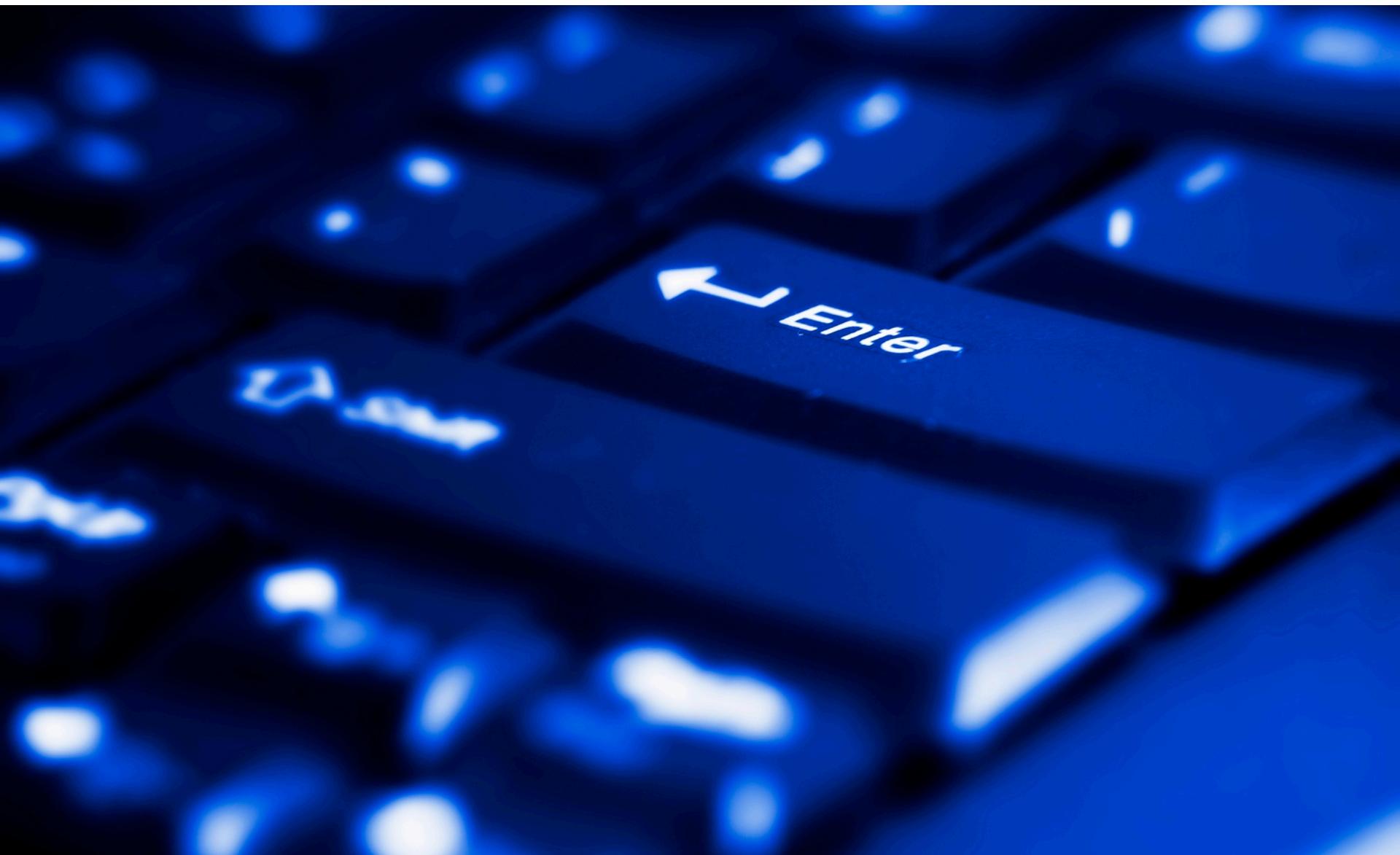




# Wstęp

Wzorce projektowe i refaktoryzacja do wzorców



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

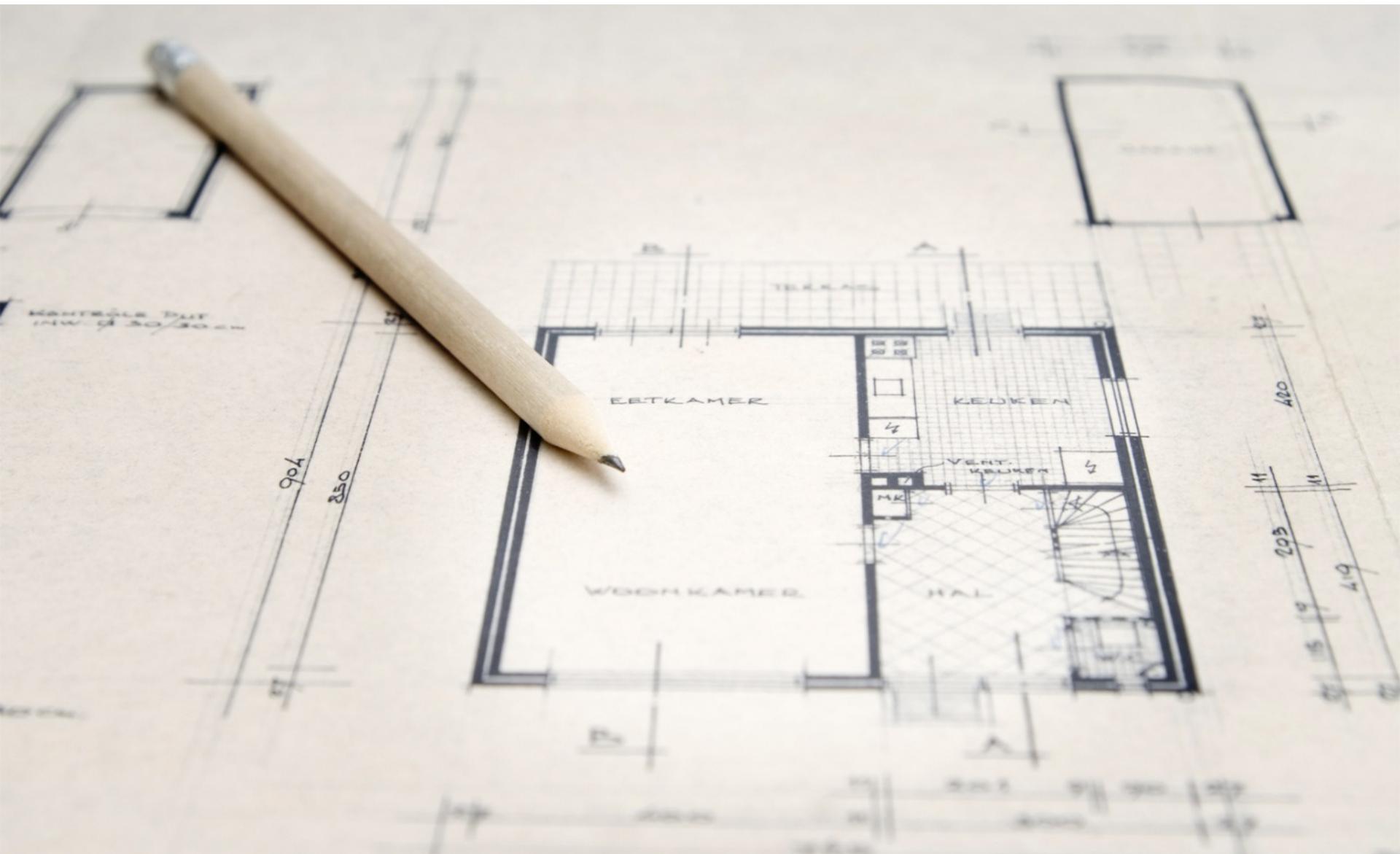


**Imię i nazwisko  
Stanowisko  
Doświadczenie  
Oczekiwania**

**Poznajmy się!**

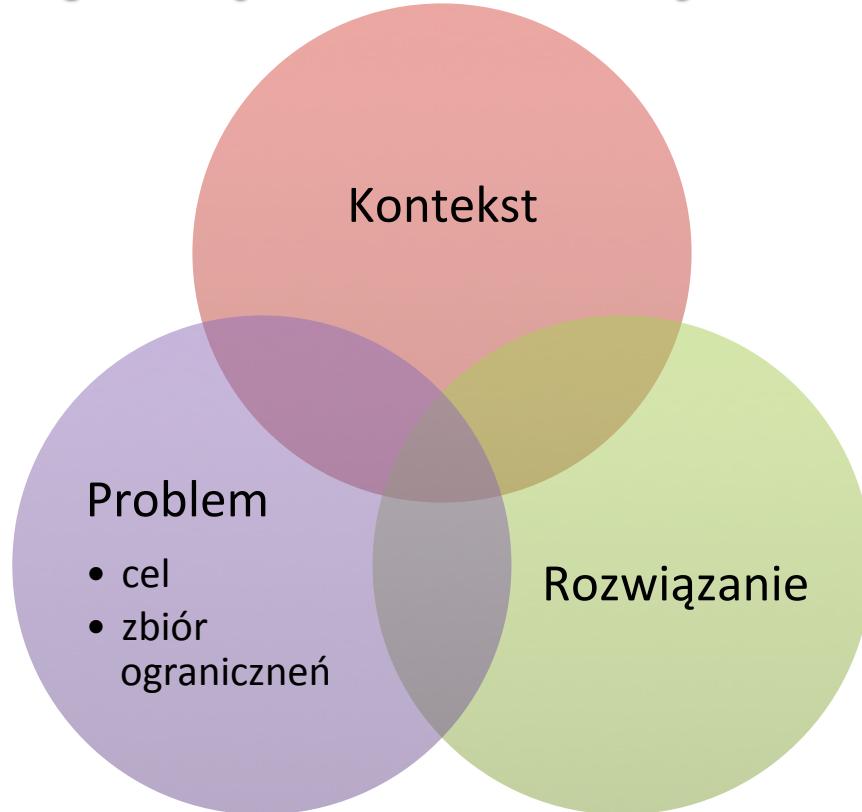
# Wprowadzenie do wzorców projektowych

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

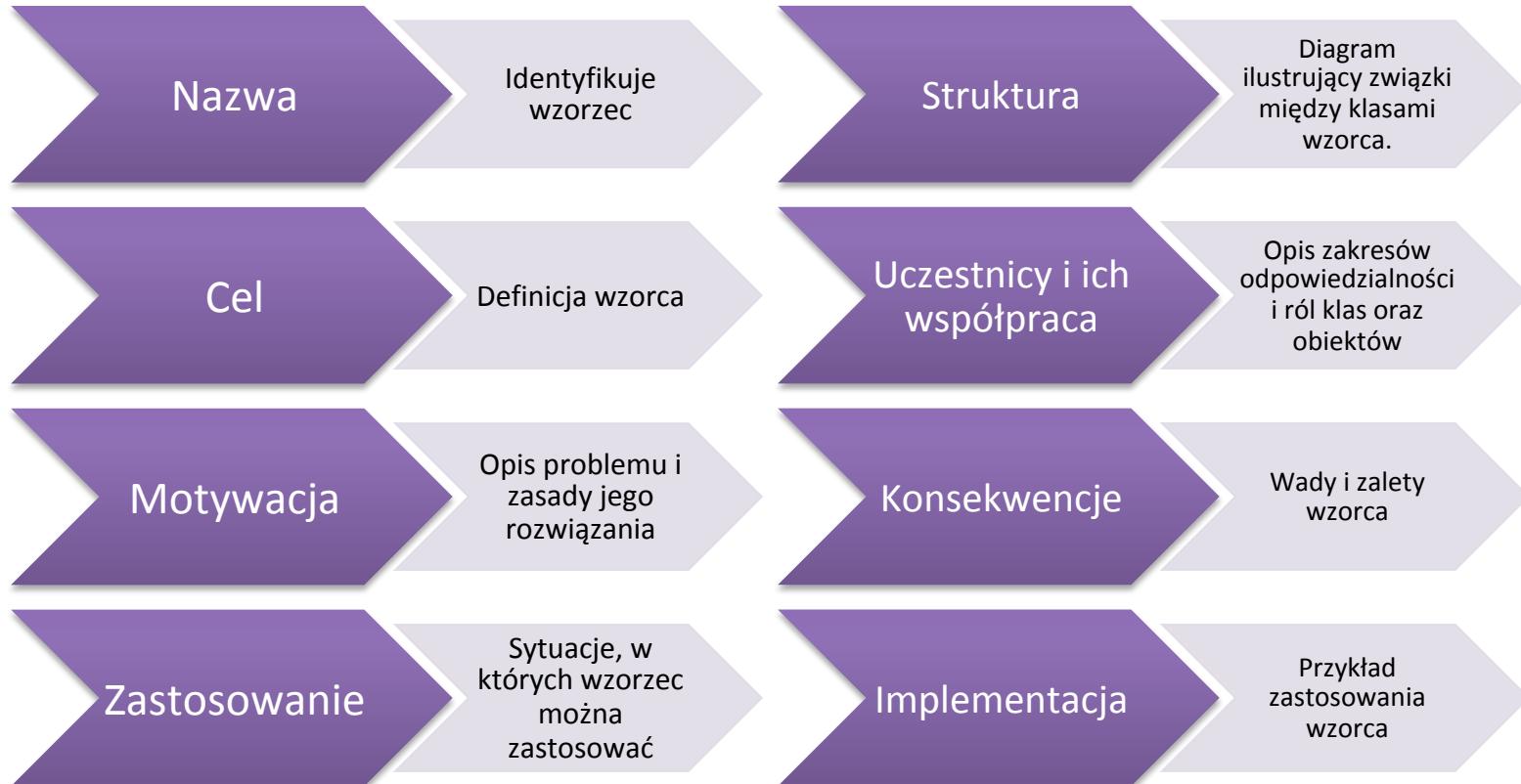
Pattern-Oriented Software Architecture (seria), 1996-2007  
Fowler, *Patterns of Enterprise Application Architecture*, 2002

## 3. Wzorce integracyjne

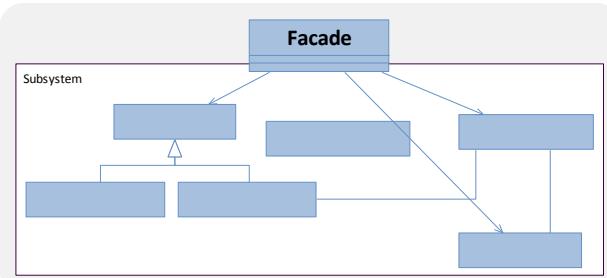
Hohpe, Woolf, <http://www.enterpriseintegrationpatterns.com>

- # 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.

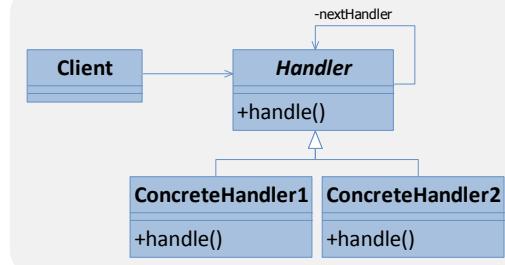
# Kluczowe elementy opisu wzorca projektowego



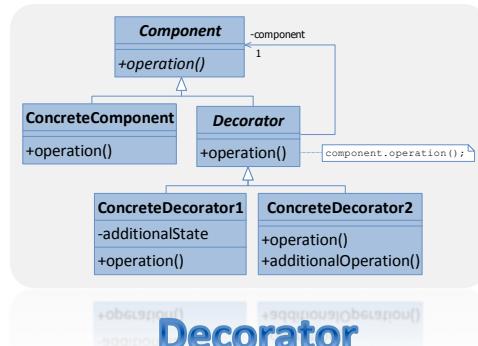
# Przykłady wzorców projektowych



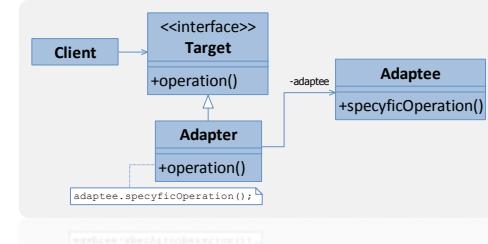
## Facade



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

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

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

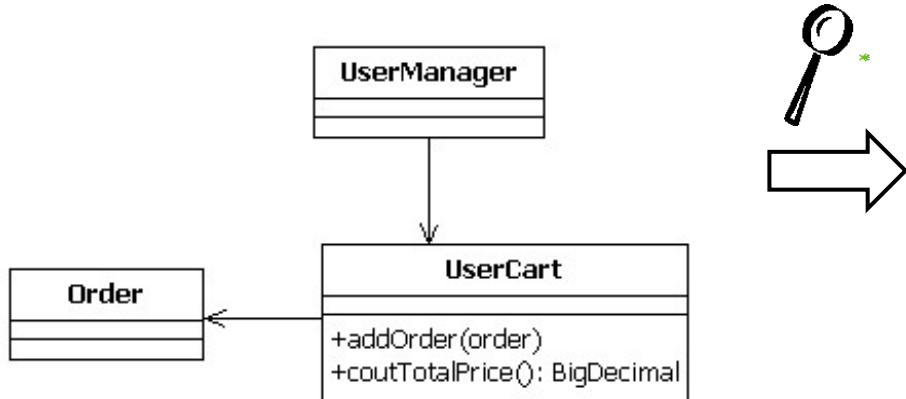
## # Do uzyskania nowej funkcjonalności używaj delegacji zamiast przeciążania

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

## # Definiuj w miarę niezmiennej sposób komunikowania się komponentu z otoczeniem

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

# Wzorce projektowe, wzorce implementacyjne



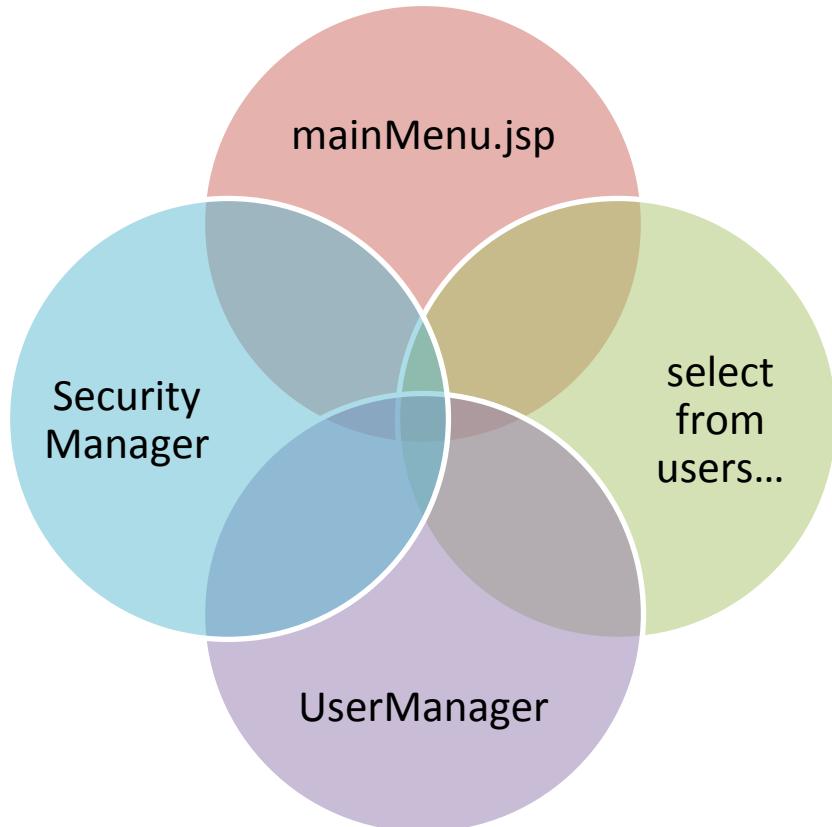
```
public class UserCart{
    public void addOrder( Order order ) {
        //...
    }

    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

```
<html><body>
<table>
<%
while( rs.next() ){
%><tr>
<td><%=rs.getString("id") %></td>
<td><%=rs.getString("date") %></td>
<td><%=rs.getString("email") %></td>
</tr>
<% }%>
<%}
catch(Exception e){e.printStackTrace();}
finally{
if(con!=null) con.close();
}
%>
</body></html>
```

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

# Silna zależność: warstwy

```
public class Department {  
    public void addEmployee( String name, long depId ) {  
        //...  
        st.executeUpdate( "insert into empls values( 1" + "," + name + "" + ")" );  
        st.executeUpdate( "update deps set empnum=" + 7 + " where id=" + depId + ")" );  
    }  
}  
  
public class EmployeesService {  
    public ResultSet findAllEmployees() {  
        //...  
        return st.executeQuery( "select * from employees;" );  
    }  
}
```

- # 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

```
public 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

```
public 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( HttpServletRequest req ) {  
        String street = (String) req.getAttribute( "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

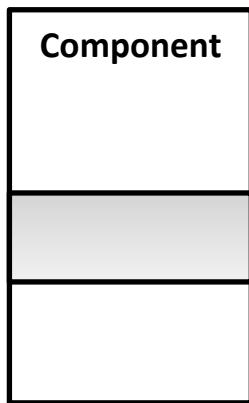
- # Programowanie poprzez interfejsy
- # Wstrzykiwanie zależności
- # Programowanie aspektowe
- # Komunikacja asynchroniczna



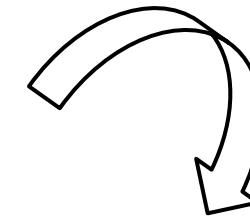
# Po co nam interfejsy?

Jakość kodu źródłowego

# Definiowanie interfejsu



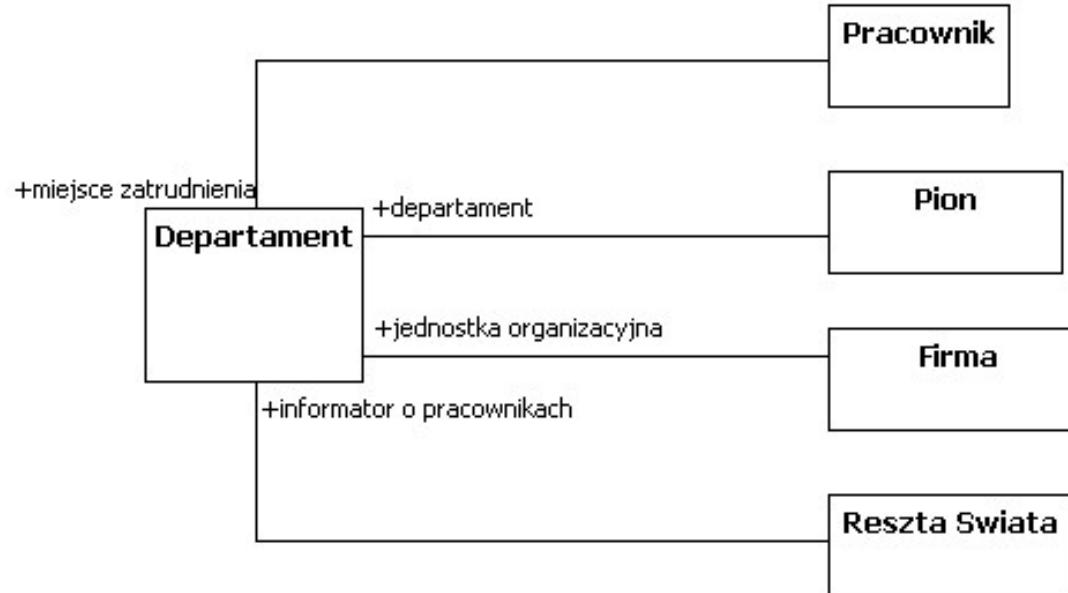
```
public class Departement {  
    public void addEmployee(empl) {  
        //...  
    }  
    public Employee getDirector() {  
        //...  
    }  
    public Employee notifyEmployees(msg) {  
        //...  
    }  
}
```



- # 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

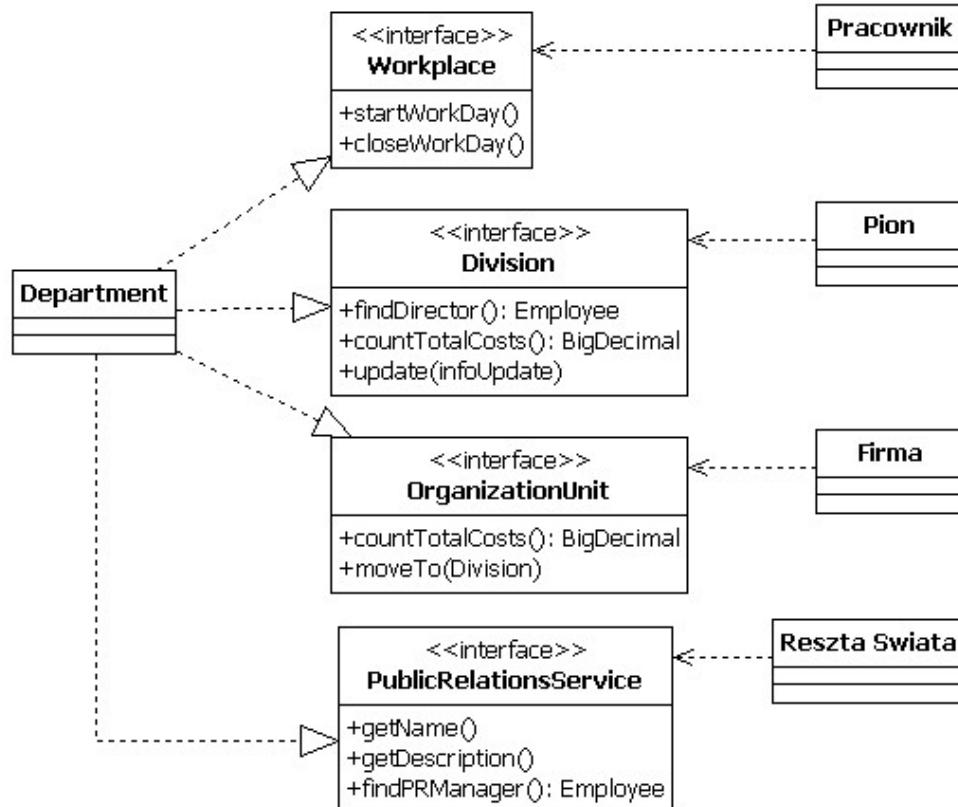
```
public interface OrganizationalUnit {  
    public void addEmployee(empl);  
    public Employee getDirector();  
    public Employee notifyEmployees(msg);  
}  
  
public class Departement  
implements OrganizationalUnit { //...
```

# Dzielenie interfejsów



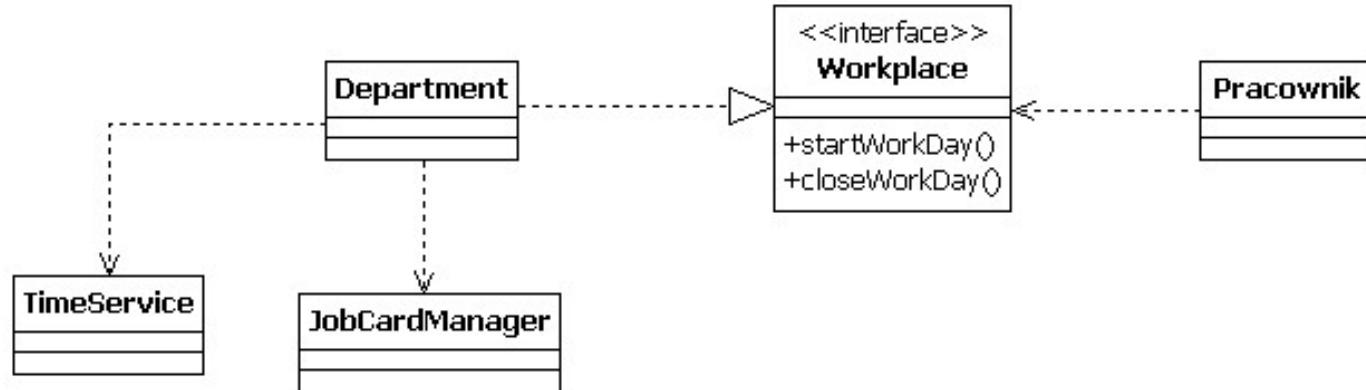
- # W relacjach z różnymi współpracownikami komponent może pełnić różne role
- # Definiuj nowy interfejs dla nowej roli, zamiast modyfikować istniejące interfejsy

## Dzielenie interfejsów



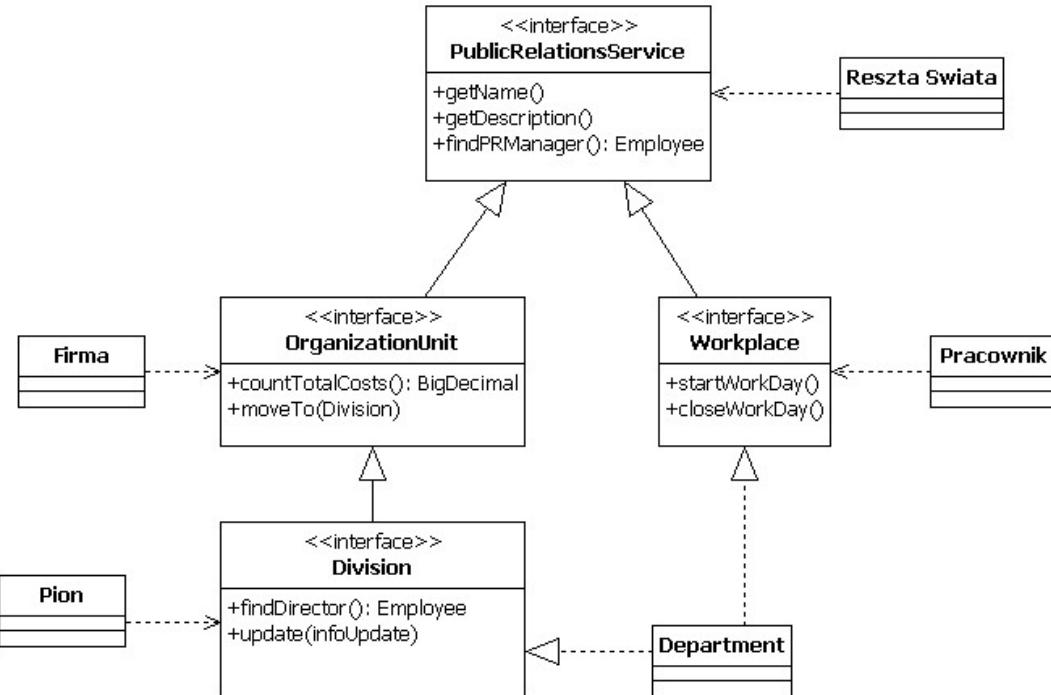
- # Komponent definiuje interfejs dla ról, którą pełni
- # Współpracownicy otrzymują tylko taką funkcjonalność, która jest im niezbędna i nic ponad to

# Dzielenie interfejsów



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

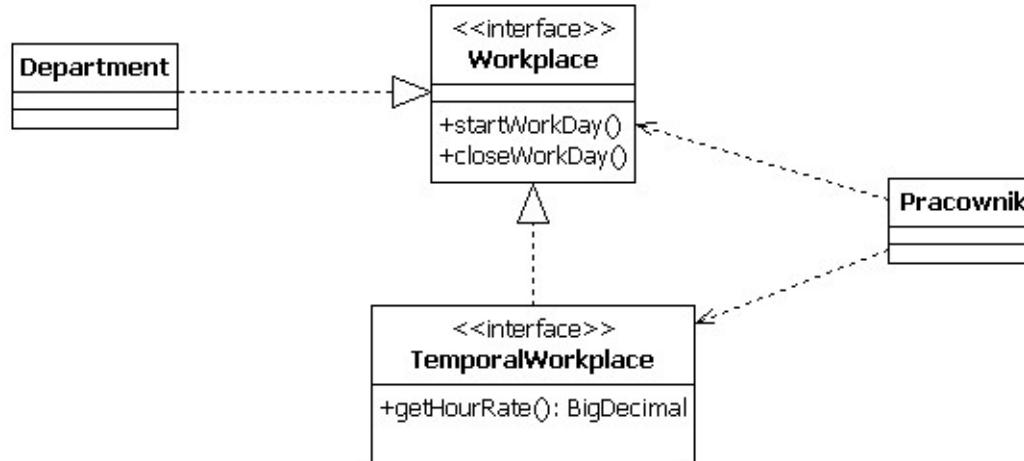
# Rozszerzanie interfejsów



# Rozszerzanie interfejsów pozwala na ustalenie zakresów dostępu do funkcjonalności obiektu

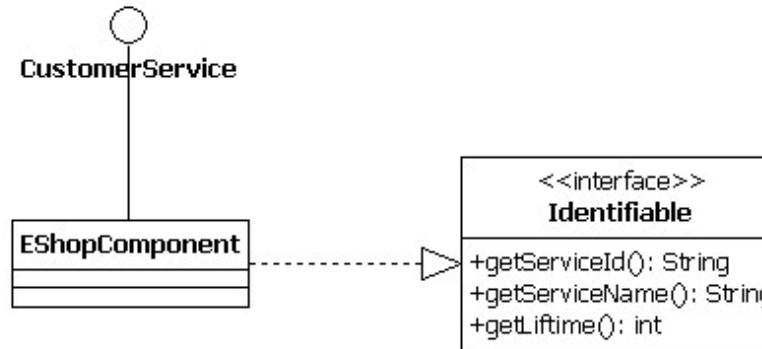
# Zapewnia logiczną ciągłość funkcjonalności

# 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



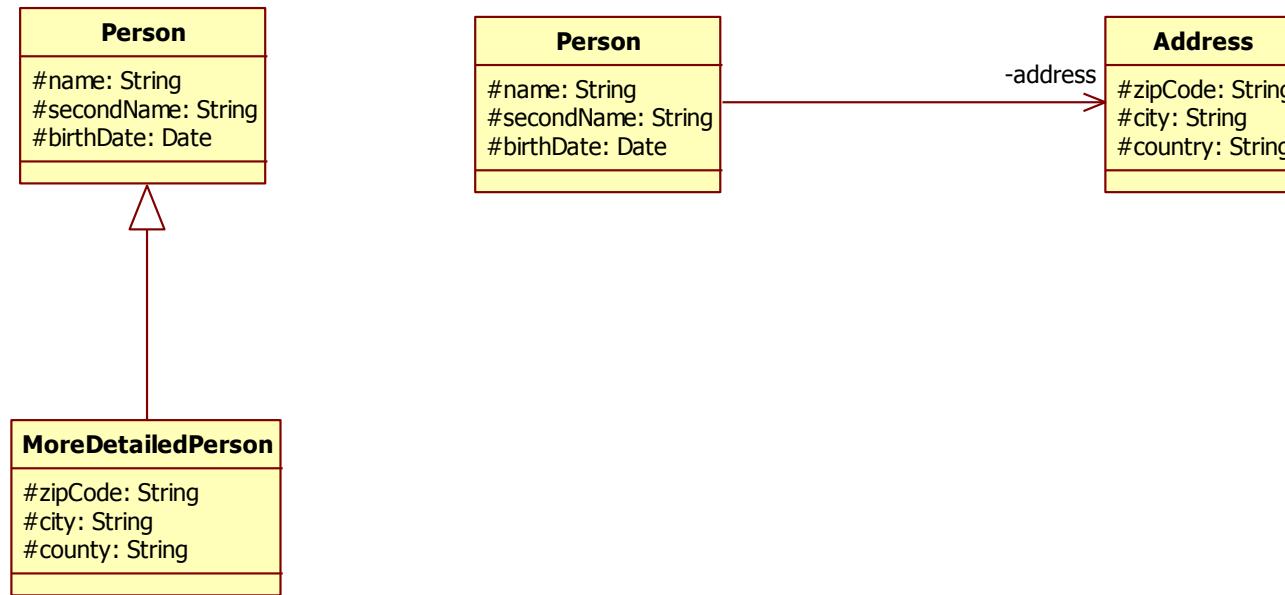
- # **Introspective Interface** jest szczególnym przypadkiem *Extension Interface*
- # Jego odpowiedzialnością jest dostarczenie informacji diagnostycznych o komponencie
- # Tego typu funkcjonalność powinna zostać oddzielona od głównych usług komponentu
- # Szczególnym przypadkiem **Introspective Interface**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

# Programowanie poprzez interfejsy

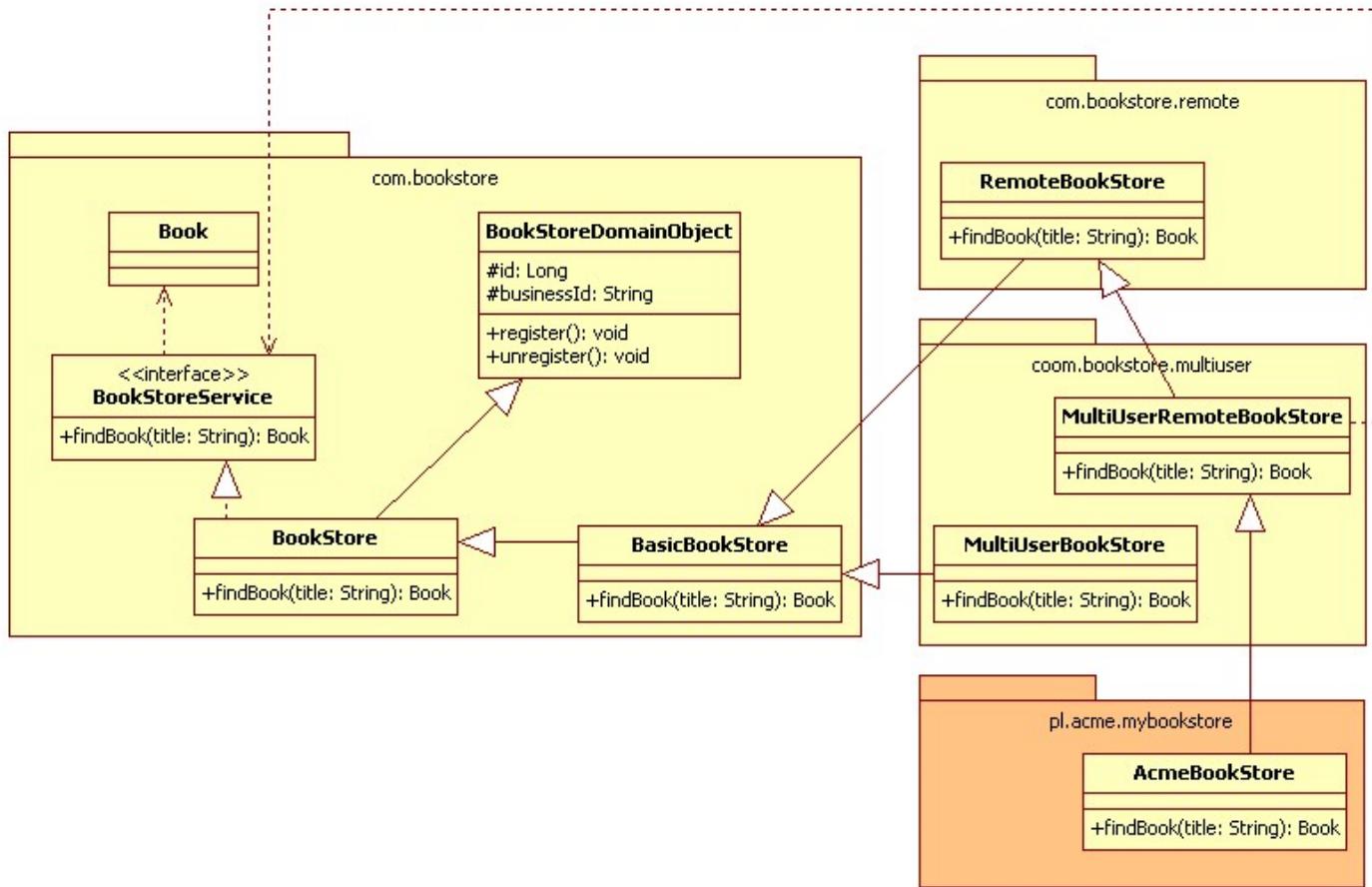
Oznacza, że:	NIE oznacza, że:
<ul style="list-style-type: none"><li>• Interfejs jest najważniejszą częścią komponentu</li><li>• Interfejs to kontrakt, który zobowiązuje się wypełnić komponent</li><li>• Dokładnie przemyśl w jaki sposób komponent komunikuje się ze światem zewnętrznym</li><li>• O ile to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

bns it }    # }   Kompozycja a dziedziczenie  
               Jakość kodu źródłowego

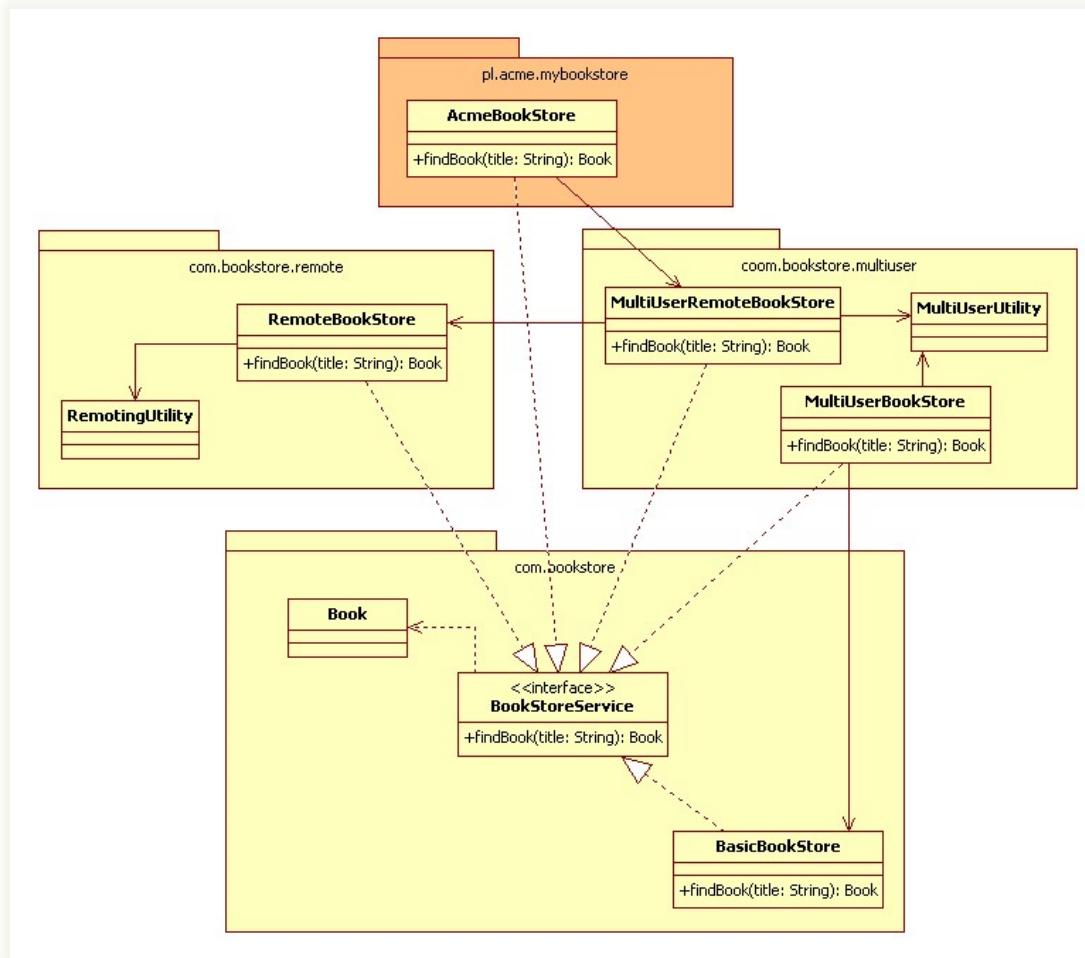
# Kompozycja a dziedziczenie (Prosty przykład)



# 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

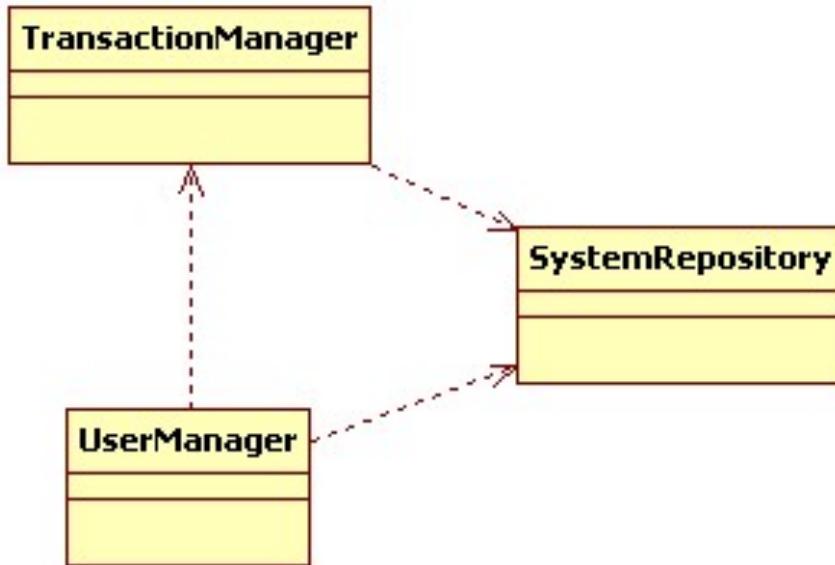
- # Więcej klas
- # Więcej kodu fabrykującego i wiążącego obiekty
- # Niezależność klas zawieranych
- # Można je używać w wielu kontekstach
- # Słabe zależności (coupling)
- # Łatwiej testować



# Wstrzykiwanie zależności

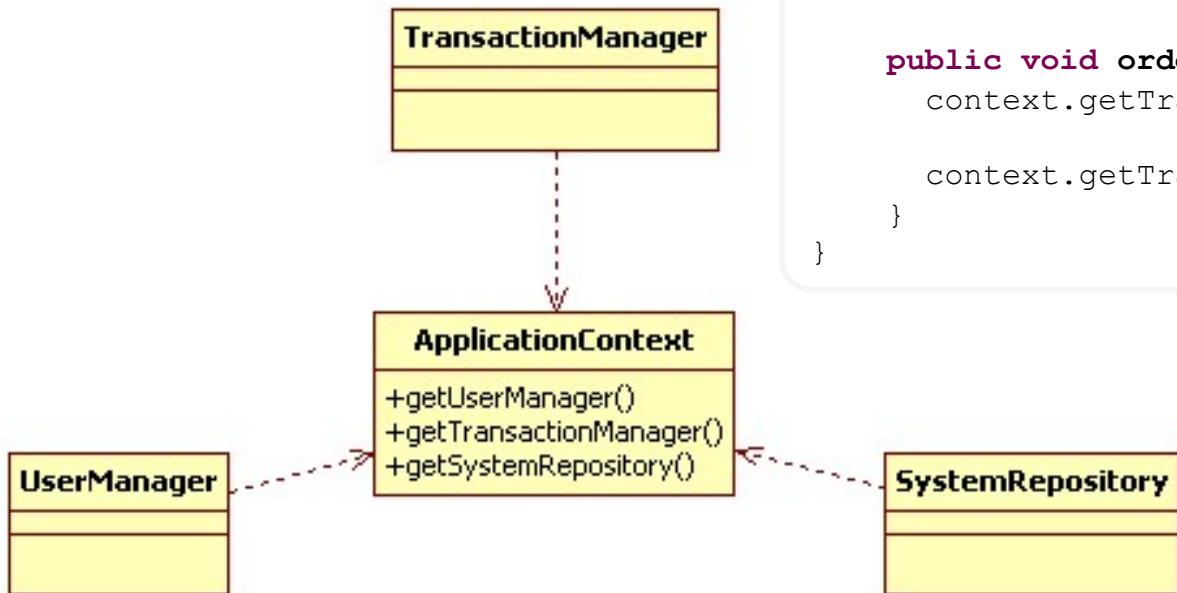
Jakość kodu źródłowego

## 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



```
public class UserManager {

    private ApplicationContext context;

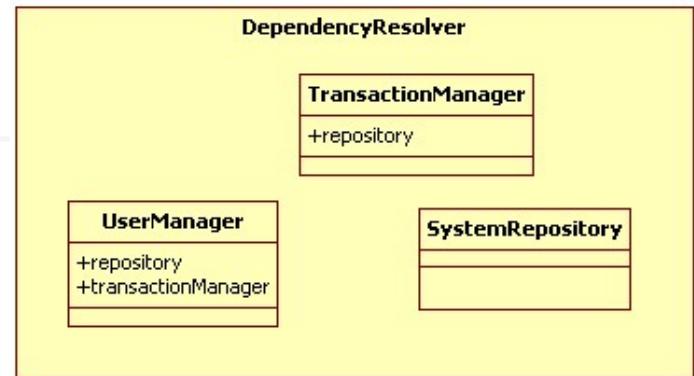
    public void order() {
        context.getTransactionManager().begin();
        //...
        context.getTransactionManager().commit();
    }
}
```

## 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

```
public class UserManager {  
    // zależności klasy  
    private SystemRepository repository;  
    private TransactionManager transactionManager;  
  
    public void setRepository(SystemRepository repository) {  
        this.repository = repository;  
    }  
    public void setTransactionManager(TransactionManager manager) {  
        this.transactionManager = manager;  
    }  
  
    //dane prywatne klasy  
    private Cart cart;  
    private int lifetime;  
}
```



## Otwarcie na wstrzykiwanie zależności

- # 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

# Refaktoryzacja

Wzorce projektowe i refaktoryzacja do wzorców



# 6 sekund na zrozumienie kodu

```
public void add(Object element) {  
    if (!readOnly) {  
        int newSize = size + 1;  
        if (newSize > elements.length) {  
            Object[] newElements = new Object[elements.length + 10];  
            for (int i = 0; i < size; i++) {  
                newElements[i] = elements[i];  
            }  
            elements = newElements;  
        }  
        elements[size++] = element;  
    }  
}
```

# 6 sekund na zrozumienie kodu

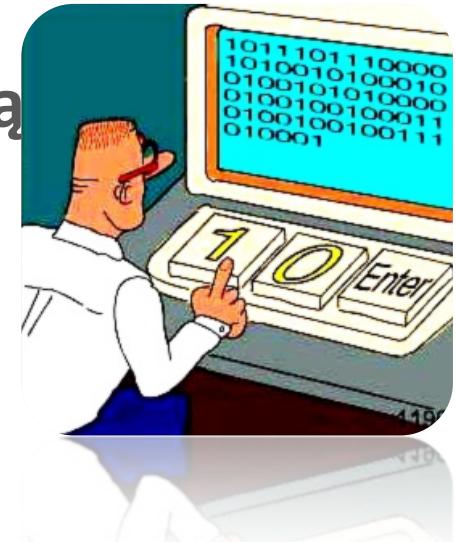
```
public void add(Object element) {  
    if (readOnly)  
        return;  
    if (atCapacity())  
        grow();  
    addElement(element);  
}
```

*„Czysty kod jest prosty i bezpośredni.  
Czysty kod czyta się jak dobrze napisaną prozę.  
Czysty kod nigdy nie zaciemnia zamiarów projektanta; jest pełen trafnych abstrakcji i prostych ścieżek sterowania.”*

*Grady Booch*

Kod zawsze pozostanie niezbędny...

- # Narzędzia do generowania kodu **nie są w stanie spełnić szczegółowych wymagań użytkowników**
- # Kod jest **specyfikacją** wymagań



- # Priorytetem jest **terminowe tworzenie działającego oprogramowania**
- # Efektem pośpiechu jest **zły kod**, którego późniejsza rozbudowa czy chociażby zrozumienie jest **niezwykle utrudnione**
- # Tworzenie złego kodu prowadzi do tzw. brodzenia (ang. wading)



# Dlaczego czystość kodu jest ważna?

- # Programy są częściej **czytane** niż pisane
- # Więcej czasu poświęcamy na **modyfikację istniejącego kodu** niż na tworzenie nowego
- # Programy to struktura oparta na operowaniu **stanem i przepływem**
- # Czytający kod musi móc łatwo przejść z od **ogółu do szczegółu** i vice versa

- # **Komunikacja** – kod źródłowy powinno się czytać jak książkę
- # **Prostota** – wprowadzaj złożoność tylko wtedy, kiedy jest to konieczne
- # **Elastyczność** – uwaga: elastyczność to także złożoność (!), bądź elastyczny tylko tam, gdzie system tego wymaga



# Koszt wytwarzania oprogramowania

**koszt całkowity = koszt stworzenia + koszt utrzymania**

**koszt utrzymania = koszt zrozumienia + koszt zmiany  
+ koszt testowania + koszt wdrożenia**

# Programista rozpoczynając projekt stoi przed wyborem:

- Napisać kod szybko, ale **niedokładnie i nieelastycznie**
- Napisać taki kod, który jest **elastyczny i będzie go można swobodnie wykorzystać w przyszłości**

# Bardzo często wybierana jest pierwsza opcja, która gwarantuje szybszy czas realizacji projektu, jednak jakiekolwiek zmiany dokonywane w późniejszym terminie są **niezwykle trudne i czasochłonne**

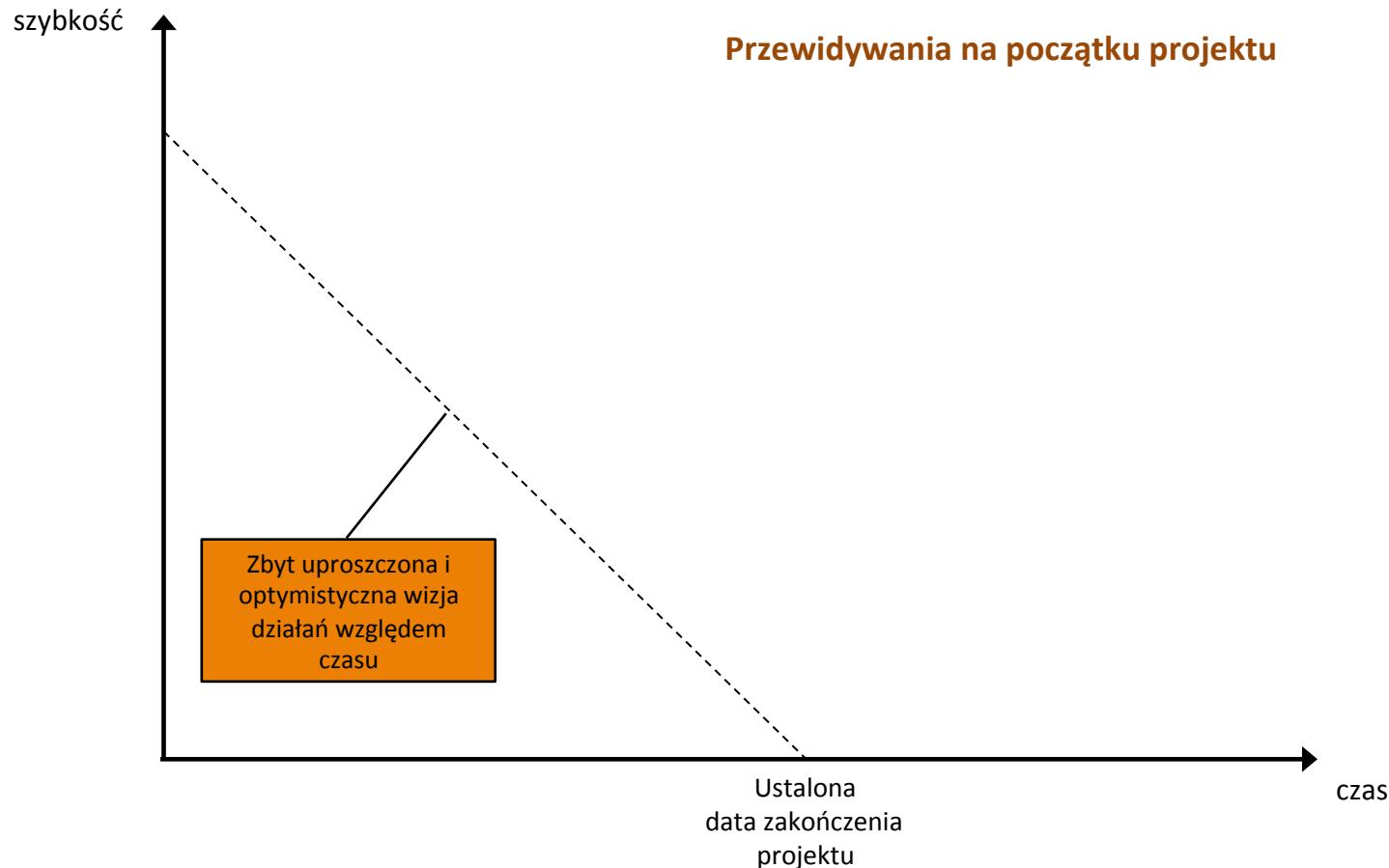


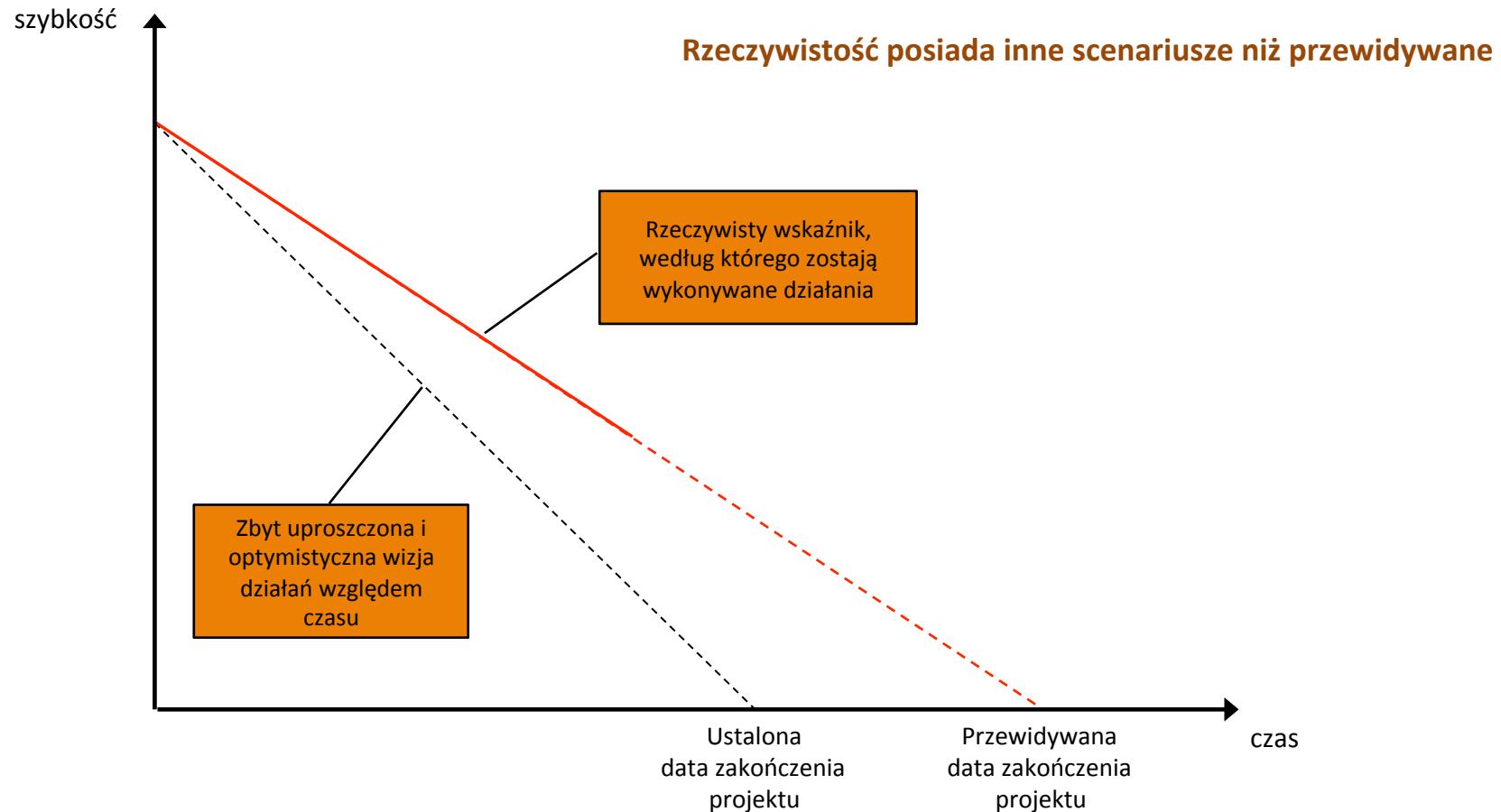
- # Dług techniczny jest metaforą opracowaną przez Warda Cunninghama
- # Metafora ta oparta jest na idei dłużu finansowego, który jest **korzystny jedynie w krótkim okresie czasu**:
  - Zaciągając pożyczkę otrzymujemy w szybki sposób dodatkowe środki finansowe
  - Do momentu, do którego nie spłacimy pożyczki w całości naliczane są odsetki
  - Szybsza spłata gwarantuje mniejsze straty



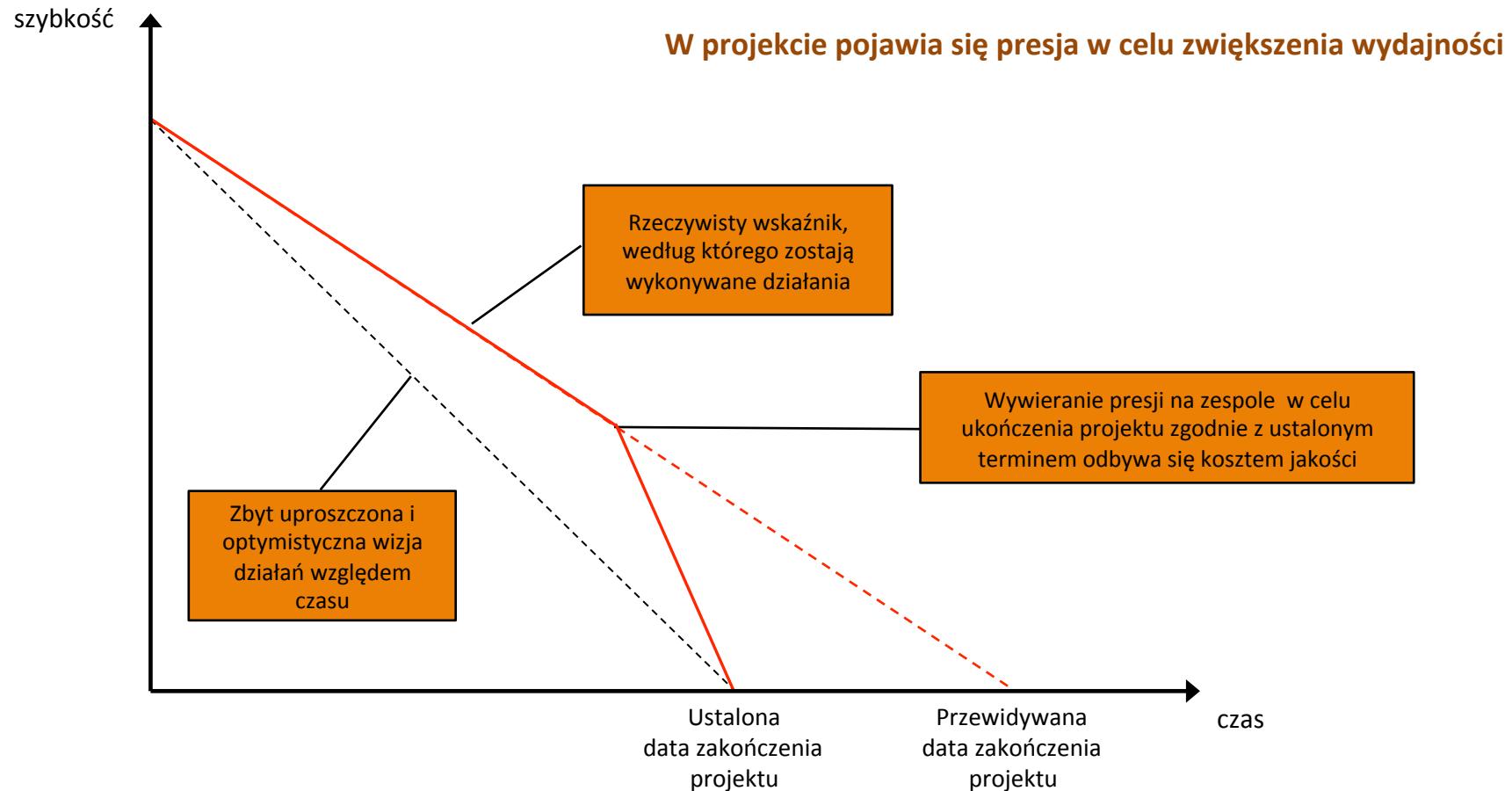
- # Dług finansowy znajduje swoje odzwierciedlenie w przypadku pisania kodu:
- # Pisanie kodu „na sztywno” pozwala na szybką realizację konkretnej funkcjonalności
- # Do momentu, do którego nie poprawimy w całości elastyczności dotychczasowego kodu, pojawiają się kolejne elementy wymagające przekształcenia



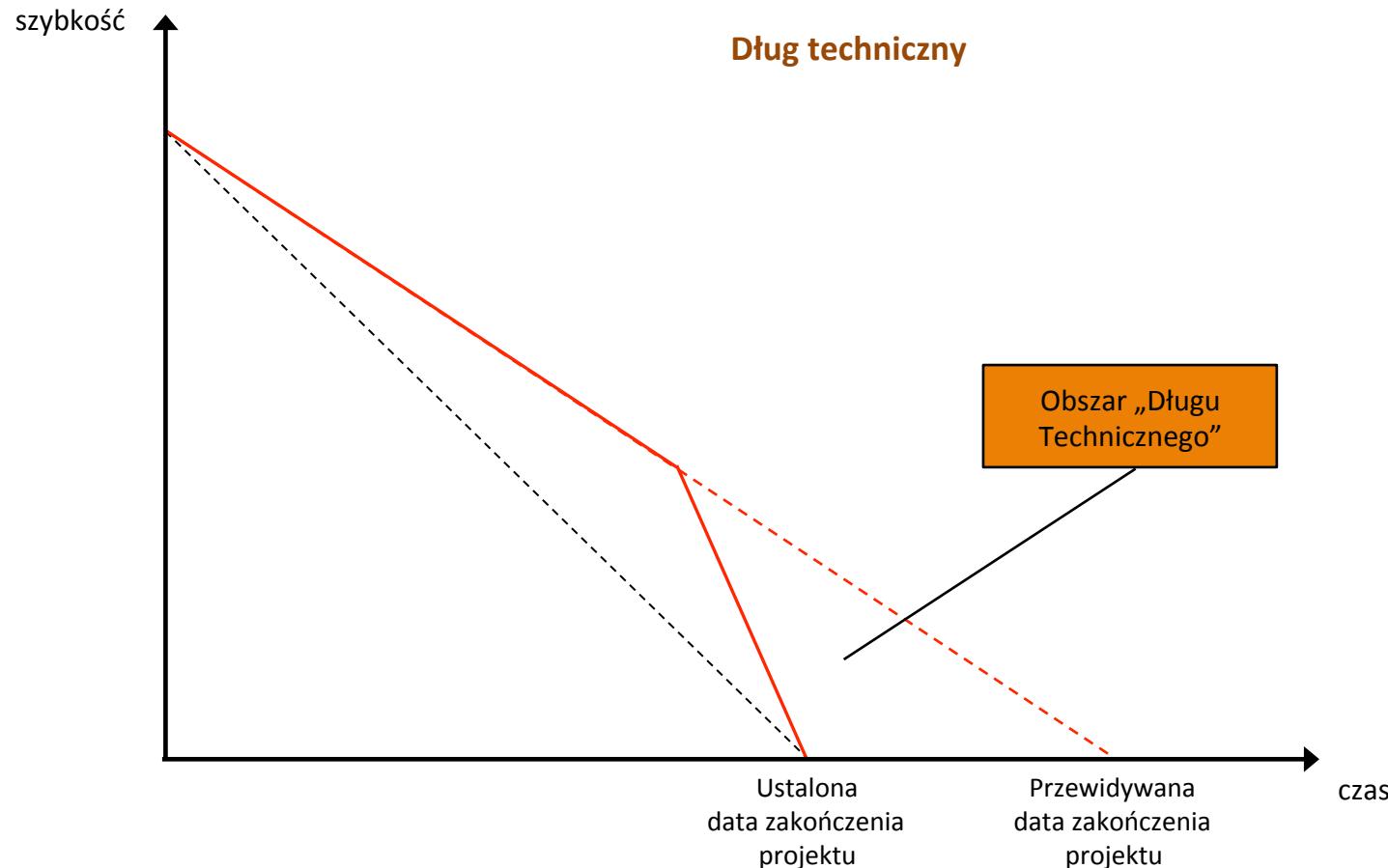




# Dług techniczny



# Dług techniczny



- # Najważniejsze jest **tworzenie elastycznego kodu** przez stosowanie **refaktoryzacji**
- # Należy **unikać** dłużu technicznego, jeśli jednak zachodzi konieczność jego zaciągnięcia:
  - W jak najkrótszym czasie należy dokonać refaktoryzacji kodu w celu przywrócenia elastyczności projektu
  - Uczynić dług widocznym
    - Zaalokuj zadania związane z refaktoryzacją kodu w ścieżce realizacji całości projektu



bns it } # Naturalny porządek refaktoryzacji  
Dlaczego refaktoryzacja

„Zmiana w wewnętrznej strukturze oprogramowania, która zwiększa jego przejrzystość i ułatwia wprowadzanie dalszych zmian, ale nie zmienia obserwowanych zachowań”

Fowler Martin, Refactoring: *Improving the Design of Existing Code*, 2000

Identyfikuję pewne cechy kodu źródłowego  
świadadczące o złym sposobie implementacji  
Są sygnałem do refaktoryzacji

- # *Duplicated Code* – identyczne fragmenty kodu w różnych miejscach projektu
- # *Long Method* – długie skomplikowane i trudne w zrozumieniu metody
- # *Conditional Complexity* – skomplikowane wyrażenia warunkowe
- # *Primitive Obsession* – nadmierne użycie typów prymitywnych
- # *Indecent Exposure* – metody lub klasy, które nie powinny być widoczne dla klientów stają się dostępne
- # *Solution Sprawl* – kod pewnego rozwiązania jest rozrzucony po wielu klasach
- # *Large Class* – klasa mająca zbyt duży zakres odpowiedzialności

# Zakresy refaktoryzacji

## Czytelność kodu

- Zmiana nazw klas, metod, zmiennych
- Wyodrębnianie metod
- Wprowadzenie konwencji kodowania
- Uproszczenie wyrażeń i sygnatur metod
- Łatwa do wprowadzenia
- Zajmuje niewiele czasu
- Programista może ją wprowadzać bez przekraczania szacowań dla zadań

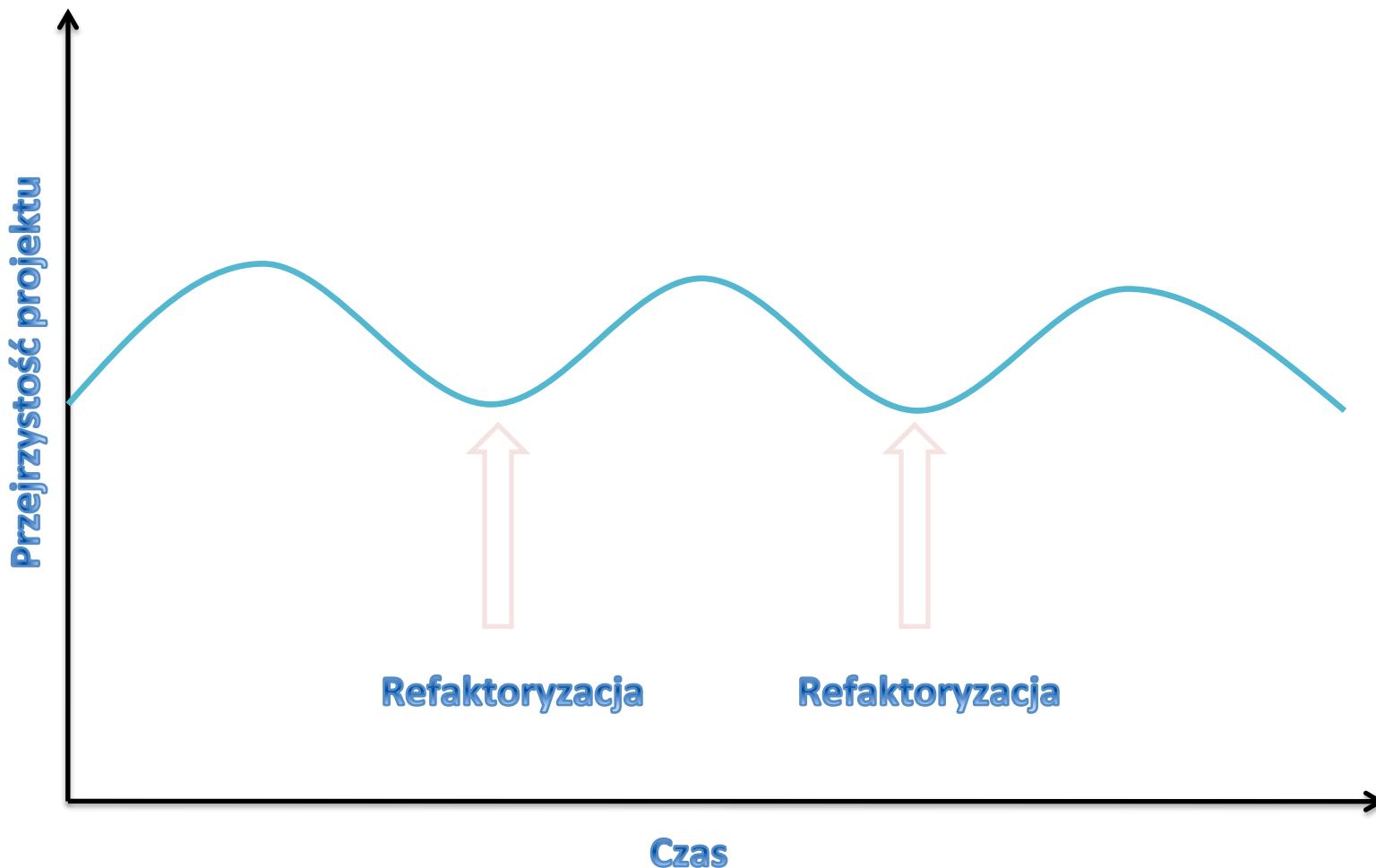
## Struktura kodu

- Wprowadzanie wzorców projektowych
- Redukcja powtórzeń
- Dość czasochłonna
- Z reguły prowadzi do dodatkowych narzutów czasowych
- Niekontrolowana skutkuje nadmiernie rozbudowanym kodem

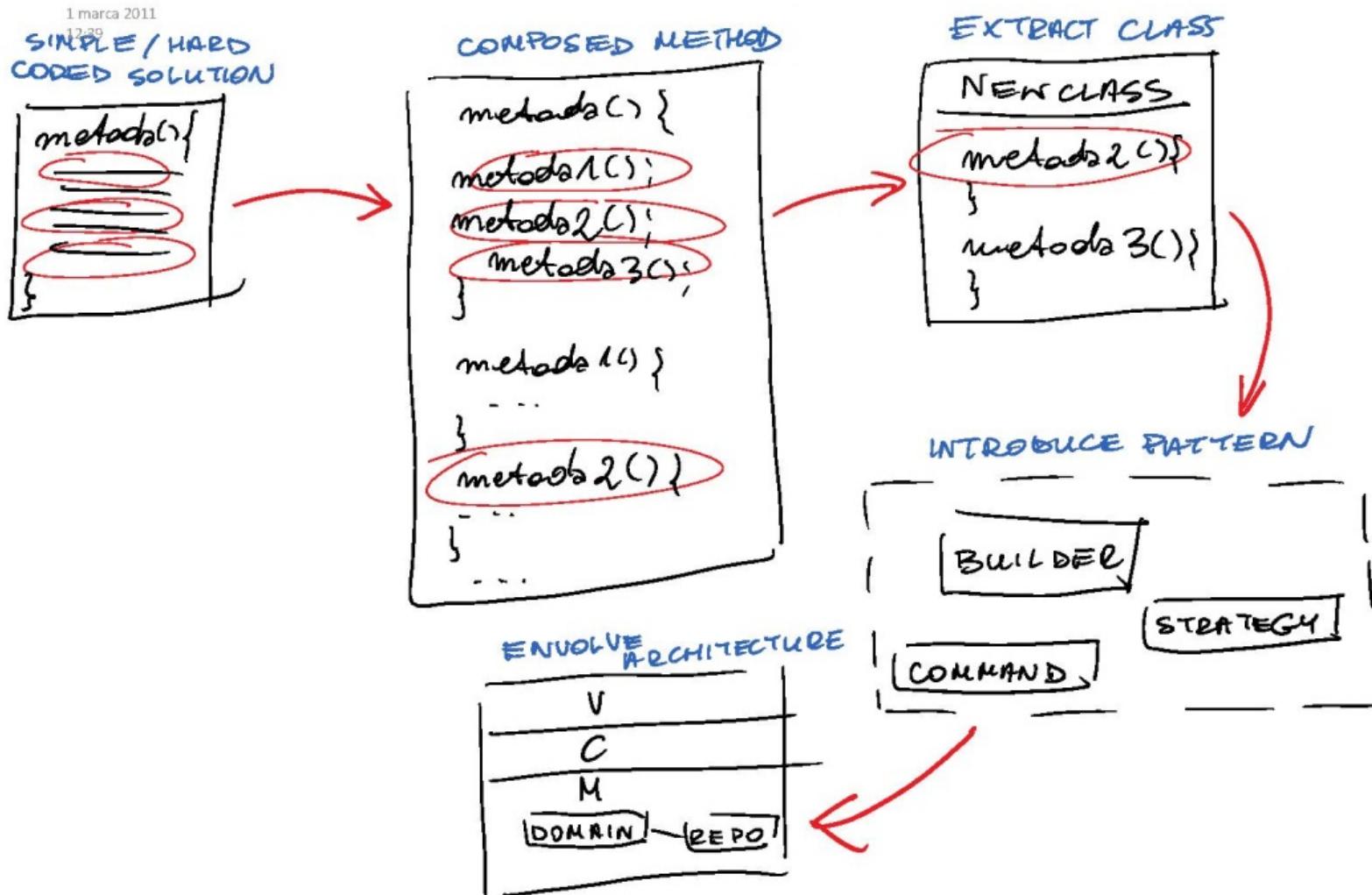
## Architektura systemu

- Wprowadzanie warstw
- Wprowadzenie lub zmiana O/RM
- Zmiana organizacji logiki biznesowej
- Wprowadzenie lub zmiana szkieletu aplikacji
- Bardzo czasochłonna
- Wymaga dogłębnej wiedzy o systemie

# Refaktoryzacja to ciągły proces



# Naturalny porządek refaktoryzacji



bns it } # Komponowanie metod  
Refaktoryzacja

1. Extract Method
2. Inline Method
3. Inline Temp
4. Replace Temp with Query
5. Introduce Explaining Variable

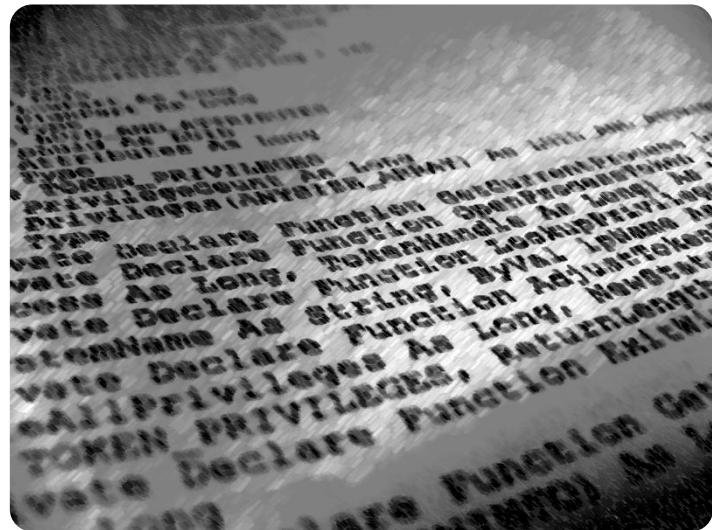


8. Split Temporary Variable
9. Remove Assignments to Parameters
10. Extract Method with the Method Object
11. Substitute Algorithm



bns it }    **#}** Extract method  
(Wyodrębnij metodę)

# Wyłączamy fragment kodu i tworzymy z niego metodę o nazwie która wyjaśnia jej działanie.



# Extract Method

## [implementacja]

```
public void printPerson(int age) {  
    printBanner();  
    //Wyświetla szczegóły  
    System.out.println("name: " + name);  
    System.out.println("age: " + age);  
}
```



```
public void printDetails(int age){  
    System.out.println("name: " + name);  
    System.out.println("age: " + age);  
}  
  
public void printPerson(int age){  
    printBanner();  
    printDetails(age);  
}
```

- # Dla długich metod z komentarzami.
- # Zamiast komentarza wprowadź metodę.



- # Tworzymy metodę której nazwa **odzwierciedla jej działanie.**
- # Ważne jest **co metoda robi**, a nie jak robi.
- # Jeśli **nie możesz wymyśleć nazwy metody** - nie wyodrębniaj jej.

- # Skopiuj kod z metody źródłowej do metody docelowej.
- # Zmienne lokalne ze skopiowanego kodu stają się zmiennymi lokalnymi lub parametrami nowej metody.

- # Zadeklaruj zmienne jeśli są używane **tylko w nowej metodzie**.
- # Jeśli nowa metoda modyfikuje zmienne ze źródła, to spróbujmy potraktować ją jako zapytanie i **przypisać do zmiennej**.

- # **Wymień** wyodrębniony kod w metodzie źródła na wywołanie metody docelowej.
- # **Usuń zmienne ze źródła** które są zadeklarowane w nowej metodzie [T].

bns it }    # }    **Inline Method**  
(Metoda w Linii)

# Odwrotność Extract Method.

# Wywołanie metody zastępujemy ciałem tej metody.



```
public boolean moreThanSevenLateDeliveries() {  
    return (numberOfLateDeliveries > 7);  
}  
  
public int GetRating() {  
    return (moreThanSevenLateDeliveries()) ? 2 : 1;  
}
```



```
public int getRating() {  
    return (numberOfLateDeliveries > 7) ? 2 : 1;  
}
```

# W celu zwiększenia  
przejrzystości  
i czytelniejszości kodu.



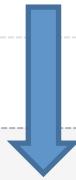
- # Upewniamy się, że metoda nie jest polimorficzna.
- # Znajdujemy wszystkie **odwołania** do metody.
- # Zamieniamy wszystkie odwołania na **ciało metody [T]**.
- # **Usuwamy definicję** metody.

bns it }    **#}** **Inline temp**  
**(zmienna w Linii)**

# Zamieniamy wszystkie referencje do zmiennej tymczasowej jej wartości.



```
double baseCost = anOrder.baseCost();  
  
return (baseCost > 1000);
```



```
return (anOrder.baseCost() > 1000);
```

# Jako część  
refaktoryzacji *Replace  
Temp with Query*.

# Do zmiennej, która  
powstała w efekcie  
zastosowania *Extract  
Method*.



- # Deklarujemy zmienną jako **final**.
- # Zastępujemy wszystkie jej odniesienia [T].
- # Usuwamy zmienną [T].

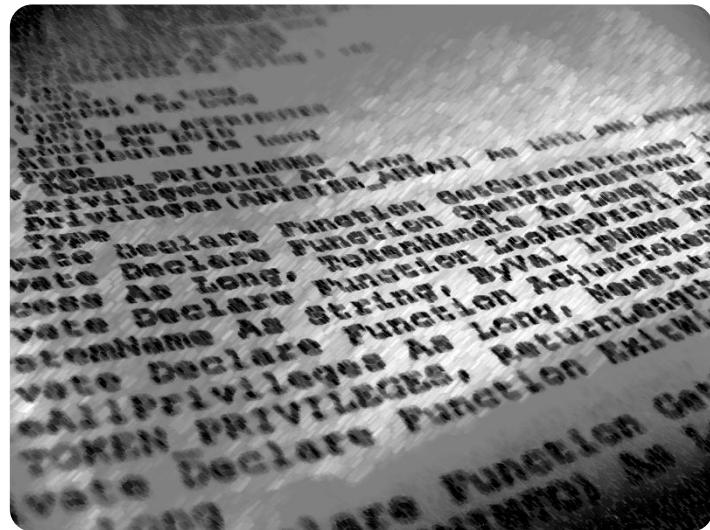


# Replace Temp with Query

(zamień zmienną z wyrażeniem)

# Wyodrębniamy wyrażenie w formie metody

# Wymieniamy odniesienia do zmiennej na wywołanie metody.



```
double baseCost = quantity * itemCost;

if (baseCost > 1000){
    return baseCost * 0.95;
}
else{
    return baseCost * 0.98;
}
```



```
private double baseCost(){
    return quantity * itemCost;
}

if ( baseCost() > 1000 ){
    return baseCost() * 0.95;
}
else{
    return baseCost() * 0.98;
}
```

# Używamy metody dostępnej w całej klasie zamiast zmiennej.

# Replace Temp with Query jest często krokiem przed Extract Method.



- # Zmienne, które są **przypisane tylko raz** deklarujemy jako const [T].
- # Jeśli występuje przypisanie więcej niż raz, to korzystamy z **Split Temporary Variable**.

- # Wyrażenie umieszczamy w metodzie.
- # Upewniamy się, że nowa metoda np. **nie modyfikuje** obiektów.
- # Jeśli modyfikuje, to stosujemy **Separate Query from Modifier [T]**.

- # Usuwamy zmienną, która nie jest wykorzystywana.
- # Używamy **Replace Temp with Query** na innych zmiennych w podobny sposób.



# Introduce Explaining Variable

(Wprowadzenie zmiennej wyjaśniającej)

# Wyrażenie umieszczamy w zmiennej o nazwie która wyjaśnia działanie (cel) tego wyrażenia.



# Introduce Explaining Variable [implementacja]

```
double price() {  
    // price is base price - quantity discount + shipping  
    return quantity * basePrice -  
        Math.max(0, quantity - 500) * basePrice * 0.05 +  
        Math.min(quantity * basePrice * 0.1, 100.0);  
}
```



```
double price() {  
    final double basePrice = quantity * itemPrice;  
    final double quantityDiscount =  
        Math.max(0, quantity - 500) * itemPrice * 0.05;  
    final double shipping = Math.min(basePrice * 0.1, 100.0);  
    return basePrice - quantityDiscount + shipping;  
}
```

# Gdy wyrażenia są  
**skomplikowane i trudne**  
do odczytania.

# Stosujemy dobrze  
nazwaną zmienną w celu  
jasności i prostoty zapisu.



- # Deklarujemy zmienną, której nazwa powinna wyrażać jej znaczenie.
- # Do zmiennej przypisujemy wyrażenie będące częścią jakiegoś większego wyrażenia.

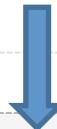
- # Zamiast wyrażenia **podstawiamy otrzymaną zmienną [T].**
- # **Powtarzamy** tę czynność dla innych części większego wyrażenia.

bns it }    **Split temporary Variable**  
[Podziel na Zmienne]

# Tworzymy osobną zmienną dla każdego zadania, które na niej mamy wykonać.



```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```



```
final double areaOfCity = 2 * (height + width);  
System.out.println(areaOfCity);  
final double areaOfVillage = height * width;  
System.out.println(areaOfVillage);
```

Gdy jednej zmiennej przypisujemy więcej niż jedno zadanie.

Korzystanie z jednej zmiennej dla dwóch różnych zadań jest bardzo mylące dla czytelnika.



- # Jeśli zmienna jest np. inkrementowana,  
wtedy nie rozdzielaj zmiennych.
- # Zmieniamy nazwę zmiennej w deklaracji i  
przy pierwszym jej użyciu.
- # Deklarujemy nową zmienną jako **const**.

- # Zmień **wszystkie odniesienia** do zmiennej aż do drugiego zadania.
- # **Zadeklaruj nową zmienną** dla drugiego zadania i postępuj tak jak poprzednio [T].



# Remove Assignments to Parameters

(Usuń przypisanie parametrów do zadań)

Zastępujemy przypisania do parametrów zmienną tymczasową.

Tworzymy osobną zmienną zamiast używać referencji podanej jako parametr.



# Remove Assignments to Parameters

## [Implementacja]

```
public int discount(int inputVal, int quantity) {  
    if (inputVal > 50) inputVal -= 2;  
    return inputVal;  
}
```



```
public int discount(int inputVal, int quantity) {  
    int result = inputVal;  
    if (inputVal > 50)  
        result -= 2;  
    return result;  
}
```

Aby mieć pewność że odwołujemy się przez wartość a nie przez referencje.



- # Tworzymy tymczasową zmienną dla parametru.
- # Zamieniamy wszystkie odniesienia do parametrów na odniesienia do zmiennej [T].

bns it } **Replace Method with Method Object**  
(Zamień metodę obiektem)

Jeśli metoda jest zbyt złożona, aby użyć wyodrębnienia metody, stwórz nowy obiekt.

Tworzymy osobny obiekt, którego metody zastąpią naszą metodę.



# Replace Method with Method Object

## [Implementacja]

```
public class Account {  
    // (...)  
    public void makeTransfer(double amount, Account destinationAccount) {  
        double transferFee = 1;  
        if (amount > 1000){  
            transferFee = 1 + amount * 0.0001;  
        }  
  
        connectToElixir();  
  
        // (...)  
        // (...)  
        // (...)  
  
        debit(amount + transferFee);  
    }  
}
```



# Replace Method with Method Object

## [Implementacja]

```
class TransferManager {  
  
    private final Account account;  
    private double amount;  
    private Account destinationAccount;  
    private double transferFee;  
  
    public TransferMaker( Account source, double amount, Account destination ){  
        // initialization  
    }  
  
    public void make(){  
        if (amount > 1000){  
            transferFee = 1 + amount * 0.0001;  
        }  
  
        account.connectToElixir();  
        //...  
        account.debit(amount + transferFee);  
    }  
}
```

```
class Account {  
    // (...)  
    public void makeTransfer(double amount,  
                            Account destinationAccount){  
        new TransferMaker(this, amount,  
                          destinationAccount).make();  
    }  
}
```

Aby móc zrefaktoryzować metodę, która początkowo jest do tego nieodpowiednia.



1. Utwórz klasę o nazwie metody pochodzącej od metody
2. Zmienne lokalne i parametry metody funkcji umieść jako pola prywatne nowej klasy
3. Referencję do pierwotnej klasy przekaż poprzez konstruktor i przypisz w prywatnego pola
4. Dodaj metodę **compute** i przenieś do niej zawartość metody źródłowej
5. Zamień ciało metody źródłowej na stworzenie nowego obiektu i wywołanie **compute**

bns it }    #} Substitute Algorithm  
(zastąpić Algorytm)

Podmieniamy ciało  
metody na kod  
zawierający **nowy**  
**algorytm**.



# Substitute Algorithm

## [Implementacja]

```
Person findPerson(Person[] people, String name1,
                  String name2, String name3, String name4){
    for (int i = 0; i < people.length; i++) {
        if (people[i].getName().equals(name1)) {
            return people[i];
        }
        if (people[i].getName().equals(name2)) {
            return people[i];
        }
        if (people[i].getName().equals(name3)) {
            return people[i];
        }
        if (people[i].getName().equals(name4)) {
            return people[i];
        }
    }
    throw new PersonNotFoundException();
}
```



```
Person findPerson(Person[] people, String name1, String name2,
                  String name3, String name4){
    List candidates =
        Arrays.asList(new String[] {name1, name2, name3, name4});
    for (int i = 0; i < people.length; i++) {
        if (candidates.contains(people[i].getName()))
            return people[i];
    }
    throw new PersonNotFoundException();
}
```

W przypadku gdy chcemy wymienić algorytm na taki, który będzie prostszy.



- # Przygotuj alternatywny algorytm.
- # Uruchom algorytm w ramach testów.
- # Jeśli wyniki są takie same to koniec.
- # Jeśli wyniki nie są takie same, użyj starego algorytmu dla porównania podczas testowania.

bns it } # Przenoszenie kodu  
Moving Features

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man
7. Introduce Foreign Method
8. Introduce Local Extension



bns it } # Move Method  
Przeniesienie Metody

**Przeniesienie metody  
do klasy bardziej  
z nią związanego.**



```
public class Account{
    //(...)

    public double overdraftCharge(){

        if (type.isPremium()) {
            double result = 10;

            if (daysOverdrawn > 7){
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        }
        else return daysOverdrawn * 1.75;
    }

    public double bankCharge(){
        //...
    }

    private AccountType type;
    private int daysOverdrawn;
}
```

```
public class AccountType{

    //(...)

    public boolean isPremium(){
        return true;
    }
}
```

```
public class Account{
    //...
    public double bankCharge(){
        double result = 4.5;
        if (daysOverdrawn > 0){
            result += type.overdraftCharge(daysOverdrawn);
        }
        return result;
    }
}

public class AccountType{
    //...
    public double overdraftCharge(Account account) {
        if (isPremium()){
            double result = 10;
            if (account.getDaysOverdrawn() > 7){
                result += (account.getDaysOverdrawn() - 7) * 0.85;
            }
            return result;
        }
        else return account.getDaysOverdrawn() * 1.75;
    }
}
```

- Jeśli metoda wymusza **zbyt duże powiązanie** między klasami.
- Gdy występują metody, które mają **mało wspólnego** ze swoją klasą.
- **Przeniesienie metod upraszcza klasy** i wpływa na poprawienie hermetyzacji.



- Jeśli istnieją **wykorzystywane metody lub pola** w klasie rozważ ich przeniesienie wraz z metodą.
- Sprawdź czy metoda występuje także w **podklasach lub nadklasach**.
- Zadeklaruj metodę **w nowej klasie**.
- **Przenieś ciało** metody do nowej klasy i dopasuj jej działanie do nowej klasy **[C]**.



- Określ **sposób odwoływania się** do docelowego obiektu z klasy źródłowej.
- Zamień metodę źródłową **na delegującą [T]**.
- **Zachowaj** metodę w formie delegującej

```
old() { newOb.NewMethod(); }
```
- lub **usuń ją**, zamieniając jej **wywołania** wywołaniami nowej metody [T].



bns it } # Move Field  
Przeniesienie Pola

**Przeniesienie pola  
do klasy będącej bardziej  
z nim związaną.**



```
public class Account{
    //...
    private AccountType type;
    private double interestRate;
    public double interestForAmount_days(double amount, int days) {
        return interestRate * amount * days / 365;
    }
}

public class AccountType{
    //...
}
```

## Move Field [Implementacja]

```
public class AccountType{
    //...
    private double interestRate;

    public double getInterestRate(){
        return interestRate;
    }

    public void setInterestRate(double interestRate){
        this.interestRate = interestRate;
    }
}

public class Account{
    //...
    private AccountType type;

    public double interestForAmount_days(double amount, int days){
        return type.getInterestRate() * amount * days / 365;
    }
}
```

- **Jeśli pole danej klasy jest używane częściej przez inną klasę niż własna.**
- **Metody działające na tym polu też warto przenieść.**



- Jeśli pole jest **publiczne**, poddaj je **hermetyzacji [T]**.
- Zadeklaruj **pole**, wraz z metodami **set i get lub właściwość** w drugiej klasie **[C]**.

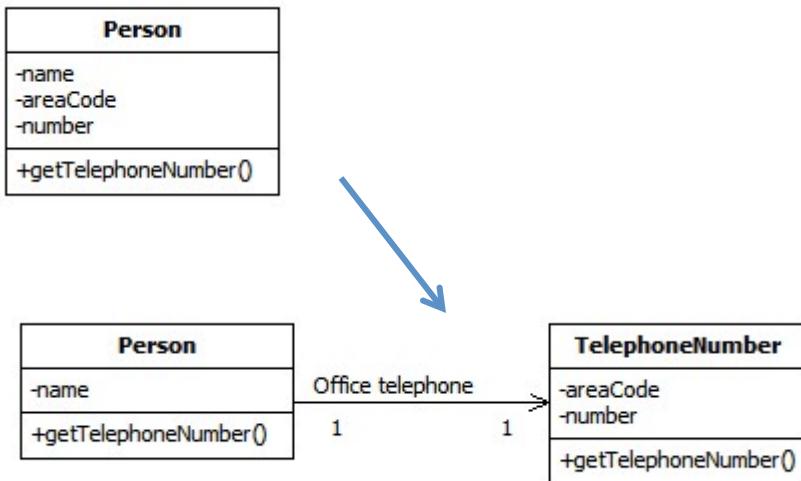


- Określ sposób odwoływania się do docelowego obiektu.
- Usuń pole z klasy pierwotnej.
- Zastąp wszystkie stare odwołania nowymi [T].



bns it }    **# }** Extract class  
Wydzielenie Klasy

**Wydzielenie części zbyt  
dużej klasy w postaci  
nowej klasy.**



# Extract Class

## [Implementacja]

```
public class Person
{
    private String name;
    private String officeAreaCode;
    private String officeNumber;

    //(...)

    public String getTelephoneNumber() {
        return "(" + officeAreaCode + ") " + officeNumber);
    }
}
```

# Extract Class

## [Implementacja]

```
public class Person
{
    private String name;
    private TelephoneNumber officeTelephone
        = new TelephoneNumber();

    //...
    public String getTelephoneNumber() {
        return officeTelephone.getTelephoneNumber();
    }
    public TelephoneNumber getOfficeTelephone() {
        return officeTelephone;
    }
}
```

```
public class TelephoneNumber
{
    private String number;
    private String areaCode;

    //...
    public String getTelephoneNumber() {
        return "(" + areaCode
            + ") " + number);
    }
    public String getNumber() {
        return number;
    }
}
```

- Gdy klasy są **zbyt rozbudowane**.
- Gdy klasy są **trudne do zrozumienia**.
- Gdy klasy mają **zbyt wiele odpowiedzialności**.



- Ustal jak **podzielić odpowiedzialność** klasy.
- Stwórz **nową klasę**.
- Utwórz **relację** między stara i nową klasą.

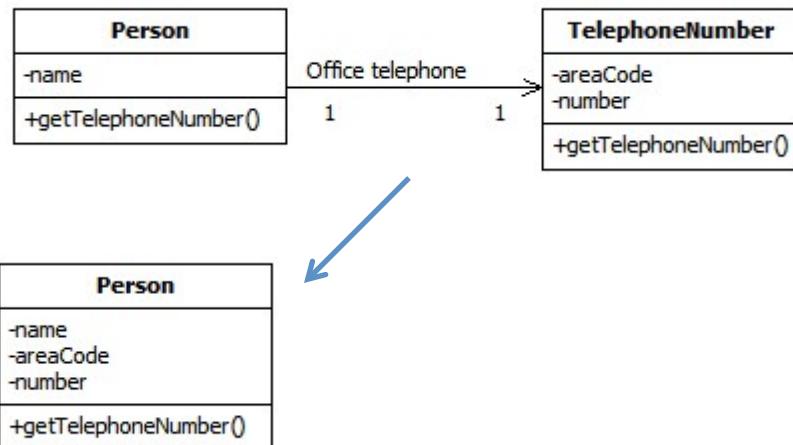


- Przenieś pola używając **Move Field [T]**.
- Zaczynając od najprostszych metod zastosuj **Move Method [T]**.
- Usuń zbędny kod.



bns it } **Inline Class**  
Rozwinięcie Klasy

**Wchłonięcie mało funkcjonalnej klasy przez klasę, która najwięcej z niej korzysta.**



# Inline Class [Implementacja]

```
public class Person
{
    private String name;
    private TelephoneNumber officeTelephone
        = new TelephoneNumber();

    //...
    public String getTelephoneNumber(){
        return officeTelephone.getTelephoneNumber();
    }
    public TelephoneNumber getOfficeTelephone() {
        return officeTelephone;
    }
}
```

```
public class TelephoneNumber
{
    private String number;
    private String areaCode;

    //...
    public String getTelephoneNumber(){
        return "(" + areaCode
            + ") " + number);
    }
    public String getNumber() {
        return number;
    }
}
```

# Inline Class

## [Implementacja]

```
public class Person
{
    private String name;
    private String officeAreaCode;
    private String officeNumber;

    //(...)

    public String getTelephoneNumber()
    {
        return "(" + officeAreaCode + ") " + officeNumber;
    }
}
```

- Stosowane w przypadku **mało samodzielnych klas** – występujących w powiązaniu z inną klasą.
- Wchłaniana przez tę klasę, która w **najczęściej** jej używa.
- **Przeciwieństwo wydzielenia klasy.**

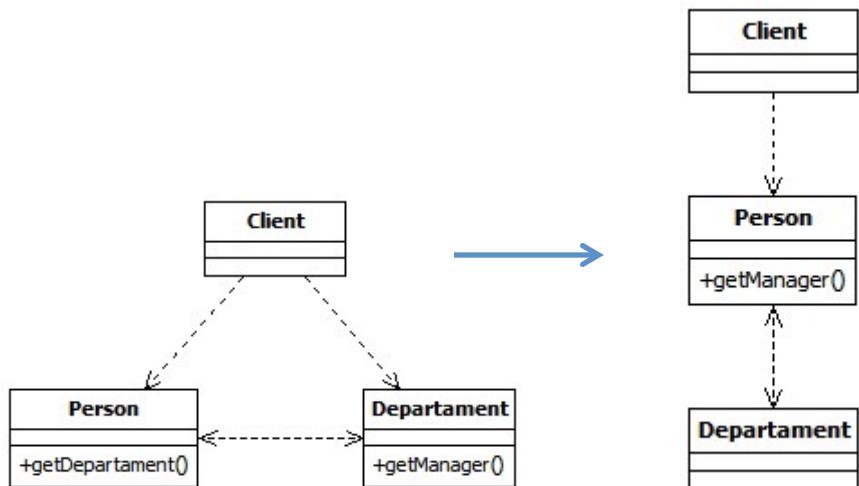


- **Utwórz metody rozwijanej klasy w klasie docelowej.**
- **Zastąp referencje do klasy źródłowej referencjami do klasy docelowej [T].**
- **Przenieś pola i metody z klasy źródłowej do docelowej.**



bns it } **#**} Hide Delegate  
Ukrycie Delegowania

# Stworzenie metody delegującej, która ukryje delegata.



#}

## Hide Delegate [Implementacja]

```
public class Person
{
    //...
    private Department department;

    public Department getDepartment() {
        return department;
    }
    public void setDepartment(Department arg) {
        department = arg;
    }
}

public class Department
{
    //...
    private Person manager;
    public Department (Person manager) {
        this.manager = manager;
    }
    public Person getManager() {
        return manager;
    }
}

manager = john.getDepartment().getManager();
```

```
public class Person
{
    //...
    private Department department;
    public Person getManager()
    {
        return
            department.getManager();
    }
}

manager = john.getManager();
```

- Stosowane aby **ograniczyć konsekwencje zmian w klasach.**
- Zmniejsza **złożoność** wywoływania metod.
- Poprawia **hermetyzację.**

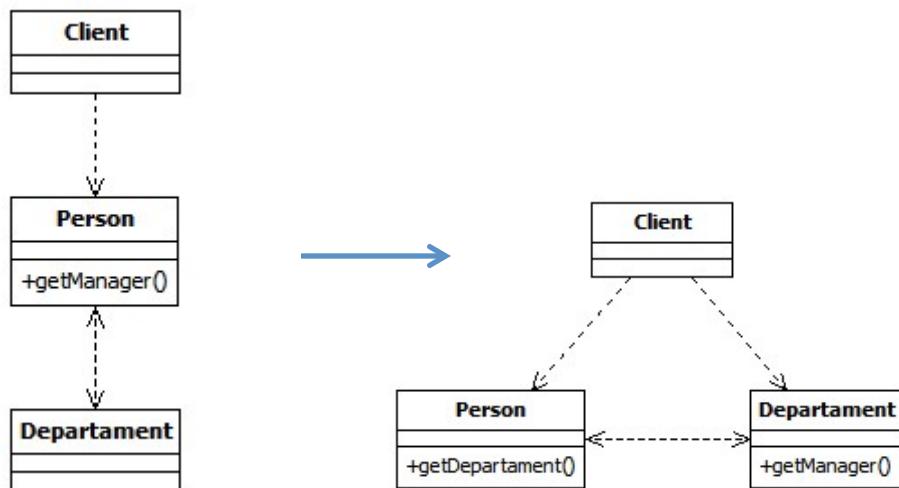


- Dla metod klasy delegata **utwórz metody delegujące** w klasie fasadowej.
- **Zmień** kod klienta, aby wywoływał metody delegujące [T].
- **Usuń** już **niepotrzebne** metody udostępniające delegata[T].



bns it } Remove Middle Man  
Usunięcie Pośrednika

## Usunięcie klasy zawierającej metody delegujące.



# Remove Middle Man

## [Implementacja]

```
public class Person
{
    //...
    private Department department;
    public Person getManager()
    {
        return
            department.getManager();
    }
}

public class Department
{
    //...
    private Person manager;
    public Person getManager() {
        return manager;
    }
}

manager = john.getManager();
```

```
public class Person
{
    //...
    private Department department;

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department arg) {
        department = arg;
    }
}

manager = john.getDepartment().getManager();
```

- Stosowane gdy klasa delegująca ogranicza się do bycia **jedynie pośrednikiem**.
- **Przeciwieństwo ukrycia delegowania.**



- Utwórz metodę zwracającą referencję do delegata.
- Zamień wywołania starych metod na nowe [T].
- Usuń niepotrzebne już metody [T].



bns it } **Introduce Foreign Method**  
Utworzenie Metody Obcej

**Utworzenie metody,  
która jest obca i powinna  
znajdować się w innej klasie.**



# Introduce Foreign Method [Implementacja]

```
double nettoWithNewTax =  
    new Money(oldValue.getBrutto(), oldValue.getTax() + 1).getNetto();
```

```
double valueWithNewTax = countNewTax(oldValue);  
  
private static double countNewTax(Money oldValue) {  
    // metoda obca,  
    // powinna znajdować się w klasie Money  
    return new Money(oldValue.getBrutto(), oldValue.getTax()  
+ 1).getNetto();  
}
```

- Stosowane gdy **chcemy** utworzyć **nową** metodę.
- Gdy **nie możemy** umieścić jej w odpowiedniej dla niej klasie.



- Utwórz pożądaną metodę **w klasie klienta**.
- Jako jej parametr **przekaż referencję do obiektu klasy serwera**.
- Do metody **dodaj komentarz //metoda obca, powinna znajdować się w klasie NazwaKlasy**.

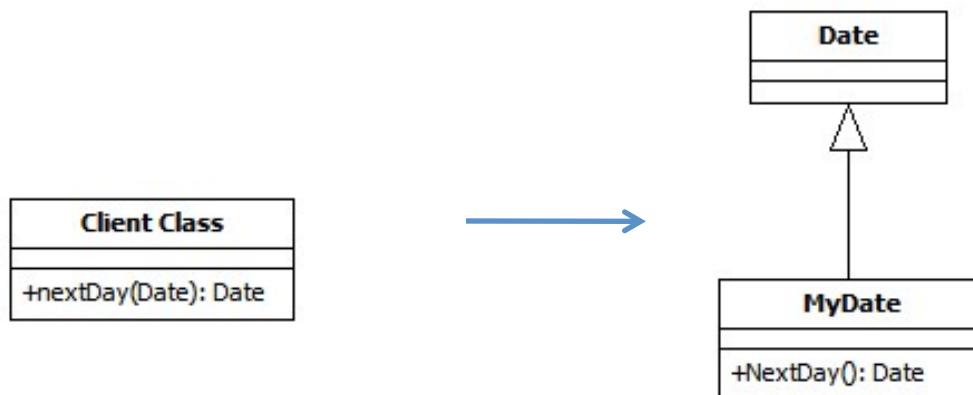




# Introduce Local Extension

Utworzenie Rozszerzenia Lokalnego

**Utworzenie podklasy lub opakowania dla klasy, której nie można modyfikować.**



# Introduce Local Extension [Implementacja]

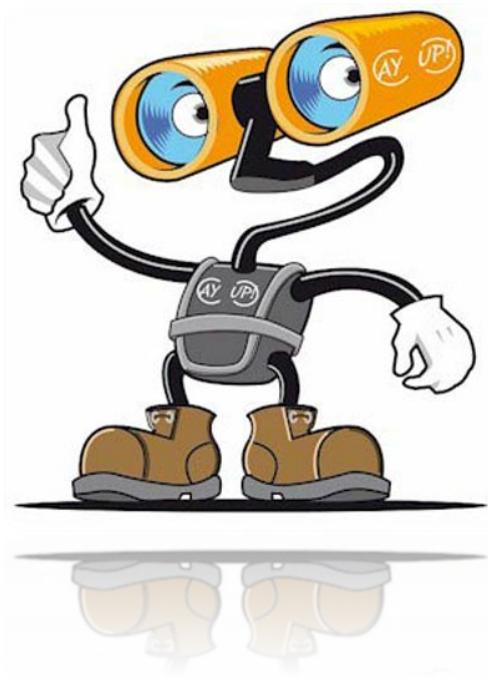
```
public static double countNewTax(Money oldValue) {
    // metoda obca,
    // powinna znajdować się w klasie Money
    return new Money(oldValue.getBrutto(), oldValue.getTax()+1).getNetto();
}
```

```
public class MyMoney extends Money{

    public MyMoney (double nettoValue) {
        super(nettoValue);
    }

    public double countNewTax(Money oldValue) {
        return new Money(oldValue.getBrutto(),
                        oldValue.getTax()+1).getNetto();
    }
}
```

- Stosowane **aby uporządkować metody obce dla danej klasy.**
- Daje możliwość **przedefiniowania lub przeciążania metod.**



- Utwórz **podklasę lub opakowanie klasy**.
- Dodaj **konstruktor konwertujący**.
- Dodaj potrzebne **metody**.
- Dokonaj pożądanych **zmian**.
- Przenieś wszystkie **metody obce** rozszerzanej klasy.



bns it } #} Upraszczanie wyrażeń warunkowych  
Simplifying Conditional Expressions

1. Decompose Conditional
2. Consolidate Conditional Expression
3. Consolidate Duplicate Conditional Fragments
4. Remove Control Flag
5. Replace Nested Conditional with Guard Clauses
6. Replace Conditional with Polymorphism
7. Introduce Null Object
8. Introduce Assertion



bns it }    #} Decompose Conditional  
              Podział Wyrażenia Warunkowego

**Uproszczenie rozbudowanej instrukcji warunkowej poprzez zastosowanie wywołań metod.**



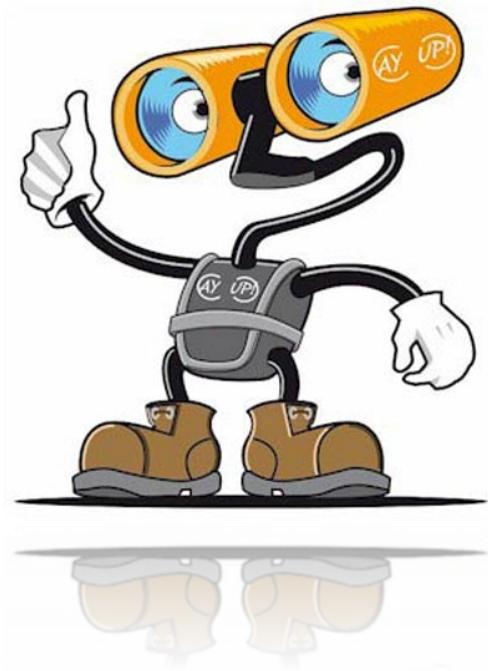
```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * winterRate + winterServiceCharge;
else charge = quantity * summerRate;
```



```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

```
private boolean notSummer(Date date) {
    return date.before(SUMMER_START) || date.after(SUMMER_END);
}
private double summerCharge(int quantity) {
    return quantity * summerRate;
}
private double winterCharge(int quantity) {
    return quantity * winterRate + winterServiceCharge;
}
```

- Stosowane dla **zbyt rozbudowanych** wyrażeń warunkowych.
- Pozwala **jasno wyrazić** intencje programisty.



- Wydziel **wyrażenie warunkowe** w odrębna metodę.
- Wydziel **obie gałęzie instrukcji if** w odrębne metody.





# Consolidate Conditional Expression

## Połączenie Wyrażeń Warunkowych

**Połączenie kilku wyrażeń warunkowych, których spełnienie daje taki sam wynik, w jedno wyrażenie.**



# Consolidate Conditional Expression [Implementacja]

```
if (onVacation()) {  
    if (lengthOfService() > 10) {  
        return 1;  
    }  
}  
return 0.5;
```



```
if (onVacation() && lengthOfService() > 10) {  
    return 1;  
}  
else {  
    return 0.5;  
}
```

- Stosowane gdy spełnienie **każdego** z wyrażeń warunkowych da **taki sam skutek**.
- Poprawia czytelność kodu.



- Warunki **nie mogą** mieć efektów ubocznych.
- Połącz ciąg warunków w jeden warunek [T].
- Rozważ **wydzielenie metody** z kodu połączzonego wyrażenia.

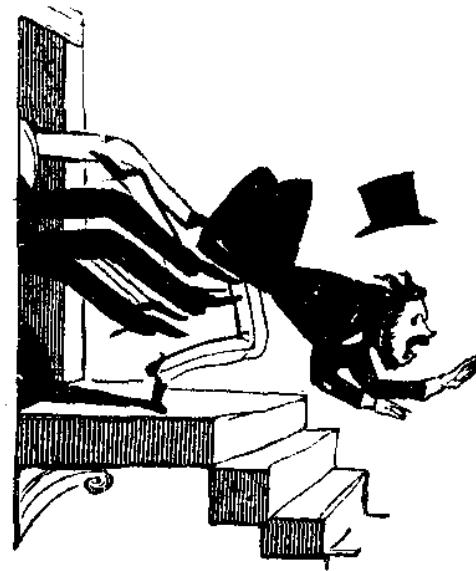


# Consolidate Duplicate Conditional Fragments

Połączenie Powielonych Fragmentów Wyrażeń Warunkowych

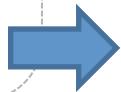


**Wyłączenie wspólnych  
fragmentów kodu z gałęzi  
wyrażenia warunkowego  
poza wyrażenie.**



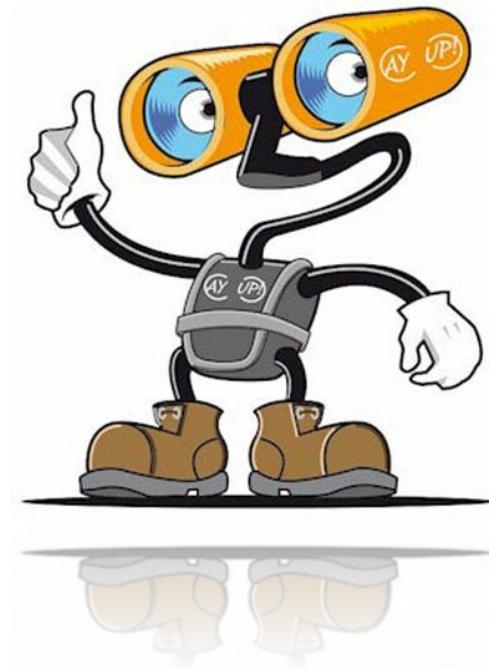
# Consolidate Duplicate Conditional Fragments [Implementacja]

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
  
else {  
    total = price * 0.98;  
    send();  
}
```



```
if (isSpecialDeal()) {  
    total = price * 0.95;  
}  
  
else {  
    total = price * 0.98;  
}  
  
send();
```

- Stosowane gdy **gałęzie** wyrażenia warunkowego **zawierają ten sam kod**.
- Pomaga **zrozumieć różnice** pomiędzy gałęziami.



- **Znajdź wspólny kod.**
- **Przenieś go przed lub za instrukcję warunkową.**
- Jeśli wydzielony kod zawiera więcej niż jedną instrukcję, zastosuj *wydzielenie metody*.



bns it } Remove Control Flag  
Usunięcie Znacznika Kontrolnego

**Usunięcie zmiennej będącej znacznikiem kontrolnym (flagą).**



```
void checkSecurity(String[] persons) {  
    boolean found = false;  
    for (int i = 0; i < persons.length; i++) {  
        if (!found) {  
            if (persons[i].equals ("Don")){  
                sendAlert();  
                found = true;  
            }  
            if (persons[i].equals ("John")){  
                sendAlert();  
                found = true;  
            }  
        }  
    }  
}
```



```
void checkSecurity(String[] persons)  
{  
    for (int i = 0; i < persons.length; i++)  
    {  
        if (persons[i].equals("Don")){  
            sendAlert();  
            break;  
        }  
        if (persons[i].equals("John")){  
            sendAlert();  
            break;  
        }  
    }  
}
```

- Stosowane w celu **zamiany** znacznika kontrolnego **na** instrukcje **break** lub **return**.
- **Poprawia czytelność** wyrażenia warunkowego.



- Odszukaj znaczniki kontrolne.
- Zastąp przypisania im wartości instrukcjami **break** lub **continue [T]**.
- Usuń deklarację znacznika i pozostałe referencje.





# Replace Nested Conditional with Guard Clauses

Zastąpienie Zagnieżdżonych Wyrażeń Warunkowych Przez Klauzule Dozorowane

**Obsługa rzadko spotykanych  
przypadków przez klauzule  
dozorowane zamiast  
konstrukcji `if-then-else`.**



## Replace Nested Conditional with Guard Clauses [Implementacja]

```
double getPayAmount() {  
    double result;  
    if (isDead) result = getdeadAmount();  
    else {  
        if (isSeparated) result = getSeparatedAmount();  
        else {  
            if (isRetired) {  
                result = getRetiredAmount();  
            }  
            else result = getNormalPayAmount();  
        }  
    }  
    return result;  
}  
  
public double getPayAmount() {  
    if (isDead) {  
        return getDeadAmount();  
    }  
    if (isSeparated) {  
        return getSeparatedAmount();  
    }  
    if (isRetired) {  
        return getRetiredAmount();  
    }  
    return getNormalPayAmount();  
}
```



- Stosowane aby **oddzielić** sytuacje **normalne** (**if-then-else**) od wyjątkowych (**if-return**).
- **Poprawia czytelność** wyrażenia warunkowego.



- **Wyszukaj warunki opisujące sytuacje wyjątkowe.**
- **Zamieniaj je kolejno na klauzule dozorowane [T].**



bns it } Replace Conditional With Polymorphism  
Zastąpienie Wyrażenia Warunkowego Przez Polimorfizm

Zastosowanie **polimorfizmu**  
do uproszczenia wyrażeń  
warunkowych **zależnych**  
od typu obiektu.



# Replace Conditional With Polymorphism

## [Implementacja(1)]

```
public class Employee{  
    public int payAmount() {  
        switch (getType()) {  
            case EmployeeType.ENGINEER:  
                return monthlySalary;  
            case EmployeeType.SALESMAN:  
                return monthlySalary + commission;  
            case EmployeeType.MANAGER:  
                return monthlySalary + bonus;  
            default:  
                throw new RuntimeException  
                    ("Incorrect Employee");  
        }  
    }  
    int getType() {  
        return type.getTypeCode();  
    }  
    private EmployeeType type;  
}
```

```
public abstract class EmployeeType{  
    public abstract int getTypeCode();  
}  
  
public class Engineer extends EmployeeType{  
    public int getTypeCode() {  
        return Employee.ENGINEER;  
    }  
}  
  
public class Salesman extends EmployeeType{  
    public int getTypeCode() {  
        return Employee.SALESMAN;  
    }  
} // ...
```



#}

# Replace Conditional With Polymorphism

## [Implementacja(1)]

```
public class Employee {  
    private EmployeeType type;  
    //(...)  
    public int GetPayAmount() {  
        return type.getPayAmount(this);  
    }  
  
    public int getMonthlySalary() {  
        //(...)  
    }  
  
    public int getCommission() {  
        //(...)  
    }  
}
```

```
public abstract class EmployeeType {  
    public abstract int GetPayAmount(Employee employee);  
}  
  
public class Salesman extends EmployeeType {  
    public int GetPayAmount(Employee employee) {  
        return employee.getMonthlySalary()  
            + employee.getCommission();  
    }  
}  
  
public class Engineer extends EmployeeType {  
    public int GetPayAmount(Employee employee) {  
        return employee.getMonthlySalary();  
    }  
}
```

- Stosowane aby **usunąć wyrażenia warunkowe zależne od typu obiektu.**
- Korzystanie z **polimorfizmu** gwarantuje proste i czytelne metody.



- Jeśli to konieczne – **wydziel instrukcję warunkową** do nowej metody.
- Umieść ją **na szczycie** hierarchii dziedziczenia.
- **Stwórz metody** w podklasach używając kodu z gałęzi [T].
- **Usuń ten kod z gałęzi [T].**



- Dla każdej gałęzi instrukcji warunkowej postępuj tak samo.
- Zamień metodę z instrukcją warunkową w metodę abstrakcyjną.



bns it }    # }   Introduce Null Object  
              Użycie Obiektu Pustego

**Zamiana wartości null na obiekt pusty.**



# #} Introduce Null Object bns it} [Implementacja]

```
public class Site{
    Customer getCustomer() {
        return _customer;
    }
    Customer customer;
}

public class Customer{
    public String getName() {...}
    public BillingPlan getPlan() {...}
}

public class BillingPlan{
    static public BillingPlan GetBasic(){
        return new BillingPlan();
    }
}

//...
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer == null)
    plan = BillingPlan.GetBasic();
else
    plan = customer.getPlan();
//...
String customerName;
if (customer == null) customerName = "occupant";
else customerName = customer.getName();
//...
```



```
class NullCustomer extends Customer {
    public booleanisNull() {
        return true;
    }
    public String getName(){
        return "occupant";
    }
}

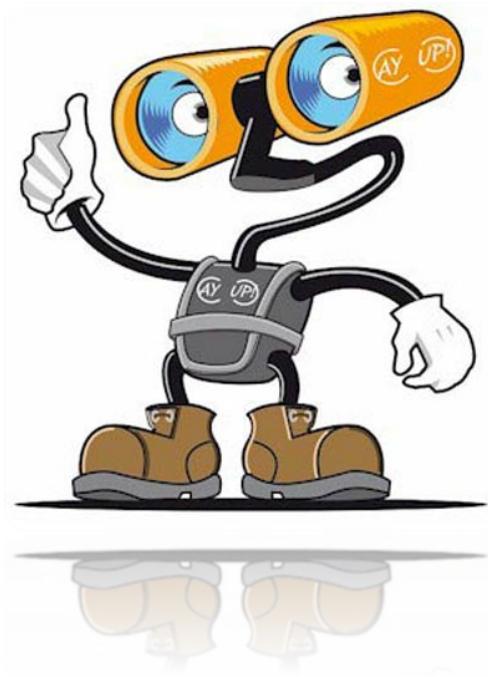
class Customer implements Nullable{
    public booleanisNull() {
        return false;
    }
    public static Customer newNull() {
        return new NullCustomer();
    }
    public String getName(){...}
    public BillingPlan getPlan(){...}
    protected Customer() {}
}

interface Nullable {
    booleanisNull();
}
```

```
class Site {
    public Customer getCustomer() {
        return (customer == null) ?
            Customer.newNull():
            customer;
    }
    Customer customer;
}

//...
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer.isNull())
    plan = BillingPlan.GetBasic();
else
    plan = customer.getPlan();
//...
String customerName = customer.getName();
//...
```

- Stosowane przy **wielokrotnym porównywaniu z null**.
- Użyteczne gdy **w większości przypadków** wartość **null** skutkowała **tym samym działaniem** – kod był powielany.



- **Utwórz podklasę odpowiedniej klasy.**
- **Dodaj metodę `IsNull()` do klasy (zwracającą `false`) i podklasy (zwracającą `true`) [T].**
- **Zastąp przymówniania obiektu do `null` wywołaniami `IsNull()`.**

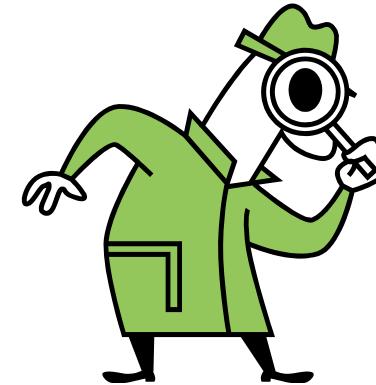


- **Zastąp wszystkie przypisania wartości `null` obiektem pustym [T].**
- **Zdefiniuj metody obiektu pustego kodem gałęzi wybieranej gdy wartość jest równa `null`.**
- **Usuń kod porównujący do `null` z miejsc wywołujących metody obiektu pustego [T].**



bns it } **Introduce Assertion**  
Dodanie Asercji

Dodanie asercji, jawnie sprawdzającej spełnienie wszystkich warunków.



# Introduce Assertion

## [Implementacja]

```
public double getExpenseLimit() {  
    return (expenseLimit != NULL_EXPENSE) ?  
        expenseLimit : primaryProject.getMemberExpenseLimit();  
}
```



```
public double getExpenseLimit() {  
    Assert.isTrue (expenseLimit != NULL_EXPENSE || primaryProject != null);  
    return (expenseLimit != NULL_EXPENSE) ?  
        expenseLimit : primaryProject.getMemberExpenseLimit();  
}
```

- Stosujemy gdy kod działa poprawnie **tylko w przypadku spełnienia pewnych założeń.**
- Asercje **ułatwiają komunikację i wyszukiwanie błędów.**



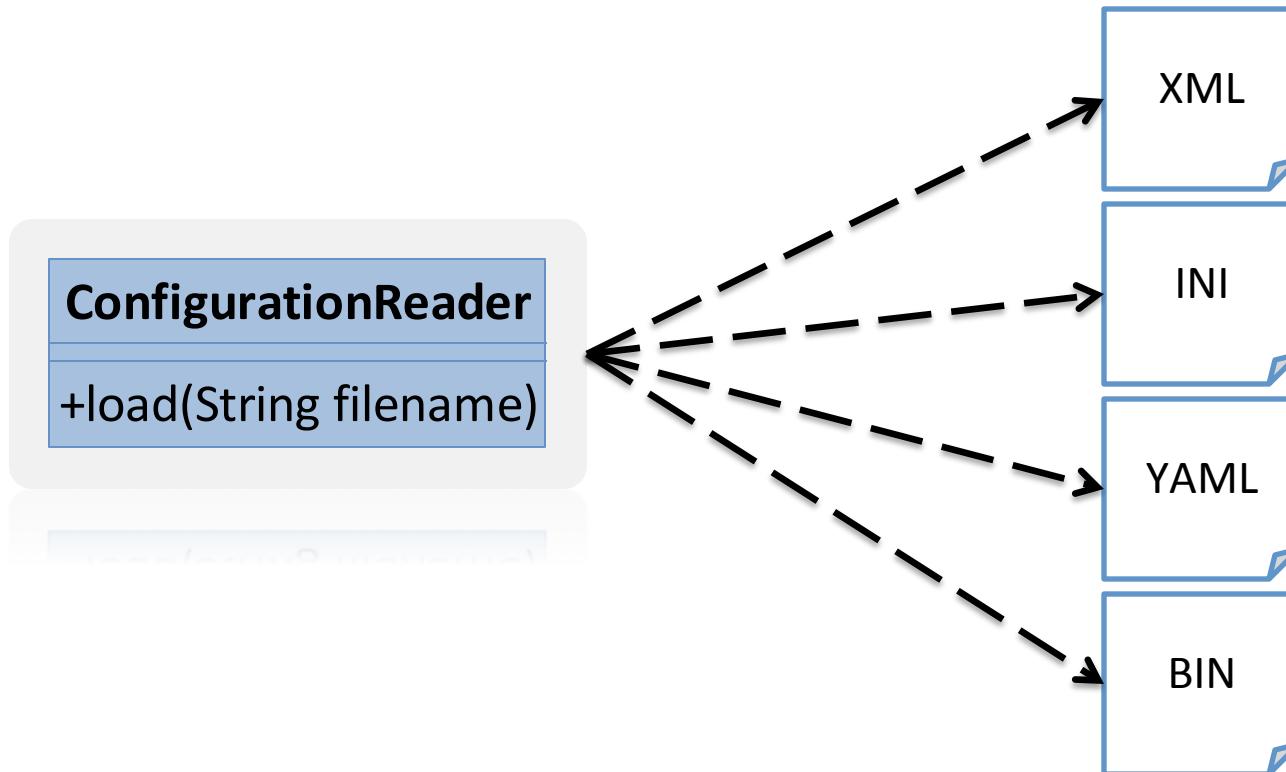
- Jeżeli zauważysz, że w kodzie **zakłada się prawdziwość pewnego warunku, dodaj sprawdzającą go asercję.**





- # 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
```

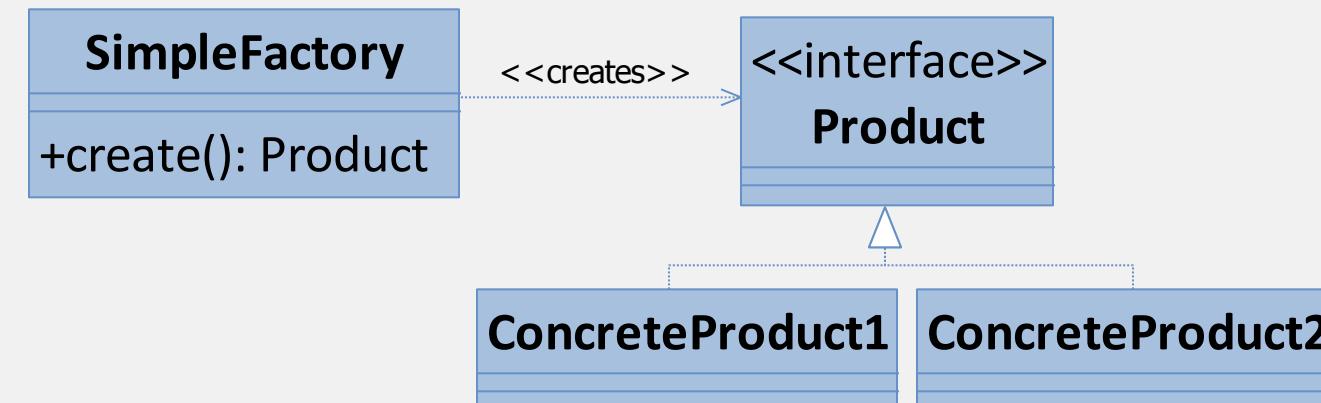
<<creates>>

**XMLSerializer**

**INISerializer**

**YAMLSerializer**

**BinarySerializer**

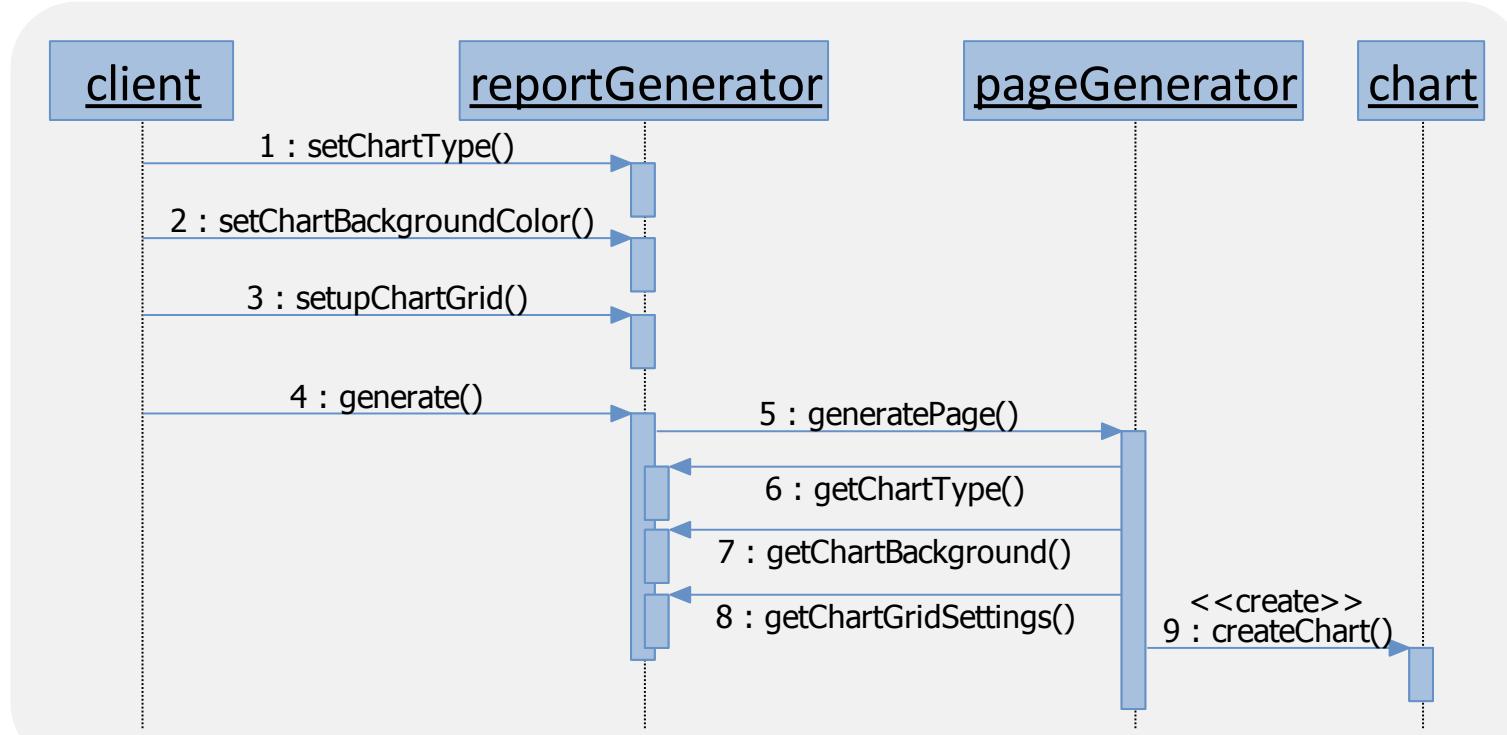


Wzorzec *Simple Factory* hermetyzuje tworzenie rodziny obiektów.

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

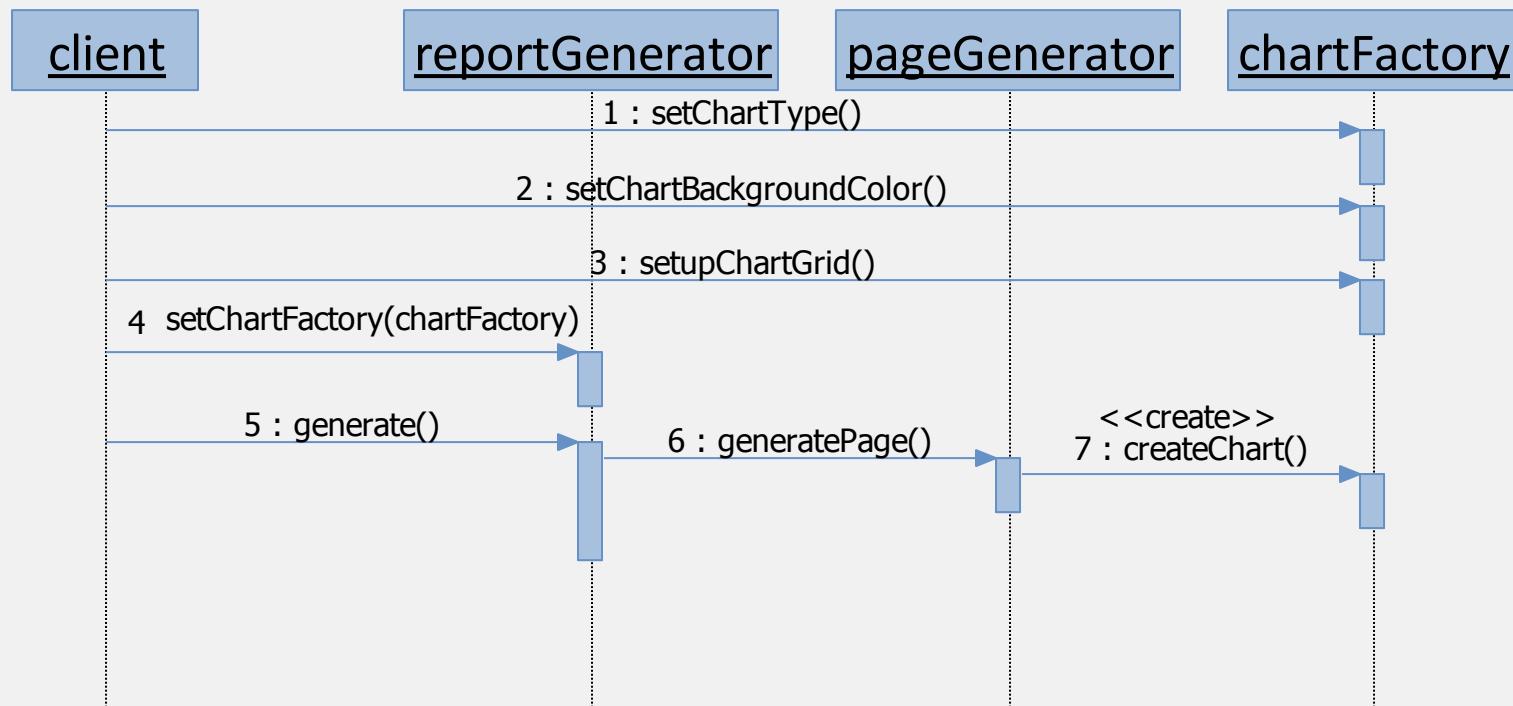
# Refaktoryzacja: *Move Creation Knowledge to Factory*



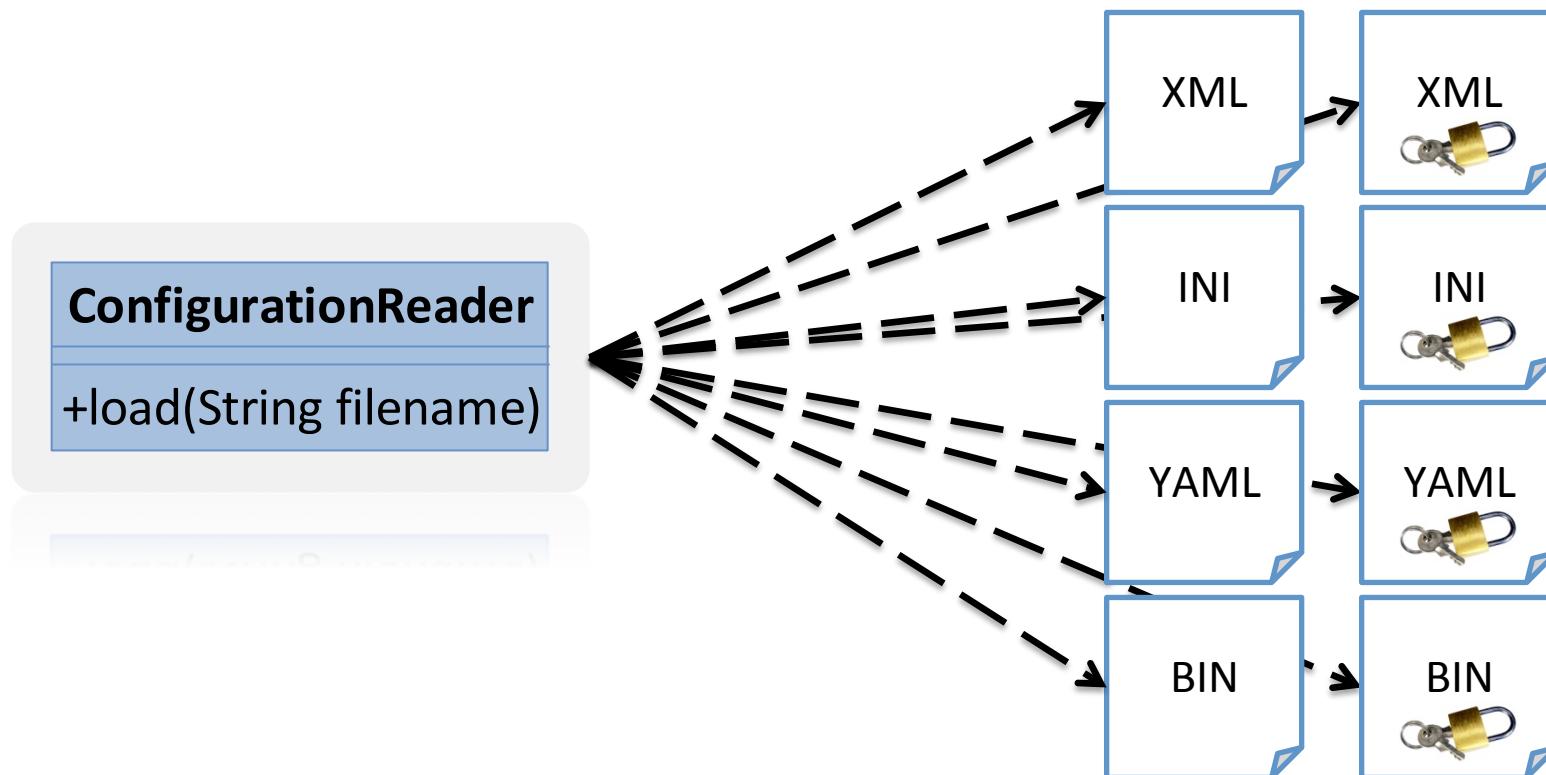
8 : getChartGridSettings()

6 : createChart()  
<<create>>

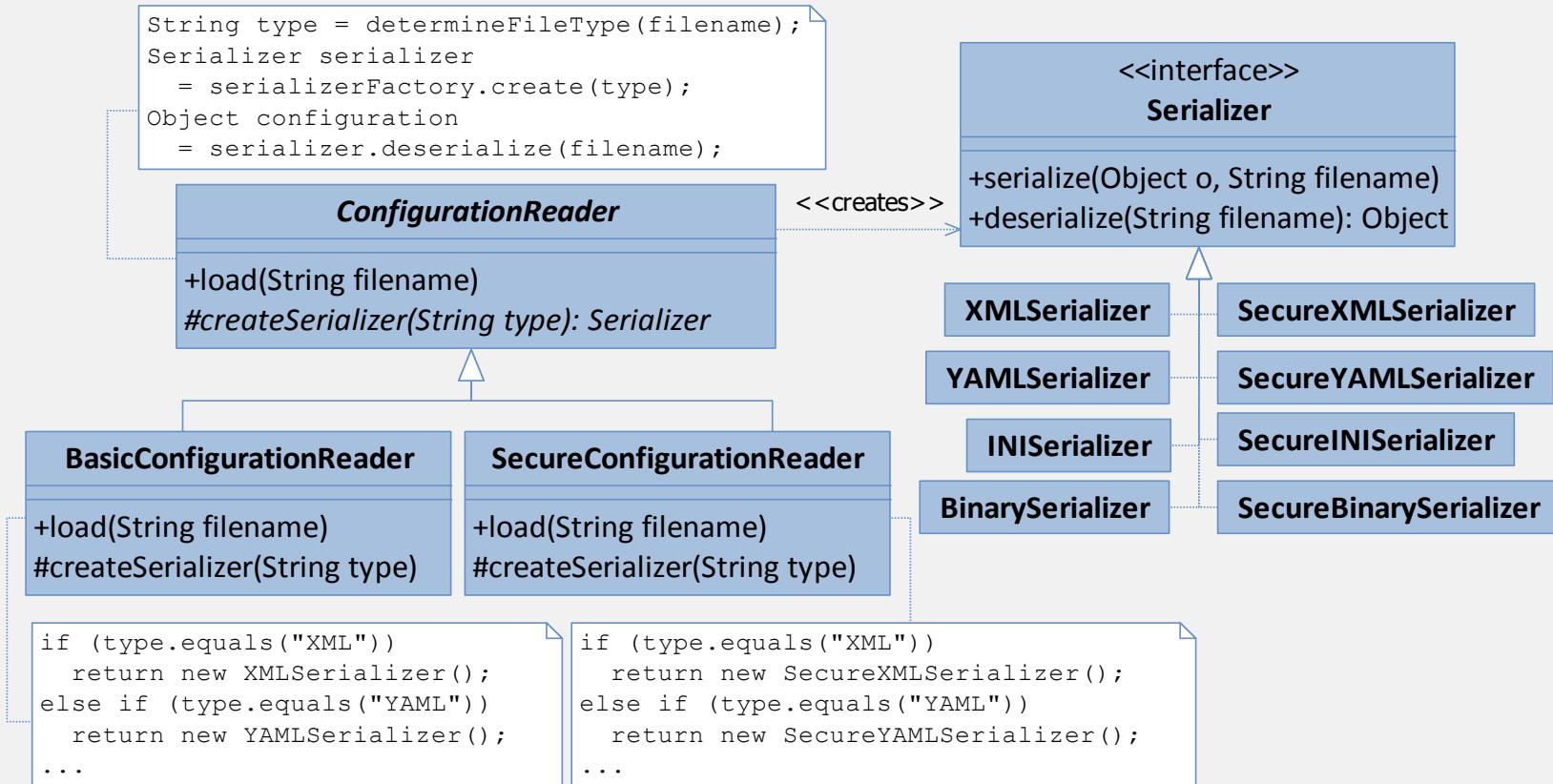
# Refaktoryzacja: *Move Creation Knowledge to Factory*

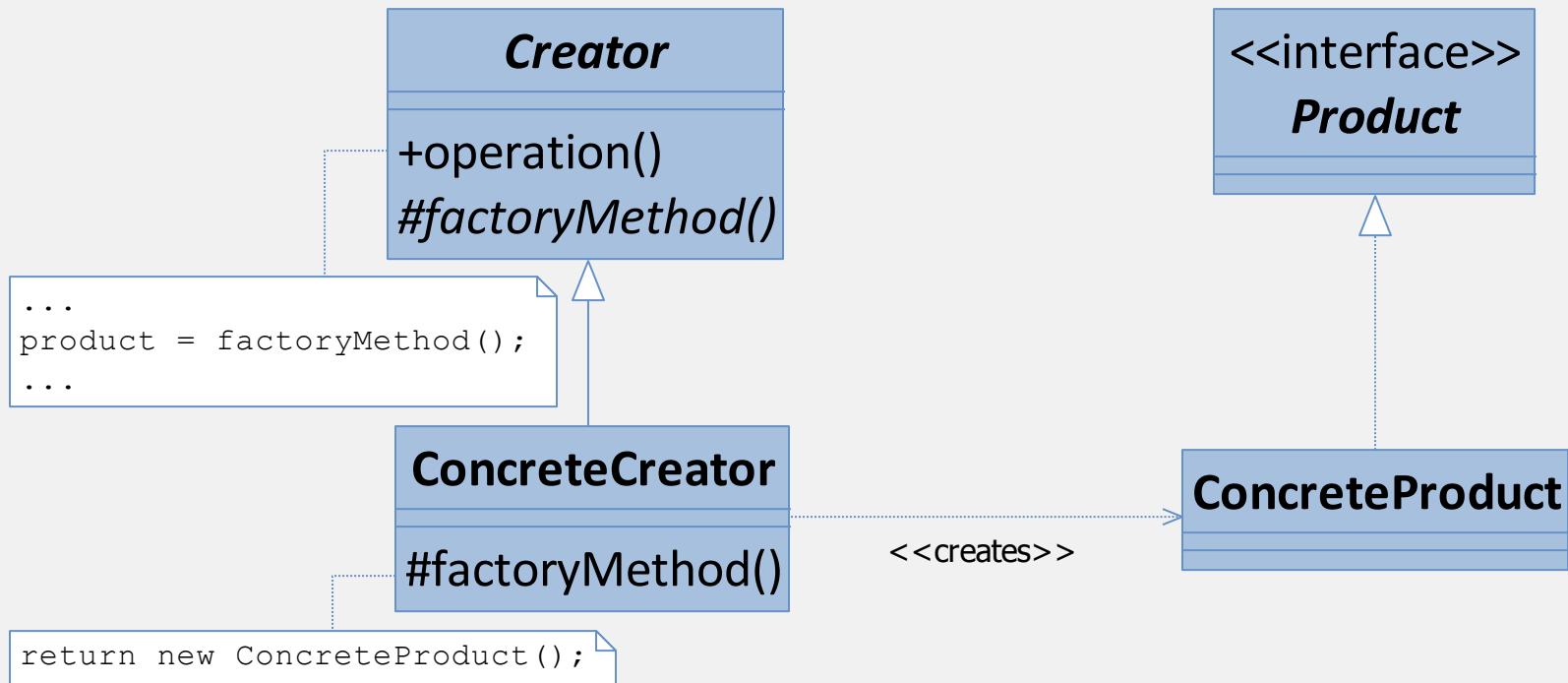


bns it } # Wzorzec Factory Method  
Wzorce kreacyjne



# Przykład





reflex u m ConcreteProduct () :

#factoryMethod()

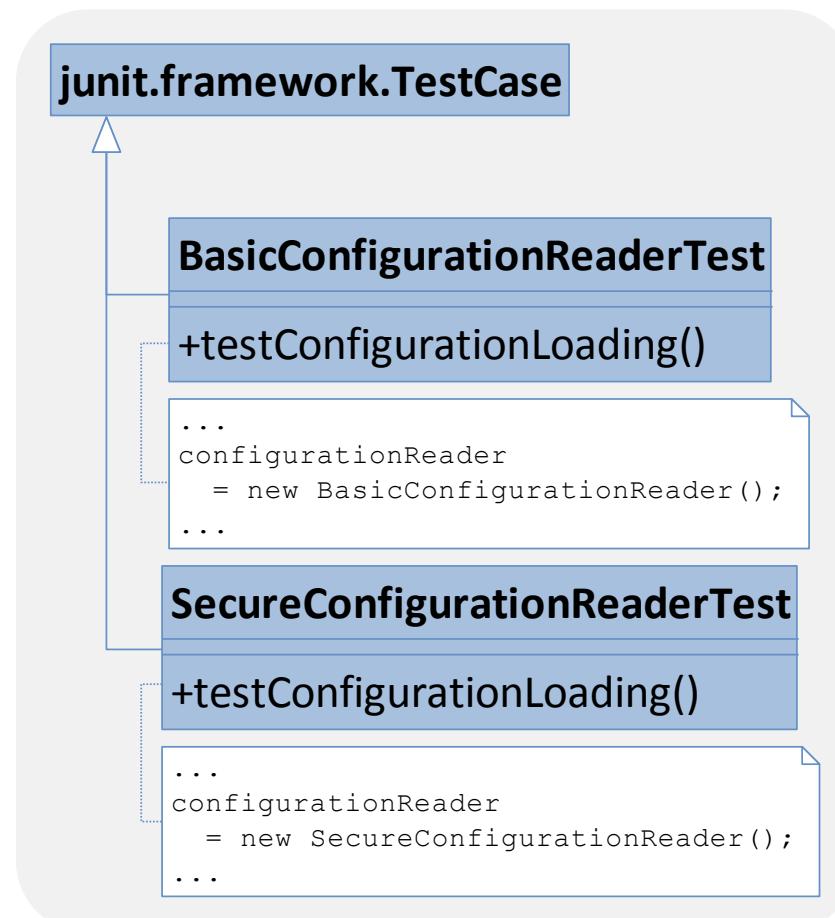
<<deleg>>

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

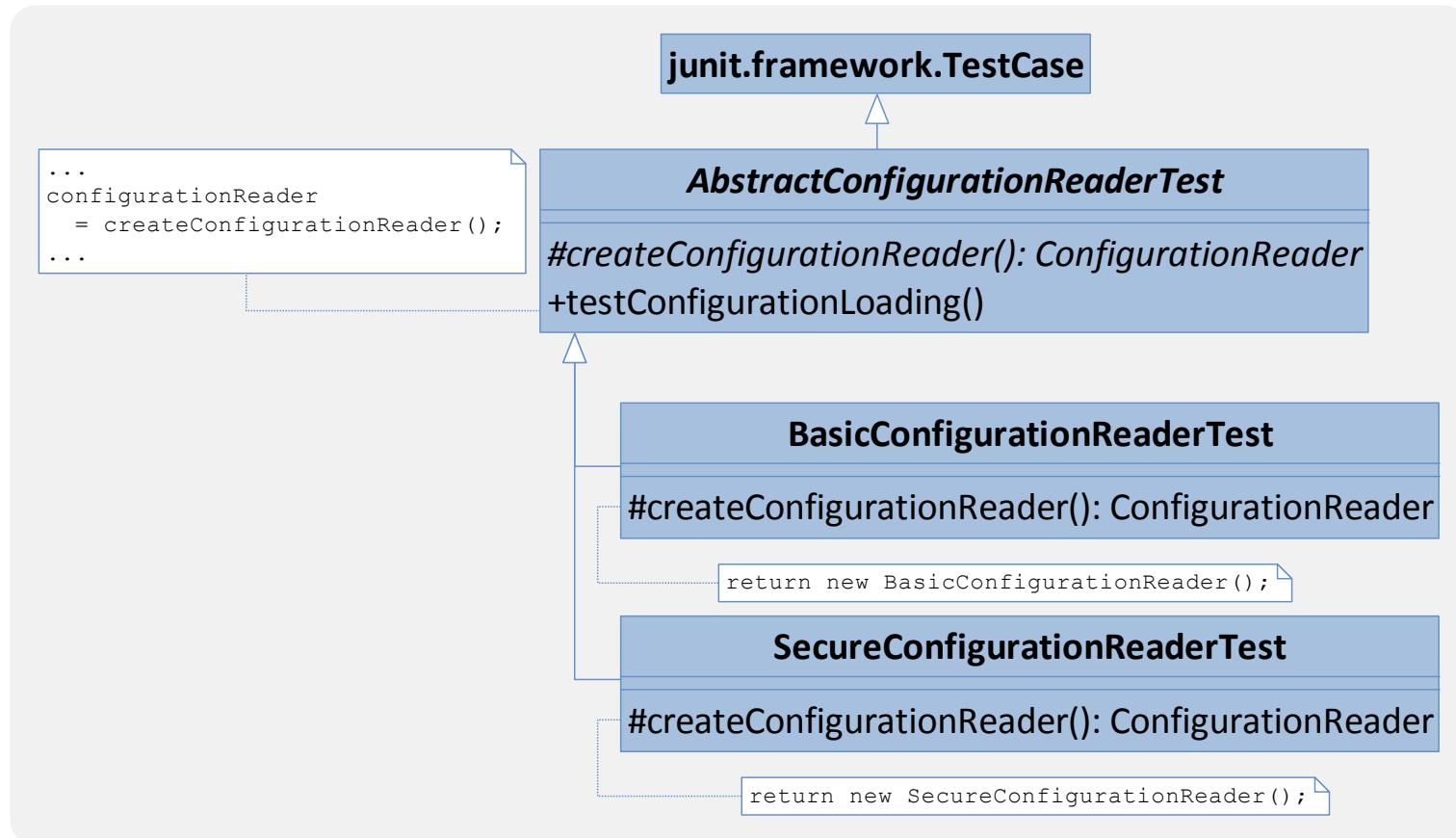
- # *Factory Method* eliminuje potrzebę współpracy z klasami *ConcreteProduct*. Klient odwołuje się tylko do interfejsu klasy *Product*.
- # W przypadku kiedy klient będzie potrzebował obiektu klasy *ConcreteProduct* ze względu na jego specyficzny interfejs, będzie musiał wykonać rzutowanie.
- # Klasa *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.

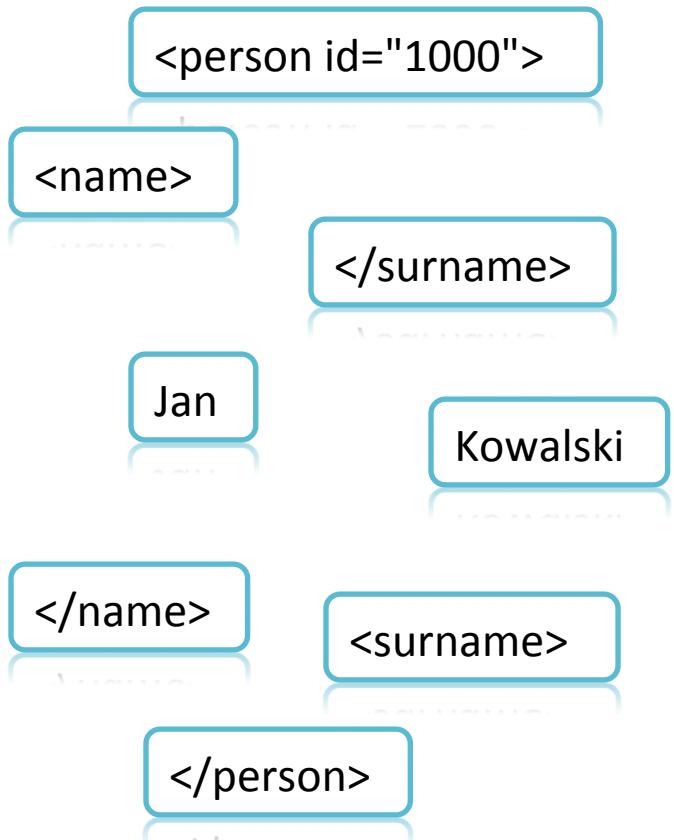
# Refaktoryzacja: *Introduce Polymorphic Creation with Factory Method*



# Refaktoryzacja: *Introduce Polymorphic Creation with Factory Method*



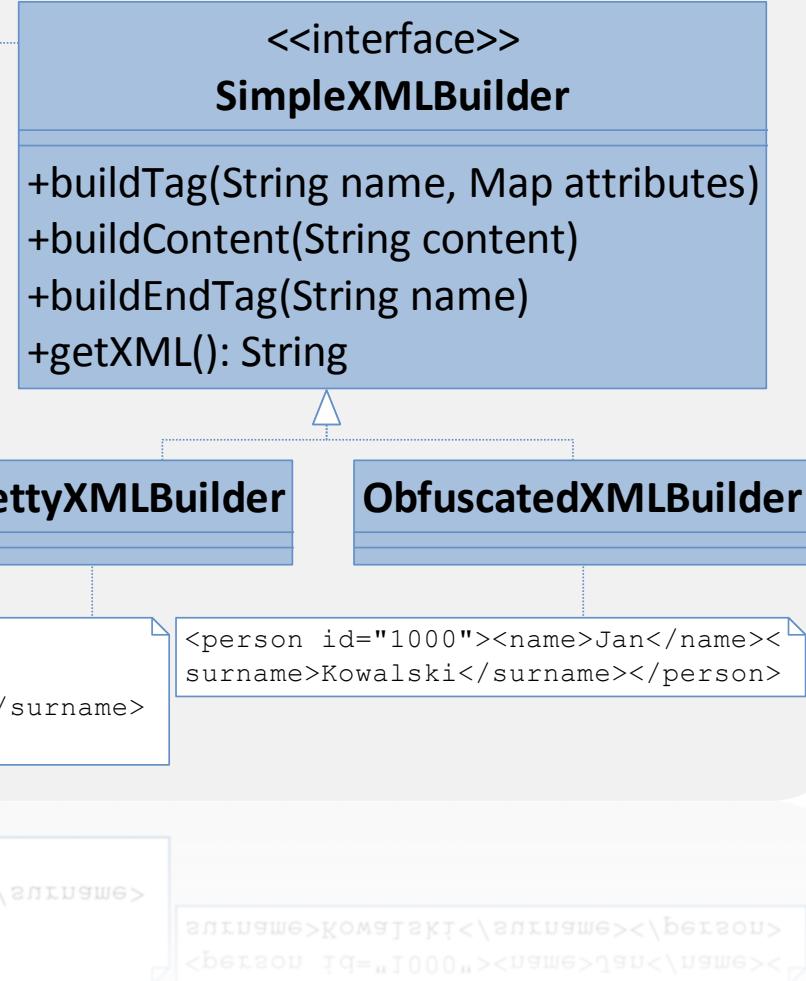
bns it} #} Wzorzec Builder  
Wzorce kreacyjne

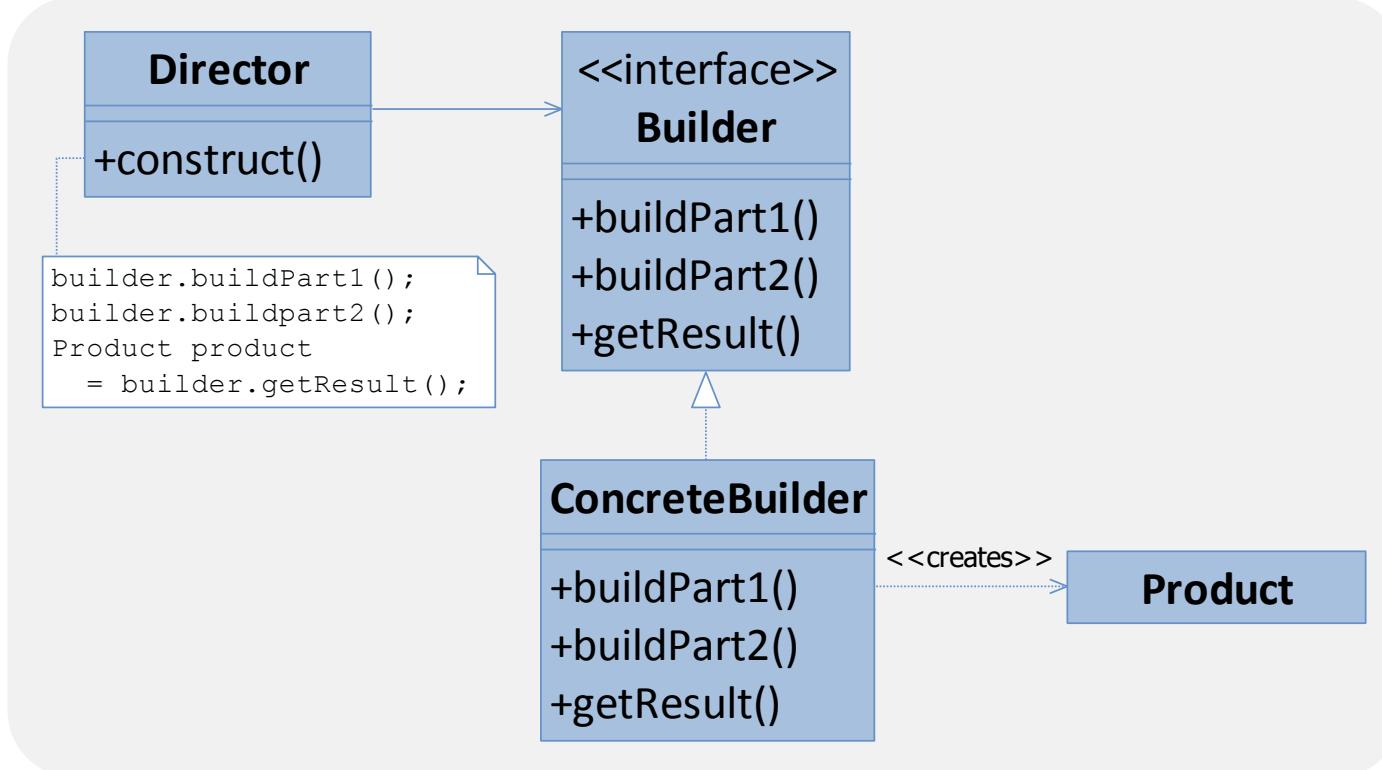


```
<person id="1000">  
  <name>Jan</name>  
  <surname>Kowalski</surname>  
</person>
```

<\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

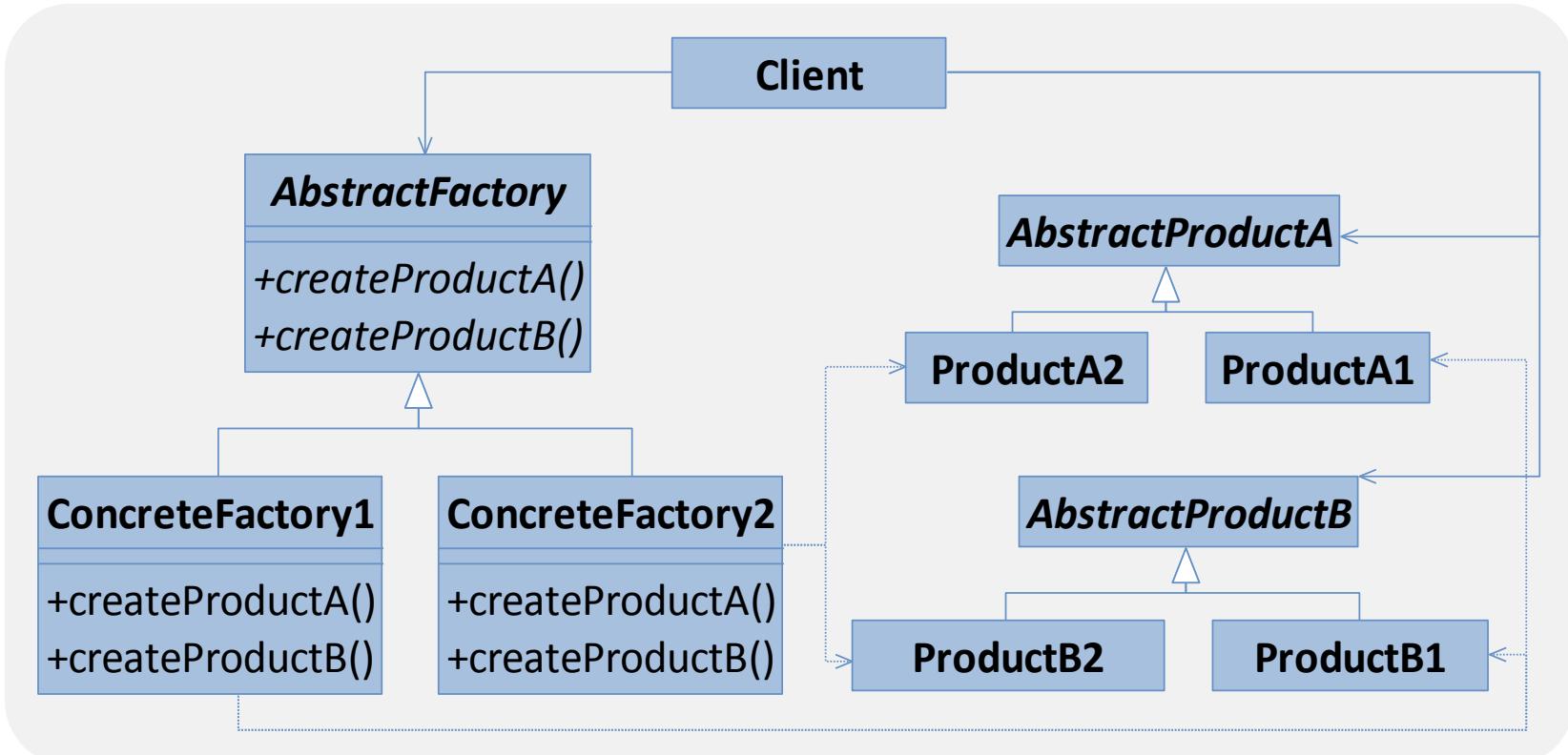
Wzorzec *Builder* pozwala konstruować obiekty z komponentów.

- # 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.



+createProductB()  
+createProductA()

+createProductB()  
+createProductA()

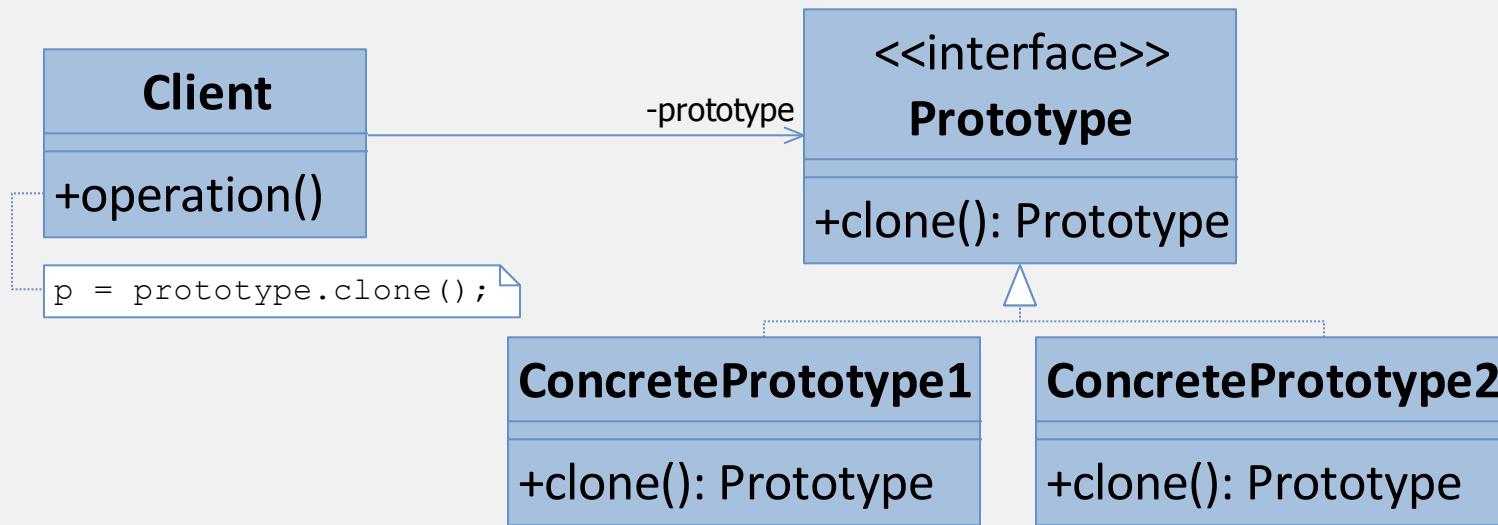
AbstractProductA

AbstractProductB

- # 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;
```

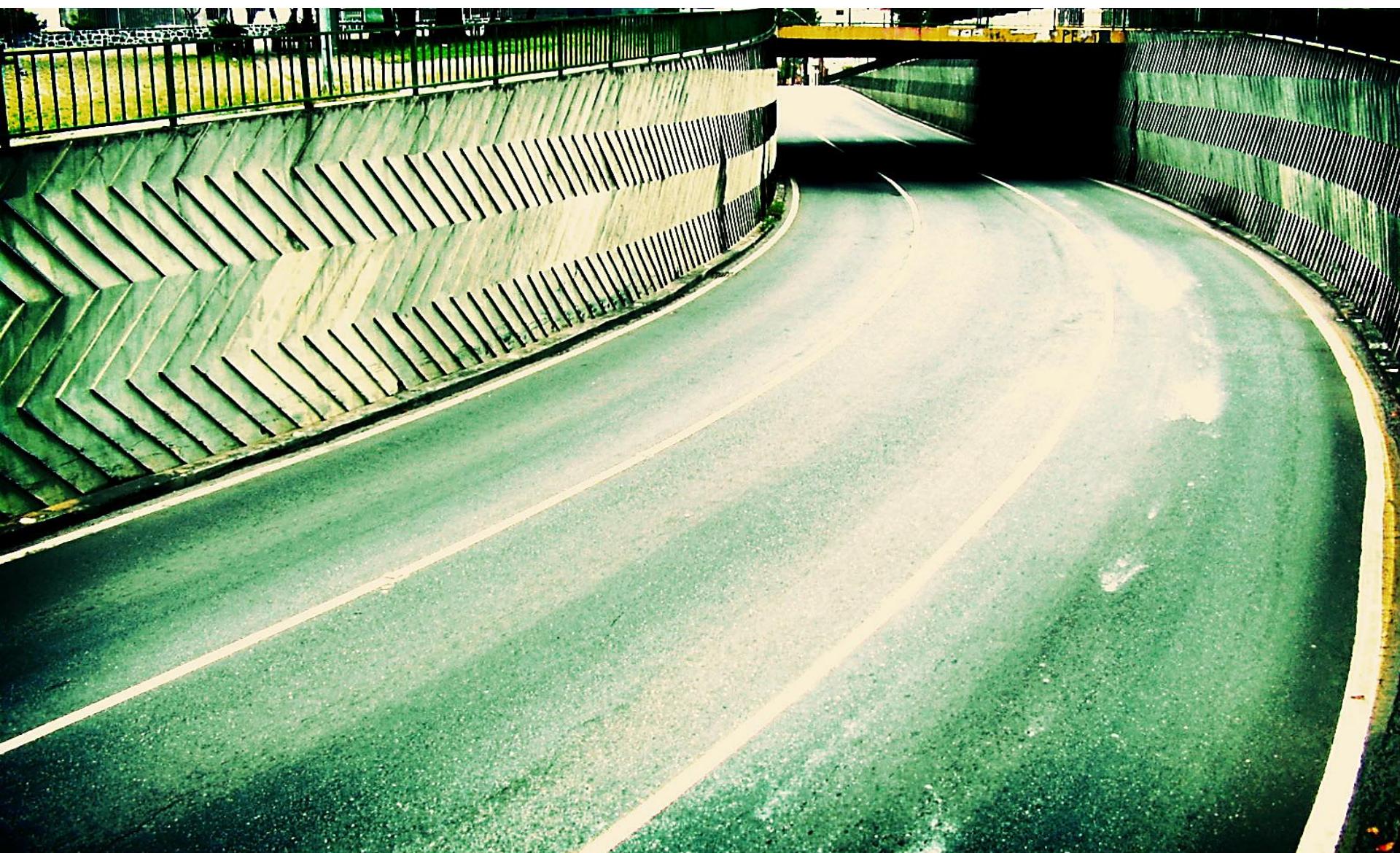
return instance;

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.



# Wzorzec Command

Wzorce behawioralne

# Przykład

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



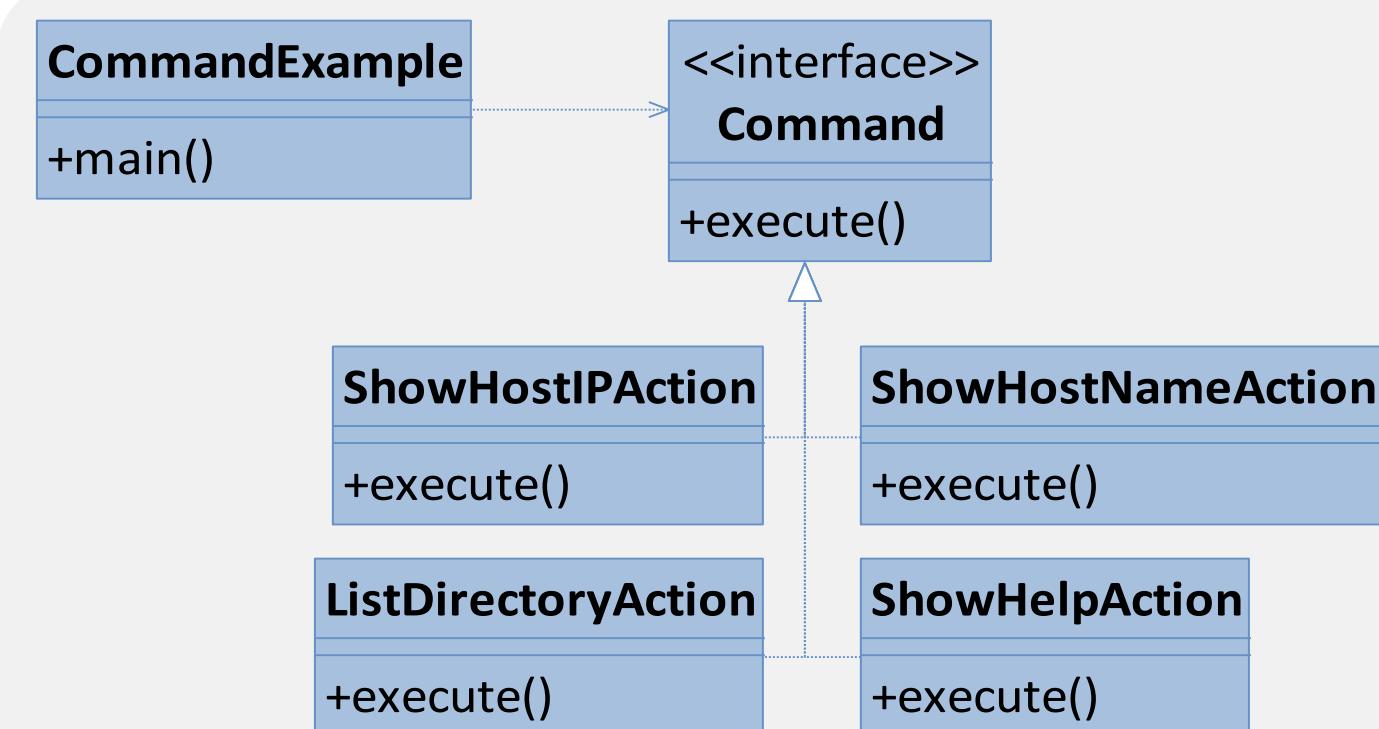
```
> java CommandExample ShowHostIP  
192.168.0.5
```

```
> java CommandExample ShowMachineName  
BNSServer
```



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

# bns it} Przykład



# Kod przykładu

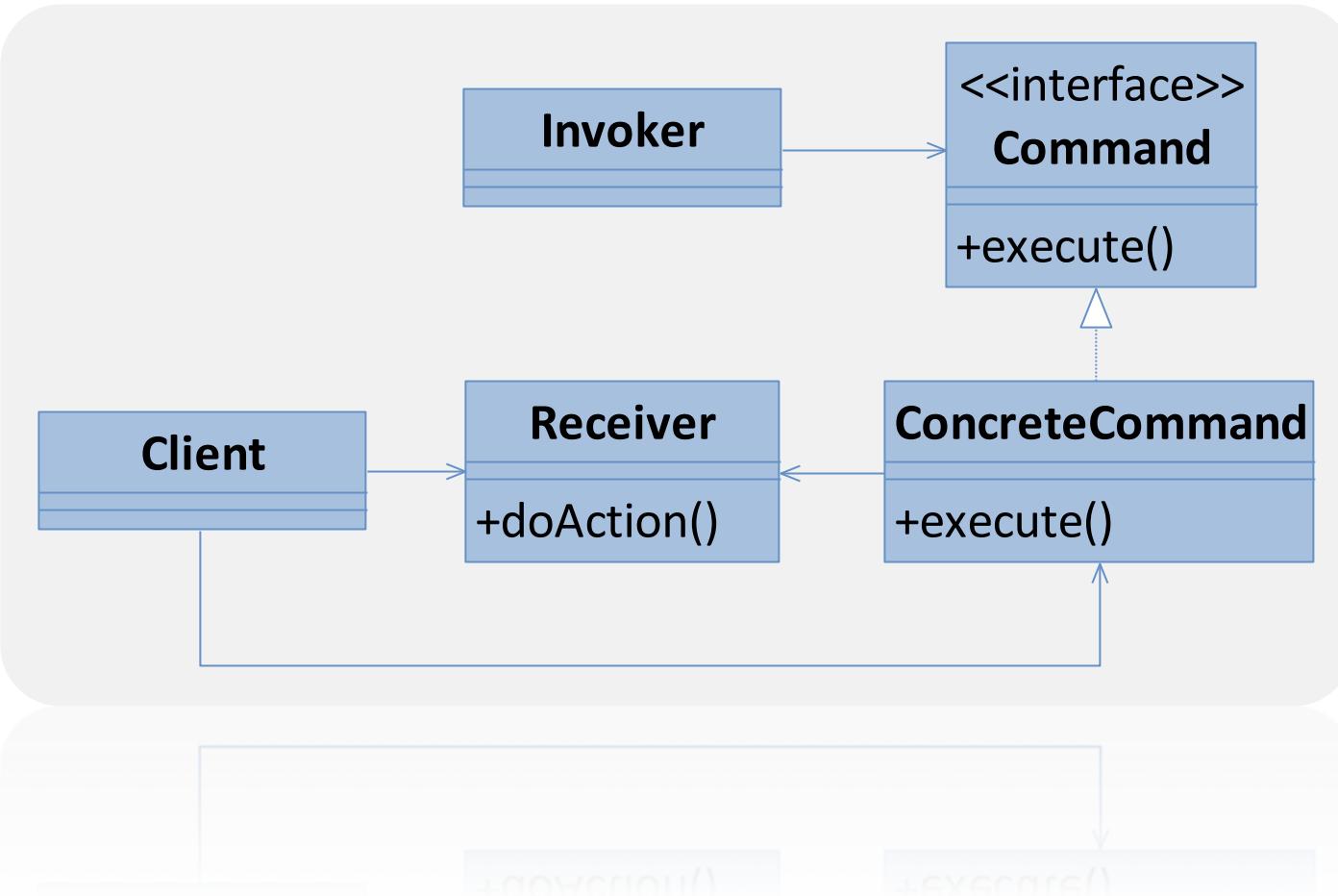
```
public interface Command {  
    public void execute();  
}
```

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

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

```
public class ShowHostIPAction  
    implements Command {  
    public void execute() {  
        String ip = "";  
        try {  
            ip = InetAddress.getLocalHost().getHostAddress();  
        } catch (UnknownHostException e) {  
            System.out.println("Cannot obtain ip.");  
        }  
        System.out.println(ip);  
    }  
}
```

```
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();  
        } else if (...) {...}  
        } else if (...) {...}  
    }  
}
```



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
- 
- # Klasy realizujące interfejs *ActionListener* do obsługi zdarzeń w języku Java

## Refaktoryzacja: *Replace Conditional Dispatcher with Command*

```
if (command.equals("ShowHelp")) {
    String msg = "ShowHostIP - Display computer IP.\n";
    msg += "ShowMachineName - Display name of the computer.\n";
    msg += "ListCurrentDirectory - Lists files in current directory.";
    System.out.println(msg);
} else if (command.equals("ShowHostIP")) {
    String ip = InetAddress.getLocalHost().getHostAddress();
    System.out.println(ip);
} else if (command.equals("ShowHostName")) {
    String name = InetAddress.getLocalHost().getHostName();
    System.out.println(name);
} else if (command.equals("ListCurrentDirectory")) {
    File dir = new File(System.getProperty("user.dir"));
    String[] subfiles = dir.list();
    for (String filename : subfiles) {
        System.out.println(filename);
    }
}
```

## Refaktoryzacja: *Replace Conditional Dispatcher with Command*

```
private Map<String, Command> commands
= new HashMap<String, Command>();

public CommandFullExample() {
    commands.put("ShowHelp", new ShowHelpAction());
    commands.put("ShowHostIP", new ShowHostIPAction());
    commands.put("ShowHostName", new ShowHostNameAction());
    commands.put("ListDirectory", new ListDirectoryAction());
}

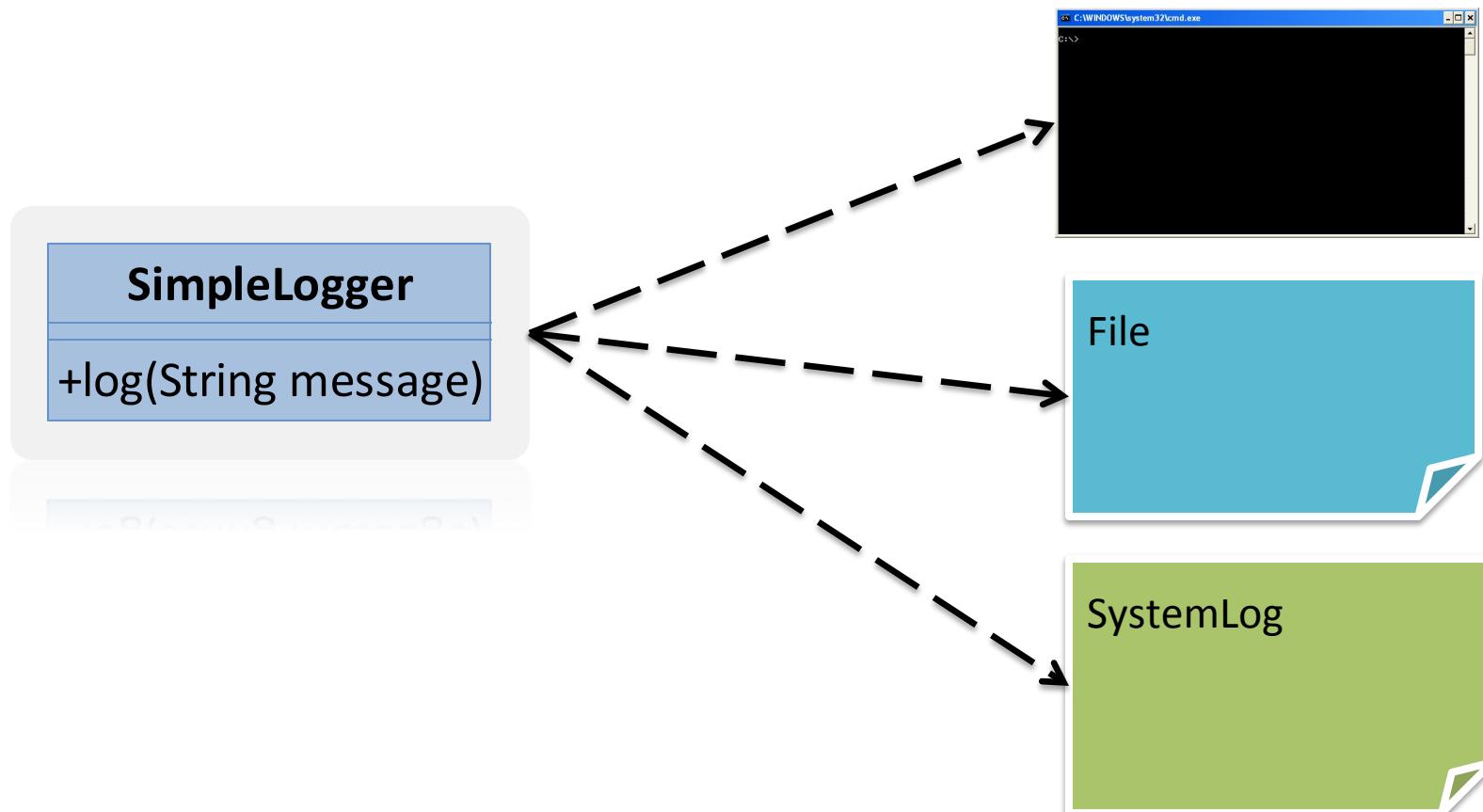
public void run(String[] args) {
    if (args.length == 0) return;

    Command command = commands.get(args[0]);

    command.execute();
}
```

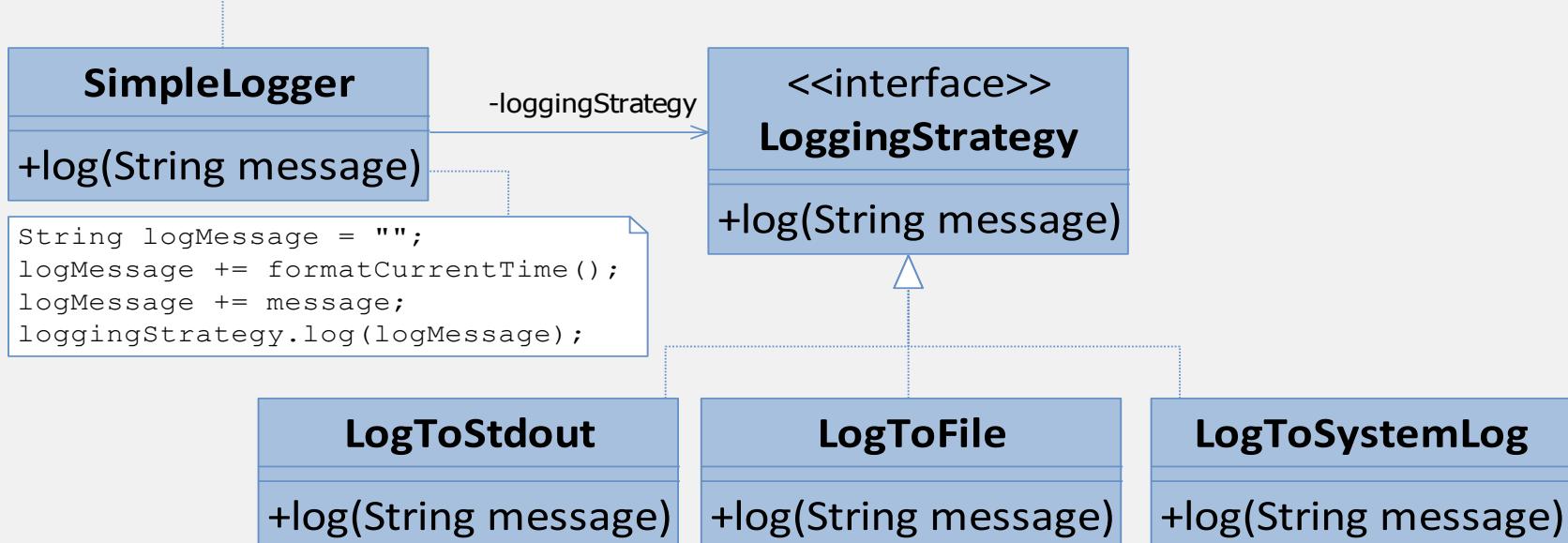
bns it } # Wzorzec Strategy  
Wzorce behawioralne

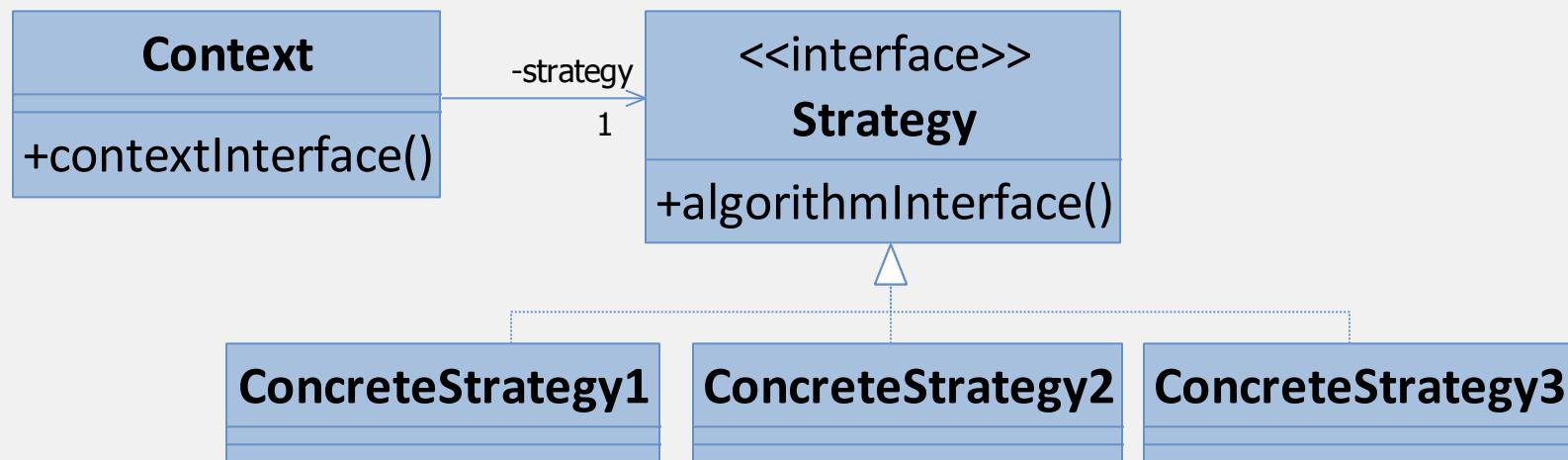
## Przykład



## 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) {  
        System.out.println(logMessage);  
    } else if (loggerType == LoggerType.FILE) {  
        try {  
            BufferedWriter out  
                = new BufferedWriter(new FileWriter("log.txt"));  
            out.write("aString");  
            out.close();  
        } catch (IOException e) {  
            throw new RuntimeException("Cannot write to file.", e);  
        }  
    } // else if ...  
    // else if ...  
    // else if ...  
}
```

## Refaktoryzacja: *Replace Conditional Logic with Strategy*

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

```
public void log(String message) {
    String logMessage = "";
    logMessage += formatCurrentTime();
    logMessage += message;

    loggingStrategy.log(logMessage);
}
```



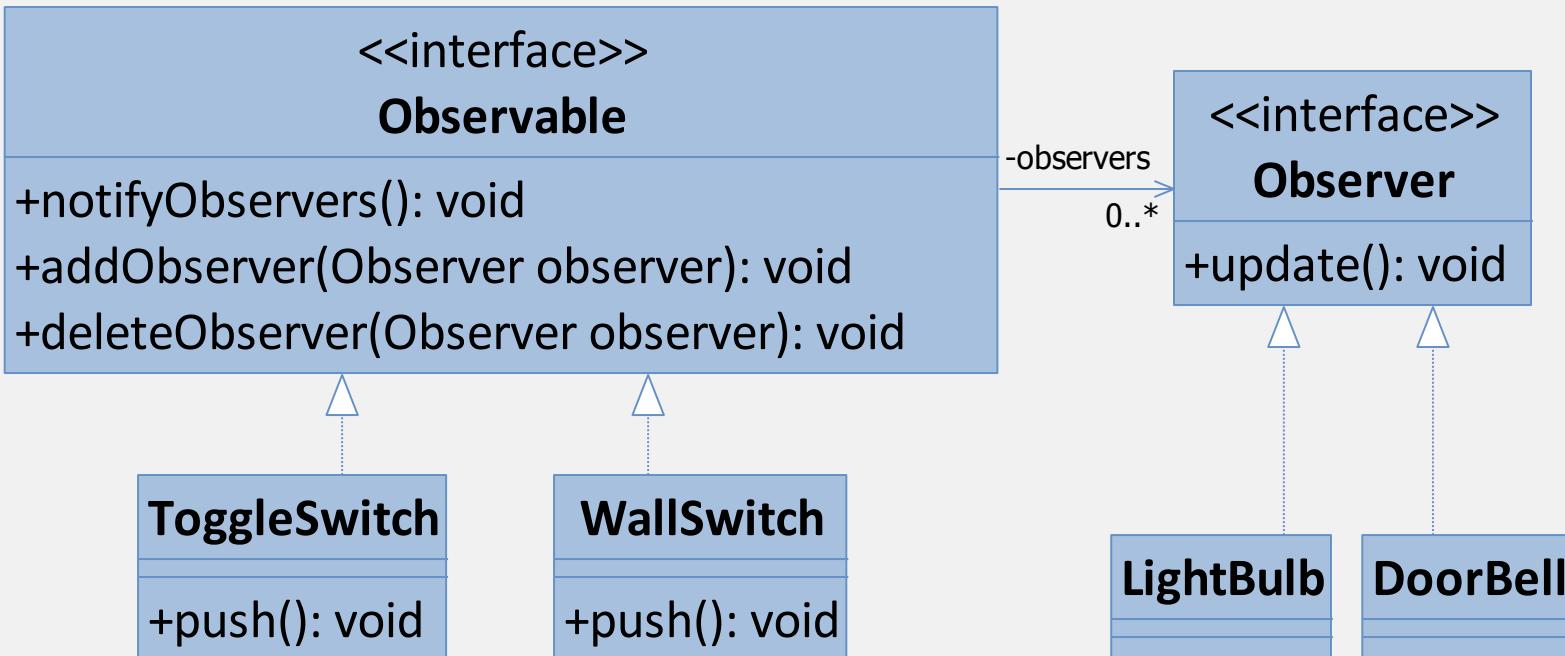
# Wzorzec Observer

Wzorce behawioralne

# Zagadnienie projektowe



# Struktura przykładu



# Kod przykładu

```
public interface Observable {  
    public void addObserver(Observer observer);  
    public void deleteObserver(Observer observer);  
    public void notifyObservers();  
}
```

```
public class WallSwitch implements Observable {  
    private List<Observer> observers  
        = new ArrayList<Observer>();  
    public void push() {  
        System.out.println("WallSwitch: Activated");  
        notifyObservers();  
    }  
    public void addObserver(Observer observer) {  
        observers.add(observer);  
    }  
    public void deleteObserver(Observer observer){  
        observers.remove(observer);  
    }  
    public void notifyObservers() {  
        for(Observer activeable : observers) {  
            activeable.update();  
        }  
    }  
}
```

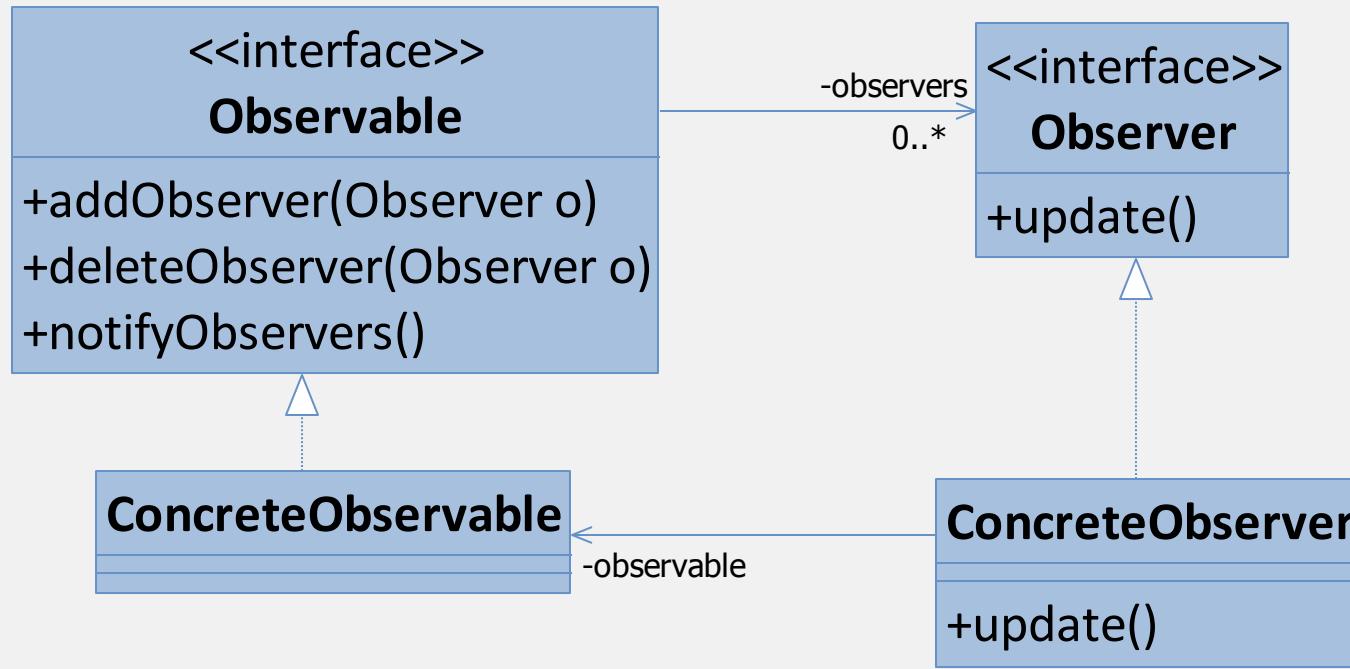
```
public interface Observer {  
    public void update();  
}
```

```
public class DoorBell  
    implements Observer {  
    public void update() {  
        System.out.println("DoorBell: Ring");  
    }  
}
```

```
public class LightBulb  
    implements 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()

-observers

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

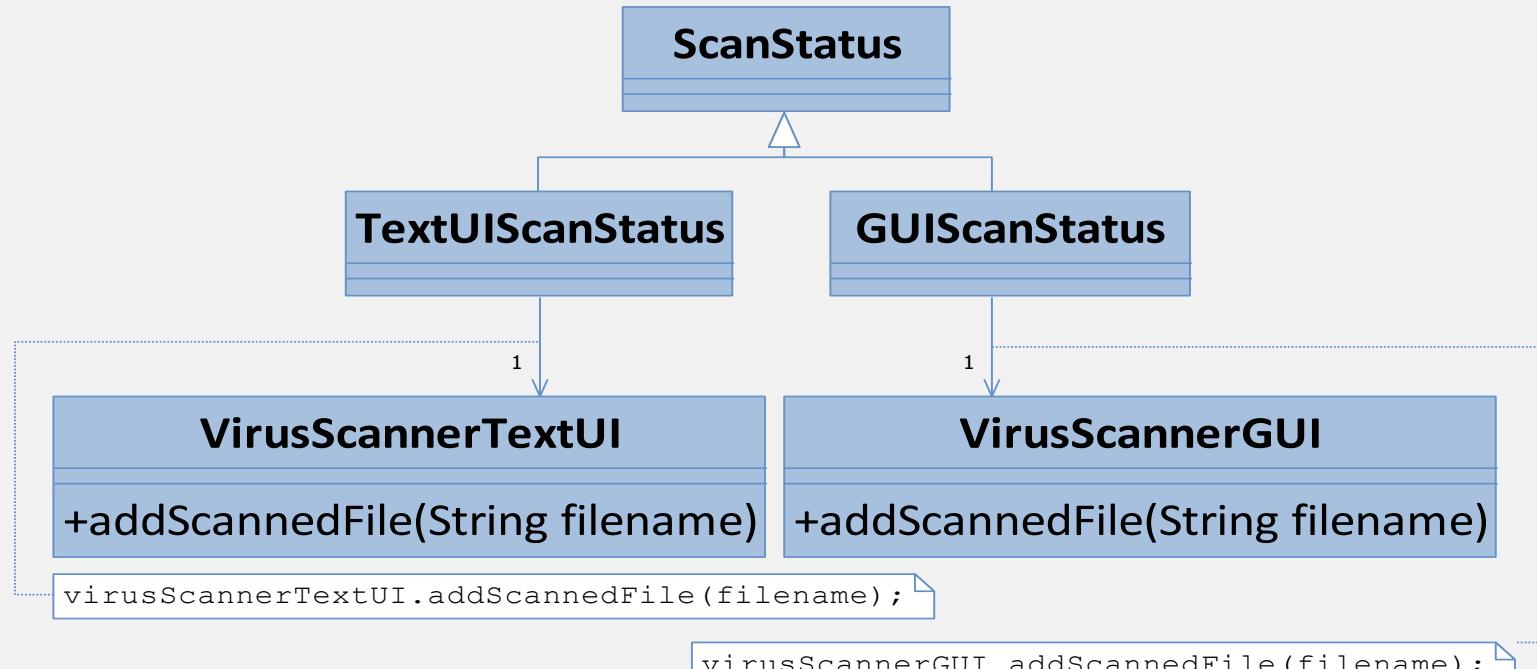
- # Wzorzec *Observer* zapewnia luźne powiązanie pomiędzy obiektami.
- # Umożliwia niezależne wymienianie obiektów obserwowanych i obserwatorów.
- # Umożliwia dodawanie obserwatorów bez konieczności modyfikowania kodu obiektu obserwowanego.
- # Powiadomienie wysyłane przez obserwowanego nie musi specyfikować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

# Informowanie o wydarzeniach zachodzących w systemie

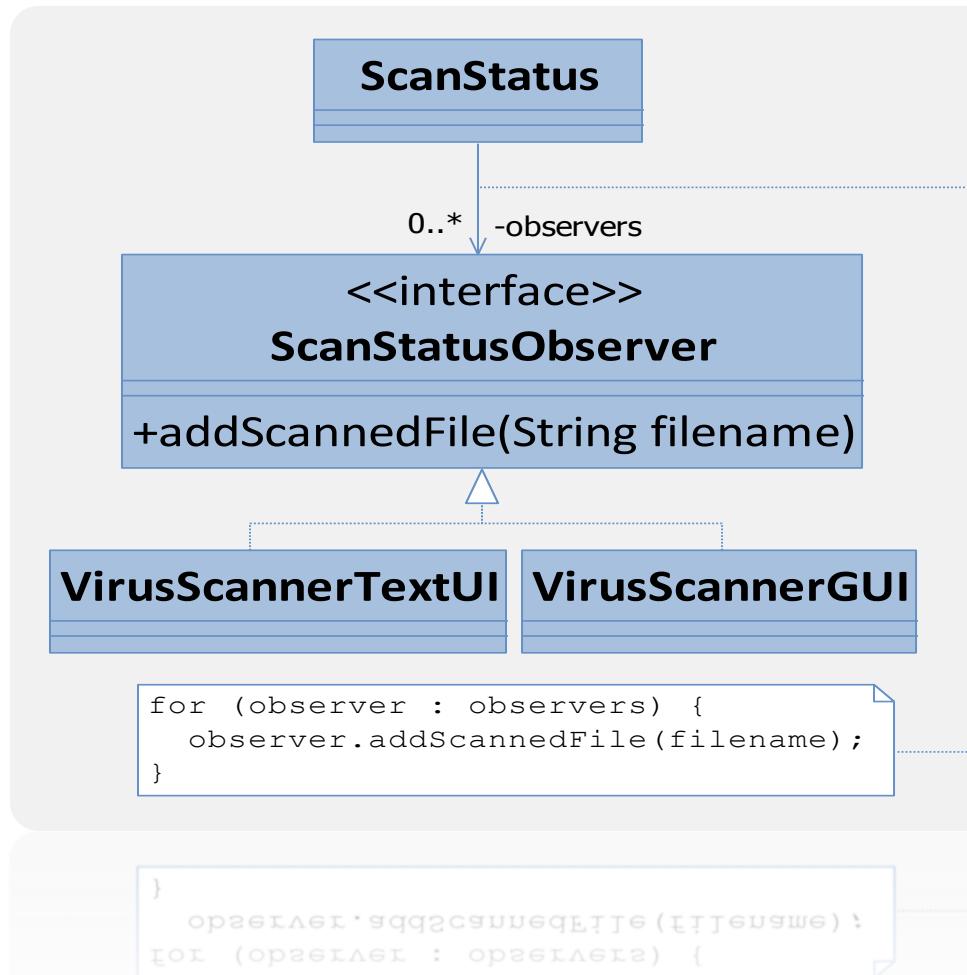
# Informowanie widoku o zmianach w modelu danych

# *java.util.Observable* i *java.util.Observer*

# Refaktoryzacja: Replace Hard-Coded Notifications with Observer



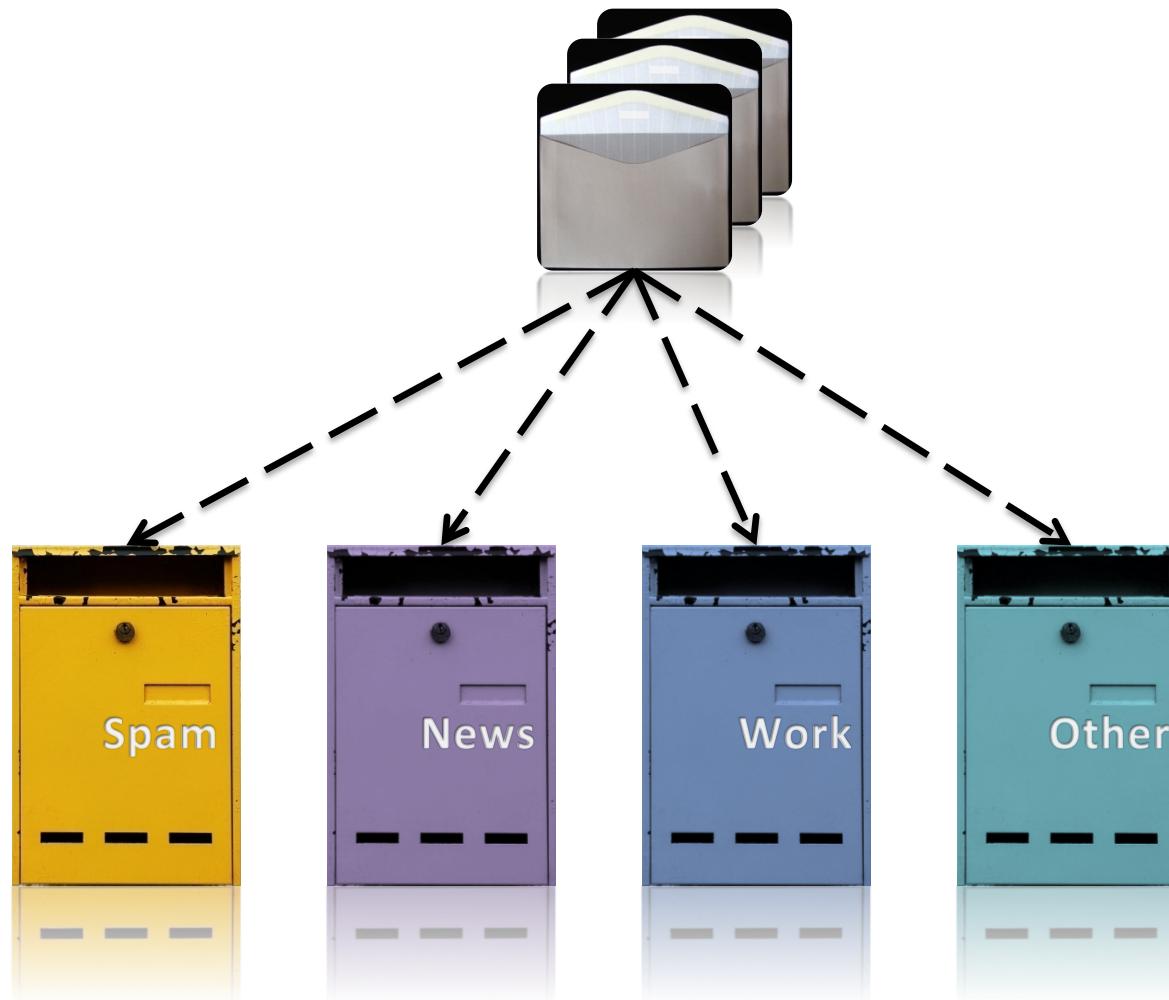
# Refaktoryzacja: Replace Hard-Coded Notifications with Observer



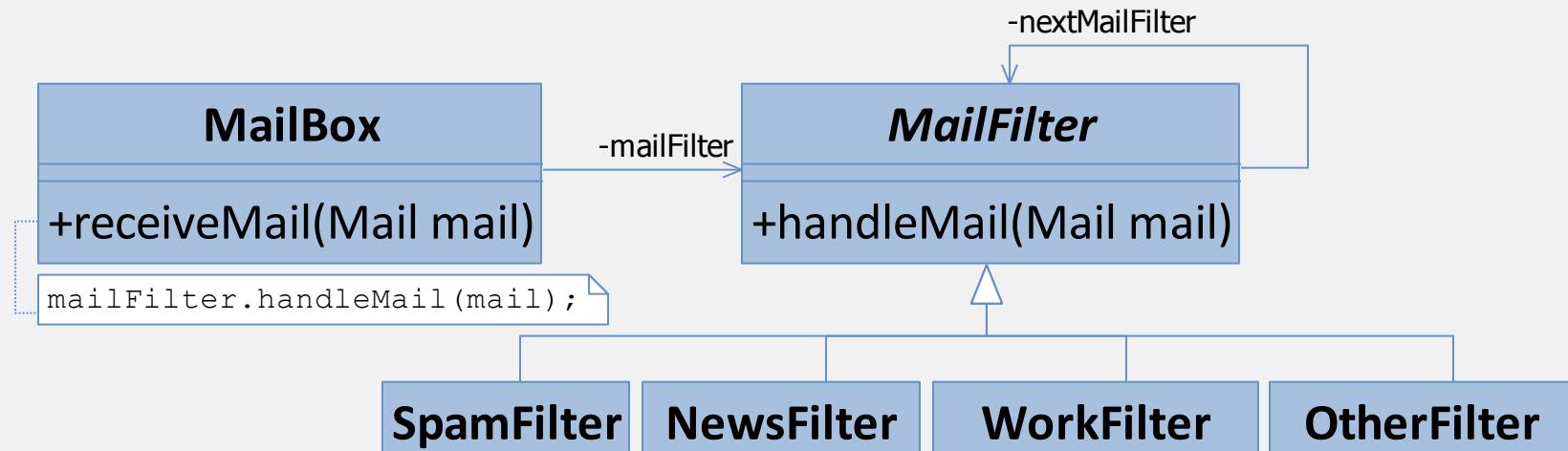
bns it }    # }    Wzorzec Chain of Responsibility

Wzorce behawioralne

# Przykład



# bns it} Przykład



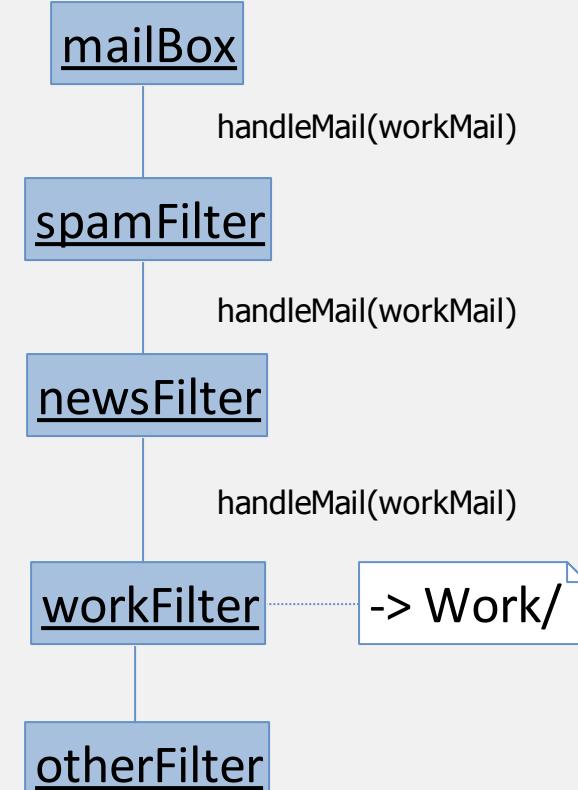
spamFilter  
newsFilter  
workFilter  
otherFilter

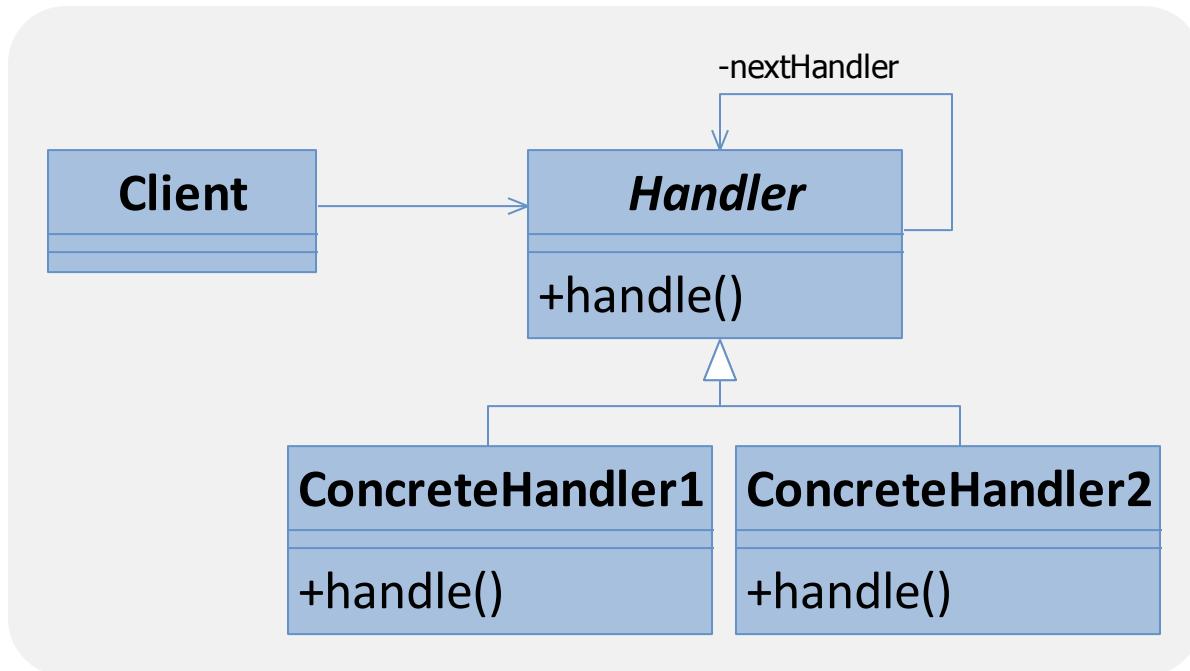
# Przykład

## 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.

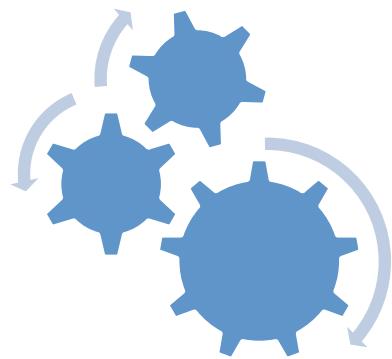
- # Obsługa zdarzeń myszy i klawiatury w systemach okienkowych.
- # Filtrowanie danych w konfigurowalny sposób.



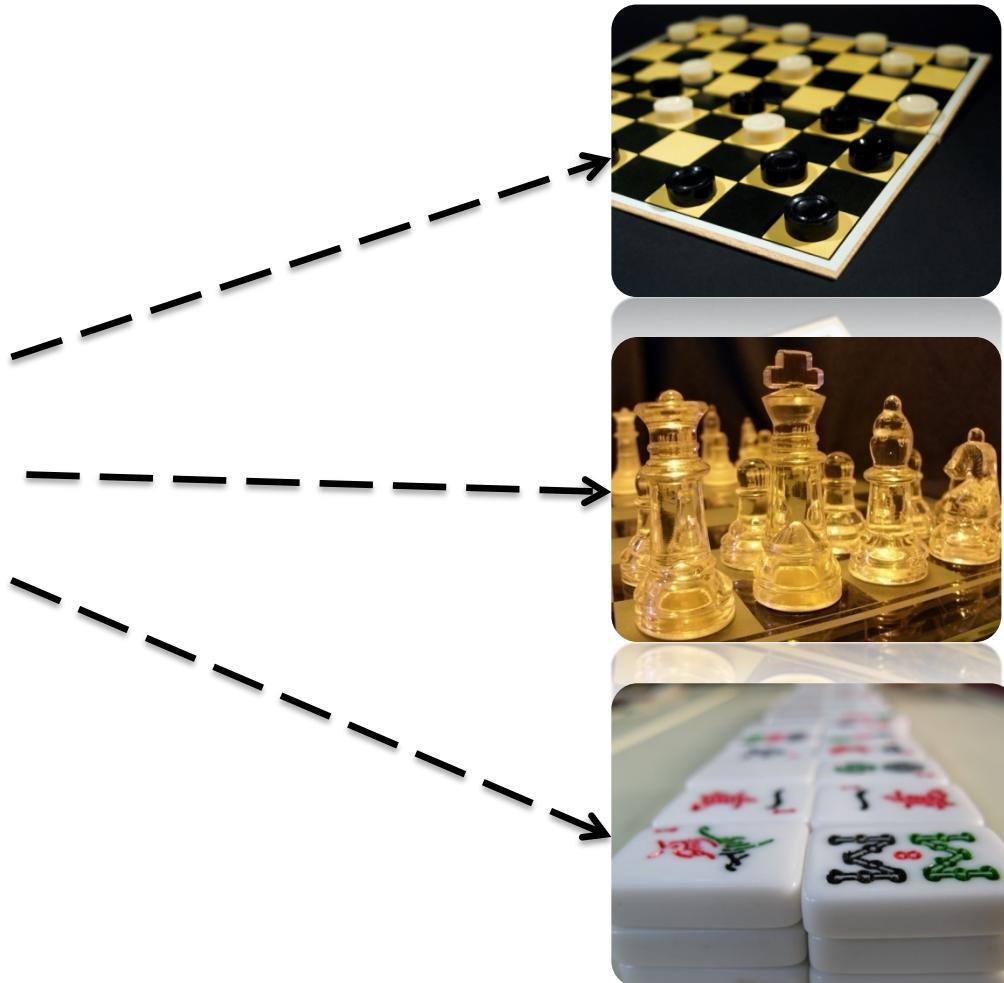
# Wzorzec Template Method

Wzorce behawioralne

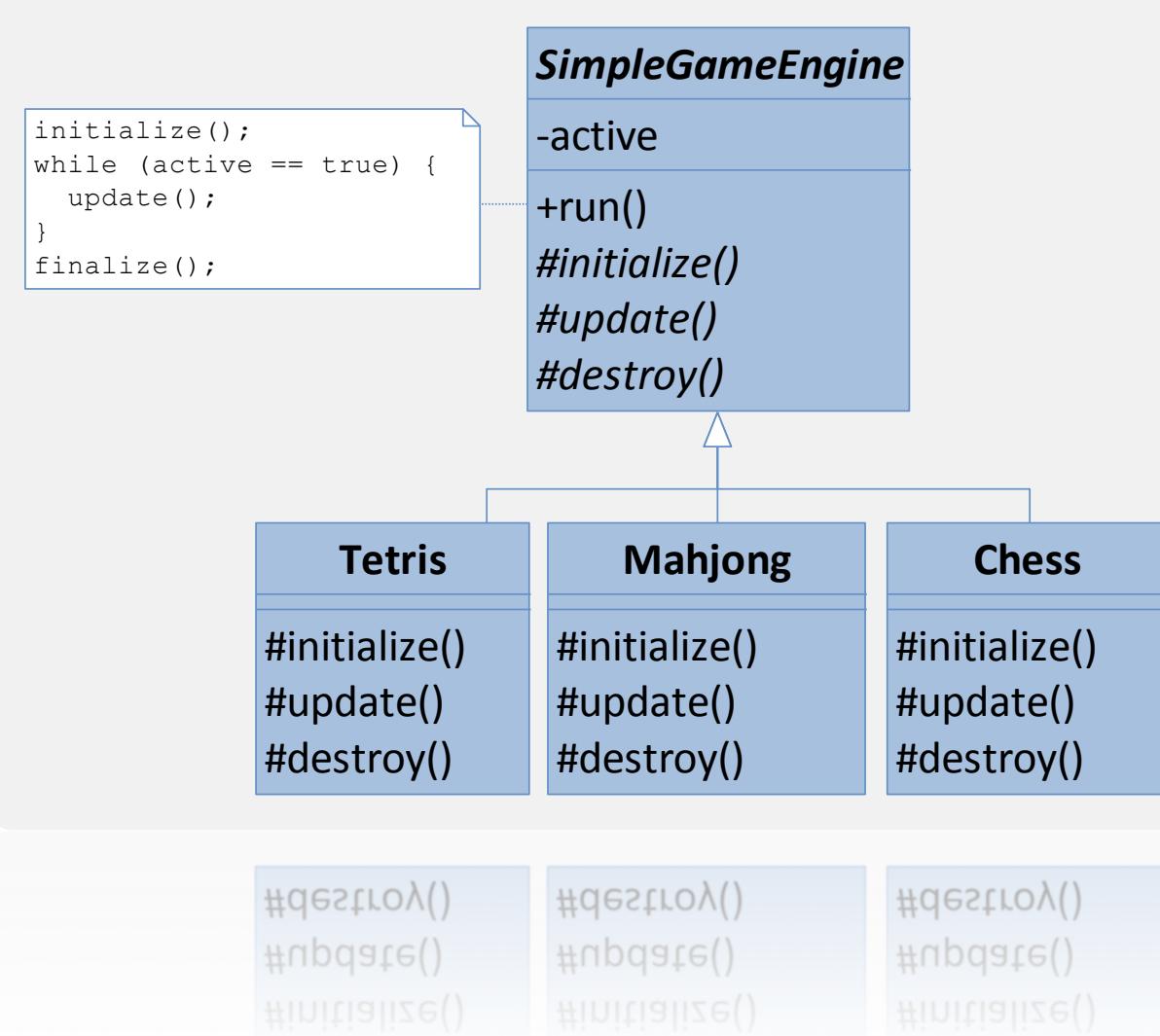
# Przykład

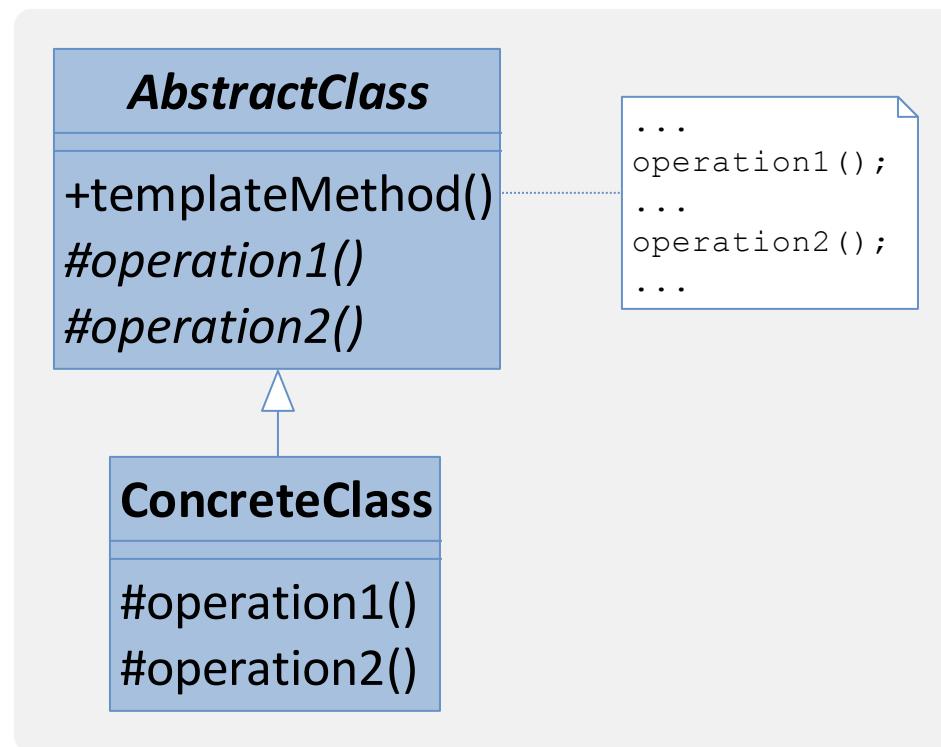


Game Engine



## 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.

# Przykład implementacji *JdbcTemplate*

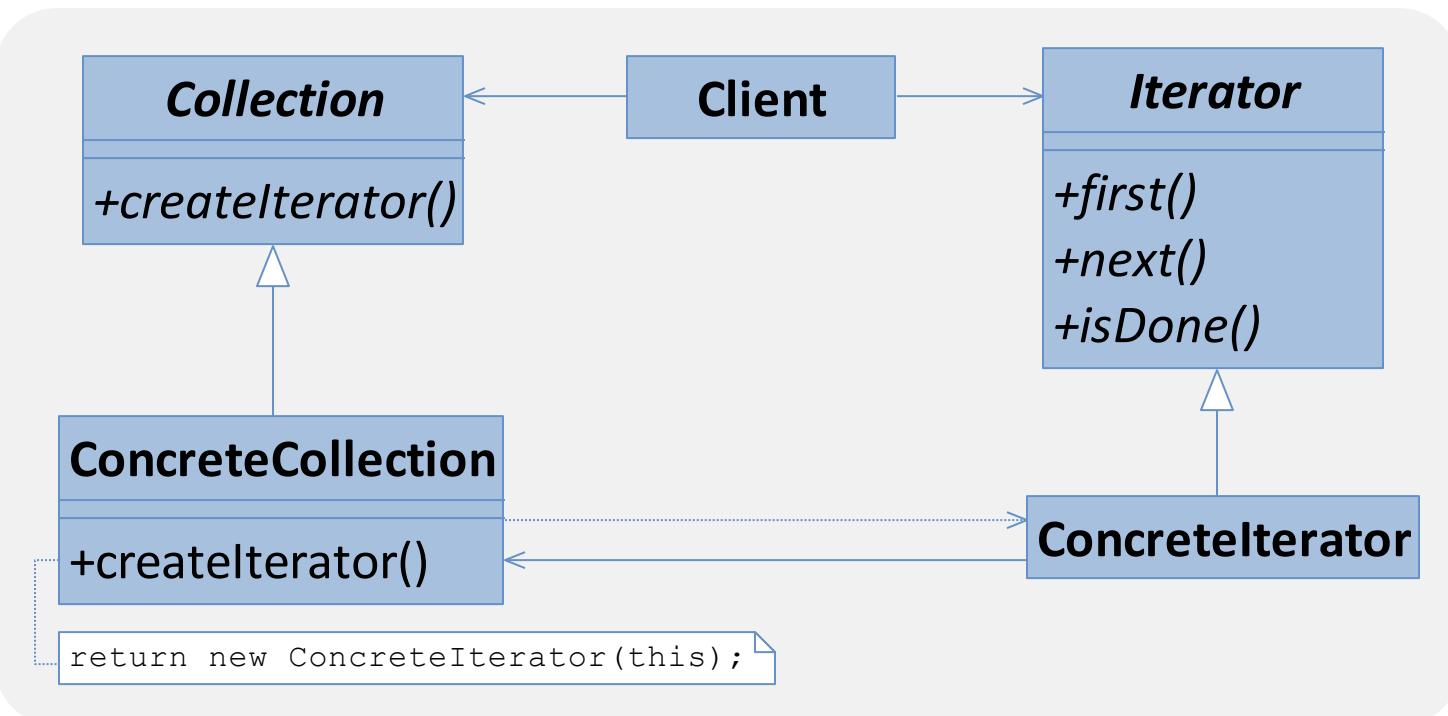
```
public abstract class JdbcQueryTemplate {  
    private DataSource dataSource;  
    public JdbcQueryTemplate( DataSource dataSource ) {  
        this.dataSource = dataSource;  
    }  
    public final ResultSet executeQuery() {  
        ResultSet resultSet = null;  
        Statement statement = createStatement();  
        try {  
            resultSet = statement.executeQuery( getCustomQuery() );  
            statement.close();  
        } catch ( SQLException e ) {  
            throw new DataAccessException( e );  
        }  
        return resultSet;  
    }  
    protected abstract String getCustomQuery();  
    protected final Statement createStatement() {  
        // open connection  
        // create and return statement  
    }  
    public void close() {  
        // close connection  
    }  
}
```

```
JdbcQueryTemplate queryTemplate  
= new JdbcQueryTemplate( dataSource ) {  
    @Override  
    protected String getCustomQuery() {  
        return "select * from employees";  
    }  
};  
  
ResultSet resultSet  
= queryTemplate.executeQuery();  
  
// extract resultSet;  
  
queryTemplate.close();
```

# 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

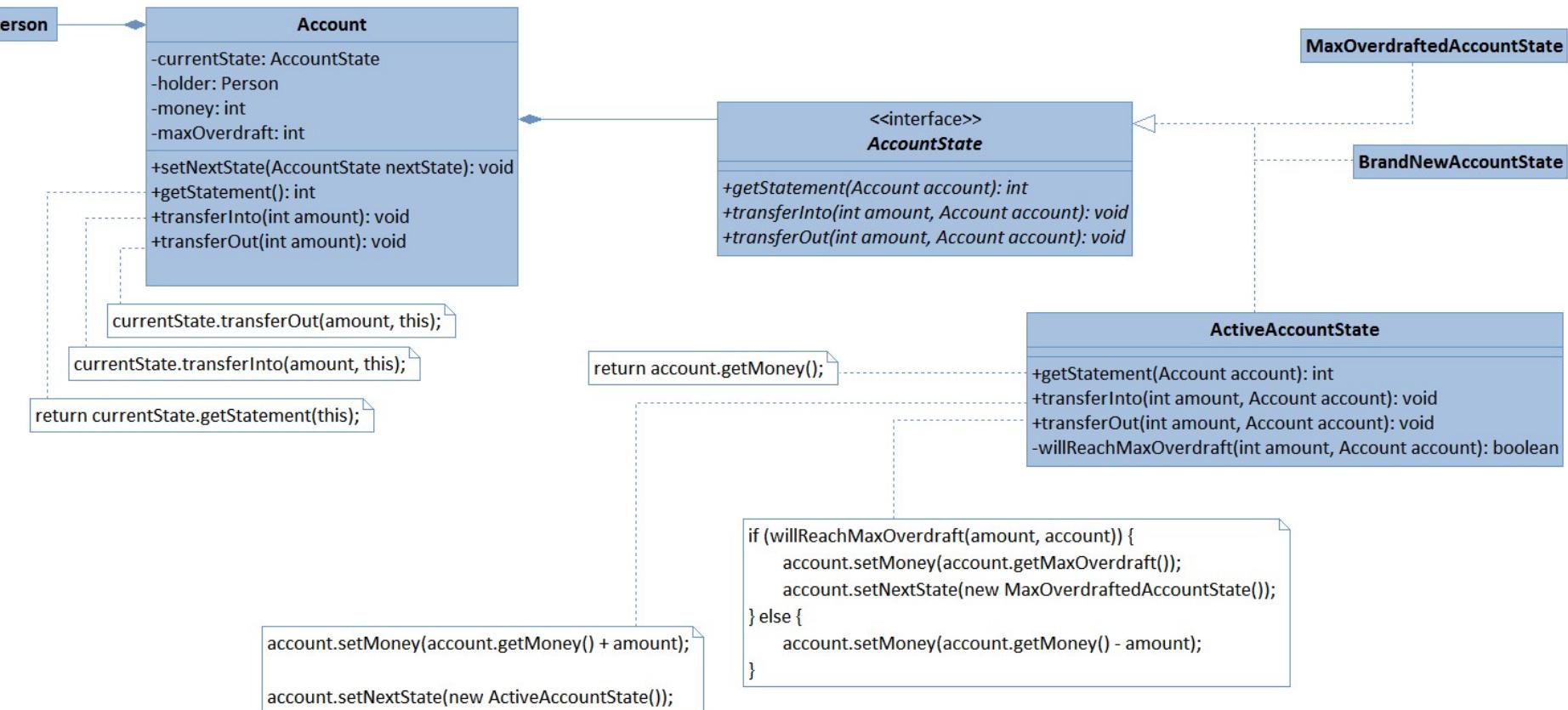
- # Wzorzec *Iterator* umożliwia zaimplementowanie różnych sposobów przehodzenia skomplikowanych kolekcji czy drzew obiektów.
- # Wyodrębnienie obiektu *Iterator* upraszcza interfejs klasy *Collection*.
- # Jednocześnie kilka obiektów typu *ConcretIterator* może przehodzić jedną kolekcję.

# bns it Wzorzec State

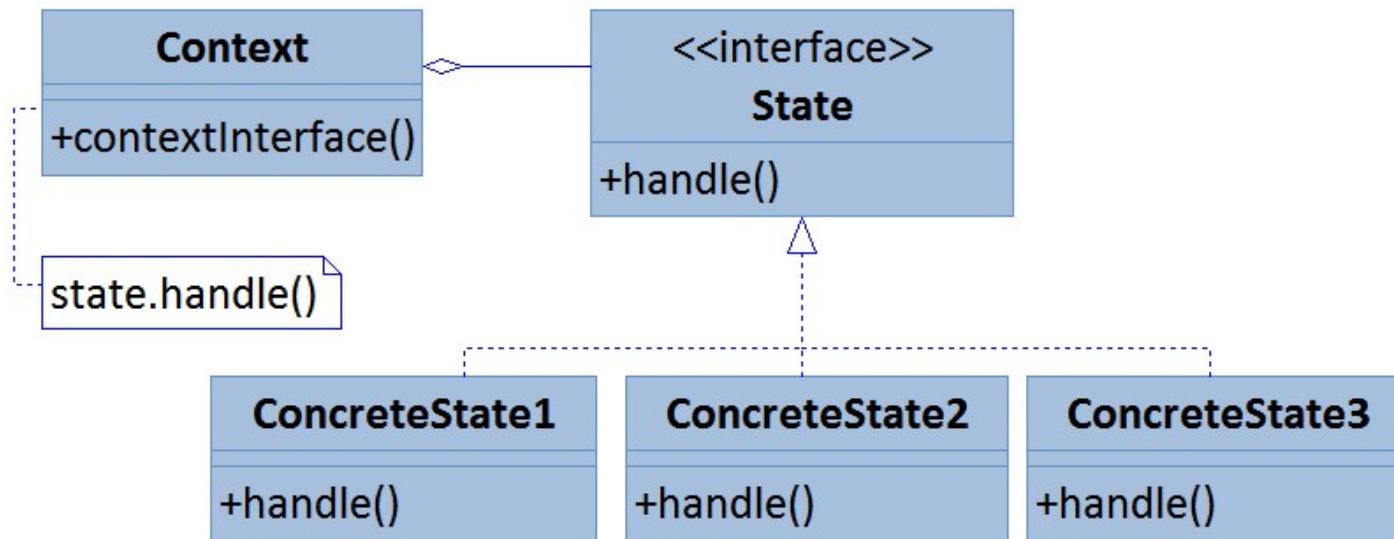
Wzorce behawioralne

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

# Przykład



# Struktura

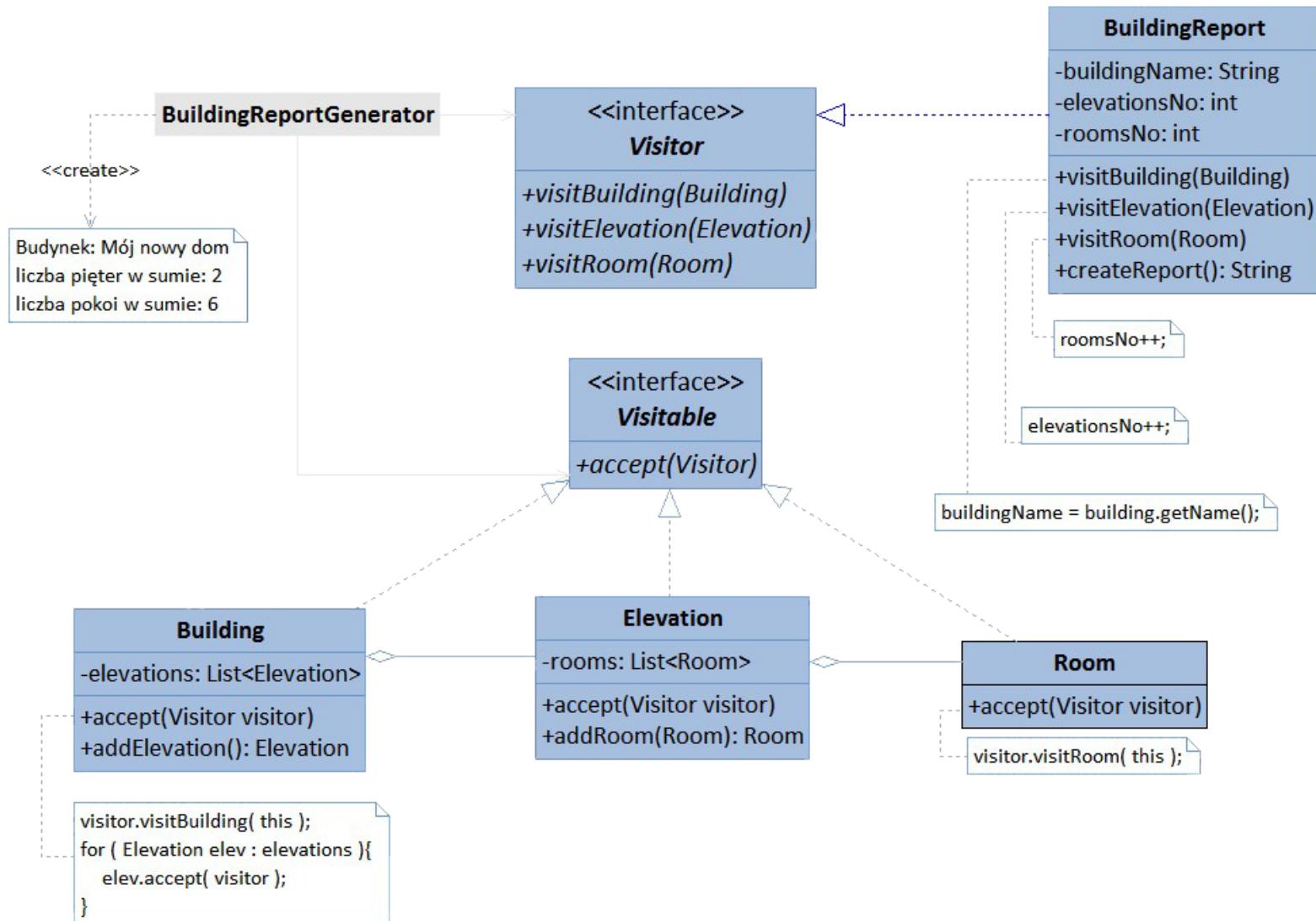


- # Wzorzec *State* zapewnia łatwość kontroli nad aktualnym stanem obiektu.
- # Pozwala uniknąć wielu nieczytelnych instrukcji warunkowych, które sprawdzały wartości zmiennych.
- # Specyficzne zachowania są podklasami klasy *State*, więc można je łatwo dodawać i edytować.
- # Jawność przejść pomiędzy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

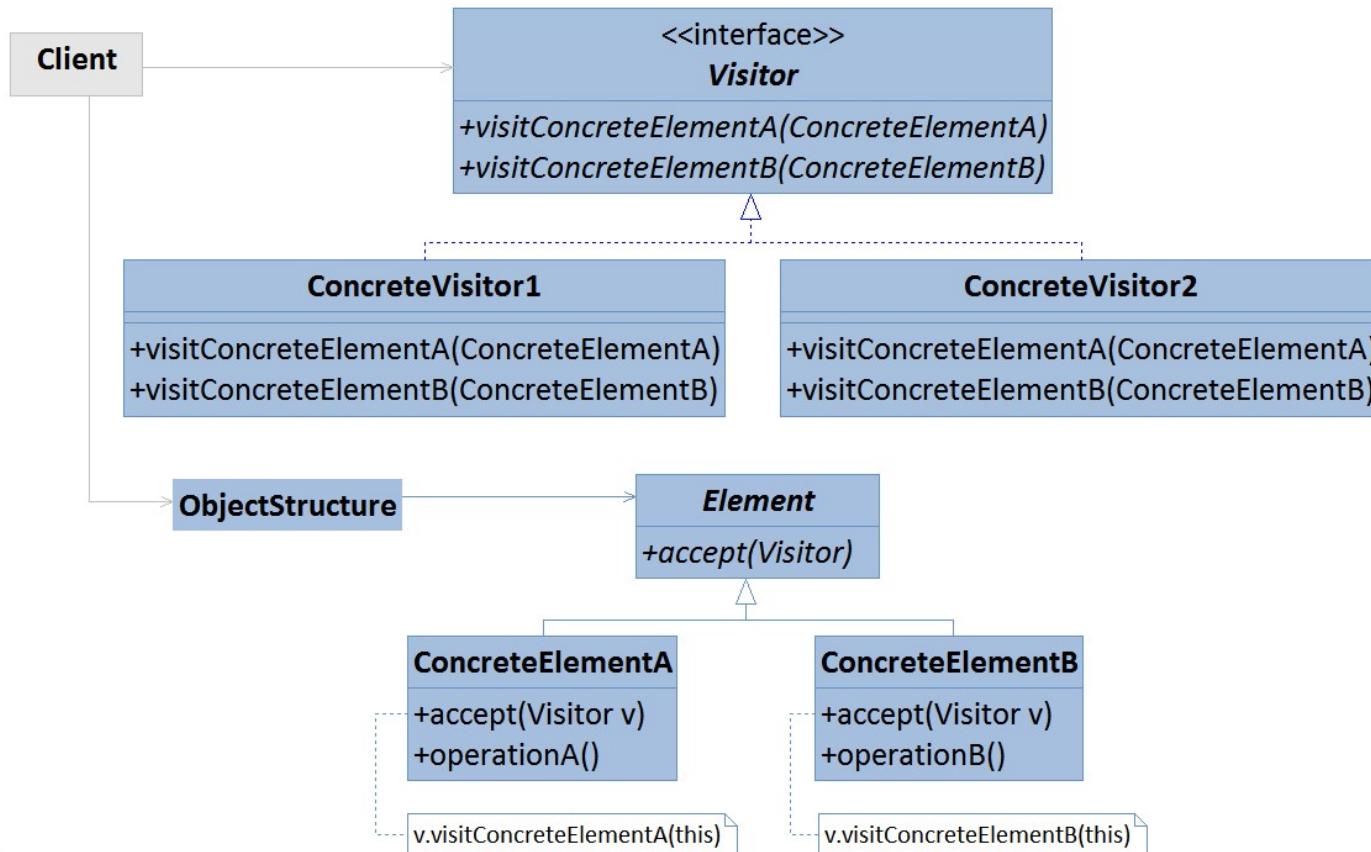
bns it} #} Wzorzec Visitor  
Wzorce behawioralne

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

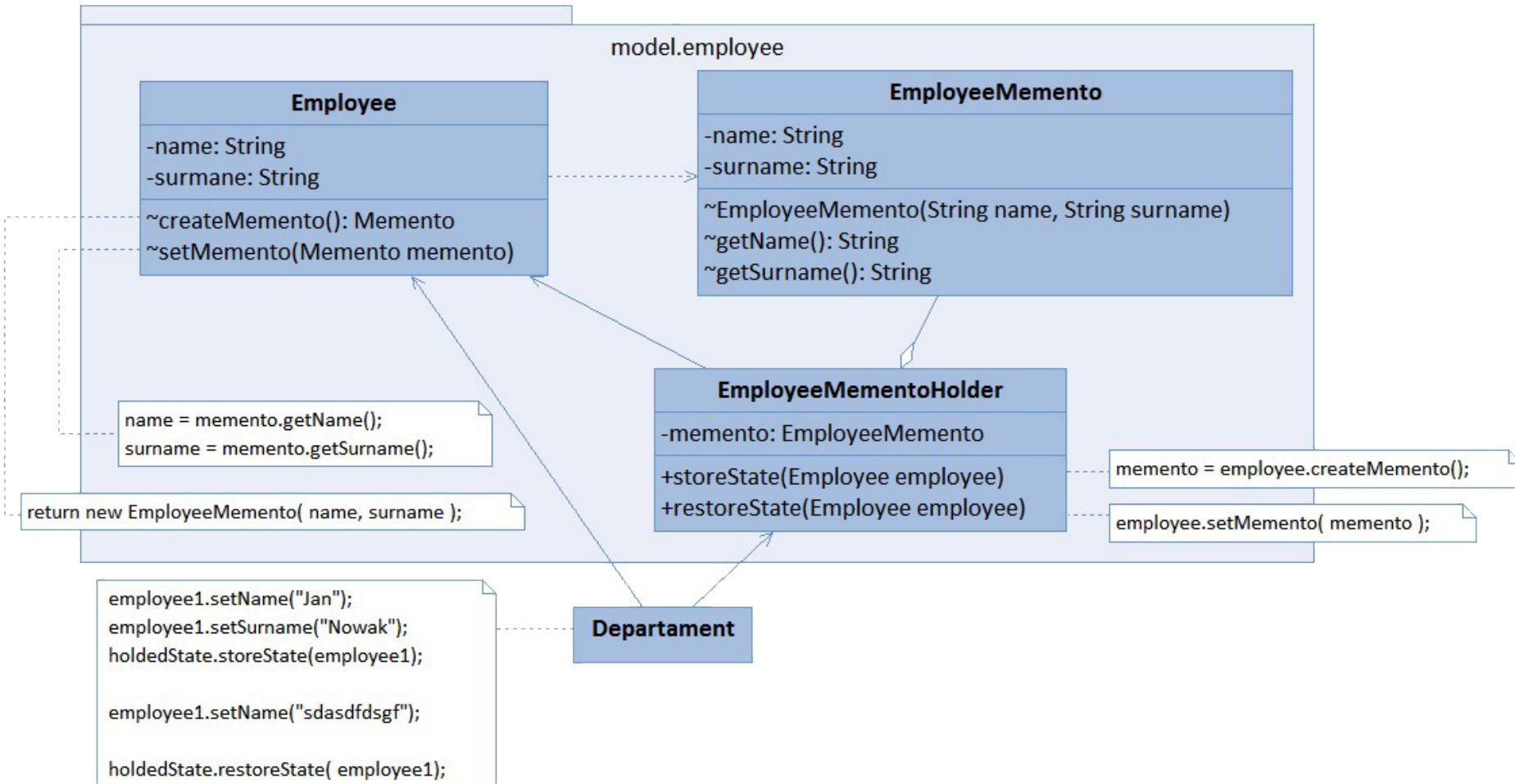


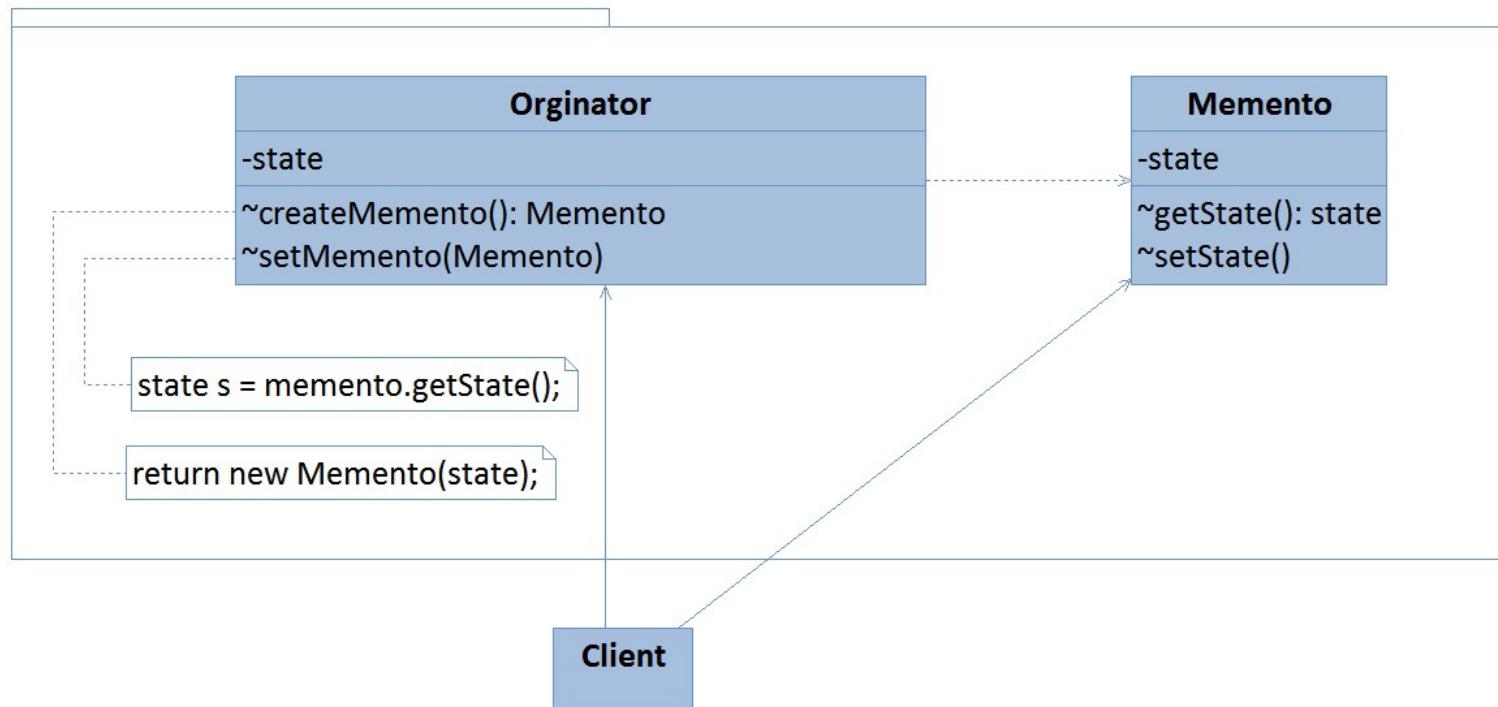
- # Wzorzec *Visitor* zapewnia łatwość dodawania nowych operacji.
- # Grupowanie powiązanych ze sobą operacji w podklasach odwiedzających.
- # Odwiedzanie całej hierarchii klas, bez względu na rodzaje powiązań obiektów.
- # Kumulowanie stanu w konkretnych wizytatorach.
- # Prawdopodobieństwo naruszenia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

bns it }    # }    Wzorzec Memento  
Wzorce behawioralne

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

## Przykład





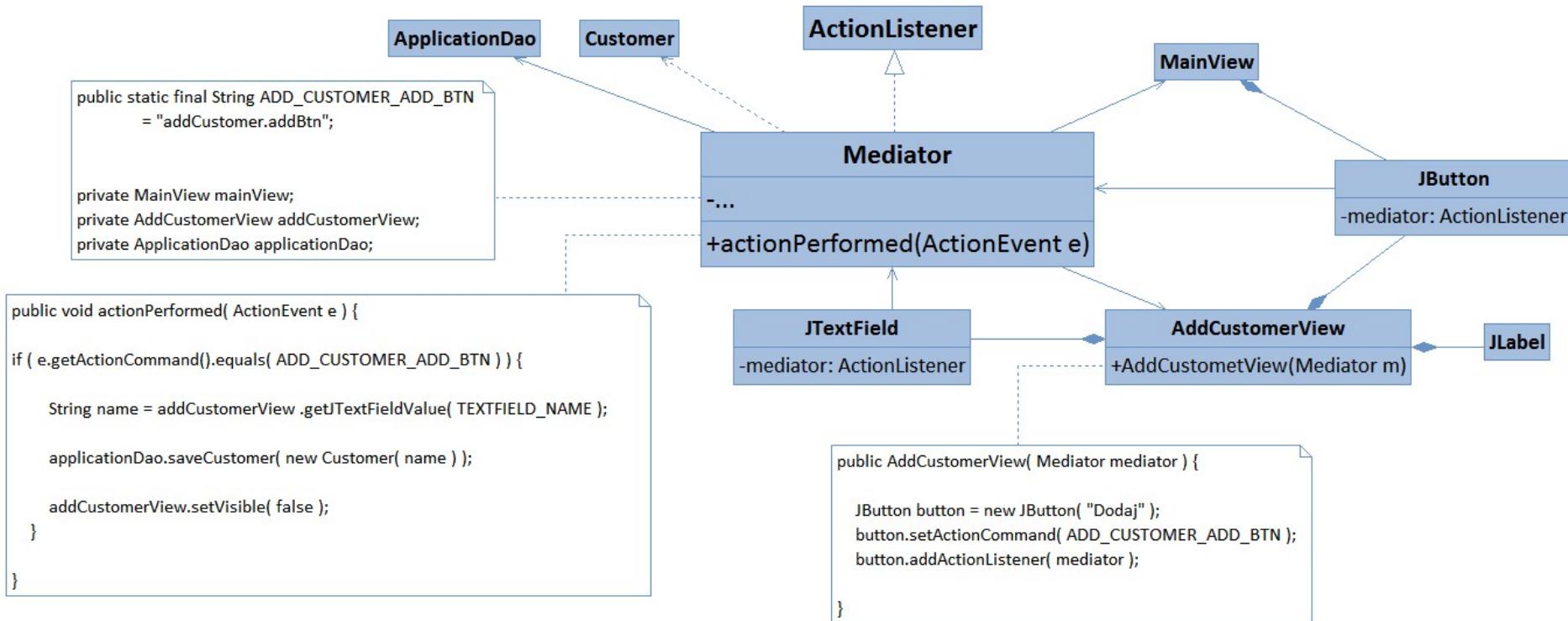
- # Wzorzec *Memento* zapewnia pewność nienaruszalności enkapsulacji, podczas zapamiętywania stanu obiektu.
- # Trzeba zagwarantować, że tylko wybrani klienci mają dostęp do zapisanego stanu, co może okazać się trudne.
- # Tworzenie i obsługa zapisanych stanów może być kosztowne .

bns it} #} Wzorzec Mediator  
Wzorce behawioralne

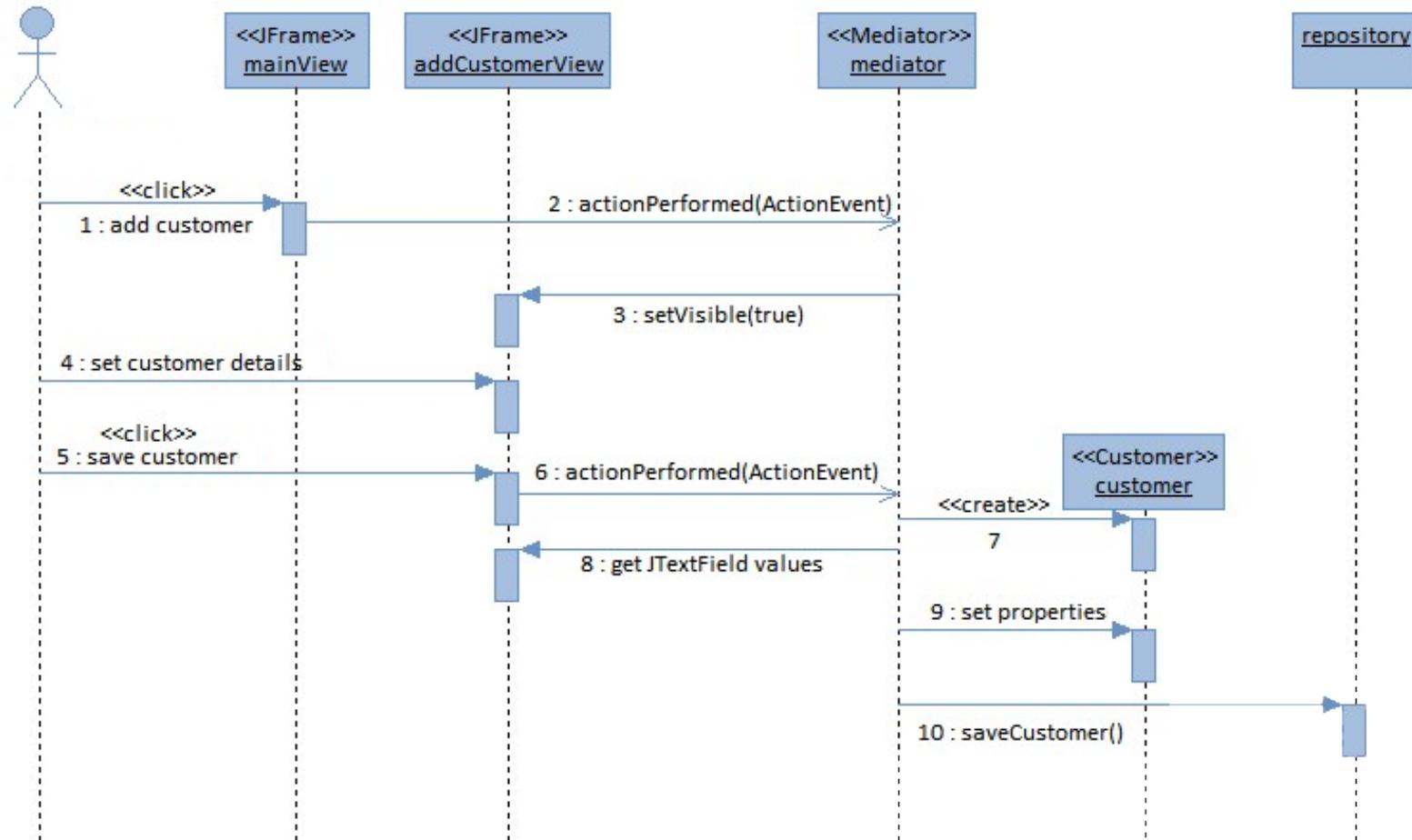
Wzorzec *Mediator* separuje obiekty od systemu.

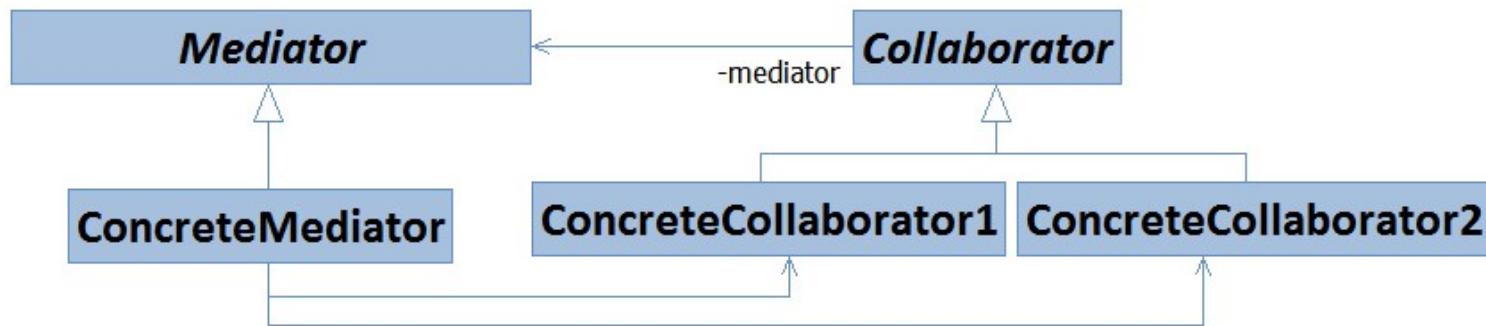
Tworzy obiekt enkapsulujący informacje o współdziałaniu obiektów w strukturze charakteryzującej się złożonością procedur komunikacji i sterowania.

## Przykład



## Przykład



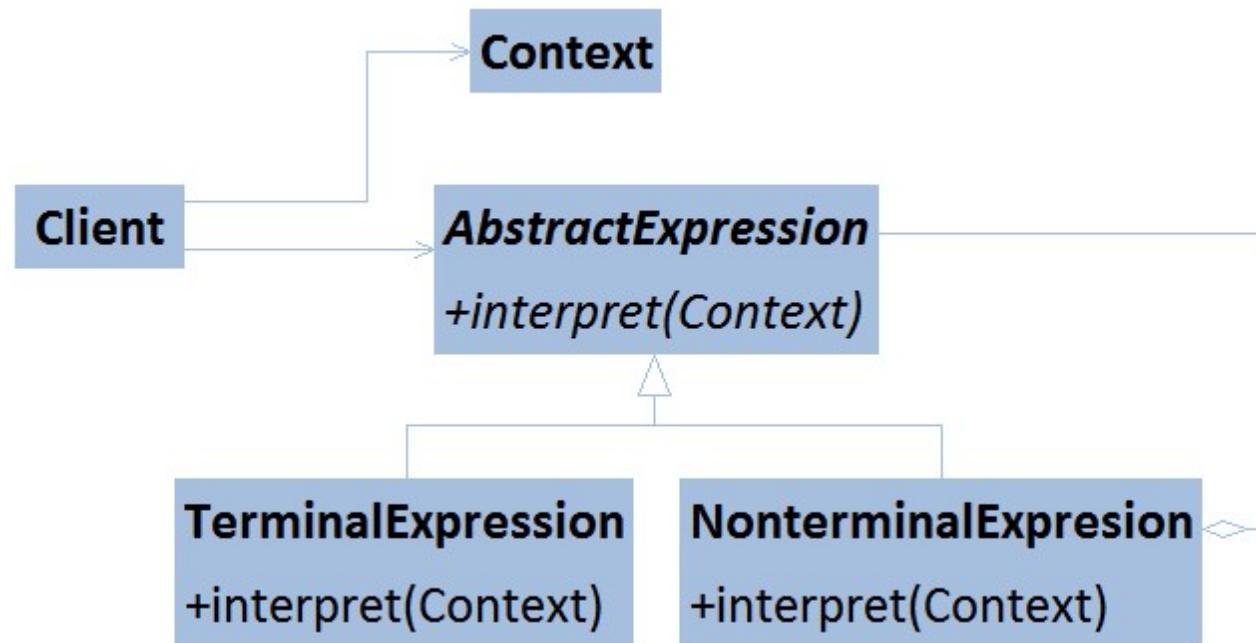


- # Wzorzec *Mediator* zwiększa szanse ponownego wykorzystania obiektów, z którym współpracuje, bo zapewnia ich separację od systemu.
- # Upraszcza system zamieniając związki wiele-do-wiele na jeden-do-wiele.
- # Skupia w jednym miejscu całą logikę sterowania.
- # Niedopracowany projekt może uczynić 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

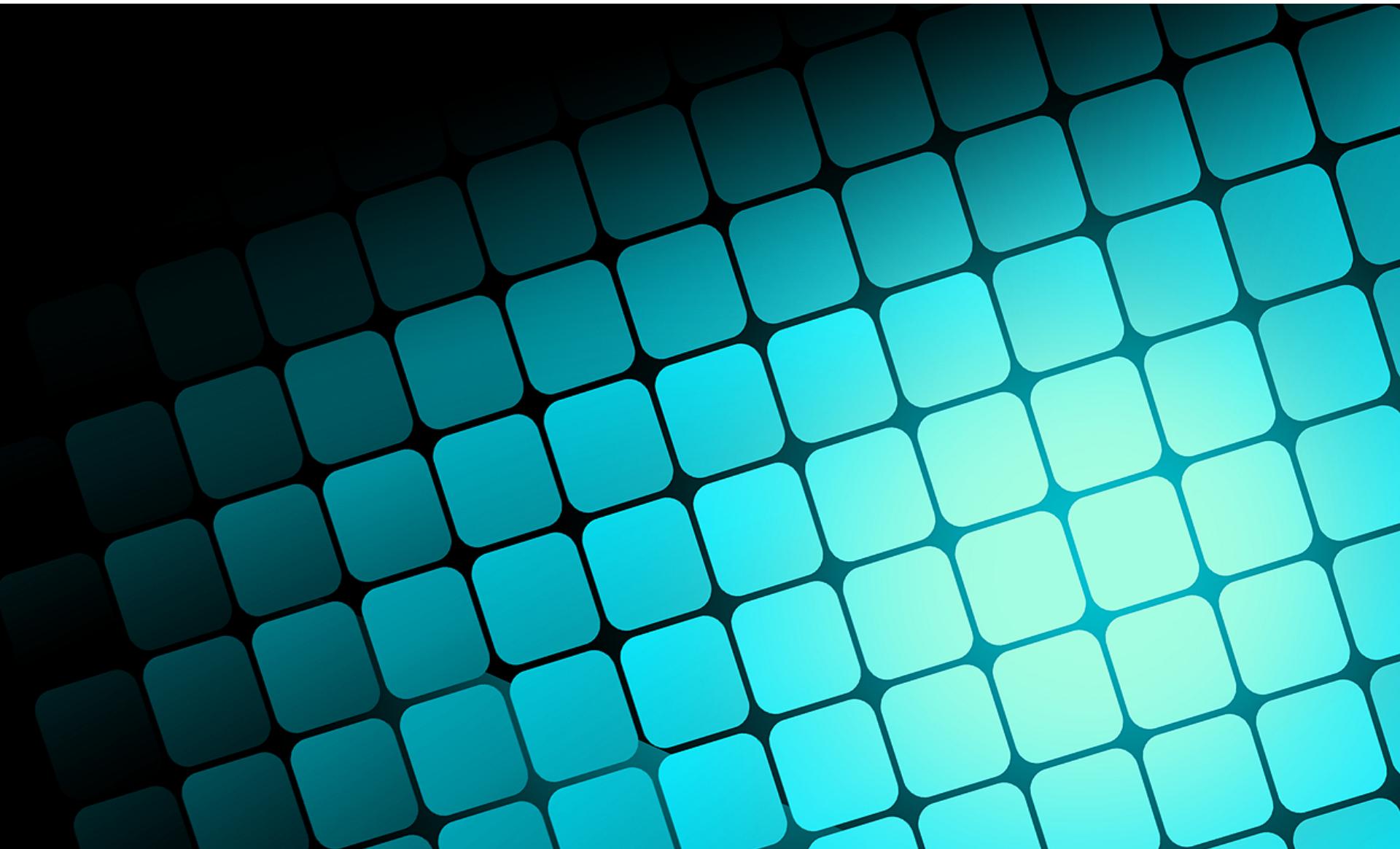
Wzorzec *Interpreter* definiuje opis gramatyki języka i umożliwia jej interpretacje.



- # 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

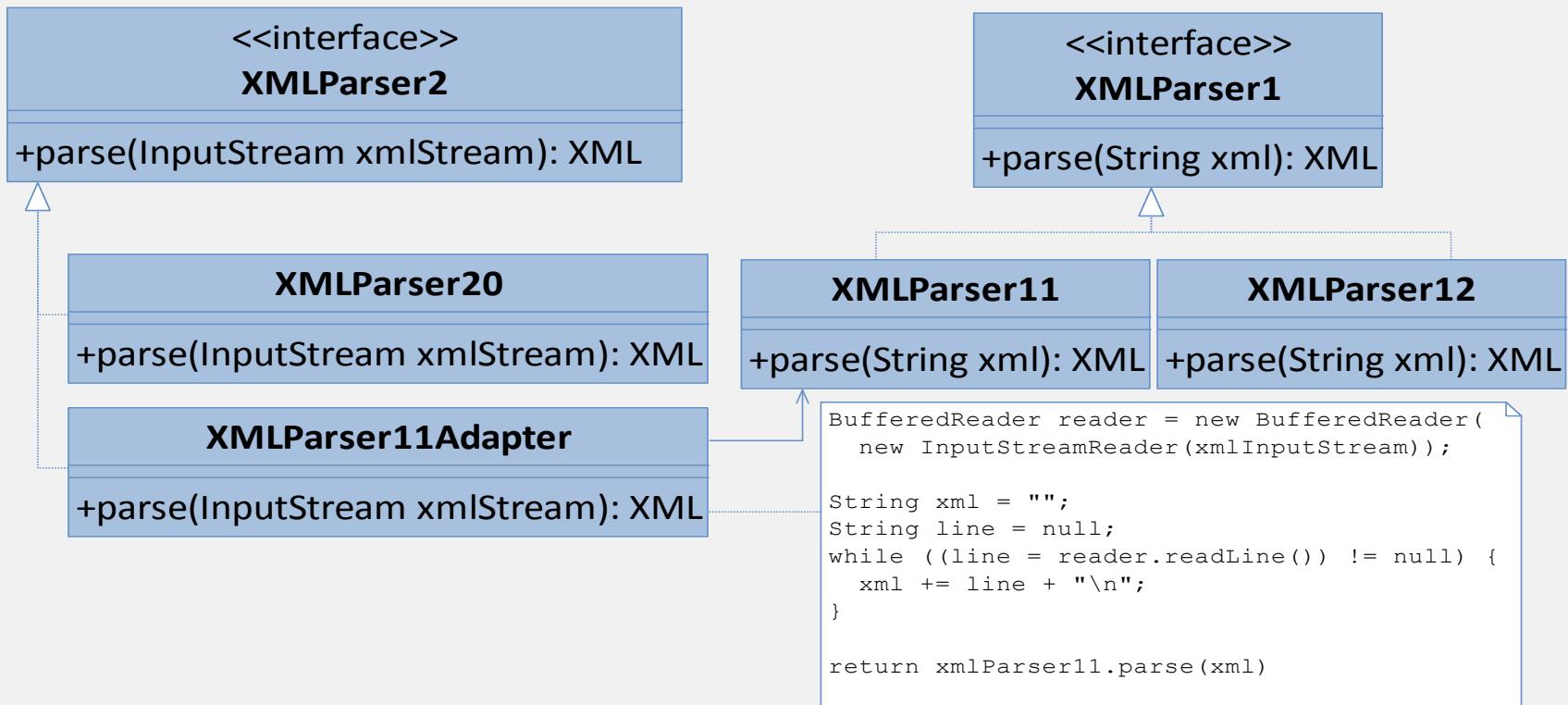
## Przykład

**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";
}

return xmlParser11.parse(xml)
```

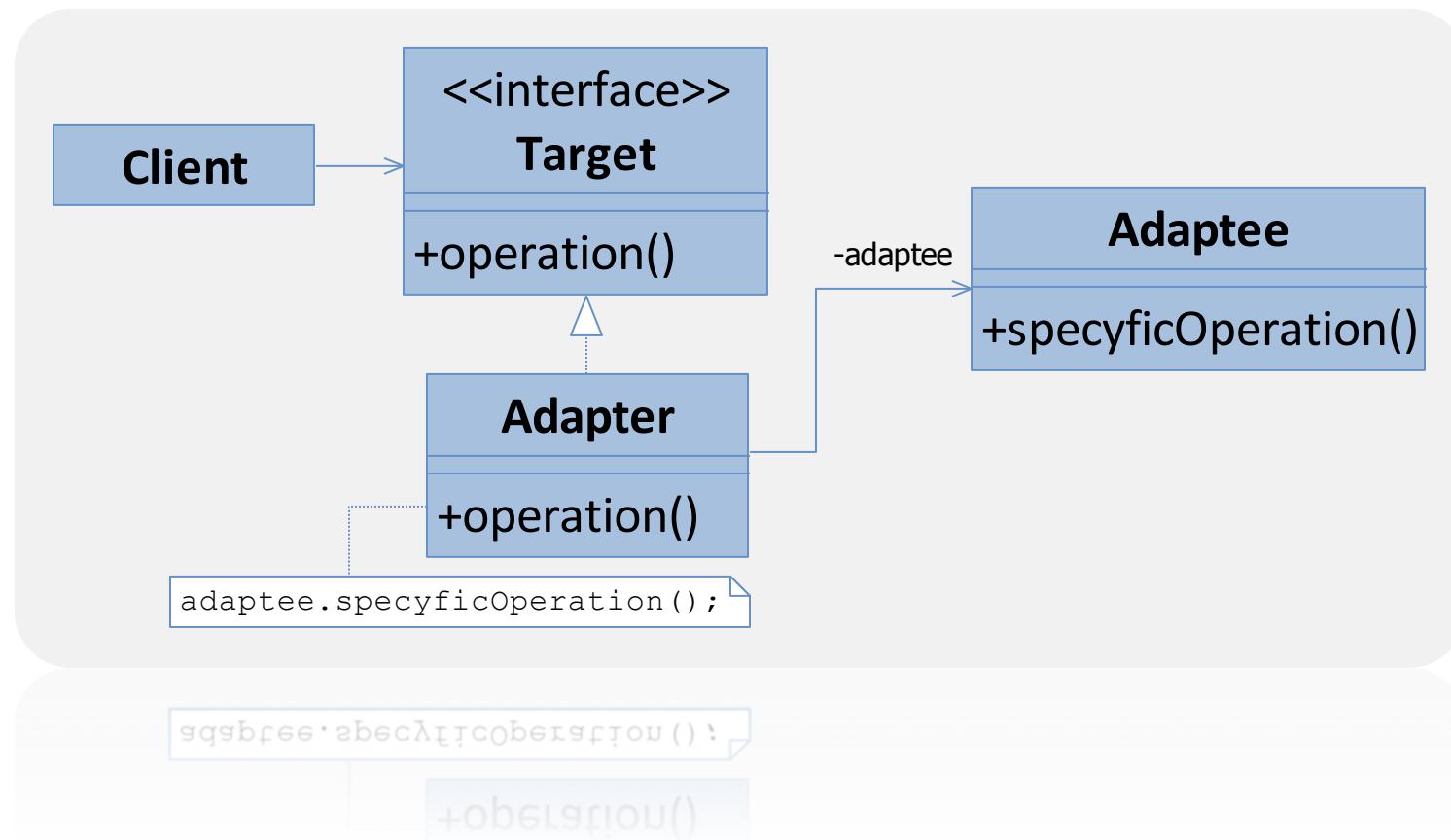
# bns it} Kod przykładu

```
public interface XMLParser2 {  
    public XML parse(InputStream xmlStream)  
        throws IOException;  
}
```

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

```
public class XMLParser11Adapter implements XMLParser2 {  
    private XMLParser11 xmlParser11 = new XMLParser11();  
    public XML parse(InputStream xmlStream)  
        throws IOException {  
        BufferedReader reader = new BufferedReader(  
            new InputStreamReader(xmlStream));  
        String xml = "";  
        String line = null;  
  
        while ((line = reader.readLine()) != null) {  
            xml += line + "\n";  
        }  
        return xmlParser11.parse(xml);  
    }  
}
```

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

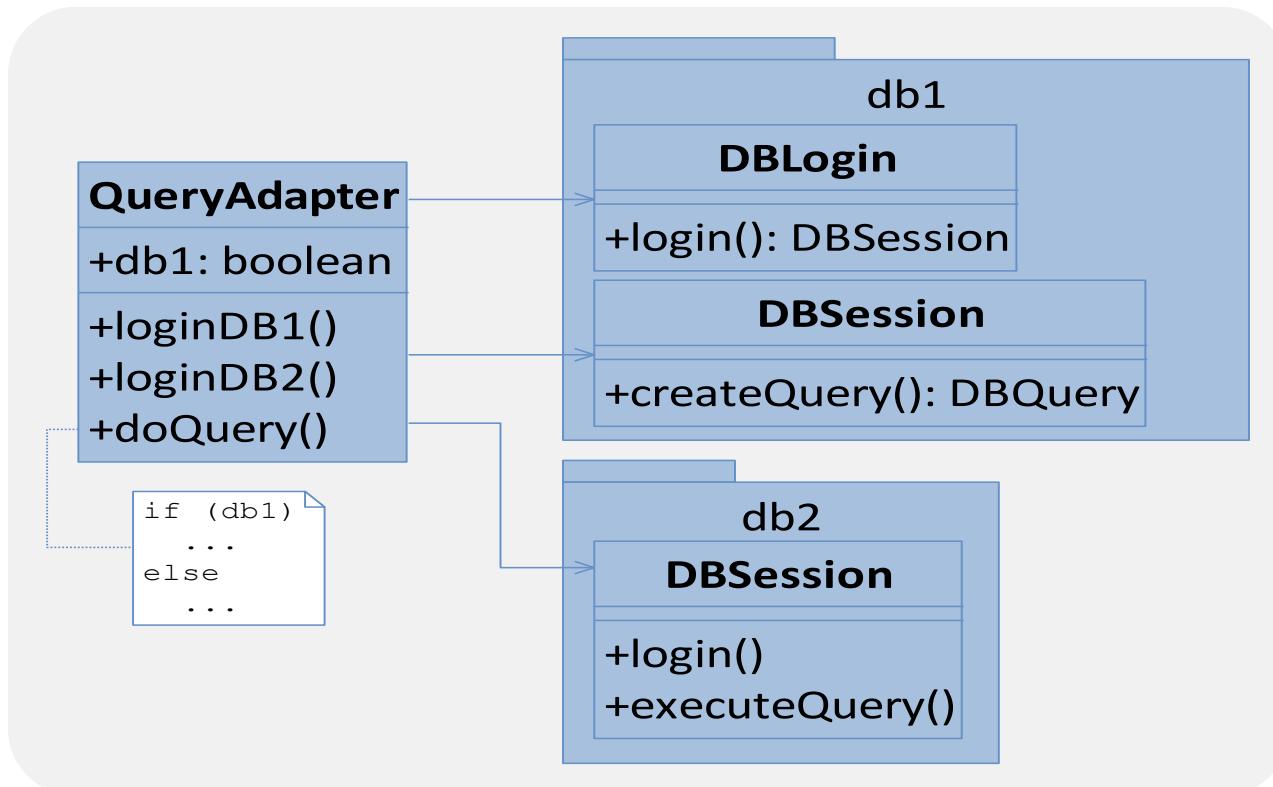


Wzorzec *Adapter* przekształca interfejs klasy do postaci, której oczekują klienci.

- # *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.

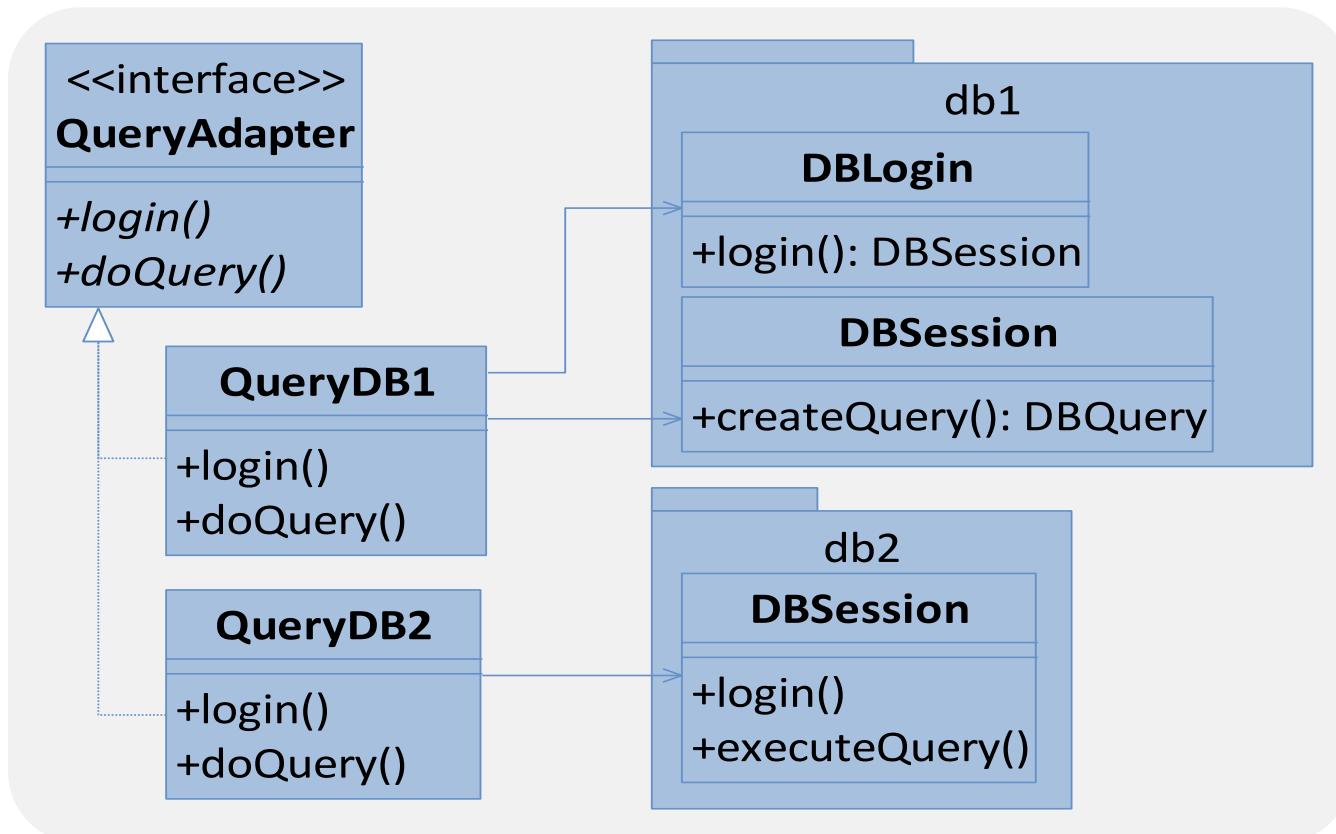
- # Włączanie klasy do systemu, który oczekuje od niej innego interfejsu.
- # Adaptowanie zewnętrznej biblioteki do własnych interfejsów.
- # Biblioteka okienkowa *wxWidgets* – adaptuje biblioteki okienkowe z różnych systemów do wspólnego interfejsu.
- # Biblioteka *commons-logging* adaptuje do jednego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

# Refaktoryzacja: *Extract Adapter*



+executeQuery()  
+login()

# Refaktoryzacja: *Extract Adapter*



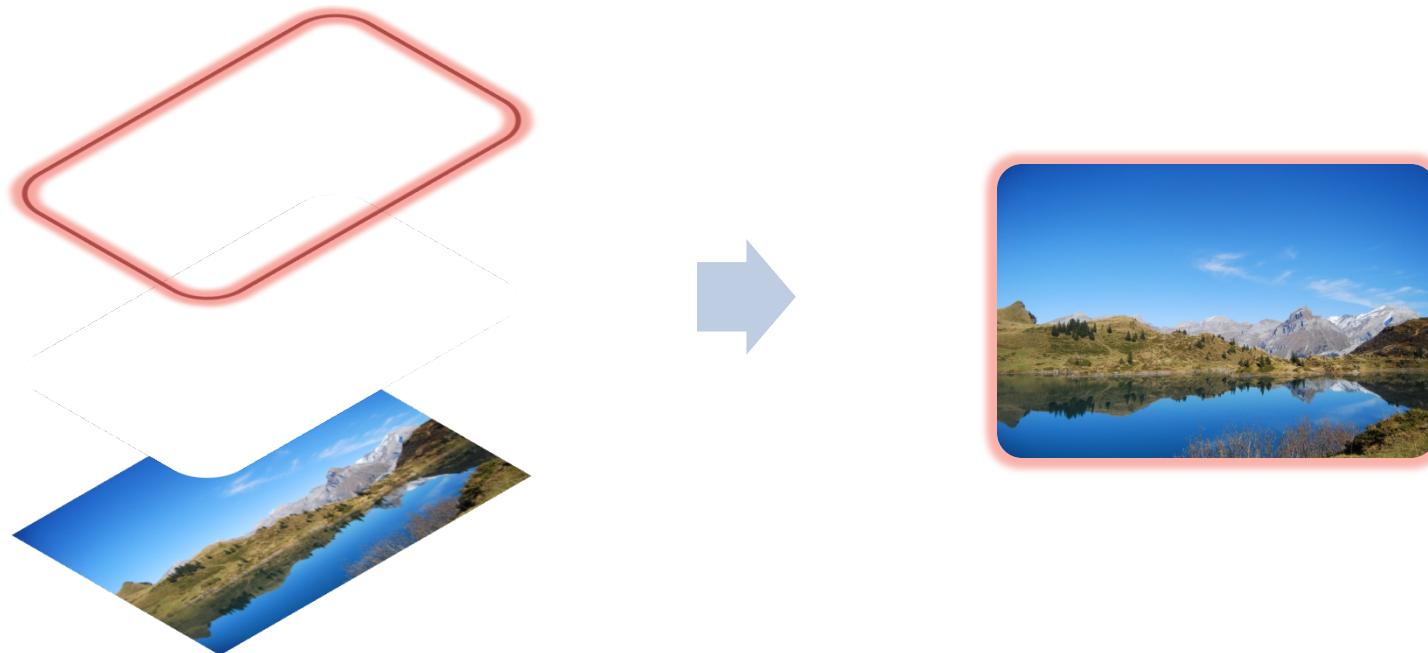
+doQuery()  
+login()

db1  
db2

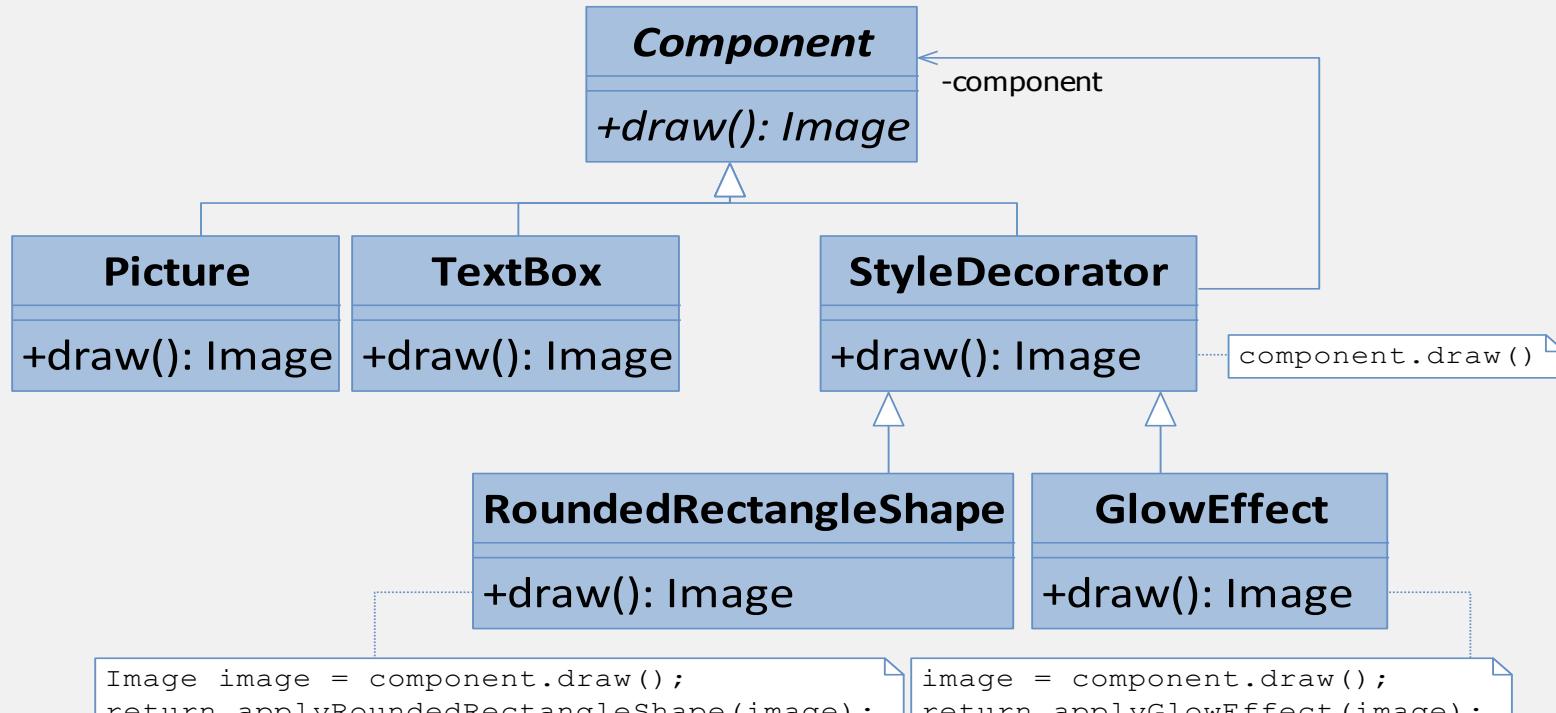
+executeQuery()  
+login()

bns it }    #} Wzorzec Decorator  
Wzorce strukturalne

## Przykład



## Przykład



```
Image image = component.draw();
return applyRoundedRectangleShape(image);
```

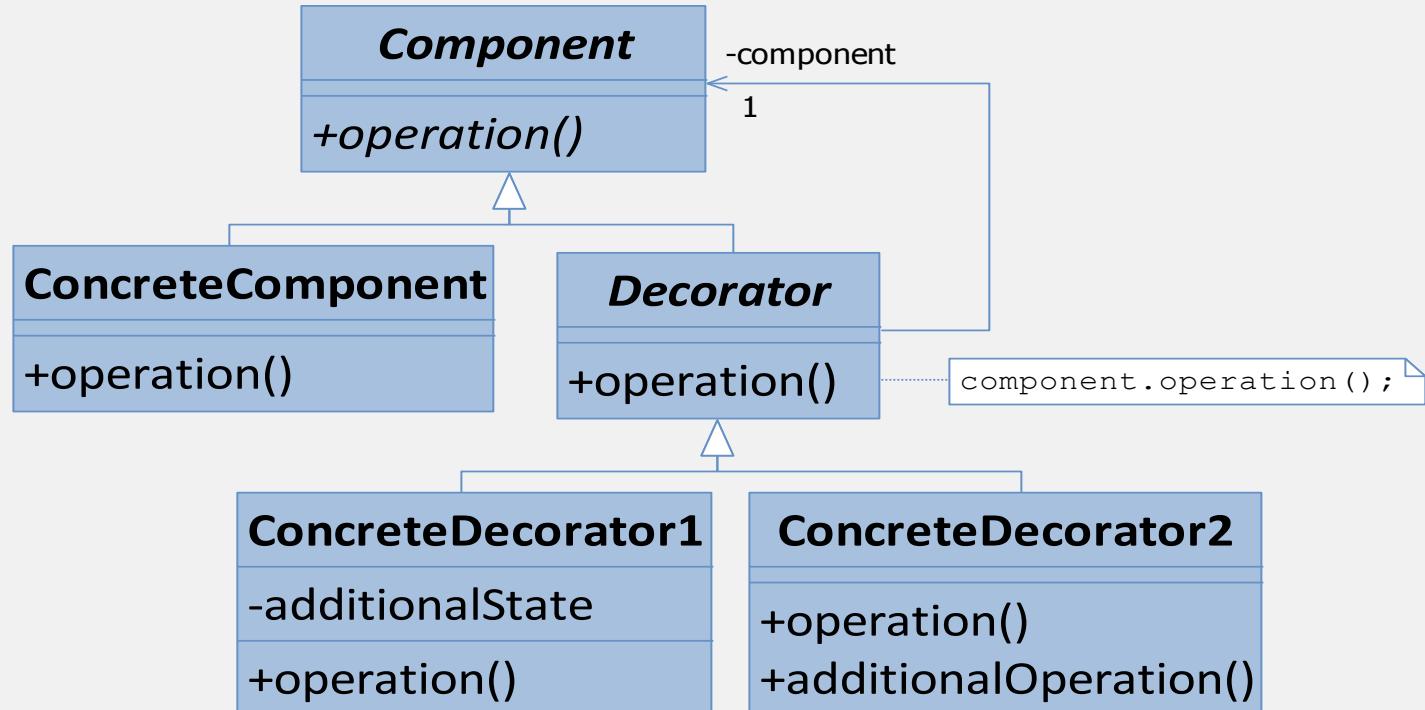
```
image = component.draw();
return applyGlowEffect(image);
```

reflexus əbbəjλəqomuñqəq्वेचरानम्जेश्वर्पे (र्मादे) :
र्मादे र्मादे = componenf·draw() :

reflexus əbbəjλəqomewētēc़ (र्मादे) :
र्मादे = componenf·draw() :

+draw(): Image

+draw(): Image



# Kod przykładu

```
public interface Component {  
    public Image draw();  
}  
  
public class Picture implements Component {  
    private Image image;  
    public Picture(String filename) {  
        image = new Image(filename);  
    }  
    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 abstract class StyleDecorator  
    implements Component {  
    protected Component component;  
    public StyleDecorator(Component component) {  
        this.component = component;  
    }  
}  
  
public class RoundedRectangleShape  
    extends StyleDecorator {  
    public Image draw() {  
        Image image = component.draw();  
        return applyRoundedRectangleShape(image);  
    }  
}  
  
public class GlowEffect  
    extends StyleDecorator {...}
```

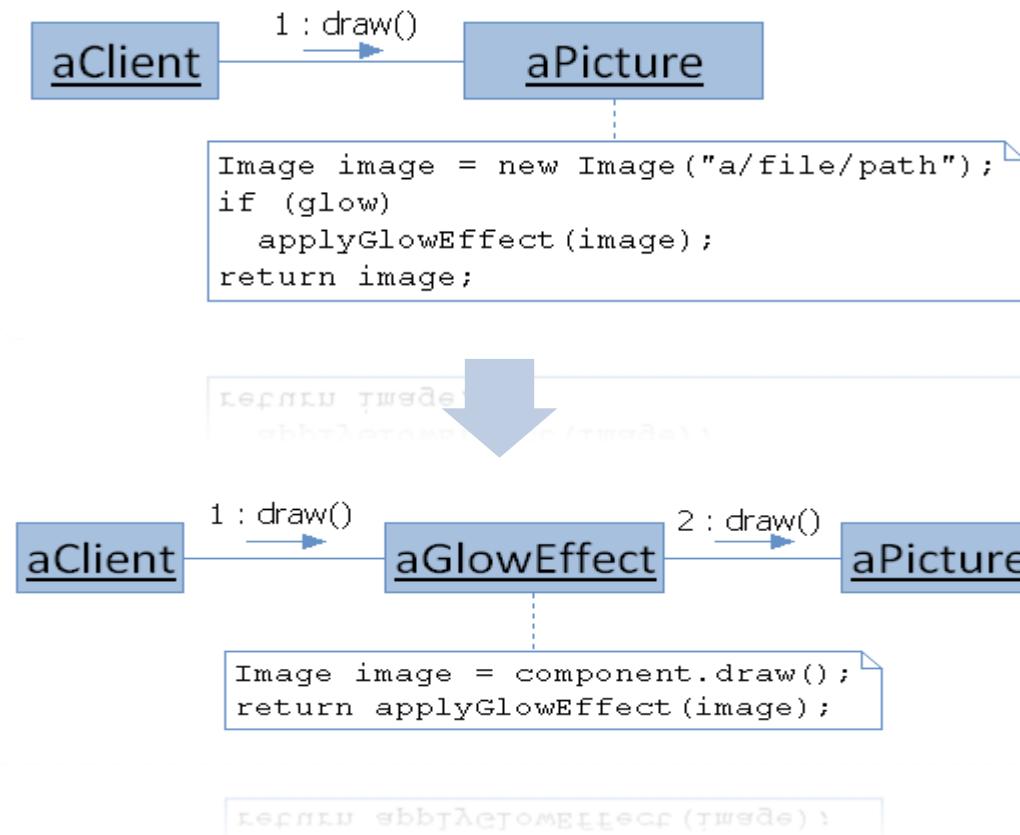
Wzorzec *Decorator* pozwala na dynamiczne rozszerzanie odpowiedzialności obiektu.

- # 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
- # Dekoratory strumieni w języku Java  
(*LineNumberInputStream*,  
*PushbackInputStream*...)
- # Sprawdzanie w czasie wykonania typów w kolekcjach (*EmployeeListDecorator*,  
*NotNullCollectionDecorator*)

# Refaktoryzacja: *Move Embellishment To Decorator*



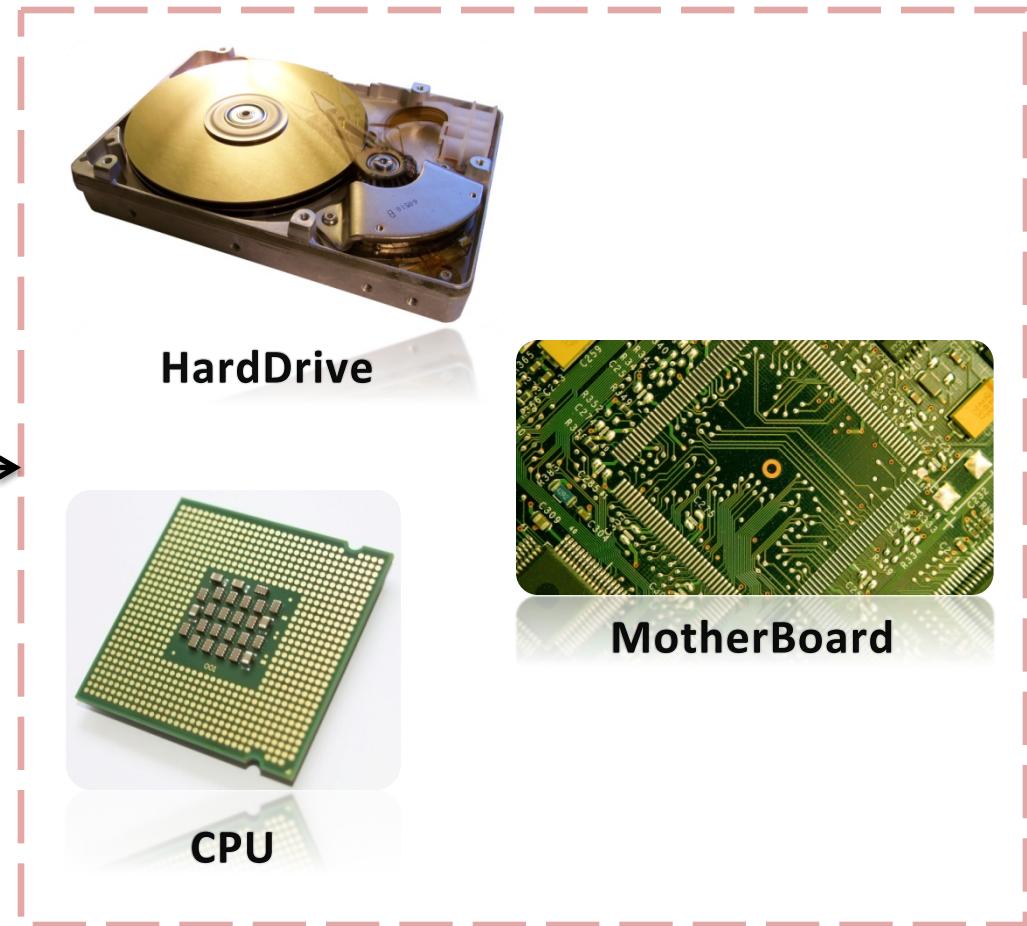
bns it} Wzorzec Facade  
Wzorce strukturalne

#}

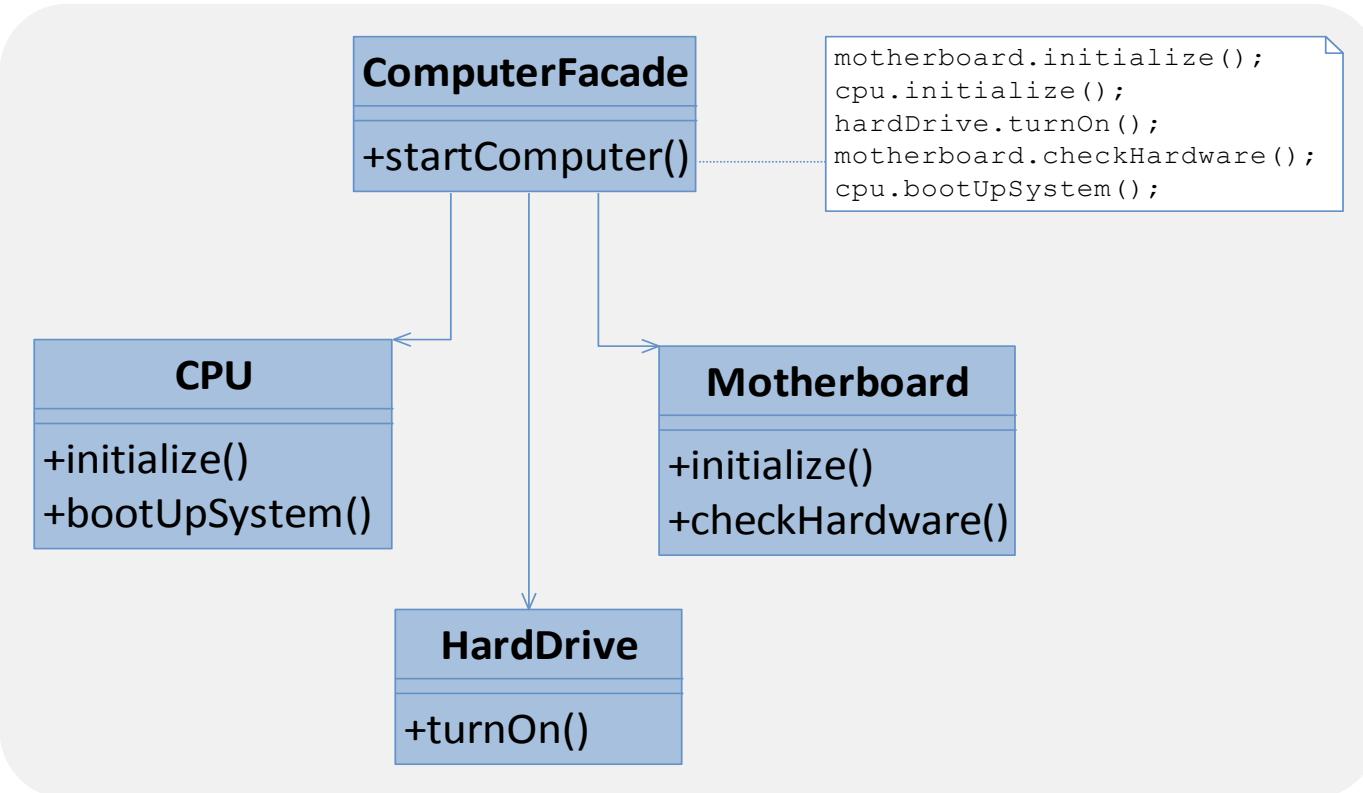
## Przykład



**start()**



## Przykład



+turnOn()  
HardDrive

# bns it} Kod przykładu

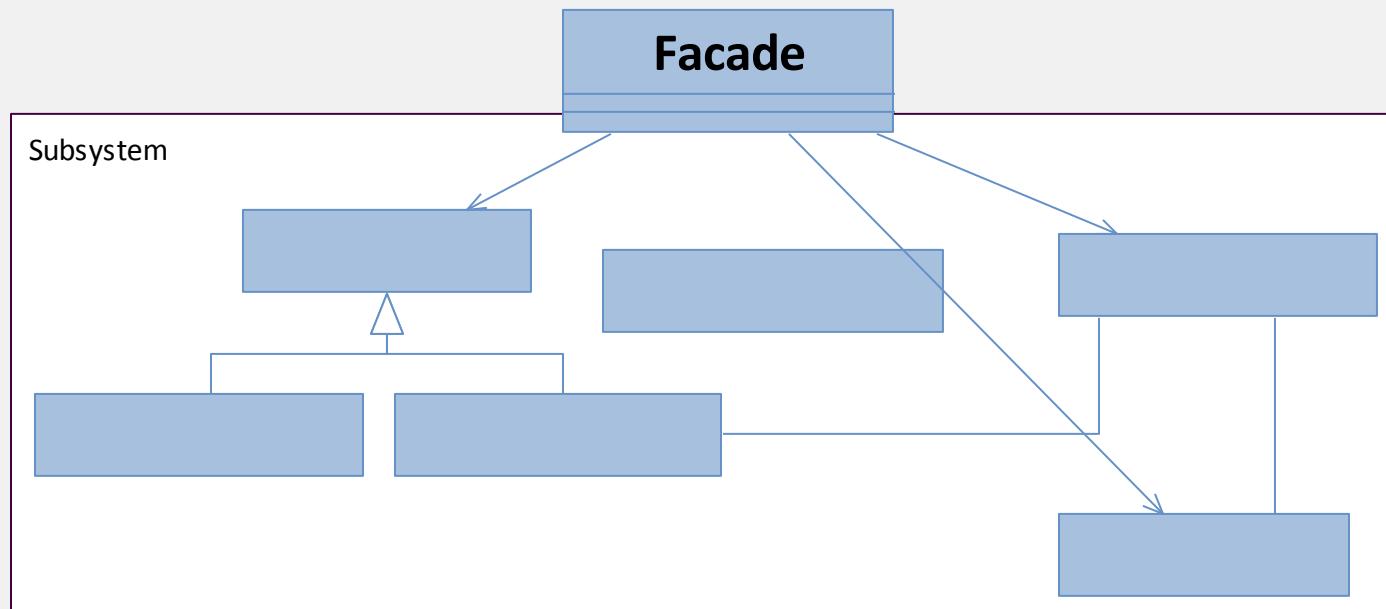
```
public class ComputerFacade {  
    private Motherboard motherboard;  
    private CPU cpu;  
    private HardDrive hardDrive;  
  
    public void startComputer() {  
        motherboard.initialize();  
        cpu.initialize();  
        hardDrive.turnOn();  
        motherboard.checkHardware();  
        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.

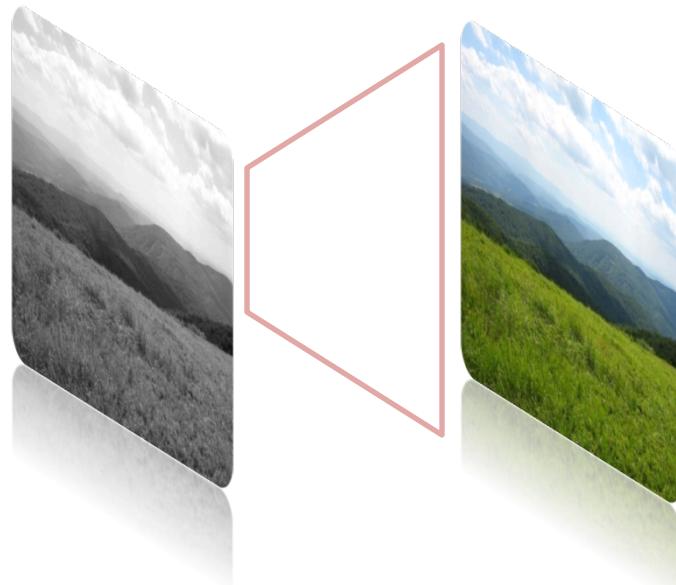
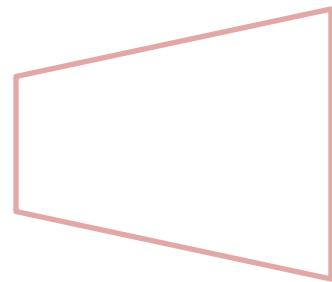
- # Ułatwienie korzystania ze skomplikowanego podsystemu
- # API biblioteki definiujące jej zunifikowany i uproszczony interfejs



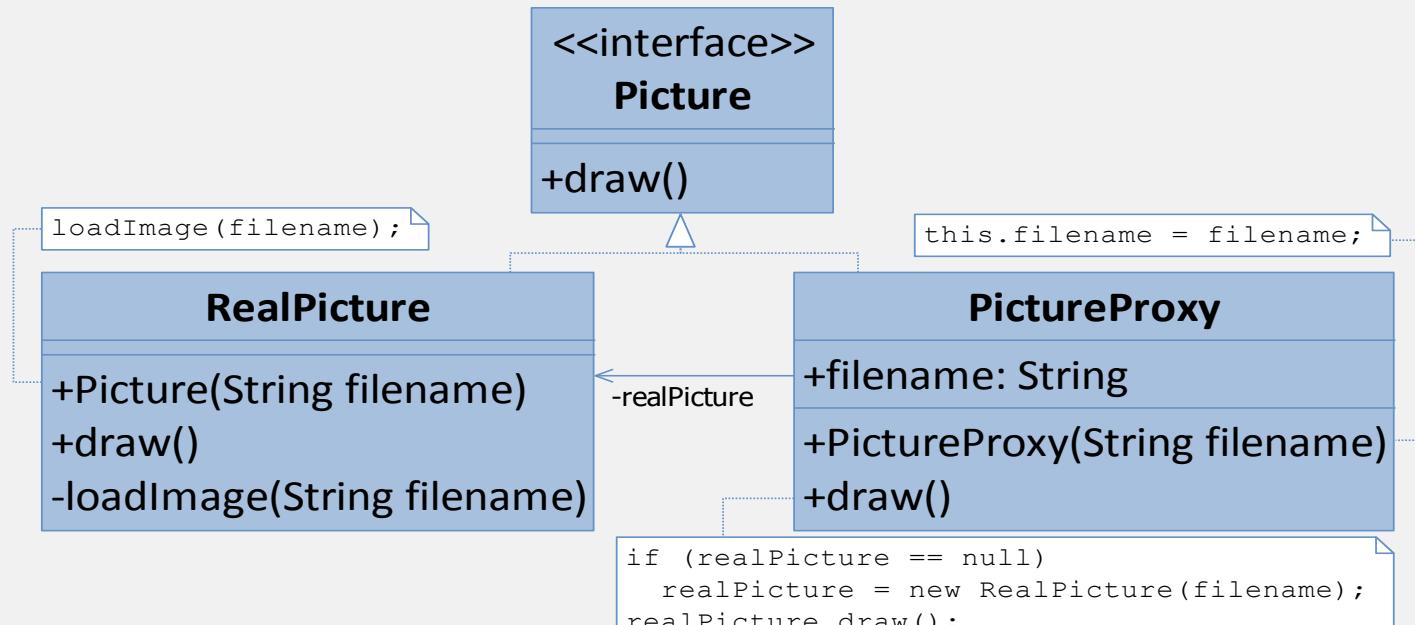
# Wzorzec Proxy

Wzorce strukturalne

# Przykład



## Przykład



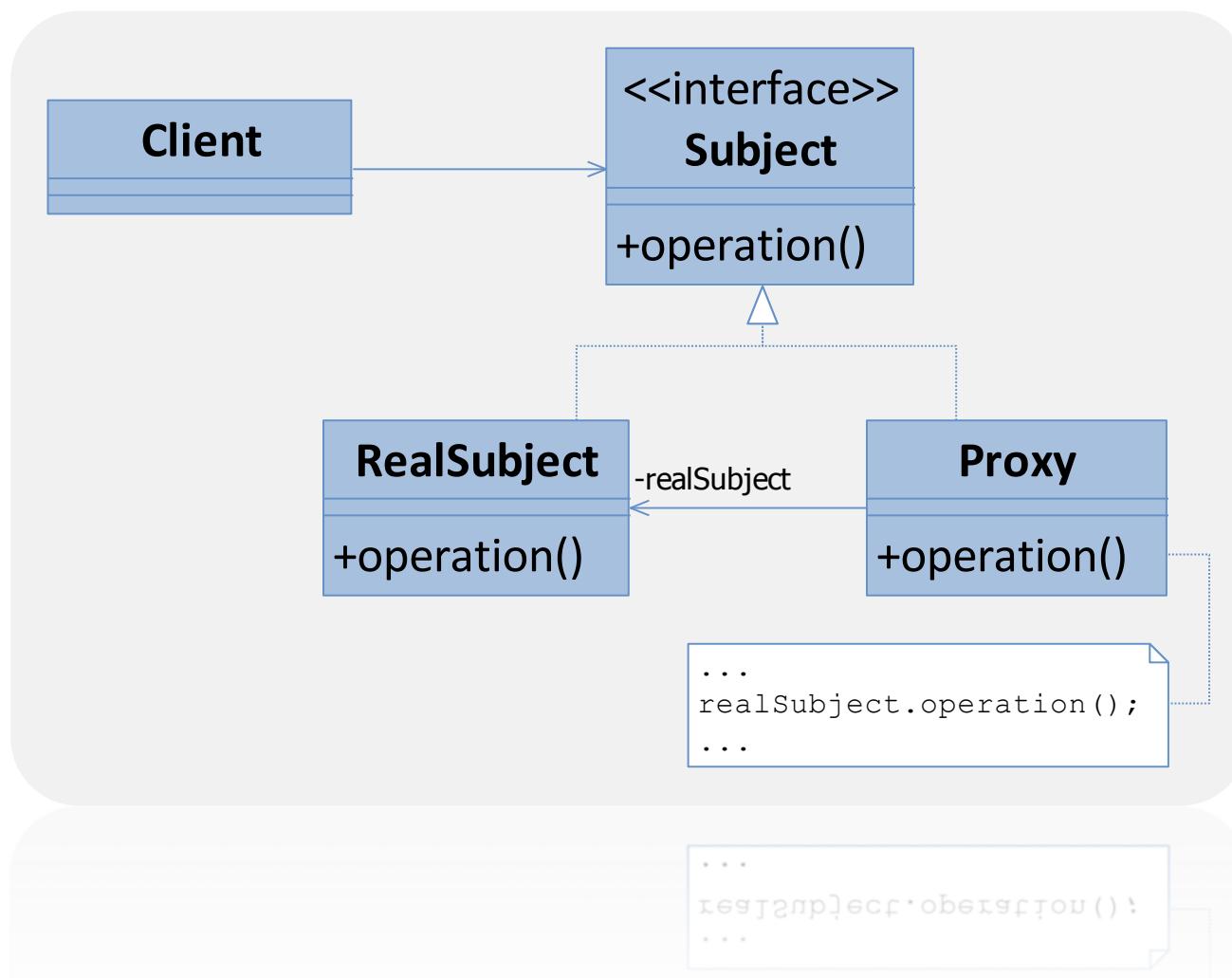
```
Picture picture1 = new PictureProxy("picture1.png");
Picture picture2 = new PictureProxy("picture2.png");
Picture picture3 = new PictureProxy("picture3.png");

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

Picture picture1  
Picture picture2

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

Wzorce projektowe i refaktoryzacja do wzorców



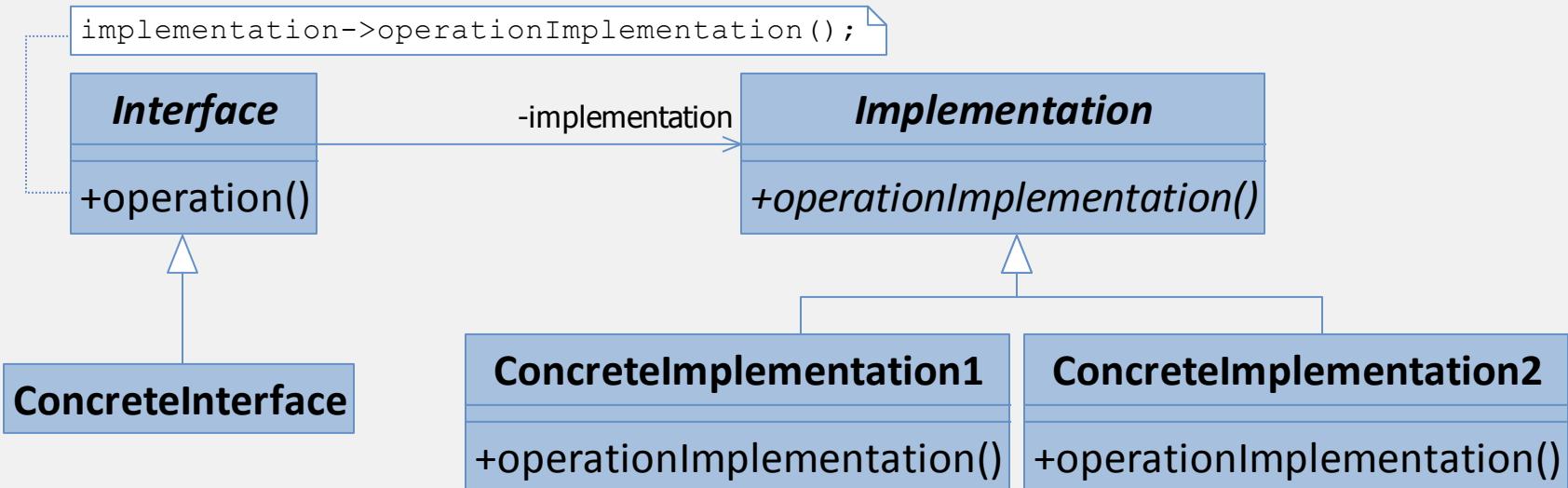
Wzorzec *Proxy* zapewnia reprezentanta obiektu, który steruje dostępem do niego.

Wprowadza dodatkową warstwę przy dostępie do obiektu.

- # *Remote Proxy* – lokalny reprezentant zdalnego obiektu
- # *Virtual Proxy* – odsuwa w czasie tworzenie 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

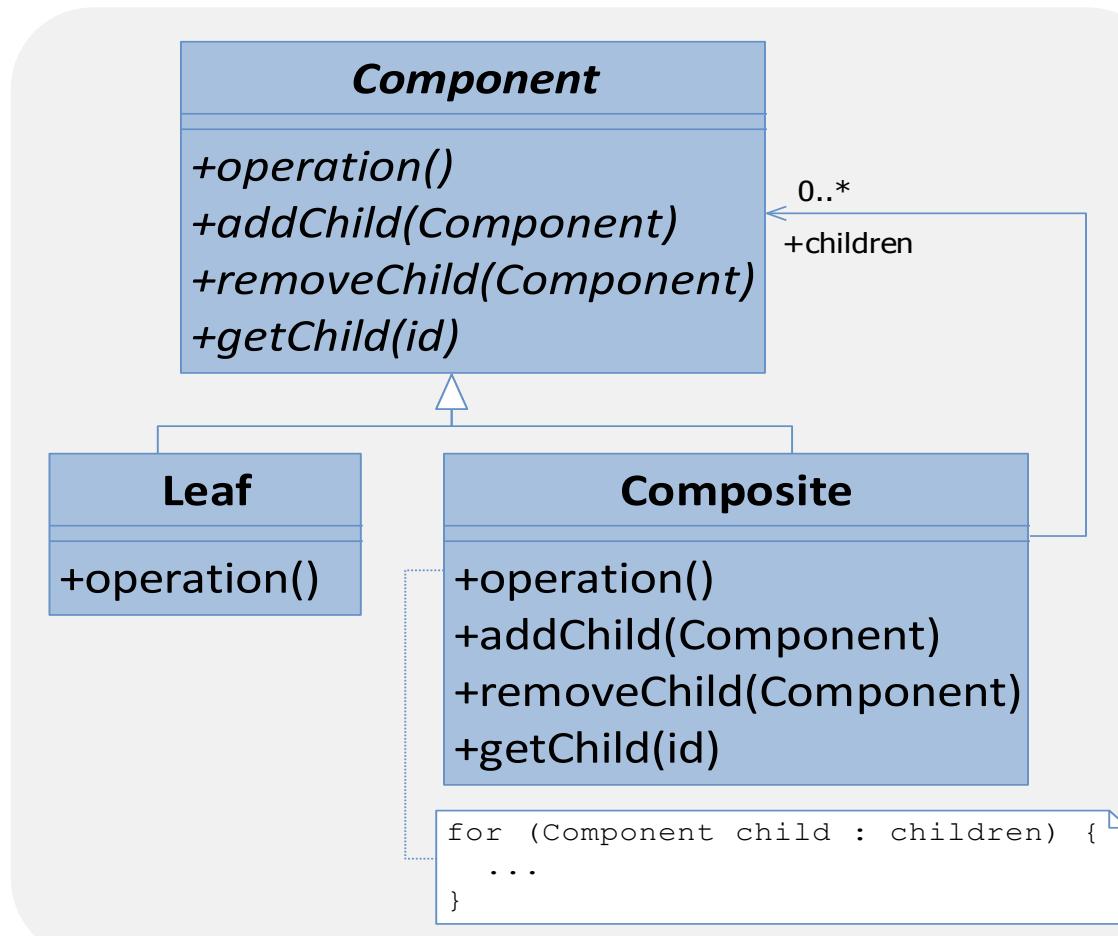
+operationImplementation()

+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.

# Wzorce GoF - podsumowanie

## Wzorce projektowe i refaktoryzacja do wzorców



## Strategie implementacji wzorców

Jest wiele sposobów implementowania każdego wzorca.

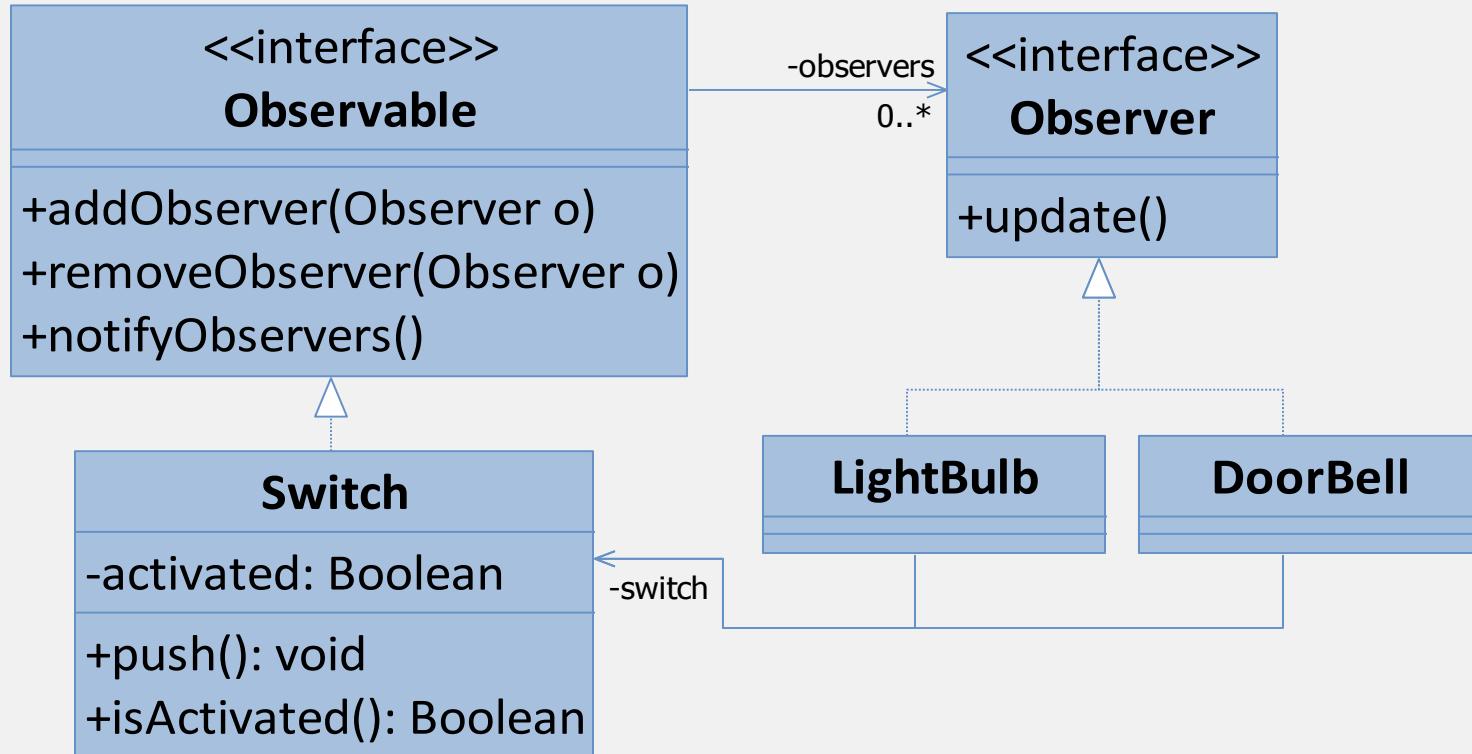
Diagram dołączany do opisu wzorca jest tylko przykładem a nie specyfikacją.

## *Observer - wersja podstawowa*

Obserwowany powiadamia obserwatora wywołując metodę *update()*.

Obserwator przechowuje odwołanie do obserwowanego i za jego pomocą pobiera informacje nt. zmian jego stanu.

# Observer - wersja podstawowa



+isActivated(): Boolean  
+push(): void

-switch

# Observer - wersja podstawowa

```
public interface Observable {  
    public void addObserver(Observer observer);  
    public void deleteObserver(Observer observer);  
    public void notifyObservers();  
}
```

```
public interface Observer {  
    public void update();  
}
```

```
public class Switch implements Observable {  
    private List<Observer> observers  
        = new ArrayList<Observer>();  
    private Boolean activated = false;  
  
    public void push() {  
        activated = true;  
        notifyObservers();  
    }  
  
    public Boolean isActivated() {  
        return activated;  
    }  
  
    public void addObserver(...) {...}  
    public void deleteObserver(...) {...}  
    public void notifyObservers() {...}  
}
```

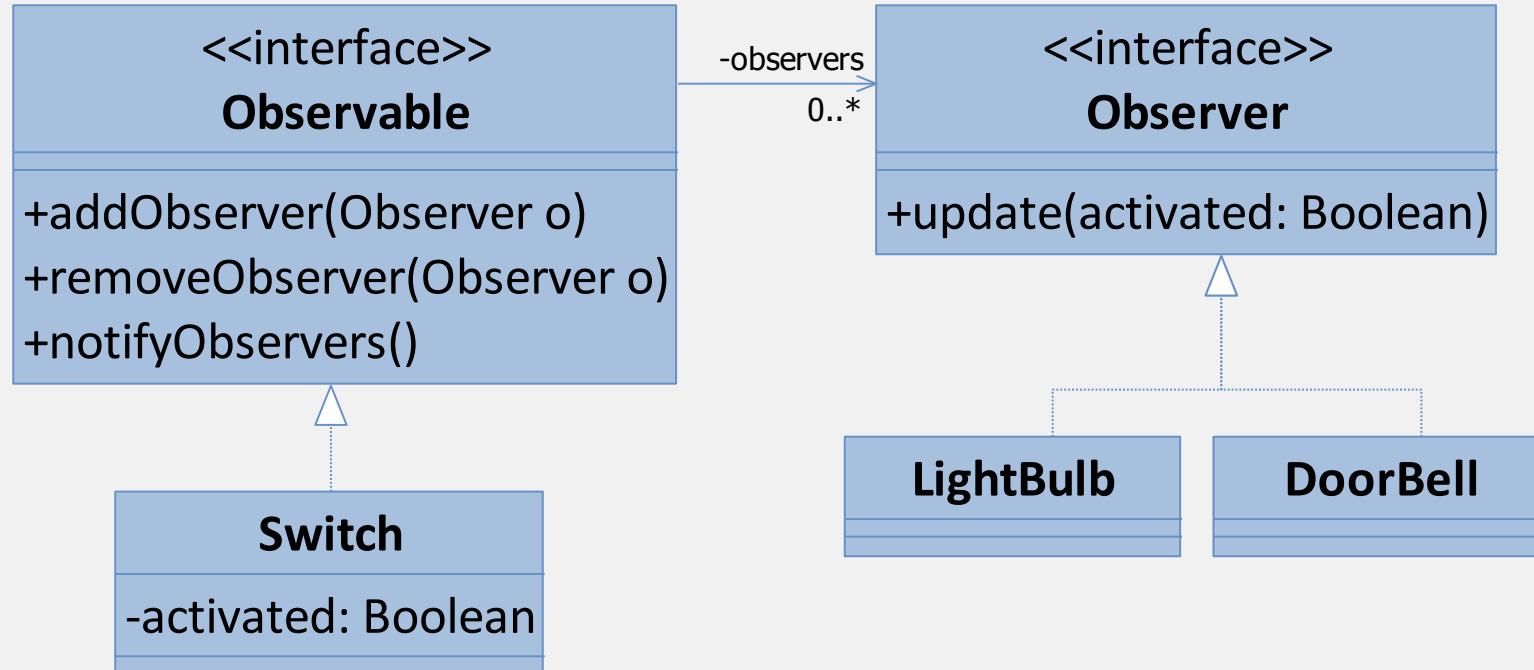
```
public class LightBulb implements Observer {  
    private Switch aSwitch = null;  
    public LightBulb(Switch aSwitch) {  
        this.aSwitch = aSwitch;  
    }  
    public void update() {  
        if (aSwitch.isActivated()) {  
            shine();  
        }  
    }  
  
    public void shine() {  
        System.out.println("Shining");  
    }  
}
```

## *Observer - wersja spараметryzowana*

Obserwowany powiadamia obserwatora wywołując metodę *update()* wraz z parametrem informującym o zmianie konkretnego stanu.

Obserwator uaktualnia swój stan na podstawie odebranego parametru.

# Observer - wersja spараметryzowana



-activated: Boolean

Switch

# Observer - wersja sparametryzowana

```
public interface Observable {  
    public void addObserver(Observer observer);  
    public void deleteObserver(Observer observer);  
    public void notifyObservers();  
}
```

```
public interface Observer {  
    public void update(boolean activated);  
}
```

```
public class Switch implements Observable {  
    private List<Observer> observers  
        = new ArrayList<Observer>();  
    private Boolean activated = false;  
  
    public void push() {  
        activated = true;  
        notifyObservers();  
    }  
  
    public void addObserver(...) {...}  
    public void deleteObserver(...) {...}  
    public void notifyObservers() {  
        for(Observer observer : observers) {  
            observer.update(activated);  
        }  
    }  
}
```

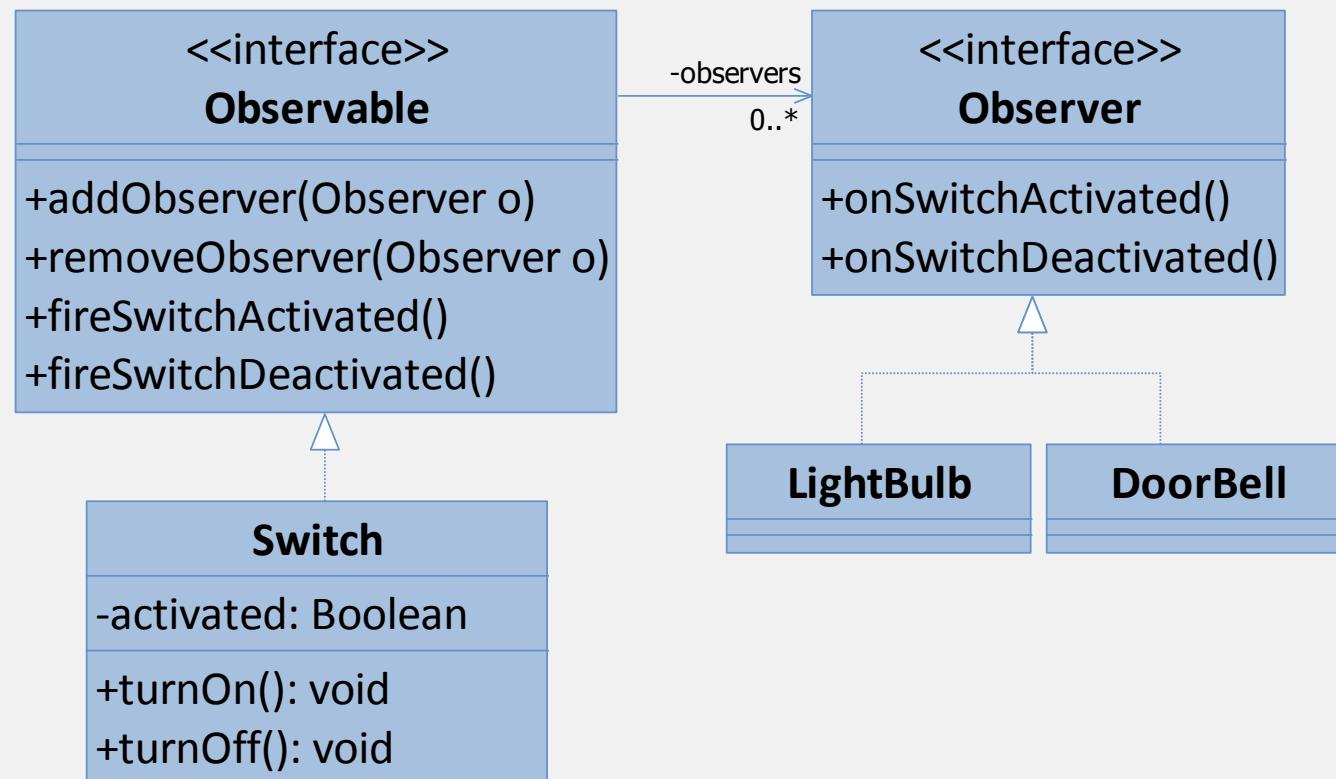
```
public class LightBulb implements Observer {  
    public void update(boolean activated) {  
        if (activated == true) {  
            System.out.println("Shining");  
        }  
    }  
}
```

## *Observer - wersja z dedykowanymi metodami*

Obserwowany powiadamia obserwatora o zdarzeniach za pomocą odpowiednich metod.

Obserwator reaguje na zdarzenia, które go interesują implementując odpowiednie metody.

# Observer - wersja z dedykowanymi metodami



biov:()#Outut+  
biov:()#Outut+

REPOSITORY

# Observer – wersja z dedykowanymi metodami

```
public interface Observable {  
    public void addObserver(Observer observer);  
    public void deleteObserver(Observer observer);  
    public void fireSwitchActivated();  
    public void fireSwitchDeactivated();  
}
```

```
public interface Observer {  
    public void onSwitchActivated();  
    public void onSwitchDeactivated();  
}
```

```
public class LightBulb implements Observer {  
    public void onSwitchActivated() {  
        System.out.println("Shining");  
    }  
    public void onSwitchDeactivated() {  
        System.out.println("Not shining");  
    }  
}
```

```
public class Switch implements Observable {  
    private List<Observer> observers  
        = new ArrayList<Observer>();  
    private Boolean activated = false;  
    public void turnOn() {  
        activated = true;  
        fireSwitchActivated();  
    }  
    public void turnOff() {  
        activated = false;  
        fireSwitchDeactivated();  
    }  
    public void fireSwitchActivated() {  
        for(Observer observer : observers) {  
            observer.onSwitchActivated();  
        }  
    }  
    public void fireSwitchDeactivated() {  
        for(Observer observer : observers) {  
            observer.onSwitchDeactivated();  
        }  
    }  
    public void addObserver(...) {...}  
    public void deleteObserver(...) {...}  
}
```

# Różnice pomiędzy podobnymi wzorcami

## *Strategy*

- # Hermetyzuje algorytm
- # Umożliwia jego dynamiczne podmienianie

## *Command*

- # Hermetyzuje żądanie
- # Umożliwia wykonanie żądania przez dowolnego klienta
- # Umożliwia kolejkowanie żądań i wykonanie ich w późniejszym czasie

# Różnice pomiędzy podobnymi wzorcami

## *Simple Factory*

- # Tworzy gotowy obiekt
- # Potrafi tworzyć różne obiekty na podstawie parametrów

## *Builder*

- # Składa gotowy obiekt z części
- # Tworzy obiekty podobne różniące się elementami składowymi

# Różnice pomiędzy podobnymi wzorcami

## Proxy

- # Zapewnia ten sam interfejs obiektu
- # Reprezentuje inny obiekt i zarządza dostępem do niego

## Decorator

- # Zapewnia rozszerzony interfejs obiektu
- # Pozwala dynamicznie dodawać do obiektu dodatkowe zobowiązania

# Różnice pomiędzy podobnymi wzorcami

## *Simple Factory*

- # Tworzy obiekt określonego typu
- # Potrafi decydować o typie tworzonego obiektu na podstawie parametrów

## *Factory Method*

- # Zapewnia interfejs do tworzenia obiektów
- # Umożliwia podklasom decydowanie o typie tworzonego obiektu

- # Metody fabrykujące są często wywoływane przez metody szablonowe
- # Klasy *Abstract Factory* są często implementowane z wykorzystaniem wzorca *Factory Method*
- # Metody fabrykujące mogą produkować obiekty *ConcreteStrategy* na podstawie przekazanego im parametru.

- # Obiekty *Strategy* są często produkowane przez obiekty *Factory*
- # Wzorzec *Strategy* może korzystać z wzorca *Template Method* w przypadku, gdy algorytm jest rozbudowany
- # Obiekt *Strategy* może korzystać ze wzorca *Observer*, aby informować inne obiekty np. o postępie w wykonaniu algorytmu

- # Wzorzec *Chain of Responsibility* może używać komend do reprezentowania żądań w postaci obiektów.
- # Obiekt *Factory* może produkować obiekty *Command*.
- # Wzorzec *Decorator* może posłużyć do dynamicznego rozszerzania odpowiedzialności obiektów *Command*.

# Antywzorce projektowania obiektowego

Antywzorzec pokazuje jak dojść od problemu do złego rozwiązania.

- pokazuje dlaczego złe rozwiązanie wydaje się korzystne
- pokazuje długofalowe skutki takiego rozwiązania
- wskazuje wzorce, które mogą doprowadzić do dobrego rozwiązania

## # *Spaghetti Code*

- Kod staje się nieczytelny na skutek używania złożonych struktur językowych.

## # *The Blob (God Class)*

- Jedna klasa implementuje zachowanie całej aplikacji, podczas gdy inne przechowują dane.

# Antywzorce projektowania obiektowego

## # *Poltergeists*

- Projekt jest „zaśmiecony” przez nadmiarowe niepotrzebnie dodane klasy - poltergeisty

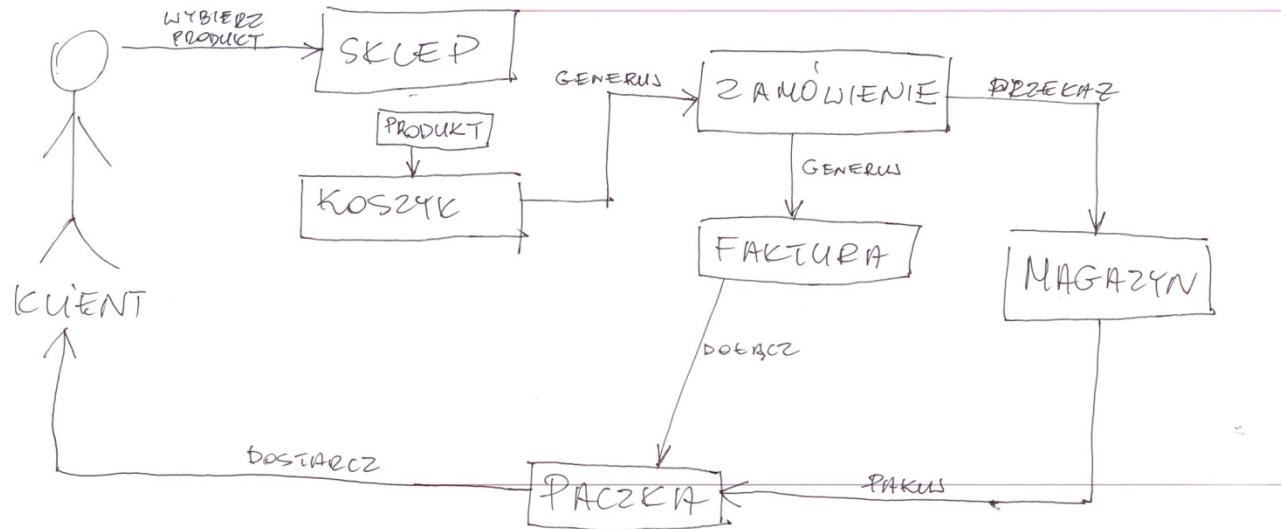
## # *Golden Hammer*

- Jedno narzędzie jest używane do rozwiązywania większości problemów.

# Wybrane wzorce architektoniczne

Wzorce projektowe i refaktoryzacja do wzorców

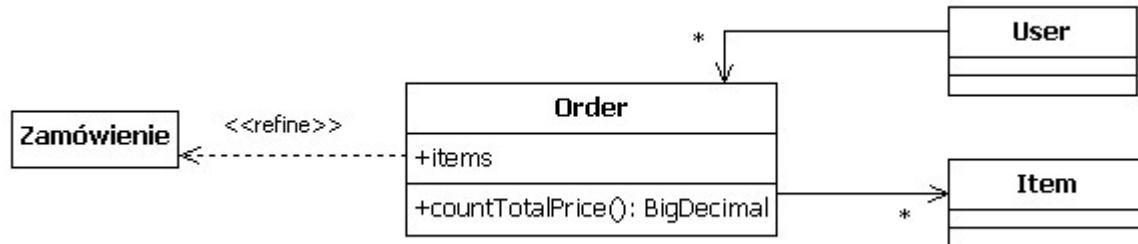




## Model dziedziny (Domain Model)

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

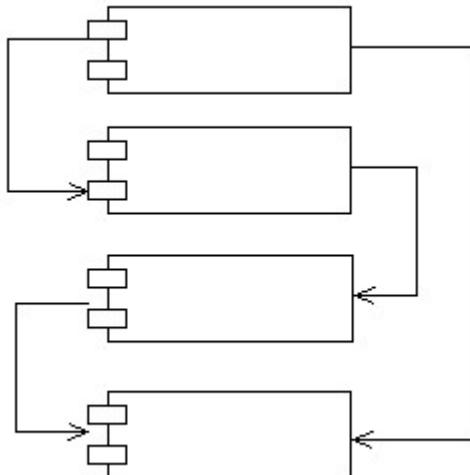
# Obiekt dziedziny



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

**Obiekt dziedziny** jest zamkniętym komponentem programistycznym, z jednoznacznie zdefiniowanym interfejsem

Obiekt dziedziny realizuje **fragment procesu** zdefiniowanego dla systemu



- # 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

- # Pojedynczą warstwę można traktować jako jedną spójną całość.
- # Można poznać sposób działania jednej warstwy bez potrzeby poznawania sposobu działania innych warstw.
- # Można wymienić dowolną warstwę na inną będącą jej alternatywną implementacją.
- # Zależności pomiędzy warstwami są minimalne.
- # Raz zaimplementowana warstwa może zostać wykorzystana do implementacji usług o szerszym zakresie działania

- # Wprowadzanie niektórych zmian może doprowadzić do kasadowego modyfikowania kodu.
- # Zbyt duża ilość warstw może skutkować obniżeniem szybkości działania aplikacji.

# Model warstwowy Fowlera

## Presentation Layer (Warstwa prezentacji)

Prezentowanie informacji użytkownikowi i obsługa żądań pochodzących od użytkownika.



## Domain Layer (Warstwa dziedziny)

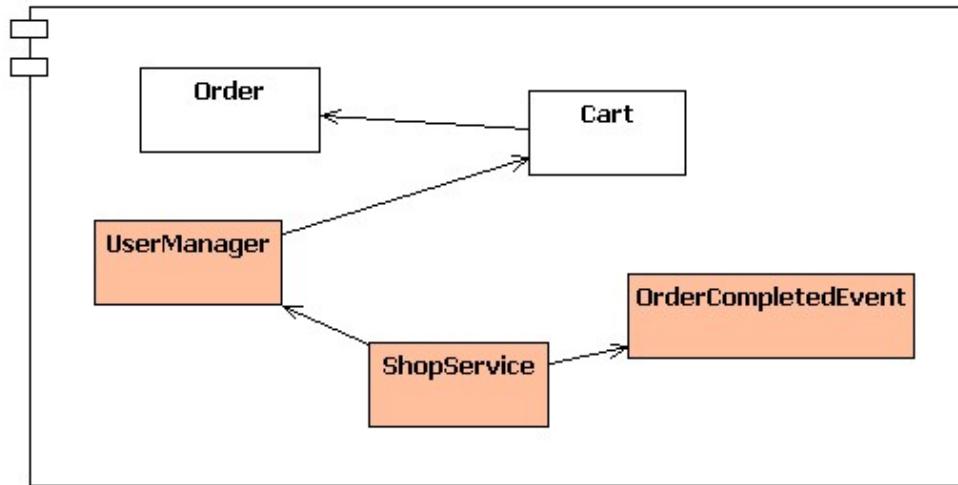
Implementacja zadań biznesowych systemu.



## Data Source Layer (Warstwa dostępu do danych)

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

# Dekomponowanie warstwy



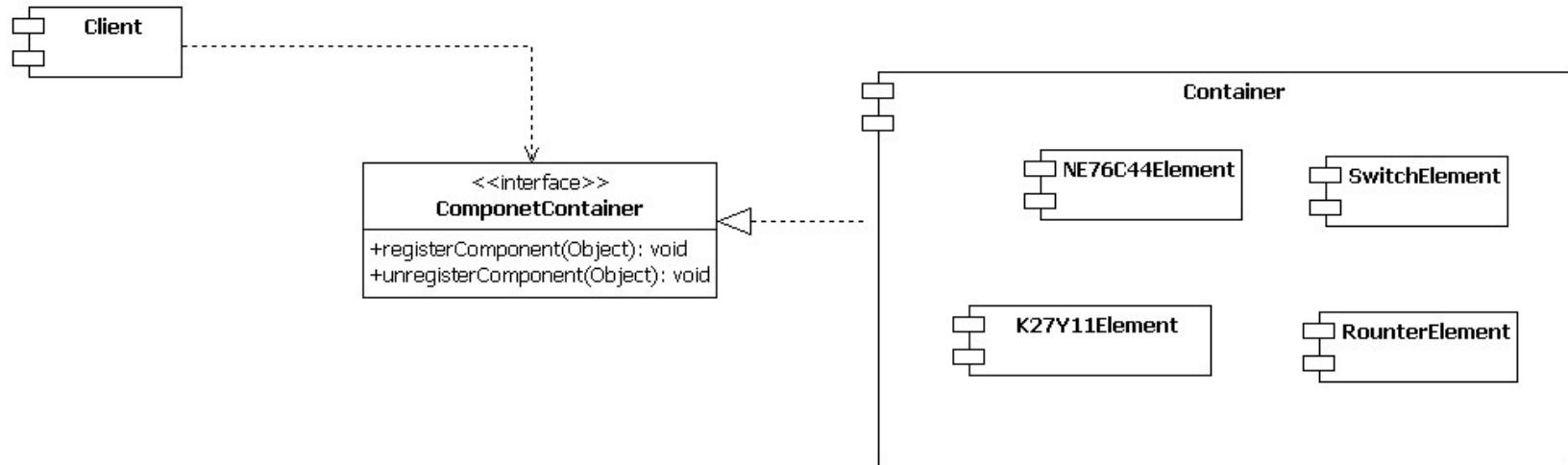
- # Warstwa jest dekomponowana na części przez **obiekty dziedziny**
- # W każdym systemie oprócz elementów modelu znajdują się również **obiekty abstrakcyjne**, które łączą system w całość

## Obiekty modelu dziedziny

- Modelują fragmenty rzeczywistości

## Obiekty abstrakcyjne

- Wyjątki, zdarzenia, usługi, klasy narzędziowe
- 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 może definiować pulę obiektów
- # Daje możliwość deklaratywnego zarządzania zależnościami oraz transakcjami w systemie
- # Może stanowić ośrodek propagacji zdarzeń w systemie
- # Daje podstawę do integracji z innymi systemami
- # Implementacje: *EJB Container, Spring Application Context, Google Guice, Nano Container, Pico Container*



# Dostęp do danych

Wybrane wzorce architektoniczne

## 1. JDBC + *ResultSet*

## 2. *iBatis*

Przeniesienie zapytań SQL poza kod źródłowy.

## 3. *JDO, Hibernate, TopLink*

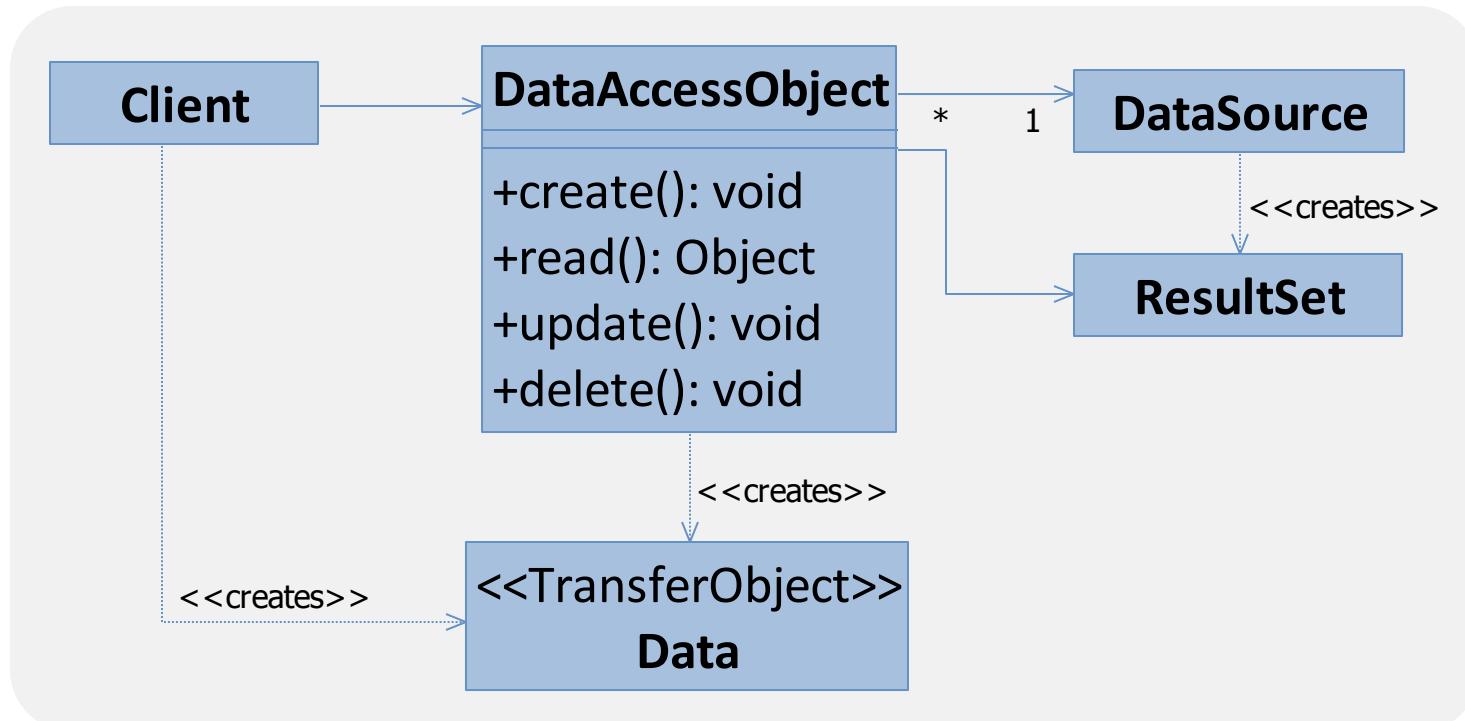
Narzędzia O-RM pozwalające pracować na obiektach, zamiast na tabelach

## 4. Standard JPA

Wzorcową implementacją jest TopLink-Essentials.

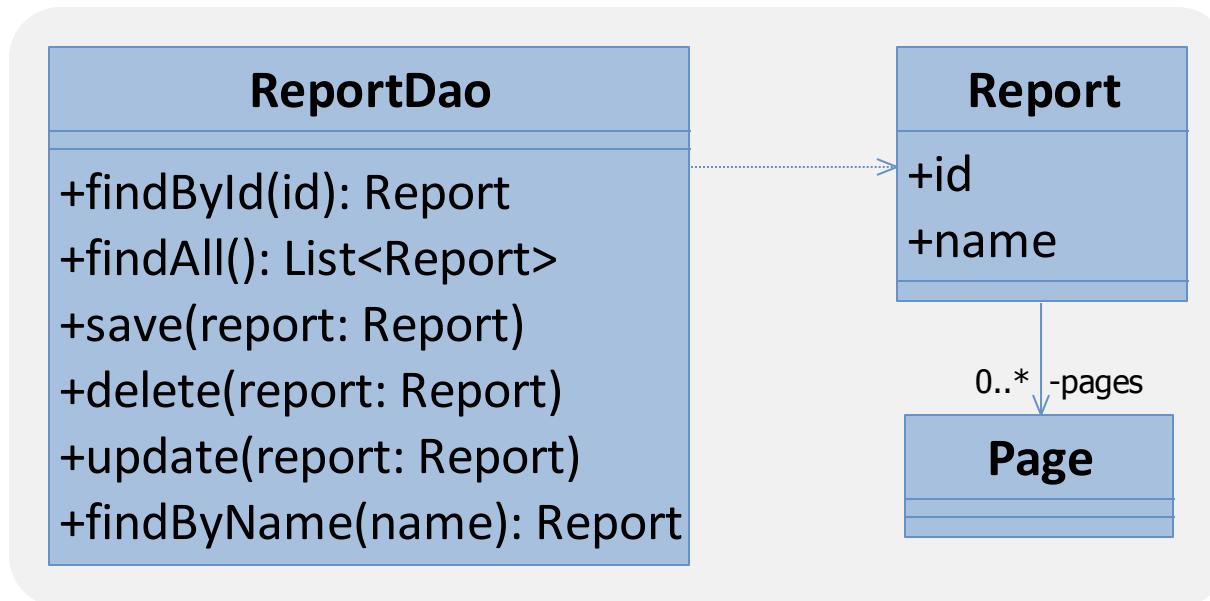
Inne implementacje: *Hibernate, JDO, OpenJPA*

# Table Data Gateway



Dat  
<<TransferObject>>

# Przykład

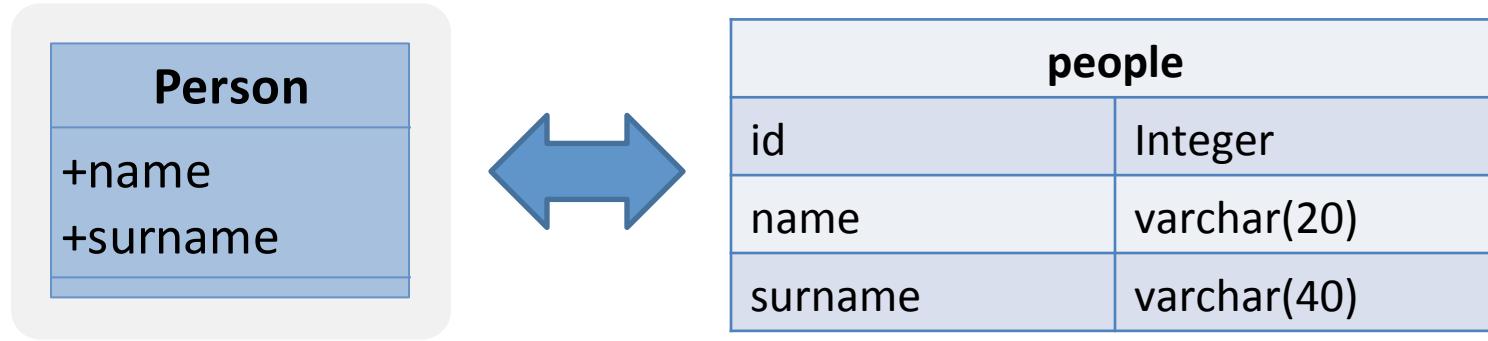


+findByName(name): Report  
+save(report: Report)

base

- # Klienci za pośrednictwem obiektu *Data Access Object* mogą korzystać z różnych źródeł danych bez znajomości ich konkretnej implementacji i lokalizacji.
- # Klient korzysta z obiektowych struktur danych. Nie musi znać schematu bazy danych.
- # Obiekt *Data Access Object* opakowuje implementację kodu dostępu do danych dzięki czemu kod klienta może być prostry.
- # Warstwa obiektów *DAO* oddziela aplikację od implementacji trwałego magazynu danych.

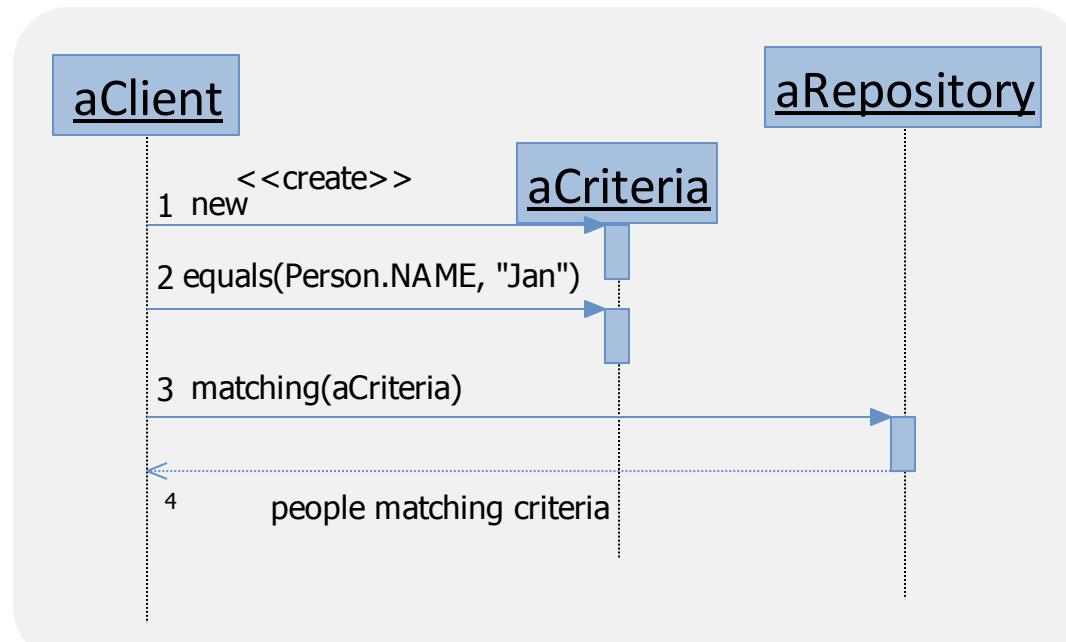
# Object-Relational Mapping



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

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

# bns it} Przykład



beobieś mądrzinię dżurę

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);
```

- # Umożliwia tworzenie zapytań w sposób deklaratywny i obiektowy poprzez odpowiednie skonfigurowanie obiektu *Criteria*.
- # Klient nie martwi się o to, gdzie przechowywane są obiekty dziedziny problemu. Ma do nich dostęp poprzez obiekt *Repository*.
- # Minimalizuje duplikacje zapytań do bazy danych.

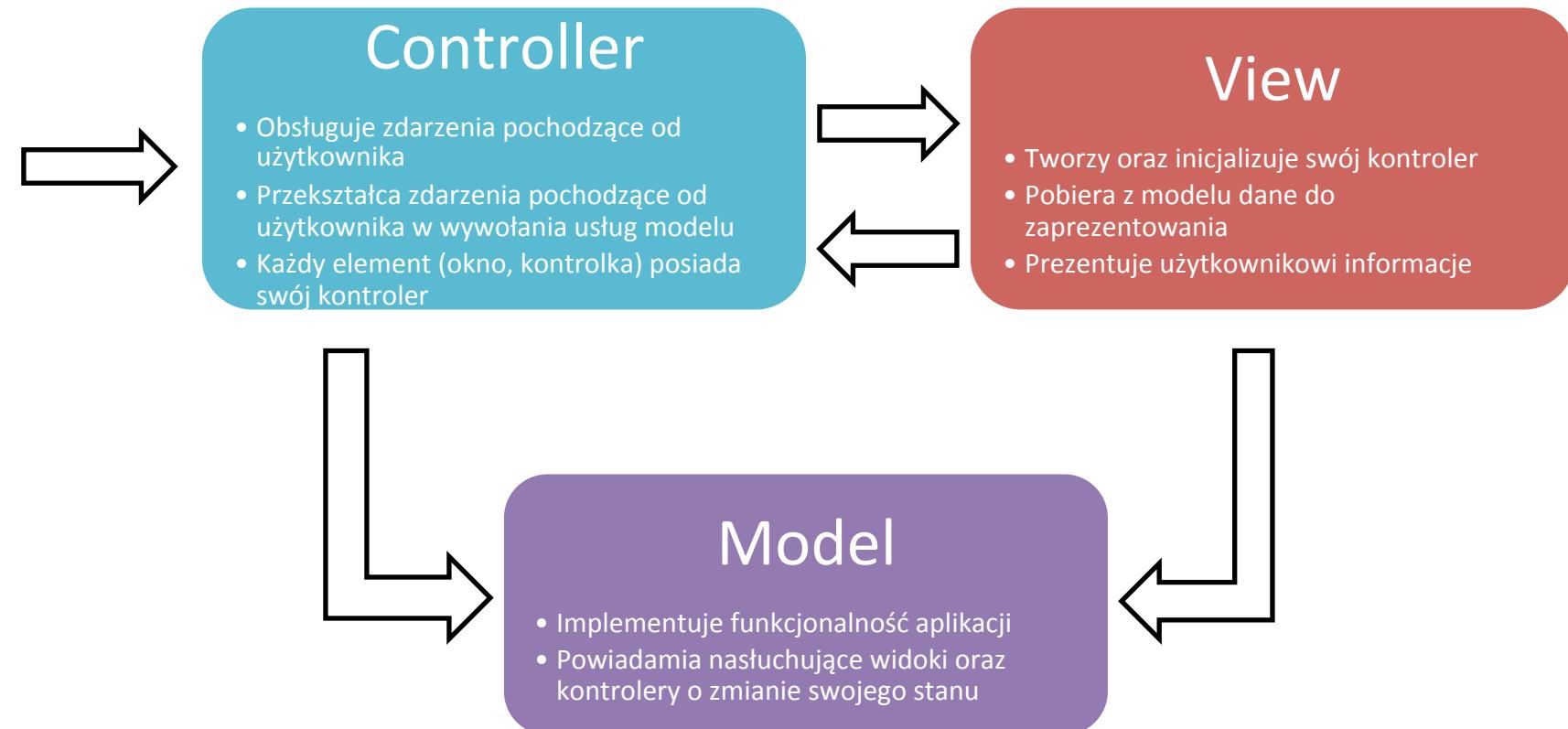
- # Niweluje potrzebę tworzenia dedykowanych metod *find\**()
- # Źródłem obiektów dla *Repository* nie musi koniecznie być relacyjna baza danych.
- # Zastosowanie wzorca *Repository* zwiększa czytelność kodu, który polega w dużej mierze na różnorodnych zapytaniach.



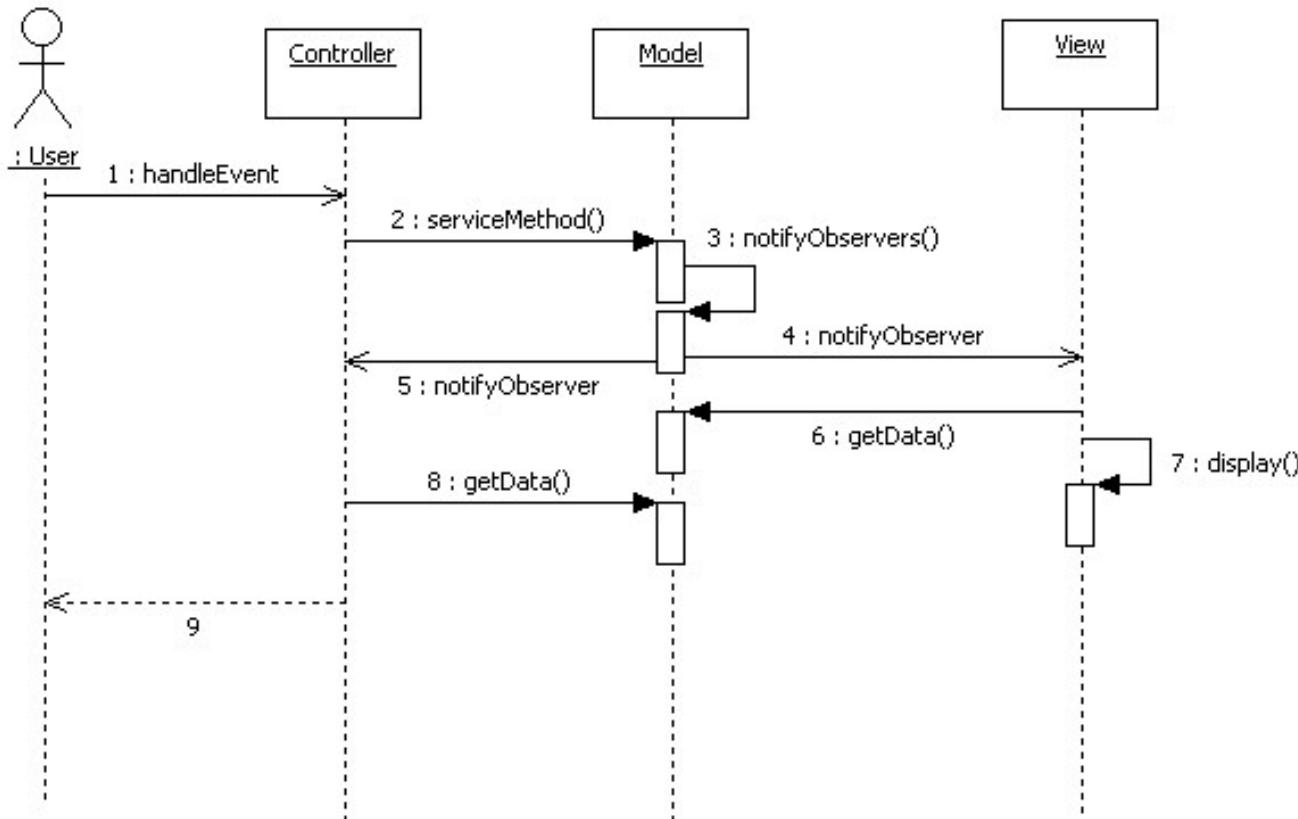
# Prezentacja i komunikacja z użytkownikiem

Wybrane wzorce architektoniczne

# Wzorzec Model View Controller

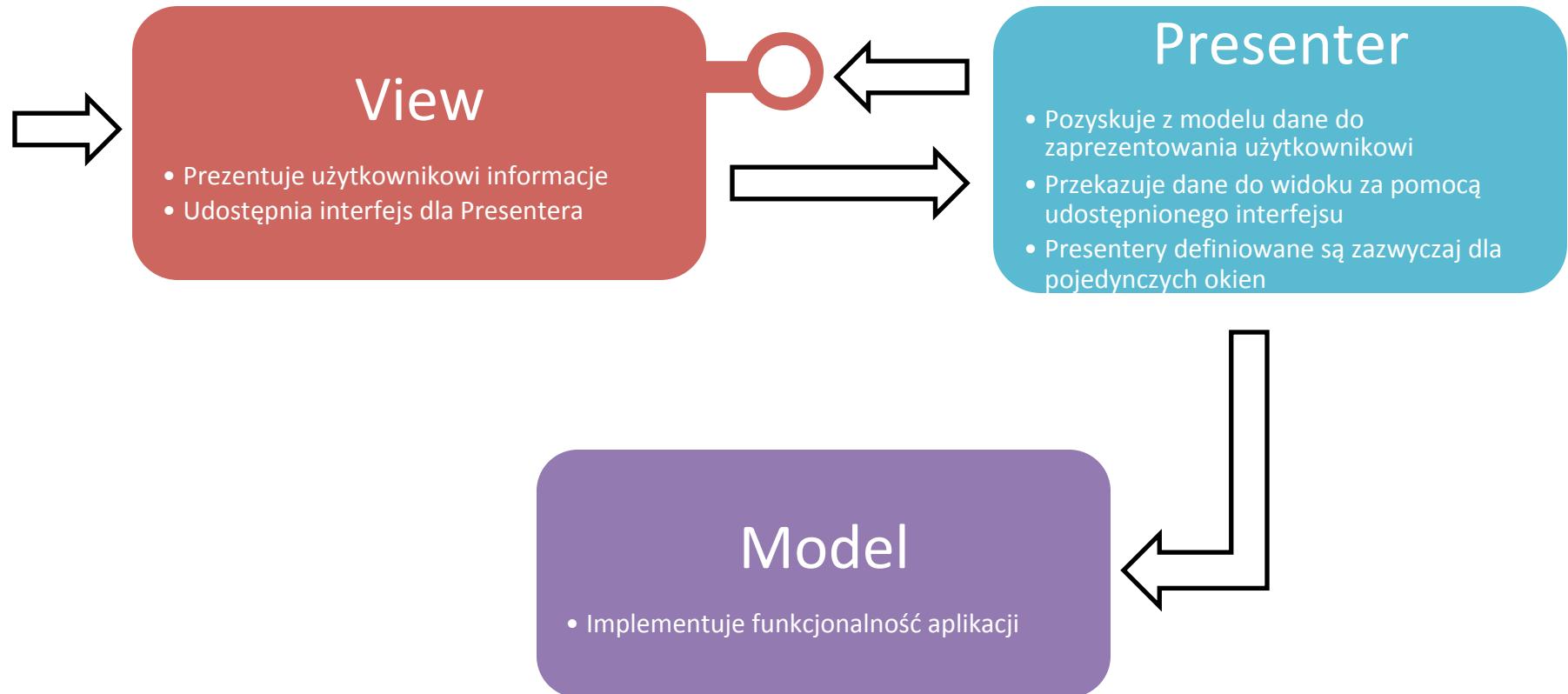


# MVC - obsługa żądania



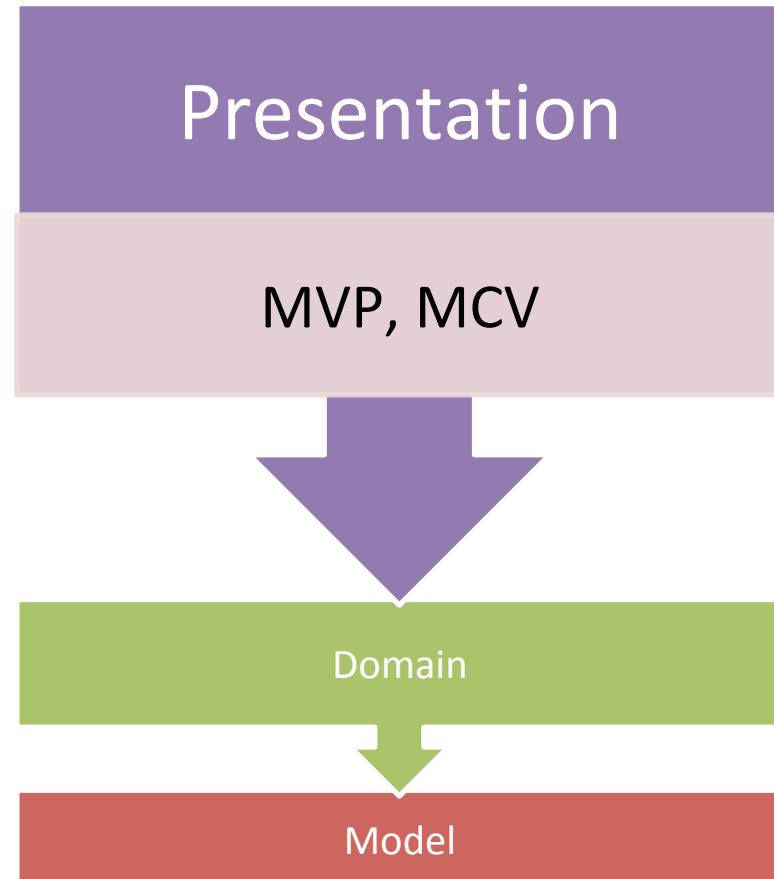
- # *View* jest obiektem konfigurowalnym przy pomocy wzorca *Strategy*. Odpowiednią strategię zapewnia *Controller*.
- # *Model* zawiera informacje, które należy zaprezentować użytkownikowi oraz usługi, które potrafią dostarczyć te informacje
- # *View* może wykorzystywać wzorzec *Composite* do składania graficznych elementów interfejsu użytkownika.

# 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.



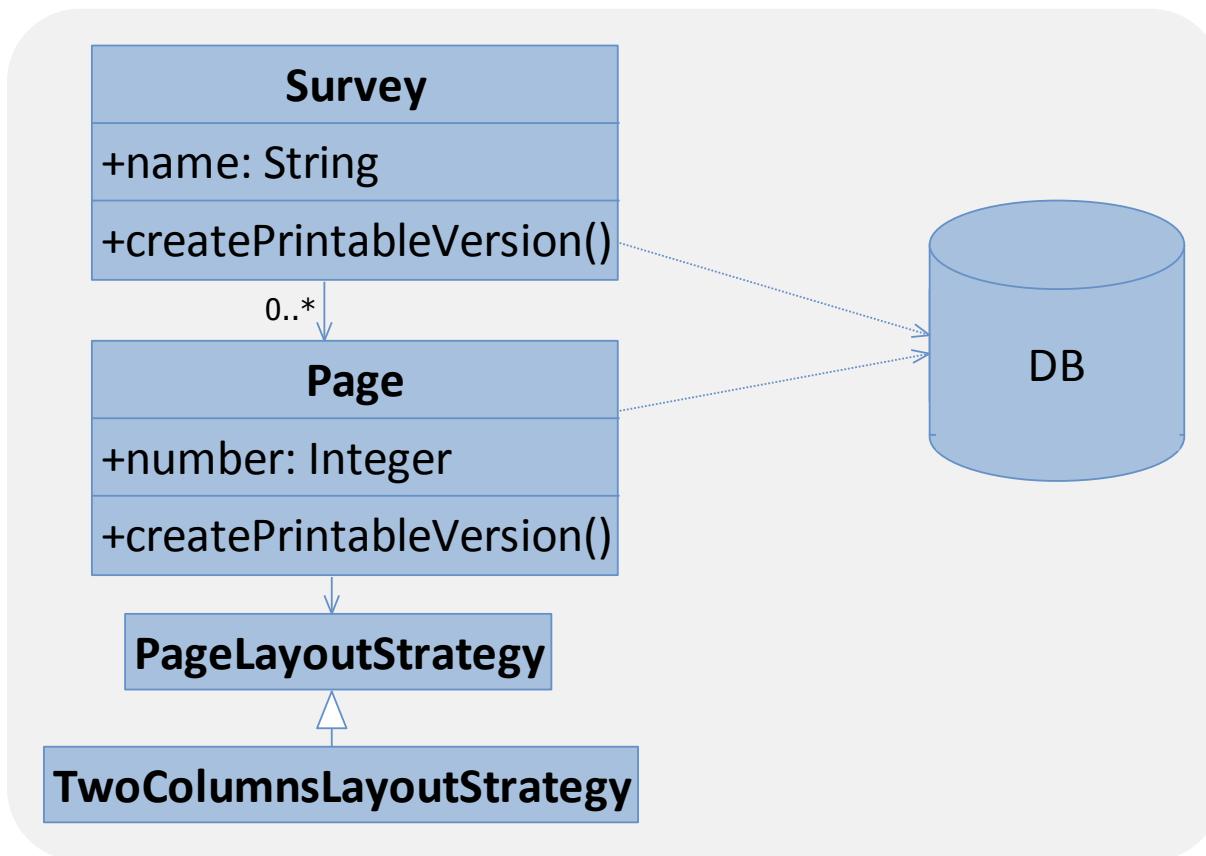


# Dziedzina

Wybrane wzorce architektoniczne

Wzorzec *Domain Model* tworzy strukturę powiązanych i współpracujących ze sobą obiektów, spośród których każdy obiekt reprezentuje niezależny element dziedziny problemu.

## Przykład



Wzorce projektowe i refaktoryzacja do wzorców

TwoolumnLayoutStrategy

- # Wykorzystanie wzorca *Domain Model* w aplikacji wiąże się z umieszczeniem w niej warstwy obiektów, które modelują dziedzinę biznesową aplikacji naśladując dane i reguły występujące w tej dziedzinie.
- # Wzorzec *Domain Model* łączy modelowane dane z modelowanym procesem.

- # Istnieje ryzyko, że obiekty dziedziny problemu będą miały nadmierną odpowiedzialność wykonując pewne czynności tylko w pojedynczych przypadkach.
- # Należy rygorystycznie pilnować odpowiedzialności obiektów. Takie podejście prowadzi do stworzenia użytecznego modelu dziedziny.

## *Simple Domain Model*

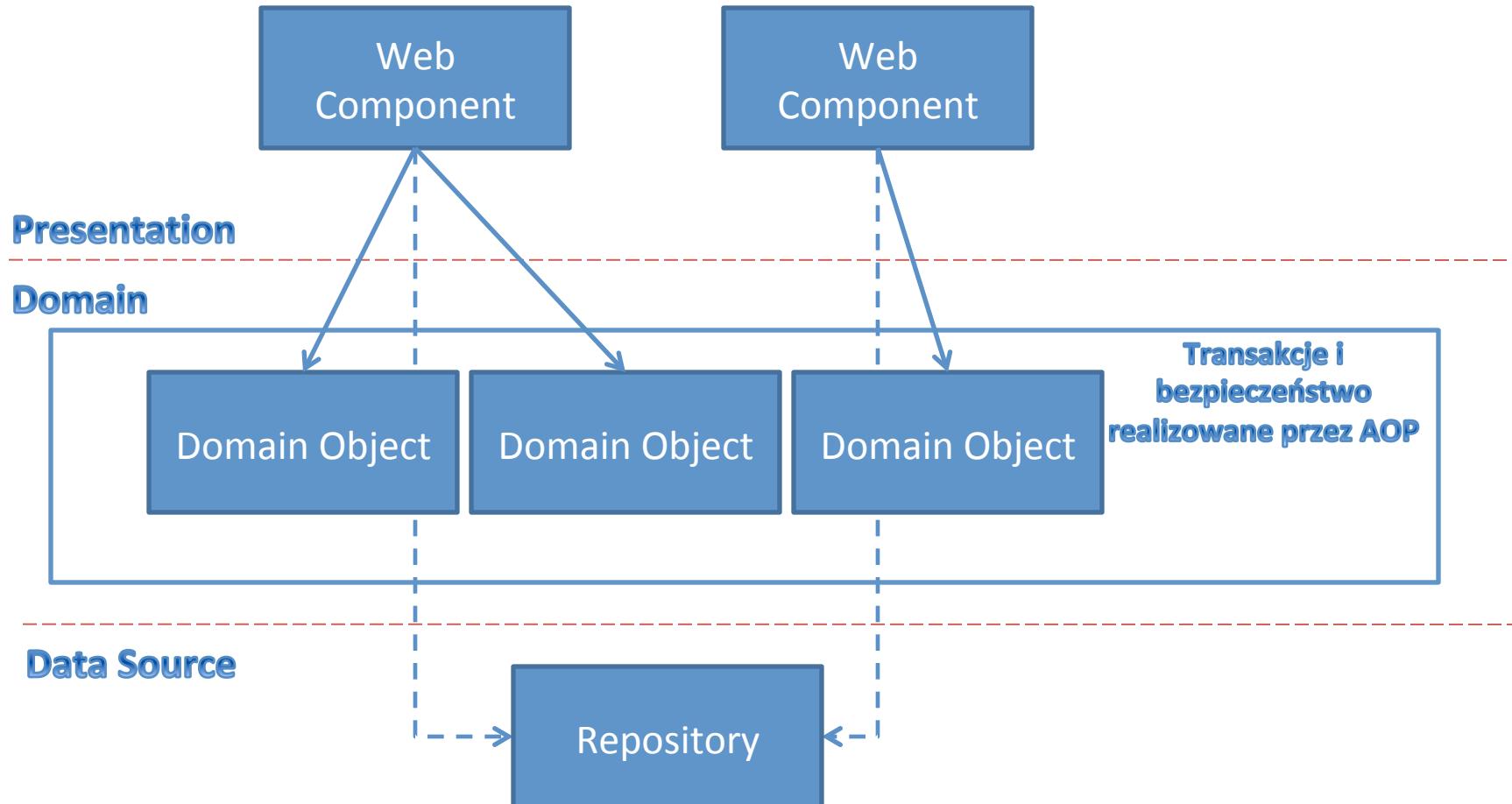
- # Jeden obiekt dziedziny problemu jest mapowany na pojedynczą tabelę w bazie danych.
- # Model obiektowy jest bardzo podobny do modelu bazodanowego danych.
- # Dobrze sprawdza się przy prostej logice biznesowej

## *Rich Domain Model*

- # Jest bardziej złożony – struktura obiektowa zawiera dziedziczenia, strategie i inne wzorce projektowe GoF.
- # Model obiektowy może znaczco różnić się od modelu bazowanego danych.
- # Dobrze sprawdza się przy złożonej logice biznesowej, natomiast trudniej zmapować go na bazę danych.

Wzorzec *Exposed Domain Model* opisuje jeden ze sposobów hermetyzacji logiki biznesowej w aplikacjach klasy Enterprise.

# Exposed Domain Model



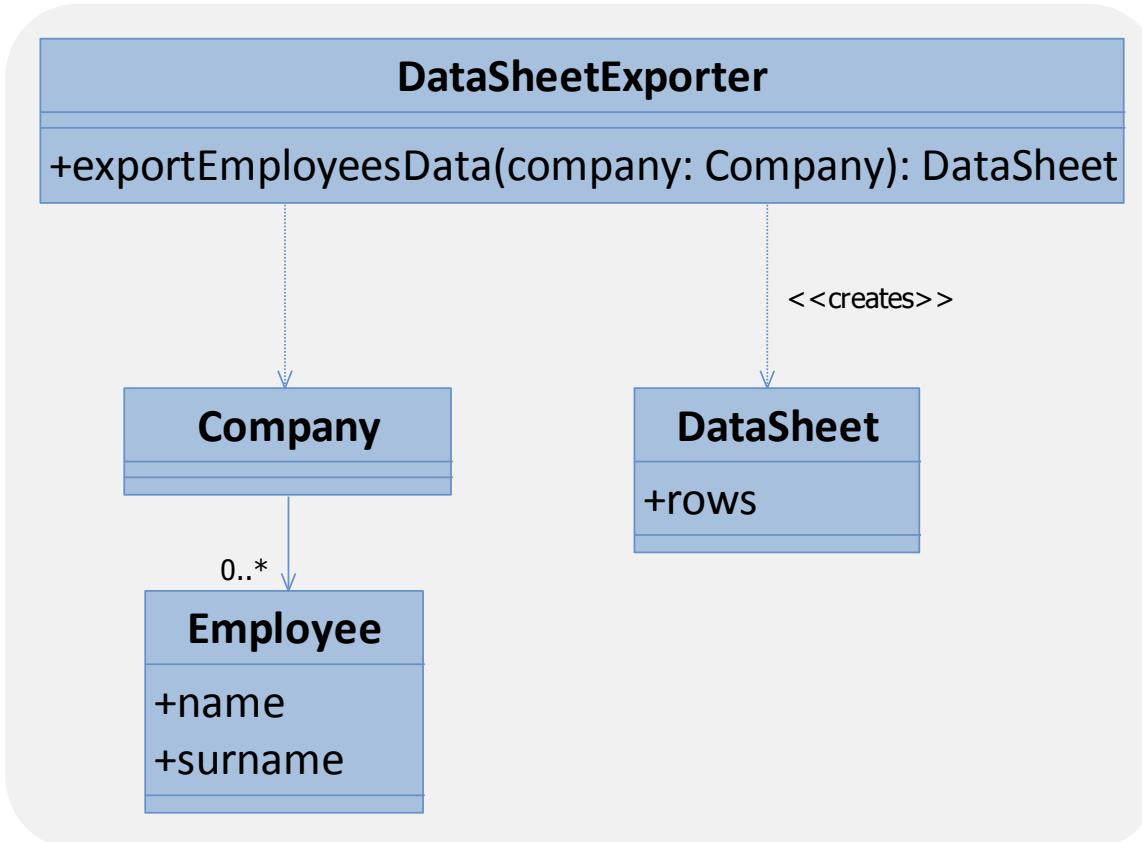
- # Obiekty warstwy *Presentation* mają bezpośredni dostęp do obiektów warstwy *Domain*.
- # Warstwa *Domain* nie musi wiedzieć jakie obiekty ma dostarczyć warstwie *Presenation*.
- # Warstwa *Presentation* staje się bardziej złożona na skutek interakcji z wieloma różnymi obiektami.

- # Wzorzec *Exposed Domain Model* wykorzystuje transparentną persystencję – obiekty *POJO* mogą być persystowane, nie ma potrzeby tworzenia warstwy *DAO*
- # Wzorzec *Exposed Domain Model* korzysta ze wzorca *Repository* w celu zapewnienia persystencji obiektów.

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

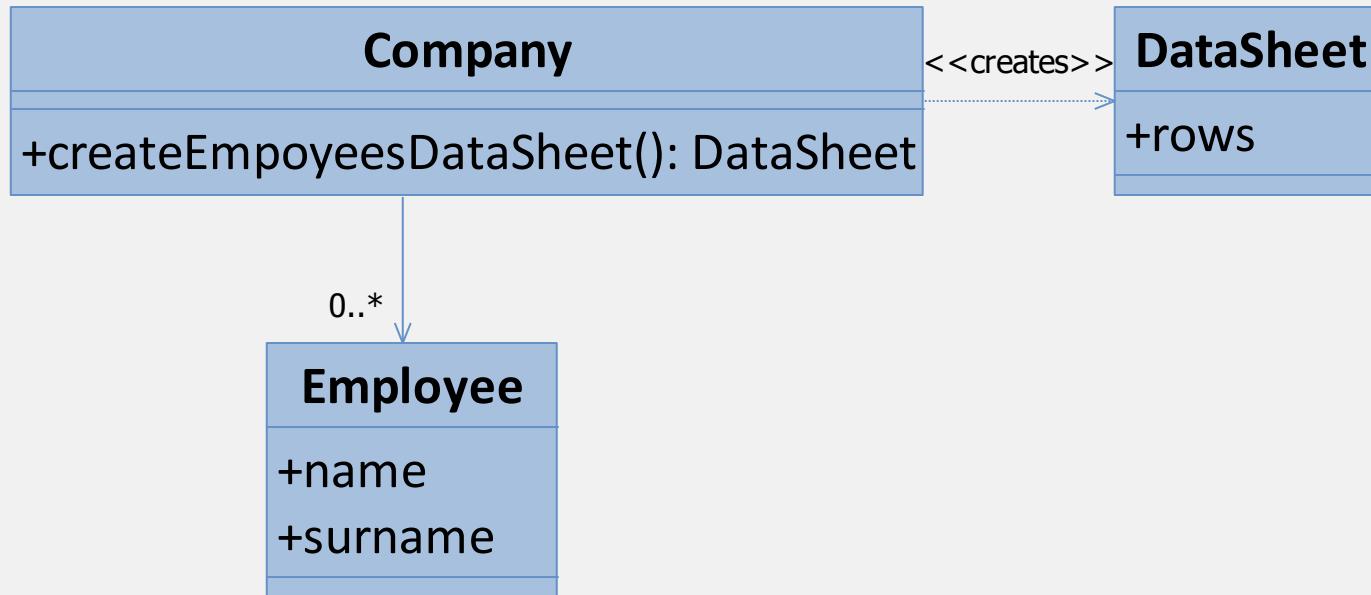
- # Obiekty dziedziny problemu są pozbawione zachowania
- # W zamian istnieje wiele obiektów usługowych, które implementują logikę zachowania natomiast nie przechowują stanu.

# Przykład: *Anemic Domain Model*



+imiona  
+nazwisko

## Przykład: *Domain Model*

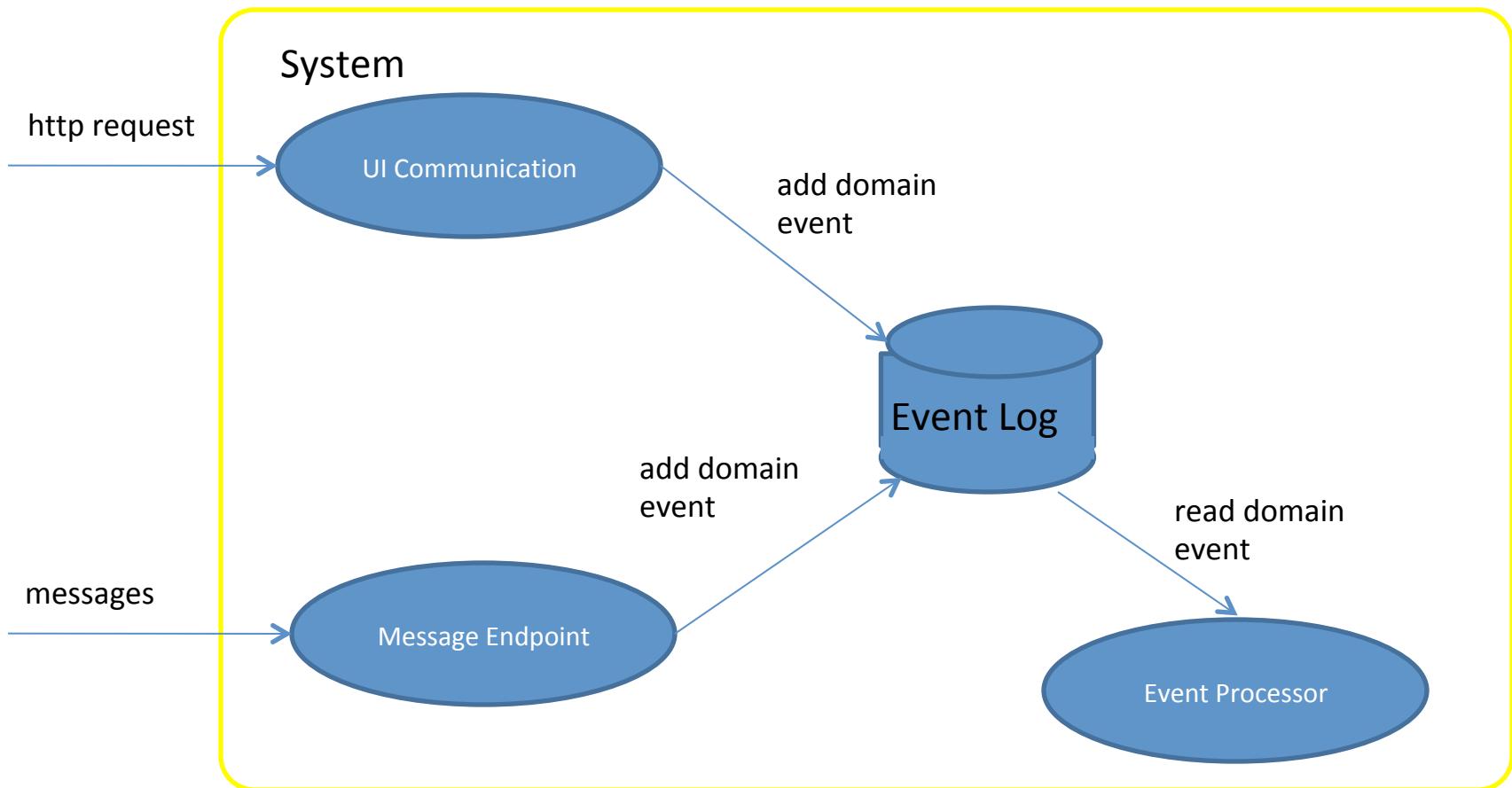


- # Zaprzeczenie paradygmatowi programowania zorientowanego obiektowo mówiącym o łączeniu danych i logiki.
- # Podążanie w kierunku programowania proceduralnego.
- # Przeniesienie odpowiedzialności implementacji logiki biznesowej z warstwy dziedziny problemu do warstwy usług.
- # Utrata korzyści wynikających z zastosowania wzorca *Domain Model*.

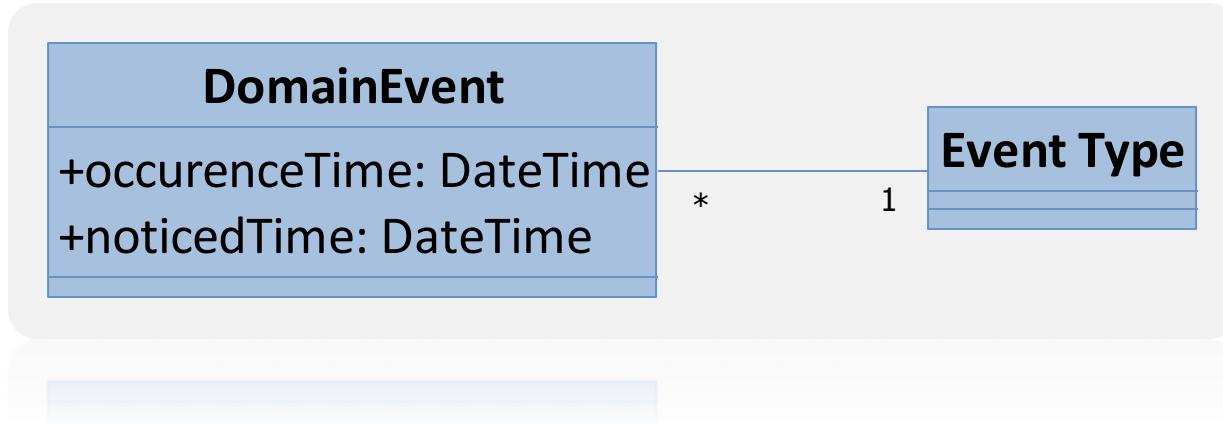
bns it }    # }    Wzorce modelu zdarzeniowego  
Wybrane wzorce architektoniczne

bns it } # Wzorzec Domain Event  
Wybrane wzorce architektoniczne

# Domain Event



## Przykładowe zdarzenie *DomainEvent*



Wzorzec *Domain Event* pozwala uchwycić zdarzenia, które mogą spowodować zmianę stanu aplikacji.

- # Zewnętrzna warstwa wejściowa systemu jest odpowiedzialna za przyjęcie żądania, zamienienie go do postaci zdarzenia *DomainEvent* i zapisania do dziennika zdarzeń *EventLog*.
- # Wewnętrzna warstwa systemu jest odpowiedzialna za przetwarzanie zdarzeń *DomainEvent*. Nie musi nic wiedzieć o źródle pochodzenia zdarzenia.

- # W praktyce wydziela się kilka dzienników zdarzeń (*EventLog*), jeżeli istnieją różne wymagania odnośnie szybkości obsługi zdarzenia.
- # Dane zapisane w dzienniku zdarzeń (*EventLog*) nigdy nie powinny się zmieniać.

- # Istnieje możliwość wystąpienia zdarzenia, które jest niepoprawne. System wysyła wtedy zdarzenie *RetroactiveEvent*, które kompensuje efekty przebiegu niepoprawnego zdarzenia.
- # Procesor zdarzeń (*EventProcessor*) może łączyć zdarzenia w celu skrócenia czasu ich przetwarzania.

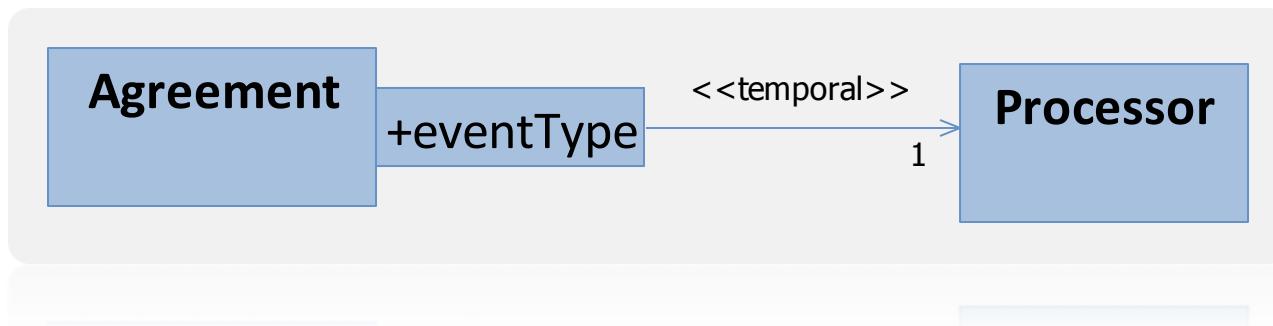
Wzorzec *Agreement Dispatcher* opisuje strukturę procesora zdarzeń *Domain Event* uwzględniającą zmieniające się reguły biznesowe.

Bank pobiera opłatę w wysokości 5 zł od każdego przelewu.

1.09 zapada decyzja aby kwotę tą zwiększyć do 6 zł.

Polecenie przelewu zostaje przyjęte 31.08 lecz zostaje wykonane 2.09.

Istotne jest aby to polecenie zostało wykonane zgodnie z regułami obowiązującymi do 31.08



- # Wzorzec *Agreement Dispatcher* umożliwia dostosowanie reguł przetwarzania zdarzeń do zmieniających się reguł biznesowych.
- # Zdarzenia, które zostały zarejestrowane przed zmianą reguł są przetwarzane zgodnie z poprzednimi regułami biznesowymi.

bns it } # Wzorce stanu tymczasowego  
Wybrane wzorce architektoniczne

Intencją wzorca *Audit Log* jest zapisywanie informacji o ważnych zdarzeniach zachodzących w systemie.

## Przykład

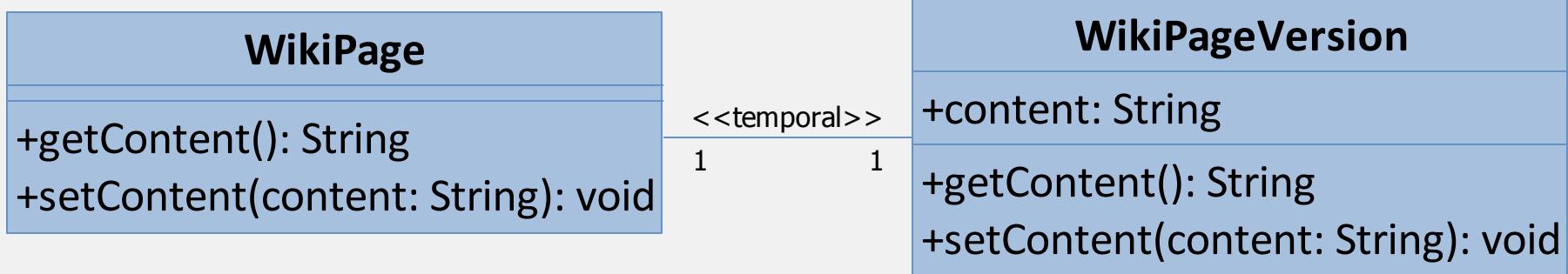
```
public class Person {  
    private String address;  
    private static Logger logger = Logger.getLogger(Person.class);  
  
    public void setAddress(String address) {  
        logger.info("Setting new address. Old address: " + this.address  
            + ". New address: " + address);  
  
        this.address = address;  
    }  
}
```



```
...  
INFO 2008-11-08 12:40:10 Setting new address.  
Old address: Piotrkowska 100. New address: Gdańska 80.  
...
```

- # Korzystanie z *Audit Log* jest bardzo proste.
- # Informacje o zmianach mogą być zapisywane do różnych formatów plików: ASCII, XML, DB.
- # Przeglądanie dziennika zdarzeń może być skomplikowane ze względu na ilość zapisywanych informacji.

# Temporal Object



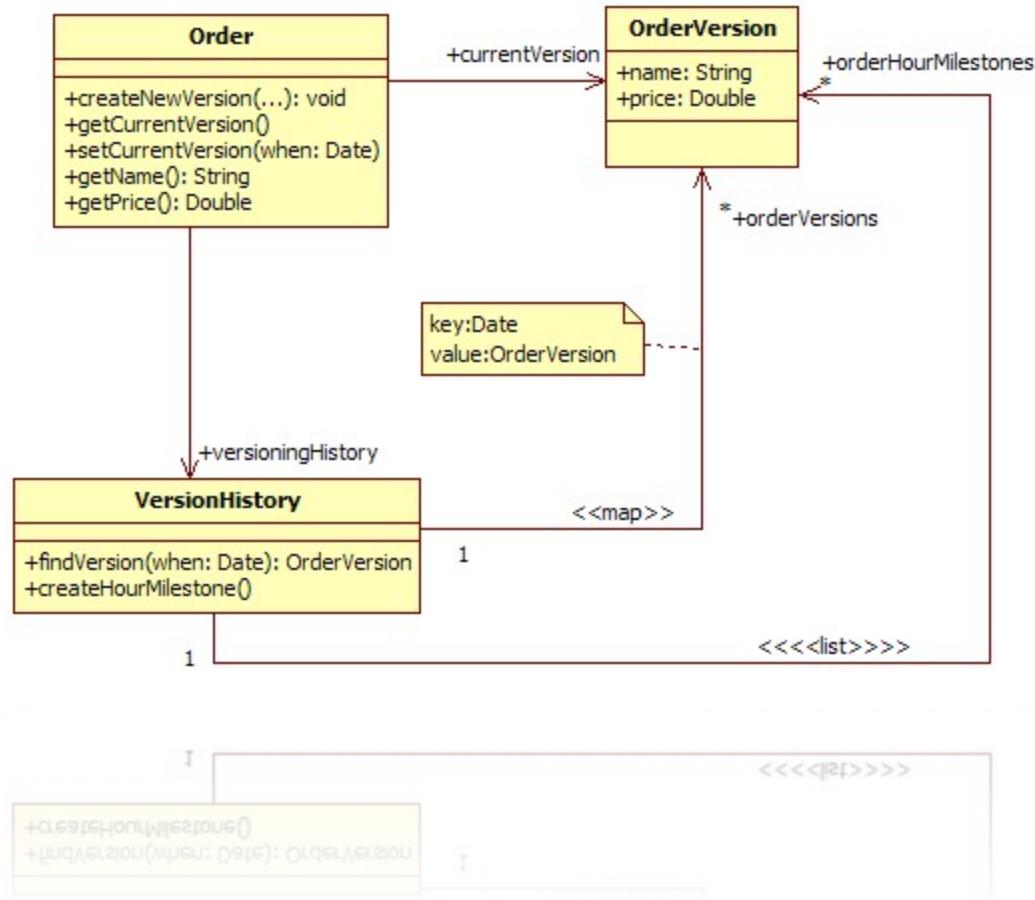
*Temporal Object* reprezentuje obiekt, który zmienia się w czasie.

# bns it} Kod przykładu

```
public class WikiPageVersion {  
    private String content;  
    public String getContent() {  
        return content;  
    }  
    public void setContent(String content) {  
        this.content = content;  
    }  
}
```

```
public class WikiPage {  
    private TemporalCollection history  
        = new SingleTemporalCollection();  
    public String getContent() {  
        return current().getContent();  
    }  
    public void setContent(String content) {  
        WikiPageVersion workingCopy = getWorkingCopy();  
        workingCopy.setContent(content);  
        history.put(workingCopy);  
    }  
    private WikiPageVersion current() {  
        return (WikiPageVersion)history.get();  
    }  
    private WikiPageVersion getWorkingCopy() {  
        return current().copy();  
    }  
}
```

## Inny przykład



# Kod przykładu

```
public class Order implements Serializable {
    private Long id;
    private VersionHistory versioningHistory = new VersionHistory();
    private OrderVersion currentVersion;
    public void createNewVersion( String productId,
        String productName, Double price ) {
        OrderVersion orderVersion = new OrderVersion();
        orderVersion.setCustomId( productId );
        orderVersion.setName( productName );
        orderVersion.setPrice( price );
        versioningHistory.addOrderVersion(
            orderVersion.getDate(), orderVersion );
        currentVersion = orderVersion;
    }
    public void setCurrentVersionTo( Date when ) {
        currentVersion
            = versioningHistory.findVersion( when );
    }
    protected OrderVersion getCurrentVersion() {
        return currentVersion;
    }
    public String getCustomId() {
        return getCurrentVersion().getCustomId();
    }
    public void setCustomId( String customId ) {
        getCurrentVersion().setCustomId( customId );
    }
    //delegacje reszty getterów i setterów
}
```

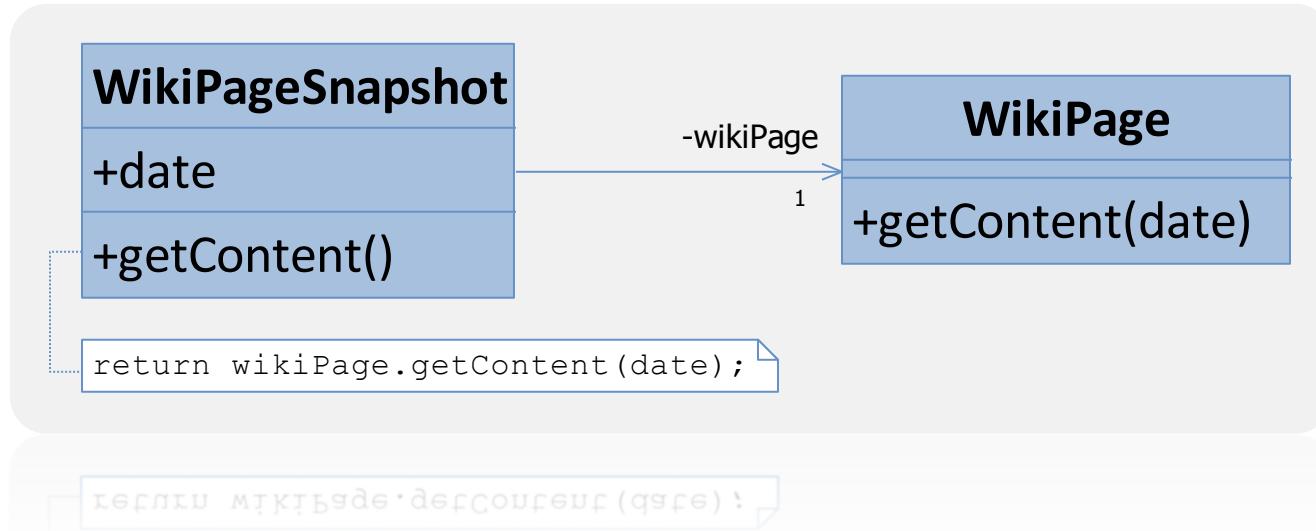
```
public class OrderVersion
    implements Serializable {
    private Long id;
    private String customId;
    private String name;
    private Double price;
    private Date date;
    public OrderVersion() {
        this.date = Utils.getNow();
    }
}
//gettery i settery
```

# bns it} Kod przykładu

```
public class VersionHistory
    implements Serializable {
    private Map orderVersions
        = new HashMap();
    private List orderHourMilestones
        = new ArrayList();
    public OrderVersion findVersion( Date date ) {
        return orderVersions.get( date );
    }
    public void addOrderVersion( Date date,
        OrderVersion version ) {
        orderVersions.put( date , version );
    }
    public void createHourMilestone() {
        //...
    }
}
```

- # Każda wersja obiektu *Temporal Object* reprezentuje jego stan w określonym przedziale czasu.
- # Klient współpracuje z aktualną wersją obiektu i nie musi nic wiedzieć o wersjach wcześniejszych.
- # Historia wersji może być przechowywana w różny sposób

# Przykład



Wzorzec *Snapshot* dostarcza obrazu obiektu w danym punkcie czasu.

- # Stworzenie obiektu *Snapshot* wymaga podania odpowiedniej daty.
- # Obiekty *Snapshot* ułatwiają dostęp do obiektów czasowych.
- # Obiekty *Snapshot* powinny być w większości przypadków immutable.
- # Modyfikacja/reorganizacja struktury obiektowej staje się trudna ze względu na zwiększenie ilości powiązań między obiektami.

# Architektury warstwy prezentacji

Projektowanie architektury aplikacji biznesowych



## # Architektury prezentacji

- Autonomous View – wszystko w jednej klasie
- Separated Presentation
- Synchronizacja stanu
- Forms and Control
- Model-View-Controller
- Presentation Model
- Model-View-Presenter

## # Poziomy danych związane z prezentacją

- Record State
  - rzeczywiste dane dziedzinowe (np. w bazie danych, w innym systemie, w pliku)
- Session State
  - kopia danych pobrana na potrzeby widoku (np. obiekt w pamięci reprezentujący osobę)
- Screen State
  - dane znajdujące się w komponentach GUI

## # Jedno z głównych wyzwań architektur GUI to synchronizacja Session State i Screen State

## # Mechanizmy synchronizacji danych w GUI

- Flow Synchronization
- Observer Synchronization
- Data binding

## # Flow synchronization

- centralny element zarządza synchronizacją
  - na przykład formularz
- jest odpowiedzialny za rozpropagowanie zmian wpływających na wiele elementów
- skomplikowane jeśli widoki są złożone i jedne elementy wpływają na inne

## # Observer synchronization

- Zmiany są propagowane z użyciem wzorca obserwatora
- Elementy zainteresowane zmianami w innych częściach GUI nasłuchują zmian (np. modelu w MVC)
- Ze względu na nie bezpośrednią komunikację trudniej wyśledzić konsekwencje zmian

## # Data Binding

- mechanizm automatycznego synchronizowania Screen State i Session State (a czasem nawet Record State)
- najczęściej realizowane z użyciem zdarzeń typu PropertyChange oraz refleksji

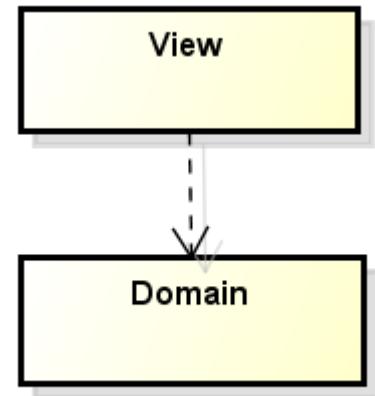
bns it } **# }** Separated Presentation  
Architektury warstwy prezentacji

## # Intencja

Oddzielić zachowania GUI  
od zachowań systemu

## # Znane użycia

- MVC, MVP, PM



powered by astah\*

*\*Bardziej podejście niż wzorzec czy architektura*

```
class AssessmentWindow...
private void updateVarianceField() {
    varianceField.setValue(currentReading.getVariance());
    varianceField.setForeground(varianceColor());
}

private Color varianceColor() {
    switch (currentReading.getVarianceCategory()) {
        case LOW:
            return Color.RED;
        case HIGH:
            return Color.GREEN;
        case NULL:
            return Color.BLACK;
        case NORMAL:
            return Color.BLACK;
        default:
            throw new IllegalArgumentException("Unknown variance category");
    }
}
```

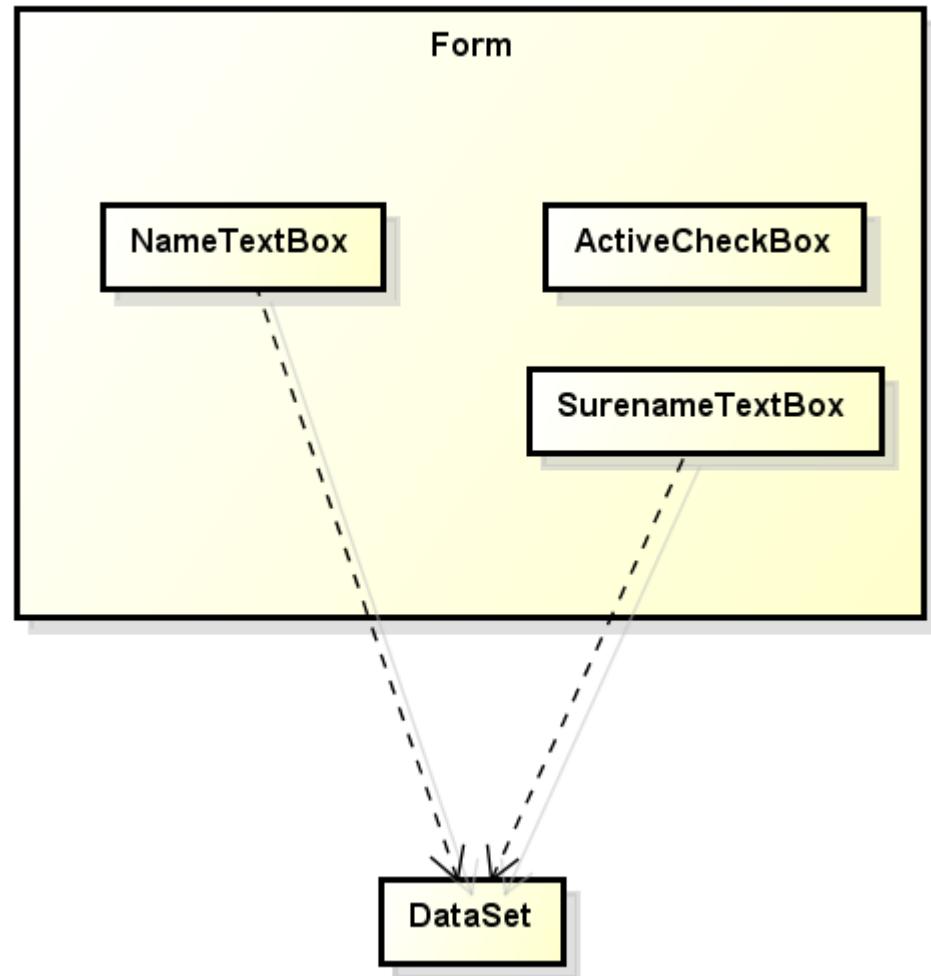
class Reading...

```
public enum VarianceCategory {  
    LOW, NORMAL, HIGH, NULL}  
  
public VarianceCategory getVarianceCategory() {  
    if (null == getVariance()) return VarianceCategory.NULL;  
    if (getVarianceRatio() < -10) return VarianceCategory.LOW;  
    else if (getVarianceRatio() > 5) return VarianceCategory.HIGH;  
    else return VarianceCategory.NORMAL;  
}
```

bns it} #} Forms and Control  
Architektury warstwy prezentacji

## # Intencja

Dostarczyć prostych  
środków do budowania  
GUI z kontrolek



powered by astah\*

## # Struktura

- Kontrolki są rozłożone na formularzu
- **Między kontrolkami a źródłami danych zazwyczaj jest Data Binding – główna siła napędowa tego podejścia**
- Formularz reaguje na zdarzenia
- Źródła danych to raczej obiekty typu DataSet niż Domain Object

## # Właściwości

- Prosty wzorzec do szybkiego tworzenia GUI
- Brak wydzielonego kontrolera – widok i zachowania są połączone
- Brak skonkretyzowanej formy organizowania złożonych interfejsów

## # Właściwości

- Zachowania realizowane poprzez zdarzenia
- Proste zmiany danych realizowane przez Data Binding
- Złożone zmiany obsługiwane przez obsługę zdarzeń

## # Przykłady

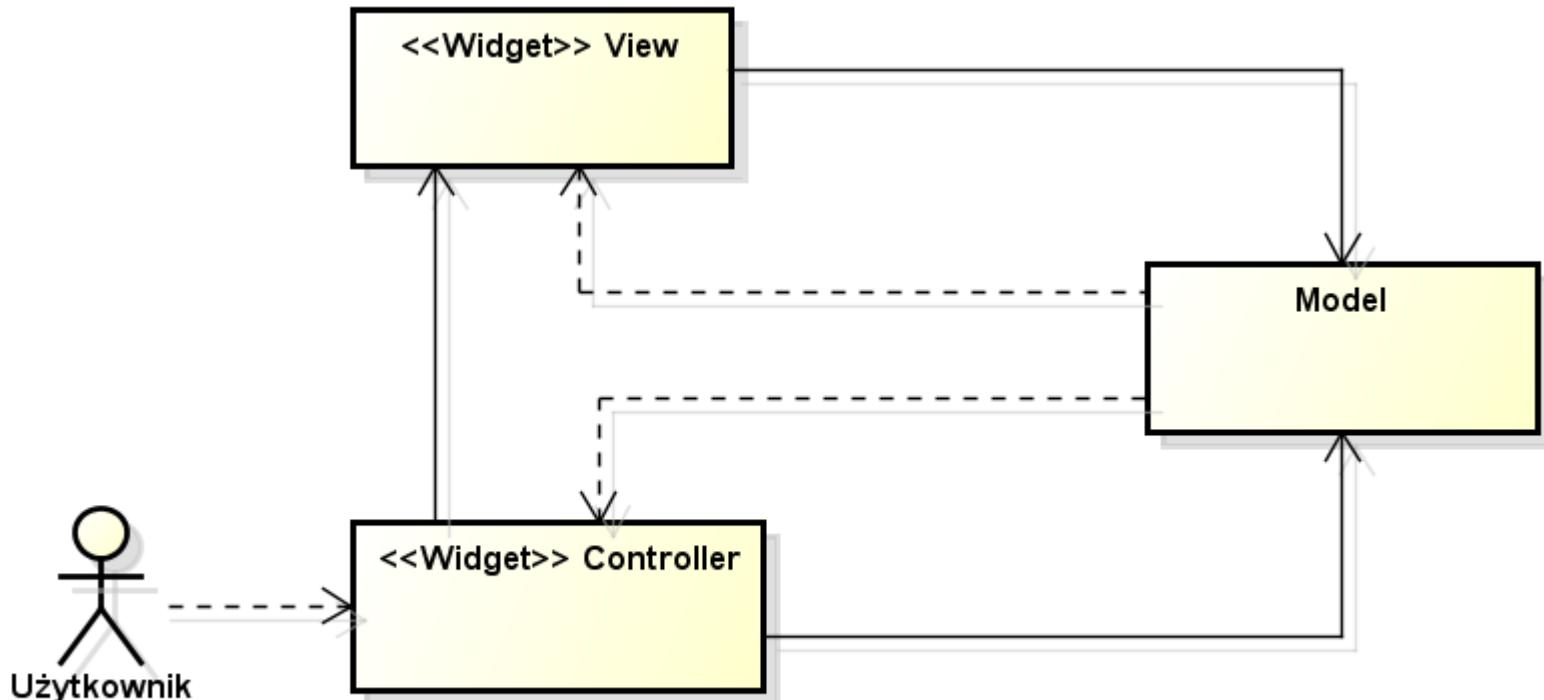
- Visual Basic for MS Office
- Wczesne rozwiązania Visual Basic, Delphi, Power Builder
- Duża część wysokopoziomowych technologii ciągle umożliwia tego typu tworzenie aplikacji (np. .NET)

bns it } # Model-View-Controller  
Architektury warstwy prezentacji

## # Intencja

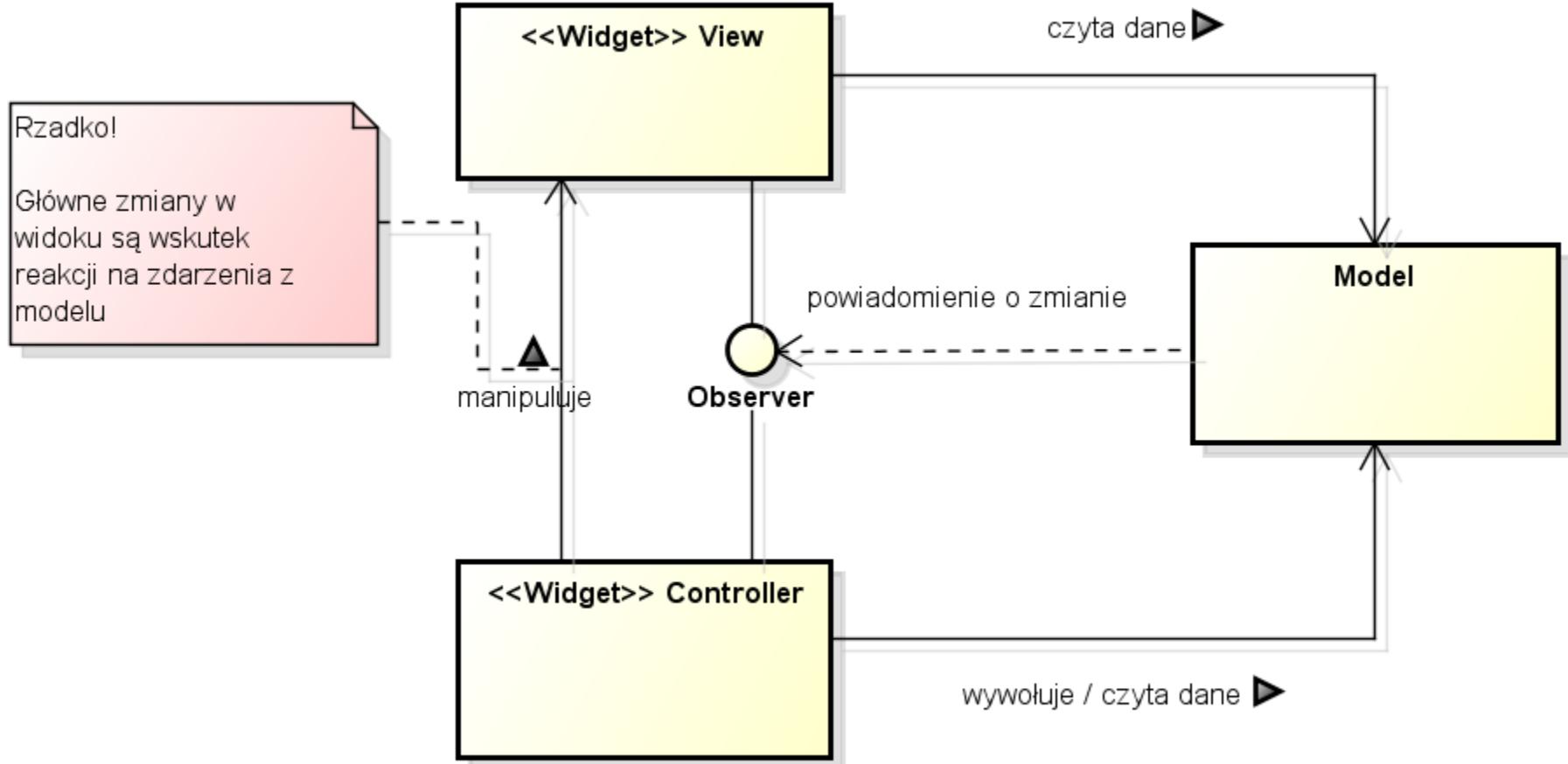
- Oddzielenie w **komponentach graficznych** części odpowiadających za komunikację ze światem zewnętrznym oraz zapewnienie spójnego mechanizmu propagacji zmian

# Model-View-Controller - struktura



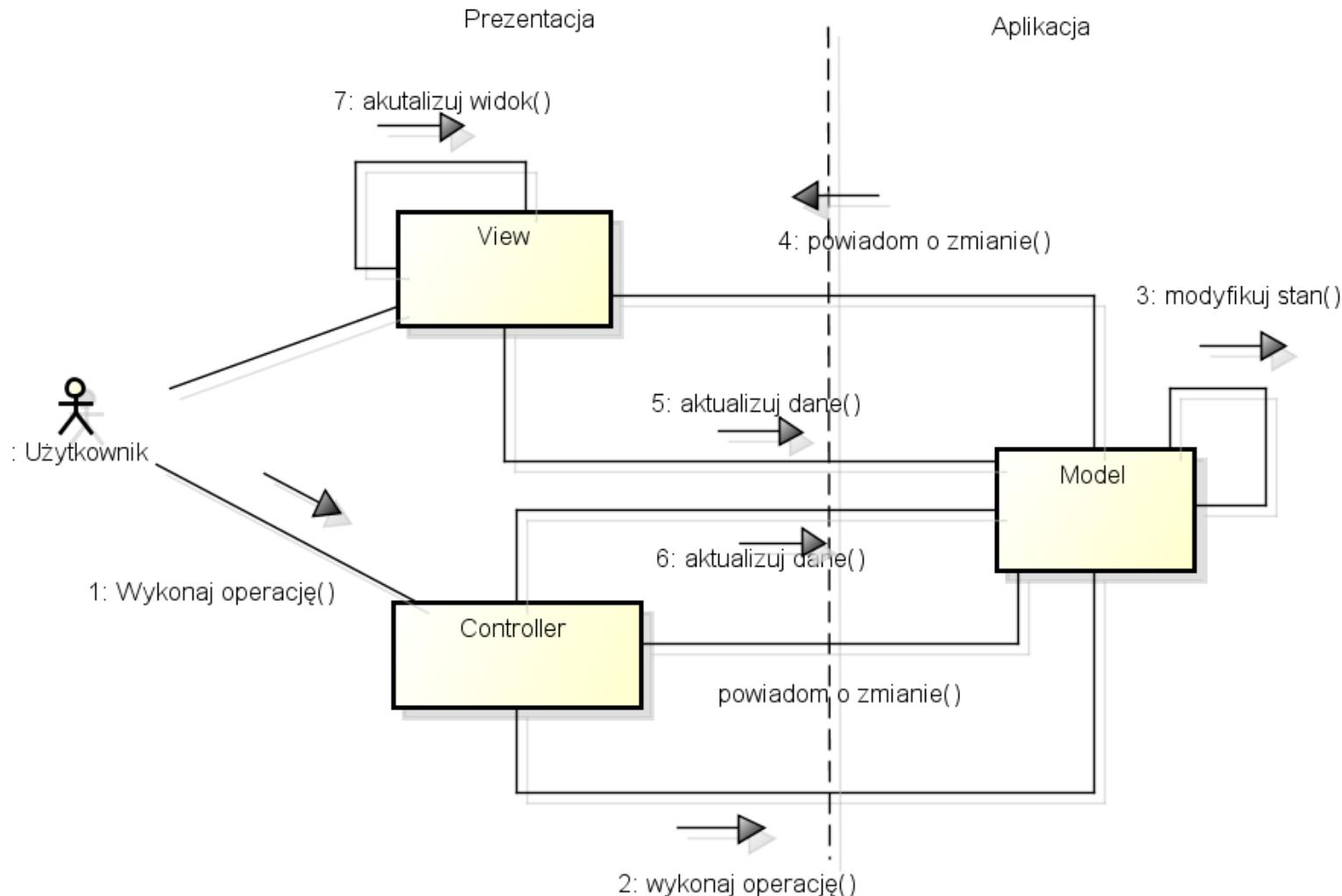
powered by astah\*

# Model-View-Controller - główne zależności

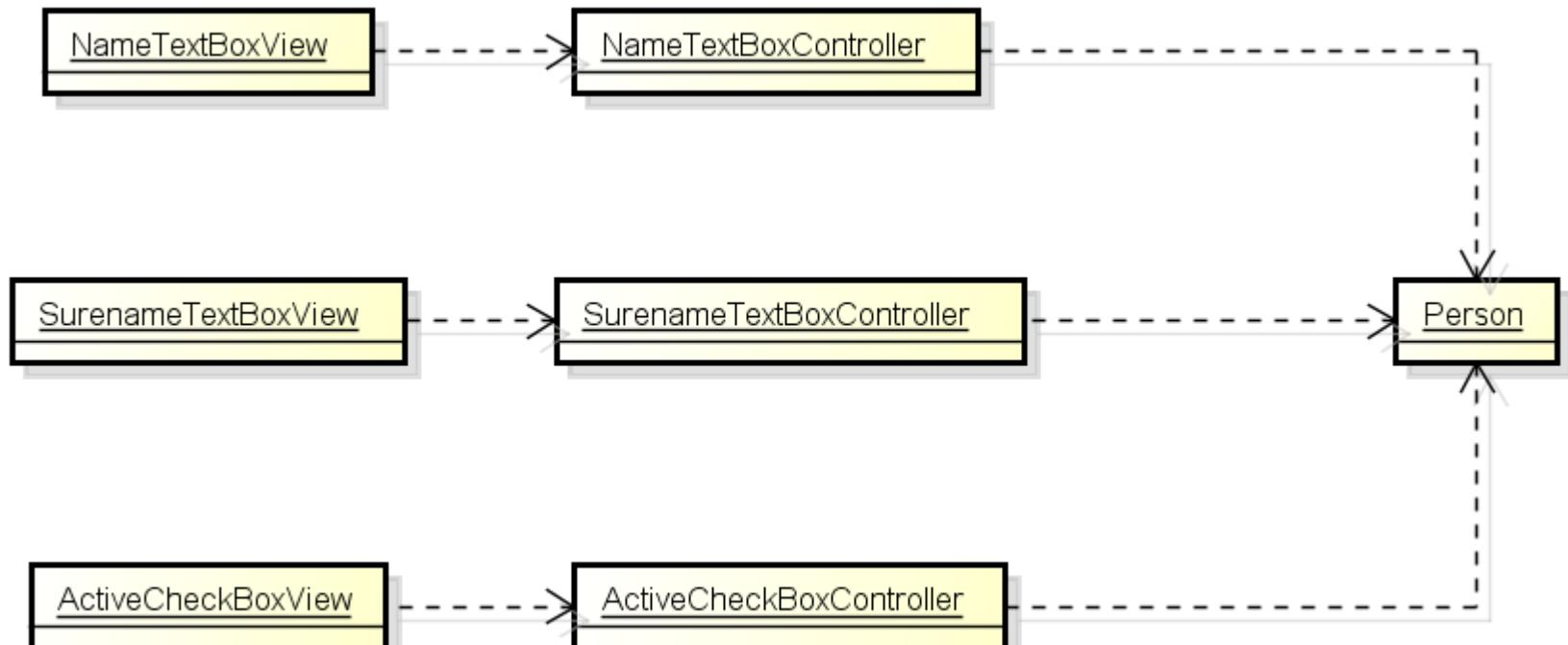


powered by astah\*

# Model-View-Controller - zasada działania



# Model-View-Controller - zasada działania



powered by astah\*

## # Model

- zasadnicza część aplikacji – zazwyczaj powiązana z danymi
- niezależny od technologii

## # View

- odpowiada za część prezentacyjną kontrolki
- reakcja na proste zdarzenia z widoku
- renderowanie danych na bazie modelu

## # Controller

- przejmuje zdarzenia i zajmuje się ich obsługą
- wywołuje operacje z modelu

- # Powiąż kontrolkę z kontrolerem, którego zadaniem jest interpretacja zdarzeń z widoku i reagowanie
- # Zmiana stanu w modelu jest propagowana za pomocą mechanizmu obserwatora
- # View i Controller to ścisła para reprezentująca kontrolkę
- # Wzorzec przydatny głównie przy tworzeniu niezależnych kontrolek



# Presentation Model

Architektury warstwy prezentacji

## # Intencja

Rozdzielić kontrolki GUI od ich stanu i zachowania

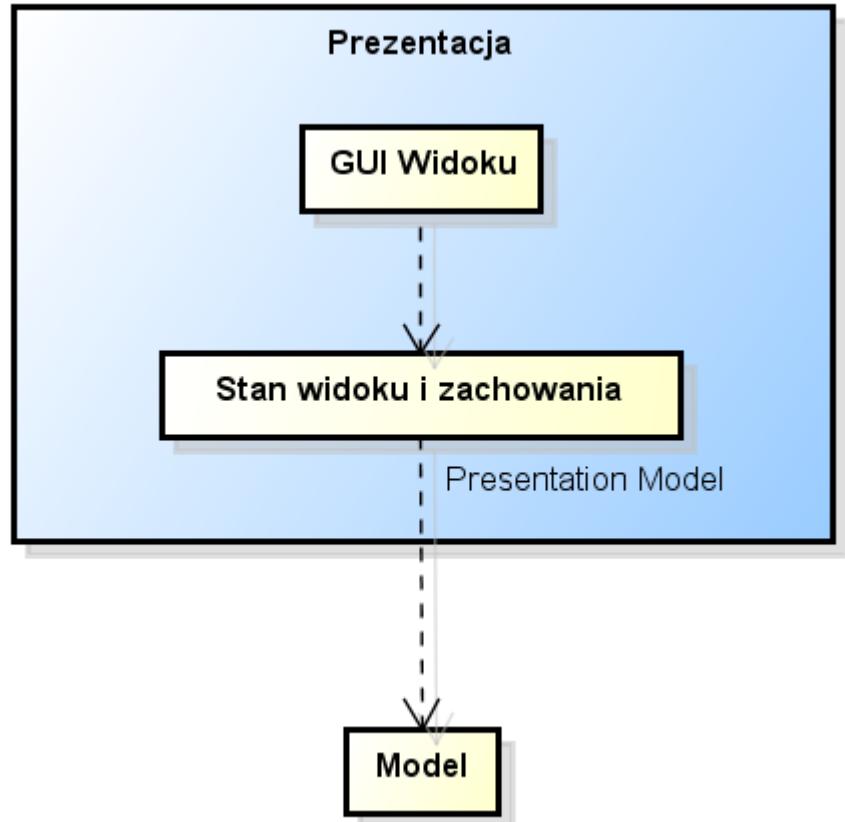
## # Cel

- Wydzielenie z warstwy prezentacji **część zależną i niezależną od technologii**
- Łatwiejsza zmiana technologii GUI
- Łatwiejsze testowanie

## # Motywacja

- Kontrolki GUI ścisłe zależą od technologii
- Kontrolki GUI przechowują stan i mogą obsługiwać zdarzenia
- Efekt
  - Trudne testowanie
  - W widokach pojawia się dużo logiki

# Presentation Model



powered by astah\*

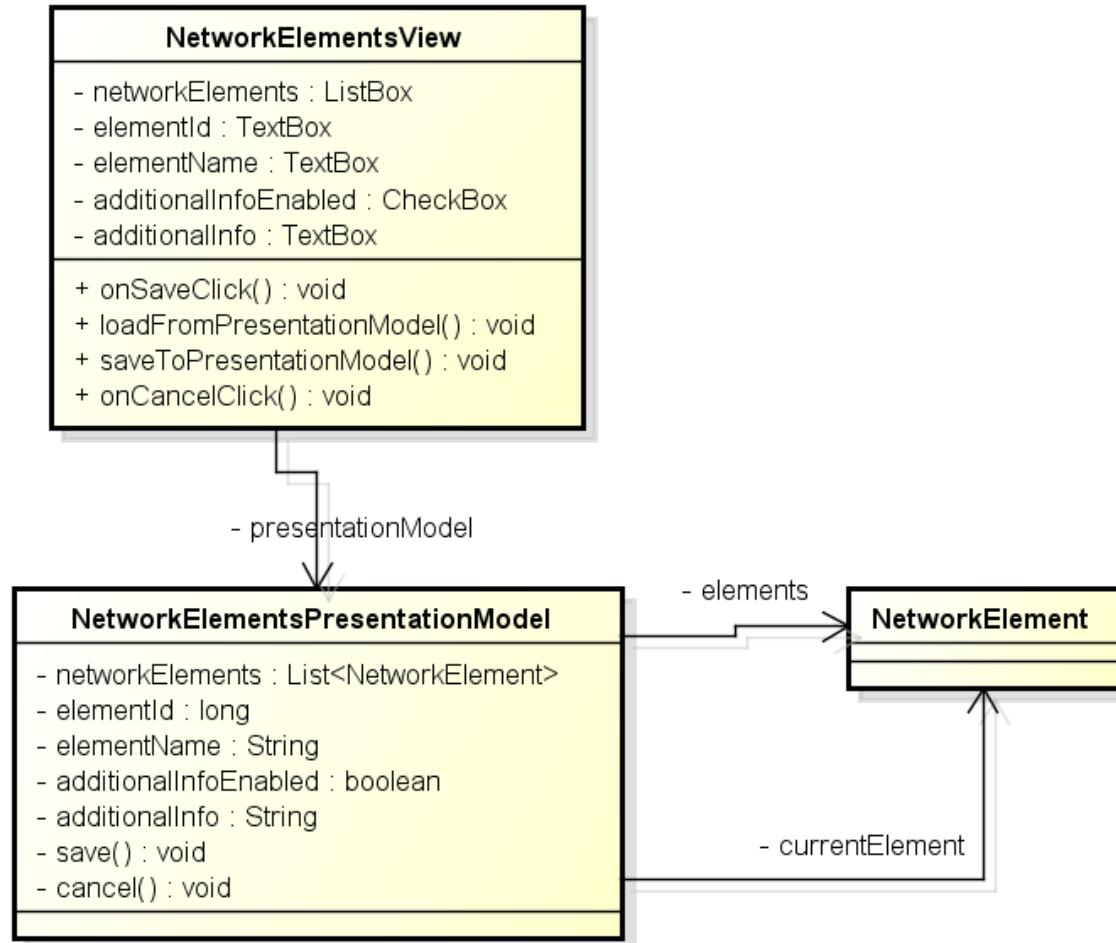
## # Presentation model

- Stan widoku – przechowuje niezależne od technologii pola odpowiadające kontrolkom oraz ich właściwościom
  - np. aktywny, nieaktywny, kolor czerwony/zielony/niebieski) jeśli się zmieniają
- Zachowania – wykonuje operacje na modelu wynikające ze zmian w widoku
- Może poinformować widok o zmianach

## # View

- Przechowuje komponenty graficzne
- Przechowuje stan GUI
- Inicjalizuje i buduje GUI
- (opcja) Nasłuchuje zmian w Presentation Model

# Presentation model - przykład



powered by astah\*

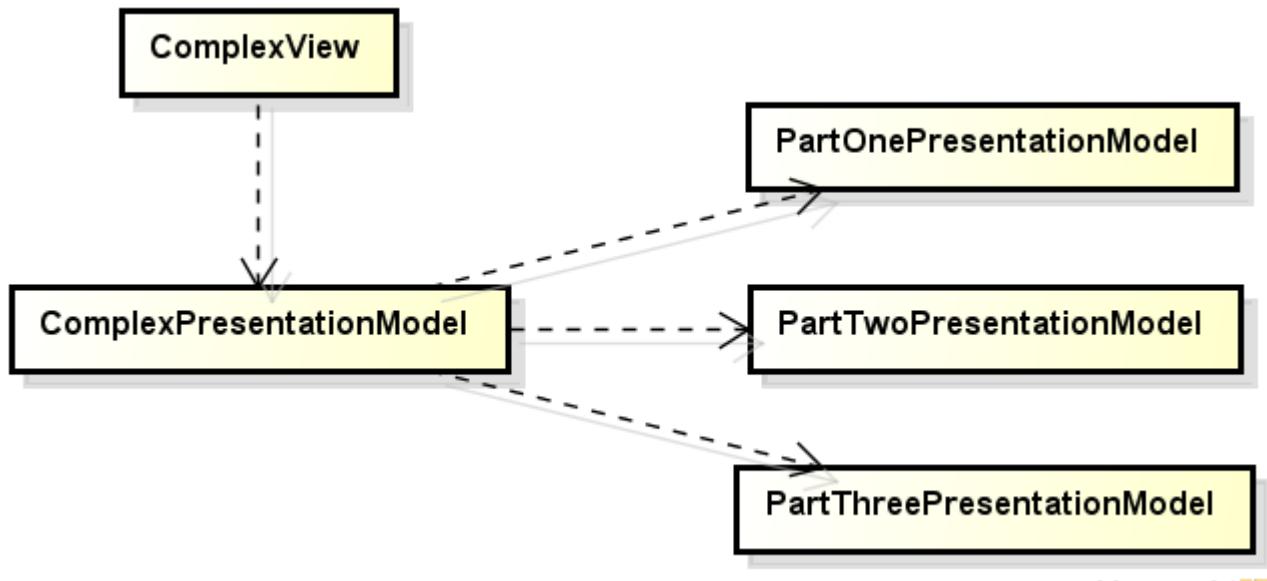
## Presentation model - strategie

- # Widok posiada referencję do Presentation Model
- # Presentation Model posiada referencję do Widoku
- # Między Widokiem a Presentation Model jest mechanizm Data Binding

*Dyskusja: Jakie są wady i zalety każdego z tych podejść?*

# Presentation model - kompozycja modeli

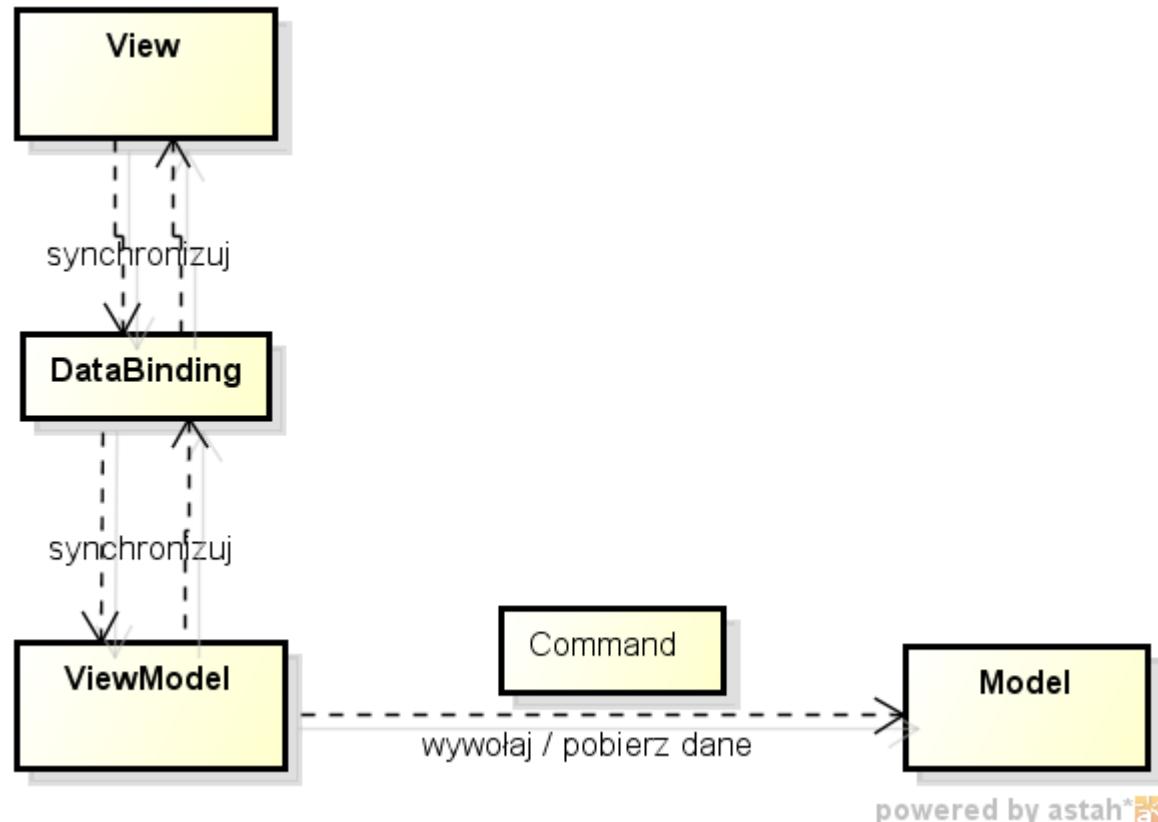
- # Kompozycja – jako forma budowania złożonych layoutów
- # Obserwator (Observer) – jako forma komunikacji między elementami



powered by astah\*

- # Wariacja na temat Presentation Model
- # Nazwa używana głównie w kontekście technologii WPF oraz Silverlight
- # ViewModel odpowiada części Presentation Model

# Model-View-ViewModel



powered by astah\*

bns it } # Model-View-Presenter  
Architektury warstwy prezentacji

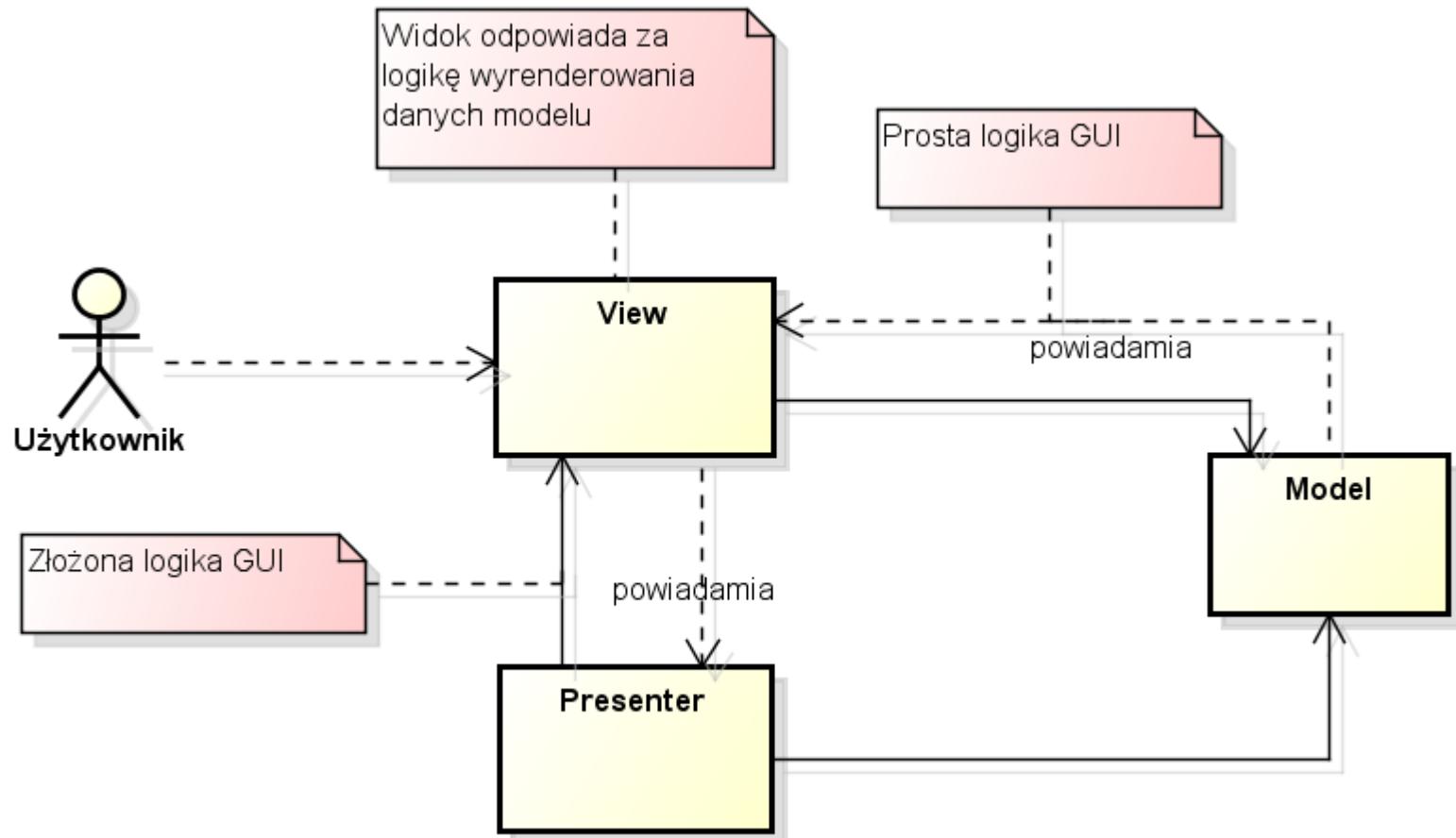
## # Intencja

- Oddzielenie w **warstwie prezentacji** części odpowiadających za komunikację ze światem zewnętrznym oraz zapewnienie spójnego mechanizmu propagacji zmian

## # Uwaga

- Mylony z MVC
- Najczęściej kiedy następuje użycie słowa Controller chodzi tak naprawdę o Presenter!

# Model-View-Presenter - wersja podstawowa



powered by astah®

## # Model

- zasadnicza część aplikacji – dziedzina
- dane i logika biznesowa
- niezależny od technologii

## # View

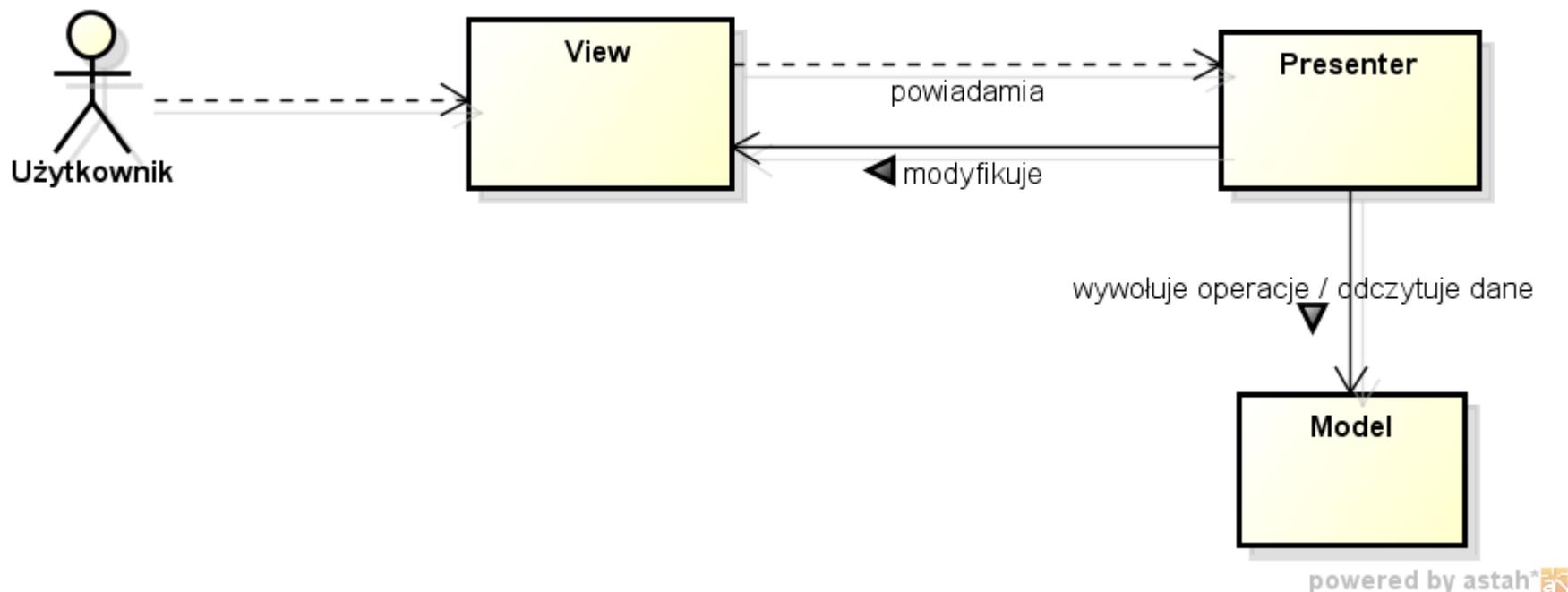
- odpowiada za większą część widoku lub za kontrolkę
- przechowuje Screen State
- renderowanie danych na bazie modelu

## # Presenter

- przejmuje zdarzenia i zajmuje się ich obsługą na modelu
- zmienia Screen State
- zajmuje się koordynacją **wszystkich elementów widoku**
- **jeden na fragment złożonego widoku**

- # Na prezenter przypada jedna lub więcej kontrollek
- # Zadaniem prezentera jest interpretacja zdarzeń z widoku i reagowanie
- # Zmiana stanu w modelu może być propagowana za pomocą mechanizmu obserwatora do widoku

# Model-View-Presenter - Passive View



# Model-View-Presenter - Passive View

- # Jedna z najczęściej stosowanych form
- # Prezenter jest aktywny
- # Widok jest pasywny
- # Przejmuje większość operacji z widoku
- # Prezenter odpowiada również za przygotowanie danych do widoku
  
- # Znaczco ułatwia testowanie
- # Prezenter pełni rolę tzw. Supervising Controller – przejmuje odpowiedzialność za koordynację zdarzeń w widoku

## # Function Controller\*

- kontroler powiązany z pewną autonomiczną częścią widoku

## # Page Controller\*

- kontroler zarządzający całym widokiem (formularzem, ramką)

## # Front Controller\*

- kontroler zarządzający całą aplikacją, interpretuje żądania i przekierowuje do innych kontrolerów i widoków

*\*Mimo nazwy Controller powyższe wzorce odpowiadają części Presenter*

## Model-View-Presenter - operacje na modelu

- # Żądania mogą być z kontrolera bezpośrednimi wywołaniami operacji z modelu
- # Można też użyć wzorca Command i Command Processor w celu enkapsulacji żądań
  - efekt: przeniesienie większej części logiki aplikacji do modelu
    - kontroler może być prostszy

## # Architektury prezentacji

- Autonomous View – wszystko w jednej klasie
- Separated Presentation
- Synchronizacja stanu
- Forms and Control
- Model-View-Controller
- Presentation Model
- Model-View-Presenter