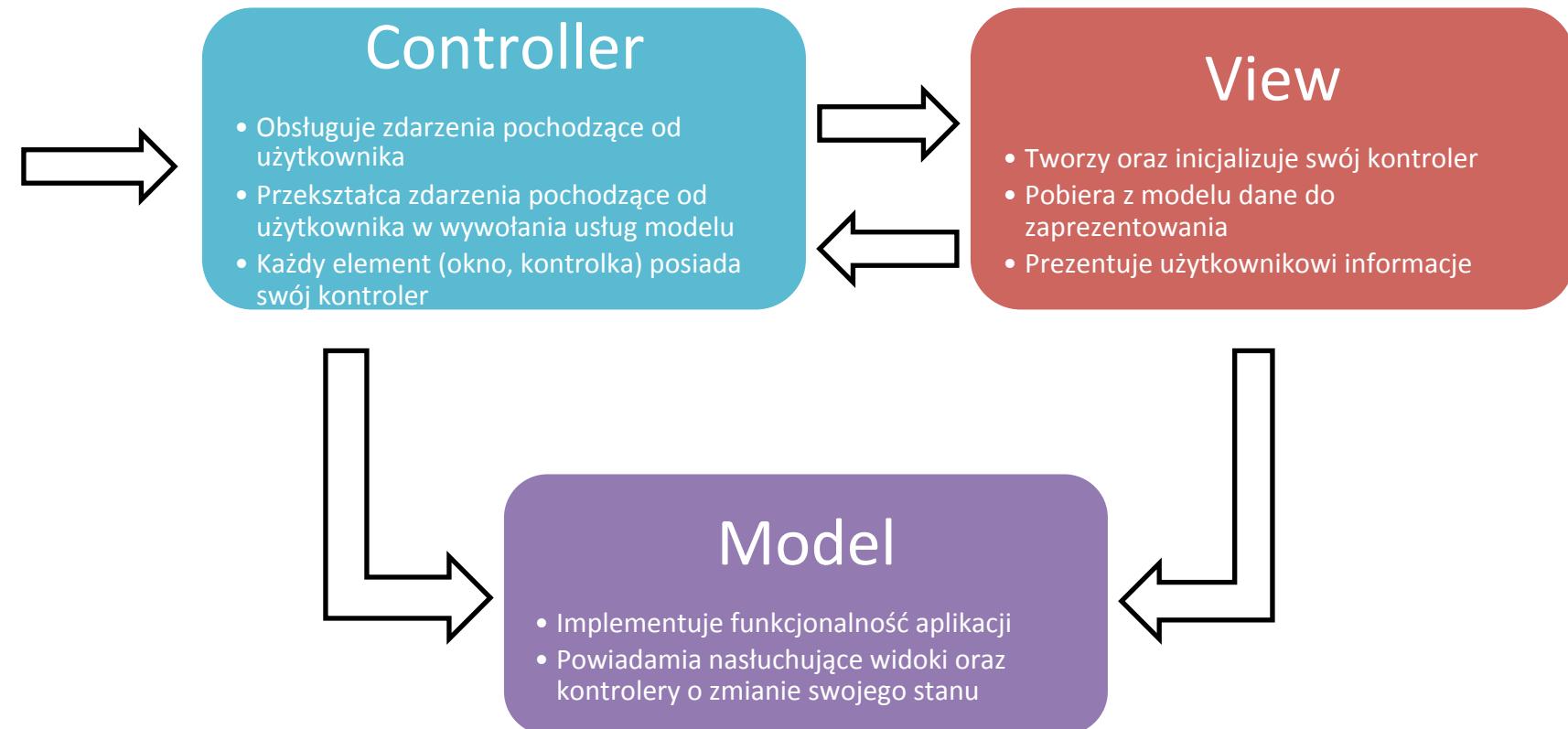
```
Criteria criteria = new Criteria();
criteria.equal(Person.FIRST_NAME, "Jan");
List<Person> persons = repository.matching(criteria);
```



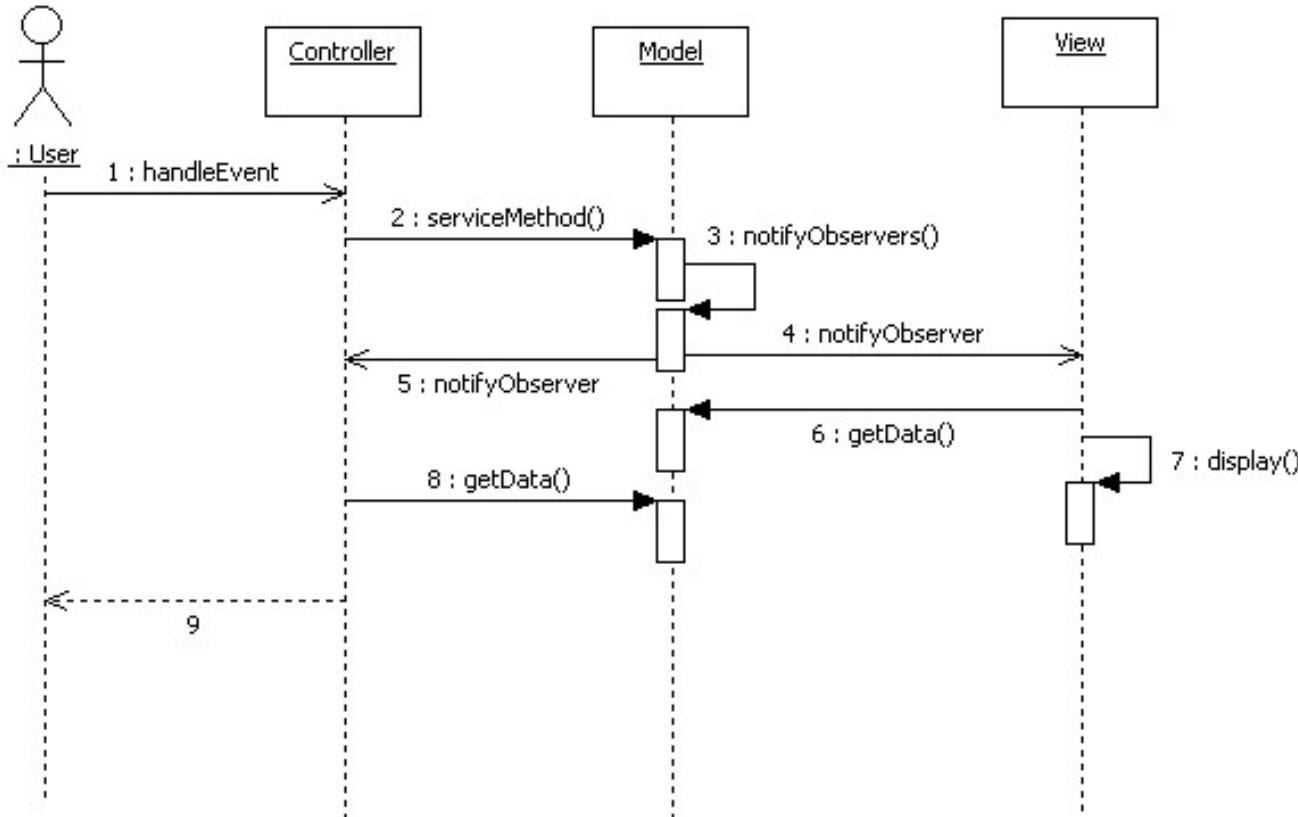
Prezentacja i komunikacja z użytkownikiem

Wybrane wzorce architektoniczne

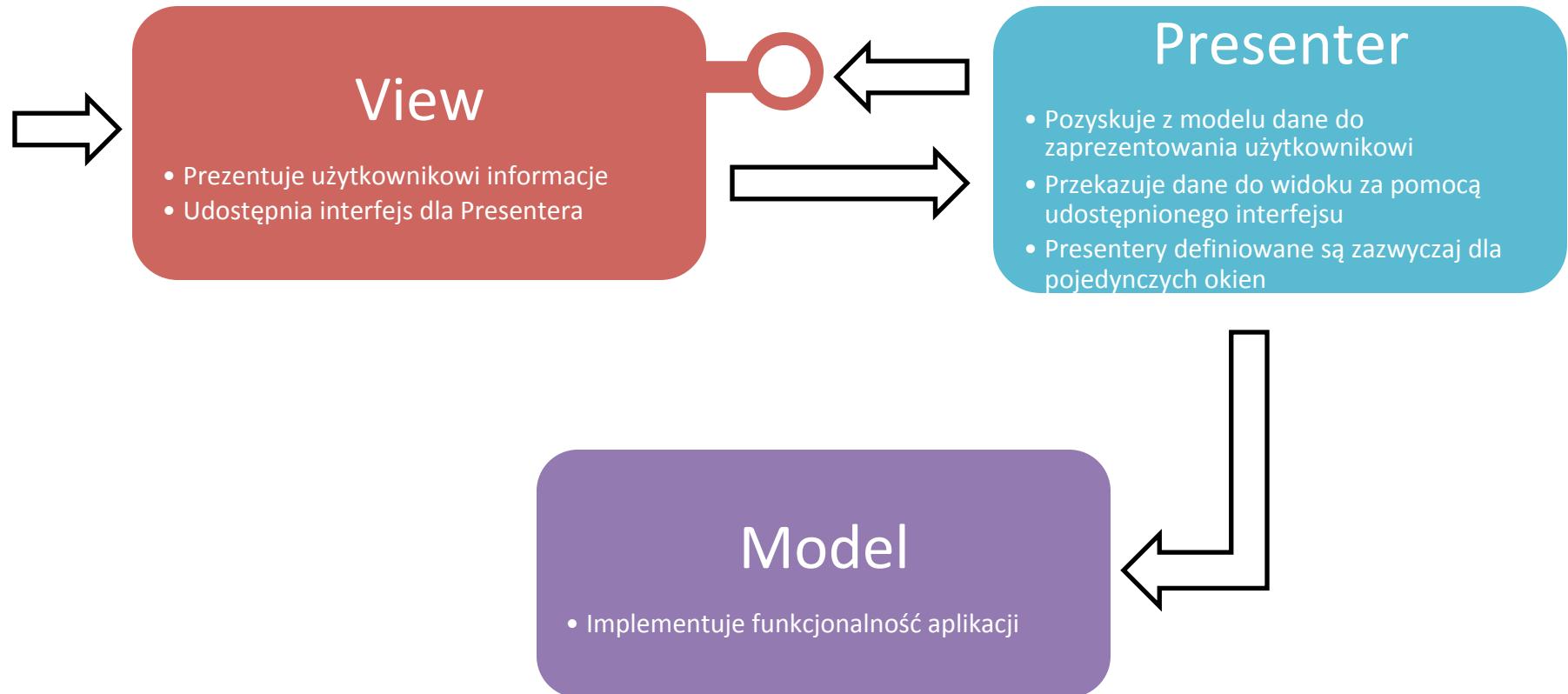
Wzorzec Model View Controller



MVC - obsługa żądania

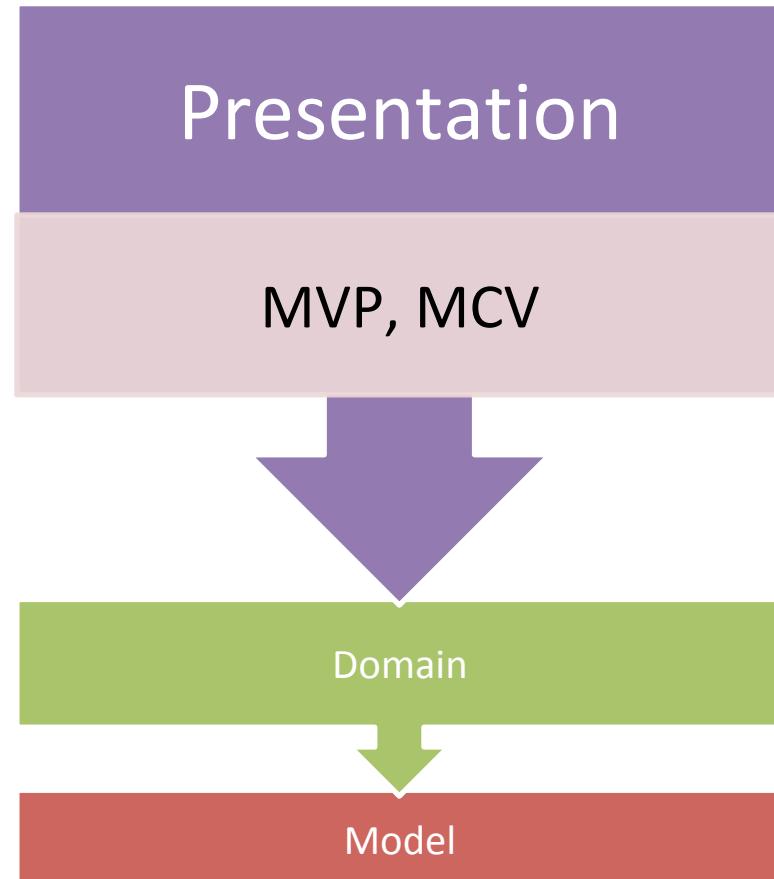


Wzorzec Model View Presenter



MVP, MVC mieszczą się w warstwie prezentacji.

Są to sposoby na zarządzanie logiką interfejsu użytkownika i uniezależnia aplikację od konkretnego sposobu prezentacji.



bns it } # ANTYWzorzec Anemic Domain Model
Wybrane wzorce architektoniczne

Anemic Domain Model to **antywzorzec** architektoniczny opisujący rozwiązania architektoniczne, w których obiekty dziedziny problemu są pozbawione zachowania

- # Obiekty dziedziny problemu są pozbawione zachowania
- # W zamian istnieje wiele obiektów usługowych, które implementują logikę zachowania natomiast nie przechowują stanu.